Effiziente Methoden zum Verifizierten Lösen von Optimierungsaufgaben und Nichtlinearen Gleichungssystemen

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

am Fachbereich Mathematik und Naturwissenschaften der
Bergischen Universität Wuppertal
genehmigte

Dissertation

von

Dipl.-Math. Thomas Beelitz

aus Würselen

Tag der mündlichen Prüfung:	20. April 2006
Referent:	Prof. Dr. Bruno Lang
Koreferent:	Prof. Dr. Andreas Frommer

Inhaltsverzeichnis

Danksagung

Die vorliegende Dissertation enstand im Zeitraum vom Juni 2002 bis Januar 2006 am Fachbereich C (Fachgruppe Mathematik und Informatik) der Bergischen Universität Wuppertal während meiner Arbeit als wissenschaftlicher Mitarbeiter.
An dieser Stelle möchte ich allen danken, die auf vielfältige Weise an der Entstehung der vorliegenden Arbeit beteiligt waren. Mein besonderer Dank gilt meinem Doktorvater Prof. Dr. Bruno Lang, der die Thematik dieser Arbeit angeregt hat, für die stetige und immer geduldige Hilfs- und Diskussionsbereitschaft. Er unterstützte meine Arbeit stets und trieb sie durch viele Anregungen voran. Herrn Prof. Dr. Andreas Frommer danke ich herzlich für die Aufnahme in seine Arbeitsgruppe und die freundliche Übernahme des Koreferats.
Allen Mitarbeiterinnen und Mitarbeitern des Instituts für Angewandte Informatik danke ich für ihre ständige Hilfsbereitschaft sowie ihre langjährige und stets gute Zusammenarbeit. Besonders erwähnen möchte ich an dieser Stelle Herrn Paul Willems, der einige Aspekte dieser Arbeit motiviert und durch seine ständige Diskussionbereitschaft zum Erfolg dieser Arbeit beigetragen hat. Darüber hinaus las er große Teile der Arbeit in mehreren unausgereiften Fassungen. Seine kritischen Kommentare trugen an vielen Stellen zu einer konsistenten Darstellung bei.
Ein ganz besonderes Dankeschön geht an meine Freundin Silke für ihre Anteilnahme, Aufheiterung und Ablenkung. Sie übernahm während der vergangenen Jahre viele eigentlich mir zufallenden Aufgaben und gab mir dadurch zusätzliche Zeit zum Arbeiten.
Meinen Eltern danke ich für ihre andauernde Unterstützung auf meinem bisherigen Lebensweg, ohne die das Gelingen dieser Arbeit sicher nicht möglich gewesen wäre.

Wuppertal, im Januar 2006 Thomas Beelitz

Motivation

„Mathematics knows no races or geographic boundaries;
for mathematics, the cultural world is one country."

David Hilbert (1862-1943)

Das Lösen von nichtlinearen Gleichungssystemen und Optimierungsproblemen gehört zu den Grundaufgaben vieler wissenschaftlicher Anwendungen. Je nach Aufgabenstellung stehen eine Vielzahl verschiedener numerischer Lösungsverfahren bereit. In vielen Fällen ist man allerdings daran interessiert, dass die berechneten Ergebnisse *garantiert*, d.h. mathematisch korrekt, sind. Zum einen können für diese Aufgabenstellung *symbolische Methoden* der Computeralgebra eingesetzt werden. „Symbolisch" bedeutet dabei, dass es das Ziel ist, eine geschlossen Form der Lösungen in einer symbolischen Darstellung zu finden. Eine Möglichkeit der symbolischen Lösung von polynomiellen Gleichungssystemen ist die so genannte *Cylindrical Algebraic Decomposition (CAD)*, die von Collins [3] entwickelt wurde und in viele bestehende Formelmanipulationssysteme integriert ist. Der entscheidende Nachteil von symbolischen Methoden liegt in deren Aufwand: Ist die Anzahl der involvierten Variablen in einem polynomiellen Gleichungssystem gleich n, so beträgt der Aufwand zur Berechnung einer CAD insgesamt $\mathcal{O}(2^{2^n})$. Aus diesem Grund sind symbolische Methoden nur für sehr kleine Systeme einsetzbar und damit für praktische Probleme im Prinzip unbrauchbar. Für größere Gleichungssysteme sollten daher Algorithmen, die auf Intervallarithmetik beruhen, eingesetzt werden.

$\diamond$

In den unterschiedlichsten technischen Anwendungen ist man an optimalen Gleichgewichtszuständen von dynamischen Modellen interessiert. Eine Reihe dynamischer Modelle, die in der Prozesstechnik eingesetzt werden, können in der Form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \eta), \tag{1}$$

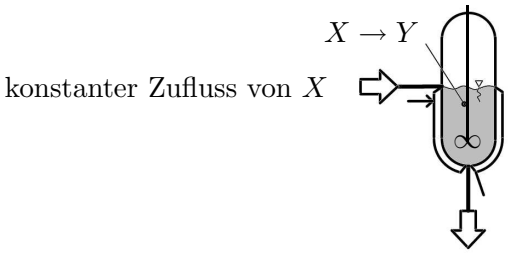

Abbildung 1: CSTR (Continuous-Flow Stirred-Tank Reactor): Modellierung von exothermen Reaktionen. Die Substanz X reagiert zu einer Substanz Y. Zwei Ventile regeln den konstanten Zu- bzw. Ablauf.

mit $\mathbf{x} \in \mathbb{R}^m$ und $\eta \in \mathbb{R}^n$ beschrieben werden. Dabei fasst $\mathbf{x}$ die so genannten Zustandsvariablen des Systems, wie z.B. die Konzentrationen verschiedener Chemikalien, und η die Kontrollparameter, wie z.B. Zuflussraten, zusammen. Die Funktion $\mathbf{f} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$ ist dabei bezüglich $\mathbf{x}$ und η „hinreichend glatt“. Als Beispiel für ein Modell aus der Prozesstechnik diene an dieser Stelle der *Continuous-Flow Stirred-Tank Reactor* (CSTR) aus Abbildung 1, der bei der Modellierung von exothermen Reaktionen seine Anwendung findet. In diesem Behälter reagiert eine Substanz X zu einer zweiten Substanz Y. Zwei Ventile regeln dabei den konstanten Zu- und Ablauf des Eduktes bzw. Produktes. Die resultierende Reaktion lässt sich durch die Differentialgleichung

$$\begin{cases} \dfrac{\partial x}{\partial t} &= (x_0 - x) - D \cdot \exp\left(\dfrac{y}{1 + g \cdot y}\right) \cdot x \\ \dfrac{\partial y}{\partial t} &= (y_0 - y) + B \cdot D \cdot \exp\left(\dfrac{y}{1 + g \cdot y}\right) \cdot x \end{cases} \tag{2}$$

mit den Betriebsparametern x_0 (Konzentration des Edukts beim Zufluss), x (Konzentration des Edukts im Reaktor), y_0 (Energiefluss beim Zufluss), y (Energiefluss im Reaktor), B (Reaktionsenergie), g (Aktivierungsenergie) und D (Verweildauer; Damköhlerzahl) beschreiben.

Im Folgenden nehmen wir stets an, dass das gewählte Modell (1) korrekt ist, d.h. der zugehörige Prozess tatsächlich durch die Gleichung (1) beschrieben wird, die Parameter η aber nur innerhalb gewisser Schranken (Intervalle) bekannt sind. Eine Methode zur Berechnung optimaler Gleichgewichtszustände einer nichtlinearen Produktionsanlage besteht in der Lösung eines so genannten nichtlinearen Programms (NLP) der Form

$$\begin{aligned} \min_{\mathbf{x},\eta} \quad & \phi(\mathbf{x}, \eta) \\ & \mathbf{f}(\mathbf{x}, \eta) = \mathbf{0} \\ & \mathbf{g}(\mathbf{x}, \eta) \geq \mathbf{0}, \end{aligned} \tag{3}$$

wobei ϕ einer reellwertigen Kostenfunktion entspricht, die hinreichend „glatt“ bezüglich der Parameter x und η ist. Diese Kostenfunktion kann zum Beispiel die Energie- und

Investitionskosten einer Produktionsanlage beinhalten. Die $l \geq 0$ Ungleichungsnebenbedingungen $\mathbf{g} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^l$ berücksichtigen dabei Schranken für Temperaturen oder Konzentrationen von unerwünschten Nebenprodukten. Diese sind im Allgemeinen nichtlinear und ebenfalls „hinreichend glatt“ bezüglich der Variablen $\mathbf{x}$ und η.

Da die Gleichung (1) ein dynamisches Modell der Anlage darstellt, können die dynamischen Eigenschaften jedes Gleichgewichtszustandes untersucht werden. Dabei ist die Stabilität des über (3) gefundenen Optimums von besonderem Interesse. Unglücklicherweise berücksichtigt das Optimierungsproblem (3) keine dynamischen Eigenschaften des Modells, d.h. die Lösung von (3) kann zu einem optimalen Betriebspunkt führen, der *nicht stabil* ist. Das Betreiben einer Produktionsanlage mit instabilen Betriebsparametern kann unangenehme Folgen haben. Es ist möglich, dass sich die erzielte Menge oder die Qualität eines Produktes verringert oder große Energiemengen freigesetzt werden, was schlimmstenfalls zu der Zerstörung einer Produktionsanlage führen kann.

Existierende Ansätze zum Auffinden eines stabilen optimalen Betriebspunktes, wie zum Beispiel die so genannte *Fortsetzungsmethode*, sind extrem schwer zu automatisieren bzw. zu implementieren. Mönnigmann und Marquardt [51] präsentieren einen neuen Ansatz, der die Nachteile der bekannten Methoden umgeht. Das Problem, zwischen stabilen und instabilen Lösungen von (3) unterscheiden zu müssen, wird dadurch gelöst, dass geeignete Nebenbedingungen zum nichtlinearen Optimierungsproblem (3) hinzugefügt werden. Diese Nebenbedingungen garantieren nicht nur die Stabilität des optimalen Betriebspunktes, sondern sollen sogar gewährleisten, dass dieser Punkt einen gewissen—vom Benutzer vorgegebenen—Mindestabstand zu allen kritischen Punkten hat.

Die hinzugefügten Nebenbedingungen garantieren allerdings nur *theoretisch* die Stabilität des optimalen Betriebspunktes. Setzt man zur Lösung des Optimierungsproblems (3) ein gewöhnliches (reelles) Optimierungstool ein, so werden die Nebenbedingungen natürlich nur diskret ausgewertet. Durch dieses Vorgehen oder das Auftreten von einfachen Rundungsfehlern kann es passieren, dass bestimmte kritische Punkte im Parameterbereich übersehen werden. Im Anschluss an den Optimierungsprozess wird aus diesem Grund der Pfad zwischen Start- und Endpunkt des Optimierungsproblems analysiert. Dazu werden, aus der Bifurkationstheorie [9, 45] bekannte, *Testfunktionen* angewendet, die auf die Existenz verschiedener Singularitäts-Typen prüfen. Wird ein kritischer Punkt entdeckt, so werden—wie oben beschrieben—geeignete Nebenbedingungen zum Optimierungsproblem hinzugefügt und ein *neuer* Optimierungsprozess gestartet. Allerdings werden die Testfunktionen erneut nur diskret, also in bestimmten Punkten auf dem Optimierungspfad, ausgewertet. Aus diesem Grund können bestimmte kritische Punkte übersehen werden. Kritische Punkte, die den Optimierungspfad gar nicht treffen, bleiben in jedem Fall unentdeckt (siehe Abbildung 2).

Mit Hilfe der Verfahren aus der Prozesstechnik erhält man folglich einen optimalen Betriebspunkt, der nur mit *hoher Wahrscheinlichkeit*, aber *nicht* garantiert, robust bezüglich der Betriebsparameter ist. Dabei ist zu beachten, dass dieser Betriebspunkt nur optimal unter den Parametereinstellungen ist, die einen gewissen Mindestabstand zu vorhandenen Mannigfaltigkeiten haben. Um *mathematisch nachweisen* zu können, dass der vorgesehene Parameterbereich zur Steuerung des Prozesses frei von kritischen

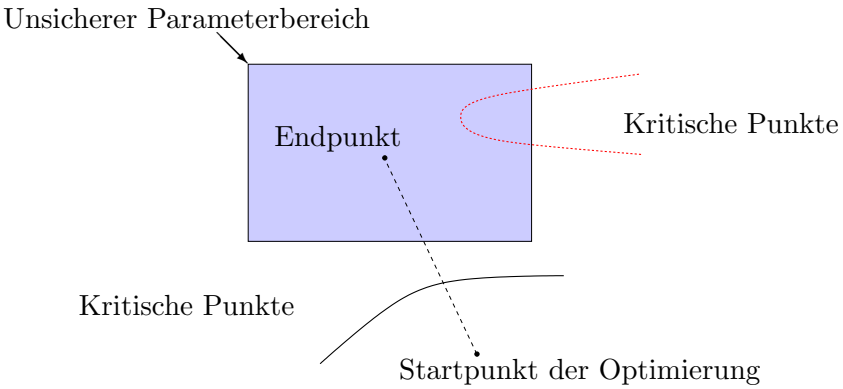

Abbildung 2: Auf dem Pfad vom Startpunkt der Optimierung zum Optimum werden kritische Punkte (durchgezogene Linie) entdeckt. Kritische Punkte, die disjunkt zum Pfad sind, werden nicht erkannt.

Punkten eines bestimmten Singularitäts-Typs ist, kann ein ergebnisverifizierender nichtlinearer Gleichungslöser eingesetzt werden. Zu diesem Zweck werden zu den Gleichungen der Gleichgewichtszustände $\mathbf{f}(\mathbf{x}, \eta) = \mathbf{0}$ zusätzliche Gleichungen, die den entsprechenden Typ der Singularität beschreiben, hinzugefügt. Eine detaillierte Beschreibung der daraus resultierenden *erweiterten Systeme* findet der interessierte Leser in [22]. Als Beispiel wollen wir an dieser Stelle die *Sattel-Knoten-Singularität* und die *Kuspe* betrachten. Für diese ergeben sich die erweiterten Systeme

$$\begin{cases} \mathbf{f}(\mathbf{x}, \eta) &= \mathbf{0} \\ \mathbf{A}\mathbf{v} &= \mathbf{0} \\ ||\mathbf{v}||_2^2 &= 1 \end{cases} \tag{4}$$

bzw.

$$\begin{cases} \mathbf{f}(\mathbf{x}, \eta) &= \mathbf{0} \\ \mathbf{A}\mathbf{v} &= \mathbf{0} \\ ||\mathbf{v}||_2^2 &= 1 \\ \mathbf{A}^T\mathbf{w} + \rho\mathbf{v} &= \mathbf{0} \\ ||\mathbf{w}||_2^2 &= 1 \\ \mathbf{w}^T\mathbf{B}\mathbf{v}\mathbf{v} &= 0 \end{cases}, \tag{5}$$

wobei $\mathbf{A} := \frac{\partial \mathbf{f}(\mathbf{x},\eta)}{\partial \mathbf{x}}$ die erste Ableitung von $\mathbf{f}$ bezüglich $\mathbf{x}$ und $\mathbf{B} := \frac{\partial^2 \mathbf{f}(\mathbf{x},\eta)}{\partial \mathbf{x}^2}$ die zweite Ableitung ist. Weiterhin ist $\rho \in \mathbb{R}$ eine *Regularisierungsvariable*, deren Wertebereich grundsätzlich unbeschränkt ist. Die Vektoren $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ stellen ausschließlich Hilfsvektoren dar. In Abbildung 3 haben wir eine sehr einfache Beispiel-Mannigfaltigkeit dargestellt: Singularitäten markieren dabei den Übergang zwischen eindeutigen und mehrdeutigen Zuständen.

Zur Lösung der nichtlinearen Gleichungssysteme können Intervallverfahren eingesetzt werden, um die Korrektheit der Ergebnisse zu garantieren. Diese Verfahren gehören zu den deterministischen Verfahren, welche nach dem Prinzip des erschöpfenden Durchsuchens arbeiten. Das betrachtete Gebiet wird so lange unterteilt und Teilgebiete, die garantiert keine Lösung enthalten können, verworfen, bis man „enge“ Einschließungen

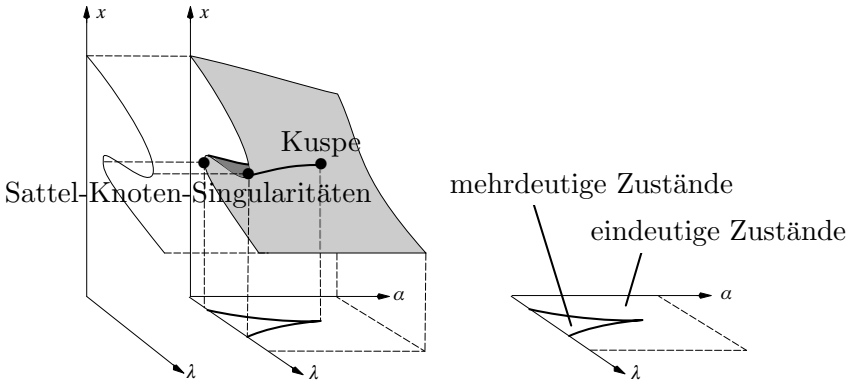

Abbildung 3: Beispiel-Mannigfaltigkeit $\mathbf{f}(\mathbf{x}, \eta) = \mathbf{0}$ mit $\mathbf{x} = x$ and $\eta = (\alpha, \lambda)^T$, die zwei Singularitäts-Typen zeigt.

aller Lösungen in $[\mathbf{x}^{(0)}]$ erhält. Die verifizierte Lösung wird erst dadurch möglich, dass *Einschließungen* und nicht Schätzungen für Wertebereiche, Lipschitzkonsatanten usw. verwendet werden, die mit Hilfe der Intervallarithmetik berechnet werden. In dieser Arithmetik, die von Moore im Jahre 1966 in dem Buch *Interval Analysis* [52] präsentiert wurde, wird mit Intervallen anstatt reeller Zahlen gerechnet. Mit Hilfe eines verifizierenden Lösers kann folglich eine *garantierte* Aussage darüber getroffen werden, ob eine Singularität eines bestimmten Typs im Parameterbereich vorliegt oder nicht.

$\diamond$

Wie wir soeben gesehen haben, spielen ergebnisverifizierende Algorithmen also eine entscheidende Rolle, wenn es um die Sicherheit und Produktivität chemischer Produktionsanlagen geht. Aus diesem Grund entstand im Rahmen des Forschungsvorhabens *Konstruktive Methoden der nichtlinearen Dynamik zum Entwurf verfahrenstechnischer Prozesse*, einem Gemeinschaftsprojekt des Lehrstuhls für Hochleistungsrechnen der RWTH Aachen und des Lehrstuhl für Prozesstechnik der RWTH Aachen ein ergebnisverifizierender Gleichungslöser. In einer weiteren Projektphase wurde dieser Löser am Lehrstuhl für Angewandte Informatik/Algorithmik an der Bergischen Universität Wuppertal um einen Optimierer erweitert und ständig weiterentwickelt. Das resultierende Tool SONIC (Solver and Optimizer for Nonlinear Problems based on Interval Computation) wird in dieser Arbeit detailliert vorgestellt.

Sowohl für die verifizierte Lösung von nichtlinearen Gleichungssystemen und Optimierungsaufgaben als auch für die Parallelisierung der Algorithmen existieren heutzutage einige hervorragende Ansätze, die allerdings vielfach auf spezielle Problemklassen ausgerichtet sind. Das wohl bekannteste und effizienteste Tool auf dem Gebiet der ergebnisverifizierenden Algorithmen ist momentan GlobSol. Leider zeigt sich, dass die existierenden Programmpakete hinsichtlich zweier wesentlicher Aspekte nicht robust genug implementiert sind, um Systeme, die aus der Prozesstechnik und anderen technischen Anwendungen stammen, in zufriedenstellender Zeit bzw. überhaupt lösen zu können. Ein wesentlicher Punkt ist dabei die Behandlung von Auswertefehlern, die

häufig durch *Überschätzungen* des tatsächlichen Wertebereichs einer Funktion entstehen, obwohl diese auf ihrem Definitionsbereich stetig ist. Existierende Algorithmen brechen den Lösungsvorgang dann unmittelbar ab, obwohl dies völlig unnötig ist. Wie sich zeigt, können aus diesem Grund die Mehrzahl der Testsysteme aus der Prozesstechnik nicht mit den vorhandenen Tools gelöst werden. Ein zweiter wesentlicher Aspekt ist die Robustheit der Algorithmen gegenüber Parametereinstellungen. Vielfach muss der Nutzer existierender Programmpakete zusätzliche, für das betrachtete Problem spezifische, Einstellungen vornehmen. Die einzelnen Verfahren und Einstellungen von SONIC wurden deshalb so ausgewählt, dass sie für ein *breites Spektrum* von Testbeispielen sehr gute Resultate erzielen. Es ist aus diesem Grund durchaus möglich, dass nicht immer für jedes System optimale Ergebnisse erzielt werden. Nach Meinung des Autors macht erst dieses Prinzip einen verifizierten Gleichungslöser bzw. globalen Optimierer zu einem guten und in der Praxis nutzbaren Tool: Ein Ingenieur, der (wahrscheinlich) kein Experte auf dem Gebiet der Intervallrechnung ist, wird in der Regel nicht in der Lage sein, gezielt und sinnvoll Parametereinstellungen vorzunehmen.

Zunächst nur auf die besonderen Anforderungen aus der Prozesstechnik ausgerichtet (z.B. die Ausnutzung der speziellen Struktur der erweiterten Systeme), entwickelte sich SONIC im Laufe der Jahre zu einem ausgezeichneten *Allrounder* auf den Gebieten des ergebnisverifizierenden Lösens von Gleichungssystemen und Optimierungsproblemen. Neben bekannten effizienten Algorithmen wurden stets Neuentwicklungen und Verbesserungen integriert. Dabei ist SONIC momentan das einzige Tool, welches grundsätzlich auch unbeschränkte Suchbereiche für die Lösungen zulässt. Wie wir an verschiedenen Beispielen sehen werden, können mit den in SONIC implementierten effizienten Verfahren heute schon nichtlineare Gleichungssysteme bzw. Optimierungsprobleme gelöst werden, bei denen dies vor einiger Zeit noch undenkbar gewesen wäre. Dennoch sind die Probleme, die aus realen Anwendungen stammen, manchmal so groß, dass einige von ihnen noch nicht in akzeptabler Zeit oder überhaupt nicht gelöst werden können. Für die Behandlung solch komplexer Probleme bietet sich die Verwendung von Parallelrechnern an. Die Berechnungen zur Lösung dieser Probleme werden dann auf p Prozessoren verteilt. Man erwartet normalerweise, dass die Probleme dann bei einer idealen Parallelisierung in einem p-tel der Zeit bearbeitet werden. Bei Parallelrechnern unterscheidet man zwischen Rechnern mit gemeinsamem und verteiltem Speicher. Für die unterschiedlichen Architekturen sind in SONIC Implementierungen mit OpenMP bzw. MPI integriert. Bei Verwendung eines Parallelrechners steht grundsätzlich mehr Speicher zur Verfügung, was die Lösung einiger Testsysteme überhaupt erst möglich macht, da schwere Probleme oft einen großen Speicherbedarf haben.

$\diamond$

Die erfolgreiche Realisierung dieses Projektes ist einer Reihe von Leuten zu verdanken, die ich an dieser Stelle kurz erwähnen möchte. Entscheidenden Anteil bei der Implementierung von SONIC hatten Prof. Dr. Bruno Lang, Thomas Beelitz und Paul Willems. Neben den gerade genannten Autoren trugen Prof. Dr. Andreas Frommer, Prof. Christian H. Bischof, Ph.D., André Kienitz, Dieter an Mey und Klaus Schulte Althoff mit vielen Ideen und Neuentwicklungen zum Erfolg von SONIC bei. Auf dem

Gebiet der Prozesstechnik standen uns stets Prof. Dr. Wolfgang Marquardt und Dr. Martin Mönnigmann mit Rat und Tat zur Seite.

◇

Nach Einführung elementarer Notation behandelt das erste Kapitel bekannte Ergebnisse aus dem Bereich der Intervallrechnung. Dazu gehört unter anderem eine kurze Einführung in die Intervallarithmetik und ein Überblick über Möglichkeiten, wie der Wertebereich einer Funktion eingeschlossen werden kann. Außerdem werden einige Grundlagen für die verwendeten Verfahren dargestellt. Die Notation der, in dieser Arbeit präsentierten, Algorithmen schließt dieses Kapitel ab.

Nach diesen Vorbereitungen werden mit Hilfe der dargestellten Grundlagen im zweiten Kapitel effiziente Verfahren für die verifizierte Lösung von nichtlinearen Gleichungssystemen vorgestellt. Zunächst wird der so genannte Branch–and–Bound-Ansatz präsentiert, bei dem eine Startbox sukzessive in Teilboxen zerlegt wird. Teilboxen, die *garantiert* keine Lösung des Gleichungssystems enthalten, werden gelöscht. Dadurch werden *alle* Lösungen immer enger eingeschlossen. Dieser einfache Ansatz muss in der Praxis mit *Kontraktionsverfahren* vervollständigt werden, um effektiv zu arbeiten. Diese Verfahren sind in der Lage, die Boxen zu verkleinern ohne dabei Lösungen des Gleichungssystems zu verlieren. Die untersuchten Kontraktionsverfahren können grob in zwei Kategorien eingeteilt werden: symbolische und numerische. In dieser Arbeit wird das *präkonditionierte Intervall-Newton-Gauß-Seidel-Verfahren* eine ganz besondere Rolle einnehmen. Das Newton-Verfahren kann neben dem hinreichend bekannten Inverse-Midpoint-Präkonditionierer auch mit optimalen Präkonditionierern, die über lineare Optimierungsprobleme berechnet werden, angewendet werden. Das numerische Newton-Verfahren ist in SONIC eng mit dem *Constraint-Propagation* gekoppelt: Das symbolische Constraint-Propagation impliziert durch Aufspaltung der Gleichungssysteme (unterschiedlich große) *Split-Systeme*, die wiederum mit dem Newton-Verfahren gelöst werden können. Da die resultierenden Split-Systeme teilweise extrem groß sind, wird auf die Anwendung des Newton-Verfahrens auf diese in den bekannten Tools zur verifizierten Lösung von nichtlinearen Gleichungssystemen verzichtet. In dieser Arbeit präsentieren wir einen *hierarchischen Ansatz*, der mit Hilfe des neu entwickelten *hybriden Newton-Verfahrens* ein exzellentes Kontraktionsverfahren darstellt. Diese Technik ermöglicht es, das Newton-Verfahren mit unterschiedlichen Präkonditionierern geschickt und effizient auf vielversprechende Gleichungen verschiedener Split-Systeme anzuwenden. Für die Teilung einer Box werden ebenfalls unterschiedliche Möglichkeiten aufgezeigt. Es wird herausgearbeitet, dass die klassischen Varianten jeweils für eine Reihe von Testsystemen gut funktionieren, aber bei anderen wiederum versagen. Aus diesem Grund entwickeln wir eine hybride Bisektionsstrategie, die die Nachteile der klassischen Unterteilungsstrategien versucht zu umgehen. Weiterhin stellen wir das Verfahren der *Splitting-Bisektion* vor. Diese Methode nutzt einen *Splitting-Präkonditionierer*, um im Newton-Gauß-Seidel-Verfahren Boxkomponenten nicht nur zu halbieren, sondern gezielt *Lücken* aus diesen herauszuschneiden. Anschließend werden Möglichkeiten vorgestellt, mit denen die Existenz bzw. die Eindeutigkeit von Lösungen in Teilboxen nachgewiesen werden kann. Diese *Verifikationstests* können bei einem weiteren

Kontraktionsverfahren eingesetzt werden, den so genannten *Exclusion-Regions*. Durch ausführliche Experimente werden wir anschließend zeigen, dass SONIC ein hervorragendes Tool zum verifizierten Lösen von Gleichungssystemen ist.

Im dritten Kapitel befassen wir uns mit dem verifizierten Lösen von Optimierungsproblemen. Nach der Präsentation effizienter Techniken für Optimierungsprobleme ohne Nebenbedingungen gehen wir auf Besonderheiten bei der Optimierung mit Nebenbedingungen ein. Da wir unser Hauptaugenmerk auf die Entwicklung effizienter Techniken zur Lösung nichtlinearer Gleichungssysteme gelegt haben, greifen wir bei der Optimierung in der Regel auf bewährte Standard-Techniken zurück. Anhand ausführlicher Untersuchungen werden wir sehen, dass SONIC auch bei der Optimierung sehr effizient arbeitet. Dieser Fakt beruht in der Hauptsache auf der sehr guten Qualität der Verfahren aus Kapitel 2, da uns die effiziente Behandlung von Gleichungssystemen bei der Lösung des *Gradientensystems* zu Gute kommt.

Das vierte Kapitel befasst sich mit der Parallelisierung der eingeführten Algorithmen, die sowohl für Systeme mit verteiltem Speicher als auch für Systeme mit gemeinsamem Speicher durchgeführt wird. Nach einer kurzen Einführung in das Thema Parallelisierung präsentieren wir die verwendeten Routinen, die für die Parallelisierung mit OpenMP und MPI benötigt werden. Für den nichtlinearen Gleichungslöser stellen wir dann drei unterschiedlich aufwändige Parallelisierungsstrategien vor, die alle gute—für die MPI-Variante sogar beinahe perfekte—*Speedups* liefern. Anschließend stellen wir unsere Strategie zur Parallelisierung der Optimierers vor. Dabei beschränken wir uns auf die Parallelisierung des Algorithmus für Optimierungsaufgaben ohne Nebenbedingungen mit MPI. Eine Anpassung dieses Algorithmus für Optimierungsaufgaben mit Nebenbedingungen ist allerdings sehr leicht möglich. Die gewählte Strategie beruht in wesentlichen Teilen auf einen Algorithmus aus der Arbeit von Berner [8], den wir an einigen Stellen geeignet abgeändert haben.

Jedes der Kapitel wird mit ausführlichen Experimenten, die die besondere Leistungsfähigkeit von SONIC untermalen, abgeschlossen. In den Kapiteln 2 und 3 folgt anschließend noch ein aussagekräftiges Anwendungsbeispiel. Jedes dieser Beispiele stellt ein Problem aus der Praxis dar und kann momentan ausschließlich von SONIC berechnet werden.

Im fünften Kapitel wird der Gesamtaufbau von SONIC erläutert. Zunächst werden die einzelnen Kern-Module des Tools vorgeführt. Es folgt eine Vorstellung der unterstützten Intervall-Bibliotheken, die durch den integrierten *generischen Intervall-Code* und den C-Präprozessor angesteuert werden können. Dadurch erhält SONIC eine besonders große Portabilität. Nachdem die integrierten (reellen) Optimierungstools erläutert wurden, gehen wir kurz auf existierende verifizierende Gleichungslöser und Optimierer ein.

Eine kurze Zusammenfassung der wichtigsten Ergebnisse, sowie einige Vorschläge für zukünftige Aufgaben schließen diese Arbeit ab. Der Anhang empfiehlt sich als Überblick über die verwendeten Testbeispiele. Es werden neben den Systemen selbst, sowohl die Wertebereiche der Variablen als auch die geforderte Genauigkeit angegeben. In vielen Fällen wird zusätzlich noch auf die gesonderte Problematik und Herkunft der Testfälle eingegangen.

Kapitel 1

Einleitung

Wie bereits in der Motivation erwähnt, ist es in vielen Fällen von entscheidender Bedeutung, die Nullstellen eines nichtlinearen Gleichungssystems oder das globale Minimum einer Funktion mit *absoluter Sicherheit* zu finden. Eine bewährte Technik dazu ist die so genannte Intervallarithmetik. Die Grundidee dabei ist, anstatt mit gerundeten Zahlen *endlicher Präzision*, mit Einschließungen von Mengen zu rechnen, die zwar zu groß sein können, aber immer die gewünschte Eigenschaft erfüllen, jede Lösung zu enthalten.

In diesem Kapitel werden zunächst einige häufig auftretende Begriffe und Notationen bereitgestellt. Anschließend betrachten wir die bereits erwähnte *Intervallarithmetik* sowie einige ihrer wichtigsten Eigenschaften. Nachdem wir auf die Problematiken bei der Implementierung auf einem Rechner eingegangen sind, werden Möglichkeiten aufgeführt, den Wertebereich einer Funktion mit Hilfe der Intervallarithmetik zu bestimmen. Eine sehr gelungene und detailliertere Einführung zu diesem Thema findet man zum Beispiel in [54]. Zum Abschluss dieses Kapitels wird auf die Notation der Algorithmen in dieser Arbeit eingegangen.

1.1 Intervalle

Die bekannten Standardwerke zur Intervallarithmetik lassen grundsätzlich nur Intervalle zu, die beschränkt sind. Wie wir bereits in der Motivation gesehen haben, sind in vielen Systemen aus der Praxis eine oder mehrere Variablen *unbeschränkt*, da z.B. nur eine untere Schranke für diese bekannt ist. Aus diesem Grund lassen wir sowohl in dieser Arbeit als auch in der Implementierung von SONIC unbeschränkte Intervalle zu.

Im Folgenden betrachten wir *Intervalle* als Menge reeller Zahlen, die zusammenhängend und abgeschlossen, aber grundsätzlich auch unbeschränkt sein können. Zur Darstellung dieser Intervalle erweitern wir zunächst die reellen Zahlen um die beiden ausgezeichneten Elemente $\{-\infty, +\infty\}$ mit der Eigenschaft

$$-\infty < a < +\infty, \quad \forall a \in \mathbb{R}.$$

Man erhält damit die so genannten *erweiterten reellen Zahlen* über die Definition $\mathbb{R}_* :=$

$\mathbb{R} \cup \{-\infty, +\infty\}$. Für $+\infty$ schreiben wir in der Regel auch einfach ∞. Die Fortsetzung der arithmetischen Operationen ist denkbar einfach und wird in [26, Seite 7] ausführlich dargestellt. Man muss dabei allerdings beachten, dass einige Resultate, wie zum Beispiel $-\infty + \infty$, undefiniert sind.

Mit Hilfe der obigen Definition lässt sich folglich jedes nicht-leere Intervall in der Form

$$[x] \equiv [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \,|\, \underline{x} \leq x \leq \overline{x},\ \ \underline{x}, \overline{x} \in \mathbb{R}_*,\ \ \underline{x} \neq \infty,\ \ \overline{x} \neq -\infty\}$$

darstellen. Zur Vereinfachung einiger Formeln erlauben wir in dieser Arbeit keine Darstellung $[\underline{x}, \overline{x}]$ mit $\underline{x}, \overline{x} \in \mathbb{R}_*$ der leeren Menge. Stattdessen verwenden wir das bekannte Symbol $\emptyset$. Die Menge aller nicht-leeren Intervalle bezeichnen wir im Folgenden mit $\mathbb{IR}_*$.

Für Intervalle wird stets die Schreibweise $[x] = [\underline{x}, \overline{x}]$ verwandt. Dabei heißt $\underline{x}$ *untere Grenze* oder *Infimum* und $\overline{x}$ *obere Grenze* oder *Supremum* des Intervalles $[x]$. Für kompliziertere Intervall-Ausdrücke benutzen wir auch $\inf([x])$ bzw. $\sup([x])$. Ein reelles Intervall $[x]$ ist *beschränkt* genau dann, wenn $\underline{x}, \overline{x} \in \mathbb{R}$ gilt. Die Menge der beschränkten Intervalle bezeichnen wir im Folgenden mit $\mathbb{IR}$. Ein beschränktes Intervall $[x]$ heißt *Punktintervall* oder *dünnes Intervall*, falls $\underline{x} = \overline{x}$ gilt.

Weiterhin definieren für ein beliebiges Intervall $[x] \in \mathbb{IR}_*$ die folgenden Funktionen:

- Mittelpunkt von $[x]$ $\qquad \mathrm{mid}([x]) := \begin{cases} \frac{\underline{x}+\overline{x}}{2} & \text{falls } \underline{x} \neq -\infty \vee \overline{x} \neq \infty \\ 0 & \text{sonst} \end{cases}$,
- Durchmesser von $[x]$ $\qquad \mathrm{diam}([x]) := \overline{x} - \underline{x}$,
- Magnitude von $[x]$ $\qquad |[x]| \equiv \mathrm{mag}([x]) := \max\{|\tilde{x}| \,|\, \tilde{x} \in [x]\}$,
- Mignitude von $[x]$ $\qquad \langle [x] \rangle \equiv \mathrm{mig}([x]) := \min\{|\tilde{x}| \,|\, \tilde{x} \in [x]\}$,
- Innere von $[x]$ $\qquad \mathrm{int}([x]) := \{\tilde{x} \in [x] \,|\, \underline{x} < \tilde{x} < \overline{x}\}$.

Diese Funktionen bilden also ein möglicherweise unbeschränktes Intervall auf $\mathbb{R}_*$ ab. Es sei darauf hingewiesen, dass der Durchmesser eines Intervalles wegen der Forderungen $\underline{x} \neq \infty$ und $\overline{x} \neq -\infty$ stets definiert ist und nicht-negative Werte annimmt. Für den Mittelpunkt schreiben wir auch oft $[\check{x}]$.

Um eine beliebige Menge $\mathcal{R} \in \mathrm{Pot}(\mathbb{R})$ von reellen Zahlen durch ein möglichst enges Intervall einzuschließen, definieren wir nun den so genannten *Box-Operator*

$$\mathrm{box}(\mathcal{R}) := \begin{cases} [\inf(\mathcal{R}), \sup(\mathcal{R})] & \text{wenn } \mathcal{R} \neq \emptyset, \\ \emptyset & \text{sonst.} \end{cases}$$

Da wir in dieser Arbeit mit unbeschränkten Intervallen arbeiten, ist der Box-Operator für alle Elemente aus $\mathrm{Pot}(\mathbb{R})$ definiert. Aus Platzgründen werden wir in einigen Fällen den Box-Operator mit $\square(\mathcal{R})$ abkürzen.

1.2 Vektoren und Matrizen

Im weiteren Verlauf dieser Arbeit werden Vektoren stets in der Form

$$\mathbf{x} := (x_1, \ldots, x_n)^T \in \mathbb{R}^n$$

und Intervallvektoren in der Form

$$[\mathbf{x}] := ([x_1], \ldots, [x_n])^T \in \mathbb{IR}^n_*$$

notiert. Die (Intervall-)Vektoren werden also durch Fettdruck von reellen Zahlen bzw. Intervallen unterschieden. In der Regel werden wir Kleinbuchstaben für die Notation von Vektoren benutzen. An einigen Stellen dieser Arbeit werden wir nur einen Teil des Vektors $\mathbf{x} \in \mathbb{R}^n_*$ benötigen. Es ist daher nützlich den durch Auslassung einer Komponente i mit $1 \leq i \leq n$ entstehenden Vektor durch

$$\mathbf{x}_{\neg i} := (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)^T \in \mathbb{R}^{n-1}_*$$

zu definieren.

Aus geometrischen Gründen werden wir einen Intervallvektor auch als *Box* oder *Suchbox* bezeichnen: Ein beschränkter Intervallvektor $[\mathbf{x}] \in \mathbb{IR}^2$ entspricht einem Rechteck in der Ebene; für $[\mathbf{x}] \in \mathbb{IR}^3$ erhält man einen Quader. Die oben definierten Funktionen für Intervalle definieren wir—soweit dies möglich ist—für Intervallvektoren *komponentenweise*. Wir erhalten

$$\begin{aligned}
\inf([\mathbf{x}]) &:= (\inf([x_1]), \ldots, \inf([x_n]))^T, \\
\sup([\mathbf{x}]) &:= (\sup([x_1]), \ldots, \sup([x_n]))^T, \\
\mathrm{mid}([\mathbf{x}]) &:= (\mathrm{mid}([x_1]), \ldots, \mathrm{mid}([x_n]))^T, \\
\mathrm{diam}([\mathbf{x}]) &:= (\mathrm{diam}([x_1]), \ldots, \mathrm{diam}([x_n]))^T, \\
|[\mathbf{x}]| &:= (|[x_1]|, \ldots, |[x_n]|)^T, \\
\langle[\mathbf{x}]\rangle &:= (\langle[x_1]\rangle, \ldots, \langle[x_n]\rangle)^T.
\end{aligned}$$

Außerdem definieren wir für $\mathcal{R} \in \mathrm{Pot}(\mathbb{R})^n$

$$\mathrm{box}(\mathcal{R}) := (\mathrm{box}(\mathcal{R}_1), \ldots, \mathrm{box}(\mathcal{R}_n))^T.$$

Im Gegensatz zu Vektoren werden Matrizen im Folgenden stets mit Großbuchstaben gekennzeichnet, d.h. wir bezeichnen Matrizen aus $\mathbb{R}^{m \times n}_*$ meist mit $\mathbf{A}$ und Matrizen aus $\mathbb{IR}^{m \times n}_*$ mit

$$[\mathbf{A}] := ([a_{i,j}])_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}}.$$

Die entsprechenden Einträge der Matrix werden weiterhin mit Kleinbuchstaben bezeichnet. Für reelle Matrizen verwenden wir die aus der Linearen Algebra bekannten Operationen wie Norm und Matrix-Matrix-Produkt. Die verwendete Notation hält sich dabei weitgehend an die gängigen Standards.

Für eine differenzierbare Funktion $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ bezeichnet $\mathbf{Df}$ die *Ableitungsmatrix* und $\mathbf{D^2f}$ die *zweite Ableitung*, falls $\mathbf{f}$ zweimal differenzierbar ist. Im Fall $m = 1$ schreiben wir für den Gradienten der Funktion auch $\nabla\mathbf{f}$.

Für Intervallmatrizen übernehmen wir die Begriffe inf, sup, mid, diam und box analog zu den Intervallvektoren, also ebenfalls komponentenweise.

Für die Auswahl bestimmter Spalten bzw. Zeilen aus einer (Intervall-)Matrix verwenden wir eine Matlab-ähnliche Notation. Über die Anweisung

$$[\mathbf{A}]_{j:k,u:v} \equiv \begin{pmatrix} [a_{j,u}] & \dots & [a_{j,v}] \\ \vdots & \ddots & \vdots \\ [a_{k,u}] & \dots & [a_{k,v}] \end{pmatrix} \in \mathbb{IR}_*^{(k-j+1)\times(v-u+1)}$$

wird eine Teilmatrix aus der ursprünglichen Matrix $[\mathbf{A}] \in \mathbb{IR}_*^{m\times n}$ herausgeschnitten. Mit Hilfe der Notation $[\mathbf{A}]_{\neg i}$ wird für $1 \leq i \leq n$ die i-te Spalte der Intervall-Matrix $[\mathbf{A}] \in \mathbb{IR}_*^{m\times n}$ entfernt.

1.3 Intervallarithmetik

Genau wie für reelle Zahlen müssen auch für Intervalle die Standardoperationen und Elementarfunktionen wie z.B. sin, cos, etc. definiert werden. Aus diesem Grund führen wir zunächst die folgende Definition ein.

Definition 1.1 Sei $f : \mathbb{R}^n \to \mathbb{R}$ eine reellwertige Funktion. Dann ist die *kanonische Mengenerweiterung* von f definiert durch die Abbildung $\hat{f}$ mit

$$\begin{aligned} \hat{f} : \mathrm{Pot}(\mathbb{R}^n) &\to \mathrm{Pot}(\mathbb{R}) \\ D &\mapsto \{y \mid \exists \mathbf{x} \in D, f(\mathbf{x}) = y\}. \end{aligned}$$

Es ist offensichtlich, dass $\hat{f}([\mathbf{x}])$ auch dann definiert ist, wenn $\hat{f}$ auf $[\mathbf{x}]$ nicht total ist. Dabei heißt eine Funktion *total* auf $[\mathbf{x}]$, wenn sie für alle $\mathbf{x} \in [\mathbf{x}]$ definiert ist. Zum Beispiel ist $\log(0)$ nicht definiert, aber es gilt $\hat{\log}([0, 0]) = \emptyset$. Die kanonische Mengenerweiterung liefert also den Wertebereich von f für alle Werte $\mathbf{x} \in [\mathbf{x}]$, an denen die Funktion definiert ist. Wie wir später noch genauer sehen werden, wird diese Definition besonders nützlich sein. Zur übersichtlicheren Darstellung schreiben wir meist auch f anstatt $\hat{f}$.

Eine wesentliche Komponente eines verifizierenden Lösers bzw. Optimierers ist es, den Wertebereich einer reellwertigen Funktion auf einer Box zu bestimmen oder zumindest eine möglichst gute Einschließung desselben, wenn sich der exakte Wertebereich nicht darstellen lässt.

Definition 1.2 Sei $f : \mathbb{R}^n \to \mathbb{R}$ eine Funktion. Eine Funktion $[f] : \mathbb{IR}_*^n \to \mathbb{IR}_*$ heißt *Intervall-Einschließung* oder einfach *Einschließung* von f, wenn für alle $[\mathbf{x}] \in \mathbb{IR}_*^n$ gilt

$$\hat{f}([\mathbf{x}]) \subseteq [f]([\mathbf{x}]).$$

Wir werden im Folgenden verschiedene Möglichkeiten zur Berechnung von Einschließungen kennenlernen, die sich teilweise in der Qualität deutlich unterscheiden. Die Notation $[f]([\mathbf{x}])$ soll hier nur *irgendeine* (natürlich korrekte) Einschließung bezeichnen.

Wir wollen nun die Frage klären, wie man Einschließungen des Wertebereiches einer Funktion möglichst exakt berechnen kann. Für die arithmetischen Operationen $+$, $-$, $\cdot$ über $\mathbb{IR}_*$ lassen sich mit Hilfe der Monotonieeigenschaften sehr leicht explizite Formeln angeben. Seien $[a,b]$ und $[c,d]$ zwei beliebige Intervalle. Dann gilt:

$$[a,b] + [c,d] = [a+c, b+d], \tag{1.1}$$

$$[a,b] - [c,d] = [a-d, b-c], \tag{1.2}$$

$$[a,b] \cdot [c,d] = [\min\{ac,ad,bc,bd\}, \max\{ac,ad,bc,bd\}]. \tag{1.3}$$

Diese Formeln findet man in nahezu allen Standardwerken über (beschränkte) Intervallarithmetik. In [27] wird gezeigt, dass diese Formeln ebenfalls für unbeschränkte Intervalle ihre Gültigkeit behalten.

Die Formeln (1.1)-(1.3) haben die Eigenschaft, dass sich ihr exakter Wertebereich stets wieder als Intervall darstellen lässt. Dies ist i.A. nicht mehr möglich, wenn die Funktion auf der Argumentbox nicht stetig ist. Betrachten wir z.B. die Funktion $f(x) = \frac{1}{x}$ auf der Suchbox $[-1,1]$. Man erhält für den Wertebereich $[-\infty,-1] \cup [1,\infty]$. Die kleinste Einschließung für diesen Wertebereich ist allerdings $\mathbb{R}_*$ selber. Nachfolgend ist die Formel für die kanonische Erweiterung der Division angegeben.

$$\frac{1}{[c,d]} = \begin{cases} [1/d, 1/c] & \text{wenn} \quad 0 \notin [c,d] \\ \emptyset & \text{wenn} \quad 0 = c = d \\ [1/d, \infty] & \text{wenn} \quad c = 0, c \neq d \\ [-\infty, 1/c] & \text{wenn} \quad d = 0, c \neq d \\ [-\infty, 1/c] \cup [1/d, \infty] & \text{wenn} \quad c < 0 < d \end{cases} \tag{1.4}$$

$$\frac{[a,b]}{[c,d]} = \begin{cases} [a,b] \cdot [1/d, 1/c] & \text{wenn} \quad 0 \notin [c,d] \\ \emptyset & \text{wenn} \quad c = d = 0 \\ [a/d, \infty] & \text{wenn} \quad a \geq 0, c = 0 \\ [-\infty, \infty] & \text{wenn} \quad a < 0 < b, c = 0 \\ [-\infty, b/d] & \text{wenn} \quad b \leq 0, c = 0 \\ [-\infty, a/c] \cup [a/d, \infty] & \text{wenn} \quad a > 0, c < 0 < d \\ [-\infty, \infty] & \text{wenn} \quad 0 \in [a,b], c < 0 < d \\ [-\infty, b/d] \cup [b/c, \infty] & \text{wenn} \quad b < 0, c < 0 < d \\ [-\infty, a/c] & \text{wenn} \quad a \geq 0, d = 0 \\ [-\infty, \infty] & \text{wenn} \quad a < 0 < b, d = 0 \\ [b/c, \infty] & \text{wenn} \quad b \leq 0, d = 0 \end{cases} \tag{1.5}$$

Für weitere Funktionen lassen sich (z.B. durch Ausnutzung von Monotonieeigenschaften) ähnliche Formeln angegeben. Auf die Angabe dieser verzichten wir an dieser Stelle und verweisen auf [54].

Willems gibt in [66, Seite 30] ein Beispiel dafür, dass die kanonische Erweiterung der Division bei der Auflösung der Multiplikation nach einer Variablen beim so genannten

Constraint-Propagation nicht ausreicht, um die Korrektheit der Ergebnisse zu garantieren. Aus diesem Grund verwenden wir an einigen Stellen die so genannte *relationale Division* $\oslash$ mit

$$[x] \oslash [y] := \{z \mid \exists x \in [x], y \in [y], x = z \cdot y\},$$

die von Ratz [59] für beschränkte Intervalle eingeführt wurde. Hickey et al. [26] erweitern diese auch auf unbeschränkte Intervalle. Für die relationale Division erhält man leicht die folgende Formel.

$$[a,b] \oslash [c,d] := \begin{cases} [a,b] \cdot [1/d, 1/c] & \text{falls} \quad 0 \notin [c,d] \\ [-\infty, \infty] & \text{falls} \quad 0 \in [a,b] \cap [c,d] \\ [b/c, \infty] & \text{falls} \quad b < 0, c < d = 0 \\ [-\infty, b/d] \cup [b/c, \infty] & \text{falls} \quad b < 0, c < 0 < d \\ [-\infty, b/d] & \text{falls} \quad b < 0, 0 = c < d \\ [-\infty, a/c] & \text{falls} \quad 0 < a, c < d = 0 \\ [-\infty, a/c] \cup [a/d, \infty] & \text{falls} \quad 0 < a, c < 0 < d \\ [a/d, \infty] & \text{falls} \quad 0 < a, 0 = c < d \\ \emptyset & \text{falls} \quad 0 \notin [a,b], c = d = 0 \end{cases} \tag{1.6}$$

Wie schon bei der kanonischen Erweiterung der Division liefert auch die relationale Division im Allgemeinen kein Intervall. Aus diesem Grund wird in einer Implementierung ein spezieller Divisionsoperator benötigt. Es gibt verschiedene Möglichkeiten diesen Operator zu realisieren. Wir werden an entsprechender Stelle auf die spezielle Implementierung in SONIC eingehen.

Wie das folgende Lemma zeigt, sollte bei Funktionsauswertungen aber stets der „normale“ Divisionsoperator bevorzugt werden. Den Beweis findet der interessierte Leser in [26].

Lemma 1.1 Seien $[x], [y] \in \mathbb{IR}_*$. Dann gilt

$$[x]/[y] \subseteq [x] \oslash [y].$$

1.4 Eigenschaften der Intervallarithmetik

Ein wesentliches Problem bei der Nutzung der Intervallarithmetik besteht in der Tatsache, dass in vielen Fällen die Rechenregeln für reelle Zahlen nicht auf Intervalle übertragen werden können. So ist zum Beispiel zu beachten, dass Subtraktion bzw. Division nicht mehr die inverse Operation zu Addition bzw. Multiplikation ist. Man sieht leicht:

$$\begin{aligned} [1,2] - [1,2] &= [-1,1] \neq 0, \\ [1,2]/[1,2] &= [0.5,2] \neq 1. \end{aligned}$$

Außerdem gilt für ein Intervall $[x] \in \mathbb{IR}_*$ im Allgemeinen *nicht* mehr $[x] \cdot [x] = [x]^2$, da man bei der Berechnung der Quadrat-Funktion die Eigenschaft ausnutzen kann, dass

der Wertebereich nicht-negativ ist. Wir definieren aus diesem Grund

$$[x]^2 := \begin{cases} [0, |[x]|^2] & 0 \in [x] \\ [\underline{x}^2, \overline{x}^2] & \underline{x} > 0 \\ [\overline{x}^2, \underline{x}^2] & \overline{x} < 0 \end{cases} .$$

Zur Veranschaulichung betrachten wir das folgende Beispiel.

$$\begin{aligned} [x] \cdot [x] &= [-2, 1] \cdot [-2, 1] = [-2, 4] \\ [x]^2 &= [-2, 1]^2 = [0, 4] \end{aligned}$$

Der Grund für dieses Verhalten ist die fehlende Kopplung gleicher Variablen bei der Benutzung von Intervallen. Dies wird in der Literatur häufig als *Dependency-Problem* bezeichnet. Beispielsweise wird bei der Subtraktion $[x] - [x]$ die Menge

$$[x] - [x] = \{x - y \mid x, y \in [x]\}$$

berechnet. Das mehrfache Vorkommen einer Variablen wird also nicht berücksichtigt, stattdessen werden sie völlig unabhängig voneinander behandelt. Dieses Problem kann durchaus als *fundamentales Problem* der Intervallarithmetik betrachtet werden. Wir werden im weiteren Verlauf dieser Arbeit Techniken vorstellen, die dieses Problem behandeln oder es zumindest versuchen.

Für die Multiplikation und Addition der Intervallarithmetik gelten weiterhin das Kommutativ- und das Assoziativgesetz. Das Distributivgesetz hingegen gilt jedoch *nicht* mehr. Es wird durch das so genannte *Subdistributivgesetz*

$$[x] \cdot ([y] + [z]) \subseteq [x][y] + [x][z]$$

ersetzt.

1.5 Rechnerarithmetik

Da auf einem Rechner nur eine endliche Anzahl von so genannten *Maschinenzahlen* zur Verfügung stehen, ist es nicht ausreichend, die oben beschriebenen Formeln bei der Implementierung einer Intervallarithmetik einfach zu übernehmen. Um die Problematik zu verdeutlichen betrachten wir zunächst das folgende

Beispiel 1.1 Seien $[x] = [-0.9999, 0.9999]$ und $[y] = [-0.01111, 0.01111]$ zwei Intervalle. Dann gilt bei exakter Rechnung

$$[x] + [y] = [-1.01101, 1.01101].$$

Auf einem Rechner mit Mantissenlänge 4 zur Basis 10 würde allerdings das falsche Ergebnis

$$[x] + [y] = [-1.011, 1.011]$$

berechnet werden.

Da solche Rundungsfehler unvermeidlich auftreten, ist es von großer Bedeutung trotzdem Einschließungen zu berechnen, die den exakten Wertebereich garantiert enthalten. Eine Möglichkeit zur Lösung dieses Problems bietet das so genannte *gerichtete Runden*. Anstatt die Ergebnisse zur nächsten Maschinenzahl zu runden, wird die untere Intervallgrenze zur nächst kleineren (in Richtung $-\infty$) und die obere Intervallgrenze zur nächst größeren Maschinenzahl (in Richtung ∞) gerundet. Dieses Vorgehen *garantiert*, dass der exakte Wertebereich der Funktion eingeschlossen wird.

Der aktuelle IEEE754-Standard bietet heute die Möglichkeit an, den Rundungsmodus umzustellen. Dadurch lassen sich sehr leicht die oben präsentierten arithmetischen Operationen implementieren. Bei komplexeren Funktionen (Exponentialfunktion, etc.) ist es allerdings notwendig auf Intervallbibliotheken zurückzugreifen, die die Wertebereiche garantiert berechnen. In SONIC sind momentan drei verschiedene Intervallbibliotheken integriert: C-XSC, filib++ und die Sun-Intervallarithmetik. Wir werden auf diese in Kapitel 5.1 noch genauer eingehen.

Der IEEE754-Standard sieht die Möglichkeit vor, mit den erweiterten reellen Zahlen zu rechnen. Da die meisten Bibliotheken keine Möglichkeit zur Behandlung unbeschränkter Intervalle integriert haben, wurde in SONIC eine entsprechende Klasse `xinterval` implementiert. Eine genaue Beschreibung und Details zur Implementierung finden sich in [66].

1.6 Intervallerweiterungen

In diesem Abschnitt werden wir verschiedene Möglichkeiten kennenlernen, wie man mit Hilfe der Intervallarithmetik Einschließungen für den Wertebereich einer Funktion berechnen kann. Die präsentierten Methoden unterscheiden sich teilweise deutlich in der Qualität der berechneten Einschließungen. Da Überschätzungen des Wertebereiches einer Funktion die Laufzeit eines Algorithmus unter Umständen sehr stark negativ beeinflussen können, ist man offensichtlich an möglichst engen Einschließungen der Wertebereiche interessiert.

Um Vergleiche zwischen verschiedenen Intervallerweiterungen bezüglich deren Qualität durchführen zu können, benötigen wir ein geeignetes Maß. Hilfreich sind dabei vor allem die asymptotischen Eigenschaften der Einschließungen, also für $||\text{diam}([\mathbf{x}])|| \to 0$. Wir vereinbaren daher die folgende

Definition 1.3 Sei $[f]([\mathbf{x}])$ eine Intervallerweiterung der Funktion $f : \mathbb{R}^n \to \mathbb{R}$ auf der Box $[\mathbf{x}]$ und bezeichne $f([\mathbf{x}])$ den exakten Wertebereich der Funktion. Gibt es eine von $[\mathbf{x}]$ unabhängige Konstante $\kappa \in \mathbb{R}$, so dass

$$||\text{diam}([f]([\mathbf{x}]))||_\infty - ||\text{diam}(\text{box}(f([\mathbf{x}])))||_\infty \leq \kappa \cdot ||\text{diam}([\mathbf{x}])||_\infty^\alpha$$

für alle $[\mathbf{x}]$ mit genügend kleinem Durchmesser und festem $0 < \alpha \in \mathbb{N}$ gilt, dann heißt $[\mathbf{f}]([\mathbf{x}])$ eine *Intervallerweiterung der Ordnung α für* $\mathbf{f}$.

Es sei an dieser Stelle erwähnt, dass die Definition natürlich nur für beschränkte In-

tervalle Sinn macht. Da wir aber nur am asymptotischen Verhalten der Intervallerweiterung interessiert sind, handelt es sich dabei um keine echte Einschränkung.

1.6.1 Natürliche Intervallerweiterung

Die wohl einfachste Möglichkeit den Wertebereich einer Funktion $f : \mathbb{R}^n \to \mathbb{R}$ über einer Box zu berechnen, besteht darin, in dem Funktionsausdruck für f die darin auftretenden Variablen durch Intervalle anstatt reeller Zahlen zu ersetzen. Anschließend wird der entstandene Ausdruck mit Hilfe der Intervallarithmetik ausgewertet. Diese Methode wird im Allgemeinen als *natürliche Intervallerweiterung* bezeichnet. Die natürliche Intervallerweiterung ist eine Intervallerweiterung der Ordnung 1.

1.6.2 Mittelwert-Form

Mit der natürlichen Intervallerweiterung haben wir bereits eine Möglichkeit kennengelernt, wie man den Wertebereich einer Funktion über einer gegebenen Box einschließen kann. In der Regel wird allerdings der tatsächliche Wertebereich durch diese Intervallerweiterung überschätzt. Die Überschätzung ist dabei oft—vor allem für Boxen mit kleinem Durchmesser—nicht unerheblich. Außer der natürlichen Intervallerweiterung existieren noch andere Möglichkeiten Einschließungen für den Wertebereich von f zu konstruieren. So lässt sich zum Beispiel aus der *Taylorentwicklung erster Ordnung* eine weitere Einschließung ableiten.

Satz 1.1 Sei $U \subseteq \mathbb{R}^n$ konvex und offen, $\mathbf{x}^*$, $\mathbf{c} \in U$ und $f : \mathbb{R}^n \to \mathbb{R}$ auf U stetig differenzierbar. Dann existiert ein $\xi = \mathbf{x}^* + \theta(\mathbf{c} - \mathbf{x}^*)$ mit $\theta \in [0, 1]$, so dass

$$f(\mathbf{x}^*) = f(\mathbf{c}) + \nabla \mathbf{f}(\xi)(\mathbf{x}^* - \mathbf{c}) \tag{1.7}$$

gilt.

Sei also $f : \mathbb{R}^n \to \mathbb{R}$ eine stetig differenzierbare Funktion, sowie $[\mathbf{x}] \in \mathbb{IR}^n_*$ und $\mathbf{c} \in [\mathbf{x}]$. Dann stellt (1.7) eine alternative Darstellung der Funktion f dar. Diese lässt sich nutzen, um eine alternative Einschließung zu berechnen. Da $\xi, \mathbf{x}^* \in [\mathbf{x}]$ gilt, erhält man eine Einschließung des Wertebereiches der Funktion f über $[\mathbf{x}]$ mit Hilfe der *Mittelwert-Form*

$$f([\mathbf{x}]) \subseteq [f](\mathbf{c}) + [\nabla \mathbf{f}]([\mathbf{x}])([\mathbf{x}] - \mathbf{c}) =: \mathcal{T}_1(f, \mathbf{c}, [\mathbf{x}]). \tag{1.8}$$

Der Vektor $\mathbf{c} \in [\mathbf{x}]$ wird dabei als *Referenzpunkt* oder auch als *Zentrum* bezeichnet. Die Gültigkeit der Mittelwert-Form folgt unmittelbar aus Satz 1.1.

Abhängig von der Wahl des Referenzpunktes ergeben sich verschiedene Entwicklungen. Diese Wahl kann die Qualität der Einschließung über die Mittelwert-Form entscheidend beeinflussen. Der folgende Satz liefert eine Möglichkeit, den Referenzpunkt in gewisser Hinsicht *optimal* zu wählen.

Satz 1.2 (Baumann) Sei $z \in \mathbb{R}$ und $[z] \in \mathbb{IR}_*$. Definiere nun

$$\mathrm{cut}(z,[z]) := \begin{cases} \overline{z} & \text{falls } z \geq \overline{z}, \\ \underline{z} & \text{falls } z \leq \underline{z}, \\ z & \text{sonst.} \end{cases}$$

Sei weiterhin $f : \mathbb{R}^n \to \mathbb{R}$ stetig differenzierbar und $[\mathbf{x}] \in \mathbb{IR}^n$. Setze

$$\begin{aligned} p_i &:= \mathrm{cut}\left(\frac{\mathrm{mid}(D_i)}{\mathrm{rad}(D_i)}, [\mathbf{x}]\right), \text{ mit } D_i := \left[\frac{\partial f}{\partial x_i}\right]([\mathbf{x}]), \\ (c_*)_i &:= \mathrm{mid}([x_i]) - p_i \cdot \mathrm{rad}([x_i]), \\ (c^*)_i &:= \mathrm{mid}([x_i]) + p_i \cdot \mathrm{rad}([x_i]). \end{aligned}$$

Dann gilt:

1. $\inf(\mathcal{T}_1(f, \mathbf{c}, [\mathbf{x}]))$ ist maximal für $\mathbf{c} = \mathbf{c}_*$,
2. $\sup(\mathcal{T}_1(f, \mathbf{c}, [\mathbf{x}]))$ ist minimal für $\mathbf{c} = \mathbf{c}^*$,
3. $\mathrm{rad}(\mathcal{T}_1(f, \mathbf{c}, [\mathbf{x}]))$ ist minimal für $\mathbf{c} = \mathrm{mid}([\mathbf{x}])$.

Wählt man folglich den Mittelpunkt von $[\mathbf{x}]$ als Referenzpunkt, so erhält man im Allgemeinen gute Ergebnisse. Ist man allerdings an *optimalen* Einschließungen mit Hilfe der Mittelwert-Form interessiert, so berechnet man $T_1(f, \mathbf{c}_*, [\mathbf{x}])$ und $T_1(f, \mathbf{c}^*, [\mathbf{x}])$ und schneidet im Anschluss die Ergebnisse. Man beachte, dass dafür nur *eine* Auswertung des Gradienten benötigt wird.

1.6.3 Steigungen

Bei genauerer Betrachtung von (1.8) wird deutlich, dass es möglicherweise unnötig ist, eine Einschließung des Gradienten $\nabla\mathbf{f}$ zu verwenden. Stattdessen wäre ein Intervallvektor $[\mathbf{s}]$ völlig ausreichend, so dass zu jedem $\mathbf{x} \in [\mathbf{x}]$ ein Vektor $\mathbf{s} \in [\mathbf{s}]$ existiert mit $f(\mathbf{x}) - f(\mathbf{c}) = \mathbf{s}^T(\mathbf{x} - \mathbf{c})$.

Definition 1.4 Sei $f : \mathbb{R}^n \to \mathbb{R}$. Der Vektor $[\mathbf{s}] \in \mathbb{IR}_*^n$ heißt *Intervall-Steigungsvektor* oder auch *Steigungsmenge* für f über der Box $[\mathbf{x}]$ mit Referenzpunkt $\mathbf{c} \in [\mathbf{x}]$, wenn zu jedem $\tilde{\mathbf{x}} \in [\mathbf{x}]$ ein $s \in [\mathbf{s}]$ existiert mit

$$f(\tilde{\mathbf{x}}) - f(\mathbf{c}) = \mathbf{s}^T(\tilde{\mathbf{x}} - \mathbf{c}). \tag{1.9}$$

Eine minimale derartige Menge bezeichnen wir mit $\mathbf{s}^{\#}(f, [\mathbf{x}], \mathbf{c})$ und Einschließungen von $\mathbf{s}^{\#}$ mit $[\mathbf{s}](f, [\mathbf{x}], \mathbf{c})$.

Demnach ist $\mathbf{s}^{\#}(f, [\mathbf{x}], \mathbf{c})$ die minimal benötigte Menge für eine Linearisierung der Funktion f. Eine Einschließung des Gradienten $[\nabla\mathbf{f}]([\mathbf{x}])$ ist offensichtlich eine Obermenge von $\mathbf{s}^{\#}(f, [\mathbf{x}], \mathbf{c})$. Im Allgemeinen beinhaltet diese allerdings eine wesentliche Überschätzung.

Verwendet man also in (1.8) einen Steigungsvektor [**s**] anstatt einer Einschließung des Gradienten, so ergibt sich folglich eine Einschließung des Wertebereiches über

$$\mathcal{T}(f, \mathbf{c}, [\mathbf{x}], [\mathbf{s}]) := [f](\mathbf{c}) + [\mathbf{s}]^T([\mathbf{x}] - \mathbf{c}). \tag{1.10}$$

Diese Einschließung wird häufig als *zentrierte Form* bezeichnet. Damit ist die Mittelwert-Form offensichtlich auch eine zentrierte Form.

1.6.3.1 Progressive Ableitungen

In der Literatur werden vielen Methoden zur Berechnung von Steigungen angegeben. Eine der wichtigsten und effektivsten Methoden stellen die so genannten *progressiven Ableitungen* [56] dar. Man erhält eine Einschließung des Wertebereiches von f über [**x**] durch

$$f([\mathbf{x}]) \subseteq [f](\mathbf{c}) + \sum_{i=1}^{n} \left[\frac{\partial f}{\partial x_i}\right] ([x_1], \ldots, [x_i], c_{i+1}, \ldots, c_n) \cdot ([x_i] - c_i). \tag{1.11}$$

Der Vorteil dieser Einschließung gegenüber (1.8) liegt offensichtlich in der Tatsache, dass einige (Intervall-)Argumente durch reelle Zahlen ersetzt werden. Dies hat zur Folge, dass die daraus resultierende Einschließung im Allgemeinen wesentlich schärfer ist. Zudem ist es völlig ausreichend, wenn die partiellen Ableitungen nur auf ihren jeweiligen Argumenten stetig sind. Eine ausführlichere Darstellung der progressiven Ableitungen mit einigen Beispielen findet sich in [35].

1.6.3.2 Steigungs-Arithmetik

Eine weitere Möglichkeit zur Berechnung einer Steigungsmenge stellt Ratz in [60] vor. Die so genannte *Steigungs-Arithmetik* basiert auf dem gleichen Prinzip wie das Automatische Differenzieren. Zunächst wird die Funktion in elementare Operationen/Funktionen aufgebrochen. Da für Variablen und Konstanten leicht Steigungsmengen bestimmt werden können, ergibt sich durch die Steigungsarithmetik nach und nach (über die elementaren Kompositionen) eine Steigungsmenge für die Gesamtfunktion. In einer Vorgängerversion von SONIC wurde diese Arithmetik implementiert. Es stellte sich allerdings heraus, dass im Vergleich zur Qualität der Einschließung zu hohen Kosten für deren Berechnung anfallen. Aus diesem Grund wurde in der aktuellen Version von SONIC die Steigungsarithmetik nicht mehr berücksichtigt. Eine genauere Untersuchung zur Qualität der Einschließungen findet der interessierte Leser in [4].

Abschließend sei an dieser Stelle erwähnt, dass die zentrierte Form bzw. die Mittelwert-Form eine Intervallerweiterung der zweiten Ordnung ist.

1.6.4 Taylorentwicklung höherer Ordnung

Weitere Möglichkeiten zur Berechnung von Einschließungen für den Wertebereich einer Funktion basieren auf der Taylorentwicklung höherer Ordnung. Exemplarisch zeigen wir hier noch die Taylorentwicklung zweiter Ordnung.

Satz 1.3 Sei $U \subseteq \mathbb{R}^n$ konvex und offen. Sei zusätzlich $\mathbf{x}^*, \mathbf{c} \in U$ und $f : \mathbb{R}^n \to \mathbb{R}$ auf U zweimal stetig differenzierbar. Dann gibt es ein $\xi = \mathbf{x}^* + \theta(\mathbf{c} - \mathbf{x}^*)$ mit $\theta \in [0, 1]$, so dass

$$f(\mathbf{x}^*) = f(\mathbf{c}) + \nabla\mathbf{f}(\mathbf{c})(\mathbf{x}^* - \mathbf{c}) + (\mathbf{x}^* - \mathbf{c})^T\mathbf{D^2f}(\xi)(\mathbf{x}^* - \mathbf{c}) \tag{1.12}$$

gilt.

Sei also $f : \mathbb{R}^n \to \mathbb{R}$ eine stetig differenzierbare Funktion, sowie $[\mathbf{x}] \in \mathbb{IR}^n_*$ und $\mathbf{c} \in [\mathbf{x}]$. Man erhält folglich eine alternative Einschließung des Wertebereiches über die Formel

$$[f](\mathbf{c}) + [\nabla\mathbf{f}](\mathbf{c})([\mathbf{x}] - \mathbf{c}) + ([\mathbf{x}] - \mathbf{c})^T[\mathbf{D^2f}]([\mathbf{x}])([\mathbf{x}] - \mathbf{c}). \tag{1.13}$$

Für kleine Boxen liefert diese Darstellung in der Regel engere Einschließungen als die Einschließungen, die mit Hilfe der Mittelwert-Form berechnet werden. Allerdings wird die Berechnung der Einschließung deutlich aufwändiger. Ausführliche Untersuchungen in [42] haben gezeigt, dass Taylorentwicklungen höherer Ordnung (größer 2) in der Praxis nicht anwendbar sind. Sie liefern zwar meist gute Einschließungen, aber der *extrem hohe* Zeitaufwand für deren Berechnung steht in keinem Verhältnis zur Qualität.

1.7 Terme und Termoptimierungen

Ein Ansatz zur Verbesserung der Einschließungen basiert auf der Idee, den darstellenden Term *geeignet* umzuformen. Aus diesem Grund haben wir in SONIC einen Parser implementiert, der alle Terme aus einer Datei als *Termbaum* einliest. Die *explizite* Darstellung von Termen ist ein besonderes Merkmal, welches SONIC von den bekannten verifiziert rechnenden Tools unterscheidet.

Ebenso wichtige wie einfache Term-Transformationen, die in einen guten Termoptimierer eingebaut sein müssen, sind Vereinfachungen der Form $x+x = 2x$ oder $x \cdot x = x^2$. Auch wenn diese Manipulationen trivial erscheinen, können sie das Laufzeitverhalten des Tools entscheidend beeinflussen. Zwar könnte man annehmen, dass ein „intelligenter Nutzer“ von SONIC solche Terme gar nicht erst formuliert. Allerdings treten bei der Berechnung von Ableitungstermen unweigerlich solche Teilausdrücke auf.

Es ist unmöglich, dass für das Dependency-Problem eine allgemeine Lösung gefunden wird. Es gibt allerdings verschiedene Ansätze, um bei Funktionen mit mehrfach auftretenden Variablen bessere Einschließungen zu berechnen. Diese Ansätze basieren alle auf den beiden folgenden Grundsätzen:

- Je weniger Variablen in einem Term vorkommen, desto besser wird in der Regel die Qualität der Einschließungen,
- Wenige arithmetische Operationen sind in Termen vorzuziehen, um Rundungsfehler zu reduzieren. Insbesondere sind triviale Identitäten wie z.B. $\exp(\log(x))$ zu vermeiden.

In vielen Fällen ist es jedoch unmöglich zu sagen, ob es sich bei der durchgeführten Term-Transformation tatsächlich um eine Termoptimierung handelt. Wie das folgende

Beispiel verdeutlicht, ist die Güte der Einschließung oft von den aktuellen Argumenten abhängig. Sei $f_1 : \mathbb{R} \to \mathbb{R}$ eine Funktion mit

$$f_1(x) := x^2 - x.$$

Dann ist die Funktion $f_2 : \mathbb{R} \to \mathbb{R}$ mit

$$f_2(x) := x \cdot (x - 1)$$

offensichtlich identisch zu f_1. Betrachten wir nun die beiden Boxen $[x_1] = [0, 1]$ und $[x_2] = [-1, 1]$, dann erhält man mit Hilfe der natürlichen Intervallerweiterung

$$[f_1]([x_1]) = [-1, 1] \supset [f_2]([x_1]) = [-1, 0]$$

und

$$[f_1]([x_2]) = [-1, 2] \subset [f_2]([x_2]) = [-2, 2].$$

In diesem Fall wäre z.B. eine mögliche Vorgehensweise, beide Einschließungen zu berechnen und anschließend diese miteinander zu schneiden. Die Qualität der Einschließungen wird im Allgemeinen dadurch verbessert, allerdings nur auf Kosten von *Rechenzeit*. Auf dem Gebiet der Termoptimierung hat es bereits einige wertvolle Arbeiten gegeben. Die wohl gelungensten findet der interessierte Leser in [24, 63].

Die explizite Darstellung der Terme in SONIC hat auch den Vorteil, dass *Ableitungsterme* durch symbolische Differentiation erzeugt werden können. Für diese Terme können dann wiederum Termvereinfachungen durchgeführt werden. Gewöhnliche Tools nutzen das so genannte *Automatische Differenzieren (AD)* [15] zur Berechnung der Ableitungswerte. Diese Methode liefert aber in der Regel schlechtere Einschließungen, da Termvereinfachungen in diesem Fall nicht durchgeführt werden können. Beim *Reverse Mode* von AD ist der Aufwand zur Berechnung der Ableitungswerte nur ungefähr 3–5 mal höher als der Aufwand für die eigentliche Funktionsauswertung. Trotzdem ist zu erwarten, dass über den Term-basierten Ansatz die Ableitungswerte in vielen Fällen deutlich schneller berechnet werden können, da durch die Termmanipulationen weniger Variablen auftreten und gemeinsame Teilterme nicht mehrfach berechnet werden müssen.

1.8 Notation von Algorithmen

In dieser Arbeit werden Algorithmen in Pseudo-Code wie in Alg. 1.1 angegeben. Alle Anweisungen werden in der Regel in C-ähnlicher Notation oder umgangssprachlich notiert. In der Parameterliste wird nicht zwischen reinen Eingabegrößen, zu modifizierenden Variablen, etc. unterschieden. Die genaue Verwendung der Parameter im Algorithmus ist in allen Fällen sehr leicht zu erkennen. Manchmal verzichten wir ganz auf die Verwendung von Parametern. In diesem Fall soll nur ein algorithmisches Prinzip verdeutlicht werden. Die Anweisungen in Schleifenrümpfen und in den Zweigen von Fallunterscheidungen werden durch Einrückungen gekennzeichnet. Dabei wird eine **for**-Schleife durch **end for** abgeschlossen, etc.. Tritt in einer **for**- oder **while**-Schleife

das Kommando **cycle** auf, so wird der Code bis zum Ende der Schleife übersprungen und die nächste Iteration durchgeführt. Die Anweisung **break** führt zum Verlassen der Schleife. Ansonsten erfolgt die Notation der Algorithmen nach den gängigen Regeln und sollte für jeden Leser verständlich sein.

Algorithmus 1.1 NAME($x, \ldots$)

Beschreibung: funktionale Beschreibung des Algorithmus

```
1: for variable = start; variable < end; ... do
2:     Anweisung(en)
3: end for
4: while Bedingung do
5:     Anweisung(en)
6: end while
7: if Bedingung then
8:     Anweisung(en)
9: else
10:    Anweisung(en)
11: end if
```

Kapitel 2

Nichtlineare Gleichungssysteme

Ein wichtiges Ziel dieses Kapitels ist die Einführung und die Präsentation der effizienten symbolischen und numerischen Techniken für das *verifizierte* Lösen von nichtlinearen Gleichungssystemen, die in die aktuelle Version von SONIC integriert sind.

Für die Funktion $\mathbf{f} : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ und die Box $[\mathbf{x}^{(0)}] \subseteq D$ betrachten wir das nichtlineare, reelle Gleichungssystem

$$\mathbf{f}(\mathbf{x}^*) = \mathbf{0} \tag{2.1}$$

mit $\mathbf{x}^* \in [\mathbf{x}^{(0)}]$. Die gegebene Startbox kann dabei möglicherweise unbeschränkt sein. Die Behandlung von unbeschränkten Boxen wird momentan von keinem anderen Tool zum verifizierten Lösen von Gleichungssystemen unterstützt. Wie wir bereits in der Motivation gesehen haben, können viele Probleme aus der Praxis nur sinnvoll modelliert werden, wenn unbeschränkte Intervalle zugelassen werden, da für einige Parameter keine Schranken bekannt sind. Dieser Fakt hebt SONIC bereits aus der Menge der Gleichungslöser heraus, weil man ohne größere Umformulierungen der Systemgleichungen ausschließlich mit diesem Tool bestimmte Systeme aus der Verfahrenstechnik lösen kann.

Setzt man Intervallarithmetik zur verifizierten Lösung von nichtlinearen Gleichungssystemen ein, so ergibt sich immer das Problem, dass man als Resultat Boxen, also nur *Einschließungen von Lösungen*, erhält. Aus diesem Grund müssen wir unsere ursprüngliche Problemstellung leicht umformulieren:

> Bestimme für ein vorgegebenes $\varepsilon \in \mathbb{R}^n_+$ und jedes $\mathbf{x}^* \in [\mathbf{x}^{(0)}]$ mit $\mathbf{f}(\mathbf{x}^*) = \mathbf{0}$ eine Box $[\mathbf{x}]$ mit folgenden Eigenschaften:
>
> - $\mathbf{x}^* \in [\mathbf{x}]$,
> - $\operatorname{diam}([x_i]) \leq \varepsilon_i$ für alle $1 \leq i \leq n$.

(2.2)

Aus diesem Grund liefert ein verifizierter nichtlinearer Löser typischerweise eine Liste von kleinen Boxen, die die tatsächliche Lösungsmenge des Gleichungssystems

überdecken. Ein weiteres Ziel dieses Kapitels ist es, Aussagen über die Existenz und Eindeutigkeit von Lösungen in jeder dieser *kleinen* Boxen machen zu können. Dieser Prozess wird meist auch mit dem Begriff der *Verifikation* von Lösungen bezeichnet. Es ist zu beachten, dass man möglicherweise Boxen erhält, die zwar die Genauigkeitsanforderung erfüllen und deshalb in die Lösungsliste eingefügt werden, die aber tatsächlich gar keine Lösung des Gleichungssystems enthalten.

2.1 Der Branch–and–Bound-Algorithmus

Die in SONIC integrierte Strategie zur verifizierten Lösung von nichtlinearen Gleichungssystemen basiert auf dem so genannten Branch–and–Bound-Ansatz. Man zerlegt bei dieser Strategie ein gegebenes Problem sukzessive in kleinere Teilprobleme (Branching). Der aus dieser Aufteilung resultierende Baum, den wir schematisch in Abbildung 2.1 dargestellt haben, wird überall dort, wo dieses möglich ist, beschnitten (Bounding). Es werden also einige Teilprobleme nicht weiter unterteilt und auch nicht weiter bearbeitet, da sie entweder gelöst sind oder *mit Sicherheit* nicht zu einer Lösung führen. Das Bounding ist sehr wichtig, um den Aufwand zur Lösung des Problems zu begrenzen. Ein wichtiges Ziel ist es, das Bounding möglichst früh im Baum durchzuführen, um möglichst wenige Boxen auf unteren Rekursionsleveln bearbeiten zu müssen: Kann eine Box nicht auf Rekursionslevel i verworfen werden, so müssen—je nach Unterteilungsstrategie—auf dem nächsten Rekursionslevel $i+1$ mindestens zwei weitere Boxen betrachtet werden. Übertragen auf die verifizierte Lösung von nichtlinearen Gleichungssystemen bedeutet dies, dass die gegebene Startbox, in der *alle* Lösungen zu suchen sind, sukzessive immer weiter unterteilt wird. Teilboxen, die definitiv keine Lösungen enthalten können, müssen nicht weiter betrachtet werden.

Die Grundstruktur des in SONIC verwendeten Ansatzes wird in Alg. 2.1 skizziert. Der Algorithmus verwaltet eine aktuelle Liste $\mathcal{L}$ von Boxen, welche zu Beginn mit der Startbox $[\mathbf{x}^{(0)}]$ initialisiert wird, und liefert eine Liste $\mathcal{R}$ von kleinen Boxen, so dass diese Boxen *alle* Lösungen des Gleichungssystems $\mathbf{f}(\mathbf{x}^*) = \mathbf{0}$ in $[\mathbf{x}^{(0)}]$ überdecken. Insbesodere hat $\mathbf{f}$ in $[\mathbf{x}^{(0)}]$ garantiert keine Nullstelle, falls $\mathcal{R} = \emptyset$.

Zu Beginn der while-Schleife wird die erste Box $[\mathbf{x}]$ aus der Liste $\mathcal{L}$ entnommen und eine Einschliessung $[\mathbf{f}]([\mathbf{x}])$ des Wertebereiches der Funktion $\mathbf{f}$ auf dieser Box berechnet. Falls nun $0 \notin [f_i]([\mathbf{x}])$ für ein i gilt, dann kann die Funktion $\mathbf{f}(\mathbf{x})$ keine Nullstelle in $[\mathbf{x}]$ besitzen und die Box kann verworfen werden. Dieses Verfahren wird in der Regel als *direkter Auswertungstest* bezeichnet. Anderenfalls werden weitere so genannte *Kontraktoren* auf die Box angewendet, um die Box zu verkleinern ohne dabei Lösungen zu verlieren. Die Implementierung der unterschiedlichen Kontraktionsverfahren ist in die Routine CheckBox (Zeile 9) integriert. Ist die Box nun „klein genug“, d.h. haben alle Boxkomponenten einen Durchmesser kleiner als eine vorgegebene Größe, so wird die Box in die potenzielle Lösungliste $\mathcal{R}$ eingefügt. Ist die Genauigkeitsanforderung noch nicht erfüllt, so muss das Branching durchgeführt werden: Man unterteilt die aktuelle Box $[\mathbf{x}]$ in zwei Teilboxen $[\mathbf{x}^{(1)}]$ und $[\mathbf{x}^{(2)}]$ und fügt diese anschließend *geeignet* in die Liste $\mathcal{L}$ ein (auf die Bedeutung von „geeignet“ wird in Abschnitt 2.5 eingegangen).

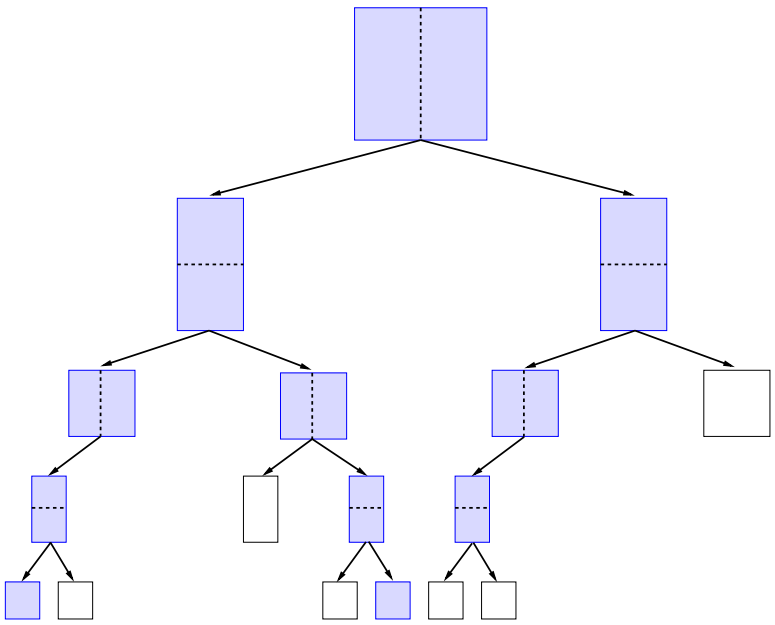

Abbildung 2.1: Baum, der durch Unterteilung (Branching) der Startbox $[\mathbf{x}^{(0)}]$ entsteht. Alle Boxen, die potenzielle Lösungen enthalten, sind schattiert. Das frühzeitige Beschneiden des Baumes verhindert, dass zu viele Boxen auf unteren Rekursionsleveln bearbeitet werden müssen. Bei zwei Boxen (schattierte Blattknoten) konnten Lösungen verifiziert werden.

Die bereits oben erwähnten Kontraktionsverfahren spielen bei der verifizierten Lösung von nichtlinearen Gleichungssystemen (sowie bei der globalen Optimierung) eine entscheidende Rolle. Ohne diese Verfahren wäre der Branch–and–Bound-Algorithmus nicht effektiv genug, um sich in der Praxis als nützlich zu erweisen.

Definition 2.1 Eine Funktion $\mu_{\mathbf{f}} : \mathbb{IR}^n_* \to \mathbb{IR}^n_*$ heißt *Kontraktor* für das Gleichungssystem $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ genau dann, wenn für alle $[\mathbf{x}] \in \mathbb{IR}^n_*$ gilt:

$$\mu_{\mathbf{f}}([\mathbf{x}]) \subseteq [\mathbf{x}].$$

Ein Kontraktor heißt *zulässig* genau dann, wenn für jedes $\mathbf{x}^*$ mit $\mathbf{f}(\mathbf{x}^*) = \mathbf{0}$ und $\mathbf{x}^* \in [\mathbf{x}]$ gilt: $\mathbf{x}^* \in \mu_{\mathbf{f}}([\mathbf{x}])$. Man spricht in diesem Fall auch von einem *zulässigen Kontraktionsverfahren.*

Ein zulässiger Kontraktor ist also ein Verfahren, um eine gegebene Box $[\mathbf{x}]$ möglichst stark zu verkleinern, ohne dabei Lösungen des Gleichungssystems zu verlieren. Ohne es zu wissen, haben wir bereits einen zulässigen Kontraktor in den grundlegenden Branch–and–Bound-Ansatz integriert: die natürliche Intervallerweiterung bzw. der direkte Auswertungstest. Allerdings liefert dieser zulässige Kontraktor nicht das, was man von einem „guten Kontraktor“ erwartet, nämlich eine Verbesserung der Einschließungen.

Die Implementierung der Kontraktionsverfahren erfolgt in SONIC in der Regel innerhalb der Routine CheckBox. Deshalb gehen wir im Folgenden zunächst davon

Algorithmus 2.1 GRUNDLEGENDER BRANCH–AND–BOUND-ANSATZ

Beschreibung: Branch–and–Bound-Ansatz für die Startbox $[\mathbf{x}^{(0)}]$ mit einer vorgegebenen Genauigkeitsanforderung ε. Potenzielle Lösungen werden in der Liste $\mathcal{R}$ gespeichert.

1: $\mathcal{L} = [\mathbf{x}^{(0)}]$; $\mathcal{R} = \emptyset$
2: **while** $\mathcal{L} \neq \emptyset$ **do**
3: entferne die erste Box $[\mathbf{x}]$ aus $\mathcal{L}$
4: **if** $\mathbf{0} \notin [\mathbf{f}]([\mathbf{x}])$ **then**
5: {Box kann verworfen werden}
6: **cycle**
7: **end if**
8: ...
9: CHECKBOX$([\mathbf{x}], \mathcal{L}, \mathcal{R})$ {Anwendung weiterer Kontraktionsverfahren}
10: ...
11: **if** $\operatorname{diam}([x_j]) \leq \varepsilon_j$ für alle $1 \leq j \leq n$ **then**
12: füge die Box $[\mathbf{x}]$ in $\mathcal{R}$ ein
13: **cycle**
14: **end if**
15: wähle eine Komponente j mit $1 \leq j \leq n$ und einen Punkt $\tilde{x} \in [x_j]$
16: $[\mathbf{x}^{(1)}] = ([x_1], \ldots, [x_{j-1}], [\underline{x}_j, \tilde{x}], [x_{j+1}], \ldots, [x_n])$
17: $[\mathbf{x}^{(2)}] = ([x_1], \ldots, [x_{j-1}], [\tilde{x}, \overline{x}_j], [x_{j+1}], \ldots, [x_n])$
18: füge $[\mathbf{x}^{(1)}]$ und $[\mathbf{x}^{(2)}]$ geeignet in $\mathcal{L}$ ein
19: **end while**

aus, dass sich der grundlegende Aufbau des Branch–and–Bound-Algorithmus *nicht* ändert. In einigen Abschnitten dieses Kapitels werden wir numerische Experimente durchführen, damit auch für den Leser stets nachvollziehbar ist, warum die gewählten Techniken in die Standardeinstellungen von SONIC integriert wurden. Bei diesen Experimenten legen wir immer den Algorithmus 2.1 zu Grunde. Die Routine CHECKBOX wird im Laufe dieses Kapitels mehrfach ergänzt und verbessert. Für numerische Experimente verwenden wir stets die letzte präsentierte Variante. Dieses Vorgehen hat den Vorteil, dass die Präsentation einigermaßen übersichtlich bleibt und große Teile des Algorithmus nicht ständig wiederholt werden müssen. Am Ende dieses Kapitels werden wir dann den endültigen, implementierten Algorithmus vorstellen.

2.2 Lineare Intervallgleichungssysteme

In diesem Abschnitt untersuchen wir zunächst Möglichkeiten für die verifizierte Lösung linearer Intervallgleichungssysteme der Form

$$[\mathbf{A}]([\mathbf{x}] - \tilde{\mathbf{x}}) = [\mathbf{b}] \tag{2.3}$$

mit $[\mathbf{A}] \in \mathbb{IR}_*^{m \times n}$, $[\mathbf{x}] \in \mathbb{IR}_*^n$, $\tilde{\mathbf{x}} \in [\mathbf{x}]$ und $[\mathbf{b}] \in \mathbb{IR}_*^m$. Die vorgestellten Verfahren werden auch bei der Lösung von nichtlinearen Gleichungssystemen eine entscheidende

Rolle spielen. Das Intervallgleichungssystem (2.3) definiert eine Menge von linearen Gleichungssystemen, deren Lösungsmenge in $[\mathbf{x}]$ wir mit

$$\sum([\mathbf{A}],[\mathbf{b}],[\mathbf{x}],\tilde{\mathbf{x}}) := \{\mathbf{x}\in[\mathbf{x}] \mid \exists \mathbf{A}\in[\mathbf{A}], \mathbf{b}\in[\mathbf{b}] \text{ mit } \mathbf{A}(\mathbf{x}-\tilde{\mathbf{x}})=\mathbf{b}\}$$

bezeichnen.

Eine weit verbreitete Methode zur Lösung des Systems (2.3) ist das iterative Intervall-Gauß-Seidel-Verfahren. Die Idee des Verfahrens ist es, jeweils die k-te Gleichung von (2.3) nach der k-ten Variablen aufzulösen, um so eine Verbesserung der Einschließung für $[x_k]$ zu erzielen. Multipliziert man das ursprüngliche Gleichungssystem (2.3) aus, so erhält man für $k=1,\ldots,m$

$$\sum_{j=1}^{n}[a_{k,j}]\cdot([x_j]-\tilde{x}_j)=[b_k]. \tag{2.4}$$

Diese m Gleichungen können nun sehr leicht mit Hilfe des linearen Kontraktors aus dem folgenden Korollar (siehe [66, Seite 31]) nach den entsprechenden Variablen aufgelöst werden.

Korollar 2.1 Sei $[\mathbf{a}],[\mathbf{x}]\in\mathbb{IR}_*^n$ und $[b]\in\mathbb{IR}_*$. Dann ist für die lineare Intervallgleichung

$$\sum_{j=1}^{n}[a_j][x_j]=[b]$$

die Vorschrift

$$[x_i']:=[x_i]\cap\left\{\left([b]-\sum_{\substack{j=1\\ j\neq i}}^{n}[a_j][x_j]\right)\oslash[a_i]\right\},\quad i=1,\ldots,n \tag{2.5}$$

ein zulässiges Kontraktionsverfahren.

Man erhält folglich ein zulässiges Kontraktionsverfahren für das lineare Intervallgleichungssystem (2.3) mit Hilfe des Operators

$$\begin{aligned}\Gamma([\mathbf{A}],[\mathbf{x}],[\mathbf{b}]):\mathbb{IR}_*^n &\to \mathbb{IR}_*^n\\ ([x_1],\ldots,[x_n]) &\mapsto ([x_1'],\ldots,[x_n']),\end{aligned}$$

definiert über die Verfahrensvorschrift

$$[x_i']=[x_i]\cap\left\{\tilde{x}_i+\left(\left([b_i]-\sum_{\substack{j=1\\ j\neq i}}^{n}[a_{i,j}]\cdot([x_j]-\tilde{x}_j)\right)\oslash[a_{i,i}]\right)\right\} \tag{2.6}$$

für $i=1,\ldots,\min\{m,n\}$. Gibt es weniger Gleichungen als Variablen, also im Fall $m<n$, so werden keine verbesserten Einschließungen für die entsprechenden Variablen berechnet.

In der Literatur wird dieses Verfahren stets nur für quadratische Systeme angegeben. Meiner Meinung nach stellt dies eine völlig unnötige Einschränkung dar: Man ist auch bei einem überbestimmten System, also für $m > n$, trotzdem in der Lage, die ersten n Gleichungen nach den zugehörigen Variablen aufzulösen.

Dem aufmerksamen Leser wird an dieser Stelle nicht entgangen sein, dass es sich bei der Formulierung (2.6) zunächst nur um das bekannte *Jacobi-Verfahren* oder auch *Gesamtschrittverfahren* handelt. Eine weitere Verbesserung des Verfahrens ergibt sich, wenn die einzelnen Komponenten von $[\mathbf{x}]$ in der natürlichen Reihenfolge für $k = 1, \ldots, n$ verbessert werden. Auf diese Weise kann man im k-ten Schritt die bereits verbesserten Komponenten $[x_j]$ für $j = 1, \ldots, k-1$ einsetzen. Erst durch dieses Vorgehen erhalten wir das bekannte *Gauß-Seidel-Verfahren* oder auch *Einzelschrittverfahren*.

Bei der Verwendung der relationalen Division zum Auflösen nach einer Variablen kann eine so genannte *Lücke* entstehen, d.h. die Einschließung einer Boxkomponente besteht dann aus der Vereinigung (mindestens) zweier disjunkter Intervalle. Wie wir später noch sehen werden, können die enstandenen Lücken für weitere Verfahren von Nutzen sein und werden deshalb separat gespeichert. Aus diesem Grund wurde in SONIC für eine Box $[\mathbf{x}]$ die Möglichkeit realisiert, intern mit einer beliebig langen, aber natürlich endlichen, Vereinigung von Intervallen zu arbeiten. Im eigentlichen Intervall-Gauß-Seidel-Verfahren wird allerdings aus Effizienzgründen die vorhandene Information über Lücken in der aktuellen Box nicht weiter ausgenutzt und mit der konvexen Vereinigung der Intervalle weitergerechnet. Es ergibt sich daraus der nachfolgende Algorithmus.

Algorithmus 2.2 GS($[\mathbf{A}], [\mathbf{x}], [\mathbf{b}], \tilde{\mathbf{x}}, [\mathbf{x}']$)

Beschreibung: Das Intervall-Gauß-Seidel-Verfahren für das lineare Gleichungssystem $[\mathbf{A}]([\mathbf{x}]-\tilde{\mathbf{x}}) = [\mathbf{b}]$. Das Resultat wird in $[\mathbf{x}']$ geschrieben. Vorhandene Lücken werden in der implementierten Datenstruktur für die Box gespeichert.

1: **for** $k = 1, \ldots, \min\{m, n\}$ **do**

2: $$[x'_k] = \tilde{x}_k - \square \left\{ \left(-[b_k] + \sum_{j=1}^{k-1} [a_{k,j}]([x'_j] - \tilde{x}_j) + \sum_{j=k+1}^{n} [a_{k,j}]([x_j] - \tilde{x}_j) \right) \oslash [a_{k,k}] \right\}$$

3: **if** $[x'_k] \cap [x_k] = \emptyset$ **then**

4: **exit**{$[\mathbf{x}]$ kann keine Lösung enthalten}

5: **else**

6: $[x'_k] = [x'_k] \cap [x_k]$

7: **end if**

8: **end for**

2.2.1 Präkonditionierer

In der Praxis spielt das Intervall-Gauß-Seidel-Verfahren seine Stärken in den meisten Fällen erst im Zusammenspiel mit der so genannten *Präkonditionierung* aus. Dazu wird das gegebene Gleichungssystem (2.3) mit einer reellen Matrix $\mathbf{C} \in \mathbb{R}^{n \times m}$—dem Präkonditionierer—von links multipliziert. Daraus ergibt sich das präkonditionierte

Gleichungssystem

$$(\mathbf{C}[\mathbf{A}]) \cdot ([\mathbf{x}] - \tilde{\mathbf{x}}) = \mathbf{C} \cdot [\mathbf{b}]. \tag{2.7}$$

Für ein quadratisches System, also $m = n$, und einen regulären Präkonditionierer $\mathbf{C}$ kann man zeigen, dass das modifizierte System (2.7) und das Originalsystem (2.3) äquivalent sind, also die gleiche Lösungsmenge besitzen. Es gilt aber in jedem Fall, dass die Lösungsmenge des präkonditionierten Systems eine Obermenge der Lösungsmenge des Originalsystems ist. Daraus ergibt sich, dass das Intervall-Gauß-Seidel-Verfahren für das Gleichungssystem (2.7) auch bei nicht-regulären Präkonditionierern ein zulässiges Kontraktionsverfahren darstellt.

2.2.1.1 Inverse-Midpoint-Präkonditionierer

Die wesentliche Aufgabe eines Präkonditionierers beim Intervall-Gauß-Seidel-Verfahren ist es, das Originalsystem so umzuformen, dass der *wesentliche Einfluss* der Variablen $[x_k]$ gerade in der Gleichung k auftritt, da nur nach dieser Variablen aufgelöst wird. Der wohl bekannteste und auch am häufigsten verwendete Präkonditionierer ist der *Inverse-Midpoint-Präkonditionierer*, den wir im Folgenden mit $\mathbf{C}^{\mathrm{IM}}$ abkürzen. Betrachten wir zunächst den Spezialfall der quadratischen Systeme. Dann ist der Inverse-Midpoint-Präkonditionierer wie folgt definiert

$$\mathbf{C}^{\mathrm{IM}} = (\mathrm{mid}([\mathbf{A}]))^{-1}, \tag{2.8}$$

sofern die Inverse der Mittelpunkts-Matrix existiert. Eine detailliertere Beschreibung des Inverse-Midpoint-Präkonditioniers findet der interessierte Leser z.B. in der Diplomarbeit von Paul Willems [66]. Grundsätzlich ist dieser Präkonditionierer natürlich auch auf nicht quadratische Systeme erweiterbar. Wir werden auf diesen Spezialfall zu einem späteren Zeitpunkt eingehen.

2.2.1.2 Optimale Präkonditionierer

Viele Jahre ging man davon aus, dass der Inverse-Midpoint-Präkonditionierer der bestmögliche Präkonditionierer für das Intervall-Gauß-Seidel-Verfahren ist. In [36] zeigt Kearfott allerdings, dass dieser Präkonditionierer in einigen Situationen nur ungenügende Ergebnisse liefert und stellt eine Möglichkeit vor, Präkonditionierer zu berechnen, die in gewisser Hinsicht *optimal* für das Intervall-Gauß-Seidel-Verfahren sind.

Für eine einfachere Darstellung vereinbaren wir zunächst, dass $\mathbf{C_k}$ die k-te Zeile des Präkonditionierers $\mathbf{C}$, also

$$\mathbf{C_k} = (c_{k,1}, \ldots, c_{k,m})$$

und $[\mathbf{A_j}]$ die j-te Spalte der Intervallmatrix $[\mathbf{A}]$, also

$$[\mathbf{A_j}] = ([a_{1,j}], \ldots, [a_{m,j}])^T$$

Algorithmus 2.3 PGS($[\mathbf{A}], [\mathbf{b}], [\mathbf{x}], \tilde{\mathbf{x}}$)

Beschreibung: Präkond. Intervall-Gauß-Seidel-Verfahren für $[\mathbf{A}]([\mathbf{x}] - \tilde{\mathbf{x}}) = [\mathbf{b}]$

1: **for** $k = 1, \ldots, \min\{m, n\}$ **do**
2: Berechne Präkonditionierungszeile $\mathbf{C_k}$
3: $$[x'_k] = \tilde{x}_k - \square \left\{ \left(-\mathbf{C_k}[\mathbf{b}] + \sum_{\substack{j=1 \\ j \neq k}}^{n} (\mathbf{C_k}[\mathbf{A_j}])([x_j] - \tilde{x}_j) \right) \oslash (\mathbf{C_k}[\mathbf{A_k}]) \right\}$$
4: **if** $[x'_k] \cap [x_k] = \emptyset$ **then**
5: **exit**{$[\mathbf{x}]$ kann keine Lösung enthalten}
6: **else**
7: $[x_k] = [x'_k] \cap [x_k]$
8: **end if**
9: **end for**

bezeichnet. Betrachtet man nun einen Intervall-Gauß-Seidel-Schritt für die k-te Komponente des präkonditionierten linearen Gleichungssystems (2.7), so ergibt sich:

$$[x'_k] = \tilde{x}_k - \left\{ -\mathbf{C_k}[\mathbf{b}] + \sum_{j=1}^{k-1} (\mathbf{C_k}[\mathbf{A_j}])([x'_j] - \tilde{x}_j) + \sum_{j=k+1}^{n} (\mathbf{C_k}[\mathbf{A_j}])([x_j] - \tilde{x}_j) \right\} \oslash (\mathbf{C_k}[\mathbf{A_k}]). \tag{2.9}$$

Datiert man im präkonditionierten Gauß-Seidel-Verfahren in jedem Schritt das Intervall $[x_k]$ mit $[x'_k] \cap [x_k]$ auf, so vereinfacht sich die obige Gleichung zu:

$$[x'_k] = \tilde{x}_k - \left\{ -\mathbf{C_k}[\mathbf{b}] + \sum_{\substack{j=1 \\ j \neq k}}^{n} (\mathbf{C_k}[\mathbf{A_j}])([x_j] - \tilde{x}_j) \right\} \oslash (\mathbf{C_k}[\mathbf{A_k}]). \tag{2.10}$$

Bereits an dieser Stelle wird klar, dass wir beim Inverse-Midpoint-Präkonditionierer eine wichtige Eigenschaft des präkonditionierten Intervall-Gauß-Seidel-Verfahrens noch nicht ausgenutzt haben: Für die Berechnung der Komponente $[x'_k]$ benötigt man *ausschließlich* die k-te Zeile der Präkonditionierungsmatrix $\mathbf{C}$. Man muss also nicht notwendigerweise vor dem Verfahren eine *vollständige* Präkonditionierungsmatrix berechnen, sondern kann diese *zeilenweise* entwickeln. Aus diesem Grund spricht man bei den in diesem Abschnitt vorgestellten Präkonditionierern auch oft von *zeilenweisen Präkonditionierern*. Zusammenfassend ergibt sich somit das in Alg. 2.3 dargestellte *präkonditionierte Intervall-Gauß-Seidel-Verfahren*.

Willems entwickelt in [66] ein Verfahren, um den Inverse-Midpoint-Präkonditionierer auch effizient zeilenweise zu berechnen. Damit ist man in der Lage, im Intervall-Gauß-Seidel-Verfahren für *jede Zeile* einen *individuellen* Präkonditionierer zu wählen. Wie wir später noch sehen werden, wird gerade diese Beobachtung die Performance von SONIC deutlich positiv beeinflussen.

Um die Darstellung im Folgenden zu vereinfachen, kürzen wir den Zähler der relationalen Division aus Gleichung (2.10) mit $[n_k](\mathbf{C_k})$ und den Nenner mit $[d_k](\mathbf{C_k})$ ab. Damit erhalten wir

$$\begin{aligned}[x'_k] &= \tilde{x}_k - [n_k](\mathbf{C_k}) \oslash [d_k](\mathbf{C_k}) \\ &\equiv \tilde{x}_k - [\underline{n}_k(\mathbf{C_k}), \overline{n}_k(\mathbf{C_k})] \oslash [\underline{d}_k(\mathbf{C_k}), \overline{d}_k(\mathbf{C_k})]. \end{aligned} \tag{2.11}$$

Novoa entwickelt in seiner Dissertation [55] eine vollständige Klassifikation der zeilenweisen Präkonditionierer für das Intervall-Gauß-Seidel-Verfahren. Diese Klassifikation basiert auf den folgenden beiden Definitionen.

Definition 2.2 Die Präkonditionierungszeile $\mathbf{C_k}$ heißt *C-Präkonditionierer* genau dann, wenn $0 \notin [d_k(\mathbf{C_k})]$. Gilt weiterhin $\underline{d}_k(\mathbf{C_k}) = 1$, so heißt $\mathbf{C_k}$ ein *normaler C-Präkonditionierer*.

Definition 2.3 Die Präkonditionierungszeile $\mathbf{C_k}$ heißt *S-Präkonditionierer* genau dann, wenn $0 \in [d_k(C_k)]$ und $0 \notin [n_k(C_k)]$. Gilt weiterhin $\underline{n}_k(\mathbf{C_k}) = 1$, so heißt $\mathbf{C_k}$ ein *normaler S-Präkonditionierer*.

Die Namensgebung rührt von der Tatsache, dass ein C-Präkonditionierer das Intervall $[x_k]$ kontrahieren (engl.: *contract*) und ein S-Präkonditionierer das Intervall $[x_k]$ mit Hilfe der erweiterten Intervallarithmetik unterteilen (engl.: *split*) soll. Es ist offensichtlich, dass ein sinnvoller zeilenweiser Präkonditionierer entweder ein C- *oder* S-Präkonditionierer sein muss. Anderenfalls wäre $0 \in [d_k(C_k)]$ und $0 \in [n_k(C_k)]$ und damit $[n_k(C_k)] \oslash [d_k(C_k)] = [-\infty, \infty]$. In diesem Fall ist also keine *Kontraktion* des Intervalles $[x_k]$ mit einem Gauß-Seidel-Schritt möglich.

Während die Existenz von S-Präkonditionierern sehr schwierig zu zeigen ist, gilt für C-Präkonditionierer das folgende Lemma.

Lemma 2.1 (Hu, [30]) Ein C-Präkonditionierer $\mathbf{C_k}$ existiert genau dann, wenn mindestens ein Element aus der k-ten Spalte von $[\mathbf{A}]$ die Null nicht enthält.

Um die Darstellung im folgenden Abschnitt vereinfachen zu können, definieren wir an dieser Stelle die so genannte Äquivalenz verschiedener Präkondierungszeilen.

Definition 2.4 Zwei Präkonditionierungszeilen $\mathbf{C_k}$ und $\mathbf{C'_k}$ heißen *äquivalent*, wenn es eine reelle Zahl $0 \neq \alpha \in \mathbb{R}$ mit $\mathbf{C_k} = \alpha \cdot \mathbf{C'_k}$ gibt.

Für C- und S-Präkonditionierer folgt aus dieser Definition unmittelbar das folgende Korollar.

Korollar 2.2 Für jeden C- und S-Präkonditionierer existiert ein äqivalenter normaler C- bzw. S-Präkonditionierer.

Beweis: Sei $\mathbf{C'_k}$ ein C-Präkonditionierer mit $\underline{d}_k(\mathbf{C'_k}) \neq 1$.

1. Gilt $\underline{d}_k(\mathbf{C}'_\mathbf{k}) > 0$, so wähle

$$\mathbf{C_k} := \frac{1}{\underline{d}_k(\mathbf{C}'_\mathbf{k})} \cdot \mathbf{C}'_\mathbf{k}.$$

2. Gilt $\overline{d}_k(\mathbf{C}'_\mathbf{k}) < 0$, so wähle

$$\mathbf{C_k} := \frac{1}{\overline{d}_k(\mathbf{C}'_\mathbf{k})} \cdot \mathbf{C}'_\mathbf{k}.$$

Dann ist $\mathbf{C_k}$ offensichtlich ein normaler C-Präkonditionierer. Der Beweis für den S-Präkonditionierer erfolgt analog.

Eine weitere wichtige Eigenschaft äquivalenter Präkonditionierer liefert das folgende Lemma.

Lemma 2.2 Sind $\mathbf{C_k}$ und $\mathbf{C}'_\mathbf{k}$ zwei äquivalente Präkonditionierer, so berechnet das präkonditionierte Intervall-Gauß-Seidel-Verfahren für beide Präkonditionierer die gleichen Einschließungen (in exakter Arithmetik).

Beweis: Laut Definition der Äquivalenz existiert ein $\alpha \neq 0$ mit $\mathbf{C}'_\mathbf{k} = \alpha \cdot \mathbf{C_k}$. Dann gilt nach Definition offensichtlich:

$$\begin{aligned} [n_k](\mathbf{C}'_\mathbf{k}) \oslash [d_k](\mathbf{C}'_\mathbf{k}) &= [n_k](\alpha\mathbf{C_k}) \oslash [d_k](\alpha\mathbf{C_k}) \\ &= (\alpha \cdot [n_k](\mathbf{C_k})) \oslash (\alpha \cdot [d_k](\mathbf{C_k})) \\ &= [n_k](\mathbf{C_k}) \oslash [d_k](\mathbf{C_k}). \end{aligned}$$

In diesem Abschnitt verfolgen wir das Ziel optimale Präkonditionierer zu entwickeln. Allerdings können Präkonditionierer in vielerlei Hinsicht optimal sein. Es bedarf also einer genauen Definition des Optimalitätsbegriffes bzw. der Charakterisierung verschiedener Optimalitätstypen. Kearfott und Novoa führen die folgenden Optimalitätskriterien für zeilenweise Präkonditionierer ein:

Definition 2.5 Sei $\mathbf{C_k}$ ein C-Präkonditionierer.

1. $\mathbf{C_k}$ heißt C^W-*Präkonditionierer*, wenn er den Durchmesser von $[x'_k]$ unter allen C-Präkonditionierern minimiert.

2. $\mathbf{C_k}$ heißt C^L-*Präkonditionierer*, wenn er den linken Randpunkt $\underline{x}'_k$ unter allen C-Präkonditionierern maximiert.

3. $\mathbf{C_k}$ heißt C^R-*Präkonditionierer*, wenn er den rechten Randpunkt $\overline{x}'_k$ unter allen C-Präkonditionierern minimiert.

4. $\mathbf{C_k}$ heißt C^M-*Präkonditionierer*, wenn er die Magnitude von $[x'_k] - \tilde{x}_k$ unter allen C-Präkonditionierern minimiert.

Definition 2.6 Sei $\mathbf{C_k}$ ein S-Präkonditionierer. Dann heißt $\mathbf{C_k}$ ein $\mathrm{S^M}$-Präkonditionierer, wenn er die Mignitude von $[x'_k] - \tilde{x}_k$ unter allen S-Präkonditionierern maximiert.

Doch wie berechnet man nun die verschiedenen zeilenweisen Präkonditionierer? Jeder der in den Definitionen 2.5 und 2.6 vorgestellten Präkonditionierer kann als *reelles* nichtlineares Optimierungsproblem beschrieben werden. Novoa zeigt in seiner Dissertation [55], dass man diese durch geschickte Umformulierung sogar in *lineare Optimierungsprobleme* der Form

$$\begin{aligned} \min \quad & \mathcal{C}^T\mathcal{X} \\ & \mathcal{A}\mathcal{X} = \mathcal{B} \\ & \mathcal{X} \geq \mathbf{0}, \end{aligned} \tag{2.12}$$

mit $\mathcal{C}^T, \mathcal{X} \in \mathbb{R}^{\tilde{n}}, \mathcal{B} \in \mathbb{R}^{\tilde{m}}$ und $\mathcal{A} \in \mathbb{R}^{\tilde{m}\times\tilde{n}}$, überführen kann, die in der Regel wesentlich leichter zu lösen sind als die ursprünglichen Optimierungsprobleme. Detaillierte Informationen zur Behandlung linearer Optimierungsprobleme findet der interessierte Leser zum Beispiel in [7]. Leider findet man in der Literatur für die optimalen Präkonditionierer keine Darstellung in der Form (2.12), die sich für eine sinnvolle Implementierung nutzen ließe: Kearfott beschränkt sich in [38] nur auf eine sehr stark vereinfachte Form des $\mathrm{C^W}$-Präkonditionierers und gibt eine *falsche* Darstellung des $\mathrm{S^M}$-Präkonditionierers an (es fehlen Variablen und eine Nebenbedingung). Aus diesem Grund werden wir im Folgenden die Implementierung der in SONIC integrierten optimalen Präkonditionierer genauer beschreiben.

Korollar 2.3 ([38, Lemma 3.4]) Sei $t \in \mathbb{R}$ und $[x] \in \mathbb{IR}$. Dann gilt

$$t \cdot [x] = t \cdot [\check{x}] + \frac{1}{2} \cdot |t| \cdot \operatorname{diam}([x]) \cdot [-1, 1].$$

Wählt man $\tilde{x}_j = [\check{x_j}]$ für $1 \leq j \leq n$ im Intervall-Gauß-Seidel-Verfahren, so vereinfachen sich der Zähler und der Nenner aus Gleichung (2.11) nach Korollar 2.3 zu

$$\begin{aligned} [n_k](\mathbf{C_k}) = -\sum_{i=1}^{m} c_{k,i}[\check{b_i}] + \frac{1}{2}\sum_{i=1}^{m} |c_{k,i}| \cdot \operatorname{diam}([b_i]) \cdot [-1,1] \\ + \frac{1}{2}\sum_{\substack{j=1 \\ j\neq k}}^{n} \operatorname{diam}([x_j]) \cdot \left|\sum_{i=1}^{m} c_{k,i}[a_{i,j}]\right| \cdot [-1,1] \end{aligned} \tag{2.13}$$

und

$$[d_k](\mathbf{C_k}) = \sum_{j=1}^{m} [\check{a_{j,k}}] + \frac{1}{2}\sum_{j=1}^{m} |c_{k,j}| \cdot \operatorname{diam}([a_{j,k}]) \cdot [-1,1]. \tag{2.14}$$

Ist $x \in \mathbb{R}$ eine beliebige reelle Zahl, so bezeichte im Folgenden $x^+ := \max\{x, 0\}$ den *positiven Anteil* von x und $x^- := \max\{-x, 0\}$ den *negativen Anteil* von x. Damit ist x stets über $x = x^+ - x^-$ darstellbar. Mit dieser Definition erhält man unmittelbar das folgende Korollar.

Korollar 2.4 ([38, Lemma 3.4]) Seien $x, y \in \mathbb{R}$ zwei reelle Zahlen. Dann gilt

1. $\max\{x, y\} = x + (y - x)^+$,

2. $\max\{x, y\} = y + (y - x)^-$.

Wie bereits oben erwähnt, lässt sich insbesondere $c_{k,i}$ über $c_{k,i} = c_{k,i}^+ - c_{k,i}^-$ darstellen. Damit erhalten wir für den Betrag aus Gleichung (2.13) für $1 \leq j \leq n$

$$\begin{aligned}
\left|\sum_{i=1}^{m} c_{k,i}[a_{i,j}]\right| &= \max\left\{-\inf\left(\sum_{i=1}^{m} c_{k,i}[a_{i,j}]\right), \sup\left(\sum_{i=1}^{m} c_{k,i}[a_{i,j}]\right)\right\} \\
&= \max\left\{-\inf\left(\sum_{i=1}^{m} (c_{k,i}^+ - c_{k,i}^-)[a_{i,j}]\right), \sup\left(\sum_{i=1}^{m} (c_{k,i}^+ - c_{k,i}^-)[a_{i,j}]\right)\right\} \\
&= \max\left\{\underbrace{-\sum_{i=1}^{m} (c_{k,i}^+ \underline{a}_{i,j} - c_{k,i}^- \overline{a}_{i,j})}_{=:\lambda_j}, \underbrace{\sum_{i=1}^{m} (c_{k,i}^+ \overline{a}_{i,j} - c_{k,i}^- \underline{a}_{i,j})}_{=:\gamma_j}\right\}, \qquad (2.15)
\end{aligned}$$

wobei wir zunächst auch $j = k$ zulassen. Weiterhin führen wir für $1 \leq j \leq n$ die Variablen

$$\begin{aligned}
v_j &:= \lambda_j + \gamma_j \\
&= \sum_{i=1}^{m} (c_{k,i}^+ \underline{a}_{i,j} - c_{k,i}^- \overline{a}_{i,j}) + \sum_{i=1}^{m} (c_{k,i}^+ \overline{a}_{i,j} - c_{k,i}^- \underline{a}_{i,j}) \\
&= \sum_{i=1}^{m} (c_{k,i}^+ \underline{a}_{i,j} - c_{k,i}^- \overline{a}_{i,j} + c_{k,i}^+ \overline{a}_{i,j} - c_{k,i}^- \underline{a}_{i,j}) \\
&= \sum_{i=1}^{m} (c_{k,i}^+ - c_{k,i}^-)(\underline{a}_{i,j} + \overline{a}_{i,j}) \qquad (2.16)
\end{aligned}$$

ein.

Mit dem ersten Teil von Korollar 2.4 erhält man damit für Gleichung (2.15) und $1 \leq j \leq n$

$$\begin{aligned}
\left|\sum_{i=1}^{m} c_{k,i}[a_{i,j}]\right| &= \max\{-\lambda_j, \gamma_j\} \\
&= -\lambda_j + (\gamma_j + \lambda_j)^+ = -\lambda_j + v_j^+ \\
&= -\sum_{i=1}^{m} (c_{k,i}^+ \underline{a}_{i,j} - c_{k,i}^- \overline{a}_{i,j}) + v_j^+ . \qquad (2.17)
\end{aligned}$$

Analog erhält mit dem zweiten Teil des Korollars 2.4

$$\begin{aligned}\left|\sum_{i=1}^{m} c_{k,i}[a_{i,j}]\right| &= \max\{-\lambda_j, \gamma_j\} \\ &= \gamma_j + (\gamma_j + \lambda_j)^- = \gamma_j + v_j^- \\ &= \sum_{i=1}^{m}(c_{k,i}^+ \overline{a}_{i,j} - c_{k,i}^- \underline{a}_{i,j}) + v_j^-. \end{aligned} \tag{2.18}$$

Wählt man nun ein beliebiges $\delta_j \in [0,1]$, so kann der obige Betrag auch als Konvexkombination der Gleichungen (2.17) und (2.18) geschrieben werden. Es folgt damit

$$\begin{aligned}\left|\sum_{i=1}^{m} c_{k,i}[a_{i,j}]\right| &= \delta_j \left(\sum_{i=1}^{m}(c_{k,i}^- \overline{a}_{i,j} - c_{k,i}^+ \underline{a}_{i,j}) + v_j^+\right) \\ &\quad +(1-\delta_j)\left(\sum_{i=1}^{m}(c_{k,i}^+ \overline{a}_{i,j} - c_{k,i}^- \underline{a}_{i,j}) + v_j^-\right). \end{aligned} \tag{2.19}$$

Diese Variante wurde von Kearfott eingeführt, um das resultierende Optimierungsproblem hinsichtlich der numerischen Stabilität zu verbessern. Da die Wahl eines unabhängigen δ_j mit $1 \le j \le n$ für jede Gleichung nicht sinnvoll erscheint, wählen wir im Folgenden ein spezielles δ mit $\delta = \delta_j$ für $1 \le j \le n$. Für die Standardeinstellungen von SONIC hat sich der Wert $\delta = \frac{1}{2}$ als besonders geeignet herausgestellt.

Mit der Definition von v_j und der Gleichung (2.16) folgt für $1 \le j \le n$

$$v_j^+ - v_j^- = \sum_{i=1}^{m}(c_{k,i}^+ - c_{k,i}^-)(\underline{a}_{i,j} + \overline{a}_{i,j})$$

und damit die Bedingung

$$\underbrace{v_j^+ - v_j^- - 2 \cdot \sum_{i=1}^{m} [\check{a_{i,j}}](c_{k,i}^+ - c_{k,i}^-)}_{=:\Gamma_{k,j}} = 0. \tag{2.20}$$

Anschließend erhalten wir mit Hilfe der Gleichungen (2.13) und (2.19) für das Infimum von $[n_k](\mathbf{C_k})$ die folgende Formel

$$\begin{aligned}\underline{n}_k(\mathbf{C_k}) &= \sum_{j=1}^{m}(c_{k,j}^+ - c_{k,j}^-)[\check{b_j}] - \frac{1}{2}\sum_{j=1}^{m}(c_{k,j}^+ + c_{k,j}^-) \cdot \operatorname{diam}([b_j]) \\ &\quad -\frac{\delta}{2}\sum_{\substack{j=1 \\ j\neq k}}^{n} \operatorname{diam}([x_j])\left(\sum_{i=1}^{m}(c_{k,i}^- \overline{a}_{i,j} - c_{k,i}^+ \underline{a}_{i,j}) + v_j^+\right) \\ &\quad -\frac{1-\delta}{2}\sum_{\substack{j=1 \\ j\neq k}}^{n} \operatorname{diam}([x_j])\left(\sum_{i=1}^{m}(c_{k,i}^+ \overline{a}_{i,j} - c_{k,i}^- \underline{a}_{i,j}) + v_j^-\right). \end{aligned} \tag{2.21}$$

Dabei ist zu beachten, dass die Variablen v_k^+ und v_k^- (wie auch in den nächsten beiden Formeln) zunächst nicht benötigt werden. Analog erhält man die Formel für das Supremum von $[n_k](\mathbf{C_k})$,

$$\begin{aligned}\overline{n}_k(\mathbf{C_k}) &= \sum_{j=1}^{m}(c_{k,j}^+ - c_{k,j}^-)[\check{b}_j] + \frac{1}{2}\sum_{j=1}^{m}(c_{k,j}^+ + c_{k,j}^-)\cdot\operatorname{diam}([b_j]) \\ &\quad + \frac{\delta}{2}\sum_{\substack{j=1\\ j\neq k}}^{n}\operatorname{diam}([x_j])\left(\sum_{i=1}^{m}(c_{k,i}^-\overline{a}_{i,j} - c_{k,i}^+\underline{a}_{i,j}) + v_j^+\right) \\ &\quad + \frac{1-\delta}{2}\sum_{\substack{j=1\\ j\neq k}}^{n}\operatorname{diam}([x_j])\left(\sum_{i=1}^{m}(c_{k,i}^+\overline{a}_{i,j} - c_{k,i}^-\underline{a}_{i,j}) + v_j^-\right)\end{aligned} \tag{2.22}$$

Daraus lässt sich unmittelbar die folgende Formel für den Durchmesser von $[n_k](\mathbf{C_k})$ ableiten

$$\begin{aligned}\operatorname{diam}([n_k](\mathbf{C_k})) &= \sum_{j=1}^{m}(c_{k,j}^+ + c_{k,j}^-)\cdot\operatorname{diam}([b_j]) \\ &\quad + \delta\sum_{\substack{j=1\\ j\neq k}}^{n}\operatorname{diam}([x_j])\left(\sum_{i=1}^{m}(c_{k,i}^-\overline{a}_{i,j} - c_{k,i}^+\underline{a}_{i,j}) + v_j^+\right) \\ &\quad + (1-\delta)\sum_{\substack{j=1\\ j\neq k}}^{n}\operatorname{diam}([x_j])\left(\sum_{i=1}^{m}(c_{k,i}^+\overline{a}_{i,j} - c_{k,i}^-\underline{a}_{i,j}) + v_j^-\right).\end{aligned} \tag{2.23}$$

Auf diese Weise kann aus Gleichung (2.14) auch eine Formel für den Nenner $[d_k](\mathbf{C_k})$ in der Gauß-Seidel-Iterationsformel abgeleitet werden. Es gilt

$$[d_k](\mathbf{C_k}) = \sum_{j=1}^{m} c_{k,j}[a_{j,k}] \tag{2.24}$$

$$= \left[\sum_{j=1}^{m}(c_{k,j}^+\underline{a}_{j,k} - c_{k,j}^-\overline{a}_{j,k}), \sum_{j=1}^{m}(c_{k,j}^+\overline{a}_{j,k} - c_{k,j}^-\underline{a}_{j,k})\right]. \tag{2.25}$$

2.2.1.2.1 $\mathbf{C^W}$-Präkonditionierer Der Durchmesser-optimierende $\mathrm{C^W}$-Präkonditionierer ist, laut Definition, Lösung des Optimierungsproblems

$$\min_{\mathbf{C_k}} \operatorname{diam}\left(\frac{[n_k](\mathbf{C_k})}{[d_k](\mathbf{C_k})}\right), \tag{2.26}$$

wobei über die Menge aller C-Präkonditionierer optimiert wird. Da nach Lemma 2.2 zu jedem C-Präkonditionierer ein äquivalenter normaler C-Präkonditionierer existiert und diese nach Korollar 2.2 die gleichen Einschließungen berechnen, kann das Optimierungsproblem auf normale C-Präkonditionierer beschränkt werden. Um das Optimierungsproblem weiter zu vereinfachen, betrachten wir zunächst das folgende Lemma.

Lemma 2.3 Seien $[a], [b] \in \mathbb{IR}_*$ zwei Intervalle. Gilt zusätzlich $0 \in [a]$ und $\underline{b} = 1$, so folgt

$$\frac{[a]}{[b]} = [a].$$

Geht man zunächst davon aus, dass für den C^W-Präkonditionierer $0 \in [n_k(\mathbf{C_k})]$ gilt, dann läßt sich dieser nach Lemma 2.3 über das Optimierungsproblem

$$\min_{\underline{d}_k(\mathbf{C_k})=1} \operatorname{diam}([n_k](\mathbf{C_k})) \tag{2.27}$$

berechnen. Offensichtlich kann man erst nach der Berechnung des Optimierungsproblems (2.27) nachprüfen, ob die Bedingung $0 \in [n_k(\mathbf{C_k})]$ tatsächlich erfüllt ist. Ist dies der Fall, so gilt nach [55], dass die Lösung des Optimierungsproblems tatsächlich den gewünschten C^W-Präkonditionierer liefert (anderenfalls erhält man nur einen völlig beliebigen Präkonditionierer). Man kann allerdings zeigen, dass für einen beliebigen C-Präkonditionierer $\mathbf{C_k}$ mit $0 \notin [n_k(\mathbf{C_k})]$ stets

$$\operatorname{diam}([x'_k] \cap [x_k]) \leq \frac{1}{2} \cdot \operatorname{diam}([x_k])$$

gilt (siehe [36, Lemma 2.2]). Die Berechnung eines Präkonditionierers über das Optimierungsproblem (2.27) macht also stets Sinn, da mit diesem auf jeden Fall eine *gute* Kontraktion bezüglich der Variablen x_k erreicht wird.

Das Normalisierungs-Constraint $\underline{d}_k(\mathbf{C_k}) = 1$ aus (2.27) kann mit Hilfe von Gleichung (2.25) sehr leicht als Nebenbedingung

$$\sum_{j=1}^{m} (c^+_{k,j} \underline{a}_{j,k} - c^-_{k,j} \overline{a}_{j,k}) = 1 \tag{2.28}$$

formuliert werden. Diese Nebenbedingung ist offensichtlich linear in den Variablen $c^+_{k,1}, \ldots, c^+_{k,m}, c^-_{k,1}, \ldots, c^-_{k,m}$. Die Zielfunktion des Optimierungsproblems (2.27) haben wir bereits in Gleichung (2.23) bestimmt. Diese ist ebenfalls linear in ihren Variablen. Weiterhin müssen die linearen Nebenbedingungen (2.20) gelten. Da die Zielfunktion die Variablen v^+_k und v^-_k nicht enthält, kann die Bedingung $\Gamma_{k,k} = 0$ entfallen.

Definition 2.7 Jede Lösung des *linearen* Optimierungsproblems in den $2(m + n - 1)$ Variablen

$$(c^+_{k,1}, \ldots, c^+_{k,m}, c^-_{k,1}, \ldots, c^-_{k,m}, v^+_1, \ldots, v^+_{k-1}, v^+_{k+1}, \ldots, v^+_n, v^-_1, \ldots, v^-_{k-1}, v^-_{k+1}, \ldots, v^-_n),$$

die einer Nicht-Negativitäts-Nebenbedingung unterliegen, mit der Zielfunktion (2.23) sowie den $n - 1$ Nebenbedingungen $\Gamma_{k,j} = 0$ für $1 \leq j \leq n, j \neq k$ aus (2.20) und der Normalisierungsbedingung (2.28) heißt *C^W-LP-Präkonditionierer* für die k-te Variable.

Es sei an dieser Stelle noch einmal darauf hingewiesen, dass im Allgemeinen ein Unterschied zwischen dem C^W- und dem C^W-LP-Präkonditionierer existiert. In der Praxis wird also zunächst der C^W-LP-Präkonditionierer berechnet. Ist man an der

Information interessiert, ob dieser Präkonditionierer tatsächlich den Durchmesser von $[x'_k]$ minimiert, so muss nach [55] noch die Bedingung $0 \in [n_k](\mathbf{C_k})$ überprüft werden. Dieser Schritt wird in der Praxis aber meist nicht durchgeführt.

Der C^W-LP-Präkonditionierer aus Definition 2.7 muss nun nur noch auf die allgemeine Form (2.12) für lineare Optimierungsprobleme gebracht werden. Mit einigen wenigen Umformungen erhält man die folgende Darstellung

$$
\begin{aligned}
\mathcal{A} &= \begin{pmatrix} ([\check{\mathbf{A}}]_{\neg k})^T & (-[\check{\mathbf{A}}]_{\neg k})^T & -\mathbf{I}_{(n-1)\times(n-1)} & \mathbf{I}_{(n-1)\times(n-1)} \\ \inf([\mathbf{A}]_{1:m,k})^T & -\sup([\mathbf{A}]_{1:m,k})^T & \mathbf{0}_{1\times(n-1)} & \mathbf{0}_{1\times(n-1)} \end{pmatrix}, \\
\mathcal{B} &= \begin{pmatrix} \mathbf{0}_{(n-1)\times 1} \\ 1 \end{pmatrix}, \\
\mathcal{C} &= \begin{pmatrix} \operatorname{diam}([b]) + \left(\operatorname{diam}([\mathbf{x}]^T_{\neg k}) \cdot \left(-\delta \cdot \inf(\mathbf{A}_{\neg k})^T + (1-\delta) \cdot \sup(\mathbf{A}_{\neg k})^T\right)\right)^T \\ \operatorname{diam}([b]) + \left(\operatorname{diam}([\mathbf{x}]^T_{\neg k}) \cdot \left(\delta \cdot \sup(\mathbf{A}_{\neg k})^T - (1-\delta) \cdot \inf(\mathbf{A}_{\neg k})^T\right)\right)^T \\ 2 \cdot \delta \cdot \operatorname{diam}([\mathbf{x}]_{\neg k}) \\ 2 \cdot (1-\delta) \cdot \operatorname{diam}([\mathbf{x}]_{\neg k}) \end{pmatrix}, \\
\mathcal{X} &= \left(\mathcal{C}^+, \ \mathcal{C}^-, \ \mathcal{V}^+, \ \mathcal{V}^- \right)^T,
\end{aligned}
$$

wobei

$$
\begin{aligned}
\mathcal{C}^+ &= (c^+_{k,1}, \ldots, c^+_{k,m}) \\
\mathcal{C}^- &= (c^-_{k,1}, \ldots, c^-_{k,m}) \\
\mathcal{V}^+ &= \tfrac{1}{2} \cdot (v^+_1, \ldots, v^+_{k-1}, v^+_{k+1}, \ldots, v^+_n) \\
\mathcal{V}^- &= \tfrac{1}{2} \cdot (v^-_1, \ldots, v^-_{k-1}, v^-_{k+1}, \ldots, v^-_n)
\end{aligned}
$$

gilt. Hat man eine Lösung des linearen Optimierungsproblems gefunden, so wird der C^W-LP-Präkonditionierer über die Anweisung

$$c_{k,i} = c^+_{k,i} - c^-_{k,i}$$

für $1 \leq i \leq m$ gebildet.

Es ist offensichtlich, dass das lineare Optimierungsproblem zur Berechnung eines C^W-LP-Präkonditionierers eine spezielle Struktur aufweist. In [34] wird gezeigt, dass die Berechnung des Durchmesser-optimierenden Präkonditioniers um den Faktor fünf beschleunigt werden kann, wenn die Struktur des Optimierungsproblems ausgenutzt wird. Desweiteren kann bei Verwendung des Simplex-Verfahrens der benötigte Speicherplatz für das Simplex-Tableau deutlich verringert werden, da sich einige Spalten aus anderen Spalten leicht berechnen lassen. Für dünnbesetzte Probleme kann der C^W-LP-Präkonditionierer genauso schnell berechnet werden wie der Inverse-Midpoint-Präkonditionierer.

Für die spezielle Wahl $\delta = 1$ ist für den C^W-LP-Präkonditionierer in GlobSol eine Fortran95-Implementierung des Simplex-Verfahrens integriert, die der speziellen Struktur des linearen Optimierungsproblems Rechnung trägt. Dieser Code wurde zu Testzwecken nach C++ portiert, um ihn in SONIC einsetzen zu können. Gleichzeitig haben wir jeweils eine Implemtierung des C^W-LP-Präkonditionierers mit dem Paket Glpk und

Galahad (eine Beschreibung der beiden Pakete zur linearen Optimierung findet der interessierte Leser in Kapitel 5.2) entwickelt. Bei umfangreichen Tests konnte festgestellt werden, dass die beiden „professionellen“ Pakete wesentlich schnellere Berechnungen durchführen als die GlobSol-Variante (mit dem Paket Glpk konnte der Präkonditionierer teilweise sechs mal schneller als mit dem GlobSol-Code berechnet werden), die die spezielle Struktur berücksichtigt. Wie wir später noch genauer sehen werden, dominiert die Berechnung der optimalen Präkonditionierer sehr deutlich das Laufzeitverhalten von SONIC. Eine schnellere Berechnung der Präkonditionierer würde folglich eine wesentlich bessere Performance des Tools zur Folge haben. Aus diesem Grund erachten wir es als besonders wichtig, auf diesem Gebiet noch weitere umfangreiche Untersuchungen zur Verbesserung der Laufzeit durchzuführen. Eine eigene C++-Implementierung, die die effizienten Techniken des Glpk ausnutzt und speziell auf die Struktur des Optimierungsproblems zugeschnitten ist, ist geplant.

Mögliche Verbesserungen könnten sich bei der Berechnung optimaler Präkonditionierer durch folgende Punkte ergeben:

- In einem ersten Versuch wurde die Anzahl der Iterationen, die im Simplex-Verfahren maximal durchgeführt werden dürfen, durch eine Konstante γ (zum Beispiel $\gamma = 10$) nach oben beschränkt. Dies hat zur Folge, dass weniger Zeit, auf Kosten der Genauigkeit des Verfahrens, benötigt wird (man beachte, dass die Präkonditionierungszeile ein reeller Vektor ist). Dieser Ansatz hat sich bei einigen Testbeispielen als vielversprechend herausgestellt. Da jedoch keine gesicherte Aussage darüber getroffen werden kann, ob dieses Vorgehen *immer* sinnvoll ist, haben wir in den Standardeinstellungen von SONIC zunächst auf diese Variante verzichtet.

- Eine Verbesserung des Laufzeitverhaltens könnte sich auch ergeben, falls andere Verfahren zur Optimierung als das Simplex-Verfahren zur Anwendung kommen würden (z.B. die Innere-Punkte-Methode auf das *ursprüngliche* Optimierungsproblem).

- Lin und Stadtherr [46] haben den so genannten Pivot-Präkonditionierer entwickelt. Bei der Berechnung dieses Präkonditionierers geht man davon aus, dass genau ein Eintrag in der Präkonditionierungszeile $\mathbf{C_k}$ nicht verschwindet und den Wert 1 annimmt. Unter dieser Annahme kann man sehr leicht einen Durchmesser-optimierenden Präkonditionierer berechnen. Dieser Pivot-Präkonditionierer wurde testweise in SONIC integriert. Die Ergebnisse waren ernüchternd und vielversprechend zugleich. Zunächst konnte festgestellt werden, dass die wesentlich schnellere Berechnung des Pivot-Präkonditionierers in der Regel nicht ausreicht, um gegen die Qualität des langsameren C^W-LP-Präkonditionierers anzukommen. Andererseits konnte auch festgestellt werden, dass die Annahme, dass nur ein Eintrag (in der Regel verschieden von 1) in der Präkonditionierungszeile besetzt ist, sich in der Praxis in einer größeren Anzahl von Fällen tatsächlich als richtig erweist. Wir wollen dies an ein paar Testsystemen demonstrieren, wobei wir in Klammern den Prozentsatz der Präkonditionierungszeilen angeben, bei dem nur ein einziger Eintrag besetzt ist: 7erSystem (23.78%), Agrawal_Hopf_1 (31.18%), DirectKinematics (30.58%), CSTR_Cusp_Large (62.32%), min-03-07 (83.26%),

Eco9 (2.71%). Da wir nach Lemma 2.2 wissen, dass äquivalente Präkonditionierer aber die gleichen Einschließungen berechnen, ist die Berechnung über ein lineares Programm offensichtlich zu teuer. Eine schnellere Berechnung des C^W-LP-Präkonditionierers könnte damit erfolgen, wenn frühzeitig abgeschätzt werden kann, wann nur ein Eintrag in der Präkonditionierungszeile besetzt ist.

An dieser Stelle sei noch einmal darauf hingewiesen, dass der C^W-LP-Präkonditionierer nur den Durchmesser des resultierenden Intervalles $[x'_k]$ optimiert. Dies ist offensichtlich nur dann optimal, wenn $[x'_k] \subset [x_k]$ gilt, da man in der Praxis eigentlich den Durchmesser des Schnittes $[x'_k] \cap [x_k]$ minimieren will. Aus diesem Grund hat der Autor einen Präkonditionierer entwickelt, der den Durchmesser des Schnittes optimiert. Leider ist es bis heute nur gelungen, das resultierende Optimierungsproblem auf ein quadratisches—und nicht lineares—Optimierungsproblem zu reduzieren. Da das Paket Galahad auch quadratische Optimierungsprobleme löst, wurde dieser Präkonditionierer testweise in SONIC integriert. In dieser Form konnte nur eine minimale Verbesserung der Boxzahlen beobachtet werden, die mit einer massiven Erhöhung der Laufzeit einhergehen. Wir haben daher auf eine endgültige Integration dieses Präkonditionierers verzichtet. Der interessierte Leser findet die Formulierung dieses Präkonditionierers im Anhang A dieser Arbeit.

2.2.1.2.2 S^M-Präkonditionierer

Nimmt man zunächst an, dass $0 \in \operatorname{int}([d_k](\mathbf{C_k}))$ und $[n_k](\mathbf{C_k}) > 0$ gilt, dann gilt nach der Definition der relationalen Division (1.6)

$$[n_k](\mathbf{C_k}) \oslash [d_k](\mathbf{C_k}) = \left[-\infty, \frac{\underline{n}_k(\mathbf{C_k})}{\underline{d}_k(\mathbf{C_k})}\right] \cup \left[\frac{\underline{n}_k(\mathbf{C_k})}{\overline{d}_k(\mathbf{C_k})}, \infty\right].$$

Da wir uns nach Korollar 2.2 und Lemma 2.2 auf normale S-Präkonditionierer beschränken können, ist dann der S^M-Präkonditionierer Lösung des nichtlinearen Optimierungsproblems

$$\max_{\underline{n}_k(\mathbf{C_k})=1} \min\left\{-\frac{1}{\underline{d}_k(\mathbf{C_k})}, \frac{1}{\overline{d}_k(\mathbf{C_k})}\right\}. \tag{2.29}$$

Wegen der Voraussetzung $0 \in \operatorname{int}([d_k](\mathbf{C_k}))$ folgt

$$\begin{aligned} \min\left\{-\frac{1}{\underline{d}_k(\mathbf{C_k})}, \frac{1}{\overline{d}_k(\mathbf{C_k})}\right\} &= \frac{1}{\max\{|\underline{d}_k(\mathbf{C_k})|, |\overline{d}_k(\mathbf{C_k})|\}} \\ &= \frac{1}{|[d_k](\mathbf{C_k})|}. \end{aligned}$$

Da $1/|[d_k](\mathbf{C_k})|$ maximal wird, wenn die Magnitude $|[d_k](\mathbf{C_k})|$ minimal wird, lässt sich das Optimierungsproblem (2.29) durch

$$\min_{\underline{n}_k(\mathbf{C_k})=1} |[d_k](\mathbf{C_k})| \tag{2.30}$$

ersetzen. Für die Zielfunktion gilt nach (2.19)

$$\begin{aligned} |[d_k](\mathbf{C_k})| &= \left| \sum_{j=1}^{m} c_{k,j}[a_{j,k}] \right| \\ &= \delta \left(\sum_{j=1}^{m} (c_{k,j}^{-} \overline{a}_{j,k} - c_{k,j}^{+} \underline{a}_{j,k}) + v_k^{+} \right) \\ &\quad +(1-\delta) \left(\sum_{j=1}^{m} (c_{k,j}^{+} \overline{a}_{j,k} - c_{k,j}^{-} \underline{a}_{j,k}) + v_k^{-} \right). \end{aligned} \tag{2.31}$$

Für die Normalisierungsbedingung $\underline{n}_k(\mathbf{C_k}) = 1$ gilt nach (2.21)

$$\begin{aligned} &\sum_{j=1}^{m} (c_{k,j}^{+} - c_{k,j}^{-})[\check{b}_j] - \frac{1}{2} \sum_{j=1}^{m} (c_{k,j}^{+} + c_{k,j}^{-}) \cdot \operatorname{diam}([b_j]) \\ &\quad -\frac{\delta}{2} \sum_{\substack{j=1 \\ j \neq k}}^{n} \operatorname{diam}([x_j]) \left(\sum_{i=1}^{m} (c_{k,i}^{-} \overline{a}_{i,j} - c_{k,i}^{+} \underline{a}_{i,j}) + v_j^{+} \right) \\ -\frac{1-\delta}{2} &\sum_{\substack{j=1 \\ j \neq k}}^{n} \operatorname{diam}([x_j]) \left(\sum_{i=1}^{m} (c_{k,i}^{+} \overline{a}_{i,j} - c_{k,i}^{-} \underline{a}_{i,j}) + v_j^{-} \right) = 1 \,. \end{aligned} \tag{2.32}$$

Definition 2.8 Jede Lösung des *linearen* Optimierungsproblems in den $2(m+n)$ Variablen

$$(c_{k,1}^{+}, \ldots, c_{k,m}^{+}, c_{k,1}^{-}, \ldots, c_{k,m}^{-}, v_1^{+}, \ldots, v_n^{+}, v_1^{-}, \ldots, v_n^{-}),$$

die einer Nicht-Negativitäts-Nebenbedingung unterliegen, mit der Zielfunktion (2.31) sowie den n Nebenbedingungen $\Gamma_{k,j} = 0$ für $1 \leq j \leq n$ aus (2.20) und der Normalisierungsbedingung (2.32) heißt *S^M-LP-Präkonditionierer* für die k-te Variable.

Der $\mathrm{S^M}$-LP-Präkonditionierer aus Definition 2.8 muss nun nur noch auf die allgemeine Form (2.12) für lineare Optimierungsprobleme gebracht werden. Mit Hilfe der Abkürzungen

$$\begin{aligned} \mathcal{A}_{1,1} &= [\check{\mathbf{b}}]^T + \tfrac{1}{2} \left(-\operatorname{diam}([\mathbf{b}])^T + \operatorname{diam}([\mathbf{x}]_{\neg k}^T)(\delta \inf([\mathbf{A}]_{\neg k})^T - (1-\delta) \sup([\mathbf{A}]_{\neg k})^T) \right), \\ \mathcal{A}_{1,2} &= -[\check{\mathbf{b}}]^T - \tfrac{1}{2} \left(\operatorname{diam}([\mathbf{b}])^T - \operatorname{diam}([\mathbf{x}]_{\neg k}^T)((1-\delta) \inf([\mathbf{A}]_{\neg k})^T) - \delta \sup([\mathbf{A}]_{\neg k})^T \right), \\ \mathcal{A}_{1,3} &= -\delta \cdot (\operatorname{diam}([x_1]), \ldots, \operatorname{diam}([x_{k-1}]), 0, \operatorname{diam}([x_{k+1}]), \ldots, \operatorname{diam}([x_n])), \\ \mathcal{A}_{1,4} &= -(1-\delta) \cdot (\operatorname{diam}([x_1]), \ldots, \operatorname{diam}([x_{k-1}]), 0, \operatorname{diam}([x_{k+1}]), \ldots, \operatorname{diam}([x_n])), \end{aligned}$$

und einigen wenigen Umformungen erhält man die folgende Darstellung

$$\begin{aligned}
\mathcal{A} &= \begin{pmatrix} \mathcal{A}_{1,1} & \mathcal{A}_{1,2} & \mathcal{A}_{1,3} & \mathcal{A}_{1,4} \\ [\check{\mathbf{A}}]^T & -[\check{\mathbf{A}}]^T & -\mathbf{I}_{n\times n} & \mathbf{I}_{n\times n} \end{pmatrix}, \\
\mathcal{B} &= \begin{pmatrix} 1 \\ \mathbf{0}_{n\times 1} \end{pmatrix}, \\
\mathcal{C} &= \begin{pmatrix} \left(-\delta \inf([\mathbf{A}]_{1:m,k})^T + (1-\delta)\sup([\mathbf{A}]_{1:m,k})^T\right) \\ \left(\delta \sup([\mathbf{A}]_{1:m,k})^T - (1-\delta)\inf([\mathbf{A}]_{1:m,k})^T\right) \\ 2\cdot\delta\cdot\mathbf{e_k} \\ 2\cdot(1-\delta)\cdot\mathbf{e_k} \end{pmatrix}, \\
\mathcal{X} &= \left(c^+_{k,1},\ldots,c^+_{k,m},c^-_{k,1},\ldots,c^-_{k,m},\tfrac{1}{2}v^+_1,\ldots,\tfrac{1}{2}v^+_n,\tfrac{1}{2}v^-_1,\ldots,\tfrac{1}{2}v^-_n\right)^T.
\end{aligned}$$

Dabei bezeichnet $\mathbf{e_k} \in \mathbb{R}^n$ den k-ten Einheitsvektor. Auch hier werden nach Berechnung des Optimierungsproblems die Einträge der Präkonditionierungszeile $\mathbf{C_k}$ über

$$c_{k,i} = c^+_{k,i} - c^-_{k,i}$$

für $1 \leq i \leq m$ bestimmt.

Da wir die verschiedenen optimalen Präkonditionierer ausschließlich im Intervall-Newton-Gauß-Seidel-Verfahren (Kapitel 2.3.2), bzw. in einigen Modifikationen dieses Verfahrens, einsetzen, werden wir erst an entsprechender Stelle auf die genaue Verwendung dieser in SONIC eingehen.

Abschließend möchte ich an dieser Stelle noch auf eine subtile Fehlerquelle hinweisen. Die Präkonditionierungszeile $\mathbf{C_k}$ ist ein reeller Vektor und wird bei den optimalen Präkonditionierern mit Hilfe eines reellen Optimierungstools bestimmt. Die Anwendung des Präkonditionierers auf das lineare Intervallgleichungssystem *muss* allerdings *verifiziert* erfolgen: Der reelle Vektor muss also vor der Präkonditionierung auf einen Intervall-Vektor ge-castet werden. Anderenfalls können Lösungen des Gleichungssystems *verloren gehen.*

2.3 Weitere Kontraktionsverfahren

In diesem Abschnitt stellen wir eine Reihe von *Kontraktionsverfahren* für nichtlineare Gleichungssysteme vor. Diese Verfahren sind in der Lage die Intervalleinschließungen der Lösungsmenge eines nichtlinearen, reellen Gleichungssystems zu verbessern. D.h. eine aktuell betrachtete Box $[\mathbf{x}]$ wird durch eine kleinere Box $[\mathbf{x}']$ ersetzt ohne dabei Nullstellen $\mathbf{x}^* \in [\mathbf{x}]$ von $\mathbf{f}$ zu verlieren.

2.3.1 Taylor-Verfahren

Das erste Kontraktionsverfahren, das wir an dieser Stelle vorstellen möchten, ist das so genannte *Taylor-Verfahren.* Die Idee dieses Verfahrens ist es, jede Gleichung des

Systems nach jeder Gleichung aufzulösen, um so eine verbesserte Einschließung zu erhalten. Um die Funktionsweise der Taylor-Verfahrens zu verstehen, reicht es völlig aus, nur eine Gleichung des Gleichungssystems zu betrachten. Die dafür entwickelten Verfahren können dann ohne Probleme auf alle anderen Gleichungen ebenfalls angewendet werden. Insbesondere ergibt sich daraus, dass das Taylor-Verfahren erster bzw. zweiter Ordnung auch auf nicht-quadratische Systeme anwendbar ist. In diesem Abschnitt gehen wir also davon aus, dass $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ eine reellwertige Funktion ist.

Da die Funktion f möglicherweise nichtlinear ist, ist das Auflösen nach einer bestimmten Variablen unter Umständen nicht ohne weiteres möglich. Aus diesem Grund betrachtet man eine Linearisierung bzw. die Taylorentwicklung zweiter Ordung für f, woraus sich das Taylor-Verfahren erster und zweiter Ordnung ergibt.

2.3.1.1 Taylor-Verfahren erster Ordung

Beim *Taylor-Verfahren erster Ordnung* wird die Funktion f zunächst über eines der in Kapitel 1 vorgestellten Verfahren linearisiert. Man erhält also für einen beliebigen Referenzpunkt $\mathbf{c} \in [\mathbf{x}]$ einen Intervall-Steigungs-Vektor $[\mathbf{s}]$ mit

$$f([\mathbf{x}]) \subseteq f(\mathbf{c}) + \sum_{i=1}^{n} [s_i] \cdot ([x_i] - c_i). \tag{2.33}$$

Da wir nach Nullstellen der Funktion f suchen, kann also alternativ die lineare Intervallgleichung

$$0 = f(\mathbf{c}) + \sum_{i=1}^{n} [s_i] \cdot ([x_i] - c_i) \tag{2.34}$$

betrachtet werden. Es ist offensichtlich, dass jede Lösung des nichtlinearen Gleichungssystems $f(\mathbf{x}) = 0$ auch eine Lösung des linearen Gleichungssystems (2.34) ist.

Wir erhalten damit über den linearen Kontraktor (2.5) die Kontraktionsvorschrift aus Algorithmus 2.4.

Algorithmus 2.4 Taylor-Verfahren erster Ordnung (mit Steigungen)

1: **for** $i = 1, \ldots, n$ **do**

2: $$[x_i'] := [x_i] \cap \left\{ c_i - \left(f(\mathbf{c}) + \sum_{\substack{j=1 \\ j \neq i}}^{n} [s_j] \cdot ([x_j] - c_j) \right) \oslash [s_i] \right\}$$

3: **end for**

Bei der oberflächlichen Betrachtung von Algorithmus 2.4 könnte sich der aufmerksame Leser fragen, warum in Zeile 3 nicht jeweils die bereits verbesserte Komponente $[x_j']$ für $1 \leq j < i$ in die Summenformel eingesetzt wird: In [66] wird gezeigt, dass dieses Vorgehen keinen Vorteil bringt, wenn nicht zwischendurch eine neue Einschließung für den Steigungsvektor $[\mathbf{s}]$ berechnet wird.

Im Spezialfall, dass man als Steigungsvektor eine Einschließung der Funktion f verwendet, spricht man auch vom *Taylor-Verfahren mit Ableitungen*, anderenfalls vom *Taylor-Verfahren mit Steigungen*.

2.3.1.2 Taylor-Verfahren zweiter Ordung

Für das Taylor-Verfahren zweiter Ordnung muss f zweimal stetig differenzierbar auf der Box $[\mathbf{x}]$ sein. Man erhält nach (1.12) für eine Nullstelle $\mathbf{x}^*$ von f

$$0 = f(\mathbf{x}^*) = f(\mathbf{c}) + \nabla\mathbf{f}(\mathbf{c})(\mathbf{x}^* - \mathbf{c}) + \frac{1}{2}(\mathbf{x}^* - \mathbf{c})^T\mathbf{D^2f}(\xi)(\mathbf{x}^* - \mathbf{c}) \tag{2.35}$$

für ein ξ zwischen dem gewählten Referenzpunkt $\mathbf{c} \in [\mathbf{x}]$ und der Nullstelle $\mathbf{x}^*$. Durch Ausnutzung der Symmetrie der Hesse-Matrix $\mathbf{H} := \mathbf{D^2f}$ und Einführung der Bezeichnungen

$$u_k := \frac{\partial f}{\partial x_k}(\mathbf{c}), \;\; h_{k,l} := \frac{\partial^2 f}{\partial x_k \partial x_l}(\xi)$$

erhält man für $1 \le j \le n$

$$\begin{aligned} 0 \;\; = \;\; & \frac{1}{2}h_{j,j}(x_j^* - c_j)^2 + \left(u_j + \sum_{k\neq j} h_{j,k}(x_k^* - c_k)\right)(x_j^* - c_j) \\ & + f(\mathbf{c}) + \sum_{k\neq j} u_k(x_k^* - c_k) + \sum_{\substack{k,l\neq j \\ k<j}} h_{k,l}(x_k^* - c_k)(x_l^* - c_l). \end{aligned}$$

Ersetzt man nun in der obigen Gleichung $y := x_j^* - c_j$, so erhält man eine Gleichung des Typs

$$0 \in [a] \cdot y^2 + [b] \cdot y + [c]$$

mit geeigneten Intervallen $[a], [b]$ und $[c]$, die alle *nicht* von y abhängen. In [20] wird gezeigt, dass im Punktfall die Lösungsformeln für diese quadratische Gleichung

$$[y_1^{(a)}] := \frac{-[b] + \sqrt{[d]}}{2[a]}, \;\; [y_2^{(a)}] := \frac{-[b] - \sqrt{[d]}}{2[a]}$$

mit $[d] := [b]^2 - 4[a][c]$ sehr anfällig für Rundungsfehler ist, falls $b \approx \pm\sqrt{d}$ ist. Damit ist zu erwarten, dass die Intervallauswertung in diesem Fall auch zu signifikanten Überschätzungen führen wird. Aus diesem Grund führen wir zusätzlich (vgl. [5]) die alternativen Lösungsformeln

$$[y_1^{(b)}] := \frac{2[c]}{-[b] - \sqrt{[d]}}, \;\; [y_2^{(b)}] := \frac{2[c]}{-[b] + \sqrt{[d]}}$$

ein, die dann bessere Einschließungen berechnen. Als Konsequenz berechnen wir die Lösungen der quadratischen Gleichung über

$$[y_1^{(a)}] \cap [y_1^{(b)}] \cap ([x_j] - c_j), \;\; [y_2^{(a)}] \cap [y_2^{(b)}] \cap ([x_j] - c_j).$$

Es ist zu beachten, dass durch das Auflösen der quadratischen Gleichung das Intervall $[x_j]$ möglicherweise in bis zu vier Teilintervalle unterteilt wird. Auf eine exakte Formulierung des Taylor-Verfahrens zweiter Ordnung wollen an dieser Stelle verzichten, da dieses völlig analog zum Verfahren erster Ordnung ist.

2.3.2 Das Newton-Gauß-Seidel-Verfahren

Zur Bestimmung von Nullstellen eines nichtlinearen Gleichungssystems stellt die Analysis das so genannte *Newton-Gauß-Seidel-Verfahren* zur Verfügung. Dieses kann leicht zu einem Intervall-Verfahren mit sehr attraktiven Eigenschaften erweitert werden. Die präkonditionierte Variante ist wohl eines der stärksten Kontraktionsverfahren und bildet ein wesentliches Kernstück des nichtlinearen Lösers.

Analog zum Taylor-Verfahren nutzt das Newton-Gauß-Seidel-Verfahren eine Linearisierung mit Ableitungen oder Steigungen der Funktion zur Kontraktion einer Box. Sei $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ eine Funktion, $[\mathbf{x}]$ eine aktuelle Box und $[\mathbf{S}]$ eine Intervall-Steigungs-Matrix für $\mathbf{f}$ und $[\mathbf{x}]$ bezüglich eines Referenzpunktes $\mathbf{c} \in [\mathbf{x}]$ (diese ergibt sich durch komponentenweise Zusammensetzung von Intervall-Steigungs-Vektoren). Dann gilt

$$\mathbf{f}([\mathbf{x}]) \subseteq [\mathbf{f}](\mathbf{c}) + [\mathbf{S}]([\mathbf{x}] - \mathbf{c}).$$

Für jede Lösung $\mathbf{x} \in [\mathbf{x}]$ des Gleichungssystems $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ muss offensichtlich

$$0 \in \mathbf{f}(\mathbf{c}) + [\mathbf{S}](\mathbf{x} - \mathbf{c})$$

gelten. Damit ist $\mathbf{x}$ aber auch offensichtlich Lösung des linearen Gleichungssystems

$$[\mathbf{S}]([\mathbf{x}] - \mathbf{c}) = -\mathbf{f}(\mathbf{c}). \tag{2.36}$$

Damit ist jede Kontraktion einer beliebigen Box $[\mathbf{x}]$ über das lineare Gleichungssystem (2.36) zulässig für das Gleichungssystem $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

Das Newton-Gauß-Seidel-Verfahren ergibt sich nun durch Anwendung des präkonditionierten Intervall-Gauß-Seidel-Verfahrens (Algorithmus 2.3) auf eine Linearisierung der Funktion $\mathbf{f}$ der Form (2.36). Den so gewonnenen Kontraktor bezeichnen wir auch als *Newton-Operator mit Steigungen*

$$\begin{array}{rcl} \mathcal{N}(\mathbf{f}, [\mathbf{x}]) : \mathbb{IR}^n_* & \to & \mathbb{IR}^n_* \\ ([x_1], \ldots, [x_n]) & \mapsto & \mathrm{PGS}([\mathbf{S}], -[\mathbf{f}](\mathbf{c}), [\mathbf{x}], \mathbf{c}). \end{array}$$

Wählt man als Spezialfall eine Linearisierung der Funktion über Ableitungen, also $[\mathbf{S}] = [\mathbf{Df}]([\mathbf{x}])$, so ergibt sich der *Newton-Operator mit Ableitungen* als

$$\begin{array}{rcl} \mathcal{N}(\mathbf{f}, [\mathbf{x}]) : \mathbb{IR}^n_* & \to & \mathbb{IR}^n_* \\ ([x_1], \ldots, [x_n]) & \mapsto & \mathrm{PGS}([\mathbf{Df}]([\mathbf{x}]), -[\mathbf{f}](\mathbf{c}), [\mathbf{x}], \mathbf{c}). \end{array}$$

An dieser Stelle wollen wir noch einmal auf die Präkonditionierung des resultierenden linearen Systems im Gauß-Seidel-Verfahren zurückkommen. Die Berechnung des Inverse-Midpoint-Präkonditionierers ist per definitionem zunächst nur für quadratische Systeme möglich, wogegen die optimalen Präkonditionierer grundsätzlich auch auf nicht-quadratische Systeme—und zwar *zeilenweise*—angewendet werden können. Wie wir später noch detailliert sehen werden, ist man vielfach nicht in der Lage, optimale Präkonditionierer zu berechnen (selbst wenn diese nach der Theorie existieren). Dies

würde aber bedeuten, dass bei nicht-quadratischen Systemen (bislang) keine Präkonditionierung im Newton-Gauß-Seidel-Verfahren eingesetzt werden kann. Da in diesem Fall das Newton-Verfahren in der Regel nicht effektiv genug ist, sind wir stark daran interessiert, einen sinnvollen Präkonditionierer zu entwickeln. Wir unterscheiden in SONIC grundsätzlich zwischen zwei verschiedenen Fällen.

Überbestimmte Systeme $(m > n)$

Dazu wird nach der ursprünglichen Idee des Inverse-Midpoint-Präkonditionierers für jede Variable, also für $k = 1, \ldots, n$, eine Präkonditionierungszeile $\mathbf{C_k}$ so berechnet, dass

$$\mathbf{C_k} \cdot \underbrace{\mathrm{mid}([\mathbf{Df}]([\mathbf{x}]))}_{=:\mathbf{A}} \approx \mathbf{e_k}^T$$

gilt. Wir bestimmen deshalb zunächst eine LU-Zerlegung $\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U}$ der Matrix $\mathbf{A}$ und lösen anschließend die Systeme

$$\left\{ \begin{array}{rcl} \mathbf{y}^T \cdot \mathbf{U} & = & \mathbf{e_k}^T \\ \mathbf{z}^T \cdot \mathbf{L} & = & \mathbf{y}^T \end{array} \right. ,$$

womit wir einen Präkonditionierer der gewünschten Form über $\mathbf{C_k} = \mathbf{z}^T \cdot \mathbf{P}$ erhalten: Der wesentliche Einfluss der k-ten Variable tritt gerade in der k-ten Gleichung auf. Bei der Implementierung ist darauf zu achten, dass die LU-Zerlegung nur *einmal* zu Beginn des Newton-Gauß-Seidel-Verfahrens bestimmt werden muss.

Unterbestimmte Systeme $(m < n)$

Für unterbestimmte Systeme verwenden wir in SONIC eine relativ einfache, aber effektive Präkonditionierung. Dazu bestimmen wir in der k-ten Spalte der Jacobi-Matrix $[\mathbf{M}] := [\mathbf{DF}]([\mathbf{x}])$ den Eintrag $[m_{i_0,k}]$ mit der größten Magnitude, also $|[m_{i_0,k}]| = \max_i |[m_{i,k}]|$. Dabei stellt die Magnitude der Einträge $[m_{i,k}]$ für $i = 1, \ldots, m$ ein ungefähres Maß für den Einfluss der Variablen i in der k-ten Gleichung dar. Anschließend setzen wir

$$\mathbf{C_k}^T = \mathbf{e}_{i_0} = (\underbrace{0, \ldots, 0}_{(i_0-1)-\mathrm{mal}}, 1, 0, \ldots, 0)^T \in \mathbb{R}^m .$$

Dieses Vorgehen macht auch noch aus einem zweiten Grund Sinn: Bei der Diskussion über die Pivot-Präkonditionierer (Seite 31) haben wir bereits gesehen, dass in vielen Fällen beim $\mathrm{C^W}$-LP-Präkonditionierer tatsächlich nur ein Eintrag in der entsprechenden Präkonditionierungszeile besetzt ist.

Dem aufmerksamen Leser wird an dieser Stelle aufgefallen sein, dass man beispielsweise für die Präkonditionierung ebenfalls eine Rang-anzeigende QR-Zerlegung verwenden kann. Diese hat gegenüber der LU-Zerlegung numerische Stabilitätsvorteile [14] und könnte eventuell geeignet zur Wahl der Variablen, nach denen bei unterbestimmten Systemen aufgelöst wird, eingesetzt werden. Allerdings ist momentan in SONIC eine hoch-optimierte LU-Zerlegung [66] integriert, die die besondere Struktur der Systeme berücksichtigt. Daher verzichten wir im Moment auf die Verwendung einer QR-Zerlegung.

Neben der Eigenschaft eines zulässigen Kontraktors bietet das Newton-Gauß-Seidel-Verfahren in vielen Fällen die Möglichkeit, die Existenz einer Lösung und bei Verwendung des Operators mit Ableitungen die Eindeutigkeit einer Lösung nachzuweisen. Um die Möglichkeit der Verifikation auszunutzen, muss der Newton-Operator allerdings modifiziert werden. Für den folgenden Satz darf im präkonditionierten Gauß-Seidel-Verfahren aus Algorithmus 2.3 der Schnitt mit dem ursprünglichen Intervall in Zeile 7 nicht durchgeführt werden. Wir erhalten somit den modifizierten Newton-Operator $\tilde{\mathcal{N}}(\mathbf{f}, [\mathbf{x}])$, für den gilt

$$\mathcal{N}(\mathbf{f}, [\mathbf{x}]) \subseteq \tilde{\mathcal{N}}(\mathbf{f}, [\mathbf{x}]) \cap [\mathbf{x}]. \tag{2.37}$$

Mit der Hilfe dieser Definition lässt sich nun der folgende Satz formulieren.

Satz 2.1 Sei $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ stetig auf der beschränkten Box $[\mathbf{x}] \in \mathbb{IR}^n$. Weiterhin seien $\mathbf{c} \in [\mathbf{x}]$ und $[\mathbf{S}]$ wie im Newton-Verfahren. Gilt für den modifizierten Newton-Operator

$$\tilde{\mathcal{N}}(\mathbf{f}, [\mathbf{x}]) \subseteq [\mathbf{x}],$$

dann folgt

(i) Die Steigungsmatrix $[\mathbf{S}]$ ist regulär.

(ii) Die Box $[\mathbf{x}]$ enthält eine Nullstelle von $\mathbf{f}$.

(iii) Ist $\mathbf{Df}([\mathbf{x}]) \subseteq [\mathbf{S}]$, also im Falle einer Linearisierung über Ableitungen, dann ist die Nullstelle sogar *eindeutig*.

Der Satz besteht aus einer Kombination von [38, Lemma 1.20], [38, Theorem 1.22] und [62, Satz 2.7], wo auch die entsprechenden Beweise zu finden sind. Gilt $\mathcal{N}(\mathbf{f}, [\mathbf{x}]) = \tilde{\mathcal{N}}(\mathbf{f}, [\mathbf{x}]) \cap [\mathbf{x}] = \emptyset$, so kann $[\mathbf{x}]$ keine Nullstelle enthalten und damit verworfen werden.

2.3.3 Constraint-Propagation (CP)

Nach den bereits präsentierten numerischen Kontraktionsverfahren wollen wir in diesem Abschnitt auch ein symbolisches Kontraktionsverfahren vorstellen. Dieses Kontraktionsverfahren bildet ein wichtiges Kernelement des Lösers und wird in der Regel als *Constraint-Propagation* bezeichnet. R. Baker Kearfott führt in [38] die alternative Bezeichnung *Substitution-Iteration* ein. Ziel dieses Verfahrens ist es, durch Ausnutzung der *symbolischen Information* aus der Termdarstellung der nichtlinearen Gleichungssysteme Suchboxen zu kontrahieren.

Das Constraint-Propagation, im Folgenden auch CP genannt, wurde in den unterschiedlichsten wissenschaftlichen Bereichen entwickelt und hauptsächlich im Bereich *Künstliche Intelligenz* und *Logikprogrammierung* eingesetzt. Detaillierte Einführungen zu diesem Thema findet der interessierte Leser in [38, 35]. Da das Thema Constraint-Propagation sehr umfangreich ist, wollen wir an dieser Stelle die verschiedenen Techniken nur an anschaulichen und ausgesuchten Beispielen erläutern. Eine sehr ausführliche Darstellung sowie eine genaue Analyse der *neuen Techniken* findet man in [66].

Als motivierendes Beispiel betrachten wir an dieser Stelle das nichtlineare Gleichungssystem $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$ mit

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} x_1^2 + \exp(x_2) \\ x_1^2 + x_2^2 - 1 \end{pmatrix} \tag{2.38}$$

und der Suchbox $[\mathbf{x}] = [0, 2]^2$. Hat man bereits mit einer beliebigen Intervallerweiterung eine Einschließung $[f_1]([\mathbf{x}]) = [0, 2]$ berechnet, so können durch *symbolisches Auflösen* nach jeder Variablen bzw. f_1 verbesserte Einschliessungen berechnet werden. Man erhält somit für die erste Gleichung des nichtlinearen Systems:

$$\begin{array}{rcll} x_1 & \in & \left(\pm\sqrt{[f_1]([\mathbf{x}]) - \exp([x_2])}\right) \cap [x_1] & = & [0, 1], \\ x_2 & \in & \ln([f_1]([\mathbf{x}]) - [x_1]^2) \cap [x_2] & = & [0, \ln 2], \\ f_1([\mathbf{x}]) & \in & \left([x_1]^2 + \exp([x_2])\right) \cap [f_1]([\mathbf{x}]) & = & [1, 2]. \end{array}$$

Das Gleichungssystem kann folglich in der angegebenen Suchbox keine Nullstelle enthalten. Bei der Berechnung der neuen Einschließung für x_1 haben wir bereits eine wichtige Technik benutzt, die im weiteren Verlauf dieser Arbeit (siehe Abschnitt 2.3.3.7) noch eine wichtige Rolle spielen wird. Als Zwischenergebnis erhalten wir bei der Berechnung $\pm\sqrt{[-e^2, 1]}$, also eine nicht-totale Funktion, da die Wurzelfunktion nur für nicht-negative Werte definiert ist. Offensichtlich ist es an dieser Stelle aber zulässig, das Intervall unter der Wurzel mit dem Definitionsbereich $[0, \infty]$ zu schneiden. Anschließend kann die Wurzel ohne Probleme mit Hilfe der Intervallarithmetik berechnet werden. Wir werden später noch genauer sehen, dass die Behandlung von nicht-totalen Funktionen ein besonderes leistungsfähiges Feature von SONIC ist.

Die Idee von Constraint-Propagation ist nun, die Prozedur des symbolischen Auflösens nach den Variablen auf jeden Knoten der Termbäume des Gleichungssystems anzuwenden. Aus diesem Grund wird in einer ersten Phase des Constraint-Propagation, der so genannten *Splitting-Phase*, das System (in diesem Abschnitt sprechen wir auch von *Constraints*) in elementare Operationen bzw. Funktionen aufgebrochen. Dies geschieht, ähnlich wie beim Automatischen Differenzieren, über die Einführung von neuen *Zwischenvariablen* $x_i (i > 2)$ für jeden Subterm des Termnetzes. Man spricht in diesem Fall vom *vollen Split des Systems*. Alternativ spricht man in einigen Fällen auch von einem *erweiterten System* oder einer *vollen Zerlegung*. Für das obige Beispiel aus (2.38) erhält man durch Aufsplittung der Funktionen f_1 und f_2:

$f_1(\mathbf{x})$	$f_2(\mathbf{x})$
$x_3 = x_1^2$	$x_5 = x_1^2$
$x_4 = \exp(x_2)$	$x_6 = x_2^2$
$0 = x_3 + x_4$	$x_7 = x_6 - 1$
	$0 = x_5 + x_7$

Diese Terme liegen in SONIC zunächst als Termwald, also als eine Liste von disjunkten Termbäumen vor. Durch Ausnutzung gemeinsamer Teilterme entsteht daraus ein

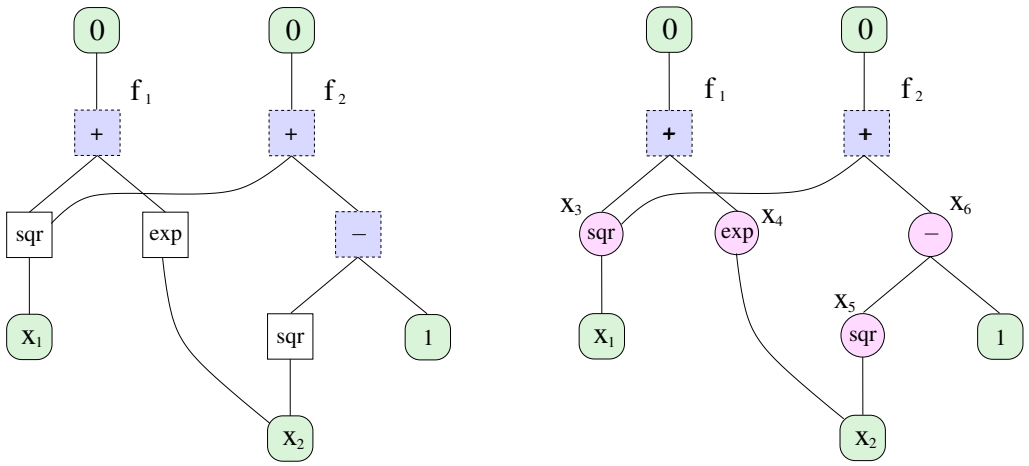

Abbildung 2.2: Die Termnetze für das Originalsystem (links) und das erweiterte System (rechts).

Termnetz. Für das Beispielsystem erhält man also unter Berücksichtigung gemeinsamer Teilterme und Umordnung der Terme das erweiterte System:

$$\mathcal{E} = \begin{cases} 0 &= x_3 + x_4 \\ 0 &= x_3 + x_6 \\ x_3 &= x_1^2 \\ x_4 &= \exp(x_2) \\ x_5 &= x_2^2 \\ x_6 &= x_5 - 1 \end{cases} \tag{2.39}$$

Dabei ist zu beachten, dass bei Entdeckung eines zweiten Vorkommens eines Teilterms die Variable für den ersten, bereits vorhandenen Term eingesetzt wird. Anschließend wird die Nummerierung der Zwischenvariablen geeignet angepasst (die Variablennummer wird um 1 reduziert). Eine schematische Darstellung der Termnetze sowohl für das Originalsystem als auch für das erweiterte System ist in Abbildung 2.2 skizziert. Das Auflösen der elementaren Gleichungen aus dem erweiterten System ist verhältnismäßig einfach. Eine ausführliche Darstellung der dabei benötigten Umkehrfunktionen ist in [66, Seite 60–70] zu finden. Wie bereits in der Einleitung erläutert, muss die Multiplikation mit Hilfe der relationalen Division aufgelöst werden. Allerdings kann durch die Anwendung der relationalen Division eine Vereinigung von Intervallen als Resultat enstehen. Aus diesem Grund arbeitet der Constraint-Propagator intern mit einer endlichen Vereinigung von Intervallen.

Um im Folgenden die Algorithmen besser darstellen zu können, vereinbaren wir, dass das erweiterte System eines Gleichungssystems $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ stets mit $\mathcal{E}$ bezeichnet wird. Das erweiterte System besitzt insgesamt N Variablen (die ersten n stimmen mit dem Originalsystem überein) und M Gleichungen. Die Anzahl der $(N - n)$ neu hinzugekommenen Variablen ist dabei gleich der Anzahl innerer Knoten (diese haben mindestens einen Vater und einen Sohn) im Termnetz des erweiterten Systems. Weiterhin gilt offensichtlich $N - n = M - m$. Für $1 \leq i \leq M$ bezeichne $\mathcal{E}_i$ die i-te Gleichung

von $\mathcal{E}$ und $\mathrm{VAR}(\mathcal{E}_i)$ die Menge der Variablen, die in $\mathcal{E}_i$ auftreten.

In der so genannten *Propagations-Phase* wird das symbolische Auflösen einzelner Gleichungen des erweiterten Systems wiederholt durchgeführt, bis keine Veränderung von System- oder Zwischenvariablen mehr auftritt. Der daraus resultierende Algorithmus ist schematisch in Alg. 2.5 dargestellt. Es ist zu beachten, dass die neu eingeführten Zwischenvariablen zunächst unbeschränkt sind, da offensichtlich keine Werte für diese bekannt sind.

Algorithmus 2.5 CONSTRAINT-PROPAGATION$(\mathcal{E}, [\mathbf{x}])$

Beschreibung: CP auf erweitertem System $\mathcal{E} : \mathbb{R}^N \to \mathbb{R}^M$ von $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ und der aktuellen Suchbox $[\mathbf{x}]$

1: $\mathcal{S} := \{1, \ldots, N\}$
2: **for** $i = (n+1), \ldots, N$ **do**
3: $\quad [x_i] = [-\infty, \infty]$
4: **end for**
5: **while** $\mathcal{S} \neq \emptyset$ **do**
6: $\quad$ entferne geeigneten „Knoten“ i aus $\mathcal{S}$ {Scheduling}
7: $\quad [\mathbf{x}'] \leftarrow$ Kontraktion von $[\mathbf{x}]$ durch symb. Auflösen von $\mathcal{E}_i$ nach allen $x_j \in \mathrm{VAR}(\mathcal{E}_i)$
8: $\quad$ **for** alle Variablen $x_j \in \mathrm{VAR}(\mathcal{E}_i)$ **do** {Propagations-Phase}
9: $\quad\quad$ **if** $[x'_j] \neq [x_j]$ **then**
10: $\quad\quad\quad \mathcal{S} = \mathcal{S} \cup \{k \mid 1 \leq k \leq N, k \neq i, x_j \in \mathrm{VAR}(\mathcal{E}_k)\}$
11: $\quad\quad\quad [x_j] = [x'_j]$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: **end while**

2.3.3.1 Zykel in Termnetzen

Der Ablauf des CP-Algorithmus wird problematisch, wenn *Zykel* im Termnetz auftreten. Diese Zykel treten auf, wenn eine Variable in einer Gleichung mehrfach vorkommt oder wenn zwei Gleichungen mehr als eine gemeinsame Variable haben. Da Variablen ebenfalls Teilterme sind, ist es ersichtlich, dass Zykel also eine Folge des Dependency-Problems sind. Treten Zykel im Termetz auf, so kann es unter Umständen sehr lange dauern, bis sich im CP-Algorithmus eine stabile Lösung einstellt. Ein sehr einfaches Beispiel für Zykel in Termnetzen ist in Abbildung 2.3 anhand der Gleichung $x_1^2 - x_1 = 0$ dargestellt (ein weiteres Beispiel für einen Zykel haben wir bereits in Abbildung 2.2 kennengelernt). Ein wesentlicher Aspekt bei der Behandlung von Zykeln ist, nach welcher Methode im Algorithmus jeweils das nächste zu kontrahierende Constraint ausgewählt wird. Wir werden darauf zu einem späteren Zeitpunkt noch gesondert eingehen.

2.3.3.2 Forward-Propagation

Nach der Durchführung der Splitting-Phase sind zunächst alle neu eingeführten Zwischenvariablen unbeschränkt, also $x_i = [-\infty, \infty]$ für $i > n$. Eine Methode zur Be-

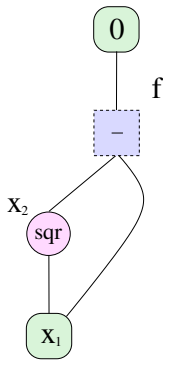

Abbildung 2.3: Zykel im Termnetz für die Gleichung $x_1^2 - x_1 = 0$.

schleunigung des CP-Algorithmus besteht nun darin, diese Variablen geeignet zu initialisieren. Dies kann sehr einfach durch die natürliche Intervallerweiterung der Terme realisiert werden. Eine neu erzeugte Splitvariable z im erweiterten System hat in der Regel die Form $z = \Phi(x, y)$, wobei Φ eine elementare Operation oder Funktion ist. Man erhält also eine geeignete Initialisierung der Zwischenvariablen z über die Auswertung $[z] = [\Phi]([x], [y])$. Dieser Vorgang wird in der Regel als *Forward-Propagation* bezeichnet. Die Namensgebung wird veranschaulicht durch die Tatsache, dass die Werte der (Zwischen-)Variablen „von unten nach oben“ durch das Termnetz *propagiert* werden.

Zu Beginn von CP erhält man mit Hilfe von Forward-Propagation eine Einschließung des Wertebereiches der Funktion auf der aktuellen Box (dies entspricht der natürlichen Intervallerweiterung). Aus diesem Grund bietet es sich an dieser Stelle an, den direkten Auswertungstest durchzuführen, da Constraint-Propagation sofort abgebrochen werden kann, wenn dieser Test fehlschlägt. Es ist allerdings zu beachten, dass man im Allgemeinen *keine* Einschließung des Wertebereiches durch Forward-Propagation mehr erhält, wenn z.B. Zeile 7 von Algorithmus 2.5 bereits einmal durchgeführt wurde. Dies ist auch der Fall, wenn das erweiterte Newton-Verfahren, das wir zu einem späteren Zeitpunkt kennenlernen, bereits angewendet wurde. Man erhält dann nur noch eine Einschließung des Wertebereiches auf den Nullstellen von $\mathbf{f}$ in der aktuellen Box. Dies ist allerdings eine wenig nützliche Aussage, da dies im optimalen Fall die Null selber sein sollte.

2.3.3.3 Backward-Propagation

Unter *Backward-Propagation* versteht man einen invertierten Forward-Propagation-Durchlauf. Angefangen bei den „Wurzeln der Termbäume“ (Knoten, die keinen Vaterknoten besitzen) werden also die neuen Werte der Variablen mit Hilfe des symbolischen Auflösens der elementaren Constraints von oben nach unten durch das Termnetz propagiert.

2.3.3.4 Forward-Backward-Propagation

Für Termnetze ohne Zykeln existiert eine effizientere Methode der symbolischen Kontraktion, als die lediglich durch die Scheduling-Heuristik (Zeile 6 in Algorithmus 2.5) gesteuerte Reihenfolge der primitiven Kontraktoren des CP-Algorithmus. Dieser Ansatz wird als *Forward-Backward-Propagation-Ansatz* (FBP) bezeichnet.

Bei Forward-Backward-Propagation werden, genau wie beim allgemeineren CP, lediglich die elementaren Gleichungen genutzt, jedoch in einer effizienteren Reihenfolge. Dieser Ansatz geht auf [35, Seite 78] zurück und ist von der folgenden Struktur:

1. Zunächst wird ein Durchlauf durch das Termnetz mit Forward-Propagation durchgeführt,
2. Anschließend folgt ein Backward-Propagation-Durchlauf. Dieser wird angewendet, um die Änderungen der Wurzelvariable(n) nach dem Forward-Propagation-Sweep wieder zu den Variablen zurückzuspielen.

Man kann zeigen, dass Forward-Backward-Propagation auf einem Constraint ohne mehrfache Variablen, also auf einem azyklischen Netz, *optimal* ist, d.h. dass kein anderes gültiges Kontraktionsverfahren eine bessere Einschließung berechnen kann [66, Satz 3.3.1].

2.3.3.5 Zusätzliche Kontraktoren in CP

Wir haben gesehen, dass Forward-Backward-Propagation auf Termnetzen ohne Zykeln ein mindestens genauso guter Kontraktor wie Constraint-Propagation ist. Es bietet sich aus diesem Grund an, das ursprüngliche Termnetz in azyklische *Subnetze* zu zerlegen, auf denen dann der Forward-Backward-Sweep durchgeführt wird. Die Subnetze werden dafür konzeptuell durch ein entsprechendes FBP-Constraint ersetzt und damit wie eine *elementare* Funktion oder Operation betrachtet. Anstatt des symbolischen Kontraktors wird innerhalb von CP nun Forward-Backward-Propagation aufgerufen.

Da im Allgemeinen ein Forward-Backward-Sweep wesentlich teurer als eine einfache symbolische Kontraktion eines elementaren Constraints ist, muss dieser Tatsache innerhalb der Scheduling-Strategie Rechnung getragen werden. Ein geeignetes Maß für die Priorität eines FBP-Constraints ist die Summe der Kosten für die symbolische Kontraktion der elementaren Constraints des Subnetzes.

Durch die Hinzunahme der FBP-Constraints hebt sich Constraint-Propagation von der Ebene der elementaren Constraints deutlich ab. Eine weitere Möglichkeit den CP-Algorithmus zu erweitern besteht darin, weitere zusätzliche Constraints hinzuzunehmen. Man kann dann zum Beispiel für einzelne Gleichungen des Systems eine Kontraktion mit dem Taylor-Verfahren erster oder zweiter Ordnung durchführen. Dieses hat den entscheidenden Vorteil, dass über den Scheduler von CP die einzelnen Kontraktionsverfahren basierend auf den Kosten und den Änderungen der Variablen gesteuert werden. Es ist allerdings zu beachten, dass eine Kontraktion mit dem Taylor-Verfahren nur für Gleichungen mit mehrfach vorkommenden Variablen Sinn macht.

2.3.3.6 Scheduling und Terminierung

Die Auswahl eines geeigneten Constraints zur Kontraktion innerhalb des CP-Algorithmus (Alg. 2.5, Zeile 6) wird als *Scheduling* bezeichnet. Durch das Scheduling wird im Wesentlichen festgelegt, wie die Änderung der Variablen im Netz propagiert werden. Aus diesem Grund wird jedem Constraint eine gewisse *Priorität* zugewiesen, die sowohl die Kosten für die Kontraktion als auch die Änderung der Variablen berücksichtigt. Eine genauere Beschreibung findet der interessierte Leser in [66, Seite 81]. Zu Beginn des CP-Algorithmus wird nun das Constraint mit der höchsten Priorität ausgewählt. Anschließend wird das entsprechende Kontraktionsverfahren ausgeführt und die Priorität auf Null zurückgesetzt.

Wir haben bereits erwähnt, dass der CP-Algorithmus, zum Beispiel beim Auftreten von Zykeln im Termnetz, unter Umständen sehr lange braucht, bis er gegen stabile Werte konvergiert. Es ist deshalb sinnvoll, den Algorithmus abzubrechen, wenn sich kaum noch Änderungen der Variablen ergeben oder bereits „zu viele“ Kontraktionen durchgeführt wurden. Der erste Fall wird dadurch realisiert, dass nur noch Constraints in $\mathcal{S}$ eingefügt werden, deren Änderung eine bestimmte untere Schranke überschreitet.

2.3.3.7 Nicht-totale Funktionen

In diesem Abschnitt wollen wir kurz auf die Behandlung nicht-totaler Funktionen eingehen. Betrachten wir dazu die simple Funktion $f(x) = \sqrt{x^2 - 1} - 1$ auf der Box $[x] = [-4, 4]$. Die Funktion ist auf der Box offensichtlich nicht total. Alle dem Autor bekannten Tools zum verifizierten Lösen von Gleichungssystemen, wie zum Beispiel GlobSol, brechen aus diesem Grund die Berechnung unmittelbar nach dem Start ab. Dies ist bei genauerer Betrachtung aber völlig unnötig, zumal die beiden Nullstellen $x = \pm\sqrt{2}$ in der Suchbox $[x]$ enthalten sind. Im Gegensatz zu den bereits vorgestellten Kontraktionsverfahren, die alle die Stetigkeit der Funktion voraussetzen, kann Constraint-Propagation trotzdem angewendet werden. Dies wird durch den Umstand ermöglicht, dass beim Auflösen der elementaren Funktionen die Werte der Variablen jeweils mit dem Definitions- bzw. dem Wertebereich geschnitten werden. Wir wollen dies an zwei Beispielen erläutern:

- Das elementare Constraint $y = \sqrt{x}$ sei aktuell mit beliebigen Werten $[x]$ bzw. $[y]$ belegt. Bei der Durchführung eines Forward-Propagation-Schrittes wird stets die Zuweisung $[y] \leftarrow \sqrt{[x] \cap [0, \infty]}$ durchgeführt.
- Ein weiteres elementares Constraint $y = \sin(x)$ sei ebenfalls mit beliebigen Werten $[x]$ und $[y]$ belegt. Bei der Durchführung von Backward-Propagation für dieses Constraint wird zunächst $[y]$ mit dem Wertebereich $[-1, 1]$ geschnitten.

Im Laufe der Zeit konnte festgestellt werden, dass bei bei vielen Systemen aus der Prozesstechnik das folgende Problem auftritt: Obwohl das nichtlineare Gleichungssystem komplett auf der gegebenen Suchbox definiert ist, treten durch starke Überschätzungen in den Intervall-Erweiterungen sehr schnell „Auswertungsfehler“ auf. Im Gegensatz zu anderen Lösern bricht SONIC die Berechnung an dieser Stelle nicht (unnötigerweise)

ab, sondern schaltet für die aktuell betrachtete Box alle Beschleunigungsverfahren ab, die die Stetigkeit der Funktion bzw. Ableitung auf der Box voraussetzen. Constraint-Propagation kann an dieser Stelle trotzdem eingesetzt werden. Wie sich in der Praxis zeigt, können durch wiederholte Anwendung von Bisektion und CP die Überschätzungen in den Intervall-Erweiterungen in der Regel so verringert werden, dass ab einem bestimmten Rekursionslevel auch weitere Verfahren hinzugeschaltet werden können. Diese besondere—und in der Tat recht simple—Vorgehensweise ermöglicht es, dass man heute nur mit Hilfe von SONIC in der Lage ist, *alle* Testsysteme aus der Verfahrenstechnik zu berechnen. Bei ca. 80% dieser Systeme versagen momentan alle anderen bekannten Tools

2.3.4 Splitting-Bisektion

In den vorherigen Abschnitten wurde bereits mehrmals darauf hingewiesen, dass innerhalb des Newton-Gauß-Seidel-Verfahrens durch die Verwendung der relationalen Division eine Box $[\mathbf{x}']$ entstehen kann, die eine oder mehrere Lücken enthält. Da bereits Constraint-Propagation auf einer endlichen Vereinigung von Intervallen (Divisionen) arbeitet, besteht deshalb in SONIC die Möglichkeit, alle Boxkomponenten *intern* als Divisionen mit gewünschter Präzision p, d.h. als Vereinigung von bis zu p Teilintervallen, zu verwalten. Greift man direkt auf bestimmte Boxkomponenten zu, so wird stets die konvexe Hülle der Divisionen zurückgegeben. Die Vereinigung der Intervalle wird in SONIC als Datentyp `Xdivision<p>` implementiert. Als besonders effektiv hat sich in der Praxis die Präzision `p = 2` erwiesen.

Setzt man den Datentyp `Xdivision<p>` beim Newton-Gauß-Seidel-Verfahren ein, so werden zunächst einmal alle Lücken in den Boxkomponenten „angesammelt“. Die nun gespeicherte Information über eventuell vorhandene Lücken kann genutzt werden, um bereits vor der Bisektion eine Komponente zu unterteilen. Dieses Vorgehen ist in vielen Fällen der normalen Bisektion vorzuziehen, da die Box nicht nur in zwei Hälften unterteilt wird, sondern sogar ein ganzes Stück aus dem Innern der Box herausgeschnitten wird. Diese Technik werden wir im Folgenden als *Splitting-Bisektion* bezeichnen.

Damit die Anzahl der Boxen nach Anwendung der Splitting-Bisektion nicht „explodiert“, verwenden wir in SONIC eine Variable `MaxSplittingBisections`, die die Anzahl der Splitting-Bisektionen, die durchgeführt werden dürfen, nach oben beschränkt. Dabei hat sich der Wert `MaxSplittingBisections = 4` als besonders geeignet herausgestellt. Nun wenden wir uns der Frage zu, welche Lücken in den Boxkomponenten zur Unterteilung genutzt werden. In der Literatur (z.B. [23]) wird vorgeschlagen, stets die größten Lücken zu nutzen. Ausführliche Tests von Paul Willems haben aber gezeigt, dass es in den meisten Fällen sinnvoll ist, den maximalen Durchmesser einer Lücke im Vergleich zur entsprechenden gesamten Boxkomponente zu nutzen. Dabei ist zu beachten, dass die Komponenten intern als Divisionen, also Vereinigung von Intervallen, gespeichert werden. Bei der Berechnung nutzen wir den Durchmesser der konvexen Vereinigung der Boxkomponente, nicht die Summe der Durchmesser der Teilintervalle. Damit nur Lücken genutzt werden, die einen entsprechend großen (relativen) Durchmesser haben, werden in SONIC nur Lücken berücksichtigt, die den

Wert `MinRelGapsizeForSplittingBisection` überschreiten. Die Belegung dieses Wertes mit 0.2 hat sich etabliert. An einigen Stellen wird in der Literatur davor gewarnt, die Bisektion tatsächlich durchzuführen, wenn die Lücke sehr nah am Rand liegt, also nur ein kleiner Teil „abgehobelt“ wird. Nach Erfahrung des Autors braucht in der aktuellen Implementierung darauf allerdings keine Rücksicht genommen werden, da durch den Faktor `MinRelGapsizeForSplittingBisection` in der Tat sichergestellt ist, dass nur Lücken entfernt werden, die bereits einen relativen Durchmesser überschreiten. In diesem Fall ist die Bisektion eher als Kontraktion zu betrachten. Allerdings sollte darauf geachtet werden, dass nicht zu kleine Werte eingestellt werden.

Die Verwendung der Splitting-Bisektion wird standardmäßig nach jeder Verwendung des Newton-Gauß-Seidel-Verfahrens durchgeführt. Die resultierenden Boxen werden anschließend bis auf eine in die Arbeitsliste $\mathcal{L}$ eingefügt. Die übriggebliebene Box wird als aktuell zu bearbeitende Box $[\mathbf{x}]$ übernommen. Sollte eine Teilbox der Genauigkeitsanforderung genügen, so wird diese in $\mathcal{R}$ eingefügt. Damit erfolgt der Aufruf des Newton-Verfahrens im Folgenden stets über

$$\textsc{Newton}([\mathbf{x}], \text{Präkonditionierer}, \mathcal{L}, \mathcal{R}, \texttt{gesplittet}) \ .$$

Wie man sieht wird nun auch der *Name* (intern eine Präkonditionierer-Nummer) des zu verwendenden Präkonditionierers übergeben. Sollte die Splitting-Bisektion tatsächlich erfolgreich durchgeführt worden sein, so wird dies in der Variablen `gesplittet` (mögliche Werte: `true`/`false`) festgehalten. Soll das Newton-Gauß-Seidel-Verfahren nur auf *einer* Gleichung durchgeführt werden, so genügt im Folgenden der Aufruf

$$\textsc{Newton}([\mathbf{x}], \text{Präkonditionierer}, \mathcal{L}, \mathcal{R}, \texttt{gesplittet}, i) \ .$$

In diesem Fall wird die i-te Gleichung des Systems nach der i-ten Variablen aufgelöst. Ist aus dem aktuellen Zusammenhang nicht ersichtlich, auf welchem System das Newton-Gauß-Seidel-Verfahren durchgeführt werden soll, so wird jeweils als erster Parameter das entsprechende Gleichungssystem eingefügt

$$\textsc{Newton}(\mathbf{f}, [\mathbf{x}], \text{Präkonditionierer}, \mathcal{L}, \mathcal{R}, \texttt{gesplittet}) \ .$$

Mit diesem Aufruf wird folglich ein kompletter Sweep des Newton-Gauß-Seidel-Verfahrens auf dem Gleichungssystem $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ mit der Startbox $[\mathbf{x}]$ durchgeführt.

2.3.5 Das erweiterte Intervall-Newton-Verfahren

In diesem Abschnitt stellen wir das so genannte (präkonditionierte) *erweiterte Newton-Gauß-Seidel-Verfahren* vor. Dieses Verfahren hat einen besonderen Bezug zu Constraint Propagation, da es ebenfalls auf den eingeführten Split- bzw. Zwischenvariablen arbeitet. Wir werden sehen, dass das geschickte Zusammenspiel von erweitertem Newton-Verfahren und Constraint-Propagation ein effektives Kontraktionsverfahren auf symbolischer *und* numerischer Ebene darstellt.

In der Splitting-Phase des Constraint-Propagation wurde ein erweitertes System durch Einführung neuer Variablen eingeführt, so dass in dem neuen System jede Gleichung aus einer einzigen elementaren Operation bzw. Funktion besteht. Für ein allgemeines, aus Übersichtsgründen quadratisches, nichtlineares Gleichungssystem $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ ist das erweiterte System $\mathcal{E}$ von der grundlegenden Struktur

$$\mathcal{E} \begin{cases} 0 &= \Phi_1(\cdot,\cdot) \\ & \dots \\ 0 &= \Phi_n(\cdot,\cdot) \\ x_{n+1} &= \Phi_{n+1}(\cdot,\cdot) \\ & \dots \\ x_N &= \Phi_N(\cdot,\cdot) \end{cases}, \tag{2.40}$$

wobei $\Phi(\cdot,\cdot)$ eine elementare Funktion oder Operation darstellt.

Da das präkonditionierte Newton-Gauß-Seidel-Verfahren nur für das Auffinden von Nullstellen eines Systems formuliert wurde, muss das erweiterte System $\mathcal{E}$ auf Normalform gebracht werden. Man erhält sehr leicht das modifizierte Gleichungssystem $\mathcal{E}'$ mit

$$\mathcal{E}' \begin{cases} 0 &= \Phi_1(\cdot,\cdot) \\ & \dots \\ 0 &= \Phi_n(\cdot,\cdot) \\ 0 &= x_{n+1} - \Phi_{n+1}(\cdot,\cdot) \\ & \dots \\ 0 &= x_N - \Phi_N(\cdot,\cdot) \end{cases} \tag{2.41}$$

Damit besteht $\mathcal{E}'$ aus N Gleichungen mit insgesamt N Variablen, ist also insbesondere wiederum quadratisch. Definiert man nun

$$\mathbf{f}_{\mathcal{E}} := \begin{pmatrix} \Phi_1(\cdot,\cdot) \\ \dots \\ \Phi_n(\cdot,\cdot) \\ x_{n+1} - \Phi_{n+1}(\cdot,\cdot) \\ \dots \\ x_N - \Phi_N(\cdot,\cdot) \end{pmatrix}, \tag{2.42}$$

so lässt sich offensichtlich das präkonditionierte Newton-Gauß-Seidel-Verfahren auf das nichtlineare Gleichungssystem $\mathbf{f}_{\mathcal{E}} = 0$ anwenden. Es ist zu beachten, dass dadurch eine Kontraktion *aller* Zwischenvariablen $x_1, \dots, x_N$ erzielt wird, obwohl man eigentlich nur an einer Kontraktion von $x_1, \dots, x_n$ interessiert ist. Aus Gründen der Übersichtlichkeit werden wir in einigen Fällen nicht mehr zwischen der Funktion $\mathbf{f}_{\mathcal{E}}$ und dem Gleichungssystem $\mathcal{E}'$ unterscheiden. Die genaue Bedeutung geht dann eindeutig aus dem Zusammenhang hervor.

Um eine geeignete Kontraktion mit dem Newton-Gauß-Seidel-Verfahren zu erreichen, müssen zunächst die Splitvariablen sinnvoll initialisiert sein. Dies könnte zum Beispiel durch einen einzigen Forward-Propagation-Sweep erreicht werden. Ausführliche Tests in [37] haben allerdings gezeigt, dass erweiterte Newton-Gauß-Seidel-Verfahren nur sinnvoll ist, wenn es im Zusammenhang mit Constraint-Propagation verwendet

wird. Aus diesem Grund betrachten wir im Folgenden das erweiterte Newton-Verfahren nicht nur als Newton-Gauß-Seidel-Verfahren für das erweiterte System, sondern integrieren als Vor- bzw. Nachlauf zusätzlich Constraint Propagation. Somit ergibt sich der in Alg. 2.6 skizzierte Algorithmus für das erweiterte Newton-Gauß-Seidel-Verfahren.

Algorithmus 2.6 ERWEITERTES NEWTON-GAUSS-SEIDEL-VERFAHREN($\mathcal{E}, [\mathbf{x}]$)

1: CONSTRAINT-PROPAGATION auf erweitertem System $\mathcal{E}$ und Box $[\mathbf{x}]$
2: führe NEWTON-GAUSS-SEIDEL-VERFAHREN auf $\mathcal{E}'$ und Box $[\mathbf{x}]$ aus
3: **if** Box $[\mathbf{x}]$ wurde genügend kontrahiert **then**
4: CONSTRAINT-PROPAGATION auf erweitertem System $\mathcal{E}$ und Box $[\mathbf{x}]$
5: **end if**

Wie die Untersuchungen in [66] zeigen, kann durch das erweiterte Newton-Verfahren die Gesamtzahl der Boxen teilweise erheblich reduziert werden. Allerdings hat dies in den meisten Fällen zur Folge, dass die Laufzeit des Algorithmus deutlich zunimmt. Die außerordentlich gute Qualität der Kontraktion ist laut [37] und [66] auf zwei Gründe zurückzuführen:

- Da das erweiterte System $\mathcal{E}'$ nur aus elementaren Operationen bzw. Funktionen besteht, ist es in gewisser Hinsicht „linearer“ als das Original-System. Aus diesem Grund sollte die im Newton-Verfahren integrierte Linearisierung effektiver sein.
- Die Ableitungsmatrix des erweiterten Systens $\mathcal{E}'$ ist extrem dünnbesetzt (maximal 3 Nicht-Null-Einträge pro Zeile) und von geringer Komplexität. In den meisten Fällen können deshalb die Ableitungswerte exakt eingeschlossen werden.

Im Hinblick auf die Laufzeit des erweiterten Newton-Verfahrens hat der Algorithmus den entscheidenden Nachteil, dass nun ein System der Größe $N \times N$ anstatt $n \times n$ berechnet werden muss, wobei in der Regel $N \gg n$ ist. So entsteht z.B. aus dem so genannten *7erSystem* mit je 7 Gleichungen bzw. Variablen ein System der Größe 430×430. Bei genauerer Betrachtung sieht man, dass im Vergleich zum Original-System nun etwa $\mathcal{O}((N/n)^3)$ mehr Operationen für *einen* Schritt des erweiterten Newton-Verfahrens anfallen, wobei in dieser Aufwandsabschätzung zusätzlich noch sehr große Konstanten enthalten sind. Man ist folglich daran interessiert, eine Strategie zu entwickeln, die einen guten Kompromiss zwischen beiden Verfahren darstellt. Aus dieser Idee ergibt sich der so genannte *hierarchische Ansatz*, den wir im nächsten Abschnitt vorstellen wollen.

2.3.6 Ein hierarchischer Ansatz

Das erweiterte Newton-Verfahren ergibt sich durch Anwendung des Newton-Gauß-Seidel-Verfahrens auf das erweiterte System $\mathcal{E}'$. Dieses erweiterte System entsteht durch die Einführung einer neuen Variable für *jeden* elementaren Operator des Termnetzes beim Constraint-Propagation. Durch die starke Kopplung dieser beiden Verfahren ergibt sich eine bedeutende Verschärfung der Kontraktion, die allerdings oftmals mit (zu) hohem Zeitaufwand einhergeht.

Eine Möglichkeit, den Zeitaufwand zu verbessern und gleichzeitig gute Kontraktionseigenschaften zu erhalten, ist es, das Newton-Gauß-Seidel-Verfahren auf *alternative Zerlegungen* des Systems anzuwenden. Bei diesen Zerlegungen wird nicht für jeden Operator eine neue Variable eingeführt, sondern nur für bestimmte.

In [66, Seite 103–106] werden einige Split-Strategien entwickelt, von denen wir an dieser Stelle nur die *Common-Subterms-Split-Strategie* vorstellen wollen. Wie der Name sagt, werden bei dieser Strategie nur neue Zwischenvariablen für gemeinsame Teilterme eingeführt. Eine genaue Analyse dieser Strategie findet man an der oben erwähnten Stelle.

Die unterschiedlichen Split-Strategien implizieren eine *Split-Hierarchie*, die von der gröbsten zur feinsten Split-Ebene reicht. In SONIC wird momentan die Split-Hierarchie $\sigma = \{$Original-System, Common-Subterms-Split, voller Split$\}$ genutzt. Die Grundidee des hierarchischen Ansatzes ist nun, für die aktuelle Suchbox die Split-Hierarchie von grob nach fein zu durchlaufen. Das erklärte Ziel dabei ist, die teilweise gewaltigen Sprünge der Laufzeit des Newton-Verfahrens nur zu akzeptieren, wenn das Verfahren auf den billigeren Split-Leveln nichts oder nur wenig bewirkt. Dies hat insbesondere den Vorteil, dass die Verfahrensaufrufe auf den feineren Leveln komplett gespart werden können, wenn die Box bereits auf einer gröberen Ebene verworfen werden kann. Für eine allgemeinere Split-Hierarchie $\sigma = \{\sigma_1, \ldots, \sigma_{\texttt{MaxSplitLevel}}\}$ ergibt sich der in Alg. 2.7 skizzierte Algorithmus.

Algorithmus 2.7 HIERARCHISCHER ANSATZ($[\mathbf{x}], \mathcal{L}, \mathcal{R}$)

```
1:  Schritte = 0
2:  SplitLevel = 1 {Original-System}
3:  while Schritte < MaxSchritte und SplitLevel ≤ MaxSplitLevel do
4:    Schritte = Schritte + 1
5:    direkter Auswertungstest für Box [x] auf σ_SplitLevel
6:    CONSTRAINT-PROPAGATION {arbeitet immer auf vollem Split}
7:    optionale Anwendung weiterer Kontraktionsverfahren (z.B. Taylor)
8:    if σ_SplitLevel ist stetig differenzierbar auf [x] then
9:      NEWTON(σ_SplitLevel, [x], *, L, R, gesplittet) {* ≙ bel. Präkonditionierer}
10:   end if
11:   if Box [x] konnte nicht genügend verkleinert werden then
12:     SplitLevel = SplitLevel + 1 {gehe auf das nächst-feinere Split-Level}
13:   else
14:     SplitLevel = 1
15:   end if
16: end while
```

Wir wollen nun kurz auf das Vorgehen auf einem bestimmten Split-Level eingehen. Wie bereits erwähnt macht das Verfahren nur im Zusammenhang mit Constraint-Propagation Sinn. Dies ist vor allem beim hierarchischen Ansatz der Fall, da CP eine geeignete Methode ist, um die Werte der Variablen aus den gröberen Split-Leveln mit denen aus der aktuellen Split-Ebene abzugleichen. Optional können nun weitere Kontraktionsverfahren angewendet werden (in den Standardeinstellungen von SONIC wird

das Taylor-Verfahren erster Ordnung zusätzlich genutzt). Die stetige Differenzierbarkeit, die für das Newton-Verfahren gebraucht wird, ergibt sich als „billiges Nebenprodukt“ des direkten Auswertungstests.

Die Einführung verschiedener Split-Level, sowie deren Einsatz innerhalb eines hierarchischen Ansatzes wurde in der Diplomarbeit [66] entwickelt. Es wird durch ausführliche Tests dargelegt, dass diese Technik bereits einen guten Kompromiss zwischen dem „normalen“ Newton-Verfahren und dem erweiterten Newton-Verfahren darstellt. Allerdings wurde in einigen, weiterführenden Untersuchungen festgestellt, dass sich die Wahl des Präkonditionierers auf den unterschiedlichen Split-Ebenen entscheidend auf die Performance von SONIC auswirken kann. Setzt man zum Beispiel den C^W-LP-Präkonditionierer im feinsten Split-Level ein, so benötigt beim 7erSystem die Ausführung von CheckBox auf *einer Box* mehrere Minuten. Wie wir später sehen werden, kann aber durch geschickte Auswahl der Verfahren das 7erSystem bereits in dieser Zeit vollständig gelöst werden. Entscheidenden Anteil hat dabei das so genannte *hybride Newton-Gauß-Seidel-Verfahren*, welches wir im nächsten Abschnitt präsentieren werden.

Der sehr vielversprechende Ansatz des *hierarchischen Ansatzes* wurde erst kürzlich entworfen und ist deshalb momentan ausschließlich in SONIC integriert. Da in der Regel das Newton-Verfahren auf dem vollen Split viel zu teuer ist, wird auf dessen Anwendung in den bekannten Tools zum verifizierten Gleichungslösen *komplett* verzichtet.

2.3.7 Ein hybrides Newton-Gauß-Seidel-Verfahren

Die Effizienz des hierarchischen Ansatzes hängt entscheidend von der geschickten Durchführung des Newton-Verfahrens auf den unterschiedlichen Split-Leveln ab. Bei der Implementierungs des Verfahrens muss nach den Erfahrungen des Autors auf zwei wesentliche Punkte besondere Rücksicht genommen werden:

1. Die Anwendung des Newton-Gauß-Seidel-Verfahrens auf ein feineres Split-Level ist in der Regel extrem *rechenintensiv*. Dies liegt in der Tatsache begründet, dass feinere Split-Ebenen oft große Gleichungssysteme implizieren. Man beachte noch einmal, dass zum Beispiel beim 7erSystem die Anzahl der Variablen von 7 im Originalsystem auf 430 im vollen Split steigt (man erhält also eine Aufblähung des Systems um den Faktor 60).

2. Im hierarchischen Ansatz (Algorithmus 2.7, Zeile 9) ist die Wahl des Präkonditionierers bislang offen geblieben. Nach Meinung des Autors bieten sich an dieser Stelle sowohl der Inverse-Midpoint-Präkonditionierer, als auch der C^W-LP-Präkonditionierer an. Dabei ist zu beachten, dass die Berechnung des zuletzt genannten Präkonditionierers ebenfalls sehr teuer ist. Eine Anwendung des Durchmesser-optimierenden Präkonditionierers auf den vollen Split erscheint deshalb nicht praktikabel.

Diese beiden Punkte haben uns dazu veranlasst, ein neues, *hybrides Newton-Gauß-Seidel-Verfahren* zu entwerfen. Dieses Verfahren wird in den hierarchischen Ansatz

integriert (Zeile 9) und dient als geschicktes Steuerungsmodul für das „normale Newton-Verfahren“.

Für die Implementierung des Verfahrens muss zunächst festgelegt werden, welche Präkonditionierer („maximal“) angewendet werden dürfen bzw. können. Aus diesem Grund wird eine Liste $\mathbf{C}^{(1)}, \ldots, \mathbf{C}^{(p)}$ möglicher Präkonditionierer festgelegt. In den aktuell gewählten Standardeinstellungen von SONIC werden—genau in dieser Reihenfolge—der C^{W}-LP- und der Inverse-Midpoint-Präkonditionierer, sowie die Identität (also keine Präkonditionierung) in die Liste eingetragen.

Das hybride Newton-Verfahren arbeitet auf jedem Split-Level komplett zeilenorientiert. Betrachten wir aus diesem Grund das Vorgehen des hybriden Newton-Verfahrens auf einer speziellen Gleichung: Für jede ausgewählte Zeile i des übergebenen Systems wird zunächst das Newton-Verfahren mit dem Präkonditionierer $\mathbf{C}^{(1)}$ angewendet. Hat sich das Volumen von $[\mathbf{x}]$ nur wenig oder gar nicht geändert, so wird auf der gleichen Zeile i erneut ein Newton-Schritt mit dem Präkonditionierer $\mathbf{C}^{(2)}$ angewendet, usw.. Dieses Vorgehen wurde unter anderem notwendig, weil für bestimmte Boxen zum Beispiel keine optimalen Präkonditionierer existieren bzw. die Berechnung des Inverse-Midpoint-Präkonditionierers fehlschlägt. Ist die Liste der gewünschten Präkonditionierer erschöpft oder ist die Box $[\mathbf{x}]$ bereits genügend kontrahiert worden, wird anschließend eine neue, geeignete Zeile i' gewählt und das Vorgehen wiederholt. Wurde zwischenzeitlich eine Boxkomponente von $[\mathbf{x}]$ sehr stark kontrahiert, so erscheint eine neue Berechnung von $[\mathbf{Df}]([\mathbf{x}])$ und $[\mathbf{f}](\mathbf{c})$ für die Linearisierung im Newton-Verfahren sinnvoll. Außerdem wird in diesem Fall erneut ein Durchlauf von Constraint-Propagation durchgeführt.

Die Qualität des hybriden Newton-Verfahrens hängt nun entscheidend von der Wahl der zu behandelnden Gleichungen ab. Da wir an einer besonders starken Kontraktion der Original-Variablen interessiert sind, wird in SONIC sichergestellt, dass für jede Zerlegung stets nach diesen aufgelöst wird. Nach einer ursprünglichen Idee von Willems sollte die Durchführung des Newton-Verfahrens auf feineren Split-Leveln nicht wesentlich länger dauern als die Durchführung auf dem Original-System. Aus diesem Grund wird nicht jede Gleichung des Split-Systems betrachtet, sondern eine bestimmte Schrittweite `StepSize` in den Zeilen gewählt, die von der Größe des Systems (oder besser des Split-Systems) abhängig ist. Man startet das Verfahren mit der Zeile $i = 0$. Um zu gewährleisten, dass stets auch die Original-Variablen kontrahiert werden (bzw. es zumindest versucht wird), wird im Fall $i < n$ die Schrittweite 1 gewählt. Der daraus resultierende Algorithmus ist in Alg. 2.8 dargestellt.

Integriert man nun das hybride Newton-Verfahren aus Alg. 2.8 in den existierenden hierarchischen Ansatz (Algorithmus 2.7), so ergibt sich bereits ein leistungsfähiges Werkzeug. In Tabelle 2.1 haben wir dies für einige repräsentative Testbeispiele dokumentiert. Dazu haben wir den hierarchischen Ansatz mit dem einfachen Newton-Verfahren (Inverse-Midpoint-Präkonditionierer auf allen Splits) und dem hybriden Newton-Verfahren (mehrere Präkonditionierer) verglichen. An den Ergebnissen wird schnell deutlich, dass das neue Verfahren großes Potential besitzt: Nur mit diesem ist es möglich, zwei Testbeispiele in angemessener Zeit zu lösen. Allerdings sieht man auch, dass bei anderen Beispielen die Anzahl der Boxen beim ursprünglichen hierarchischen

Algorithmus 2.8 HYBRIDES NEWTON-GAUSS-SEIDEL-VERFAHREN($[\mathbf{x}], \mathcal{E}, \mathcal{L}, \mathcal{R}$)

Kommentar: Gegeben sei Liste gewünschter Präkonditionierer $\mathbf{C}^{(1)}, \ldots, \mathbf{C}^{(p)}$

```
1: wähle geeignete Schrittweite StepSize (in Abhängigkeit von E)
2: do
3:    i = 0 {kann gegebenenfalls anders gewählt werden}
4:    while i ≤ n do {n entspricht der Anzahl der Zeilen von E}
5:       if i = 0  or  Volumen von [x] wurde stark verändert then
6:          werte Ableitung für die Linearisierung neu aus
7:          CONSTRAINT-PROPAGATION {arbeitet immer auf vollem Split}
8:       end if
9:       l = 1
10:      do
11:         NEWTON(E, [x], C^(l), L, R, gesplittet, i)
12:         l = l + 1
13:      while l ≤ p and [x_i] konnte nicht genügend kontrahiert werden
14:      if i < Anzahl Original-Variablen then
15:         i++ {stellt sicher, dass Original-Variablen kontrahiert werden}
16:      else
17:         i += StepSize
18:      end if
19:   end while
20: while [x] noch nicht klein genug and Volumen von [x] wurde stark verkleinert
```

Ansatz zwar zunimmt, die Gesamtzeit aber auch sinkt. Wie bereits in der Motivation erwähnt, verfolgen wir mit SONIC das Ziel, einen Löser zu entwickeln, der möglichst robust bezüglich der Parametereinstellungen ist. Da aber bei mindestens zwei Testbeispielen die ursprüngliche Variante dazu führt, dass die Ergebnisse nicht mehr akzetabel sind und mit dem neuen Verfahren nur einige leichte Einbußen in der Laufzeit zu verzeichnen sind, haben wir uns dafür entschieden, den hierarchischen Ansatz mit dem hybriden Newton-Verfahren *zunächst* als Standard-Einstellung zu wählen.

In einem nächsten Schritt hat der Autor Untersuchungen durchgeführt, warum das neue Verfahren zwar in einigen Fällen eine bessere Gesamtanzahl von betrachteten Boxen liefert, aber Einbußen in der Laufzeit hingenommen werden müssen. Wie zu erwarten war, hat sich herausgestellt, dass die Ursache im Umfeld der Berechnung des C^W-LP-Präkonditionierers zu finden ist. Es konnten zwei interessante Beobachtungen gemacht werden:

- In *allen* Testbeispielen, bei denen der hierarchische Ansatz mit dem einfachen Newton-Verfahren bessere Laufzeiten liefert als die neue Version, schlägt die Berechnung des C^W-LP-Präkonditionierers für eine große Anzahl von Boxen fehl (unbeschränkte Zielfunktion). In diesem Fall werden unnötig System-Ressourcen vergeudet, da ein rechenintensives lineares Programm gestartet wird, welches in der Regel nicht zum Erfolg führt. Offensichtlich kann an dieser Stelle die Laufzeit deutlich verbessert werden, wenn die Berechnung des LP-Präkonditionierers für

System	**Hierarchischer Ansatz**			
	mit Newton		mit hybridem Newton	
	Boxen	Zeit [s]	Boxen	Zeit [s]
CSTR_Cusp_Large	1 243	6.29	998	9.49
Agrawal_Hopf	9 451	118.29	4 993	115.37
Eco9	82 862	162.02	51 407	201.97
DesignProblem	64 226	302.83	39 393	415.57
DirectKinematics	80 556	1427.84	16 293	356.68
7erSystem	$> 50\,000$	$> 12\,769.02$	6 288	125.51
Arith. Mittel	48 056	2 464.38	19 895	204.09
Geom. Mittel	25 219	295.48	10 056	126.58

Tabelle 2.1: Anzahl der betrachteten Boxen für den hierarchischen Ansatz mit integriertem Newton-Verfahren und mit dem hybriden Newton-Verfahren für verschiedene Testprobleme.

geeignete Boxen abgeschaltet wird.

- Ist die Berechnung des C^W-LP-Präkonditionierers erfolgreich, so kann in Verbindung mit dem Newton-Verfahren stets eine *beachtliche* Kontraktion der aktuell betrachteten Box erreicht werden, die in dieser Form mit dem Inverse-Midpoint-Präkonditionierers in den meisten Fällen nicht geleistet werden kann. Ein komplettes Entfernen des C^W-LP-Präkonditionierers—auch im Hinblick auf die Lösbarkeit des 7erSystems—wäre also fahrlässig.

In einem weiteren Versuch wurde deshalb zunächst untersucht, ob Lemma 2.1 sinnvoll in unseren Ansatz integriert werden kann. Mit Hilfe dieses Lemmas kann *vor* der Berechnung entschieden werden, ob überhaupt ein C-Präkonditionierer für die aktuelle Box existiert. Leider konnten wir auf diese Weise keine Verbesserung in der Laufzeit verzeichnen. Im Gegenteil konnte die folgende erstaunliche Beobachtung gemacht werden: Wendet man den C^W-LP-Präkonditionierer nur an, wenn dieser nach Lemma 2.1 existiert, so ergibt sich bei einigen Systemen eine wesentlich größere Anzahl von betrachteten Boxen. Es scheint also, dass das lineare Optimierungsproblem in einigen Fällen den berechneten Präkonditionierer zumindest in „die Nähe“ eines guten Präkonditionierers bringt, auch wenn dieser in der Theorie gar nicht existiert. Aus diesem Grund haben wir uns gegen die Implementierung dieses Ansatzes entschieden.

Bei den zahlreichen, durchgeführten Untersuchungen konnte für *alle* problematischen Testbeispiele die folgende interessante Beobachtung gemacht werden. Lässt man sich für jede Box ausgeben, ob die Berechnung des C^W-LP-Präkonditionieres erfolgreich war, oder nicht, so sieht man folgendes Phänomen: Schlägt einmal die Berechnung fehl, so ist die Wahrscheinlichkeit relativ groß, dass die Berechnung auch auf den nächsten Boxen nicht funktioniert. Nach einer gewissen Zeit, kann die Berechnung des C-Präkonditionierers für eine größere Zahl von Boxen wieder erfolgreich durchgeführt werden, bis wiederum eine Reihe von Optimierungsproblemen nicht gelöst werden können. Aus diesem Grund führen wir in der aktuellen SONIC-Implementierung zwei neue Zähler

System	Hierarchischer Ansatz			
	hybr. Newton (Version 1)		hybr. Newton (Version 2)	
	Boxen	Zeit [s]	Boxen	Zeit [s]
CSTR_Cusp_Large	998	9.49	1 156	7.31
Agrawal_Hopf	4 993	115.37	5 621	104.92
Eco9	51 407	201.97	56 777	166.19
DesignProblem	39 393	415.57	45 564	128.61
DirectKinematics	16 293	356.68	25 184	311.27
7erSystem	6 288	125.51	6 288	125.77
Arith. Mittel	19 895	204.09	23 432	140.85
Geom. Mittel	10 056	126.58	11 772	92.87

Tabelle 2.2: Anzahl der betrachteten Boxen und Zeiten für den hierarchischen Ansatz mit der ursprünglichen und der zweiten Version des hybriden Newton-Verfahrens für verschiedene Testprobleme.

ein: `NoLP` und `NoSuccLP`. Die erste Variable zählt dabei, wie oft die Berechnung eines C^W-LP-Präkonditionieres vorgenommen wurde. Die zweite Variable führt Statistik darüber, wie oft diese Berechnung erfolgreich war. Gilt nun

$$\texttt{NoSuccLP} < \alpha \cdot \texttt{NoLP} \qquad \text{und} \qquad \texttt{NoLP} > \beta \tag{2.43}$$

für geeignetes α und β, so wird der C^W-LP-Präkonditionierer für eine gewisse Anzahl von Boxen im hybriden Newton-Verfahren abgeschaltet. Dabei garantiert die zweite Bedingung, dass die erste Bedingung nicht *zu früh* greift. Dieses Vorgehen ist vor allem zu Beginn des Algorithmus wichtig, da ansonsten beim ersten Fehlschlagen auf den C^W-LP-Präkonditionierer verzichtet wird. Ist eine gewisse Zeit ohne den LP-Präkonditionierer gerechnet worden, so werden beide Zähler erneut auf Null gesetzt. Greift zu einem späteren Zeitpunkt die Bedingung (2.43) erneut, so wird das Vorgehen wiederholt.

Wir haben diese Technik in das hybride Newton-Verfahren integriert und die daraus resultierenden Ergebnisse in Tabelle 2.2 aufgeführt. Wie zu erwarten war, hat sich im Mittel die Anzahl der betrachteten Boxen durchaus leicht erhöht. Dies liegt vor allem daran, dass in einigen Fällen der Durchmesser-optimierende Präkonditionierer nicht eingesetzt wurde, obwohl dies theoretisch möglich gewesen wäre. Weiterhin kann beobachtet werden, dass die Vermeidung von unnötigen Berechnungen des C^W-LP-Präkonditionierer tatsächlich eine Verbesserung in der Performance von SONIC bewirkt, obwohl eine Steigerung der Boxzahlen beobachtet werden kann. Einzig das 7erSystem liefert bei gleicher Boxzahl eine leicht erhöhte Laufzeit. Dies liegt allerdings daran, dass bei diesem Testsystem in keinem Fall die Berechnung des linearen Programms fehlschlägt (es existiert damit gegenüber der ursprünglichen Variante ein gewisser Overhead).

Bei genauerer Betrachtung sieht man, dass die implementierte Methode noch weiter verbessert werden kann. Die Entscheidung, dass man als Startzeile im hybriden Newton-Verfahren immer die erste Zeile wählt, bringt leider einen kleinen Nachteil mit sich: In feineren Split-Leveln werden bestimmte Gleichungen niemals betrachtet, da die

dort verwendetete Schrittweite `StepSize` in der Regel sehr viel größer als 1 ist. Es ist vermutlich effektiver, wenn man eine variable Startzeile zulässt, die von Zeit zu Zeit geändert wird. Dies hat zur Folge, dass in „gewisser Regelmäßigkeit“ alle Gleichungen einmal betrachtet werden. Aus diesem Grund haben wir in SONIC die Möglichkeit integriert, optional eine Startzeile (genauer gesagt: die Startzeile *nach* den Originalvariablen, da wir nach diesen *immer* auflösen wollen) über einen Zufallsgenerator zu wählen. Erste Tests haben gezeigt, dass sich durch dieses Vorgehen durchaus leichte Verbesserungen in der Performance ergeben. Allerdings hat dieses Vorgehen auch einen Nachteil bei der Parallelisierung des Algorithmus bzw. bei der Bewertung des parallelen Algorithmus. Selbst bei Verwendung eines „reproduzierbaren“ Zufallsgenerators kann nicht gewährleistet werden, dass für jede Box bei unterschiedlicher Prozessorzahl dieselbe Startzeile gewählt wird (jeder Prozessor hat in diesem Fall einen *eigenen* Zufallsgenerator). In der Regel werden sich also für verschiedene Prozessorzahlen unterschiedliche Gesamtboxzahlen ergeben, welches eine Vergleichbarkeit der Ergebnisse nicht mehr zulässt.

Im Folgenden wird im *hierarchischen Ansatz* (Algorithmus 2.7) die Anwendung des Newton-Verfahrens durch die Anwendung der zweiten Version des *hybriden Newton-Verfahrens* ersetzt (Zeile 9). Der Aufruf des Verfahrens erfolgt im Folgenden über

$$\textsc{Hierarchischer Ansatz}([\mathbf{x}], \mathcal{L}, \mathcal{R}, \texttt{gesplittet}) \quad .$$

Es ist zu beachten, dass gegenüber der ursprünglichen Variante ein zusätzlicher Parameter `gesplittet` eingefügt wurde, der aufgrund des Einsatzes des hybriden Newton-Verfahrens notwendig wurde.

Es sei an dieser Stelle noch einmal erwähnt, dass SONIC momentan das einzige Tool ist, welches das Newton-Verfahren auch auf alternativen Zerlegungen anwendet. In allen anderen Lösern ist die einzige Alternative die Durchführung des erweiterten Newton-Verfahrens auf dem *vollen Split*, was—wie oben bereits erwähnt—in der Regel viel „zu teuer“ ist. Die qualitativen Vorteile, die sich durch feinere Splits ergeben, werden damit nicht ausgenutzt.

2.4 Bisektionsstrategien

In diesem Abschnitt stellen wir einige Möglichkeiten vor, wie die im Verfahren gerade behandelte Box unterteilt werden kann. Für einen effektiven Branch–and–Bound-Ansatz spielen sowohl die Wahl der Unterteilungsrichtung, orthogonal zu der die Box geteilt wird, als auch die Wahl des Unterteilungspunktes eine sehr wichtige Rolle. In dieser Arbeit stellen wir allerdings nur Strategien vor, die nicht bezüglich des Verhältnisses, in dem das entsprechende Intervall geteilt wird, unterscheiden. Es erfolgt in jedem Fall eine Halbierung in der gewählten Richtung. Erste Untersuchungen zur Wahl des Unterteilungspunktes lassen allerdings vermuten, dass eine geschickte Wahl die Performance eines Branch–and–Bound-Lösers wesentlich verbessern kann.

Eine weitere Möglichkeit die aktuelle Box zu unterteilen ist die so genannte *Multisektion*. Bei dieser Methode wird die Box in insgesamt 2^l mit $2 < l \in \mathbb{N}$ Teilboxen

unterteilt. Diese Strategie findet ihre Anwendung allerdings fast ausschließlich in der globalen Optimierung (siehe Abschnitt 3.1.7). Eine interessante Alternative zur Bisektion könnte die Multisektion auch bei der Parallelisierung von nichtlinearen Lösern sein (siehe Kapitel 4).

Im Folgenden stellen wir insgesamt fünf *Grundstrategien* zur Unterteilung der aktuell betrachteten Box vor. Bei diesen Strategien gehen wir zunächst davon aus, dass die betrachtete Box beschränkt ist. Wir werden feststellen, dass jede dieser Strategien geeignet für einige Testbeispiele ist und wiederum für andere sehr schlecht funktioniert. Aus diesem Grund führen wir im Anschluss eine neue hybride Bisektionsstrategie ein, die darauf ausgerichtet ist, die für die aktuelle Box beste Grundstrategie auszuwählen. Zum Abschluss werden wir dann auf die Bisektion von Boxen eingehen, die teilweise oder komplett unbeschränkt sind.

2.4.1 MaxDiam

Die wohl älteste und bekannteste Bisektionsstrategie unterteilt die Box orthogonal zu der Komponente mit dem größten Durchmesser. Im Folgenden kürzen wir dieser Strategie mit **MaxDiam** ab. Durch diese bestimmte Wahl der Komponente wird sichergestellt, dass die Boxen in der aktuellen Arbeitsliste nach einer endlichen Zahl von Bisektionsschritten in jeder Komponente die geforderte Genauigkeit erreichen. Die Bisektionsstrategie **MaxDiam** bietet also die Sicherheit, dass der Löser stets terminiert.

2.4.2 MaxSmear

Diese Strategie wurde zuerst von Kearfott und Novoa in [41] vorgestellt. Sie geht auf eine Idee zurück, die wir bereits in einem der vorigen Abschnitte vorgestellt haben. Mit Hilfe der Taylor-Entwicklung erster Ordnung kann der Wertebereich der Funktion f_i über der Box $[\mathbf{x}]$ eingeschlossen werden:

$$[w_i] := f_i(\mathbf{c}) + \sum_{j=1}^{n} [J_{i,j}] \cdot ([x_j] - c_j),$$

wobei $[\mathbf{J}]$ eine Einschließung der Jacobi-Matrix $\mathbf{J} = \partial \mathbf{f} / \partial \mathbf{x}$ über der Box $[\mathbf{x}]$ und $\mathbf{c}$ der Mittelpunkt von $[\mathbf{x}]$ ist. Für den Durchmesser dieser Einschließungen gilt:

$$\operatorname{diam}([w_i]) = \sum_{j=1}^{n} |[J_{i,j}]| \cdot \operatorname{diam}([x_j]).$$

Die **MaxSmear**-Strategie wählt nun die Komponente j_0 der aktuellen Box, die den größten Einfluss auf einen der Durchmesser von $[w_i]([\mathbf{x}])$ hat, also

$$|[J_{i_0,j_0}]| \cdot \operatorname{diam}([x_{j_0}]) = \max_{i,j} |[J_{i,j}]| \cdot \operatorname{diam}([x_j]), \tag{2.44}$$

mit geeignetem i_0. Durch diese spezielle Wahl der Unterteilungsrichtung kann es passieren, dass eine Boxkomponente, die eigentlich schon die geforderte Genauigkeitsanforderung erreicht hat, weiter unterteilt wird. Alternativ könnte also Gleichung (2.44)

durch

$$|[J_{i_0,j_0}]| \cdot \operatorname{diam}([x_{j_0}]) = \max_{\substack{i,j \\ \operatorname{diam}([x_j])>\varepsilon_j}} |[J_{i,j}]| \cdot \operatorname{diam}([x_j]) \tag{2.45}$$

ersetzt werden, um diese „unnötige“ Unterteilung zu verhindern und die Terminierung des Lösers zu garantieren.

2.4.3 MaxSmearDiam

Die Strategie **MaxSmearDiam** ist—wie der Name bereits vermuten lässt—der Strategie **MaxSmear** sehr ähnlich. Sie wurde zuerst von Hansen [23] vorgestellt. Hier wird einfach die Magnitude aus Gleichung (2.44) durch den entsprechenden Durchmesser ersetzt. Es ergibt sich also für die Unterteilungsrichtung j_0 mit

$$\operatorname{diam}([J_{i_0,j_0}]) \cdot \operatorname{diam}([x_{j_0}]) = \max_{i,j} \operatorname{diam}([J_{i,j}]) \cdot \operatorname{diam}([x_j]), \tag{2.46}$$

und geeignetem i_0. Alternativ können auch hier wiederum nur die Komponenten berücksichtigt werden, die die gewünschte Genauigkeitsanforderung noch nicht erfüllen.

2.4.4 MaxSumMagnitude

Die Strategie **MaxSumMagnitude** [23] wählt die Unterteilungsrichtung j_0 so, dass sie den Beitrag der Richtung zu *allen* Komponenten von $\mathbf{f}$ maximiert, also

$$\sum_{i=1}^{m} |[J_{i,j_0}]| \cdot \operatorname{diam}([x_{j_0}])^{\alpha} = \max_{j} \sum_{i=1}^{m} |[J_{i,j}]| \cdot \operatorname{diam}([x_j])^{\alpha}.$$

Hierbei kann das Gewicht $\alpha > 0$ vom Benutzer frei eingestellt werden. Als Standard-Einstellung wird in SONIC der Wert $\alpha_{\text{default}} = 1$ gewählt. Eine detailliertere Beschreibung dieser Unterteilungsstrategie findet man in [62]. Um die Terminierung des Algorithmus zu erzwingen, können auch hier nur die Boxkomponenten berücksichtigt werden, die noch „zu groß“ sind.

2.4.5 ZeroNearBound

Das erklärte Ziel dieser Bisektionsstrategie ist es, die aktuelle Box so zu unterteilen, dass zumindest eine der neu entstandenen Teilboxen im nächsten Rekursionslevel verworfen werden kann, weil die Null nicht mehr in einer der Einschließungen $[f_i]([\mathbf{x}])$ mit $i = 1, \ldots, m$ enthalten ist. Liegt in einer der Komponenten i_0 des Bildes die Null sehr nah am Rand von $[f_{i_0}]([\mathbf{x}])$, so ist in vielen Fällen zu erwarten, dass eine Unterteilung der Box orthogonal zu der Komponente j_0, die $[f_{i_0}]([\mathbf{x}])$ am meisten beeinflusst, erfolgversprechend ist (siehe Abbildung 2.4).

Ein vielversprechender Kandidat für die Wahl von i_0 ist also die Komponente, die den Ausdruck

$$\delta_i := \min\{-\inf([f_i]([\mathbf{x}])), \sup([f_i]([\mathbf{x}]))\} \tag{2.47}$$

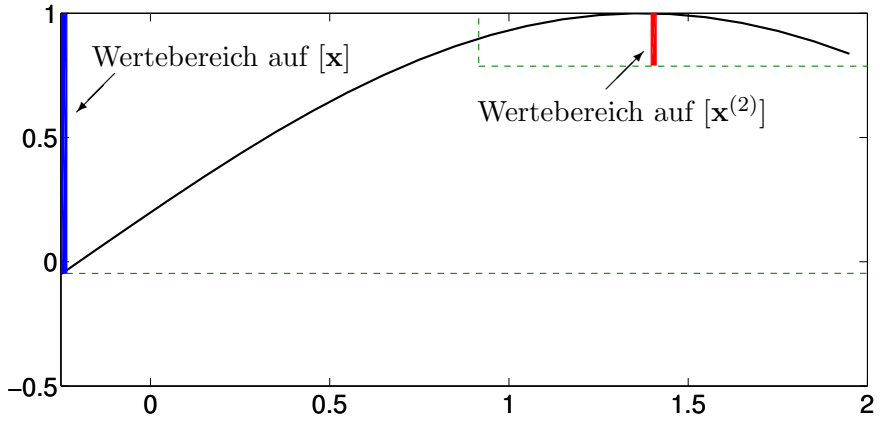

Abbildung 2.4: Exakter Wertebereich der Funktion $f = \sin(x + 0.2)$ auf der Suchbox $[x] = [-0.25, 2]$ und auf der Teilbox $[x^{(2)}]$. Die Box $[x^{(2)}]$ kann mit hoher Wahrscheinlichkeit im nächsten Rekursionlevel mit Hilfe des direkten Auswertungs-Tests verworfen werden.

minimiert. Hat man einmal die Komponente i_0 bestimmt, so wird als Unterteilungsrichtung die Komponente j_0 gewählt, die—ähnlich zur Bisektionsstrategie **MaxSmear**—den Ausdruck

$$|[J_{i_0,j}]| \cdot \operatorname{diam}([x_j])^\beta$$

maximiert. Hierbei kann das Gewicht $\beta > 0$ vom Benutzer ebenfalls frei eingestellt werden. In SONIC wird der Standardwert $\beta_{\text{default}} = 1$ genutzt.

Bei der Strategie **ZeroNearBound** kann die Terminierung des Algorithmus nicht mehr garantiert werden. Es kann passieren, dass stets $i_0 = 1$ gewählt wird und diese Komponente ausschließlich von einer einzigen Variable abhängt. In diesem Fall würde nur diese spezielle Komponente unterteilt. Auch wenn diese Situation dem Leser auf den ersten Blick konstruiert erscheint, so tritt sie in vielen praktischen Fällen doch recht häufig auf.

In (2.47) haben wir zur Auswahl einer geeigneten Gleichung ein absolutes Kriterium verwendet. Dieses kann natürlich auch durch ein relatives Kriterium für die Exzentrizität von $[f_i]([\mathbf{x}])$ ersetzt werden. Wir haben daher (2.47) testweise durch den Ausdruck

$$\tilde{\delta}_i := \frac{\min\{-\inf([f_i]([\mathbf{x}])), \sup([f_i]([\mathbf{x}])\}}{\max\{-\inf([f_i]([\mathbf{x}])), \sup([f_i]([\mathbf{x}])\}}$$

ersetzt, wobei wir wiederum die Komponente i_0, die $\tilde{\delta}_i$ minimiert, auswählen. Obwohl diese Variante in der Theorie besser geeignet scheint, hat sich in Tests herausgestellt, dass sich auf diese Weise keine wesentlichen Verbesserungen ergeben (die Anzahl der betrachteten Boxen variieren um unter ±2%). Da bei dem neuen Kriterium pro Gleichung zusätzlich eine Division anfällt und bei der Implementierung darauf geachtet werden muss, dass nicht durch Null geteilt wird, verzichten wir „aus Kostengründen“ auf dessen Anwendung.

System	A	B	C	D	E	Hybrid
Brown	4	2	6	3	6	6
Comb.	14	5	4	5	6	5
Cusp	9	51	12	16	14	24
D1	29	96	30	81	13	39
Agrawal1	75	60	37	56	47	47
Agrawal2	>1 000	16	10	13	244	13
Reac29	>1 000	148	422	195	>1 000	37
HHD	310	389	1 942	332	285	458
Reac7	4 950	13 539	3 953	12 746	7 945	3 770
Arith. Mittel	821	1 590	713	1 494	1 062	489
Geom. Mittel	106	71	60	59	83	47

Tabelle 2.3: Anzahl der betrachteten Boxen für verschiedene Testprobleme (in Tausend). Zuordnung: (A) **MaxDiam**, (B) **MaxSmear**, (C) **MaxSmearDiam**, (D) **MaxSumMagnitude**, (E) **ZeroNearBound**

2.4.6 Eine hybride Bisektionsstrategie

In den vorangehenden Abschnitten haben wir insgesamt fünf Grundstrategien zur Unterteilung der aktuellen (beschränkten) Box in zwei Teilboxen vorgestellt, die für den Einsatz in einem Branch–and–Bound-Löser geeignet sind. Da die Wahl der Bisektionsstrategie die Performance des Lösers entscheidend beeinflussen kann, stellt sich schnell die Frage, welche Strategie wohl die beste ist. Zur Beantwortung dieser Frage haben wir eine Vielzahl von Tests durchgeführt und eine repräsentative Auswahl in Tabelle 2.3 dargestellt. Um die Qualität der unterschiedlichen Bisektionsstrategien besser unterscheiden zu können, wurden sowohl das arithmetische als auch das geometrische Mittel über die Anzahl der betrachteten Boxen pro Testproblem berechnet. Das geometrische Mittel wird in dieser Arbeit zusätzlich angegeben, da das arithmetische Mittel in einigen Fällen einen entscheidenden Nachteil hat: Liegen in einer Testreihe sehr viele „große Werte“ und nur wenige „kleine Werte“ vor, so gehen die kleinen Werte weniger stark in das (arithmetische) Mittel ein. Das geometrische Mittel berücksichtigt alle Werte—unabhängig von ihrem genauen Wert—ungefähr gleich stark.

In einigen Fällen waren die Bisektionsstrategien **MaxDiam** und **ZeroNearBound** nicht in der Lage, die Berechnung innerhalb von 1 Millionen Boxen zu beenden. In diesen Fällen wurde für die Berechnung des arithmetischen und geometrischen Mittels jeweils 1 Millionen Boxen zu Grunde gelegt.

Es ist offensichtlich, dass die Frage nach der besten Bisektionsstrategie nicht zufriedenstellend beantwortet werden kann. Jede der fünf Grundstrategien liefert für eine Reihe von Testproblemen sehr gute Ergebnisse, führt aber auch in einigen Fällen zu einer massiven Anzahl von betrachteten Boxen (im Vergleich zu den anderen Strategien). Eine wichtige Zielsetzung bei der Implementierung war die *Robustheit* von SONIC gegenüber Parametereinstellungen: Die Grundeinstellungen des Lösers sollen bei *allen* Testproblemen ohne „Ausreißer“ gute Ergebnisse liefern. Aus diesem Grund haben

wir eine *hybride Bisektionsstrategie*—im Folgenden nur Strategie **Hybrid** genannt—entwickelt, die die schlechten Fälle verhindern soll, indem genau die Grundstrategie ausgewählt wird, die am vielversprechendsten für die aktuelle Box erscheint.

Eine wichtige Erfahrung, die wir in den letzten Jahren gemacht haben, ist die Tatsache, dass Boxen mit stark unterschiedlich großen Kantenlängen die Performance des Lösers negativ beeinflussen. In einem ersten Schritt der Strategie **Hybrid** wird also zunächst überprüft, ob die Variation der Kantenlängen ein gewisses Maß überschreitet. Dazu bezeichne

$$d_{\text{avg}} := \frac{1}{n} \sum_{i=1}^{n} \text{diam}([x_i])$$

die *durchschnittliche Kantenlänge* und

$$d_{\max} := \max_{1 \le i \le n} \text{diam}([x_i])$$

die *maximale Kantenlänge* der aktuellen Box $[\mathbf{x}]$. Falls der Wert $d_{\max}/d_{\text{avg}}$ größer als ein Parameter γ ist, so wenden wir die Strategie **MaxDiam** an. Als Standardwert hat sich in SONIC der Werte $\gamma_{\text{default}} = 20$ etabliert.

Ist die Variation der Kantenlängen nicht zu groß, so muss eine vielversprechendere Bisektionsstrategie gewählt werden. Es ist offensichtlich, dass die Strategie **ZeroNearBound** erst sinnvoll ist, wenn die Null sehr nah am Rand des Wertebereiches liegt. Eine weitere Beobachtung die wir gemacht haben, ist die Tatsache, dass diese Strategie nur für nicht zu kleine Boxen effektiv ist. Wir bestimmen also zunächst die Komponente i_0 und die Unterteilungsrichtung j_0 wie in der Strategie **ZeroNearBound**. Nur wenn die Einschließung $[f_{i_0}]([\mathbf{x}])$ nicht zu klein, also $\text{diam}([f_{i_0}]([\mathbf{x}])) > \delta$, und die Einschließung möglichst unsymmetrisch um den Ursprung, also $\text{diam}([f_{i_0}]([\mathbf{x}])) > \xi \cdot \delta_{i_0}$, ist, unterteilen wir die aktuelle Box orthogonal zu der Komponente j_0. Als geeignete—und vor allem robuste—Standardwerte haben sich die Parameter $\delta_{\text{default}} = 10^{-2}$ und $\xi_{\text{default}} = 3$ herausgestellt.

Erscheint weder die Strategie **MaxDiam** noch die Strategie **ZeroNearBound** erfolgversprechend für die aktuelle Box, so werden in einem letzten Schritt der hybriden Bisektionsstrategie die Komponenten i_0 und j_0 der Strategie **MaxSmearDiam** bestimmt. Ist der Durchmesser der Einschließung $[J_{i_0,j_0}]$ nicht zu klein, also $|[J_{i_0,j_0}]| < \lambda \cdot \text{diam}([J_{i_0,j_0}])$, so wählen wir als Unterteilungsstrategie **MaxSmearDiam**, anderenfalls **MaxSmear**. Als geeigneter Standardparameter hat sich in SONIC $\lambda_{\text{default}} = 1$ bewährt.

Die Tabelle 2.3 zeigt, dass die hybride Bisektionsstrategie für alle Testprobleme sehr gute—aber nicht notwendigerweise die besten—Ergebnisse liefert. Außerdem ist die ursprüngliche Forderung an die neue Strategie eindeutig erfüllt. Im Vergleich zu den wohlbekannten Grundstrategien reduziert unser Ansatz das Risiko, dass der Löser vor allem für große Probleme eine „unnötig hohe Anzahl“ von Boxen betrachtet. Auch über die hier betrachteten Beispiele hinaus ist dem Autor kein Testproblem bekannt, bei dem die Strategie **Hybrid** im Vergleich zu den Grundstrategien zu Ausreißern führt. Momentan ist SONIC das einzige Tool, welches eine hybride Strategie zur Bisektion nutzt. Bei den anderen bekannten Gleichungslösern muss der Benutzer entweder eine

(feste) Bisektionsstrategie wählen oder es steht gar nur eine Strategie—mit den oben beschriebenen Nachteilen—zur Verfügung.

2.4.7 Bisektion bei unbeschränkten Boxen

Ist die aktuell betrachtete Box unbeschränkt, so können die in den vorigen Abschnitten präsentierten Bisektionstrategien nicht genutzt werden, da die dort verwendeten Bewertungen in der Regel von der Boxgröße bzw. den Jacobi-Matrix-Einträgen abhängen. Um möglichst schnell die gute Qualität des hybriden Bisektionsverfahrens ausnutzen zu können, werden zwei Phasen durchlaufen. In der ersten Phase wird überprüft, ob eine beidseitig unbeschränkte Boxkomponente existiert. Ist dies der Fall, so wird diese in einem Punkt ε unterteilt (in einigen Tests hat sich herausgestellt, dass die Wahl von 0 als Unterteilungspunkt ungünstig ist). Dieses Vorgehen wird solange wiederholt, bis nur noch maximal halb-unbeschränkte Boxkomponenten übrig bleiben. Dabei beginnen wir in der Regel mit der ersten Boxkomponente und suchen dann zyklisch weiter. In der zweiten Phase sind folglich nur noch Komponenten der Form $[-\infty, a]$ bzw. $[a, \infty]$ vorhanden. Für diese Intervalle wird nun ein Unterteilungpunkt in der Nähe von a festgelegt. In der aktuellen Implementierung wählen wir

$$[-\infty, a] = [-\infty, a - |a|] \cup [a - |a|, a]$$

bzw.

$$[a, \infty] = [a, a + |a|] \cup [a + |a|, \infty] \ .$$

Bei diesem Vorgehen ist darauf zu achten, dass im Fall $a = 0$ eine leicht andere Variante gewählt wird, da sonst $|a| = 0$ gilt und keine sinnvolle Unterteilung mehr stattfindet.

Mit Hilfe dieser Methode ist man nun in der Lage, einige Systeme aus der Prozesstechnik *tatsächlich* verifiziert zu rechnen. Bisher hat man für einige Variablen nur sinnvolle Einschließungen verwendet, die ausschließlich auf den Erfahrungen der Ingenieure beruhen und damit keineswegs verifiziert sind. Nun ist es zum Beispiel für die Regularisierungsvariable aus dem System CSTR_Cusp_Large möglich, das Suchintervall vom geschätzten Bereich $[-0.000999, 0.001001]$ auf den tatsächlich „bekannten“ Bereich $[-\infty, \infty]$ zu setzen. Während die ursprüngliche Variante von SONIC insgesamt 999 Boxen und 11.12 Sekunden benötigte, steigt beim modifizierten System mit dem unbeschränkten Bereich die Anzahl der betrachteten Boxen auf 1241 in einer Laufzeit von 13.81 Sekunden (GlobSol benötigt für die ursprüngliche Variante bereits 3359 Boxen).

Es ist offensichtlich, dass dieses Vorgehen nur für eine bestimmte Problemklasse zum Erfolg führen kann. Eine mögliche Alternative könnte in diesen Fällen eventuell eine geeignete Variablen-Transformation darstellen. Hat eine Boxkomponente x den aktuellen Wert $[x] = [-\infty, a]$, so kann diese Komponente offensichtlich mit Hilfe der Transformation

$$x' = \frac{a - a^*}{x - a^*} \qquad \text{mit} \qquad a^* > a$$

in ein beschränktes Intervall überführt werden. Jedes Vorkommen von x in einem Term

wird anschließend durch den Ausdruck

$$a^* + \frac{a - a^*}{x'}$$

mit $[x'] = [0, 1]$ ersetzt. Dabei ist darauf zu achten, dass diese Variablentransformation vor der Berechnung der Ableitungen durchgeführt wird, um die Korrektheit der Ergebnisse zu garantieren. Für Variablen mit Einschließungen der Form $[a, \infty]$ bzw. $[-\infty, \infty]$ lassen sich ähnliche Transformationen angeben. Diese Methode wurde in SONIC integriert und kann optional zugeschaltet werden. Bei bisherigen Untersuchungen musste der Autor allerdings feststellen, dass diese Variante in Fällen, in denen das Standard-Verfahren fehlschlägt, ebenfalls keine Verbesserung erreicht werden kann. Dieses Verhalten könnte sich nach Meinung des Autors durchaus noch verändern, wenn andere Variablentransformationen eingesetzt werden. Eine genaue Untersuchung steht allerdings noch aus.

Um einige Beispiele aus der Prozesstechnik auch mit GlobSol verifiziert—also mit unbeschränkter Regularisierungsvariable—rechnen zu können, wurde eine Variablentransformation von Hand durchgeführt und in die entsprechenden Systeme integriert. Leider führte dieses Vorgehen ebenfalls nicht zum Erfolg, da GlobSol stets mit der Fehlermeldung „arithmetic exception“ abbricht. Nach Aussage von J.-P. Merlet ist der Löser ALIAS ebenfalls nicht in der Lage, Lösungen dieser Systeme (CSTR_Cusp, Teymour, Agrawal) zu berechnen. Damit ist SONIC momentan das einzige Tool, welches es ermöglicht, die gewünschten Systeme aus der Verfahrenstechnik verifiziert zu lösen.

2.4.8 Gezielte Splitting-Bisektion

In Kapitel 2.3.4 haben wir gesehen, dass sich die so genannte Splitting-Bisektion zum Aufteilen einer Box nach dem Newton-Gauß-Seidel-Verfahren nutzen lässt. Bei dieser Technik werden Lücken in den Boxkomponenten ausgenutzt, die durch die Anwendung der relationalen Division zum Auflösen nach den einzelnen Variablen entstehen. In der jetzigen Form entstehen die Lücken dabei *eher zufällig*.

Eine Methode, die Splitting-Bisektion *gezielt* durchzuführen, liegt in der Anwendung des S^M-LP-Präkonditionierers im Newton-Verfahren. Dieser Präkonditionierer zielt darauf, möglichst große Lücken während Durchführung des Verfahrens zu erzeugen. Eine mögliche Ergänzung des grundlegenden Branch–and–Bound-Algorithmus könnte also sein, vor der eigentlichen (konventionellen) Bisektion, einen *einzigen* Schritt des Newton-Verfahrens mit dem S^M-LP-Präkonditionierer auf das System anzuwenden. Konnte eine Lücke in einer Boxkomponente erzeugt werden, so wird diese verwendet und die hybride Bisektionsstrategie (für diese Box) abgeschaltet, da die Box nicht *zu oft* unterteilt werden soll. Dieses Vorgehen ist nach Meinung des Autors nur dann sinnvoll, wenn nicht zuvor bei der Anwendung der Routine CheckBox *zufällig* eine Lücke in der Box entstanden ist. In diesem Fall kann die in einer Boxkomponente erzeugte Lücke natürlich sofort genutzt werden. Mit diesen Überlegungen erhalten wir einen leicht modifizierten Branch–and–Bound-Algorithmus, der in Alg. 2.9 skizziert ist.

Die Ergebnisse des resultierenden Algorithmus haben wir in Tabelle 2.4 zusammengefasst. Wie man sieht, ergeben sich in den meisten Fällen durchaus Verbesserungen

Algorithmus 2.9 VERBESSERTER BRANCH–AND–BOUND-ANSATZ

1: $\mathcal{L} = [\mathbf{x}^{(0)}]; \mathcal{R} = \emptyset$
2: **while** $\mathcal{L} \neq \emptyset$ **do**
3: entferne die erste Box $[\mathbf{x}]$ aus $\mathcal{L}$
4: CHECKBOX($[\mathbf{x}], \mathcal{L}, \mathcal{R}$, `gesplittet`)
5: **if** $[\mathbf{x}]$ konnte nicht verworfen werden **then**
6: **if** `gesplittet` = `false` **then**
7: NEWTON($[\mathbf{x}]$, S^M-LP-Präkonditionierer, $\mathcal{L}, \mathcal{R}$, `gesplittet`)
{Zunächst auf allen Gleichungen, später auf einer vielversprechenden}
8: **end if**
9: **if** `gesplittet` = `false` **then**
10: führe HYBRIDE BISEKTIONSSTRATEGIE durch
11: **end if**
12: **end if**
13: ... {weiter wie im Original-Algorithmus}
14: **end while**

in der Anzahl der betrachteten Boxen. Allerdings gehen bei fast allen Testsystemen diese Verbesserungen mit einer gleichzeitigen (leichten) Erhöhung der Laufzeit einher. Besonders auffällig ist hier allerdings das DesignProblem: Obwohl die Boxzahl um mehr als 10% sinkt, verdreifacht sich die Laufzeit des Algorithmus. Dieses Verhalten ist bei den anderen Systemen (auch bei den zusätzlich durchgeführten Tests) in diesem starken Ausmaß nicht beobachtbar. Eine genauere Untersuchung hat ergeben, dass die Berechnung des linearen Programmes für den S^M-LP-Präkonditionierers dafür verantwortlich ist. Bei diesem Testproblem werden im Schnitt 300 Iterationen des Simplex-Algorithmus benötigt, während in der Regel bei anderen Systemen nur 50 bis maximal 100 Iterationen ausgeführt werden. Dieses Verhalten bestätigt also die—bereits zu Beginn dieser Arbeit getroffene—Vermutung, dass eine Obergrenze für die maximale Anzahl von Iterationen festgelegt werden sollte.

System	**Original-Algorithmus**		**mit Splitting-Bisektion**	
	Boxen	Zeit [s]	Boxen	Zeit [s]
CSTR_Cusp_Large	1 156	7.31	1 055	8.14
Agrawal_Hopf	5 621	104.92	5 798	112.99
Eco9	56 777	166.19	53 912	182.11
DesignProblem	45 564	128.61	39 429	366.18
DirectKinematics	25 184	311.27	23 317	391.21
7erSystem	6 288	125.77	4 282	102.25
Arith. Mittel	23 432	140.85	21 299	193.81
Geom. Mittel	11 772	92.87	10 445	116.13

Tabelle 2.4: Vergleich von Original-Algorithmus und dem Algorithmus mit anschließender Splitting-Bisektion an verschiedenen Testbeispielen.

Um die Performance von SONIC weiter zu verbessern, ist es unser Ziel, die gewonnene Reduktion der Boxzahlen beizubehalten, den LP-Präkonditionierer aber nur noch gezielt—also für bestimmte Boxen oder Funktionsgleichungen—einzusetzen. Wie unsere Erfahrungen zeigen, entstehen im Laufe des Algorithmus in der Regel nur sehr wenige, genügend große, Lücken in einzelnen Boxkomponenten, die sich *sinnvoll* für die Bisektion nutzen lassen. Auch aus diesem Grund ist ein kompletter Sweep des Newton-Verfahrens über alle Systemgleichungen bzw. für jede Box in den meisten Fällen kontraproduktiv, da ein rechenintensiver LP-Präkonditionierer bestimmt wird, der weder zu einem Split noch zu einer Kontraktion der aktuell betrachteten Box führt.

Kearfott trifft in seinem Buch [38] die Aussage, dass S-Präkonditionierer nur für Boxen effektiv arbeiten, die zwischen Lösungen liegen. Er schlägt vor, den S^M-LP-Präkonditionierer immer nur dann einzusetzen, wenn

$$\frac{\min\{|\inf([f_i]([\mathbf{x}]))|, |\sup([f_i]([\mathbf{x}]))|\}}{\max\{|\inf([f_i]([\mathbf{x}]))|, |\sup([f_i]([\mathbf{x}]))|\}} < \beta \tag{2.48}$$

für (mindestens) ein i und $\beta = 0.25$ gilt. Diese Aussage konnte in ausführlichen Tests, bei denen auch unterschiedliche Werte für β getestet wurden, in dieser strikten Formulierung nicht verifiziert werden. Nach Meinung des Autors ist momentan keine Heuristik veröffentlicht, die sinnvoll entscheiden kann, ob der Splitting-Präkonditionierer für eine bestimmte *Box* sinnvoll ist, oder nicht.

Wir haben uns aus diesem Grund dafür entschieden, die Newton-Bisektion mit dem S^M-LP-Präkonditionierer für jede Box durchzuführen, sofern nicht schon während der CheckBox-Phase eine Lücke entstanden ist. Allerdings wenden wir das Newton-Verfahren nicht auf *jede* Gleichung des Systems an, sondern nur auf eine *einzige, vielversprechende*, um Zeit einzusparen. Zu diesem Zweck verwenden wir ein Kriterium, welches wir bereits in Abschnitt 2.4.5 (Bisektionsstrategie **ZeroNearBound**) kennengelert haben. Folglich wählen wir als Gleichung i_0 für die Newton-Bisektion diejenige, die den Ausdruck δ_i in (2.47) minimiert.

Die Ergebnisse, die sich aus dieser Implementierung ergeben, haben wir für einige Testbeispiele in Tabelle 2.5 zusammengefasst. Offensichtlich ergibt sich durch das neue Verfahren nur eine leichte Erhöhung der betrachteten Boxen. Dieses Verhalten war aber zu erwarten, da nur durch eine relativ einfache *Heuristik* gesteuert wird, welche Systemgleichung für die Splitting-Bisektion ausgesucht wird. Es wird aber auch deutlich, dass die implementierte Heuristik durchaus vielversprechende Gleichungen auswählt. Gleichzeitig ergibt sich durch die neue Methode eine leichte bis mittlere Verbesserung in den Laufzeiten, die darauf zurückzuführen ist, dass nur noch *ein* lineares Optimierungsproblem pro Box gelöst werden muss. Auffällig (und leider auch unerklärlich) ist bei den Testergebnissen allerdings, dass beim 7erSystem im Prinzip keine Verbesserung der Laufzeit zu verzeichnen ist, obwohl die Anzahl der betrachteten Boxen praktisch gleich bleibt.

System	vollständiger Newton		Newton auf einer Gleichung	
	Boxen	Zeit [s]	Boxen	Zeit [s]
CSTR_Cusp_Large	1 055	8.14	1 130	7.59
Agrawal_Hopf	5 798	112.99	5 812	106.32
Eco9	53 912	182.11	54 001	162.48
DesignProblem	39 429	366.18	39 639	122.19
DirectKinematics	23 317	391.21	23 441	299.50
7erSystem	4 282	102.25	4 288	102.11
Arith. Mittel	21 299	193.81	21 385	133.37
Geom. Mittel	10 445	116.13	10 593	88.79

Tabelle 2.5: Vergleich von Splitting-Bisektion auf allen Gleichungen und der Splitting-Bisektion mit einer ausgewählten Gleichung an verschiedenen Testbeispielen.

2.5 Listenverwaltung

In diesem Abschnitt beschäftigen wir uns mit der Verwaltung von Boxen im grundlegenden Branch–and–Bound-Löser. In Zeile 19 von Algorithmus 2.1 werden die beiden durch Bisektion erhaltenen Teilboxen „geeignet“ in die Liste $\mathcal{L}$ eingefügt. Bis jetzt haben wir völlig offen gelassen, was man unter dem Begriff *geeignet* versteht.

Die wohl intuitivste und einfachste Art und Weise diese Operation zu realisieren, ist das Einfügen der beiden Teilboxen am *Ende* der Liste $\mathcal{L}$. Durch diese spezielle Anordnung der Boxen in der aktuellen Arbeitsliste wird der Rekursionsbaum aus Abbildung 2.1 mit Hilfe der Breitensuche (engl.: breadth-first) durchlaufen.

Der in Alg. 2.1 vorgestellte Branch–and–Bound-Löser von SONIC liefert als Resultat eine Liste $\mathcal{R}$ von „kleinen“ Boxen, die *alle* Lösungen des nichtlinearen Gleichungssystems in der Startbox $[\mathbf{x}^{(0)}]$ überdecken. Enthält die Lösungsmenge eine Mannigfaltigkeit M, dann enthält die Liste $\mathcal{R}$ eine Überdeckung von $[\mathbf{x}^{(0)}] \cap M$. Dies kann bei einer hohen Genauigkeitsanforderung zu einer extrem großen Anzahl von Boxen in $\mathcal{R}$ führen.

Wie wir bereits in der Einleitung erläutert haben, ist man bei vielen Gleichungssystemen aus der Verfahrenstechnik nur daran interessiert nachzuweisen, dass Lösungen in der Startbox existieren oder eben nicht. Im ersten Fall werden für den nächsten Optimierungsschritt nicht alle Lösungen, sondern nur *eine* benötigt. Es wäre also in diesem Fall (eine weitere Anwendung ergibt sich in Abschnitt 2.7) eine unnötige Verschwendung von Ressourcen, wirklich alle Lösungen berechnen zu lassen.

Man ist also in vielen Fällen besonders daran interessiert eine Lösung möglichst schnell zu finden. Neben der Verbesserung der so genannten Beschleunigungsmechanismen spielt dabei vor allem die *Umordnung der Listenelemente* in $\mathcal{L}$ eine entscheidende Rolle. Um diese Umordnung zu realisieren, weisen wir jeder Box aus der Liste $\mathcal{L}$ eine bestimmte *Kennzahl* oder auch *Priorität* zu. Je kleiner die Kennzahl einer Box ist, desto höher soll die Wahrscheinlichkeit sein, dass diese Box eine Lösung enthält. Die Zuweisung und das Auslesen der Kennzahl einer Box erfolgt in SONIC über die Methoden SET_RANK bzw. GET_RANK aus der Klasse `Box`.

System	Breadth-First	ME	MMPE
Combustion	85	28	136
Reactor_SaddleNode	126	231	311
Brent	702	336	340
Agrawal_Hopf	1 498	550	629
7erSystem	4 218	527	1 985
DirectKinematics	5 463	3 055	3 699
DesignProblem	5 840	724	30 661
Arithm. Mittel	2 562	779	5 349
Geom. Mittel	1 061	391	1 107

Tabelle 2.6: Vergleich der drei Strategien zur Verwaltung von Boxen

```
class Box{
  private:
    IVector *aBox;
    double  rank;
    ...
  public:
    void   set_rank( double );
    double get_rank( void );
    ...
};
```

Eine Zuweisung der Priorität 0.7 zu einer Box $[\mathbf{x}]$ erfolgt damit über die Anweisung $[\mathbf{x}]$.SET_RANK(0.7).

Als geeignete Datenstruktur für $\mathcal{L}$ erweist sich der *Heap* [12]. Dieser erreicht zwar keine vollständige Ordnung seiner Elemente, es kann aber garantiert werden, dass der Wert eines Vaterknotens immer kleiner oder gleich dem Wert seiner Söhne ist. Ein Vergleich der Prioritäten der Söhne ist nicht möglich. Damit ist gewährleistet, dass in der Wurzel des Heaps stets die Box mit der kleinsten Priorität gehalten wird. Durch die Auswahl des Heaps ergibt sich gegenüber der ursprünglichen linearen Liste ein gewisser Overhead. Während beim Breadth-First-Ansatz alle Listenoperationen einen Aufwand von $\mathcal{O}(1)$ haben, ergibt sich bei der Einfüge- und Lösch-Operation für einen Heap der Länge n jeweils ein Aufwand von $\mathcal{O}(\log n)$. Beide Datenstrukturen sind in SONIC in der Klasse `boxmanager` implementiert. Durch Setzen der boolschen Klassenvariablen `boxmanager::UseAsHeap` wird eine der Datenstrukturen ausgewählt. Dadurch wird sichergestellt, dass auch für die lineare Liste kein Overhead in Kauf genommen werden muss.

Wie wir oben festgestellt haben, müssen wir nur noch jeder Box eine Kennzahl zuweisen. Die Idee ist nun, dass diese Kennzahl ungefähr die Wahrscheinlichkeit beschreiben soll, dass die Box eine Lösung des Systems enthält. Wir folgen hier einem Ansatz von Jean-Pierre Merlet [47], der insgesamt zwei Bewertungsmodelle vorstellt.

Maximum equation ordering (ME)

$$\begin{aligned} C &:= \max_{1\le k\le m} \{|\inf([F_k]([\mathbf{x}]))|, |\sup([F_k]([\mathbf{x}]))|\} \\ &= \max_{1\le k\le m} \{-\inf([F_k]([\mathbf{x}])), \sup([F_k]([\mathbf{x}]))\} \end{aligned} \tag{2.49}$$

Maximum middle-point equation ordering (MMPE)

$$C := \max_{1\le k\le m} \{|\inf([F_k]([\tilde{\mathbf{x}}]))|, |\sup([F_k]([\tilde{\mathbf{x}}]))|\} \tag{2.50}$$

Bei diesen beiden Anordnungen besitzt diejenige Box mit der *kleinsten* Kennzahl die höchste Wahrscheinlichkeit, dass sie eine Lösung enthält.

Mit einer Vielzahl von Tests haben wir untersucht, welche der drei vorgestellten Strategien schneller zu einer Lösung führt. Die Ergebnisse haben wir in Tabelle 2.6 dargestellt, wobei wiederum nur eine kleine Anzahl repräsentativer Testbeispiele ausgewählt wurde. Man sieht deutlich, dass die Maximum-Equation-Anordnung den anderen Strategien überlegen ist.

Wird in SONIC also das besonders schnelle Auffinden einer (möglichen) Lösung gewünscht, so ist der User über die boolsche Variable `TerminationOnFirstHit` in der Lage, die Maximum-Equation-Anordnung der Liste $\mathcal{L}$ einzustellen. Ist eine Lösung (verifiziert) gefunden, so wird der Berechnungsprozess unmittelbar abgebrochen. Dieses Feature von SONIC ist momentan einzigartig unter den bekannten Tools, obwohl es extrem einfach zu implementieren ist. Bei einigen Systemen aus der Prozesstechnik konnte so der Berechnungsaufwand von einigen Stunden oder Minuten auf wenige Sekunden reduziert werden.

2.6 Verifikation

Ein wichtiges Ziel eines verifizierenden nichtlinearen Lösers ist es, Aussagen über die Existenz und Eindeutigkeit von Lösungen in Suchboxen zu treffen. Dieser Prozess wird im Allgemeinen als *Verifikation von Lösungen* bezeichnet.

Bereits zu Beginn dieses Abschnittes möchte ich auf eine subtile Fehlerquelle hinweisen. Ergibt sich durch den direkten Auswertungstest, also durch Anwendung einer beliebigen Intervallerweiterung, dass die Null in der entsprechenden Einschließung liegt, so verifiziert dies aus zwei Gründen *nicht* die Existenz einer Nullstelle. Zunächst ist es möglich, dass die Null ausschließlich durch Überschätzungen in der Einschließung des Wertebereiches der Funktion liegt. Ein weiterer Grund ist die Tatsache, dass beispielsweise die erste Gleichung des Systems durch einen Punkt $\tilde{\mathbf{x}} \in [\mathbf{x}]$ und die zweite Gleichung nur durch einen weiteren Punkt $\mathbf{z} \in [\mathbf{x}]$ mit $\tilde{\mathbf{x}} \neq \mathbf{z}$ erfüllt ist. Damit ist weder $\tilde{\mathbf{x}}$ noch $\mathbf{z}$ eine Lösung des Gleichungssystems.

Neben der aus Satz 2.1 bekannten Verifikation mit Hilfe des Newton-Verfahrens, das sogar Eindeutigkeit von Lösungen nachweisen kann, stellen wir zunächst neue bzw. weiterentwickelte Verfahren vor, die die Existenz von Nullstellen nachweisen. Anschließend werden wir Methoden vorstellen, wie man in bestimmten Fällen von der Existenz

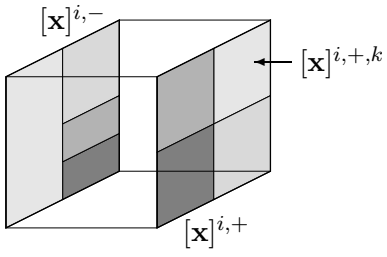

Abbildung 2.5: Aufteilung gegenüberliegender Facetten für den Miranda-Test.

auf die Eindeutigkeit einer Lösung schließen kann. Während die Verifikation mit Hilfe des Newton-Verfahrens auch *online*, also innerhalb der Routine CheckBox, geschieht, werden die nachfolgend präsentierten Verifikationsroutinen in der Regel nach *Beendigung* des Branch–and–Bound-Algorithmus durchgeführt. Dazu werden sukzessive alle Elemente der Lösungsliste $\mathcal{R}$ betrachtet. Auf die genaue Implementierung werden wir am Ende dieses Abschnittes eingehen.

Für einen (beschränkten) Intervallvektor $[\mathbf{x}] \in \mathbb{IR}^n$ definieren wir n Paare gegenüberliegender *Facetten* über

$$\begin{aligned} [\mathbf{x}]^{i,+} &:= ([x_1], \ldots, [x_{i-1}], \overline{x}_i, [x_{i+1}], \ldots, [x_n])^T \\ [\mathbf{x}]^{i,-} &:= ([x_1], \ldots, [x_{i-1}], \underline{x}_i, [x_{i+1}], \ldots, [x_n])^T \end{aligned}$$

mit $1 \leq i \leq n$. Damit ist der *topologische Rand* $\partial[\mathbf{x}]$ des Intervallvektors $[\mathbf{x}]$ gerade die Vereinigung dieser n Paare:

$$\partial[\mathbf{x}] = \bigcup_{i=1}^{n} \bigcup_{s \in \{+,-\}} [\mathbf{x}]^{i,s}. \tag{2.51}$$

Damit kann der topologische Rand mit $2n$ Boxen aus $\mathbb{IR}^n$ überdeckt werden, wobei jede dieser Boxen mindestens eine Komponente mit Durchmesser Null besitzt.

2.6.1 Miranda-Test

Der Satz von Miranda [50] verallgemeinert den Zwischenwert-Satz auf höhere Dimensionen: Nimmt eine stetige Funktion $\mathbf{f}$ auf allen gegenüberliegenden Facetten unterschiedliche konstante Vorzeichen an, dann besitzt $\mathbf{f}$ eine Nullstelle in $\mathbf{x}$.

Satz 2.2 (Miranda) Sei $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ eine stetige Funktion und $[\mathbf{x}] \in \mathbb{IR}^n$ eine Box. Falls

$$f_i(\mathbf{x}^-) \leq 0 \leq f_i(\mathbf{x}^+) \tag{2.52}$$

für alle $\mathbf{x}^+ \in [\mathbf{x}]^{i,+}, \mathbf{x}^- \in [\mathbf{x}]^{i,-}$ und $i = 1, \ldots, n$ gilt, dann besitzt $\mathbf{f}$ in $[\mathbf{x}]$ eine Nullstelle.

Um den Miranda-Test in der Praxis effektiv zu nutzen, wird die Bedingung (2.52) für eine präkonditionierte Funktion $\mathbf{g}(\mathbf{x}) = \mathbf{C} \cdot \mathbf{f}(\mathbf{x})$ überprüft, wobei $\mathbf{C} \in \mathbb{R}^{n \times n}$ eine

nicht-singuläre Matrix ist. In der Regel werden die Präkonditionierer $\mathbf{C} = \mathbf{I}_{n\times n}$ und $\mathbf{C} \approx \mathbf{Df}(\check{\mathbf{x}})^{-1}$ verwendet. Somit impliziert die Gültigkeit der Bedingung

$$\sup([g_i]([\mathbf{x}]^{i,-})) \leq 0 \leq \inf([g_i]([\mathbf{x}]^{i,+})) \tag{2.53}$$

für alle $i = 1, \ldots, n$, die Existenz einer Nullstelle in der Box $[\mathbf{x}]$. Um die Auswirkungen von Überschätzungen, die unweigerlich bei der Auswertung von (2.53) auftreten, zu reduzieren, werden die Facetten in Teilfacetten $[\mathbf{x}]^{i,+} = \bigcup_k [\mathbf{x}]^{i,+,k}$ und $[\mathbf{x}]^{i,-} = \bigcup_l [\mathbf{x}]^{i,-,l}$ unterteilt. Die Unterteilung kann dabei völlig unabhängig voneinander erfolgen. Allerdings müssen die Teilfacetten disjunkt (bis auf die Ränder) sein. Es ergibt sich der alternative Test

$$\begin{cases} \sup([g_i]([\mathbf{x}]^{i,-,l})) & \leq & 0 \\ \inf([g_i]([\mathbf{x}]^{i,+,k})) & \geq & 0 \end{cases} \tag{2.54}$$

für alle $i = 1, \ldots, n$ und alle l bzw. k.

2.6.2 Borsuk-Test

Neben den Newton- und Miranda-Verifikationstests, die beide auf dem Fixpunktsatz von Brouwer basieren, existiert ein weiterer Test, der die Existenz einer Nullstelle in einer Box bei quadratischen Systemen nachweist. Der so genannte *Borsuk-Test* basiert auf dem folgenden Satz aus [10].

Satz 2.3 (Borsuk) Sei $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ eine stetige Funktion und $[\mathbf{x}] \in \mathbb{IR}^n$. Gilt

$$\mathbf{f}(\text{mid}([\mathbf{x}]) + \mathbf{y}) \neq \lambda \mathbf{f}(\text{mid}([\mathbf{x}]) - \mathbf{y}) \tag{2.55}$$

für alle $\mathbf{y}$, so dass $\text{mid}([\mathbf{x}]) + \mathbf{y}$ in einer Facette $[\mathbf{x}]^{i,+}$ enthalten ist und alle $\lambda > 0$, dann hat $\mathbf{f}$ eine Nullstelle in $[\mathbf{x}]$.

Zunächst sei an dieser Stelle angemerkt, dass aus $\text{mid}([\mathbf{x}]) + \mathbf{y} \in [\mathbf{x}]^{i,+}$ stets folgt, dass $\text{mid}([\mathbf{x}]) - \mathbf{y} \in [\mathbf{x}]^{i,-}$ (die gegenüberliegende Facette) gilt. In der Praxis wird die Bedingung (2.55) nicht für die Funktion $\mathbf{f}$ selber geprüft, sondern für die präkonditionierte Funktion $\mathbf{g}(\mathbf{x}) := \mathbf{C} \cdot \mathbf{f}(\mathbf{x})$. Ist $\mathbf{C} \in \mathbb{R}^{n\times n}$ eine nicht-singuläre Matrix, impliziert die Bedingung (2.55) für $\mathbf{g}$ ebenfalls eine Nullstelle von $\mathbf{f}$. Nach [17] gilt Bedingung (2.55) für $\mathbf{g}$, falls

$$\bigcap_{j=1}^{n} \frac{[g_j]([\mathbf{x}]^{i,+})}{[g_j]([\mathbf{x}]^{i,-})} \cap (0, \infty) = \emptyset \tag{2.56}$$

für alle $i = 1, \ldots, n$ gilt. Dabei erlauben wir zur besseren Darstellung an dieser Stelle die Verwendung eines offenen Intervalles. In vielen Fällen kann der Test (2.56) die Existenz einer Nullstelle nicht nachweisen, obwohl die Bedingungen von Satz 2.3 erfüllt sind. Dies ist vor allem auf die Tatsache zurückzuführen, dass der Wertebereich der Funktion $\mathbf{g}$ stark überschätzt wird.

Um diese Überschätzung zu reduzieren, werden die Facetten $[\mathbf{x}]^{i,\pm}$ wiederum in *Teilfacetten* $[\mathbf{x}]^{i,\pm} = \bigcup_k [\mathbf{x}]^{i,\pm,k}$ so unterteilt, dass diese disjunkt bis auf ihre jeweiligen

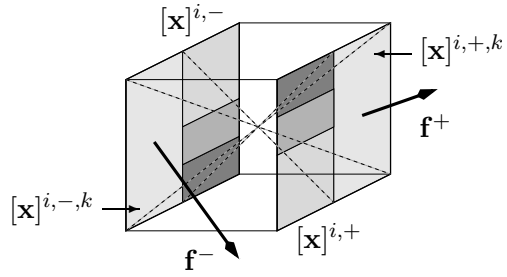

Abbildung 2.6: Aufteilung gegenüberliegender Facetten für den Borsuk-Test.

Ränder sind. Für den Borsuk-Test muss zusätzlich gelten, dass gegenüberliegende Teilfacetten symmetrisch zum Mittelpunkt der Box sind (siehe Abbildung 2.6). Folglich kann die Bedingung (2.56) durch die folgende Bedingung ersetzt werden: Die Box $[\mathbf{x}]$ enthält eine Lösung, falls

$$\bigcap_{j=1}^{n} \frac{[g_j]([\mathbf{x}]^{i,+,k})}{[g_j]([\mathbf{x}]^{i,-,k})} \cap (0,\infty) = \emptyset \tag{2.57}$$

für alle $i = 1, \ldots, n$ und alle k.

Wie beim Miranda-Test können auch beim Borsuk-Test die Präkonditionierer $\mathbf{C} = \mathbf{I}_{n\times n}$ und $\mathbf{C} \approx \mathbf{Df}(\check{\mathbf{x}})^{-1}$ eingesetzt werden. Beim Miranda-Test musste der *gleiche* Präkonditionierer für alle Teilfacetten benutzt werden. Dagegen entfällt beim Borsuk-Test diese unangenehme Einschränkung. In [18] stellen Frommer und Lang einen neuen Präkonditionierer vor, der für jedes gegenüberliegendes Paar von Facetten neu gewählt werden kann. Der jeweilige Präkonditionierer $\mathbf{C}^{i,k}$ wird dabei so gewählt, dass die Funktion $\mathbf{g} = \mathbf{C}^{i,k} \cdot \mathbf{f}$ möglichst in entgegengesetzte Richtungen auf $[\mathbf{x}]^{i,+,k}$ und $[\mathbf{x}]^{i,-,k}$ zeigt. Um den Präkonditionierer zu formulieren, definieren wir zunächst $\mathbf{f}^+ := \mathbf{f}(\mathrm{mid}([\mathbf{x}]^{i,+,k}))$ und $\mathbf{f}^- := \mathbf{f}(\mathrm{mid}([\mathbf{x}]^{i,-,k}))$. Der Präkonditionierer $\mathbf{C}^{i,k}$ wird nun so gewählt, dass $\mathbf{g}^+ = \mathbf{C}^{i,k} \cdot \mathbf{f}^+ = (100, 1, 0, \ldots, 0)^T$ und $\mathbf{g}^- = \mathbf{C}^{i,k} \cdot \mathbf{f}^- = (-100, 1, 0, \ldots, 0)^T$. Um den Präkonditionierer zu erhalten, werden zunächst zwei QR-Zerlegungen $(\mathbf{f}^+, \mathbf{f}^-) = \mathbf{Q_f} \cdot \mathbf{R_f}$ und $(\mathbf{g}^+, \mathbf{g}^-) = \mathbf{Q_g} \cdot \mathbf{R_g}$ durchgeführt. Anschließend setzt man

$$\mathbf{C}^{i,k} = (\mathbf{g}^+, \mathbf{g}^-, \mathbf{Q_g}(:, 3:n)) \cdot (\mathbf{f}^+, \mathbf{f}^-, \mathbf{Q_f}(:, 3:n))^{-1}. \tag{2.58}$$

Ist diese Matrix schlecht konditioniert oder sogar singulär, so zeigen entweder $\mathbf{f}^+$ und $\mathbf{f}^-$ bereits in beinahe entgegengesetzte Richung (in diesem Fall ist keine Präkonditionierung nötig) oder sie zeigen in die gleiche Richtung (dann kann Präkonditionierung nicht helfen).

2.6.3 Topologischer Abbildungsgrad

Ein weiterer Test, der die Existenz einer Nullstelle in einer Box bei quadratischen Systemen nachweist, basiert—wie der Satz von Borsuk—auf dem so genannten *topologischen Abbildungsgrad*. Sei $\Omega \subseteq \mathbb{R}^n$ eine offene und beschränkte Menge und $\mathbf{f} : \overline{\Omega} \to \mathbb{R}^n$ eine stetige Abbildung. Weiterhin gelte für alle $\mathbf{x} \in \partial\Omega$, dass $\mathbf{f}(\mathbf{x}) \neq \mathbf{0}$. Dann gibt uns der

topologische Abbildungsgrad $d(\mathbf{f}, \Omega, \mathbf{0})$ Aufschluss über die Anzahl der Lösungen $\mathbf{x} \in \Omega$ von $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. Eine genauere Einführung zum topologischen Abbildungsgrad, der stets einer ganzen Zahl entspricht, findet man z.B. in [13, 58]. Für die in diesem Abschnitt vorgestellten Verifikationstests benötigen wir nur den folgenden Satz.

Satz 2.4 (i) Ist $\mathbf{f}$ differenzierbar und die Ableitungsmatrix $\mathbf{Df}(\mathbf{x})$ nicht-singulär in allen Punkten $\mathbf{x} \in \Omega$ mit $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, dann gilt

$$d(\mathbf{f}, \Omega, \mathbf{0}) = \sum_{\mathbf{x} \in \Omega, \mathbf{f}(\mathbf{x}) = \mathbf{0}} \operatorname{sign}(\det \mathbf{Df}(\mathbf{x}))$$

(ii) Ist $\mathbf{h} : [0,1] \times \overline{\Omega} \to \mathbb{R}^n$ eine stetige Funktion, so dass $\mathbf{h}(t, \mathbf{x}) \neq \mathbf{0}$ für alle $t \in [0,1]$ und alle $\mathbf{x} \in \partial\Omega$, dann hängt $d(\mathbf{h}(t, \cdot), \Omega, \mathbf{0})$ nicht von t ab.

(iii) Ist $d(\mathbf{f}, \Omega, \mathbf{0}) \neq 0$, dann existiert ein $\mathbf{x} \in \Omega$, so dass $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

Wählt man $\Omega = \operatorname{int}([\mathbf{x}])$, also das Innere der Box $[\mathbf{x}]$, so gilt für einen Punkt $\check{\mathbf{x}} \in \operatorname{int}([\mathbf{x}])$ nach Satz 2.4 (i)

$$d(\mathbf{x} - \check{\mathbf{x}}, \operatorname{int}([\mathbf{x}]), \mathbf{0}) = 1. \tag{2.59}$$

Um nun die Existenz einer Nullstelle einer gegebenen stetigen Funktion $\mathbf{f}$ nachzuweisen, betrachten wir die Homotopie

$$\mathbf{h}(t, \mathbf{x}) = t \cdot \mathbf{g}(\mathbf{x}) + (1 - t) \cdot (\mathbf{x} - \check{\mathbf{x}}), \tag{2.60}$$

wobei $\mathbf{g}(\mathbf{x}) = \mathbf{C} \cdot \mathbf{f}(\mathbf{x})$ mit einer geeigneten, nicht-singulären Präkonditionierungsmatrix $\mathbf{C} \in \mathbb{R}^{n \times n}$. Besitzt $\mathbf{h}(t, \mathbf{x})$ in $[0,1] \times \partial[\mathbf{x}]$ keine Nullstelle, so gilt nach (2.59) und Satz 2.4 (ii), dass $d(\mathbf{g}, \operatorname{int}([\mathbf{x}]), \mathbf{0}) = 1$. Nach Satz 2.4 (iii) hat damit $\mathbf{g}$ eine Nullstelle $\mathbf{x}^*$ im Inneren von $[\mathbf{x}]$. Da zusätzlich gefordert wurde, dass $\mathbf{C}$ nicht-singulär ist, ist $\mathbf{x}^*$ ebenfalls eine Nullstelle von $\mathbf{f}$. Eine gute Wahl für den Präkonditionierer ist $\mathbf{C} \approx \mathbf{Df}(\check{\mathbf{x}})^{-1}$, vorausgesetzt, dass $\mathbf{f}$ differenzierbar und $\mathbf{C}$ gut konditioniert ist.

Wie bei den Tests von Miranda und Borsuk, kann der Suchbereich $[0,1] \times \partial[\mathbf{x}]$ in $2n$ Teilboxen unterteilt werden:

$$[0,1] \times \partial[\mathbf{x}] = \bigcup_{i=1}^{n} \bigcup_{s \in \{+,-\}} [0,1] \times [\mathbf{x}]^{i,s}.$$

Um die Existenz einer Nullstelle in $[\mathbf{x}]$ nachzuweisen, müssen folglich mindestens $2n$ Gleichungssysteme gelöst werden, da wir erneut eine weitere Unterteilung der einzelnen Facetten in Teilfacetten zulassen. Somit ergibt sich der in Alg. 2.10 skizzierte *Degree-Test*.

Abhängig davon, wie man Lösungen von $\mathbf{h}(t, \mathbf{x})$ in $[0,1] \times [\mathbf{y}]$ sucht (Zeile 8 von Alg. 2.10), ergeben sich offensichtlich unterschiedliche Existenz-Tests. Zunächst erscheint es unsinnig, die Lösung von $2n$ Gleichungssystemen einem einzigen Gleichungssystem vorzuziehen. Allerdings muss ja „nur“ gezeigt werden, dass die Gleichungssysteme *keine* Lösung haben. Aus diesem Grund genügt es oft, einzig und allein den direkten Auswertungstest für $\mathbf{h}$ auf der aktuellen Box anzuwenden. Den daraus resultierenden Test werden wir im Folgenden als DegreeTest bezeichnnen.

Algorithmus 2.10 DEGREE_TEST()

```
1: for i = 1, ..., n do
2:   for s = +, − do
3:     L_0 = {[0,1] × [x]^{i,s}}
4:     b = 1
5:     for l = 0, ..., bis L_l = ∅ do
6:       L_{l+1} = ∅
7:       for alle [t] × [y] ∈ L_l do
8:         if h(t, x) hat keine Lösung in [t] × [y] then
9:           {In diesem Fall ist nichts zu tun}
10:        else
11:          if l < l_max und b ≤ b_max − d then
12:            unterteile [t] × [y] in d Teilboxen und füge diese in L_{l+1} ein
13:            b = b + d
14:          else
15:            return false {weitere Unterteilung nicht erlaubt}
16:          end if
17:        end if
18:      end for
19:    end for
20:  end for
21: end for
22: return true {h(t, x) hat keine Nullstelle in [t] × [y]}
```

Alternativ kann offensichtlich auch ein verifizierender nichtlinearer Löser für die Nullstellensuche eingesetzt werden. Dieses hat den Vorteil, dass wiederum effiziente Techniken wie das Newton-Verfahren oder Constraint-Propagation zum Einsatz kommen können. Da bereits in Algorithmus 2.10 eine Unterteilung der Boxen vorgenommen wird, sollte es genügen, einen einzigen Aufruf von CHECKBOX für das Gleichungssystem durchzuführen.

Um die Effektivität des letzten Ansatzes zu zeigen, betrachten wir die sehr einfache Abbildung

$$\mathbf{f}(u,v) = \begin{pmatrix} 4 - 2(u-1)^2 \\ (2-(u+1)^2) \cdot (2-(v-1)^2) \end{pmatrix}, \tag{2.61}$$

welche vier Nullstellen $(1 \pm \sqrt{2}, 1 \pm \sqrt{2})$ im $\mathbb{R}^2$ hat. Wir haben nun den DegreeTest, den DegreeTest mit Anwendung von CHECKBOX und den Borsuk-Test auf verschiedene Boxen

$$[\mathbf{z}^{(k)}] = 2^{k/8} \cdot \begin{pmatrix} [-0.25, 0.25] \\ [-0.25, 0.25] \end{pmatrix},$$

für $k = 0, \ldots, 32$ angewendet. Diese Boxen wachsen mit einem konstanten Faktor $\sqrt[8]{2}$ von der kleinsten Box $[\mathbf{z}^{(0)}]$ zur größten Box $[\mathbf{z}^{(32)}]$ an. In Abbildung 2.7 haben wir die Ergebnisse zusammengefasst. Bei den Ergebnisse ist zu beachten, dass die Boxen $[\mathbf{z}^{(0)}]$ bis $[\mathbf{z}^{(5)}]$ keine Lösung enthalten. Die Boxen $[\mathbf{z}^{(27)}]$ bis $[\mathbf{z}^{(32)}]$ enthalten alle vier Lösun-

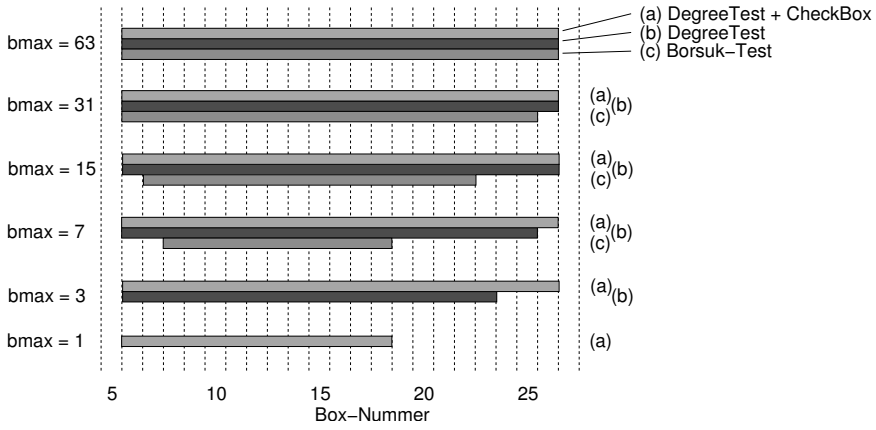

Abbildung 2.7: Nummern der Boxen, in denen mit den Existenztests (a) DegreeTest + CheckBox, (b) DegreeTest und (c) Borsuk-Test Nullstellen verifiziert werden können.

gen. Für diese Boxen kann der DegreeTest keine Lösung nachweisen, da der topologische Abbildungsgrad dort gerade ist.

Man sieht deutlich, dass alle Tests für alle (möglichen) Boxen Nullstellen verifizieren können, wenn man viele Unterteilungen zulässt. Da aber der Aufwand des Verifikationstests proportional zur Anzahl der betrachteten Teilboxen ist, sind wir natürlicherweise daran interessiert, möglichst wenige Unterteilungen zuzulassen. Man sieht an Abbildung 2.7 deutlich, dass nur der DegreeTest mit einem CheckBox-Aufruf für bemerkenswert viele Boxen $[\mathbf{z}^{(k)}]$ erfolgreich ist, bei denen die Unterteilung komplett abgeschaltet wurde.

Die präsentierten Ergebnisse legen den Schluss nahe, dass der DegreeTest immer stärker als der Borsuk-Test ist. Allerdings steht ein Beweis dieser Theorie noch aus. Eventuell könnte der DegreeTest sogar noch weiter verbessert werden, wenn man eine andere Homotopie als (2.60) verwendet. Eine genauere Untersuchung der beiden zuletzt genannten Punkte ist in näherer Zukunft geplant.

2.6.4 Implementierungsdetails

Für die Implementierung ist zunächst zu beachten, dass der zu wählende Präkonditionierer im Miranda-Test für *alle* i gleich sein muss, wogegen der Präkonditionierer im Borsuk-Test zeilenweise gewählt werden kann. Bei genauerer Betrachtung sieht man ebenfalls, dass die drei präsentierten Verifikationstests alle *zeilenweise* durchgeführt werden können. Diese Eigenschaft erlaubt es uns, eine Hierarchie von Verifikationstests festzulegen, d.h., für jedes i wird in SONIC zunächst der Miranda-Test, bei Fehlschlagen der Borsuk-Test und zuletzt der Degree-Test angewendet. Das Vorgehen haben wir gewählt, da der Aufwand, der für die Durchführung der Tests benötigt wird, aufsteigend in der gewählten Reihenfolge ist (wir beginnen also mit dem „billigsten“ Test). Es sei an dieser Stelle erwähnt, dass die Qualität der Verifikationstests ebenfalls zunimmt. Bei gleichem Präkonditionierer ist der Borsuk-Test besser als der Miranda-Test [1]. Es

wird vermutet, dass der Degree-Test seinerseits besser als der Borsuk-Test ist. Diese Vermutung konnte bislang aber nicht bewiesen werden.

2.6.5 Weitere Eindeutigkeitsaussagen

In der Praxis kommt es vor, dass nur die Existenz einer Nullstelle in einer Box nachgewiesen werden kann, obwohl diese sogar eindeutig ist. In diesem Abschnitt werden wir Möglichkeiten vorstellen, wie man aus der Existenz auf die Eindeutig schließen kann, sofern dies überhaupt möglich ist. Grundlage dieser Techniken ist das folgende Lemma.

Lemma 2.4 Sei $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. Die Funktion $\mathbf{f}$ sei auf der Box $[\mathbf{x}] \in \mathbb{IR}^n$ stetig differenzierbar. Ist die Intervall-Matrix $[\mathbf{Df}]([\mathbf{x}])$ regulär, dann ist jede Nullstelle von $\mathbf{f}$ in $[\mathbf{x}]$ eindeutig.

Beweis: Angenommen, $\mathbf{x} \in [\mathbf{x}]$ und $\tilde{\mathbf{x}} \in [\mathbf{x}]$ sind zwei Nullstellen von $\mathbf{f}$. Dann existiert eine Matrix $\mathbf{S} \in [\mathbf{Df}]([\mathbf{x}])$ mit

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(\tilde{\mathbf{x}}) + \mathbf{S} \cdot (\mathbf{x} - \tilde{\mathbf{x}}).$$

Daraus folgt unmittelbar

$$\mathbf{0} = \mathbf{f}(\mathbf{x}) - \mathbf{f}(\tilde{\mathbf{x}}) = \mathbf{S} \cdot (\mathbf{x} - \tilde{\mathbf{x}}).$$

Da $\mathbf{S}$ regulär ist, muss offensichtlich $\mathbf{x} - \tilde{\mathbf{x}} = \mathbf{0}$ gelten. Folglich ist eine Nullstelle von $\mathbf{f}$ in $[\mathbf{x}]$ stets eindeutig.

Hat man also einmal die Existenz einer Nullstelle mit eine der Methoden aus den vorigen Abschnitten nachgewiesen, so genügt es zu zeigen, dass die Jacobi-Matrix regulär ist, um die Eindeutigkeit zu folgern. Offensichtlich gibt es mehrere Möglichkeiten die Regularität der Jacobi-Matrix zu zeigen. Die nach Erfahrung des Autors geschickteste Methode ist der Nachweis der Ungleichung

$$|| \mathbf{I} - \mathbf{C} \cdot [\mathbf{Df}]([\mathbf{x}]) ||_\infty < 1 \ , \tag{2.62}$$

wobei $\mathbf{C} := (\mathrm{mid}([\mathbf{Df}]([\mathbf{x}])))^{-1}$ gewählt werden sollte.

2.6.6 Verifikation bei nicht-quadratischen Systemen

Die präsentierten Verifikationstests sind ausschließlich für den Einsatz bei quadratischen Systemen geeignet. In diesem Abschnitt werden wir auf Methoden der Verifikation bei nicht-quadratischen Systemen eingehen.

2.6.6.1 Unterbestimmte Systeme

Sei $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ eine Funktion mit $m < n$ und $[\mathbf{x}] \in \mathbb{IR}^n$ eine Suchbox. Funktionen dieses Typs werden in der Regel als *unterbestimmtes System* bezeichnet. Hansen

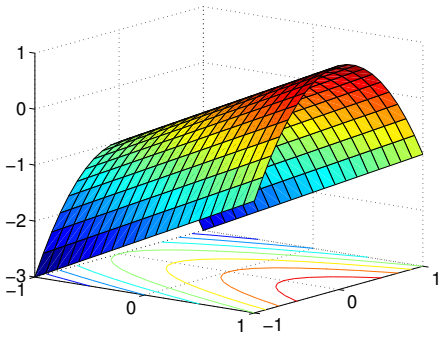

Abbildung 2.8: Plot der Funktion $\mathbf{f}(\mathbf{x}) := x_1 - 2x_2^2$ auf der Box $[\mathbf{x}] = [-1, 1]$.

[23] schlägt in diesem Fall vor, $(n - m)$ der Variablen zu fixieren. Betrachten wir zum Beispiel ein System mit $n = 2$ (2 Variablen) und $m = 1$ (1 Gleichung), so dass die Lösungsmenge von $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ einer Kurve im $\mathbb{R}^2$ entspricht. Eine der oben vorgestellten Methoden könnte nun die Existenz einer Nullstelle entlang einer Geraden parallel zu einer Koordinatenrichtung nachweisen. Allerdings muss man beachten, dass der Nachweis einer eindeutigen Lösung des erzeugten quadratischen Systems *nicht* die Eindeutigkeit beim ursprünglichen System impliziert. Zur Verdeutlichung betrachten wir das Gleichungssystem $\mathbf{f}(\mathbf{x}) := x_1 - 2x_2^2$ auf der Box $[\mathbf{x}] = [-1, 1]^2$, das in Abb 2.8 abgebildet ist. Die Lösungsmenge ist offensichtlich eine Parabel. Fixiert man die Variable x_2 auf $\text{mid}([x_2]) = 0$, so sucht man eine Nullstelle auf einer Geraden. Mit Hilfe des Newton-Verfahrens kann hier die Eindeutigkeit nachgewiesen werden. Für das ursprüngliche System ist damit aber lediglich die Existenz einer Nullstelle nachgewiesen.

Die Grundidee der existieren Methoden ist, genau die Koordinatenrichtungen festzuhaltenden, in denen das System am wenigsten sensitiv ist. Hansen führt deshalb Gauß-Eliminationsschritte aus und fixiert die Variablen, die in die letzten $(n - m)$ Zeilen permutiert wurden. Einen ähnlichen Ansatz verfolgen wir in SONIC. Hier wird allerdings eine Rang-anzeigende QR-Zerlegung verwendet, die auf Golub und Van Loan [21] zurückgeht. Anschließend werden auch bei diesem Ansatz die Variablen festgehalten die durch Permutation in die letzten $(n - m)$ Zeilen gelangt sind.

Eine weitere Möglichkeit der Verifikation besteht darin, dass man nicht einzelne Koordinaten in ihrem Mittelpunkt festhält, sondern das gesamte Intervall als Konstante in die Systemgleichungen einsetzt. Gelingt die Verifikation, so ist sogar die Existenz einer Schar von Lösungen nachgewiesen. Bei genauerer Betrachtung sieht man, dass dies bereits in der ursprünglichen Version der Fall war, da wir in der Implementierung eine *Einschließung* des jeweiligen Mittelpunktes verwenden müssen.

2.6.6.2 Überbestimmte Systeme

Im überbestimmten Fall, also falls mehr Gleichungen als Variablen existieren, kann nach der ursprünglichen Definition keine Verifikation mehr durchgeführt werden. Daher weichen wir für überbestimmte Systeme den Verifikationsbegriff ein wenig auf.

Zunächst wählen wir—wie im unterbestimmten Fall—mit Hilfe einer Rang-anzeigenden QR-Zerlegung n Gleichungen aus und erhalten ein quadratisches System. Für dieses System führen wir einen beliebigen Verifikationstest durch. Ist dieser erfolgreich, so haben wir folgendes „nachgewiesen“:

1. Für die ersten n Gleichungen (man beachte, dass man möglicherweise bereits eine Permutation der Gleichungen durchgeführt hat) existiert in der aktuell betrachteten Box eine, eventuell eindeutige, Lösung.
2. Für die übrigen Gleichungen können durch Anwendung einer Intervallerweiterung nur *Fehlerschranken* angegeben werden.

Auch wenn die Verifikation einer Nullstelle auf diese Weise nicht gelingt, so erhält der Nutzer von SONIC zumindest einen starken Hinweis auf die Existenz einer Nullstelle.

2.7 Exlusion-Regions

Wie von Kearfott und Du [40] im Kontext der globalen Optimierung erläutert wurde, leiden viele Branch–and–Bound-Algorithmen am so genannten *Cluster-Effekt*. Unter diesem versteht man das extreme Unterteilen einer Box in der Nähe einer Lösung bzw. das Fehlschlagen des direkten Auswertungstests für sehr viele Boxen, die keine Lösung enthalten. Die mathematischen Gründe für den Cluster-Effekt und dessen „Behandlung“ werden ausführlich in [61] beschrieben.

2.7.1 Bestimmung von Exlusion-Regions

Eine Möglichkeit dem Cluster-Effekt entgegenzuwirken, bieten die so genannten *Exclusion-Regions*. Sei $\mathbf{x}^*$ eine Lösung des nichtlinearen Gleichungssystems (2.1). Gesucht ist eine Exclusion-Region $\mathrm{ER}(\mathbf{x}^*)$ um $\mathbf{x}^*$ mit der Eigenschaft, dass keine weitere Lösung des Gleichungssystems in $\mathrm{ER}(\mathbf{x}^*)$ liegt. Diese Exclusion-Region muss offensichtlich nicht weiter in einem Branch–and–Bound-Löser nach Nullstellen untersucht werden, da die einzig vorhandene Nullstelle bereits gefunden wurde.

Hat man bereits eine *eindeutige* Lösung $[\mathbf{x}^*]$ des Gleichungssystems gefunden, so kann eine Exclusion-Region durch eine Kombination aus ε-Inflation und Verifikation berechnet werden. Zunächst wird die berechnete Lösung in jeder Komponente $[x_i]$ für $1 \leq i \leq n$ durch

$$[x_i'] := (1 + \varepsilon) \cdot [x_i] - \varepsilon \cdot [x_i] \tag{2.63}$$

vergrößert. Der Betrag, um den $[x_i]$ dabei vergrößert wird, hängt also vom Durchmesser von $[x_i]$ und ε ab. Anschließend wird mit Hilfe der Verifikationstechniken aus Abschnitt 2.6 versucht, nachzuweisen, dass die neue Box $[\mathbf{x}']$ ebenfalls eine eindeutige Lösung enthält. Dieses Verfahren wird solange wiederholt, bis die Eindeutigkeit der Lösung in $[\mathbf{x}']$ nicht mehr gezeigt werden kann. Der daraus resultierende Algorithmus ist in Alg. 2.11 skizziert. Es ist zu beachten, dass eine Lösung immer nur als Box, die die tatsächliche Lösung enthält, vorliegt. Damit ist $[\mathbf{x}^*]$ selber bereits eine Exclusion-Region, die

allerdings nur von begrenztem Nutzen ist, da diese zu allen anderen Boxen disjunkt ist.

Algorithmus 2.11 EXCLUSION-REGION($[\mathbf{x}^*], \mathcal{E}$)

```
1: eindeutig = true
2: ER(x*) = [x'] = [x*]
3: while eindeutig = true do
4:    bestimme [x'] über ε-Inflation aus Gleichung (2.63)
5:    führe Verifikation für [x'] aus
6:    if [x'] enthält eindeutige Lösung then
7:       ER(x*) = [x']
8:    else
9:       eindeutig = false
10:   end if
11: end while
12: füge ER(x*) in E ein
```

Die Methode wurde in dieser Form zunächst in SONIC integriert und mit Hilfe einer Vielzahl von Testbeispielen analysiert. Leider stellt sich heraus, dass bei fast jedem untersuchten System auf diese Weise nur extrem kleine Exclusion-Regions erzeugt werden können. Dies hat zur Folge, dass beinahe jede im Algorithmus betrachtete Box disjunkt zu den berechneten Exclusion-Regions ist. In den meisten Fällen ist es aus diesem Grund nicht möglich gewesen, Boxen mit dieser Technik komplett zu verwerfen. Der zusätzliche Mehraufwand, der durch dieses Verfahren entsteht, hat sogar die Performance von SONIC verschlechtert.

Neumaier stellt in [61, Satz 4.3] eine weitere Methode vor, wie man Exclusion-Regions berechnen kann. Diese Methode nutzt Informationen über die Steigungsmatrix zweiter Ordnung aus. Um die Resultate mit denen unserer Implementierung zu vergleichen, wurde dieses Verfahren testweise in Matlab implementiert. Es kann festgestellt werden, dass das Verfahren von Neumaier in der Regel bessere—also größere—Exclusion-Regions berechnet. Allerdings kann kein deutlicher Qualitätssprung verzeichnet werden. Betrachten wir dazu das durchaus repräsentative Gleichungssystem

$$x_1^i + x_2^i + x_3^i + x_4^i + x_5^i + x_6^i = \sum_{k=1}^{6} k^i$$

für $i = 1, \ldots, 6$, welches offensichtlich $\mathbf{x} = (1, 2, 3, 4, 5, 6)^T$ als Lösung besitzt. Die Berechnung der Exclusion-Regions mit Hilfe des Verfahrens von Neumaier und der Variante aus SONIC ergeben

$$\begin{pmatrix} [0.9850, 1.0145] \\ [1.9914, 2.0021] \\ [2.9871, 3.0129] \\ [3.9886, 4.0125] \\ [4.9925, 5.0199] \\ [5.9860, 6.0112] \end{pmatrix} \text{ bzw. } \begin{pmatrix} [0.98016, 1.01987] \\ [1.99210, 2.00011] \\ [2.99945, 3.00009] \\ [3.99910, 4.00019] \\ [4.99998, 5.00065] \\ [5.99769, 6.00015] \end{pmatrix}.$$

Man sieht deutlich, dass die Neumaier-Variante größere Exclusion-Regions liefert: Der Durchmesser der einzelnen Komponenten liegt hier durchschnittlich bei 10^{-2}, während die SONIC-Variante in der Regel den Durchmesser 10^{-3} liefert. Setzt man nun die mit Hilfe der Neumaier-Technik berechnete Exclusion-Region „per Hand“ in SONIC ein (und zwar bereits zu Beginn der Berechnung), so muss ebenfalls ernüchternd festgestellt werden, dass nur *eine einzige* Box zusätzlich verworfen werden kann. Ähnliche Erfahrungen hat der Autor auch mit weiteren Systemen gemacht: Bei *schweren* Testproblemen liefern beiden Techniken nur sehr kleine Exclusion-Regions, wobei die Neumaier-Variante in der Regel um einen Faktor 10 im (maximalen) Durchmesser besser ist.

Neumaier präsentiert in seinem einführenden Paper [61] einige Testbeispiele, bei denen seine neue Technik wesentlich größere—manchmal sogar optimale—Exclusion-Regions liefert. Allerdings muss an dieser Stelle festgehalten werden, dass diese Beispiele nur sehr leichte Testsysteme mit quadratischen Gleichungen darstellen. Für jede in [61] präsentierte Exclusion-Region konnte der Autor zeigen, dass *eine einzige* Anwendung von CheckBox genügt, um die entsprechende Exclusion-Region $\mathrm{ER}(\mathbf{x}^*)$ auf einen *Punkt* zu kontrahieren. Damit ist zu erwarten, dass der Einsatz von Exclusion-Regions im Gleichungslöser auch für einfachere Testsysteme keine Vorteile bringt, da jede Box, die ganz in $\mathrm{ER}(\mathbf{x}^*)$ liegt, genauso stark kontrahiert wird.

Abschließend sei erwähnt, dass die ε-Inflation in einigen Fällen auch bei einer ursprünglichen Verifikationstechnik eingesetzt wird. Hat nämlich eine Box einen extrem kleinen Durchmesser, so ist aufgrund der Rechengenauigkeit nicht zu erwarten, dass man eine Box mit dem Newton-Verfahren so weit verkleinern kann, dass diese im Innern der Box liegt.

2.7.2 Pruning mit Hilfe von Exclusion-Regions

Bei genauerer Betrachtung sieht man, dass man Exclusion-Regions vom reinen Wegwerf-Test zu einem „echten“ zulässigen Kontraktionsverfahren erweitern kann. Liegt die aktuelle Suchbox zu einem Teil in einer Exclusion-Region, so braucht dieser Teil nicht weiter betrachtet zu werden. Zur Veranschaulichung betrachten wir Abbildung 2.9 für den 2D- und den 3D-Fall.

Liegt die aktuelle Box $[\mathbf{x}]$ *bis auf eine Komponente* i_0 in einer Exclusion-Region $\mathrm{ER}(\mathbf{x}^*)$, so kann gewährleistet werden, dass der Bereich, der nach der Kontraktion noch betrachtet werden muss, durch *maximal zwei* Boxen darstellbar ist. Dabei ist zu beachten, dass die Boxkomponente $[x_{i_0}]$ mit der entsprechenden Komponente der Exclusion-Region einen nicht-leeren Schnitt haben muss, da anderenfalls *kein* Teil von $[\mathbf{x}]$ komplett in $\mathrm{ER}(\mathbf{x}^*)$ fällt. Dann wird über

$$[a] \cup [b] := [x_{i_0}] \setminus (\mathrm{ER}(\mathbf{x}^*))_{i_0}$$

der Bereich aus $[x_{i_0}]$ herausgeschnitten, der garantiert keine weitere Nullstelle mehr enthalten kann. Anschließend werden zwei Kopien $[\mathbf{x}^{(1)}]$ und $[\mathbf{x}^{(2)}]$ der Box $[\mathbf{x}]$ erzeugt und deren Komponente i_0 durch $[a]$ bzw. $[b]$ ersetzt (ist eines der Intervalle $[a]$ oder $[b]$ leer, so wird natürlich nur eine neue Box erzeugt). Diese Methode bezeichnen wir im Folgenden als *Pruning mit Exclusion-Regions*.

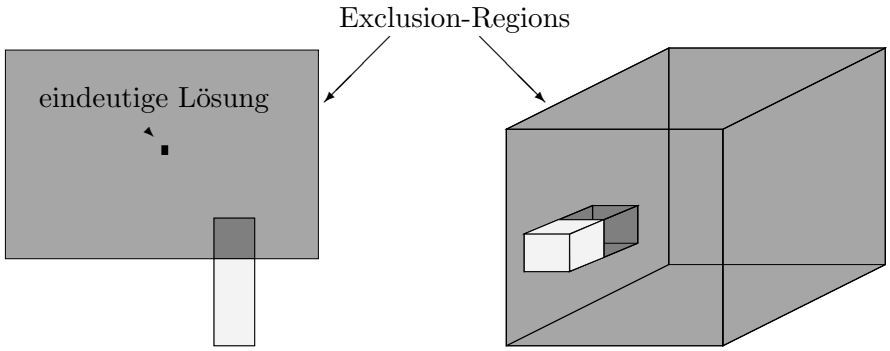

Abbildung 2.9: Mögliches Pruning mit Hilfe von Exclusion-Regions für 2D- (links) und 3D-Fall (rechts). Im weiteren Verlauf müssen nur noch die helleren Bereiche betrachtet werden, die dunklen Bereiche können verworfen werden.

Aufgrund der Tatsache, dass wir momentan nur in der Lage sind, sehr kleine Exclusion-Regions zu berechnen, hat sich bei ersten Untersuchungen herausgestellt, dass diese Technik keine wesentlichen Verbesserungen in der Performance bringt. Allerdings konnte festgestellt werden, dass bei vielen Testsystemen eine Reihe von Boxen durch dieses Verfahren deutlich kontrahiert werden konnten. Eine Methode, die deutlich größere Exclusion-Regions berechnet, als die momentan bekannten Varianten, dürfte das Pruning mit Exclusion-Regions in der Zukunft zu einem starken, zulässigen Kontraktionsverfahren machen. Da nach Meinung des Autors in dieser Technik ein großes Potential verborgen liegt, sind für die nähere Zukunft weitere Untersuchungen geplant. Dabei sollte auch untersucht werden, ob Exclusion-Regions definiert werden können, die definitiv *keine* Lösung des Gleichungssystems enthalten.

Treten für verschiedene Systeme sehr viele Exclusion-Regions auf, so könnte eine spezielle Verwaltung von Exclusion-Regions sinnvoll sein. Anstatt jede Box sequentiell gegen alle Exclusion-Regions zu testen, könnten zum Beispiel geschickte Verfahren aus der algorithmischen Geometrie, wie zum Beispiel die Intervall-Bäume [12] oder der Range-Tree-Ansatz, eingesetzt werden.

2.8 Das serielle Verfahren

Im Folgenden wird das serielle Verfahren zur verifizierten Lösung nichtlinearer Gleichungssysteme beschrieben, das auf den präsentierten Verfahren aus den vorigen Abschnitten beruht. Wie zu Beginn haben wir den resultierenden Algorithmus in zwei grobe Teile gegliedert: Die Routine CheckBox, die die individuelle Behandlung einer Box abwickelt und den verwendeten Branch–and–Bound-Ansatz. Wir haben diese Unterteilung beibehalten, da die Rountine CheckBox ebenfalls noch im Kapitel 4 benötigt wird. Die Beschreibung der beiden Algorithmen soll an dieser Stelle nur sehr kurz erfolgen, da wir bei der Herleitung bereits ausführlich darauf eingegangen sind, warum bestimmte Verfahren gewählt wurden, oder eben nicht. Im hierarchischen Ansatz wird standardmäßig das Taylor-Verfahren erster Ordnung zur Auflösung der FBP-

Algorithmus 2.12 CheckBox($[\mathbf{x}], \mathcal{L}, \mathcal{R}$, `gesplittet`)

1: Hierarchischer Ansatz($[\mathbf{x}], \mathcal{L}, \mathcal{R}$, `gesplittet`)
2: führe optional Taylor-Verfahren erster oder zweiter Ordnung durch

Constraints (Forward-Backward-Propagation) und Constraint-Propagation durchgeführt, die beide optional abschaltbar sind. Die Auflösung der FBP-Constraints kann zusätzlich durch das Taylor-Verfahren zweiter Ordnung erfolgen, was im Standardverfahren allerdings nicht gemacht wird. In Zeile 2 von Algorithmus 2.12 werden bei Verwendung der Standardeinstellungen keine zusätzlichen Kontraktionsverfahren benutzt. In einigen Beispielen ist es durchaus möglich, dass die Hinzunahme weiterer Verfahren an dieser Stelle leichte Verbesserungen bringen.

Der Branch–and–Bound-Ansatz folgt in großen Teilen der zuletzt präsentierten Variante aus Algorithmus 2.9. Wir haben zusätzlich einen Zähler integriert, der Statistik über die Anzahl der durchgeführten Iterationen führt. Erreicht dieser Zähler den Wert `MaxIterations`, so wird die Berechnung unmittelbar abgebrochen. In Zeile 6 des seriellen Algorithmus werden nach einer gewissen Anzahl von Boxen wichtige Informationen über den aktuellen Status der Berechnung auf dem Bildschirm ausgegeben. Ebenfalls können alle Daten—gesteuert über einen weiteren Zähler—an dieser Stelle in eine Datei gespeichert werden, um die Berechnung zu einem späteren Zeitpunkt fortzusetzen. Alle integrierten Verfahren, wie zum Beispiel die Splitting-Bisektion, können auch hier optional aus- bzw. eingeschaltet werden. Aus Gründen der Übersichtlichkeit haben wir hier nur den Algorithmus, der sich durch die Wahl der Standardeinstellungen ergibt, präsentiert.

2.9 Numerische Ergebnisse

Die Zeit zur Lösung von nichtlinearen Gleichungssystemen und Optimierungsproblemen wird sehr oft in der Einheit *Standard Time Unit* (STU) angegeben, um eine gewisse Unabhängigkeit vom verwendeten Rechner und Compiler zu erhalten. Eine *Standard Time Unit* entspricht der Berechnungszeit für 1000 reelle Auswertungen der Shekel-5-Funktion. Der Nachteil dieser Methode ist, dass ausschließlich *reelle* Auswertungen berücksichtigt werden. Allerdings spielen bei verifizierenden Algorithmen unterschiedlich schnelle Intervall-Bibliotheken eine große Rolle (bei den in SONIC verwendeten Intervall-Bibliotheken können die Laufzeiten um einen Faktor 10 variieren). Diese werden bei der STU aber *nicht* berücksichtigt. Aus diesem Grund verzichten wir an dieser Stelle auf die Angabe der gemessenen Zeiten in STU.

Die numerischen Experimente wurden alle mit den *Standard-Einstellungen* von SONIC durchgeführt. Diese wurden so gewählt, dass sie in der Regel für alle Systeme gute Ergebnisse liefern, da ein potenzieller Nutzer von SONIC ohne umfangreiche Veränderungen am Löser eine gute Performance des Tools erwartet. In einigen Fällen wäre es durchaus möglich gewesen, durch die gezielte Wahl bestimmter Einstellungen bessere Zeiten zu erzielen. Um eine gewisse Unabhängigkeit von der Last auf den verschiedenen Rechnersystemen zu bekommen, haben wir alle Messungen mindestens drei

Algorithmus 2.13 DER SERIELLE ALGORITHMUS

Beschreibung: Gegeben sei eine Startbox $[\mathbf{x}^{(0)}]$. Die Liste $\mathcal{L}$ verwaltet die noch zu betrachtenden Boxen. In $\mathcal{R}$ werden die potentiellen Lösungen gespeichert.

```
1:  L = [x^(0)]
2:  R = ∅
3:  Iterations = 0
4:  while L ≠ ∅ and Iterations < MaxIterations do
5:    entferne die erste Box [x] aus L
6:    gebe optional Informationen aus oder speichere notwendige Daten in einer Datei
7:    Iterations = Iterations + 1
8:    HIERARCHISCHER ANSATZ([x], L, R, gesplittet)
9:    if [x] konnte nicht verworfen werden then
10:     if gesplittet = false then
11:       bestimme Komponente i, in der das Bild möglichst exzentrisch zur Null liegt
12:       NEWTON-GAUSS-SEIDEL([x], S^M-LP-Präkonditionierer, L, R, gesplittet, i)
13:     end if
14:     if gesplittet = false then
15:       führe HYBRIDE BISEKTIONSSTRATEGIE durch
16:     end if
17:   end if
18:   if [x] ist klein genug then
19:     füge [x] in Liste der potentiellen Lösungen R ein
20:   else
21:     füge [x] geeignet in die Arbeitsliste L ein
22:   end if
23:   if TerminationOnFirstHit and es konnte eine Lösung verifiziert werden then
24:     exit
25:   end if
26: end while
27: führe für alle Boxen in R die Verifikation durch
```

mal durchgeführt und den Mittelwert der Ergebnisse gebildet. Es sei an dieser Stelle angemerkt, dass sich schlimmstenfalls Abweichungen im Bereich von Zehntel-Sekunden ergeben haben.

Wir wollen SONIC an dieser Stelle detailliert mit dem hervorragenden Tool GlobSol vergleichen. GlobSol [19] enthält ebenfalls einen verifizierenden nichtlinearen Gleichungslöser und wird momentan als eines der besten Pakete im Bereich des ergebnisverifizierendes Rechnens erachtet. Auch die numerischen Experimente von GlobSol haben wir jeweils mit den *Standard-Einstellungen* durchgeführt. Auch bei diesem Tool wären sicherlich durch veränderte Einstellungen in manchen Fällen bessere Ergebnisse zu erzielen gewesen. Da GlobSol ausschließlich den Einsatz auf einer Sun-Maschine erlaubt, haben wir ingesamt drei verschiedene Testreihen durchgeführt. Zunächst haben wir SONIC und GlobSol für eine Vielzahl von Testbeispielen auf demselben Sun Fire-Server mit 8 Prozessoren (1.2 MHz) und insgesamt 36 GB Speicher durchgeführt. Dabei haben

wir für SONIC die Intervallbibliothek von Sun eingesetzt. Desweiteren haben wir diese Systeme auch auf einem weiteren Rechner mit einem Pentium IV M Prozessor mit 1.7 GHz und 1 GB Hauptspeicher (mit der Intervallbibliothek filib++) durchgeführt. Die resultierenden Ergebnisse haben wir in Tabelle 2.7 zusammengefasst.

An den Ergebnissen ist zunächst auffällig, dass GlobSol für eine größere Anzahl von Testbeispielen keine Ergebnisse berechnen konnte. Dieses Verhalten kann grundsätzlich auf zwei Gründe zurückgeführt werden. Ein Teil der Systeme, die aus der Prozesstechnik kommen, sind nicht quadratisch. Bei diesen Testsystemen bricht GlobSol die Berechnung unmittelbar mit der Fehlermeldung „`Coordinate 1 is degenerate`“ ab (die entsprechenden Systeme sind in Tabelle 2.7 mit „—“ gekennzeichnet). Anschließend wird jeweils die erste Boxkomponente ausgegeben, deren Durchmesser in allen Fällen größer Null ist. Diese Aussage ist zumindest für den Autor sehr verwirrend, da sie wohl der eigentlichen Definition einer degenerierten Boxkomponente widerspricht. Dieses Verhalten konnte auch mit Hilfe von R. Baker Kearfott leider nicht aufgeklärt werden. Der zweite Grund liegt in der Behandlung von Auswertefehlern. Wie bereits in diesem Kapitel erwähnt wurde, treten in einigen Fällen viele Auswertefehler durch Überschätzungen auf, obwohl die Funktion in der kompletten Box total ist. Ein Abbrechen des Lösers ist in diesem Fall also völlig unnötig, da diese Problematik durch Bisektion oder Anwendung von CP gelöst werden kann. GlobSol dagegen unterbricht in diesem Fall die Berechnung und gibt „`arithmetic exception`“ aus (die entsprechenden Systeme haben wir mit † gekennzeichnet). Damit ist die Berechnung beinahe aller Systeme, die aus dem Bereich der Prozesstechnik kommen, mit Hilfe von GlobSol nicht möglich. Die einzige Ausnahme bildet hier das CSTR, welches wir in der Motivation vorgestellt haben. Bei einigen Systemen haben wir die Ergebnisse von GlobSol mit einem Stern gekennzeichnet. In diesem Fall war die Berechnung des Beispiels über ein nichtlineares Gleichungssystem nicht erfolgreich. Auf Anraten von Kearfott hat der Autor diese Systeme in Optimierungsprobleme umgeschrieben (konstante Zielfunktion) und diese anschließend mit dem in GlobSol integrierten Optimierer gelöst. Diese Methode liefert offensichtlich nicht die besten Resultate, da sich durch die Berechnung eines Optimierungsproblems ein deutlicher Overhead ergibt. Allerdings war es die einzige Möglichkeit, die Testsysteme mit GlobSol zu berechnen.

Bei den präsentierten Testsystemen sieht man deutlich, dass SONIC—sowohl in der Anzahl der betrachteten Boxen als auch in der Laufzeit—GlobSol deutlich überlegen ist. Die erzielten Ergebnisse werden in der Qualität sogar noch gesteigert, wenn wir die Plattform wechseln, was mit GlobSol nicht möglich ist. In diesem Fall kommt uns, neben der schnelleren Intervallbibliothek, natürlich auch die leicht erhöhte Prozessorleistung (Faktor 1.4) zu Gute. Neben den hier präsentierten Beispielen hat der Autor weitere umfangreiche Untersuchungen durchgeführt. In *allen* Beispielen konnte gezeigt werden, dass SONIC bei Verwendung der Standard-Einstellungen dem Löser von GlobSol überlegen ist.

Einige, in diesem Abschnitt präsentierte, Testsysteme stammen von der Homepage von J.-P. Merlet [47]. Für diese Systeme hat Merlet mit seinem verifizierten Löser ALIAS ebenfalls numerische Experimente auf einem Rechner mit 1.7 GHz durchgeführt (weitere Details sind dem Autor leider nicht bekannt), deren Resultate wir ebenfalls in

Tabelle 2.7 dargestellt haben. Dabei ist zu beachten, dass ein leerer Eintrag in der Spalte bedeutet, dass entsprechende Resultate nicht bekannt sind. Leider sind dem Autor nur die gemessenen Zeiten und nicht die Anzahl der betrachteten Boxen bekannt. Für jedes durchgeführte Experiment wurden *unterschiedliche* und auf das spezielle System *zugeschnittene* Einstellungen verwendet. Die dabei erzielten Resultate konnten nur in einem Ausnahmefall vom Autor mit den Standard-Einstellungen reproduziert werden. Ein *fairer* Vergleich der beiden Tools ist deshalb leider nicht möglich. Wie man sieht, liefert SONIC aber auch bei diesen Testbeispielen bessere Ergebnisse.

System	m	n	M	N	SONIC		SONIC (Sun)		GlobSol (Sun)		ALIAS
					Boxen	Zeit [s]	Boxen	Zeit [s]	Boxen	Zeit [s]	Zeit [s]
CSTR_Cusp	9	9	69	69	17	1.86	17	1.71	382	15.75	—
CSTR_Cusp_Large	9	9	69	69	997	11.12	999	23.82	3 359	86.81	—
7erSystem	7	7	430	430	4 288	102.11	4 288	276.90	26 330	767.05	43 203.0
Agrawal_Hopf	12	13	82	82	5 812	106.32	5 812	288.13	†	†	—
Agrawal_SaddleNode	5	6	52	52	9	2.55	9	3.61	†	†	—
Brent	8	8	70	70	86 189	146.68	86 190	397.51	—	—	1 902.6
Combustion	5	5	24	24	314	5.95	314	14.45	*1 493	32.85	
DirectKinematics	11	11	184	184	23 441	299.50	23 441	810.29	†	†	2 880.00
DesignProblem	9	9	91	91	39 639	122.19	38 725	1118.76	—	—	398.0
G7	7	7	79	79	1 919	10.22	1 920	27.84	*48 170	9450.59	
Eco9	8	8	65	65	54 001	162.48	54 001	342.76	—	—	208.0
HumanHeartDipole	8	8	67	67	36 649	388.31	36 649	896.99	—	—	
Reactor	29	29	171	171	344	22.10	344	59.89	22 478	1026.70	
Robotics	8	8	48	48	31	0.89	32	2.42	259	12.69	
Teymour_SaddleNode	10	11	458	458	267	74.61	267	83.19	†	†	—
min-03-07	19	11	179	179	65 198 359	102 460.00			—	—	

Tabelle 2.7: Vergleich von SONIC, GlobSol und (wenn möglich) ALIAS. Neben der Anzahl der Gleichungen und Variablen wird für jedes System ebenfalls die Größe des vollen Splits angegeben. Bei SONIC und GlobSol wurden jeweils die Standardeinstellungen verwendet, während bei ALIAS jeweils spezielle Einstellungen vorgenommen wurden. Die numerischen Ergebnisse in den Spalten „SONIC (Sun)“ und „GlobSol (Sun)“ wurden auf demselben Rechner durchgeführt.

2.10 Ein Anwendungsbeispiel

Wie wir bereits mehrmals erwähnt haben, wird durch die Verwendung ergebnisverifizierender Algorithmen *garantiert*, dass die berechneten Ergebnisse mathematisch korrekt sind. Aus diesem Grund führt SONIC stets einen mathematischen Beweis durch. Interessanterweise kann diese Eigenschaft auf einigen Gebieten ausgenutzt werden, bei denen man dies zunächst gar nicht vermuten würde. Wir wenden uns in diesem Abschnitt einem Beweis durch SONIC in der Algebra zu. Dieser Beweis ist momentan mit analytischen Methoden *nicht* zu führen.

Definition 2.9 Eine Menge von n Punkten $\zeta = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$ auf der Einheitskugel $S^2 = \{\mathbf{x} \in \mathbb{R}^3 \,|\, ||\mathbf{x}||_2 = 1\}$ heißt *spherical t-design* genau dann, wenn die Gleichung

$$\frac{1}{4\pi} \int_{S^2} f(\mathbf{x}) \; dS^2 = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{p}_i) \tag{2.64}$$

für alle $f \in \mathbb{P}_t$ gilt. Dabei bezeichne $\mathbb{P}_t$ die Menge aller Polynome vom Grad $\leq t$.

Eine ausführliche Einführung zu diesem Thema findet der interessierte Leser zum Beispiel in [25]. Bajnok hat im Fall $t = 3$ für $n = 6$, $n = 8$ und $n \geq 10$ nachgewiesen, dass *spherical designs* existieren. Man kann ebenfalls beweisen, dass die Anzahl von Punkten, die für ein *spherical t-design* benötigt werden, nach unten durch

$$\begin{aligned} n &\geq \frac{(t+1)(t+3)}{4} \text{ , falls } t \text{ ungerade} \\ n &\geq \frac{(t+2)^2}{4} \text{ , falls } t \text{ gerade} \end{aligned}$$

abgeschätzt werden kann. Damit ist gezeigt, dass für $n \leq 5$ kein *3-design* existiert. Für die Fälle $n = 7$ und $n = 9$ kann momentan nur vermutet werden, dass keine *spherical 3-designs* existieren. Diese Vermutung konnte *bisher* aber noch *nicht* bewiesen werden.

Die analytische Berechnung des Flächenintegrals aus (2.64) erfolgt mit den Methoden aus der Analysis durch Umwandlung in ein Doppel-Integral. Dazu benötigen wir eine Parametrisierung der Sphäre S^2. Mit Hilfe der üblichen Kugelkoordinaten erhalten wir diese über

$$\mathbf{x}(\varphi, \theta) = \begin{pmatrix} \cos(\varphi)\sin(\theta) \\ \sin(\varphi)\sin(\theta) \\ \cos(\theta) \end{pmatrix}, \tag{2.65}$$

wobei $0 \leq \varphi < 2\pi$ und $0 \leq \theta \leq \pi$ ist. Damit gilt

$$\begin{aligned} \frac{1}{4\pi} \int_{S^2} f(\mathbf{x}) dS^2 &= \int_0^{2\pi} \int_0^{\pi} f(\mathbf{x}(\varphi,\theta)) \cdot |\mathbf{x}_\varphi \times \mathbf{x}_\theta| \; d\varphi \; d\theta \\ &= \int_0^{2\pi} \int_0^{\pi} f(\mathbf{x}(\varphi,\theta)) \cdot \sin(\theta) \; d\varphi \; d\theta \end{aligned} \tag{2.66}$$

Die Gleichung (2.64) muss für alle Polynome $f \in \mathbb{P}_3$ erfüllt sein. Aus der Algebra ist bekannt, dass es genügt, die Gleichung für alle Monome vom Grad ≤ 3 zu zeigen.

Nach (2.66) gilt für ein Monom $x_1^i x_2^j x_3^k$ mit $i+j+k \leq 3$

$$\begin{aligned}\frac{1}{4\pi}\int_{S^2} x_1^i x_2^j x_3^k \, dS^2 &= \frac{1}{4\pi}\int_0^{2\pi}\int_0^{\pi} \cos^i(\varphi)\sin^i(\theta)\sin^j(\varphi)\sin^j(\theta)\cos^k(\theta)\sin(\theta)\; d\varphi\; d\theta \\ &= \frac{1}{4\pi}\int_0^{2\pi}\int_0^{\pi} \cos^i(\varphi)\sin^{i+j+1}(\theta)\sin^j(\varphi)\cos^k(\theta)\; d\varphi\; d\theta.\end{aligned}$$

Durch Nachrechnen erhält man für die Monome die folgenden Integrale, die wir mit Hilfe der symbolischen Toolbox von Matlab verifiziert haben (siehe Anhang C).

Monom	Integral	Monom	Integral	Monom	Integral	Monom	Integral
1	1	x_2x_3	0	x_1	0	$x_1x_2^2$	0
x_3	0	$x_2x_3^2$	0	x_1x_3	0	x_1^2	$\frac{1}{3}$
x_3^2	$\frac{1}{3}$	x_2^2	$\frac{1}{3}$	$x_1x_3^2$	0	$x_1^2x_3$	0
x_3^3	0	$x_2^2x_3$	0	x_1x_2	0	$x_1^2x_2$	0
x_2	0	x_2^3	0	$x_1x_2x_3$	0	x_1^3	0

Die rechte Seite von Gleichung (2.64) erhält man durch Einsetzen der Punkte in die entsprechenden Monome. Weiterhin muss beachtet werden, dass ein Punkt auf der Kugel festgehalten und zusätzlich ein weiterer Punkt auf den Null-Meridian gelegt wird. Anderenfalls könnte aus einer Lösung durch Rotation der Einheitskugel beliebig viele andere Lösungen bestimmt werden. Deshalb legen wir im Folgenden $\varphi_1 = \varphi_2 = 0$ und $\theta_1 = \pi$ fest. Daraus ergibt sich das nichtlineare Gleichungssystem

$$\mathbf{f}(\varphi, \theta) : \mathbb{R}^5 \times \mathbb{R}^6 \to \mathbb{R}^{19}$$

mit $\mathbf{f}(\varphi, \theta)$ aus (2.67). Die Startbox für dieses System ist offensichtlich gegeben durch $\varphi_i = [0, \pi]$ für $i = 3, \ldots, 7$ und $\theta_j = [0, 2\pi]$ für $j = 2, \ldots, 7$. Mit Hilfe von SONIC konnte gezeigt werden, dass dieses Gleichungssystem keine Lösung in der gegebenen Startbox besitzt. Dadurch ist *mathematisch bewiesen*, dass keine Linearkombination der Monome die Gleichung (2.64) erfüllt (der Nachweis, dass nur ein Monom diese Eigenschaft nicht erfüllt, hätte offensichtlich auch genügt). Damit gilt der folgende

Satz 2.5 (Frommer, Lang, Beelitz, Willems)
Es gibt keine Menge von 7 Punkten $\zeta = \{\mathbf{p}_1, \ldots, \mathbf{p}_7\}$ auf der Einheitskugel, welche ein *spherical 3-design* ist.

$$
\mathbf{f}(\varphi,\theta) = \left(\begin{array}{l}
\sin(\theta_2)+(\sin(\theta_3)\cos(\varphi_3))+(\sin(\theta_4)\cos(\varphi_4))+(\sin(\theta_5)\cos(\varphi_5))+(\sin(\theta_6)\cos(\varphi_6))+(\sin(\theta_7)\cos(\varphi_7)) \\
(\sin(\theta_3)\sin(\varphi_3))+(\sin(\theta_4)\sin(\varphi_4))+(\sin(\theta_5)\sin(\varphi_5))+(\sin(\theta_6)\sin(\varphi_6))+(\sin(\theta_7)\sin(\varphi_7)) \\
1+\cos(\theta_2)+\cos(\theta_3)+\cos(\theta_4)+\cos(\theta_5)+\cos(\theta_6)+\cos(\theta_7) \\
\sin(\theta_2)^2+(\sin(\theta_3)\cos(\varphi_3))^2+(\sin(\theta_4)\cos(\varphi_4))^2+(\sin(\theta_5)\cos(\varphi_5))^2+(\sin(\theta_6)\cos(\varphi_6))^2+(\sin(\theta_7)\cos(\varphi_7))^2-7/3 \\
(\sin(\theta_3)\cos(\varphi_3))(\sin(\theta_3)\sin(\varphi_3))+(\sin(\theta_4)\cos(\varphi_4))(\sin(\theta_4)\sin(\varphi_4))+(\sin(\theta_5)\cos(\varphi_5))(\sin(\theta_5)\sin(\varphi_5))+ \\
\qquad (\sin(\theta_6)\cos(\varphi_6))(\sin(\theta_6)\sin(\varphi_6))+(\sin(\theta_7)\cos(\varphi_7))(\sin(\theta_7)\sin(\varphi_7)) \\
\sin(\theta_2)\cos(\theta_2)+(\sin(\theta_3)\cos(\varphi_3))\cos(\theta_3)+(\sin(\theta_4)\cos(\varphi_4))\cos(\theta_4)+(\sin(\theta_5)\cos(\varphi_5))\cos(\theta_5)+(\sin(\theta_6)\cos(\varphi_6))\cos(\theta_6)+ \\
\qquad (\sin(\theta_7)\cos(\varphi_7))\cos(\theta_7) \\
(\sin(\theta_3)\sin(\varphi_3))^2+(\sin(\theta_4)\sin(\varphi_4))^2+(\sin(\theta_5)\sin(\varphi_5))^2+(\sin(\theta_6)\sin(\varphi_6))^2+(\sin(\theta_7)\sin(\varphi_7))^2-7/3 \\
(\sin(\theta_3)\sin(\varphi_3))\cos(\theta_3)+(\sin(\theta_4)\sin(\varphi_4))\cos(\theta_4)+(\sin(\theta_5)\sin(\varphi_5))\cos(\theta_5)+(\sin(\theta_6)\sin(\varphi_6))\cos(\theta_6)+(\sin(\theta_7)\sin(\varphi_7))\cos(\theta_7) \\
1+\cos(\theta_2)^2+\cos(\theta_3)^2+\cos(\theta_4)^2+\cos(\theta_5)^2+\cos(\theta_6)^2+\cos(\theta_7)^2-7/3 \\
\sin(\theta_2)^3+(\sin(\theta_3)\cos(\varphi_3))^3+(\sin(\theta_4)\cos(\varphi_4))^3+(\sin(\theta_5)\cos(\varphi_5))^3+(\sin(\theta_6)\cos(\varphi_6))^3+(\sin(\theta_7)\cos(\varphi_7))^3 \\
(\sin(\theta_3)\cos(\varphi_3))^2(\sin(\theta_3)\sin(\varphi_3))+(\sin(\theta_4)\cos(\varphi_4))^2(\sin(\theta_4)\sin(\varphi_4))+(\sin(\theta_5)\cos(\varphi_5))^2(\sin(\theta_5)\sin(\varphi_5))+ \\
\qquad (\sin(\theta_6)\cos(\varphi_6))^2(\sin(\theta_6)\sin(\varphi_6))+(\sin(\theta_7)\cos(\varphi_7))^2(\sin(\theta_7)\sin(\varphi_7)) \\
\sin(\theta_2)^2\cos(\theta_2)+(\sin(\theta_3)\cos(\varphi_3))^2\cos(\theta_3)+(\sin(\theta_4)\cos(\varphi_4))^2\cos(\theta_4)+(\sin(\theta_5)\cos(\varphi_5))^2\cos(\theta_5)+(\sin(\theta_6)\cos(\varphi_6))^2\cos(\theta_6)+(\sin(\theta_7)\cos(\varphi_7))^2\cos(\theta_7) \\
(\sin(\theta_3)\cos(\varphi_3))(\sin(\theta_3)\sin(\varphi_3))^2+(\sin(\theta_4)\cos(\varphi_4))(\sin(\theta_4)\sin(\varphi_4))^2+(\sin(\theta_5)\cos(\varphi_5))(\sin(\theta_5)\sin(\varphi_5))^2+(\sin(\theta_6)\cos(\varphi_6))(\sin(\theta_6)\sin(\varphi_6))^2+ \\
\qquad (\sin(\theta_7)\cos(\varphi_7))(\sin(\theta_7)\sin(\varphi_7))^2 \\
(\sin(\theta_3)\cos(\varphi_3))(\sin(\theta_3)\sin(\varphi_3))\cos(\theta_3)+(\sin(\theta_4)\cos(\varphi_4))(\sin(\theta_4)\sin(\varphi_4))\cos(\theta_4)+ \\
\qquad (\sin(\theta_5)\cos(\varphi_5))(\sin(\theta_5)\sin(\varphi_5))\cos(\theta_5)+(\sin(\theta_6)\cos(\varphi_6))(\sin(\theta_6)\sin(\varphi_6))\cos(\theta_6)+(\sin(\theta_7)\cos(\varphi_7))(\sin(\theta_7)\sin(\varphi_7))\cos(\theta_7) \\
\sin(\theta_2)\cos(\theta_2)^2+(\sin(\theta_3)\cos(\varphi_3))\cos(\theta_3)^2+(\sin(\theta_4)\cos(\varphi_4))\cos(\theta_4)^2+(\sin(\theta_5)\cos(\varphi_5))\cos(\theta_5)^2+(\sin(\theta_6)\cos(\varphi_6))\cos(\theta_6)^2+(\sin(\theta_7)\cos(\varphi_7))\cos(\theta_7)^2 \\
(\sin(\theta_3)\sin(\varphi_3))^3+(\sin(\theta_4)\sin(\varphi_4))^3+(\sin(\theta_5)\sin(\varphi_5))^3+(\sin(\theta_6)\sin(\varphi_6))^3+(\sin(\theta_7)\sin(\varphi_7))^3 \\
(\sin(\theta_3)\sin(\varphi_3))^2\cos(\theta_3)+(\sin(\theta_4)\sin(\varphi_4))^2\cos(\theta_4)+(\sin(\theta_5)\sin(\varphi_5))^2\cos(\theta_5)+(\sin(\theta_6)\sin(\varphi_6))^2\cos(\theta_6)+(\sin(\theta_7)\sin(\varphi_7))^2\cos(\theta_7) \\
(\sin(\theta_3)\sin(\varphi_3))\cos(\theta_3)^2+(\sin(\theta_4)\sin(\varphi_4))\cos(\theta_4)^2+(\sin(\theta_5)\sin(\varphi_5))\cos(\theta_5)^2+(\sin(\theta_6)\sin(\varphi_6))\cos(\theta_6)^2+(\sin(\theta_7)\sin(\varphi_7))\cos(\theta_7)^2 \\
1+\cos(\theta_2)^3+\cos(\theta_3)^3+\cos(\theta_4)^3+\cos(\theta_5)^3+\cos(\theta_6)^3+\cos(\theta_7)^3
\end{array} \right) \tag{2.67}
$$

Kapitel 3

Globale Optimierung

Die in vielen Anwendungen auftretenden globalen Optimierungsprobleme lassen sich in der Regel mit lokalen Optimierungsverfahren nicht lösen, da die Funktion in dem zu betrachtenden Gebiet mehrere lokale Minimalpunkte besitzt. Die Bestimmung eines globalen Minimalpunktes ist unter diesen Bedingungen oft nur mit erheblichem Rechenaufwand möglich. Das Problem der globalen nichtlinearen Minimierung, das hier betrachtet werden soll, lässt sich wie folgt beschreiben.

$$\begin{cases} \min & f(\mathbf{x}) \\ \text{s.d.} & p_i(\mathbf{x}) \leq 0 \qquad (i = 1, \ldots, m) \\ & q_i(\mathbf{x}) = 0 \qquad (i = 1, \ldots, r) \\ & \mathbf{x} \in [\mathbf{x}^{(0)}] \end{cases} \tag{3.1}$$

Dabei ist $f : \mathbb{R}^n \to \mathbb{R}$ eine zunächst beliebige Funktion, die wir im Folgenden auch als *Zielfunktion* bezeichnen werden. Weiterhin seien zwei Funktionen $\mathbf{p} : \mathbb{R}^n \to \mathbb{R}^m$ und $\mathbf{q} : \mathbb{R}^n \to \mathbb{R}^r$ gegeben, die wir als *Nebenbedingungen* bezeichnen. Bereits an dieser Stelle möchte der Autor darauf hinweisen, dass in der Standardliteratur zur verifizierten Optimierung stets gefordert wird, dass die Zielfunktion (mindestens) zweimal stetig differenzierbar und die beiden Funktionen $\mathbf{p}$ und $\mathbf{q}$, die die Nebenbedingungen beschreiben, (mindestens) einmal stetig differenzierbar auf der Box $[\mathbf{x}^{(0)}]$ sind. Diese Forderungen sind nach Meinung des Autors viel zu stark und werden nicht für alle Kontraktionsverfahren benötigt. Wir haben uns aus diesem Grund zu einer robusten Implementierung in SONIC entschieden, die stets vor der Anwendung eines Kontraktionsverfahrens die entsprechenden Voraussetzungen überprüft. Wir nehmen im Folgenden an, dass eine Startbox $[\mathbf{x}^{(0)}] \in \mathbb{IR}^n_*$ gegeben ist. Gesucht ist das globale Minimum f^* von $f(\mathbf{x})$ in der Startbox $[\mathbf{x}^{(0)}]$ unter den gegebenen Nebenbedingungen $\mathbf{p}$ und $\mathbf{q}$. Neben dem globalen Minimum sind wir auch an der Menge der Minimalpunkte

$$X^* := \{\mathbf{x} \in [\mathbf{x}^{(0)}] \mid f(\mathbf{x}) = f^*, \mathbf{p}(\mathbf{x}) \leq \mathbf{0}, \mathbf{q}(\mathbf{x}) = \mathbf{0}\}. \tag{3.2}$$

interessiert.

Da das globale Maximum einer Funktion f gerade das globale Minimum der Funktion $-f$ ist, kann das Problem der globalen Optimierung mit dem der globalen Mini-

mierung (3.1) gleichgesetzt werden. In dieser Arbeit verstehen wir deshalb unter Optimierung immer die Minimierung einer Funktion f.

Die Bestimmung des globalen Minimums, mit oder ohne Nebenbedingungen, ist sehr viel aufwändiger als die eines lokalen Minimums. Es existieren leider keine lokalen Kriterien, um festzustellen, dass ein lokales Minimum auch dem globalen Minimum entspricht oder nicht.

Prinizipiell werden wir das Problem der globalen Optimierung genauso angehen wie das der Lösung nichtlinearer Gleichungssysteme. Auch hier wird ein Branch–and–Bound-Ansatz eingesetzt, bei dem der Bounding-Schritt entsprechend angepasst wird. Wie wir später im Detail sehen werden, profitiert der in SONIC integrierte Optimierer in hohem Maße von der Qualität des implementierten Gleichungslösers.

3.1 Optimierung ohne Nebenbedingungen

In diesem Abschnitt wollen wir zunächst auf Techniken zur Lösung von globalen Optimierungsproblemen *ohne* Nebenbedingungen, also für das Optimierungsproblem (3.1) mit $m = 0$ und $r = 0$, eingehen. Im nächsten Abschnitt werden wir dann zeigen, wie diese Techniken auch bei Problemen mit Nebenbedingungen eingesetzt werden können.

Der Branch–and–Bound-Ansatz kann unter Ausnutzung der Differenzierbarkeitseigenschaften von f beschleunigt werden. Als mögliche Beschleunigungsverfahren (Bounding) werden an dieser Stelle der Monotonie-Test, der Konkavitäts-Test, der Cut-Off-Test und ein spezielles Newton-Verfahren vorgestellt. Während beim Monotonie-Test die einmalige stetige Differenzierbarkeit von f ausreicht, muss für die anderen Beschleunigungsverfahren f zweimal stetig differenzierbar sein. Bei allen präsentierten Verfahren ist sichergestellt, dass keine Teilboxen, die globale Minimalpunkte enthalten, gelöscht werden. Es handelt sich also um zulässige Kontraktionsverfahren. Für das Branching, also für die Zerlegung des Problems in mehrere Teilprobleme, verwenden wir die so genannte Multisektion.

3.1.1 Monotonie-Test

Sei zunächst $[\mathbf{x}] \subset [\mathbf{x}^{(0)}]$ eine Box, die den Rand der Startbox nicht schneidet. Ist nun die Funktion f in der Box $[\mathbf{x}]$ *streng monoton* bezüglich einer Koordinatenrichtung, so kann die Box verworfen werden, da sie in diesem Fall keinen stationären Punkt und somit auch kein globales Minimum enthalten kann. Hat die zu betrachtende Box $[\mathbf{x}]$ dagegen mit dem Rand des ursprünglichen Optimierungsgebietes $[\mathbf{x}^{(0)}]$ einen nichtleeren Schnitt, so reicht die strenge Monotonie als Ausschlusskriterium nicht mehr aus, da Randminima keine stationären Punkte sind. In diesem Fall muss die entsprechende Boxkomponente auf den Randpunkt reduziert werden. Dieser Test wird in der Literatur meist als *Monotonie-Test* bezeichnet. Zusammenfassend ergibt sich Algorithmus 3.1.

Um die Funktionsweise des Monotonie-Tests zu verdeutlichen, betrachten wir nun das eindimensionale Beispiel aus Abbildung 3.1. Auf der Box $[x^{(3)}]$ ist die zu minimierende Funktion streng monoton steigend. Diese kann offensichtlich keine globalen

Algorithmus 3.1 MONOTONIE-TEST($[\mathbf{x}]$)

```
1: for i = 1, ..., n do
2:   if sup([f_i'][x]) < 0 then
3:     if sup([x_i]) = sup([x_i^(0)]) then {liegt auf Rand der Startbox}
4:       [x_i] = [x̄_i, x̄_i]
5:     else
6:       return „kann verworfen werden“
7:     end if
8:   end if
9:   if inf([f_i'][x]) > 0 then
10:    if inf([x_i]) = inf([x_i^(0)]) then {liegt auf Rand der Startbox}
11:      [x_i] = [x_i, x_i]
12:    else
13:      return „kann verworfen werden“
14:    end if
15:  end if
16: end for
```

Minimalstellen enthalten und folglich gelöscht werden. Die Funktion ist ebenfalls streng monoton steigend in den Boxen $[x^{(1)}]$ und $[x^{(4)}]$. Diese beiden Boxen treffen allerdings den Rand des ursprünglichen Suchbereiches. Trotzdem kann die Box $[x^{(4)}]$ ebenfalls gelöscht werden, da die Funktion streng monoton steigend ist und der *rechte* Rand getroffen wird. Für die Box $[x^{(1)}]$ muss dagegen der Rand gesondert betrachtet werden. Man reduziert in diesem Fall die Box $[x^{(1)}]$ auf das Punktintervall $[\inf([x^{(1)}]), \inf([x^{(1)}])]$. Die Box $[x^{(2)}]$ bleibt durch den Monotonie-Test unverändert.

Für eine effiziente Implementierung ist darauf zu achten, dass für die Anwendung des Monotonie-Tests die Zielfunktion nicht bezüglich *aller* Variablen auf der Box $[\mathbf{x}]$ stetig differenzierbar sein muss: Jede Koordinatenrichtung, in der f auf $[\mathbf{x}]$ stetig differenzierbar ist, kann auf strenge Monotonie untersucht werden.

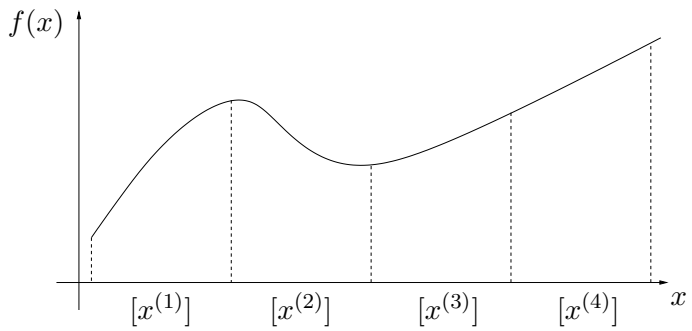

Abbildung 3.1: Der Monotonie-Test: Die Boxen $[x^{(3)}]$ und $[x^{(4)}]$ können verworfen werden. Die Box $[x^{(1)}]$ reduziert sich auf $\inf([x^{(1)}])$.

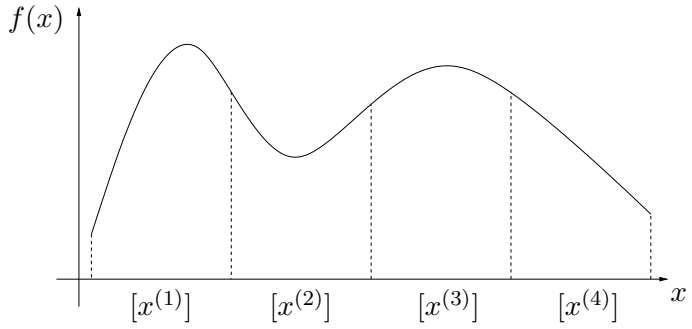

Abbildung 3.2: Der Konkavitäts-Test: Die Box $[x^{(3)}]$ kann verworfen werden. Die Boxen $[x^{(1)}]$ und $[x^{(4)}]$ reduzieren sich auf die entsprechenden Randpunkte.

3.1.2 Der Konkavitäts-Test

In diesem Abschnitt beschreiben wir den so genannten *Konkavitätstest*, der eigentlich die „Nicht-Konvexität" der Funktion f in einer Box $[\mathbf{x}]$ überprüft. Er erhielt seinen Namen in der Literatur wohl zur Vereinfachung der Sprechweise. Wahlweise sprechen wir also auch vom *Nicht-Konvexitäts-Test*.

Grundlage für diesen Test ist die Tatsache, dass die Konvexität eine notwendige Bedingung für ein lokales Minimum darstellt. Ausnahme bildet auch hier der Rand des ursprünglichen Optimierungsgebietes $[\mathbf{x}^{(0)}]$. Trifft die aktuelle Box den Rand von $[\mathbf{x}^{(0)}]$ und ist die Funktion f an keinem Punkt konvex, so muss nur noch der entsprechende Rand untersucht werden. Um die Eigenschaft der Nicht-Konvexität nachzuprüfen, muss man zusätzlich voraussetzen, dass die zu minimierende Funktion zweimal stetig differenzierbar in $[\mathbf{x}]$ ist. Nach Satz 1.6.3 in [67] ist eine Funktion f in einer Box $[\mathbf{x}]$ *konvex*, wenn die Hessematrix überall in $[\mathbf{x}]$ positiv semidefinit ist. Eine notwendige Bedingung für die positive Semidefinitheit ist, dass *alle* Diagonalelemente der Hessematrix nichtnegativ sind. Ist $[\mathbf{H}]$ die Hessematrix der Funktion f ausgewertet auf der Box $[\mathbf{x}]$ und gilt $\overline{h}_{i,i} < 0$ für ein i mit $i = 1, \ldots, n$, so kann die Box $[\mathbf{x}]$ verworfen werden, sofern sie den Rand der Startbox $[\mathbf{x}^{(0)}]$ nicht trifft.

Wir betrachten nun die Funktionsweise des Konkavitäts-Testes an dem eindimensionalen Beispiel aus Abbildung 3.2. Während der Test die Box $[x^{(3)}]$ verwerfen kann, müssen für die Boxen $[x^{(1)}]$ und $[x^{(4)}]$ die Ränder gesondert betrachtet werden. Die Box $[x^{(2)}]$ bleibt völlig unverändert.

3.1.3 Der Cut-Off-Test

Sei $\tilde{f}$ eine garantierte obere Schranke für das globale Minimum f^* und $[\mathbf{x}] \subseteq [\mathbf{x}^{(0)}]$. Gilt nun $\inf([f]([\mathbf{x})) > \tilde{f}$, so kann die Box $[\mathbf{x}]$ verworfen werden, da damit gezeigt wurde, dass *alle* Funktionswerte auf dieser Box garantiert größer als das globale Minimum sind. Dieser Test wird in der Literatur meist als *Cut-Off-Test* bezeichnet und kann ohne weitere Foderung an die Zielfunktion durchgeführt werden. Damit dieses Verfahren zulässig ist, muss ausschließlich garantiert werden, dass die Einschließung des Wertebereiches—auch für nicht-stetige Funktionen—korrekt ist. Die Abbildung 3.3

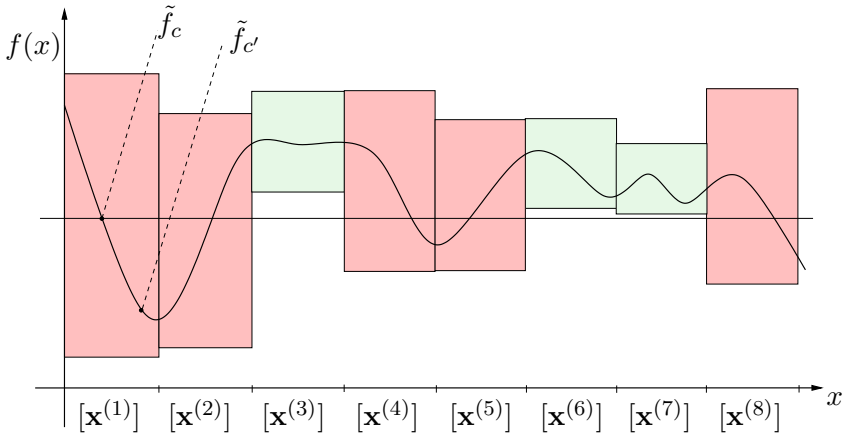

Abbildung 3.3: Der Cut-Off-Test: Die Boxen $[\mathbf{x}^{(3)}], [\mathbf{x}^{(6)}]$ und $[\mathbf{x}^{(7)}]$ können verworfen werden. Die schraffierten Flächen stellen jeweils Intervallauswertungen über dem entsprechenden Teil der x-Achse dar.

verdeutlicht die Funktionsweise des Cut-Off-Tests an einem eindimensionalen Beispiel. Mit der Höhenlinie auf dem Niveau von $\tilde{f}$ sieht man sofort, dass dort die hell schraffierten Flächen kein globales Minimum enthalten können.

An diesem Beispiel wird ebenfalls deutlich, dass der Cut-Off-Test umso effektiver wird, je besser der Wert des aktuellen globalen Minimums $\tilde{f}$ ist. Eine Verbesserung des Verfahrens könnte sich also ergeben, wenn an dieser Stelle *approximative* lokale Verfahren zur Verbesserung des „Auswertepunktes“ c eingesetzt würden. Eine weitere Variante könnte sein, dass die Funktion in mehreren Punkten der aktuellen Box ausgewertet wird und das Minimum von diesen Funktionswerten und dem „alten globalen Minimum“ als neues $\tilde{f}$ verwendet wird. Die Bestimmung des aktuellen globalen Minimums über den Referenzpunkt c' würde in Abbildung 3.3 sogar dazu führen, dass alle Boxen $[x^{(k)}]$ mit $k = 3, \ldots, 8$ verworfen werden könnten. Diese Möglichkeiten wurde bislang in SONIC nicht realisiert, da erste Untersuchungen in [65] gezeigt haben, dass sich das Laufzeitverhalten des Optimierers bei schweren Problemen nicht wesentlich ändert, wenn man bereits mit dem (bekannten) globalen Minimum startet.

Bei der Implementierung unterscheiden wir grundsätzlich zwischen zwei Varianten des Cut-Off-Tests. Zum einen kann der Cut-Off-Test auf *einer* einzigen Box durchgeführt werden. Zum anderen kann, wie bereits die Abbildung 3.3 nahelegt, dieser Test für eine ganze *Liste* von Boxen durchgeführt werden. Da die Effizienz dieser Methode aber stark von der Verwaltung der noch gehaltenen Boxen abhängt, verweisen wir an dieser Stelle auf die Diskussion in Kapitel 3.1.8.

3.1.4 Ein spezielles Newton-Verfahren

Entscheidend für das nächste Vefahren ist die Beobachtung, dass lokale Minima im Inneren der Startbox $[\mathbf{x}^{(0)}]$ stationäre Punkte sind, d.h. dort verschwindet der Gradient $\nabla f(\mathbf{x})$. Es ist also möglich, die aktuelle Suchbox zu kontrahieren, indem man das

System	**hybrider Newton**		**Newton-Verfahren**	
	Boxen	Zeit [s]	Boxen	Zeit [s]
Griewank	255	2.3	4 095	11.0
Levy	9 500	23.7	13 203	34.4
Kowalik	27 976	216.5	77 216	523.0
Siirola6	38 518	261.0	54 555	424.5
Sphere4	172 291	1 047.5	273 296	1 969.4
Arithm. Mittel	49 708	310.2	84 473	592.5
Geom. Mittel	13 509	79.8	36 209	175.3

Tabelle 3.1: Vergleich des Optimierers mit integriertem Newton-Verfahren bzw. hybriden Newton-Verfahren an verschiedenen Testbeispielen.

(reine oder hybride) Newton-Verfahren oder den hierarchischen Ansatz auf das Gleichungssystem $\nabla f(\mathbf{x}) = 0$ anwendet. Um dieses Verfahren durchführen zu können, muss die Zielfunktion allerdings *zweimal* stetig differenzierbar auf $[\mathbf{x}]$ sein, da im Newton-Verfahren die Differenzierbarkeit der Funktion vorausgesetzt wird. Im Folgenden werden wir das Gleichungssystem $\nabla f(\mathbf{x}) = 0$ auch als *Gradientensystem* bezeichnen. In der Literatur wird stets vorgeschlagen, nur einen *einzigen* Schritt des reinen Newton-Verfahrens anzuwenden, da das Verfahren in der Regel sehr teuer ist. In der aktuellen SONIC-Version wird ein Schritt des hybriden Newton-Verfahrens angewendet (mit dem C^W-LP- und dem Inverse-Midpoint-Präkonditionierer auf dem Originalsystem). Dies bedeutet zwar einen Mehraufwand (der vor allem durch die Berechnung der optimalen Präkonditionierer hervorgerufen wird), der sich aber durchaus wegen der Qualität des Verfahrens lohnt. Dagegen hat sich der Einsatz des hierarchischen Ansatzes zur Lösung des Gradientensystems in verschiedenen Testreihen als zu aufwändig erwiesen. Tabelle 3.1 zeigt einen Vergleich zwischen dem verwendeten hybriden Newton-Verfahren und der Anwendung eines einzigen Schrittes des Newton-Verfahrens mit dem Inverse-Midpoint-Präkonditionierer. Dabei stehen die präsentierten Testbeispiele für ein breites Spektrum an durchgeführten Tests. Während der Testphase konnte der Autor in der Tat kein System finden, bei dem sich durch das teurere hybride Newton-Verfahren eine schlechtere Gesamtzeit bzw. Gesamtboxzahl ergab.

3.1.5 Einschluss des globalen Minimums

Wie bei der Lösung von nichtlinearen Gleichungssystemen tritt auch bei der Lösung von globalen Optimierungsproblemen der so genannte Cluster-Effekt auf. In [61] wird gezeigt, dass eine Auswertung von f mit (mindestens) der Mittelwert-Form dem Cluster-Effekt bei Optimierungsproblemen erfolgreich entgegengewirkt. Da beim integrierten hybriden Newton-Verfahren sogar die Hesse-Matrix der zu minimierenden Funktion berechnet wird, setzen wir bei SONIC neben der natürlichen Intervallerweiterung und der Taylorform erster Ordnung auch die Taylorform zweiter Ordnung ein (die berechneten Ergebnisse werden dann jeweils miteinander geschnitten). Wir haben bereits in Kapitel 1 gesehen, dass die natürliche Intervallauswertung für Boxen mit großem Durchmesser

in einigen Fällen eine bessere Einschließung liefert als die Taylorform. Aus diesem Grund berechnen wir an geeigneter Stelle beide Einschließungen und bilden anschließend den Schnitt dieser Intervalle.

3.1.6 Erweiterung des Newton-Verfahrens

Setzt man voraus, dass die Zielfunktion im globalen Minimum stetig differenzierbar ist, so kann das in SONIC integrierte Newton-Verfahren um eine weitere interessante Technik ergänzt werden. Zu diesem Zweck führen wir eine zusätzliche Variable y ein, für die

$$f(\mathbf{x}) = y \qquad \text{bzw.} \qquad f(\mathbf{x}) - y = 0 \tag{3.3}$$

gilt. Eine Einschließung der Variablen $[\mathbf{y}]$ gibt damit zunächst eine Einschließung des Wertebereiches der Zielfunktion auf der aktuellen Box $[\mathbf{x}]$ an. Als globale Minimierer kommen allerdings ausschließlich Boxen $[\tilde{\mathbf{x}}]$ in Frage, für die $f^* \in f([\tilde{\mathbf{x}}])$ gilt: Der Wertebereich der Zielfunktion auf der Box muss offensichtlich das globale Minimum enthalten. Aus diesem Grund ist es zu jedem Zeitpunkt zulässig, die obere Schranke der Variablen $[y]$ mit einer oberen Schranke für das globale Minimum $\tilde{f}$ aufzudatieren, ohne dabei Lösungen des Gleichungssystems zu verlieren. Bei der Implementierung sollte darauf geachtet werden, dass stets

$$[y] \leftarrow \left[\underline{y}\ ,\ \min\left\{\overline{y}, \tilde{f}\right\}\right]$$

gesetzt wird. Um die beschriebene Eigenschaft auszunutzen, wird in SONIC die Gleichung $f(\mathbf{x}) - y = 0$ in das Gradientsystem aufgenommen. Anschließend können die vorgestellten Kontraktionsverfahren auf das (modifizierte) Gradientensystem angewendet werden. Wie in Kapitel 3.1.4 beschrieben, wird in SONIC momentan ein einziger Schritt des hybriden Newton-Verfahrens angewendet. Der Wertebereich der Variablen y wird zu Beginn der Berechnung zunächst entweder über einen Forward-Sweep von CP oder eine beliebige Intervall-Erweiterung mit dem Wert $[f]([\mathbf{x}^{(0)}]) \equiv [y]$ initialisiert. An dieser Stelle möchte der Autor auf eine weitere subtile Fehlerquelle hinweisen: Die obere Schranke für das globale Minimum darf *nicht* mit dem Wert der Variablen $[y]$ aufdatiert werden, da diese nach Anwendung von CP keine Einschließung des Wertebereiches von f auf $[\mathbf{x}]$ mehr darstellen muss.

Die Idee der so genannten „*one-dimensional Newton iteration*“ wurde der Dissertation von H. Tapamo [65] entnommen. Dort wird vorgeschlagen einen (eindimensionalen) Newton-Schritt auf die Gleichung $f(\mathbf{x}) - y = 0$ anzuwenden (es wird also nach *einer einzigen*, festzulegenden Variablen aufgelöst). Die Technik wurde in den verifizierten globalen Optimierer von Tapamo und Frommer integriert und dort als *abgekoppeltes*, also eigenständiges, Verfahren betrachtet; eine Integration in das Gradientensystem erfolgt also nicht.

In vielen Fällen kann diese Technik die Effizienz eines globalen Optimierers stark verbessern. Das wesentliche Problem bei dieser Technik, welche in [65] genauer untersucht wird, ist allerdings die Wahl der Variablen, nach der im Newton-Verfahren aufgelöst wird. Momentan ist noch keine effiziente Technik zur Wahl der Variablen bekannt. Tapamo bevorzugt aus diesem Grund eine zyklische Wahl der Variablen.

Die Integration der Gleichung $f(\mathbf{x}) - y = 0$ in das Gradientensystem gewinnt nach Meinung des Autors noch mehr an Attraktivität, wenn zusätzlich auch die *symbolische Komponente* des Verfahrens berücksichtigt wird. Dazu betrachten wir noch einmal das modifizierte Gradientensystem

$$\left\{ \begin{array}{rcl} f(\mathbf{x}) - y & = & 0 \\ \partial f/\partial x_1(\mathbf{x}) & = & 0 \\ & \vdots & \\ \partial f/\partial x_n(\mathbf{x}) & = & 0 \end{array} \right. . \tag{3.4}$$

Wie bereits erwähnt, wird in SONIC die Möglichkeit geboten, während der Vorverarbeitungsphase gemeinsame Teilterme zu erkennen. Dadurch kann die günstige Situation eintreten, dass im System (3.4) die Variable y zusätzlich auch in weiteren Gleichungen vorkommt (man denke an die Situation, in der zum Beispiel die Exponential-Funktion in $f(\mathbf{x})$ vorkommt). Die hinzugefügte Gleichung wird mit dem ursprünglichen Gradientensystem und damit mit den Original-Variablen stark verzahnt. Es entsteht also eine Kopplung, die in der ursprünglichen Formulierung nicht vorhanden ist. Durch diesen Umstand kann die Variable y nicht nur mit Hilfe der ersten Gleichung (wie in [65]), sondern auch über eine mögliche Änderung der Original-Variablen $x_1, \ldots, x_n$ kontrahiert werden. Durch Einsatz des hybriden Newton-Verfahrens, welches auch die Anwendung von CP beinhaltet, werden damit Änderungen in den Original-Variablen auch in die Variable y und umgekehrt propagiert. Das beschriebene Verfahren kann in SONIC durch die boolsche Variable `UseAdditionalConstraintInOptimizer` ein- bzw. ausgeschaltet werden. In den Standardeinstellungen wird diese Variable auf `true` gesetzt.

Die vorgestellte Technik ist in seiner jetzigen Form ausschließlich in SONIC integriert. Wie in [65] gezeigt wird, können bereits mit der ursprünglichen Version der Technik sehr gute Ergebnisse erzielt werden. Durch Hinzunahme der symbolischen Komponente wird die Qualität der erzielten Ergebnisse sogar noch gesteigert. Um dies zu verdeutlichen, haben wir SONIC zunächst mit der ursprünglichen Version von Tapamo [65] (Newton auf Gradientensystem und eindimensionaler Newton auf der zusätzlichen Gleichung) und anschließend mit dem Newton-Verfahren auf dem modifizierten Gradientensystem ausgeführt. Die daraus resultierenden Ergebnisse haben wir in Tabelle 3.2 für einige aussagekräftige Beispiele dargestellt. Man sieht, dass sich durch die neue symbolische Komponente des Verfahrens eine Einsparung an betrachteten Boxen um ca. 5–10% ergibt.

3.1.7 Multisektion

In diesem Abschnitt werden wir eine Möglichkeit vorstellen, wie die im Verfahren gerade behandelte Box unterteilt werden kann, um eine engere Einschließung des Wertebereiches und damit letztendlich des globalen Minimums zu erhalten. Auch bei der globalen Optimierung wird die Box ausschließlich in der gewählten Richtung halbiert. Eine Untersuchung zur Wahl des Unterteilungspunktes findet hier nicht statt. Erste Tests haben aber gezeigt, dass an dieser Stelle weiteres Potential zur Verbesserung der Performance liegt.

System	urspr. Variante	mod. Gradientensystem
HM4	37 615	35 014
KOW	23 757	21 469
Siirola5	15 372	14 167
Siirola6	38 518	35 398
Shubert	66 571	62 177
W4	172 291	159 541
Arithm. Mittel	59 021	54 628
Geom. Mittel	42 709	39 398

Tabelle 3.2: Anzahl betrachteter Boxen für verschiedene Systeme bei Verwendung der ursprünglichen „one-dimensional Newton iteration“ bzw. bei Integration der Gleichung in das Gradientensystem.

Die Wahl der Unterteilungsrichtung bezüglich der die aktuell betrachtete Box halbiert werden soll, wurde bereits in vielen Arbeiten genau untersucht. Eine detaillierte und trotzdem verständliche Analyse findet der interessierte Leser in [8]. Dort wird zunächst herausgearbeitet, dass die Unterteilung der Box in insgesamt *vier*—nicht wie bei der Bisektion in zwei—Teilboxen am günstigsten ist. Es wird gezeigt, dass diese Strategie bereits im seriellen Fall hervorragend arbeitet und seine Vorteile vor allem bei der Parallelisierung des Verfahrens hat. Es ergibt sich das in Alg. 3.2 dargestellte Verfahren, welches in der Originalarbeit als Strategie C (mit $l = 2$) beschrieben ist.

Algorithmus 3.2 MULTISEKTION($[\mathbf{x}], \mathcal{L}$)

Beschreibung: Führt Multisektion auf der Box $[\mathbf{x}]$ durch und fügt die resultierenden Boxen in die (zunächst leere) Liste $\mathcal{L}$ ein.

1: **for** $i = 1, \ldots, n$ **do**
2: $\quad w_i = |(\nabla f)_i([\mathbf{x}])| \cdot \operatorname{diam}([x_i])$
3: **end for**
4: $k = \min\{j \mid w_j([\mathbf{x}]) = \max_{i=1,\ldots,n} w_i([\mathbf{x}])\}$
5: $w_k = w_k/2$
6: $l = \min\{j \mid w_j([\mathbf{x}]) = \max_{i=1,\ldots,n} w_i([\mathbf{x}])\}$
7: unterteile die Box $[\mathbf{x}]$ zunächst orthogonal zur Richtung l
8: unterteile die entstandenen Boxen orthogonal zur Richtung k
9: füge die resultierenden vier Teilboxen in $\mathcal{L}$ ein

Die Wahl der Unterteilungsrichtung erfolgt über die Verwendung des Gradienten. Die vorgestellte Strategie zielt darauf hin, die Box orthogonal zu einer Richtung zu unterteilen, die am stärksten zum Durchmesser von $f(\mathbf{x})$ beiträgt:

$$|(\nabla f)_i([\mathbf{x}])| \cdot \operatorname{diam}([x_i]).$$

Dieses Vorgehen haben wir bereits bei der Bisektion für die Lösung von nichtlinearen Gleichungssystemen kennengelernt.

3.1.8 Listenverwaltung

In der Literatur sind drei gängige Strategien zur Auswahl der nächsten zu behandelnden Box bekannt. Die „richtige“ Auswahl kann den Ablauf eines globalen Optimierers entscheidend beeinflussen. Bislang unterscheidet man in der Regel zwischen den folgenden Kriterien:

Oldest-First-Strategie
Bei dieser Strategie wird die Box ausgewählt, die sich am längsten in der Liste befindet. Daraus ergibt sich eine FIFO-Datenstruktur (first in first out), d.h. die Liste wird als Warteschlange implementiert. Diese Strategie hat den Nachteil, dass der Algorithmus langsamer als die anderen Strategien gegen das globale Minimum konvergiert. Dadurch werden in der Regel mehr Boxen betrachtet, da der Cut-Off-Test in diesem Fall nicht effektiv genug ist.

Depth-First-Strategie
Bei dieser Strategie werden die Boxen in einer LIFO-Datenstruktur verwaltet, d.h. die Liste wird als Stack implementiert. Dadurch wird eine Box so lange unterteilt, bis sie ein Abbruchkriterium erfüllt. Der Vorteil dieser Strategie ist, dass in der Regel weniger Boxen in der Liste verwaltet werden müssen als bei den anderen Strategien (es wird also weniger Speicher benötigt). Die Strategie hat allerdings auch einen entscheidenden Nachteil. Man stelle sich eine monoton fallende Zielfunktion vor. Wird nun nach der Bisektion jeweils die linke Hälfte der Box an erster Stelle in den Stack eingefügt, so werden alle Boxen „auf der linken Seite“ völlig unnötig unterteilt.

Best-First-Strategie
Wählt man als nächstes zu bearbeitendes Element im Branch–and–Bound-Algorithmus immer dasjenige mit minimaler unterer Schranke $\inf([f](\mathbf{x}))$ für den Wertebereich über dieser Box, so soll dies hier als *Best-First-Strategie* bezeichnet werden. Auf diese Weise wird immer die vielversprechenste Box als nächstes gewählt, da eine kleinere untere Schranke für den Wertebereich die Wahrscheinlichkeit erhöht, dass in dieser Box das globale Minimum angenommen wird.

In [8, Satz 2.2] wird gezeigt, dass die *Best-First-Strategie* (ohne Initialisierung mit dem globalen Minimum) mindestens so gut ist, wie die beiden anderen Strategien *mit* Initialisierung des globalen Minimums. Aus diesem Grund ist in SONIC die Best-First-Strategie implementiert.

Wie bereits zuvor erwähnt, kann der Cut-Off-Test nicht nur auf einer einzigen Box, sondern natürlich auch für eine komplette Liste von Boxen durchgeführt werden. Dies hat den Vorteil, dass die Listenlänge in einigen Fällen „auf einen Schlag“ stark reduziert werden kann und damit wieder mehr Speicherplatz zur Verfügung steht. In den bekannten Tools (wie z.B. GlobSol) wird stets eine sortierte Liste zur Speicherung der Boxen verwendet. Dabei werden alle Listenelemente bezüglich der zugehörigen unteren Schranke für f auf dem entsprechenden Teilbereich der Suchbox aufsteigend sortiert. Beim Cut-Off-Test für die sortierte Liste werden nun alle Listenelemente—beginnend

beim ersten—sequentiell abgearbeitet. Findet man eine Box $[\mathbf{z}]$ für die $\inf([f]([\mathbf{z}]) > \tilde{f}$ gilt, so können auch alle Listenelemente, die auf diese Box in der Liste folgen, verworfen werden, da deren zugehörigen unteren Schranken garantiert größer als $\tilde{f}$ sind und diese damit kein globales Minimum enthalten können.

Eine Alternative zur Speicherung von Boxen ist der *Heap*. Im Gegensatz zur sortierten Liste, wird beim Heap lediglich ein Aufwand von $\mathcal{O}(\log n)$ anstatt $\mathcal{O}(n)$ zum Einfügen eines Elementes benötigt. Allerdings ergeben sich für den Heap Nachteile beim Cut-Off-Test: Der Heap ist ein binärer Baum, bei dem ausschließlich garantiert werden kann, dass die Priorität des Vaterknotens kleiner oder gleich der Prioritäten der Söhne ist. Führt man nun für den Cut-Off-Test einen Baumdurchlauf durch und findet dabei einen oder mehrere Knoten, dessen zugehörigen unteren Schranken größer als $\tilde{f}$ sind, so können *nur* die darunter liegenden Teilbäume komplett gelöscht werden. Bei der Implementierung des Heaps mit Hilfe eines Feldes können auf diese Weise Lücken (diese repräsentieren die entsprechenden Teilbäume) entstehen. Die entstandenen Lücken müssen nach dem Cut-Off-Test mit dem Aufwand $\mathcal{O}(n)$ „geschlossen“ werden (z.B. über Methode `heapify` in [12]): Dabei werden zunächst die Lücken mit Elementen vom Ende des Feldes gefüllt. Anschließend wird eine relativ einfache Umwandlung des Feldes in einen Heap durchgeführt. Bei der sortierten Liste entfällt dieser Arbeitsschritt, da diese ab einer gewissen Stelle bis zum Ende abgeschnitten wird. Der erste Teil der Liste bleibt völlig unverändert. In [65] wurde untersucht, wie sich unterschiedliche Datenstrukturen auf die Performance des globalen Optimierers auswirken. Es konnte festgestellt werden, dass in vielen Fällen der Heap der sortierten Liste vorzuziehen ist.

Nach Erfahrung des Autors ist es allerdings bei vielen, vor allem bei schweren, Problemen der Fall, dass nach einer gewissen Zeit in der Datenstruktur nur noch Elemente gehalten werden, bei denen die unteren Schranken für den Wertebereich auf diesen Boxen gleich sind. Damit wird jede Box immer an den *Anfang* der Datenstruktur eingefügt. Ist dies der Fall, so macht die Verwendung eines Heaps in meinen Augen keinen Sinn mehr. Aus diesem Grund wird in der Startphase des Algorithmus stets ein Heap als Datenstruktur verwendet. Ändert sich nach einer vorgegebenen Anzahl von Schritten das globale Minimum nicht mehr, so wird auf eine sortierte Liste „umgeschaltet“.

Eine weitere Methode, Boxen in einer Liste anzuordnen, ist der so genannte *RejectIndex*, welcher von Csendes et al. entwickelt wurde. Dazu bezeichne $[w] := [f]([\mathbf{x}])$ eine Einschließung des Wertebereiches der Zielfunktion auf der aktuell betrachteten Box $[\mathbf{x}]$. Dann ist der RejectIndex definiert als

$$pf^*([\mathbf{x}]) := \frac{f^* - \underline{w}}{\overline{w} - \underline{w}} \ .$$

Für den RejectIndex muss das globale Minimum f^* oder zumindest eine gute Approximation des Minimums $\tilde{f}$ bekannt sein. Über die Qualität dieses Ansatzes lässt sich leider momentan noch nicht viel aussagen. In [65] wird eine Variante des RejectIndex vorgestellt, die vielversprechend erscheint. Nachteilig bei der Verwendung des RejectIndex ist allerdings, dass der Cut-Off-Test *gar nicht mehr* auf der Liste (Heap) $\mathcal{L}$ durchgeführt kann, da keine Anordnung bezüglich des Infimums des Wertebereiches vorliegt. Wir verzichten aus diesem Grund momentan auf die Implemtierung dieser Technik in SONIC.

3.1.9 Das serielle Verfahren

Im Folgenden wird das serielle Verfahren für die globale Optimierung ohne Nebenbedingungen beschrieben, das auf den Verfahren aus den vorigen Abschnitten beruht. Das Verfahren ist dabei—wie bei der verifizierten Lösung von nichtlinearen Gleichungssystemen—in der Darstellung in zwei Teile gegliedert, den Rahmen (Algorithmus 3.3) und die Bearbeitung eines Listenelementes (Algorithmus 3.4). Diese Unterteilung erfolgt lediglich aus Übersichtsgründen und der Tatsache, dass die Routine CheckElementGOP auch bei der Parallelisierung verwendet wird.

Wir verwenden in SONIC wesentliche Teile des Algorithmus von Berner [8], welcher am Lehrstuhl für Angewandte Informatik/Algorithmik an der Bergischen Universität Wuppertal entwickelt wurde und sich in der Praxis als effizient herausgestellt hat. Zwischen der Implementierung von SONIC und der von Berner gibt es im Wesentlichen zwei Unterschiede. Der erste liegt in der Anwendung des Newton-Verfahrens. Berner schlägt vor, das Newton-Verfahren nur anzuwenden, wenn die aktuell betrachtete Box $[\mathbf{x}]$ einen gewissen Durchmesser $\varepsilon_{\text{newton}}$ unterschreitet (es wird vorgeschlagen, dass jede Komponente maximal den Durchmesser 0.01 aufweisen darf). Begründet wird dieses Vorgehen durch die Vermutung, dass das Newton-Verfahren nur auf kleinen Boxen eine sinnvolle Kontraktion erreicht. Diese Beobachtung kann der Autor beim Einsatz des „normalen“ Newton-Verfahrens für einige Testsysteme durchaus bestätigen. Wie bereits erwähnt, setzen wir in SONIC das hybride Newton-Verfahren ein. Dabei konnte beobachtet werden, dass *für alle getesteten Systeme* die besten Ergebnisse erzielt werden, wenn der Wert $\varepsilon_{\text{newton}}$ auf ∞ gesetzt wird, also das hybride Newton-Verfahren *immer* durchgeführt wird. Der zweite Unterschied existiert bei der Implementierung des Abbruchkriteriums. Berner fügt ausschließlich eine Box $[\mathbf{x}]$ in die Liste der (potentiellen) Lösungen ein, wenn das Kriterium $\operatorname{diam}([f]([\mathbf{x}])) < \varepsilon_f$ für ein vorgegebenes ε_f erfüllt ist. Wie wir später noch sehen werden, stellt dieses Vorgehen in den Augen des Autors eine schlechte Strategie dar: Betrachtet man ein Optimierungsproblem mit einer extrem flachen Zielfunktion (vergl. das Tibken-Problem), so ist das von Berner gewählte Kriterium sehr schnell erfüllt und relativ große Boxen werden bereits frühzeitig in die Lösungsliste eingefügt. Nach Meinung des Autors sollte ein gutes Abbruchkriterium daher *auch* die Größe der aktuell betrachteten Box berücksichtigen. Wir haben uns aus diesem Grund für eine Kombination beider Kriterien entschieden. Eine Box wird damit in die Lösungsliste eingefügt, wenn die Box einen gewissen Durchmesser ε_x unterschreitet *oder* der *relative* Durchmesser der Einschließung der Zielfunktion auf der Box *extrem klein* ist, zum Beispiel $\varepsilon' = 10^{-12}$.

Geht man davon aus, dass das Newton-Verfahren für jede Box eingesetzt wird, so sollte der ursprüngliche Algorithmus leicht umstrukturiert werden, um eine noch bessere Performance zu erzielen. Da es sich dabei im Wesentlichen aber nur um Umordnungen handelt, verweisen wir für detaillierte Erläuterungen des Algorithmus auf die Dissertation von Berner [8].

Algorithmus 3.3 GLOBALE OPTIMIERUNG (ohne Nebenbedingungen/seriell)

Beschreibung: Suche globales Optimum in Startbox $[\mathbf{x}^{(0)}]$. Eine obere Schranke für das globale Minimum wird in $\tilde{f}$ gespeichert. Noch zu bearbeitende Boxen werden in $\mathcal{L}$ verwaltet und potentielle Minimierer in $\mathcal{R}$ gespeichert.

1: $\tilde{f} = \sup([f](\text{mid}([\mathbf{x}^{(0)}])))$
2: $[\mathbf{x}^{(0)}]$.SET_RANK($\inf([f]([\mathbf{x}^{(0)}]))$)
3: $\mathcal{L} = \{[\mathbf{x}^{(0)}]\}$; $\mathcal{R} = \emptyset$
4: **while** $\mathcal{L} \neq \emptyset$ **do**
5: entnehme erstes Element $[\mathbf{x}]$ aus Arbeitsliste $\mathcal{L}$
6: CHECKELEMENTGOP($\mathcal{L}, \mathcal{R}, \tilde{f}, [\mathbf{x}]$)
7: CUT-OFF-TEST($\tilde{f}, \mathcal{L}$)
8: **end while**
9: CUT-OFF-TEST($\tilde{f}, \mathcal{R}$)

3.1.10 Numerische Ergebnisse

Die numerischen Ergebnisse wurden alle mit den Standard-Einstellungen von SONIC durchgeführt. Um eine gewisse Unabhängigkeit von der Last der verschiedenen Rechnersysteme zu bekommen, haben wir alle Tests mit SONIC mindestens drei mal durchgeführt und den Mittelwert der Messergebnisse genommen. Vergleichen wollen wir an dieser Stelle wiederum mit dem bewährten verifizierten Optimierer von GlobSol. Dazu haben wir sowohl SONIC als auch GlobSol auf einem Sun Fire-Server mit acht Prozessoren (1.2 GHz) und 36 GB Hauptspeicher getestet. Desweiteren vergleichen wir SONIC für einige ausgewählte Testbeispiele mit dem Optimierer von Tapamo und Frommer [65]. Dazu haben wir die beiden Tools auf einem 2.8 Ghz Pentium4 Linux PC getestet. Dabei verwendet SONIC die Intervall-Bibliothek *filib++* und der Optimierer von Tapamo und Frommer *C-XSC*. Die resultierenden numerischen Ergebnisse sind in Tabelle 3.3 zusammengefasst.

Auch bei der verifizierten Optimierung ohne Nebenbedingungen ist zu erkennen, dass SONIC bis auf wenige Ausnahmen bessere Ergebnisse bezüglich der benötigten Laufzeit liefert. Dabei kann beobachtet werden, dass SONIC gerade bei schwereren Testproblemen eine besondere Effizienz unter Beweis stellt. Auffällig ist allerdings, dass die Anzahl der betrachteten Boxen bei SONIC in vielen Fällen deutlich höher ist, als bei den anderen Tools. Dieses Phänomen ist aber in der Regel nicht auf die Qualität des verwendeten Verfahrens zurückzuführen, sondern liegt in der Tatsache begründet, dass bei den verschiedenen Tools unterschiedlich gezählt wird: Nach der Durchführung der Bisektion werden bei den anderen Verfahren zusätzlich noch weitere Kontraktionsverfahren auf die entstandenen Teilboxen angewendet, die dadurch möglicherweise verworfen werden können. Dies hat in manchen Fällen zur Folge, dass pro Iterationsschritt in SONIC noch vier weitere Boxen gezählt werden, die bei den anderen Methoden eventuell nicht mehr gezählt werden.

Algorithmus 3.4 CHECKELEMENTGOP($\mathcal{L}, \mathcal{R}, \tilde{f}, [\mathbf{x}]$)

Beschreibung: Bearbeitet das Listenelement $[\mathbf{x}]$ aus dem seriellen Algorithmus

if $[\mathbf{x}]$.GET_RANK() $> \tilde{f}$ **then**
  $\mathcal{L} = \emptyset$ {da die untere Schranke minimal in $\mathcal{L}$ ist}
  return
end if
führe NICHT-KONVEXITÄTS-TEST für $[\mathbf{x}]$ durch
if Box $[\mathbf{x}]$ konnte gelöscht werden **then**
  return
end if
if $\inf([f]([\mathbf{x}])) > \tilde{f}$ **then** {CUT-OFF-TEST mit Taylorentwicklung 2. Ordnung}
  return
end if
führe MONOTONIE-TEST für $[\mathbf{x}]$ durch
if Box $[\mathbf{x}]$ kann verworfen werden **then**
  return
end if
führe HYBRIDES NEWTON-VERFAHREN für modifiziertes Gradientensystem durch
if Box $[\mathbf{x}]$ kann verworfen werden **then**
  return
end if
MULTISEKTION($[\mathbf{x}], \mathcal{Q}$)
for $i = 1, \ldots,$ Anzahl der erzeugten Teilboxen $[\mathbf{x}^{(i)}]$ in $\mathcal{Q}$ **do**
  bestimme $[w] := [f]([\mathbf{x}^{(i)}])$ mit Mittelwert-Form
  $[\mathbf{x}^{(i)}]$.SET_RANK($\underline{w}$)
  $\tilde{f} := \min\{\tilde{f}, \sup([f](\mathbf{c}))\}$ wobei $\mathbf{c}$ der Mittelpunkt von $[\mathbf{x}^{(i)}]$ ist
  if reldiam$([w]) \leq \varepsilon_f$ **oder** diam$([\mathbf{x}^{(i)}]) < \varepsilon_x$ **then**
    füge $[\mathbf{x}^{(i)}]$ in $\mathcal{R}$ ein
  else
    füge $[\mathbf{x}^{(i)}]$ in $\mathcal{L}$ ein
  end if
end for

System	SONIC		SONIC (Sun)		GlobSol (Sun)		Tapamo	
	Boxen	Zeit [s]	Boxen	Zeit [s]	Boxen	Zeit [s]	Boxen	Zeit [s]
Trefethen	287	1.07	287	2.75	244	6.79		
GP	960	4.47	961	11.21	1 004	22.36		
G7	1 356	1.04	1 356	2.89	?	1.26		
Levy	9 500	23.71	9 501	68.70	20 253	8 826.74	14 628	12.44
Hansen	773	3.26	773	8.15	703	17.19		
HM3	3 967	6.64	3 968	13.60	631	9.36	19 444	250.01
HM4	35 014	65.76	35 014	163.09	88 857	31 122.49		
Kowalik	27 976	216.50	27 976	636.85	—	—	109 521	1 537.80
Siirola5	15 372	116.74	15 372	290.68	30 481	9 803.24	124 356	334.25
Siirola6	38 518	261.01	38 518	707.31	†	†	867 537	3 266.77
Shubert	66 571	78.82	66 572	157.41	23 913	423.75	46 698	92.03
W4	172 291	1 047.52	172 299	2 659.11	†	†	—	—
Tibken	1 437 612	1 170.06			†	†	—	—

Tabelle 3.3: Vergleich von SONIC, GlobSol und dem Optimierer von Tapamo. Beim Problem G7 wurde von GlobSol keine Angabe über die Anzahl der betrachteten Boxen gemacht.

3.2 Optimierung mit Nebenbedingungen

Bei der Optimierung mit Nebenbedingungen können nicht alle Algorithmen aus dem vorherigen Abschnitt übernommen werden. Zur Veranschaulichung betrachten wir das folgende Optimierungsproblem

$$\begin{cases} \min & x+y \\ \text{s.d.} & x^2+y^2=1 \end{cases}$$

mit den Einschließunge $[x] = [-1,1]$ und $[y] = [-1,1]$. Da offensichtlich für den Gradienten $\nabla f(x,y) = (1,1)$ gilt, würde der Monotonie-Test die Startbox sofort verwerfen oder zumindest auf den Rand, der das globale Minimum *nicht enthält*, reduzieren. Ähnliche Beispiele können auch für den Nicht-Konvexitäts-Test konstruiert werden. Bei der Optimierung mit Nebenbedingungen dürfen diese beiden Verfahren also *nicht* mehr eingesetzt werden. Auf diese wichtige Besonderheit wird in einigen Algorithmen zur verifizierten Lösung von Optimierungsproblemen (wie zum Beispiel [38, Algorithmus 14]) nicht geachtet.

Selbst die Aufdatierung des globalen Minimums ist bei der Optimierung mit Nebenbedingungen nicht mehr ohne weiteres durchführbar. Dazu muss zunächst sichergestellt sein, dass die Nebenbedingungen zumindest in einem Punkt erfüllt sind. Um zu garantieren, dass die Rigorosität unseres Algorithmus erhalten bleibt, führen wir nun den Begriff der *Feasibilität* ein.

Definition 3.1 Eine Box $[\mathbf{x}]$ heißt *feasible* für das Optimierungsproblem (3.1), falls ein $\mathbf{x} \in [\mathbf{x}]$ existiert mit $\mathbf{p}(\mathbf{x}) \leq \mathbf{0}$ und $\mathbf{q}(\mathbf{x}) = \mathbf{0}$.

Für den Nachweis der Feasibilität einer Box können die Verfahren zur Verifikation bei nichtlinearen Gleichungssystemen aus Abschnitt 2.6 eingesetzt werden. Es ist zu beachten, dass bei der globalen Optimierung mit Nebenbedingungen hauptsächlich *unterbestimmte Systeme* vorkommen. Der Nachweis der Bedingung $\mathbf{q}([\mathbf{x}]) \leq \mathbf{0}$ erfolgt mit Hilfe einer beliebigen Intervallerweiterung. Gilt folglich $\sup([q_i]([\mathbf{x}])) \leq 0$ für alle i und ist die Existenz einer Lösung von $\mathbf{q}(\mathbf{x}) = \mathbf{0}$ verifiziert, so ist die Box feasible.

Bei der Aufdatierung des aktuellen globalen Minimums muss neben der Feasibilitäts-Prüfung noch eine weitere Änderung vorgenommen werden. Da das aktuelle globale Minimum nur für Punkte aktualisiert werden darf, die feasible sind, so reicht eine Auswertung der Funktion auf dem Mittelpunkt der Box nicht aus. Hat man die Feasibilität einer Box nachgewiesen, so ist immer noch nicht bekannt, *welcher* Punkt aus der Box die Nebenbedingungen erfüllt. Das Update muss also in diesem Fall über $\tilde{f} = \min\{\tilde{f}, \sup([f]([\mathbf{x}]))\}$ erfolgen.

3.2.1 Die Fritz-John-Bedingungen

In diesem Abschnitt betrachten wir die so genannten *Fritz-John-Bedingungen*, die notwendigerweise in einem lokalen oder globalen Minimum eines Optimierungsproblems

mit Nebenbedingungen gelten. Die Fritz-John-Bedingungen sind wie folgt definiert

$$\begin{pmatrix} \sum_{i=0}^{m} u_i + \sum_{i=1}^{r} Ev_i - 1 \\ u_0 \nabla f(\mathbf{x})^T + \sum_{i=1}^{m} u_i \nabla p_i(\mathbf{x})^T + \sum_{i=1}^{r} v_i \nabla q_i(\mathbf{x})^T \\ u_1 p_1(\mathbf{x}) \\ \vdots \\ u_m p_m(\mathbf{x}) \\ q_1(\mathbf{x}) \\ \vdots \\ q_r(\mathbf{x}) \end{pmatrix} = \mathbf{0} \ , \tag{3.5}$$

wobei $u_0, \ldots, u_m$ und $v_1, \ldots, v_r$ *Lagrange-Multiplikatoren* sind. Die erste Gleichung entspricht einer *linearen* Normalisierungsbedingung [23], für die $E = [1, 1 + \epsilon_0]$ gilt, wobei ϵ_0 der Maschinengenauigkeit entspricht. In der Literatur werden durchaus verschiedene Normalisierungsbedingungen verwendet, die allerdings zusätzliche Forderungen an das Optimierungsproblem (3.1) stellen. Da wir auf diese Einschränkungen verzichten wollen, setzen wir ausschließlich die lineare Normalisierungsbedingung ein.

Die Fritz-John-Bedingungen können nun mit Hilfe des hybriden Newton-Verfahrens oder anderen Techniken gelöst werden. Es ist zu beachten, dass dieses Verfahren auch ohne den Nachweis der Feasibilität durchgeführt werden kann. Allerdings benötigen wir dafür noch Einschließungen für die Lagrange-Multiplikatoren. Wir nehmen dazu an, dass wir aktuell eine Teilbox $[\mathbf{x}] \subseteq [\mathbf{x}^{(0)}]$ betrachten. Definiert man für ein beliebiges $\mathbf{x} \in [\mathbf{x}]$ die Matrix

$$\mathbf{A}(\mathbf{x}) = \begin{pmatrix} 1 & 1 & \cdots & 1 & E & \cdots & E \\ \nabla f(\mathbf{x})^T & \nabla p_1(\mathbf{x})^T & \cdots & \nabla p_m(\mathbf{x})^T & \nabla q_1(\mathbf{x})^T & \cdots & \nabla q_r(\mathbf{x})^T \end{pmatrix} \tag{3.6}$$

und den Vektor

$$[\mathbf{w}]^T = ([u_0], \ldots, [u_m], [v_1], \ldots, [v_r])^T \ , \tag{3.7}$$

dann können die ersten $(n+1)$ Gleichungen des Fritz-John-Systems (3.5) mit Hilfe des Gleichungssystems

$$\mathbf{A}(\mathbf{x}) \cdot \mathbf{w} = \mathbf{e_1} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{3.8}$$

beschrieben werden. Da $\mathbf{x} \in [\mathbf{x}]$ beliebig gewählt war, erhält man Einschließungen für die Lagrange-Multiplikatoren über die Lösung des linearen Intervallgleichungssystems

$$\mathbf{A}([\mathbf{x}]) \cdot [\mathbf{w}] = \mathbf{e_1} \ , \tag{3.9}$$

Da keine Anfangseinschließungen für die Lagrange-Multiplikatoren bekannt sind, wird mit Hilfe der (präkonditionierten) Intervall-Gauß-Elimination (siehe zum Beispiel [38])

das Intervallgleichungssystem auf obere Dreiecksgestalt gebracht. Schlägt diese Prozedur fehl, da durch ein Intervall geteilt werden muss, welches die Null enthält, so können keine Einschließungen für die Lagrange-Multiplikatoren berechnet werden. In diesem Fall muss die Box in mehrere Teilboxen unterteilt und die Prozedur wiederholt werden. Ist die Gauß-Elimination erfolgreich, so ergibt sich im Fall $m + r < n$ ein Intervall-Gleichungsystem der Form

$$[\mathbf{R}] \cdot [\mathbf{w}] = [\mathbf{b}^{(1)}] \tag{3.10}$$

$$\mathbf{0} = [\mathbf{b}^{(2)}] \tag{3.11}$$

Im nächsten Schritt wird überprüft, ob $\mathbf{0} \in [\mathbf{b}^{(2)}]$ gilt. Ist dies nicht der Fall, so kann die aktuell betrachtete Box verworfen werden, da das Gleichungssystem (3.11) inkonsistent ist. Anderenfalls kann das System eine Lösung besitzen und (3.10) muss betrachtet werden. Dieses Gleichungssystem wird nun mit Hilfe der Rückwärts-Substitution gelöst. Damit erhält man die gewünschten Einschließungen für die Lagrange-Multiplikatoren. Hat man einmal Einschließungen bestimmt, so kann die oben beschriebene Methode offensichtlich in jeder Iteration wieder eingesetzt werden, um verbesserte Einschließungen zu berechnen. Numerische Experimente haben aber gezeigt, dass dieses Vorgehen keine Verbesserung bringt: Die Kontraktion der Variablen durch Newton und CP in der CheckBox-Phase ist anscheinend völlig auszureichend bzw. qualitativ besser als die gerade vorgestellte Variante.

3.2.2 Das serielle Verfahren

Im Folgenden wird das implementierte serielle Verfahren zur verifizierten Optimierung mit Nebenbedingungen beschrieben. Wie bereits zu Beginn dieser Arbeit erwähnt, lag das Hauptaugenmerk dieses Projektes bei der verifizierten Lösung von nichtlinearen Gleichungssystemen. Wir haben aus diesem Grund eine Implementierung gewählt, die sich auf die aktuellen Standard-Verfahren [38, 23] stützt und an einigen Stellen geeignet ergänzt wurden. Das resultierende Verfahren ist in Pseudo-Code in Algorithmus 3.5 dargestellt.

Zunächst wird die obere Schranke $\tilde{f}$ für das globale Minimum mit dem Wert ∞ initialisiert. Anschließend wird eine Einschließung des Wertebereiches der Zielfunktion auf der Startbox berechnet und das Infimum dieses Wertes als Priorität in die Box eingetragen. Dann wird die Box in den *Heap* $\mathcal{L}$ eingefügt.

Solange der Heap nun nicht leer ist, wird die folgende Prozedur durchgeführt. Als erstes wird die Box mit der kleinsten, zugehörigen unteren Schranke für den Wertebereich (das oberste Element im Heap) aus $\mathcal{L}$ entnommen. Für diese wird nun mit Hilfe der Mittelwert-Form eine Einschließung des Wertebereiches der Zielfunktion berechnet. Ist das Infimum dieser Einschließung größer als die aktuelle obere Schranke für das Minimum, so kann die betrachtete Box verworfen werden. Anderenfalls aktualisieren wir mit diesem Wert die Priorität der Box. Man beachte, dass dieser Cut-Off-Test für alle Boxen durchgeführt werden kann. Anschließend wenden wir Constraint-Propagation auf das System der Nebenbedingungen an. Sind noch keine Einschließungen für die Lagrange-Multiplikatoren bekannt, so werden diese in einem nächsten Schritt berech-

Algorithmus 3.5 GLOBALE OPTIMIERUNG (mit Nebenbedingungen/seriell)

Beschreibung: Führt globale Optimierung mit Nebenbedingungen für Startbox $[\mathbf{x}^{(0)}]$ durch. Potentielle Lösungen werden in $\mathcal{S}$ gespeichert. Noch zu behandelnde Boxen werden in $\mathcal{L}$ gehalten.

1: $[\mathbf{x}^{(0)}]$.SET_RANK($\inf([f]([\mathbf{x}^{(0)}]))$)
2: $\mathcal{L} = \{[\mathbf{x}^{(0)}]\}$
3: $\mathcal{S} = \emptyset$
4: $\tilde{f} = \infty$
5: **while** $\mathcal{L} \neq \emptyset$ **do**
6: entnehme erste Box $[\mathbf{x}]$ aus $\mathcal{L}$
7: **if** $\inf([f]([\mathbf{x}])) > \tilde{f}$ **then** {mit Mittelwert-Form}
8: **cycle**
9: **end if**
10: $[\mathbf{x}]$.SET_RANK($\inf([f]([\mathbf{x}]))$)
11: führe CONSTRAINT-PROPAGATION für das System aller Nebenbedingungen durch {also Gleichungen und Ungleichungen}
12: **if** Box $[\mathbf{x}]$ kann verworfen werden **then**
13: **cycle**
14: **end if**
15: bestimme gegebenenfalls Einschließungen für die Lagrange-Multiplikatoren
16: führe CHECKBOX für Fritz-John-System aus {nichtlineare Gleichungssysteme}
17: **if** Box $[\mathbf{x}]$ kann verworfen werden **then**
18: **cycle**
19: **end if**
20: führe Feasibilitäts-Prüfung durch
21: **if** Box ist feasible **then**
22: $\tilde{f} = \min\{\tilde{f}, \sup([f]([\mathbf{x}]))\}$
23: **end if**
24: **if** $\tilde{f}$ wurde seit dem letzten CUT-OFF-TEST geändert **then**
25: führe CUT-OFF-TEST auf $\mathcal{L}$ und $\mathcal{S}$ durch
26: **if** Box $[\mathbf{x}]$ kann verworfen werden **then**
27: **cycle**
28: **end if**
29: **end if**
30: MULTISEKTION($[\mathbf{x}], \mathcal{Q}$)
31: **for** $i = 1, \ldots,$ Anzahl erzeugter Teilboxen $[\mathbf{x}^{(i)}]$ in $\mathcal{Q}$ **do**
32: **if** Box $[\mathbf{x}^{(i)}]$ ist klein genug **then**
33: füge $[\mathbf{x}^{(i)}]$ in $\mathcal{S}$ ein {aktualisiere Priorität}
34: **else**
35: füge $[\mathbf{x}^{(i)}]$ in $\mathcal{L}$ ein
36: **end if**
37: **end for**
38: **end while**

System	SONIC	GlobSol
Wolfe	3.22	0.41
Parabola	11.41	9.87
GCD	19.29	15.01
Paradolos	21.44	23.65
TBill	32.18	76.13

Tabelle 3.4: Vergleich von SONIC und GlobSol für die Lösung globaler Optimierungsproblem mit Nebenbedingungen für einige Testbeispiele. Angegeben ist jeweils die benötigte Zeit in Sekunden.

net. Anschließend wird die effiziente CheckBox-Routine aus dem nichtlinearen Löser auf das Fritz-John-System angewendet. Dieses kann aber offensichtlich nur gemacht werden, wenn Einschließungen für die Lagrange-Multiplikatoren bekannt sind. Kann im folgenden Schritt nachgewiesen werden, dass die Box $[\mathbf{x}]$ feasible ist, so wird anschließend die aktuelle obere Schranke für das globale Minimum aktualisiert. Dabei ist zu beachten, dass die Schranke nur mit dem Supremum einer Einschließung der Zielfunktion auf der *gesamten* Box aktualisiert werden darf. Konnte das globale Minimum verbessert werden, so wird anschließend der Cut-Off-Test auf den beiden Listen $\mathcal{L}$ und $\mathcal{S}$ durchgeführt. Anschließend wird die Box in mehrere Teilboxen mit Hilfe der Multisektion unterteilt und diese entweder in die Lösungsliste oder in den Heap $\mathcal{L}$ eingetragen. Für die Boxen, die in $\mathcal{L}$ eingetragen werden, wird jeweils die Priorität mit Hilfe der Mittelwert-Form aktualisiert.

3.3 Numerische Ergebnisse

Die numerischen Ergebnisse wurden alle mit den Standard-Einstellungen von SONIC und GlobSol auf einem Sun Fire-Server mit 8 Prozessoren (1.2 GHz) und 36 GB Hauptspeicher durchgeführt. Für SONIC haben wir die Intervallbibliothek filib++ gewählt. Da bei den meisten Testbeispielen GlobSol keine Angaben zur Anzahl der betrachteten Boxen gemacht hat (Ausgabe: ***), geben wir an dieser Stelle nur die erzielten Laufzeiten an. Die Resultate haben wir in Tabelle 3.4 zusammengefasst. Die Ergebnisse zeigen, dass bei den Testsystemen SONIC vergleichbare Resultate zu GlobSol liefert. Interessanterweise konnte für das „schwerste“ System ein deutlich besseres Resultat mit SONIC erzielt werden. Eine abschließende Beurteilung der Qualität des implementierten globalen Optimierers mit Nebenbedingungen soll allerdings an dieser Stelle aber aus zwei Gründen nicht erfolgen: Bis dato wurden dazu noch nicht ausreichend viele Testsysteme untersucht. Desweiteren soll in einer weiteren Projektphase der erst kürzlich implementierte Optimierer verbessert werden.

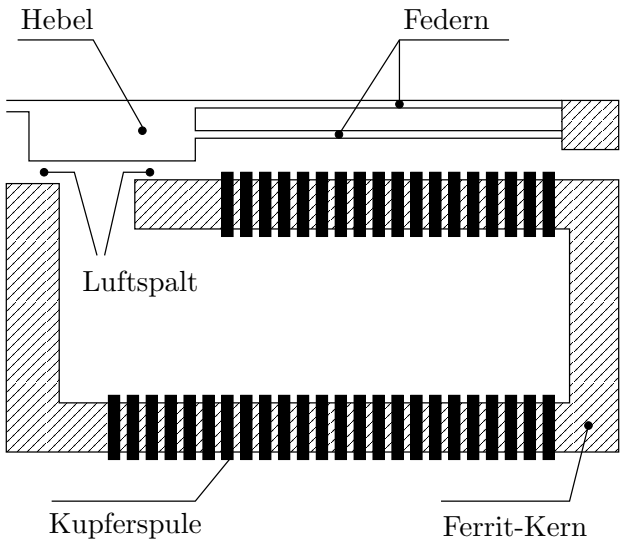

Abbildung 3.4: Schematische Darstellung eines Microrelais

3.4 Ein industrielles Anwendungsbeispiel

In diesem Abschnitt beschäftigen wir uns mit einem Problem aus der Industrie: Das Design eines Mikrorelais. Dieses Testproblem wurde uns freundlicherweise von Prof. Dr. Bernd Tibken von der Bergischen Universität Wuppertal zur Verfügung gestellt. Das betrachtete Mikrorelais wurde ursprünglich am Forschungszentrum Karlsruhe entwickelt. Die Maße dieses Bauteils, welches wir schematisch in Abbildung 3.4 dargestellt haben, betragen ungefähr 2mm × 3mm, wobei die Abstände zwischen den einzelnen Kontakten nur 30μm betragen. Die mathematische Modellierung dieses Bauteils liefert eine Differentialgleichung zweiter Ordnung

$$z''(t) + 2 \cdot \omega_0 \cdot z'(t) + \omega_0^2 \cdot z(t) + b(z(t)) \cdot I^2(t) = 0 \ , \tag{3.12}$$

wobei $z(t)$ die Abweichung des Hebels von seinem Ruhezustand beschreibt. Die Parameter d und $f_0 = \frac{\omega_0}{2\pi}$ stehen für die Dämpfungskonstante bzw. die Basisfrequenz. Der Parameter $I(z)$ steht für den Fahrdrahtstrom durch die Kupferspule. Die nichtlineare Funktion $b(z)$ ist gegeben durch

$$b(z) = \frac{\omega_0^2 \cdot A_G}{c_g \cdot \mu_0} \cdot \left(\frac{N \cdot \mu_0 \cdot \mu_r}{\frac{L_{Fe} \cdot A_G}{A_{Fe}} + 2\mu_r \cdot (\delta_0 + z)} \right)^2 \tag{3.13}$$

Bis auf die Parameter μ_r und c_g sind in (3.12) und (3.13) alle weiteren Parameter bekannt. Es gilt Für die Schätzung von μ_r und c_g werden nun verschiedene Spannungen $I_1, \ldots, I_{23}$ an die Spule angelegt. Nach der Modellgleichung (3.13) gilt für die Abweichung des Hebels von seinem Ruhezustand die folgende kubische Gleichung

$$z^3 + 2\gamma \cdot z^2 + \gamma^2 \cdot z + \nu = 0 \ , \tag{3.14}$$

Parameter	physikalische Bedeutung	Wert
d	Dämpfungskonstante	0.012
f_0	Basisfrequenz	829.7 Hz
A_G	Querschnitt des Luftspaltes	$20000\mu\text{m}^2$
μ_0	Magnetfeldkonstante	$1.2566 \cdot 10^{-6}$
N	Windungszahl der Spule	44
L_{F_e}	Länge des Ferritkerns	11.3mm
A_{F_e}	Querschnitt des Ferritkerns	$40000\mu\text{m}^2$

wobei die Abkürzungen

$$\begin{aligned} \gamma(\mu_r) &= \frac{L_{F_e} \cdot A_G}{2\mu_r \cdot A_{F_e}} + \delta_0 \\ \nu(c_g, I) &= \frac{N^2 \cdot A_G \cdot \mu_0 \cdot I^2}{4c_g} \end{aligned}$$

verwendet werden. Nach [11] gilt für die einzige—für dieses Bauteil relevante—Lösung

$$z(\mu_r, c_g, I(t)) = \frac{2\gamma(\mu_r)}{3} \cdot \left(\cos\left(\frac{\arccos\left(1 - \frac{27\nu(c_g, I(t))}{2\gamma(\mu_r)^3}\right)}{3} \right) - 1 \right) .$$

Mit Hilfe dieser Gleichung kann der Wert der Funktion z für unterschiedliche Ströme gemessen werden. Dafür definieren wir die Gleichung $\mathbf{y} = \mathbf{h}(\mu_r, c_g)$, wobei

$$h_i(\mu_r, c_g) = \frac{2\gamma(\mu_r)}{3} \cdot \left(\cos\left(\frac{\arccos\left(1 - \frac{27\nu(c_g, I_i(t))}{2\gamma(\mu_r)^3}\right)}{3} \right) - 1 \right)$$

für $i = 1, \ldots, 23$ gilt. Die gemessenen Werte für $I_i(t)$ und y_i für $i = 1, \ldots, 23$ kann der interessierte Leser in [31] nachlesen. Die Schätzungen für die Parameter μ_r und c_g werden nun über das Optimierungsproblem

$$\min_{\mu_r, c_g} ||\mathbf{y} - \mathbf{h}(\mu_r, c_g)||^2 = \min_{\mu_r, c_g} \sum_{i=1}^{23} (y_i - h_i(\mu_r, c_g))^2 \tag{3.15}$$

berechnet. Die Startbox für das Optimierungsproblem lautet $[500, 2500] \times [5.2, 6.7]$. Die Genauigkeitsanforderung legen wir auf $\varepsilon_j = 10^{-14}$ für $j = 1, 2$ fest.

Die große Herausforderung dieses Problems liegt in der Tatsache, dass die zu Zielfunktion nicht überall auf der Startbox definiert ist. Die zu minimierende Funktion ist offensichtlich nur für

$$c_g > \frac{454.3496}{2 \cdot \left(\frac{282.5}{\mu_r} + 3.182 \right)^3}$$

definiert. Diese Einschränkung hat zur Folge, dass dieses Optimierungsproblem bislang von keinem verifizierten Optimierer richtig gelöst werden kann (GlobSol zum Beispiel

bricht die Berechnung mit der Fehlermeldung „arithmetic exception“ ab). Bislang existiert ein Ansatz [31, 65], um dieses Problem zu lösen: Bei diesem wird zunächst überprüft, ob die Funktion auf der ganzen Box ausgewertet werden kann. Ist dies der Fall, so werden die normalen Optimierungs-Routinen aufgerufen. Anderenfalls wird die Box sofort unterteilt. Hat eine Box den Durchmesser ε' erreicht und kann die Funktion immer noch nicht fehlerfrei ausgewertet werden, so wird diese in eine separate Liste $\mathcal{L}_{\varepsilon'}$ gespeichert. Dieses Vorgehen hat aber den entscheidenden Nachteil, dass die Lösung *nicht verifiziert* ist, falls nach Ende der Berechnung $\mathcal{L}_{\varepsilon'} \neq \emptyset$ gilt. In diesem Fall könnte in einer der Boxen aus $\mathcal{L}_{\varepsilon'}$ ein kleinerer Funktionswert in der Zielfunktion angenommen werden. Da sowohl in [31] als auch in [65] $\mathcal{L}_{\varepsilon'} \neq \emptyset$ gilt, kann bislang keine verifizierte Aussage über das globale Minimum getroffen werden.

Mit Hilfe von SONIC kann dieses Problem recht elegant gelöst werden. Durch den Einsatz von Constraint-Propagation im hybriden Newton-Verfahren werden alle nichttotalen Bereiche aus der Box *automatisch* herausgeschnitten, die umständliche und nicht verifizierte Implementierung aus [31] kann damit entfallen. Nach 67 693 betrachteten Boxen und 3372.44 Sekunden liefert SONIC die obere Schranke für das globale Minimum 5.72959908703836e-13 und insgesamt 8090 Minimierer. Die berechneten Minimalstellen liegen (vergl. auch Abbildung 3.5) dicht beeinander und werden in der Regel in die Lösungsliste eingefügt, da die Intervallerweiterungen extrem kleine Einschließungen berechnen.

In Abbildung 3.5 haben wir die erzielten Resultate im Vergleich zur Variante von Ibraev dargestellt. Man sieht deutlich, dass die Qualität der Lösung von SONIC deutlich besser ist: Während mit SONIC nachgewiesen werden kann, dass die der Minimierer im Bereich $[698.545, 698.567] \times [5.3242, 5.3243]$ liegt, können mit der Variante von Ibraev nur Einschließungen im Bereich $[696.4, 701] \times [5.32, 5.33]$ berechnet werden.

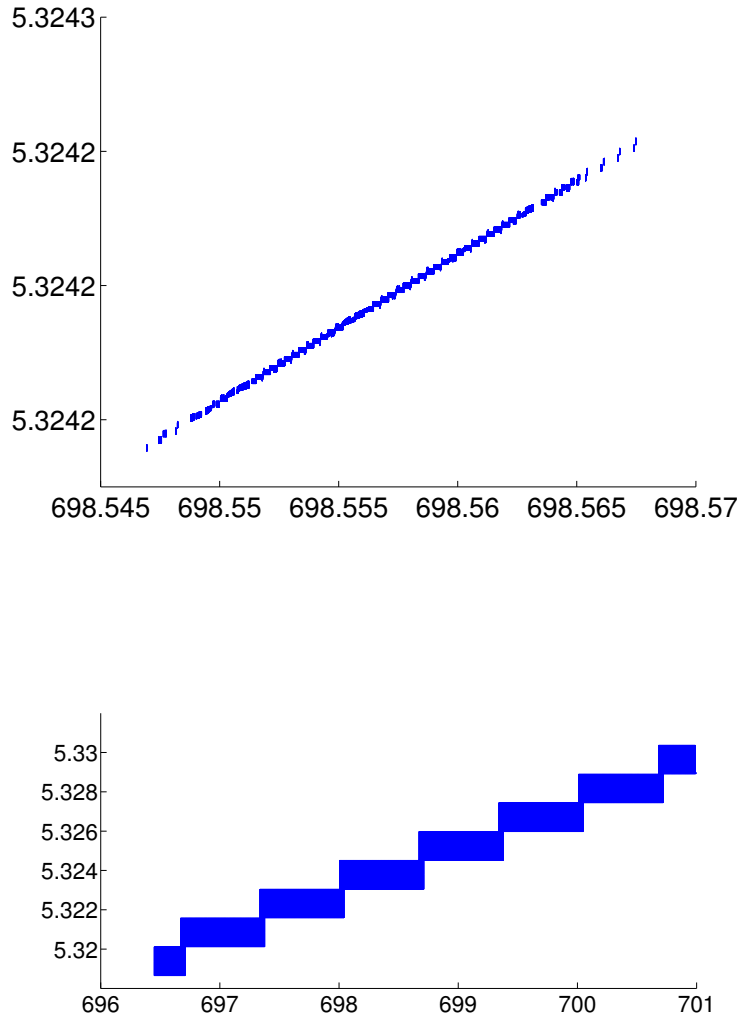

Abbildung 3.5: Vergleich der berechneten Minimalstellen zwischen SONIC (oben) und der Variante von Ibraev (unten).

Kapitel 4

Parallelisierung

In den bisherigen Abschnitten haben wir effiziente Techniken vorgestellt, die SONIC zu einem hervorragenden verifizierenden nichtlinearen Gleichungslöser bzw. Optimierer machen. Trotzdem muss festgestellt werden, dass Intervall-Verfahren immer noch sehr rechenintensiv sind. Ist man daran interessiert „realistische Probleme“ zu lösen, so sind gewöhnliche serielle Rechner diesen oft nicht gewachsen. Dies kann zum einen an zu langen Ausführungszeiten oder an einem zu hohen Speicherbedarf liegen. Aus diesem Grund versucht man, so genannte Höchstleistungsrechner zur Lösung dieser Probleme einzusetzen. Im Allgemeinen werden diese Höchstleistungsrechner in zwei Klassen, Vektorrechner und Parallelrechner, eingeteilt. Im Folgenden wenden wir uns ausschließlich der zweiten Kategorie zu, da nur diese zur Parallelisierung der präsentierten Verfahren geeignet erscheint.

4.1 Parallelrechner

4.1.1 Klassifizierung von Parallelrechnern

Nach einer Klassifizierung von Flynn [16] unterscheidet man bei Parallelrechnern im Wesentlichen zwischen so genannten *SIMD*- (single instruction stream, multiple data stream) und *MIMD*-Rechnern (multiple instruction stream, multiple data stream). Bei einem SIMD-Rechner arbeit ein Kontrollprozessor ein zentrales Programm ab. Dabei wird taktsynchron derselbe Befehl auf allen Prozessoren durchgeführt, wobei die verwendeten Daten unterschiedlich sein können. Auf einem MIMD-Rechner dagegen können alle Prozessoren völlig unterschiedliche Programme gleichzeitig ausführen.

Eine zusätzliche Klassifikation von Parallelrechnern kann anhand der Organisation des Speichers vorgenommen werden. Es stehen dabei grundsätzlich zwei Möglichkeiten zur Verfügung. In der ersten Kategorie teilen sich alle Prozessoren einen zentralen *gemeinsamen Speicher* (engl.: shared memory), auf den alle Prozessoren beliebig zugreifen können. Müssen Prozessoren bei der Bearbeitung ihrer Teilprobleme Daten austauschen, so findet die notwendige Kommunikation über den gemeinsamen Speicher statt.

In der zweiten Kategorie verfügt jeder Prozessor über seinen eigenen *lokalen* Speicher, ein gemeinsamer Speicher steht dabei nicht zur Verfügung. Diese Rechner bezeichnet man auch als Rechner mit verteiltem Speicher (engl.: distributed memory). Auf die verschiedenen Speicherkonzepte werden wir bei der Präsentation der unterschiedlichen Parallelisierungs-Ansätze zurückkommen.

4.1.2 Prozesse/Threads

Unter einem *Prozess* versteht man ein Programm während seiner Ausführung. Dieser Prozess besteht aus dem eigentlichen Programm, einem Programmzähler, den Programmdaten und einem Stack. Die Programmausführung geschieht über die sequentielle Abarbeitung der Instruktionen innerhalb des Programmcodes. Die dabei entstehende „Ablaufbahn“ durch die Instruktionen bezeichnet man als *Thread*. Ein Prozess besitzt somit mindestens einen Thread. Prozesse können normalerweise nur über Nachrichten kommunizieren, weil es keinen gemeinsamen Speicherbereich gibt. Threads dagegen sind so genannte Sub-Prozesse (oder auch leichtgewichtige Prozesse), die alle auf denselben Adressraum des zugrunde liegenden Mutter-Prozesses zugreifen. Er teilt das im Speicher liegende Programm mit den anderen Threads, arbeitet dieses aber unabhängig von den anderen ab (er hat also seinen eigenen Programmzähler) und besitzt seinen eigenen Stack.

4.1.3 Bewertung von parallelen Algorithmen

In der Regel wird zur Bewertung der Qualität eines parallelen Algorithmus der so genannte *Speedup* herangezogen. Wir definieren diesen durch

$$S(p) := \frac{\text{Ausführungszeit des seriellen Algorithmus}}{\text{Ausführungszeit des parallelen Algorithmus auf } p \text{ Prozessoren}}.$$

Der Speedup $S(p)$ gibt somit an, wie groß die Beschleunigung des parallelen Verfahrens gegenüber dem seriellen ist. Allerdings weicht diese Definition leicht von der Standard-Definition des Speedup ab: Normalerweise wird die Laufzeit des *schnellsten* seriellen Algorithmus mit der Laufzeit des parallelen Algorithmus verglichen. Da für die Parallelisierung des nichtlinearen Lösers von SONIC keine wesentlichen Änderungen am Grundalgorithmus vorgenommen werden mussten, sind diese beiden Definitionen im Grunde genommen gleichwertig.

Die Implementierung eines Verfahrens kann auf einem Parallelrechner nur dann sinnvoll sein, wenn der Anteil an Operationen, die von p Prozessoren parallel bearbeitet werden können, groß genug ist. Aufgrund des Gesetzes von Amdahl ist es ansonsten unmöglich, eine gute Beschleunigung nahe p zu erreichen.

Satz 4.1 (Amdahlsches Gesetz) Sei p die Anzahl der parallel arbeitenden Prozessoren und $a \in [0, 1]$ der parallelisierbare Anteil des Programms. Dann ist der maximal erreichbare Speedup

$$S_{\max}(p) = \frac{1}{(1-a) + \frac{a}{p}}.$$

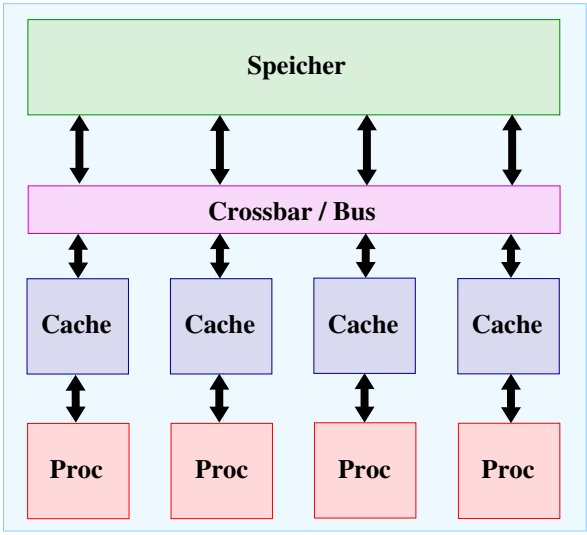

Abbildung 4.1: Parallelrechner mit gemeinsamem Speicher

Ist also nur die Hälfte des seriellen Verfahrens geeignet parallelisierbar, so kann gegenüber dem seriellen Programm—*unabhängig* von der Anzahl der verwendeten Prozessoren—maximal die doppelte Geschwindigkeit erreicht werden.

4.2 Parallelisierung mit OpenMP

In diesem Abschnitt präsentieren wir Möglichkeiten zur Parallelisierung auf Parallelrechnern mit gemeinsamem Speicher. Die Prozessoren kommunizieren bei diesem Konzept (eine schematische Darstellung findet sich in Abb. 4.1) über den gemeinsamen Speicher, der über so genannte *Crossbar-Switches* bzw. den *Bus* angesprochen wird. Ein wesentlicher Punkt bei der Parallelisierung mit gemeinsamen Speicher ist die Tatsache, dass Speicherzugriffe auf globale Daten mit Hilfe von Sperrmechanismen synchronisiert werden müssen, um nicht zum Beispiel bei gleichzeitigen Schreibzugriffen—so genannten *Race Conditions*—einen inkonsistenten Speicherinhalt zu erhalten. Dies kann vor allem bei einer großen Anzahl von Prozessoren problematisch werden. Ein weiterer Nachteil dieses Konzeptes liegt in der steigenden Komplexität des Verbindungsnetzwerkes.

Das so genannte *OpenMP Application Program Interface (OpenMP API)* spezifiziert eine Sammlung von Compiler-Direktiven, Bibliotheken und Umgebungsvariablen, welche die Parallelisierung auf Rechnern mit gemeinsamem Speicher für die Programmiersprachen C, C++ und Fortran standardisiert. Die Parallelisierung erfolgt hier auf Thread-Ebene. Der erste OpenMP-Standard wurde 1997 zunächst für Fortran veröffentlicht. Es folgten 1998 auch die Standardisierungen für die Programmiersprachen C und C++. Die aktuelle Version [57] wurde im Mai 2005 herausgegeben.

Um die parallelen Verfahren im Folgenden als Pseudocode darstellen zu können, stellen wir an dieser Stelle ausschließlich die aus dem umfangreichen OpenMP-Standard

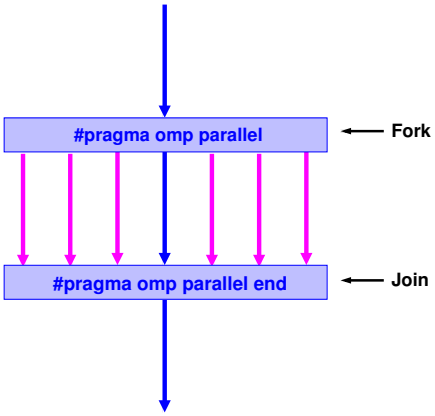

Abbildung 4.2: Das Fork-Join-Prinzip: Der Master-Thread erzeugt zu Beginn der parallelen Region ein Thread-Team. Am Ende werden alle Threads—bis auf den Master-Thread—beendet.

benötigten Routinen (nur für die Programmiersprache C++), die so genannten OpenMP-Direktiven, vor. Alle OpenMP-Direktiven für C++ werden über `#pragma`-Präprozessor-Direktiven spezifiziert. Jede dieser Direktiven beginnt standardmäßig mit `#pragma omp`.

- `Parallele Region / #prama omp parallel`

  Diese OpenMP-Direktive ist das wesentliche Konstrukt, welches eine so genannte *parallele Region* startet. Die Syntax dieses Konstrukts sieht wie folgt aus.

  ```
  #pragma omp parallel private( <list> ), shared( <list> ) {
      <Anweisungsblock>
  }
  ```

  Die Abarbeitung des Quellcodes eines beliebigen OpenMP-Programmes beginnt mit *einem* einzigen Thread, dem so genanntem Master-Thread. Sobald eine parallele Region erreicht wird, erzeugt dieser ein ganzes Team von Threads, deren Anzahl wird über die Umgebungsvariable `OMP_NUM_THREADS` gesteuert wird. Über die Anweisung `private(<list>)` wird eine Liste von Variablen vereinbart, die *privat* sind. Dies bedeutet, dass für jede Variable der Liste eine *uninitialisierte* Kopie für jeden Thread des Teams angelegt wird. Gemeinsame Variablen, die im gemeinsamen Adressraum liegen und damit von jedem Thread verändert werden können, werden über die Anweisung `shared(<list>)` vereinbart. Jede Anweisung aus dem Anweisungsblock wird von allen Teammitgliedern parallel ausgeführt, sofern dies nicht durch spezielle OpenMP-Direktiven „verhindert“ wird. Am Ende der parallelen Region werden alle Threads des Teams durch eine implizite Barriere synchronisiert. Anschließend fährt ausschließlich der Master-Thread mit der Ausführung fort. Dieses Prinzip wird in der Literatur meist als *Fork-Join-Prinzip* bezeichnet (eine schematische Darstellung findet der interessierte Leser in Abbildung 4.2).

- `Kritische Bereiche / #pragma omp critical`

  Durch die OpenMP-Direktive

```
#pragma omp critical {
    <Anweisungsblock>
}
```

  werden so genannte *kritische Bereiche* in parallelen Regionen definiert. Diese Anweisung bewirkt, dass sich nur jeweils *ein* Thread des Teams in diesem kritischen Bereich befinden darf, um den entsprechenden Anweisungsblock auszuführen. Durch die Verwendung von kritischen Bereiche können die oben erwähnten Race Conditions vermieden werden.

- `Parallele Schleife / #pragma omp for`

  Wir haben gesehen, dass jede Anweisung des Anweisungsblockes innerhalb einer parallelen Region von allen Threads ausgeführt wird. OpenMP stellt allerdings auch Direktiven zur Verfügung, die die Arbeit auf einzelne Threads verteilt. Die wohl wichtigste Möglichkeit dieser Art ist die Parallelisierung von Schleifen.

```
#pragma omp for schedule( <Strategie>, <chunk> ) {
    for ( i = a; i < e; ++i ) {
        <Schleifenrumpf>
    }
}
```

  Hierbei stehen die Schleifenvariable `i` und die Ausdrücke `a` und `e` für Integer-Variablen. Diese dürfen innerhalb des Schleifenrumpfes *nicht* verändert werden, da für die korrekte Ausführung die Anzahl der Iterationen *vor* Eintritt in die Schleife bekannt sein muss. Die Schleifenvariable `i` wird implizit zu einer privaten Variable für jeden Thread des Teams gemacht, damit die einzelnen Iterationen völlig *unabhängig* voneinander und auch in einer eventuell *anderen Reihenfolge* durch die Threads abgearbeitet werden können.

  Die Aufteilung der Iterationen auf die einzelnen Threads erfolgt über die Wahl von <`Strategie`>. Dabei stehen unter anderem die folgenden beiden Strategien zur Verfügung.

  static
  : Die Iterationen werden zunächst in Blöcke der Größe <`chunk`> aufgeteilt. Diese werden anschließend *zyklisch* auf die einzelnen Threads verteilt.

  dynamic
  : Die Iterationen werden in Blöcken der Größe <`chunk`> den verschiedenen Threads zugewiesen, wenn diese durch die Threads angefordert werden. Ein Thread führt also zunächst seinen Iterations-Block aus und fordert dann einen neuen Block an, solange bis alle Blöcke abgearbeitet sind.

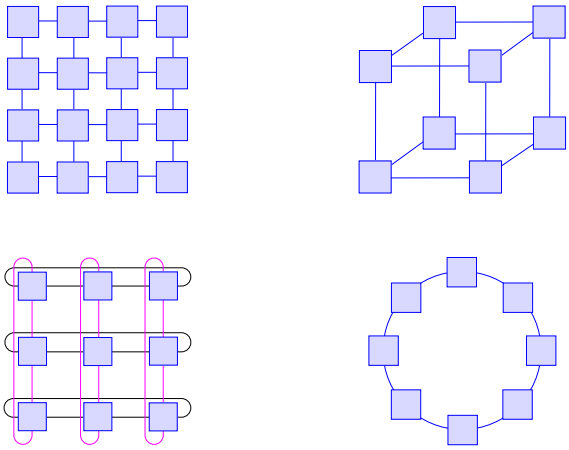

Abbildung 4.3: Parallelrechner mit verteiltem Speicher: Gitter (links oben), Hypercube (rechts oben), 2D-Torus (links unten) und Ring (rechts unten).

- `#pragma omp single`

  Die OpenMP-Direktive

  ```
  #pragma omp single {
      <Anweisungsblock>
  }
  ```

  stellt sicher, dass der entsprechende Anweisungsblock nur von *einem* Thread ausgeführt wird. Dies ist zum Beispiel sinnvoll, wenn globale Daten initialisiert werden: Die Startbox $[\mathbf{x}^{(0)}]$ muss genau einmal in die Arbeitsliste $\mathcal{L}$ eingefügt werden.

4.3 Parallelisierung mit MPI

In diesem Abschnitt gehen wir auf die Möglichkeit zur Parallelisierung auf Parallelrechnern mit verteiltem Speicher ein. In diesem Fall erfolgt die Kommunikation, also der Austausch von Daten, über Kommunikationskanäle oder einen Bus. Da es in der Regel sehr aufwändig ist, jeden Prozessor mit jedem anderen Prozessor zu verbinden, werden andere Verbindungsmuster verwendet. Die Prozessoren werden z.B. in einem Gitter, einem Torus, einem Hypercube oder ähnlichem angeordnet (eine schematische Darstellung findet man in Abbildung 4.3). Die Kommunikation zwischen zwei nicht direkt miteinander verbundenen Prozessoren verläuft dann über die dazwischenliegenden Prozessoren.

MPI steht für *Message Passing Interface* und ist der de-facto Standard für das so genannte Message Passing. Dieser spezifiziert eine Sammlung von Routinen und Umgebungsvariablen, welche die Parallelisierung auf Rechnern mit verteiltem Speicher für die Programmiersprachen C, C++ und Fortran standardisiert. Nach einem ersten

Workshop 1992, bei dem erste Vereinbarungen für einen Standard getroffen wurden, wurde im Juni die Version 1.0 [48] veröffentlicht. Der aktuelle Standard ist Version 2.0 [49] und besteht momentan aus ca. 323 Subroutinen.

Um die parallelen Verfahren bei Rechnern mit verteiltem Speicher als Pseudocode darstellen zu können, soll an dieser Stelle die Notation für die verwendeten Kommunikationsroutinen eingeführt werden. Im Gegensatz zur Implementierung mit OpenMP wollen wir hier auf eine exakte—an die tatsächlichen Routinen angelehnte—Notation verzichten, da dies den Sachverhalt unnötig verkomplizieren würde. Die hier eingeführten Kommunikationsroutinen basieren auf [8].

Bei jedem Versenden wird eine Nachricht mit einem *tag* versehen. Von einer Empfangsroutine werden nur solche Nachrichten empfangen, die mit dem gewünschten `tag` versehen sind. Die in den parallelen Verfahren verwendeten Kennzeichnungen für `tag` werden an den entsprechenden Stellen beschrieben. Bei den folgenden Kommunikationsroutinen bezeichnet jeweils `var` die Daten, die kommuniziert werden sollen. Da es sich hierbei oft um Felder handelt, wird zusätzlich ein weiterer Parameter `size` zur Verfügung gestellt, der die Länge der Nachricht angibt. Die benötigten Kommunikationsroutinen lassen sich damit wie folgt beschreiben:

- SEND(`i, tag, var, size`)
  Senden an Prozessor `i`. MPI sorgt dafür, dass nach dem Zurückkehren der Funktion der Puffer wieder beschrieben werden kann. Entweder wurde die Nachricht bereits gesendet oder der Pufferinhalt wurde in einen Systemspeicher kopiert.

- RECEIVE(`i, tag, var, size`)
  Blockierendes Empfangen von Prozessor `i`. MPI garantiert, dass nach Ausführung der Operation die Nachricht in der addressierten Empfangsvariablen abgelegt wurde und somit zur weiteren Verarbeitung zur Verfügung steht.

- SEND-ASYNC(`i, tag, var, size`)
  Asynchrones Senden an Prozessor `i`. Der Sender kann sofort weiterarbeiten, auch wenn der Empfänger noch nicht bereit ist, die Nachricht zu empfangen.

- MESSAGE-WAIT(`i, tag`)
  Diese Funktion gibt an, ob eine Nachricht von Prozessor `i` mit Kennzeichnung `tag` darauf wartet, empfangen zu werden.

- LAST-MESSAGE-SENT(`tag, var`)
  Diese Funktion gibt an, ob das letzte asynchrone Versenden der Nachricht an Prozessor `i` mit Kennzeichnung `tag` bereits beendet worden ist.

Wird in den obigen Empfangsroutinen die Nummer i eines Prozesses durch '?' ersetzt, so bedeutet dies, dass die Nachricht unabhängig vom Absender empfangen werden soll. In den präsentierten parallelen Algorithmen bezeichne `myrank` die eigene Nummer des Prozesses und `manager` die Nummer des so genannten Manager-Prozesses.

Im Folgenden stehe die Zahl p für die Anzahl der zur Verfügung stehenden Prozessoren bzw. Threads. Im Falle der MPI-Implementierung werden die Prozessoren von 0

bis $p-1$ durchnummeriert. Dabei steht die Prozessornummer 0 in der Regel für den Manager-Prozess.

4.4 Parallelisierung des Lösers

In diesem Abschnitt werden wir zwei Möglichkeiten vorstellen, den nichtlinearen Löser von SONIC mit Hilfe von OpenMP zu parallelisieren. Nachdem wir die Effizienz der Parallelisierungen genau untersucht haben, wenden wir uns der Parallelisierung des Lösers mit Hilfe von MPI zu.

4.4.1 Parallelisierung des Breadth-First-Algorithmus

Die wohl einfachste Methode ist die Parallelisierung des Breadth-First-Algorithmus. Zu diesen Zweck ändern wir den ursprünglichen Algorithmus 2.1 leicht ab. Zu Beginn der while-Schleife wird nun die aktuelle Länge l der Arbeitsliste $\mathcal{L}$ bestimmt. Anschließend wird eine for-Schleife integriert, die sukzessive alle Elemente der Liste $\mathcal{L}$ abarbeitet. Entstehen durch Kontraktionsverfahren oder Bisektion neue Boxen, so werden diese zunächst in einer weiteren Liste $\mathcal{N}$ gespeichert. Sind alle Elemente aus der Liste $\mathcal{L}$ bearbeitet, so werden die neuen Boxen aus $\mathcal{N}$ in die—nun leere—Arbeitsliste $\mathcal{L}$ verschoben. Man beachte, dass dieser Schritt sehr einfach implementiert werden kann, wenn man mit Zeigern auf Listen arbeitet: Es müssen dann nur die beiden Zeiger vertauscht werden. Wir erhalten damit die in Alg. 4.1 skizzierte Variante.

Algorithmus 4.1 Modifizierter Breadth-First-Algorithmus

```
 1: L = [x^(0)]; N = ∅; R = ∅
 2: while L ≠ ∅ do
 3:   l = Länge von L
 4:   for i = 1,...,l do
 5:     entferne die i-te Box [x] aus L
 6:     CheckBox( [x] )
 7:     if [x] ist „klein genug“ then
 8:       füge [x] in R ein
 9:     else
10:       unterteile [x] in zwei Teilboxen [x^(1)] und [x^(2)]
11:       füge [x^(1)] und [x^(2)] in N ein
12:     end if
13:   end for
14:   vertausche die Listen L und N
15: end while
```

Die Parallelisierung dieses Algorithmus mit OpenMP ist denkbar einfach. Von entscheidender Bedeutung ist dabei die Beobachtung, dass die Bearbeitung von allen Boxen $[\mathbf{x}]$ aus der Liste $\mathcal{L}$ (Zeilen 5–12) völlig *unabhängig* voneinander erfolgen kann. Die Parallelisierung kann dadurch sehr einfach über die Direktive `#pragma omp for` erfolgen.

System	$p =$	2	4	8	16	32
Robotics		1.97	3.81	7.41	10.26	12.21
Reactor		1.97	3.88	7.02	11.14	13.62
Hansen		1.98	3.89	7.49	10.01	11.22
Brent		1.98	3.84	7.51	12.19	18.99
Eco		1.97	3.89	7.70	12.28	19.81
Griewank		1.97	3.91	7.77	13.51	20.67
Human heart dipole		1.99	3.93	7.78	13.72	21.55
Design problem		1.99	3.94	7.80	13.94	22.97

Tabelle 4.1: Speedups der Breadth-First-OpenMP-Variante für p Prozessoren.

Doch welche globalen Daten müssen nun noch durch spezielle OpenMP-Direktiven vor Race Conditions geschützt werden? Implementiert man die Liste $\mathcal{L}$ als Feld von Zeigern auf Boxen (bei einer verketteten Liste wäre eine zusätzliche Synchronisation notwendig, da die Zeiger für Vorgänger bzw. Nachfolger verändert werden müssten) und führt jeder Prozessor eine private Lösungsliste $\mathcal{R}$, so muss ausschließlich das Einfügen der beiden Teilboxen $[\mathbf{x}^{(1)}]$ und $[\mathbf{x}^{(2)}]$ in die Liste $\mathcal{N}$ (Zeile 11) durch die Verwendung eines kritischen Bereiches geschützt werden. Die aktuelle Box $[\mathbf{x}]$ wird innerhalb der for-Schleife als private Variable vereinbart. Am Ende des Algorithmus werden alle privaten Lösungslisten $\mathcal{R}$ zu einer globalen Lösungsliste $\mathcal{S}$ zusammengefügt. Der Zugriff auf $\mathcal{S}$ muss ebenfalls durch einen kritischen Bereich geschützt werden. Wir erhalten dadurch den in Abbildung 4.2 skizzierten Breadth-First-OpenMP-Algorithmus.

Um die Effizienz dieser Parallelisierungsstrategie darzustellen, haben wir insgesamt acht verschiedene Testprobleme ausgesucht, die eine repräsentative Auswahl einer Vielzahl von getesteten Systemen darstellen. Tabelle 4.1 fasst die Speedups des Breadth-First-Ansatzes für unterschiedliche Prozessoranzahlen zusammen. Die numerischen Experimente wurden alle auf einem Sun Fire E25K-Server mit insgesamt 72 Prozessoren (1,05 GHz) und 228 GB Hauptspeicher durchgeführt. Auf diesen läuft Solaris 10 und die Version 5.6 des Sun C++-Compilers.

Man sieht deutlich, dass der Breadth-First-OpenMP-Ansatz für eine moderate Anzahl von Prozessoren gute Speedups liefert, obwohl er extrem einfach zu implementieren ist. Ausführliche Experimente deuten allerdings darauf hin, dass diese Implementierung bei einer weiteren Erhöhung der Prozessorzahl nicht mehr zu einem linearen Speedup führen wird.

Für die Parallelisierung haben wir in Algorithmus 4.2 die Scheduling-Strategie `schedule(static,1)` gewählt. Da die Arbeit, die für eine Box investiert werden muss, sehr stark variieren kann, reicht diese Scheduling-Strategie nicht aus, die Last bei einer größeren Anzahl von Threads grob zu balancieren: Es kann passieren, dass ein Thread ausschließlich schwierige Boxen bearbeitet (d.h. alle implementierten Kontraktionsverfahren müssen angewendet werden) und alle anderen Threads ihre Boxen nach dem *direkten Auswertungstest* verwerfen können. Die Strategie `dynamic` versucht, den Lastausgleich zu verbessern. Allerdings verhindert der große Overhead dieser Strategie

Algorithmus 4.2 BREADTH-FIRST-OPENMP-ALGORITHMUS

Kommentar: Die Listen $\mathcal{L}, \mathcal{N}, \mathcal{S}$ und $\mathcal{R}$ seien im Code bereits deklariert.

```
1: #pragma omp parallel private(R), shared(L, N, S)
2:   #pragma omp single
3:     L = [x^(0)]; N = ∅;
4:   end #pragma omp single
5:   R = ∅
6:   while L ≠ ∅ do
7:     l = Länge von L
8:     #pragma omp for private([x], i, R), shared(L, N, l), schedule(static)
9:       for i = 1, ..., l do
10:        entferne die i-te Box [x] aus L {benötigt Array-Implementierung}
11:        CheckBox( [x] )
12:        if [x] ist „klein genug“ then
13:          füge [x] in R ein
14:        else
15:          unterteile [x] in zwei Teilboxen [x^(1)] und [x^(2)]
16:          #pragma omp critical
17:            füge [x^(1)] und [x^(2)] in N ein
18:          end #pragma omp critical
19:        end if
20:      end for
21:    end #pragma omp for
22:    #pragma omp single
23:      vertausche die Listen L und N
24:    end #pragma omp single
25:  end while
26:  l = Länge von R
27:  for i = 1, ..., l do {füge private Lösungslisten zusammen}
28:    #pragma omp critical
29:      füge i-te Box aus R in S ein {Zugriff auf R braucht nicht geschützt werden}
30:    end #pragma omp critical
31:  end for
32: end #pragma omp parallel
```

exzellente Speedups. Tatsächlich konnte der Autor in keinem der umfangreichen Tests eine Verbesserung der Laufzeit mit Hilfe der Strategie `dynamic` feststellen.

4.4.2 Der Task-Queue-Ansatz

Wie wir soeben gesehen haben, liefert die Parallelisierung des Breadth-First-Ansatzes—trotz der sehr einfachen Implementierung—gute Speedups für eine moderate Anzahl von Prozessoren. Ausführliche Experimente haben gezeigt, dass diese Implementierung bei einer weiteren Erhöhung der Prozessorzahl nicht mehr zu einem linearen Speedup führt.

Neben den oben bereits erwähnten Problemen hinsichtlich der Scheduling-Strategie liegt das Hauptproblem unseres ersten OpenMP-Ansatzes in der Tatsache, dass wir erst Boxen aus der Liste $\mathcal{N}$ bearbeiten können, wenn die Arbeit für die Elemente aus Liste $\mathcal{L}$ vollständig beendet ist. Bei einer höheren Prozessorzahl und einer kurzen Arbeitsliste $\mathcal{L}$ werden dadurch, unabhängig von der gewählten Scheduling-Strategie, zufriedenstellende Speedups unmöglich gemacht.

Bis jetzt haben wir ausschließlich die Parallelisierung einer for-Schleife kennengelernt, um die Arbeit innerhalb einer parallelen Region unter den verschiedenen Threads zu verteilen. Einige Compiler, wie der KAP/Pro Toolset's GuideC++ [32] und der Intel C++-Compiler [33], bieten eine neue Möglichkeit, die darauf ausgelegt ist, die Probleme des Breadth-First-Ansatzes zu umgehen. Bei diesen Compilern ist man in der Lage, eine so genannte *Taskq* innerhalb einer parallelen Region zu definieren, die verschiedene Tasks enthält. Zur Laufzeit wird dann die Ausführung dieser Tasks den aktiven Threads *asynchron* zugeteilt. Leider ist dieses Taskq-Konzept noch nicht im neuesten OpenMP-Standard [57] integriert und wird deshalb auch nicht vom Sun C++-Compiler unterstützt. Der Sun C++-Compiler ist momentan der einzige (dem Autor) zur Verfügung stehende Compiler, der die Parallelisierung mit OpenMP unterstützt. Allerdings wurde dieses exzellente Konzept von Dieter an Mey [2] mit Hilfe der Standard-OpenMP-Direktiven für einen Code zur adaptiven Integration „nachgebildet". Bei genauerer Betrachtung sieht man, dass dieses Konzept, welches wir im Folgenden stets als *Task-Queue-Konzept* bezeichnen, sehr einfach in SONIC integriert werden kann.

Zur Parallelisierung von Algorithmus 2.13 muss demnach jeder Zugriff auf die Arbeitsliste $\mathcal{L}$ (in diesem Fall auch Task-Queue genannt) durch kritische Regionen geschützt werden. Weiterhin dürfen unbeschäftigte Prozessoren nicht ohne weiteres die Schleife verlassen und am Ende der parallelen Region warten, falls die aktuelle Arbeitsliste weniger Einträge als Threads hat. Offensichtlich tritt diese Situation zumindest am Anfang der Berechnung auf, da für p wartende Threads nur die Startbox zu Verfügung steht. Aus diesem Grund wird in den Algorithmus eine zusätzliche Variable `busy` integriert, die die Anzahl der Threads zählt, die aktuell eine Box bearbeiten. Ein Prozessor darf die Schleife also nur verlassen, wenn dieser Zähler auf Null gesetzt wurde und die eigene *lokale* Arbeit getan ist. Der daraus resultierende Algorithmus ist in Alg. 4.3 dargestellt.

Tabelle 4.2 fasst die Speedups der Task-Queue-Variante für unterschiedliche Prozessorzahlen zusammen. Die numerischen Experimente wurden dabei auf dem gleichen Sun Fire-Server wie bei dem Breadth-First-Ansatz aus dem vorigen Abschnitt durchgeführt. Die Daten zeigen, dass das neue Task-Queue-Konzept dem alten Breadt-First-Ansatz überlegen ist. Weiterhin zeigt dieser Ansatz für $p \leq 16$ Prozessoren einen exzellenten Speedup und arbeitet immer noch gut, allerdings nicht perfekt, für $p = 32$ Prozessoren.

Der Vorteil des Task-Queue-Konzeptes liegt eindeutig in der Tatsache, dass dieser Algorithmus *automatisch* die Last gleichmäßig auf die verschiedenen Prozessoren verteilt, falls die Kosten für die Bearbeitung einer Box signifikant variieren. Aus diesem Grund hängt die Skalierbarkeit nur zu einem kleinen Teil von den Kontraktionsverfahren in Relation zum direkten Auswertungstest ab. Die Skalierbarkeit wird hier prinzipiell nur vom Overhead für den Queue-Mechanismus bestimmt, der in der Regel aber

Algorithmus 4.3 Der Task-Queue-Ansatz

```
1: L = [x^(0)]
2: busy = 0
3: #pragma omp parallel shared( L, R, busy ), private( [x], idle, ready )
4:   idle = true
5:   ready = false
6:   for (;;) do
7:     #pragma omp critical
8:       if L = ∅ then
9:         if idle = false then
10:          idle = true; busy = busy - 1;
11:        end if
12:        if busy = 0 then
13:          ready = true;
14:        end if
15:      else
16:        entnehme eine Box [x] aus der Liste L
17:        if idle = true then
18:          idle = false; busy = busy + 1;
19:        end if
20:      end if
21:    end #pragma omp critical
22:    if idle = true then
23:      if ready = true then
24:        exit
25:      end if
26:    else
27:      wende Beschleuniger für Box [x] an
28:      if [x] kann nicht verworfen werden then
29:        if [x] ist „klein genug“ then
30:          #pragma omp critical
31:            füge [x] in Liste der möglichen Lösungen R ein
32:          end #pragma omp critical
33:        else
34:          unterteile [x] in zwei Teilboxen [x^(1)] und [x^(2)]
35:          #pragma omp critical
36:            füge [x^(1)] und [x^(2)] in L ein
37:          end #pragma omp critical
38:        end if
39:      end if
40:    end if
41:  end for
42: end #pragma omp parallel
```

System	$p =$	2	4	8	16	32
Robotics		1.98	3.89	7.59	11.87	15.68
Reactor		1.97	3.91	7.27	12.52	18.20
Hansen		1.99	3.91	7.64	13.71	19.11
Brent		1.98	3.90	7.67	13.36	23.54
Eco		1.99	3.94	7.80	14.06	24.14
Griewank		2.01	3.99	7.88	14.51	27.16
Human heart dipole		1.99	4.00	7.91	15.11	29.65
Design problem		1.99	3.98	7.85	15.42	30.26

Tabelle 4.2: Speedups des neuen Task-Queue-Ansatzes für p Prozessoren.

eher klein ist.

Die Skalierbarkeit von Algorithmus 4.3 kann sogar noch weiter verbessert werden. Es ist offensichtlich nicht nötig, in Zeile 37 *beide* Boxen in die Liste $\mathcal{L}$ einzufügen. Fügt man ausschließlich die Box $[\mathbf{x}^{(1)}]$ in $\mathcal{L}$ ein und arbeitet unmittelbar mit der Box $[\mathbf{x}^{(2)}]$ weiter, so wird weniger Zeit in einer kritischen Region benötigt. Diese Vorgehensweise bietet sich offensichtlich auch beim Breadth-First-OpenMP-Algorithmus an. Grundsätzlich kann man davon ausgehen, dass die Skalierbarkeit eines Algorithmus umso besser ist, je weniger Zeit in einer kritischen Region verbracht wird. Da die aktuelle Strategie aber sehr gute Speedups liefert und mit der Parallelisierung für verteilten Speicher eine noch leistungsfähigere Version zur Verfügung steht, haben wir auf die Implementierung dieser Variante verzichtet.

4.4.3 Parallelisierung mit MPI

Steht kein Rechner mit gemeinsamem Speicher zur Verfügung oder benötigen wir für realistische Probleme eine noch höhere Anzahl von Prozessoren, so müssen wir eine MPI-basierte Implementierung in SONIC integrieren, um die Skalierbarkeit weiter zu erhöhen.

Die gewählte Implementierung basiert dabei auf dem hinreichend bekannten *Manager-Bearbeiter-Ansatz*. Dies bedeutet, dass ein Manager-Prozess ausschließlich für die Verteilung der Boxen auf die verschiedenen Prozessoren zuständig ist. Die $p-1$ Bearbeiter besitzen eigene private Teillisten und bearbeiten alle Boxen, die der Manager ihnen schickt. Der Manager übernimmt keine Bearbeiter-Funktion, da dieser dann sehr leicht zu einem *bottleneck* werden kann: Bei Verwendung einer größeren Prozessorzahl kann es passieren, dass viele Prozessoren auf neue Boxen warten müssen, während der Manager eigene Boxen bearbeitet. Geht man davon aus, dass das Programm auf einem Prozessor in den Speicher passt, so kann mit diesem Manager-Bearbeiter-Ansatz also maximal ein Speedup von $p-1$ erreicht werden.

Jeder Bearbeiter besitzt zwei private Teillisten $\mathcal{L}$ und $\mathcal{N}$, die beide mit der leeren Menge initialisiert werden. Schickt nun der Manager Boxen an einen Bearbeiter, so werden diese stets in die Liste $\mathcal{L}$ eingefügt. Anschließend beginnt der Bearbeiter damit, der

Algorithmus 4.4 CODE FÜR DIE BEARBEITER (Gleichungssysteme)

```
 1: ready = false; S = ∅
 2: while ready != true do
 3:   L = N = ∅
 4:   RECEIVE(manager, box-tag, box-array, size)
 5:   if size > 0 then
 6:     füge erhaltene Boxen in L ein
 7:     for alle Boxen [x] ∈ L do
 8:       CHECKBOX([x], L, N, S)
 9:       if [x] kann nicht verworfen werden then
10:         if [x] ist „klein genug“ then
11:           füge [x] in N ein
12:         else
13:           unterteile [x] in zwei Teilboxen [x^(1)] und [x^(2)]
14:           füge [x^(1)] und [x^(2)] in N ein
15:         end if
16:       end if
17:     end for
18:     speichere alle Elemente aus N in box-array {Anzahl sei size}
19:     SEND(manager, box-tag, box-array, size)
20:   else
21:     ready = true
22:   end if
23: end while
24: speichere alle Elemente aus S in box-array {Anzahl sei size}
25: SEND(manager, solution-tag, box-array, size)
```

Reihe nach alle Boxen aus seiner Arbeitsliste $\mathcal{L}$ abzuarbeiten. Entstehen neue Boxen durch Bisektion oder andere Kontraktionsverfahren, so werden diese neuen Boxen in die Liste $\mathcal{N}$ eingefügt. Hat der Bearbeiter die Liste $\mathcal{L}$ komplett abgearbeitet, so schickt er zunächst alle Boxen aus der Liste $\mathcal{N}$ an den Manager und wartet dann auf weitere Boxen oder weitere Instruktionen. Der Algorithmus ist schematisch in Alg. 4.4 dargestellt. Es ist dabei zu beachten, dass eine neue MPI-Datenstruktur für die Boxen zur Verwendung bei den Kommunikationsroutinen entworfen wurde, die zusätzliche Informationen beinhaltet. Diese Datenstruktur enthält zum Beispiel die Priorität einer Box, die für die Verwaltung der Box in einem Heap benötigt wird.

Der Manager verwaltet eine einzige globale Liste $\mathcal{Q}$, in die zu Beginn der Berechnung die Startbox $[\mathbf{x}^{(0)}]$ eingefügt wird. Im Laufe des Algorithmus durchläuft der Manager immer wieder drei Haupt-Phasen. In den ersten Phase wird überprüft, ob ein Bearbeiter neue Boxen geschickt hat. Ist dies der Fall, so werden diese Boxen in die Liste $\mathcal{Q}$ eingefügt. Anschließend wird getestet, ob es unbeschäftigte Bearbeiter gibt. Ist dies der Fall, so werden die vorhandenen Boxen gleichmäßig auf diese Prozessoren verteilt. Dabei ist zu beachten, dass man die Boxen nicht nur *quantitativ*, sondern auch *qualitativ* gut verteilt: Bei der Verwendung von Exclusion-Regions oder für das Feature *Termina-*

Algorithmus 4.5 CODE FÜR DEN MANAGER (Gleichungssysteme)

```
 1: Q = [x^(0)]; ready = false; idle = p − 1, S = ∅
 2: while ready != true do
 3:   if MESSAGE-WAIT(?, box-tag) then
 4:     RECEIVE(?, box-tag, box-array, size)
 5:     füge diese (size) Boxen in Q ein
 6:     idle = idle + 1
 7:   end if
 8:   if Q ≠ ∅ and idle > 0 then
 9:     for i = 1, ..., idle do {unbeschäftigter Prozessor hat die Nummer p_i}
10:       l = min{δ, (Länge von Q) / idle }
11:       speichere l Boxen in box-array
12:       SEND(p_i, box-tag, box-array, l)
13:       idle = idle - 1
14:     end for
15:   end if
16:   if idle = p − 1 and Q = ∅ then
17:     for i = 1, ..., p − 1 do
18:       SEND(i, box-tag, box-array, 0)
19:     end for
20:   end if
21: end while
22: for i = 1, ..., p − 1 do
23:   RECEIVE(i, solution-tag, box-array, size)
24:   füge Boxen in S ein
25: end for
```

tionOnFirstHit erhält jede Box eine bestimmte Priorität. Würden die zu versendenden Boxen nun ausschließlich vom Anfang der Liste gewählt, so würde ein Prozessor ausschließlich auf Erfolg versprechenden Boxen arbeiten und andere Prozessoren nur wenig aussichtsreiche Boxen erhalten. Aus diesem Grund werden das 1., $(p+1)$-te, usw. Element des Heaps an den Bearbeiter geschickt. Diese Prozedur kann bei der Verwendung einer einfachen Liste natürlich entfallen. Um eine grobe (quantitative) Lastverteilung zu garantieren, wird die Anzahl der Boxen, die maximal gesendet werden dürfen, auf einen Wert δ gesetzt. In den Standard-Einstellungen von SONIC wird momentan der Wert $\delta = 5$ genutzt. Sind keine weiteren Boxen in der Liste $\mathcal{Q}$ vorhanden und ist kein Bearbeiter beschäftigt, so werden in einer dritten Phase alle Bearbeiter „von ihrer Arbeit entbunden“. Dies geschieht durch das Senden einer Nachricht der Länge Null, die mit einem `box-tag` versehen wird. Anschließend müssen nur noch die lokal berechneten Lösungen von den Bearbeitern empfangen und gesammelt werden. Eine schematische Darstellung der sich daraus ergebenden Kommunikationsstruktur ist in Abbildung 4.4 zu finden.

Bei der Parallelisierung des Gleichungslösers haben wir ausschließlich Kommunikationsroutinen verwendet, die blockierend sind. Natürlich könnte man an dieser Stelle

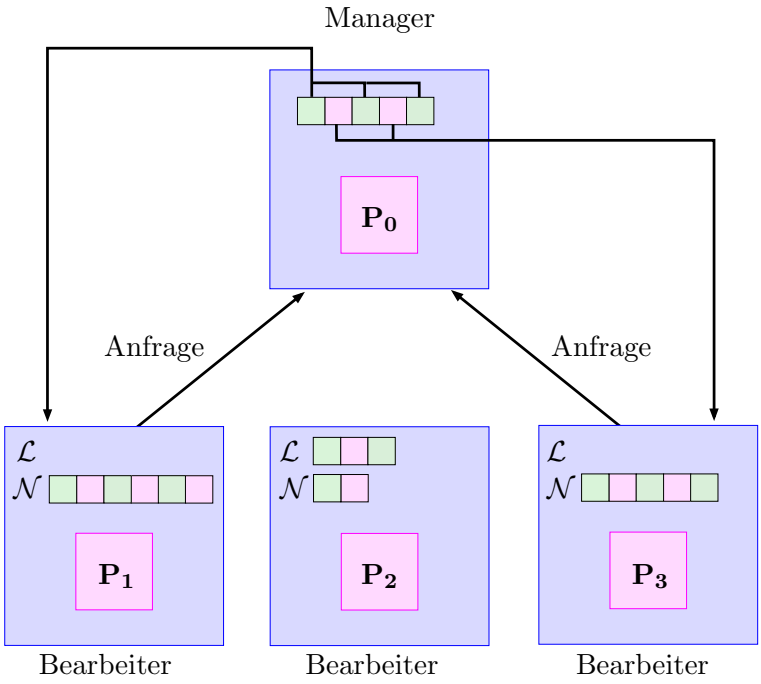

Abbildung 4.4: Ein Prozessor übernimmt die Rolle des Managers. Die restlichen Prozessoren (Bearbeiter) bearbeiten die Boxen ihrer lokalen Listen.

auch nicht-blockierende Kommunikation einsetzen. Bei genauerer Betrachtung des Codes sieht man allerdings, dass dies keine Verbesserung der Effizienz des Algorithmus zur Folge hätte, wenn *ausschließlich* die entsprechenden Kommunikationsroutinen ausgetauscht würden:

1. Der Bearbeiter sendet nur, wenn er *keine* Boxen zur Bearbeitung mehr übrig hat. Er kann seine Arbeit erst fortsetzen, wenn er neue Boxen vom Manager zugeschickt bekommt. Dieses kann aber erst geschehen, wenn der Manager die Boxen des Bearbeiters empfangen hat. Die Verwendung von nicht-blockierender Kommunikation hätte zur Folge, dass der Bearbeiter innerhalb einer Schleife immer wieder testen müsste, ob neue Boxen zur Bearbeitung geschickt wurden. Offensichtlich wäre dieses Vorgehen für das Erreichen guter Speedups eher abträglich.

2. Für das Aufteilen vorhandener Boxen auf die unbeschäftigten Bearbeiter wird ebenfalls blockierendes Senden verwendet (Alg. 4.5, Zeile 9–13). Dies wird notwendig durch die Tatsache, dass für das Senden stets das gleiche `box-array` benutzt wird. Würde man an dieser Stelle nicht-blockierendes Senden verwenden, so könnte erst wieder in das Feld geschrieben werden, wenn die Nachricht bereits *empfangen* wurde. Da in der Schleife nur noch die Anweisung `idle=idle-1` durchgeführt wird, die im Vergleich zur Kommunikation eher marginalen Aufwand darstellt, würde somit das nicht-blockierende Senden in der Praxis zu einem blockierendem Senden werden.

System	$p =$	4	8	16	32	48	64
Robotics		2.88	6.56	11.21	17.31	19.44	20.88
Reactor		2.91	6.77	13.92	22.58	28.11	32.51
Hansen		2.96	6.81	13.87	25.05	33.34	38.84
Brent		2.97	6.89	14.79	26.92	36.69	44.71
Eco		3.01	7.00	14.99	29.69	43.89	56.02
Griewank		2.99	6.99	14.89	30.37	44.36	58.34
Human heart dipole		3.00	7.02	14.96	30.74	44.75	57.57
Design problem		2.97	6.95	14.89	30.68	46.28	60.66

Tabelle 4.3: Speedups für die MPI-Version für p Prozessoren.

Eine mögliche Verbesserung der Effizienz könnte sich trotzdem durch die Verwendung asynchroner Kommunikation ergeben. Es ist denkbar, dass jeder Bearbeiter Teile seiner privaten Liste $\mathcal{N}$ asynchron an den Manager zurückschickt, wenn genügend Boxen da sind. Da, wie die Resultate im nächsten Abschnitt zeigen, der gewählte Ansatz allerdings sehr gute Speedups liefert, haben wir auf die Implementierung dieser Idee verzichtet.

Die numerischen Ergebnisse wurden diesmal auf dem *ALiCEnext* Cluster der Bergischen Universität Wuppertal (eine genaue Beschreibung des Clusters findet der interessierte Leser im Anhang) durchgeführt. Auf diesem Rechner ist ausschließlich die Nutzung der C-XSC Intervall-Bibliothek möglich. Die anderen von SONIC unterstützten Bibliotheken lassen sich nicht auf dieser 64-Bit-Architektur compilieren. Im Gegensatz zu den beiden OpenMP-Implementierungen erreichen wir nun beinahe perfekte Speedups für bis zu 64 Prozessoren. Bei einem flüchtigen Blick auf Tabelle 4.3 könnte zunächst der Eindruck entstehen, dass zumindest für die ersten drei Testbeispiele unser Ansatz versagt. Bei genauerer Betrachtung sieht man aber, dass hier das Amdahlsche Gesetz eine wichtige Rolle einnimmt. Die Ausführung von SONIC mit 64 Prozessoren ist bei diesen Beispielen bereits nach wenigen Sekunden beendet. Allerdings sind bei der Zeitmessung sowohl Vorverarbeitung als auch Verifikation enthalten, die beide nicht parallel durchgeführt werden und natürlich eine gewisse Zeit benötigen. Es ist offensichtlich, dass eine Berechnung dieser Beispiele für eine höhere Zahl von Prozessoren keinen Sinn macht. Weitere numerische Experiment zeigen, dass unsere MPI-Parallelisierung in der Tat für schwere Systeme erfolgreich ist, also für Systeme, die eine hohe Laufzeit und Boxenzahl haben.

Die gewählte MPI-Implementierung ist in der Theorie sicherlich nicht optimal, was bei einem Blick auf die Kommunikationsstrategie (Abbildung 4.4) sofort auffällt: Hat ein Bearbeiter alle seine Boxen bearbeitet, so schickt er die Boxen aus der Liste $\mathcal{N}$ an den Manager. Dieses Vorgehen kann sich aus zwei Gründen nachteilig auswirken:

1. Der Bearbeiter erhält unter Umständen im nächsten Schritt wieder die (ungefähr) gleiche Anzahl von Boxen vom Manager. In diesem Fall hat also unnötig viel Kommunikation stattgefunden. Eine einfache Anweisung vom Manager an den Bearbeiter, dass er seine Boxen weiter bearbeiten soll, hätte also völlig ausgereicht.

2. Bei extrem schweren Problemen enthält die globale Liste des Managers automatisch sehr viele Boxen. Dies kann unter Umständen zu einem Speicher-Problem führen. Eine gleichmäßige Verteilung der Boxen auf die einzelnen Prozessoren ist in diesem Fall also vorzuziehen. Ein weiteres Problem liegt darin begründet, dass einige Listenoperationen bei sehr langen Listen immer aufwändiger werden.

Um die geschilderten Probleme zu beheben, könnte der Mangager die Rolle eines *zentralen Vermittlers* übernehmen. Dieses Konzept haben wir bei der Parallelisierung des Optimierers von SONIC (siehe Abschnitt 4.5) verwendet und könnte mit wenig Aufwand auch für den nichtlinearen Gleichungslöser eingesetzt werden. Der zentrale Vermittler verwaltet nur eine kleine Zahl von Boxen selber und vermittelt nur zwischen den Prozessoren, die weitgehend unabhängig die Boxen ihrer lokalen Liste bearbeiten. Es wird so vermieden, dass die Boxen in einer zentralen Liste gespeichert werden und dass fast jede Bearbeitung einer Box mit Kommunikation—dem Senden einer Box vom Manager zum Bearbeiter—verbunden ist. Diese Strategie könnte durch Anpassung des Parameters δ in der aktuellen Implementierung von SONIC nachgeahmt werden. Je höher dieser Wert gewählt wird, desto mehr Boxen werden auf die einzelnen Bearbeiter verteilt. Allerdings hat die Erhöhung des Parameters δ zur Folge, dass kein guter Lastausgleich zwischen den Prozessoren durchgeführt wird. Da wir mit der aktuellen Implementierung hervorragende Speedups erreichen, ist die Integration eines „zentralen Vermittlers“ momentan aber nicht geplant.

Eine weitere Verbesserung könnte für alle drei präsentierten Parallelisierungen integriert werden, falls zu einem bestimmten Zeitpunkt der Algorithmen weniger Boxen als Prozessoren bzw. Threads vorhanden sind. In diesem Fall könnte eine der vorhandenen (vielversprechenden) Boxen durch Multisektion in mehr als zwei Teilboxen unterteilt werden, um *alle* Prozessoren zu beschäftigen. Dieses würde vor allem in der Startphase Sinn machen, da bereits dort nur eine einzige Box für p Prozessoren vorhanden ist. Für schwere Probleme konnte der Autor allerdings beobachten, dass in der Regel „ausreichend viele“ Boxen zur Verfügung stehen, da dann der Teil der Berechnung dominiert, bei dem die Listenlänge sehr groß ist.

4.4.4 Ein abschließender Vergleich

In dem letzten Abschnitt haben wir drei Strategien zur Parallelisierung eines ergebnisverifizierenden nichtlinearen Lösers, der auf einem Branch–and–Bound-Ansatz basiert, präsentiert und untersucht. Mit dem Breadth-First-OpenMP-Algorithmus haben wir einen Ansatz vorgestellt, der sehr leicht zu implementieren ist und gute Speedups für eine moderate Anzahl von Prozessoren liefert. Mit unserem zweiten Ansatz wurde eine Task-Queue-Strategie für Rechner mit gemeinsamem Speicher implementiert. Wir haben anhand von mehreren Testbeispielen gesehen, dass diese Stragie dem naiven OpenMP-Ansatz überlegen ist. Dieses Konzept liefert für eine mittlere Anzahl von Prozessoren exzellente Speedups. Da realistische Probleme eine sehr große Anzahl von Prozessoren benötigen, haben wir zusätzlich eine MPI-Implementierung für Rechner mit verteiltem Speicher in SONIC integriert, die auf einem Manager-Bearbeiter-Algorithmus beruht. Dieser Ansatz zeigt beinahe perfekte Speedups für eine große Anzahl von Pro-

zessoren.

In der Literatur wird häufig beschrieben, dass der Manager bei einem klassischen Mangager-Bearbeiter-Ansatz und der Verwendung vieler Prozessoren sehr leicht zum *bottleneck* werden kann. Dieser Effekt konnte bei der Verwendung von bis zu 128 Prozessoren für kein Testbeispiel festgestellt werden. Bei extrem hohen Prozessorzahlen könnte dieser Effekt allerdings auftreten. In diesem Fall könnte die Implementierung eines *zentralen Vermittlers*, den wir im folgenden Abschnitt für den Optimierer von SONIC vorstellen, helfen.

In Abbildung 4.5 haben wir repräsentative Ergebnisse für einige Testbeispiele von unterschiedlicher Größe zusammengefasst. Eine ausführliche Analyse der präsentierten Parallelisierungs-Strategien findet man in [6].

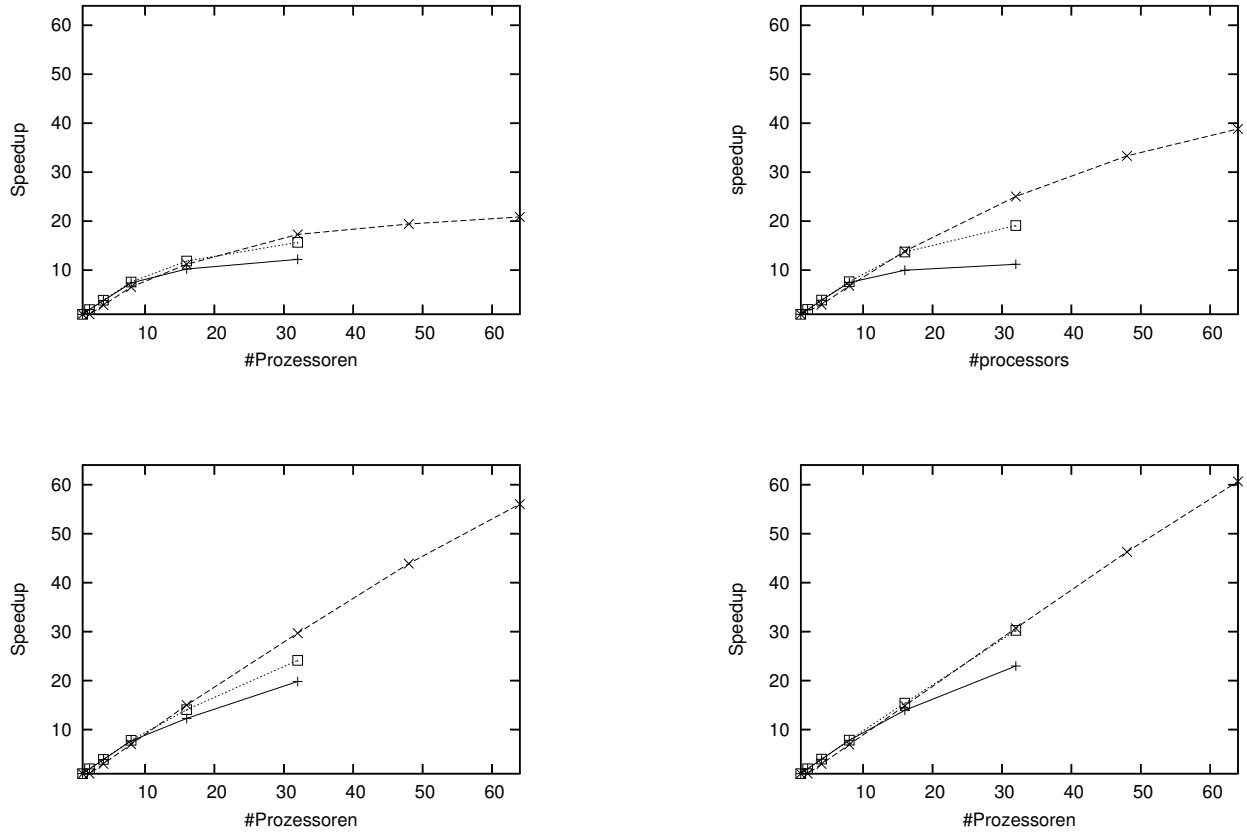

Abbildung 4.5: Speedups für vier Testprobleme: Robotics (oben links), Hansen (oben rechts), Eco (unten links) und Design problem (unten rechts) für die OpenMP-Implementierungen mit Breadth-First-Ansatz (durchgezogene Linie) und Task-Queue-Strategie (gepunktete Linie) und die MPI-Implementierung (gestrichelte Linie).

4.5 Parallelisierung des Optimierers

Die Parallelisierung des Optimierers von SONIC ist momentan ausschließlich für Rechner mit verteiltem Speicher realisiert. Weiterhin haben wir uns auf die Parallelisierung des Optimierers *ohne* Nebenbedingungen beschränkt, um die Darstellung an einigen Stellen einfacher gestalten zu können. Die Erweiterung des Codes für die Optimierung mit Nebenbedingungen ist ohne weiteres möglich.

Bei der gewählten Implementierung greifen wir auf wesentliche Teile der Parallelisierungsstrategie von Berner [8] zurück, die sich in der Praxis als erfolgreich herausgestellt hat. Da wir einige, für SONIC spezifische, Änderungen vorgenommen haben, wollen wir hier nur kurz die wesentlichen Punkte des parallelen Verfahrens beschreiben. Eine ausführliche Analyse des zu Grunde liegenden Verfahrens kann in der oben angeführten Dissertation nachgelesen werden.

Algorithmus 4.6 CODE FÜR DIE BEARBEITER (Optimierung)

Beschreibung: Heap $\mathcal{L}$ sei bereits durch Aufteilen der Startbox initialisiert. Eine obere Schranke $\tilde{f}$ sei bekannt. Lösungen werden in $\mathcal{S}$ gespeichert.

```
 1: ready = false
 2: do
 3:   while L ≠ ∅ do
 4:     f̃_old = f̃
 5:     entnehme erstes Element aus Heap L
 6:     CheckElementGOP(L, S, f̃, [x])
 7:     EMPFANGE_OBERE_SCHRANKE(f̃_old)
 8:     f̃ = min{f̃, f̃_old}
 9:     if f̃ < f̃_old then
10:       VERSENDE_OBERE_SCHRANKE(f̃)
11:     end if
12:     if nach letztem CUT-OFF-TEST wurde f̃ verbessert then
13:       führe CUT-OFF-TEST mit L durch
14:     end if
15:     BOXAUSGLEICH
16:   end while
17:   SEND(manager, request-tag, myrank, 1)
      {sende Anfrage nach Boxen an den zentralen Vermittler}
18:   RECEIVE(manager, box-tag, box-array, size)
19:   if size > 0 then
20:     füge size Boxen aus array in L ein
21:   else
22:     ready = true
23:   end if
24: while ready ≠ true
25: EMPFANGE_OBERE_SCHRANKE(f̃)
26: führe CUT-OFF-TEST mit S durch
```

Algorithmus 4.7 EMPFANGE_OBERE_SCHRANKE(f)

Beschreibung: Empfang einer möglicherweise verbesserten oberen Schranke für das globale Minimum. Der aktualisierte Wert wird in f gespeichert.

```
1: while MESSAGE-WAIT(?, min-tag) do
2:    RECEIVE(?, min-tag, x, 1)
3:    if x < f then
4:       f = x
5:    end if
6: end while
```

Um das parallele Gesamtverfahren besser darstellen zu können, teilen wir die Algorithmen wiederum jeweils in Code für den Master und die Bearbeiter auf. Dies macht vor allem Sinn, da sie sich deutlich in ihrer Aufgabenstellung voneinander unterscheiden.

In Algorithmus 4.6 gehen wir davon aus, dass die Unterteilung der Startbox bereits stattgefunden hat und die entsprechenden Boxen auf die einzelnen Listen $\mathcal{L}$ (meist als Heap implementiert) der Bearbeiter verteilt worden sind. Die Unterteilung erfolgt dabei mit Hilfe der Multisektion. Dieser Schritt wird durchgeführt, um zu garantieren, dass jeder Bearbeiter zu Beginn beschäftigt ist. Da bei der Bearbeitung von schweren Problemen in der Regel sehr lange Listen $\mathcal{L}$ entstehen, spielt dieser Schritt für die Effizienz des Verfahrens nach den Erfahrungen des Autors nur eine untergeordnete Rolle. Wir wollen deshalb an dieser Stelle auf eine genaue Betrachtung dieses Schrittes verzichten.

Ist der Heap der noch zu bearbeitenden Elemente noch nicht leer, wird in einem ersten Schritt $\tilde{f}_{\text{old}} = \tilde{f}$ der aktuelle Wert für das globale Minimum zugewiesen. Anschließend wird das erste Element aus dem Heap $\mathcal{L}$ entnommen und bearbeitet. Die dabei eventuell entstehenden Boxen werden entweder in $\mathcal{L}$ oder $\mathcal{S}$ gespeichert. Dann wird EMPFANGE_OBERE_SCHRANKE aufgerufen, wobei $\tilde{f}_{\text{old}}$ als Parameter übergeben wird. Damit kann festgestellt werden, ob die *eigene* Schranke $\tilde{f}$ an die anderen Prozessoren über VERSENDE_OBERE_SCHRANKE verschickt werden muss. Beim Versenden der

Algorithmus 4.8 VERSENDE_OBERE_SCHRANKE(f)

```
1: for i = 0, ..., p − 1, i ≠ myrank do
2:    if LAST-MESSAGE-SENT(i, min-tag) then
3:       SEND-ASYNC(i, min-tag, f̃, 1)
4:    end if
5: end for
```

oberen Schranke kann darauf verzichtet werden, an Prozessoren erneut eine Nachricht zu schicken, die noch nicht das letzte von diesem Prozessor verschickte $\tilde{f}$ empfangen haben. Diese erhalten—durch das *asynchrone* Senden der Nachricht—automatisch den richtigen Wert. Hat sich der Wert für $\tilde{f}$ verändert, so wird nun auf dem Heap ein CUT-OFF-TEST durchgeführt. Anschließend wird BOXAUSGLEICH aufgerufen. Dafür wird vom Manager ein neuer Wert für die Variable `max` empfangen und eventuell überschüssi-

Algorithmus 4.9 BOXAUSGLEICH

```
1: if LAST-MESSAGE-SENT(manager, box-tag) then
2:   empfange max vom Manager
3:   if Heap L enthält mehr als max Elemente then
4:     entnehme 2., 4., usw. Element aus L.
5:     speichere size ≤ max_send Elemente in box-array
6:     SEND-ASYNC(master, box-tag, box-array, size)
7:   end if
8: end if
```

ge Boxen an diesen geschickt, falls der lokale Heap $\mathcal{L}$ mehr als `max` Elemente besitzt. Es werden in der Regel die Hälfte der Boxen, jedoch nicht mehr als `max_send` Boxen, verschickt. In SONIC wird momentan die Standardeinstellung `max_send`=5 genutzt. Die Routinen zum Versenden und Empfangen eines neuen Wertes für `max` läuft völlig analog zum Versenden und Empfangen einer neuen oberen Schranke ab: An Prozessoren, die ihre letzte Nachricht noch nicht empfangen haben, wird keine neue geschickt. Die Aktualisierung von `max` reicht völlig aus. Ist der lokale Heap abgearbeitet, so wird eine Anfrage nach Boxen an den Manager geschickt. Wird diese Anfrage mit „Null Boxen“ beantwortet, so ist die Arbeit für den entsprechenden Bearbeiter beendet. Werden tatsächlich Boxen empfangen, so werden diese im Heap $\mathcal{L}$ gespeichert und der Algorithmus beginnt von vorne.

Der Manager übernimmt die Rolle eines *zentralen Vermittlers*. Dabei arbeitet er selber nicht auf Boxen, sondern ist ausschließlich für den Lastausgleich zwischen den

Algorithmus 4.10 CODE FÜR DEN ZENTRALEN VERMITTLER (Optimierer)

Beschreibung: Heap $\mathcal{L}$ sei bereits durch Aufteilen der Startbox initialisiert. $\mathcal{Q}$ enthält die Nummern der Bearbeiter, die keine Boxen bearbeiten. Eine Schranke $\tilde{f}$ für das globale Minimum liege vor.

```
1: max = ∞; Q = ∅
2: while Länge von Q < p − 1 do
3:   SAMMLE_ANFRAGEN(Q)
4:   EMPFANGE_OBERE_SCHRANKE(f̃)
5:   EMPFANGE_BOXEN(L)
6:   EMPFANGE_OBERE_SCHRANKE(f̃)
7:   BEANTWORTE_ANFRAGEN(Q, L)
8:   if nach letztem CUT-OFF-TEST wurde f̃ verbessert then
9:     führe CUT-OFF-TEST mit L durch
10:  end if
11:  BESTIMME_NEUES_MAX
12: end while
13: for i = 0, ..., p − 1, i ≠ manager do
14:   SEND(i, box-tag, box-array, 0)
15: end for
```

unterschiedlichen Bearbeitern zuständig. Nach der Initialisierung der Variablen `max` wird eine Warteschlange $\mathcal{Q}$ angelegt, die die Nummern der Bearbeiter enthält, die momentan keine Boxen bearbeiten. Durch die Startphase ist sichergestellt, dass jeder Prozessor zumindest eine Box in seinem Heap hält. Dadurch kann $\mathcal{Q}$ als leere Warteschlange initialisiert werden.

Solange die Bearbeitung des Problems noch nicht abgeschlossen ist, wird die folgende Hauptschleife durchlaufen. Zunächst werden alle Anfragen gesammelt, d.h. alle mit einem `request-Tag` versehenen Nachrichten empfangen, und die Nummer des entsprechenden Absenders in $\mathcal{Q}$ gespeichert. Bevor man diese Anfragen beantwortet, wird zunächst überprüft, ob es einen besseren Wert für $\tilde{f}$ gibt. Anschließend werden über die Routine EMPFANGE_BOXEN nicht nur Boxen von anderen Prozessoren empfangen, sondern auch an wartende Bearbeiter verschickt. Die in EMPFANGE_BOXEN benötigte

Algorithmus 4.11 EMPFANGE_BOXEN($\mathcal{L}$)

```
 1: old-max = max
 2: while MESSAGE-WAIT(?, box-tag) do
 3:    RECEIVE(?, box-tag, box-array, ?)
 4:    size = Anzahl der empfangenen Boxen
 5:    if size+ Länge von L ≥ Länge von Q then
 6:       max = ∞
 7:    else
 8:       max = old-max
 9:    end if
10:    if max hat sich sei letztem Verschicken verändert then
11:       VERSENDE_MAX
12:    end if
13:    füge empfangene Boxen in L ein
14:    if MESSAGE-WAIT(?, box-tag) then
15:       BEANTWORTE_ANFRAGEN(Q, L)
16:       SAMMLE_ANFRAGEN(Q)
17:    end if
18: end while
```

Routine VERSENDE_MAX läuft dabei analog zu VERSENDE_OBERE_SCHRANKE ab: Es ist durch das asynchrone Versenden nicht nötig, eine erneute Nachricht zu schicken, die Aktualisierung der Variablen max genügt völlig. Nach dem Empfang der Boxen wird in Algorithmus 4.10 erneut EMPFANGE_OBERE_SCHRANKE aufgerufen und bei Bedarf der CUT-OFF-TEST durchgeführt. Anschließend werden die restlichen, noch anstehenden, Anfragen beantwortet. Abschließend wird in Alg. 4.10 die Schranke max geeignet angepasst und eventuell an die Bearbeiter versendet, was durch die Routine BESTIMME_NEUES_MAX realisiert wird. Für das geeignete Anpassen der Schranke `max` existieren in der Literatur einige Realisierungsmöglichkeiten. Umfangreiche Tests in [8] haben gezeigt, dass die gewählte Strategie durchaus sinnvoll ist. Dort wird für bestimmte „Notfallsituationen“ eine weitere Möglichkeit präsentiert. Diese haben wir momentan nicht integriert, da unsere Erfahrungen gezeigt haben, dass für schwere Probleme die

Algorithmus 4.12 BEANTWORTE_ANFRAGEN($\mathcal{Q}, \mathcal{L}$)

```
1: while L ≠ ∅ and Q ≠ ∅ do
2:    entnehme erstes Element [x] aus Heap L
3:    if inf([f]([x̌])) > f̃ then
4:       L = ∅
5:    else
6:       entnehme Nummer i des am längsten wartenden Bearbeiters aus Q
7:    end if
8:    SEND(i, box-tag, [x], 1)
9: end while
```

Variante aus Alg. 4.13 völlig ausreicht.

Algorithmus 4.13 BESTIMME_NEUES_MAX

```
1: if Länge von L > p/4 then
2:    max = ∞
3: else
4:    max = Länge von L
5: end if
6: if max wurde um mehr als 1 verändert then
7:    VERSENDE_MAX
8: end if
```

In Abbildung 4.6 haben wir die sich aus dem Algorithmus ergebende Kommunikationsstruktur dargestellt.

4.5.1 Numerische Ergebnisse

Die numerischen Ergebnisse wurden wiederum auf dem *ALiCEnext* Cluster der Bergischen Universität Wuppertal mit 1024 AMD 1.8 GHz Opteron-Prozessoren (2 GB Hauptspeicher, 64-Bit) durchgeführt. Um die Effizienz des parallelen Verfahrens zu ermitteln, wurde dieses für die betrachteten Probleme auf 4, 8, 16, 32 und 64 Prozessoren ausgeführt. Die resultierenden Speedups sind in Tabelle 4.4 angegeben. Um möglichst unabhängig von der Auslastung der Maschine zu sein, haben wir jeden Test mindestens drei mal durchgeführt und den Mittelwert der Laufzeiten gebildet.

Den bei den Experimenten erreichten Speedup haben wir ebenfalls grafisch dargestellt. Dabei haben wir in Abbildung 4.7 zunächst die Testprobleme, deren Lösung bereits mit weniger Aufwand gefunden werden kann, zusammengestellt. Man sieht bereits hier, dass trotz der geringen Laufzeit des seriellen Verfahrens ein guter Speedup für bis zu 32 Prozessoren erreicht werden kann. Bei Hinzunahme von mehr Prozessoren nimmt allerdings die Effizienz des Verfahrens teilweise deutlich ab. In diesen Fällen hat der Overhead des parallelen Verfahrens einen größeren Anteil am gesamten Ablauf. Viele Prozessoren arbeiten dann auf wenig vielversprechenden Boxen. Die Beobachtung des Autors, ist allerdings, dass die Effizienz des Verfahrens deutlich zunimmt,

System	$p =$	4	8	16	32	64
Hansen		3.76	8.04	13.99	24.45	30.91
Levy		3.89	7.99	13.56	23.93	35.76
HM4		3.97	8.15	17.33	28.31	45.18
Kowalik		4.01	8.24	17.73	30.42	52.71
Ratz		3.97	9.82	18.77	34.26	58.11
Shubert		3.98	9.99	18.71	35.14	60.09
Siirola		3.99	9.92	19.64	36.62	62.71
Womersley		4.00	8.99	18.46	34.25	59.84
Tibken		4.29	11.12	22.14	39.66	66.52

Tabelle 4.4: Speedups für p Prozessoren.

wenn man schwerere Testbeispiele—also Systeme, die auch in der Praxis tatsächlich eine Parallelisierung benötigen—betrachtet. Diese Systeme haben wir in Abbildung 4.8 zusammengefasst. Man sieht deutlich, dass wir mit der gewählten Implementierung sehr gute Speedups für bis zu 64 Prozessoren verzeichnen können. Durchgeführte Untersuchungen haben gezeigt, dass sich schwere Probleme in der Regel dem Verlauf der Kurven in Abbildung 4.8 anpassen.

Der aufmerksame Leser hat sich sicherlich bereits gefragt, warum die Parallelisierung an einigen Stellen überlinearen Speedup, also $S(p) > p$, erreicht, da man im Allgemeinen

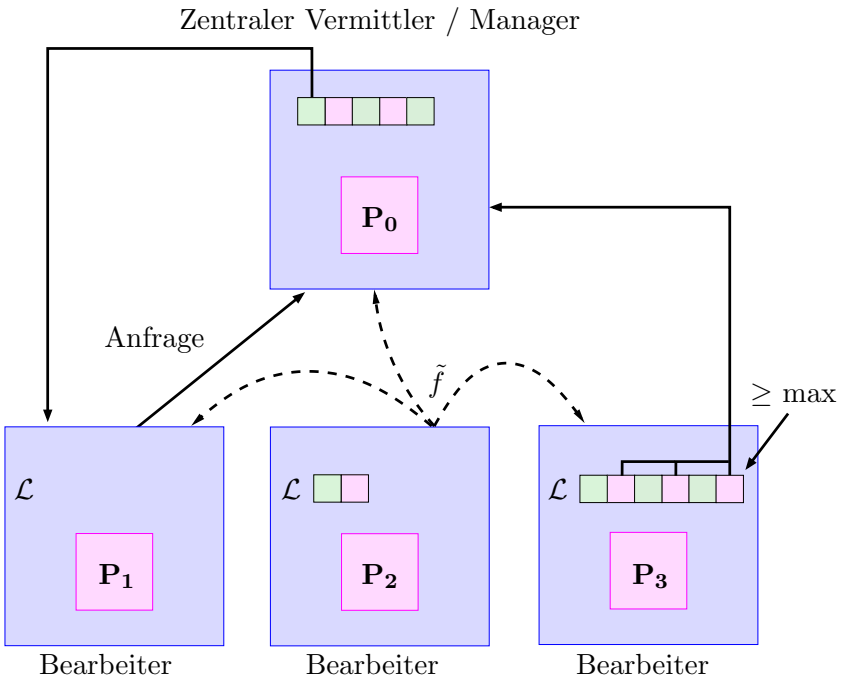

Abbildung 4.6: Ein Prozessor übernimmt die Rolle des zentralen Vermittlers (Manager). Die restlichen Prozessoren (Bearbeiter) bearbeiten die Boxen ihrer lokalen Listen.

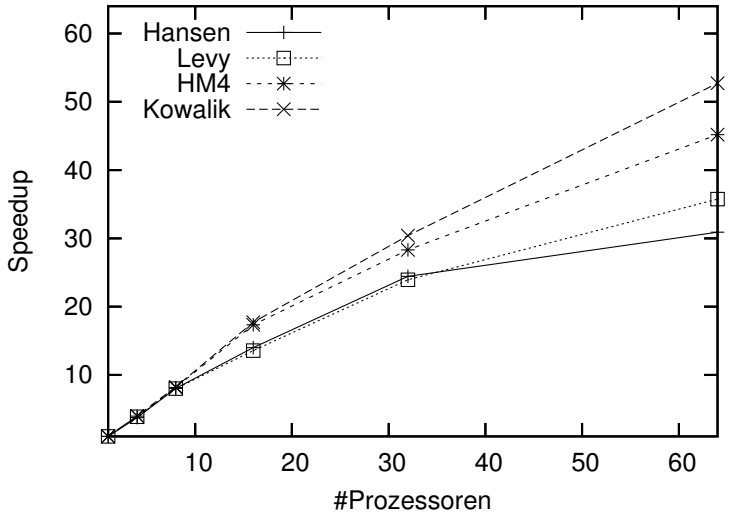

Abbildung 4.7: Erreichter Speedup für p Prozessoren für vier Probleme mit geringer Laufzeit des seriellen Verfahrens

erwartet, dass $1 \leq S(p) \leq p$ gilt. Diese Tatsache ist sicherlich darauf zurükzuführen, dass an einigen Stellen durch die Verwendung einer größeren Prozessorzahl schneller eine bessere obere Schranke für das globale Minimum gefunden wird und damit mehr Boxen bereits frühzeitig aufgrund des Cut-Off-Tests verworfen werden können. In der Literatur werden oft Parallelisierungsansätze vorgestellt und für diese in numerischen Experimenten gezeigt, dass zum Beispiel für 32 Prozessoren ein Speedup von über 150 erzielt wird. Nach Meinung des Autors sind diese Untersuchungen aber nicht „fair“, da mit hoher Wahrscheinlichkeit ein zu ineffizienter *serieller* Algorithmus verwendet wird (vergleiche auch [53]). In der Regel sollte bei einer guten Parallelisierung kein wesentlicher überlinear Speedup zu erwarten sein.

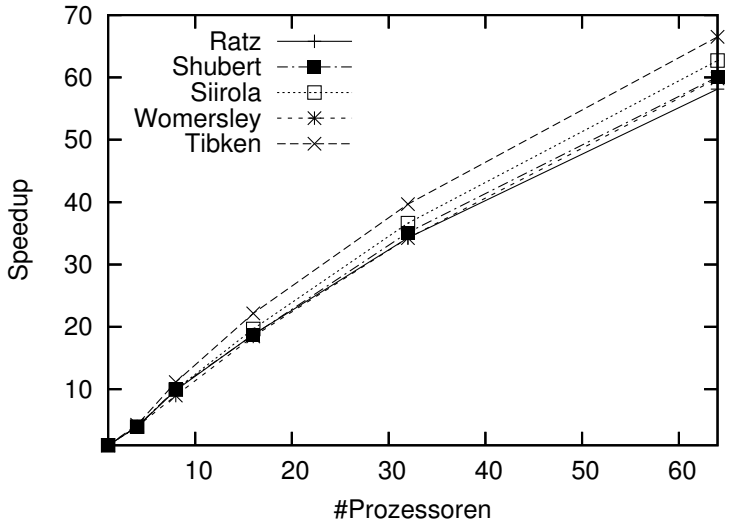

Abbildung 4.8: Erreichter Speedup für p Prozessoren für vier schwerere Probleme

Kapitel 5

Konzept und Gesamtaufbau von SONIC

Wie bereits in der Motivation erwähnt, steht der Name SONIC für *Solver and Optimizer for Nonlinear Problems based on Interval Computation*. SONIC bietet also die Möglichkeit, nichtlineare Gleichungssysteme und Optimierungsprobleme mit Hilfe von verifizierenden Algorithmen zu lösen. Das Programm-Paket wurde in der Programmiersprache C++ von den Autoren Prof. Dr. Bruno Lang, Thomas Beelitz und Paul Willems implementiert, und umfasst zum aktuellen Zeitpunkt ungefähr 35 000 Zeilen Quellcode. Der konzeptuelle Gesamtaufbau von SONIC ist schematisch in Abbildung 5.1 dargestellt. Man sieht, dass das Programm-Pakte grob in sechs Module unterteilt werden kann.

Im Modul **Vorverarbeitung** wird das System aufbereitet, welches später dem verifizierten Gleichungslöser bzw. Optimierer übergeben wird. Die Vorverarbeitung beinhaltet die folgenden Punkte:

- Einlesen der Problem-Spezifikation (Namen von Parametern und Variablen sowie deren Wertebereich, (Un-)Gleichungen für das System, evtl. Singularitäts-Typen, etc.),
- Generierung von Ableitungstermen durch symbolisches Differenzieren und evtl. Aufstellen eines neuen Systems durch Erweiterung des Originalsystems,
- Termoptimierung für System- und Ableitungsterme,
- Einlesen der Einstellungen (werden keine keine Einstellungen angegeben, so werden die Standardeinstellungen verwendet).

Eine wichtige Eigenschaft des Programm-Paketes beteht darin, dass bestimmte Einstellungen (Verwendung von Verfahren, etc.), die Vereinfachungsregeln für den integrierten, regelbasierten Termoptimierer und das zu lösende Problem in Textdateien gespeichert werden, die von SONIC eingelesen und geparst werden. Dies hat den entscheidenden Vorteil, dass auch für unterschiedliche Systeme das Programm-Paket nicht *erneut* compiliert und gelinkt werden muss. Um ein größtmögliches Maß an Modularität zu gewährleisten, können dabei beinahe alle integrierten Features unabhängig

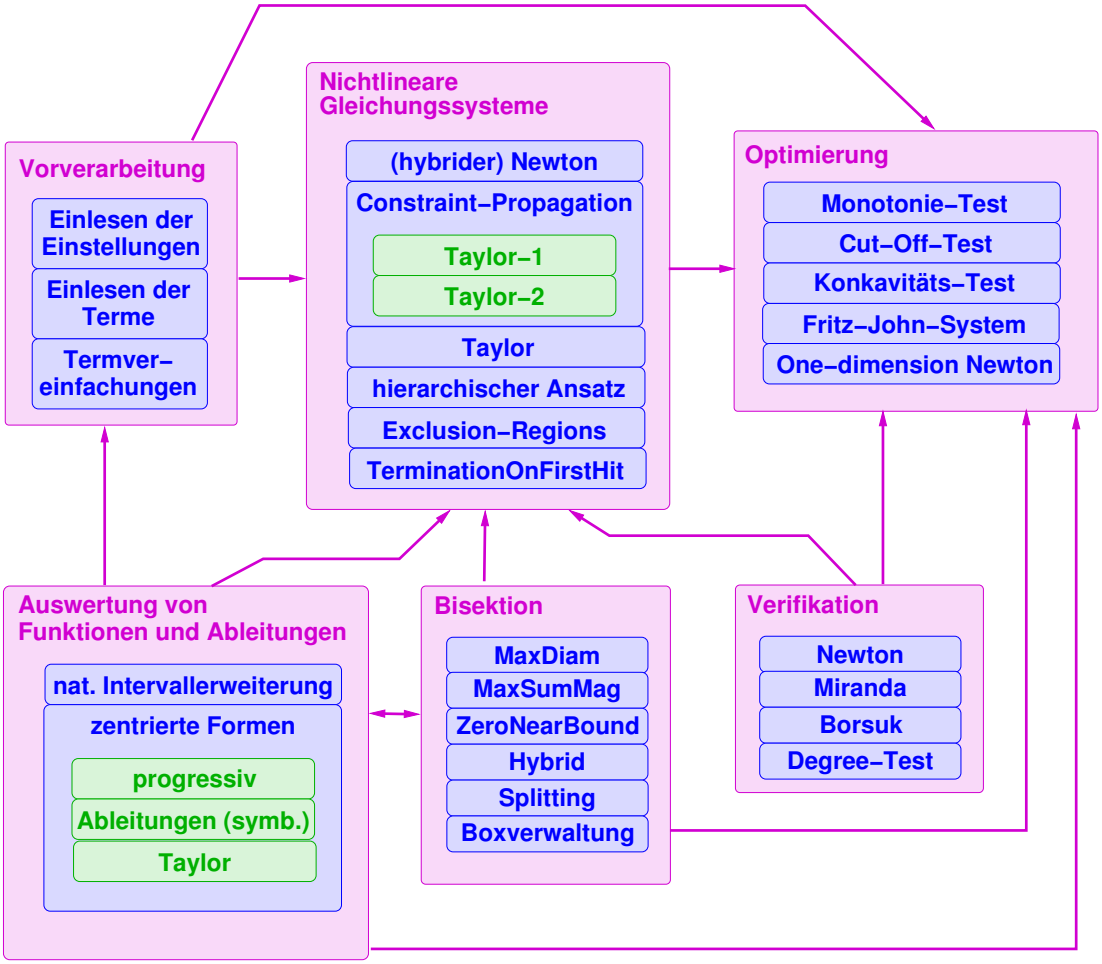

Abbildung 5.1: Die grundlegende Struktur von SONIC

voneinander ein- bzw. ausgeschaltet werden. Eine genaue Beschreibung des Aufbaus dieser Dateien findet der interessierte Leser in [62, 4]. Die zu verwendenden Kontroll- und Eingabedateien lassen sich über Kommandozeilenparameter einstellen.

In dieser Arbeit wurden einige Techniken vorgestellt, die teilweise eine umfangreiche Initialisierung in der Vorverarbeitung von SONIC erfordern. Der Zeitaufwand für die Vorverarbeitung ist aber—außer bei sehr kleinen Problemen—niemals signifikant und liegt im „vernachlässigbaren“ Bereich (in der Regel unter einer Sekunde).

Das Modul **Auswertung** stellt verschiedene Routinen zur Auswertungen der Funktionen und Ableitungen zur Verfügung. Die integrierten Routinen wurden ausführlich in Kapitel 1 beschrieben. Auch auf die Beschreibung der weiteren Module soll an dieser Stelle verzichtet werden, da sie bereits ausführlich in den vorigen Kapiteln besprochen wurden.

5.1 Intervall-Bibliotheken

Bei dem Design von SONIC haben wir darauf Wert gelegt, dass wir uns nicht auf eine bestimmte Implementierung einer Intervall-Bibliothek festlegen. Aus diesem Grund nutzen wir so genannten generischen Intervall-Code, um diesen mit Hilfe des C-Präprozessors auf die zur Verfügung stehenden Intervall-Bibliotheken abzubilden. Momentan sind in SONIC die Bibliotheken von C-XSC, FILIB++ und Sun integriert, die wir im Folgenden kurz beschreiben werden.

C-XSC

C-XSC ist ein hervorragendes Tool für die Entwicklung von numerischen Algorithmen, die höchste Genauigkeit und automatisch verifizierte Ergebnisse liefern. Es wird eine große Zahl vordefinierter numerischer Datentypen und Operatoren von höchster Genauigkeit zur Verfügung gestellt (z.B. das exakte Skalarprodukt). Die Datentypen sind als C++-Klassen implementiert. Das C-XSC-Paket ist für alle Rechner mit einem C++-Compiler, der den C++-Standard unterstützt, erhältlich. Die Sourcen dieses Tools sind frei erhältlich. Neben der bereits erwähnten hoch-genauen Intervall-Bibliothek hält das Paket z.B. auch Möglichkeiten für das Automatische Differenzieren, die Lösung von globalen Optimierungsproblemen und nichtlinearen Gleichungssystemen bereit. In dem mitgelieferten Paket *Toolbox* ist eine Klasse für unbeschränkte Intervalle implementiert, die allerdings „Exceptions wirft“, falls z.B. eine leeres Intervall während einer Berechnung entsteht. Die wichtigsten Eigenschaften von C-XSC werden in [28] zusammengefasst. Abschließend sei an dieser Stelle erwähnt, dass C-XSC momentan die *einzige* unterstützte Intervall-Bibliothek ist, die auf 64Bit-Architekturen übersetzt werden kann. Sie ist aus diesem Grund für SONIC unersetzlich geworden, da nur mit ihr eine Parallelisierung des Sourcecodes mit MPI auf der *ALiCEnext* möglich ist.

SUN Intervall-Arithmetik

Mit dem aktuellen Sun Forte C++-Compiler (Version 5.6) wird zusätzlich eine eigene Intervall-Bibliothek geliefert. Im Gegensatz zu C-XSC und filib++ ist dieser Compiler allerdings nicht frei erhältlich, sondern ein kommerzielles Produkt. Neben dem normalen Intervall-Datentyp ist auch hier eine erweiterte Intervall-Arithmetik integriert. Die Berechnung von Intervall-Operationen kann durch so genannte *Containment-Sets ohne* Exceptions durchgeführt werden. Eine genaue Beschreibung der implementierten Intervall-Funktionen und Operatoren findet der interessiert Leser in [64]. Der wesentliche Nachteil ist, dass die Intervall-Bibliothek offensichtlich nur auf Sun-Rechnern eingesetzt werden kann. Ein weiterer Nachteil besteht darin, dass einige (wünschenswerte) Methoden und Funktionen nicht implementiert sind.

filib++

filib++ ist eine Erweiterung der Intervall-Bibliothek *filib*, die ursprünglich an der Universität Karlsruhe entwickelt wurde. Das wichtigste Ziel von *filib* war die schnelle Berechnung von garantierten Schranken für Intervall-Versionen einer umfassenden Reihe von Elementarfunktionen. Die Intervall-Bibliothek *filib++* erweitert diese Bibliothek in zwei Aspekten. Zunächst wird durch den *erweiterten*

Modus die Möglichkeit eröffnet, Intervall-Operationen ohne Exceptions zu berechnen. Weiterhin werden Templates und Traits-Klassen genutzt, um eine effiziente und portable Bibliothek zu erhalten, die dem aktuellen C++ Standard entspricht. Eine ausführliche Beschreibung der Intervall-Bibliothek filib++ findet man in [29]. *filib++* ist frei erhältlich und kann auf den unterschiedlichsten Plattformern übersetzt werden.

Eine detaillierte Studie der drei präsentierten Intervall-Bibliotheken wird in [44] vorgestellt. Die von SONIC unterstützten Bibliotheken weisen—neben einigen Gemeinsamkeiten—doch relativ viele Unterschiede auf: So werden zum Beispiel die meisten Member-Funktionen der jeweiligen Intervall-Klassen unterschiedlich geschrieben. Als Beispiel diene hier der Mittelpunkt eines Intervalles `x`: Dieser wird über `mid(x)`, `x.mid()` bzw. `Mid(x)` bestimmt. Ein wesentlicher Punkt ist allerdings die Tatsache, dass die implementierten erweiterten (unbeschränkten) Intervalle unüberbrückbare Unterschiede aufweisen. Man beachte, dass zum Beispiel C-XSC in einigen Fällen Exceptions wirft. Dieser Umstand macht eine vernünftige Implementierung von Constraint Propagation unmöglich. Aus diesem Grund wurde eine eigene Klasse `xinterval` für unbeschränkte Intervalle implementiert, die ohne Exceptions auskommt. Innerhalb dieser Klasse werden die verschiedenen Intervall-Operationen und Funktionen der jeweiligen Intervall-Bibliothek mit Hilfe des C-Präprozessors angesteuert. Eine Erweiterung von SONIC um weitere Intervall-Bibliotheken ist deshalb sehr leicht möglich.

Ausführliche Tests haben gezeigt, dass die Verwendung von *filib++* im Hinblick auf die Rechenzeit am vielversprechensten ist. Ist der Einsatz dieser Bibliothek nicht möglich, so sollte *C-XSC* benutzt werden, da diese am vielseitigsten verwendbar ist. Je nach verwendetem Testbeispiel ist *filib++* fünf bis zehn mal schneller als *C-XSC* und ungefähr zwei bis vier mal schneller als die SUN Intervallarithmetik.

5.2 Lineare Optimierung

Für die Berechnung eines *optimalen Präkonditionierers* muss ein lineares Programm gelöst werden. Da die Berechnung linearer Programme sehr aufwändig ist, sollten dafür ausgefeilte Tools mit entsprechendem Know-How einsetzt werden. Momentan ist in SONIC die Nutzung des *GLPK* und von *Galahad* vorgesehen.

GLPK

GLPK steht für **G**nu **L**inear **P**rogramming **K**it und ist momentan unter der Internetpräsenz `ftp://ftp.gnu.org` frei erhältlich. Das Tool fasst eine große Zahl von Routinen zusammen, die im *ANSI C* Standard geschrieben und in einer aufrufbaren Bibliothek organisiert sind. Eine ausführliche Beschreibung der Bibliothek ist bei ihrem Download erhältlich. Der Aufruf aus C++ über so genannte API Routinen ist sehr leicht möglich. Der Vorteil des GLPK liegt eindeutig in der Schnelligkeit der Berechnungen. Sie hat allerdings auch zwei wesentliche Nachteile:

- Tritt während des Lösungsvorganges im *GLPK* ein Fehler auf, so wird dieser unmittelbar über `exit(EXIT_FAILURE)` abgebrochen. Dieser „harte Exit“ kann leider nicht mehr von Exception-Handlern abgefangen werden. Dies hat zur Folge, dass SONIC mit einer für den Nutzer unverständlichen Meldung abbricht. Eine Änderung des Quelltextes des *GLPK* hätte natürlich durch den Autor erfolgen können. Einerseits wäre dies aber extrem aufwändig gewesen (mehr als 1 000 Routinen hätten abgeändert werden müssen) und andererseits hätten diese Änderungen für *jede* neue Version des *GLPK* erneut durchgeführt werden müssen.
- Ein wesentlicher Nachteil ist allerdings, dass der *GLPK* nicht thread-safe ist. Dies hat zur Folge dass dieses Tool nicht eingesetzt werden kann, falls eine Parallelisierung mit OpenMP erwünscht ist. In diesem Fall muss entweder *GALAHAD* oder eine eigene Implementierung des Simplex-Verfahrens eingesetzt werden.

GALAHAD
GALAHAD ist eine Bibliothek für die Lösung großer, nichtlinearer Optimierungsprobleme, welche auf Fortran 90-Routinen basiert. Diese Routinen können ohne großen Aufwand in SONIC (C++-Code) integriert werden, falls ein entsprechender Fortran 90-Compiler zur Verfügung steht. *GALAHAD* hat im Gegensatz zu *GLPK* den Vorteil, dass alle enthaltenden Routinen thread-safe sind. Weiterhin verfügt *GALHAD* über eine vernünftige Fehlerbehandlung. Dies hat zur Folge, dass SONIC bei einer Fehlfunktion nicht wie beim *GLPK* unnötigerweise abgebrochen wird. Ein wesentlicher Nachteil der Bibliothek ist die Performance: Zeitmessungen haben ergeben, dass sich die Rechenzeit bei Einsatz von *GALAHAD* im Vergleich zum Einsatz des *GLPK* verdoppelt.

Simplex von Kearfott
Wie bereits erwähnt, haben wir den GlobSol-Code für die Berechnung des vereinfachten C^W-LP-Präkonditionierers zum Einsatz in SONIC von Fortran nach C++ testweise portiert. Bei ausführlichen Untersuchungen konnte allerdings gezeigt werden, dass der Präkonditionierer mit diesem Code nur ungefähr sechs mal langsamer als mit dem *GLPK* berechnet werden kann.

5.3 Weitere Features

In diesem Abschnitt wollen wir noch kurz auf einige Features eingehen, die in SONIC integriert sind. Es besteht grundsätzlich die Möglichkeit, nach einer gewissen Anzahl von durchgeführten Schritten alle relevanten Daten abzuspeichern. Die Schrittzahl wird über die Variable `BoxesBetweenSavingData` im Control-File eingestellt. Damit ist man in der Lage, die Berechnung zu einem späteren Zeitpunkt fortzusetzen. Dieses Feature in einigen Fällen sehr hilfreich gewesen: Fällt zwischenzeitlich ein Rechner aus, so muss die Berechnung nicht komplett von vorne gestartet werden. Möchte man keinen Gebrauch von dieser Möglichkeit machen, so wird die Variable auf ∞ gesetzt.

Nach einer gewissen Anzahl von Boxen können ebenfalls einige wichtige Informationen ausgegeben werden. Dazu zählt neben der aktuellen Gesamtboxzahl und der Länge der Arbeitsliste $\mathcal{L}$ auch die Anzahl der bereits berechneten Lösungen. Die Schrittweite wird über die Variable `BoxesBetweenPrintProgress` gesteuert.

5.4 Plattformen für die parallelen Berechnungen

ALiCEnext

Ein Teil der Berechnungen wurde auf dem ALiCEnext Cluster der Bergischen Universität Wuppertal durchgeführt. Dieser war im Sommer 2004 der leistungsfähigste Parallelrechner aller deutscher Universitäten und belegt momentan Rang 167 in der Top500-Liste. ALiCEnext besteht aus insgesamt 1024 AMD-Opteron Prozessoren, die auf 512 Blades verteilt sind. Auf jedem dieser Blades befinden sich

1. 2 × AMD Opteron 1.8 GHz Prozessoren,
2. 2 × 250 GB Festplatten,
3. 2 × 1024 MByte RAM,
4. 6 × Gigabit-Ethernet Anschlüsse.

Die 512 Blades sind in insgesamt 11 Racks eingebaut, jeweils 12 nebeneinander in 4 Reihen. Das Netzwerk ist eine Kombination aus Gigabit-Ethernet und einem 2-d Torus.

Sun Fire-Server

Ein weiterer Teil der parallelen Berechnungen wurden auf einem Sun Fire E25K Server der RWTH Aachen durchgeführt. Dieser besitzt insgesamt 72 UltraSPARC IV 1.05 GHz Prozessoren mit 288 GB gemeinsamem Speicher, und läuft unter Solaris 10.

5.5 Bekannte Tools

GlobSol

Dieses Tool wurde von Kearfott et al. entwickelt [19] und ist momentan das bekannteste Paket auf dem Gebiet des ergebnisverifizierenden Rechnens. Es ist in Fortran90 geschrieben und momentan nur auf Sun-Maschinen lauffähig. Es setzt die Intervallbibliothek *INTLIB* [39], die in Fortran77 implementiert wurde, ein. GlobSol ist hauptsächlich zur Lösung von globalen Optimierungsproblemen geschrieben worden, bietet mittlerweile aber auch die Möglichkeit, nichtlineare Gleichungssysteme verifiziert zu lösen. Die Funktionen müssen vom User als Fortran-Programm implementiert und bei jedem Start von GlobSol compiliert werden. Anschließend werden von GlobSol so genannte Code-Listen für die Funktionen erzeugt. Dies hat gegenüber SONIC den Vorteil, dass zusätzlich Funktionen angegeben werden können, die keine Termdarstellung haben. Nachteilig ist allerdings, dass keine Termvereinfachungen durchgeführt werden können. Durch die

Code-Listen liegen in GlobSol symbolische Informationen über die Funktionen vor, die bei der Differentiation der Funktionen (Code-Listen) genutzt werden. Wesentliche Kontraktionverfahren von GlobSol sind das Newton-Verfahren und der *Substitution-Iteration-Ansatz*, welcher ähnlich zum Constraint-Propagation von SONIC ist. Auf die Anwendung des erweiterten Newton-Verfahrens wird aus Kostengründen verzichtet. Es kann ebenfalls eine vereinfachte Version des C^W-LP-Präkonditionierers angewendet werden. GlobSol verfügt bei der globalen Optimierung über eine zusätzliche, interesannte Methode: Der so genannte Peeling-Prozess, welcher in einer nächsten Projektphase ebenfalls—zumindest testweise—in SONIC integriert werden soll. Eine genaue Beschreibung von GlobSol findet der interessiert Leser in [38].

ALIAS
Dieses Tool wurde von der Arbeitsgruppe von Jean-Pierre Merlet [47] entwickelt. Es ist in der Programmiersprache C++ geschrieben und setzt auf die Intervallbibliothek PROFIL/BIAS [43] auf. ALIAS verfügt wie SONIC über einen integrierten verifizierten nichtlinearen Gleichungslöser und einen globalen Optimierer. Die Parallelisierung ist ebenfalls möglich (leider ist unbekannt, ob mit MPI oder OpenMP). Der wesentliche Nachteil von ALIAS ist die Tatsache, dass der User sich selbst einen Löser „zusammenbauen“ muss: Es stehen verschiedene Lösungsroutinen zur Verfügung, die jeweils miteinander kombiniert werden können. Dazu muss ein C++ Programm geschrieben werden, in dem auch die Funktions- und Ableitungsterme des zu lösenden Systems *implementiert* werden müssen. Seit kurzer Zeit steht auch ein Maple-Interface zur Verfügung, aus dem ALIAS gestartet werden kann. Dies hat den Vorteil, dass die Vorteile eines Formelmanipulationssystems ausgenutzt werden können (Stichwort: Termvereinfachungen).

5.6 Beispiel für die Benutzung von SONIC

Wie bereits erwähnt, wird SONIC über drei verschiedene Textdateien gesteuert: Das Control-File, das Rule-File und das System-File. Wir werden am Beispiel des Optimierungsproblems *Trefethen* kurz erläutern, wie diese Dateien speziell eingesetzt werden.

5.6.1 Control-File

Mit Hilfe des Control-File können spezielle Einstellungen von SONIC vorgenommen werden. Die hier angegebenen Einstellungen *überschreiben* die Standard-Einstellungen. Wird folglich kein Control-File angegeben, so werden ausschließlich die Standard-Einstellungen verwendet.

```
UseCP                          on
UseNewton                      on
UseTaylor1Separate             off
UseTaylor2Separate             off
```

```
  UseCPWithTaylor1                    on
  UseCPWithTaylor2                    off

  UseRandomizedFirstVarInNewton       off

  UseExclusionRegions                 off
  TerminationOnFirstHit               off

  BoxOrderingStrategy                 1
                                      # 0 = no ordering strategy
                                      # 1 = maximum equation ordering

  StartWithSavedData                  off
  BoxesBetweenSavingData              150000
  BoxesBetweenPrintProgress           500

  Bisection                           HybridBisection
                                      # MaxSmear
                                      # MaxSmearDiam
                                      # MaxSumMagnitude
                                      # ZeroNearBound
                                      # MaxDiam

  UseSplittingPrecondForBisection     off
  MaxPreBisections                    2
  MinRelGapsizeForPreBisection        0.1

  GlobalMinimumOnBoundary             on
```

5.6.2 Rule-File

Über das Rule-File können die Regeln für den in SONIC integrierten, regelbasierten Termvereinfacher angegeben werden. Momentan sind insgesamt 81 verschiedene Regeln integriert, die sehr leicht um weitere erweitert werden können. Desweiteren ist es möglich, konstante Parameter, wie zum Beispiel `pi = 4 * arctan( 1 )`, festzulegen.

```
ConstantParameters :

  e = exp( 1.0 ) ,
  pi = 4 * arctan( 1.0 ) ,
  two_pi = 8 * arctan( 1.0 ) .

Rules :
```

```
- ( - t ) = t ,

t + 0 = t ,
0 + t = t ,
( - t1 ) + ( - t2 ) = - ( t1 + t2 ) ,
( - t1 ) + t2 = t2 - t1 ,
t1 + ( - t2 ) = t1 - t2 ,
t1 * t2 + t1 * t3 = t1 * ( t2 + t3 ) ,
t2 * t1 + t3 * t1 = ( t2 + t3 ) * t1 ,
t1 / t2 + t3 / t2 = ( t1 + t3 ) / t2 ,
t1 + ( t2 - t3 ) = ( t1 + t2 ) - t3 ,
( t1 - t2 ) + t3 = ( t1 + t3 ) - t2 ,
( t1 - t2 ) + ( t3 - t4 ) = ( t1 + t3 ) - ( t2 + t4 ) ,

t - 0 = t ,
0 - t = - t ,
t - t = 0 .
```

5.6.3 System-File

Im System-File werden Wertebereiche der Variablen, Systemgleichungen, etc. deklariert. Als Beispiel haben wir hier das System-File für das Optimierungsproblem von Trefethen angegeben.

```
MaxIterations : 10000000

ConstantParameters : .

Variables :

  x1  = [ -1 , 1 ] , 1.0E-12 ,
  x2  = [ -1 , 1 ] , 1.0E-12 .

ActiveVariables : .

ObjectiveFunction :

        -sin( 10*(x1+x2) ) + ( x1^2 + x2^2 ) / 4
        + exp( sin(50*x1) ) + sin( 60*exp(x2) )
        + sin( 70*sin(x1) ) + sin( sin( 80*x2 ) ) .

System : .
```

```
SingularityType : PlainSolution
```

Mit diesen drei Dateien ist der User nun in der Lage, das nichtlineare Gleichungssystem bzw. Optimierungsproblem zu lösen. Als Aufruf genügt

```
SONIC -S <System-File> -R <Rule-File> -C <Control-File> .
```

Desweiteren können diverse Ausgaben von SONIC, wie zum Beispiel Statistiken und Resultate, in unterschiedliche Dateien oder den Bildschirm ausgegeben werden. Die dafür benötigten Optionen können sehr leicht über den Aufruf

```
SONIC -help
```

abgefragt werden. Zudem steht ein umfangreiches, aber trotzdem leicht zu bedienendes Makefile zur Verfügung. Mit diesem kann SONIC mit verschiedenen Compilern, Intervallbibliotheken und Optionen übersetzt werden.

Kapitel 6

Zusammenfassung

In dieser Arbeit haben wir den ergebnisverifizierenden nichtlinearen Gleichungslöser und Optimierer SONIC vorgestellt, der auf effizienten symbolischen und numerischen Verfahren basiert. Eines der effektivsten Kontraktionsverfahren in SONIC ist das so genannte Intervall-Newton-Gauß-Seidel-Verfahren, welches aber erst in Zusammenspiel mit der Präkonditionierung seine Stärken ausspielt. Neben dem hinlänglich bekannten Inverse-Midpoint-Präkonditionierer konnten auch zwei „optimale Präkonditionierer“, die von Kearfott und Novoa entwickelt wurden, in SONIC integriert werden. Bezeichnet $[x'_k]$ das resultierende Intervall nach einem Intervall-Newton-Gauß-Seidel-Schritt für die k-te Variable vor dem Schnitt mit dem ursprünglichen Suchbereich $[x_k]$, so minimiert der so genannte C^W-LP-Präkonditionierer den Durchmesser von $[x'_k]$. Während mit diesem Präkonditionierer also eine möglichst starke Kontraktion erzielt wird, erreicht man mit dem S^M-LP-Präkonditionierer ein besonders starkes Aufsplitten von $[x_k]$ in zwei disjunkte Teilintervalle. Diese beiden Präkonditionierer werden über nichtlineare Optimierungsprobleme, die in lineare Programme überführt werden können, berechnet. Leider existiert in der Literatur nur eine *vereinfachte* Darstellung des C^W-LP-Präkonditionierers und eine *falsche* Darstellung des S^M-LP-Präkonditionierers. In Kapitel 2.2.1.2 konnte der Autor erstmals eine vollständige und korrekte Darstellung dieser optimalen Präkonditionierer als lineares Programm entwickeln. Wir haben in dieser Arbeit dargelegt, dass man eigentlich daran interessiert ist, den Durchmesser des Intervalles $[x'_k] \cap [x_k]$ zu minimieren. Der Autor konnte für diesen Fall einen optimalen Präkonditionierer (Anhang A) entwickeln, der allerdings bis heute nur in ein quadratisches Optimierungsproblem überführt werden konnte und daher aus Effizienzgründen momentan nicht eingesetzt wird.

Desweiteren konnte der von Willems entwickelte hierarchische Ansatz, welcher das numerische Intervall-Newton-Gauß-Seidel-Verfahren und das symbolische Constraint-Propagation eng miteinander verknüpft, in SONIC integriert werden. Bei dieser Methode wird das Intervall-Newton-Gauß-Seidel-Verfahren auf verschiedene Zerlegungen des ursprünglichen Gleichungssystems angewendet. Es konnte gezeigt werden, dass durch die Anwendung des Newton-Verfahrens auf alle Gleichungen des vollen Splits eine sehr gute Kontraktion erzielt werden kann, die allerdings mit einem extrem hohen Aufwand

einhergeht. Aus diesem Grund wird in bestehenden Implementierungen auf die Anwendung dieses Verfahrens verzichtet. Der Autor konnte ein hybrides Newton-Verfahren (Kapitel 2.3.7) entwickeln, welches in den hierarchischen Ansatz integriert werden kann. Dieses Verfahren implementiert eine intelligente Steuerung von zu bearbeitenden Gleichungen in den einzelnen Zerlegungen und zu verwendenden Präkonditionierern. Die Auswahl der Gleichungen erfolgt dabei nach dem „Kostenprinzip“. Dazu wird, abhängig von der Größe des gerade betrachteten Split-Systems, eine bestimmte Schrittweite i festgelegt. Das Newton-Verfahren wird dann nur noch auf jeder i-ten Gleichung durchgeführt, um zu gewährleisten, dass für große Zerlegungen nicht zu viel Zeit benötigt wird. Die Wahl der zu verwendenden Präkonditionierer erfolgt zusätzlich nach einem „Bewährungsprinzip“. In ausführlichen Untersuchungen konnte festgestellt werden, dass der C^W-LP-Präkonditionierer sehr effektiv arbeitet, dessen Berechnung über ein lineares Programm aber oft für eine Reihe von hintereinanderliegenden Iterationsschritten fehlschlägt. Übersteigt daher die Anzahl der fehlgeschlagenen Berechnungen einen gewissen Prozenzsatz, so wird für eine vorgegebene Anzahl von Boxen auf den C^W-LP-Präkonditionierer verzichtet. Im Rahmen dieser Arbeit konnte durch ausführliche Untersuchungen gezeigt werden, dass damit der hierarchische Ansatz mit integriertem hybriden Newton-Verfahren zu einem effektiven Kontraktionsverfahren wird.

Ein Vergleich von fünf bekannten Strategien zur Unterteilung einer aktuell betrachteten Box hat in dieser Arbeit gezeigt, dass diese für eine Vielzahl von Testbeispielen sehr effizient arbeiten, aber auch in einigen Fällen zu einer—verglichen mit den anderen Strategien—extrem hohen Anzal von betrachteten Box führen. Der Autor konnte eine hybride Bisektionsstrategie (Kapitel 2.4.6) entwickeln, die versucht, für die gerade betrachtete Box eine der etablierten Strategien geeignet auszuwählen. Durch ausführliche Experimente konnte gezeigt werden, dass mit der neuen Strategie die Peaks in der Anzahl der betrachteten Boxen, die alle Grundstrategien aufweisen, vermieden werden können. Desweiteren konnte der Autor eine Unterteilungsstrategie entwickeln, welches das Newton-Verfahren in Kombination mit dem optimalen S^M-LP-Präkonditionierer zum Splitten einer Box nutzt (Kapitel 2.4.8). Durch diese Methode wird die Box nicht nur in zwei Hälften geteilt, sondern sogar eine Lücke aus dem Inneren der Box herausgeschnitten. Es konnte gezeigt werden, dass diese effektive Methode weiter verbessert werden kann, wenn anstatt aller Systemgleichungen nur eine vielversprechende Komponente betrachtet wird.

Die so genannten Exclusion-Regions, die von Neumaier eingeführt wurden, geben Teilgebiete der Startbox an, die von einem Branch–and–Bound-Löser nicht weiter betrachtet werden müssen. Während Neumaier diese Gebiete mit Hilfe des Satzes von Newton-Kantorovich berechnet, setzen wir in SONIC eine Variante ein, die mit der Newton-Verifikation arbeitet. In Untersuchungen konnte festgestellt werden, dass die Methode von Neumaier leicht bessere (also größere) Exclusion-Regions liefert. Trotzdem haben beide Methoden den Nachteil, dass die berechneten Exclusion-Regions noch zu klein sind, um effektiv in SONIC eingesetzt werden zu können. Zudem konnte der Autor ein Verfahren entwickeln, welches die Exclusion-Regions von einem „Alles-oder-Nichts-Test“ zu einem echten Kontraktionsverfahren (Kapitel 2.7.2) erweitert.

Der Autor hat ebenfalls je ein Verfahren zur verifizierten Lösung von Optimie-

rungsproblemen ohne bzw. mit Nebenbedingungen implementiert. Die verwendeten Verfahren basieren auf bewährten Techniken, die in der Regel in irgendeiner Form in alle gängigen globalen Optimierer integriert sind. Allerdings profitiert SONIC bei der Lösung des Gradientensystems bzw. des Fritz-John-Systems wesentlich von der ausgezeichneten Qualität des implementierten nichtlinearen Gleichungslösers. Zusätzlich konnte eine Variante der „one-dimensional Newton iteration“ von Tapamo und Frommer in SONIC implementiert werden. Gegenüber dem ursprünglichen Ansatz, dass das zusätzliche Constraint nach einer zu wählenden Variablen aufgelöst wird, erhalten wir durch Integration dieser Gleichung in das Gradientensystem ebenfalls eine symbolische Kopplung mit den Originalvariablen und dadurch eine stärkere Kontraktion (Kapitel 3.1.6). Es konnte durch ausführliche Experimente gezeigt werden, dass das implementierte Gesamt-Verfahren zur Lösung von Optimierungsproblemen ohne Nebenbedingungen dem bewährten Ansatz von GlobSol in vielen Fällen—besonders für schwere Testsysteme—überlegen ist. Bei der Optimierung mit Nebenbedingungen werden bereits sehr gute Ergebnisse erzielt, die vergleichbar mit GlobSol sind. Da wir bis zum jetzigen Zeitpunkt noch wenig Erfahrung auf diesem Gebiet haben, ist zu erwarten, dass sich die erzielten Ergebnisse noch weiter verbessern lassen.

In Kapitel 4 dieser Arbeit konnte der Autor unterschiedliche Methoden entwickeln, um den vorgestellten Algorithmus zur Lösung von nichtlinearen Gleichungssystemen auf Rechnern mit gemeinsamem und verteiltem Speicher zu parallelisieren. Neben einer einfachen und intuitiven OpenMP-Implementierung konnte der Autor zusätzlich das so genannte *Taskq*-Konzept für Rechner mit gemeinsamem Speicher, welches einige OpenMP-Compiler zur Verfügung stellen, in SONIC integrieren. Dieses Konzept ermöglicht es, bestimmte Tasks asynchron auf aktive Threads zu verteilen. Leider ist dieses Konzept noch nicht in den OpenMP-Standard integriert. Auf Vorschlag von Dieter an Mey konnte der Autor dieses effiziente Mittel mit den Standard-OpenMP-Direktiven nachbilden. Für Rechner mit verteiltem Speicher konnte der Autor eine aufwändige, aber effiziente Parallelisierung mit MPI entwickeln. Obwohl die Effizienz der Ansätze jeweils mit dem Aufwand, der für die Implementierung benötigt wird, zunimmt, werden bereits für die „einfachste“ Variante gute Speedups erreicht. Bei der Parallelisierung des Optimierers ohne Nebenbedingungen greifen wir auf den bewährten Ansatz von Berner zurück. Diese Strategie implementiert einen *zentralen Vermittler*, der für einen Ausgleich bezüglich der zu bearbeitenden Boxen—qualitativ und quantitativ—zwischen den Prozessoren sorgt. Die dargestellten numerischen Ergebnisse zeigen deutlich, dass Verfahren entwickelt bzw. implementiert werden konnten, die insbesondere auf Problemen mit hoher Laufzeit sehr effizient arbeiten.

Ausführliche Vergleiche haben gezeigt, dass die implementierten Gesamtverfahren zur Lösung von nichtlinearen Gleichungssystemen und Optimierungsaufgaben ohne Nebenbedingungen den bewährten Tools GlobSol und ALIAS für ein breites Spektrum von Standard-Testbeispielen überlegen sind. Desweiteren konnte gezeigt werden, dass das implementierte Verfahren zur Lösung von Optimierungsaufgaben mit Nebenbedingungen konkurrenzfähig zu GlobSol ist. SONIC ist momentan das einzige Tool, welches grundsätzlich nicht-totale Funktionen zulässt. Wir haben herausgearbeitet, dass mit Hilfe dieses einfachen Features viele Systeme gelöst werden können, bei denen GlobSol und ALIAS frühzeitig mit einer Fehlermeldung abbrechen. Durch die Robust-

heit bezüglich der Parametereinstellungen wird es dem Nutzer ermöglicht, die Testsysteme bereits mit den Standard-Einstellungen von SONIC effizient zu lösen. Durch den implementierten generischen Intervall-Code erhält man den zusätzlichen Vorteil der Portabilität, welcher bei ALIAS nur bedingt und bei GlobSol gar nicht gegeben ist. Desweiteren ist SONIC momentan das einzige Tool, welches prinzipiell auch unbeschränkte Suchbereiche zulässt. Mit Hilfe des effizienten Gesamtpaketes ist man heute in der Lage, verschiedene Systeme aus der Prozesstechnik sowie weitere anwendungsorientierte Beispiele zu lösen, bei dem Tools wie GlobSol und ALIAS „versagen“.

In Tabelle 6.1 haben wir noch einmal alle Methoden und Features, die in SONIC, GlobSol und ALIAS integriert sind, zusammengefasst. Ist eine Methode bzw. Feature in eines der Tools integriert, so wird dies über ein '+' gekennzeichnet. Anderenfalls wird ein '–' in die Tabelle eingetragen. Abschließend wollen wir noch einmal kurz die Punkte zusammenfassen, die vom Autor in dieser Arbeit entwickelt bzw. untersucht wurden (in Tabelle 6.1 kursiv gedruckt):

- Konzeptuelle Entwicklung von SONIC in Zusammenarbeit mit Prof. Dr. Bruno Lang und Dipl.-Inf. Paul Willems
- Hauptverantwortliche Implementierung von SONIC (einige Teile wurden durch Dipl.-Inf. Paul Willems implementiert)
- Korrekte und vollständige Formulierung verschiedener optimaler Präkonditionierer als lineare Programme (Kapitel 2.2.1.2)
- Entwicklung eines neuen optimalen Präkonditionierers (Anhang A)
- Erweiterung des hierarchischen Ansatzes um ein effizientes hybrides Newton-Verfahren, welches als Steuerungsmodul für die geschickte Auswahl vielversprechender Gleichungen und Präkonditionierer innerhalb der Split-Hierarchie dient (Kapitel 2.3.7)
- Entwicklung einer hybriden Bisektionsstrategie (Kapitel 2.4.6)
- Entwurf der gezielten Splitting-Bisektion mit Hilfe des Newton-Verfahrens und des S^M-LP-Präkonditionierers (Kapitel 2.4.8)
- Untersuchungen zur Qualität der Exclusion-Regions und Erweiterung des daraus resultierenden „Alles-oder-Nichts-Test“ zu einem echten Kontraktionsverfahren (Kapitel 2.7.2)
- Entwicklung verschiedener Parallelisierungsstrategien auf Rechnern mit gemeinsamem und verteiltem Speicher zur verifizierten Lösung von nichtlinearen Gleichungssystemen (Kapitel 4)

Methode	SONIC	GlobSol	ALIAS
Unbeschränkte Suchboxen	+	–	–
Nicht-quadr. Systeme	+	–	+
Nicht-totale Funktionen	+	–	–
Termvereinfachungen	+	–	(+)
Ableitungen	symbolisch	AD	(symbolisch)
TerminationOnFirstHit	+	–	–
Restart-Möglichkeit	+	–	–
Taylor-Verfahren	+	–	+
(Einfacher) Newton	+	+	+
C^{IM}*-Präkonditionierer*	+	+	+
C^{W}*-LP-Präkonditionierer*	+	+	–
S^{M}*-LP-Präkonditionierer*	+	–	–
Constraint-Propagation	+	+	–
Taylor in CP	+	–	–
Simplex f. lineare Systeme	–	–	+
Erweiterter Newton	+	–	–
Hierarchischer Ansatz	+	–	–
Hybrider Newton	+	–	–
Exclusion-Regions	+	–	–
Statische Bisektionstrat.	5	1	1
Hybride Bisektionsstrat.	+	–	–
Splitting-Bisektion	+	–	–
Newton-Verifikation	+	+	+
Miranda	+	–	–
Borsuk	+	–	–
Degree-Test	+	–	–
Nicht-quadr. Verifikation	+	–	–
Cut-Off-Test	+	+	+
Monotonie-Test	+	+	+
Konkavitäts-Test	+	–	+
Peeling	–	+	–
Lokaler Optimierer	–	+	–
One-dimensional Newton	+	–	–
Fritz-John-System	+	+	+
Box-Verwaltung	sort. Liste/Heap	sort. Liste	sort. Liste
Intervallbibliotheken	C-XSC,filib++,Sun	INTLIB	PROFIL/BIAS
Compiler	alle (C++)	Sun C++ V5.6	gcc 2.95
Parallelisierung	OpenMP/MPI	–	+

Tabelle 6.1: Zusammenfassung der vorhandenen Methoden und Features von SONIC, GlobSol und ALIAS. Die vom Autor in dieser Arbeit entwickelten bzw. untersuchten Verfahren sind jeweils kursiv gedruckt.

Anhang

A Präkonditionierer

Der in Kapitel 2.2.1.2 vorgestellte C^W-LP-Präkonditionierer minimiert den Durchmesser des Bildes $[x'_k]$ eines Gauß-Seidel-Schrittes *vor* dem Schnitt mit dem ursprünglichen Intervall $[x_k]$. Es wäre allerdings wünschenswert, wenn man sogar den Durchmesser des Schnittes $S := [x'_k] \cap [x_k]$ minimieren könnte.

Definition 6.1 Ein C-Präkonditionierer $\mathbf{C_k}$ heißt genau dann *$C^{W'}$-Präkonditionierer*, wenn er den Durchmesser von $[x'_k] \cap [x_k]$ unter allen C-Präkonditionierern minimiert.

Der $C^{W'}$-Präkonditionierer ist folglich die Lösung des (nichtlinearen) Optimierungsproblems

$$\min_{\underline{d}_k(\mathbf{C_k})=1} \operatorname{diam}\left([x_k] \cap \left(\check{x}_k - \frac{[n_k](\mathbf{C_k})}{[d_k](\mathbf{C_k})}\right)\right) \ , \tag{A.1}$$

da wir uns auf normale C-Präkonditionierer beschränken können. Wir gehen zunächst davon aus, dass die Menge S nicht leer ist. Ist nun $0 \in [n_k](\mathbf{C_k})$ (gleiche Voraussetzung wie beim C^W-LP-Präkonditionierer), so lässt sich nach Lemma 2.3 für den Durchmesser von S die folgende Formel ableiten:

$$\begin{aligned}
\operatorname{diam}(S) &= \operatorname{diam}([\max\{\underline{x}'_k, \underline{x}_k\}, \min\{\overline{x}'_k, \overline{x}_k\}]) \\
&= \min\{\overline{x}'_k, \overline{x}_k\} - \max\{\underline{x}'_k, \underline{x}_k\} \\
&= -\max\{-\overline{x}'_k, -\overline{x}_k\} - \max\{\underline{x}'_k, \underline{x}_k\} \\
&= -\left(\max\{-\sup(\check{x}_k - [n_k](\mathbf{C_k})), -\overline{x}_k\} + \max\{\inf(\check{x}_k - [n_k](\mathbf{C_k})), \underline{x}_k\}\right) \\
&= -\left(\max\{-\check{x}_k + \underline{n}_k(\mathbf{C_k}), -\overline{x}_k\} + \max\{\check{x}_k - \overline{n}_k(\mathbf{C_k}), \underline{x}_k\}\right) \\
&= -\left(-\check{x}_k + \max\{\underline{n}_k(\mathbf{C_k}), \check{x}_k - \overline{x}_k\} + \check{x}_k + \max\{-\overline{n}(\mathbf{C_k}), \underline{x}_k - \check{x}_k\}\right) \\
&= -(\underbrace{\max\{\underline{n}_k(\mathbf{C_k}), \check{x}_k - \overline{x}_k\}}_{=:\alpha} + \underbrace{\max\{-\overline{n}(\mathbf{C_k}), \underline{x}_k - \check{x}_k\}}_{=:\beta}).
\end{aligned} \tag{A.2}$$

Die oben gemachte Einschränkung, dass S nicht leer sein soll, bezieht sich offensichtlich nur auf die Darstellung von S: In der Einleitung haben wir keine Intervalle $[x]$ mit $\underline{x} > \overline{x}$

zugelassen. Trotzdem würde Gleichung (A.2) auch für $S = \emptyset$ eine *sinnvolle Formel* darstellen: In diesem Fall würde ein negativer Wert für den Durchmesser angenommen. Bei genauerer Betrachtung sieht man allerdings, dass immer $\text{diam}(S) \geq 0$ gilt, da stets $\alpha, \beta \leq 0$ gilt.

Für die beiden Ausdrücke α und β führen wir nun jeweils zwei Schlupfvariablen α_1 und α_2 bzw. β_1 und β_2 ein, um die Maximums-Funktionen aufzulösen. Damit erhalten wir

$$\begin{aligned} \alpha &= \underline{n}_k(\mathbf{C_k}) + \alpha_1 \\ \alpha &= \check{x}_k - \overline{x}_k + \alpha_2 \\ \beta &= -\overline{n}_k(\mathbf{C_k}) + \beta_1 \\ \beta &= \underline{x}_k - \check{x}_k + \beta_2 \end{aligned} \tag{A.3}$$

mit $\alpha_1, \alpha_2, \beta_1, \beta_2 \geq 0$ sowie $\alpha_1 \cdot \alpha_2 = 0$ und $\beta_1 \cdot \beta_2 = 0$. Die vier Bedingungen (A.3) können dann nach (2.21) und (2.22) in lineare Gleichungen überführt werden. Da wird den Durchmesser von S minimieren wollen, ergibt sich die (lineare) Zielfunktion

$$\mathcal{O}_{\mathrm{C^{W'}}} = -(\alpha + \beta) \ . \tag{A.4}$$

Definition 6.2 Jede Lösung des quadratischen Optimierungsproblems mit den $2(m + n + 2)$ Variablen

$$\begin{aligned} &(c^+_{k,1}, \ldots, c^+_{k,m}, c^-_{k,1}, \ldots, c^-_{k,m}, \\ &\ v^+_1, \ldots, v^+_{k-1}, v^+_{k+1}, \ldots, v^+_n, v^-_1, \ldots, v^-_{k-1}, v^-_{k+1}, \ldots, v^-_n, \alpha, \alpha_1, \alpha_2, \beta, \beta_1, \beta_2) \end{aligned}$$

mit der Zielfunktion (A.4) und den $n+4$ linearen Nebenbedingungen (A.3), (2.20) und (2.28) sowie den beiden quadratischen Nebenbedingungen $\alpha_1 \cdot \alpha_2 = 0$ und $\beta_1 \cdot \beta_2 = 0$ heißt *$C^{W'}$-LP-Präkonditionierer*. Dabei werden $\underline{n}_k(\mathbf{C_k})$ und $\overline{n}_k(\mathbf{C_k})$ mit Hilfe der Gleichungen (2.21) bzw. (2.22) dargestellt.

Die zugehörige Präkonditionierungszeile $\mathbf{C_k}$ wird nach Berechnung des quadratischen Optimierungsproblems über die Formel $c_{k,i} = c^+_{k,i} - c^-_{k,i}$ für $i = 1, \ldots, m$ bestimmt. Wir gehen nun der Frage nach wann der $\mathrm{C^{W'}}$-LP-Präkonditionierer das Optimierungsproblem (A.1) löst. Notwendige Voraussetzung dafür sind die Eigenschaften $c^+_{k,i} \cdot c^-_{k,i} = 0$ für $i = 1, \ldots, m$ und $v^+_j \cdot v^-_j = 0$ für $j = 1, \ldots, n$. Während die erste Bedingung nach Berechnung des Optimierungsproblems überprüft werden muss (oder alternativ als Nebenbedingung hinzugefügt wird), ergibt sich die zweite Bedingung *automatisch*, wie der folgende Beweis zeigt.

Um im Folgenden die Abhängigkeit der Funktion $\mathcal{O}_{\mathrm{C^{W'}}}$ von seinen Variablen zu verdeutlichen, schreiben wir auch

$$\mathcal{O}_{\mathrm{C^{W'}}}(\mathbf{c}^+, \mathbf{c}^-, \mathbf{v}^+, \mathbf{v}^-, \alpha, \alpha_1, \alpha_2, \beta, \beta_1, \beta_2) := O_{\mathrm{C^{W'}}} \ .$$

Nach Berechnung des quadratischen Optimierungsproblems repräsentieren die $2(n-1)$ Variablen $\mathbf{v}^+$ bzw. $\mathbf{v}^-$ zunächst einmal beliebige Werte und nicht die jeweiligen positiven bzw. negativen Anteile von $\mathbf{v}$. Um dies zu verdeutlichen, bezeichnen wir diese Werte mit $\mathbf{v}'$ bzw. $\mathbf{v}''$.

Korollar 6.1 Für eine beliebige Zahl $a \in \mathbb{R} \setminus \{0\}$ sei eine Darstellung $a = a' - a''$ mit $a', a'' \geq 0$ gegeben. Ist $a^+ \neq a'$ und $a^- \neq a''$, dann gilt

1. $x' \geq x^+$ und $x'' \geq x^-$,
2. Es gilt $x' > x^+$ und/oder $x'' > x^-$.

Beweis:

1. trivial
2. Beweis durch Widerspruch: Nach 1. gilt dann $a' = a^+$ und $a'' = a^-$.

Wir zeigen nun, dass der minimale Wert der Zielfunktion für $\mathbf{v}^+$ und $\mathbf{v}^-$ angenommen wird. Angenommen, das quadratische Optimierungsproblem liefert als Ergebnis $\mathbf{v}' \neq \mathbf{v}^+$ und $\mathbf{v}'' \neq \mathbf{v}^-$. Dann gilt aber nach Korollar 6.1 und (2.23)

$$
\begin{aligned}
O_{C^{W'}}\ & (\mathbf{c}', \mathbf{c}'', \mathbf{v}', \mathbf{v}'', \alpha', \alpha_1, \alpha_2, \beta', \beta_1, \beta_2) \\
&= -(\alpha + \beta) \\
&= -(\underline{n_k}(\mathbf{C_k}) + \alpha_1 - \overline{n}_k(\mathbf{C_k}) + \beta_1) \\
&= \operatorname{diam}([n_k](\mathbf{C_k})) - (\alpha_1 + \beta_1) \\
&= \sum_{j=1}^{m} (c'_{k,j} + c''_{k,j}) \cdot \operatorname{diam}([b_k]) \\
&\quad + \delta \cdot \sum_{\substack{j=1 \\ j \neq k}}^{n} \operatorname{diam}([x_j]) \left(\sum_{i=1}^{m} (c''_{k,i} \overline{a}_{i,j} - c'_{k,i} \underline{a}_{i,j}) + v'_j \right) \\
&\quad + (1 - \delta) \cdot \sum_{\substack{j=1 \\ j \neq k}}^{n} \operatorname{diam}([x_j]) \left(\sum_{i=1}^{m} (c'_{k,i} \overline{a}_{i,j} - c''_{k,i} \underline{a}_{i,j}) + v''_j \right) - (\alpha_1 + \beta_1) \\
&> \sum_{j=1}^{m} (c'_{k,j} + c''_{k,j}) \cdot \operatorname{diam}([b_k]) \\
&\quad + \delta \cdot \sum_{\substack{j=1 \\ j \neq k}}^{n} \cdot \operatorname{diam}([x_j]) \left(\sum_{i=1}^{m} (c''_{k,i} \overline{a}_{i,j} - c'_{k,i} \underline{a}_{i,j}) + v^+_j \right) \\
&\quad + (1 - \delta) \cdot \sum_{\substack{j=1 \\ j \neq k}}^{n} \operatorname{diam}([x_j]) \left(\sum_{i=1}^{m} (c'_{k,i} \overline{a}_{i,j} - c''_{k,i} \underline{a}_{i,j}) + v^-_j \right) - (\alpha_1 + \beta_1) \\
&= O_{C^{W'}}(\mathbf{c}', \mathbf{c}'', \mathbf{v}^+, \mathbf{v}^-, \alpha', \alpha_1, \alpha_2, \beta', \beta_1, \beta_2)
\end{aligned}
$$

Dies bedeutet aber, dass das zuvor berechnete Tupel keine Minimalstelle der Zielfunktion ist. Damit wird das Minimum immer für $\mathbf{v}^+$ und $\mathbf{v}^-$ angenommen.

B Untersuchte Testprobleme

Um die Qualität von SONIC für ein breites Spektrum von Testproblemene zu belegen, wurden in dieser Arbeit eine große Zahl von globalen Optimierungsproblemen und nichtlinearen Gleichungssystemen betrachtet.

Die untersuchten Testprobleme sind dabei den unterschiedlichsten Gebieten entnommen. Um Vergleiche mit bestehenden Tools durchführen zu können, ist ein Großteil der Probleme verschiedenen Testsuites entnommen. Ein weitere wichtige Problemklasse stellen nichtlineare Gleichungssysteme aus der Verfahrenstechnik (siehe Einleitung) dar. Eine Vielzahl weiterer Probleme soll die Qualität von SONIC vor allem für Probleme aus der Praxis belegen.

Neben der Definition der Funktionen (nichtlineares Gleichungssystem oder Optimierungaufgabe) werden sowohl die zu untersuchende Startbox $[\mathbf{x}]$ als auch die geforderten Genauigkeiten ε_j für die Boxkomponenten angegeben. Werden keine weiteren Angaben gemacht, so wird stets vorausgesetzt, dass Optimierungsprobleme ihr Optimum im Innern der Startbox annehmen. Auf die Angabe der Lösungen zu den einzelnen Problemen wollen wir an dieser Stelle verzichten. Sollten diese von besonderem Interesse sein, so wurde bereits in einem der vorigen Kapitel auf diese eingegangen.

Eine Unterscheidung der Probleme in leichte, mittlere und schwere Probleme—wie es in vielen Arbeiten gemacht wird—erfolgt an dieser Stelle nicht, da diese Begriffe stark von dem zur Verfügung stehenden Löser bzw. Optimierer abhängen. Es kommt relativ häufig vor, dass sich Tools auf bestimmte Problemklassen spezialisieren. Dies kann dazu führen, dass ein Problem durch Verwendung zweier unterschiedlicher Löser oder Optimierer in völlig andere Kategorien eingestuft würde.

B.1 Gleichungssysteme

B.1.1 7erSystem

Die Funktion $\mathbf{f} = (f_1, \ldots, f_7)^t$, deren Nullstellen gesucht werden, ist in Abbildung 6.1 bis 6.3 dargestellt. Als Startbox wurde die Box $[\mathbf{x}] = [-2, 2]^7$ betrachtet. Die Genauigkeitsanforderung liegt bei $\varepsilon_j = 10^{-8}$ für $j = 1, \ldots, 7$.

$$
\begin{aligned}
f_1(\mathbf{x}) &:= x_1 + x_2 + x_3 - 1 \\
f_2(\mathbf{x}) &:= -0.6675958407 + 0.2055525566x_1 + 0.6625749516\text{E-}1\ x_2 + \\
&\quad 0.2135733917\text{E-}1\ x_3 - 0.3294350663x_4 - 0.2878424112x_5 - \\
&\quad 0.2515010154x_6 - 0.2197478839x_7 + 0.2003078577\text{E-}2\ x_1x_2 + \\
&\quad 0.6456692700\text{E-}3\ x_1x_3 + 0.2081240402\text{E-}3\ x_2x_3 + \\
&\quad 0.2099295270\text{E-}1\ x_1x_4 + 0.6766836107\text{E-}2\ x_2x_4 + \\
&\quad 0.2181211551\text{E-}2\ x_3x_4 + 0.1834249824\text{E-}1\ x_1x_5 + \\
&\quad 0.5912492690\text{E-}2\ x_2x_5 + 0.1905823807\text{E-}2\ x_3x_5 + \\
&\quad 0.5954522808\text{E-}1\ x_4x_5 + 0.1602667555\text{E-}1\ x_1x_6 + \\
&\quad 0.5166013963\text{E-}2\ x_2x_6 + 0.1665205002\text{E-}2\ x_3x_6 + \\
&\quad 0.5202737589\text{E-}1\ x_4x_6 + 0.4545868627\text{E-}1\ x_5x_6 + \\
&\quad 0.1400323585\text{E-}1\ x_1x_7 + 0.4513781523\text{E-}2\ x_2x_7 + \\
&\quad 0.1454965400\text{E-}2\ x_3x_7 + 0.4545868627\text{E-}1\ x_4x_7 + \\
&\quad 0.3971932318\text{E-}1\ x_5x_7 + 0.3470458044\text{E-}1\ x_6x_7 + \\
&\quad 0.3107104497\text{E-}2\ x_1^2 + 0.3228346350\text{E-}3\ x_2^2 + \\
&\quad 0.3354319164\text{E-}4\ x_3^2 + 0.3407469747\text{E-}1\ x_4^2 + \\
&\quad 0.2601368795\text{E-}1\ x_5^2 + 0.1985966159\text{E-}1\ x_6^2 + \\
&\quad 0.1516148573\text{E-}1\ x_7^2 \\
f_3(\mathbf{x}) &:= -0.5828529757 + 0.1228340113x_1 + 0.2805508994\text{E-}1\ x_2 + \\
&\quad 0.6407737265\text{E-}2\ x_3 - 0.3414023228x_4 - 0.2406648259x_5 - \\
&\quad 0.1696519167x_6 - 0.1195927686x_7 + 0.1073433627\text{E-}2\ x_1x_2 + \\
&\quad 0.2451705080\text{E-}3\ x_1x_3 + 0.5599654833\text{E-}4\ x_2x_3 + \\
&\quad 0.1806580429\text{E-}1\ x_1x_4 + 0.4126200541\text{E-}2\ x_2x_4 + \\
&\quad 0.9424175444\text{E-}3\ x_3x_4 + 0.1273513199\text{E-}1\ x_1x_5 + \\
&\quad 0.2908683592\text{E-}2\ x_2x_5 + 0.6643386384\text{E-}3\ x_3x_5 + \\
&\quad 0.4705176468\text{E-}1\ x_4x_5 + 0.8977379819\text{E-}2\ x_1x_6 + \\
&\quad 0.2050419062\text{E-}2\ x_2x_6 + 0.4683124045\text{E-}3\ x_3x_6 + \\
&\quad 0.3316821239\text{E-}1\ x_4x_6 + 0.2338127636\text{E-}1\ x_5x_6 + \\
&\quad 0.6328426630\text{E-}2\ x_1x_7 + 0.1445402428\text{E-}2\ x_2x_7 + \\
&\quad 0.3301275819\text{E-}3\ x_3x_7 + 0.2338127636\text{E-}1\ x_4x_7 + \\
&\quad 0.1648216907\text{E-}1\ x_5x_7 + 0.1161877962\text{E-}1\ x_6x_7 + \\
&\quad 0.2349915088\text{E-}2\ x_1^2 + 0.1225852540\text{E-}3\ x_2^2 + \\
&\quad 0.6394760633\text{E-}5\ x_3^2 + 0.3337334754\text{E-}1\ x_4^2 + \\
&\quad 0.1658410619\text{E-}1\ x_5^2 + 0.8241084535\text{E-}2\ x_6^2 + \\
&\quad 0.4095214628\text{E-}2\ x_7^2
\end{aligned}
$$

Abbildung 6.1: *Funktion des 7er Systems, Teil 1*

$$
\begin{aligned}
f_4(\mathbf{x}) \;:=\; & -0.5196091865 \;+\; 0.7952023194\text{E-}1\; x_1 \;+ \\
& 0.1369550717\text{E-}1\; x_2 \;+\; 0.2358732012\text{E-}2\; x_3 \;- \\
& 0.3231249836 x_4 \;-\; 0.1842648187 x_5 \;-\; 0.1050786078 x_6 \;- \\
& 0.5992198558\text{E-}1\; x_7 \;+\; 0.5914814012\text{E-}3\; x_1x_2 \;+ \\
& 0.1018688901\text{E-}3\; x_1x_3 \;+\; 0.1754454283\text{E-}4\; x_2x_3 \;+ \\
& 0.1415543336\text{E-}1\; x_1x_4 \;+\; 0.2437943581\text{E-}2\; x_2x_4 \;+ \\
& 0.4198789790\text{E-}3\; x_3x_4 \;+\; 0.8072258394\text{E-}2\; x_1x_5 \;+ \\
& 0.1390258429\text{E-}2\; x_2x_5 \;+\; 0.2394396220\text{E-}3\; x_3x_5 \;+ \\
& 0.3198484963\text{E-}1\; x_4x_5 \;+\; 0.4603275220\text{E-}2\; x_1x_6 \;+ \\
& 0.7928069027\text{E-}3\; x_2x_6 \;+\; 0.1365425170\text{E-}3\; x_3x_6 \;+ \\
& 0.1823963734\text{E-}1\; x_4x_6 \;+\; 0.1040131107\text{E-}1\; x_5x_6 \;+ \\
& 0.2625057537\text{E-}2\; x_1x_7 \;+\; 0.4521049982\text{E-}3\; x_2x_7 \;+ \\
& 0.7786455212\text{E-}4\; x_3x_7 \;+\; 0.1040131107\text{E-}1\; x_4x_7 \;+ \\
& 0.5931437666\text{E-}2\; x_5x_7 \;+\; 0.3382453669\text{E-}2\; x_6x_7 \;+ \\
& 0.1717159416\text{E-}2\; x_1^2 \;+\; 0.5093444506\text{E-}4\; x_2^2 \;+ \\
& 0.1510819362\text{E-}5\; x_3^2 \;+\; 0.2804415971\text{E-}1\; x_4^2 \;+ \\
& 0.9119818669\text{E-}2\; x_5^2 \;+\; 0.2965718833\text{E-}2\; x_6^2 \;+ \\
& 0.9644367408\text{E-}3\; x_7^2 \\
f_5(\mathbf{x}) \;:=\; & -0.9708221316 \;+\; 0.8863865849 x_1 \;+\; 0.8095263737 x_2 \;+ \\
& 0.7393308528 x_3 \;-\; 0.9374687356\text{E-}2\; x_4 \;- \\
& 0.2969634573\text{E-}2\; x_5 \;-\; 0.9406958507\text{E-}3\; x_6 \;- \\
& 0.2979857157\text{E-}3\; x_7 \;+\; 0.4233432703\text{E-}3\; x_1x_2 \;+ \\
& 0.3866343967\text{E-}3\; x_1x_3 \;+\; 0.3531086170\text{E-}3\; x_2x_3 \;+ \\
& 0.2009129356\text{E-}3\; x_1x_4 \;+\; 0.1834914054\text{E-}3\; x_2x_4 \;+ \\
& 0.1675805281\text{E-}3\; x_3x_4 \;+\; 0.6364350907\text{E-}4\; x_1x_5 \;+ \\
& 0.5812486333\text{E-}4\; x_2x_5 \;+\; 0.5308474951\text{E-}4\; x_3x_5 \;+ \\
& 0.2648440225\text{E-}4\; x_4x_5 \;+\; 0.2016045525\text{E-}4\; x_1x_6 \;+ \\
& 0.1841230509\text{E-}4\; x_2x_6 \;+\; 0.1681574025\text{E-}4\; x_3x_6 \;+ \\
& 0.8389506080\text{E-}5\; x_4x_6 \;+\; 0.2657557139\text{E-}5\; x_5x_6 \;+ \\
& 0.6386259364\text{E-}5\; x_1x_7 \;+\; 0.5832495068\text{E-}5\; x_2x_7 \;+ \\
& 0.5326748692\text{E-}5\; x_3x_7 \;+\; 0.2657557139\text{E-}5\; x_4x_7 \;+ \\
& 0.8418385875\text{E-}6\; x_5x_7 \;+\; 0.2666705437\text{E-}6\; x_6x_7 \;+ \\
& 0.2317687279\text{E-}3\; x_1^2 \;+\; 0.1933171983\text{E-}3\; x_2^2 \;+ \\
& 0.1612449597\text{E-}3\; x_3^2 \;+\; 0.4180362680\text{E-}4\; x_4^2 \;+ \\
& 0.4194753040\text{E-}5\; x_5^2 \;+\; 0.4209192938\text{E-}6\; x_6^2 \;+ \\
& 0.4223682540\text{E-}7\; x_7^2
\end{aligned}
$$

Abbildung 6.2: *Funktion des 7er Systems, Teil 2*

$$
\begin{aligned}
f_6(\mathbf{x}) \;:=\; & -0.9802721137 + 0.9219460300x_1 + 0.8672034774x_2 + \\
& 0.8157113830x_3 - 0.4450172272\text{E-}2\, x_4 - \\
& 0.9941956844\text{E-}3\, x_5 - 0.2221093924\text{E-}3\, x_6 - \\
& 0.4962059574\text{E-}4\, x_7 + 0.2128632096\text{E-}3\, x_1x_2 + \\
& 0.2002239931\text{E-}3\, x_1x_3 + 0.1883352576\text{E-}3\, x_2x_3 + \\
& 0.6717220595\text{E-}4\, x_1x_4 + 0.6318370998\text{E-}4\, x_2x_4 + \\
& 0.5943203966\text{E-}4\, x_3x_4 + 0.1500668136\text{E-}4\, x_1x_5 + \\
& 0.1411562698\text{E-}4\, x_2x_5 + 0.1327748090\text{E-}4\, x_3x_5 + \\
& 0.4276502662\text{E-}5\, x_4x_5 + 0.3352584336\text{E-}5\, x_1x_6 + \\
& 0.3153517342\text{E-}5\, x_2x_6 + 0.2966270385\text{E-}5\, x_3x_6 + \\
& 0.9553968318\text{E-}6\, x_4x_6 + 0.2134414915\text{E-}6\, x_5x_6 + \\
& 0.7489878308\text{E-}6\, x_1x_7 + 0.7045150477\text{E-}6\, x_2x_7 + \\
& 0.6626829330\text{E-}6\, x_3x_7 + 0.2134414915\text{E-}6\, x_4x_7 + \\
& 0.4768413372\text{E-}7\, x_5x_7 + 0.1065292692\text{E-}7\, x_6x_7 + \\
& 0.1131501407\text{E-}3\, x_1^2 + 0.1001119965\text{E-}3\, x_2^2 + \\
& 0.8857622082\text{E-}4\, x_3^2 + 0.9571140702\text{E-}5\, x_4^2 + \\
& 0.4776984159\text{E-}6\, x_5^2 + 0.2384206686\text{E-}7\, x_6^2 + \\
& 0.1189964491\text{E-}8\, x_7^2 \\
f_7(\mathbf{x}) \;:=\; & 0.3617367006x_1 + 0.1724108153x_2 + 0.8217438040\text{E-}1\, x_3 - \\
& 0.2477761981x_4 - 0.2472344192x_5 - 0.2466938251x_6 - \\
& 0.2461544129x_7 - 0.7736815316 + 0.3187193360\text{E-}2\, x_1x_2 + \\
& 0.1519078946\text{E-}2\, x_1x_3 + 0.7240228578\text{E-}3\, x_2x_3 + \\
& 0.1746280745\text{E-}1\, x_1x_4 + 0.8323116969\text{E-}2\, x_2x_4 + \\
& 0.3966960997\text{E-}2\, x_3x_4 + 0.1742462389\text{E-}1\, x_1x_5 + \\
& 0.8304917930\text{E-}2\, x_2x_5 + 0.3958286979\text{E-}2\, x_3x_5 + \\
& 0.4371338635\text{E-}1\, x_4x_5 + 0.1738652381\text{E-}1\, x_1x_6 + \\
& 0.8286758688\text{E-}2\, x_2x_6 + 0.3949631927\text{E-}2\, x_3x_6 + \\
& 0.4361780417\text{E-}1\, x_4x_6 + 0.4352243100\text{E-}1\, x_5x_6 + \\
& 0.1734850704\text{E-}1\, x_1x_7 + 0.8268639147\text{E-}2\, x_2x_7 + \\
& 0.3940995796\text{E-}2\, x_3x_7 + 0.4352243100\text{E-}1\, x_4x_7 + \\
& 0.4342726636\text{E-}1\, x_5x_7 + 0.4333230981\text{E-}1\, x_6x_7 + \\
& 0.3343539697\text{E-}2\, x_1^2 + 0.7595394732\text{E-}3\, x_2^2 + \\
& 0.1725417563\text{E-}3\, x_3^2 + 0.2190458898\text{E-}1\, x_4^2 + \\
& 0.2180890209\text{E-}1\, x_5^2 + 0.2171363318\text{E-}1\, x_6^2 + \\
& 0.2161878045\text{E-}1\, x_7^2
\end{aligned}
$$

Abbildung 6.3: *Funktion des 7er Systems, Teil 3*

B.1.2 Reactor

Für dieses System definieren wir zunächst die folgenden Konstanten

$$\begin{array}{lcllcl}
\delta_1 &=& 2006.2696326884725 & \delta_2 &=& 0.02026048822046789 \\
\delta_3 &=& 0.12661204406731648 & & & \\
\gamma_1 &=& 0.070328 & \gamma_2 &=& 0.440539 \\
\gamma_3 &=& 0.02857142857142857 & \gamma_4 &=& -36.84211 \\
\gamma_5 &=& 0.1390824095604175E{-}3 & \gamma_6 &=& 0.2605108778006071E{-}4 \\
\gamma_7 &=& 0.1504208124796024E{-}4 & \gamma_8 &=& 0.1832123538232914E{-}3 \\
\gamma_9 &=& 19910.76346781887 & \gamma_{10} &=& 1.54924278922385 \\
\gamma_{11} &=& 516.5430719881741 & \gamma_{12} &=& 0.9697059518160267E{-}7 \\
\gamma_{13} &=& 167.137902 & \gamma_{14} &=& 0.489133 \\
\gamma_{15} &=& 0.3356676659347092E{-}11 & \gamma_{16} &=& 0.9597770416126526E{-}3 \\
\gamma_{17} &=& 0.5541820119662886E{-}3 & \gamma_{18} &=& 0.6749929692916622E{-}2
\end{array}$$

Damit ergibt sich das System $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ mit $\mathbf{f} = (f_1, \ldots, f_{29})^T : \mathbb{R}^{29} \to \mathbb{R}^{29}$ mit insgesamt 29 Gleichungen und 29 Unbekannten.

$$\begin{aligned}
f_1(\mathbf{x}) &:= x_{29}\delta_3 - \frac{1}{100}x_1x_2 - \frac{1}{6}x_{10} \\
f_2(\mathbf{x}) &:= \gamma_1 x_{29} - 100x_1x_3 + \frac{1}{6}x_{10} \\
f_3(\mathbf{x}) &:= \gamma_2 x_{29} - 100x_1x_4 \\
f_4(\mathbf{x}) &:= -100x_1x_5 + \frac{2}{3}x_{10} + x_{29}x_{14} \\
f_5(\mathbf{x}) &:= -x_1x_6 + x_7 - x_8 + x_9 \\
f_6(\mathbf{x}) &:= -1 + \frac{1}{10000}x_2 + x_3 + x_4 + x_5 + \frac{1}{100}x_6 \\
f_7(\mathbf{x}) &:= \gamma_3 x_8 - \gamma_3 x_9 - \gamma_3 x_{10} \\
f_8(\mathbf{x}) &:= \gamma_4\delta_1\Big(\gamma_5 - \gamma_6 x_4 - \gamma_7 x_5 - \gamma_8 x_6\Big) + x_7 \\
f_9(\mathbf{x}) &:= x_8 - \gamma_9\delta_2 x_6\Big(1 - \frac{1}{10}x_{11} - \frac{1}{10}x_{12}\Big) \\
f_{10}(\mathbf{x}) &:= x_9 - \gamma_{10}\delta_2 x_{11} \\
f_{11}(\mathbf{x}) &:= x_{10} - \gamma_{11}\delta_2 x_2 x_{11} \\
f_{12}(\mathbf{x}) &:= -\frac{\gamma_8}{100}\frac{x_2}{\gamma_{12} + \frac{\gamma_8}{100}x_2} + \frac{1}{10}x_{12} \\
f_{13}(\mathbf{x}) &:= -\gamma_{13}x_{29}\delta_3 + 35\delta_2 x_{13} \\
f_{14}(\mathbf{x}) &:= -\gamma_{14} + \delta_3 + x_{14}
\end{aligned}$$

$$
\begin{aligned}
f_{15}(\mathbf{x}) &:= -\frac{1}{100}x_{15}x_2 - 100x_{16}x_3 - 100x_{17}x_4 - 100x_{18}x_5 - x_{19}x_6 \\
f_{16}(\mathbf{x}) &:= -\frac{1}{100}x_{15}x_1 + \frac{1}{10000}x_{20} - \gamma_{11}x_{25}\delta_2 x_{11} \\
&\quad + x_{26}\Big(-\frac{\gamma_8}{100}\frac{1}{\gamma_{12} + \frac{\gamma_8}{100}x_2} + \gamma_{15}\frac{x_2}{(\gamma_{12} + \frac{\gamma_8}{100}x_2)^2}\Big) \\
f_{17}(\mathbf{x}) &:= -100x_{16}x_1 + x_{20} \\
f_{18}(\mathbf{x}) &:= -100x_{17}x_1 + x_{20} + \gamma_{16}x_{22}\delta_1 \\
f_{19}(\mathbf{x}) &:= -100x_{18}x_1 + x_{20} + \gamma_{17}x_{22}\delta_1 \\
f_{20}(\mathbf{x}) &:= -x_{19}x_1 + \frac{1}{100}x_{20} + \gamma_{18}x_{22}\delta_1 - \gamma_9 x_{23}\delta_2\Big(1 - \frac{1}{10}x_{11} - \frac{1}{10}x_{12}\Big) \\
f_{21}(\mathbf{x}) &:= x_{19} + x_{22} \\
f_{22}(\mathbf{x}) &:= -x_{19} + \gamma_3 x_{21} + x_{23} \\
f_{23}(\mathbf{x}) &:= x_{19} - \gamma_3 x_{21} + x_{24} \\
f_{24}(\mathbf{x}) &:= -\frac{1}{6}x_{15} + \frac{1}{6}x_{16} + \frac{2}{3}x_{18} - \gamma_3 x_{21} + x_{25} \\
f_{25}(\mathbf{x}) &:= \frac{\gamma_9}{10}x_{23}\delta_2 x_6 - \gamma_{10}x_{24}\delta_2 - \gamma_{11}x_{25}\delta_2 x_2 \\
f_{26}(\mathbf{x}) &:= \frac{\gamma_9}{10}x_{23}\delta_2 x_6 + \frac{1}{10}x_{26} \\
f_{27}(\mathbf{x}) &:= 35x_{27}\delta_2 \\
f_{28}(\mathbf{x}) &:= x_{18}x_{29} + x_{28} \\
f_{29}(\mathbf{x}) &:= x_{15}^2 + x_{16}^2 + x_{17}^2 + x_{18}^2 + x_{19}^2 + x_{20}^2 + x_{21}^2 + x_{22}^2 + x_{23}^2 + x_{24}^2 \\
&\quad + x_{25}^2 + x_{26}^2 + x_{27}^2 + x_{28}^2 - 1
\end{aligned}
$$

Für den Reactor ist als Startbox die Box in Abbildung 6.4 gegeben.

B.1.3 CSTR_Cusp

$[\mathbf{x}] = ([0,1],[0,20],[-20,0],[0.39,0.4],[-0.000999,0.001001],[0,1],[-1,1],[0,1],[-1,1])^T$ mit $B = 1.914$, $\gamma = 0.02542$ und $b_0 = 1.083$. Die Genauigkeitsanforderung liegt bei

$$
\begin{array}{rcr}
[x_1] = [\ 1E{-}6, & 100\ ] \\
[x_3] = [\ 1E{-}6, & 1\ ] \\
[x_5] = [\ 1E{-}6, & 1\ ] \\
[x_7] = [\ 1E{-}6, & 10\ ] \\
[x_9] = [\ 1E{-}6, & 1\ ] \\
[x_{11}] = [\ 1E{-}6, & 10\ ] \\
[x_{13}] = [\ 1E{-}6, & 1000\ ] \\
[x_{15}] = [\ 0, & 1\ ] \\
[x_{17}] = [\ -1, & 1\ ] \\
[x_{19}] = [\ -1, & 1\ ] \\
[x_{21}] = [\ -1, & 1\ ] \\
[x_{23}] = [\ -1, & 1\ ] \\
[x_{25}] = [\ -1, & 1\ ] \\
[x_{27}] = [\ -1, & 1\ ] \\
[x_{29}] = [\ 0, & 10\ ]
\end{array}
\qquad
\begin{array}{rcr}
[x_2] = [\ 1E{-}6, & 1\ ] \\
[x_4] = [\ 1E{-}6, & 1\ ] \\
[x_6] = [\ 1E{-}6, & 1\ ] \\
[x_8] = [\ 1E{-}6, & 10\ ] \\
[x_{10}] = [\ 1E{-}6, & 10\ ] \\
[x_{12}] = [\ 1E{-}6, & 10\ ] \\
[x_{14}] = [\ 1E{-}6, & 1\ ] \\
[x_{16}] = [\ -1, & 1\ ] \\
[x_{18}] = [\ -1, & 1\ ] \\
[x_{20}] = [\ -1, & 1\ ] \\
[x_{22}] = [\ -1, & 1\ ] \\
[x_{24}] = [\ -1, & 1\ ] \\
[x_{26}] = [\ -1, & 1\ ] \\
[x_{28}] = [\ -1, & 1\ ]
\end{array}
$$

Abbildung 6.4: *Startbox für den Reactor*

$\varepsilon_j = 10^{-8}$ für $j = 1, \ldots, 9$.

$$
\begin{aligned}
f_1(\mathbf{x}) &:= x_1\Big(1 + \exp(x_3 + \tfrac{x_2}{1+\gamma x_2})\Big) - x_4 \\
f_2(\mathbf{x}) &:= x_2 - \Big(b_0 + B\ \exp(x_3)\ \exp(\tfrac{x_2}{1+\gamma x_2})x_1\Big) \\
f_3(\mathbf{x}) &:= \Big(1 + \exp(x_3 + \tfrac{x_2}{1+\gamma x_2})\Big)x_6 + x_1\ \exp(x_3 + \tfrac{x_2}{1+\gamma x_2})\tfrac{x_7}{(1+\gamma x_2)^2} \\
f_4(\mathbf{x}) &:= \Big(1 - B\ \exp(x_3)\ \exp(\tfrac{x_2}{1+\gamma x_2})\tfrac{x_1}{(1+\gamma x_2)^2}\Big)x_7 - B\ \exp(x_3)\ \exp(\tfrac{x_2}{1+\gamma x_2})x_6 \\
f_5(\mathbf{x}) &:= 1 - \Big(x_6^2 + x_7^2\Big) \\
f_6(\mathbf{x}) &:= x_5x_6 + B\ \exp(x_3)\ \exp(\tfrac{x_2}{1+\gamma x_2})x_9 - \Big(1 + \exp(x_3 + \tfrac{x_2}{1+\gamma x_2})\Big)x_8 \\
f_7(\mathbf{x}) &:= x_5x_7 - \Bigg(x_1\ \exp(x_3 + \tfrac{x_2}{1+\gamma x_2})\tfrac{x_8}{(1+\gamma x_2)^2} \\
&\quad + \Big(1 - B\ \exp(x_3)\ \exp(\tfrac{x_2}{1+\gamma x_2})\tfrac{x_1}{(1+\gamma x_2)^2}\Big)x_9\Bigg) \\
f_8(\mathbf{x}) &:= 1 - \Big(x_8^2 + x_9^2\Big) \\
f_9(\mathbf{x}) &:= \exp(x_3 + \tfrac{x_2}{1+\gamma x_2})\Bigg(x_8\Big(\tfrac{2x_6x_7}{(1+\gamma x_2)^2} + x_1x_7^2\Big(\tfrac{1}{(1+\gamma x_2)^4} - \tfrac{2\gamma}{(1+\gamma x_2)^3}\Big)\Big) \\
&\quad - x_9\tfrac{2Bx_6x_7}{(1+\gamma x_2)^2}\Bigg) - x_1x_7^2x_9B\ \exp(x_3)\ \exp(\tfrac{x_2}{1+\gamma x_2})\Big(\tfrac{1}{(1+\gamma x_2)^4} - \tfrac{2\gamma}{(1+\gamma x_2)^3}\Big)
\end{aligned}
$$

B.1.4 Combustion

$[\mathbf{x}] = [-100000, 100000]^4$, $\varepsilon_j = 10^{-8}$ für $j = 1, \ldots, 4$

$$
\begin{array}{rcl}
f_1(\mathbf{x}) & := & -1.697E7 \cdot x_2 x4 + 2.177E7 \cdot x_2 + 0.55 x_1 x_4 + 0.45 x_1 - x_4 \\
f_2(\mathbf{x}) & := & 1.58510E14 x_2 x_4 + 4.126E7 x_1 x_3 - 8.285E6 x_1 x_4 + 2.284E7 x_3 x_4 \\
 & & -1.918E7 x_3 + 48.4 x_4 - 27.73 \\
f_3(\mathbf{x}) & := & x_1^2 - x_2 \\
f_4(\mathbf{x}) & := & x_4^2 - x_3
\end{array}
$$

B.1.5 HumanHeartDipole

$[\mathbf{x}] = [-3, 3]^8, \varepsilon_j = 10^{-8}$ für $j = 1, \ldots, 8$

$$
\begin{array}{rcl}
f_1(\mathbf{x}) & := & x_1 + x_2 - 1.4 \\
f_2(\mathbf{x}) & := & x_3 + x_4 - 1.8 \\
f_3(\mathbf{x}) & := & x_5 x_1 + x_6 x_2 - x_7 x_3 - x_8 x_4 - 2 \\
f_4(\mathbf{x}) & := & x_7 x_1 + x_8 x_2 - x_5 x_3 + x_6 x_4 - 2 \\
f_5(\mathbf{x}) & := & x_1(x_5^2 - x_7^2) - 2 x_3 x_5 x_7 + x_2(x_6^2 - x_8^2) - 2 x_4 x_6 x_8 - 1.2 \\
f_6(\mathbf{x}) & := & x_3(x_5^2 - x_7^2) + 2 x_1 x_5 x_7 + x_4(x_6^2 - x_8^2) + 2 x_2 x_6 x_8 - 1.3 \\
f_7(\mathbf{x}) & := & x_1 x_5(x_5^2 - 3x_7^2) + x_3 x_7(x_7^2 - 3x_5^2) + x_2 x_6(x_6^2 - 3x_8^2) + x_4 x_8(x_8^2 - 3x_6^2) - 1.9 \\
f_8(\mathbf{x}) & := & x_3 x_5(x_5^2 - 3x_7^2) - x_1 x_7(x_7^2 - 3x_5^2) + x_4 x_6(x_6^2 - 3x_8^2) - x_2 x_8(x_8^2 - 3x_6^2) - 1.91
\end{array}
$$

B.1.6 Eco9

$[\mathbf{x}] = [-100, 100]^9, \varepsilon_j = 10^{-8}$ für $j = 1, \ldots, 9$

$$
\begin{array}{rcl}
f_1(\mathbf{x}) & := & x_1 + x_2(x_1 + x_3) + x_4(x_3 + x_5) + x_6(x_5 + x_7) - (x_8((1/8) - x_7)) \\
f_2(\mathbf{x}) & := & x_2 + x_3(x_1 + x_5) + x_4(x_2 + x_6) + x_5 x_7 - (x_8((2/8) - x_6)) \\
f_3(\mathbf{x}) & := & x_3(1 + x_6) + x_4(x_1 + x_7) + x_2 x_5 - (x_8((3/8) - x_5)) \\
f_4(\mathbf{x}) & := & x_4 + x_1 x_5 + x_2 x_6 + x_3 x_7 - (x_8((4/8) - x_4)) \\
f_5(\mathbf{x}) & := & x_5 + x_1 x_6 + x_2 x_7 - (x_8((5/8) - x_3)) \\
f_6(\mathbf{x}) & := & x_6 + x_1 x_7 - (x_8((6/8) - x_2)) \\
f_7(\mathbf{x}) & := & x_7 - (x_8((7/8) - x_1)) \\
f_8(\mathbf{x}) & := & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + 1
\end{array}
$$

B.1.7 DesignProblem

$[\mathbf{x}] = [0,10]^9, \varepsilon_j = 10^{-8}$ für $j = 1,\ldots,9$

$$
\begin{aligned}
f_1(\mathbf{x}) &:= 23.3037x_2 + (1-x_1x_2)x_3(\exp(x_5(0.485 - 0.0052095x_7 - 0.0285132x_8)) \\
&\quad -1) - 28.5132 \\
f_2(\mathbf{x}) &:= -28.5132x_1 + (1-x_1x_2)x_4(\exp(x_6(0.116 - 0.0052095x_7 + 0.0233037x_9)) \\
&\quad -1) + 23.3037 \\
f_3(\mathbf{x}) &:= 101.779x_2 + (1-x_1x_2)x_3(\exp(x_5(0.752 - 0.0100677x_7 - 0.1118467x_8)) \\
&\quad -1) - 111.8467 \\
f_4(\mathbf{x}) &:= -111.8467x_1 + (1-x_1x_2)x_4(\exp(x_6(-0.502 - 0.0100677x_7 + 0.101779x_9)) \\
&\quad -1) + 101.779 \\
f_5(\mathbf{x}) &:= 111.461x_2 + (1-x_1x_2)x_3(\exp(x_5(0.869 - 0.0229274x_7 - 0.1343884x_8)) \\
&\quad -1) - 134.3884 \\
f_6(\mathbf{x}) &:= -134.3884x_1 + (1-x_1x_2)x_4(\exp(x_6(0.166 - 0.0229274x_7 + 0.111461x_9)) \\
&\quad -1) + 111.461 \\
f_7(\mathbf{x}) &:= 191.267x_2 + (1-x_1x_2)x_3(\exp(x_5(0.982 - 0.0202153x_7 - 0.2114823x_8)) \\
&\quad -1) - 211.4823 \\
f_8(\mathbf{x}) &:= -211.4823x_1 + (1-x_1x_2)x_4(\exp(x_6(-0.473 - 0.0202153x_7 + 0.191267x_9)) \\
&\quad -1) + 191.267 \\
f_9(\mathbf{x}) &:= x_1x_3 - x_2x_4
\end{aligned}
$$

B.1.8 DirectKinematics

Gegeben ist die Startbox

$$
\begin{pmatrix} [x_c] \\ [y_c] \\ [z_c] \\ [p] \\ [t] \\ [x_1] \\ [y_1] \\ [z_1] \\ [x_2] \\ [y_2] \\ [z_2] \end{pmatrix} = \begin{pmatrix} [-12.80624847 & , & 12.80624847] \\ [-12.80624847 & , & 12.80624847] \\ [0 & , & 12.80624847] \\ [-1.570796327 & , & 1.570796327] \\ [-1.570796327 & , & 1.570796327] \\ [-12 & , & -2] \\ [10 & , & 20] \\ [-5 & , & 5] \\ [2 & , & 12] \\ [10 & , & 20] \\ [-5 & , & 5] \end{pmatrix}
$$

mit der Genauigkeitsanforderung $\varepsilon = 10^{-8}$ für jede Variable.

$$
\begin{aligned}
f_1(\mathbf{x}) &:= x_c^2 + y_c^2 + z_c^2 - 164 \\
f_2(\mathbf{x}) &:= 304.0192 - 20x_c - 300\cos(p) + 100\sin(p)\cos(t) - 10y_c - 150\sin(p) \\
&\quad -50\cos(p)\cos(t) + 30x_c\cos(p) - 10x_c\sin(p)\cos(t) + 30y_c\sin(p)
\end{aligned}
$$

$$
\begin{aligned}
& +10y_c\cos(p)\cos(t) + 10z_c\sin(t) \\
f_3(\mathbf{x}) &:= 304.0192 + 20x_c - 300\cos(p) - 100\sin(p)\cos(t) - 10y_c + 150\sin(p) \\
& -50\cos(p)\cos(t) - 30x_c\cos(p) - 10x_c\sin(p)\cos(t) - 30y_c\sin(p) \\
& +10y_c\cos(p)\cos(t) + 10z_c\sin(t) \\
f_4(\mathbf{x}) &:= (x_1+7)^2 + (y_1-15)^2 + z_1^2 - 25 \\
f_5(\mathbf{x}) &:= (\cos(p)(x_1-x_c) + \sin(p)(y_1-y_c) + 7)^2 + (-\sin(p)\cos(t)(x_1-x_c) \\
& + \cos(p)\cos(t)(y_1-y_c) + \sin(t)(z_1-z_c) - 7.011678)^2 + (\sin(p)\sin(t)\cdot \\
& (x_1-x_c) - \cos(p)\sin(t)(y_1-y_c) + \cos(t)(z_1-z_c) + 4.065716)^2 - 25.4601 \\
f_6(\mathbf{x}) &:= (x_2-7)^2 + (y_2-15)^2 + z_2^2 - 25 \\
f_7(\mathbf{x}) &:= (\cos(p)(x_2-x_c) + \sin(p)(y_2-y_c) - 7)^2 + (-\sin(p)\cos(t)(x_2-x_c) \\
& + \cos(p)\cos(t)(y_2-y_c) + \sin(t)(z_2-z_c) - 7.011678)^2 + \\
& (\sin(p)\sin(t)(x_2-x_c) - \cos(p)\sin(t)(y_2-y_c) + \cos(t)(z_2-z_c) \\
& +4.065716)^2 - 25.46010 \\
f_8(\mathbf{x}) &:= (28.04672y_1 - 420.7008)\sin(t) + (243.9430 - 16.26286y_1)\cos(t) \\
& +28z_1\sin(p) + (-16.26286\sin(t)z_1 - 28.04672\cos(t)z_1)\cos(p) \\
& +(-60+4y_1)z_c + 60z_1 - 4z_1y_c \\
f_9(\mathbf{x}) &:= (-196.327 - 28.04672x_1)\sin(t) + (113.84 + 16.26286x_1)\cos(t) \\
& +(-16.26286\sin(t)z_1 - 28.04672\cos(t)z_1)\sin(p) - 28z_1\cos(p) \\
& +(-4x_1-28)z_c + 28z_1 + 4z_1x_c \\
f_{10}(\mathbf{x}) &:= 60z_2 - 4z_2y_c + (-420.7008 \\
& +28.04672y_2)\sin(t) + (243.943 - 16.26286y_2)\cos(t) - 28z_2\sin(p) \\
& +(-28.04672\cos(t)z_2 - 16.26286\sin(t)z_2)\cos(p) + (4y_2-60)z_c \\
f_{11}(\mathbf{x}) &:= -28z_2 + 4z_2x_c + (196.3270 - 28.04672x_2)\sin(t) + (-113.84 \\
& +16.26286x_2)\cos(t) + (-28.04672\cos(t)z_2 - 16.26286\sin(t)z_2)\sin(p) \\
& +28z_2\cos(p) + (-4x_2+28)z_c
\end{aligned}
$$

B.1.9 Griewank (G7)

$[\mathbf{x}] = [-3,3]^7, \varepsilon_j = 10^{-6}$ für $j = 1,\ldots,7$. Zur besseren Darstellung führen wir $\tilde{x}_i := \frac{x_i}{\sqrt{i}}$ mit $i = 1,\ldots,7$ ein.

$$
\begin{aligned}
f_1(\mathbf{x}) &:= \frac{x_1}{2000} + \frac{1}{\sqrt{1}}\sin(x_1)\cos(\tilde{x}_2)\cos(\tilde{x}_3)\cos(\tilde{x}_4)\cos(\tilde{x}_5)\cos(\tilde{x}_6)\cos(\tilde{x}_7)\\
f_2(\mathbf{x}) &:= \frac{x_2}{2000} + \frac{1}{\sqrt{2}}\cos(x_1)\sin(\tilde{x}_2)\cos(\tilde{x}_3)\cos(\tilde{x}_4)\cos(\tilde{x}_5)\cos(\tilde{x}_6)\cos(\tilde{x}_7)\\
f_3(\mathbf{x}) &:= \frac{x_3}{2000} + \frac{1}{\sqrt{3}}\cos(x_1)\cos(\tilde{x}_2)\sin(\tilde{x}_3)\cos(\tilde{x}_4)\cos(\tilde{x}_5)\cos(\tilde{x}_6)\cos(\tilde{x}_7)\\
f_4(\mathbf{x}) &:= \frac{x_4}{2000} + \frac{1}{\sqrt{4}}\cos(x_1)\cos(\tilde{x}_2)\cos(\tilde{x}_3)\sin(\tilde{x}_4)\cos(\tilde{x}_5)\cos(\tilde{x}_6)\cos(\tilde{x}_7)\\
f_5(\mathbf{x}) &:= \frac{x_5}{2000} + \frac{1}{\sqrt{5}}\cos(x_1)\cos(\tilde{x}_2)\cos(\tilde{x}_3)\cos(\tilde{x}_4)\sin(\tilde{x}_5)\cos(\tilde{x}_6)\cos(\tilde{x}_7)\\
f_6(\mathbf{x}) &:= \frac{x_6}{2000} + \frac{1}{\sqrt{6}}\cos(x_1)\cos(\tilde{x}_2)\cos(\tilde{x}_3)\cos(\tilde{x}_4)\cos(\tilde{x}_5)\sin(\tilde{x}_6)\cos(\tilde{x}_7)\\
f_7(\mathbf{x}) &:= \frac{x_7}{2000} + \frac{1}{\sqrt{7}}\cos(x_1)\cos(\tilde{x}_2)\cos(\tilde{x}_3)\cos(\tilde{x}_4)\cos(\tilde{x}_5)\cos(\tilde{x}_6)\sin(\tilde{x}_7)
\end{aligned}
$$

B.1.10 Brent

$[\mathbf{x}] = [-10^8, 10^8]^8, \varepsilon_j = 10^{-8}$ für $j = 1,\ldots,8$

$$
\begin{aligned}
f_1(\mathbf{x}) &:= 3x_1(x_2 - 2x_1) + x_2^2/4\\
f_2(\mathbf{x}) &:= 3x_2(x_3 - 2x_2 + x_1) + (x_3 - x_1)^2/4\\
f_3(\mathbf{x}) &:= 3x_3(x_4 - 2x_3 + x_2) + (x_4 - x_2)^2/4\\
f_4(\mathbf{x}) &:= 3x_4(x_5 - 2x_4 + x_3) + (x_5 - x_3)^2/4\\
f_5(\mathbf{x}) &:= 3x_5(x_6 - 2x_5 + x_4) + (x_6 - x_4)^2/4\\
f_6(\mathbf{x}) &:= 3x_6(x_7 - 2x_6 + x_5) + (x_7 - x_5)^2/4\\
f_7(\mathbf{x}) &:= 3x_7(x_8 - 2x_7 + x_6) + (x_8 - x_6)^2/4\\
f_8(\mathbf{x}) &:= 3x_8(20 - 2x_8 + x_7) + (20 - x_7)^2/4
\end{aligned}
$$

B.1.11 Robotics

$[\mathbf{x}] = [-1,1]^8, \varepsilon_j = 10^{-8}$ für $j = 1,\ldots,8$

$$\begin{aligned}
f_1(\mathbf{x}) &:= 4.731\text{E-}3x_1x_3 - 0.3578x_8x_3 - 0.1238x_1 - 1.637\text{E-}3x_8 - 0.9338x_4 + x_7 \\
&\quad -0.3571 \\
f_2(\mathbf{x}) &:= 0.2238x_1x_3 + 0.7623x_8x_3 + 0.2638x_1 - 0.07745x_8 - 0.6734x_4 - 0.6022 \\
f_3(\mathbf{x}) &:= x_6x_2 + 0.3578x_1 + 4.731\text{E-}3x_8 \\
f_4(\mathbf{x}) &:= -0.7623x_1 + 0.2238x_8 + 0.3461 \\
f_5(\mathbf{x}) &:= x_1^2 + x_8^2 - 1 \\
f_6(\mathbf{x}) &:= x_3^2 + x_4^2 - 1 \\
f_7(\mathbf{x}) &:= x_5^2 + x_6^2 - 1 \\
f_8(\mathbf{x}) &:= x_7^2 + x_2^2 - 1
\end{aligned}$$

B.2 Optimierungsprobleme

B.2.1 Trefethen

$[\mathbf{x}] = [-1,1]^2, \varepsilon_j = 10^{-12}$ für $j = 1,2$

$$\begin{aligned}
f(\mathbf{x}) &= -\sin(10(x_1+x_2)) + 0.25\cdot(x_1^2+x_2^2) + \exp(\sin(50x_1)) \\
&\quad + \sin(60\exp(x_2)) + \sin(70\sin(x_1)) + \sin(\sin(80x_2))
\end{aligned}$$

Es ergibt sich die Lösungsbox

$$\begin{pmatrix} [\ -0.0244030796943752216,\ -0.0244030796943751141\ ] \\ [\ \ 0.2106124271553556640,\ \ 0.2106124271553562190\ ] \end{pmatrix}$$

mit dem globalen Minimum $f^* = -3.30686864747521$.

B.2.2 HM2

$[\mathbf{x}] = [-1,1]^7, \varepsilon_j = 10^{-6}$ für $j = 1,\ldots,7$

$$f(\mathbf{x}) = \mathbf{y}^T\mathbf{A}\mathbf{y},$$

mit $y_i = \sin(x_i)$ für alle $i = 1,\ldots,7$ und

$$\mathbf{A} = \begin{pmatrix}
1 & -1 & & & & & 0 \\
-1 & 2 & -1 & & & & \\
 & -1 & 2 & -1 & & & \\
 & & -1 & 2 & -1 & & \\
 & & & -1 & 2 & -1 & \\
 & & & & -1 & 2 & -1 \\
0 & & & & & -1 & 2
\end{pmatrix}.$$

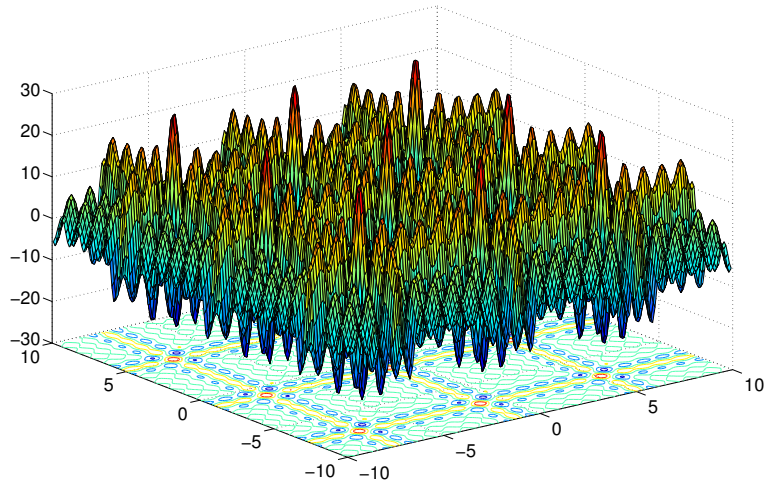

Abbildung 6.5: Darstellung der Funktion HM3 auf der Startbox.

B.2.3 HM3

$[\mathbf{x}] = [-10, 10]^2, \varepsilon_j = 10^{-6}$ für $j = 1, 2$

$$\begin{aligned} f(\mathbf{x}) \quad = \quad & -(\sin(2x_1 + 1) + 2\sin(3x_1 + 2) + 3\sin(4x_1 + 3) + 4\sin(5x_1 + 4) \\ & +5\sin(6x_1 + 5) + \sin(2x_2 + 1) + 2\sin(3x_2 + 2) + \\ & 3\sin(4x_2 + 3) + 4\sin(5x_2 + 4) + 5\sin(6x_2 + 5)) \end{aligned}$$

B.2.4 L3

$[\mathbf{x}] = [-50, 50]^2, \varepsilon_j = 10^{-6}$ für $j = 1, 2$

$$f(\mathbf{x}) = \sum_{i=1}^{5} i\cos((i+1)x_1 + i) \sum_{j=1}^{5} j\cos((j+1)x_2 + j)$$

B.2.5 Ratz

$[\mathbf{x}] = [-3, 3]^2, \varepsilon_j = 10^{-6}$ für $j = 1, 2$

$$f(\mathbf{x}) = \sin(x_1^2 + 2x_2^2)\exp(-x_1^2 - x_2^2)$$

B.2.6 Womersley

Sogenannte Minimale-Energie-Systeme (engl.: minimum energy system) sind Punkte $\mathbf{x}^{(j)}$ mit $j = 1, \ldots, m$ auf der *Sphäre*

$$S^2 := \{\mathbf{x} \in \mathbb{R}^3 \,|\, ||\mathbf{x}||_2 = 1\},$$

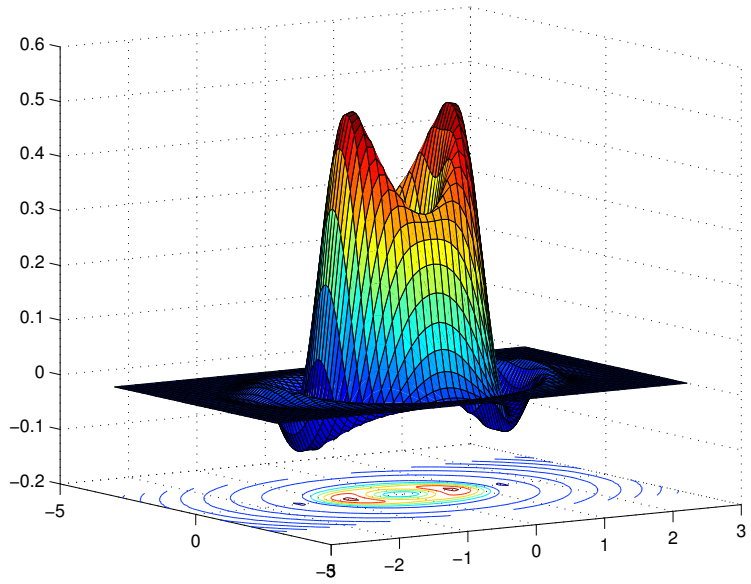

Abbildung 6.6: Darstellung der Funktion von Ratz auf der Startbox.

die die *potentielle Energie*

$$W_{\text{pot}} = \sum_{i=1}^{m} \sum_{j=i+1}^{m} \frac{1}{||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2}$$

minimieren. Dieses Problem entspricht auf den ersten Blick einem Optimierungsproblem mit Nebenbedingungen. Wir können allerdings diese Nebenbedingungen umgehen, wenn wir für die Punkte $\mathbf{x}^{(j)}$ mit $j = 1, \ldots, m$ statt karthesischer Koordinaten die sogenannten *Kugelkoordinaten*

$$\mathbf{x}^{(j)} = \begin{pmatrix} \sin\theta_j \cdot \cos\varphi_j \\ \sin\theta_j \cdot \sin\varphi_j \\ \cos\theta_j \end{pmatrix}$$

mit $\theta_j \in [0, \pi]$ und $\varphi_j \in [0, 2\pi]$ einsetzen. Dies hat den positiven Nebeneffekt, dass sich die Anzahl der Variablen pro Punkt $\mathbf{x}^{(j)}$ verringert.

Wir betrachten nun für eine unterschiedliche Anzahl von Punkten m die daraus resultierenden Optimierungsprobleme ohne Nebenbedingungen.

W3 $(m = 3)$
In diesem Fall suchen wir also nach 3 Punkten auf der Einheitskugel, die maximalen Abstand voneinander haben. Es ist offensichtlich, dass wir bei der obigen Problemstellung noch zu viele Freiheitsgrade haben. Hat man eine Lösung des Optimierungsproblems berechnet, so ergeben sich weitere Lösungen durch Drehung der Punkte auf der Einheitskugel. In diesem Fall würde man als Lösungsmenge

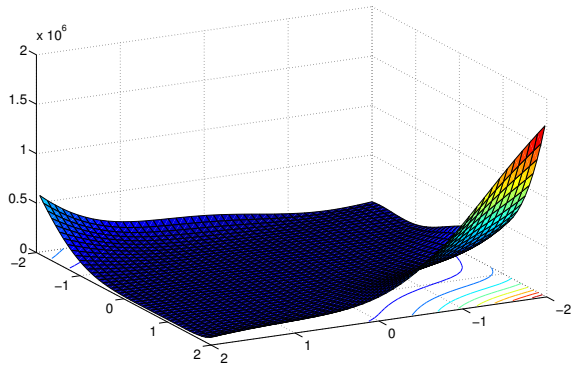

Abbildung 6.7: Darstellung der Goldstein-Price-Funktion auf der Startbox.

eine vollständige *Überdeckung* der Einheitskugel erhalten. Aus diesem Grund wird hier o.B.d.A. der Punkt $\mathbf{x}^{(1)}$ durch $\theta_1 = 0, \varphi_1 = 0$ festgehalten und mit $\varphi_2 = 0$ eine Ebene festgelegt. Man erhält das Optimierungsproblem

$$\begin{aligned} f(\mathbf{x}) &= \frac{1}{\sqrt{\sin^2\theta_2 + (1-\cos\theta_2)^2}} + \frac{1}{\sqrt{\sin^2\theta_3 + (1-\cos\theta_3)^2}} \\ &\quad + \frac{1}{\sqrt{(\sin\theta_2 - \sin\theta_3\cos\varphi_3)^2 + \sin^2\theta_3\sin^2\varphi_3 + (\cos\theta_2 - \cos\theta_3)^2}} \end{aligned}$$

mit der Startbox $\theta_2, \theta_3 \in [0, \pi]$ und $\varphi_3 \in [0, 2\pi]$. Als Genauigkeitsanforderung wird $\varepsilon_j = 10^{-8}$ für $j = 1, 2, 3$ festgelegt.

W4 $(m = 4)$
gleiche Einstellungen wie bei **W3**

B.2.7 Siirola

$[\mathbf{x}] = [\tilde{x}_i - 20, \tilde{x}_i + 20]$ und $\varepsilon_i = 10^{-6}, \tilde{x}_i = 3$ für $i = 1, \ldots, n$

$$f(\mathbf{x}) = 100 \prod_{i=1}^{n} \sum_{j=1}^{5} \left(\frac{j^5}{4425} \cos(j + j x_i) \right) + \frac{1}{n} \sum_{i=1}^{n} (x_i - \tilde{x}_i)^2$$

B.2.8 GP

$[\mathbf{x}] = [-2, 2]^2$, $\varepsilon_j = 10^{-6}$ für $j = 1, 2$

$$\begin{aligned} f(\mathbf{x}) &= (1 + (x_1 + x_2 + 1)^2 (19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)) \cdot \\ &\quad (30 + (2x_1 - 3x_2)^2 (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)) \end{aligned}$$

C Matlab-Code für spherical t-designs

```
syms phi theta x y z

array_phi = sym( zeros(1:N) );
array_theta = sym( zeros(1:N) );

x = cos( phi ) * sin( theta );
y = sin( phi ) * sin( theta );
z = cos( theta );

% Create variables
for i = 1 : N
    eval(['syms phi_' num2str(i) ';']);
    eval(['array_phi(i)=phi_' num2str(i) ';']);
    eval(['syms theta_' num2str(i) ';']);
    eval(['array_theta(i)=theta_' num2str(i) ';']);
end

for i = 0 : 3
 for j = 0 : 3
   for k = 0 : 3
     if ( i + j + k <= 3 )
       f = x^i * y^j * z^k * sin( theta );
       lhs = int( int( f, theta, 0, pi ), phi, 0, 2*pi ) / ( 4 * pi )
       rhs = 0.0;
       for p = 1 : N
         rhs = rhs + eval(['(cos( array_phi(p) ) * sin( array_theta(p) ))^i *
                           (sin( array_phi(p) ) * sin( array_theta(p) ))^j *
                            cos( array_theta(p) )^k']);
       end
       rhs = rhs / N
     end
   end
 end
end
```

Tabellenverzeichnis

Abbildungsverzeichnis

Symbolverzeichnis

$\exists$	Existenzquantor (es existiert)
$\mathbb{N}$	Menge der natürlichen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{R}^n$	Menge aller reellen Spaltenvektoren der Dimension n ($n \in \mathbb{N}$)
$\mathbb{IR}_*$	Menge der nichtleeren, reellen Intervalle
$\mathbb{IR}$	Menge der nichtleeren, reellen, beschränkten Intervalle
$\mathbb{IR}_*^n$	Menge der nichtleeren, reellen Intervallspaltenvektoren der Dimension n ($n \in \mathbb{N}$)
$\mathbb{IR}^n$	Menge der nichtleeren, reellen, beschränkten Intervallspaltenvektoren der Dimension n ($n \in \mathbb{N}$)
$\mathbb{ID}$	Menge aller nichtleeren, reellen Intervallvektoren in einer beliebigen Menge $D \subseteq \mathbb{R}^n$
$\mathbb{R}^{m \times n}$	Menge aller reellen $m \times n$ Matrizen ($m, n \in \mathbb{N}$)
$\mathbb{IR}_*^{m \times n}$	Menge aller nichtleeren, reellen $m \times n$ Intervallmatrizen ($m, n \in \mathbb{N}$)
$[x]$	Intervall
$[x_i]$	i-te Komponente des Intervallvektors $[\mathbf{x}]$
$\underline{x} = \inf([x])$	untere Grenze oder Infimum von $[x]$
$\overline{x} = \sup([x])$	obere Grenze oder Supremum von $[x]$
$\check{[x]} = \mathrm{mid}([x])$	Mittelpunkt von $[x]$
$\mathrm{diam}([x])$	Durchmesser von $[x]$
$\lvert[x]\rvert$	Magnitude von $[x]$
$\langle[x]\rangle$	Mignitude von $[x]$
$\mathrm{int}([x])$	Innere von $[x]$
$[\mathbf{x}^{(0)}]$	Startbox
$\mathrm{box}(D)$, $\square(D)$	engstmögliche Intervalleinschließung der Menge D
$\oslash$	relationale Division nach Ratz

Algorithmenverzeichnis

Literaturverzeichnis

[1] G. Alefeld, A. Frommer, G. Heindl, and J. Mayer. On the Existence Theorems of Kantorovich, Miranda and Borsuk.

[2] D. An Mey. Two OpenMP programming patterns. In D. an Mey, editor, *Proc. EWOMP '03, September 22-26, 2003, Aachen, Germany*, pages 51–60, Aachen, Germany, 2004. Center for Computing and Communication, Aachen University.

[3] D. S. Arnon, G. E. Collins, and S. M. Callum. Cylindrical Algebraic Decomposition I: The basic algorithm. *SIAM J. of Computing*, 13(4):865–877, 1984.

[4] T. Beelitz. *Methoden zum Einschluss von Funktions- und Ableitungswerten.* Diplomarbeit, RWTH Aachen, Deutschland, April 2002.

[5] T. Beelitz, C. Bischof, B. Lang, and K. Schulte Althoff. Result-Verifying Solution of Nonlinear Systems in the Analysis of Chemical Processes. In R. Alt et al., editors, *Numerical Software with Result Verification*, number 2991 in LNCS, pages 198–205, Heidelberg, 2004. Springer.

[6] T. Beelitz, C. H. Bischof, and B. Lang. Efficient Task Scheduling in the Result-Verifying Solution of Nonlinear Systems. *Reliable Computing*, 12(2):141–151, 2006.

[7] E.-P. Beisel and M. Mendel. *Optimierungsmethoden des Operations Research.* Vieweg, Braunschweig, Wiesbaden, 1987.

[8] S. Berner. *Ein paralleles Verfahren zur verifizierten globalen Optimierung.* Dissertation, Bergische Universität Wuppertal, Deutschland, August 1995.

[9] W.-J. Beyn, A. Champneys, E. Doedel, W. Govaerts, Y. A. Kuznetsov, and B. Sanstede. Numerical continuation, and computation of normal forms. In B. Fiedler, editor, *Handbook of Dynamical Systems*, volume 2, pages 149–219. Elsevier Science, North-Holland, 2002.

[10] K. Borsuk. Drei Sätze über die n-dimensionale Sphäre. *Fund. Math.*, 20:177–190, 1933.

[11] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Teubner Verlag, Leipzig, 1987.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[13] K. Deimling. *Nonlinear Functional Analysis*. Springer, Berlin, Heidelberg, New York, 1985.

[14] P. Deuflhard and A. Hohmann. *Numerische Mathematik I*. Lehrbuch. de Gruyter, 1993.

[15] H. Fischer. Special problems in automatic differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 43–50. SIAM, Philadelphia, PA, 1991.

[16] M. Flynn. Very high-speed computing systems. In *Proc. of the IEEE*, volume 54, pages 1901–1909, Dezember 1966.

[17] A. Frommer and B. Lang. Existence tests for solutions of nonlinear equations using Borsuk's theorem, 2003. To appear in SIAM J. Numer. Anal.

[18] A. Frommer and B. Lang. On preconditioners for the Borsuk existence test. *Proc. Appl. Math. Mech.*, 4(1):638–639, 2004.

[19] GlobSol home page, `http://www.mcsu.mu.edu/~globsol`.

[20] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[21] G. H. Golub and C. F. Van Loan. Number second edition. John Hopkins University Press, Baltimore, MD, UDA, 1989.

[22] M. Golubitsky and D. G. Schaeffer. *Singularities and Groups in Bifurcation Theory, Volume I*. Springer-Verlag, New York, 1985.

[23] E. Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, 1992.

[24] E. R. Hansen. Sharpness in interval computations. *Reliable Computing*, 3:17–29, 1997.

[25] R. H. Hardin and N. J. A. Sloane. Mclaren's improved snub cube and other new spherical designs in three dimensions. *DISCRETE COMPUTATIONAL GEOMETRY*, 15:429, 1996.

[26] T. J. Hickey, Q. Ju, and M. van Emden. Interval arithmetic: from principles to implementation. Technical report, Brandeis University CS, April 1999.

[27] T. J. Hickey, M. van Emden, and H. Wu. A unified framework for interval constraints and interval arithmetic. In M. Maher and J.-F. Puget, editors, *Principles and Practice of Constraint Programming*, volume 1520 of *LNCS*, pages 250–264. Springer-Verlag, 1998.

[28] W. Hofschuster and W. Krämer. C-XSC 2.0—A C++ library for extended scientific computing. In R. Alt et al., editors, *Numerical Software with Result Verification*, number 2991 in LNCS, pages 15–35. Springer, Heidelberg.

[29] W. Hofschuster, W. Krämer, M. Lerch, G. Tischler, and J. Wolff v. Gudenberg. filib++—A fast interval library supporting containment computations, 2004. To appear in ACM Trans. Math. Software.

[30] C. Hu. *Optimal Preconditioners for the Interval Newton Method*. Dissertation, University of Southwestern Louisiana, USA, 1990.

[31] S. Ibraev. *A new parallel method for verified global optimization*. PhD thesis, Bergische Universiät Wuppertal, Deutschland, 2001.

[32] Intel Corporation. KAI C++ User's guide, 2001. `http://www.kai.com/kpts/guide`.

[33] Intel Corporation. Intel C++ Compiler User's Guide, 2003. `http://developer.intel.com/software/products/compilers`.

[34] C.-H. Jan. *Expression Parsing and Rigorous Computation of Bounds on all Solutions to Practical Nonlinear Systems*. Doktorarbeit, University of Southwestern Louisiana, may 1992.

[35] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer-Verlag, 2001.

[36] R. B. Kearfott. Preconditioners for the interval Gauss-Seidel method. *SIAM J. Numer. Anal.*, 27(3):804–822, Juni 1990.

[37] R. B. Kearfott. Decomposition of arithmetic expressions to improve the behavior of interval iteration for nonlinear systems. *Computing*, 1991.

[38] R. B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer Academic Publishers, 1996.

[39] R. B. Kearfott, M. Dawande, D. K.-S., and C.-Y. Hu. Algorithm 737: INTLIB, a portable FORTRAN 77 interval standard function library. *ACM Trans. Math. Software*, 3(5):96–105, 1994.

[40] R. B. Kearfott and K. Du. The cluster problem in multivariate global optimization. *Journal Global Optimization 5*, pages 253–265, 1994.

[41] R. B. Kearfott and M. Novoa III. Algorithm 681: INTBIS, a portable interval Newton/bisection package. *ACM Trans. Math. Software*, 16(2):152–157, Juni 1990.

[42] A. Kienitz. *Untersuchungen zum Einsatz von Taylormodellen bei der verifizierten Lösung von Gleichungssystemen.* Diplomarbeit, RWTH Aachen, Germany, Apr. 2003.

[43] O. Knüppel. A PROFIL/BIAS Implementation of a Global Minimization Algorithm.

[44] W. Krämer. Advanced Software Tools for Validated Computing. In *Thirty First Spring Conference of the Union of Bulgarian Mathematicians*, pages 344–355, Borovets, 2002.

[45] Y. A. Kuznetsov. *Elements of Applied Bifurcation Theory.* Springer Verlag, 2nd edition, 1999.

[46] Y. Lin and M. A. Stadtherr. Lp-based strategies for modeling and optimization using interval methods. In *Fourth International Conference on Foundations of Computer Aided Process Operations (FOCAPO 2003)*, Coral Springs, Florida, Januar 2003.

[47] J.-P. Merlet. http://www.sop-inria.fr.

[48] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994. `http://www.mpi-forum.org`.

[49] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, 1997. `http://www.mpi-forum.org`.

[50] C. Miranda. Un' osservazione su un teorema di Brouwer. *Bolletino Unione Mathematica Italiana*, pages 5–7, 1940.

[51] M. Mönnigmann, W. Marquardt, C. H. Bischof, T. Beelitz, B. Lang, and P. Willems. A Hybrid Approach for Efficient Robust Design of Dynamic Systems, 2004. Submitted for publication.

[52] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[53] R. E. Moore, E. Hansen, and A. Leclerc. Rigorous methods for global optimization. *Recent advances in global optimization*, 1992.

[54] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, UK, 1990.

[55] M. Novoa III. *Existence / Uniqueness Theory with Interval Newton Methods, and Formulas for Slopes of Powers.* Dissertation, University of Southwestern Louisiana, USA, 1993.

[56] J. B. Oliveira. New slope methods for sharper interval functions and a note on Fischer's acceleration method. *Reliable Comput.*, 2(3):299–320, 1996.

[57] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, Version 2.5, 2005. `http://www.openmp.org`.

[58] J. Ortega and W. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York, 1970.

[59] D. Ratz. On extended interval arithmetic and inclusion isotonicity. 1996.

[60] D. Ratz. *Automatic Slope Computation and its Application in Nonsmooth Global Optimization*. Habilitationsschrift, Karlsruhe, Germany, 1998.

[61] H. Schichl and A. Neumaier. Exclusion regions for systems of equations. 2003.

[62] K. Schulte Althoff. *Algorithmen zum verifizierten Lösen nichtlinearer Gleichungssysteme*. Diplomarbeit, Aachen University, Germany, May 2002.

[63] V. Stahl. A sufficient condition for non-overestimation in interval arithmetic. *Computing*, 59:349–363, 1997.

[64] Sun Microsystems. Sun Studio 9: C++ interval arithmetic programming reference. `http://docs.sun.com/app/docs/doc/817-6705`.

[65] H. Tapamo. *Some New Acceleration Mechanism in Verified Global Optimization*. Dissertation, Bergische Universität Wuppertal, Deutschland, 2005.

[66] P. R. Willems. *Symbolisch-numerische Techniken zum Verifizierten Lösen nichtlinearer Gleichungssysteme*. Diplomarbeit, Bergische Universität Wuppertal/RWTH Aachen, Deutschland, Mai 2004.

[67] M. Wolfe. *Numerical Methods for Unconstrained Optimization–An Introduction*. Van Nostrand Reinhold, 1978.

Index