TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Simulation
und Systemoptimierung

SICOM

a Solver-Independent Continuation Method

Diplomarbeit

Autor : André König
Betreuung : Dipl.-Math. Markus Glocker
Aufgabenstellung : Prof. Dr. Oskar von Stryk

Gib mir nicht, was ich mir wünsche,
sondern was ich brauche.
Lehre mich die Kunst der kleinen Schritte.

Antoine de Saint - Exupéry

Inhalt

1 Vorwort

Ein Schwachpunkt vieler numerischer Lösungsverfahren ist das schlechte Konvergenzverhalten bei sensitiven Problemstellungen oder ungünstiger Startnäherung. So kann z.B. der Konvergenzradius des Newton - Verfahrens zur Nullstellenbestimmung, wie in Abbildung 1-1 dargestellt, so klein werden, daß die Vorgabe einer ausreichend genauen Startnäherung nahezu unmöglich wird.

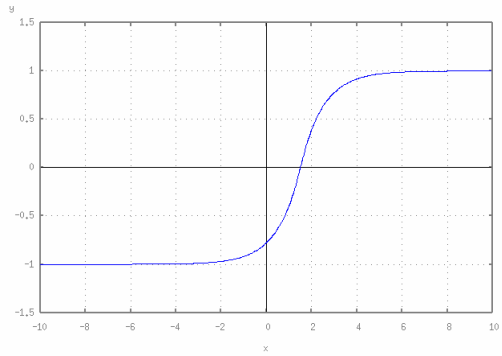

Abbildung 1-1:
Funktion mit minimalem Konvergenzradius für das Newton-Verfahren

Der zusätzliche Einsatz eines Fortsetzungs- oder Homotopieverfahrens ermöglicht die Erzeugung globaler Konvergenz bzw. die Beschleunigung des Konvergenzverhaltens. Erreicht wird dies durch die schrittweise Überführung eines Problems mit bekannter oder leicht zu berechnender Lösung in das Problem, dessen Lösung allein mit dem entsprechenden numerischen Lösungsverfahren nicht, oder nur schwer, also ineffizient berechnet werden kann.

Im Rahmen dieser Arbeit wurde SICOM, ein für die Verwendung mit verschiedenen externen, numerischen Lösungsverfahren geeignetes Homotopieverfahren entwickelt.

Im weiteren Verlauf werden zunächst Ursprung und Grundlagen von Homotopieverfahren erläutert. Bevor die Implementierung von SICOM dargestellt und dessen Funktionalität anhand mehrerer Beispiele veranschaulicht wird, soll die Vorgehensweise anderer, in der Praxis verwendeter Verfahren gezeigt werden. Vorgestellt werden das Homotopieverfahren von E. Grigat zur Berechnung von Optimalflugbahnen in Fallwindgebieten sowie HOMPACK von L. T. Watson, S.C. Billups und A.P. Morgan, ein Softwarepaket zur Lösung nichtlinearer Gleichungssysteme. Ziel ist dabei nicht die detailgetreue Wiedergabe und Bewertung der Verfahren, sondern die Darstellung der grundlegenden Konzepte.

2 Ursprung und Bedeutung des Begriffs "Homotopie"

2.1 Etymologischer Ursprung

Aus sprachwissenschaftlicher Sicht stammt der Begriff Homotopie aus dem Griechischen. Abgeleitet aus den Worten

> Homeios - <gr.> das Gleiche, das Ähnliche und
> Topos - <gr.> der Ort, der Platz,

ergibt sich die Bedeutung des Wortes demnach zu “Ortsähnlichkeit“.

2.2 Mathematischer Ursprung

Der Ursprung numerischer Homotopieverfahren liegt im Bereich der Topologie. Dort findet sich folgende Definition [1]:

> Zwei stetige Abbildungen $f, g: X \to Y$ zwischen topologischen Räumen heißen homotop, wenn es eine Homotopie h zwischen ihnen gibt, d.h. eine stetige Abbildung $h: X \times [0,1] \to Y$ mit $h(x,0)=g(x)$ und $h(x,1)=f(x)$ für alle $x \in X$.

Anschaulich ist eine Homotopie also eine Deformation einer Abbildung in eine Andere, was durch folgendes Beispiel verdeutlicht werden soll.

Gegeben seien dafür die stetigen Funktionen $f, g: X \to Y$. Eine Homotopie zwischen f und g kann dann z.B. beschrieben werden durch die Funktionenschar

$h(x,p)=p \cdot f(x) + (1 - p) \cdot g(x)$ mit $x \in X$ und $p \in [0,1]$.

Der Scharparameter p gibt dabei den Grad der Deformation an. Für $p=0$ gilt $h(x,p) = h(x,0) = g(x)$, $p=1$ ergibt $h(x,p) = h(x,1) = f(x)$.

Wächst p von 0 auf 1 wird g schrittweise in f deformiert. Im Bezug auf die etymologische Bedeutung des Begriffs Homotopie gilt also für zwei Werte $p_1, p_2 \in [0,1]$ des Parameters p, daß $h(x, p_1)$ und $h(x, p_2)$ um so ähnlicher sind, je kleiner $|p_1 - p_2|$. Der entsprechende Effekt wird in Abbildung 2-1 dargestellt.

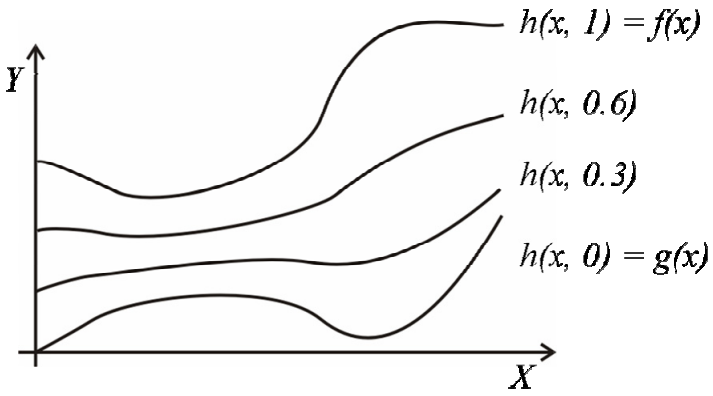

Abbildung 2-1: Homotopie zwischen f und g

2.3 Analogie Morphing

Der Begriff Morphing aus dem Bereich der Bildverarbeitung bezeichnet in Analogie zur topologischen Homotopie ein Verfahren, daß es ermöglicht, ein Bild schrittweise in ein anderes zu überführen.

Abbildung 2-2: Morphing [2]

Eine einfache Variante dieses Verfahrens, die z.B. in Präsentationssoftware als Überblendungseffekt zum Folienübergang eingesetzt wird, kann mathematisch als Homotopie beschrieben werden.
Ein Bild bzw. eine Folie sei dafür dargestellt durch eine endliche Menge B von Bildpunkten $b(\vec{s},\vec{c})$, wobei $\vec{s}$ die Position und $\vec{c}$ den Farbwert des jeweiligen Bildpunktes angibt. Die Homotopie h zwischen zwei Bildern B_1 mit Bildpunkten $b_1(\vec{s}_1,\vec{c}_1)$ und B_2 mit Bildpunkten $b_2(\vec{s}_2,\vec{c}_2)$ kann dann definiert werden als

$$h(\vec{s}_1,\vec{s}_2,\vec{c}_1,\vec{c}_2,p) = p \cdot \vec{c}_1 + (1 - p) \cdot \vec{c}_2$$

für $p \in [0,1]$ und alle $b_1 \in B_1$, $b_2 \in B_2$ mit $\vec{s}_1 = \vec{s}_2$.
Wächst p nun von 0 auf 1, werden die Farbwerte der Bildpunkte mit gleicher Position ineinander überführt und so B_1 mit B_2 überblendet. Im Bezug auf 2.1 gilt auch hier, das für $p_1, p_2 \in [0,1]$ die entsprechen Bilder $h(\vec{s}_1,\vec{s}_2,\vec{c}_1,\vec{c}_2,p_1)$ und $h(\vec{s}_1,\vec{s}_2,\vec{c}_1,\vec{c}_2,p_2)$ ähnlicher werden, je kleiner $|p_1 - p_2|$.

2.4 Homotopie, Psychologie und Bionik

Das Lösen komplexer Aufgaben, indem sie in einfache Teilaufgaben zerlegt bzw. auf bekannte Probleme zurückgeführt werden, ist ein Verfahren, das sicher nicht nur in der numerischen Mathematik Anwendung findet. Eine ähnliche Vorgehensweise beschreiben z.B. nach [3] Versuchspersonen bei der Entscheidung, ob zwei räumlich unterschiedlich angeordnete, dreidimensionale Objekte deckungsgleich sind. Hierfür vollzogen die Probanden nach eigenen Angaben eine schrittweise "geistige Drehung" der Objekte mit einer Schrittweite von durchschnittlich 60° pro Sekunde, versuchten also, eine Homotopie zwischen den beiden Objekten zu finden.

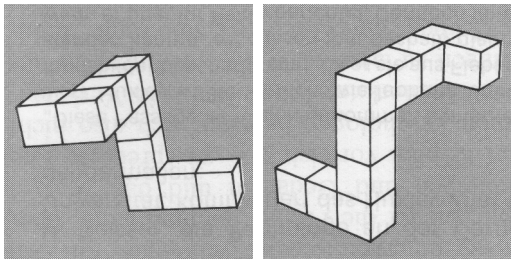

Abbildung 2-3:
Deckungsgleiche Objekte mit unterschiedlicher räumlicher Anordnung

In [4] wird dieses und verwandte Probleme der Problemklasse der Umformung zugeordnet. Alle Probleme dieser Klasse werden gelöst, indem ein Startzustand über Zwischenschritte in einem gegebenen Aktionsraum zur gesuchten Lösung umgeformt wird, wobei die Lösung eines Teilproblems jeweils Grundlage für den nächsten Schritt ist.

Betrachtet man nun folgende Definition der Bionik [5],

> Bionik als Wissenschaftsdisziplin befaßt sich systematisch mit der technischen Umsetzung und Anwendung von Konstruktionen, Verfahren und Entwicklungsprinzipien biologischer Systeme.

kann ein numerisches Homotopieverfahren als bionischer Ansatz zur Lösung eines mathematischen Problems bezeichnet werden.

3 Anwendung von Homotopieverfahren in der Numerik

Wie bereits im Vorwort angeführt, sind die meisten numerischen Lösungsverfahren nicht in der Lage, für jedes Problem bzw. für beliebige Startnäherungen eines Problems eine Lösung zu berechnen. Dies soll am Beispiel des Newton - Verfahrens, welches auch Grundlage für viele weitere Lösungs- und Optimierungsverfahren ist, verdeutlicht werden.

3.1 Das Newton-Verfahren

Als die wohl bekannteste Methode zur Lösung nichtlinearer Gleichungen wird das Newton - Verfahren vielfach in Literatur über numerische Mathematik, wie z.B. [6], beschrieben, weshalb nachfolgend nur kurz die Funktionsweise geometrisch erläutert werden soll.

Gegeben sei die Funktion

$$f(x)=ln(x).$$

Um eine Lösung für das Problem $f(x)=0$ zu finden, wird eine Startnäherung x_0, also eine Schätzung der Nullstelle benötigt. Hier sei $x_0=0.1$.

Folgende Schritte beschreiben das Newton-Verfahren zur Lösung der Gleichung:

1. Bilde die Tangente von f am Punkt x_0
2. Bestimme den Schnittpunkt $x_S = x_0 - \frac{f(x_0)}{f'(x_0)}$ der Tangente

   mit der x-Achse

3. Wiederhole ab Schritt 1 mit $x_0=x_S$ bis $f(x_S)$ in gewünschtem Toleranzbereich liegt

Bei einer Toleranz von 0.01 wird die Nullstelle von f, wie in der folgenden Abbildung dargestellt, in 4 Newton-Iterationen gefunden.

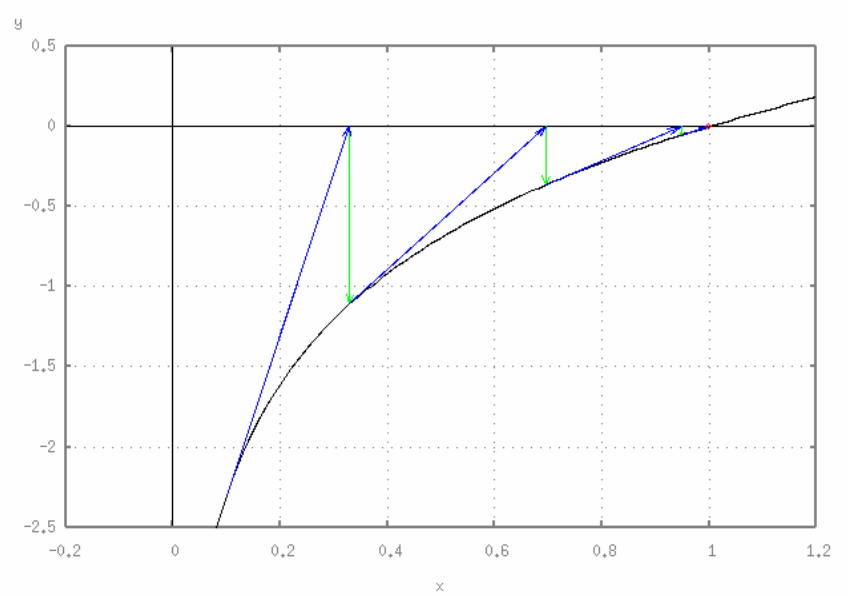

Abbildung 3-1: Newton-Verfahren für $f(x)=ln(x)$

3.2 Problemfälle im Newton-Verfahren

Die folgenden Beispiele zeigen einige Probleme, deren Lösung ausgehend von der gegebenen Startnäherung durch das Newton - Verfahren nicht bzw. nur ineffizient berechnet werden kann.

3.2.1 Newton-Iteration verläßt Definitionsbereich

Ist eine Funktion nicht über ganz $\mathbb{R}$ definiert, besteht die Möglichkeit, daß ihr Definitionsbereich während des Newton - Verfahrens verlassen wird.

Gegeben sei wieder

$$f(x)=ln(x)$$

mit dem zu lösenden Problem $f(x)=0$. Als Startnäherung wird $x_0=4.0$ gewählt.

Da das Newton - Verfahren, wie in Abbildung 3-2 gezeigt, bereits in der ersten Iteration den Definitionsbereich von f verläßt, kann für diesen Fall keine Lösung berechnet werden.

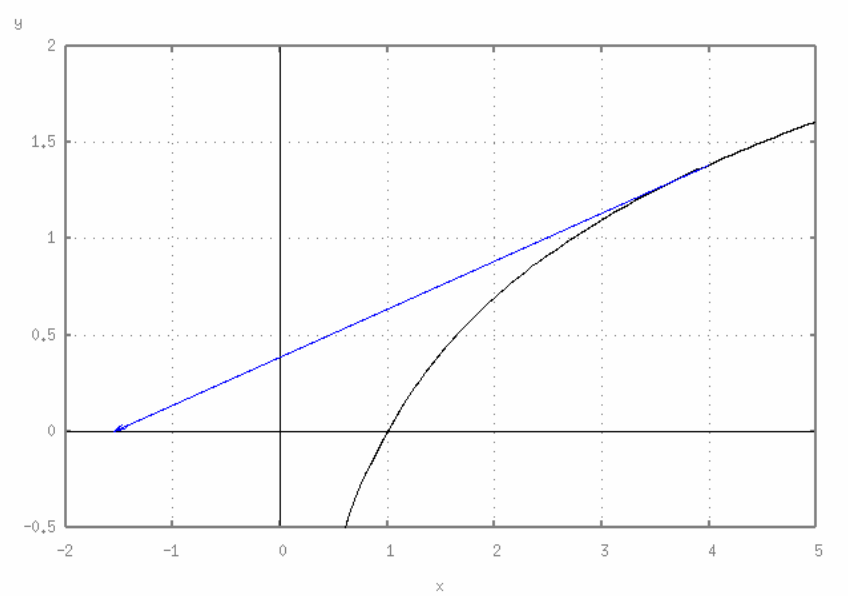

Abbildung 3-2: Newton-Iteration außerhalb des Definitionsbereichs von f

3.2.2 Endlosschleifen

Gegeben sei die Funktion

$$f(x) = \begin{cases} e^{x-1.5} - 1 & \text{für } x < 1.5 \\ 1 - e^{-(x-1.5)} & \text{für } x \geq 1.5 \end{cases}$$

aus Abbildung 1-1.

Diese Funktion besitzt nur einen geringen Konvergenzbereich, in dem die Wahl einer Startnäherung zu einer Lösung führt. Näherungen außerhalb des Konvergenzbereichs führen zu einer Endlosschleife.

Die folgende Abbildung zeigt die ersten drei Newton-Iterationen für $x_0=3.0$. In der Theorie läuft das Verfahren in diesem Fall mit zunehmender Distanz zur eigentlichen Lösung endlos weiter, in der Praxis wird jedoch mit der vierten Iteration bereits der Gültigkeitsbereich der Programmintern verwendeten Variablen verlassen.

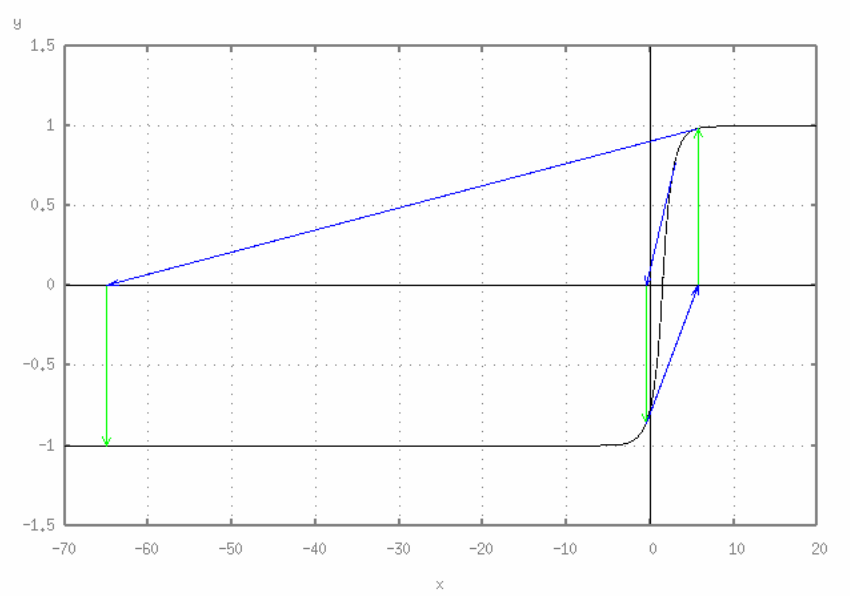

Abbildung 3-3: Beginn einer Endlosschleife im Newton-Verfahren

3.2.3 Lokale Extremwerte - “Zufällige Konvergenz“

Ein Beispiel für einen Fall, in dem das Newton-Verfahren zwar eine Lösung findet, jedoch von Konvergenz im eigentlichen Sinn nicht mehr die Rede sein kann stellt die Funktion

$$f(x)=2\cdot x^3+17\cdot x^2+6\cdot x+50$$

dar. Wieder soll $f(x)=0$ gelöst werden. Die Startnäherung sei $x_0=-5.0$.

Lokale Extremwerte wie in diesem Fall im Besonderen das lokale Minimum bei $x=0$ sind für das Newton-Verfahren schwer zu überwinden, da dort die für die Berechnung der Tangente benötigte erste Ableitung f' verschwindet, was, wie auch das Verlassen des Definitionsbereichs oder des Gültigkeitsbereichs der verwendeten Variablen ein Abbruch des Verfahrens bedeutet. Zwar wird in der Praxis nur selten genau der Punkt getroffen an dem $f'(x)=0$ gilt, jedoch ist die Steigung in dessen Umgebung ausreichend gering, um eine effiziente Berechnung der Lösung zu verhindern.

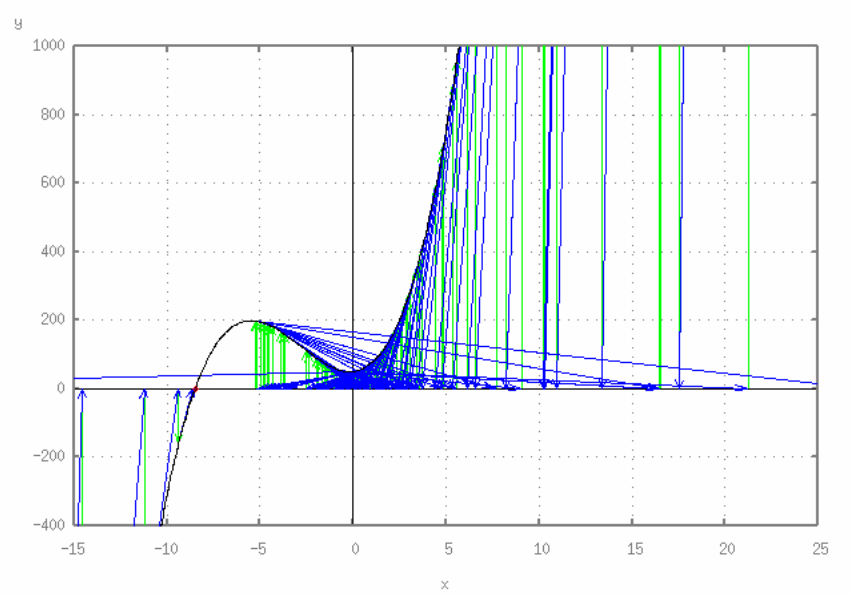

Abbildung 3-4: "Zufällige Konvergenz" in 183 Iterationen

3.3 Aufgabe eines Homotopieverfahrens

Die vorangegangenen Beispiele zeigen einige Schwachstellen des Newton - Verfahrens. Da dieses Verfahren oft als Grundlage für andere Lösungs- und Optimierungsverfahren verwendet wird, ist es nötig, das Konvergenzverhalten zu verbessern. Einen Ansatz, der mit beliebigen Lösungsverfahren eingesetzt werden kann, bieten Homotopieverfahren. Ihre Aufgabe ist es, ein Startproblem P_S mit bekannter bzw. leicht zu berechnender Lösung schrittweise in das zu lösende Zielproblem P_Z zu deformieren. Dabei wird jedes entstehende Zwischenproblem von dem, in das Homotopieverfahren eingebetteten numerischen Löser, wie z.B. dem Newton-Verfahren, gelöst. Die gefundene Lösung wird dann (im einfachsten Fall) als Startnäherung für das nächste Zwischenproblem verwendet.

Mögliche Startprobleme für die Zielprobleme aus 3.2 sind z.B. lineare Funktionen der Form $f(x)=m \cdot x + b$. Für diese Funktionen gilt $f'(x)=m$. Die Tangente ist also identisch zu f, so daß das Newton-Verfahren global in einer Iteration konvergiert.

Abstrakt kann eine Homotopie h zwischen P_S und P_Z, analog zur Topologie, z.B. beschrieben werden durch

$$h(P_S(x(p-\Delta p)), P_Z(x(p-\Delta p)), p)=p \cdot P_S(x(p-\Delta p)) + (1 - p) \cdot P_Z(x(p-\Delta p)).$$

Dabei ist $p \in [0,1]$ der die Deformation steuernde Homotopieparameter und $x(p-\Delta p)$ die jeweilige Startnäherung aus dem vorherigen Homotopieschritt, also die von dem eingebetteten numerischen Lösungsverfahren für das letzte Zwischenproblem gefundene Lösung.

Es gilt

$$x(p)=Löser(\ h(P_S(x(p-\Delta p)),\ P_Z(x(p-\Delta p)),\ p)).$$

Folgendes Diagramm veranschaulicht die abstrakte Funktionsweise eines Homotopieverfahrens in der Numerik.

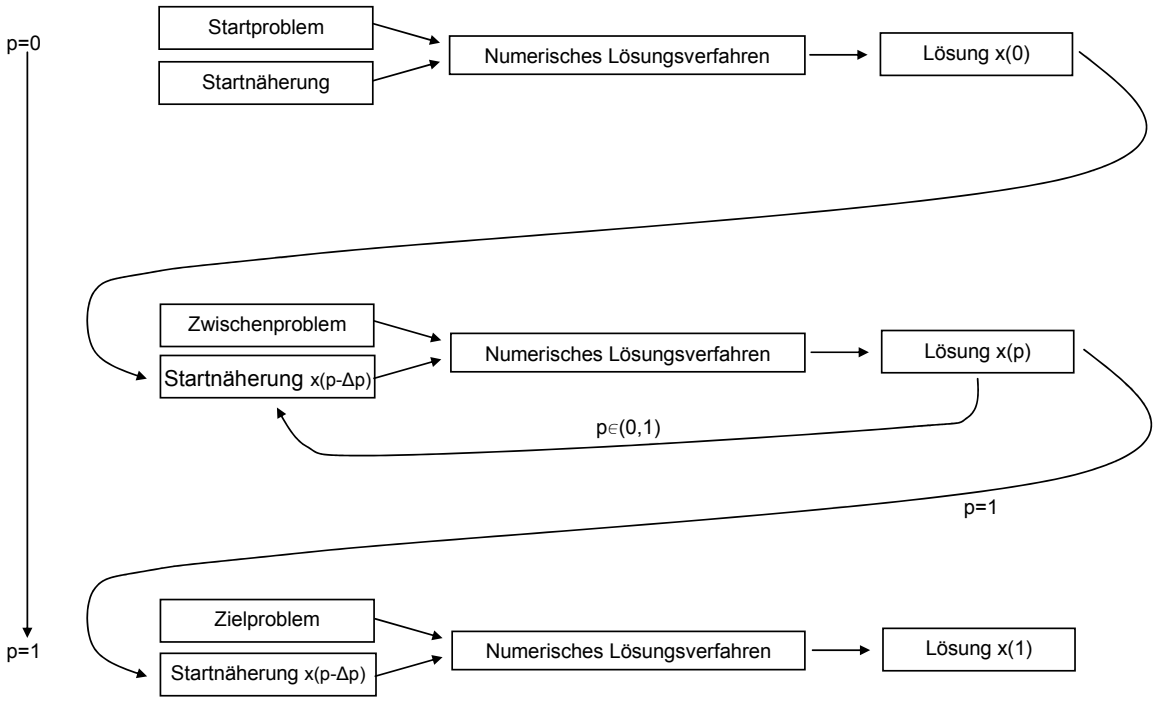

Abbildung 3-5: Abstrakte Darstellung eines Homotopieverfahrens

3.4 Kernproblem Homotopie

Die Wahl der Homotopie, also der Einbettung des Zielproblems in eine Schar von Problemen, und des entsprechenden Startproblems ist der grundlegende Schritt für ein Homotopieverfahren. An dieser Stelle soll auch, wie in [11] zwischen natürlichen und künstlichen Homotopieen unterschieden werden.

Natürliche Homotopieen entstehen z.B. bei der Beschreibung physikalischer Systeme bzgl. Schwerkraft oder Temperatur. Durch den entsprechenden natürlichen Parameter entsteht eine Schar von Funktionen, die allerdings im Allgemeinen nicht die, für die Anwendung eines Homotopieverfahrens notwendigen Voraussetzungen erfüllt.

Insbesondere Stetigkeit und Differenzierbarkeit der Lösungskurve, also des Verlaufs der Lösungen der Zwischenprobleme bzgl. des Homotopieparameters sind häufig gestellte Anforderungen an eine Homotopie, die durch eine natürliche Homotopie nicht zwangsläufig gegeben sind.

Künstliche Homotopieen werden dagegen hinsichtlich dieser benötigten Kriterien entworfen. Der entsprechende Homotopieparameter p wird in diesem Fall dem betreffenden Problem als zusätzliche Variable hinzugefügt, ist also nicht dessen natürlicher, für die mathematische Formulierung benötigter Bestandteil, sondern dient ausschließlich der Steuerung der Deformation des Startproblems in das Zielproblem. Daß jedoch auch in diesem Fall die Wahl der Homotopie nicht trivial ist, zeigen folgende Beispiele.

3.4.1 Bifurkationen der Lösungskurve

Gegeben sei das Zielproblem

$$f(x) \overset{!}{=} 0 \text{ mit}$$
$$f : \Re \to \Re,\ f(x) = x^3 - x.$$

Mit Hilfe einer Einbettung des Newton - Verfahrens in ein Homotopieverfahren soll nun eine Nullstelle von f bestimmt werden. Die dafür verwendete Homotopie sei gegeben durch

$$h : \Re \times [0,1] \to \Re,\ h(x,p) = p \cdot f(x) + (1-p) \cdot g(x)$$

mit dem Startproblem

$$g(x) = x - a,\ a \in \Re \text{ beliebig aber konstant.}$$

Abbildung 3-6 zeigt einige Deformationsschritte zwischen g und f, Abbildung 3-7 den entsprechenden Verlauf der Lösungskurve $x(p)$ für $a=0.0$.

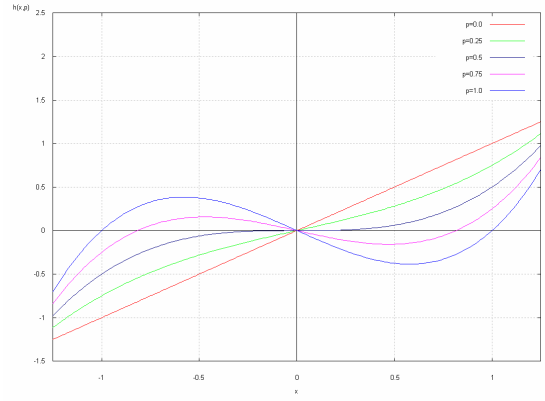

Abbildung 3-6: *h(x,p)* für a=0.0

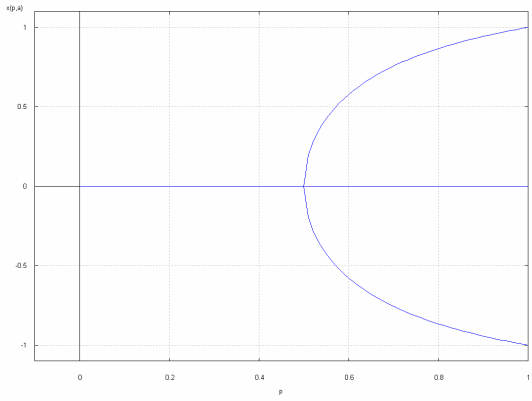

Abbildung 3-7:
Verlauf der Lösungen *x(p)*

Bei der Wahl dieser Homotopie ergibt sich eine Bifurkation der Lösungskurve *x(p)* an der Stelle *p=0.5*. Da an dieser Stelle *x(p)* dann weder stetig noch differenzierbar ist, führt dies zu Problemen, wenn während des Homotopieverfahrens wie z.B. in [8] oder [11] eine Methode zur numerischen Pfadverfolgung eingesetzt werden soll.
Eine Möglichkeit zur Lösung des Problems ist eine geringfügige Variation des Startproblems durch den Parameter *a*.

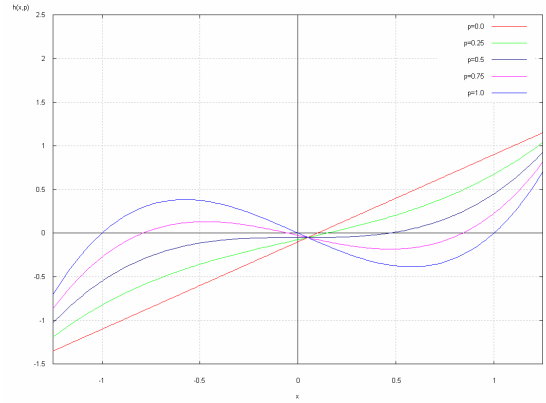

Abbildung 3-8: *h(x,p)* für *a=0.1*

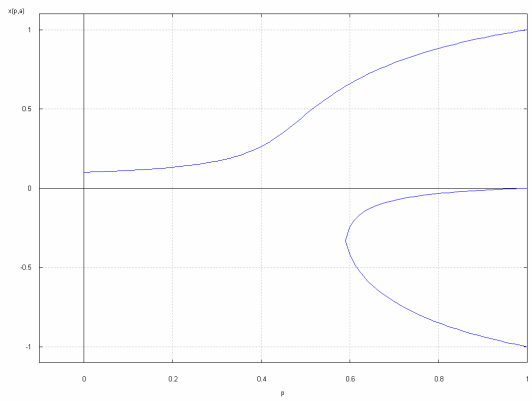

Abbildung 3-9:
Verlauf der Lösungen *x(p)*

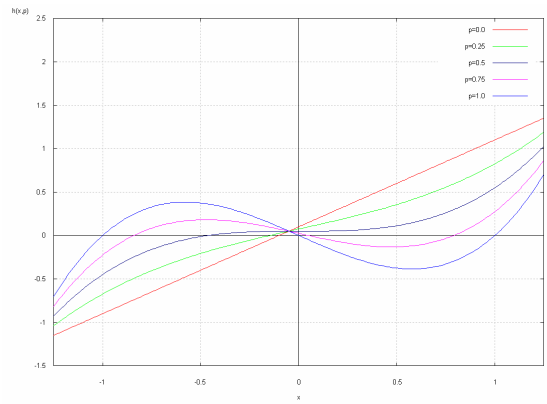

Abbildung 3-10: *h(x,p)* für *a=-0.1*

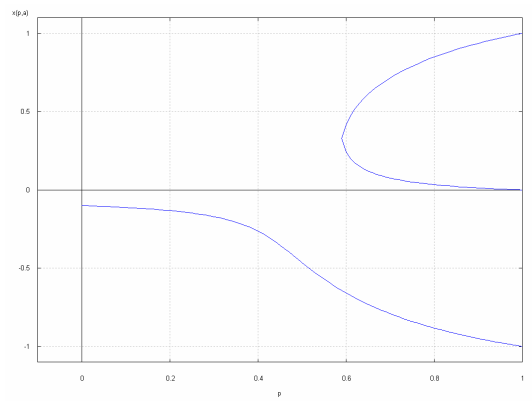

Abbildung 3-11:
Verlauf der Lösungen *x(p)*

Eine andere mögliche Lösung ergibt sich durch eine Veränderung der Einbettung. Wählt man

$$\bar{h}(x,p) = p \cdot f(x) - (1-p) \cdot g(x)$$

statt

$$h(x,p) = p \cdot f(x) + (1-p) \cdot g(x),$$

ergibt sich für $a=0.0$ folgendes Bild für die Deformation bzw. den Verlauf der Lösungen $x(p)$:

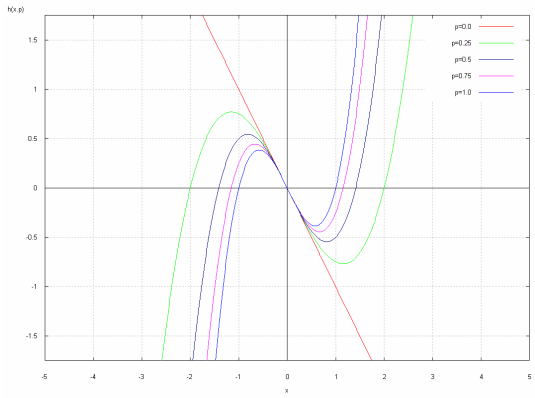

Abbildung 3-12: $\bar{h}(x,p)$ für $a=0.0$

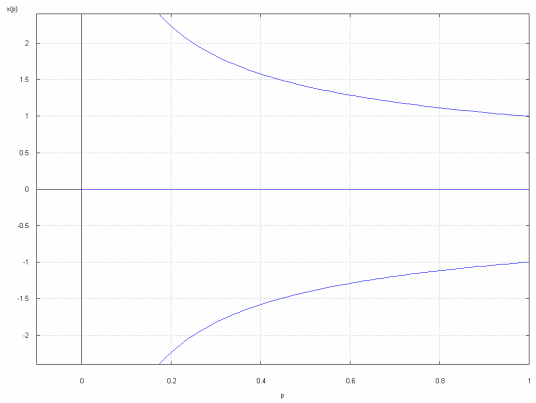

Abbildung 3-13:
Verlauf der Lösungen $x(p)$

3.4.2 Unterbrochene Lösungskurven

Das Zielproblem sei gegeben durch

$$f(x) \overset{!}{=} 0 \text{ mit}$$
$$f : \Re \to \Re,\ f(x) = x^2 - 1.$$

Wie in 3.4.1 soll das Zielproblem mit Hilfe des Newton - Verfahrens in Verbindung mit einem Homotopieverfahren gelöst werden. Die Homotopie sei wieder

$$h : \Re \times [0,1] \to \Re,\ h(x,p) = p \cdot f(x) + (1-p) \cdot g(x) \text{ mit}$$
$$g(x) = x - a,\ a \in \Re \text{ beliebig aber konstant.}$$

Wie Abbildung 3-14 zeigt, entstehen für $a=-1.5$ während des Homotopieverfahrens Zwischenprobleme, die keine Nullstelle aufweisen. Für die entsprechenden Werte des Homotopieparameters p kann also keine Lösung berechnet werden. $x(p)$ ist demnach nicht über das gesamte Intervall $[0,1]$ definiert, woraus, wie in Abbildung 3-15 zu sehen, eine Unterbrechung der Lösungskurve resultiert.

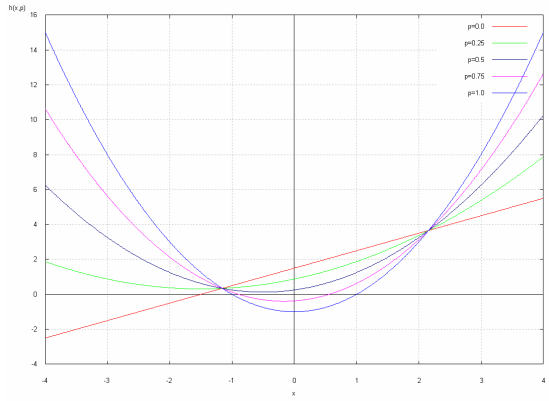

Abbildung 3-14: h(x,p) für a=-1.5

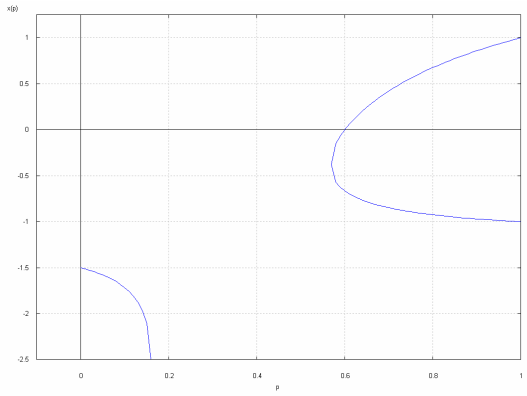

Abbildung 3-15:
Verlauf der Lösungen x(p)

Auch hier kann das entstandene Problem durch eine Variation des Startproblems behoben werden. Die folgenden Abbildungen zeigen $h(x,p)$ bzw. $x(p)$ für $a=0.0$:

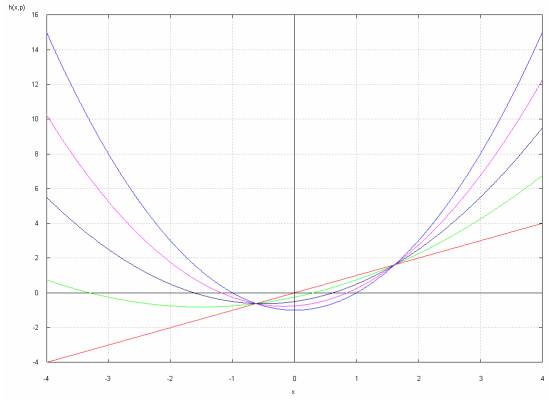

Abbildung 3-16: h(x,p) für a=0.0

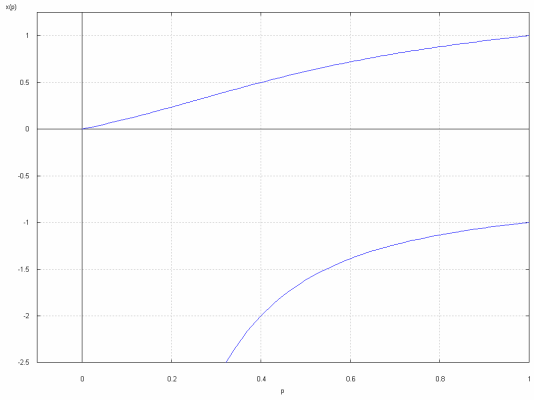

Abbildung 3.17:
x(p) für a=0.0

3.4.3 Konstante Startprobleme - nicht injektives $p \rightarrow x(p)$

Weitere Schwierigkeiten ergeben sich bei der Wahl konstanter Startprobleme in Verbindung mit einer Homotopie, die z.B. eine lineare Verschiebung entlang der y-Achse erzeugt.

Gegeben sei das Zielproblem aus 3.4.1

$$f(x) \overset{!}{=} 0 \text{ mit}$$
$$f : \Re \to \Re,\ f(x) = x^3 - x,$$

sowie die Homotopie

$$h : \Re \times [0,1] \to \Re,\ h(x,p) = f(x) - (1-p) \cdot f(x_0).$$

x_0 ist hier die für das Newton - Verfahren benötigte Startnäherung der Nullstelle, für die allein mit dem Newton - Verfahren keine Lösung gefunden werden kann.

Diese Homotopie stellt, auch im Bezug auf das in 3.2.3 vorgestellte Problem, eine ungünstige Einbettung des Zielproblems dar, da die dort vorhandenen Extremwerte, wie Abbildung 3-18 zeigt, bei einer linearen Verschiebung weiterhin erhalten bleiben. Gelingt es, diese Problemstellen z.B. mit Hilfe zusätzlicher Globalisierungs- oder Dämpfungsstrategien des Newton - Verfahrens zu überwinden, ergibt sich der in Abbildung 3-19 dargestellte Lösungsverlauf.

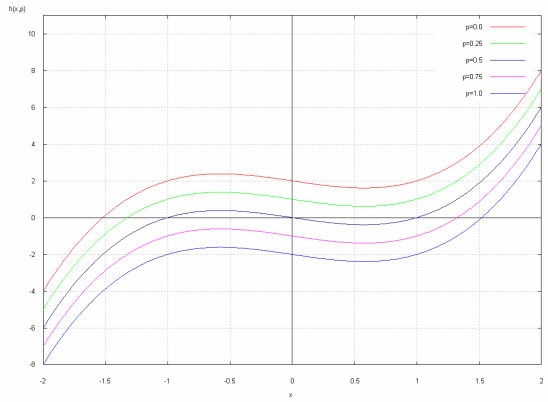

Abbildung 3-18: Linear verschobene Zwischenprobleme

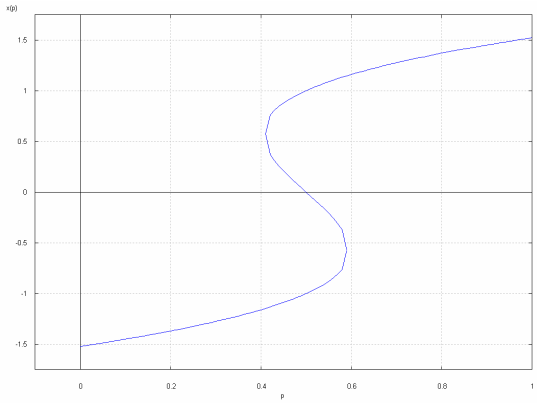

Abbildung 3-19: Lösungsverlauf bzgl. des Homotopieparameters p

Entgegengesetzt zu den bisher vorgestellten Homotopieen und deren Lösungsverläufen ist hier die Abbildung $p \to x(p)$ für den zu Verfolgenden Abschnitt der Lösungskurve nicht injektiv. Zur Verfolgung des Verlaufs aus Abbildung 3-19 darf der Homotopieparameter p also nicht kontinuierlich vergrößert werden.
Dies führt, auch im Bezug auf die in 3.5 vorgestellten Probleme, bei der Steuerung der Homotopieschrittweite zu weiteren Schwierigkeiten, da bei einer Verkleinerung von p zusätzlich unterschieden werden muß, ob diese aus einem mit der gegeben Startnäherung nicht lösbaren Zwi-

schenproblem resultiert, oder zur Verfolgung des Lösungsverlaufs notwendig ist.
Gravierender ist in diesem Fall allerdings die fehlende Eindeutigkeit der Lösungskurve, wodurch der Einsatz von numerischen Methoden zur Pfadverfolgung unmöglich wird.
Gelöst wird dieses Problem, wie auch in [11] oder [12] durch die zusätzliche Parametrisierung des Homotopieparameters p bzgl. der Länge l der Lösungskurve, wodurch sich der in Abbildung 3-20 dargestellte Verlauf ergibt.

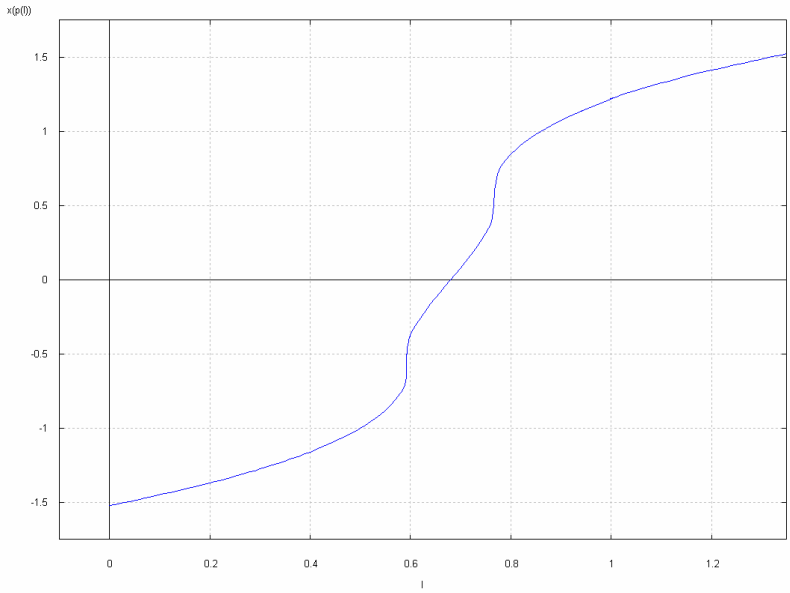

Abbildung 3-20: Lösungsverlauf bzgl. der Länge l der Lösungskurve

3.5 Kernproblem Homotopieschrittweite

Die Steuerung des Homotopieparameters p bzw. dessen Schrittweite Δp entscheidet über die Effizienz des Verfahrens. Da mit jedem Zwischenschritt ein Teilproblem entsteht, welches wiederum mit dem, meist sehr zeitaufwendigen, externen numerischen Lösungsverfahren gelöst werden muß, ist es Ziel, P_S mit einer möglichst geringen Anzahl von Zwischenschritten in P_Z zu überführen. Um dies zu erreichen, gilt es, folgende Situationen zu vermeiden:

1. Δp zu klein:
Ist die Schrittweite Δp sehr klein, kann zwar das nächste entstehende Zwischenproblem mit der im aktuellen Schritt berechneten Lösung als Startnäherung wieder gelöst werden, die Berechnung der Lösung wäre aber durchaus auch mit einer größeren Schrittweite möglich gewesen.

Um unnötige Zwischenschritte, die den Gesamtaufwand erhöhen, zu vermeiden, soll also die Steigung von p bzgl. der Anzahl der Homotopieschritte maximiert werden.

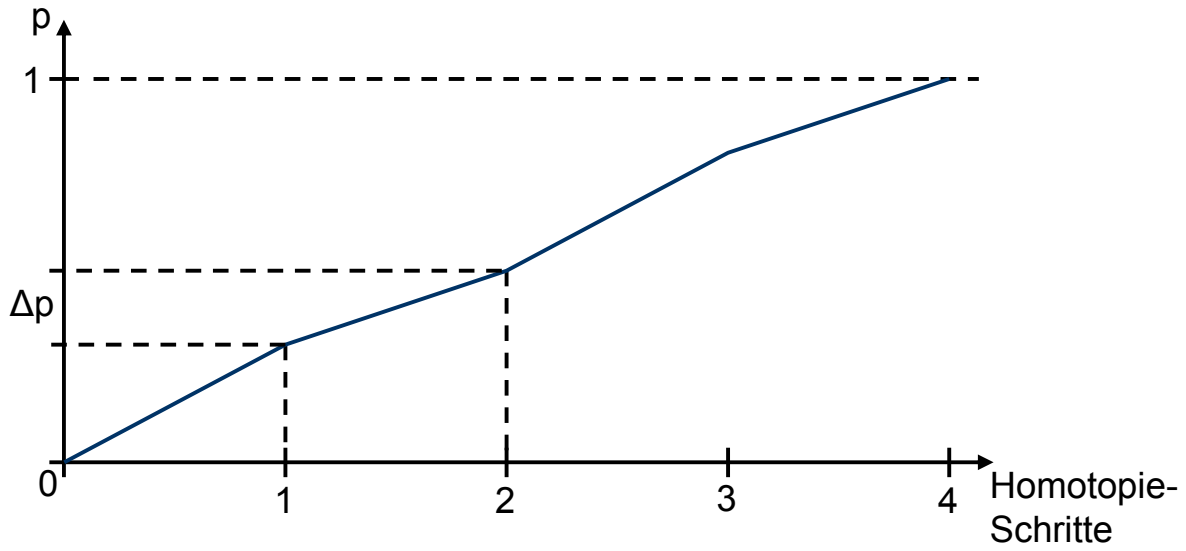

Abbildung 3-21: kleines Δp zwischen Schritt 1 und 2

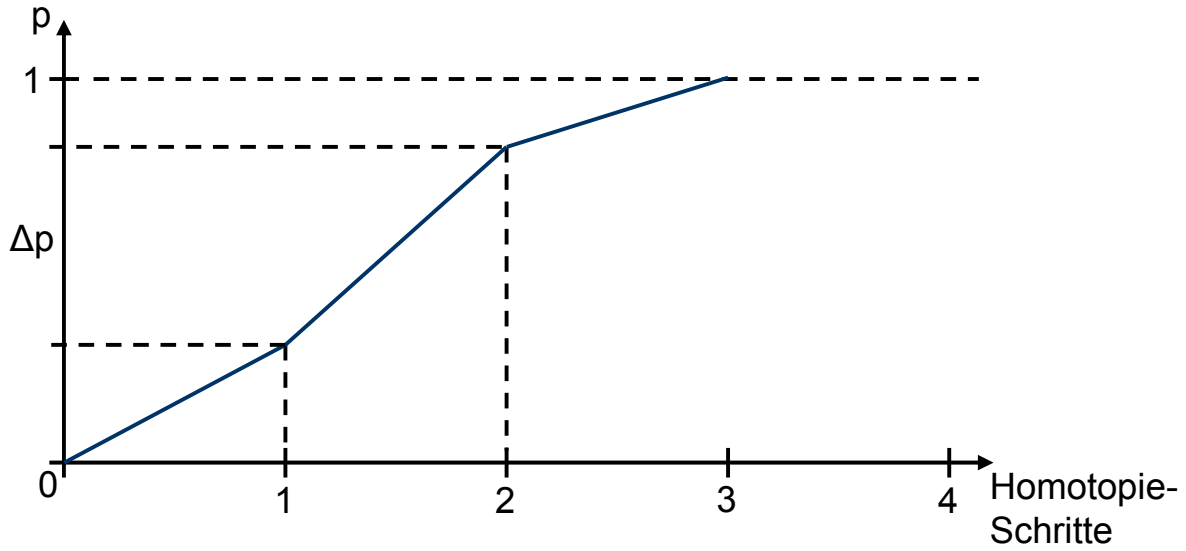

Abbildung 3-22: Δp zwischen Schritt 1 und 2 maximiert

2. Δp zu groß:
Wurde Δp zu groß gewählt, kann das nächste Zwischenproblem mit der im aktuellen Schritt berechneten Lösung als Startnäherung nicht mehr gelöst werden. p muß also für den nächsten Schritt wieder verkleinert werden, um die Deformation Richtung Zielproblem zu reduzieren, und somit ein Zwischenproblem zu erzeugen, daß mit der aktuellen Startnäherung gelöst werden kann.

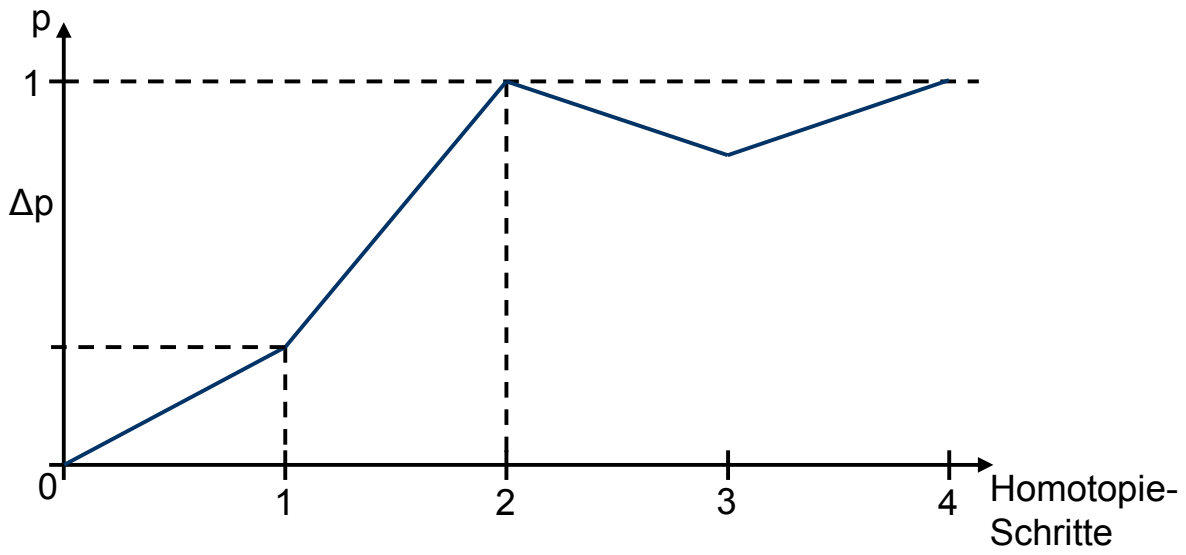

Abbildung 3-23: Δp zwischen Schritt 1 und 2 zu groß

Für diesen Rückwärtsschritt ergeben sich nun wieder zwei Randfälle:

2.i: Δp zu klein:
Eine zu geringe Schrittweite Δp bei einem Rückwärtsschritt resultiert in einer ungenügenden Reduktion der Deformation zwischen P_S und P_Z. Das entstehende Zwischenproblem kann also wieder nicht mit der vorhandenen Startnäherung gelöst werden, so daß weitere Rückwärtsschritte notwendig werden.

2.ii: Δp zu groß:
Analog zu dem Fall eines zu kleinen Δp für einen Vorwärtsschritt bedeutet eine zu große Schrittweite bei einem Rückwärtsschritt, daß das entstehende Zwischenproblem mit der gegebenen Startnäherung wieder gelöst werden kann. Eine geringere Verminderung der Deformation, also ein kleineres Δp, wäre aber schon ausreichend gewesen, um dies zu erreichen.

4 Das Prädiktor - Korrektor Homotopieverfahren von E. Grigat

E. Grigat entwickelte in [8] ein Homotopieverfahren, das in Verbindung mit einem Lösungsverfahren für Randwertprobleme zur Berechnung von Optimalflugbahnen in Fallwindgebieten eingesetzt wird. Im Folgenden sollen zuerst die Grundlagen der mathematischen Modellierung dieser Problemklasse, sowie die daraus resultierende Notwendigkeit eines Homotopieverfahrens anhand einiger Beispiele geringerer Komplexität erläutert werden. Anschließend wird das Homotopieverfahren selbst und seine Schrittweitensteuerung vorgestellt.

4.1 Differentialgleichungen

Zahlreiche Vorgänge aus verschiedenen naturwissenschaftlichen Disziplinen, wie auch Flugbahnen, lassen sich mathematisch durch (Systeme von) Differentialgleichungen beschreiben.
Ein Beispiel ist die in [7] entwickelte Vorwärtskinematik differentialgesteuerter Fahrzeuge.

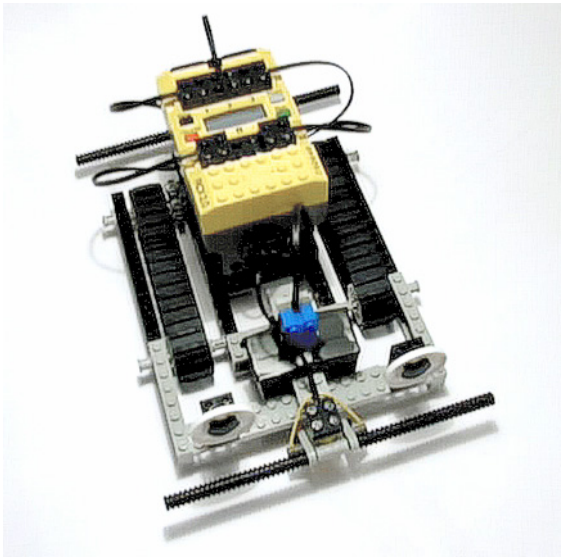

Abbildung 4-1: Differentialgesteuerter Lego-Mindstorms Roboter

Die Bewegung dieser Fahrzeuge in der x,y - Ebene eines vom Fahrzeug unabhängigen Koordinatensystems wird mit Hilfe der sensorisch einfach zu messenden [9] Rad- bzw. Kettengeschwindigkeiten durch das folgende Differentialgleichungssystem dargestellt:

$$\dot{x}(t) = \frac{v_r(t) + v_l(t)}{d} \cdot \cos\theta(t)$$
$$\dot{y}(t) = \frac{v_r(t) + v_l(t)}{d} \cdot \sin\theta(t) \qquad (4.1\text{-}1)$$
$$\dot{\theta}(t) = \frac{v_r(t) - v_l(t)}{d}$$

Dabei bezeichnet $v_l(t)$ bzw. $v_r(t)$ die Geschwindigkeit des linken bzw. rechten Antriebsrades zum Zeitpunkt t, d den Abstand der beiden Antriebsräder und θ die Orientierung des Fahrzeugs bzgl. der x-Achse. $\dot{x}$ und $\dot{y}$ bezeichnen die Ableitung der Position des Fahrzeugs nach der Zeit, also dessen lineare Geschwindigkeit, $\dot{\theta}$ dementsprechend die Winkelgeschwindigkeit.

4.2 Anfangswertprobleme

Die Position des Fahrzeugs aus 4.1 kann durch Integration des Differentialgleichungssystems (4.1-1) bzgl. der Zeit t bestimmt werden.
Die analytische Integration von (4.1-1) ist zwar, im Gegensatz zur Integration weit komplexerer Differentialgleichungssysteme, durchaus noch möglich, allerdings resultieren daraus veränderte sensorische Anforderungen an das Fahrzeug. So müßte für die Antriebsräder dann nicht mehr die Geschwindigkeit zum Zeitpunkt t, sondern jeweils der bis dahin zurückgelegte Weg gemessen werden. Da sich (4.1-1) aber aus der Verwendung von Geschwindigkeitssensoren ergibt, ist eine analytische Lösung in diesem Fall also nicht geeignet.

Wird für den linken Rand des Integrationsintervalls $[t_0;t_f]$ ein Anfangswert vorgegeben, ist also zum Beispiel die Position des Fahrzeugs zum Zeitpunkt t_0 bekannt, kann die Integration des Differentialgleichungssystems mittels numerischer Approximationsverfahren durchgeführt werden. Hierfür existieren verschiedene Methoden unterschiedlicher Genauigkeit, die auch im weiteren Verlauf dieses Kapitels noch von Bedeutung sein werden.

Zwei dieser Methoden, die Verfahren von Euler und Runge, sollen hier vorgestellt werden. Beide Verfahren gehören zur Klasse der Einschrittverfahren, die allgemein für eine gegebene Differentialgleichung der Form

$$\dot{x}(t) = f(x,t)$$

durch die Verfahrensvorschrift

$$\tilde{x}(t+\Delta t) = \tilde{x}(t) + \phi_f(\tilde{x}(t),t,\Delta t)$$

beschrieben werden können, wobei $\tilde{x}$ die numerisch ermittelte Näherung der analytischen Lösung bezeichnet.

Für das Euler - Verfahren gilt nun

$$\phi_f(\tilde{x}(t),t,\Delta t) = \Delta t \cdot f(\tilde{x}(t),t).$$

Zur Approximation der Lösung wird hier die Differentialgleichung f an den diskreten Zeitpunkten $i \cdot \Delta t$ ausgewertet. Mit diesen ermittelten Steigungen wird dann die Approximation für $\tilde{x}(t)$ ausgehend von der jeweils letzten berechneten Schätzung $\tilde{x}(t-\Delta t)$ bestimmt.

Im Unterschied zum Euler - Verfahren wird bei der Methode von Runge die Differentialgleichung zusätzlich zu den diskreten Zeitpunkten $i \cdot \Delta t$ noch an den Stellen $i \cdot \Delta t + 0.5 \cdot \Delta t$ ausgewertet. Es gilt

$$\begin{aligned} k_1 &= f(\tilde{x}(t),t) \\ k_2 &= f(\tilde{x}(t) + 0.5 \cdot \Delta t \cdot k_1, t + 0.5 \cdot \Delta t) \\ \phi_f(\tilde{x}(t),t,\Delta t) &= \Delta t \cdot k_2 . \end{aligned}$$

Mit den zusätzlichen Auswertungen von f wird einerseits eine höhere Genauigkeit der Approximation erreicht, andererseits steigt der Rechenaufwand, wenn die Anzahl der Auswertungen von f als Maß benutzt wird, um den Faktor 2.

Abbildung 4-2 zeigt die beiden Verfahren am Beispiel des Anfangswertproblems

$$\begin{aligned} &x: \Re \rightarrow \Re,\ \dot{x}(t) = e^t \cdot x(t) \\ &\text{mit } t_0 = 0,\ t_f = 1 \text{ und } x(t_0) = 1. \end{aligned} \qquad (4.2\text{-}1)$$

Die analytische Lösung dieses Anfangswertproblems kann mit Hilfe des Ansatzes

$$\int_{x_0}^{x(t)} (h(\xi))^{-1} d\xi = \int_{t_0}^{t} g(\vartheta) d\vartheta$$

zur Lösung von Differentialgleichungen mit getrennten Veränderlichen der Form

$$\dot{x}(t) = g(t) \cdot h(x(t))$$

bestimmt werden. Mit $g(t) = e^t$ und $h(x(t)) = x(t)$ folgt dann

$$x(t) = e^{e^t - 1 + \ln(x(t_0))}.$$

Für $x(t_0)=1$ gilt demnach

$$x(t) = e^{e^t - 1}.$$

Als Diskretisierung der Zeitachse für die numerische Näherung mit den Verfahren von Euler und Runge wurde $\Delta t=0.25$ gewählt, die analytische Lösung wird also mit vier linearen Teilabschnitten approximiert.

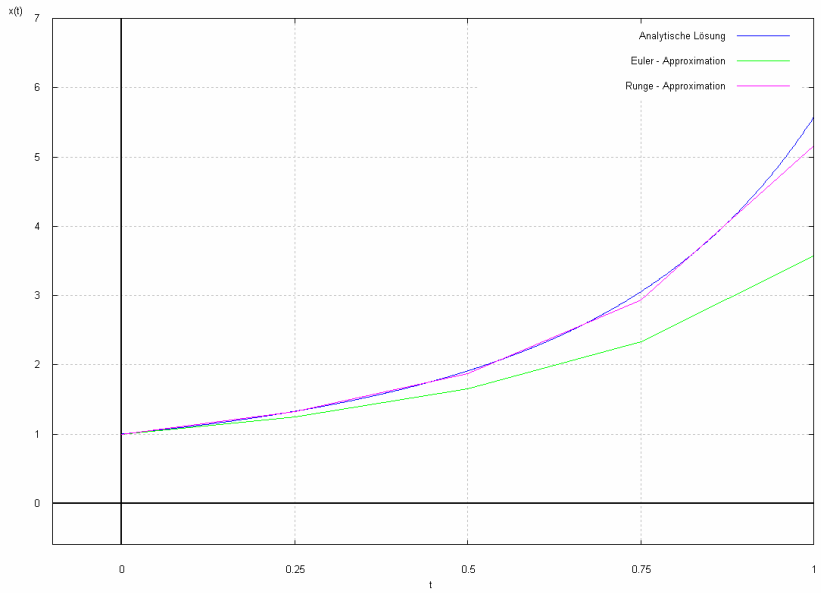

Abbildung 4-2: Numerische Integrationsverfahren von Euler und Runge

4.3 Randwertprobleme

Anders als bei Anfangswertproblemen werden bei einem Randwertproblem Bedingungen an beide Grenzen des Integrationsintervalls gestellt. Allgemein besteht ein Randwertproblem also aus einer oder mehreren Differentialgleichungen der Form

$$\dot{x}(t) = f(x,t)$$

mit gegebenen Randbedingungen

$$r(x(t_0), x(t_f))=0. \qquad (4.3\text{-}1)$$

Im Fall des in 4.1 vorgestellten Fahrzeugs wäre ein mögliches Randwertproblem z.B. eine Fahrt, die zum Zeitpunkt t_0 in Punkt p_1 mit $\theta=90°$ startet und zum Zeitpunkt t_f in Punkt p_2 mit $\theta=270°$ endet.

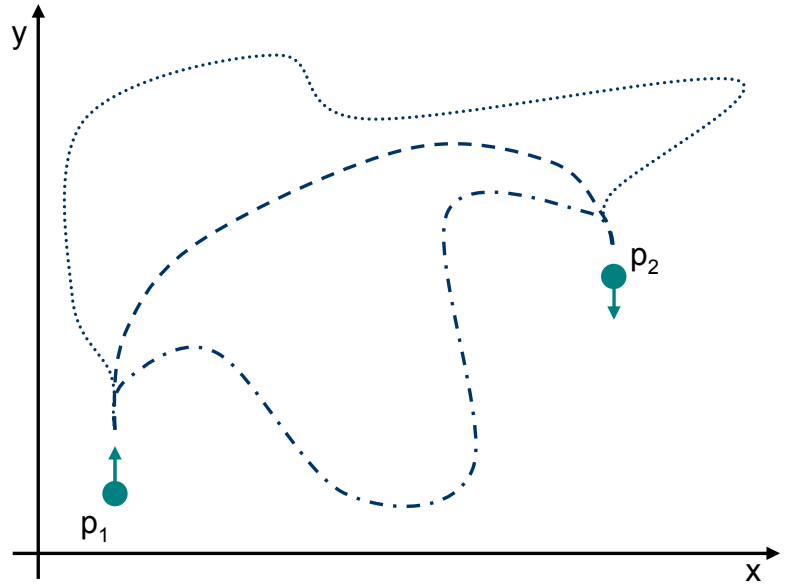

Abbildung 4-3: Verschiedene Trajektorien zwischen p_1 und p_2

Eine Möglichkeit zur Lösung von Randwertproblemen, die auch E. Grigat verwendet, bieten Schießverfahren.
Dabei wird die Lösung des durch die Randbedingung (4.3-1) gegebenen Nullstellenproblems z.B. mit Hilfe des in 3.1 vorgestellten Newton - Verfahrens ermittelt. Zur Berechnung des für die Auswertung der Randbedingung (4.3-1) benötigten Wertes $x(t_f)$ am rechten Rand des Integrationsintervalls $[t_0;t_f]$ wird dann, ausgehend von einer gegebenen Startnäherung $\tilde{x}(t_0)$ für den linken Rand des Integrationsintervalls, in jeder Newton - Iteration ein Anfangswertproblem gelöst.

Durch die Verwendung des Newton - Verfahrens zur Lösung von Randwertproblemen bestehen für Schießverfahren die Probleme, wie sie schon in 3.2 für das Newton - Verfahren vorgestellt wurden. Ist eine hinreichend genaue Startnäherung $\tilde{x}(t_0)$ für die Lösung $x(t_0)$ des Randwertproblems nicht bekannt, wird also der Einsatz eines Homotopieverfahrens nötig.

Ein weiteres Beispiel für ein Randwertproblem ergibt sich durch eine entsprechend geänderte Formulierung des Anfangswertproblems (4.2-1). Gegeben sei die Differentialgleichung

$$x:\Re\to\Re,\ \dot{x}(t)=e^t\cdot x(t), \qquad (4.3\text{-}2)$$
$$t_0=0,\ t_f=1$$

mit der Randbedingung

$$r(x(t_0),x(t_f))=x(t_f)-x(t_0)-(e^{e^t-1}-1)\overset{!}{=}0.$$

Diese Randbedingung ist für $x(t_0)=1$ erfüllt. Die analytische Lösung hat also denselben Verlauf wie die des Anfangswertproblems (4.2-1).
Abbildung 4-4 zeigt einige (analytische) Lösungen der Anfangswertprobleme die im Verlauf des Schießverfahrens berechnet werden müssen, Abbildung 4-5 zeigt den Verlauf der Randbedingung bezüglich $x(t_0)$. Da r in diesem Fall nicht über ganz $\Re$ definiert ist, ist wie in 3.2.1 eine Newton- Iteration außerhalb des Definitionsbereichs von r möglich.

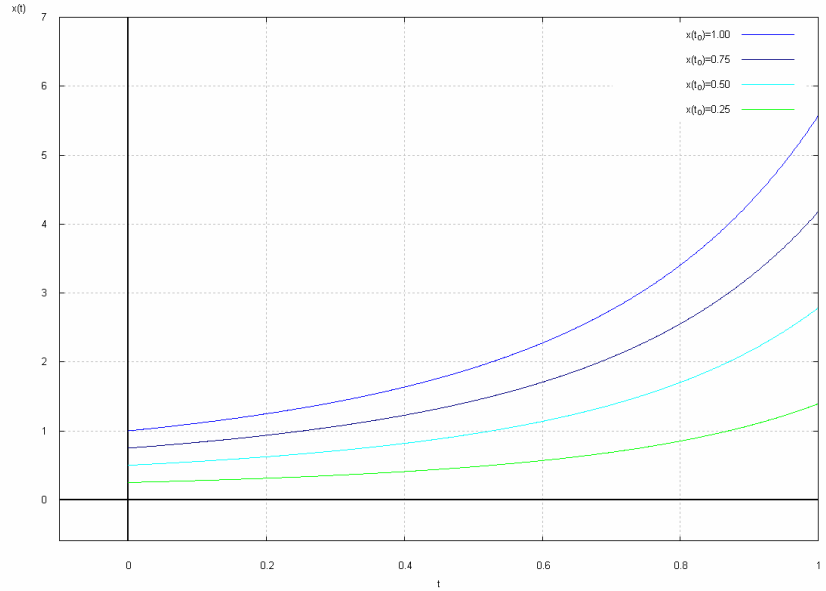

Abbildung 4-4: Analytische Lösungen von (4.3-2) für verschiedene $x(t_0)$

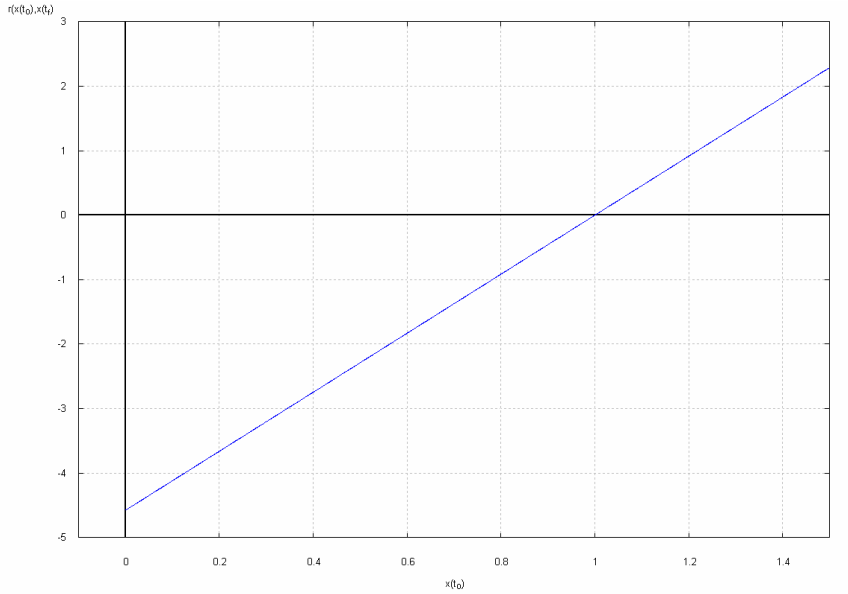

Abbildung 4-5: Verlauf der Randbedingung von (4.3-2) bzgl. $x(t_0)$

4.4 Optimierung

Abbildung 4-3 zeigt, daß durchaus mehrere Lösungen eines Randwertproblems existieren können. Wie auch in der Arbeit von E. Grigat wird aber meist eine bzgl. eines Kriteriums wie z.B. Zeit, Energieverbrauch oder Materialverschleiß optimale Lösung gesucht. Zur Berechnung dieser optimalen Lösung muß das Minimum einer entsprechend formulierten Zielfunktion bestimmt werden. Ein notwendiges Kriterium für Minima einer Funktion ist das Verschwinden ihrer ersten Ableitung an dieser Stelle. Das resultierende Nullstellenproblem kann z.B. wieder mit dem Newton - Verfahren gelöst werden, wobei in jeder Newton - Iteration ein Randwertproblem berechnet werden muß. Auch hier ergeben sich durch die Verwendung des Newton - Verfahrens wieder die bereits in 3.2 vorgestellten Probleme, die wiederum zum notwendigen Einsatz eines Homotopieverfahrens führen. Ähnliche Überlegungen gelten für die, von E. Grigat verwendete Optimierung über Funktionsräumen.

4.5 Das Prädiktor - Korrektor Verfahren

Der Ansatz für das Homotopieverfahren von E. Grigat geht zurück auf eine Arbeit von D.F. Davidenko [10]. Die grundlegende Idee ist, die Homotopie auf ein Anfangswertproblem zurückzuführen und damit den Verlauf der Lösungen $x(p)$ der erzeugten Zwischenprobleme bzgl. des Homotopieparameters p mittels eines Prädiktor - Korrektor Verfahrens zur numerischen Pfadverfolgung zu berechnen.

Der Prädiktor stellt dafür ein Verfahren zur Lösung von Anfangswertproblemen zur Verfügung, der Korrektor entspricht dem, in das Homotopieverfahren eingebetteten numerischen Lösungsverfahren.
Aufgabe des Prädiktors ist es, Startnäherungen $\tilde{x}(p)$ für den Korrektor zu erzeugen, von welchen ausgehend dieser die Lösung $x(p)$ des aktuellen Zwischenproblems berechnen kann.

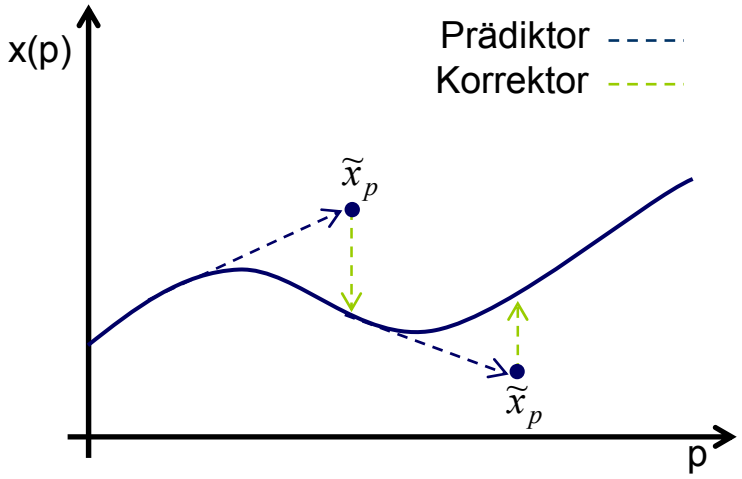

Abbildung 4-6: Prädiktor - Korrektor Pfadverfolgung

Das für den Prädiktor benötigte Anfangswertproblem wird durch Differenzieren der Homotopie $h(x,p)$ nach dem Homotopieparameter p erzeugt. Ist $x(p)$ die Lösung des durch den Parameter p erzeugten Zwischenproblems, gilt also o.B.d.A. $h(x(p),p)=0$, so folgt

$$\frac{\partial h}{\partial p} = \frac{\partial h}{\partial x(p)} \cdot \frac{\partial x(p)}{\partial p} = 0 \text{ bzw.}$$

$$\frac{\partial x(p)}{\partial p} = -\frac{\partial h}{\partial p} \cdot \left(\frac{\partial h}{\partial x(p)} \right)^{-1} \qquad (4.5\text{-}1)$$

Zusammen mit der für $p=0$ bekannten bzw. einfach zu berechnenden Lösung ergibt sich das erste gewünschte Anfangswertproblem, das im Prädiktor z.B. mittels der in 4.2 vorgestellten Verfahren gelöst werden kann. Da diese Verfahren aber, wie schon in Abbildung 4-2 gezeigt, nicht die exakte Lösung $x(p)$, sondern nur eine Approximation $\tilde{x}(p)$ derer liefern, wird diese ermittelte Näherung dem Korrektor als Startnäherung übergeben. Im Fall des Prädiktor - Korrektor - Verfahrens von E. Grigat ist der Korrektor eine Variante des bereits vorgestellten Schießverfahrens. Mit der im Korrektor berechneten exakten Lösung $x(p)$ als neuem Anfangswert wird dann der nächste Prädiktor gestartet, der wiederum die Startnäherung $\tilde{x}(p+\Delta p)$ für den folgenden Korrektor - Schritt bestimmt.

4.6 Prädiktoren unterschiedlicher Ordnung

Per Definition ist ein numerisches Integrationsverfahren bzw. der daraus abgeleitete Prädiktor von der Ordnung u wenn für ein konstantes $c \in \Re$

$$\left\| x(t_0 + \Delta t) - \tilde{x}(t_0 + \Delta t) \right\| \leq c \cdot \Delta t^{u+1}. \qquad (4.6\text{-}1)$$

Dabei bezeichnet $x(t_0+\Delta t)$ die analytische Lösung der Differentialgleichung an der Stelle $t_0+\Delta t$ und $\tilde{x}(t_0 + \Delta t)$ deren durch den Prädiktor berechnete Näherung.

E. Grigat verwendet für sein Homotopieverfahren Prädiktoren erster bis fünfter Ordnung. Die ersten drei werden im Folgenden vorgestellt und anhand der Verfolgung eines gemeinsamen Lösungsverlaufs $x(p)$ verglichen.

Um den Rechenaufwand des Korrektors möglichst gering zu halten, ist es Ziel, die Homotopieschrittweite Δp so zu wählen, daß sich die vom Prädiktor ermittelte Schätzung $\tilde{x}(p)$ der Lösung $x(p)$ im Toleranzbereich des Korrektors befindet, so daß dieser die Lösung der Zwischenprobleme in einer Iteration berechnen kann. Der Toleranzbereich bezieht sich in diesem Fall nicht wie in 3.1 auf die analytische Nullstelle, sondern auf die Korrektur einer Newton - Iteration. Bezeichnen

$$x_i^{Newton}(p),\ i \in [0,f]$$

die während des Newton - Verfahrens berechneten Achsen - Schnittpunkte, wobei $x_0^{Newton}(p) = \tilde{x}(p)$ und $x_f^{Newton}(p) = x(p)$, soll also

$$\left\| x_0^{Newton}(p) - x_1^{Newton}(p) \right\| \leq tol$$

gelten. Der Toleranzbereich und die Grenze tol sind dabei nicht notwendigerweise identisch. Ebenfalls muß der Toleranzbereich, wie zur Vereinfachung der Darstellung in den folgenden Abbildungen auch bei konstantem tol nicht konstant sein.

Da, durch die Verwendung des Schießverfahrens, in jeder Korrektor - Iteration ein Randwertproblem gelöst werden muß, folgt aus einem Prädiktor außerhalb des Toleranzbereichs ein erheblicher Anstieg des Rechenaufwands im Korrektor.

4.6.1 Der Prädiktor erster Ordnung (Trivialer Prädiktor)

Für den trivialen Prädiktor gilt

$$\tilde{x}(p_i + \Delta p) = x(p_i).$$

Wieder bezeichnet p den Homotopieparameter und $x(p)$ die Lösung des von p erzeugten Zwischenproblems. $\tilde{x}(p_i + \Delta p)$, als die vom Prädiktor berechnete Startnäherung für den Korrektor im nächsten Homotopieschritt, ist im Fall des trivialen Prädiktors also das Ergebnis des aktuellen Korrektors. Der Rechenaufwand für diesen Prädiktor ist demzufolge vernachlässigbar gering, allerdings ergeben sich im Vergleich zu Prädiktoren höherer Ordnung kürzere Homotopieschrittweiten, was zu einer relativ hohen Anzahl benötigter Korrektor - Aufrufe führt.
Für den Lösungsverlauf des Beispiels in Abbildung 4-7 werden bei Anwendung des Trivialen Prädiktors 9 Homotopieschritte (Korrektoraufrufe) zur Lösung des Zielproblems benötigt.

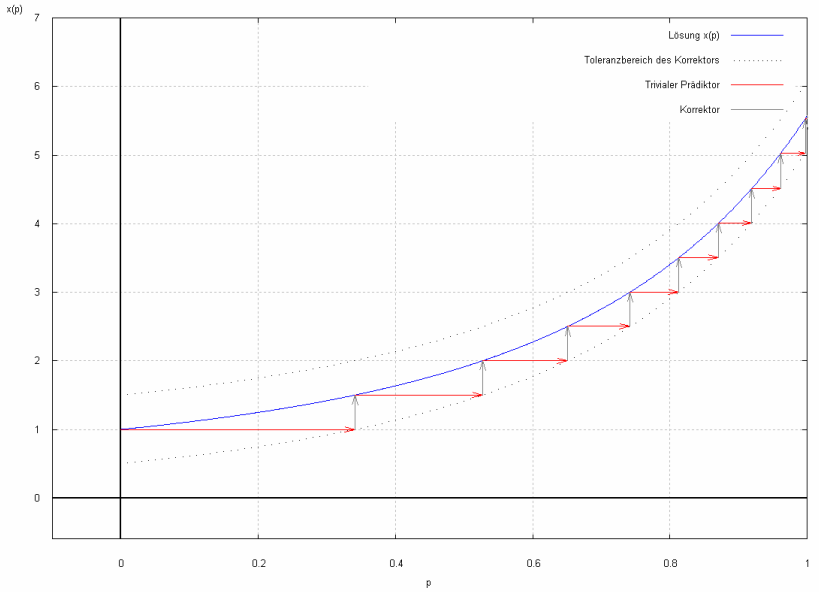

Abbildung 4-7: Trivialer Prädiktor

4.6.2 Der Prädiktor zweiter Ordnung (Euler - Prädiktor)

Der Prädiktor zweiter Ordnung besteht aus dem in 4.2 vorgestellten Euler - Verfahren zur Lösung von Anfangswertproblemen. Im Vergleich zum trivialen Prädiktor liefert der Euler - Prädiktor genauere Approximationen der Lösungskurve und ermöglicht so größere Homotopieschrittweiten, benötigt dafür aber auch einen höheren Rechenaufwand.

Abbildung 4-8 zeigt den Euler - Prädiktor an dem auch für den trivialen Prädiktor verwendeten Verlauf der Lösungen bzgl. des Homotopieparameters. Bei Gebrauch des Euler - Prädiktors werden nur noch 4 Homotopieschritte benötigt.

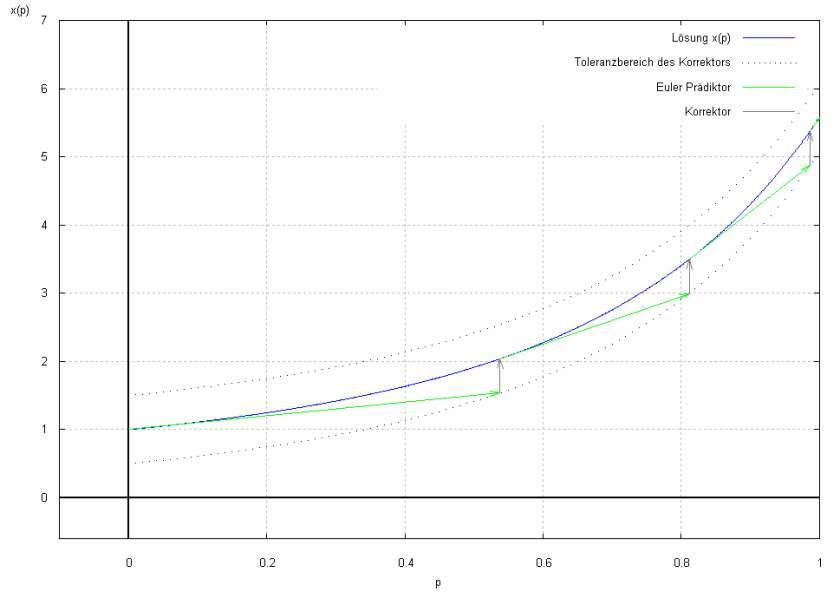

Abbildung 4-8: Euler – Prädiktor

4.6.3 Der Prädiktor dritter Ordnung (Runge - Prädiktor)

Basierend auf dem in 4.2 vorgestellten Runge - Verfahren liefert der Runge - Prädiktor im Vergleich zu den vorher vorgestellten Prädiktoren die genauesten Approximationen der exakten Lösung und ermöglicht so die größten Homotopieschrittweiten, benötigt dafür aber auch den höchsten Rechenaufwand.
Zum besseren Vergleich mit den Prädiktoren geringerer Ordnung wurde für folgende Abbildung der Prädiktor jeweils für die größtmögliche Schrittweite $\Delta p_{i+1}=1-p_i$ berechnet und diese anschließend wieder soweit reduziert, das sich der entsprechende Prädiktor im Toleranzbereich des Korrektors befindet. Bei Verwendung des trivialen Prädiktors werden so 2 Homotopieschritte zur Lösung des Zielproblems benötigt.

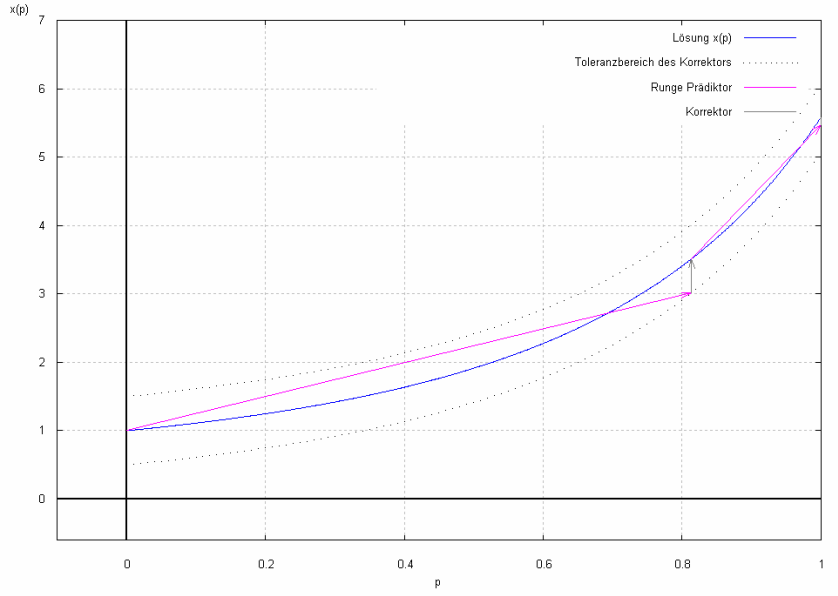

Abbildung 4-9: Runge – Prädiktor

4.7 Aufwandsbedingte Ordnungssteuerung

Im Bezug auf die in 3.4 vorgestellten Probleme bei der Steuerung der Homotopieschrittweite ist es Ziel, den Prädiktorschritt so genau zu wählen, daß sich einerseits die ermittelte Näherung zumindest noch im Konvergenzbereich des im Korrektor verwendeten Lösungsverfahrens befindet, andererseits aber der Rechenaufwand für einen Homotopieschritt, also eine Prädiktor - Korrektor Kombination möglichst gering gehalten wird. So genügt für einen Bereich mit annähernd konstantem oder linearem Verlauf der Lösungskurve ein Prädiktor niedriger Ordnung, um eine große Homotopieschrittweite bei ausreichend genauer Startnäherung für den Korrektor zu erreichen. Für sensitivere Bereiche der Davidenko - Differentialgleichung (4.5-1) muß dagegen ein Prädiktor höherer Ordnung eingesetzt werden.

Als Maß zur Bewertung des Rechenaufwands benutzt E. Grigat die Anzahl der Auswertungen des zu lösenden Differentialgleichungssystems. Daraus ergeben sich drei Maßzahlen:

N_{pr}^{i} — Rechenaufwand im Prädiktor des i. Homotopieschrittes

N_{co}^{i} — Rechenaufwand im Korrektor des i. Homotopieschrittes

$W^{i} = \frac{N_{pr}^{i} + N_{co}^{i}}{\Delta p_{i-1}}$ — Gesamter, bzgl. der Homotopieschrittweite normierter Rechenaufwand des i. Homotopieschrittes

Mit Hilfe dieser Werte kann nun die Ordnung des Prädiktors gesteuert werden. Es ergeben sich drei mögliche Fälle, die Rückschlüsse auf die Sensitivität der Lösungskurve ermöglichen:

1. $W^i > W^{i-1}$

Ein Anstieg des Rechenaufwands folgt aus einer ungünstigen Wahl der Ordnung für den aktuellen Bereich der Lösungskurve, deutet also auf einen Änderung der Sensitivität von (4.5-1) hin.

Wurde die Ordnung des Prädiktors im aktuellen Homotopieschritt i bzgl. des vorangegangenen Schritts $i-1$ nicht verändert oder sogar verringert, gilt also $N_{pr}^{i} \le N_{pr}^{i-1}$, so folgt $N_{co}^{i} \ge N_{co}^{i-1}$.
Der Anstieg des Rechenaufwands wurde also vom Korrektor aufgrund einer ungenauen Startnäherung in einem Bereich höherer Sensitivität der Lösungskurve verursacht, so daß die Ordnung des Prädiktors erhöht werden muß.

Bei einer vorausgegangen Erhöhung der Ordnung muß der vom Prädiktor verursachte Rechenaufwand mit dem des Korrektors verglichen werden.
Wenn $\frac{N_{pr}^{i-1}}{N_{co}^{i-1}} < \frac{N_{pr}^{i}}{N_{co}^{i}}$,ist der für den Prädiktor benötigte Rechenaufwand bzgl. dem des Korrektors gestiegen. Die Sensitivität der Lösungskurve hat also abgenommen und die Ordnung kann verringert werden.
Gilt dagegen $\frac{N_{pr}^{i-1}}{N_{co}^{i-1}} \ge \frac{N_{pr}^{i}}{N_{co}^{i}}$, hat der Rechenaufwand des Prädiktors bzgl. dem des Korrektors nicht zugenommen. Der Anstieg des Rechenaufwands wurde also vom Korrektor verursacht, so daß die Ordnung des Prädiktors erhöht werden muß.

2. $W^i < W^{i-1}$

Bei einer Verringerung des Rechenaufwands kann die Ordnung für den nächsten Prädiktor beibehalten werden. Für den Fall daß $\frac{N_{pr}^{i-1}}{N_{co}^{i-1}} < \frac{N_{pr}^{i}}{N_{co}^{i}}$, ist zusätzlich der vom Korrektor verursachte Rechenaufwand bzgl. dem des Prädiktors gesunken. Die Sensitivität der Lösungskurve wurde dann im entsprechenden Bereich geringer, so daß die Ordnung im nächsten Schritt weiter reduziert werden kann.

3. $W^i \approx W^{i-1}$

Bleibt der Rechenaufwand annähernd gleich, war die Wahl des Prädiktors für den aktuellen Bereich der Lösungskurve erfolgreich. Die Ordnung kann dann für den nächsten Prädiktor beibehalten werden. Um eine schleichende Erhöhung bzw. Verringerung des Rechenaufwands zu erkennen, wird dieser in diesem Fall zusätzlich für die letzten drei Homotopieschritte verglichen. Zeichnet sich ein Trend in eine Richtung ab, wird die Ordnung gemäß Fall 1 bzw. 2 angepaßt.

4.8 Steuerung der Homotopieschrittweite

Aus der Ordnung des Prädiktors kann mit (4.6-1) und der Lipschitz - Bedingung

$$\left\| \frac{h_x(y,p) - h_x(z,p)}{h_x(\tilde{x},p)} \right\| \le l \cdot \|y - z\| \quad \forall y,z \in D(h_x) \subseteq \Re$$
$$\text{mit } h_x = \frac{\partial h}{\partial x},\ l \in \Re \text{ konstant,} \tag{4.8-1}$$

die maximale Homotopieschrittweite Δp berechnet werden. Es folgt [8]

$$\Delta p = \left(\frac{\sqrt{2 \cdot tol \cdot l + 1} - 1}{c \cdot l} \right)^{\frac{1}{u+1}}.$$

Dabei dient die Lipschitzbedingung (4.8-1) zur Abschätzung des Toleranzbereichs des Korrektors, aus (4.6-1) folgt der "worst - case Verlauf" des Prädiktors. Der auf die p-Achse projizierte Schnittpunkt des Prädiktors mit dem Toleranzbereich des Korrektors ergibt die maximale Schrittweite Δp.

Da in der Praxis die exakten Werte der Konstanten c und l aus (4.6-1) bzw. (4.8-1) nicht bekannt sind, werden diese durch entsprechende Approximationen ersetzt.

Abbildung 4-10 zeigt die Bestimmung der maximalen Schrittweite für den ersten Homotopie - Schritt am Beispiel des Euler Prädiktors.

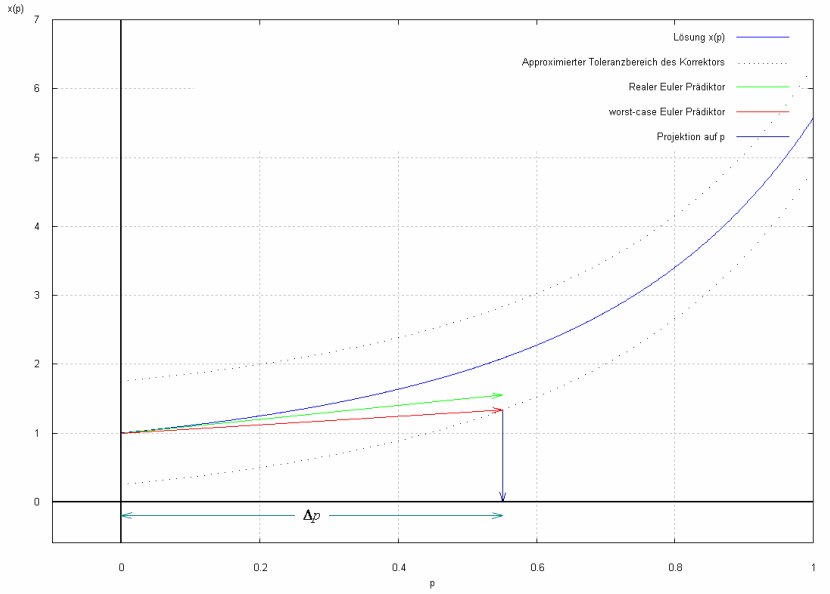

Abbildung 4-10: Bestimmung der maximalen Homotopieschrittweite

5 HOMPACK

HOMPACK als eine Sammlung von Fortran - Algorithmen zur Lösung nichtlinearer Gleichungssystemen mit Hilfe von Homotopieverfahren wurde bereits 1987 von L.T. Watson, S.C. Billups und A.P. Morgan entwickelt [11]. Im Laufe der Jahre überarbeitet und an aktuelle Fortran - Versionen angepaßt [13], hat sich jedoch an den ursprünglichen Lösungsmethoden, die im weiteren Verlauf dieses Kapitel vorgestellt werden sollen, nur wenig verändert.
HOMPACK bietet drei verschiedene Ansätze für Homotopieverfahren, von denen zwei, wie auch das im vorigen Kapitel vorgestellte Verfahren von E. Grigat nach dem Prädiktor - Korrektor Prinzip arbeiten.

Im Bezug auf die in 3.4 vorgestellten Probleme werden in HOMPACK neben dem die Deformation steuernden Homotopieparameter p die zusätzlichen Parameter a zur Variation des Startproblems, und l zur Parametrisierung der Lösungskurve bzgl. deren Länge eingeführt. Die entsprechende Homotopie ergibt sich also zu

$$h_a(x, p(l)) = p(l) \cdot f(x) + (1 - p(l)) \cdot (g(x) - a) \text{ mit}$$
$$f : \Re^n \to \Re^m,\ f(x) \overset{!}{=} 0 \text{ Zielproblem,}$$
$$g : \Re^n \to \Re^m,\ g(x) \overset{!}{=} 0 \text{ Startproblem,}$$
$$p : L \subset \Re \to [0,1],\ l \in \Re,\ x, a \in \Re^n.$$

Hieraus folgt

$$h_a(x(p(l)), p(l)) = 0 \text{ für alle } l \in L.$$

5.1 ODE basiertes Verfahren

Das ODE basierte Verfahren verfolgt analog zu den Prädiktoren von E. Grigat den Ansatz, durch Differenzieren der Homotopie $h_a(x(p(l)), p(l))$ nach der Länge l der Lösungskurve das Anfangswertproblem

$$\frac{dx_a(p(l))}{dl} = -\left(\frac{\partial h_a(x_a(p(l)), p(l)}{\partial x_a}\right)^{-1} \cdot \frac{\partial h_a(x_a(p(l)), p(l)}{\partial p} \cdot \frac{dp}{dl},$$
$$\text{mit } x_a(0) = a,\ p(0) = 0,$$

zu erzeugen, welches dann mit numerischen Verfahren zur Behandlung von gewöhnlichen Differentialgleichungen (Ordinary Differential Equations) gelöst werden kann. Der dabei entstehende Approximationsfehler wird durch Ausnutzung des Davidenko - Flow korrigiert. Dieses in den Abbildungen 5-1 bis 5-5 für die Homotopie

$$h_a(x, p(l)) = p(l) \cdot f(x) + (1 - p(l)) \cdot (g(x) - a),$$
$$a \in \Re,\ f, g : \Re \to \Re \text{ mit } f(x) = x^3 + 0.5,\ g(x) = x$$

dargestellte Phänomen beschreibt den unterschiedlichen Verlauf der Lösungskurve $x_a(p(l))$ für verschiedene Werte des Parameters a.

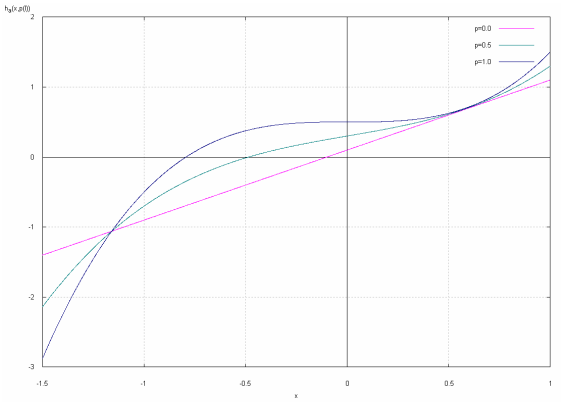

Abbildung 5-1:
$h_a(x, p(l))$ für a=-0.1

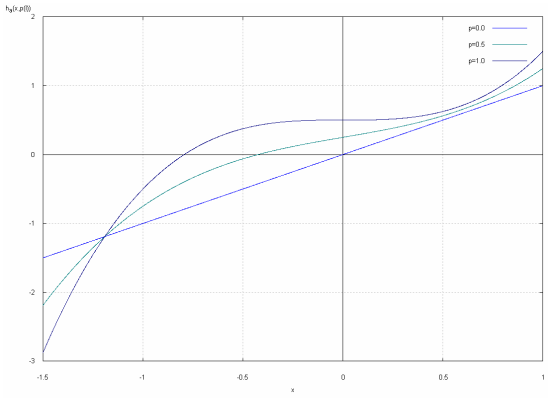

Abbildung 5-2:
$h_a(x, p(l))$ für a=0.0

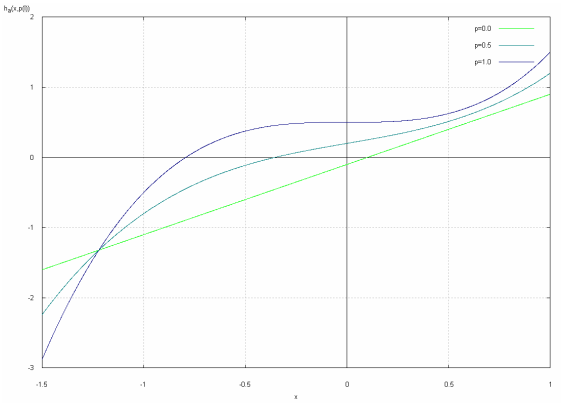

Abbildung 5-3:
$h_a(x, p(l))$ für a=0.1

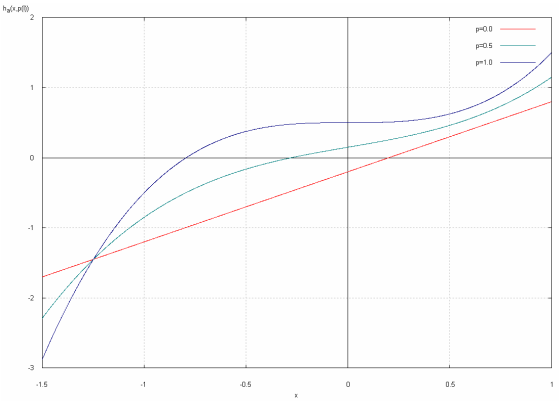

Abbildung 5-4:
$h_a(x, p(l))$ für a=0.2

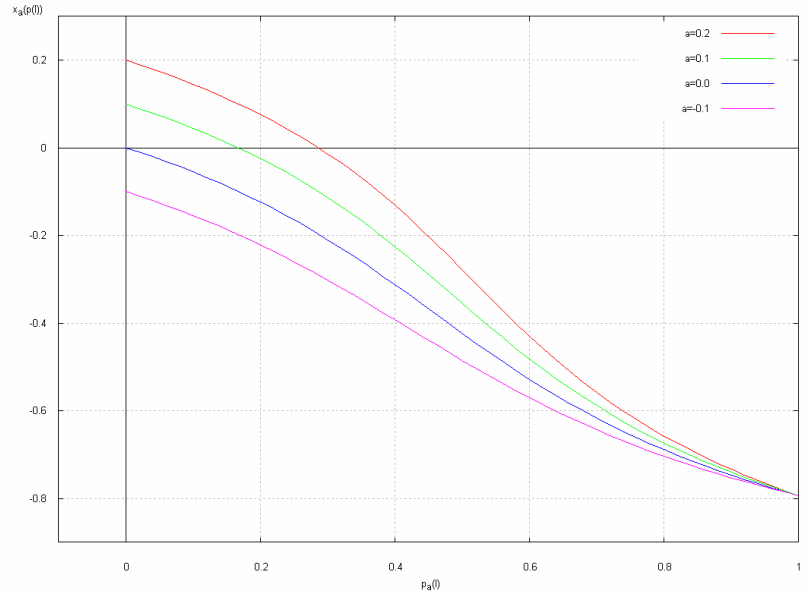

Abbildung 5-5: Davidenko - Flow der Lösungskurve $x_a(p(l))$

Prädiktor - Korrektor basierte Verfahren führen die im Prädiktor ermittelte Näherung $\tilde{x}_a(p(l))$ der exakten Lösung $x_a(p(l))$ mit Hilfe des Korrektors auf die entsprechende Lösungskurve zurück.
Im Gegensatz dazu wird im ODE - basierten Verfahren aus HOMPACK bei einer Unterschreitung der für die Lösung des Zielproblems gewünschten Genauigkeit die Lösungskurve durch Änderung des Parameters a so angepaßt, daß sie durch die vom Lösungsverfahren für das Anfangswertproblem erzeugte Näherung $\tilde{x}_a(p(l))$ verläuft. Da alle Lösungskurven des Davidenko - Flow für das Zielproblem f im Punkt $p(l)=1$ identisch sind, kann dann im weiteren Verlauf die angepaßte Lösungskurve verfolgt werden.
Die bis zur Unterschreitung der gewünschten Genauigkeit jeweils mögliche, maximale Schrittweite für das ODE - Verfahren kann wieder mittels (4.6-1) abgeschätzt werden.

5.2 Erstes Prädiktor - Korrektor Verfahren: "Normal Flow"

Abweichend von E. Grigat's Verfahren basiert der Prädiktor des Normal - Flow Verfahrens aus HOMPACK nicht auf numerischen Integrationsverfahren zur Lösung von Anfangswertproblemen, sondern wird durch Hermit - Kubische Interpolation zweier Punkte

$$P_1(l_1)=(x(p(l_1)),p(l_1)) \text{ und } P_2(l_2)=(x(p(l_2)),p(l_2))$$

mit entsprechenden Tangenten

$$P_1'(l_1) = \left(\frac{dx}{dl}(l_1), \frac{dp}{dl}(l_1) \right) \text{ und } P_2'(l_2) = \left(\frac{dx}{dl}(l_2), \frac{dp}{dl}(l_2) \right)$$

auf der Lösungskurve $x(p(l))$ gegeben.
Abbildung 5-6 zeigt den Hermit - Kubischen Prädiktor für den Lösungsverlauf der Homotopie

$$h(x, p(l)) = p(l) \cdot f(x) + (1 - p(l)) \cdot g(x),$$
$$f, g : \Re \to \Re \text{ mit } f(x) = x^3 + 0.5,\ g(x) = x,$$

an den Stützstellen $p_1(l_1) = 0$ und $p_2(l_2) = 0.25$.

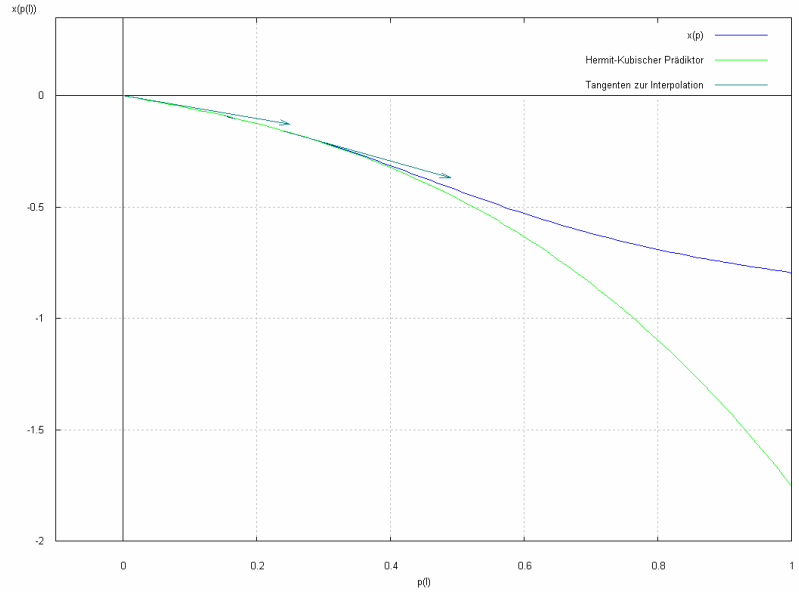

Abbildung 5-6: Hermit-Kubischer Prädiktor

Der Korrektor des Normal - Flow Verfahrens besteht aus einer Implementierung des Newton - Verfahrens für den mehrdimensionalen Fall. Die Newton - Iteration wird durch die Vorschrift

$$(x, p(l))_{i+1}^{Newton} = (x, p(l))_i^{Newton} + \left[Dh((x, p(l))_i^{Newton} \right]_{MP}^{-1} \cdot h((x, p(l))_i^{Newton}$$

beschrieben, wobei $\left[Dh((x, p(l))_i^{Newton} \right]_{MP}^{-1}$ die Moore - Penrose Pseudoinverse der Jakobi - Matrix von h bezeichnet. Im Gegensatz zu der von E. Grigat verwendeten Korrektor - Iteration ist hier also auch der Homotopieparameter $p(l)$ eine Variable im Newton - Verfahren des Korrektors.
Da während des Newton - Verfahrens die Länge l der Lösungskurve konstant bleibt, verlaufen die Iterationen nicht wie während des Korrek-

tors von E. Grigat orthogonal zu p, sondern ausgehend von der im Prädiktor ermittelten Näherung $\tilde{x}(p(l))$ entlang der Normale des Davidenko - Flow, woraus sich die Bezeichnung Normal - Flow Verfahren ableitet.

Ziel der Schrittweitensteuerung ist, wie auch bei E. Grigat, den vom Korrektor benötigten Rechenaufwand möglichst gering zu halten. Hierfür werden verschiedene Quotienten zur Bewertung des Konvergenzverhaltens des Korrektors definiert, anhand derer die maximal mögliche Schrittweite abgeschätzt werden kann.

5.3 Zweites Prädiktor - Korrektor Verfahren: "Augmented Jacobian"

Der Prädiktor des Augmented - Jacobian Verfahrens wird wie auch im Normal - Flow Verfahren mittels Hermit - Kubischer Interpolation realisiert.

Für den Korrektor wird die Jakobimatrix

$$Dh((x, p(l))_i^{Newton}$$

um die Tangente am zweiten zur Interpolation verwendeten Punkt auf der Lösungskurve erweitert ("Augmented Jacobian").
Das im Korrektor zu lösende Gleichungssystem ergibt sich so zu

$$\begin{pmatrix} h(x, p(l)) \\ (P_2'(l_2))^t \cdot ((x, p(l)) - (\tilde{x}, \tilde{p}(l))) \end{pmatrix} = 0.$$

Die Newton - Iteration verläuft also nicht wie beim Normal - Flow Verfahren entlang der Normale des Davidenko - Flow, sondern orthogonal zur Tangente am Punkt P_2.
Die Erweiterung der Jakobimatrix führt in Verbindung mit einem entsprechend angepaßten Newton - Verfahren zu einer Verringerung des Rechenaufwands, da zur Berechnung der Tangente benötigte Schritte im Korrektor wiederverwendet werden können.

Das Ziel der Schrittweitensteuerung ist wie beim Normal - Flow Verfahren, den vom Korrektor benötigten Rechenaufwand konstant (und gering) zu halten. Grundlage für die Berechnung der optimalen Schrittweite ist hier allerdings nicht mittelbar das Konvergenzverhalten des Korrektors, sondern eine Approximation der Krümmung der Lösungskurve.

5.4 Viele Möglichkeiten

Weder gibt es von den drei vorgestellten Verfahren in HOMPACK "das" Beste, noch läßt sich ein Regelwerk bestimmen, welches erlauben würde, für ein gegebenes Problem das Verfahren auszuwählen, mit dessen Hilfe die Lösung am schnellsten gefunden werden kann.

Das auch schon an verschiedenen Stellen einer Lösungskurve durchaus Unterschiede zwischen den Verfahren bestehen, zeigt der folgende grafische Vergleich des Normal - Flow und des Augmented - Jacobian Korrektors. Zur Vereinfachung wurde statt des Hermit - Kubischen ein linearer Euler - Prädiktor verwendet und der Normal - Flow Korrektor ebenfalls linear, orthogonal zur Lösungskurve dargestellt. Der qualitative Unterschied der beiden Korrektor - Verfahren bleibt jedoch erhalten. Generell gilt, daß die Wahrscheinlichkeit für die Konvergenz des Korrektors steigt, je näher sich die vom Prädiktor ermittelte Startnäherung bzgl. des jeweiligen Korrektors an der Lösungskurve befindet, je kürzer also der Weg ist, der vom Korrektor zurückgelegt werden muß.

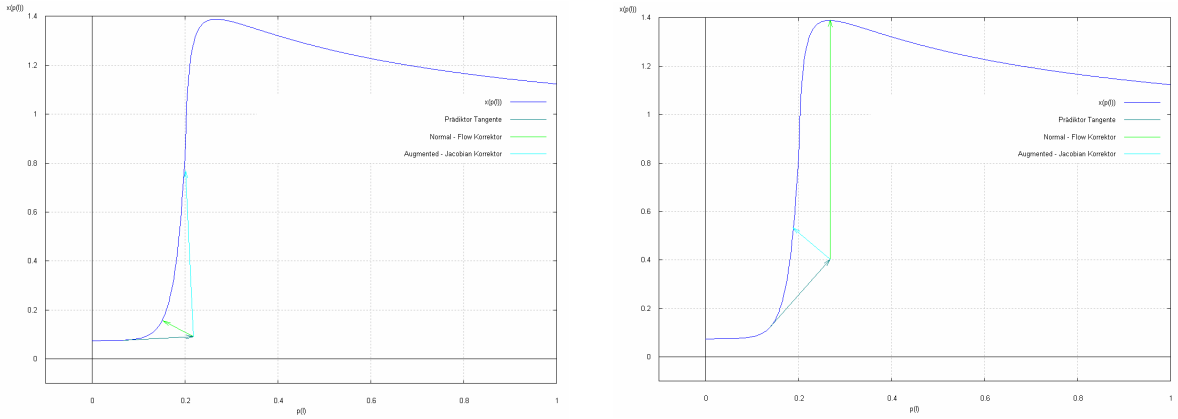

Abbildung 5-7: Normal - Flow und Augmented - Jacobian Korrektor

6 SICOM

Während der letzten beiden Kapitel wurden zwei, in der Praxis eingesetzte Homotopieverfahren und deren unterschiedliche Ansätze vorgestellt. Gemeinsam sind diesen Verfahren die enge Verbindung mit dem jeweiligen numerischen Lösungsverfahren, sowie der daraus resultierende, intensive Datenaustausch. So muß etwa zur Aufstellung der von beiden Verfahren benötigten Davidenko - Differentialgleichung dem Homotopieverfahren das zu lösende Problem und die verwendete Homotopie bekannt sein. Ebenso werden zur Steuerung der Homotopieschrittweite die Ergebnisse des numerischen Lösungsverfahrens innerhalb des Homotopieverfahrens benötigt. In Gegenrichtung müssen dem numerischen Lösungsverfahren die durch das Homotopieverfahren ermittelten Startnäherungen zur Lösung der entstehenden Zwischenprobleme übergeben werden. Zum Teil werden sogar erforderliche Berechnungsschritte gegenseitig wiederverwendet.

SICOM wurde mit der Zielsetzung entwickelt, weitestgehend unabhängig bzgl. des verwendeten numerischen Lösungsverfahrens zu arbeiten. Zu diesem Zweck wurde der erforderliche Datenaustausch auf das benötigte Minimum reduziert. Die Information, die SICOM über das verwendete numerische Lösungsverfahren benötigt, ist beschränkt darauf, wie das Verfahren aufgerufen werden kann und ob die Berechnung eines Problems erfolgreich war. Basierend darauf liefert SICOM als Rückgabe neue Werte der zu steuernden Homotopieparameter. SICOM ist hier nicht auf einen Homotopieparameter beschränkt. Theoretisch ist die Steuerung beliebig vieler Parameter unabhängig voneinander möglich.
Über die Benutzerschnittstelle von SICOM können Ergebnisse des externen numerischen Lösungsverfahrens gespeichert und geladen werden. Berechnete Lösungen können so dem externen Verfahren als Startnäherung für die Berechnung nachfolgender Zwischenprobleme zur Verfügung gestellt werden, ohne daß die Lösung bzw. deren Format SICOM explizit bekannt ist. Im Bezug auf das in Kapitel 4 vorgestellte Homotopieverfahren von E. Grigat ist SICOM also ein Prädiktor - Korrektor Verfahren erster Ordnung.

Durch die Minimalisierung der Schnittstelle kann SICOM mit einem beliebigen numerischen Lösungsverfahren kombiniert werden, ohne eventuell weitreichende Anpassungen des Verfahrens vornehmen zu müssen, die zum Teil, insbesondere wenn der entsprechende Quellcode bzw. die benötigten Kenntnisse dessen nicht zur Verfügung stehen, unmöglich sind.

SICOM wurde unter Linux für den GNU - Compiler in C++ implementiert. Um die Unabhängigkeit von SICOM auch bzgl. der verwendeten Plattform zu gewährleisten wurden jedoch ausschließlich die C / C++ Standardbibliotheken verwendet. Der Einsatz von SICOM in Verbindung mit anderen Betriebssystemen und C++ Compilern ist also möglich. Erfolgreich getestet wurde SICOM auch in Verbindung mit dem Cygnus C++ Compiler und dem Visual C++ Compiler unter Windows XP.

6.1 Die Schnittstellen

Die zur Verbindung zwischen SICOM und einem numerischen Lösungsverfahren benötigte Schnittstelle, sowie die Benutzerschnittstelle werden in den drei C++ Objektklassen <Homotopy>, <Parameter> und <Result> zur Verfügung gestellt. Eine detaillierte Beschreibung dieser Klassen sowie ein Anwendungsbeispiel sind im Anhang zu finden. Abbildung 6-1 zeigt eine schematische Darstellung der Kommunikation von SICOM mit dem numerischen Lösungsverfahren aus der Sicht des Benutzers.

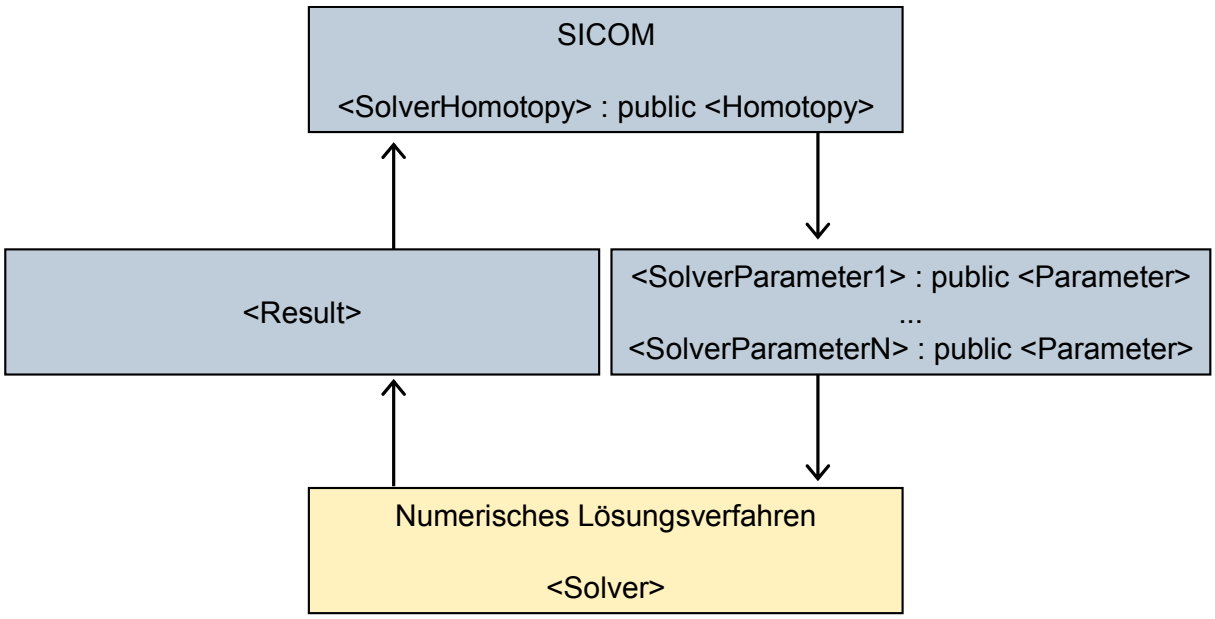

Abbildung 6-1:
Kommunikation von SICOM und dem numerischen Lösungsverfahren

6.1.1 Benutzerebene der abstrakten Klasse <Homotopy>

Die abstrakte Klasse <Homotopy> stellt verschiedene virtuelle Methoden zur Verfügung, die durch den Benutzer in einer abgeleiteten Klasse implementiert werden sollen. Sie dienen zum Aufruf des externen numerischen Lösungsverfahrens, sowie zum (externen) Speichern und Laden dessen Ergebnisse.

Weiterhin enthält <Homotopy> Methoden zur Übergabe der zu steuernden Homotopieparameter an SICOM, zum An- bzw. Ausschalten des grafischen und textbasierten Trace, zur Definition der maximalen Anzahl der Homotopieschritte und zum Start des Homotopieverfahrens.

6.1.2 Benutzerebene der Klasse <Result>

Mit Hilfe der in der Klasse <Result> bereitgestellten Methoden wird SICOM die zur Schrittweitensteuerung benötigte Information über die erfolgreiche oder fehlgeschlagene Berechnung eines Zwischenproblems übergeben. Zusätzlich kann die Anzahl der für die Berechnung notwendigen Iterationen des externen Verfahrens angegeben werden. Diese wird allerdings nur dann benötigt, wenn die Schrittweite basierend auf dieser Information gesteuert werden soll.

6.1.3 Benutzerebene der abstrakten Klasse <Parameter>

Da die Änderung eines Homotopieparameters durch SICOM weitere notwendigen Anpassungen innerhalb des externen Lösungsverfahrens nach sich ziehen kann, wird in <Parameter> eine virtuelle Methode zur Verfügung gestellt, durch deren Aufruf SICOM den geänderten Wert extern bekanntgeben kann.
Die weiteren Methoden dieser Klasse dienen zur Definition von Start- und Zielwert des Parameters, initialem, minimalem und maximalem Intervall der Schrittweite sowie zur Auswahl und Anpassung der Variante der Schrittweitensteuerung.

6.2 Die internen Klassen

Die bisher vorgestellten Klassen <Homotopy>, <Parameter> und <Result> enthalten neben den für die Benutzerschnittstelle benötigten, öffentlichen Methoden weitere Methoden und Variablen, die ausschließlich zur Verwendung innerhalb von SICOM bestimmt sind und deshalb im privaten Abschnitt der entsprechenden Klasse deklariert werden. So wird vermieden, daß diese von SICOM benötigten Bestandteile in der vom Benutzer abgeleiteten Klasse überladen werden.
Da der Zugriff auf private Bereiche allerdings nur innerhalb der jeweiligen Klasse erfolgen kann, werden die übrigen SICOM - Klassen als befreundete Klassen der Benutzerschnittstelle definiert. So wird die Beschränkung des Zugriffs aufgehoben, während die Integrität der privaten Methoden und Variablen gewährleistet bleibt.

Die Beziehungen aller Klassen werden in Abbildung 6-2 dargestellt. Die Hierarchische Anordnung der Klassen <Homotopy>, <Parameter> sowie <ForwardStepController>, <BackstepController> und deren abgeleiteter Klassen resultiert aus der Position dieser Klassen auf unterschiedlichen Abstraktionsebenen bzgl. des Ablaufs des Homotopieverfahrens.

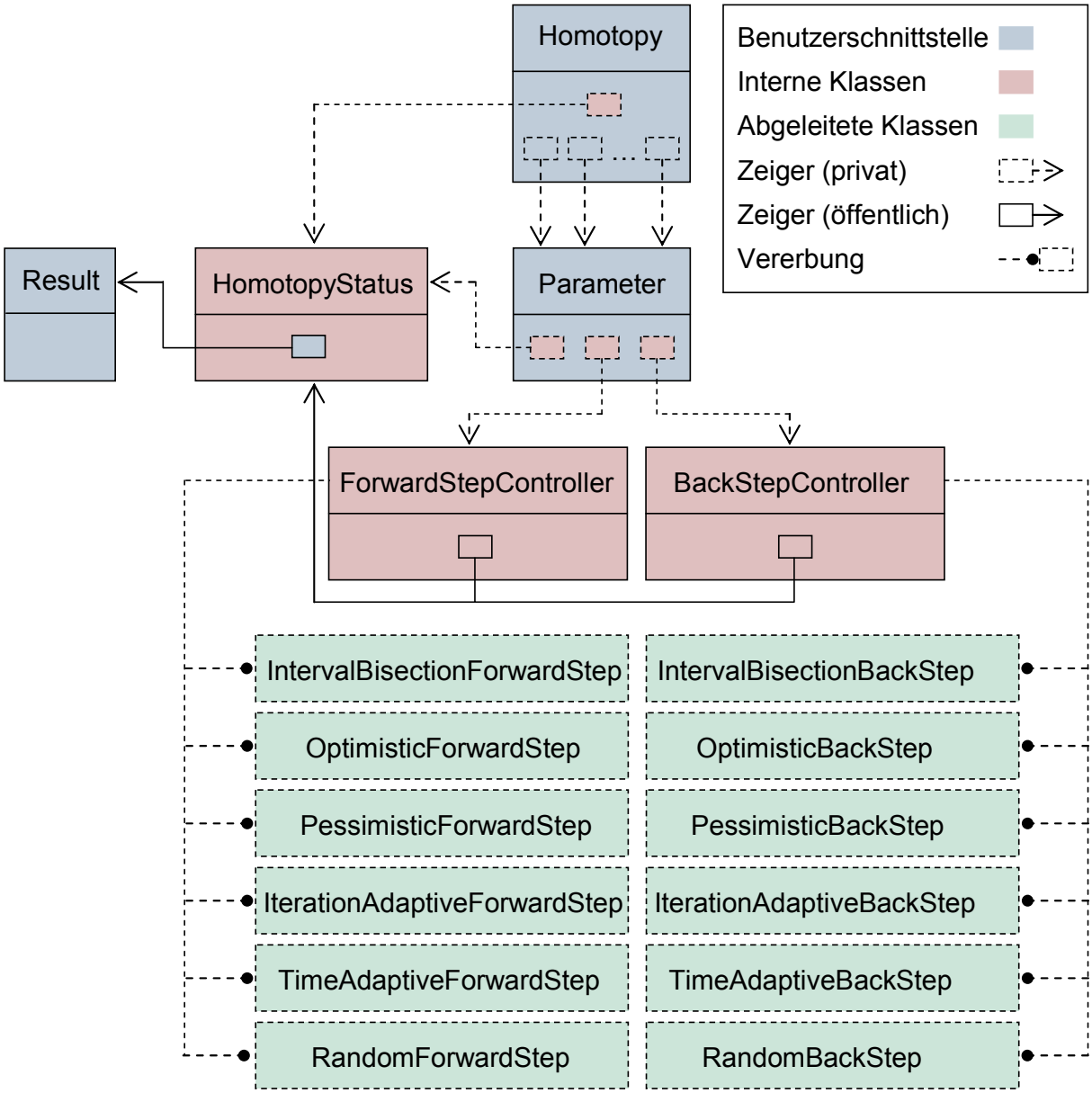

Abbildung 6-2: SICOM - Klassen und deren Beziehungen

6.2.1 Interne Ebene der Klasse <Result>

Die Klasse <Result> stellt auf privater Ebene Methoden zur Verfügung, die das Abrufen der, wie in 6.1.2 beschrieben, durch den Benutzer angegebenen Informationen über das Ergebnis des externen numerischen Lösungsverfahrens ermöglichen.

6.2.2 Die Klasse <HomotopyStatus>

Der Datenaustausch der einzelnen Klassen wird über die Klasse <HomotopyStatus> realisiert. Das entsprechende Objekt wird auf oberster Ebene der Abstraktionshierarchie in der Klasse <Homotopy> erzeugt. Ein Zeiger auf dieses Objekt wird zur jeweils nächsten Ebene weitergereicht. Alle Informationen, die klassenübergreifend benötigt werden, können so in <HomotopyStatus> gespeichert und abgerufen werden.

6.2.3 Interne Ebene der Klasse <Homotopy>

In der Klasse <Homotopy> wird das Homotopieverfahren auf der obersten Abstraktionsebene realisiert. Der Ablauf des Homotopieverfahrens auf dieser Ebene wird durch den in Abbildung 6-3 gezeigten deterministischen, endlichen Automaten gesteuert, der durch den Benutzer im öffentlichen Teil der Klasse <Homotopy> gestartet werden kann.
Anhand der über den indirekten Zeiger auf die Klasse <Result> erreichbaren Information, ob ein Problem durch das externe numerische Lösungsverfahren erfolgreich gelöst werden konnte, wird entschieden, ob ein Vorwärts- oder ein Rückwärtsschritt ausgeführt, werden soll. Die entsprechende Entscheidung wird an alle zu steuernden Homotopieparameter weitergeleitet. Im privaten Bereich enthält <Homotopy> dafür Zeiger auf diese Parameter, die in Form einer einfach verketteten Liste gespeichert werden.
Die für die übrigen Zustandsübergänge benötigten Informationen über den Status der einzelnen Parameter, wie das Erreichen des Zielwertes oder der minimalen Schrittweite, können aus <HomotopyStatus> abgerufen werden.
Ist eine Lösung des Zielproblems auch im Bezug auf die in 3.4 vorgestellten Probleme mit Hilfe des Homotopieverfahrens nicht möglich, wird ein Fehlerzustand erreicht und eine entsprechende Fehlermeldung ausgegeben. So kann z.B. eine Unterbrochene Lösungskurve $x(p)$ oder das Erreichen eines Punktes p mit

$$\lim_{\Delta p \to 0} \frac{x(p+\Delta p) - x(p)}{\Delta p} = \infty ,$$

der auf einen Verlauf der Lösungskurve wie in Abbildung 3-19 dargestellt schließen läßt, durch das Unterschreiten der minimalen Schrittweite Δp_{min} der Parameter erkannt werden. Da in diesen Fällen nach einem

Vorwärtsschritt mit beliebig kleinem Δp keine Lösung des entstehenden Zwischenproblems berechnet werden kann, werden Rückwärtsschritte ausgeführt bis Δp_{min} erreicht und damit der Übergang in den Fehlerzustand ausgelöst wird.

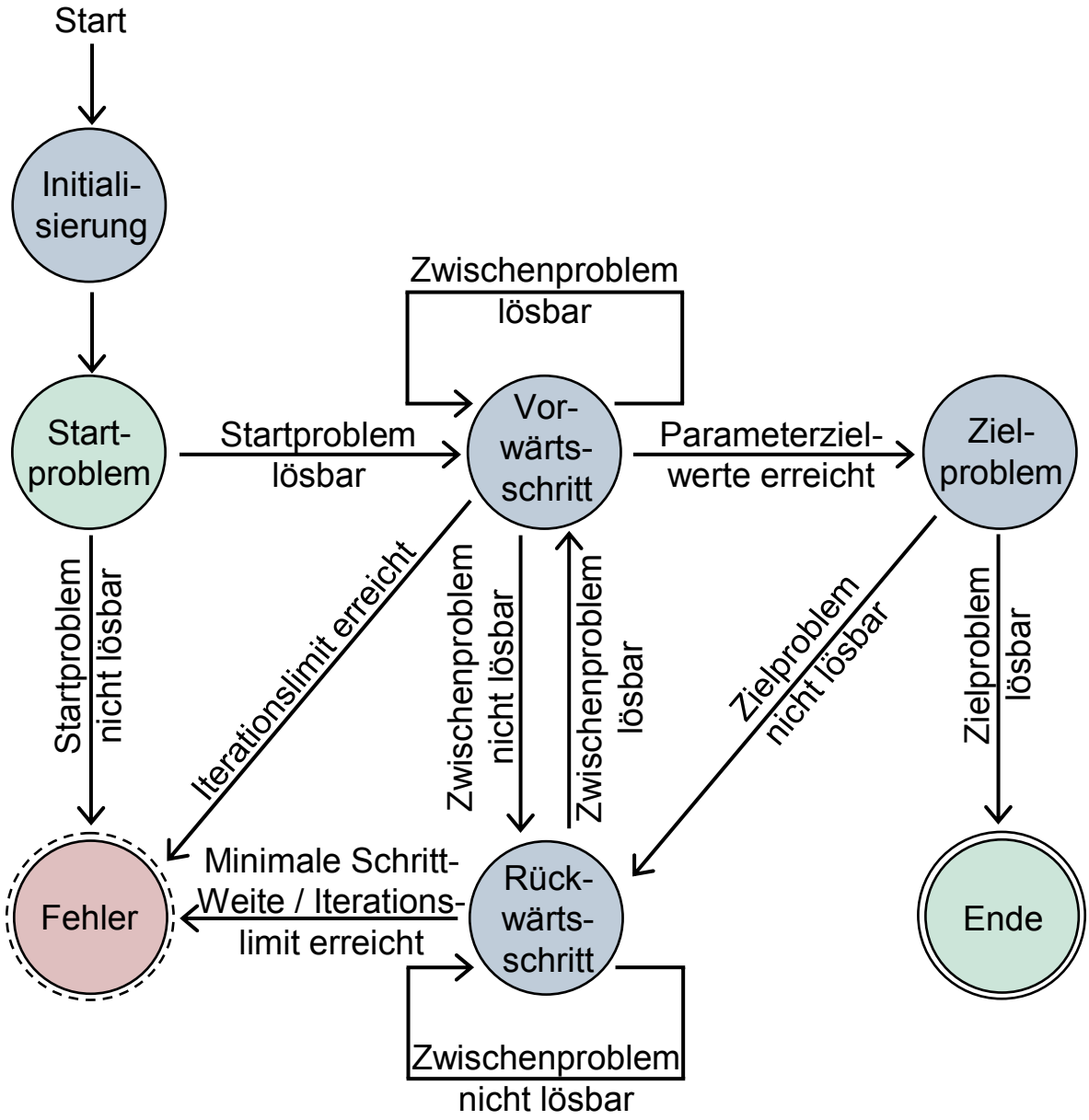

Abbildung 6-3:
Deterministischer, endlicher Automat des Homotopieverfahrens

Wird vom Benutzer ein Trace des Homotopieverfahrens gewünscht, erzeugt <Homotopy> die textbasierte Ausgabe in einer Konsole. Weiterhin werden Dateien geschrieben, die Informationen über den Verlauf der Parameterwerte und die vom externen numerischen Lösungsverfahren pro Aufruf benötigte CPU - Zeit und, falls angegeben,

dessen Iterationen enthalten. Zur grafischen Ausgabe dieser Informationen wird eine GNUplot - Steuerdatei erzeugt.
Die von SICOM intern ermittelte CPU - Zeit für den jeweils letzten Aufruf des externen Verfahrens wird auch für eine Variante der Schrittweitensteuerung benötigt und deshalb in der Klasse <HomotopyStatus> gespeichert.

6.2.4 Interne Ebene der Klasse <Parameter>

Die Klasse <Parameter> enthält Variablen für den Start-, Ziel- und aktuellen Wert des Parameters und dessen minimale, maximale und aktuelle Schrittweite. Methoden zur Beeinflussung und Abfrage dieser Werte werden bereitgestellt. Diese Methoden gewährleisten auch, daß der Wert und die Schrittweite des Parameters innerhalb der vorgegebenen Grenzen bleiben. So muß z.B. nicht in jeder Variante der Schrittweitensteuerung geprüft werden, ob der Zielwert des Parameters durch die neu berechnete Schrittweite überschritten wird. Ist das der Fall, wird die Schrittweite durch die Klasse <Parameter> entsprechend angepaßt.
Bei Erreichen des Zielwertes oder der minimalen Schrittweite eines Parameters, wird diese, zur Steuerung des im vorigen Abschnitt beschriebenen Automaten benötigte Information an die Klasse <HomotopyStatus> weitergeleitet.
Während der Initialisierungsphase des Automaten findet eine Plausibilitätsprüfung der durch den Benutzer angegebenen Werte für die Schrittweite statt. Sind diese fehlerhaft oder nicht angegeben, werden sie durch Standardwerte ersetzt, bevor das Homotopieverfahren gestartet wird.

6.2.5 Die abstrakten Klassen <ForwardStepController> und <BackStepController>

Verantwortlich für die Steuerung der Schrittweite für Vorwärts- bzw. Rückwärtsschritte eines Parameters sind die abstrakten Klassen <ForwardStepController> bzw. <BackStepController>. Aufgaben, die für alle Varianten der Schrittweitensteuerung identisch sind, wie z.B. die Berechnung des Parameterwertes anhand der neuen Schrittweite, werden in diesen Klassen ausgeführt. Für die von der gewählten Variante abhängige Berechnung der neuen Schrittweite steht eine virtuelle Methode zur Verfügung, die jeweils in den in Abbildung 6-2 dargestellten, abgeleiteten Klassen implementiert wird. Durch diese Vererbungsstruktur ist das Hinzufügen neuer Varianten zur Schrittweitensteuerung möglich, ohne Anpassungen der übrigen Klassen vornehmen zu müssen.

6.3 Verschiedene Varianten zur Schrittweitensteuerung

SICOM bietet basierend auf den zur Verfügung stehenden Informationen über die Parameter und das externe numerische Lösungsverfahren verschiedene Varianten zur Steuerung der Schrittweite s der Homotopieparameter.

Alle Varianten bestimmen die neue Schrittweite s_{neu} anhand des Abstands

$$d = p_{akt} - p_{prä}$$

zwischen dem aktuellen Wert p_{akt} eines Parameters und seinem Wert $p_{prä}$ bei der letzten erfolgreichen (externen) Berechnung eines Zwischenproblems.

Bei Vorwärtsschritten und dem ersten in einer Folge von Rückwärtsschritten entspricht d der aktuellen Schrittweite s_{akt}. Müssen mehrere Rückwärtsschritte in Folge ausgeführt werden, unterscheiden sich jedoch diese Werte.

Für die Bestimmung der neuen Schrittweite gilt generell

$$s_{neu} = d \cdot \delta .$$

Dabei ist

$\delta > 0$ für einen Vorwärtsschritt,
$0 < \delta < 1$ für einen Rückwärtsschritt.

Der neue Wert p_{neu} des Parameters ergibt sich dann durch

$p_{neu} = p_{akt} + s_{neu}$ für einen Vorwärtsschritt,
$p_{neu} = p_{akt} - s_{neu}$ für einen Rückwärtsschritt.

Mit $0 < \delta < 1$ für einen Rückwärtsschritt folgt daraus, daß auch bei mehreren aufeinanderfolgenden Rückwärtsschritten immer gilt, daß

$$p_{neu} > p_{prä} \text{ wenn } p_{start} < p_{ziel} \text{ bzw.}$$
$$p_{neu} < p_{prä} \text{ wenn } p_{start} > p_{ziel}.$$

p_{start} bezeichnet dabei den Startwert des Parameters, p_{ziel} entsprechend den Zielwert. Dies bedeutet, daß ein Parameterwert, für den das entsprechende Zwischenproblem erfolgreich gelöst werden konnte, im weiteren Verlauf, wie in Abbildung 6-4 dargestellt, ausschließlich in Richtung des Zielwertes des Parameters verändert wird. Die Verfolgung von Lösungskurven $x(p)$ wie z.B. der in Abbildung 3-19 dargestellten ist folglich nicht möglich.
SICOM setzt also zumindest eine Homotopie voraus, die eine injektive Abbildung $p \rightarrow x(p)$ erzeugt und stellt damit geringere Anforderungen an die verwendete Einbettung als das Homotopieverfahren von E. Grigat oder HOMPACK, die beide ein bzgl. p stetig differenzierbares $x(p)$ fordern.

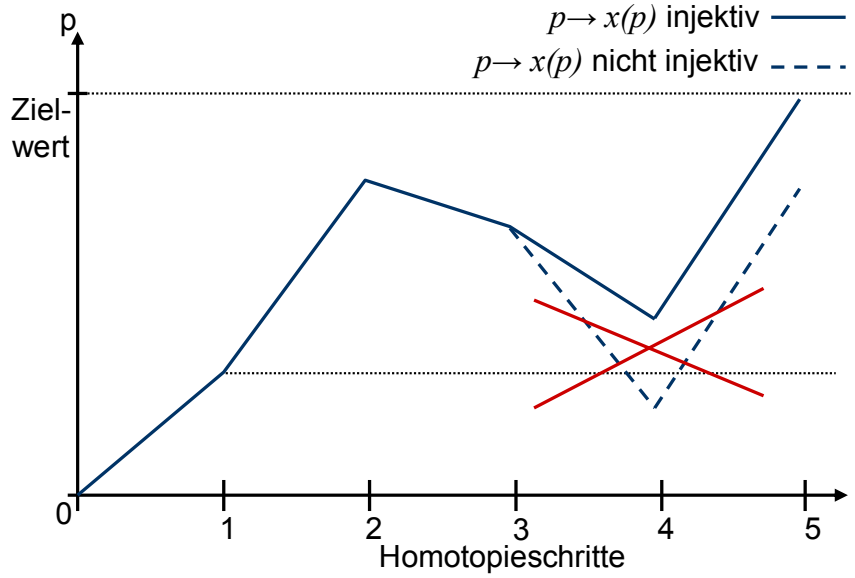

Abbildung 6-4:
Verlauf der Parameterwerte bzgl. Injektivität von $p \rightarrow x(p)$

Ein Sonderfall entsteht, wenn SICOM zur Steuerung mehrerer Homotopieparameter eingesetzt wird. Da die Schrittweitensteuerung der einzelnen Parameter unabhängig voneinander erfolgen kann, ist es durchaus möglich, daß die Zielwerte der Parameter zu verschiedenen Zeitpunkten erreicht werden.

In diesem Fall gilt nach einer erfolgreichen Auswertung des entsprechenden Zwischenproblems durch das externe numerische Lösungsverfahren

$$p_{akt} = p_{prä} = p_{ziel}$$

für alle Parameter, deren Zielwerte erreicht wurden. Daraus folgt

$$d = 0 \text{ bzw. } s_{neu} = 0$$

für alle weiteren Schritte der betreffenden Parameter, so daß auch bei evt. nachfolgenden Rückwärtsschritten die Werte dieser Parameter, wie Abbildung 6-5 zeigt, unverändert bleiben, die Deformation der Zwischenprobleme also nicht mehr beeinflussen.

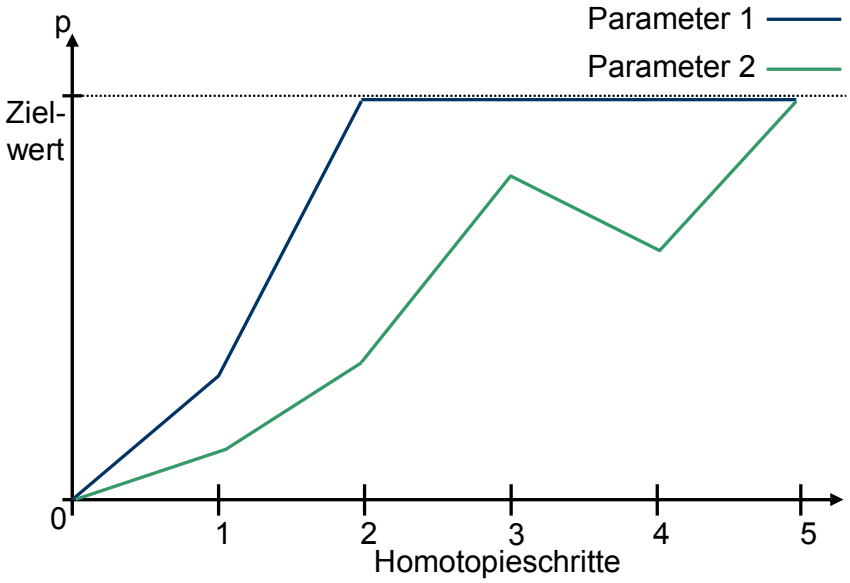

Abbildung 6-5: Konstanter Parameterverlauf bei erreichtem Zielwert

Die Berechnung von p_{neu} ist identisch für alle Methoden der Schrittweitensteuerung und wird entsprechend der in Abbildung 6-2 gezeigten Abstraktionshierarchie in den Klassen <ForwardStepController> und <BackStepController> ausgeführt.
Aufgabe der, in den von <ForwardStepController> und <BackStepController> abgeleiteten Klassen jeweils implementierten Variante der Schrittweitensteuerung ist die Berechnung von s_{neu}. Jedem zu steuernden Parameter kann unabhängig eine der nachfolgend vorgestellten Varianten zugewiesen werden. Dabei ist auch die Kombination verschiedener Methoden für Vorwärts- und Rückwärtsschritte möglich.

6.3.1 Intervallbisektion

Eine einfache Möglichkeit zur Steuerung der Schrittweite, die auch in [14] vorgeschlagen wird, ist die in den Klassen <IntervalBisectionForwardStep> und <IntervalBisectionBackStep> realisierte Intervallbisektion. Hier wird zur Bestimmung der neuen Schrittweite s_{neu} lediglich beachtet, ob ein Zwischenproblem durch das externe numerische Lösungsverfahren gelöst werden konnte.
Für den Faktor δ zur Berechnung der neuen Schrittweite gilt dann

$\delta = 2$ bei Vorwärtsschritten bzw.
$\delta = 0.5$ bei Rückwärtsschritten.

Wurde ein Zwischenproblem erfolgreich gelöst, wird also die Schrittweite des Parameters für den nächsten Homotopieschritt verdoppelt. Konnte die Lösung nicht berechnet werden, wird der Wert des Parameters auf die Mitte des Intervalls zwischen dem aktuellem Wert und dem Wert bei der letzten erfolgreichen Berechnung eines Zwischenproblems gesetzt.

Der Benutzer kann durch eine Veränderung der Werte für δ die Schrittweitensteuerung beeinflussen. Läßt sich der Verlauf der Lösungskurve $x(p)$ bzw. der Konvergenzradius des externen numerischen Lösungsverfahrens abschätzen, kann z.B. für eine Lösungskurve mit geringer Steigung und / oder großem Konvergenzradius ein großes δ für Vorwärtsschritte gewählt werden.

6.3.2 Optimistische Schrittweitensteuerung

Zusätzlich zu der Information, ob ein Zwischenproblem gelöst werden konnte, wird die Schrittweite bei optimistischer Steuerung durch die Anzahl c aufeinanderfolgender Vorwärts- bzw. Rückwärtsschritte beeinflußt. δ ergibt sich hier zu

$\delta = c + 1$ für Vorwärtsschritte und
$(\delta = c + 1)^{-1}$ für Rückwärtsschritte.

Für den jeweils ersten einer Reihe von Schritten gleicher Richtung entspricht die optimistische Schrittweitensteuerung der Intervallbisektion. Danach ergeben sich im Vergleich größere Vorwärtsschritte und kleinere Rückwärtsschritte.
Geeignet ist die optimistische Schrittweitensteuerung damit für Lösungskurven mit geringer Steigung bzw. großem Konvergenzradius. Im Bezug

auf die in 3.5 vorgestellten Schwierigkeiten bei der Schrittweitensteuerung kann die Anzahl der benötigten Homotopieschritte hier durch die schnell wachsende Vorwärtsschrittweite verringert werden. Die kleiner werdenden Rückwärtsschritte sollen im Gegenzug vermeiden, daß der optimale Wert des Parameters, für den mit der gegebenen Startnäherung wieder eine Lösung des entsprechenden Zwischenproblems möglich ist, zu weit in Richtung des Wertes bei der letzten erfolgreichen externen Auswertung überschritten wird.

Die optimistische Schrittweitensteuerung kann durch die Wahl eines anderen Startwertes für den Zähler c angepaßt werden. Die Schrittweite ändert sich dabei proportional zu c für Vorwärtsschritte, antiproportional zu c für Rückwärtsschritte. Je größer also der Startwert für c ist, desto "optimistischer" ist die Steuerung der Schrittweite.

6.3.3 Pessimistische Schrittweitensteuerung

Wie auch bei der optimistischen Steuerung, allerdings mit gegensätzlichem Verhalten, ist die Schrittweite bei der pessimistischen Steuerung abhängig von der Anzahl c aufeinanderfolgender Schritte gleicher Richtung. Für δ gilt

$$\delta = \frac{c+1}{c} \text{ bei Vorwärtsschritten und}$$

$$\delta = \frac{c}{c+1} \text{ bei Rückwärtsschritten.}$$

Wieder entspricht die Schrittweite für den jeweils ersten in einer Folge von Schritten gleicher Richtung der Intervallbisektion. Dann folgen bzgl. der Intervallbisektion kürzere Vorwärts- bzw. längere Rückwärtsschritte.
Die pessimistische Schrittweitensteuerung eignet sich dementsprechend für Lösungskurven $x(p)$ mit großer Steigung und kleinem Konvergenzradius.
Bzgl. 3.5 wird durch die kleiner werdenden Vorwärtsschritte erreicht, daß der maximale Parameterwert, für den die erfolgreiche Berechnung eines Zwischenproblems noch möglich ist, nicht massiv überschritten wird. Die wachsende Rückwärtsschrittweite führt in wenigen Schritten zu einem Zwischenproblem, daß mit der aktuellen Startnäherung wieder gelöst werden kann.

Der Startwert für c kann durch den Benutzer geändert werden. Größere Werte verstärken das "pessimistische Verhalten" der Schrittweitensteuerung.

6.3.4 Zwischenstand

Die Abbildungen 6-6 und 6-7 zeigen einen grafischen Vergleich von Intervallbisektion, optimistischer und pessimistischer Schrittweitensteuerung für Vorwärtsschritte und Rückwärtsschritte.
Ein Verlauf des Parameters wie in 6-6 entsteht dabei durch die Verfolgung einer Lösungskurve $x(p)$ mit geringer Steigung bzw. großem Konvergenzradius. Für den in 6-7 dargestellten Verlauf gelten gegenteilige Eigenschaften der Lösungskurve.

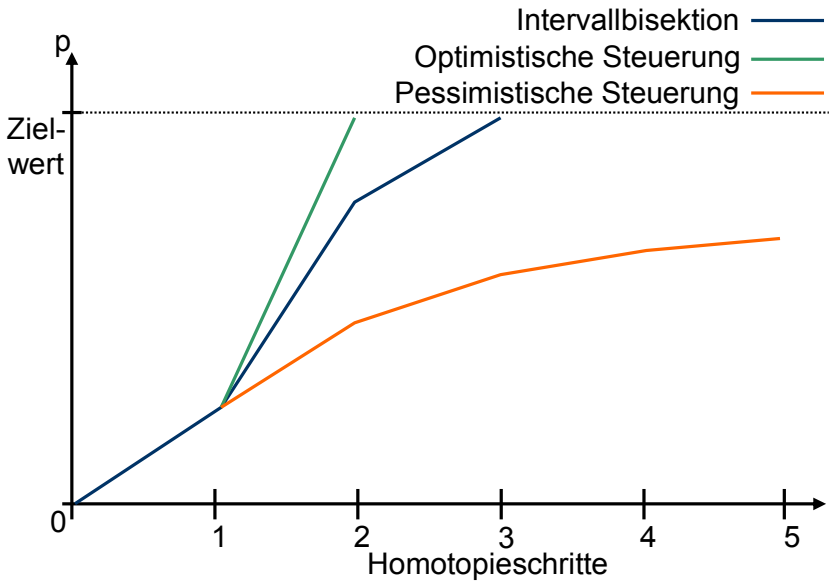

Abbildung 6-6: Vorwärtsschritte

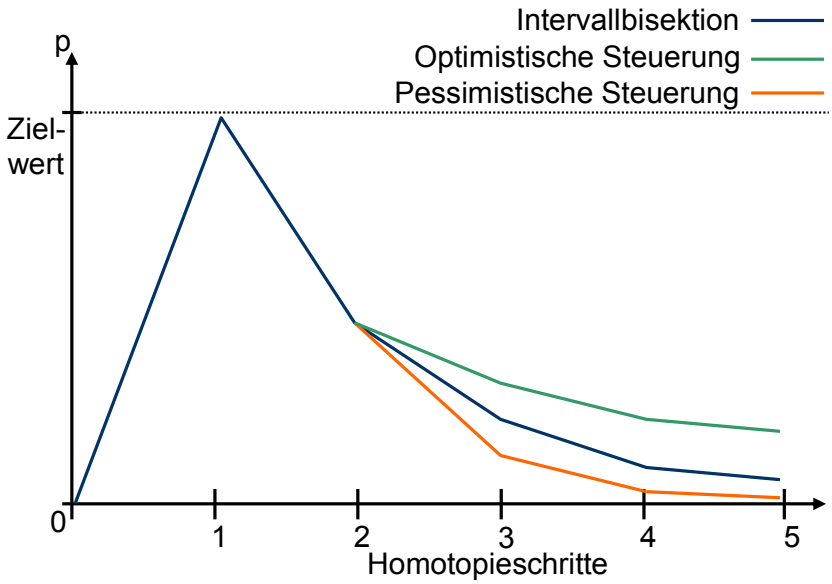

Abbildung 6-7: Rückwärtsschritte

Deutlich wird, daß die pessimistische Schrittweitensteuerung für Vorwärtsschritte zu einem starken Anstieg der Anzahl insgesamt benötigter Homotopieschritte führen kann, wenn sie für eine Lösungskurve mit geringer Steigung und / oder großem Konvergenzradius eingesetzt wird. Da in diesem Fall keine oder nur wenige Rückwärtsschritte ausgeführt werden müssen, wird die Vorwärtsschrittweite ständig weiter reduziert. Zum Erreichen des Zielwertes des Parameters werden folglich viele Schritte benötigt.
Entsprechendes gilt für optimistische Rückwärtsschritte in Verbindung mit einer Lösungskurve mit großer Steigung bzw. kleinem Konvergenzradius. Nach einem Vorwärtsschritt mit zu großer Schrittweite werden hier, da die Schrittweite der Rückwärtsschritte fortlaufend verkleinert wird, demzufolge viele Rückwärtsschritte benötigt, bevor wieder ein mit der gegebenen Startnäherung lösbares Zwischenproblem entsteht.

6.3.5 Iterations - adaptive Schrittweitensteuerung

Optimistische und pessimistische Schrittweitensteuerung bieten gute Ansätze für Lösungskurven die eine geringe globale Varianz bzgl. Steigung und Konvergenzradius aufweisen. Für verschiedene Bereiche von $x(p)$ können sich diese Eigenschaften allerdings durchaus stark unterscheiden. Zudem ist der Verlauf der Lösungskurve meist nur schwer prognostizierbar.

Die Iterations - adaptive Schrittweitensteuerung löst diese Probleme, indem die Schrittweite dynamisch an den Verlauf der Lösungskurve angepaßt wird. Der Rückschluß über den Verlauf erfolgt auf der Basis der durch das externe Verfahren pro Aufruf benötigten Iterationen. Dies ist notwendig, da SICOM der konkrete Verlauf der Lösungskurve nicht bekannt ist. Die Grundlegende Idee und das resultierende Verhalten sollen durch die Unterscheidung der beiden folgenden Extremfälle für Vorwärtsschritte, wie in Abbildung 6-8 dargestellt, verdeutlicht werden:

- Ist die Lösung eines Zwischenproblems in wenigen Iterationen des externen numerischen Lösungsverfahrens möglich, folgt, daß die durch die Lösung des letzten Problems gegebene Startnäherung nahe an der Lösung des aktuellen Zwischenproblems gelegen war. Die Lösungskurve weist also im aktuellen Bereich eine geringe Steigung auf. Für den folgenden Vorwärtsschritt kann dann eine entsprechend große Schrittweite gewählt werden.

- Waren zur Lösung des Zwischenproblems viele Iterationen nötig, läßt dies auf einen großen Abstand zwischen Startnäherung und exakter Lösung, also auf eine große Steigung im aktuellen Bereich der Lösungskurve schließen. Die Schrittweite für den nächsten Vorwärtsschritt wird dementsprechend reduziert.

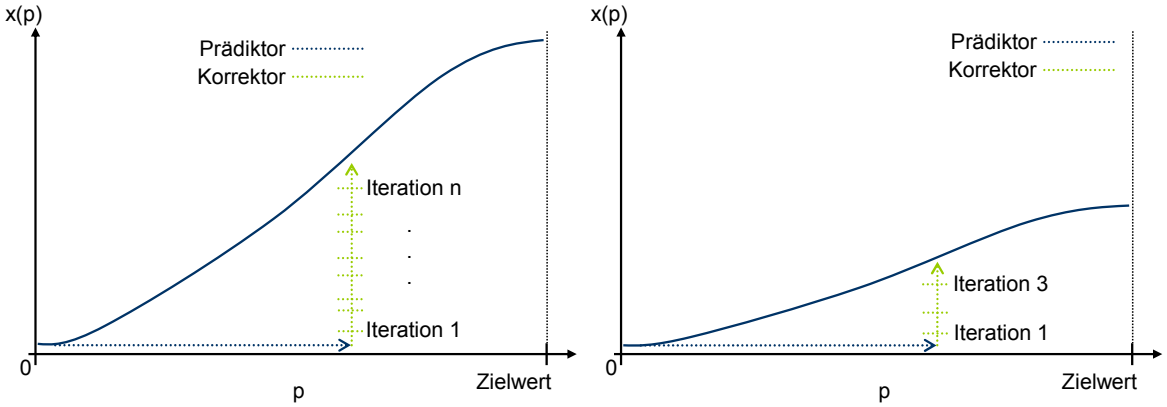

Abbildung 6-8: Iterationen des externen Verfahrens (Korrektor) bei großer und kleiner Steigung der Lösungskurve $x(p)$

Ist das externe numerische Lösungsverfahren in der Lage, das Konvergenzverhalten während der Berechnung eines Problems zu bewerten und einen Abbruch der Berechnung bei langsamer bzw. nicht gegebener Konvergenz auszulösen, können für den resultierenden Rückwärtsschritt wieder zwei Randfälle unterschieden werden:

- Erfolgte der Abbruch des externen Verfahrens nach wenigen Iterationen, war das Konvergenzverhalten ausgehend von der entsprechenden Startnäherung schlecht. Der Abstand der Startnäherung zur exakten Lösung war also wesentlich zu groß. Für den notwendigen Rückwärtsschritt wird dann eine große Schrittweite gewählt, um möglichst schnell wieder ein lösbares Zwischenproblem zu erzeugen.

- Wurden im externen Lösungsverfahren bis zum Abbruch der Berechnung viele Iterationen durchgeführt, liegt nahe, daß die Deformation des Zwischenproblems im Bezug auf die letzte erfolgreiche Auswertung nicht erheblich zu groß war. Die Schrittweite für den folgenden Rückwärtsschritt kann dementsprechend klein gewählt werden.

Abhängig von der Anzahl i der für in einem Aufruf des externen Verfahrens benötigten Iterationen ergibt sich δ zu

$$\delta = \frac{i_{ref}}{i_{akt}} \text{ für Vorwärtsschritte bzw. zu}$$

$$\delta = \left(1 + \frac{i_{ref}}{i_{akt}} \right)^{-1} \text{ für Rückwärtsschritte.}$$

Dabei bezeichnet i_{akt} die vom externen Lösungsverfahren im aktuellen Aufruf benötigten Iterationen. Wie auch in [8] wird i_{akt} bzgl. der Schrittweite normiert. Dies vermeidet, daß z.B. nach einem langen Vorwärtsschritt mit nachfolgender großer Anzahl benötigter Iterationen des externen Verfahrens, die Schrittweite zu stark reduziert wird.
Die zur Bewertung von i_{akt} benötigte Referenz i_{ref} entspricht dem Mittel der in den jeweils insgesamt vorangegangenen Schritten gleicher Richtung benötigten Iterationen des externen Verfahrens.

Da das Startproblem meist ein einfaches, also in wenigen externen Iterationen lösbares Problem darstellt, für die folgenden Zwischenprobleme aber durchaus erheblich mehr Iterationen benötigt werden können, kann der Benutzer den Startwert für i_{ref} festlegen und so die Schrittweitensteuerung in der Anfangsphase des Homotopieverfahrens beeinflussen.

6.3.6 Rechenzeit - adaptive Schrittweitensteuerung

Abweichend von der beschriebenen Minimalität der Schnittstelle von SICOM und der Unabhängigkeit bzgl. des verwendeten externen numerischen Lösungsverfahrens, basiert die Iterations - adaptive Schrittweitensteuerung auf der Anzahl der pro Aufruf des externen numerischen Lösungsverfahrens benötigten Iterationen. Dieser Wert muß bei Verwendung dieser Variante der Schrittweitensteuerung SICOM als zusätzliche Information durch den Benutzer angegeben werden.

Die Rechenzeit - adaptive Schrittweitensteuerung bietet ebenfalls eine dynamische, an der Verlauf der Lösungskurve *x(p)* angepaßte Steuerung der Schrittweite, gründet diese allerdings auf die durch das externe Verfahren pro Aufruf benötigte Rechenzeit. Da dieser Wert innerhalb von SICOM ermittelt werden kann, muß der Benutzer keine zusätzlichen Informationen bzgl. der Dauer eines Aufrufs des externen Verfahrens zur

Verfügung stellen. Als Maß für die Rechenzeit werden die durch das externe Verfahren benötigten Taktzyklen genutzt, welche mit Hilfe der C++ Standardbibliotheken unabhängig von der übrigen Auslastung des Systems bestimmt werden können. Die Rechenzeit - adaptive Schrittweitensteuerung wird also nicht durch andere laufende Programme beeinflußt.

Da die benötigte Rechenzeit des externen Verfahrens in proportionalem Verhältnis zur Anzahl der Iterationen steht, gelten für die Bestimmung der neuen Schrittweite dieselben Überlegungen wie für die Iterations - adaptive Schrittweitensteuerung. Der Wert für δ ergibt sich dementsprechend analog.

Über einen Faktor für die Gewichtung der für die Lösung des Startproblems benötigten Rechenzeit kann die Schrittweite in der Anfangsphase des Homotopieverfahrens wieder durch den Benutzer angepaßt werden.

6.3.7 Zufällige Schrittweitensteuerung

Zur Evaluierung der übrigen Varianten zur Schrittweitensteuerung und weiteren Testzwecken bietet SICOM eine auf pseudo - Zufallszahlen basierende Steuerung der Schrittweite.

Sei $rand_{seed} \in (0,1]$ eine durch die Initialisierung des Zufallszahlengenerators mit $seed$ berechnete Zufallszahl, folgt

$$s_{neu} = s_{max} \cdot rand_{seed}$$ für einen Vorwärtsschritt und
$$s_{neu} = d \cdot rand_{seed}$$ für einen Rückwärtsschritt.

Abweichend von bisher vorgestellten Schrittweitensteuerungen wird die neue Vorwärtsschrittweite hier nicht anhand des Abstands d des aktuellen Parameterwertes zu dem Wert bei der letzten erfolgreichen Auswertung eines Zwischenproblems berechnet. Die neue Schrittweite s_{neu} ergibt sich als Bruchteil der maximalen Schrittweite s_{max} des Parameters.

Da der verwendete Zufallzahlengenerator aus der C++ Standardbibliothek nach einer Initialisierung mit gleichem $seed$ immer dieselbe Folge von Zufallszahlen liefert, ist der Parameterverlauf für die zufällige Schrittweitensteuerung reproduzierbar. $seed$ kann hierfür durch den Benutzer angegeben werden. Erfolgt keine Angabe, wird $seed$ mit der aktuellen Systemzeit belegt.

6.3.8 Einfrieren der Vorwärtsschrittweite

Bei allen vorgestellten Methoden zur Schrittweitensteuerung kann zusätzlich die Vorwärtsschrittweite eingefroren werden. Dies bedeutet, daß nach einer Folge mehrerer Rückwärtsschritte die durch den letzten Rückwärtsschritt erreichte Schrittweite für die nachfolgenden Vorwärtsschritte beibehalten wird, bis der Wert des Parameters, der den ersten Rückwärtsschritt ausgelöst hat wieder erreicht bzw. überschritten wird. So kann bei Abschnitten einer Lösungskurven $x(p)$ mit extrem großer Steigung vermieden werden, daß durch eine Vergrößerung der Vorwärtsschrittweite nicht lösbare Zwischenprobleme entstehen, wodurch die Anzahl der in diesem Fall notwendigen Rückwärtsschritte reduziert wird.

6.4 Rechenaufwand von SICOM

Die von SICOM zur Steuerung der Parameter benötigte Rechenzeit ist minimal. Für jeden zu steuernden Parameter werden pro Schritt

- eine Addition zur Berechnung von d,
- maximal zwei Additionen und Divisionen für die Berechnung von δ,
- eine Addition zur Bestimmung der neuen Schrittweite und
- eine (bedingte) Addition zur Steuerung des in 6.2.3 vorgestellten endlichen Automaten benötigt.

Der Rechenaufwand von SICOM liegt mit konstantem Aufwand für die Initialisierung in $O(n)$, ist also linear bzgl. der Anzahl n benötigter Homotopieschritte.
Nicht einbezogen wird hier die zum Speichern und Laden der Zwischenergebnisse benötigte Zeit. Diese kann zwar durchaus dem Homotopieverfahren zugerechnet werden, ist aber abhängig von der durch den Benutzer gegebenen Implementierung der entsprechenden virtuellen Methoden aus der Benutzerschnittstelle von SICOM.

Mehrere Testläufe mit dem Newton - Verfahren als externem numerischen Lösungsverfahren haben gezeigt, daß in dieser Kombination SICOM (inklusive dem Speichern und Laden der Zwischenergebnisse) unter 5% der insgesamt zur Lösung eines Zielproblems benötigten Rechenzeit beansprucht.

7 Anwendungsbeispiele

Die folgenden Abschnitte zeigen einige Anwendungsbeispiele von SICOM in Kombination mit zwei verschiedenen externen numerischen Lösungsverfahren.
Zuerst sollen mit Hilfe von SICOM die in 3.2 vorgestellten Problemfälle des Newton - Verfahrens gelöst werden. In diesem Zusammenhang wird neben der Verwendung von SICOM zur Erzeugung globaler Konvergenz auch der mögliche Einsatz zur Beschleunigung des Konvergenzverhaltens externer, numerischer Lösungsverfahren demonstriert.
Die Konvergenzbeschleunigung wird im Anschluß nochmals in Verbindung mit MINOCS, einem von M. Glocker in [15] entwickelten Verfahren zur Lösung kontinuierlicher und gemischt - ganzzahliger Optimalsteuerungsprobleme, gezeigt. Berechnet werden dabei verschiedene, optimale Wege des in 4.1 vorgestellten, differentialgesteuerten Fahrzeugs. Durch den Einsatz von SICOM wurde hier, als zusätzliche Anwendungsmöglichkeit, auch die Formulierung des entsprechenden Optimalsteuerungsproblems teilweise deutlich vereinfacht.

7.1 SICOM und Newton

Die für die folgenden Beispiele verwendete Implementierung des Newton - Verfahrens entspricht der in 3.1 beschriebenen Vorgehensweise. Auf zusätzliche Erweiterungen des Newton - Verfahrens zur Globalisierung oder Beschleunigung der Konvergenz wurde dabei bewußt verzichtet.
Da in den Beispielen für jeden Aufruf des Newton - Verfahrens nur wenige Iterationen zur Bestimmung der Lösung benötigt werden, wurde das Newton - Verfahren aber zur Verbesserung der grafischen Darstellung der Rechenzeit in jeder Iteration durch busy - waiting um einen konstanten Faktor verlangsamt.

7.1.1 Verlassen des Definitionsbereichs

Gegeben sei das, aus 3.2.1 bekannte, zu lösende Problem

$$f(x) \stackrel{!}{=} 0 \text{ mit}$$
$$f : \Re^{+} \to \Re, \quad f(x) = ln(x).$$

Als Startwert für das Newton - Verfahren wird $x_0 = 4.0$ gewählt.

Wie schon in Abbildung 3-2 gezeigt, verläßt das Newton - Verfahren bereits in der ersten Iteration den Definitionsbereich von f.

Das Problem soll nun mit Hilfe von SICOM gelöst werden. Die verwendete Homotopie ist gegeben durch

$$h : \Re^+ \times [0,1] \to \Re,$$
$$h(x,p) = p \cdot f(x) + (1-p) \cdot g(x).$$

Als Startproblem wird

$$g(x) \overset{!}{=} 0 \text{ mit}$$
$$g : \Re^+ \to \Re, \quad g(x) = x - 4.0$$

verwendet. Der Startwert x_0 des Newton - Verfahrens bleibt für das Homtopieverfahren unverändert, die Lösung des Startproblems entspricht also x_0, so daß die Abbruchbedingung des Newton - Verfahrens für das Startproblem in der ersten Iteration erfüllt ist.

Während des Homotopieverfahrens wird der durch SICOM gesteuerte Homotopieparameter p ausgehend von dem Startwert $p_{start} = 0$ bis zu seinem Zielwert $p_{ziel} = 1$ vergrößert, wodurch das Startproblem g in das Zielproblem f deformiert wird. Als Startnäherung für die, durch das Newton - Verfahren zu lösenden Zwischenprobleme wird dabei jeweils die Lösung des zuletzt erfolgreich berechneten Zwischenproblems genutzt.

Da zur Lösung des Zielproblems nur wenige Homotopieschritte benötigt werden, entstehen durch die Verwendung verschiedener Varianten zur Schrittweitensteuerung nur minimale Unterschiede im Verlauf des Homotopieparameters p. Gezeigt wird deshalb nur die Schrittweitensteuerung durch Intervallbisektion. Allerdings kann durch die Wahl verschiedener, initialer Schrittweiten s_{start} des Parameters dessen Verlauf und damit die Anzahl der zur Lösung des Zielproblems benötigten Homotopieschritte durchaus günstig beeinflußt werden.

7.1.1.1 Intervallbisektion, $s_{start} = 1$

Aus der Verwendung der Anfangsschrittweite $s_{start} = 1$ folgt, daß nach der Lösung des Startproblems der Homotopieparameter p sofort den Zielwert $p_{ziel} = 1$ erreicht. Das Startproblem $g(x) \overset{!}{=} 0$ wird also ohne weitere Zwischenschritte in das Zielproblem $f(x) \overset{!}{=} 0$ deformiert. Als Startnäherung für das Zielproblem wird dadurch die Lösung $x(p_{start}) = 4.0$ des Startproblems verwendet, wodurch wiederum in der ersten Newton - Iteration der Definitionsbereich von f verlassen wird. p muß also im nächsten Homotopieschritt wieder verkleinert werden, um ein mit der gegebenen Startnäherung $x(p_{start}) = 4.0$ lösbares Zwischenproblem zu erzeugen.

Die Abbildungen 7-1 und 7-2 zeigen den entstehenden Verlauf des Homotopieparameters und die entsprechenden Schrittweiten. Die bei jedem Homotopieschritt durch das Newton - Verfahren benötigten Iterationen und die daraus resultierende Rechenzeit werden in den Abbildungen 7-3 und 7-4 dargestellt. Die entsprechenden Werte für Rückwärtsschritte, also für Aufrufe des Newton - Verfahrens bei denen keine Lösung berechnet werden konnte, werden, zur Verbesserung der Übersicht, im negativen Bereich notiert. Deutlich wird auch, daß das Newton - Verfahren in diesem Fall bis zum Erreichen seines Iterationslimits (hier 20 Iterationen) arbeitet, bevor der Lösungsversuch abgebrochen wird.

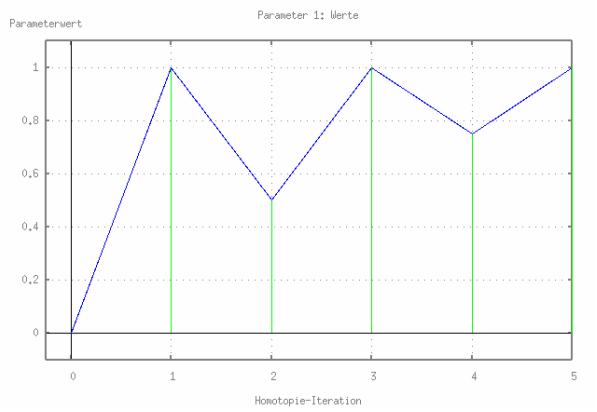

Abbildung 7-1: Verlauf des Homotopieparameters

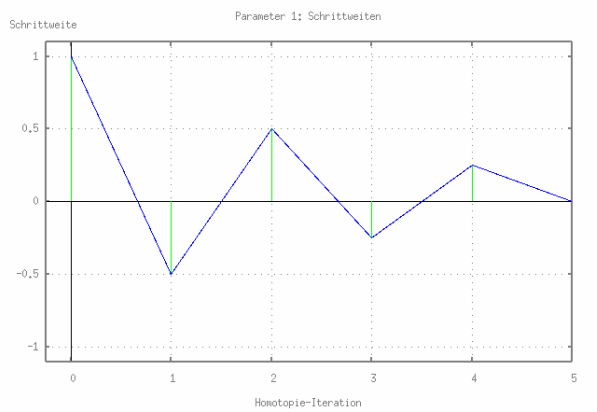

Abbildung 7-2: Schrittweiten

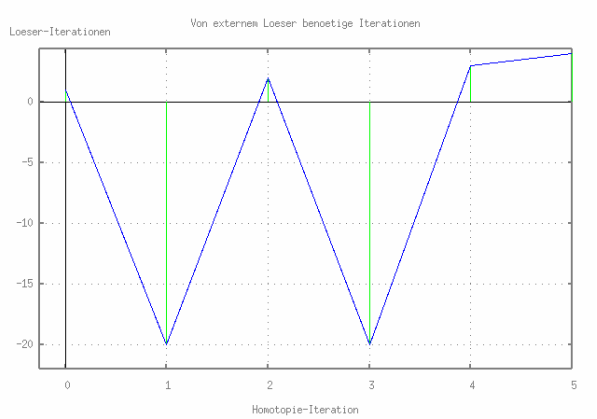

Abbildung 7-3: Iterationen im Newton - Verfahren

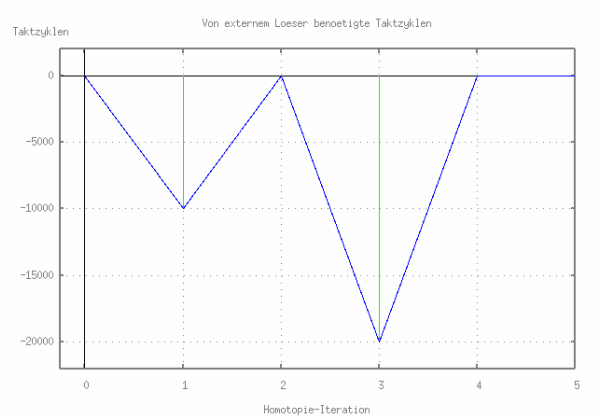

Abbildung 7-4: Rechenzeit des Newton - Verfahrens

Der Verlauf des Parameters erzeugt die in den Abbildungen 7-5 bis 7-8 dargestellten Zwischenprobleme mit den entsprechenden Iterationen des Newton - Verfahrens. Gezeigt werden dabei nur die Zwischenprobleme für die eine Lösung berechnet werden konnte. Abbildung 7-9 zeigt die sich ergebende Prädiktor - Korrektor Pfadverfolgung der Lösungskurve $x(p)$.

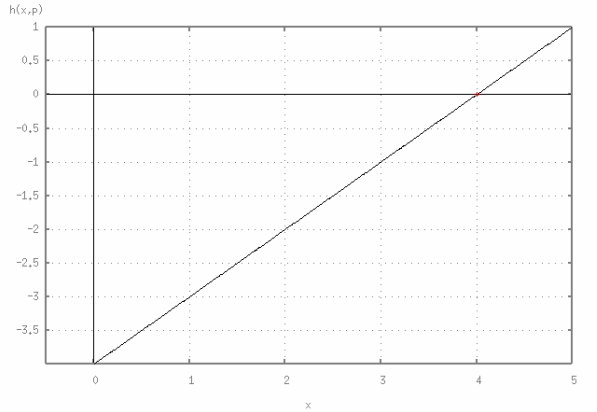

Abbildung 7-5: Startproblem, $p = 0$

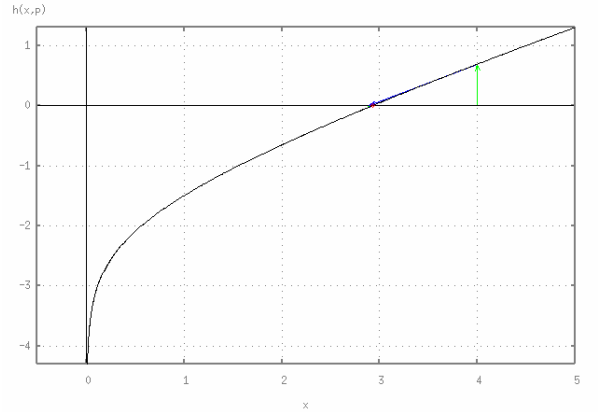

Abbildung 7-6: $p = 0.5$

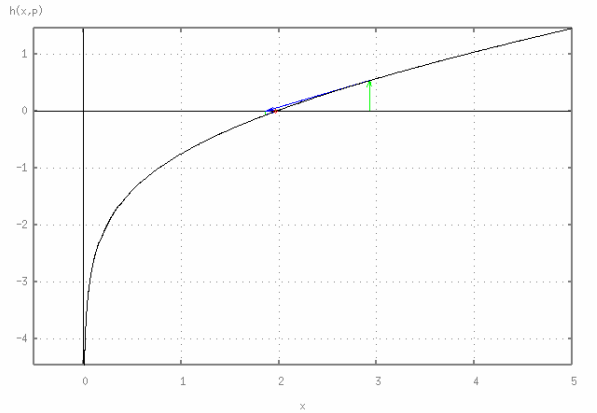

Abbildung 7-7: $p = 0.75$

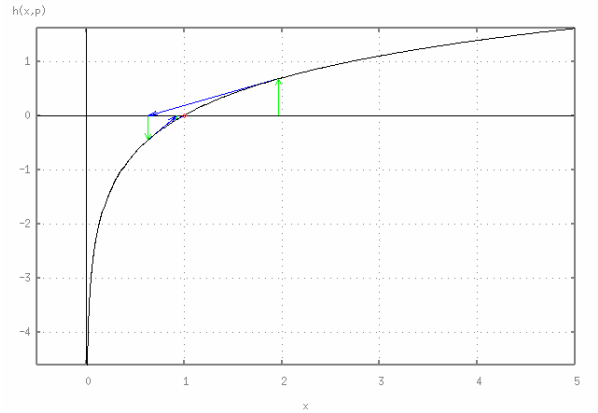

Abbildung 7-8: Zielproblem, $p=1$

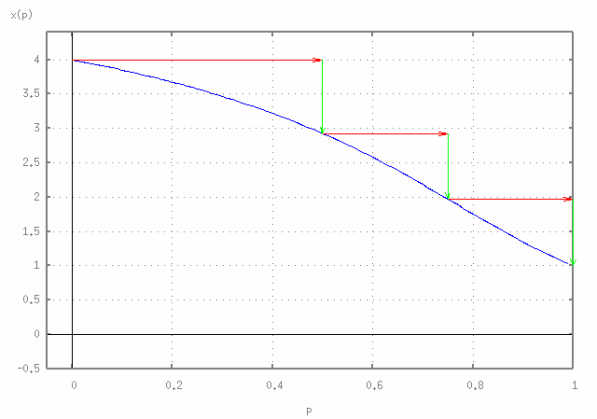

Abbildung 7-9: Verfolgung der Lösungskurve

7.1.1.2 Intervallbisektion, $s_{start} = 0.25$

Die Reduktion der Anfangsschrittweite auf $s_{start} = 0.25$ bewirkt, daß während des Homotopieverfahrens ausschließlich lösbare Zwischenprobleme entstehen. Die Ausführung von Rückwärtsschritten ist in diesem Fall also nicht notwendig, wodurch die Anzahl der zur Lösung des Zielproblems benötigten Homotopieschritte von 5 auf 3 reduziert wird.
Der resultierende Verlauf der Parameterwerte und die zugehörigen Schrittweiten, sowie die im Newton - Verfahren benötigten Iterationen und die entsprechend benötigte Rechenzeit werden in den Abbildungen 7-10 und 7-13 gezeigt.

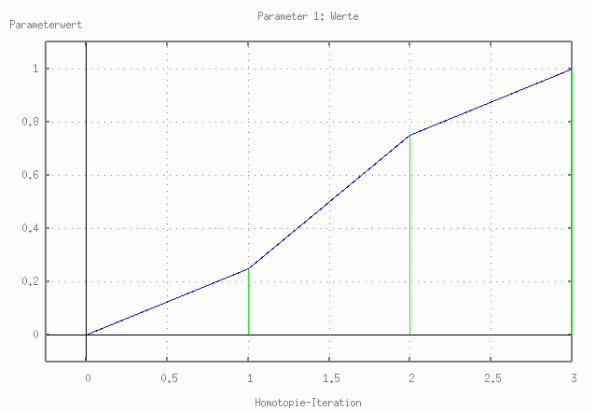

Abbildung 7-10: Verlauf des Homotopieparameters

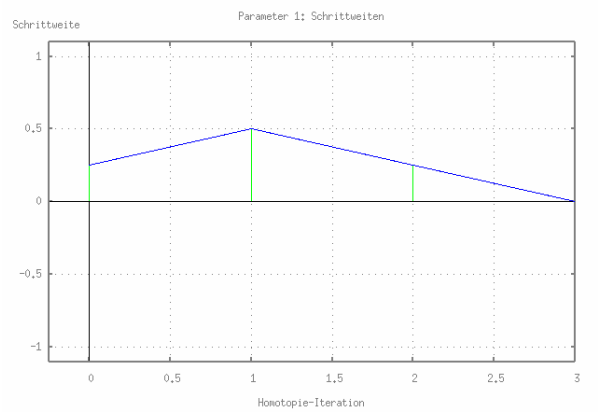

Abbildung 7-11: Schrittweiten

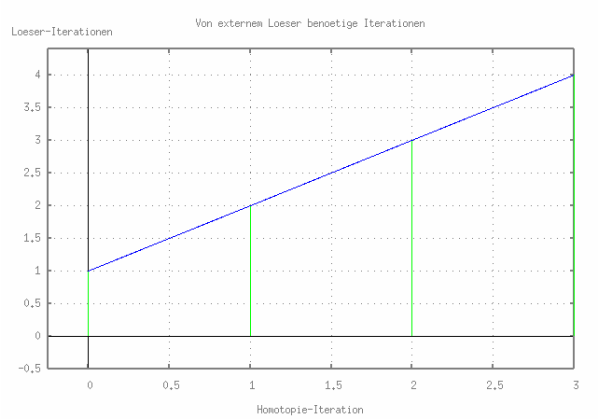

Abbildung 7-12: Iterationen im Newton - Verfahren

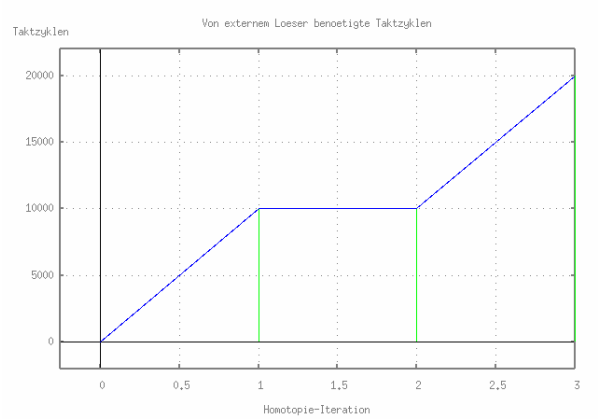

Abbildung 7-13: Rechenzeit des Newton - Verfahrens

Durch den Verlauf des Homotopieparameters ergeben sich die folgenden Zwischenprobleme und die zugehörige Prädiktor - Korrektor Verfolgung der Lösungskurve $x(p)$.

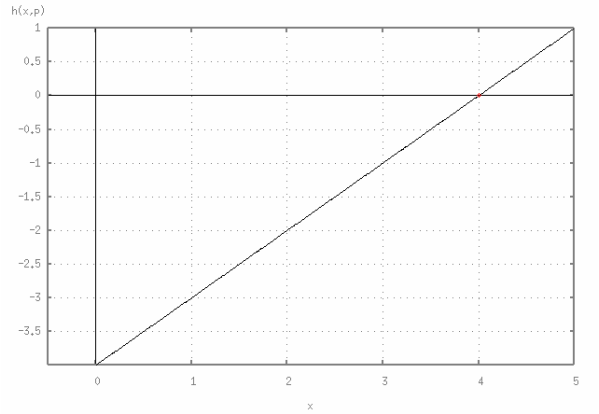

Abbildung 7-14: Startproblem, $p=0$

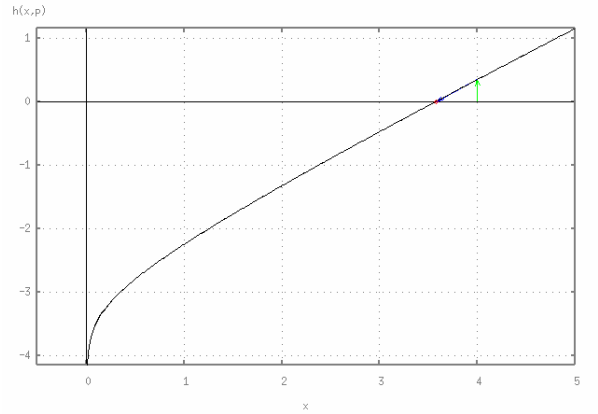

Abbildung 7-15: $p=0.25$

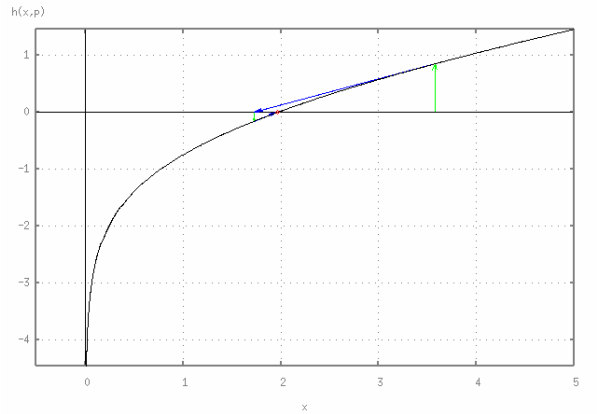

Abbildung 7-16: $p=0.75$

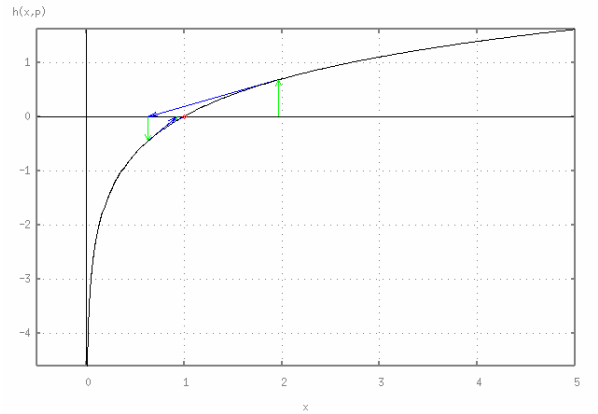

Abbildung 7-17: $p=1$

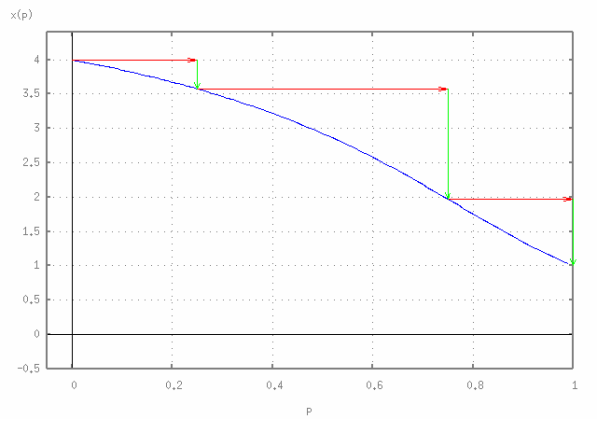

Abbildung 7-18: Verfolgung der Lösungskurve

7.1.2 Endlosschleifen

Das zu lösende Zielproblem sei, wie in 3.2.2 gegeben durch

$$f(x) \overset{!}{=} 0 \text{ mit}$$

$$f : \Re \to \Re \;,\quad f(x) = \begin{cases} e^{x-1.5} - 1 & \text{für } x < 1.5 \\ 1 - e^{-(x-1.5)} & \text{für } x \geq 1.5 \end{cases}.$$

In Verbindung mit dem Startwert $x_0 = 3.0$ für das Newton - Verfahren entsteht dabei die in Abbildung 3-3 gezeigte Endlosschleife, die hier allerdings durch ein Limit von maximal 20 Newton - Iterationen unterbrochen wird.

Für das Homotopieverfahren wird die Einbettung

$$h : \Re \times [0,1] \to \Re \text{ mit}$$
$$h(x,p) = p \cdot f(x) + (1-p) \cdot g(x)$$

verwendet. Das Startproblem sei

$$g(x) \overset{!}{=} 0 \text{ mit}$$
$$g : \Re \to \Re,\quad g(x) = x - 3.0\,.$$

Zur Deformation des Startproblems in das Zielproblem gilt für den Homotopieparameter p wieder $p_{start} = 0$ und $p_{ziel} = 1$.

Wie auch in 7.1.1 ist die Anzahl der zur Lösung des Zielproblems benötigten Homotopieschritte zu gering, um den Unterschied verschiedener Varianten der Schrittweitensteuerung zu demonstrieren. Verwendet wird wieder die Intervallbisektion mit zwei unterschiedlichen Anfangsschrittweiten s_{start} des Homotopieparameters p.

7.1.2.1 Intervallbisektion, $s_{start} = 1$

Die initiale Schrittweite $s_{start} = 1$ hat den schon in 7.1.1.1 beschriebenen Effekt. Wieder ist also die Ausführung von Rückwärtsschritten zur Erzeugung lösbarer Zwischenprobleme nötig. Der dabei entstehende Parameterverlauf und die entsprechenden Schrittweiten, sowie die Anzahl der Iterationen im Newton - Verfahren und die resultierende Rechenzeit werden in den folgenden Abbildungen dargestellt.

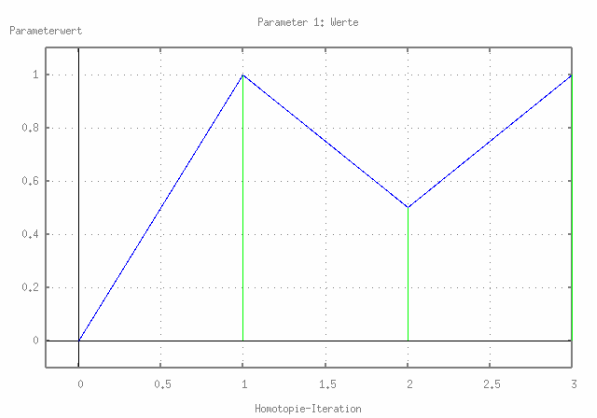

Abbildung 7-19: Verlauf des Homotopieparameters

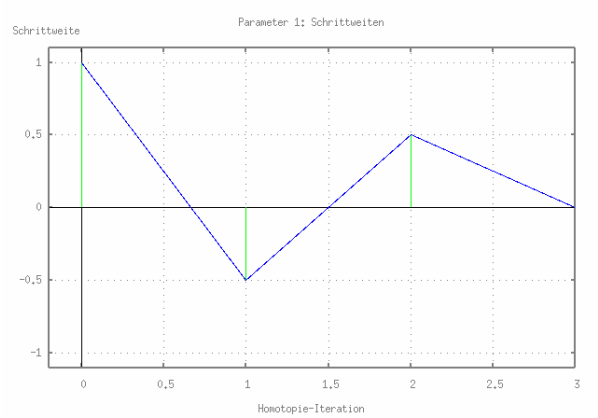

Abbildung 7-20: Schrittweiten

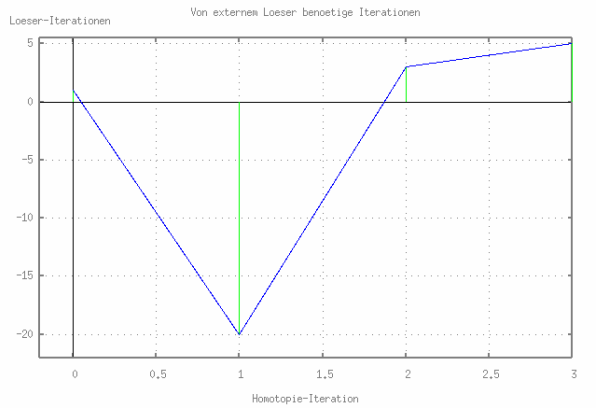

Abbildung 7-21: Iterationen im Newton - Verfahren

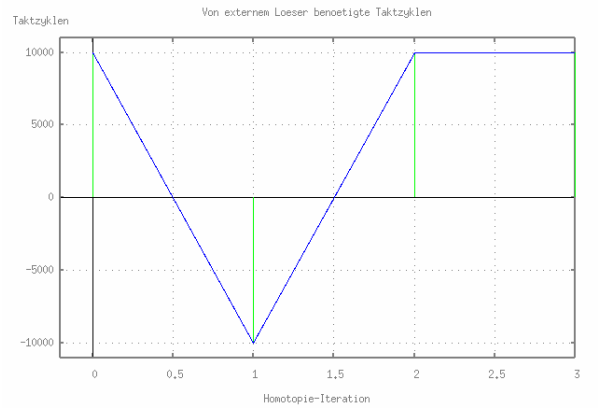

Abbildung 7-22: Rechenzeit des Newton - Verfahrens

Durch den Verlauf des Homotopieparameters p ergeben sich die nachfolgenden Zwischenprobleme und die Prädiktor - Korrektor Verfolgung der Lösungskurve $x(p)$. Dargestellt werden wieder nur lösbare Zwischenprobleme.

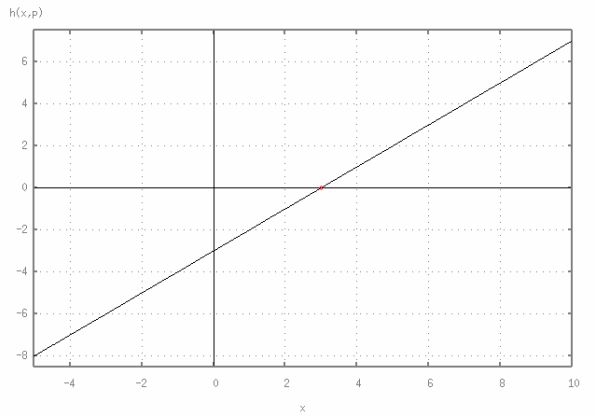

Abbildung 7-23: Startproblem, $p=0$

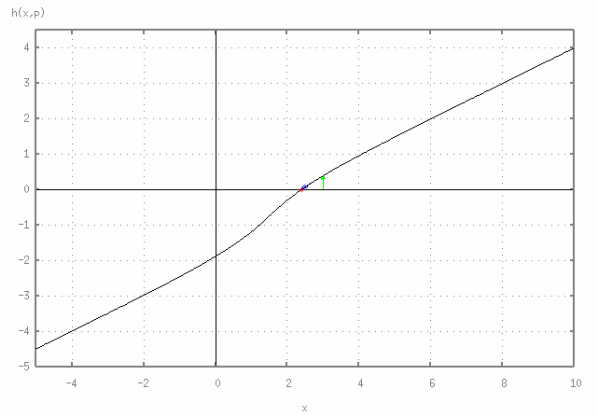

Abbildung 7-24: $p=0.5$

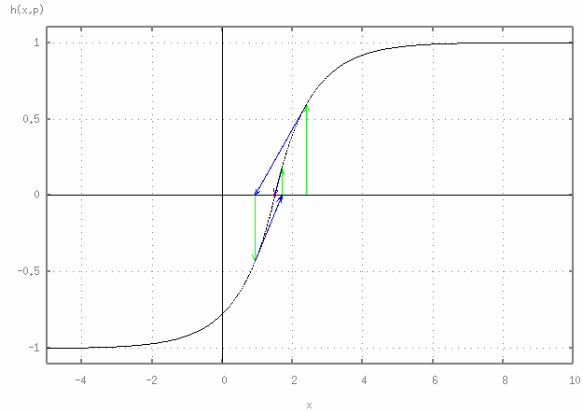

Abbildung 7-25: Zielproblem, $p=1$

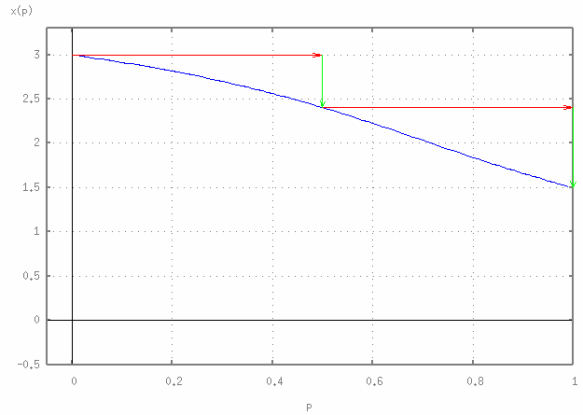

Abbildung 7-26: Verfolgung der Lösungskurve

7.1.2.2 Intervallbisektion, $s_{start} = 0.5$

Analog zu 7.1.1.2 führt eine verminderte Anfangsschrittweite des Homotopieparameters dazu, daß alle, im Verlauf des Homotopieverfahrens entstehenden Zwischenprobleme lösbar sind, also keine Rückwärtsschritte ausgeführt werden müssen. Für $s_{start} = 0.5$ entsprechen die entstehenden, lösbaren Zwischenprobleme denen aus 7.1.2.1. Nachfolgend sollen deshalb nur der veränderte Verlauf der Parameterwerte und die Schrittweiten, sowie die im Newton - Verfahren benötigten Iterationen und der daraus resultierende Rechenaufwand gezeigt werden.

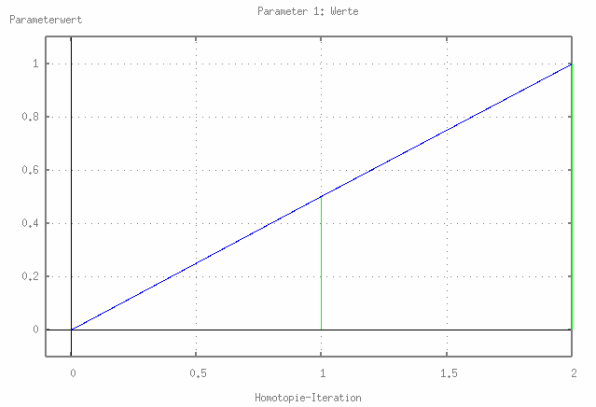

Abbildung 7-27: Verlauf des Homotopieparameters

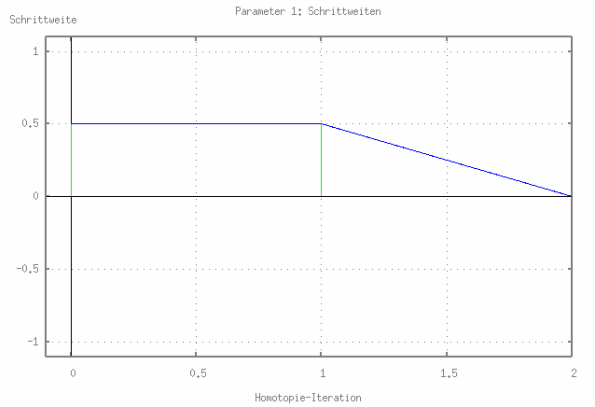

Abbildung 7-28: Schrittweiten

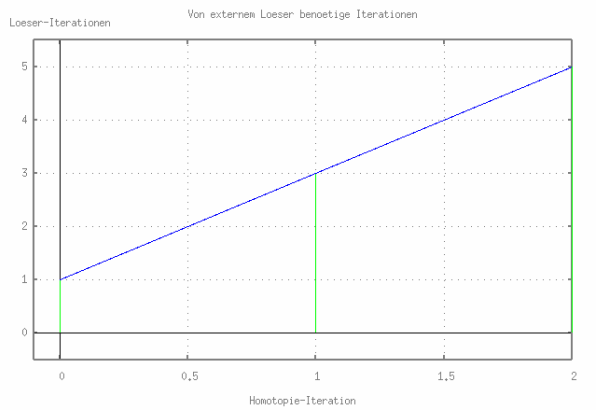

Abbildung 7-29: Iterationen im Newton - Verfahren

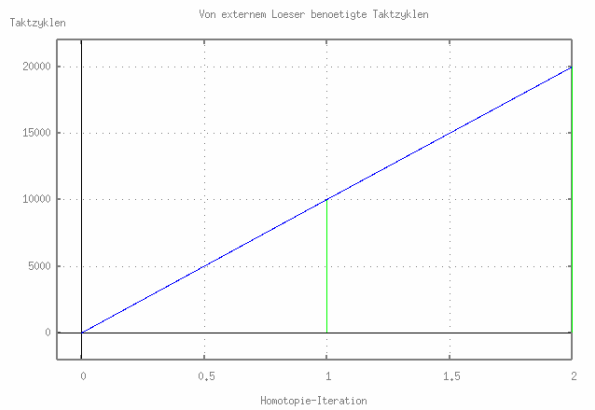

Abbildung 7-30: Rechenzeit im Newton - Verfahren

7.1.3 “Zufällige Konvergenz“

Anhand des folgenden Zielproblems aus 3.2.3 soll die Verwendung von SICOM zur Beschleunigung des Konvergenzverhaltens des Newton - Verfahrens gezeigt werden.

Das Zielproblem ist gegeben durch

$$f(x) \stackrel{!}{=} 0 \text{ mit}$$
$$f : \Re \to \Re \;,\quad f(x) = 2 \cdot x^3 + 17 \cdot x^2 + 6 \cdot x + 50 .$$

Zwar kann die Lösung des Zielproblems ausgehend von dem Startwert $x_0 = -5.0$ gefunden werden, allerdings sind dafür, wie in Abbildung 3-4 dargestellt, 183 Iterationen des Newton - Verfahrens nötig.

In Verbindung mit der Homotopie

$$h : \Re \times [0,1] \to \Re \;,\quad h(x,p) = p \cdot f(x) + (1-p) \cdot g(x) \text{ mit}$$
$$p_{start} = 0 \;,\quad p_{ziel} = 1$$

und dem Startproblem

$$g(x) \stackrel{!}{=} 0 \text{ mit}$$
$$g : \Re \to \Re \;,\quad g(x) = x + 5.0 ,$$

soll dieses “Konvergenzverhalten“ nun beschleunigt werden.
Zur Vermeidung der “zufälligen Konvergenz“ auch in Zwischenproblemen, wird das Newton - Verfahren dafür um eine zusätzliche Abbruchbedingung erweitert.
Da das Newton - Verfahren für hinreichend genaue Startnäherungen quadratisch konvergiert, sollte also zumindest ein lineares Konvergenzverhalten gegeben sein. Bezeichnen $x_i^{Newton}(p)$, $i \in [0,f]$ die während des Newton - Verfahrens berechneten Schnittpunkte der Tangenten mit der x-Achse, wobei $x_0^{Newton}(p)$ der Startwert und $x_f^{Newton}(p)$ die Lösung des durch den Homotopieparameter p erzeugten Zwischenproblems ist, soll dementsprechend die Bedingung

$$\left| x_{i+1}^{Newton}(p) \right| < \left| x_i^{Newton}(p) \right|$$

erfüllt sein. Ist dies nicht der Fall, wird die Berechnung des aktuellen Zwischenproblems abgebrochen.

Nachfolgend sollen die, durch verschiedene Varianten der Schrittweitensteuerung entstehenden Parameterverläufe, die entsprechende Prädiktor - Korrektor Verfolgung der Lösungskurve und die entstehenden Iterationen im Newton - Verfahren verglichen werden. Auf die Darstellung der jeweils entstehenden Zwischenprobleme wird dabei verzichtet. Exemplarisch werden deshalb vorab in den folgenden Abbildungen 7-31 bis 7-34 das Start- und Zielproblem sowie zwei Zwischenprobleme gezeigt.
Die Lösungskurve $x(p)$ verläuft, wie im weiteren Verlauf noch dargestellt wird, im Anfangsbereich sehr steil, bleibt dann aber bis zum Zielwert des Homotopieparameters p fast konstant. Der größte Teil der Deformation des Startproblems in das Zielproblem findet also in einem kleinen Bereich von p nahe dem Startwert $p_{start} = 0$ statt.

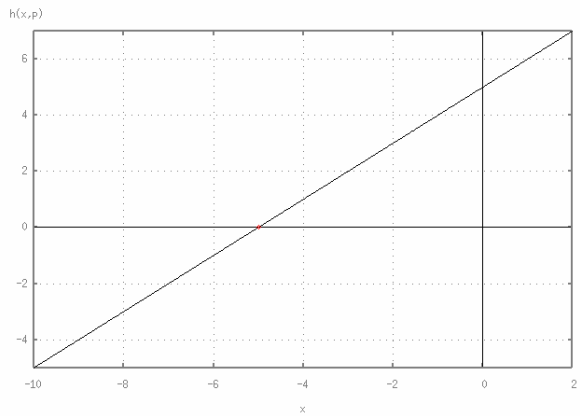

Abbildung 7-31: Startproblem, $p=0$

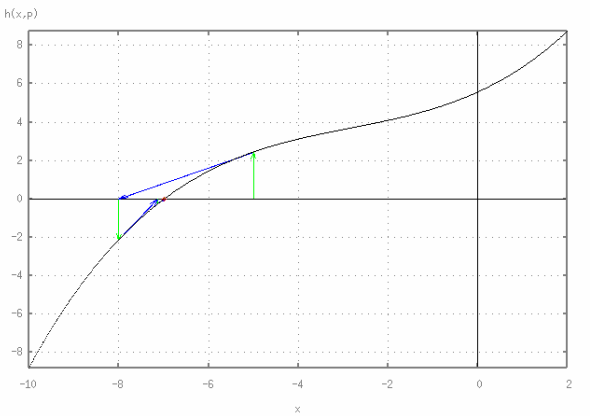

Abbildung 7-32: $p=0.0125$

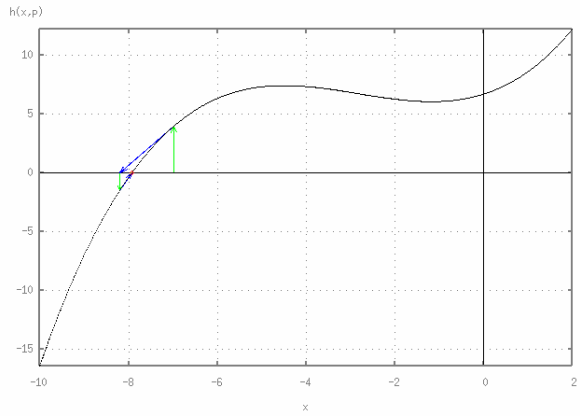

Abbildung 7-33: $p=0.0375$

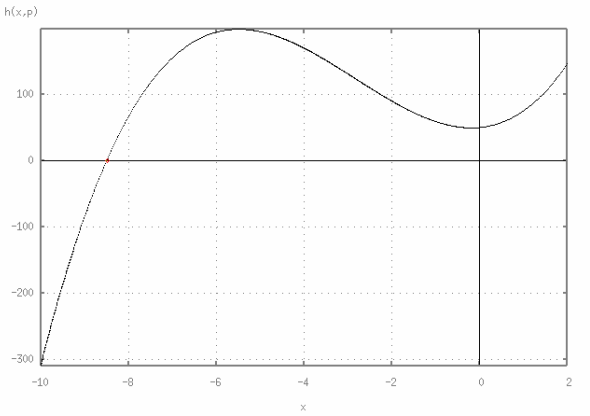

Abbildung 7-34: Zielproblem, $p=1$

7.1.3.1 Intervallbisektion, $s_{start} = 1$

Die Verwendung der Intervallbisektion mit einer Anfangsschrittweite $s_{start} = 1$ führt zu folgender Statistik. Der Beschleunigungsfaktor ergibt sich dabei aus dem Verhältnis der ohne SICOM zur Lösung des Zielproblems benötigten 183 Newton - Iterationen zur Gesamtanzahl der bei Verwendung von SICOM entstehenden Iterationen des Newton - Verfahrens.

Benötigte Homotopieschritte	:	16
Rückwärtsschritte	:	7
Newton - Iterationen (gesamt)	:	28
Beschleunigungsfaktor	:	6.5

Abbildung 7-37 zeigt den Effekt des oben beschriebenen zusätzlichen Abbruchkriteriums im Newton - Verfahren bzgl. linearer Konvergenz. Da die entsprechende Abbruchbedingung hier bereits nach der jeweils ersten Newton - Iteration erfüllt ist, kann die Anzahl der insgesamt im Newton - Verfahren benötigten Iterationen von 183 auf 28 reduziert werden.

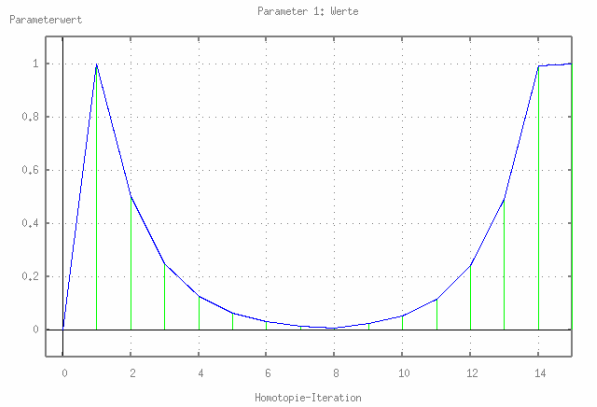

Abbildung 7-35: Verlauf des Homotopieparameters

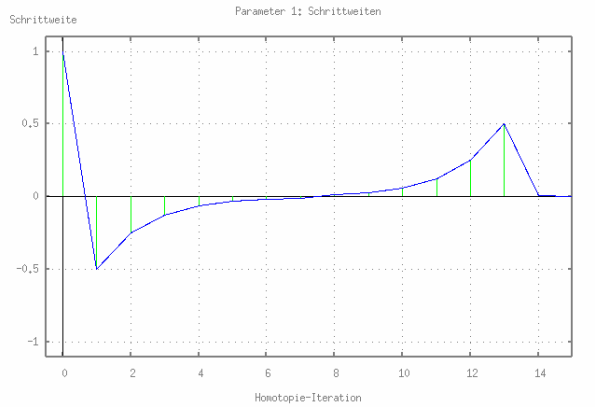

Abbildung 7-36: Schrittweiten

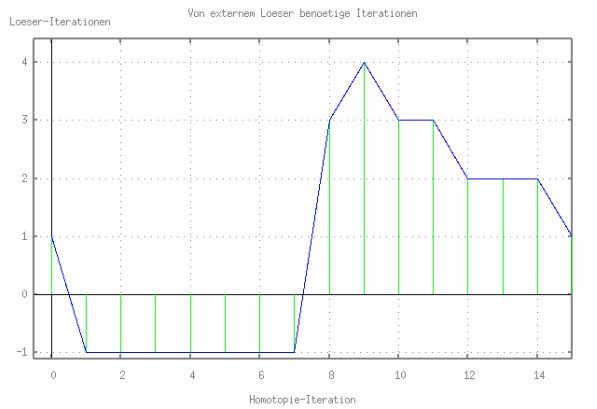

Abbildung 7-37: Iterationen im Newton - Verfahren

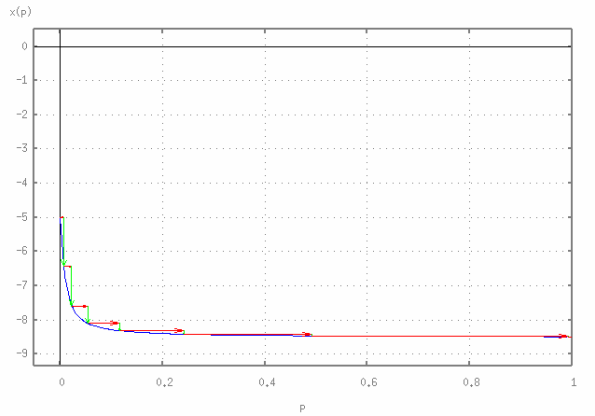

Abbildung 7-38: Verfolgung der Lösungskurve

7.1.3.2 Intervallbisektion, $s_{start} = 0.1$

Die Intervallbisektion mit verkleinerter Anfangsschrittweite ergibt folgende Werte:

Benötigte Homotopieschritte	:	11
Rückwärtsschritte	:	3
Newton - Iterationen (gesamt)	:	24
Beschleunigungsfaktor	:	7.6

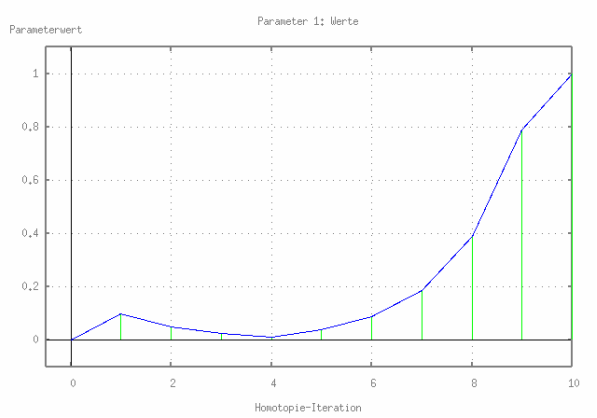

Abbildung 7-39: Verlauf des Homotopieparameters

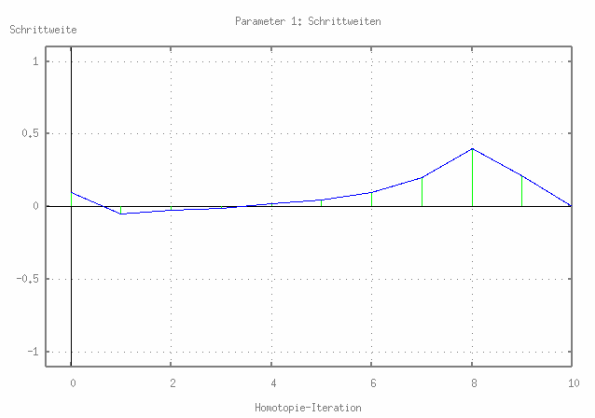

Abbildung 7-40: Schrittweiten

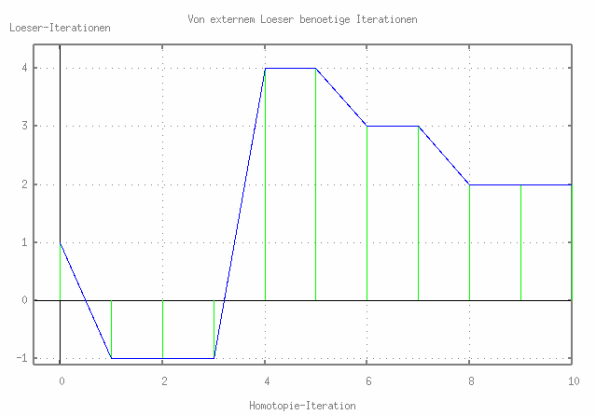

Abbildung 7-41: Iterationen im Newton - Verfahren

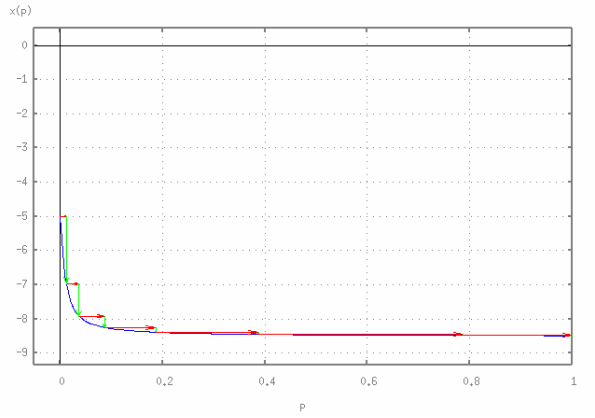

Abbildung 7-42: Verfolgung der Lösungskurve

7.1.3.3 Optimistische Schrittweitensteuerung, $s_{start} = 1$

Bei Verwendung der optimistischen Schrittweitensteuerung mit einer Anfangsschrittweite $s_{start} = 1$ ergibt sich für die Rückwärtsschritte das bereits in 6.3.4 beschriebene Problem. Da der Parameterwert wieder bis nahe an den Startwert verkleinert werden muß, bevor ein lösbares Zwischenproblem entsteht, die optimistischen Rückwärtsschritte aber, wie in Abbildung 7-44 zu sehen, zunehmend kleiner werden, folgt ein starker Anstieg der Anzahl benötigter Rückwärtsschritte.
Die Anzahl benötigter Vorwärtsschritte kann allerdings im Vergleich zur Intervallbisektion reduziert werden.

Benötigte Homotopieschritte	:	83
Rückwärtsschritte	:	76
Newton - Iterationen (gesamt)	:	95
Beschleunigungsfaktor	:	1.9

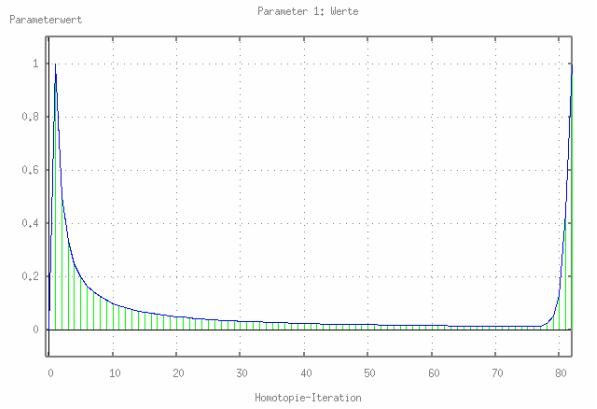

Abbildung 7-43: Verlauf des Homotopieparameters

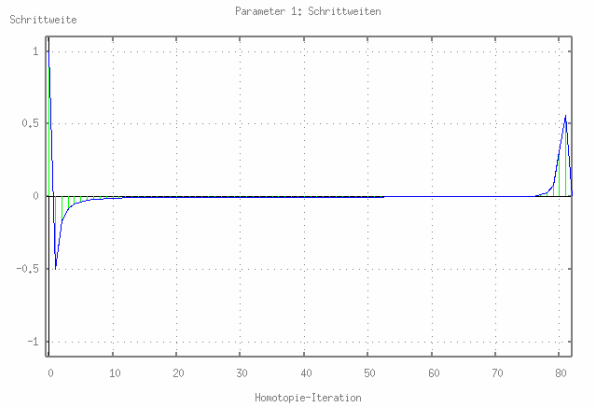

Abbildung 7-44: Schrittweiten

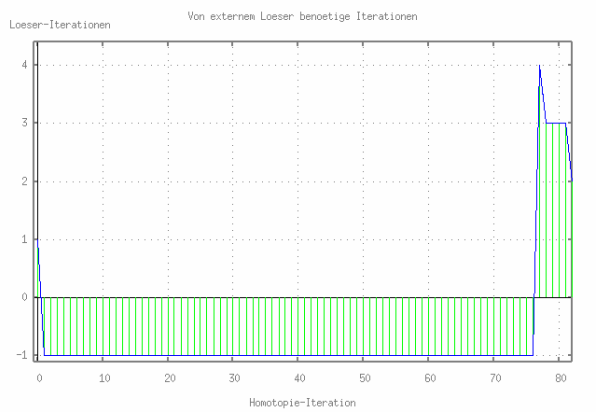

Abbildung 7-45: Iterationen im Newton - Verfahren

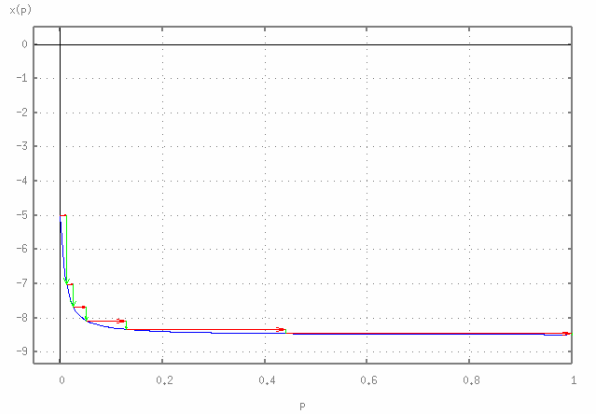

Abbildung 7-46: Verfolgung der Lösungskurve

7.1.3.4 Optimistische Schrittweitensteuerung, $s_{start} = 0.1$

Wird die Anfangsschrittweite für die optimistische Schrittweitensteuerung auf $s_{start} = 0.1$ verkleinert, sind nur noch kurze Rückwärtsschritte auszuführen, womit das Verhalten im Vergleich zu 7.1.3.3 deutlich verbessert werden kann.

Benötigte Homotopieschritte	:	14
Rückwärtsschritte	:	7
Newton - Iterationen (gesamt)	:	26
Beschleunigungsfaktor	:	7.0

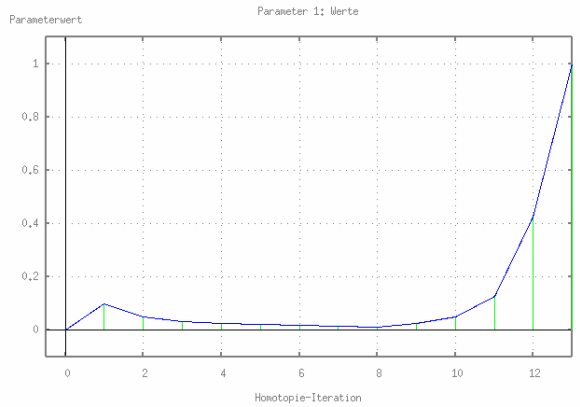

Abbildung 7-47: Verlauf des Homotopieparameters

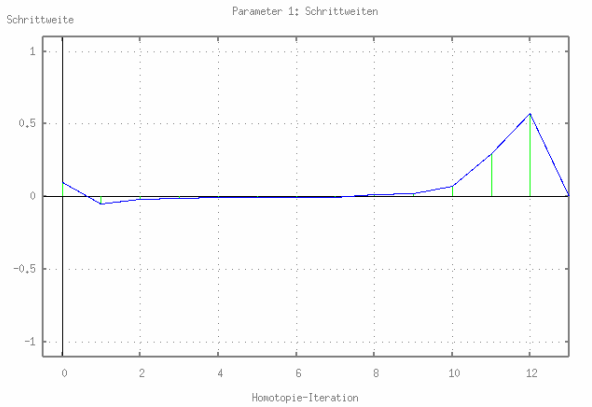

Abbildung 7-48: Schrittweiten

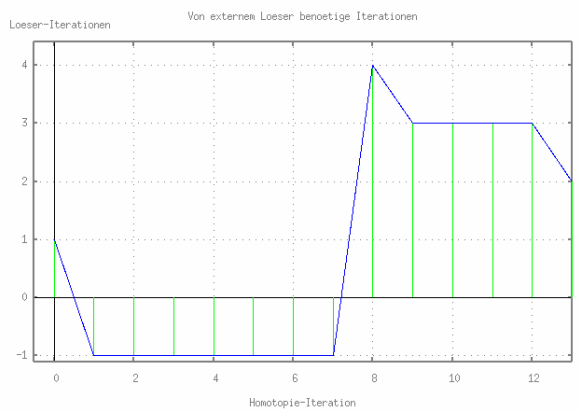

Abbildung 7-49: Iterationen im Newton - Verfahren

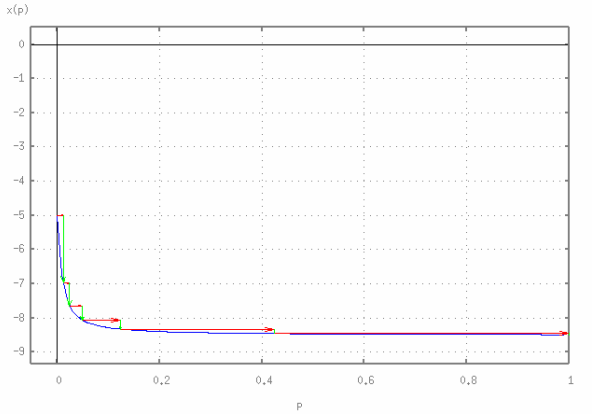

Abbildung 7-50: Verfolgung der Lösungskurve

7.1.3.5 Pessimistische Schrittweitensteuerung, $s_{start} = 1$

Die pessimistische Schrittweitensteuerung ist, aufgrund des in 6.3.4 beschriebenen Effekts, vor allem bei der Steuerung der Vorwärtsschrittweite wegen des zum Großteil flachen Verlaufs der Lösungskurve $x(p)$, zur Lösung dieses Problems eher ungeeignet. Die Anzahl der benötigten Vorwärtsschritte steigt, wie in den Abbildungen 7-51 und 7-52 gezeigt, aufgrund der ständig kürzer werdenden Schrittweite, stark an.
Zum Vergleich aller Varianten der Schrittweitensteuerung soll das Ergebnis hier aber trotzdem aufgeführt werden.

Benötigte Homotopieschritte	:	120
Rückwärtsschritte	:	4
Newton - Iterationen (gesamt)	:	149
Beschleunigungsfaktor	:	1.2

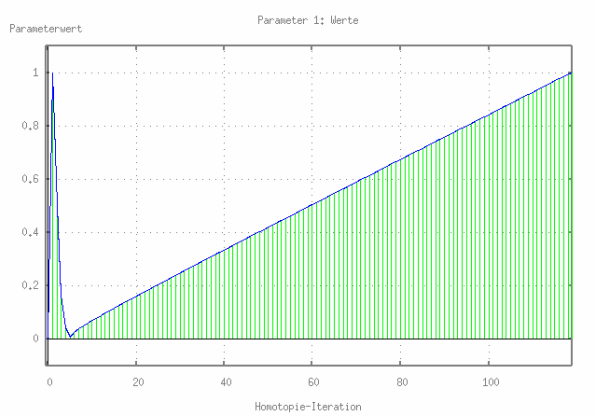

Abbildung 7-51: Verlauf des Homotopieparameters

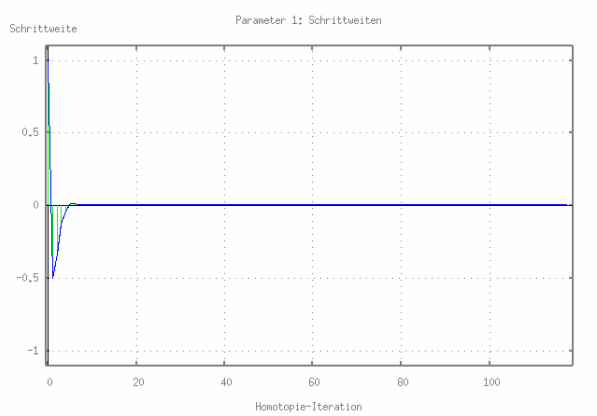

Abbildung 7-52: Schrittweiten

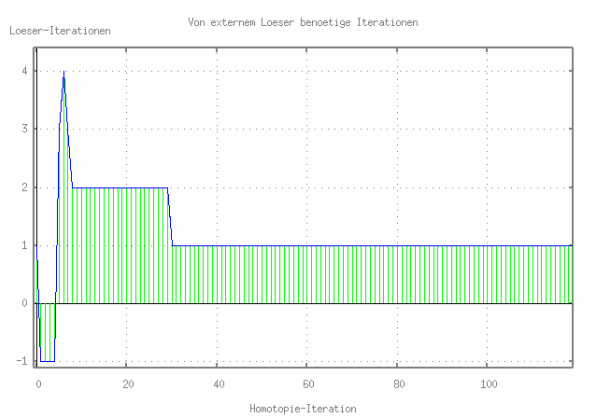

Abbildung 7-53: Iterationen im Newton - Verfahren

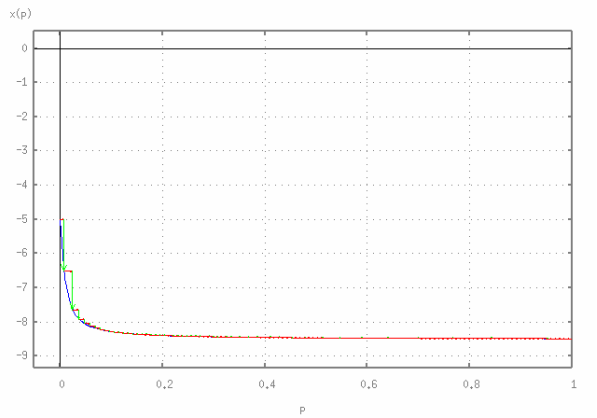

Abbildung 7-54: Verfolgung der Lösungskurve

7.1.3.6 Optimistische und pessimistische Schrittweitensteuerung

Durch die Kombination der optimistischen Schrittweitensteuerung für Vorwärtsschritte mit der pessimistischen Schrittweitensteuerung für Rückwärtsschritte können die beschriebenen Nachteile der beiden Varianten ausgeglichen werden. Daraus folgt auch für die gewählte Anfangsschrittweite $s_{start} = 1$ das, im Vergleich zu den bisher vorgestellten Varianten beste Resultat.

Benötigte Homotopieschritte	:	11
Rückwärtsschritte	:	4
Newton - Iterationen (gesamt)	:	21
Beschleunigungsfaktor	:	8.7

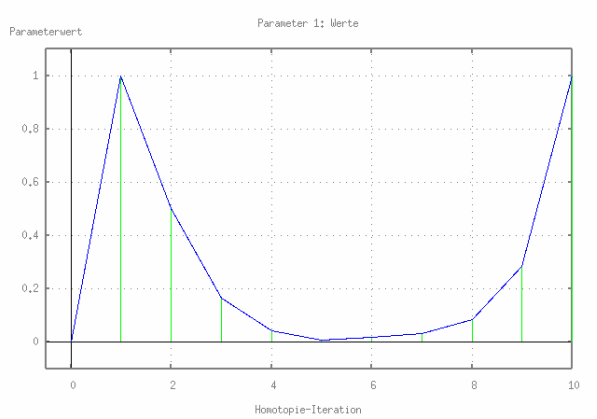

Abbildung 7-55: Verlauf des Homotopieparameters

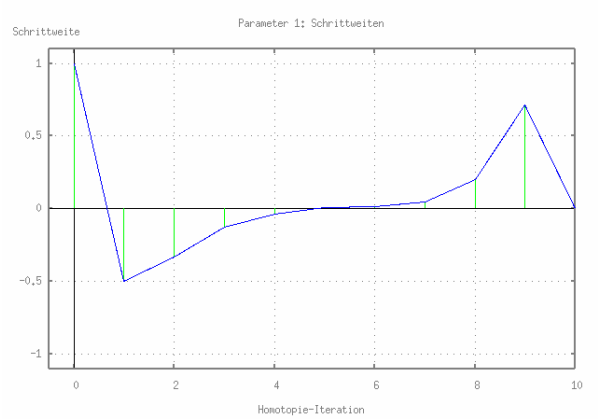

Abbildung 7-56: Schrittweiten

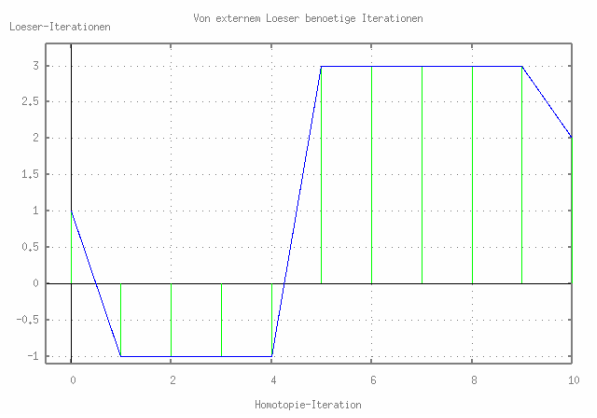

Abbildung 7-57: Iterationen im Newton - Verfahren

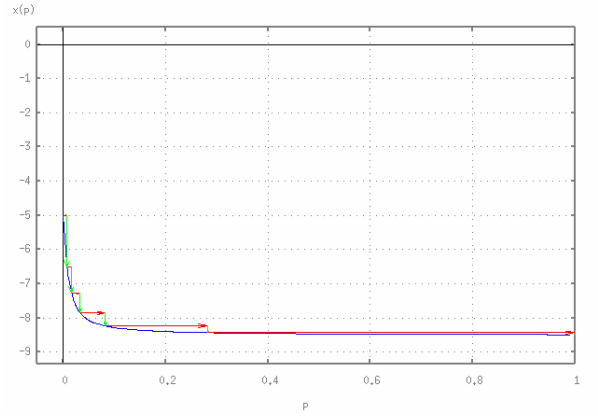

Abbildung 7-58: Verfolgung der Lösungskurve

7.1.3.7 Iterations - adaptive Schrittweitensteuerung, $s_{start} = 1$

Die Iterations - adaptive Schrittweitensteuerung liefert mit der Anfangsschrittweite $s_{start} = 1$ ein Ergebnis daß mit dem der optimistischen Schrittweitensteuerung mit der angepaßten, initialen Schrittweite $s_{start} = 0.1$ vergleichbar ist. Durch die dynamische Steuerung der Schrittweite kann also, ohne Kenntnis des Verlaufs der Lösungskurve $x(p)$ und einer entsprechend angepaßten Anfangsschrittweite, ein gutes Resultat erzielt werden.

Benötigte Homotopieschritte	:	13
Rückwärtsschritte	:	5
Newton - Iterationen (gesamt)	:	26
Beschleunigungsfaktor	:	7.0

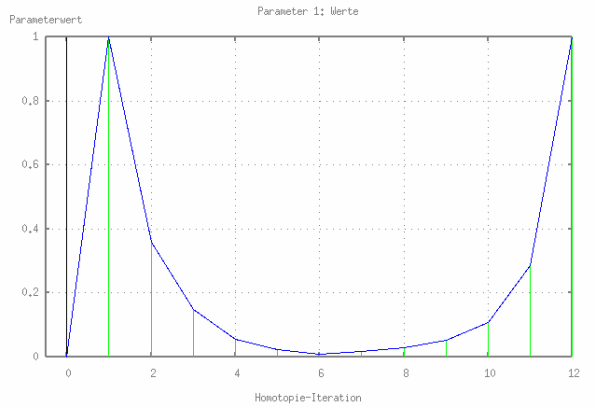

Abbildung 7-59: Verlauf des Homotopieparameters

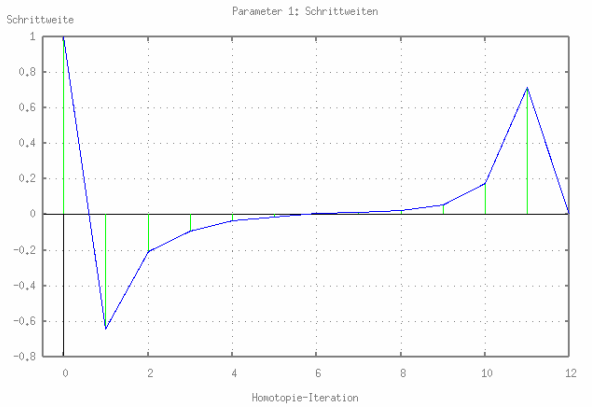

Abbildung 7-60: Schrittweiten

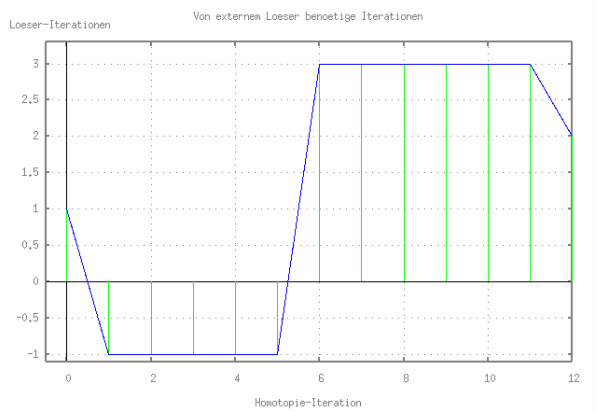

Abbildung 7-61: Iterationen im Newton - Verfahren

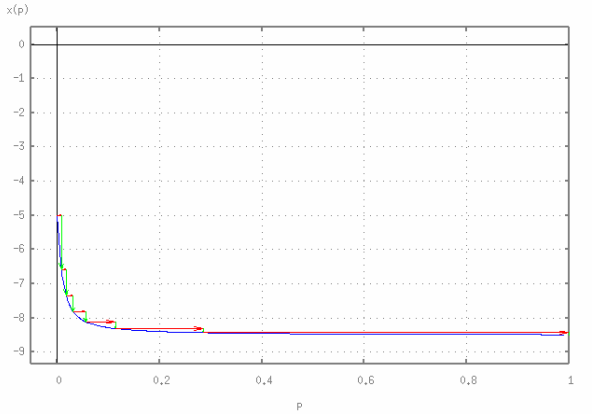

Abbildung 7-62: Verfolgung der Lösungskurve

7.1.3.7 Iterations - adaptive Schrittweitensteuerung, $s_{start} = 0.1$

Durch die reduzierte Anfangsschrittweite $s_{start} = 0.1$ kann das Ergebnis der Iterations - adaptiven Schrittweitensteuerung weiter verbessert werden. Die Anzahl der benötigten Homotopieschritte erreicht so bzgl. dieses Zielproblems das Minimum aller gezeigten Varianten der Schrittweitensteuerung. Der Beschleunigungsfaktor ist nur geringfügig kleiner als der, durch die Kombination von pessimistischer und optimistischer Schrittweitensteuerung entstehende. Die Kenntnis der Lösungskurve $x(p)$, auf der die Wahl dieser Kombination basiert, wird allerdings bei Verwendung der Iterations - adaptiven Schrittweitensteuerung nicht benötigt.

Benötigte Homotopieschritte	:	9
Rückwärtsschritte	:	1
Newton - Iterationen (gesamt)	:	22
Beschleunigungsfaktor	:	8.3

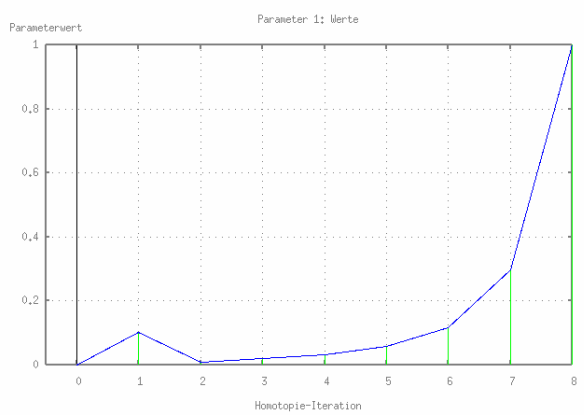

Abbildung 7-63: Verlauf des Homotopieparameters

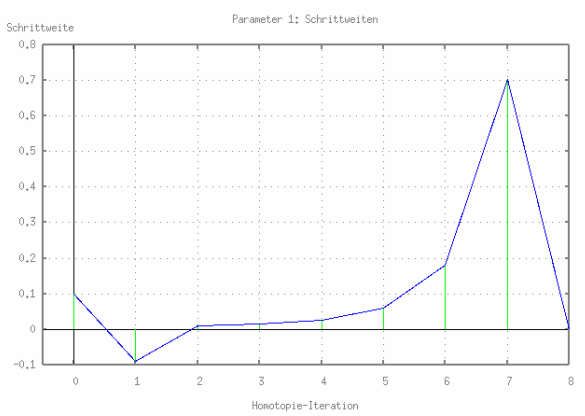

Abbildung 7-64: Schrittweiten

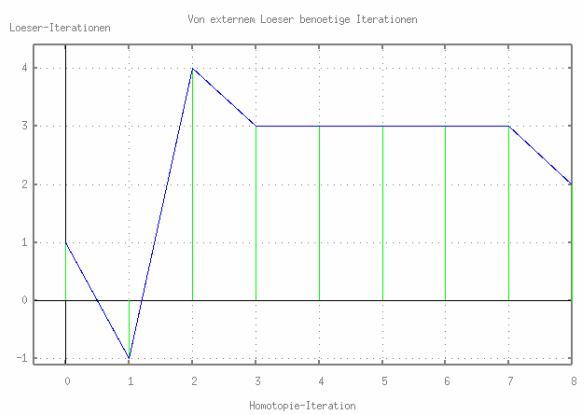

Abbildung 7-65: Iterationen im Newton - Verfahren

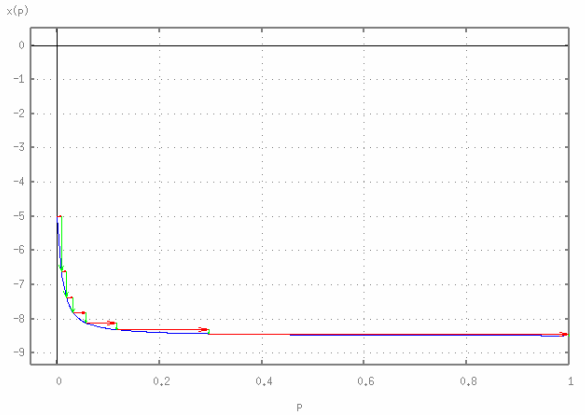

Abbildung 7-66: Verfolgung der Lösungskurve

7.1.3.8 Rechenzeit - adaptive Schrittweitensteuerung

Die Ergebnisse der Rechenzeit - adaptiven Schrittweitensteuerung unterscheiden sich für beide Anfangsschrittweiten nicht von denen der Iterations - adaptiven Schrittweitensteuerung. So können für das gegebene Zielproblem ohne Kenntnis des Verlaufs der Lösungskurve $x(p)$ und ohne die zusätzliche Angabe der im externen Lösungsverfahren benötigten Iterationen sehr gute Resultate erzielt werden.

7.2 SICOM und MINOCS

MINOCS gehört zur Klasse der direkten Kollokationsverfahren. Hierbei wird das gemischt - ganzzahlige, diskret - kontinuierliche Optimalsteuerungsproblem durch Zeitdiskretisierung in ein großes, dünn besetztes, nicht konvexes Optimierungsproblem transformiert, welches mit einem strukturausnutzenden, sequentiellen, quadratischen Optimierungsverfahren (SQP) gelöst wird.
Die Funktionsweise dieses Verfahrens soll hier nicht näher erläutert werden. Wie auch die in Kapitel 4 beschriebenen, von E. Grigat zur Optimierung verwendeten Schießverfahren, basieren SQP - Verfahren allerdings auf Newton - ähnlichen Iterationsschritten. Sie sind damit ebenso anfällig für die in 3.2 vorgestellten Probleme des Newton - Verfahrens, wodurch der Einsatz zusätzlicher Globalisierungsstrategien, wie sie z.B. durch Homotopieverfahren geboten werden, notwendig wird.

Alle nachfolgenden Beispiele wurden auf einem Notebook mit einem Intel Mobile Pentium IV Prozessor mit einer Taktfrequenz von 1.8 GHz und 512 MB RAM unter SUSE Linux 9.1 berechnet.

7.2.1 Zeitminimaler Weg mit Hindernis

In diesem Beispiel soll ein zeitminimaler Weg für das 4.1 vorgestellte, differentialgesteuerte Fahrzeug gefunden werden. Zu berechnen ist dabei ein Weg in der x,y - Ebene des globalen Koordinatensystems von dem Startpunkt $x(t_0)=0$, $y(t_0)=0$ zum Zielpunkt $x(t_f)=10$, $y(t_f)=0$. In der Mitte dieses Weges, am Punkt $x_{pylon}=5$, $y_{pylon}=0$ befindet sich eine Pylone, die mit einem Mindestabstand von 3 Umfahren werden soll. Das entsprechende Optimalsteuerungsproblem ist gegeben durch

$$\min t_f \quad \text{mit}$$

$$\dot{x}(t)=\frac{v_r(t)+v_l(t)}{d}\cdot\cos\theta(t),$$

$$\dot{y}(t)=\frac{v_r(t)+v_l(t)}{d}\cdot\sin\theta(t),$$

$$\dot{\theta}(t)=\frac{v_r(t)-v_l(t)}{d}.$$

Die Nebenbedingungen sind

$$
\begin{aligned}
& t_0 = 0, \\
& x(t_0) = 0,\ x(t_f) = 10, \\
& y(t_0) = 0,\ y(t_f) = 0, \\
& (x(t) - x_{pylon})^2 + (y(t) - y_{pylon})^2 \geq 9.
\end{aligned}
$$

Da es für die Zeitminimalität des Weges unerheblich ist, ob dieser oberhalb oder unterhalb des Hindernisses vorbeiführt, existieren für dieses Problem, wie in den Abbildungen 7-67 und 7-68 gezeigt, zwei optimale Lösungen.

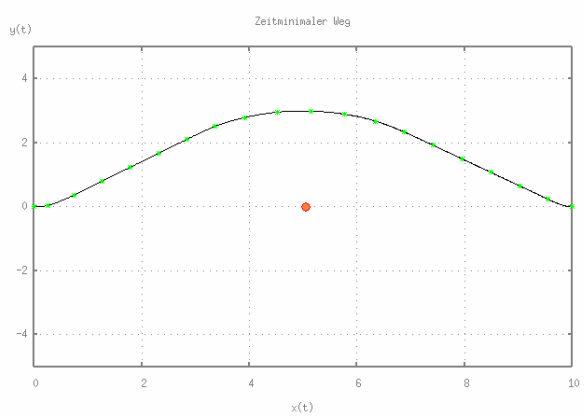

Abbildung 7-67: Zeitminimaler Weg oberhalb des Hindernisses

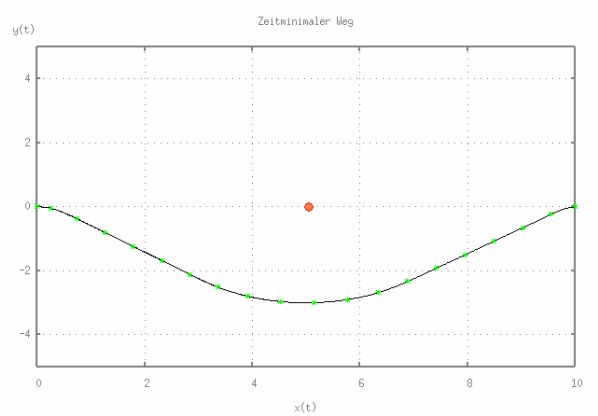

Abbildung 7-68: Zeitminimaler Weg unterhalb des Hindernisses

Ohne weitere Nebenbedingungen liefert MINOCS den unteren Weg als optimale Lösung. Wird als Startnäherung eine direkte Verbindungsgerade zwischen Start- und Zielpunkt verwendet, benötigt MINOCS zur Berechnung dieser Lösung eine Rechenzeit von 1.4 Sekunden.
Durch den Einsatz von SICOM kann die Rechenzeit unter gleichen Voraussetzungen auf 0.8 Sekunden reduziert werden. Dies entspricht einer Beschleunigung um den Faktor 1.75.
Um dies zu erreichen, wird die Pylone schrittweise in den Weg des Fahrzeugs geschoben, wobei die Lösung eines Zwischenproblems jeweils als Startnäherung für das folgende Zwischenproblem verwendet wird. Die y-Koordinate y_{pylon} der Pylone wird SICOM dafür als zu steuernder Homotopieparameter übergeben und so in drei Schritten von $y_{pylon} = 3$ auf $y_{pylon} = 0$ verkleinert. Die entsprechenden zeitminimalen Wege sind in den Abbildungen 7-69 bis 7-71 dargestellt.

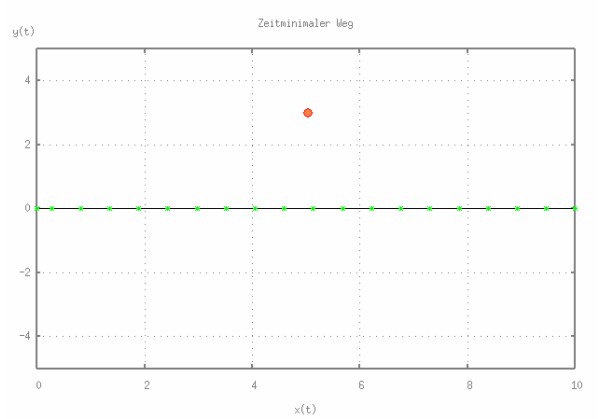

Abbildung 7-69: Zeitminimaler Weg für $y_{pylon} = 3$

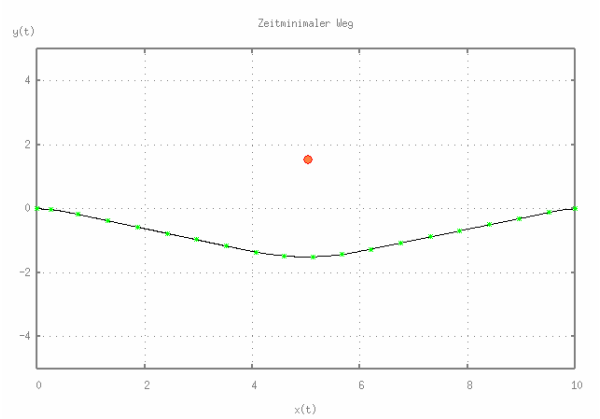

Abbildung 7-70: Zeitminimaler Weg für $y_{pylon} = 1.5$

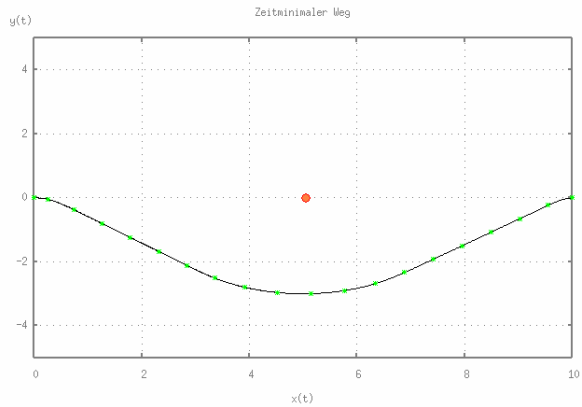

Abbildung 7-71: Zeitminimaler Weg für $y_{pylon} = 0$

Durch die Steuerung von y_{pylon} durch SICOM ergibt sich eine weitere Anwendungsmöglichkeit des Homotopieverfahrens. Wird mit Hilfe von SICOM die Pylone nicht von oben, sondern von unten in den Weg des Fahrzeugs geschoben, kann der zweite, in Abbildung 7-67 dargestellte, zeitminimale Weg als optimale Lösung des Problems erzwungen werden. Möglich ist dies, da für alle $y_{pylon} < 0$ der jeweilige zeitminimale Weg sicher oberhalb der Pylone verläuft und ausgehend von einem solchen Weg als Startnäherung die Berechnung des oberen Weges bei $y_{pylon} = 0$ für das Optimierungsverfahren “näher liegt“ als die Berechnung des unteren Weges.
Die Abbildungen 7-72 bis 7-74 zeigen die Lösungen der entsprechenden Zwischenprobleme.

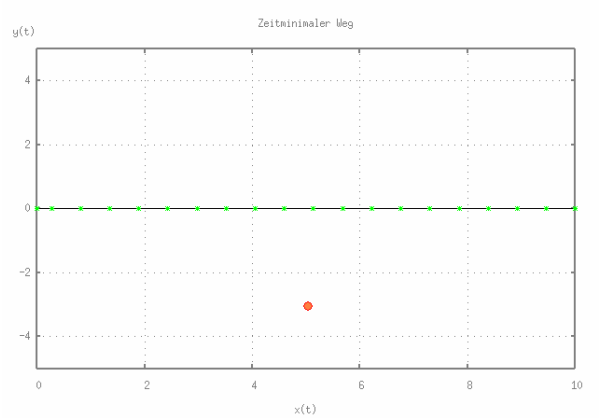

Abbildung 7-72: Zeitminimaler Weg für $y_{pylon} = -3$

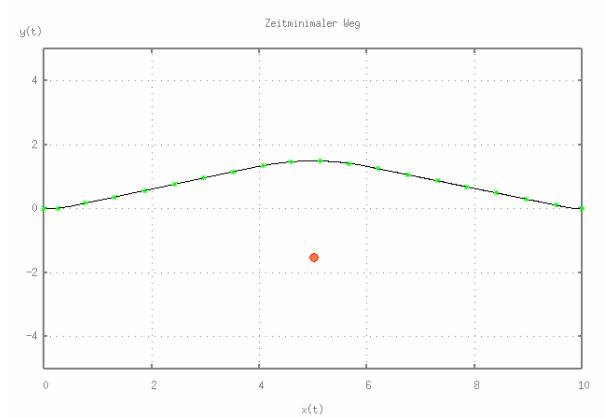

Abbildung 7-73: Zeitminimaler Weg für $y_{pylon} = -1.5$

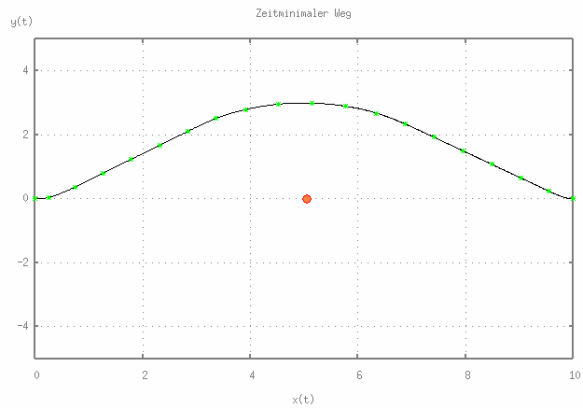

Abbildung 7-74: Zeitminimaler Weg für $y_{pylon} = 0$

Die Berechnung dieses oberen, zeitminimalen Weges ist durchaus auch ohne die Verwendung von SICOM möglich. Allerdings ist dafür die Formulierung weiterer Nebenbedingungen für das Optimalsteuerungsproblem notwendig.
Um den oberen Weg allein mit MINOCS zu berechnen, muß das Problem in zwei Phasen unterteilt werden. In der ersten Phase verläuft dabei der Weg des Fahrzeugs von seinem Startpunkt bis zur x-Koordinate $x_{pylon} = 5$ der Pylone. In der zweiten Phase wird der übrige Weg bis zum Zielpunkt zurückgelegt. Für die erste Phase kann dann eine Randbedingung für die y-Koordinate des Fahrzeugs formuliert werden, durch die erreicht wird, daß der Weg des Fahrzeugs oberhalb der Pylone verläuft. In einer weiteren Nebenbedingung muß der Übergang zwischen den beiden Phasen beschrieben werden.

Aus den zusätzlichen, an den Optimierer gestellten Bedingungen folgt ein Anstieg der zur Bestimmung der Lösung erforderlichen Rechenzeit. Durch die Verwendung von SICOM kann in diesem Fall also nicht nur die Formulierung des Problems vereinfacht, sondern auch eine weitere Erhöhung der zur Berechnung benötigten Zeit vermieden werden.

7.2.2 Zeitminimaler Weg mit zwei Hindernissen

Die Problemstellung aus 7.2.1 soll nun um ein weiteres Hindernis erweitert werden. Dazu wird neben der ersten Pylone am Punkt $x_{pylon1} = 5$, $y_{pylon1} = 0$ eine Zweite am Punkt $x_{pylon2} = 10$, $y_{pylon2} = 0$ plaziert. Ziel ist die Berechnung einer Slalomfahrt mit Mindestabstand 2 um die Pylonen, ausgehend vom Startpunkt $x(t_0) = 0$, $y(t_0) = 0$ zum Zielpunkt $x(t_f) = 15$, $y(t_f) = 0$.

Die geänderten Nebenbedingungen für das Optimalsteuerungsproblem aus 7.2.1 ergeben sich dementsprechend zu

$$
\begin{aligned}
& t_0 = 0, \\
& x(t_0) = 0,\ x(t_f) = 15, \\
& y(t_0) = 0,\ y(t_f) = 0, \\
& (x(t) - x_{pylon1})^2 + (y(t) - y_{pylon1})^2 \geq 4, \\
& (x(t) - x_{pylon2})^2 + (y(t) - y_{pylon2})^2 \geq 4.
\end{aligned}
$$

Soll eine Slalomfahrt ohne die Verwendung von SICOM berechnet werden, ist eine Formulierung des Problems in mehreren Phasen unumgänglich, da sonst der zeitminimale Weg nur oberhalb oder unterhalb beider Pylonen verläuft. Dieser Effekt wird in den Abbildungen 7-75 und 7-76 dargestellt.

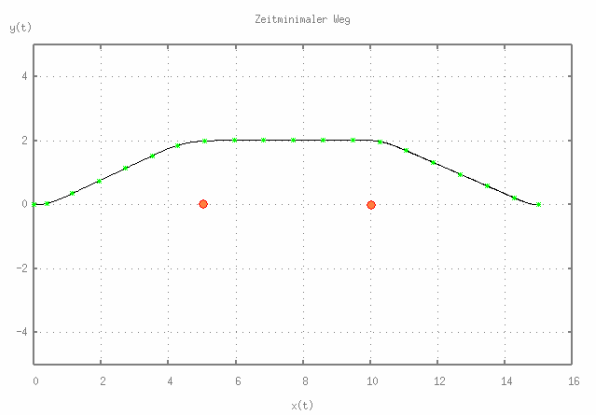

Abbildung 7-75: Zeitminimaler Weg oberhalb beider Pylonen

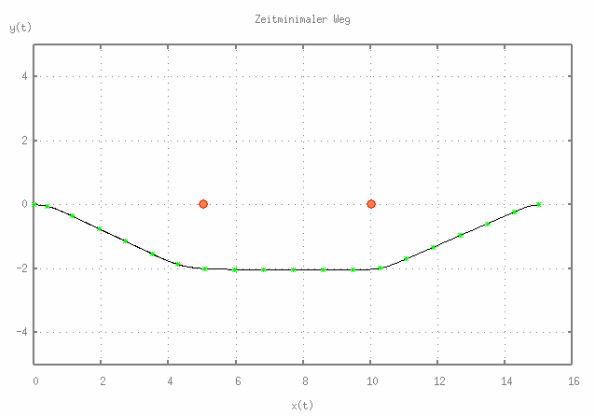

Abbildung 7-76: Zeitminimaler Weg unterhalb beider Pylonen

Durch die Unterteilung des Problems in mehrere Phasen, kann analog zu der in 7.2.1 beschriebenen Vorgehensweise, die Slalomfahrt erzwungen werden. MINOCS benötigt dann 5.3 Sekunden zur Berechnung der Lösung.

Auch hier kann mit Hilfe von SICOM die Formulierung des Problems deutlich vereinfacht und zusätzlich eine Verringerung der zur Bestimmung der Lösung benötigten Rechenzeit auf 3.3 Sekunden bewirkt werden, was einer Beschleunigung um den Faktor 1.6 entspricht.
Die Pylonen werden dafür wieder in drei Homotopieschritten von unten bzw. von oben in den Weg des Fahrzeugs geschoben. SICOM wird also zur Steuerung der y-Koordinaten beider Pylonen verwendet.
Die Abbildungen 7-77 bis 7-79 zeigen die Lösungen der entstehenden Zwischenprobleme.

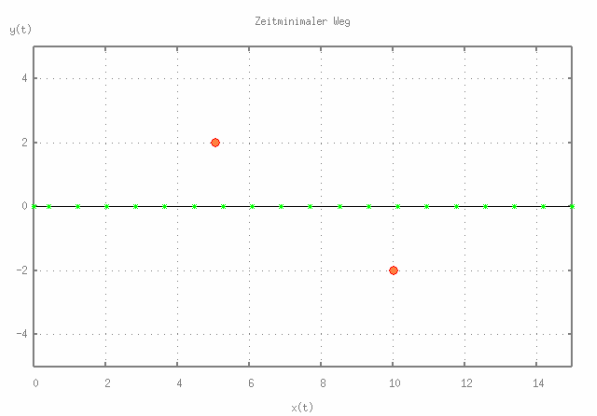

Abbildung 7-77: Zeitminimaler Weg für $y_{pylon1} = 2$, $y_{pylon2} = -2$

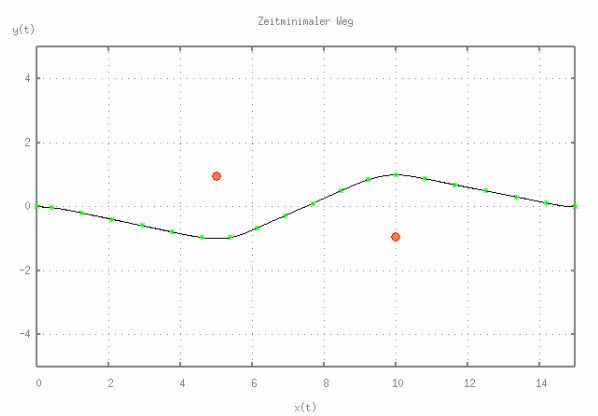

Abbildung 7-78: Zeitminimaler Weg für $y_{pylon1} = 1$, $y_{pylon2} = -1$

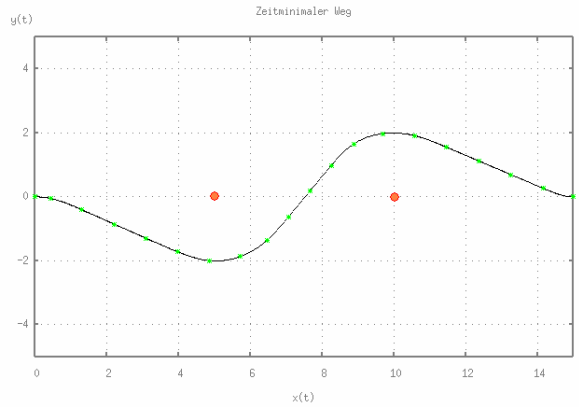

Abbildung 7-79: Zeitminimaler Weg für $y_{pylon1} = 0$, $y_{pylon2} = 0$

8 Fazit

Die Idee zur Lösung numerischer Probleme durch den Einsatz von Homotopieverfahren läßt sich von der menschlichen Vorgehensweise bei der Lösung von Problemen ableiten. Auch hier werden schwierige Aufgaben gelöst, indem sie in einfacher lösbare, aufeinander aufbauende Teilschritte zerlegt werden.
Numerische Homotopieverfahren betten dazu das zu lösende mathematische Problem in eine Schar von Problemen ein. Der Homotopieparameter dient dabei zur Deformation des einfach zu lösenden Startproblems in das Zielproblem, wobei jeweils die Lösung eines Zwischenproblems als Startnäherung für das nächste entstehende Problem genutzt wird.
Notwendig wird diese Vorgehensweise, da viele numerische Lösungsverfahren auf dem Newton - Verfahren zur Lösung nichtlinearer Gleichungssysteme aufbauen. Da das Newton - Verfahren allerdings oft nur einen geringen Konvergenzradius aufweist, wird der Einsatz zusätzlicher Globalisierungsstrategien, wie sie durch Homotopieverfahren geboten werden, notwendig.

Der Einsatz bzw. die Entwicklung eines solchen Verfahrens sind allerdings durchaus nicht so trivial, wie sie auf den ersten Blick erscheinen mögen. Wichtig ist die Wahl einer Homotopie, die einen Verlauf der Lösungen der Zwischenprobleme erzeugt, dessen Verfolgung mit Hilfe numerischer Methoden möglich ist. Ein weiteres Kernproblem von Homotopieverfahren stellt die Steuerung der Schrittweite des Homotopieparameters dar. Sie beeinflußt direkt die Anzahl der zur Lösung des Zielproblems benötigten Homotopieschritte und entscheidet damit über die Effizienz des Verfahrens.

Mit dem von E. Grigat in [8] entwickelten Homotopieverfahren zur Berechnung von Optimalflugbahnen in Fallwindgebieten und HOMPACK, einem Verfahren zur Lösung allgemeiner, nichtlinearer Gleichungssysteme wurden zwei Homotopieverfahren aus der Praxis vorgestellt. Beide Verfahren wurden für die direkte Zusammenarbeit mit dem jeweiligen numerischen Lösungsverfahren entwickelt, so daß sie nicht ohne evt. weitreichende Anpassungen in Verbindung mit anderen numerischen Lösungsverfahren eingesetzt werden können.

Ziel bei der Entwicklung von SICOM war, ein weitestgehend vom externen numerischen Lösungsverfahren unabhängiges Homotopieverfahren zur Verfügung zu stellen. Auf einen umfangreichen Datenaustausch mit dem externen numerischen Lösungsverfahren, wie er bei dem Verfahren von E. Grigat und HOMPACK gegeben ist, wurde daher verzichtet. SICOM bietet verschiedene, heuristische und adaptive Methoden zur Steuerung der Homotopieschrittweite, mit deren Hilfe trotz der geringen zur Verfügung stehenden Information über das externe numerische Lösungsverfahren gute Resultate erzielt werden können. Ist der Verlauf der Lösungen der Zwischenprobleme annähernd bekannt oder vorhersagbar, kann die Schrittweitensteuerung von SICOM durch den Benutzer entsprechend angepaßt werden.

Die Anwendungsbeispiele von SICOM haben gezeigt, daß das Homotopieverfahren nicht nur zur Erzeugung globaler Konvergenz des externen numerischen Lösungsverfahrens eingesetzt werden kann. Möglich ist der Einsatz von SICOM auch zur Beschleunigung des Konvergenzverhaltens sowie zur Vereinfachung der mathematischen Formulierung eines Problems.

9 Literaturverzeichnis

[1] Jänich, Klaus: *Topologie*, 7. Aufl., Springer 2001

[2] Alexa, Marc; Cohen-Or, Daniel; Levin David: As-Rigid-As-Possible Shape Interpolation, www.dgm.informatik.tu-darmstadt.de/pub/arap.pdf

[3] R. Shepard, J. Metzler: Mental rotation of three-dimensional objects, Science, 1971

[4] D. Krech, R. S. Crutchfield: Grundlagen der Psychologie, Psychologie Verlags Union, 1992

[5] R. Bappert, S. Benner, B. Häcker, U. Kern, G. Zweckbronner: Bionik, Zukunftstechnik lernt von der Natur, Schultheis Printservice, 2003

[6] Herrmann, Martin: Numerische Mathematik, Oldenbourg, 2001

[7] von Stryk, Oskar: Skript zur Vorlesung "Robotik 1", Technische Universität Darmstadt, Fachgebiet Simulation und Systemoptimierung, 2004

[8] Grigat, Ernst: Berechnung von Optimalflugbahnen in Fallwindgebieten mittels eines Homotopieverfahrens variabler Ordnung, Herbert Utz Verlag, 1997

[9] von Stryk, Oskar: Skript zur Vorlesung "Mobile und sensorgeführte Robotiksysteme", Technische Universität Darmstadt, Fachgebiet Simulation und Systemoptimierung, 2004

[10] Davidenko, D.F.: On a new method of numerical solution of systems of nonlinear equations, Doklady Akad. Nauk, 1953

[11] Layne T. Watson, Stephen C. Billups, Alexander P. Morgan: ALGORITHM 652 - HOMPACK: A Suite of Codes for Globally Convergent Homotopy Algorithms, ACM Transactions on Mathematical Software, Vol. 13, No. 3, September 1987

[12] Jorge Nocedal, Stephen J. Wright: Numerical Optimization, Springer Series in Operations Research, Springer 1999

[13] Layne T. Watson, Maria Sosonkina, Robert C. Melville, Alexander P. Morgan, Homer F. Walker: Algorithm 777 - HOMPACK90: A Suite of Fortran 90 Codes for Globally Convergent Homotopy Algorithms, ACM Transactions on Mathematical Software, Vol. 23, No. 4, Dezember 1997

[14] H. Schwetlick, H. Kretzschmar: Numerische Verfahren für Naturwissenschaftler und Ingenieure, Fachbuchverlag GmbH Leipzig, 1991

[15] M. Glocker: Modellierung und Numerik Gemischt-Ganzzahliger Optimalsteuerungsprobleme, Diplomarbeit 1999/2000, Technische Universität München, Lehrstuhl M2 Höhere Mathematik und Numerische Mathematik

A Beschreibung der Schnittstellen von SICOM

A.1 Die abstrakte Klasse <Homotopy>

Die abstrakte Klasse <Homotopy> dient zum Aufruf des externen numerischen Lösungsverfahrens, sowie zum Laden und Speichern dessen Ergebnisse.

A.1.1 <virtual Result* evaluate(void)>

Die Methode <evaluate> wird in der von <Homotopy> abgeleiteten Klasse durch den Benutzer implementiert und soll SICOM die Information zur Verfügung stellen, wie das verwendete numerische Lösungsverfahren aufgerufen werden kann. Muß während des Homotopieverfahrens ein entstehendes Zwischenproblem durch das numerischen Lösungsverfahren gelöst werden, wird innerhalb von SICOM die Methode <evaluate> ausgeführt.
Der Rückgabewert dieser Methode ist ein Zeiger auf ein Objekt vom Typ <Result>, welches die Information beinhaltet, ob das entsprechende Zwischenproblem gelöst werden konnte.

A.1.2 <virtual void saveResult(int)> und <virtual void loadResult(int)>

Mit Hilfe dieser Methoden kann SICOM Ergebnisse des numerischen Lösungsverfahrens speichern bzw. diese als Startnäherung für ein zu lösendes Problem laden. Wie auch <evaluate> werden diese Methoden in der von <Homotopy> abgeleiteten Klasse durch den Benutzer implementiert und während des Homotopieverfahrens von SICOM aufgerufen. Um auf weiter zurückliegende Berechnungen zugreifen zu können, werden die Ergebnisse mit einem entsprechenden Index versehen. Dazu erhalten die Methoden einen Eingabewert vom Typ Integer.

A.1.3 <void addParameter(Parameter*)> und <void removeParameter(Parameter*)>

Bevor das Homotopieverfahren gestartet werden kann, müssen SICOM die zu steuernden Parameter übergeben werden. Dies geschieht mit Hilfe der Methode <addParameter>. Soll SICOM ohne vorhergehende Zerstörung des Objekts der Superklasse <Homotopy> erneut aufgerufen

werden, können nicht mehr benötigte bzw. von SICOM zu kontrollierende Parameter mittels <removeParameter> aus der Liste der zu steuernden Parameter entfernt werden. Das entsprechende Objekt der Oberklasse <Parameter> wird dadurch nicht beeinflußt. Beide Methoden erhalten als Eingabewert einen Zeiger auf ein Objekt der Oberklasse <Parameter>.

A.1.4 <void setTrace(bool)>

Mit dieser Methode kann der Trace des Homotopieverfahrens an- bzw. ausgeschaltet werden. In der Konsole erfolgt bei eingeschaltetem Trace eine Ausgabe der Werte der Parameter und der entsprechenden Schrittweiten. Diese und der pro Homotopieschritt durch das externe Lösungsverfahren verursachte Rechenaufwand in Taktzyklen werden zusätzlich in Dateiform erfaßt. Erzeugt wird weiterhin die Steuerdatei "HomotopyTrace.gpf" für GNUplot. So können die gesammelten Daten grafisch dargestellt werden.

A.1.5 <void setIterationLimit(int)>

Um die maximale Anzahl der Homotopieschritte zu beschränken, kann durch <setIterationLimit> eine obere Grenze hierfür gesetzt werden. Falls dies nicht durch den Benutzer geschieht, wird eine Schranke von 1000 Iterationen vorgegeben.

A.1.6 <void start(void)>

Durch den Aufruf dieser Methode wird das Homotopieverfahren gestartet. Alle zu steuernden Parameter müssen SICOM zu diesem Zeitpunkt bekannt sein. Das Homotopieverfahren endet, wenn alle Parameter den gesetzten Zielwert erreicht haben, wenn also das Zielproblem erfolgreich gelöst werden konnte. Ein Abbruch erfolgt bei Erreichen des Limits der Homotopieschritte oder bei Unterschreiten einer vorgegebenen Schranke für die Schrittweite eines Parameters.

A.1.7 <void reset(void)>

<reset> wird, wie auch <removeParameter>, benötigt, falls das Homotopieverfahren mehrfach gestartet werden soll, ohne das Objekt der Oberklasse <Homotopy> zu zerstören. Der Aufruf setzt den Automaten zur Steuerung des Homotopieverfahens zurück in den Startzustand. Weiterhin werden die Werte aller Homotopieparameter mit den entsprechenden Startwerten belegt.

A.2 Die abstrakte Klasse <Parameter>

<Parameter> stellt Methoden zur Gestaltung der von SICOM zu steuernden Homotopieparameter zur Verfügung. Zeiger auf diese Parameter werden, wie in A.1 beschrieben, der durch den Benutzer abgeleiteten Klasse der Oberklasse <Homotopy> übergeben.

A.2.1 <virtual void update(double)>

In einer von <Parameter> abgeleiteten Klasse soll diese Methode durch den Benutzer implementiert werden.
Bei jeder Änderung des Parameterwertes durch SICOM wird diese Methode aufgerufen, um dem externen numerischen Lösungsverfahren den neuen Wert bekannt zu geben.

A.2.1 <void setInitialValue(double)> und <void setFinalValue(double>

Mit Hilfe dieser Methoden können der Startwert p_{start} bzw. der Zielwert p_{ziel} des betreffenden Parameters festgelegt werden. Während des Homotopieverfahrens bewegen sich die Werte des Parameter in dem Intervall $[\, p_{start}, p_{ziel} \,]$.

A.2.2 <void setInitialInterval(double)>

<setInitialInterval> dient zur Festsetzung der Schrittweite s_{start} für den ersten Homotopieschritt. Bei Aufruf findet eine Plausibilitätsprüfung des angegebenen Wertes statt. Bewirkt dieser einen Schritt, der das Intervall $[\, p_{start}, p_{ziel} \,]$ verlassen würde, wird der angegebene Wert ignoriert.

Der Aufruf von <setInitialInterval> ist nicht zwingend notwendig. Wird vom Benutzer keine bzw. eine fehlerhafte Anfangsschrittweite angegeben, gilt

$$s_{start} = \frac{p_{ziel} - p_{start}}{2}.$$

A.2.3 <void setMinimumInterval(double)> und <void setMaximumInterval(double>

Diese Methoden dienen zur Angabe der minimalen Schrittweite s_{min} und der maximalen Schrittweite s_{max} des Parameters. Auch hier findet eine Plausibilitätsprüfung analog zu A.2.2 statt. Werden durch den Benutzer keine Werte angegeben, gilt

$$s_{min} = (p_{ziel} - p_{start}) \cdot 10^{-4} \text{ bzw.}$$
$$s_{max} = \frac{p_{ziel} - p_{start}}{2} .$$

A.2.4 <void setForwardStepMehtod(int)>, <void setForwardStepMethod(int, int), <void setForwardStepMethod(int, double)>, <void setBackStepMethod(int)>, <void setBackStepMethod(int, int)>, <void setBackStepMethod(int, double)>

Durch diese Methoden kann für jeden Parameter und jede Schrittrichtung eine der in 6.3 vorgestellten Varianten zur Schrittweitensteuerung ausgewählt werden. Dies geschieht durch die Angabe des ersten erwarteten Eingabewertes. Im Namensraum der Klasse Parameter werden dafür die Konstanten

- IntervalBisection,
- OptimisticSteps,
- PessimisticSteps,
- IterationAdaptiveSteps,
- TimeAdaptiveSteps und
- RandomSteps

bereitgestellt. Über den zweiten, möglichen Eingabewert kann die Schrittweitensteuerung, wie in 6.3 beschrieben, angepaßt werden.
Falls keine Angabe zur verwendenden Variante der Schrittweitensteuerung erfolgt, wird als Standard die Intervallbisektion genutzt.

A.2.5 <void reset(void)>

Soll das Homotopieverfahren mehrfach gestartet werden, kann der Wert eines Parameters durch Aufruf von <reset> auf seinen Startwert zurückgesetzt werden.

A.3 Die Klasse <Result>

Über einen Zeiger auf ein Objekt des Typs <Result> erhält SICOM die Information, ob die Lösung eines Zwischenproblems durch das externe numerische Lösungsverfahren berechnet werden konnte.

A.3.1 <void setResult(bool)> und <void setResult(bool, int)>

Der Eingabewert für <setResult> ist true, falls in einem Aufruf des externen Verfahrens eine Lösung berechnet werden konnte, false, falls die Berechnung der Lösung nicht möglich war.
Über einen zweiten, ganzzahligen Eingabewert, muß, falls die Iterations - adaptive Schrittweitensteuerung verwendet werden soll, die Anzahl der in einem Aufruf des externen Verfahrens durchlaufenen Iterationen angegeben werden.

A.3.2 <void setInvalidProblem(void)>

Durch den Aufruf dieser Methode kann das Homotopieverfahren zu jedem Zeitpunkt unterbrochen werden. So kann, falls die Wahl der Homotopie zu ungültigen Zwischenproblemen führt, die Ausführung weiterer, erfolgloser Homotopieschritte vermieden werden.

A.4 Ein Implementierungsbeispiel

Das folgende Beispiel zeigt die Implementierung eines in SICOM eingebetteten Newton - Verfahrens. Gelöst werden soll das, aus 3.2.1 und 7.1.1 bekannte Zielproblem

$$f(x) \overset{!}{=} 0 \text{ mit}$$
$$f : \Re^+ \to \Re, \quad f(x) = ln(x).$$

Die verwendete Homotopie ist gegeben durch

$$h : \Re^+ \times [0,1] \to \Re,$$
$$h(x,p) = p \cdot f(x) + (1-p) \cdot g(x).$$

Als Startproblem wird

$$g(x) \overset{!}{=} 0 \text{ mit}$$
$$g : \Re^+ \to \Re, \quad g(x) = x - 4.0$$

verwendet. Der Startwert für den ersten Aufruf des Newton - Verfahrens sei $x_0 = 4.0$.

A.4.1 Implementierung der abgeleiteten <Parameter> Klasse

In der nachfolgenden, von <Parameter> abgeleiteten Klasse <NewtonHomotopyParameter> wird der für das Homotopieverfahren benötigte Homotopieparameter implementiert. Neben der benötigten Implementierung der virtuellen Methode <update> aus der Oberklasse <Parameter> wird die Methode <getValue> zum Abfragen des Parameterwertes zur Verfügung gestellt.

```
#include "Parameter.h"

class NewtonHomotopyParameter : public Parameter{
  private:
    double parameterValue;

  public:
    void update(double value){
      parameterValue=value;
    }

    double getValue(void){
      return parameterValue;
    }
};
```

A.4.2 Implementierung der abgeleiteten <Homotopy> Klasse

Die von <Homotopy> abgeleitete Klasse <NewtonHomotopy> implementiert das in SICOM eingebettete Newton - Verfahren zur Lösung des oben genannten Zielproblems.

Im privaten Bereich enthält sie dafür einen Zeiger auf Objekte des Typs <NewtonHomotopyParameter> und <Result>. Weiterhin werden eine Variable für die Startnäherung des Newton - Verfahrens und ein Array zum Speichern der lokal ermittelten Lösungen bereitgestellt.

Der Konstruktor dient zum Initialisieren der privaten Zeiger und Variablen, zum Festlegen des Start- und Zielwertes des Homotopieparameters und zur Übergabe des Homotopieparameters an SICOM.

In der Methode <evaluate> findet der Aufruf des Newton Verfahrens, für das durch den Homotopieparameter gegebene Zwischenproblem statt. Über den Zeiger auf das Objekt vom Typ <Result> wird die Information, ob die Lösung berechnet werden konnte, an SICOM übergeben.

Die Methode <saveResult> speichert die in <evaluate> gefundenen Lösungen der Zwischenprobleme in dem entsprechenden Array, von wo sie von <loadResult> als Startnäherungen für das Newton - Verfahren geladen werden können.

<printResult> dient zur textbasierten Ausgabe eines Ergebnisses.

```
#include <iostream>

#include "Homotopy.h"
#include "NewtonHomotopyParameter.h"

class NewtonHomotopy : public Homotopy{
  private:
    NewtonHomotopyParameter* p; //Homotopieparameter
    Result* result;             //Zur Übergabe des Ergebnisses an SICOM
    double x0;                  //Startnäherung für das Newton - Verfahren
    double localResults[500];   //Array zum Speichern der Ergebnisse

  public:
    NewtonHomotopy(void){
      //Homotopieparameter initialisieren, Start- und Zielwert festlegen
      p=new NewtonHomotopyParameter();
      p->setInitialValue(0.);
      p->setFinalValue(1.);

      //Objekt vom Typ <Result> initialisieren
      result=new Result();

      //Homotopieparameter an SICOM übergeben
      this->addParameter(p);

      //Startnäherung für ersten Aufruf des Newton - Verfahren festlegen
      x0=4.0;
    }

    Result* evaluate(void){
      double f, df;
      bool solutionFound=false;

      //Newton - Iterationen bis Lösung gefunden
      //oder Iterationslimit erreicht
      for(int i=0; i<20 && !solutionFound; i++){
        //Berechnen von Funktionswert und Ableitung an der Stelle x0
        f =p->getValue() *  log(x0) + (1.0 - p->getValue()) * (x0 - 4.0);
        df=p->getValue() * (1.0/x0) + (1.0 - p->getValue()) * 1.0;

        //Falls Lösung gefunden Abbruch,
        //sonst x0 auf Schnittpunkt der Tangente mit x-Achse
        if(fabs(f) < 0.001){
          solutionFound=true;
        }else{
          x0-=f / df;
        }
      }

      //Lösung gefunden / nicht gefunden an SICOM melden
      result->setResult(solutionFound);
      return result;
    }

    void saveResult(int index){localResults[index]=x0;}

    void loadResult(int index){x0=localResults[index];}

    void printResult(void){std::cout << x0 << std::endl;}
};
```

A.4.3 Der Start des Homotopieverfahrens

Die folgende <main> - Methode startet das Homotopieverfahren und gibt das Endergebnis in einer Konsole aus.

```
#include "NewtonHomotopy.h"

int main(void){
   NewtonHomotopy* h=new NewtonHomotopy();
   h->start();
   h->printResult();
}
```

```
#ifndef _Homotopy_
#define _Homotopy_

#include <iostream>
#include <sstream>
#include <fstream>
#include <time.h>

#include "Result.h"
#include "Parameter.h"
#include "HomotopyStatus.h"
#include "ForwardStepController.h"
#include "BackStepController.h"

class Homotopy{
   friend class testHomotopy; //Zum Test des Verfahrens ohne externen Loeser

   private:
      Parameter* firstParameter; //erster Parameter der Parameterliste
      int numberOfParameters; //Laenge der Parameterliste
      bool trace; //Trace? (Ausgabe der Parameterwerte und Schrittweiten)
      HomotopyStatus* homotopyStatus; //Zeiger auf Instanz von HomotopyStatus fuer den Datenaustausch
      int iterationLimit; //Maximale Anzahl der Homotopie - Iterationen

   public:
      Homotopy(void); //Standardkonstruktor

      //Hinzufuegen und Loeschen von Parametern
      void addParameter(Parameter*);
      void removeParameter(Parameter*);

      //Zugriff auf private - Variablen trace und iterationLimit
      void setIterationLimit(int);
      void setTrace(bool);

      void start(void); //Start des Homotopieverfahrens
      void reset(void); //Alles auf Anfang

      //Vom Benutzer zu implementierende, virtuelle Methoden zum Aufruf
      //des externen Loesers sowie zum Speichern und Laden dessen Ergebnisse
      virtual Result* evaluate(void)=0;
      virtual void saveResult(int)=0;
      virtual void loadResult(int)=0;
};

#endif
```

```
#include "Homotopy.h"

//--Standardkonstruktor--
//Initialisierung der Parameterliste mit der leeren Liste, Trace einschalten,
//Instanz von HomotopyStatus fuer den Datenaustausch der Klassen bereitstellen,
//Iterationslimit auf 1000 setzen
Homotopy::Homotopy(void){
   firstParameter=0;
   numberOfParameters=0;
   trace=true;
   homotopyStatus=new HomotopyStatus();
   iterationLimit=1000;
}

//--Hinzufuegen eines Parameters--
//Parameter werden in umgekehrter Reihenfolge in eine Verkettete Liste
//mit minimaler Funktionalitaet -> Verwaltungsaufwand eingetragen
void Homotopy::addParameter(Parameter* in){
   in->setHomotopyStatusPointer(homotopyStatus);
   in->setNextParameter(firstParameter);
   firstParameter=in;
   numberOfParameters++;
}

//--Loeschen eines Parameters--
//Wenn zu loeschender Parameter erster in Liste -> zweiter Parameter neuer Listenkopf
//Sonst Vorgaenger von zu loeschendem Parameter suchen und als dessen Nachfolger den
//Nachfolger des zu loeschenden Parameters eintragen
void Homotopy::removeParameter(Parameter* in){
   Parameter *currentParameter;
   currentParameter=firstParameter;
   if(firstParameter==in){
      firstParameter=firstParameter->getNextParameter();
   }else{
      while(currentParameter->getNextParameter()!=in){
         currentParameter=currentParameter->getNextParameter();
      }
      currentParameter->setNextParameter((currentParameter->getNextParameter())->getNextParameter());
   }
   numberOfParameters--;
}

//--Setzen des Iterationslimits--
void Homotopy::setIterationLimit(int in){
   iterationLimit=in;
}

//--Trace ein-/ausschalten--
void Homotopy::setTrace(bool in){
   trace=in;
```

```
}

//--Start des Homotopieverfahrens--
void Homotopy::start(void){
   //Wenn Parameterliste leer ist erfolgt ein Aufruf des Loesers -> im Loeser muss nicht geprueft werden,
   //ob Loesung mittels Homotopie oder durch einmaliges Anwenden des Loesungsverfahrens erzeugt werden soll
   if(firstParameter==0){
      evaluate();
   }else{ //Wenn Parameterliste nicht leer ist, Start des Homotopieverfahrens
      Parameter *currentParameter; //Zeiger auf aktuellen Parameter
      int storePosition=0; //Speicherposition fuer Ergebnis des Loesers
      int iterationCounter=0; //Zaehler fuer Homotopie-Iterationen

      //Variablen fuer Trace und Statistik
      int parameterCounter=numberOfParameters;
      clock_t homotopyTimeStamp1;
      clock_t homotopyTimeStamp2;
      double homotopyTime;
      clock_t solverTimeStamp1;
      clock_t solverTimeStamp2;
      double solverTime=0.;
      int currentSolverCycles;
      std::ostringstream solverCycles;
      int minSolverCycles=0;
      int maxSolverCycles=0;
      std::ostringstream solverIterations;
      int minSolverIterations=0;
      int maxSolverIterations=0;

      homotopyTimeStamp1=clock(); //Zeit zu Begin des Homotopieverfahrens festhalten

      //Vorbereiten der Parameter
      currentParameter=firstParameter;
      while(currentParameter!=0){
         currentParameter->initiate();
         currentParameter=currentParameter->getNextParameter();
      }

      //Durchfuehren des Homotopieverfahrens solange Homotopie-FSM nicht in End- oder Fehlerzustand,
      //oder das Iterationslimit ueberschritten wird
      while(homotopyStatus->getHomotopyStatus()!=HomotopyStatus::Finished &&
            homotopyStatus->getHomotopyStatus()!=HomotopyStatus::PrecisionError &&
            iterationCounter<iterationLimit){
         //Aufruf des externen Loesungsverfahrens, bereitstellen des Ergebnisses
         //in Instanz von HomotopyStatus und Berechnen der benoetigten Zeit
         solverTimeStamp1=clock();
         homotopyStatus->setSolverResultPointer(evaluate());
         solverTimeStamp2=clock();
         currentSolverCycles=solverTimeStamp2 - solverTimeStamp1;
         homotopyStatus->setCurrentSolverCycles(currentSolverCycles);
```

```
solverTime+=currentSolverCycles;

if((homotopyStatus->solverResult)->getInvalidProblem()){
   homotopyStatus->setHomotopyStatus(HomotopyStatus::PrecisionError);
}else{
   currentParameter=firstParameter; //Zeiger fuer aktuellen Parameter auf ersten Parameter setzen
   if((homotopyStatus->solverResult)->getResult()){ //Wenn Ergebnis des Loesers ok -> Vorwaertsschritt
      if(trace){ //Trace, wenn gewuenscht
         std::cout << std::endl
                   << "Homotopie-Iteration " << (iterationCounter+1) << " (vorwaerts)" << std::endl;

         //Von externem Loeser benoetigte Takzyklen aufzeichnen
         solverCycles << iterationCounter << " " << currentSolverCycles << std::endl;
         if(currentSolverCycles > maxSolverCycles){maxSolverCycles=currentSolverCycles;}

         //Von externem Loeser benoetigte Iterationen aufzeichnen (wenn angegeben)
         if((homotopyStatus->solverResult)->getIterations()!=-1){
            solverIterations << iterationCounter << " " << (homotopyStatus->solverResult)->getIterations() << std::endl;
            if((homotopyStatus->solverResult)->getIterations() > maxSolverIterations){
               maxSolverIterations=(homotopyStatus->solverResult)->getIterations();
            }
         }
      }

      //Vorwaertsschritt fuer jeden Parameter ausfuehren
      while(currentParameter!=0){
         //Trace: Parameterwerte vor Schritt ausgeben
         if(trace){currentParameter->printStatusBeforeStep(iterationCounter, parameterCounter);}

         (currentParameter->forwardStep)->execute(); //Vorwaetsschritt ausfuehren

         if(trace){ //Trace: Parameterwerte nach Schritt
            currentParameter->printStatusAfterStep(iterationCounter);
            parameterCounter--;
         }

         currentParameter=currentParameter->getNextParameter(); //Naechsten Parameter auswaehlen
      }

      //Speichern des Loeser-Ergebnisses und Aktualisieren des Homotopie-Status
      if(homotopyStatus->getPreviousStepDirection()==HomotopyStatus::BackStep){storePosition++;}
      //saveResult(storePosition);
      //loadResult(storePosition);
      saveResult(0);
      loadResult(0);
      storePosition++;
      homotopyStatus->updateForwardStep();
   }else{ //Wenn Loeser-Ergebnis nciht ok -> Rueckwaertsschritt
      if(trace){ //Trace, wenn gewuenscht
         std::cout << std::endl
```

```
                        << "Homotopie-Iteration " << (iterationCounter+1) << " (rueckwaerts)" << std::endl;

               //Von externem Loeser benoetigte Takzyklen aufzeichnen
               solverCycles << iterationCounter << " " << -currentSolverCycles << std::endl;
               if(-currentSolverCycles < minSolverCycles){minSolverCycles=-currentSolverCycles;}

               //Von externem Loeser benoetigte Iterationen aufzeichnen (wenn angegeben)
               if((homotopyStatus->solverResult)->getIterations()!=-1){
                  solverIterations << iterationCounter << " " << -(homotopyStatus->solverResult)->getIterations() << std::endl;
                  if(-(homotopyStatus->solverResult)->getIterations() < minSolverIterations){
                     minSolverIterations=-(homotopyStatus->solverResult)->getIterations();
                  }
               }
            }

            //Rueckwaertsschritt fuer jeden Parameter ausfuehren
            while(currentParameter!=0){
               //Trace: Parameterwerte vor Schritt ausgeben
               if(trace){currentParameter->printStatusBeforeStep(iterationCounter, parameterCounter);}

               (currentParameter->backStep)->execute(); //Rueckwaertsschritt ausfueren

               if(trace){ //Trace: Parameterwerte nach Schritt
                  currentParameter->printStatusAfterStep(iterationCounter);
                  parameterCounter--;
               }

               currentParameter=currentParameter->getNextParameter(); //Naechsten Parameter auswaehlen
            }

            //Laden des letzten Erfolreichen Loeser-Ergebnisses und Aktualisieren des Homotopie-Status
            if(homotopyStatus->getPreviousStepDirection()==HomotopyStatus::ForwardStep){storePosition--;}
            //loadResult(storePosition);
            loadResult(0);
            homotopyStatus->updateBackStep();
         }

         iterationCounter++; //Iterationszaehler erhoehen
         parameterCounter=numberOfParameters; //Parameterzaehler (fuer Trace) zuruecksetzen
      }
   }

   //Trace: Parameterwerte, Schrittweiten, Takzyklen und Iterationen in Datei schreiben und GNUPlot-Datei erzeugen
   if(trace){
      //Von externem Loeser benoetige Taktzyklen in Datei schreiben
      std::ofstream solverCycleFile("SolverCycles.dat");
      solverCycleFile << solverCycles.str();

      //Von externem Loeser benoetige Iterationen (falls angegeben) in Datei schreiben
      if((homotopyStatus->solverResult)->getIterations()!=-1){
```

```
   std::ofstream solverIterationFile("SolverIterations.dat");
   solverIterationFile << solverIterations.str();
}

//Parameterwerte in Datei schreiben
currentParameter=firstParameter;
parameterCounter=numberOfParameters;
while(currentParameter!=0){
   currentParameter->writeFile(parameterCounter);
   currentParameter=currentParameter->getNextParameter();
   parameterCounter--;
}

//GNUPlot-Datei erzeugen
std::ofstream gnuPlotFile("HomotopyTrace.gpf");
gnuPlotFile << "set terminal X11" << std::endl
            << "set xzeroaxis lt 1" << std::endl
            << "set yzeroaxis lt 1" << std::endl
            << "set xrange[-.5:" << iterationCounter-1 << "]" << std::endl
            << "set grid" << std::endl
            << "set nokey" << std::endl
            << "set zero 1.e-90" << std::endl
            << "set title \"Von externem Loeser benoetigte Taktzyklen\"" << std::endl
            << "set xlabel \"Homotopie-Iteration\"" << std::endl
            << "set ylabel \"Taktzyklen\"" << std::endl
            << "set yrange[" << minSolverCycles+(minSolverCycles*.1) << ":"
                             << maxSolverCycles+(maxSolverCycles*.1) << "]" << std::endl
            << "plot \"SolverCycles.dat\" using 1:2 with impulses lt 3,"
            << "\"SolverCycles.dat\" using 1:2 with line lt 2" << std::endl;
if((homotopyStatus->solverResult)->getIterations()!=-1){
   gnuPlotFile << "pause -1 \"Return fuer Loeser-Iterationen...\"" << std::endl
               << "set title \"Von externem Loeser benoetige Iterationen\"" << std::endl
               << "set xlabel \"Homotopie-Iteration\"" << std::endl
               << "set ylabel \"Loeser-Iterationen\"" << std::endl
               << "set yrange[" << minSolverIterations+(minSolverIterations*.1) << ":"
                                << maxSolverIterations+(maxSolverIterations*.1) << "]" << std::endl
               << "plot \"SolverIterations.dat\" using 1:2 with impulses lt 3,"
               << "\"SolverIterations.dat\" using 1:2 with line lt 2" << std::endl;
}
gnuPlotFile << "pause -1 \"Return fuer Parameterwerte...\"" << std::endl;
for(int i=1; i<=numberOfParameters; i++){
   gnuPlotFile << "set title \"Parameter " << i << ": Werte\"" << std::endl
               << "set xlabel \"Homotopie-Iteration\"" << std::endl
               << "set ylabel \"Parameterwert\"" << std::endl
               << "set yrange[*:*]" << std::endl
               << "plot \"ParameterValues" << i << ".dat\" using 1:2 with impulses lt 3,"
               << "\"ParameterValues" << i << ".dat\" using 1:2 with line lt 2" << std::endl
               << "pause -1 \"Return fuer Schrittweiten...\"" << std::endl
               << "set title \"Parameter " << i << ": Schrittweiten\"" << std::endl
               << "set xlabel \"Homotopie-Iteration\"" << std::endl
```

```
                        << "set ylabel \"Schrittweite\"" << std::endl
                        << "set yrange[*:*]" << std::endl
                        << "plot \"StepWidthValues" << i << ".dat\" using 1:2 with impulses lt 3,"
                        << "\"StepWidthValues" << i << ".dat\" using 1:2 with line lt 2" << std::endl
                        << "pause -1 \"Return fuer naechsten Parameter...\"" << std::endl;
            }
        }

        //Berechnen der vom Homotopieverfahren (ohne externen Loeser) benoetigten Zeit
        homotopyTimeStamp2=clock();
        homotopyTime=((double)(homotopyTimeStamp2 - (homotopyTimeStamp1 + solverTime))) / (double)CLOCKS_PER_SEC;
        solverTime=((double)solverTime) / (double)CLOCKS_PER_SEC;

        //Statistik ausgeben
        std::cout << std::endl
                  << "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -" << std::endl;
        if(iterationCounter==iterationLimit && homotopyStatus->getHomotopyStatus()!=HomotopyStatus::Finished){
           std::cout << "Keine Loesung gefunden      : Iterationslimit erreicht" << std::endl
                     << std::endl;
        }
        if(homotopyStatus->getHomotopyStatus()==HomotopyStatus::PrecisionError){
           std::cout << "Keine Loesung gefunden      : Minimale Schrittweite erreicht" << std::endl
                     << std::endl;
        }
        std::cout << "Homotopie-Iterationen         : " << iterationCounter << std::endl
                  << "    davon Erfolgreich         : " << storePosition << std::endl
                  << std::endl
                  << "Zeit fuer Homotopie-Verfahren : " << homotopyTime << " Sekunden" << std::endl
                  << "Zeit fuer Loesungsverfahren   : " << solverTime << " Sekunden" << std::endl
                  << "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -" << std::endl
                  << std::endl;
    }
}

//--Alles auf Anfang
void Homotopy::reset(void){
    Parameter* currentParameter;
    currentParameter=firstParameter;
    while(currentParameter!=0){
        currentParameter->reset();
        currentParameter=currentParameter->getNextParameter();
    }
    homotopyStatus->setHomotopyStatus(HomotopyStatus::Start);
}
```

```
#ifndef _Parameter_
#define _Parameter_

#include <iostream>
#include <sstream>
#include <fstream>
#include <string>

#include "HomotopyStatus.h"

//Vordefinition der Klassen fuer die Schrittweitensteuerung
class ForwardStepController;
class BackStepController;

class Parameter{
   //Deklaration der Friend-Klassen fuer Zugriff auf private-Methoden
   friend class Homotopy;
   friend class ForwardStepController;
   friend class BackStepController;
   friend class IntervalBisectionForwardStep;
   friend class IntervalBisectionBackStep;
   friend class PessimisticForwardStep;
   friend class PessimisticBackStep;
   friend class OptimisticForwardStep;
   friend class OptimisticBackStep;
   friend class IterationAdaptiveForwardStep;
   friend class IterationAdaptiveBackStep;
   friend class TimeAdaptiveForwardStep;
   friend class TimeAdaptiveBackStep;
   friend class RandomForwardStep;
   friend class RandomBackStep;
   friend class testHomotopy; //fuer Test ohne externes Loesungsverfahren

   private:
      double initialValue; //Startwert des Parameters
      double finalValue; //Zielwert
      double currentValue; //aktueller Wert
      double lastSuccessfulValue; //Wert bei letzter erfolgreicher Auswertung
      double backStepTriggerValue; //Wert, der Rueckwaertsschritt(e) ausgeloest hat
      double initialInterval; //Start-Schrittweite
      bool setInitialIntervalCalled; //Start-Schrittweite von Benutzer festgelegt?
      double minimumInterval; //Minimale Schrittweite -> Abbruchbedingung
      bool setMinimumIntervalCalled; //Minimale Schrittweite von Benutzer festgelegt?
      double maximumInterval; //Maxmimale Schrittweite
      bool setMaximumIntervalCalled; //Maximale Schrittweite vom Benutzer festgelegt?
      double currentInterval; //aktuelle Schrittweite
      Parameter* nextParameter; //Zeiger auf naechsten Parameter in Liste
      ForwardStepController *forwardStep; //Zeiger auf Schrittweiten-Controller (vorwaerts)
      bool setForwardStepMethodCalled; //Methode fuer Vorwaertsschritt von Benutzer festgelegt?
      BackStepController *backStep; //Zeiger auf Schrittweiten-Controller (rueckwawerts)
```

```
    bool setBackStepMethodCalled; //Methode fuer Rueckwaertsschritt von Benutzer festgelegt?
    HomotopyStatus *homotopyStatus; //Zeiger auf HomotopyStatus fuer Datenaustausch
    bool forwardStepsFrozen; //Vorwaertsschrittweite eingefroren?
    std::ostringstream parameterValues; //String-Stream fuer graph. Ausgabe d. Parameterwerte
    std::ostringstream stepWidthValues; //String-Stream fuer graph. Ausgabe d. Schrittweiten

    //Methoden zum Zugriff auf private-Variablen
    //(private Methoden werden ausschliesslich fuer das Homotopieverfahren genutzt,
    //sie stehen dem Benutzer nicht zur Verfuegung)
    void setCurrentValue(double);
    double getCurrentValue(void);
    void saveLastSuccessfulValue(void);
    double getLastSuccessfulValue(void);
    void saveBackStepTriggerValue(void);
    double getBackStepTriggerValue(void);
    double getMaximumInterval(void);
    void setCurrentInterval(double);
    double getCurrentInterval(void);
    void setNextParameter(Parameter*);
    Parameter* getNextParameter(void);
    void setHomotopyStatusPointer(HomotopyStatus*);

    //Methode zum Vorbereiten des Parameters auf das Homotopieverfahren
    void initiate(void);

    //Methoden zum Ausgeben (graphisch und Konsole) der Parameterwerte fuer Trace
    void printStatusBeforeStep(int, int);
    void printStatusAfterStep(int);
    void writeFile(int);

public:
    //Konstanten zur Auswahl des Verfahrens zur Schrittweitensteuerung
    static const int IntervalBisection=0;
    static const int PessimisticSteps=1;
    static const int OptimisticSteps=2;
    static const int IterationAdaptiveSteps=3;
    static const int TimeAdaptiveSteps=4;
    static const int RandomSteps=5;

    Parameter(void); //Standardkonstruktor
    virtual ~Parameter(void){}; //Standarddestruktor wg. globaler String-Streams

    //Methoden zum Festlegen der Parameterwerte und der Schrittweitensteuerung
    void setInitialValue(double);
    void setFinalValue(double);
    void setInitialInterval(double);
    void setMinimumInterval(double);
    void setMaximumInterval(double);
    void setForwardStepMethod(int);
    void setForwardStepMethod(int, int);
```

```
      void setForwardStepMethod(int, double);
      void freezeForwardSteps(void);
      void setBackStepMethod(int);
      void setBackStepMethod(int, int);
      void setBackStepMethod(int, double);

      void reset(void); //Parameter auf Startwerte zuruecksetzen

      //Vom Benutzer zu implementierende, virtuelle Methode zum
      //externen Aktualisieren des Parameters
      virtual void update(double)=0;
};

#include "Parameter.inline"

#endif
```

```
//--Aktuellen Parameterwert setzen--
inline void Parameter::setCurrentValue(double in){
   if((finalValue>initialValue && in<finalValue) ||
      (finalValue<initialValue && in>finalValue)){ //Neuer Wert kleiner als Zielwert?
      currentValue=in; //Neuen Wert setzen
      //Parameterstatus an Instanz von HomotopyStatus uebergeben
      homotopyStatus->tellParameterStatus(HomotopyStatus::Variable);
   }else{ //Neuer Wert groesser als Zielwert?
      currentValue=finalValue; //Wert auf Zielwert setzen
      currentInterval=finalValue - lastSuccessfulValue; //Interval korrigieren
      //Parameterstatus an Instanz von HomotopyStatus uebergeben
      homotopyStatus->tellParameterStatus(HomotopyStatus::FinalValueReached);
   }
   update(currentValue); //Neuen Parameterwert extern bekanntgeben
}

//--Aktuellen Parameterwert abfragen--
inline double Parameter::getCurrentValue(void){
   return currentValue;
}

//--Parameterwert bei letzter Erfolgreicher Auswertung speichern--
inline void Parameter::saveLastSuccessfulValue(void){
   lastSuccessfulValue=currentValue;
}

//--Parameterwert bei letzter Erfolgreicher Auswertung abfragen--
inline double Parameter::getLastSuccessfulValue(void){
   return lastSuccessfulValue;
}

//--Wert, der Rueckwaertsschritt ausgeloest hat speichern--
inline void Parameter::saveBackStepTriggerValue(void){
   backStepTriggerValue=currentValue;
}

//--Wert, der Rueckwaertsschritt ausgeloest hat abfragen--
inline double Parameter::getBackStepTriggerValue(void){
   return backStepTriggerValue;
}

//--Maximale Schrittweite abfragen--
inline double Parameter::getMaximumInterval(void){
   return maximumInterval;
}

//--Aktuelle Schrittweite setzen--
inline void Parameter::setCurrentInterval(double in){
      if(((in<0. && in>minimumInterval)||(in>0. && in<minimumInterval)) &&
         currentValue!=finalValue){ //Pruefen, ob Schrittweite zu klein fuer Maschinen-Praezision
```

```
            homotopyStatus->setHomotopyStatus(HomotopyStatus::PrecisionError);
        }
        if((maximumInterval>0. && in<maximumInterval) ||
            (maximumInterval<0. && in>maximumInterval)){ //Neue Schrittweite kleiner als Maximalschrittweite?
            currentInterval=in; //Schrittweite setzen
        }else{ //Neue Schrittweite groesser als Maximalschrittweite?
            currentInterval=maximumInterval; //Schrittweite anpassen
        }
}

//--Aktuelle Schrittweite abfragen--
inline double Parameter::getCurrentInterval(void){
    return currentInterval;
}

//--Zeiger auf naechsten Parameter setzen--
inline void Parameter::setNextParameter(Parameter *in){
    nextParameter=in;
}

//--Zeiger auf naechsten Parameter abfragen--
inline Parameter* Parameter::getNextParameter(void){
    return nextParameter;
}

//--Zeiger auf Instanz von HomotopyStatus zum Datenaustausch setzen--
inline void Parameter::setHomotopyStatusPointer(HomotopyStatus *in){
    homotopyStatus=in;
}

//--Trace: Ausgabe der Parameterwerte vor Schritt (graphisch und text)--
inline void Parameter::printStatusBeforeStep(int iteration, int parameter){
    //Speichern der Parameterwerte fuer GNUPlot
    parameterValues << iteration << " " << currentValue << std::endl;

    //Textausgabe in Konsole
    std::cout << "   Parameter " << parameter << ":" << std::endl
              << "        Wert zum Auswertungszeitpunkt: " << currentValue << std::endl
              << "        Altes Interval               : " << currentInterval << std::endl;
}

//--Trace: Ausgabe der Parameterwerte nach Schritt (graphisch und text)--
inline void Parameter::printStatusAfterStep(int iteration){
    //Speichern der Schrittweite fuer GNUPlot
    //if(currentValue > lastSuccessfulValue && !(homotopyStatus->solverResult)->getResult()){
    if(!(homotopyStatus->solverResult)->getResult()){
        stepWidthValues << iteration << " -" << currentInterval << std::endl;
    }else{
        stepWidthValues << iteration << " " << currentInterval << std::endl;
    }
```

```
   //Textausgabe in Konsole
   std::cout << "        Neues Interval               : " << currentInterval << std::endl
             << "        Neuer Wert                   : " << currentValue << std::endl;
}

//--Parameterwerte und Schrittweiten in Datei schreiben--
inline void Parameter::writeFile(int parameter){
   //Dateinummer aus uebergebener Parameternummer erzeugen
   std::ostringstream fileNumber;
   fileNumber << parameter;

   //Dateinamen fuer Parameterwerte erzeugen
   std::string parameterFileName("ParameterValues");
   parameterFileName+=fileNumber.str();
   parameterFileName+=".dat";

   //Parameterwerte in Datei schreiben
   std::ofstream parameterValueFile(parameterFileName.c_str());
   parameterValueFile << parameterValues.str();

   //Dateinamen fuer Schrittweiten erzeugen
   std::string stepWidthFileName("StepWidthValues");
   stepWidthFileName+=fileNumber.str();
   stepWidthFileName+=".dat";

   //Schrittweiten in Datei schreiben
   std::ofstream stepWidthFile(stepWidthFileName.c_str());
   stepWidthFile << stepWidthValues.str();
}
```

```
#include "Parameter.h"
#include "ForwardStepController.h"
#include "BackStepController.h"
#include "IntervalBisectionForwardStep.h"
#include "IntervalBisectionBackStep.h"
#include "PessimisticForwardStep.h"
#include "PessimisticBackStep.h"
#include "OptimisticForwardStep.h"
#include "OptimisticBackStep.h"
#include "RandomForwardStep.h"
#include "RandomBackStep.h"
#include "IterationAdaptiveForwardStep.h"
#include "IterationAdaptiveBackStep.h"
#include "TimeAdaptiveForwardStep.h"
#include "TimeAdaptiveBackStep.h"

//--Standardkonstruktor--
//Initialisieren aller Parameterwerte mit 0, Benutzer hat noch keine Werte festgelegt
Parameter::Parameter(void){
   initialValue=0.;
   finalValue=0.;
   currentValue=0.;
   lastSuccessfulValue=0.;
   backStepTriggerValue=0.;
   initialInterval=0.;
   setInitialIntervalCalled=false;
   minimumInterval=0.;
   setMinimumIntervalCalled=false;
   maximumInterval=0.;
   setMaximumIntervalCalled=false;
   currentInterval=0.;
   nextParameter=0;
   forwardStepsFrozen=false;
   setForwardStepMethodCalled=false;
   setBackStepMethodCalled=false;
}

//--Vorbereiten des Parameters auf das Homotopieverfahren--
void Parameter::initiate(void){
   //Start- und Maximalschrittweite und Verfahren zur Schrittweitensteuerung
   //werden auf Standardwerte gestzt, falls nicht von Benutzer festgelegt
   if(setInitialIntervalCalled){
      if(finalValue < initialValue && initialInterval >= 0.){
         setInitialInterval((finalValue - initialValue) / 2.);
         initialInterval=-initialInterval;
      }
      if(finalValue > initialValue && initialInterval <= 0.){
         setInitialInterval((finalValue - initialValue) / 2.);
      }
   }
```

```
   if(!setInitialIntervalCalled){
      setInitialInterval((finalValue - initialValue) / 2.);
      if(finalValue < initialValue){initialInterval=-initialInterval;}
   }
   if(setMinimumIntervalCalled){
      if(finalValue < initialValue && minimumInterval >= 0.){
         setMinimumInterval((finalValue - initialValue) / 10000.);
         minimumInterval=-minimumInterval;
      }
      if(finalValue > initialValue && minimumInterval <= 0.){
         setMinimumInterval((finalValue - initialValue) / 10000.);
      }
   }
   if(!setMinimumIntervalCalled){
      setMinimumInterval((finalValue - initialValue) / 10000.);
      if(finalValue < initialValue){minimumInterval=-minimumInterval;}
   }
   if(setMaximumIntervalCalled){
      if(finalValue < initialValue && maximumInterval >= 0.){
         setMaximumInterval((finalValue - initialValue) / 2.);
         maximumInterval=-maximumInterval;
      }
      if(finalValue > initialValue && maximumInterval <= 0.){
         setMaximumInterval((finalValue - initialValue) / 2.);
      }
   }
   if(!setMaximumIntervalCalled){
      setMaximumInterval((finalValue - initialValue) / 2.);
      if(finalValue < initialValue){maximumInterval=-maximumInterval;}
   }
   if(!setForwardStepMethodCalled){
      setForwardStepMethod(IntervalBisection);
   }
   if(forwardStepsFrozen){
      forwardStep->freezeSteps();
   }
   if(!setBackStepMethodCalled){
      setBackStepMethod(IntervalBisection);
   }

   //Uebergabe von Zeigern auf Parameter und Instanz von HomotopyStatus
   //an die Schrittweiten-Controller
   forwardStep->setControlledParameter(this);
   forwardStep->setHomotopyStatusPointer(homotopyStatus);
   backStep->setControlledParameter(this);
   backStep->setHomotopyStatusPointer(homotopyStatus);
}

//--Startwert setzen--
void Parameter::setInitialValue(double in){
```

```
   initialValue=in;
   currentValue=in;
   update(in); //Startwert extern setzen
}

//--Zielwert setzen--
void Parameter::setFinalValue(double in){
   finalValue=in;
}

//--Startschrittweite setzen--
void Parameter::setInitialInterval(double in){
   initialInterval=in;
   currentInterval=in;
   setInitialIntervalCalled=true; //Benutzer hat Startschrittweite festgelegt
}

//--Minimale Schrittweite setzen--
void Parameter::setMinimumInterval(double in){
   minimumInterval=in;
   setMinimumIntervalCalled=true;
}

//--Maximale Schrittweite setzen--
void Parameter::setMaximumInterval(double in){
   maximumInterval=in;
   setMaximumIntervalCalled=true; //Benutzer hat Maximalschrittweite festgelegt
}

//--Verfahren zur Schrittweitensteuerung fuer Vorwaertsschritt setzen--
void Parameter::setForwardStepMethod(int in){
   setForwardStepMethodCalled=true; //Benutzer hat Vorwaertsschritt-Variante festgelegt
   switch(in){
      case IntervalBisection:
         forwardStep=new IntervalBisectionForwardStep(2.);
         break;

      case PessimisticSteps:
         forwardStep=new PessimisticForwardStep(0.);
         break;

      case OptimisticSteps:
         forwardStep=new OptimisticForwardStep(0.);
         break;

      case IterationAdaptiveSteps:
         forwardStep=new IterationAdaptiveForwardStep(-1);
         break;

      case TimeAdaptiveSteps:
```

```
            forwardStep=new TimeAdaptiveForwardStep(1);
            break;

         case RandomSteps:
            forwardStep=new RandomForwardStep(time(0));
            break;

         default:
            forwardStep=new IntervalBisectionForwardStep(2.);
      }
}

//--Verfahren zur Schrittweitensteuerung fuer Vorwaertsschritt setzen--
//Schrittweite kann ueber <int supplement> zusaetzlich beeinflusst werden
void Parameter::setForwardStepMethod(int method, int supplement){
   setForwardStepMethodCalled=true; //Benutzer hat Vorwaertsschritt-Variante festgelegt
   switch(method){
      case IntervalBisection:
         forwardStep=new IntervalBisectionForwardStep((double)supplement);
         break;

      case PessimisticSteps:
         forwardStep=new PessimisticForwardStep((double)supplement);
         break;

      case OptimisticSteps:
         forwardStep=new OptimisticForwardStep((double)supplement);
         break;

      case IterationAdaptiveSteps:
         forwardStep=new IterationAdaptiveForwardStep(supplement);
         break;

      case TimeAdaptiveSteps:
         forwardStep=new TimeAdaptiveForwardStep(supplement);
         break;

      case RandomSteps:
         forwardStep=new RandomForwardStep(supplement);
         break;

      default:
         forwardStep=new IntervalBisectionForwardStep(2.);
   }
}

//--Verfahren zur Schrittweitensteuerung fuer Vorwaertsschritt setzen--
//Schrittweite kann ueber <double supplement> zusaetzlich beeinflusst werden
void Parameter::setForwardStepMethod(int method, double supplement){
   setForwardStepMethodCalled=true; //Benutzer hat Vorwaertsschritt-Variante festgelegt
```

```
   switch(method){
      case IntervalBisection:
         forwardStep=new IntervalBisectionForwardStep(supplement);
         break;

      case PessimisticSteps:
         forwardStep=new PessimisticForwardStep(supplement);
         break;

      case OptimisticSteps:
         forwardStep=new OptimisticForwardStep(supplement);
         break;

      case IterationAdaptiveSteps:
         forwardStep=new IterationAdaptiveForwardStep((int)supplement);
         break;

      case TimeAdaptiveSteps:
         forwardStep=new TimeAdaptiveForwardStep((int)supplement);
         break;

      case RandomSteps:
         forwardStep=new RandomForwardStep((int)supplement);
         break;

      default:
         forwardStep=new IntervalBisectionForwardStep(2.);
   }
}

//--Vorwaertsschrittweite einfrieren--
void Parameter::freezeForwardSteps(void){
   forwardStepsFrozen=true;
}

//--Verfahren zur Schrittweitensteuerung fuer Rueckwaertsschritt setzen--
void Parameter::setBackStepMethod(int in){
   setBackStepMethodCalled=true; //Benutzer hat Rueckwaertsschritt-Variante festgelegt
   switch(in){
      case IntervalBisection:
         backStep=new IntervalBisectionBackStep(2.);
         break;

      case PessimisticSteps:
         backStep=new PessimisticBackStep(0.);
         break;

      case OptimisticSteps:
         backStep=new OptimisticBackStep(0.);
         break;
```

```
      case IterationAdaptiveSteps:
         backStep=new IterationAdaptiveBackStep(-1);
         break;

      case TimeAdaptiveSteps:
         backStep=new TimeAdaptiveBackStep(1);
         break;

      case RandomSteps:
         backStep=new RandomBackStep(time(0));
         break;

      default:
         backStep=new IntervalBisectionBackStep(2.);
   }
}

//--Verfahren zur Schrittweitensteuerung fuer Rueckwaertsschritt setzen--
//Schrittweite kann ueber <int supplement> zusaetzlich beeinflusst werden
void Parameter::setBackStepMethod(int method, int supplement){
   setBackStepMethodCalled=true; //Benutzer hat Rueckwaertsschritt-Variante festgelegt
   switch(method){
      case IntervalBisection:
         backStep=new IntervalBisectionBackStep((double)supplement);
         break;

      case PessimisticSteps:
         backStep=new PessimisticBackStep((double)supplement);
         break;

      case OptimisticSteps:
         backStep=new OptimisticBackStep((double)supplement);
         break;

      case IterationAdaptiveSteps:
         backStep=new IterationAdaptiveBackStep(supplement);
         break;

      case TimeAdaptiveSteps:
         backStep=new TimeAdaptiveBackStep(supplement);
         break;

      case RandomSteps:
         backStep=new RandomBackStep(supplement);
         break;

      default:
         backStep=new IntervalBisectionBackStep(2.);
   }
```

```
}

//--Verfahren zur Schrittweitensteuerung fuer Rueckwaertsschritt setzen--
//Schrittweite kann ueber <double supplement> zusaetzlich beeinflusst werden
void Parameter::setBackStepMethod(int method, double supplement){
   setBackStepMethodCalled=true; //Benutzer hat Rueckwaertsschritt-Variante festgelegt
   switch(method){
      case IntervalBisection:
         backStep=new IntervalBisectionBackStep(supplement);
         break;

      case PessimisticSteps:
         backStep=new PessimisticBackStep(supplement);
         break;

      case OptimisticSteps:
         backStep=new OptimisticBackStep(supplement);
         break;

      case IterationAdaptiveSteps:
         backStep=new IterationAdaptiveBackStep((int)supplement);
         break;

      case TimeAdaptiveSteps:
         backStep=new TimeAdaptiveBackStep((int)supplement);
         break;

      case RandomSteps:
         backStep=new RandomBackStep((int)supplement);

      default:
         backStep=new IntervalBisectionBackStep(2.);
   }
}

//--Parameter auf Startwerte zuruecksetzen--
void Parameter::reset(void){
   setCurrentValue(initialValue); //Wert zuruecksetzen
   setCurrentInterval(initialInterval); //Schrittweite zuruecksetzen
   update(currentValue); //Wert extern aktualisieren
}
```

```
#ifndef _Result_
#define _Result_

#include <iostream>

class Result{
   //Deklaration der Friend-Klassen fuer Zugriff auf private-Methoden
   friend class Homotopy;
   friend class Parameter;
   friend class IterationAdaptiveForwardStep;
   friend class IterationAdaptiveBackStep;

   private:
      //Variablen zur Kommunikation von externem Loeser mit Homotopie
      bool solutionFound; //(Externe) Loesung gefunden?
      bool invalidProblem; //Ungueltiges Zwischenproblem?
      int iterations; //Dafuer benoetigte Iterationen
      int iterationReference; //Referenz fuer Bewertung der Iterationszahl

      //Methoden zum Zugriff auf private-Variablen
      //(private Methoden werden ausschliesslich fuer
      //das Homotopieverfahren genutzt,
      //sie stehen dem Benutzer nicht zur Verfuegung)
      bool getResult(void);
      bool getInvalidProblem(void);
      int getIterations(void);
      int getIterationReference(void);

   public:
      Result(void); //Standardkonstruktor
      void setResult(bool); //Externe Loesung ok / nicht ok uebergeben
      void setResult(bool, int); //Zusaetzlich benoetigte Iterationen uebergeben
      void setInvalidProblem(void); //Ungueltiges Zwischenproblem
};

#include "Result.inline"

#endif
```

```
//--Externe Loesung ok / nicht ok abfragen
inline bool Result::getResult(void){
   return solutionFound;
}

//--Ungueltiges Zwischenproblem abfragen--
inline bool Result::getInvalidProblem(void){
   return invalidProblem;
}

//--Von externem Loeser benoetigte Iterationen abfragen--
inline int Result::getIterations(void){
   return iterations;
}

//--Referenz fuer Bewertung benoetigter Iterationen abfragen--
inline int Result::getIterationReference(void){
   return iterationReference;
}
```

```
#include "Result.h"

//--Standardkonstruktor--
//Variablen werden mit Standardwerten belegt
Result::Result(void){
   solutionFound=true;
   invalidProblem=false;
   iterations=-1;
   iterationReference=-1;
}

//--Externe Loesung ok / nicht ok uebergeben
void Result::setResult(bool in){
   solutionFound=in;
}

//--Externe Loesung mit Iterationsanzahl uebergeben--
//Iterationsreferenz wird auf Anzahl benoetigter Iterationen fuer
//ersten Loeser-Aufruf gesetzt (wird benutzt wenn nicht von Benutzer
//angegeben);
//Minimale Iterationsanzahl ist 1 -> vermeiden von Division by Zero
void Result::setResult(bool found, int iter){
   if(iterationReference==-1){
      iterationReference=(iter<1) ? 1 : iter;
   }
   iterations=(iter<1) ? 1 : iter;
   solutionFound=found;
}

//--Ungueltiges Zwischenproblem--
//Benutzer kann Abbruch des Homotopieverfahrens selbst ausloesen
void Result::setInvalidProblem(void){
   invalidProblem=true;
}
```

```
#ifndef _HomotopyStatus_
#define _HomotopyStatus_
#include "Result.h"
class HomotopyStatus{
   private:
      int homotopyStatus; //aktueller Zustand der Homotopie-FSM
      int previousStepDirection; //letzte Schrittrichtung
      int currentSolverCycles; //Von externem Loeser benoetigte Taktzyklen
      int referenceSolverCycles; //Von externem Loeser in erstem Aufruf benoetigte Taktzyklen
      double forwardStepCounter; //Anzahl aufeinanderfolgender Vorwaertsschritte
      double backStepCounter; //Anzahl aufeinanderfolgender Rueckwaertsschritte
      bool allFinalValuesReached; //Zielwert bei allen Parametern erreicht?

   public:
      Result *solverResult; //Zeiger auf Ergebnis des externen Loesers

      //Konstaten fuer den Parameterzustand (Parameter::parameterStatus)
      static const bool FinalValueReached=true;
      static const bool Variable=false;
      //ZustÃ¤nde der Homotopie-FSM (homotopyStatus)
      static const int Start=0;
      static const int Running=1;
      static const int LastStep=2;
      static const int Finished=3;
      static const int PrecisionError=4;
      //Konstanten fuer letzte Schrittrichtung (previousStepDirection)
      static const int ForwardStep=4;
      static const int BackStep=5;

      HomotopyStatus(void); //Standardkonstruktor

      //Methoden zum Zugriff auf private-Variablen
      int getHomotopyStatus(void);
      int getPreviousStepDirection(void);
      double getForwardStepCounter(void);
      double getBackStepCounter(void);
      int getCurrentSolverCycles(void);
      int getReferenceSolverCycles(void);

      void setCurrentSolverCycles(int); //Von externem Loeser benoetigte Taktzyklen uebergeben
      void tellParameterStatus(bool); //Parameterstatus mitteile
      void setHomotopyStatus(int); //Homotopie - FSM beeinflussen
      void setSolverResultPointer(Result*); //Zeiger auf externes Loeser-Ergebnis setzen
      void updateForwardStep(void); //Homotopiestatus nach Vorwaertsschritt aktualisieren
      void updateBackStep(void); //Homotopiestatus nach Rueckwaertsschritt aktualisieren
};
#include "HomotopyStatus.inline"
#endif
```

```
//--Standardkonstruktor--
//Homotopie-FSM in Start-Zustand setzen, Variablen Standardwerte zuweisen
inline HomotopyStatus::HomotopyStatus(void){
   homotopyStatus=Start;
   allFinalValuesReached=true;
   previousStepDirection=ForwardStep;
   forwardStepCounter=1.;
   backStepCounter=1.;
   referenceSolverCycles=-1;
}

//--Von externem Loeser benoetigte Taktzylen uebergeben--
inline void HomotopyStatus::setCurrentSolverCycles(int in){
   currentSolverCycles=in;
   if(referenceSolverCycles==-1){referenceSolverCycles=in;}
}

//--Von externem Loeser benoetigte Taktzyklen abfragen--
inline int HomotopyStatus::getCurrentSolverCycles(void){
   return currentSolverCycles;
}

//--Von externem Loeser in erstem Aufruf benoetigte Taktzyklen abfragen--
inline int HomotopyStatus::getReferenceSolverCycles(void){
   return referenceSolverCycles;
}

//--Parameterstatus mitteilen--
//Pruefen, ob alle Paramter Zielwert erreicht haben
inline void HomotopyStatus::tellParameterStatus(bool in){
   allFinalValuesReached=allFinalValuesReached && in;
}

//--Aktuellen Zustand der Homotopie-FSM abfragen--
inline int HomotopyStatus::getHomotopyStatus(void){
   return homotopyStatus;
}

//--Zeiger auf externes Loeser-Ergebnis setzen--
inline void HomotopyStatus::setSolverResultPointer(Result *in){
   solverResult=in;
}

//--Letzte Schrittrichtung abfragen--
inline int HomotopyStatus::getPreviousStepDirection(void){
   return previousStepDirection;
}

//--Anzahl aufeinanderfolgender Vorwaertsschritte abfragen--
inline double HomotopyStatus::getForwardStepCounter(void){
```

```
   return forwardStepCounter;
}

//--Anzahl aufeinanderfolgender Rueckwaertsschritte abfragen--
inline double HomotopyStatus::getBackStepCounter(void){
   return backStepCounter;
}

//--Homotopiestatus nach Vorwaertsschritt aktualisieren--
inline void HomotopyStatus::updateForwardStep(void){
   previousStepDirection=ForwardStep; //Letzte Schrittrichtung auf vorwaerts setzen
   if(homotopyStatus!=PrecisionError){ //Zustand der Homotopie-FSM bestimmen
      if(homotopyStatus==Start && allFinalValuesReached){homotopyStatus=Running;}
      homotopyStatus=allFinalValuesReached ? ++homotopyStatus : Running;
   }
   allFinalValuesReached=true; //Annahme: Zielwert bei allen Parametern erreicht
   forwardStepCounter+=1.; //Zaehler fuer aufeinanderfolgende Vorwaersschritte erhoehen
   backStepCounter=1.; //Zaehler fuer aufeinanderfolgende Rueckwaertsschritte auf 1
}

//--Homotopiestatus nach Rueckwaertsschritt aktualisieren--
inline void HomotopyStatus::updateBackStep(void){
   previousStepDirection=BackStep; //Letzte Schrittrichtung auf rueckwaerts setzen
   if(homotopyStatus!=PrecisionError){ //Zustand der Homotopie-FSM bestimmen
      if(homotopyStatus==Start && allFinalValuesReached){homotopyStatus=Running;}
      homotopyStatus=allFinalValuesReached ? ++homotopyStatus : Running;
   }
   allFinalValuesReached=true; //Annahme: Zielwert bei allen Parametern erreicht
   backStepCounter+=1.; //Zaehler fuer aufeinanderfolgende Rueckwaertsschritte erhoehen
   forwardStepCounter=1.; //Zaehler fuer aufeinanderfolgende Vorwaertsschritte auf 1
}

//--Homotopie - FSM beeinflussen
inline void HomotopyStatus::setHomotopyStatus(int in){
   homotopyStatus=in;
}
```

```
#ifndef _ForwardStepController_
#define _ForwardStepController_
#include "Parameter.h"
#include "HomotopyStatus.h"
class ForwardStepController{
   public:
      Parameter *controlledParameter; //Zeiger auf gesteuerten Parameter
      HomotopyStatus *homotopyStatus; //Zeiger auf Instanz von HomotopyStatus
      bool stepsFrozen; //Vorwaertsschrittweite eingefroren?

      //Hilfsstruktur fuer eingefrorene Vorwaertsschrittweite
      struct FrozenSteps{
         double interval; //Schrittweite
         double thresholdValue; //Schwellenwert, bis zu dem Schrittweite verwendet wird
         FrozenSteps* nextEntry; //Zeiger auf naechste Instanz
         //Standardkonstruktor zum initialisieren von Variablen und Zeiger
         inline FrozenSteps(void){
            interval=0.;
            thresholdValue=0.;
            nextEntry=0;
         }
         //Eintrag am Ende der Liste hinzufuegen
         inline FrozenSteps* push(double inter, double threshold){
            FrozenSteps* temp=new FrozenSteps();
            temp->interval=inter;
            temp->thresholdValue=threshold;
            temp->nextEntry=this;
            return temp;
         }
         //Ersten Eintrag der Liste zurueckgeben
         inline FrozenSteps* pop(void){
            return this->nextEntry;
         }
      };
      FrozenSteps* frozenSteps; //Zeiger auf <struct FrozenSteps>
      ForwardStepController(void); //Standardkonstruktor
      //Methoden zum Zugriff auf Variablen
      void setControlledParameter(Parameter*);
      void setHomotopyStatusPointer(HomotopyStatus*);
      void freezeSteps(void);

      void execute(void); //Vorwaertsschritt ausfuehren
      //Virtuelle Methode zur Bestimmung der Schrittweite, die von der jeweiligen Variante zur Schrittweitensteuerung implementiert wird
      virtual void executeStep(void)=0;
};

#include "ForwardStepController.inline"
#endif
```

```
//--Standardkonstruktor--
//Vorwaertsschrittweite ist nicht eingefroren
inline ForwardStepController::ForwardStepController(void){
   stepsFrozen=false;
}
//--Zeiger auf gesteuerten Parameter setzen--
inline void ForwardStepController::setControlledParameter(Parameter *in){
   controlledParameter=in;
}
//--Zeiger auf Instanz von HomotopyStatus setzen--
inline void ForwardStepController::setHomotopyStatusPointer(HomotopyStatus *in){
   homotopyStatus=in;
}
//--Vorwaertsschrittweite einfrieren--
inline void ForwardStepController::freezeSteps(void){
   stepsFrozen=true;
}
//--Vorwaertsschritt ausfuehren--
inline void ForwardStepController::execute(void){
   //Schrittweite eingefroren und letzter Schritt rueckwaerts ->
   //aktuelle Schrittweite und aktuellen Wert als Schwellenwert in <struct FrozenSteps> eintragen
   if(stepsFrozen && homotopyStatus->getPreviousStepDirection() == HomotopyStatus::BackStep){
      frozenSteps = frozenSteps->push(controlledParameter->getCurrentInterval(), controlledParameter->getBackStepTriggerValue());
   }

   //Schrittweite eingefroren und Schwellenwert fuer aktuelle Schrittweite ueberschritten ->
   //ersten Eintrag in <struct FrozenSteps> loeschen
   if(stepsFrozen && frozenSteps!=0 && controlledParameter->getCurrentValue() > frozenSteps->thresholdValue){
      frozenSteps = frozenSteps->pop();
   }

   //Neue Schrittweite bestimmen
   if(homotopyStatus->getHomotopyStatus() != HomotopyStatus::Start){ //Wenn nicht erster Schritt
      //Interval nach Rueckwaertsschritt(en) anpassen
      if(homotopyStatus->getPreviousStepDirection() == HomotopyStatus::BackStep){
         controlledParameter->setCurrentInterval(controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue());
      }
            if(stepsFrozen && frozenSteps != 0){ //Wenn Schrittweite eingefroren und Eintrag in <struct FrozenSteps> vorhanden
         controlledParameter->setCurrentInterval(frozenSteps->interval); //Schrittweite aus <struct FrozenSteps> verwenden
      }else{ //Wenn Schrittweite nicht eingefroren oder keine Eintraege in <struct FrozenSteps>
         executeStep(); //Schrittweite mit Schrittweitensteuerung bestimmen
      }
   }

   //Wert bei letzter erfolgreicher Auswertung speichern und neuen Wert setzen
   controlledParameter->saveLastSuccessfulValue();
   controlledParameter->setCurrentValue(controlledParameter->getCurrentValue() + controlledParameter->getCurrentInterval());
}
```

```
#ifndef _BackStepController_
#define _BackStepController_

#include "Parameter.h"
#include "HomotopyStatus.h"

class BackStepController{
   public:
      Parameter *controlledParameter; //Zeiger auf gesteuerten Parameter
      HomotopyStatus *homotopyStatus; //Zeiger auf Instanz von HomotopyStatus

      BackStepController(void); //Standardkonstruktor

      //Methoden zum Zugriff auf Variablen
      void setControlledParameter(Parameter*);
      void setHomotopyStatusPointer(HomotopyStatus*);

      void execute(void); //Rueckwaertsschritt ausfuehren

      //Virtuelle Methode zur Bestimmung der Schrittweite,
      //die von der jeweiligen Variante zur
      //Schrittweitensteuerung implementiert wird
      virtual void executeStep(void)=0;
};

#include "BackStepController.inline"

#endif
```

```
//--Standardkonstruktor--
inline BackStepController::BackStepController(void){}

//--Zeiger auf gesteuerten Parameter setzen--
inline void BackStepController::setControlledParameter(Parameter *in){
   controlledParameter=in;
}

//--Zeiger auf Instanz von HomotopyStatus setzen--
inline void BackStepController::setHomotopyStatusPointer(HomotopyStatus *in){
   homotopyStatus=in;
}

//--Rueckwaertsschritt ausfuehren--
inline void BackStepController::execute(void){
   //Wert der Rueckwaertsschritt(e) ausgeloest hat speichern
   if(homotopyStatus->getPreviousStepDirection() == HomotopyStatus::ForwardStep){
      controlledParameter->saveBackStepTriggerValue();
   }

   executeStep(); //Neue Schrittweite bestimmen

   //Neuen Wert setzen
   controlledParameter->setCurrentValue(controlledParameter->getCurrentValue() - controlledParameter->getCurrentInterval());
}
```

```
#ifndef _IntervalBisectionForwardStep_
#define _IntervalBisectionForwardStep_

#include "ForwardStepController.h"

class IntervalBisectionForwardStep : public ForwardStepController{
   public:
      double supplement; //Anpassen des Intervalmultiplikators

      IntervalBisectionForwardStep(double); //Konstruktor zur Uebergabe des Multiplikators

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "IntervalBisectionForwardStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe des Multiplikators zur Berechung der Schrittweite
inline IntervalBisectionForwardStep::IntervalBisectionForwardStep(double supp){
   supplement=(supp >= 1.) ? supp : 2.; //Pruefe: Supplement >= 1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = aktuelle Schrittweite * Supplement
inline void IntervalBisectionForwardStep::executeStep(void){
   controlledParameter->setCurrentInterval(controlledParameter->getCurrentInterval() * supplement);
}
```

```
#ifndef _IntervalBisectionBackStep_
#define _IntervalBisectionBackStep_

#include "BackStepController.h"

class IntervalBisectionBackStep : public BackStepController{
   public:
      double supplement; //Anpassen des Intervalmultiplikators

      IntervalBisectionBackStep(double); //Konstruktor zur Uebergabe des Multiplikators

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <BackStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "IntervalBisectionBackStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe des Multiplikators zur Berechung der Schrittweite
inline IntervalBisectionBackStep::IntervalBisectionBackStep(double supp){
   supplement=(supp > 1.) ? supp : 2.; //Pruefe: Supplement > 1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = (aktueller Wert - Wert bei letzter erfolgreicher Auswertung) / Supplement
inline void IntervalBisectionBackStep::executeStep(void){
   controlledParameter->setCurrentInterval(
      (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) / supplement
   );
}
```

```
#ifndef _OptimisticForwardStep_
#define _OptimisticForwardStep_

#include "ForwardStepController.h"

class OptimisticForwardStep : public ForwardStepController{
   public:
      double supplement; //Anpassen des Intervalmultiplikators

      OptimisticForwardStep(double); //Konstruktor zur Uebergabe des Supplement

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "OptimisticForwardStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe des initialen Multiplikators zur Berechnung der Schrittweite
//und Bestimmung des Inkrements des Multiplikators
inline OptimisticForwardStep::OptimisticForwardStep(double supp){
   supplement=(supp >= 0.) ? supp : 0.; //Pruefe: Supplement >= 0
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = aktuelle Schrittweite * (ForwardSteps + Supplement)
inline void OptimisticForwardStep::executeStep(void){
   controlledParameter->setCurrentInterval(controlledParameter->getCurrentInterval() * (homotopyStatus->getForwardStepCounter() + supplement));
}
```

```
#ifndef _OptimisticBackStep_
#define _OptimisticBackStep_

#include "BackStepController.h"

class OptimisticBackStep : public BackStepController{
   public:
      double supplement; //Anpassen des Intervalmultiplikators

      OptimisticBackStep(double); //Konstruktor zur Uebergabe des Supplement

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "OptimisticBackStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe des initialen Multiplikators zur Berechnung der Schrittweite
//und Bestimmung des Inkrements des Multiplikators
inline OptimisticBackStep::OptimisticBackStep(double supp){
   supplement=(supp > -1.) ? supp : 0.; //Pruefe: Supplement > -1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = (aktueller Wert - Wert bei letzter erfolgreicher Auswertung) / (Backsteps + Supplement +1)
inline void OptimisticBackStep::executeStep(void){
   controlledParameter->setCurrentInterval(
      (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) /
      (homotopyStatus->getBackStepCounter() + supplement + 1.)
   );
}
```

```
#ifndef _PessimisticForwardStep_
#define _PessimisticForwardStep_

#include "ForwardStepController.h"

class PessimisticForwardStep : public ForwardStepController{
   public:
      double interval; //Referenzschrittweite
      double supplement; //Anpassen des Intervalmultiplikators

      PessimisticForwardStep(double); //Konstruktor zur Uebergabe des supplement

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "PessimisticForwardStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe des supplement zur Anpassung der Schrittweitensteuerung,
//Initialisieren des Referenzintervals
inline PessimisticForwardStep::PessimisticForwardStep(double supp){
   interval=-1.; //Referenzschrittweite mit -1 vorbelegen
   supplement=(supp>-1.) ? supp : 0.; //Pruefe: supplement > -1
}

//--Neue Schrittweite und Referenzintervall bestimmen--
//Referenzintervall = Intervall nach letztem Rueckwaertsschritt
//Neue Schrittweite = Referenzintervall * (Vorwaertsschritte + Supplement + 1) / (Vorwaertsschritte + Supplement)
inline void PessimisticForwardStep::executeStep(void){
   if(homotopyStatus->getPreviousStepDirection() == HomotopyStatus::BackStep || interval==-1.){
      interval=controlledParameter->getCurrentInterval();
   }

   controlledParameter->setCurrentInterval(
      interval * ((homotopyStatus->getForwardStepCounter() + supplement + 1.) / (homotopyStatus->getForwardStepCounter() + supplement))
   );
}
```

```
#ifndef _PessimisticBackStep_
#define _PessimisticBackStep_

#include "BackStepController.h"

class PessimisticBackStep : public BackStepController{
   public:
      double supplement; //Anpassen des Intervalmultiplikators

      PessimisticBackStep(double); //Konstruktor zur Uebergabe des supplement

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "PessimisticBackStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe des initialen Multiplikators zur Berechnung der
//Schrittweite, und Bestimmung des Inkrements des Multiplikators
inline PessimisticBackStep::PessimisticBackStep(double supp){
   supplement=(supp>-1.) ? supp : 0.; //Pruefe: supplement > -1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = (aktueller Wert - Wert bei letzter erfolgreicher Auswertung) *
//                    (Rueckwaertsschritte + Supplement) / (Rueckwaertsschritte + Supplement + 1)
inline void PessimisticBackStep::executeStep(void){
   controlledParameter->setCurrentInterval(
      (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) *
      ((homotopyStatus->getBackStepCounter() + supplement) / (homotopyStatus->getBackStepCounter() + supplement + 1))
   );
}
```

```
#ifndef _IterationAdaptiveForwardStep_
#define _IterationAdaptiveForwardStep_

#include "ForwardStepController.h"

class IterationAdaptiveForwardStep : public ForwardStepController{
   public:
      int iterationReference; //Iterationsreferenz

      IterationAdaptiveForwardStep(int); //Konstruktor zur Uebergabe einer I.referenz

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "IterationAdaptiveForwardStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe einer Iterationsreferenz
inline IterationAdaptiveForwardStep::IterationAdaptiveForwardStep(int in){
   iterationReference=(in<1) ? -1 : in; //Pruefe: Iterationsreferenz >=1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = aktuelle Schrittweite * Iterationsreferenz / im letzten Loeseraufruf benoetigte Iterationen
//Wird keine Iterationsreferenz vom Benutzer angegeben wird die Anzahl
//der Iterationen des ersten Loeseraufrufs als Referenz benutzt
inline void IterationAdaptiveForwardStep::executeStep(void){
   if(iterationReference==-1){
      iterationReference=(homotopyStatus->solverResult)->getIterationReference();
   }else{
      //Nicht Normiert
      //iterationReference=(int)((iterationReference + (homotopyStatus->solverResult)->getIterations()) * .5);
      //Normiert
      iterationReference=(int)((iterationReference + ((homotopyStatus->solverResult)->getIterations() /
                               (controlledParameter->getCurrentValue()-controlledParameter->getLastSuccessfulValue()))) * .5);
   }

   //Nicht Normiert
   /*controlledParameter->setCurrentInterval(
      controlledParameter->getCurrentInterval() *
      (1. + (double)iterationReference / (double)(homotopyStatus->solverResult)->getIterations())
   );*/
   //Normiert
   controlledParameter->setCurrentInterval(
      controlledParameter->getCurrentInterval() *
      (1. + (double)iterationReference / ((double)(homotopyStatus->solverResult)->getIterations() /
      (controlledParameter->getCurrentValue()-controlledParameter->getLastSuccessfulValue())))
   );
}
```

```
#ifndef _IterationAdaptiveBackStep_
#define _IterationAdaptiveBackStep_

#include "BackStepController.h"

class IterationAdaptiveBackStep : public BackStepController{
   public:
      int iterationReference; //Iterationsreferenz

      IterationAdaptiveBackStep(int); //Konstruktor zur Uebergabe einer I.referenz

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "IterationAdaptiveBackStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe einer Iterationsreferenz
inline IterationAdaptiveBackStep::IterationAdaptiveBackStep(int in){
   iterationReference=(in<1) ? -1 : in; //Pruefe: Iterationsreferenz >=1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = (aktueller Wert - Wert bei letzter erfolgreicher Auswertung) *
//                    (Iterationsreferenz - Benoetigte Iterationen) / Iterationsreferenz
inline void IterationAdaptiveBackStep::executeStep(void){
   if(iterationReference==-1){
      iterationReference=(homotopyStatus->solverResult)->getIterationReference();
   }else{
      //Nicht Normiert
      //iterationReference=(int)((iterationReference + (homotopyStatus->solverResult)->getIterations()) * .5);
      //Normiert
      iterationReference=(int)((iterationReference + ((homotopyStatus->solverResult)->getIterations() /
                                                      controlledParameter->getCurrentInterval())) * .5);
   }

   //Nicht Normiert
   //double iterationRatio=(double)iterationReference / (double)(homotopyStatus->solverResult)->getIterations();
   double iterationRatio=(double)iterationReference /
                         (double)((homotopyStatus->solverResult)->getIterations() / controlledParameter->getCurrentInterval());
   if(iterationRatio==0. || !(iterationRatio==iterationRatio)){
      controlledParameter->setCurrentInterval(
         (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) * .5
      );
   }else{
      controlledParameter->setCurrentInterval(
         (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) / (1. + iterationRatio)
      );
   }
}
```

```
#ifndef _TimeAdaptiveForwardStep_
#define _TimeAdaptiveForwardStep_

#include "ForwardStepController.h"

class TimeAdaptiveForwardStep : public ForwardStepController{
   public:
      int cycleReference; //Referenz zur Bewertung benoetigter Taktzyklen
      int referenceWeight; //Gewichtung der Iterationsreferenz

      TimeAdaptiveForwardStep(int); //Konstruktor zur Uebergabe der Gewichtung

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "TimeAdaptiveForwardStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe einer Iterationsreferenz
inline TimeAdaptiveForwardStep::TimeAdaptiveForwardStep(int in){
   cycleReference=-1;
   referenceWeight=(in>0) ? in : 1; //Pruefe: Gewichtung >=1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = aktuelle Schrittweite * Iterationsreferenz / im letzten Loeseraufruf benoetigte Iterationen
//Wird keine Iterationsreferenz vom Benutzer angegeben wird die Anzahl
//der Iterationen des ersten Loeseraufrufs als Referenz benutzt
inline void TimeAdaptiveForwardStep::executeStep(void){
   if(cycleReference==-1){
      cycleReference=referenceWeight * homotopyStatus->getReferenceSolverCycles();
   }else{
      //Nicht Normiert
      //cycleReference=(int)((cycleReference + homotopyStatus->getCurrentSolverCycles()) * .5);
      //Normiert
      cycleReference=(int)((cycleReference +
                            (homotopyStatus->getCurrentSolverCycles() /
                            (controlledParameter->getCurrentValue()-controlledParameter->getLastSuccessfulValue()))) * .5);
   }

   //Nicht Normiert
   /*controlledParameter->setCurrentInterval(
      controlledParameter->getCurrentInterval() * (1. + (double)cycleReference / (double)homotopyStatus->getCurrentSolverCycles())
   );*/
   //Normiert
   controlledParameter->setCurrentInterval(
      controlledParameter->getCurrentInterval() *
      (1. + (double)cycleReference / ((double)homotopyStatus->getCurrentSolverCycles() /
      (controlledParameter->getCurrentValue()-controlledParameter->getLastSuccessfulValue())))
   );
}
```

```
#ifndef _TimeAdaptiveBackStep_
#define _TimeAdaptiveBackStep_

#include "BackStepController.h"

class TimeAdaptiveBackStep : public BackStepController{
   public:
      int cycleReference; //Referenz zur Bewertung benoetigter Taktzyklen
      int referenceWeight; //Gewichtung der Iterationsreferenz

      TimeAdaptiveBackStep(int); //Konstruktor zur Uebergabe der Gewichtung

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "TimeAdaptiveBackStep.inline"

#endif
```

```
//--Konstruktor--
//Uebergabe einer Iterationsreferenz
inline TimeAdaptiveBackStep::TimeAdaptiveBackStep(int in){
   cycleReference=-1;
   referenceWeight=(in>0) ? in : 1; //Pruefe: Gewichtung >=1
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = (aktueller Wert - Wert bei letzter erfolgreicher Auswertung) *
//                    (Iterationsreferenz - Benoetigte Iterationen) / Iterationsreferenz
inline void TimeAdaptiveBackStep::executeStep(void){
   if(cycleReference==-1){
      cycleReference=referenceWeight * homotopyStatus->getReferenceSolverCycles();
   }else{
      //Nicht Normiert
      //cycleReference=(int)((cycleReference + homotopyStatus->getCurrentSolverCycles()) * .5);
      //Normiert
      cycleReference=(int)((cycleReference + (homotopyStatus->getCurrentSolverCycles() /
                                              controlledParameter->getCurrentInterval())) * .5);
   }

   //Nicht Normieret
   //double cycleRatio=(double)cycleReference / (double)homotopyStatus->getCurrentSolverCycles();
   //Normiert
   double cycleRatio=(double)cycleReference /
                     ((double)homotopyStatus->getCurrentSolverCycles()/controlledParameter->getCurrentInterval());
   if(cycleRatio==0. || !(cycleRatio==cycleRatio)){
      controlledParameter->setCurrentInterval(
         (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) * .5
      );
   }else{
      controlledParameter->setCurrentInterval(
         (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) / (1. + cycleRatio)
      );
   }
}
```

```
#ifndef _RandomForwardStep_
#define _RandomForwardStep_

#include "ForwardStepController.h"

class RandomForwardStep : public ForwardStepController{
   public:
      RandomForwardStep(int); //Konstruktor zur Uebergabe des Random-Seed

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "RandomForwardStep.inline"

#endif
```

```
//--Konstruktor--
//Initialisierung des Zufallszahlengenerators uebergebenem Random-Seed
inline RandomForwardStep::RandomForwardStep(int in){
   srandom(in);
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = maximale Schrittweite * Zufallszahl zwischen 0 und 1
inline void RandomForwardStep::executeStep(void){
   double randomStep;
   do{
      randomStep=((double)random()) / ((double)RAND_MAX);
   }while(randomStep==0.);

   controlledParameter->setCurrentInterval(controlledParameter->getMaximumInterval() * randomStep);
}
```

```
#ifndef _RandomBackStep_
#define _RandomBackStep_

#include "BackStepController.h"

class RandomBackStep : public BackStepController{
   public:
      RandomBackStep(int); //Konstruktor zur Uebergabe des Random-Seed

      //Implementierung der virtuellen Methode <executeStep> aus
      //der Superklasse <ForwardStepController> zur Bestimmung der
      //neuen Schrittweite
      void executeStep(void);
};

#include "RandomBackStep.inline"

#endif
```

```
//--Konstruktor--
//Initialisierung des Zufallszahlengenerators uebergebenem Random-Seed
inline RandomBackStep::RandomBackStep(int in){
   srandom(in);
}

//--Neue Schrittweite bestimmen--
//Neue Schrittweite = (aktueller Wert - Wert bei letzter erfolgreicher Auswertung) * Zufallszahl zwischen 0 und 1
inline void RandomBackStep::executeStep(void){
   double randomStep;
   do{
      randomStep=((double)random()) / ((double)RAND_MAX);
   }while(randomStep==0.);

   controlledParameter->setCurrentInterval(
      (controlledParameter->getCurrentValue() - controlledParameter->getLastSuccessfulValue()) * randomStep
   );
}
```

C Vergleich: SICOM und Newton - HOMPACK

Im Folgenden sollen die Laufzeiten von SICOM und HOMPACK anhand einiger Beispiele zur Nullstellenbestimmung von nichtlinearen Gleichungssystemen verglichen werden. Als externes Lösungsverfahren für SICOM wird dabei die, auch für die Anwendungsbeispiele aus 7.1 verwendete, einfache Implementierung des Newton - Verfahrens eingesetzt. Um eine Vergleichbarkeit mit HOMPACK zu gewährleisten, wird der in 7.1.3 eingeführte Abbruch des Newton - Verfahrens bei nicht gegebener (linearer) Konvergenz beibehalten. Zusätzlich werden, wie auch bei HOMPACK, unterschiedliche Toleranzen für Zwischenprobleme und Zielproblem gefordert. Während der Verfolgung der Lösungskurve, also für Werte des Homotopieparameters p im halboffenen Intervall $[0,1)$ wird allgemein eine geringere Genauigkeit der Lösung gefordert als für die Lösung des Zielproblems bei $p=1$.
Als Vergleichsmaß dient die Anzahl der Auswertungen des zu lösenden Gleichungssystems bzw. der entsprechenden Inversen der Jacobi - Matrix. Ebenfalls angegeben wird die Gesamtzahl der Homotopieschritte, sowie deren Unterteilung in Vorwärts- und Rückwärtsschritte. Bedingt durch die unterschiedlichen Vorgehensweisen von SICOM und HOMPACK kann dieser Wert allerdings nicht direkt zum Vergleich der beiden Verfahren herangezogen werden.
Zur Übersichtlichkeit der Vergleichstabellen ist die Auswahl der Schrittweitensteuerungen für SICOM auf die Intervallbisektion, als einfachste Variante und die rechenzeitadaptive Schrittweitensteuerung, die jeweils die besten Ergebnisse erzielte, beschränkt.
An dieser Stelle sei darauf hingewiesen, daß aufgrund der folgenden Beispiele keine globale Aussage über SICOM und HOMPACK im Sinne von besser / schlechter getroffen werden kann. Deutlich wird allerdings, daß SICOM durchaus eine Alternative zur Implementierung eines, an das jeweilig verwendete externe, numerische Lösungsverfahren angepaßten Homotopieverfahrens bietet.

C.1 Eindimensionales Gleichungssystem

Gegeben sei das, auch in 7.1.3 verwendete Zielproblem

$$f(x) \overset{!}{=} 0 \text{ mit}$$
$$f : \Re \to \Re \;,\quad f(x) = 2 \cdot x^3 + 17 \cdot x^2 + 6 \cdot x + 50 .$$

Die Homotopie ist sowohl für SICOM, als auch für HOMPACK definiert durch

$$h : \Re \times [0,1] \to \Re \;,\quad h(x,p) = p \cdot f(x) + (1-p) \cdot g(x) \text{ mit}$$
$$p_{start} = 0 \;,\quad p_{ziel} = 1 .$$

Das Startproblem sei, wiederum für beide Verfahren

$$g(x) \overset{!}{=} 0 \text{ mit}$$
$$g : \Re \to \Re \;,\quad g(x) = x + 5.0 .$$

Folgende Tabellen zeigen einen Vergleich von SICOM und HOMPACK. Angegeben werden jeweils die verwendete Variante der Schrittweitensteuerung, die geforderte Genauigkeit der Lösungen von Zwischenproblemen und Zielproblem, die Gesamtanzahl der zur Lösung des Zielproblems erforderlichen Homotopieschritte und der darin enthaltenen Rückwärtsschritte, sowie die Anzahl der benötigten Auswertungen des Gleichungssystems (der Homotopie).
Abgesehen von dem ODE basierten Algorithmus aus HOMPACK, bewegen sich SICOM und HOMPACK bzgl. der Anzahl der benötigten Homotopieschritte trotz der unterschiedlichen Vorgehensweisen im selben Bereich.
Vergleicht man die jeweils besten Ergebnisse, fällt auf, daß SICOM in diesem Fall weniger Homotopieschritte benötigt als HOMPACK, die Schrittweiten also dementsprechend größer sind. Da bei SICOM aufgrund des verwendeten, trivialen Prädiktors die Startnäherungen für die entstehenden Zwischenprobleme weiter von der Lösung entfernt sind als die Näherungen des hermit - kubischen Prädiktors von HOMPACK, benötigt SICOM aber trotz der geringeren Anzahl von Homotopieschritten mehr Iterationen des Newton - Verfahrens. Die Anzahl benötigter Auswertungen des zu lösenden Gleichungssystems ist daher bei SICOM größer als bei HOMPACK.

Verfahren	Toleranz $p \in [0,1) \mid p=1$	Homotopieschritte Gesamt | Rückw.	Auswertungen Gleichungssys.
SICOM IntervalBisection $s_{start} = 0.5$	$10^{-8} \mid 10^{-12}$	15 | 6	48
SICOM TimeAdaptive $s_{start} = 0.1$	$10^{-8} \mid 10^{-12}$	9 | 1	39
HOMPACK ODE	$10^{-8} \mid 10^{-12}$	126 | 125	258
HOMPACK Normal Flow	$10^{-8} \mid 10^{-12}$	15 | -	40
HOMPACK Augmented Jac.	$10^{-8} \mid 10^{-12}$	14 | -	15

C.2 Achtdimensionales Gleichungssystem

Das zu lösende Zielproblem, eines der Beispielprobleme aus HOMPACK, ist gegeben durch

$$F(x) \overset{!}{=} 0 \text{ mit}$$
$$F : \Re^8 \to \Re^8,\ x \equiv (x_1,...,x_8)^T,\ F(x) \equiv (f_1(x),...,f_8(x))^T.$$

Dabei gilt

$$f_1(x) = x_1^3 + 6 \cdot x_2 \cdot x_3 + 2 \cdot x_1 - 1$$
$$f_2(x) = 6 \cdot x_1 \cdot x_3 + x_2^4 \cdot x_5 + 3 \cdot x_2 - 1$$
$$f_3(x) = 6 \cdot x_1 \cdot x_2 + x_3 \cdot x_5 + 4 \cdot x_3 - 1$$
$$f_4(x) = x_4^3 \cdot x_8 + 2 \cdot x_4 - 1$$
$$f_5(x) = \frac{x_2^5}{5} + \frac{x_3^2}{2} + x_8 \cdot x_5 + 3 \cdot x_5 - 1$$
$$f_6(x) = x_6 \cdot x_8 + 4 \cdot x_6 - 1$$
$$f_7(x) = x_7^2 \cdot x_8^3 + 2 \cdot x_7 - 1$$
$$f_8(x) = \frac{x_4^4}{4} + \frac{x_5^2}{2} + \frac{x_6^2}{2} + x_7^3 \cdot x_8^2 + 3 \cdot x_8 - 1$$

Die Homotopie sei analog zu C.1

$$h : \Re^8 \times [0,1] \to \Re^8,\ h(x,p) = p \cdot F(x) + (1-p) \cdot G(x) \text{ mit}$$
$$p_{start} = 0,\ \ p_{ziel} = 1.$$

Als Startproblem wird entsprechend

$$G(x) \overset{!}{=} 0,$$
$$G : \Re^8 \to \Re^8,\ x \equiv (x_1,...,x_8)^T,\ G(x) \equiv (g_1(x),...,g_8(x))^T \text{ mit}$$
$$g_i = x_i - 0.5,\ \ i \in [1..8].$$

verwendet.

Unter diesen Bedingungen ergeben sich in Verbindung mit den in der Tabelle aufgeführten Toleranzen folgende Werte, wobei bei diesem Beispiel auffällt, daß sowohl die Anzahl der benötigten Homotopieschritte, als auch die der Auswertungen des zu lösenden Gleichungssystems bei SICOM geringer ist, als bei HOMPACK.
Wie auch im vorigen Beispiel werden von SICOM entsprechend größere Schrittweiten bei geringerer Genauigkeit des linearen Prädiktors erzielt. Da aber die Toleranz für die Lösungen der Zwischenprobleme im Vergleich zu C.1 relativ groß ist, werden jeweils nur wenige Iterationen des Newton - Verfahrens benötigt.

Verfahren	Toleranz $p \in [0,1)$ | $p=1$	Homotopieschritte Gesamt | Rückw.	Auswertungen Gleichungssys.
SICOM IntervalBisection $s_{start} = 0.5$	10^{-2} | 10^{-4}	3 | -	8
SICOM TimeAdaptive $s_{start} = 0.5$	10^{-2} | 10^{-4}	3 | -	8
HOMPACK ODE	10^{-2} | 10^{-4}	10 | 1	24
HOMPACK Normal Flow	10^{-2} | 10^{-4}	7 | -	10
HOMPACK Augmented Jac.	10^{-2} | 10^{-4}	5 | -	16

C.3 Fünfdimensionales Gleichungssystem

Ein weiteres, HOMPACK entnommenes Beispiel ist gegeben durch die Zielfunktion

$$F(x) \overset{!}{=} 0 \text{ mit}$$
$$F : \Re^5 \to \Re^5,\ x \equiv (x_1,...,x_5)^T,\ F(x) \equiv (f_1(x),...,f_5(x))^T,$$

$$f_1(x) = x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 - 1$$
$$f_2(x) = x_1 + 2 \cdot x_2 + x_3 + x_4 + x_5 - 6$$
$$f_3(x) = x_1 + x_2 + 2 \cdot x_3 + x_4 + x_5 - 6$$
$$f_4(x) = x_1 + x_2 + x_3 + 2 \cdot x_4 + x_5 - 6$$
$$f_5(x) = x_1 + x_2 + x_3 + x_4 + 2 \cdot x_5 - 6$$

Die Homotopie sei wieder

$$h : \Re^5 \times [0,1] \to \Re^5,\ h(x,p) = p \cdot F(x) + (1-p) \cdot G(x) \text{ mit}$$
$$p_{start} = 0,\ \ p_{ziel} = 1,$$

mit

$$G(x) \overset{!}{=} 0,$$
$$G : \Re^5 \to \Re^5,\ x \equiv (x_1,...,x_5)^T,\ G(x) \equiv (g_1(x),...,g_5(x))^T,$$

$$g_i = x_i,\ \ i \in [1..5].$$

Bei der Lösung dieses Gleichungssystems überwiegt klar der Vorteil des genaueren hermit - kubischen Prädiktors von HOMPACK. So erzielt für den Vergleich der jeweils schnellsten Varianten HOMPACK hier im Vergleich zu SICOM bessere Werte bzgl. benötigter Homotopieschritte und Auswertungen des Gleichungssystems.
Trotz der geringen Menge an Informationen, auf die SICOM die Schrittweitensteuerung aufbaut, liegen deren Ergebnisse aber noch vor dem ODE basierten Algorithmus von HOMPACK.

Verfahren	Toleranz $p \in [0,1) \mid p=1$	Homotopieschritte Gesamt | Rückw.	Auswertungen Gleichungssys.
SICOM IntervalBisection $s_{start} = 0.5$	$10^{-4} \mid 10^{-8}$	18 | 8	56
SICOM TimeAdaptive $s_{start} = 0.5$	$10^{-4} \mid 10^{-8}$	10 | 2	31
HOMPACK ODE	$10^{-4} \mid 10^{-8}$	29 | 1	62
HOMPACK Normal Flow	$10^{-4} \mid 10^{-8}$	5 | -	17
HOMPACK Augmented Jac.	$10^{-4} \mid 10^{-8}$	6 | -	11

C.4 Gleichungssystem variabler Ordnung *n*

Das Zielproblem ist gegeben durch

$$F(x) \overset{!}{=} 0 \text{ mit}$$
$$F: \Re^n \to \Re^n,\ x \equiv (x_1,...,x_n)^T,\ F(x) \equiv (f_1(x),...,f_n(x))^T.$$

Für f_i mit $i \in [1..n]$ gilt

$$f_1(x) = 3 \cdot x_1 - 2 \cdot x_1^2 - 2 \cdot x_2 + 1$$
$$f_j(x) = 3 \cdot x_j - 2 \cdot x_j^2 - x_{j-1} - 2 \cdot x_{j+1} + 1 \quad \text{für} \quad j \in [2..(n-1)]$$
$$f_n(x) = 3 \cdot x_n - 2 \cdot x_n^2 - 2 \cdot x_{n-1} + 1$$

Die Homotopie sei wieder gegeben durch

$$h: \Re^n \times [0,1] \to \Re^n,\ h(x,p) = p \cdot F(x) + (1-p) \cdot G(x) \text{ mit}$$
$$p_{start} = 0,\ \ p_{ziel} = 1$$

mit dem Startproblem

$$G(x) \overset{!}{=} 0,$$
$$G: \Re^n \to \Re^n,\ x \equiv (x_1,...,x_n)^T,\ G(x) \equiv (g_1(x),...,g_n(x))^T \text{ mit}$$
$$g_i = x_i,\ \ i \in [1..n].$$

C.4.1 n=3

Wie auch bei den vorherigen Beispielen, bewegt sich die Anzahl der zur Lösung des Zielproblems benötigten Homotopieschritte (abgesehen vom ODE basierten Algorithmus aus HOMPACK) für SICOM und HOMPACK in derselben Größenordnung.
Beim Vergleich der jeweils besten Ergebnisse bzgl. der Auswertungen des zu lösenden Gleichungssystems liegt HOMPACK aufgrund des genaueren hermit - kubischen Prädiktors trotz gleicher Anzahl von Homotopieschritten wiederum vor SICOM.

Verfahren	Toleranz $p \in [0,1) \mid p=1$	Homotopieschritte Gesamt | Rückw.	Auswertungen Gleichungssys.
SICOM IntervalBisection $s_{start} = 0.5$	$10^{-4} \mid 10^{-8}$	3 | -	11
SICOM TimeAdaptive $s_{start} = 0.25$	$10^{-4} \mid 10^{-8}$	3 | -	10
HOMPACK ODE	$10^{-4} \mid 10^{-8}$	23 | 1	48
HOMPACK Normal Flow	$10^{-4} \mid 10^{-8}$	4 | -	15
HOMPACK Augmented Jac.	$10^{-4} \mid 10^{-8}$	3 | -	4

C.4.2 n=100

Für das Gleichungssystem mit 100 Gleichungen und 100 unbekannten gilt, wie auch bei C.2, daß SICOM weniger Homotopieschritte benötigt als HOMPACK. Wieder ist allerdings aufgrund der unterschiedlichen Prädiktoren die Anzahl benötigter Auswertungen des Gleichungssystems bei HOMPACK geringer.
Setzt man die jeweils besten Ergebnisse jedoch in Relation zu C.4.1 steigt die Anzahl der Auswertungen des Gleichungssystems bei SICOM mit einem Faktor von 1.3 geringer als bei HOMPACK (Faktor 2.5).

Verfahren	Toleranz $p \in [0,1) \mid p=1$	Homotopieschritte Gesamt | Rückw.	Auswertungen Gleichungssys.
SICOM IntervalBisection $s_{start} = 0.5$	$10^{-4} \mid 10^{-8}$	5 | 1	16
SICOM TimeAdaptive $s_{start} = 0.25$	$10^{-4} \mid 10^{-8}$	4 | -	13
HOMPACK ODE	$10^{-4} \mid 10^{-8}$	33 | 1	68
HOMPACK Normal Flow	$10^{-4} \mid 10^{-8}$	9 | -	22
HOMPACK Augmented Jac.	$10^{-4} \mid 10^{-8}$	9 | -	10

C.4.3 n=1000

Der von C.4.1 nach C.4.2 erkennbare Trend der Entwicklung der insgesamt ausgeführten Auswertungen des Gleichungssystems setzt sich für die Ordnung 1000 weiter fort.
Für SICOM bleibt sowohl die Anzahl der Homotopieschritte, als auch die der erforderlichen Funktionsauswertungen im Vergleich zu C.4.2 konstant. Beide Werte steigen bei HOMPACK weiter an, so daß SICOM im Vergleich zu HOMPACK in diesem Fall nur ein sechstel der Homotopieschritte benötigt.
Die Anzahl der Auswertungen des Gleichungssystems wächst für HOMPACK, wie auch von C.4.1 zu C.4.2 wieder um den Faktor 2.5. Trotz des verwendeten, trivialen Prädiktors benötigt SICOM hier nur noch rund die Hälfte der von HOMPACK ausgeführten Auswertungen des Gleichungssystems.

Verfahren	Toleranz $p \in [0,1) \mid p=1$	Homotopieschritte Gesamt | Rückw.	Auswertungen Gleichungssys.
SICOM IntervalBisection $s_{start} = 0.5$	$10^{-4} \mid 10^{-8}$	5 | 1	16
SICOM TimeAdaptive $s_{start} = 0.25$	$10^{-4} \mid 10^{-8}$	4 | -	13
HOMPACK ODE	$10^{-4} \mid 10^{-8}$	37 | 1	76
HOMPACK Normal Flow	$10^{-4} \mid 10^{-8}$	25 | -	30
HOMPACK Augmented Jac.	$10^{-4} \mid 10^{-8}$	24 | -	25