Modellierung Nichtlinearer Systeme mit Unscharfen Regeln

Diplomarbeit

Manfred Männle

Nancy, den 1. August 1995

Referent: Prof. Dr.-Ing. U. Rembold
Korreferent: Prof. Dr. A. Richard
Betreuer: Dipl.-Ing. T. Dörsam

Als erstes habe ich die angenehme Aufgabe, all denen zu danken, die mich während meiner Arbeit in Nancy unterstützt haben:

Herrn U. Rembold, Professor am IPR[1], Universität Karlsruhe, ebenso wie Herrn M. Aubrun, Professor am CRAN[2], Université de Nancy I, für die freundliche Aufnahme in die jeweilige Forschungsgruppe,

Herrn A. Richard, Professor am CRAN, Université de Nancy I, für das interessante Thema und die fachliche Unterstützung während der gesamten Arbeit,

Herrn T. Dörsam, Wissenschaftlicher Assistent am IPR, Universität Karlsruhe, für die fruchtbaren Diskussionen und die Korrekturhinweise für diesen Bericht,

Frau Müller-Klumpp und Herrn A. Richard, die mir als Ansprechpartner des IAR[3] bei organisatorischen Fragen zur Seite standen und mir bei der Finanzierung durch ein ERASMUS-Stipendium halfen,

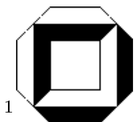
Herrn M. Weihrauch für seine hilfreichen Anregungen und das Korrekturlesen dieser Arbeit.

Für die freie Bereitstellung von Softwarewerkzeugen: Herrn R. Davies (C++ Mathematikbibliothek), der GNU Free Software Foundation (C++ Compiler, C++ Standardbibliothek und Graphikwerkzeug Gnuplot), sowie Herrn D. Knuth und L. Lamport (Textsatzsystem LaTeX),

schließlich noch allen Mitgliedern des CRAN, ganz besonders Harald Oehlmann, Didier Henry, Marius Ghetie, Luan Nguyen, Christoph Fonte, Günther Schneider und Johannes Hönig, die eine lockere Atmosphäre schafften und zu einem guten Arbeitsklima und einem angenehmen Aufenthalt in Nancy beigetragen haben.

Diese Arbeit ist im Rahmen eines Studentenaustauschprogramms des Deutsch-Französischen Instituts IAR entstanden.

---

[1] 
Institut für Prozeßrechentechnik und Robotik (IPR)
Universität Karlsruhe, Postfach 6980, D-76128 Karlsruhe
Manfred.Maennle@Informatik.Uni-Karlsruhe.De

[2]

Centre de Recherche en Automatique de Nancy (CRAN)
Université H. Poincaré Nancy I, BP 239, F-54506 Vandœuvre Cedex
maennle@cran.u-nancy.fr

[3]

Deutsch-Französisches Institut für Automation und Robotik (IAR)
Universität Karlsruhe, Kaiserstraße 12, D-76128 Karlsruhe

Erklärung

Ich versichere hiermit an Eides statt, daß ich diese Diplomarbeit selbständig in dem Zeitraum vom 1. Februar 1995 bis 1. August 1995 ohne fremde Hilfe und nur unter Zuhilfenahme der angegebenen Literaturquellen angefertigt habe.

Nancy, den 1. August 1995

Inhaltsverzeichnis

1 Einleitung **1**

1.1 State of the art 1

1.2 Problemstellung 2

1.3 Vorgehensweise 3

2 Einführung in verwandte Verfahren **4**

2.1 Übersicht 4

2.2 Regelbasierte Verfahren 5

2.3 Entstehung und Einordnung des entwickelten Verfahrens 7

3 Der unscharfe Modellierungsalgorithmus **9**

3.1 Das unscharfe Modell 10

3.2 Identifikation der Parameter 12

3.2.1 Identifikation mit der Methode der kleinsten Quadrate . . 13

3.2.2 Identifikation mit dem Algorithmus RPROP 15

3.3 Bestimmung der Modellstruktur 18

3.4 Aufwandsabschätzung 23

3.5 Gütekriterien 24

3.6 Prädiktion und Simulation 25

4 Anwendungsbeispiele **28**

4.1 Zwei Gaußglocken 28

4.2 Benchmark von Frank 38

4.3 Porosität von Papier 42

4.4 Datensatz von Box-Jenkins 45

4.5 Die Karlsruher Hand 51

5 Schlußwort **56**

A Implementierung **59**

A.1 Installation 59

A.2 Benutzung von `fzyident` 61

A.2.1 Format der Ein- und Ausgabedateien 61

A.2.2 Optionen beim Programmaufruf 63

A.2.3 Betriebsmodi 66

A.3 Benutzung von `fzy2pixl` 67

A.4 Vorgaben 68

A.5 Entwurf und Programmierung 69

B Dateiausdrucke **71**

B.1 Makefile 71

B.2 fuzzsets.h 73

B.3 fuzzsets.cc 79

B.4 fzyident.cc 90

Symbolverzeichnis

y	Ausgangsvariable (scharf)
$\hat{y}$, $\tilde{y}$	Ausgänge (scharf) von unscharfen Modellen
u, `X`	Eingangsvariablen (scharf)
N	Dimension von $\vec{u}$ (Vektor aller Eingangsvariablen)
$\mathcal{U}$, $\mathcal{Y}$	Universen
M	unscharfes Modell
R, `R`	unscharfe Regel
r	Anzahl unscharfer Regeln eines Modells M
I	Indexmenge, $I \subset I\!N$
F, `F`	unscharfe Menge
$F(u)$	Zugehörigkeitswert von u zur Menge F, $F(u) \in [0,1]$
μ	Verschiebungsparameter sigmoider Zugehörigkeitsfunktionen
σ	Steigungsparameter sigmoider Zugehörigkeitsfunktionen
$\wedge$	Konjunktion (unscharfer Klauseln)
$w_k(u)$	Zugehörigkeit von u zur Regel R_k
$v_k(u)$	normierte Zugehörigkeit von u zur Regel R_k
f_k	implizierter Wert (Konsequenz) der Regel R_k
ε	Fehler, $\varepsilon = y - \hat{y}$
T	Rechenaufwand, Anzahl von Rechenoperationen
$\mathrm{O}(\cdot)$	O-Kalkül
UC, R^2	Gütekriterien
$V(\cdot)$	Varianz
$E(\cdot)$	Erwartungswert (arithmetischer Mittelwert)

Listings, Programmausgaben und aus Dateien stammende Texte sind durchgehend in der Schriftart `typewriter` gesetzt.

Kapitel 1

Einleitung

Seit der Formulierung unscharfer[1] Mengen in [Zad65] wurden viele Anwendungen für diese Theorie entwickelt, besonders in der Regelungstechnik. Während bei der unscharfen Regelung die Modellierung von Expertenwissen im Vordergrund steht, unscharfe Regeln also „von Hand“ von einem Experten vorgegeben werden, soll in dieser Arbeit der umgekehrte Weg gegangen werden. Ausgehend von einer Menge von Datenbeispielen (Meßdaten oder simulierten Daten) werden automatisch unscharfe Regeln extrahiert, um damit komplexe Systeme zu beschreiben.

Über den vorgestellten Algorithmus ist auch ein interner Artikel des CRAN in französischer Sprache verfügbar [Mä95].

1.1 State of the art

Das Problem der Modellierung wird schon seit langem untersucht, es gibt deshalb viele verschiedene Lösungsansätze. Der erste Ansatz ist die Bildung eines mathematischen Modells [Föl92] mit Hilfe der das System beschreibenden physikalischen Gesetze. Ist dies zu schwierig oder zu aufwendig, wird versucht, mit Hilfe von Meßdaten näherungsweise ein Modell aufzustellen. Dazu gibt es wiederum viele verschiedene Ansätze. Einige davon sind:

- *Interpolation* nach Newton oder Lagrange. Sie liefert zwar eine mathematisch perfekte Funktionsapproximation, ist aber in der Praxis für die Modellierung nicht brauchbar, da sie selbst bei kleinsten Störungen in den Beispieldaten völlig vom eigentlichen System verschiedene Ergebnisse liefert.

[1] In der deutschen Literatur wird der entsprechende englische Ausdruck *fuzzy* oft unübersetzt beibehalten.

- *Klassische* lineare und nichtlineare *Modelle* sind für einen großen Teil der praktischen Probleme anwendbar. Es bleiben allerdings immer noch viele Aufgaben, bei denen lineare Modelle zu ungenau und nichtlineare Modelle nicht oder nur unter großem Aufwand berechnet werden können. Hierfür werden alternative Methoden gesucht.

- *Neuronale Netze* sind, als universelle Funktionsapproximatoren mit guten Fehlertoleranzeigenschaften, ein solcher neuer und allgemeiner Ansatz.

- *Stückweise lineare Modelle* erweitern die Möglichkeiten linearer Modelle, indem sie nichtlineare Funktionen durch mehrere lineare Teilstücke approximieren.

- *Unscharfe Modelle*, wie sie in dieser Arbeit vorgestellt werden, versuchen die Vorteile mehrerer Verfahren zu vereinigen. Sie enthalten hier lineare Teilmodelle, werden insgesamt aber durch die unscharfen Übergänge zwischen ihnen nichtlinear und können somit als Erweiterung der stückweise linearen Modelle betrachtet werden. Sie stellen, wie die Neuronalen Netze, ein allgemeines und automatisches Verfahren dar. Im Gegensatz zu den Neuronalen Netzen bleiben unscharfe Modelle aber durch die Darstellung mit unscharfen Regeln vom Menschen interpretierbar. Eine Motivation zum Einsatz unscharfer Logik für die Modellierung gibt Zadeh selbst in [Zad94].

1.2 Problemstellung

Ziel dieser Arbeit ist die Entwicklung, die Implementierung und der Test eines Algorithmus zur Modellierung nichtlinearer (statischer) MISO[2] Systeme mit unscharfen Regeln, ausgehend vom Verfahren von Sugeno [TS85, SK88]. Zudem soll ein Vergleich mit anderen Algorithmen gezogen werden. Die Aufgabe entspricht somit einer Funktionsapproximation aufgrund gegebener Stützwerte. Formal lautet sie:

- Gegeben sind m gemessene oder simulierte Eingabevektoren $\vec{u}_1, \ldots, \vec{u}_m$, $\vec{u}_i \in \mathcal{U}_1 \times \cdots \times \mathcal{U}_N \subset I\!R^N$, und m dazugehörende Ausgabewerte $y_1, \ldots, y_m$, $y_i \in \mathcal{Y} \subset I\!R$.

- Gesucht ist eine Funktionsapproximation $\hat{f} : \mathcal{U}_1 \times \cdots \times \mathcal{U}_N \mapsto \mathcal{Y}$ von y mit $\hat{y}_i = \hat{f}(\vec{u}_i) \stackrel{!}{=} y_i$ für alle $i = 1, \ldots, m$.

Die Eingaben u sind hier, wie auch bei den meisten Anwendungen in der Regelungstechnik, keine unscharfen Mengen sondern scharfe Werte (singletons).

[2]Multiple Input, Single Output.

Allerdings ist nicht eine möglichst genaue Approximation der Beispieldaten y_i gewünscht, sondern die Identifikation des zugrunde liegenden Modells. Zusätzliche Schwierigkeiten bereiten dabei verrauschte Meßdaten und nicht oder nur schwach mit dem Ausgang y korrelierte Eingänge u_j.

1.3 Vorgehensweise

Die Struktur des unscharfen Modells und die Heuristik zu deren Aufbau wurde von Sugeno [TS85, SK88] übernommen. Da die Struktur- und die Parameterbestimmung des Modells relativ unabhängig voneinander sind, bestand die Idee nun darin, einen anderen, effizienten und nichtlinearen Optimierungsalgorithmus für die Parameterbestimmung anzuwenden, um das Verfahren zu verbessern und zu beschleunigen. Nach Kapitel 2, einer Übersicht über verwandte Verfahren, behandelt Kapitel 3 ausführlich die notwendigen Herleitungen für die Anwendung dieses Optimierungsalgorithmus auf das vorliegende Problem. Die zu Versuchszwecken erstellten Programme werden in Anhang A beschrieben. In Kapitel 4 wird die Funktionsweise des Verfahrens an einem Beispiel veranschaulicht und die Leistung anhand mehrerer Beispiele untersucht. Kapitel 5 schließt diesen Bericht mit einer Zusammenfassung und einigen Schlußbemerkungen ab.

Kapitel 2

Einführung in verwandte Verfahren

Seit der Einführung unscharfer Mengen wurden verschiedene Typen unscharfer Systeme entwickelt. Allgemeines über Definitionen und Sätze der unscharfen Logik und unscharfer Systeme findet sich zum Beispiel in [Goo93, BG93, KGK93, Men95]. Eine Teilübersicht über unscharfe Modellierungsalgorithmen gibt [JS95].

2.1 Übersicht

Grob lassen sich die unscharfen Modellierungsverfahren in regelbasierte Verfahren, relationsbasierte Verfahren, Verfahren zur unscharfen linearen Regression und Verfahren zur unscharfen qualitativen Modellierung einteilen. Abbildung 2.1 gibt eine Übersicht, die keinen Anspruch auf Vollständigkeit erhebt.

unscharfe Modellierung / Identifikation

relationsbasiert	regelbasiert	u. lineare Regression	unsch. qual. Mod.
Mamdani	Takagi/Sugeno/Kang	Tanaka et al.	Fishwick
Vrba	Yager/Filev	Xizhao/Minghu	Guariso et al.
Pedrycz	Tanaka/Ye/Tanino	Lai/Chang	Shen/Leitch
Yi/Chung	Negoita/Canciu	...	Sugeno/Yasukawa
Chen/Lu/Chen	Nakamori/Ryoke		...
...	Abe/Lan		
	...		

Abbildung 2.1: Übersicht über unscharfe Modellierungsverfahren.

Relationsbasierte Verfahren haben im allgemeinen eine fest vorgegebene unscharfe Partitionierung des Eingaberaumes durch unscharfe Mengen (meist Dreiecke

oder Trapeze). Die unscharfe Relation zwischen Ein- und Ausgang kann dann mit verschiedenen Verfahren berechnet und als Relationsmatrix dargestellt werden. Ein Vergleich und Anwendungen zu den Verfahren von Mamdani, Vrba (α-cut), Pedrycz (optimiert) und Yi/Chung (max/prod) steht in [Bö94]. Ein online Verfahren von Chen/Lu/Chen wird in [CLC94] vorgestellt. Ein Nachteil relationsbasierten Verfahren ist die oft hohe Regelanzahl, die eine Interpretation durch den Menschen erschwert.

Unscharfe lineare Regressionsverfahren arbeiten mit unscharfen Zahlen und einer entsprechenden Arithmetik. Da sie mit dem in dieser Arbeit verwendeten Verfahren wenig gemeinsam haben, wird nicht weiter auf sie eingegangen. Weitere Informationen zu einigen Verfahren finden sich in: Tanaka et al. [TYT94, INT93], Xizhao/Minghu [XM92] und Lai/Chang [LC94]. Ein Vergleich mehrerer Verfahren ist in [RW94] zu finden.

Bei der *unscharfen qualitativen Modellierung* wird der Anspruch auf numerische Genauigkeit ganz aufgegeben. Es werden grobe, nur qualitative Modelle bestimmt. Diese Verfahren sind für die hier gestellte Aufgabe zu ungenau. Verfahren gibt es zum Beispiel von Fishwick [Fis91], Guariso et al. [GRW92], Shen/Leitch [SL93] oder Sugeno [SY93].

Das in dieser Arbeit vorgestellte Verfahren gehört zu den regelbasierten. Auf diese Verfahren wird in Kapitel 2.2 deshalb gesondert eingegangen.

2.2 Regelbasierte Verfahren

Regelbasierte Verfahren zeichnen sich, im Gegensatz zu den relationsbasierten Verfahren, dadurch aus, daß sie eine variable Regelzahl mit variablen unscharfen Mengen und einen festen Inferenzmechanismus haben. Sie kommen im allgemeinen mit wesentlich weniger Regeln zur Modellbeschreibung aus, da die Regeln aufgrund ihrer individuellen Form aussagekräftiger sind als bei relationalen Modellen. Die Verfahren unterscheiden sich untereinander in der Form der unscharfen Mengen und vor allem in der Art, wie die Regeln gefunden werden.

Takagi, Sugeno & Kang (TSK)

Die Struktur des TSK-Modells [TS85, SK88] wird ausführlich in Kapitel 3 behandelt, da sie für das in dieser Arbeit entwickelte Verfahren übernommen wurde. Takagi, Sugeno und Kang verwenden *Trapezfunktionen* als Zugehörigkeitsfunktionen. Die Konsequenzparameter werden mit der Methode der kleinsten Quadrate und die Prämissenparameter mit der sogenannten *Simplex-Methode* optimiert. Die Modellstruktur wird durch einen heuristischen Suchalgorithmus bestimmt, das heißt, das Verfahren teilt schrittweise den Eingaberaum immer feiner auf und

bestimmt dann für jeden Teileingaberaum ein lineares Modell. Im Unterschied zu stückweise linearen Modellen enthält das TSK-Modell aber fließende oder unscharfe Übergänge von einem Teilmodell zum anderen.

Yager & Filev

Der Algorithmus von Yager/Filev [YF93] geht auch von einer TSK-Struktur aus, benutzt aber Glockenkurven als Zugehörigkeitsfunktionen für die unscharfen Mengen. Die Optimierung der Parameter geschieht durch eine Kombination der Methode der kleinsten Quadrate und einer *Gradientenabstiegsmethode*.

Tanaka, Ye & Tanino

Bei diesem Verfahren [TYT94] wird von der Struktur eines TSK-Modells ausgegangen. Die Konsequenzparameter werden wiederum mit der Methode der kleinsten Quadrate berechnet. Zur Bestimmung der Prämissen wird hier aber ein *genetischer Algorithmus* verwendet. Dazu wird die Partitionierung des Eingaberaumes als String kodiert und mit den genetischen Methoden Reproduktion, Kreuzung und Mutation optimiert.

Negoita & Canciu

Die Methode von Negoita/Canciu benutzt im Gegensatz zu Sugeno eine *Bézier Interpolation* in der Konsequenzbildung. Für nähere Informationen hierzu siehe [NC92].

Die obigen vier Methoden gehen alle von einer initialen Modellstruktur aus, optimieren die Konsequenzparameter, passen dann die Struktur an, optimieren wieder die Konsequenzparameter, und so weiter. Die nächsten drei Methoden gehen sozusagen den umgekehrten Weg: Es wird nicht von einer Modellstruktur, das heißt einer Aufteilung des *Eingaberaumes*, ausgegangen, sondern zuerst der *Ausgaberaum* betrachtet, um damit dann die Partition des Eingaberaumes direkt herzuleiten.

Sugeno & Yasukawa

Das Verfahren von Sugeno/Yasukawa [SY93] teilt zuerst den Ausgaberaum mit Hilfe der *fuzzy c-means* Methode in Cluster auf. Die Cluster sind dabei rund (bei Verwendung der Euklidnorm L_2) oder rechteckig (bei Verwendung der Maximumsnorm L_∞). Die Zugehörigkeit eines Punktes im Raum zu einem Cluster kann hierbei unscharf sein und wird mit einem Wert aus $[0;1]$ angegeben. Diese Cluster werden auf die Achsen der Eingabevariablen projiziert und dann mit

Trapezfunktionen approximiert. Sie ergeben somit direkt die unscharfen Regeln. Diese Methode liefert im Vergleich zum TSK-Algorithmus im allgemeinen mehr Regeln trotz ihrer geringeren Genauigkeit, benötigt aber weniger Rechenaufwand.

Nakamori & Ryoke

Nakamori und Ryoke [NR94] gehen prinzipiell denselben Weg wie Sugeno und Yasukawa, verwenden jedoch eine andere Clustermethode. Sie verwenden das sogenannte „fuzzy c-varieties", um damit *hyperellipsoidale Cluster* zu erzeugen. Diese Methode ist im allgemeinen genauer, sie ist allerdings sehr komplex.

Abe & Lan

Abe und Lan stellen in [AL95] eine Methode vor, die diese komplexen und manchmal nicht gut arbeitenden Clusteringmethoden umgeht. Hier wird die Ausgabe in n Niveaus unterteilt. Jedes Niveau bildet somit auf einfache Weise einen Cluster, der aber nicht zusammenhängend zu sein braucht. Diese Cluster werden wieder auf den Eingaberaum projiziert. Hierbei werden rekursiv Überschneidungen der projizierten Cluster aufgelöst. Aus den Projektionen können dann direkt die Geltungsbereiche der unscharfen Regeln abgeleitet werden, das Niveau des zugehörigen Clusters bestimmt dann den Wert der Konsequenz. Durch die Unschärfe der unscharfen Mengen wird außerdem ein fließender Übergang zwischen den Geltungsbereichen der einzelnen Regeln erreicht.

2.3 Entstehung und Einordnung des entwickelten Verfahrens

Bei dem in dieser Arbeit entwickelten und verwendeten Verfahren wird von der Modellstruktur das TSK-Modells ausgegangen. Übernommen wird auch der heuristische Suchalgorithmus zur Strukturbestimmung (siehe Kapitel 3.3).

Die Idee, das TSK-Verfahren zu modifizieren, entspringt aus der Tatsache, daß die Strukturbestimmung und die Paramteroptimierung zwei unabhängige Probleme sind. Dabei ist die Berechnung der Konsequenzparameter ein lineares, die der Prämissenparameter ein nichtlineares Optimierungsproblem. Die Veränderung besteht darin, ein aus den Neuronalen Netzen bekanntes, effizientes Optimierungsverfahren einzusetzen. Gewählt wird hierzu RPROP[1] (siehe Kapitel 3.2.2), wegen seiner guten Laufzeiteffizienz und seiner leichten Programmierbarkeit.

[1] Resilient propagation.

RPROP ist ein Gradientenabstiegsverfahren und setzt deshalb differenzierbare Zugehörigkeitsfunktionen voraus. Yager/Filev verwenden hierzu Glockenkurven. Um allgemeinere Glockenkurven zu erhalten werden hier Sigmoidfunktionen als Zugehörigkeitsfunktionen gewählt. Eine Glockenkurve entsteht aus der Multiplikation zweier Sigmoidkurven. Durch nach außen hin ansteigende Sigmoidkurven ist auch für Werte an den Rändern der Universen immer gewährleistet, daß zumindest eine Regel anspricht. Kapitel 3 beschreibt ausführlich das verwendete Modell und enthält die Herleitungen der für die Optimierung mit RPROP nötigen Formeln.

Kapitel 3

Der unscharfe Modellierungsalgorithmus

Die Modellierung geschieht in zwei Phasen (siehe Abb. 3.1). Zuerst wird nach einer Heuristik eine neue Modellstrukturen generiert und dann deren Parameter optimiert und deren Güte untersucht. Diese Schleife wiederholt sich, bis das Ergebnis zufriedenstellend ist oder eine vorgegebene maximale Modellgröße, d.h. eine maximale Anzahl von unscharfen Regeln, erreicht wird.

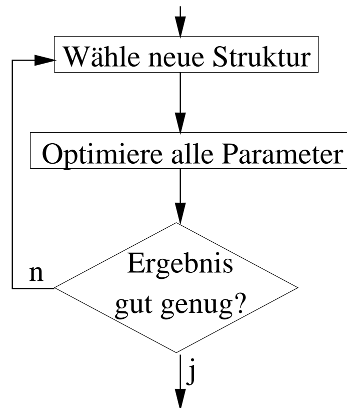


Abbildung 3.1: Schema der Modellierung.

Kapitel 3.1 gibt eine mathematische Beschreibung des verwendeten unscharfen Modells, Kapitel 3.2 behandelt die Parameteroptimierung und Kapitel 3.3 die Strukturbestimmung mittels der heuristischen Suche. Eine Abschätzung des Rechenaufwands wird in Kapitel 3.4 gegeben. In Kapitel 3.5 werden zwei mögliche Gütekriterien und in Kapitel 3.6 schließlich zwei Verwendungsmöglichkeiten der Modelle vorgestellt.

3.1 Das unscharfe Modell

Das hier verwendete unscharfe Modell entspricht im wesentlichen dem von Takagi, Sugeno und Kang in [TS85, SK88], es wird in der Literatur auch als TSK-Modell bezeichnet. Es besteht aus einer Anzahl unscharfer Regeln, von denen jede eine Prämisse und eine Konsequenz enthält. Die Konsequenz besteht aus einer (scharfen) Konstante oder einer Linearkombination der Eingänge[1]. Die Prämisse der Regel R_k ist eine Konjunktion von n_k unscharfen Klauseln der Form $u_{i_j k}$ `is` F_{jk}, d.h. sie berücksichtigt diejenigen n_k Eingänge, deren Indizes durch die Indexmenge I_k gegeben sind. Die Einführung der Indexmengen ist notwendig, da die Prämissen nicht notwendigerweise vollständig sein müssen, d.h. nicht alle Eingänge $u_j, j = 1, \ldots, N$ berücksichtigen müssen.

Definition 3.1 *Ein* **unscharfes Modell** M *besteht aus einer Menge von* r *unscharfen Regeln* R_k*,* $k = 1, \ldots, r$

$$M := \{R_1, \ldots, R_r\} \tag{3.1}$$

und c *bezeichne sein Gütekriterium.*

Definition 3.2 *Die* **Indexmenge** I_k *der Regel* R_k *ist definiert als*

$$I_k := \{i_{1k}, \ldots, i_{n_k k}\} \tag{3.2}$$

mit $i_{jk} \in \{1, \ldots, N\}$*, d.h.* $I_k \subset \{1, \ldots, N\}$.

Definition 3.3 *Eine* **unscharfe Regel** R_k *hat für* $I_k \neq \emptyset$ *die Form*

$$\texttt{if } u_{i_{1k}} \texttt{ is } F_{1k} \texttt{ and} \ldots \texttt{and } u_{i_{n_k k}} \texttt{ is } F_{n_k k} \texttt{ then } f_k = p_{0k} \underbrace{+ p_{1k} \cdot u_1 + \ldots + p_{Nk} \cdot u_N}_{optional}$$

und für $I_k = \emptyset$

$$\texttt{if TRUE then } f_k = p_{0k} \underbrace{+ p_{1k} \cdot u_1 + \ldots + p_{Nk} \cdot u_N}_{optional}, \tag{3.3}$$

wobei f_k *die Konsequenz der Regel* R_k *mit ihren Parametern* $p_{0k}, \ldots, p_{Nk}$*,* $p_{jk} \in \mathbb{R}$ *bezeichnet.*

Als Zugehörigkeitsfunktionen der unscharfen Mengen F_{jk} werden hierbei Sigmoidfunktionen verwendet. Die Parameter μ beschreiben die Verschiebungen, σ die Steigungen der Zugehörigkeitsfunktionen. Als Beispiel hierfür siehe Abbildung 3.2 unten. Sie zeigt zwei Sigmoidfunktionen mit $\mu = 0$ und $\sigma = 5$ (durchgezogen) und $\sigma = -5$ (gestrichelt).

[1] Das heißt, einer Hyperebene im Ein-/Ausgaberaum

Definition 3.4 *Die* **unscharfe Menge** F_{jk} *ist somit* $\forall j \in I_k;\ k = 1, \ldots, r;$ $i = 1, \ldots, m;\ u_{ji} \in I\!R$

$$F_{jk}(u_{ji}) \ := \ \frac{1}{1 + e^{\sigma_{jk} \cdot \left(u_{ji} - \mu_{jk}\right)}}. \tag{3.4}$$

Definition 3.5 *Die* **Zugehörigkeit** w_k *von* $\vec{u}_q$ *zur Regel* R_k *mit* $q = 1, \ldots, m;$ $\vec{u}_q \in \mathcal{U}_1 \times \cdots \times \mathcal{U}_N;\ \mathcal{U}_l \subset I\!R$ *ist*

$$w_k(\vec{u}_q) \ := \ \bigwedge_{j \in I_k} F_{jk}(u_{jq}) \tag{3.5}$$

und mit dem *Produkt* als t-norm ergibt sich

$$w_k(\vec{u}_q) \ = \ \prod_{j \in I_k} F_{jk}(u_{jq}). \tag{3.6}$$

Definition 3.6 *Die* **normierte Zugehörigkeit** $v_k(\vec{u})$ *sei*

$$v_k(\vec{u}) \ := \ \frac{w_k(\vec{u})}{\sum\limits_{i=1}^{r} w_i(\vec{u})} \qquad k = 1, \ldots, r. \tag{3.7}$$

Dabei gilt $\forall \vec{u} \in \mathcal{U}_1 \times \cdots \times \mathcal{U}_N$

$$\sum_{k=1}^{r} v_k(\vec{u}) \ = \ 1 \tag{3.8}$$

und v_k kann auch als Möglichkeitsverteilung (possibility distribution) aufgefaßt werden.

Das Modell berechnet eine Abbildung $\hat{y} : \mathcal{U}_1 \times \cdots \times \mathcal{U}_N \mapsto \mathcal{Y}$ mit $\mathcal{U}_j \subset I\!R$ und $Y \subset I\!R$. Die Modellausgabe berechnet sich mittels *Produktinferenz* (Larsen) und *gewichtetem Mittelwert* zu

$$\hat{y}(\vec{u}) \ = \ \frac{\sum\limits_{k=1}^{r} w_k(\vec{u}) \cdot f_k(\vec{u})}{\sum\limits_{k=1}^{r} w_k(\vec{u})}. \tag{3.9}$$

Abbildung 3.2 (oben) zeigt die Ausgabe des unscharfen Modells

$\mathtt{R}_1 :$ `if` u_1 `is` F_1 `then` $f_1 = 1 + 0,5 \cdot u_1$
$\mathtt{R}_2 :$ `if` u_1 `is` F_2 `then` $f_2 = 1 - 1,5 \cdot u_1$

mit $\mu_1 = \mu_2 = 0$, $\sigma_1 = 5$ und $\sigma_2 = -5$ für die unscharfen Mengen F_1 und F_2.

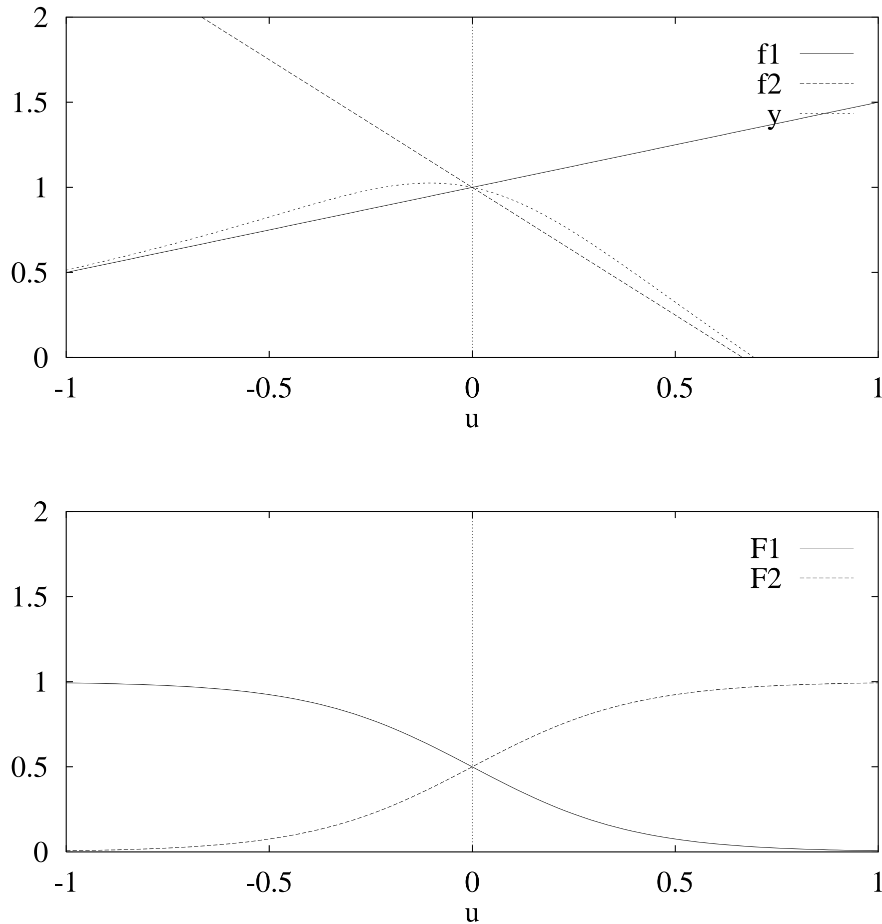


Abbildung 3.2: Eindimensionales Beispiel für $\hat{y}$ bei zwei Regeln.

Es ist zu sehen, wie die Modellausgabe $\hat{y}$ als Überlagerung der beiden linearen Teilmodelle f_1 und f_2 entsteht. Für $u < -0,5$ ist F_2 fast 0, F_1 beinahe 1. Die Ausgabe $\hat{y}$ wird deshalb für $u < -0,5$ fast nur von f_1 bestimmt und $\hat{y}$ nähert sich für noch kleinere u immer mehr an f_1 an. Für $u > 0,5$, überwiegt umgekehrt der Einfluß der zweiten Regel und $\hat{y}$ nähert sich immer mehr an f_2 an. Dazwischen, im sogenannten unscharfen Bereich, findet ein fließender Übergang zwischen den Teilmodellen f_1 und f_2 statt.

3.2 Identifikation der Parameter

Die Identifikation der Parameter geschieht durch Minimierung des über alle Beispiele ermittelten Fehlers

$$||\vec{\varepsilon}||^2 := ||\vec{y} - \vec{\hat{y}}\,||^2, \tag{3.10}$$

wobei $\vec{\hat{y}} := (\hat{y}_1, \ldots, \hat{y}_m)^T$ mit $\hat{y}_i := \hat{y}(\vec{u}_i)$ für $i = 1, \ldots, m$.

Werden die Parameter μ und σ der unscharfen Mengen konstant gehalten, so können die Konsequenzparameter p durch Aufstellen eines linearen Gleichungssystems und Lösen mit der Methode der kleinsten Quadrate wie in Kapitel 3.2.1 direkt bestimmt werden. Die Identifikation der Prämissenparameter ist schwieriger, da es sich hierbei um eine nichtlineare Optimierung handelt. Da für den Modellierungsalgorithmus eine gleichzeitige iterative Optimierung der Konsequenz- *und* Prämissenparameter erforderlich ist, wird auch für die Konsequenzparameter ein iteratives Verfahren benötigt. Die Wahl fiel auf den aus dem Gebiet der Neuronalen Netze stammenden Algorithmus RPROP, da er wesentlich laufzeiteffizienter ist als herkömmliche Gradientenabstiegsverfahren, aber trotzdem, im Gegensatz zum z.B. Levenberg-Marquardt Algorithmus [HM94], nur die erste Ableitung benötigt.

3.2.1 Identifikation mit der Methode der kleinsten Quadrate

Sei die $m \times (N+1) \cdot r$ Matrix

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1(N+1)\cdot r} \\ \vdots & & \\ a_{1m} & \ldots & a_{m(N+1)\cdot r} \end{pmatrix} \tag{3.11}$$

wie folgt aufgebaut:

$$A := \begin{pmatrix} v_1(\vec{u}_1) & v_1(\vec{u}_1)\cdot u_{11} & \ldots & v_1(\vec{u}_1)\cdot u_{N1} & \ldots & v_r(\vec{u}_1) & \ldots & v_r(\vec{u}_1)\cdot u_{N1} \\ \vdots & & & & & & & \\ v_1(\vec{u}_q) & v_1(\vec{u}_q)\cdot u_{1q} & \ldots & v_1(\vec{u}_q)\cdot u_{Nq} & \ldots & v_r(\vec{u}_q) & \ldots & v_r(\vec{u}_q)\cdot u_{Nq} \\ \vdots & & & & & & & \\ v_1(\vec{u}_m) & v_1(\vec{u}_m)\cdot u_{1m} & \ldots & v_1(\vec{u}_m)\cdot u_{Nm} & \ldots & v_r(\vec{u}_m) & \ldots & v_r(\vec{u}_m)\cdot u_{Nm} \end{pmatrix} \tag{3.12}$$

mit v_k wie in (3.7) und der Parametervektor der Konsequenzen

$$\vec{p} := \begin{pmatrix} p_{01} & p_{11} & \ldots & p_{N1} & \ldots & p_{0r} & p_{1r} & \ldots & p_{Nr} \end{pmatrix}^T, \tag{3.13}$$

so berechnet sich der Ausgabevektor $\vec{\hat{y}}$ aller Beispiele zu

$$\vec{\hat{y}} = A \cdot \vec{p}. \tag{3.14}$$

Die v_k in der Matrix A hängen von den Parametern μ und σ der unscharfen Mengen ab, die während der Identifikation der Konsequenzparameter als konstant angenommen werden.

Der Fehler $\|\vec{\varepsilon}\|^2$ hängt somit nur von den Parametern p_k ab und so gilt in dessen Minimum

$$\frac{\partial \|\vec{\varepsilon}\|^2}{\partial p_k} = 0. \tag{3.15}$$

Unter Verwendung der Euklidnorm L_2 für $\|\cdot\|$ ergibt sich für (3.10)

$$\|\vec{\varepsilon}\|^2 = \left(\sqrt{\sum_{q=1}^{m} \varepsilon_q^2}\right)^2 = \sum_{q=1}^{m}\left(y_q - \sum_{j=1}^{(N+1)\cdot r} a_{qj} p_j\right)^2, \tag{3.16}$$

und damit für (3.15) mit $k = 1, \ldots, (N+1)\cdot r$

$$\frac{\partial \|\vec{\varepsilon}\|^2}{\partial p_k} = \sum_{q=1}^{m} 2\left(y_q - \sum_{j=1}^{(N+1)\cdot r} a_{qj} p_j\right)(-a_{qk}) = 0. \tag{3.17}$$

In Matrixschreibweise gilt

$$\begin{aligned} 2\left(A\vec{p} - \vec{y}\right)^T A &= 0 \\ A^T\left(A\vec{p} - \vec{y}\right) &= 0 \\ A^T A\vec{p} &= A^T\vec{y} \\ \vec{p} &= \left(\left(A^T A\right)^{-1} A^T\right)\vec{y}. \end{aligned} \tag{3.18}$$

Dieses lineare, überbestimmte Gleichungssystem läßt mit der die Methode der kleinsten Quadrate lösen. Eine numerisch stabile Lösung liefert zum Beispiel SVD (singular value decomposition) [KL80, CLR92]. SVD berechnet drei Matrizen U, D und V, so daß $A = UDV^T$, wobei $U^T U = E$, D diagonal und V orthogonal ist. Damit ergibt sich

$$\begin{aligned} \vec{p} &= \left(VD^TU^TUDV^T\right)^{-1} VD^TU^T\vec{y} \qquad (A = UDV^T) \\ &= \left(VD^TDV^T\right)^{-1} VD^TU^T\vec{y} \qquad (U^TU = E) \\ &= \left((D^TD)VV^T\right)^{-1} VD^TU^T\vec{y} \qquad (D^TD \text{ diagonal}) \\ &= \left(D^TD\right)^{-1} VD^TU^T\vec{y} \qquad (V \text{ orthogonal} \Rightarrow VV^T = E) \\ &= V\left(D^TD\right)^{-1} D^TU^T\vec{y} \\ &= VD^{-1}\left(D^T\right)^{-1} D^TU^T\vec{y} \\ &= VD^{-1}U^T\vec{y}. \end{aligned} \tag{3.19}$$

Mit der Methode der kleinsten Quadrate werden die Parameter optimal bestimmt. Sie wird deshalb bei dem in dieser Arbeit vorgestellten Verfahren für das initiale (lineare) Modell verwendet, das dann als Ausgangspunkt für die nichtlineare Modellierung dient.

3.2.2 Identifikation mit dem Algorithmus RPROP

Der nichtlineare Optimierungsalgorithmus RPROP [Men94, ZMS+94, BR92, BR93] gehört zur Familie der Gradientenabstiegsverfahren mit variabler Schrittweite. Er ändert unabhängig für jeden zu optimierenden Parameter sein Verhalten unter Betrachtung der lokalen Topologie der Fehlerfunktion.

Bei jeder Iteration t wird für alle Beispiele $\vec{u}_i, i = 1, \ldots, m$, der Ausgang $\hat{y}(\vec{u}_i)$ und $\|\vec{\varepsilon}^t\|^2$ berechnet. Dann wird untersucht, ob sich das Vorzeichen des partiellen Gradienten ändert, d.h. ob $\frac{\partial\|\vec{\varepsilon}^{t-1}\|^2}{\partial p_j} \cdot \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} < 0$. Wenn nein, so heißt dies, daß die Richtung der Parameteränderung mit dem für jeden Parameter eigenen Δ_j^t richtig war; sie wird deshalb, je nach dem Gradienten $\frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j}$, mit positiver oder negativer Schrittweite fortgeführt. Wenn sich das Vorzeichen ändert, so ist das ein Zeichen, daß ein Extremum durchlaufen wurde; die letzte Parameteränderung war also zu groß und wird zurückgenommen. Die Parameteränderung beträgt somit

$$p_j^t = \begin{cases} p_j^{t-1} - \Delta_j^t & \text{falls } \frac{\partial\|\vec{\varepsilon}^{t-1}\|^2}{\partial p_j} \cdot \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} > 0 \text{ und } \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} > 0 \\ p_j^{t-1} + \Delta_j^t & \text{falls } \frac{\partial\|\vec{\varepsilon}^{t-1}\|^2}{\partial p_j} \cdot \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} > 0 \text{ und } \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} < 0 \\ p_j^{t-2} & \text{falls } \frac{\partial\|\vec{\varepsilon}^{t-1}\|^2}{\partial p_j} \cdot \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} < 0 \\ p_j^{t-1} & \text{sonst} \end{cases} . \qquad (3.20)$$

Gleichfalls wird je nach Änderung des Gradientenvorzeichens die Schrittweite Δ_j^t modifiziert. Falls keine Änderung auftritt, wird die Schrittweite erhöht, bei einer Änderung, also nach einem zu großen Schritt, wird sie verkleinert. Sie berechnet sich zu

$$\Delta_j^t = \begin{cases} \eta^+ \cdot \Delta_j^{t-1} & \text{falls } \frac{\partial\|\vec{\varepsilon}^{t-1}\|^2}{\partial p_j} \cdot \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} > 0 \\ \eta^- \cdot \Delta_j^{t-1} & \text{falls } \frac{\partial\|\vec{\varepsilon}^{t-1}\|^2}{\partial p_j} \cdot \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_j} < 0 \\ \Delta_j^{t-1} & \text{sonst} \end{cases} , \qquad (3.21)$$

mit $0 < \eta^- < 1 < \eta^+$. Die Unterschiede bei den Ergebnissen sind für verschiedene η^+ und η^- nicht sehr groß. Praktische Untersuchungen ergaben beste Ergebnisse für $\eta^+ = 1,2$ und $\eta^- = 0,5$ [ZMS+94].

Am Anfang werden alle Schrittweiten auf einen Anfangswert Δ_0 gesetzt (Voreinstellung ist hier 0,001). Da der Algorithmus weitgehend unabhängig von Δ_0 ist und sich $\eta^+ = 1,2$ und $\eta^- = 0,5$ als optimal herausstellten, werden diese Parameter nicht modifiziert.

Im Unterschied zu anderen Gradientenabstiegsverfahren benutzt RPROP nicht die *Beträge*, sondern nur die *Vorzeichen* der partiellen Gradienten, was den Algorithmus robust gegenüber unerwünschten Betragsschwankungen macht. Um mehr Information zu erhalten, wird die Vorzeichenänderung untersucht, die sich als verläßlicher Hinweis auf die lokale Topologie der Fehlerfunktion herausstellte.

Zur Anwendung von RPROP auf das vorliegende Optimierungsproblem werden bei jeder Iteration t für die Identifikation der Konsequenzparameter sämtliche partiellen Ableitungen $\frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_{jk}}, j = 0, \ldots, N; k = 1, \ldots, r$ benötigt. Entsprechend müssen für die Prämissenparameter alle Ableitungen $\frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial \mu_{jk}}$ und $\frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial \sigma_{jk}}$ für alle $j \in I_k, k = 1, \ldots, r$ bestimmt werden.

Ableitung nach den Konsequenzparametern

Unter Verwendung der Euklidnorm L_2 ist nach (3.10)

$$\|\vec{\varepsilon}\|^2 = \sum_{i=1}^{m} \varepsilon_i^2 = \sum_{i=1}^{m} \left(y_i - \sum_{q=1}^{r} v_q(\vec{u}_i) \cdot f_q(\vec{u}_i) \right)^2 . \tag{3.22}$$

Für die Konsequenzparameter ist mit $u_{0i} := 1$, für $k = 1, \ldots, r$ und $\forall j \in I_k$

$$\begin{aligned} \frac{\partial\|\vec{\varepsilon}^t\|^2}{\partial p_{jk}} &= (-2) \sum_{i=1}^{m} (y_i - \hat{y}_i) \sum_{q=1}^{r} v_q(\vec{u}_i) \frac{\partial f_q(\vec{u}_i)}{\partial p_{jk}} \\ &= 2 \sum_{i=1}^{m} (\hat{y}_i - y_i) v_k(\vec{u}_i) u_{ji}, \end{aligned} \tag{3.23}$$

da $\frac{\partial f_q(\vec{u}_i)}{\partial p_{jk}} = u_{ji}$.

Ableitung nach den Prämissenparametern

Für die partiellen Ableitungen ergibt sich mit (3.22) für alle $j \in I_k$ und $k = 1, \ldots, r$

$$\begin{aligned} \frac{\partial\|\vec{\varepsilon}\|^2}{\partial \mu_{jk}} &= 2 \sum_{i=1}^{m} \left((y_i - \hat{y}_i) \cdot (-1) \cdot \sum_{q=1}^{r} f_q(\vec{u}_i) \cdot \frac{\partial v_q(\vec{u}_i)}{\partial \mu_{jk}} \right) \\ &= 2 \sum_{i=1}^{m} \left((\hat{y}_i - y_i) \cdot \sum_{q=1}^{r} f_q(\vec{u}_i) \cdot \frac{\partial v_q(\vec{u}_i)}{\partial \mu_{jk}} \right) \end{aligned} \tag{3.24}$$

und analog

$$\frac{\partial\|\vec{\varepsilon}\|^2}{\partial \sigma_{jk}} = 2 \sum_{i=1}^{m} \left((\hat{y}_i - y_i) \cdot \sum_{q=1}^{r} f_q(\vec{u}_i) \cdot \frac{\partial v_q(\vec{u}_i)}{\partial \sigma_{jk}} \right) . \tag{3.25}$$

Es ergibt sich durch Ableiten von (3.7) für alle $q = 1, \ldots, r;\ q \neq k$

$$\frac{\partial v_q(\vec{u}_i)}{\partial \mu_{jk}} = \frac{-w_q(\vec{u}_i)}{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right)^2} \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \mu_{jk}} \tag{3.26}$$

und für $q = k$

$$\frac{\partial v_k(\vec{u}_i)}{\partial \mu_{jk}} = \frac{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right) \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \mu_{jk}} - w_k(\vec{u}_i) \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \mu_{jk}}}{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right)^2}$$

$$= \frac{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right) - w_k(\vec{u}_i)}{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right)^2} \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \mu_{jk}} \tag{3.27}$$

mit

$$\frac{\partial w_k(\vec{u}_i)}{\partial \mu_{jk}} = \prod_{\substack{d \in I_k \\ d \neq j}} F_{dk}(u_{di}) \cdot \frac{\partial F_{jk}(u_{ji})}{\partial \mu_{jk}} \tag{3.28}$$

und

$$\frac{\partial F_{jk}(u_{ji})}{\partial \mu_{jk}} = \frac{-e^{\sigma_{jk} \cdot \left(u_{ji} - \mu_{jk}\right)} \cdot (-\sigma_{jk})}{\left(1 + e^{\sigma_{jk} \cdot \left(u_{ji} - \mu_{jk}\right)}\right)^2}$$

$$= \sigma_{jk} \cdot F_{jk}(u_{ji}) \cdot \frac{1 + e^{\sigma_{jk} \cdot \left(u_{ji} - \mu_{jk}\right)} - 1}{1 + e^{\sigma_{jk} \cdot \left(u_{ji} - \mu_{jk}\right)}}$$

$$= \sigma_{jk} \cdot F_{jk}(u_{ji}) \cdot (1 - F_{jk}(u_{ji}))\,. \tag{3.29}$$

Entsprechend ist für alle $q = 1, \ldots, r;\ q \neq k$

$$\frac{\partial v_q(\vec{u}_i)}{\partial \sigma_{jk}} = \frac{-w_q(\vec{u}_i)}{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right)^2} \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \sigma_{jk}} \tag{3.30}$$

und für $q = k$

$$\frac{\partial v_k(\vec{u}_i)}{\partial \sigma_{jk}} = \frac{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right) \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \sigma_{jk}} - w_k(\vec{u}_i) \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \sigma_{jk}}}{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right)^2}$$

$$= \frac{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right) - w_k(\vec{u}_i)}{\left(\sum\limits_{s=1}^{r} w_s(\vec{u}_i)\right)^2} \cdot \frac{\partial w_k(\vec{u}_i)}{\partial \sigma_{jk}} \tag{3.31}$$

mit

$$\frac{\partial w_k(\vec{u}_i)}{\partial \sigma_{jk}} = \prod_{\substack{d \in I_k \\ d \neq j}} F_{dk}(u_{di}) \cdot \frac{\partial F_{jk}(u_{ji})}{\partial \sigma_{jk}} \tag{3.32}$$

und

$$\begin{aligned}
\frac{\partial F_{jk}(u_{ji})}{\partial \sigma_{jk}} &= \frac{-e^{\sigma_{jk} \cdot (u_{ji} - \mu_{jk})} \cdot (u_{ji} - \mu_{jk})}{\left(1 + e^{\sigma_{jk} \cdot (u_{ji} - \mu_{jk})}\right)^2} \\
&= (\mu_{jk} - u_{ji}) \cdot F_{jk}(u_{ji}) \cdot \frac{1 + e^{\sigma_{jk} \cdot (u_{ji} - \mu_{jk})} - 1}{1 + e^{\sigma_{jk} \cdot (u_{ji} - \mu_{jk})}} \\
&= (\mu_{jk} - u_{ji}) \cdot F_{jk}(u_{ji}) \cdot (1 - F_{jk}(u_{ji}))\,.
\end{aligned} \tag{3.33}$$

3.3 Bestimmung der Modellstruktur

Eine unscharfe Regel kann als (lineares) Teilmodell betrachtet werden, dessen Gültigkeit durch den normierten Zugehörigkeitswert $v_k(\vec{u}_i)$ nach (3.7) zur Prämisse gegeben ist. Die Parameter der unscharfen Mengen legen dann noch fest, an welchen Stellen und mit welcher Schärfe die Übergänge zwischen den Teilmodellen stattfinden.

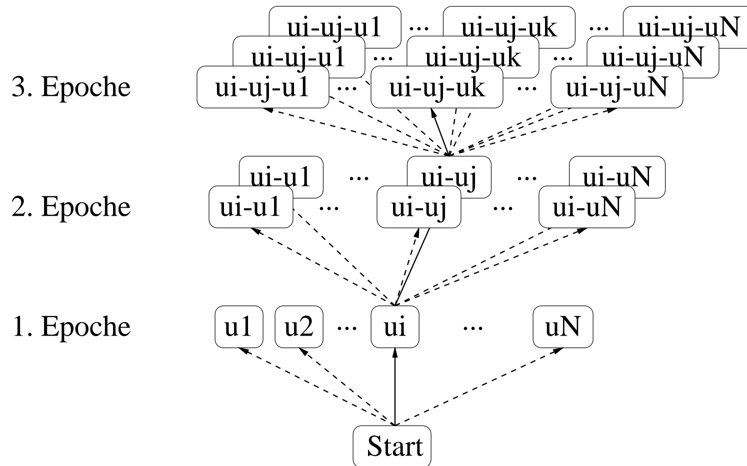


Abbildung 3.3: Bestimmung der Aufteilung des Eingaberaumes.

Die Bestimmung der optimalen Struktur ist ein kombinatorisches Problem. Deshalb wird ein heuristischer Suchalgorithmus verwendet, dessen Aufgabe es ist, eine gute, möglichst optimale Aufteilung des Eingaberaumes in Teilräume zu finden. Bei Anwendung dieser Heuristik ist das Finden der optimalen Struktur nicht

garantiert. Für die Bestimmung der optimalen Struktur ist zur Zeit kein effizienterer Algorithmus als das Betrachten aller Möglichkeiten bekannt. Durch die Heuristik wird aber in der Regel eine gute Struktur gefunden, und sie stellt somit einen Kompromiß aus Strukturgüte und Rechenaufwand dar.

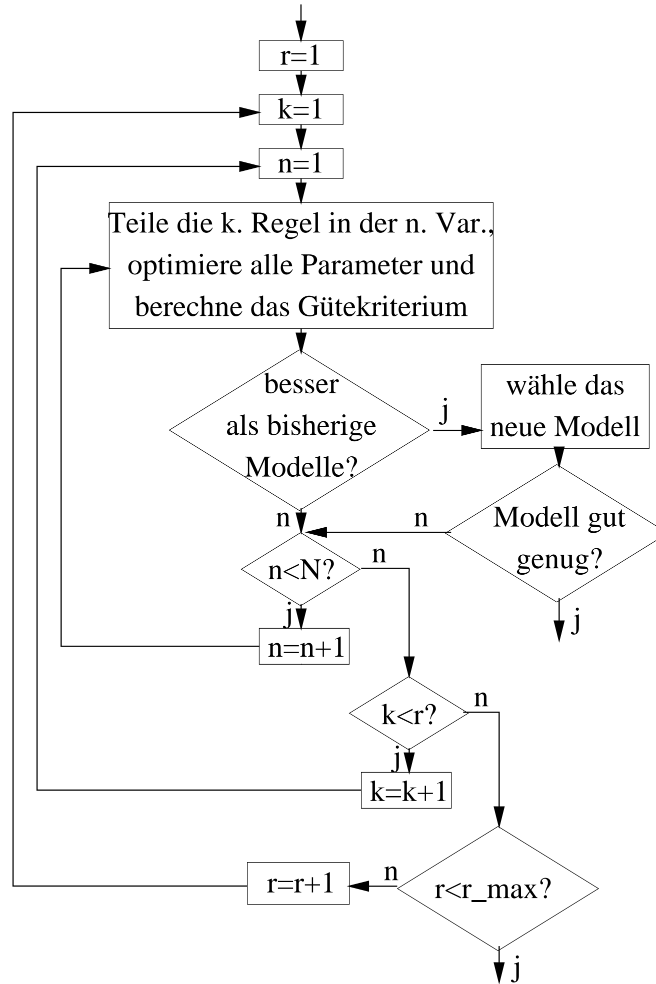


Abbildung 3.4: Heuristik zur Strukturbestimmung.

Abbildung 3.3 zeigt, wie in jeder Epoche r alle r Regeln der Epoche $r-1$ in allen N Variablen verfeinert werden. Am Start wird von einem linearen Modell (mit genau einer Regel) ausgegangen. In der ersten Epoche ergeben sich somit zwei Regeln mit jeweils einer Klausel. Unter der Annahme, daß die Verfeinerung in u_i die beste war, werden dann in der zweiten Epoche wieder für jede Regel alle N Aufteilungsmöglichkeiten, ausgehend vom Modell u_i, untersucht. Eine weitere Aufteilung in u_j führt zum Modell $u_i - u_j$ und schließlich eine Aufteilung in u_k zum Modell $u_i - u_j - u_k$. Es sind somit in jeder Epoche r genau $r \cdot N$ Modelle zu untersuchen. Im schlimmsten Fall terminiert das Verfahren für $r = r_{max}$. Es

sind dann

$$\sum_{r=1}^{r_{max}} r \cdot N = N \cdot \frac{r_{max}(r_{max}+1)}{2} \tag{3.34}$$

Modelle zu untersuchen.

Abbildung 3.4 zeigt den Programmablauf als Flußdiagramm. Der Algorithmus terminiert, wenn die Güte des bisher besten Modells ausreicht oder eine Obergrenze r_{max} von Regeln erreicht wird.

Mit den Definitionen aus Kapitel 3.1 läßt sich der Suchalgorithmus wie in Abbildung 3.5 in einer Pseudoprogrammiersprache angeben.

```
r := 1
optimiere M_1    /* mit SVD */
berechne c_1
M_opt := M_1
c_opt := c_1
while r < r_max and (not Modell gut genug) do
    r := r + 1
    c_r := ∞
    for q := 1 to r − 1 do
        for j := 1 to N do
            M_Kand := M_{r−1}
            M_Kand := M_Kand + q. Regel gespalten in der j. Variablen
            optimiere alle Parameter mit RPROP
            berechne c_Kand
            if c_Kand < c_r    /* M_Kand besser als M_r */
              M_r := M_Kand
              c_r := c_Kand
            end if
        end for
    end for
    if c_r < c_opt    /* M_r besser als M_opt */
      M_opt := M_r
      c_opt := c_r
    end if
end while
```

Abbildung 3.5: Schema der Modellierung als Pseudoprogramm.

Hierbei wird bei jedem Durchlauf der `while`-Schleife für die aktuelle Epoche r das beste Modell M_r bestimmt, indem jede schon existierende Regel in allen

Variablen einmal mehr aufgespalten und somit verfeinert wird. Über alle diese Verfeinerungen wird das optimale Modell M_r gespeichert. Ausgangspunkt der Aufspaltungen ist jeweils das optimale Modell M_{r-1} der vorherigen Epoche. Über alle Optima M_r wird schließlich das globale Optimum M_{opt} aller betrachteten Modelle bestimmt.

Verfeinern der unscharfen Regeln

Das Verfeinern einer Regel, das heißt das Aufteilen in zwei neue Regeln, geschieht durch das Kopieren der alten und anschließendes Hinzufügen jeweils einer neuen Klausel zu jeder der zwei Regeln. Die unscharfe Menge der einen neuen Klausel erhält hierbei ein positives, die der anderen ein negatives σ. Dies bewirkt, daß die zwei neuen Regeln zusammen den Bereich der alten überdecken, jede für sich aber, wegen der zusätzlichen Klausel, nur in einer Hälfte gültig ist. Der Gültigkeitsbereich der alten Regel wurde also in zwei kleinere Bereiche aufgeteilt.

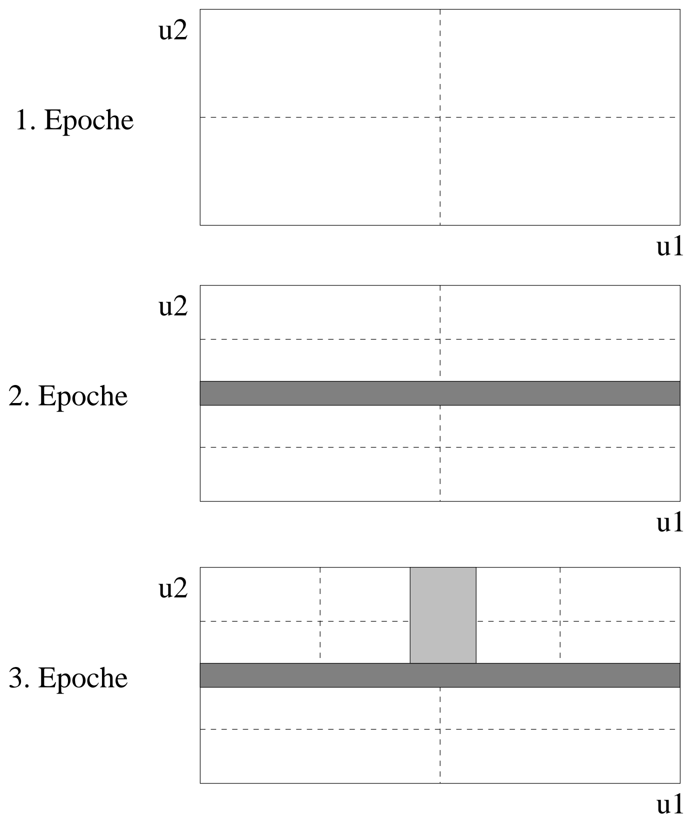


Abbildung 3.6: Beispiel einer Aufteilung eines zweidimensionalen Eingaberaumes.

Abbildung 3.6 zeigt solch eine Aufteilung eines zweidimensionalen Eingaberaumes. Das obere Bild deutet mit den zwei gestrichelten Linien die Möglichkeiten

der 1. Epoche an, das Modell durch Aufteilen in jeweils zwei Untermodelle zu verfeinern. Beim mittleren Bild wird davon ausgegangen, daß die Aufteilung in u_2 besser war. Es zeigt die nun vier Möglichkeiten in der 2. Epoche, die nun zwei vorhandenen Regeln weiter aufzuteilen. Der graue Bereich soll die unscharfe Zone des Übergangs von einem Teilmodell zum anderen verdeutlichen. Sei nun beispielsweise die Aufteilung der oberen Regel in u_1 die beste Möglichkeit der 2. Epoche. Dann ergibt sich das untere Bild, das die nun drei Teilmodelle mit ihren unscharfen Übergängen und den nun sechs möglichen Aufteilungen zeigt.

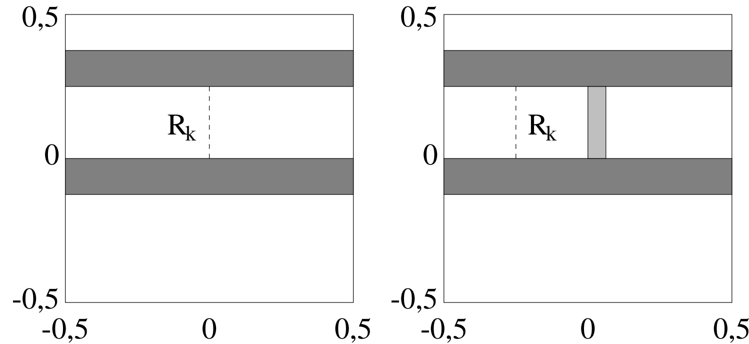


Abbildung 3.7: u_j ist in keiner (linkes Bild) bzw. in einer links gültigen Klausel (rechtes Bild) enthalten.

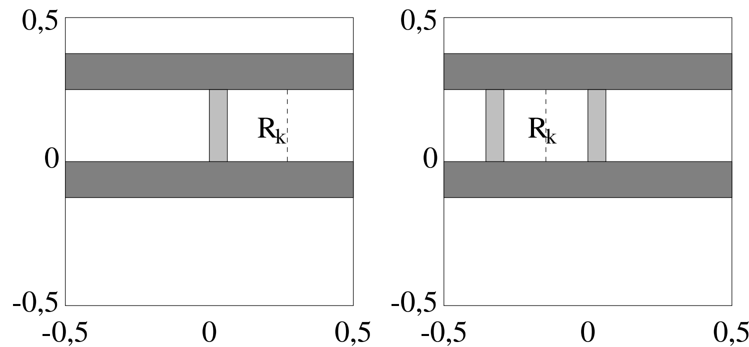


Abbildung 3.8: u_j ist in einer rechts gültigen (linkes Bild) bzw. in zwei Klauseln (rechtes Bild) enthalten.

Wird allgemein die Regel R_k in der Variablen u_j verfeinert, so sind folgende Fälle bei ihrer Teilung zu beachten:

1. Keine Klausel von R_k enthält u_j. Dann erstreckt sich R_k über das gesamte Universum $\mathcal{U}_j$ (siehe Abbildung 3.7 links). Eine der neuen Regeln erhält dann eine linksseitig gültige, die andere eine rechtsseitig gültige zusätzliche Klausel.

2. R_k enthält genau eine, links gültige Klausel. Das heißt, die Regel deckt einen Randbereich ab (Abbildung 3.7 rechts). Sie erhält dann eine zweite, rechtsseitige Klausel. Bei der neuen Regel, die als Kopie von R_k initialisiert ist, wird die u_j enthaltende Klausel gestrichen und durch eine nach außen verschobene links gültige Klausel ersetzt. Das μ der neuen unscharfen Menge wird dabei durch den Mittelwert aus dem alten μ und dem Randwert $-0,5$ initialisiert[2].

3. R_k enthält genau eine, rechts gültige Klausel. Das heißt, die Regel deckt den anderen Randbereich ab (Abbildung 3.8 links). Dieser Fall wird genau spiegelbildlich zum obigen Fall behandelt.

4. In R_k finden sich zwei u_j enthaltende Klauseln. Dies sind zwangsläufig eine rechts- und eine linksseitig gültige. Die Regel bezieht sich auf einen inneren Bereich des Eingaberaums (Abbildung 3.8 rechts). Bei der alten Regel wird die linksseitige Klausel, bei der neuen die rechtsseitige, gelöscht und durch eine neue, sich in die Mitte des Bereichs befindende Klausel ersetzt. Somit haben danach beide Regeln wieder jeweils eine rechts- und eine linksseitig gültige Klausel.

3.4 Aufwandsabschätzung

Bei jeder einzelnen Modelluntersuchung überwiegt der Aufwand durch die Optimierung mit RPROP den Aufwand für alles Sonstige, zum Beispiel das Kopieren der Modelle oder die Berechnung der Modellgüte. Die Optimierung eines Modells mit r Regeln, Eingabedimension N und m Beispielen benötigt $\mathrm{O}(rNmI)$ Schritte, wenn I die maximale[3] Zahl für RPROP-Iterationen ist. Es ergibt sich damit unter Verwendung von (3.34) ein von r_{max}, N, m und I abhängiger *Gesamtaufwand* T von maximal

$$\begin{aligned} T(r_{max}, N, m, I) &= \mathrm{O}\left(\sum_{r=1}^{r_{max}} r \cdot N \cdot (rNmI)\right) \\ &= \mathrm{O}\left(N^2 mI \cdot \frac{r_{max}(r_{max}+1)(r_{max}+2)}{6}\right) \\ &= \mathrm{O}\left(r_{max}^3 N^2 mI\right) \end{aligned} \tag{3.35}$$

Schritten. Zur Definition und einigen Eigenschaften des O-Kalküls siehe auch [CLR92] Seite 26 ff.

[2] Um diese Initialisierung sinnvoll zu machen und um die anschließende Parameteroptimierung zu erleichtern, sollten die Eingabewerte auf das Intervall $[-0,5; 0,5]$ normiert sein.

[3] In praktischen Versuchen zeigte sich $I = 100$ als ausreichend.

Bemerkenswert ist der Faktor r_{max}^3, das heißt der Aufwand wächst kubisch in der Anzahl der benötigten unscharfen Regeln. Die betrachteten Beispiele in Kapitel 4 zeigen, daß sich dies in der Praxis jedoch nicht sehr negativ auswirkt, da meist nur wenige Regeln (im Bereich von 5–15) gebraucht werden. Positiv ist das nur lineare Anwachsen des Aufwands in der Datenanzahl m, da besonders für Probleme mit vielen zu bestimmenden Parametern umfangreiche Datenmengen gebraucht werden. Der quadratische Anstieg in N wird durch die Heuristik zur Strukturbestimmung erreicht. Durch eine noch strengere Heuristik, die zum Beispiel mehrfach nicht als wichtig erachtete Eingänge nicht weiter untersucht, läßt sich dieser Faktor noch weiter verkleinern. Dieser Weg wurde hier jedoch nicht eingeschlagen. Beispiele in Kapitel 4 zeigen auch, daß manchmal auch wichtige Eingänge erst spät (beispielsweise erst in der siebten Regel) berücksichtigt werden. Dies gilt besonders dann, wenn viele ungefähr gleich wichtige Eingänge vorhanden sind.

3.5 Gütekriterien

Das Gütekriterium bestimmt den Zeitpunkt der Terminierung des Algorithmus. Terminiert wird, wenn die Erhöhung der Regelanzahl r keine Verbesserung im Sinne des Gütekriteriums mehr bewirkt. Eine andere Möglichkeit ist, eine maximale Regelanzahl r_{max} vorzugeben und ohne Berücksichtigung eines Kriteriums durchzurechnen, bis $r = r_{max}$ erreicht wird.

UC

Das Kriterium UC („unbiased criterion“) wird unter anderem von Sugeno et al. in [TS85, SK88] verwendet. Hierbei wird die Menge der Meßbeispiele in zwei etwa gleich große Mengen A und B aufgeteilt und damit werden zwei Modelle $\hat{M}$ und $\tilde{M}$ berechnet. Die Idee ist, daß, wenn unabhängig von der Datenmenge das wahre Modell identifiziert wird, die beiden Modelle auch bei beiden Mengen gleiche Ausgaben haben und UC minimal wird. Andererseits, wenn zum Beispiel Rauschen modelliert wird, so werden beide Modelle unterschiedlich und UC steigt an. Zur Gütebestimmung wird also für jede Beispielmenge die Ausgabe der beiden Modelle berechnet und verglichen. Somit ist UC definiert als

$$UC = \sqrt{\sum_{i=1}^{n_A} (\hat{y}_i^A - \tilde{y}_i^A)^2 + \sum_{i=1}^{n_B} (\hat{y}_i^B - \tilde{y}_i^B)^2}, \tag{3.36}$$

wobei n_A die Anzahl der Beispiele in A, $\hat{y}_i^A$ die Ausgabe des auf A trainierten Modells für A und $\hat{y}_i^B$ des auf A trainierten Modells für B ist. UC ist zu minimieren, das Optimum wird für UC=0 erreicht.

Das Kriterium UC führte bei dem hier vorgestellten unscharfen Modellierungsalgorithmus zu nicht befriedigenden Ergebnissen, so daß ein anderes gewählt werden mußte.

$\mathbf{R^2}$

Das Kriterium R^2 wie es auch in [Fra95] zur Gütebestimmung verwendet wird ist definiert als

$$R^2 = 1 - \frac{V(\varepsilon)}{V(y)} \tag{3.37}$$

$$= 1 - \frac{\sum_{i=1}^{n}(\varepsilon_i - E(\varepsilon_i))^2}{\sum_{i=1}^{n}(y_i - E(y_i))^2},$$

mit der Varianz $V(\cdot)$, dem Erwartungswert (arithmetischen Mittelwert) $E(\cdot)$ und dem Fehler $\varepsilon_i = y_i - \hat{y}_i$.

Um dennoch, ähnlich wie bei UC, das Eintrainieren von zum Beispiel Rauschen zu erkennen, wird wie folgt verfahren: Die Beispieldatenmenge wird wieder in zwei Mengen A und B aufgeteilt. Mit den Daten aus A werden (mit RPROP) die Parameter des Modells bestimmt. Danach wird R^2 *für B berechnet*. Falls mit A das wahre Modell identifiziert wurde, so sinkt auch der Fehler und damit wächst R^2 auch für B; falls das Rauschen in A modelliert wird, so sinkt R^2 bei B. R^2 gilt es zu maximieren, es kann bestenfalls gleich 1 werden.

3.6 Prädiktion und Simulation

Das unscharfe Modell ist zur allgemeinen Funktionsapproximation gedacht und ist somit in erster Linie für die Identifikation statischer Systeme geeignet. Es kann aber auch zur Modellierung dynamischer Systeme herangezogen werden. Für die Verarbeitung im Rechner wird dabei immer die diskrete Darstellung verwendet.

Abbildung 3.9 zeigt, wie ein Modell zur Prädiktion eingesetzt wird. In [NP90] wird diese Anordnung als Reihen-Parallel-Modell bezeichnet. Hier trifft das Modell eine Vorhersage (Prädiktion) des Ausgangs im nächsten Schritt t. Für einen Prozeß der Ordnung n stellt das unscharfe Modell somit die Abbildung

$$\hat{y}_t = \hat{f}(y_{t-1}, \ldots, y_{t-n}, u_{t-1}, \ldots, u_{t-n}) \tag{3.38}$$

dar.

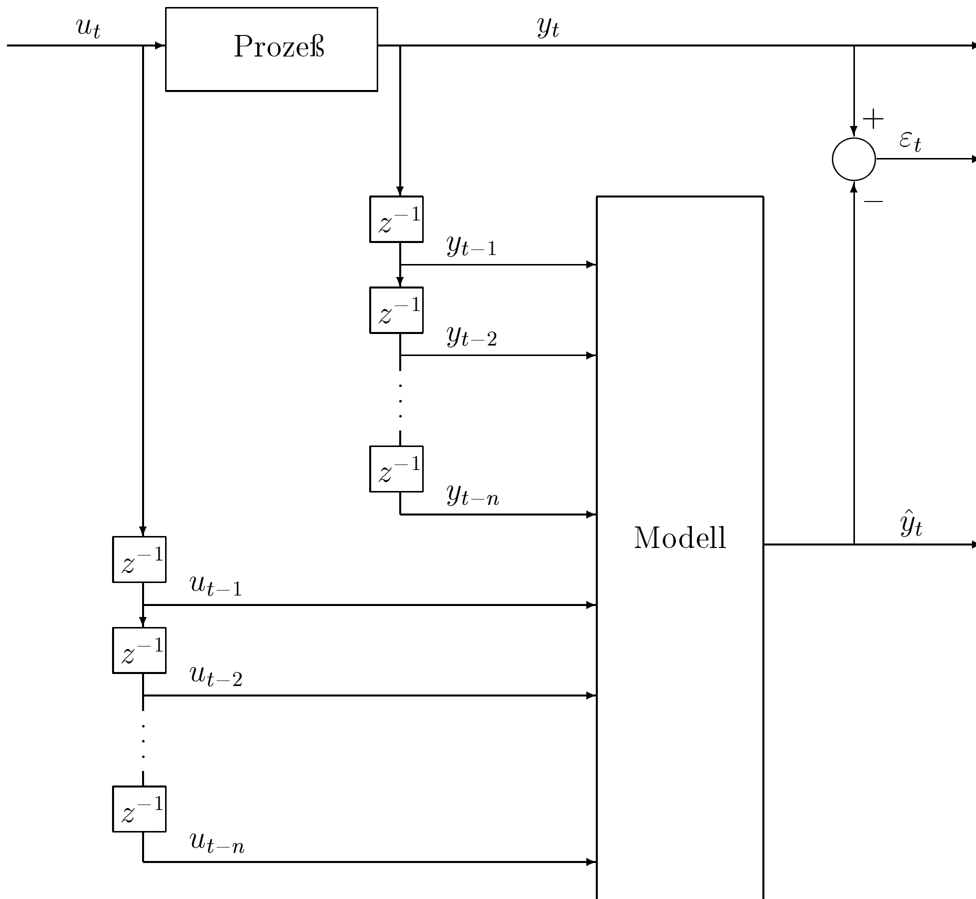


Abbildung 3.9: Prädiktion mit einem (unscharfen) Modell.

Auch bei der Identifikation wird, um RPROP verwenden zu können, vom Serien-Parallel-Modell ausgegangen, wobei zu jedem Zeitpunkt der aktuelle Fehler ε_t zur Parameteroptimierung verwendet wird.

Um einen Prozeß zu simulieren wird das unscharfe Modell wie in Abbildung 3.10 eingesetzt. In [NP90] heißt diese Anordnung Parallel-Modell. Ein gut identifiziertes Modell verhält sich dann bei gegebenem Startvektor y_0 und gegebenen Prozeßeingaben $u_{t-1}, \ldots, u_{t-n}$ wie der identifizierte Prozeß, auch wenn es wie in dieser Arbeit nach dem Serien-Parallel-Modell identifiziert wird. Das unscharfe Modell bewirkt somit die Abbildung

$$\hat{y}_t = \hat{f}(\hat{y}_{t-1}, \ldots, \hat{y}_{t-n}, u_{t-1}, \ldots, u_{t-n}). \tag{3.39}$$

Durch die Fehlerfortpflanzung, die hier durch Rückkopplung des Ausgangs an den Eingang entsteht, können sich allerdings auch kleine Fehler schnell verstärken.

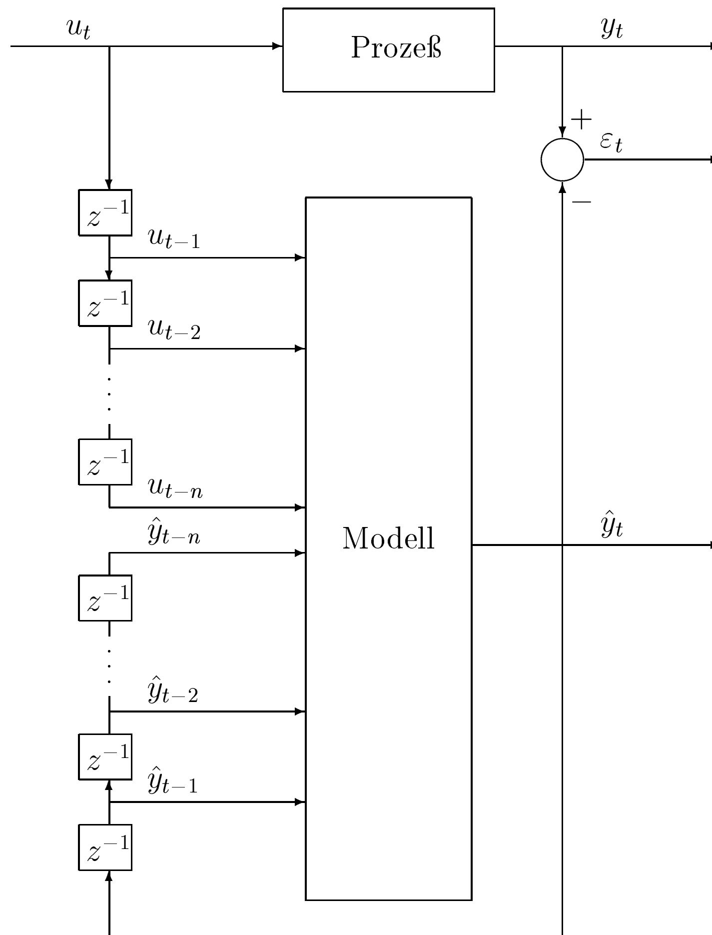


Abbildung 3.10: Simulation mit einem (unscharfen) Modell.

Kapitel 4

Anwendungsbeispiele

Die vorgestellten Anwendungen des entwickelten Verfahrens haben drei Ziele:

1. Veranschaulichung und Test des Algorithmus.
2. Untersuchung der Leistung und Vergleich mit anderen Verfahren.
3. Anwendung auf reale Meßdaten.

Zur Veranschaulichung dient Kapitel 4.1. Kapitel 4.2 untersucht die Leistungsfähigkeit im Vergleich mit vier anderen nichtlinearen Modellierungsalgorithmen. Die Kapitel 4.3, 4.4 behandeln Beispiele mit realen, Kapitel 4.5 mit simulierten Meßdaten.

4.1 Zwei Gaußglocken

Um das Programm zu testen und um das Verfahren anschaulich darstellen zu können, wurden als erstes zwei Gaußglocken im dreidimensionalen Raum als zu approximierende, nichtlineare Funktion gewählt. Es sind $u_1, u_2 \in [-0,5; 0,5]$. Als Eingangsdaten dienen die 25x25 äquidistanten und normierten Stützstellen der Funktion, wie sie in Abbildung 4.1 zu sehen sind.

Um die Ausgabe leichter interpretieren zu können, wurde hier die Modellierung mit Konstanten als Regelkonsequenzen (siehe (3.3)) verwendet. Die einzelnen Teilmodelle können nur zur u_1–u_2 Ebene parallele Ebenen bilden. Alle andere Strukturen der Originalfunktion müssen hier also durch unscharfe Übergänge zwischen den Niveaus nachgebildet werden.

In den Abbildungen 4.2 bis 4.8 wird deutlich, wie das Verfahren durch Bildung von Teilmodellen und deren Übergänge immer genauer die Originalfunktion nachbildet. Die Prädiktion mit einer Regel (Abbildung 4.2) ergibt den Mittelwert der

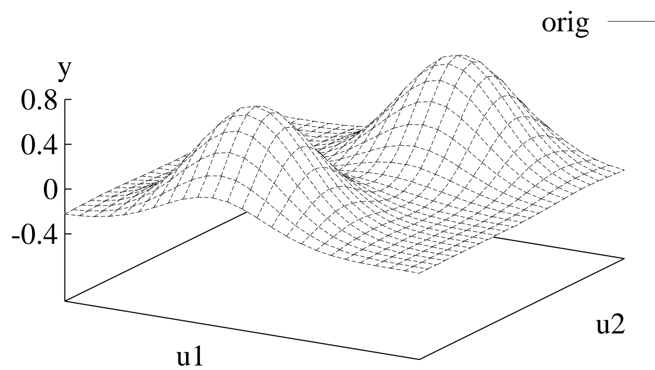


Abbildung 4.1: Originalfunktion: Zwei Gaußglocken.

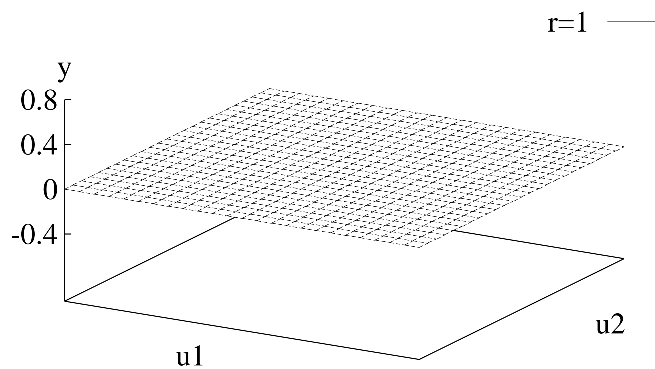


Abbildung 4.2: Modellausgabe bei einer Regel.

Funktion, da die einzige Regel nur einen Parameter enthält. Dieser ist wegen der Normierung gleich Null. Mit zwei Regeln wie in Abbildung 4.3 bildet das Modell zwei Niveaus mit unscharfem Übergang. Für $u_1 < -0,4$ ist $\hat{y} \approx -0,23$; für $u_1 > -0,1$ ist $\hat{y} \approx 0,06$.

Die folgende Aufspaltung des Eingaberaumes geschieht wieder bei u_1 (Abbildung 4.4). Die Regeln im linken und rechten Bereich haben beide negative Konsequenzen. Die rechte enthält in der Prämisse eine unscharfe Menge mit positivem, die linke mit negativem σ. Die Regel für die Mitte hat eine positive Konsequenz

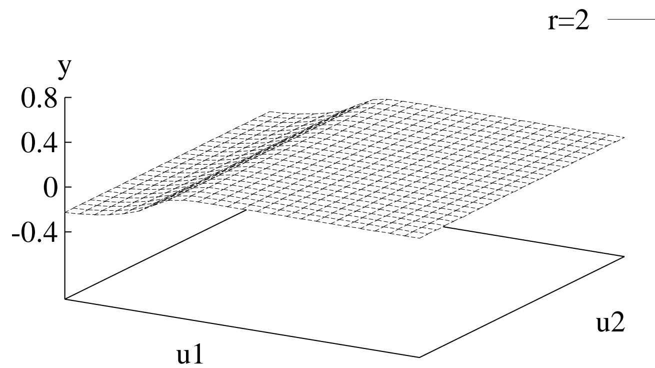


Abbildung 4.3: Modellausgabe bei zwei Regeln.

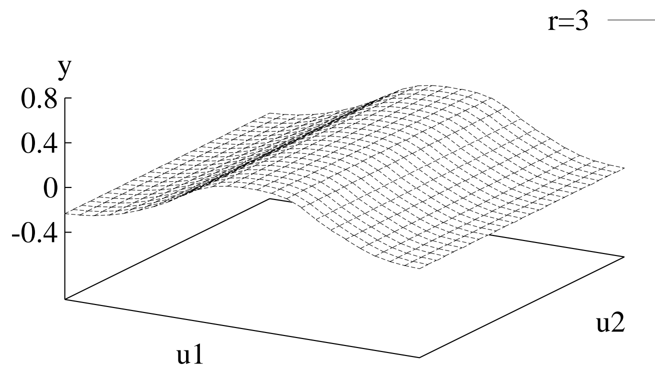


Abbildung 4.4: Modellausgabe bei drei Regeln.

und in der Prämisse zwei unscharfe Mengen, die nach links und rechts ihr Einflußgebiet begrenzen. Die Bildung der vierten Regel (Abbildung 4.5) bewirkt ein Auftrennen des mittleren Niveaus in u_2 und läßt die Originalfunktion schon erahnen.

Die nächste Verfeinerung, die das Gütekriterium am meisten verbessert, berücksichtigt die Verschiebung der beiden Gaußglocken (Abbildung 4.6). Obwohl nur eine Regel hinzugefügt wurde, verändern sich beide Glocken etwas. Das liegt daran, daß bei jeder Änderung sämtliche Parameter neu optimiert werden und damit

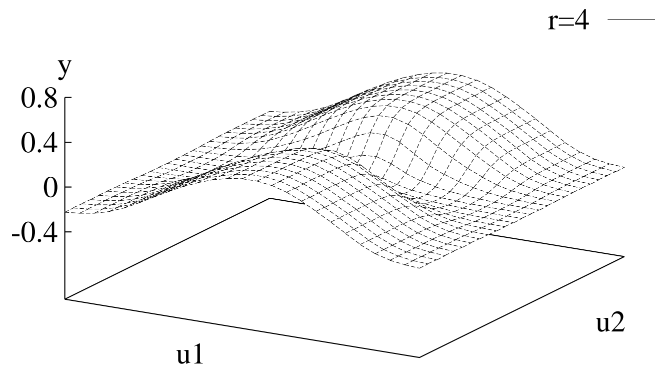


Abbildung 4.5: Modellausgabe bei vier Regeln.

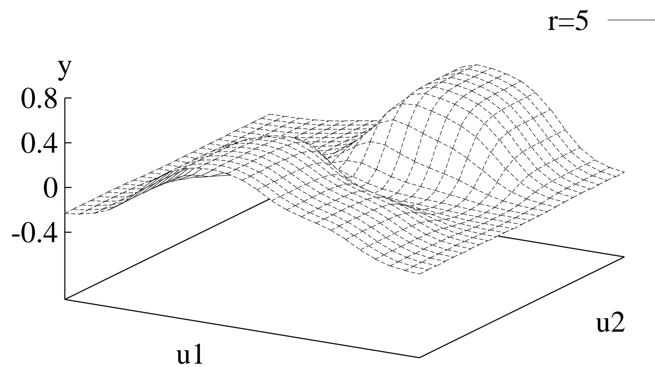


Abbildung 4.6: Modellausgabe bei fünf Regeln.

sich die Einflußbereiche aller Regeln bei Hinzufügen einer neuen ändern können. In Abbildung 4.7 wird durch die fünfte Regel nun auch ein Randbereich angepaßt. Es ist deutlich das Einflußgebiet der neuen Regel, der kleine Randbereich für mittleres u_1 und negatives u_2, erkennbar.

Sieben Regeln (Abbildung 4.8) reichen aus, um die Grobstruktur der Funktion zu erfassen. Die siebte Regel adaptiert hierzu den Randbereich der hinteren Glocke. Hinzufügen weiterer Regeln kann deshalb zu nur geringfügigen Verbesserungen der Approximation führen, was auch Abbildung 4.9 (die Entwicklung des Krite-

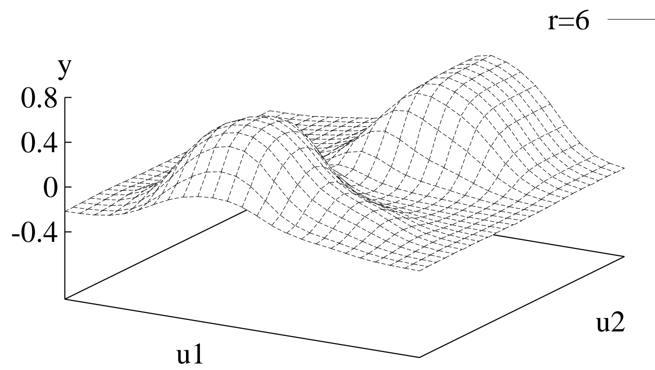


Abbildung 4.7: Modellausgabe bei sechs Regeln.

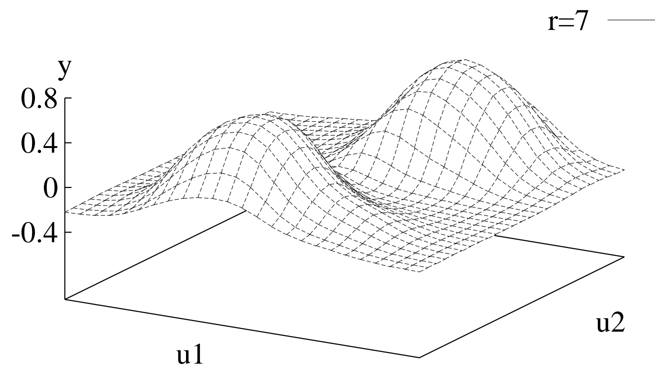


Abbildung 4.8: Modellausgabe bei sieben Regeln.

riums R^2) erkennen läßt.

Die Abbildungen 4.10 bis 4.16 zeigen die unscharfen Modelle mit Regelbasen von einer bis zu sieben Regeln. Die unscharfen Mengen sind durch ihre Zugehörigkeitsfunktionen dargestellt. Glockenkurven, wie in Regel $\mathtt{R_1}$ aus Abbildung 4.12, entstehen durch die Überlagerung (d.h. Multiplikation) zweier Sigmoidfunktionen.

Die Regelbasis aus Abbildung 4.12 läßt sich leicht interpretieren. Regel $\mathtt{R_1}$ sorgt

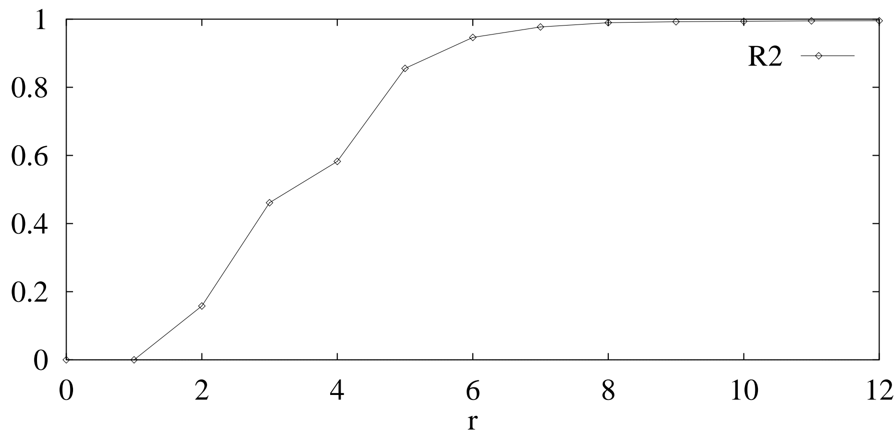


Abbildung 4.9: Entwicklung von R^2 bis $r_{max} = 12$.

`R`$_1$: `if TRUE then` $f_1 = 0,0000$

Abbildung 4.10: Modell mit konstanter Konsequenz und $r = 1$.

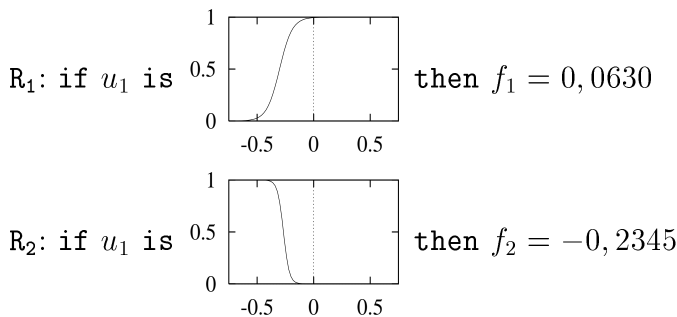


Abbildung 4.11: Modell mit konstanter Konsequenz und $r = 2$.

mit $f_1 = 0,2672$ für den Berg in der Mitte (siehe Abbildung 4.4), während die beiden anderen die tiefen Zonen bilden.

Der Einschnitt in den Hügel in Abbildung 4.5 wird auch aus dem Modell aus Abbildung 4.13 ersichtlich. Die den Hügel erzeugende Regel `R`$_1$ wurde aufgespalten und ist für $u_2 \approx 0$ nicht mehr gültig. Sie erzeugt jetzt nur noch die hintere der zwei Erhebungen ($u_2 > 0$), während `R`$_4$ die vordere ($u_2 < 0$) bewirkt.

Im Modell aus Abbildung 4.15 bildet die sechste Regel eine Verfeinerung der vierten. Durch den kleinen Wert $f_6 = 0,0250$ für $u_2 < 0,5$ wird die Abrundung des vorderen Hügels in Abbildung 4.7 erzeugt. In diesem Regelsatz fällt auf, daß die zweite unscharfe Menge in `R`$_4$ eine maximale Zugehörigkeit von etwa $0,5$ hat.

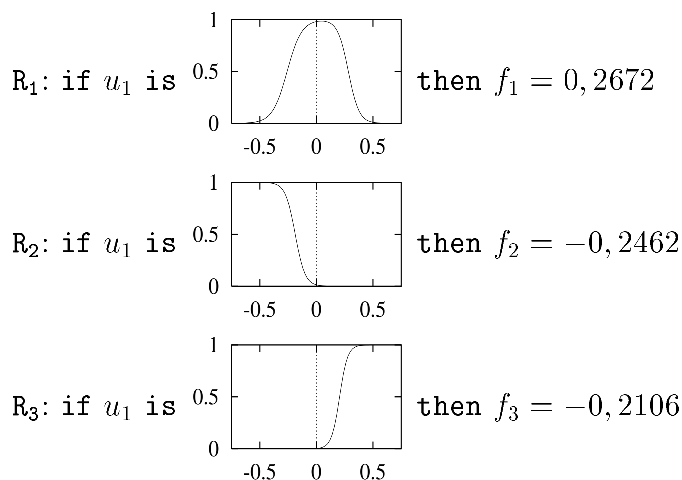


Abbildung 4.12: Modell mit konstanter Konsequenz und $r = 3$.

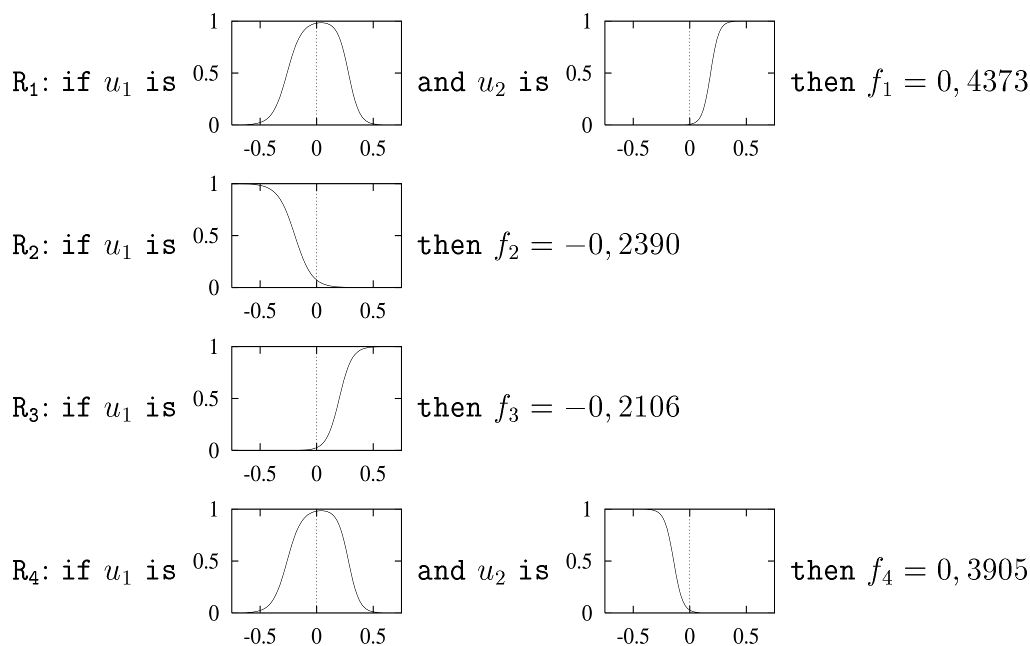


Abbildung 4.13: Modell mit konstanter Konsequenz und $r = 4$.

Durch die konjunktive Verknüpfung der unscharfen Klauseln ist somit auch die Zugehörigkeit zu $\mathtt{R}_4$ auf $0,5$ begrenzt. Diese Regel hat dadurch, im Vergleich mit den anderen, an Einfluß verloren.

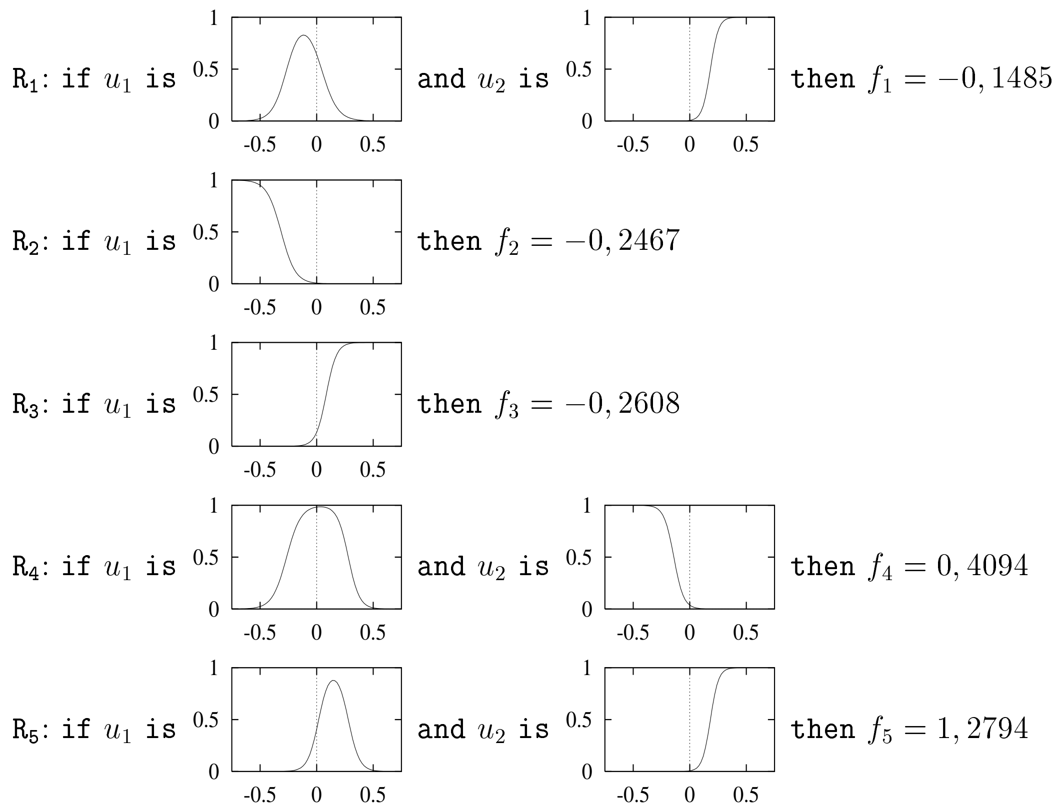


Abbildung 4.14: Modell mit konstanter Konsequenz und $r = 5$.

In Abbildung 4.16 ist die siebte Regel eine Verfeinerung der von R_5 und bewirkt die Abrundung der hinteren Erhebung. Trotzdem ist hier f_7 mit $0,9390$ recht groß und gibt nicht direkt die kleineren Werte in diesem Bereich wieder. Das entsteht durch die zusätzliche Überlagerung von R_1.

Zusammenfassend läßt sich sagen, daß die gefundenen Modelle durchaus interpretierbar sind, eine Interpretation aber mit steigender Regelanzahl immer schwerer wird. Besonders eine Überlagerung mehrerer Regeln, wie zum Beispiel von R_1, R_5 und R_7 in Abbildung 4.16, ist schwer auf den ersten Blick interpretierbar.

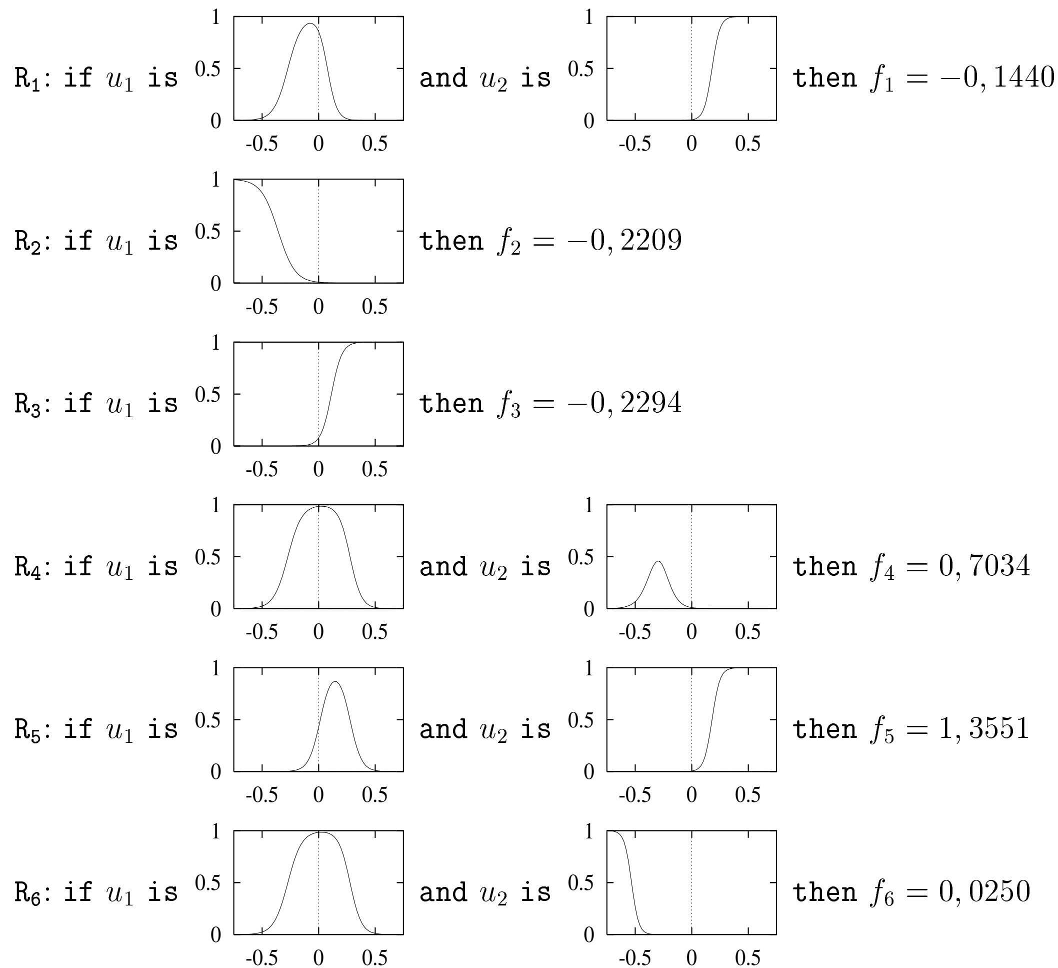


Abbildung 4.15: Modell mit konstanter Konsequenz und $r = 6$.

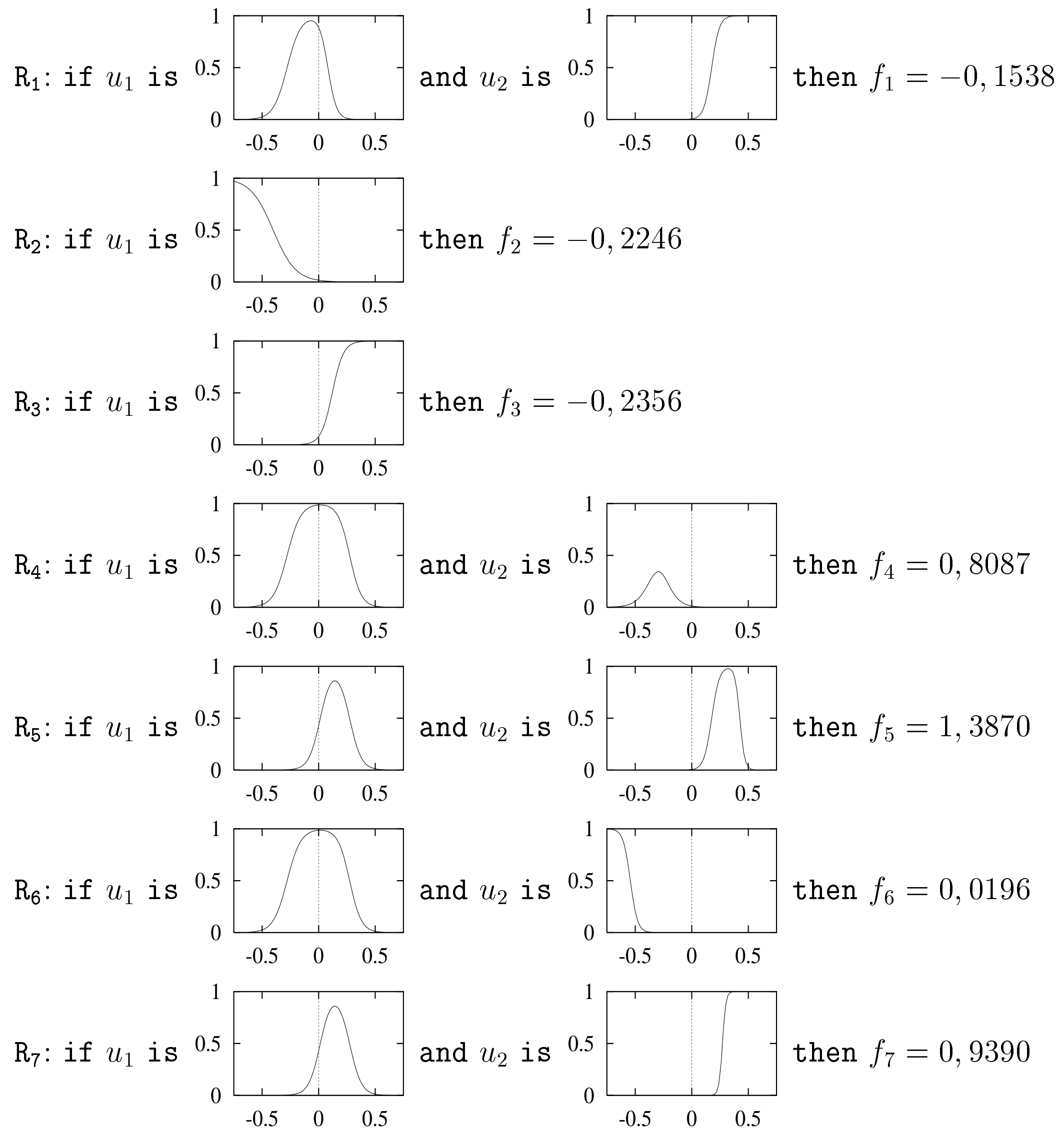


Abbildung 4.16: Modell mit konstanter Konsequenz und $r = 7$.

4.2 Benchmark von Frank

Um die Leistungsfähigkeit des Verfahrens zu testen und mit der anderer Algorithmen zu vergleichen, wurde ein Benchmark von I. Frank [Fra95] durchgeführt. Bei diesem Benchmark wurden vier vierdimensionale Funktionen (A–D) und ein Beispiel mit praktischen Meßdaten (E) aus der Lebensmittelchemie gewählt. Gleichungen (4.1) bis (4.4) zeigen die Funktionen, Tabelle 4.1 die Meßdaten. Die erste Funktion ist linear, alle anderen sind nichtlinear.

$$\text{A:} \quad y = u_1 + u_2 + u_3 + u_4 \tag{4.1}$$

$$\text{B:} \quad y = 2(u_1 - 1)^2 + 2(u_2 - 1)^2 + \sin(2\pi u_3) + \ln(u_4 + 0,1) \tag{4.2}$$

$$\text{C:} \quad y = \sin(u_1 + u_2 + u_3 + u_4) + \ln(u_1 + u_2 + u_3 + u_4) \tag{4.3}$$

$$\text{D:} \quad y = 10\sin(u_1 u_2) + 20(u_3 - 0,5)^2 + 5u_4 \tag{4.4}$$

Zu den Modellen A bis D wurden Datensätze mit gleichverteilten $u_i \in [0;1]$ mit 100 Trainings- und 1000 Testbeispielen erzeugt. Dazu wurde normalverteiltes (weißes) Rauschen auf den Ausgang y gelegt; jeweils mit den Signal-zu-Rausch Verhältnissen SRV $= 1$, 3, 7 und ∞. Zusätzlich wurde jedes der so entstandenen Modelle einmal mit den inhärenten vier, ein zweites mal mit vier zusätzlichen Eingängen erzeugt, die nur weißes Rauschen enthalten[1]. Mit den fünf Modellen, vier Rauschniveaus und zwei Eingangsdimensionen ergeben sich $5 \times 4 \times 2 = 40$ verschiedene simulierte und zu identifizierende Modelle.

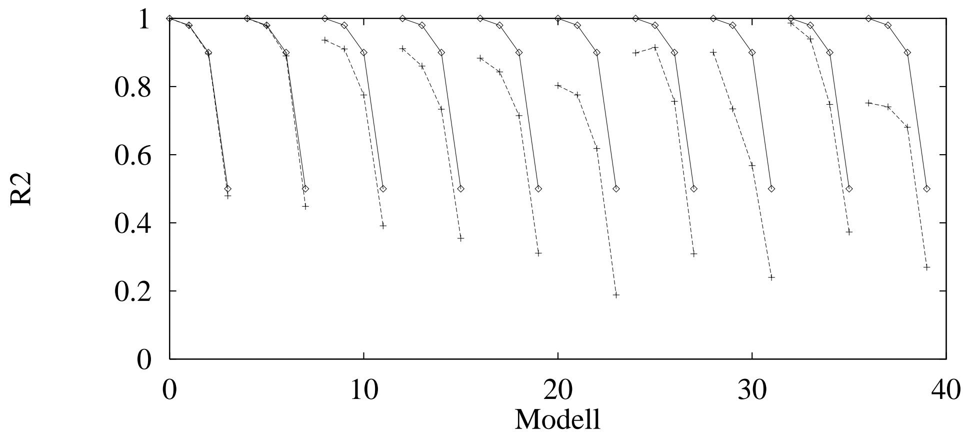


Abbildung 4.17: R^2 für 40 Benchmarkmodelle.

Der Datensatz für die 40 verschiedenen Modelle wurde fünf mal mit verschiedenen Startwerten des Zufallsgenerators erzeugt und die unscharfe Modellierung mit vollständigen Konsequenzen durchgeführt. Die Rechenzeit auf einer Sun Sparc IPX betrug dabei durchschnittlich etwa 10 Minuten pro Modellierung.

[1] Es sind also mit dem Ausgang unkorrelierte Eingänge, die sich im allgemeinen störend auf die Modellierung auswirken.

u_1	u_2	u_3	y	u_1	u_2	u_3	y
0,0990	89,360	606,88	150,8	0,0820	95,230	607,51	150,9
0,1390	87,150	602,19	122,8	0,0560	96,850	611,76	149,7
0,0570	97,740	612,75	166,7	0,2040	76,300	596,86	88,3
0,0590	98,430	615,14	157,1	0,0460	95,830	609,60	127,7
0,0680	98,000	609,43	143,0	0,0680	97,330	612,01	155,1
0,1070	94,980	607,06	140,1	0,0910	94,080	607,15	152,0
0,1010	90,650	607,12	159,7	0,2320	78,660	594,14	71,4
0,0920	93,380	605,65	150,2	0,0810	91,580	609,27	146,7
0,1130	89,810	605,68	152,8	0,0530	94,290	612,78	154,4
0,0820	96,500	606,33	131,5	0,0850	95,740	609,91	157,4
0,0600	96,890	612,08	168,3	0,0790	93,290	611,08	157,5
0,1940	81,230	598,77	100,1	0,0800	93,740	610,56	165,3
0,0750	94,620	610,36	160,9	0,1410	83,970	603,80	128,0
0,0640	97,820	612,17	153,2	0,0530	97,000	611,39	114,9
0,0930	94,120	608,57	156,3	0,0900	94,230	610,49	164,7
0,0470	97,350	616,78	157,4	0,1130	91,900	606,08	145,3
0,0950	94,150	605,79	123,5	0,0830	95,190	609,11	154,7
0,1190	91,020	602,87	134,6	0,0730	95,400	612,19	165,7
0,0420	98,670	617,30	122,4	0,0620	96,800	610,44	154,7
0,1220	91,960	604,44	126,3	0,0730	93,840	611,45	164,7
0,0870	94,080	608,88	157,4	0,0460	96,500	615,18	148,9
0,0820	94,600	605,04	130,7	0,0680	97,130	612,22	147,1
0,1060	94,800	602,28	128,0	0,0990	93,260	607,00	143,0
0,1180	90,430	603,76	149,4	0,0420	97,370	615,06	155,7
0,1160	90,160	602,07	148,8	0,0870	93,860	608,28	159,4
0,1630	86,350	598,90	121,2	0,0650	95,490	613,58	160,6
0,1190	89,590	604,39	143,1	0,0930	96,520	606,11	134,2
0,0450	97,630	613,28	136,5	0,0400	98,020	619,43	123,3
0,0760	96,790	607,82	151,8				
0,0720	95,700	611,21	165,3				

Tabelle 4.1: Datenmenge für Beispiel E.

Abbildung 4.17 zeigt die aus den 5 Durchläufen gemittelten R^2 für die 40 verschiedenen Modelle (gestrichelte Linie). Die durchgezogene Linie verdeutlicht die theoretisch optimalen Werte. Diese sind für SRV = 1, 3, 7 und ∞ gleich 0,5; 0,9; 0,98 und 1. Dabei sind von links her die Modelle von A mit zuerst vier, dann acht Eingängen aufgetragen. Das Gleiche dann für B, usw. bis E.

Abbildung 4.18 zeigt die Summe von R^2 aller 40 Modelle. Ein Wert von $10 \cdot (1,0 + 0,98 + 0,9 + 0,5) = 33,8$ ist dabei maximal zu erreichen. Im übergreifenden Vergleich mit vier anderen nichtlinearen Modellierungsalgorithmen aus [Fra95]

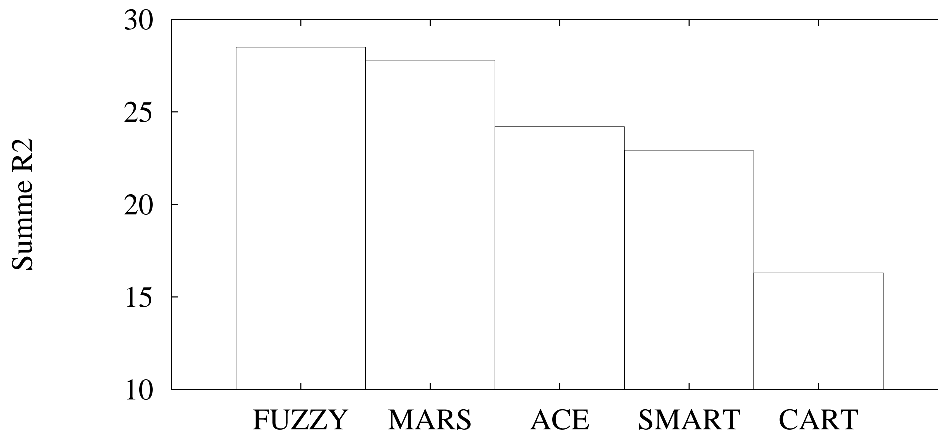


Abbildung 4.18: Vergleich mit anderen Algorithmen.

schnitt der hier vorgestellte unscharfe Modellierungsalgorithmus als bester ab.

Mit dem Benchmark sollen folgende Fragen untersucht werden, die hier für den vorgestellten Algorithmus beantwortet werden. Siehe hierzu Abbildung 4.17 und [Fra95].

1. *Erfassen diese nichtlinearen Modellierungsalgorithmen auch lineare Beziehungen?* Ja. Alle acht untersuchten linearen Modelle sind selbst bei irrelevanten Eingängen und starkem Rauschen fast optimal identifiziert worden. Dies war auch zu erwarten, da die erste Epoche des Algorithmus ein lineares Modell liefert, dessen Parameter nach der kleinsten Quadrate Methode berechnet werden.

2. *Erfassen sie die nichtlineare Funktion?* Ja. Alle fünf Funktionen wurden hinreichend gut identifiziert.

3. *Wie werden schwer zu approximierende Funktionen identifiziert?* Unterschiedlich gut. Jeder der Algorithmen zeigt Schwachstellen bei bestimmten Funktionen. So zeigt MARS ([Fra95] Abbildung 12), der in Frank's Artikel als bester Algorithmus abschnitt, bei den Modellen C und E nur mäßige Ergebnisse. SMART, fast perfekt bei C, funktioniert sehr schlecht bei B, D und E ([Fra95] Abbildung 11). CART schneidet bei den realen Daten E als bester ab, bei allen anderen Modellen, insbesondere auch bei dem linearen Modell A, ist er jedoch sehr schlecht ([Fra95] Abbildung 13). Der unscharfe Modellierungsalgorithmus zeigt jeweils bei Modell C und E bei der Variation mit den zusätzlichen unkorrelierten Eingängen nur mäßige Ergebnisse (Abbildung 4.17).

4. *Wie reagieren sie auf irrelevante Eingänge?* Unkorrelierte Eingänge werden, zumindest am Anfang der Modellbildung, nicht in die Modelle mit-

aufgenommen. Die R^2 für die Modelle mit acht Eingängen sind meist auch nicht schlechter als die der anderen. Modell E mit den realen Meßdaten zeigt aber, daß unter Umständen unkorrelierte Eingänge die Modellierung auch stark erschweren können.

5. *Wie verschlechtert sich die Modellierung bei verschiedenen Rauschniveaus?* Stark. Bei den nichtlinearen Modellen C und D verschlechtert sich das Ergebnis sehr, wenn $SRV < 3$.

6. *Gibt es eine eine global beste Methode?* Nein. Im Vergleicht mit z.B. MARS ist die unscharfe Modellierung schlechter bei den Modellen B und D, aber besser bei C und bei E ohne zusätzliche Eingänge. Jede Methode ist jeweils für eine bestimmte Funktionenklasse optimal. Die hier vorgestellte Methode ist besonders für teilweise lineare Funktionen mit fließenden Übergängen geeignet. Sie zeigte sich bisher auch bei realen Meßdaten als erfolgreich.

4.3 Porosität von Papier

Als Beispiel einer statischen Identifikation war die Identifikation von Papiereigenschaften bei einer komplexen Papiermaschine vorgesehen. Eine Beschreibung der Maschine befindet sich in [Arr93] und in [Hih93]. Allgemeines über Papiermaschinen und ihre Identifikation bietet [Che95].

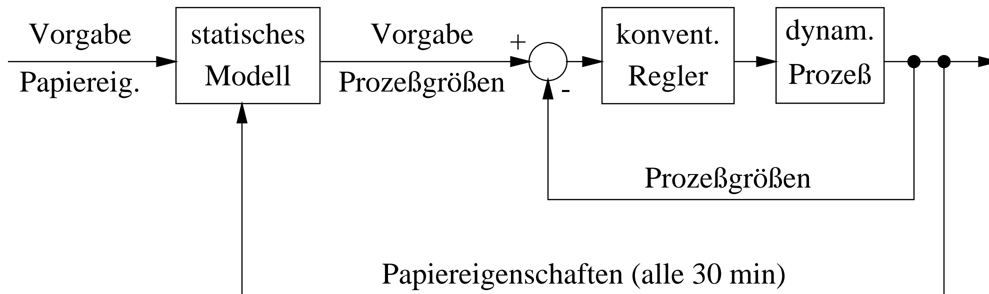


Abbildung 4.19: Blockschema der Regelung einer Papiermaschine.

Die Maschine an sich ist dynamisch und erfordert auch eine dynamische Regelung (siehe Abbildung 4.19). Die Papiereigenschaften wie z.B. die Porosität können aber nur im Labor in größeren Zeitabständen untersucht und gemessen werden. Deshalb wird ca. alle 30 Minuten eine Papierprobe genommen und auf die Papiereigenschaften hin untersucht. Befand sich die Maschine seit der letzten Messung in statischem Betrieb, so können die Prozeßgrößen zusammen mit den gemessenen Papiereigenschaften in den Datensatz aufgenommen werden. Mit Hilfe eines (statischen) Modells sollen dann für eine bestimmte Vorgabe der Papiereigenschaften die nötigen Vorgaben der Prozeßgrößen berechnet werden. Es wurden insgesamt drei Versuche mit verschiedenen Eingabevariablen durchgeführt. Am erfolgreichsten zeigte sich der mit den wenigsten Variablen. Ein großes Problem hierbei war, daß die Zahl der verfügbaren Meßwerte mit der Hinzunahme von Eingangsvariablen sank, da die Datentabelle Lücken enthielt (aufgrund von zeitweise nicht statischen Variablen oder Ausfällen bei der Messung). Das hier vorgestellte Beispiel enthält 11 Eingangsvariablen bei nur 518 Beispielen, von denen 344 (zwei Drittel) zur Identifikation und 174 zur Validation Verwendung fanden.

Abbildung 4.20 zeigt ein weiteres Problem. Sie zeigt die Entwicklung von R^2 bei der Modellierung mit 11 Eingangsvariablen und vollen Konsequenzen. Der verfügbare Datensatz wurde dabei geteilt und die Hälfte der Beispiele zur Identifikation, die andere Hälfte zur Validation eingesetzt. Bei der Identifikation wurde das Modell auch immer genauer, R^2 nähert sich immer mehr an 1 an (Abbildung 4.20, durchgezogene Linie). Die Validation (gestrichelte Linie) hat aber bei $r = 7$ mit $R^2 \approx 0,55$ ihr Maximum und fällt danach stark ab. Dies ist ein Zeichen für stark verrauschte Meßdaten, so daß genauere Modelle mit $r > 7$ nicht

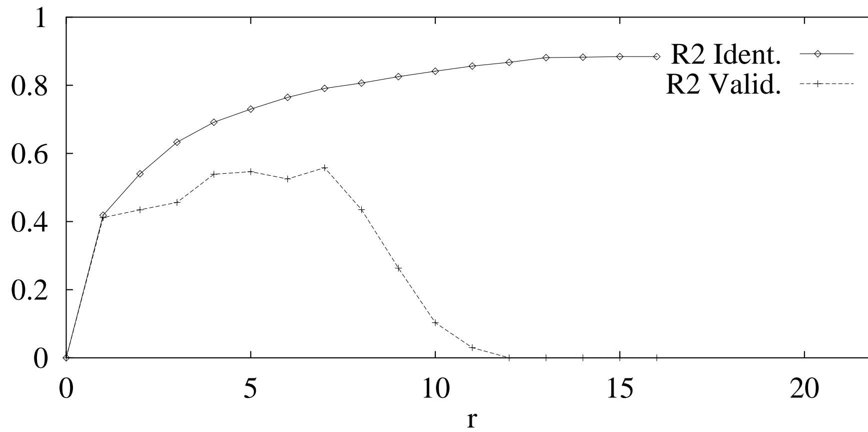


Abbildung 4.20: R^2 für die Papiermaschine.

mehr das vorliegende Modell, sondern das Rauschen identifizieren. Auch konventionelle Methoden aus [Arr93] erzielten keine besseren Werte als $R^2 = 0,55$. Die unscharfe Modellierung zeigte hier also keine besseren Ergebnisse als statistische Methoden, war aber auch nicht schlechter. Ein anderer Aspekt ist auch die Modellierung mit Konstanten in den Konsequenzen. Die erzielten Modelle sind dann in diesem Beispiel zwar erheblich schlechter als bei Modellen mit voller Konsequenz, die gefundenen Modelle lassen aber leichter auf Variablenzusammenhänge schließen.

r	SR	OL	VT1	VT2	PL1	Grm	QBMA	QCATO	VE	FK	BB
1						x					
3	x					x					
4	x					x				x	
5	x		x			x				x	
6	x		x	x		x				x	
7	x		x	x		x				x	x
8	x		x	x		x		x		x	x

Tabelle 4.2: Schrittweise Hinzunahme korrelierter Eingangsvariablen.

Tabelle 4.2 zeigt, welche Eingangsvariablen in welcher Regel auftauchen. Fünf der sieben hierbei als relevant gefundenen Variablen werden auch von Experten als wichtig erachtet (Tabelle 4.3). Es sind dies: SR, Grm, VT1, VT2 und QCATO. Zur Bedeutung der Variablen siehe [Arr93].

Die Betrachtung der in den Regelprämissen auftauchenden Eingänge gibt somit qualitativ einen Hinweis auf Eingänge, die stark mit der Ausgabe verknüpft sind. Das Verfahren eignet sich also, selbst wenn die Qualität des gefundenen Modells nicht ausreichend ist, trotzdem noch zur Detektion relevanter Eingänge.

	Regelgrößen		
Ausgang	Haupteinfluß	Zusätzliches	Störungen
Porosität	SR	OL	PL1
		VT1	Grm
		VT2	QBMA
			QCATO

Tabelle 4.3: Relevante Variablen nach Expertenmeinung.

4.4 Datensatz von Box-Jenkins

Untersucht wurde auch der in der Literatur schon oft behandelte Datensatz von Box-Jenkins aus [BJ76]. Es handelt sich hierbei um einen dynamischen Prozeß, einen Gasofen, bei dem die CO_2-Konzentration in der Brennkammer identifiziert werden soll. Es sollen dann sowohl Prädiktionen als auch Simulationen durchgeführt werden, wie sie in Kapitel 3.6 definiert sind.

Hierzu wurden die 296 Meßwerte normalisiert. Zur Modellierung wurden die ersten 145 Werte verwendet, zur Prädiktion und Simulation alle 296 Werte. Als Eingänge dienen $u_{t-1}, \ldots, u_{t-6}$ und $y_{t-1}, \ldots, y_{t-4}$.

Vollständige Konsequenz

Der erste Versuch wurde mit vollständiger Konsequenz durchgeführt. Abbildung 4.21 zeigt dessen Entwicklung von R^2. Schon das lineare Modell mit $r = 1$ hat ein $R^2 = 0,9978$, so daß weitere Regeln kaum noch Verbesserungen bringen können.

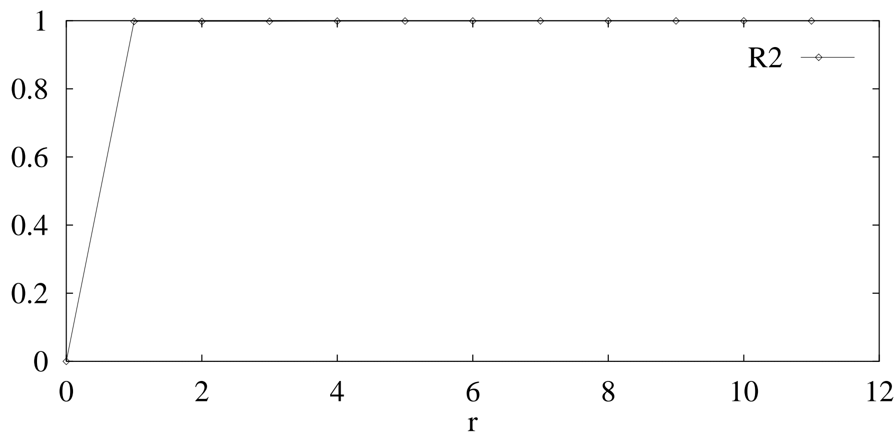


Abbildung 4.21: Entwicklung von R^2 für vollständige Konsequenz.

$$\begin{aligned} \hat{y}_t \quad = \quad & 0,000 + 0,040u_{t-1} - 0,104u_{t-2} - 0,221u_{t-3} + \\ & 0,010u_{t-4} - 0,052u_{t-5} + 0,091u_{t-6} + \\ & 0,818y_{t-1} + 0,086y_{t-2} - 0,104y_{t-3} - 0,009y_{t-4} \end{aligned} \tag{4.5}$$

Gleichung (4.5) gibt das lineare Modell für den Gasofen wieder. Seine Prädiktion bzw. Simulation sind in den Abbildungen 4.22 bzw. 4.23 für alle 296 Werte zu sehen.

Beide, Prädiktion und Simulation, sind recht gut. Die ständigen Abweichungen bei der Simulation für $t > 220$ entstehen durch Fehlerpropagation, wie sie bei

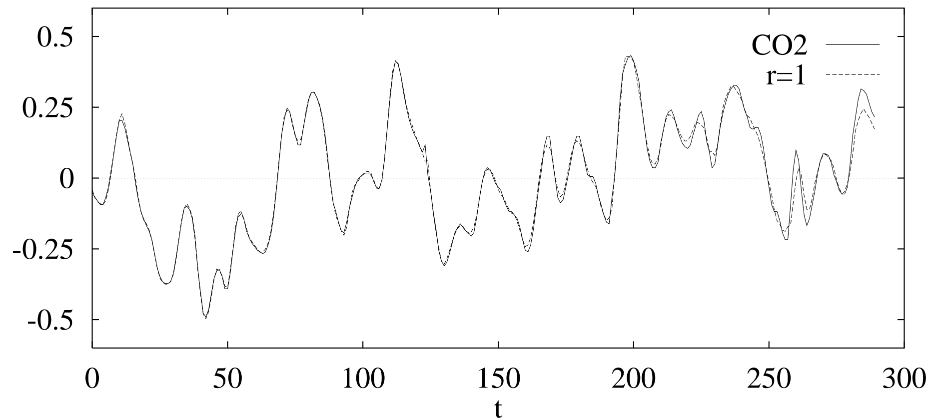


Abbildung 4.22: Prädiktion des linearen Modells.

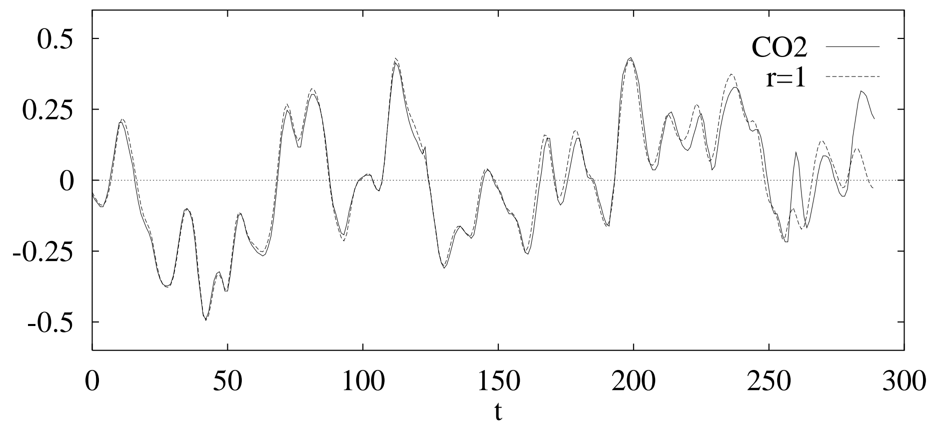


Abbildung 4.23: Simulation des linearen Modells.

Simulationen leicht auftreten kann. Das lineare Modell ist aber schon so gut, daß eine Verbesserung durch nichtlineare Modelle kaum mehr erreicht werden kann und somit uninteressant ist.

Konstante Konsequenz

Bei der Betrachtung der Modellierung mit konstanten Konsequenzen ist interessant, daß auch hier das Kriterium R^2 schnell ansteigt und es schon für $r = 2$ gleich 0,934 und für $r = 3$ gleich 0,967 ist. Abbildung 4.24 zeigt seine Entwicklung.

Schon ein Modell mit drei Regeln liefert dabei eine passable Simulation (Abbildung 4.25). Diese verbessert sich mit der Regelanzahl, aber ab $r = 7$ stellen sich in der Simulation Ausbrecher ein (in Abbildung 4.26 bei $t \approx 40$, die sich bei noch höherer Regelanzahl verstärken. Es macht allerdings auch wenig Sinn, das Modell

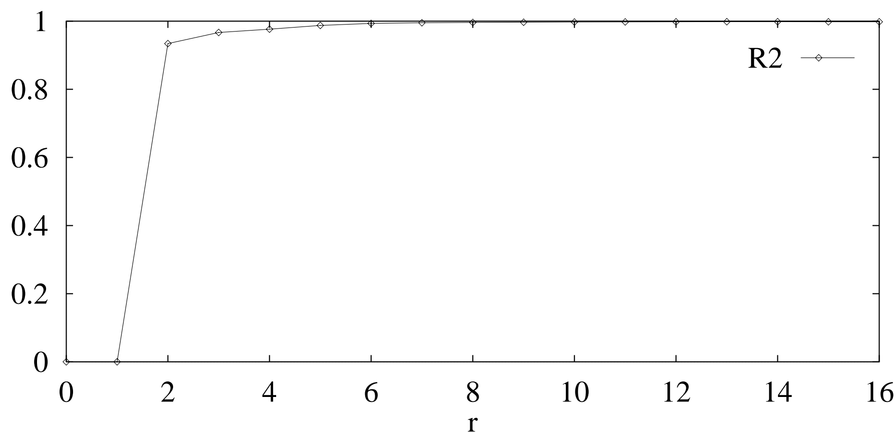


Abbildung 4.24: Entwicklung von R^2 für konstante Konsequenz.

über $r = 7$ hinaus weiter zu verfeinern, da hier $R^2 = 0,995$ kaum mehr verbessert werden kann.

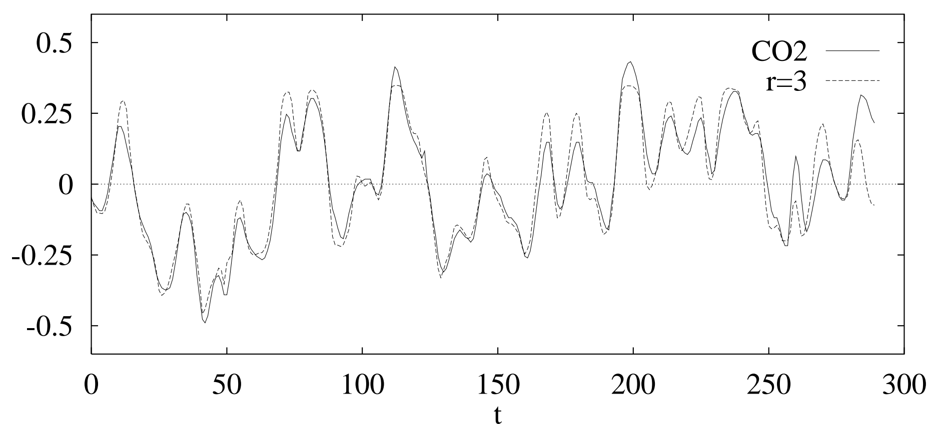


Abbildung 4.25: Simulation mit konstanter Konsequenz, $r = 3$.

Abbildung 4.27 zeigt das unscharfe Modell für $r = 3$. Solch ein kleines Modell ist leicht zu interpretieren. Es teilt den zweidimensionalen Eingaberaum in drei Bereiche mit den drei Niveaus f_1 bis f_3 und den fließenden Übergängen.

Die Abbildungen 4.28 und 4.29 enthalten das unscharfe Modell für $r = 7$. Es ist kaum mehr interpretierbar, da die Überlagerung der einzelnen Regeln zu unübersichtlich ist. Wiederum lassen sich aber aus den Prämissen die mit dem Ausgang korrelierten Eingänge ablesen.

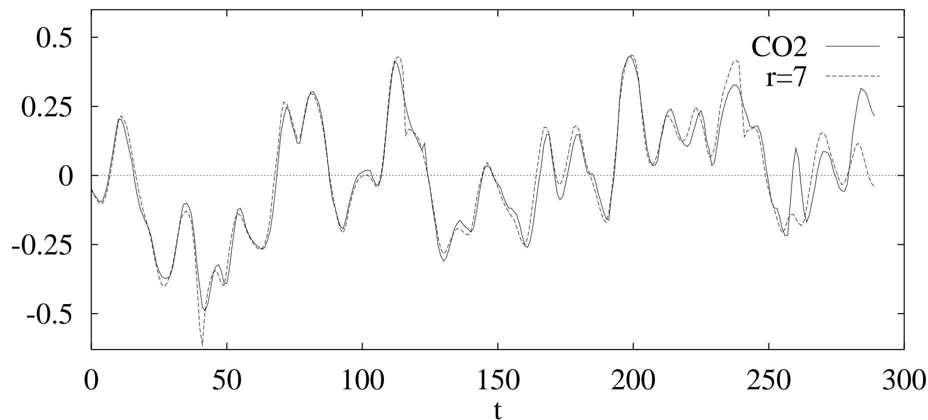


Abbildung 4.26: Simulation mit konstanter Konsequenz, $r = 7$.

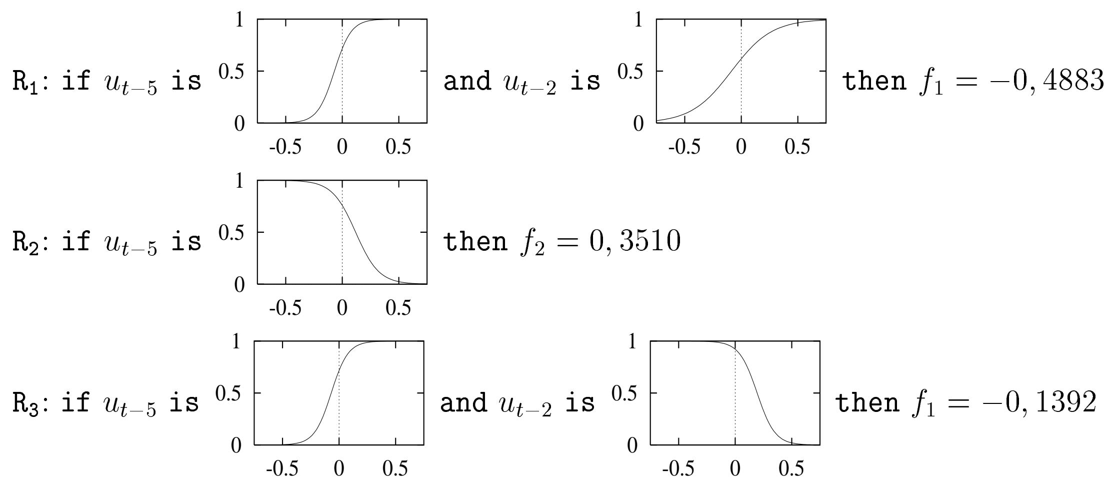


Abbildung 4.27: Modell mit konstanter Konsequenz und $r = 3$.

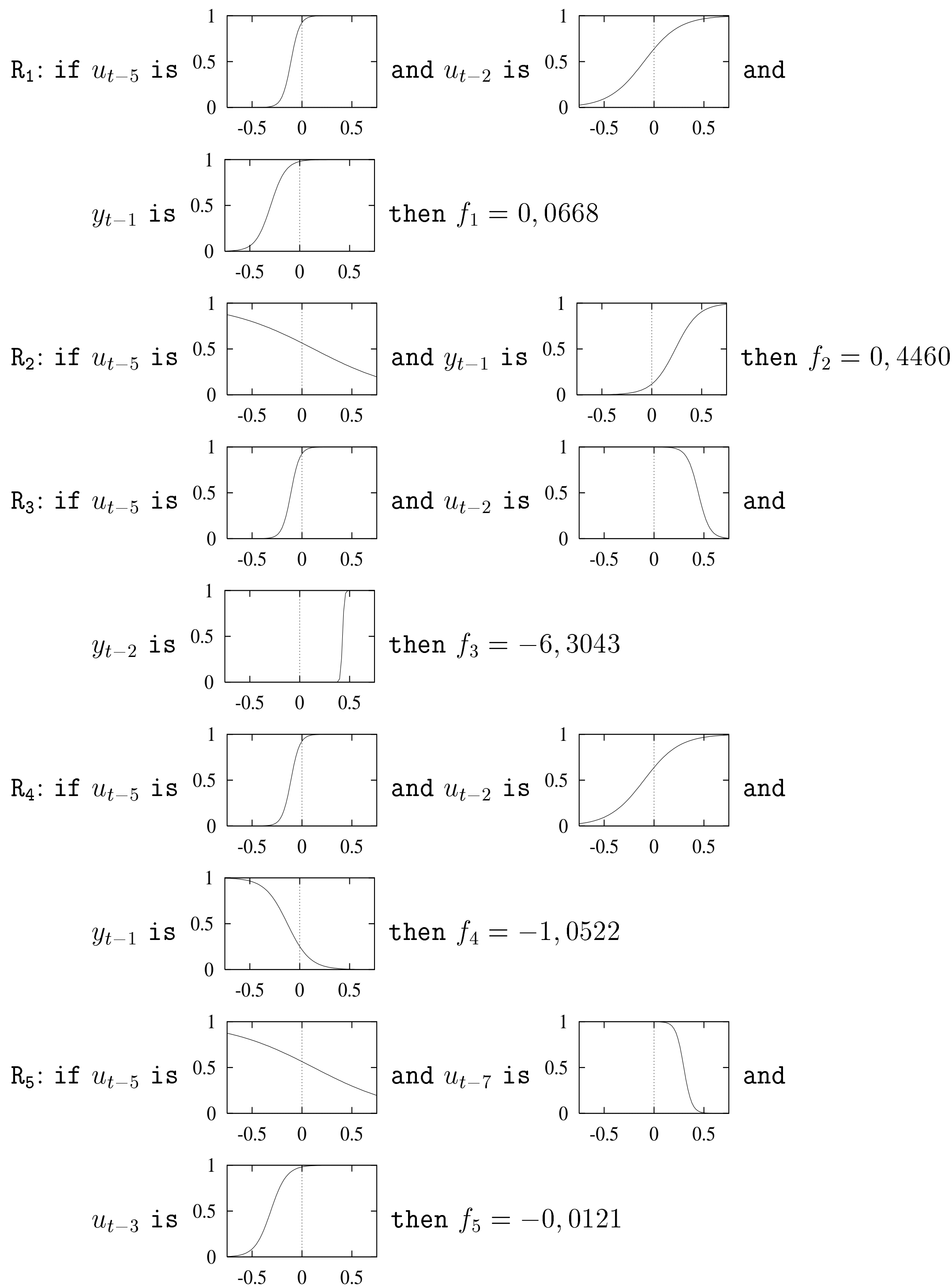


Abbildung 4.28: Modell mit konstanter Konsequenz und $r = 7$, 1. Teil.

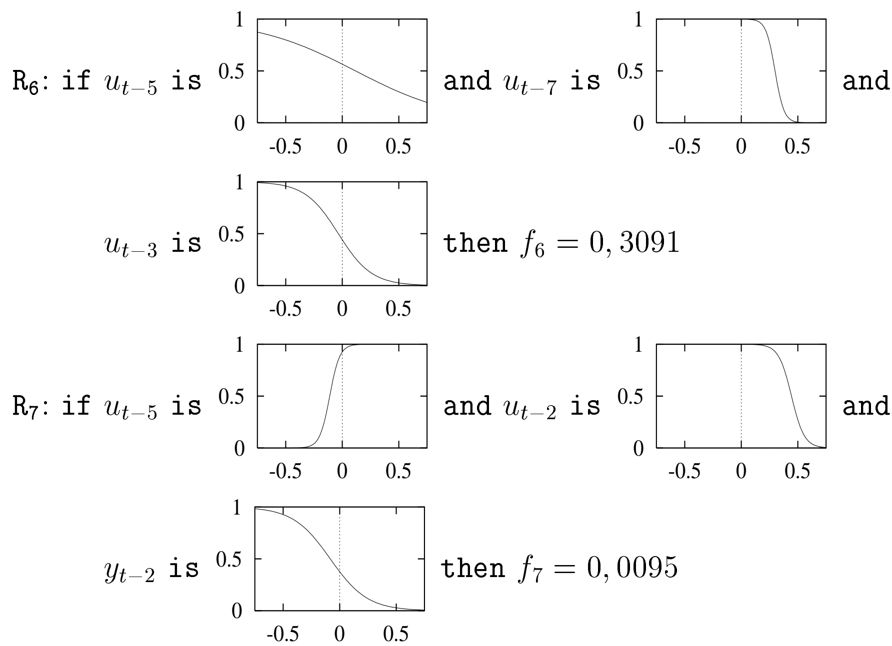


Abbildung 4.29: Modell mit konstanter Konsequenz und $r = 7$, 2.Teil.

4.5 Die Karlsruher Hand

In diesem Versuch wird die Karlsruher Hand durch ein physikalisches Modell, einem linearen Modell mit einer Totzone[2] und einer zusätzlichen Schwerkraftkomponente, angenähert. Mit diesem Modell werden 1000 Datenbeispiele erzeugt, wovon die ersten 500 zur Modellierung verwendet werden. Versuche zur Identifikation mit unscharfen Relationen finden sich in [Bö94]. Bei den folgenden Versuchen mit unscharfen Regeln werden zur Modellierung die Eingänge M_{t-1}, M_{t-2}, M_{t-3} und φ_{t-1}, φ_{t-2}, und φ_{t-3} zur Verfügung gestellt. Gesucht ist also φ_t.

Modellierung 1: Erregung mit Stufen

Als erster Versuch wird die Erregung des Systems durch eine Stufenfunktion (Abbildung 4.30) vorgegeben. Die Abtastzeit beträgt $T_s = 0,01s$. Abbildung 4.31 zeigt die Systemantwort.

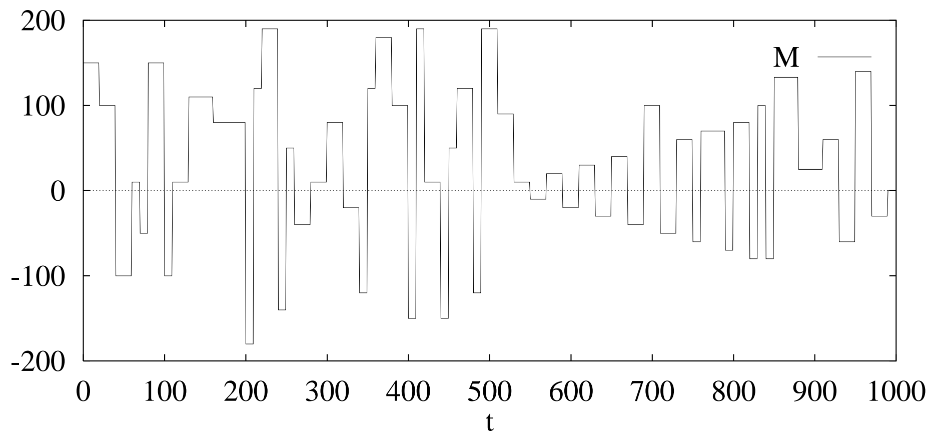


Abbildung 4.30: Erregung M des Modells 1.

Bei der Modellierung mit vollständiger Konsequenz erreicht schon das lineare Modell

$$\texttt{R}_1\texttt{: if TRUE then } f_1 = 0,00000 + 0,00623M_{t-1} + 0,00141M_{t-2} - 0,00663M_{t-3} + 2,75465\varphi_{t-1} - 2,52260\varphi_{t-2} + 0,76791\varphi_{t-3}$$

ein $R^2 = 0,999997$, das nicht mehr zu verbessern ist. Die Prädiktion des linearen Modells (Abbildung 4.32) deckt sich mit der Originalkurve. Die Simulation (gestrichelte Linie in Abbildung 4.33) zeigt aber die Schwächen des Modells. Kritisch ist nicht so sehr eine langsam zunehmende Abweichung von der Originalkurve

[2]Der Bereich, in dem der Prozeß trotz einer Erregung noch keine Reaktion zeigt, wird als Totzone bezeichnet.

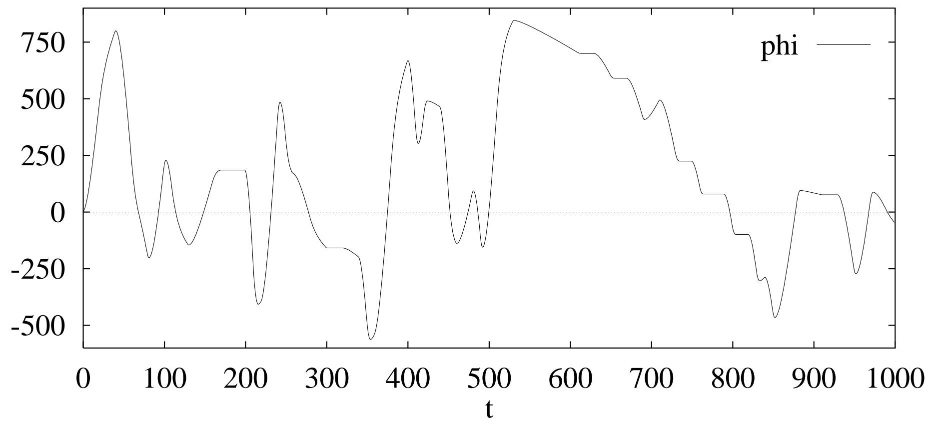


Abbildung 4.31: Antwort φ des Modells 1.

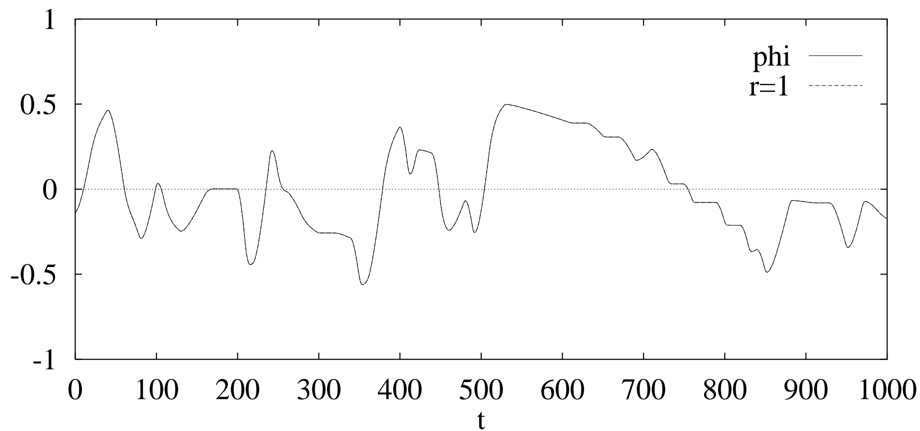


Abbildung 4.32: Prädiktion des linearen Modells 1 (gestrichelte Linie), die sich vollständig mit der Originalkurve (durchgezogene Linie) deckt.

– diese entsteht durch Fehlerfortpflanzung – sondern vielmehr das Nichterkennen der Totzone, wie sie z.B. für $150 < t < 200$ auftritt. Trotz nahezu perfekter Prädiktion wurde das System nicht vollständig identifiziert. Das liegt hier nicht am Modellierungsalgorithmus, denn ein R^2 von 0,999997 ist nicht mehr verbesserbar, sondern an der Generierung der Meßbeispiele.

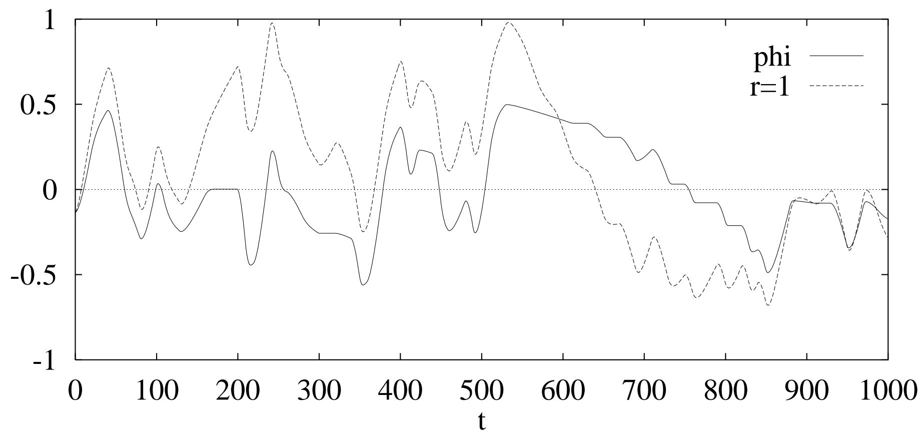


Abbildung 4.33: Simulation des linearen Modells 1 (gestrichelte Linie).

Modellierung 2: Zufallserregung

In einem zweiten Versuch wurde eine größere Abtastzeit $T_s = 0,05$ (um einen größeren Zeitraum zu erfassen) und eine „wildere“ Erregung gewählt: Eine Zufallserregung, bei der circa 80% der Erregungen in der Totzone $-50 < M < 50$ liegen, um diesen Effekt besonders hervorzuheben (Abbildung 4.34). Die Systemantwort ist in Abbildung 4.35 dargestellt. Wiederum wurde mit vollständigen Konsequenzen modelliert.

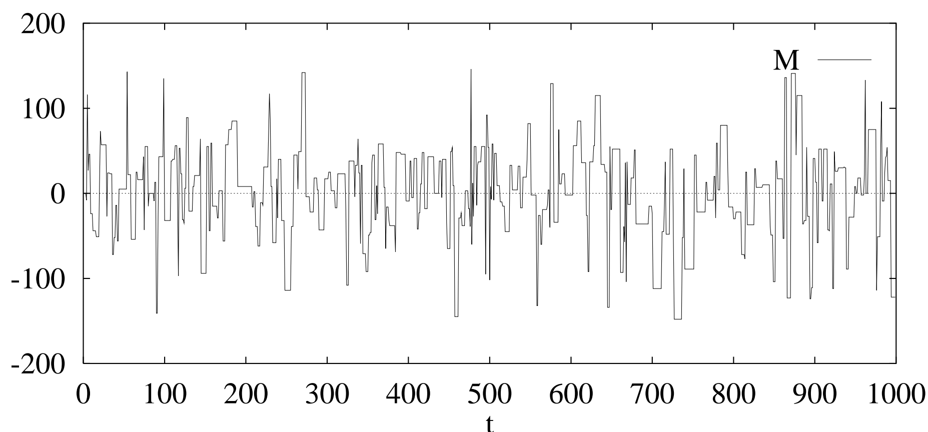


Abbildung 4.34: Erregung M des Modells 2.

Aber auch hier ist die Prädiktion mit $R^2 = 1,0000$ aus Abbildung 4.36 nahezu perfekt, während die Simulation aus Abbildung 4.37 nicht zufriedenstellend ist. Deutlich wird wieder für $400 < t < 600$, daß sich die Simulation „zuviel bewegt“, d.h. wahrscheinlich die Totzone nicht richtig modelliert wurde.

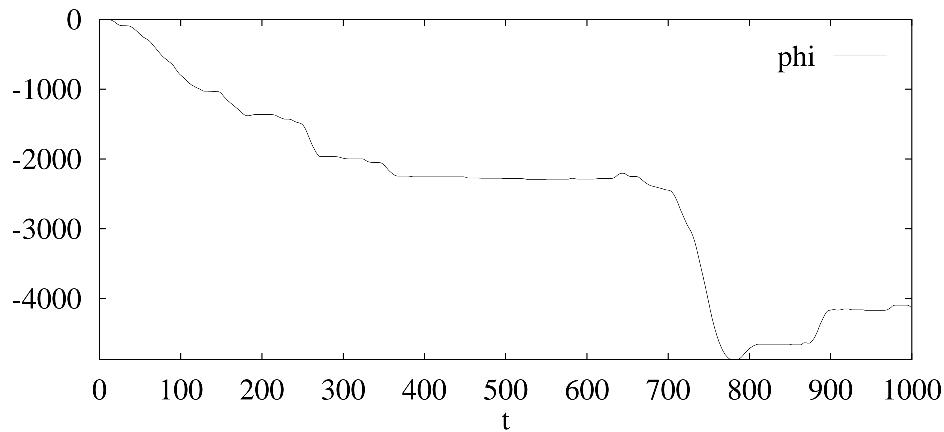


Abbildung 4.35: Antwort φ des Modells 2.

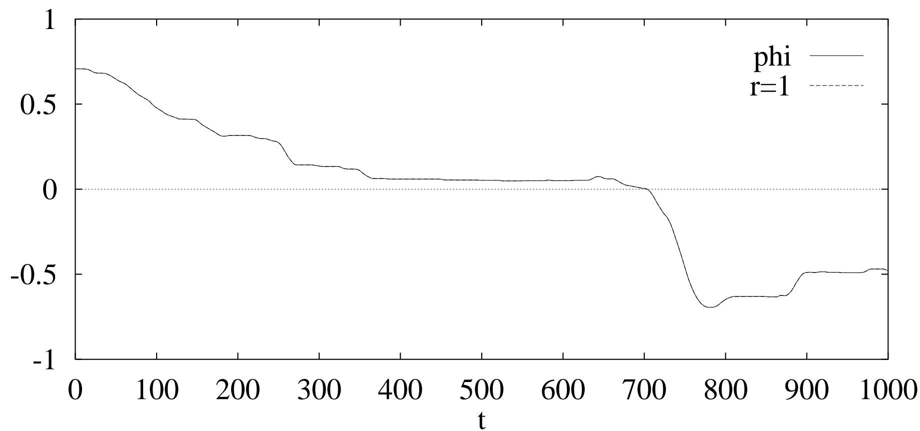


Abbildung 4.36: Sich mit der Originalkurve überdeckende Prädiktion des linearen Modells 2.

Die gezeigte Prädiktion wurde mit dem linearen Modell

$$\texttt{R}_1\texttt{: if TRUE then } f_1 = -0,00003 + 0,00007M_{t-1} + 0,00110M_{t-2} - 0,00062M_{t-3} + 2,54062\varphi_{t-1} - 2,10022\varphi_{t-2} + 0,55952\varphi_{t-3}$$

erreicht und kann somit die Nichtlinearitäten wie die Totzone und die Schwerkraft nicht modellieren. Trotzdem ist hier die Prädiktion fast perfekt, so daß der unscharfe Modellierungsalgorithmus, der die Prädiktion optimiert, erst gar nicht zum Zuge kommen kann. Gebraucht werden also simulierte Meßdaten, bei denen die Nichtlinearitäten auch bei der Prädiktion deutlich zum Vorschein kommen. Ein vielversprechender Ansatz wäre, mit der Winkelgeschwindigkeit $\dot{\varphi}$ zu arbeiten. In der Totzone ist sie gleich Null und auch kleine Winkeländerungen werden durch $\dot{\varphi} \neq 0$ deutlich. Auf den Winkel φ kann in der Eingabe aber nicht ver-

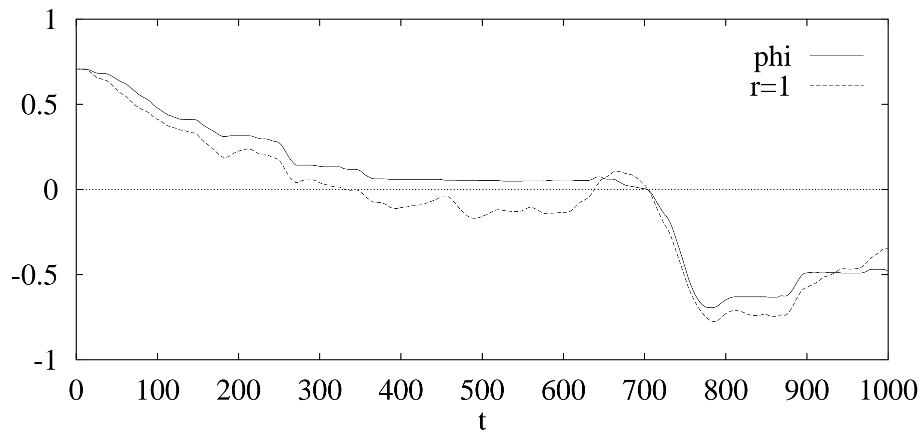


Abbildung 4.37: Simulation des linearen Modells 2 (gestrichelte Linie).

zichtet werden, da die Schwerkraft alleine von ihm abhängt. Für die Simulation mit Winkel *und* Winkelgeschwindigkeit als Eingangs- und Ausgangsgrößen ist jedoch die Modellierung eines MIMO[3] Systems erforderlich. Hierzu müßte das Modellierungsprogramm noch erweitert werden.

Eine andere Möglichkeit ist das Modellieren der Simulation anstatt der Prädiktion, das heißt, schon bei der Modellierung wird das Parallel-Modell (siehe Kapitel 3.6) verwendet. Hierfür kann allerdings RPROP nicht benutzt werden, da aufgrund der Rekursivität beim Parallel-Modell ein rekursives Optimierungsverfahren benötigt wird. Dies würde umfangreiche Erweiterungen am Programm erfordern.

Eine dritte Möglichkeit ist, die Struktur des Modells teilweise von Hand vorzugeben. Das Vorwissen, in diesem Fall das Vorhandensein einer Totzone, kann dadurch modelliert werden, daß für den Bereich der Totzone im Eingaberaum eine Regel von Hand angegeben wird. Ausgehend von solch einem mit Vorwissen versehenen Initialmodell kann das automatische Modellierungsverfahren die Parameter optimieren und auch eventuell weitere Verfeinerungen an den Regeln vornehmen.

[3]Multiple Input, Multiple Output.

Kapitel 5

Schlußwort

In dieser Arbeit wurde, ausgehend vom Modell von Sugeno, ein unscharfes Modellierungsverfahren entwickelt. Hierzu wurde RPROP als effizientes und relativ einfaches nichtlineares Optimierungsverfahren auf das vorliegende Problem angewandt. Die erforderlichen Formeln für den Optimierungsalgorithmus wurden hergeleitet und das gesamte Modellierungsverfahren in C++ implementiert und getestet.

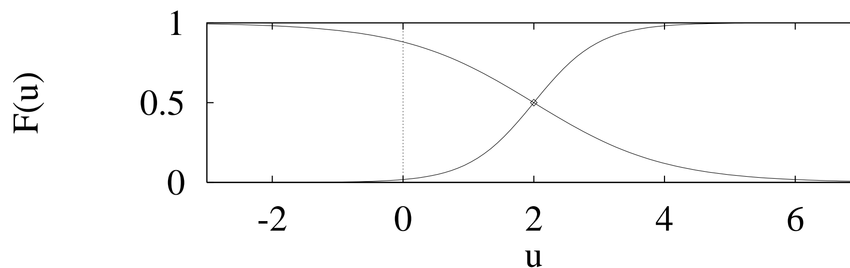


Abbildung 5.1: Gemeinsames μ bei $2,0$.

Bei Tests und Versuchen mit realen Meßdaten zeigte sich, daß die unscharfen Mengen manchmal so weit divergierten, daß nicht mehr für alle Eingaben eine Regel ansprach, das heißt, die Überdeckung des gesamten Eingaberaumes ging verloren. Da dies nur in seltenen Fällen und nur bei sehr langem Optimieren mit RPROP auftrat, wurde hier die einfachste Lösung gewählt: Bei einem solchen Fall wurde die Optimierung abgebrochen und das letzte Modell mit noch vollständiger Überdeckung behalten. Eine anderer Möglichkeit wäre, die vollständige Überdeckung zu erzwingen. Dies kann zum Beispiel dadurch geschehen, daß benachbarte Mengen wie in Abbildung 5.1 ein gemeinsames μ erhalten. Daß dadurch Parameter verloren gehen, kann durch allgemeinere unscharfe Mengen mit einem zusätzlichen Parameter a wie in Gleichung (5.1) wieder ausgeglichen wer-

den. Dieser Parameter kann dann als Gewichtung der Regel interpretiert werden.

$$F(u) \quad := \quad a \cdot \frac{1}{1 + e^{\sigma \cdot (u - \mu)}}, \qquad a \in [0;1] \tag{5.1}$$

Unerwartet war auch, daß mit dem in [TS85, SK88] vorgeschlagenen Kriterium UC keine guten Ergebnisse erzielt wurden. Das lag daran, daß dabei die Optimierung immer zu früh abgebrochen wurde. Die Verwendung von R^2 hat sich als Lösung dieses Problems bewährt.

Das hier vorgestellte Modellierungsverfahren war ursprünglich für statische Systeme gedacht. Bei dynamischen Systemen bestimmt das Optimierungsverfahren die Parameter hinsichtlich einer optimalen Prädiktion. Die so berechneten Modelle wurden auch zur Simulation verwendet (zum Beispiel erfolgreich beim Gasofen von Box/Jenkins). Trotz guter Prädiktion versagte die Simulation jedoch beim Beispiel der Karlsruher Hand. Außer der Wahl von geeigneteren Beispielmengen bietet hier die Erweiterung um rekursive Optimierungsverfahren eine Verbesserungsmöglichkeit.

Eine Möglichkeit zur Beschleunigung des gesamten Verfahrens liegt in der Anwendung einer noch restriktiveren Heuristik, um den Suchraum für die Modellstruktur weiter einzuschränken. Beispielsweise können Eingänge, die mehrmals zu nur unterdurchschnittlichen Modellverbesserungen beigetragen haben, ganz aus der Suche ausgeschlossen werden. Da die Strukturbestimmung und die Parameteroptimierung getrennt behandelt werden können, sind auch ganz andere Verfahren, zum Beispiel auch genetische Algorithmen, zur Strukturbestimmung anwendbar.

Zusammenfassend läßt sich sagen:

- Das Verfahren läßt sich durch unscharfe Mengen mit gemeinsamen μ, einer komplizierteren und effizienteren Heuristik oder ganz anderer Algorithmen zur Strukturbestimmung, und rekursiver Optimierungsverfahren für die Simulation erweitern und damit sicherlich noch verbessern und beschleunigen.

- Eine höhere Regelanzahl erhöht die Genauigkeit, erschwert aber eine Interpretation des gefundenen Modells. Das Hinzufügen von Regeln führt, bis ein gewisses Optimum erreicht wird, natürlich auch zu einer Verbesserung der Genauigkeit. Ein Modell mit vielen Regeln ist aber, aufgrund der teilweisen Überlagerungen in den unscharfen Zonen, dem Menschen schwerer verständlich.

- Zur Verbesserung der Interpretierbarkeit der gefundenen Modelle kann statt einer Darstellung wie in Abbildung 4.11 zu linguistischen Aussagen wie „`if` u_1 `is` *small* `then` ...“ übergegangen werden.

- Der Algorithmus unterscheidet relevante und irrelevante Eingangsvariablen. Bei den Versuchen des Benchmarks, dem Gasofen von Box-Jenkins und der Papiermaschine wurden jeweils die stark mit dem Ausgang korrelierten Variablen gefunden.

- Die Leistung des Verfahrens ist mit der Leistung anderer nichtlinearer Modellierungsalgorithmen vergleichbar. Im hier untersuchten Beispiel, dem Benchmark von Frank, war es im globalen Vergleich das beste Verfahren.

Anhang A

Implementierung

Der in Kapitel 3 beschriebene Modellierungsalgorithmus wurde in Standard C/C++ [KR90, Bre94, Str92] implementiert. In den Anhängen A.1, A.2 und A.3 werden Hinweise zur Installation und Bedienung der entwickelten Programme gegeben. Die Anhänge A.4 und A.5 schließlich behandeln die Vorgaben, Entwurfsentscheidungen und einige Aspekte der Programmentwicklung; sie zu lesen ist für die Installation oder Benutzung der Programme nicht notwendig.

A.1 Installation

Die zur Installation benötigten Dateien haben folgenden Inhalt:

- `fuzzsets.h` enthält die Klassendeklarationen und einige inline Funktionsdefinitionen für die Implementierung eines unscharfen Systems vom Typ Sugeno. Es werden Operationen wie das Kopieren, die Inferenz oder die Parameteroptimierung mit RPROP zur Verfügung gestellt. Die Datei ist in Anhang B.2 aufgeführt.

- In `fuzzsets.cc` sind die in `fuzzsets.h` deklarierten Methoden ausformuliert.

- `fzyident.cc` enthält das Hauptprogramm, einen Parser für die Programmoptionen sowie Funktionen für die Modellierung, Prädiktion, etc. Es werden die Klassen und Methoden aus `fuzzsets.h` verwendet.

- `fzy2pixl.cc` enthält eine Routine zum Parsen von `.fzy`-Dateien und gibt eine graphische Darstellung der unscharfen Mengen in Dateien aus. Es werden teilweise Operationen aus `fuzzsets.h` aufgerufen.

- Das `Makefile` definiert die Vorschriften, Optionen und Pfade zum Kompilieren der Dateien und Linken der Objekte zu ausführbaren Programmen. Ein Listing des Makefiles ist in Anhang B.1 zu finden.

Die benötigten und eventuell vorher zu installierenden Bibliotheken sind:

- `newmat08`; eine C++ Mathematikbibliothek. Sie bietet die arithmetischen Grundoperationen +, - und * für verschiedene Matrizen- und Vektortypen und weitere Operationen wie zum Beispiel Matrixdekomposition mit SVD. In C++ unter Verwendung der Bibliothek `newmat08` wird der Parametervektor $\vec{p}$ aus Gleichung (3.19) durch folgendes Programmstück berechnet:

```
Matrix U, V;
DiagonalMatrix D;
SVD(A, D, U, V);
ColumnVector p = V * (D.i() * (U.t() * y));
```

- `libg++`; die GNU C++ Bibliothek enthält Klassen für Listen, Hashtabellen, etc.

- `libm`, `libiostream`; C/C++ Standardbibliotheken.

Die notwendigen Bibliotheken `newmat08` und `libg++` und die Compiler `gcc/g++` sind für fast alle Rechner mit Unix-Betriebssystem verfügbar. Da die selbst entwickelten Programme keine maschinenspezifischen oder graphischen Operationen verwenden, sollten sie problemlos auch auf andere Unix-Plattformen übertragen werden können.

Die Installation der Programme für Unix Systeme sieht wie folgt aus:

- Installiere die Bibliothek `newmat08`. Siehe hierzu die beiliegende Installationsbeschreibung in [Dav95]. Setze dabei die gewünschte Genauigkeit: Gerechnet wird mit Variablen des Typs `Real`, der entweder gleich `float` oder `double` gesetzt werden kann.

- Falls nötig, installiere `gcc/g++` und die C++ Bibliothek `libg++`. Weitere Informationen geben [Sta94] und [Lea94].

- Kopiere dann die fünf Dateien `fuzzsets.h`, `fuzzsets.cc`, `fzyident.cc`, `fzy2pixl.cc` und `Makefile` in ein Verzeichnis.

- Rufe `make` auf.

Dabei können zum Beispiel folgende Probleme auftreten: Falls das Programm `make` das Format des Makefiles nicht kennt, so muß als Abhilfe zum Beispiel `gmake` installiert werden. Werden beim Kompilieren Header-Dateien nicht gefunden, so muß im Makefile `IPATH` auf die Pfade der entsprechenden Dateien gesetzt werden. Werden beim Linken Funktionen nicht gefunden, dann ist im allgemeinen auch ein fehlender Pfad die Ursache. Als Abhilfe ist `LPATH` im Makefile auf die Pfade der benötigten Bibliotheken, insbesondere auf die Bibliothek `nmat08`, zu setzen.

Getestet wurden die Programme für folgende Konfigurationen:

- Sun Sparc IPX unter SunOS 4.1.1, gcc 2.6.3, g++-lib 2.5.3.
- PC unter Linux 1.1.18 (Slackware 2.0.1), gcc 2.5.8.

Die Bibliotheken `newmat08` und `libg++` sowie der `gcc/g++` Compiler sind auch für PC unter MS-DOS verfügbar. Eine Portierung für MS-DOS sollte also möglich sein, sie wurde aber bisher nicht durchgeführt. Eine Ausführung des Programms `fzy2pixl` wird unter MS-DOS aufgrund der stark eingeschränkten Länge der Dateinamen nicht möglich sein.

A.2 Benutzung von `fzyident`

Das Programm wird durch Eingabe von `fzyident <Optionen>` direkt von einer Unix-Shell aus aufgerufen. Ausgehend von Matlab heißt der entsprechende Aufruf `! fzyident <Optionen>`. Die Ein- und Ausgabe von Daten geschieht ausschließlich über Textdateien im ASCII-Format. Die Ausgabedateien haben die Suffixe

- `.out` für Prädiktionen und Simulationen. Sie enthalten in der i. Zeile den Wert $\hat{y}(\vec{u}_i)$.
- `.fzy` für berechnete unscharfe Modelle. Sie enthalten die vollständige Information, sowohl über die Struktur, als auch über die Parameter eines unscharfen Modells.

A.2.1 Format der Ein- und Ausgabedateien

Format der Eingabedateien

Die Eingabedateien können beliebige Namen haben. Die Datei selbst muß im Textformat die Eingänge u_1 bis u_N als Spalten und als letzte Spalte y enthalten.

Das Dateiformat ist somit

$$\begin{array}{lllll} u_{11} & u_{21} & \cdots & u_{N1} & y_1 \\ \vdots & \ddots & & & \\ u_{1i} & u_{2i} & \cdots & u_{Ni} & y_i \\ \vdots & & & & \\ u_{1m} & u_{2m} & \cdots & u_{Nm} & y_m \end{array},$$

wobei u_{ji} die j. Eingangsvariable des i. Beispiels ist. Als Trennzeichen sind beliebig viele Leerzeichen oder Tabulatorzeichen erlaubt. Die Tabelle darf keine Lücken enthalten. Die Eingaben u_1 bis u_N müssen *normiert* vorliegen. Sie sollten, jeweils mit Mittelwert 0, ungefähr im Intervall $[-0,5; 0,5]$ liegen. Der Grund hierfür liegt darin, daß bei der Modellierung die initialen Modelle von dieser Normierung ausgehen und die Optimierung mit RPROP dann leichter vonstatten geht.

Sollen später auch Simulationen durchgeführt werden und tauchen *order* zeitversetzte y-Werte, also $y^{t-1}, \ldots, y^{t-order}$ als Eingangsvariablen auf, so müssen diese unbedingt als letzte Spalten angegeben werden. Das Format der Eingabedatei ist somit allgemein:

$$\begin{array}{llllll} u_{11} & \cdots & u_{j1} & y_1^{t-1} & \cdots & y_1^{t-order} & y_1^t \\ \vdots & \ddots & & & & & \\ u_{1i} & \cdots & u_{ji} & y_i^{t-1} & \cdots & y_i^{t-order} & y_i^t \\ \vdots & & & & & & \\ u_{1m} & \cdots & u_{jm} & y_m^{t-1} & \cdots & y_m^{t-order} & y_m^t \end{array}$$

Format der Ausgabedateien `.fzy`

Nach Angabe der Anzahl der unscharfen Regeln und Mengen und der Dimension der Konsequenz[1] erscheinen zeilenweise zuerst die unscharfen Mengen, dann die unscharfen Regeln. Die i. unscharfe Menge hat die Form:

```
Fi xm sigma grad_xm grad_sigma old_grad_xm old_grad_sigma delta_xm delta_sigma
```

Interessant für den Benutzer sind hierbei lediglich $\mu = \texttt{xm}$ und $\sigma = \texttt{sigma}$. Die anderen Werte sind die Gradienten $\frac{\partial \|\vec{\varepsilon}^{\,t}\|^2}{\partial \mu_i}$, $\frac{\partial \|\vec{\varepsilon}^{\,t}\|^2}{\partial \sigma_i}$ und $\frac{\partial \|\vec{\varepsilon}^{\,t-1}\|^2}{\partial \mu_i}$, $\frac{\partial \|\vec{\varepsilon}^{\,t-1}\|^2}{\partial \sigma_i}$ und das $\Delta_{\mu_i}^{t-1}$ und $\Delta_{\sigma_i}^{t-1}$. Sie sind aber für ein eventuelles Weiteroptimieren mit RPROP notwendig.

[1] Diese Angaben sind vor dem eigentlichen Einlesen des Modells für die Speicherplatzreservierung notwendig. Sie sind redundant und könnten aus den übrigen Informationen gewonnen werden, wurden aber eingeführt, um einen zweiten Parserdurchlauf einzusparen.

Die i. unscharfe Regel hat die Form:

```
Ri  fcl1 ... fcln  q0 grad_p0 old_grad_p0 delta_p0  ...  pN grad_pN old_grad_pN delta_pN
```

Nach der Regelnummer kommt zuerst eine Liste aller n Klauseln `fcl1 ... fcln` der Form `Xj is Fk` und dann $N+1 =$ `cons_dim` mal die Konsequenzparameter p_0 bis p_N, jeweils wieder mit den Gradienten zur Iteration t und $t-1$ und dem Δ_p. Beispielsweise hat die folgende Regelbasis[2]

$$\mathtt{R}_1 \ \mathtt{if}\ u_1\ \mathtt{is}\ F_0\ \mathtt{and}\ u_2\ \mathtt{is}\ F_2\ \mathtt{then}\ f_1 = 0,000 + 0,000 \cdot u_1 + 0,207 \cdot u_2$$
$$\mathtt{R}_2 \ \mathtt{if}\ u_1\ \mathtt{is}\ F_1\ \mathtt{then}\ f_2 = -0,003 - 0,224 \cdot u_1 + 0,024 \cdot u_2$$
$$\mathtt{R}_3 \ \mathtt{if}\ u_1\ \mathtt{is}\ F_0\ \mathtt{and}\ u_2\ \mathtt{is}\ F_3\ \mathtt{then}\ f_3 = 0,000 + 0,000 \cdot u_1 + 1,043 \cdot u_2$$

damit folgendes Aussehen als `.fzy`-Datei:

```
n_frules 3
n_fsets 4
cons_dim 3
F0 -0.248526 -70.511864 0.000000 0.000000 0.000000 0.000000 0.000121 0.895795
F1 -0.434989 7.233281 0.000000 0.000000 0.000001 -0.000000 0.000005 0.010745
F2 -0.000445 -16.997019 0.000000 0.000000 0.000000 0.000000 0.000535 0.128367
F3 -0.023399 34.532864 0.000000 0.000000 -0.000000 0.000000 0.000093 0.321440
R0 X1 is F0  X2 is F2  0.000041 0.000000 -0.000686 0.000000  0.000114 0.000000
 -0.000138 0.000000   0.207084 0.000000 0.000045 0.000000
R1 X1 is F1  -0.002782 0.000000 -0.000337 0.000000  -0.224107 0.000000 0.000080
 0.000013  0.024079 0.000000 0.000080 0.000001
R2 X1 is F0  X2 is F3  0.000225 0.000000 -0.002433 0.000001  0.000013 0.000000
 0.000496 0.000001  1.043288 0.000000 0.000201 0.000003
```

A.2.2 Optionen beim Programmaufruf

Bei Aufruf von `fzyident` ohne Parameter gibt das Programm folgende Hilfe aus:

```
NAME
     fzyident - fuzzy identification of a Sugeno type fuzzy model

SYNOPSIS
     fzyident {-estim || -sim || -ident || -model} -f1 <file_a> -f2 <file_b> -ex <n>
         -xdim <n> [-f3 <file_c>] [-cdim <n>] [-imp <n>] [-frules <n>] [-minopt <n>]
         [-maxopt <n>]  [-optbest <n>]   [-optlast <n>]   [-gnuplot1 || -gnuplot2 ||
         -matlab1 || -matlab2] [-info <n>] [-n_ext <str>] [-nlines <n>] [-uc || -r2]
         [-ord <n>]

OPTIONS
     -estim         set mode = estimation;     needs: f1 = model.fzy, f2 = data
     -sim           set mode = simulation;     needs: f1 = model.fzy, f2 = data, -ord order
     -ident         set mode = identification; needs: f1 = model.fzy, f2 = dataA, f3 = dataB
     -model         set mode = modelisation;   needs: f1 = dataA,     f2 = dataB
     -f1 <file_a>   input file
     -f2 <file_b>   input file
     -f3 <file_c>   input file
     -ex <n>        number of examples, <n> > 0
```

[2]Hier mit gerundeten Koeffizienten.

```
     -xdim <n>     dimension of input space, <n> > 0
     -cdim <n>     dimension of rule consequences, 1 <= <n> <= xdim+1, default: xdim+1
     -imp <f>      minimal improvement rate, <f> <= 100.0, default: 5.0
     -frules <n>   maximal number of fuzzy rules, 1 <= <n> <= 100, default: 10
     -minopt <n>   minimal candidate optimization iterations, <n> > 0, default: 5
     -maxopt <n>   maximal candidate optimization iterations, <n> > 0, dafault: 200
     -optbest <n>  additional optimization of each best model, <n> >= 0, default: 0
     -optlast <n>  additional optimization of the last model, <n> >= 0, default: 0
     -uc           use UC criterion to stop modelisation and identification
     -r2           use R2 criterion to stop modelisation and identification (default)
     -gnuplot1     set output format to 1-dimensional gnuplot
     -gnuplot2     set output format to 2-dimensional gnuplot
     -matlab1      set output format to 1-dimensional matlab (default)
     -matlab2      set output format to 2-dimensional matlab
     -info <n>     information level: 0: no information
                                      1: minimal information, print only last model
                                      2: print best model of each epoch (default)
                                      3: print all candidates
     -n_ext <str>  name extension for savefile names, <str> must be a string
     -nlines <n>   save an empty line every <n> lines, <n> >= 0, default: 0 (no empty lines)
     -ord <n>      order of the y data for simulation

DESCRIPTION
     fzyident searches iterativly an optimal structure and the parameters
     of a Sugeno type fuzzy model.
     [-a <n>]: parameter -a <n> is optional
     { -a || -b}: choose either -a or -b
...
```

Zwingende Parameter

Zu den zwingend notwendigen Parametern gehört die Auswahl des Modus: `-estim` für Prädiktion, `-sim` für Simulation, `-ident` für Identifikation oder `-model` für Modellierung. Außerdem die Angabe der Eingabedateien nach `-f1` und `-f2` und die Anzahl der darin enthaltenen Beispiele `-ex` und deren x-Dimension `-xdim`[3]. Die vier Betriebsmodi werden in Kapitel A.2.3 näher erläutert. Sie haben teilweise zusätzliche notwendige Parameter.

Zusätzliche Optionen

Zusätzliche Optionen sind:

- `cdim` bestimmt die Dimension der Regelkonsequenzen. Übliche Werte sind 1 oder `xdim`+1.
- `imp`[4] wird zur Terminierung der heuristischen Suche benutzt. Es bezeichnet die geforderte minimale Verbesserung des Gütekriteriums in Prozent von einer Epoche zur anderen. Bei Angabe eines negativen Wertes werden auch Verschlechterungen zugelassen.

[3] Da immer MISO-Systeme betrachtet werden, d.h. y ist immer eindimensional, ist `xdim` = Anzahl der Spalten − 1.

[4] Abgekürzt für improvement.

- `frules` dient wiederum zur Terminierung des Suchalgorithmus. Wird trotz wiederholter Verbesserung um `imp` Prozent die Anzahl von `frules` Regeln erreicht, so wird der Algorithmus beendet. Vor allem, wenn durch negatives `imp` auch Verschlechterungen zugelassen sind, sollte `frules` nicht zu groß gewählt werden, damit das Verfahren in vertretbarer Zeit terminiert.

- `minopt` bezeichnet die minimale Iterationszahl bei der Optimierung mit RPROP. Nach `minopt` Iterationen wird RPROP noch solange ausgeführt bis keine Verbesserung[5] mehr eintritt. Diese Option hat im allgemeinen wenig Einfluß auf das Endergebnis.

- `maxopt` ist die maximale Iterationszahl für RPROP. Der eingestellte Vorgabewert von 200 wird sehr selten erreicht und soll nur eine Terminierung sicherstellen. Diese Option braucht im allgemeinen nicht geändert werden.

- `optbest` gibt die Anzahl von RPROP Iterationen an, mit der das jeweils beste Modell einer Epoche weiter optimiert wird. So können, fast ohne zusätzlichen Zeitaufwand, die Parameter der weiterhin verwendeten Modelle zusätzlich optimiert werden. Diese Option wurde während der in Kapitel 4 vorgestellten Versuche nicht verwendet.

- `optlast` ist wie `optbest`, bezieht sich aber nur auf das vom Algorithmus als bestes erachtete Modell.

- `uc` wählt das UC-Kriterium zur Terminierung des heuristischen Suchalgorithmus.

- `r2` wird wie `uc` verwendet, wählt dafür aber das R^2-Kriterium.

- `gnuplot1` bewirkt dasselbe Ausgabeformat wie `matlab1`.

- `gnuplot2` wählt das zweidimensionale Ausgabeformat für Gnuplot. Es enthält spaltenweise zuerst alle x, dann als letzte Spalte $\hat{y}$. Leerzeilen geben den Anfang neuer Linien an.

- `matlab1` stellt die eindimensionale Ausgabe für Matlab und Gnuplot ein. Hier wird $\hat{y}$ als Spalte in die Ausgabedatei geschrieben.

- `matlab2` wählt das zweidimensionale Format für Matlab. Hierbei wird $\hat{y}$ als Matrix in die Ausgabedatei geschrieben. Bei allen vier Optionen zum Ausgabeformat geschieht das Schreiben von Werten im ASCII-Format.

- `info` gibt an, wieviele Informationen während des Programmlaufs auf dem Bildschirm dargestellt werden.

[5] RPROP benutzt den quadrierten Fehler als Gütekriterium. Siehe auch Kapitel 3.2.

- `n_ext`[6] bezeichnet die Namenserweiterung der Ausgabedateien. Dieser Name wird dann noch um die Epoche und die Art der Datei erweitert. Beispielsweise erhält das Modell der zwölften Epoche die Erweiterung `_12.fzy`.
- `nlines` wird nur für `gnuplot2` benötigt. Es bezeichnet die Anzahl der Zeilen, nach der jeweils eine Leerzeile ausgegeben werden soll.
- `order` gibt die Ordnung von y bei der Simulation an. Es bewirkt, daß $y^{t-1}, \ldots, y^{t-order}$ durch Kopieren von $\hat{y}^t$ als Eingabe verwendet werden.

A.2.3 Betriebsmodi

Die vier möglichen Betriebsarten sind:

Prädiktion

Hier wird das in `-f1` gegebene unscharfe Modell gelesen und dann eine Prädiktion auf die Daten in der zweiten Datei berechnet. Die durch `-f2` angegebene Datei muß, wie alle Eingabedateien, neben den Spalten für u_1 bis u_N, eine letzte Spalte für y enthalten. Diese kann sinnlose Werte enthalten, da sie für die Prädiktion nicht benötigt wird, sie muß aber vorhanden sein.

Simulation

Bei der Simulation muß zusätzlich zur Prädiktion die Ordnung, das heißt die Anzahl der zeitversetzten y, angegeben werden, da diese während der Simulation durch die Rückkopplung weiterverwendet werden. Es ist auf die richtige Anordnung der Eingabegrößen in der Eingabedatei zu achten (siehe Kapitel A.2.1). Eine *gleiche Anordnung* der Größen in der Eingabedatei während der Modellierung und der Simulation ist unbedingt erforderlich.

Modellierung

Hier werden zwei Dateien mit Beispieldaten benötigt. Es wird dann, ausgehend vom linearen Modell mit einer Regel, schrittweise ein immer genaueres unscharfes Modell berechnet. Zur Optimierung mit RPROP wird der erste, zur Berechnung der Modellgüte mit R^2 der zweite Datensatz verwendet. Durch Angabe derselben Datei bei `-f1` und `-f2` wird derselbe Datensatz für beides benützt. Während der Modellierung wird bei jeder Epoche i das unscharfe Modell in eine Datei

[6] Abkürzung für „name extension“.

`name_i.fzy` und die Prädiktion für den *zweiten* Datensatz in `name_i.out` geschrieben. Außerdem schreibt das Programm bei jeder Epoche den R^2-Wert für diese Prädiktion in eine Datei `name_.r2`. Mit Hilfe dieser Datei kann zum Beispiel das Fortschreiten der Modellierung während des Rechenvorgangs beobachtet werden. Der Aufruf einer Modellierung ist beispielsweise:

```
fzyident -model -f1 dat_a -f2 dat_b -ex 500 -xdim 6 -cdim 7
```

Identifikation

Bei der Identifikation wird zusätzlich zu den zwei Datendateien eine Datei `name.fzy` mit einem unscharfen Modell angegeben. Für dieses Modell werden dann die Parameter optimiert. Diese Betriebsart ist dazu gedacht, die Struktur eines Modells von Hand vorzugeben und dann mit Hilfe des Programms lediglich seine Parameter zu bestimmen.

A.3 Benutzung von fzy2pixl

Das Programm `fzy2pixl` dient zur Visualisierung der von `fzyident` erzeugten unscharfen Modelle, wie sie in `.fzy` Dateien abgespeichert werden.

Mit dem Aufruf `fzy2pixl <name.fzy>` wird zuerst das Modell geladen. Das Ergebnis des Parserdurchlaufs wird dann zur Kontrolle auf dem Bildschirm ausgegeben. Schließlich wird für jede Regel i und jede in ihr vorkommende Variable x_j eine Datei `Name.ri_xj` erzeugt, die den Verlauf der Zugehörigkeitsfunktion von x_j angibt. Die Ausgabedatei enthält hierzu zwei Spalten mit jeweils 100 Werten, in der ersten $x \in [-1; 1]$ und in der zweiten der Verlauf der Zugehörigkeitsfunktion. So ergibt zum Beispiel das Modell

```
F0 -0.258681 -15.484666 0.000000 0.000000 -0.318781 -0.002386 0.001854 0.096275
F1 -0.186870 23.180916 0.000000 0.000000 -0.944356 0.001175 0.001288 0.257939
F2 0.205322 -28.277918 0.000000 0.000000 0.959000 -0.002266 0.000519 0.373248
F3 0.283004 21.230915 0.000000 0.000000 -0.064295 -0.000686 0.000716 0.288882
R0 X1 is F0  X1 is F3  0.267204 0.000000 0.010138 0.001280
R1 X1 is F1  -0.246244 0.000000 -0.409665 0.000222
R2 X1 is F2  -0.210628 0.000000 0.820216 0.001280
```

die Dateien `name.r1_x1`, `name.r2_x1` und `name.r3_x1`. Die drei Dateien enthalten die Kurven, die zur Veranschaulichung der unscharfen Mengen in Abbildung 4.12 verwendet werden.

A.4 Vorgaben

Bei der Erstellung der Programme waren einige Vorgaben einzuhalten. Erforderliche Programmeigenschaften waren:

- Anwendbarkeit für *große Eingabedimensionen* N und *umfangreiche Datensätze* (großes m). Diese Vorgabe hatte vor allem Einfluß auf die Wahl des Algorithmus. Algorithmen, die zum Beispiel mit 2^N Regeln arbeiten, mußten von vornherein ausgeschlossen werden. Auch ist es für große N nicht möglich, den Raum aller Partitionierungsmöglichkeiten vollständig abzusuchen. Deshalb wurde die heuristische Suche wie sie Sugeno verwendet gewählt.

- *Leichte Bedienbarkeit*. Um dies zu gewährleisten, wurde ein Optimierungsverfahren gesucht, das im allgemeinen ohne Angabe zusätzlicher Parameter auskommt. Die Bedienung des Programms geschieht durch Parameter in der Kommandozeile, von denen die meisten optional sind und für übliche Anwendungen nicht gebraucht werden.

Zusätzlich waren noch folgende gewünschten Eigenschaften soweit wie möglich zu berücksichtigen:

- *Laufzeiteffizienz*. Aufgrund seiner vielfältigen Datentypen und der relativ guten Laufzeiteffizienz wurde C/C++ als Programmiersprache gewählt. Die Verwendung vorgefertigter Spezialroutinen, wie sie zum Beispiel Matlab zur Verfügung stellt, konnte teilweise durch C/C++ Bibliotheken erreicht werden.

- *Erweiterbarkeit*. Um die Programme leichter erweitern zu können, wurde das unscharfe Modell und die Operationen hierfür gesondert in die Dateien `fuzzsets.cc` geschrieben. Damit können neue Anwendungen, die das unscharfe Modell benutzen, einfach die in `fuzzsets.h` deklarierten Klassen und Methoden verwenden. Auch das Klassenkonzept von C++ trägt zur Erweiterbarkeit bei.

- Benutzung mit *Matlab*. Um die Ergebnisse mit Matlab und anderen Graphikwerkzeugen wie zum Beispiel *Gnuplot* verwenden zu können, findet die Ausgabe der Ergebnisse in Dateien im Standard-ASCII-Format statt.

- Für Entwurf, Programmierung und Test waren *5-8 Wochen* vorgesehen.

A.5 Entwurf und Programmierung

In diesem Kapitel wird kurz auf den Entwurf und die Programmierung eingegangen. Das während der Programmierung verwendete Makefile ist in Anhang B.1 gelistet. Die erste Entwurfsentscheidung war, das unscharfe System von seinen Anwendungen zu trennen. Dies sollte sich später als vorteilhaft erweisen, denn außer der Verwendung zur Modellierung in `fzyident` wurde es später auch für eine zweite Anwendung, der Visualisierung mit `fzy2pixl`, gebraucht.

Das Hauptprogramm `fzyident`

Das Hauptprogramm enthält einen Parser für die Kommandozeile, eine Routine zur Ausgabe von Fehlermeldungen und einer Hilfe, sowie der Funktionen zur Ausführung der vier Betriebsarten, von denen die Modellierung die aufwendigste ist. Sie ist eine Ausformulierung des Schemas aus Abbildung 3.5 und verwendet dabei Methoden aus `fuzzsets.h` wie zum Beispiel `FRuleBaseSugeno::RPROP(...)` für eine RPROP-Iteration.

Das unscharfe System in `fuzzsets.h`

Ein unscharfes System besteht aus folgenden drei Teilen: Der *Regelbasis* (die Zusammenfassung aller unscharfen Regeln), den unscharfen *Mengen* und dem *Inferenzmechanismus*. Diese Einteilung soll im Entwurf wiedergegeben werden. Aus Gründen der Einfachheit wird dabei auf die Erstellung von Basisklassen und Vererbung verzichtet und stattdessen mit `typedef` gearbeitet (siehe Anhang B.2). Dies ist nach [Str92] Kapitel 11 und 12 dann angebracht, wenn wie hier keine großen Änderungen zu erwarten sind. Die Eingaben liegen nicht als unscharfe Mengen, sondern als scharfe Werte vor und können direkt verarbeitet werden. Die Schärfung des Ergebnisses, hier mittels des Schwerpunktes, ist als Methode der Regelbasis implementiert.

Folgende Klassen wurden deshalb in `fuzzsets.h` entworfen:

- `FSet` ist die Klasse für die Unscharfe Menge. Im weiteren wird die Ausformulierung `FSetSugeno` verwendet, die neben den beiden Parametern $\mu =$ `xm` und $\sigma =$ `sigma` auch die für den Gradientenabstieg beziehungsweise RPROP notwendigen Gradienten und Deltawerte für μ und σ enthält. Das Setzen und Lesen der Werte geschieht über entsprechende Methoden. Um Rechenzeit zu sparen ist die sigmoide Zugehörigkeitsfunktion als Tabelle implementiert.

- `FSetBase` ist die Menge aller unscharfen Mengen. Implementiert wurde sie als Vektor von unscharfen Mengen. Als Methode wird zum Beispiel, unabhängig von der Art der unscharfen Mengen, eine Dereferenzierung mit

(aus Effizienzgründen) ausschaltbarer Bereichsüberprüfung zur Verfügung gestellt.

- `FClause` spezifiziert die unscharfen Klauseln. Die Beziehung Eingabevariable – unscharfe Menge wird durch ein Indextupel dargestellt.

- `FRule` ist, hier durch `FRuleSugeno` definiert, die Klasse für unscharfe Regeln. Sie besteht aus einer Liste von unscharfen Klauseln und einem Vektor, der die Konsequenzparameter enthält. Neben Methoden für den Zugriff auf diese Parameter enthält sie auch Methoden zur Suche nach speziellen Klauseln, wie sie bei der iterativen Aufteilung des Eingaberaums gebraucht werden, und eine Methode für die Inferenz.

- `FRuleBase` ist die Menge aller unscharfen Regeln. Es wird hierbei die Spezialisierung `FRuleBaseSugeno` verwendet. Diese enthält einen Vektor von unscharfen Regeln des Typs `FRuleSugeno`. Außer den Dereferenzierungsmethoden und einer Methode `inference_all(...)` für die Inferenz und Schärfung, kommen noch umfangreiche, speziell für das unscharfe System vom Typ Sugeno erforderliche Methoden hinzu. Es sind dies beispielsweise `RPROP(...)`, die eine RPROP-Iteration durchführt, `save_estimation(...)`, die eine Prädiktion berechnet und in eine Datei ausgibt, oder `load(...)`, die ein unscharfes Modell aus einer `.fzy`-Datei lädt.

Wegen der Größe des Quellcodes sind nur die wichtigsten Teile in den Anhängen gelistet. Durch die Verwendung langer und ausführlicher Variablennamen sind die meisten Namen selbsterklärend und auf Kommentare konnte dadurch oft verzichtet werden. Anhang B.3 enthält die Ausformulierungen einiger Methoden aus `fuzzsets.cc`. In Anhang B.4 sind das Hauptprogramm `main(...)` und einige weitere wichtige Funktionen gelistet, zum Beispiel `void modelisation_r2(...)`, die Prozedur, die auch die heuristische Bestimmung der Modellstruktur enthält.

Anhang B

Dateiausdrucke

B.1 Makefile

```
# Makefile to build:
#  - a fuzzy identification tool: fzyident
#  - a tool to visualize fuzzy rule bases .fzy: fzy2pixl
#
# Diplomarbeit, Manfred Maennle, Febr. 95
# last change: June 10th 95

# ***** compiler, compiler options, link options
CC      = g++

#CFLAGS = -Wall
#LFLAGS = -Wall
CFLAGS  = -O2
LFLAGS  = -O2

# ***** librarys to link with
LIBS    = -lnmat08 -liostream -lg++ -lm

# ***** additional include path and library path
IPATH   = -I/usr/local/gnu/libg++/lib/g++-include -I$(HOME)/include/nmat08
LPATH   = -L/usr/local/gnu/libg++/lib -L$(HOME)/lib

# ***** executable programs to make
PRG1    = fzyident
PRG2    = fzy2pixl
PRG3    =

# ***** objects to build
OBJC    = fuzzsets.o                           # common object files
OBJ1    = $(PRG1).o                            # exclusive objects for PRG1
OBJ2    = $(PRG2).o                            # exclusive objects for PRG2
OBJ3    = # $(PRG3).o                          # exclusive objects for PRG3

all :   $(PRG1) $(PRG2) $(PRG3)


# *************** link several objects to executables *******************
$(PRG1):$(OBJC) $(OBJ1)
        $(CC) $(LFLAGS) $(LPATH) $(OBJC) $(OBJ1) -o $(PRG1) $(LIBS)
```

```
$(PRG2):$(OBJC) $(OBJ2)
	$(CC) $(LFLAGS) $(LPATH) $(OBJC) $(OBJ2) -o $(PRG2) $(LIBS)

#$(PRG3):$(OBJC) $(OBJ3)
#	$(CC) $(LFLAGS) $(LPATH) $(OBJ3) -o $(PRG3) $(LIBS)

# ************************* build all objects ****************************
.cc.o:
	$(CC) $(CFLAGS) $(IPATH) -c $< -o $*.o
.c.o:
	$(CC) $(CFLAGS) $(IPATH) -c $< -o $*.o
.c.s:
	$(CC) $(CFLAGS) $(IPATH) -S $< -o $*.s

# *************************** clean disk *********************************
clean :
	rm -f *.o *.s *~ *.bak *.BAK *% core $(PRG1) $(PRG2) $(PRG3)
```

B.2 fuzzsets.h

```
/*
 * fuzzsets.h
 *  - classes to define a sugeno type fuzzy system
 *  - function declarations: t-norm, inference, sigmoid
 *
 * Diplomarbeit, Manfred Maennle, April 95
 * last change: July 4th 95
 *
 */

#ifndef _FUZZSETS_H_
#define _FUZZSETS_H_

extern "C"
{
#include <math.h>      // needs libm
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
}

#include <iostream.h>  // needs libiostream
#include <SLList.h>    // needs libg++
#include <newmat.h>    // needs libnmat08
#include <newmatio.h>  // needs libnmat08
#include <newmatap.h>  // needs libnmat08

///// uncomment the following line to switch off range assertion
#define ASSERT_OFF

#ifndef ASSERT_OFF
  #include <assert.h>
#endif

///// for control outputs
#define PRINT(X) cout << (#X) << " = " << (X) << endl;

///// define product t-norm
inline Real t_norm(Real a, Real b)
{ return (a * b); }

///// define product inference
inline Real inference(Real prem, Real cons)
{ return (prem * cons); }

enum out_format { GNUPLOT1, GNUPLOT2, MATLAB1, MATLAB2 };

/*
 * ********** FSetSigmoid
 */

///// constants for the RPROP optimisation
const Real delta_0_cons = 0.001;
const Real delta_0_xm = 0.001;
const Real delta_max_xm = 0.02;
const Real delta_0_sigma = 0.1;
const Real delta_max_sigma = 2.0;
const Real eta_plus = 1.2;
const Real eta_minus = 0.5;
const Real sigma_max = 200.0;
```

```
///// sigmoidal membership functions
Real sigmoid(Real x);       /// via table
Real exact_sigmoid(Real x); /// via 1/(1+exp(x))

class FSetSigmoid
{
protected:
  Real xm;
  Real sigma;
  Real grad_xm;
  Real old_grad_xm;
  Real grad_sigma;
  Real old_grad_sigma;
  Real delta_xm;
  Real delta_sigma;
public:
  FSetSigmoid()
    { xm = 0.0;
      sigma = 25.0;
      grad_xm = 0.0;
      grad_sigma = 0.0;
      old_grad_xm = 0.0;
      old_grad_sigma = 0.0;
      delta_xm = delta_0_xm;
      delta_sigma = delta_0_sigma;
    }
  FSetSigmoid(Real x, Real s)
    { xm = x;
      if (fabs(s) <= sigma_max)
         sigma = s;
       else
         { if (sigma >= 0)
            sigma = sigma_max;
           else
            sigma = -sigma_max;
         }
      grad_xm = 0.0;
      grad_sigma = 0.0;
      old_grad_xm = 0.0;
      old_grad_sigma = 0.0;
      delta_xm = delta_0_xm;
      delta_sigma = delta_0_sigma;
    }
  void set_xm(Real x)
    { xm = x; }
  void set_sigma(Real s)
    {  if (fabs(s) <= sigma_max)
         sigma = s;
       else
         { if (sigma >= 0)
            sigma = sigma_max;
           else
            sigma = -sigma_max;
         }
    }
  void set_grad_xm(Real x)
    { grad_xm = x; }
  void set_grad_sigma(Real s)
    { grad_sigma = s; }
  void set_old_grad_xm(Real x)
    { old_grad_xm = x; }
  void set_old_grad_sigma(Real s)
    { old_grad_sigma = s; }
  void set_delta_xm(Real x)
    {  if (fabs(x) <= delta_max_xm)
```

```
        delta_xm = x;
      else
        delta_xm = delta_max_xm;
    }
  void set_delta_sigma(Real s)
    {  if (fabs(s) <= delta_max_sigma)
        delta_sigma = s;
      else
        delta_sigma = delta_max_sigma;
    }
  Real get_xm()
    { return xm; }
  Real get_sigma()
    { return sigma; }
  Real get_grad_xm()
    { return grad_xm; }
  Real get_grad_sigma()
    { return grad_sigma; }
  Real get_old_grad_xm()
    { return old_grad_xm; }
  Real get_old_grad_sigma()
    { return old_grad_sigma; }
  Real get_delta_xm()
    { return delta_xm; }
  Real get_delta_sigma()
    { return delta_sigma; }
  Real get_membership(Real x)
    { return sigmoid(sigma * (x - xm)); }
  void print(ostream& strm);
  friend ostream& operator << (ostream& strm, const FSetSigmoid& fset);
};

//typedef FSetGauss PremiseFSet;
typedef FSetSigmoid PremiseFSet;

/*
 * ********** FSetBase
 */

class FSetBase
{
protected:
  PremiseFSet* start;
  int n_fsets;
  int max_n_fsets;
public:
  FSetBase(int maxnfsets);
  ~FSetBase();
  PremiseFSet& operator [] (int fbase_index)
    #ifdef ASSERT_OFF
      { return start[fbase_index]; }
    #else
      { Tracer tr("PremiseFSet& operator [] (int fbase_index)");
        assert(fbase_index>=0);
        assert(fbase_index<n_fsets);
        return start[fbase_index];
      }
    #endif
  PremiseFSet& operator [] (int fbase_index) const
    #ifdef ASSERT_OFF
      { return start[fbase_index]; }
    #else
      { Tracer tr("PremiseFSet& operator [] (int fbase_index) const");
        assert(fbase_index>=0);
        assert(fbase_index<n_fsets);
```

```
        return start[fbase_index];
      }
    #endif
  FSetBase& operator = (const FSetBase& fsetbase);
  int get_n_fsets()
    { return n_fsets; }
  int get_n_fsets() const
    { return n_fsets; }
  void add_fset(PremiseFSet& fset);
  friend ostream& operator << (ostream& strm, const FSetBase& fsetbase);
};

/*
 * ********** FClause
 */

class FClause
{
protected:
  int fvar;
  int fset;
public:
  FClause();
  FClause(const Matrix& inmatrix, const FSetBase& fsetbase, int var, int set);
  int get_fvar()
    { return fvar; }
  int get_fset()
    { return fset; }
  void print(ostream& strm, const FSetBase& fsetbase);
  friend ostream& operator << (ostream& strm, const FClause& fclause);
};

const FClause empty_fclause;

/*
 * ********** FRuleSugeno
 */

class FRuleSugeno
{
protected:
  SLList<FClause> premise;
  ColumnVector consequence;
  ColumnVector grad_cons;
  ColumnVector old_grad_cons;
  ColumnVector delta_cons;
public:
  FRuleSugeno();
  FRuleSugeno(int cons_dim);
  ~FRuleSugeno();
  void ReDimension(int cons_dim);
  FRuleSugeno& operator = (FRuleSugeno& frule);
  int get_n_prem_fclauses()
    { return premise.length(); }
  Pix get_p_first_premise()
    { return premise.first(); }
  void get_p_next_premise(Pix& plist)
    { premise.next(plist); }
  FClause& get_premise_fclause(Pix plist)
    { return premise(plist); }
  int get_cons_dim()
    { return consequence.Nrows(); }
  int get_cons_dim() const
    { return consequence.Nrows(); }
  void add_prem_fclause(FClause& fclause)
```

```
    { premise.append(fclause); }
  Real get_cons_param(int index)
    { return consequence[index]; }
  Real get_grad_cons(int index)
    { return grad_cons[index]; }
  Real get_old_grad_cons(int index)
    { return old_grad_cons[index]; }
  Real get_delta_cons(int index)
    { return delta_cons[index]; }
  void set_cons_param(int index, Real x)
    #ifdef ASSERT_OFF
      { consequence[index]=x; }
    #else
      { Tracer tr("void set_cons_param(int index, Real x)");
        assert(index>=0);
        assert(index<consequence.Nrows());
        consequence[index]=x;
      }
    #endif
  void set_grad_cons(int index, Real x)
    #ifdef ASSERT_OFF
      { grad_cons[index]=x; }
    #else
      { Tracer tr("void set_grad_cons(int index, Real x)");
        assert(index>=0);
        assert(index<grad_cons.Nrows());
        grad_cons[index]=x;
      }
    #endif
  void set_old_grad_cons(int index, Real x)
    #ifdef ASSERT_OFF
      { old_grad_cons[index]=x; }
    #else
      { Tracer tr("void set_old_grad_cons(int index, Real x)");
        assert(index>=0);
        assert(index<old_grad_cons.Nrows());
        old_grad_cons[index]=x;
      }
    #endif
  void set_delta_cons(int index, Real x)
    #ifdef ASSERT_OFF
      { delta_cons[index]=x; }
    #else
      { Tracer tr("void set_delta_cons(int index, Real x)");
        assert(index>=0);
        assert(index<delta_cons.Nrows());
        delta_cons[index]=x;
      }
    #endif
  void remove_prem_fclause(FClause& fclause);
  FClause& get_prem_fclause(int i);
  const FClause& get_existing_left_fclause(FSetBase& fsetbase, int variable_index);
  const FClause& get_existing_right_fclause(FSetBase& fsetbase, int variable_index);
  Real premise_value(const FSetBase& fsetbase, const RowVector& invect);
  Real consequence_value(const RowVector& invect);
  void print(ostream& strm, const FSetBase& fsetbase);
  friend ostream& operator << (ostream& strm, FRuleSugeno& frule);
};

typedef FRuleSugeno FRule;

/*
 * ********** FRuleBaseSugeno
 */
```

```
class FRuleBaseSugeno
{
protected:
  FRuleSugeno* start;
  int n_frules;
  int max_n_frules;
public:
  FRuleBaseSugeno(int maxnfrules, int cons_dim);
  ~FRuleBaseSugeno();
  FRuleSugeno& operator [] (int frulebase_index)
    #ifdef ASSERT_OFF
      { return start[frulebase_index]; }
    #else
      { Tracer tr("FRuleSugeno& operator [] (int frulebase_index)");
        assert(frulebase_index>=0);
        assert(frulebase_index<n_frules);
        return start[frulebase_index];
      }
    #endif
  FRuleSugeno& operator [] (int frulebase_index) const
    #ifdef ASSERT_OFF
      { return start[frulebase_index]; }
    #else
      { Tracer tr("FRuleSugeno& operator [] (int frulebase_index) const");
        assert(frulebase_index>=0);
        assert(frulebase_index<n_frules);
        return start[frulebase_index];
      }
    #endif
  FRuleBaseSugeno& operator = (const FRuleBaseSugeno& frulebase);
  int get_rule_cons_dim()
    { return start[0].get_cons_dim(); }
  int get_n_frules()
    { return n_frules; }
  int get_n_frules() const
    { return n_frules; }
  void add_frule(FRuleSugeno& frule);
  void fit_cons_param_LMS(const FSetBase& fsetbase, const Matrix& X, const Matrix& y);
  Real inference_all(const FSetBase& fsetbase, const RowVector& invect,
                     int info_level, int& exception);
  ///// RPROP returns the error sum[(^y - y)^2]
  Real RPROP(FSetBase& fsetbase, Matrix& U, const Matrix& y,
             FSetGauss& dummy, int info_level, int& exception);
  Real RPROP(FSetBase& fsetbase, Matrix& U, const Matrix& y,
             FSetSigmoid& dummy, int info_level, int& exception);
  void RPROP_backstep(FSetBase& fsetbase, FSetSigmoid& dummy, int& exception);
  Real R2(FSetBase& fsetbase, Matrix& X, Matrix& y, int info_level, int& exception);
  void print(ostream& strm, FSetBase& fsetbase);
  void save_estimation(FSetBase& fsetbase, Matrix& X, char* outfilename, int output_format,
                       int newlines, int info_level);
  void save_simulation(FSetBase& fsetbase, Matrix& X, int order, char* outfilename,
                       int output_format, int newlines, int info_level);
  void save_R2(FSetBase& fsetbase, Matrix& XA, Matrix& ya, char* r2_filename, int info_level);
  ///// dump a model in a .fzy file
  void dump(FSetBase& fsetbase, const char* filename);
  ///// load a model from a .fzy file
  void load(FSetBase& fsetbase, const char* filename, Matrix& X);
  friend ostream& operator << (ostream& outstrm,
                               const FRuleBaseSugeno& frulebase);
};

typedef FRuleBaseSugeno FRuleBase;

#endif /** _FUZZSETS_H_ **/
```

B.3 fuzzsets.cc

```
/*
 * fuzzsets.cc
 *  - class methods for a Sugeno type fuzzy system
 *  - definitions of the methods declared in fuzzsets.h
 *
 * Diplomarbeit, Manfred Maennle, April 95
 * last change: July 8th 95
 *
 */

#include "fuzzsets.h"

/*
 * ********** FSetSigmoid
 */

ostream& operator << (ostream& strm, const FSetSigmoid& fset)
{
  #ifndef ASSERT_OFF
    Tracer tr("ostream& operator << (ostream& strm, const FSetSigmoid& fset)");
  #endif
  strm << '[' << fset.xm << ',' << fset.sigma << ']';
  return strm;
}

/*
 * ********** FSetBase
 */

FSetBase::FSetBase(int maxnfsets)
{
  #ifndef ASSERT_OFF
    Tracer tr("FSetBase::FSetBase(int maxnfsets)");
  #endif
  max_n_fsets = maxnfsets;
  start = new PremiseFSet[max_n_fsets];
  n_fsets = 0;
}

FSetBase::~FSetBase()
{
  #ifndef ASSERT_OFF
    Tracer tr("FSetBase::~FSetBase()");
  #endif
  delete[] start;
}

FSetBase& FSetBase::operator = (const FSetBase& fsetbase)
{
  #ifndef ASSERT_OFF
    Tracer tr("FSetBase& FSetBase::operator = (const FSetBase& fsetbase)");
  #endif
  if (this != &fsetbase)
    { delete[] start;
      n_fsets  = fsetbase.n_fsets;
      max_n_fsets = fsetbase.max_n_fsets;
      start = new PremiseFSet[max_n_fsets];
      for(int i=0; i<n_fsets; i++)
        start[i] = fsetbase.start[i];
    }
  return *this;
}
```

```
void FSetBase::add_fset(PremiseFSet& fset)
{
  #ifndef ASSERT_OFF
    Tracer tr("void FSetBase::add_fset(PremiseFSet& fset)");
  #endif
  if (n_fsets < max_n_fsets)
   { start[n_fsets] = fset;
     n_fsets++;
   }
  else
   { cerr << "FSetBase::add_fset error: Couldn't append new fuzzy set:
              Maximum number of sets reached" << endl;
     exit(-1);
   }
}

ostream& operator << (ostream& strm, const FSetBase& fsetbase)
{
  #ifndef ASSERT_OFF
    Tracer tr("ostream& operator << (ostream& strm, const FSetBase& fsetbase)");
  #endif
  for(int k=0; k<fsetbase.get_n_fsets(); k++)
    strm << "F" << k << ": " << fsetbase[k] << endl;
  return strm;
}

/*
 * ********** FClause
 */

FClause::FClause()
{
  #ifndef ASSERT_OFF
    Tracer tr("FClause::FClause()");
  #endif
  fvar = -1;
  fset = -1;
}

FClause::FClause(const Matrix& inmat, const FSetBase& fsetbase,
                 int var, int set)
#ifdef ASSERT_OFF
  { fvar = var;
    fset = set;
  }
#else
  { Tracer tr("FClause::FClause(const Matrix& inmat, const FSetBase& fsetbase,
                               int var, int set)");
    assert(var >= 1);
    assert(var <= inmat.Ncols());
    assert(set >= 0);
    assert(set < fsetbase.get_n_fsets());
    fvar = var;
    fset = set;
  }
#endif

ostream& operator << (ostream& strm, const FClause& fclause)
{
  #ifndef ASSERT_OFF
    Tracer tr("ostream& operator << (ostream& strm, const FClause& fclause)");
  #endif
  if (fclause.fvar == -1)
    strm << "empty_fclause ";
```

```
  else
    strm << "X" << fclause.fvar << " is F" << fclause.fset << " ";
  return strm;
}

/*
 * ********** FRuleSugeno
 */

FRuleSugeno::FRuleSugeno()
{
  #ifndef ASSERT_OFF
    Tracer tr("FRuleSugeno::FRuleSugeno()");
  #endif
  SLList<FClause> premise; /// single linked list of FClauses
  ColumnVector consequence;
  ColumnVector grad_cons;
  ColumnVector old_grad_cons;
  ColumnVector delta_cons;
}

FRuleSugeno::FRuleSugeno(int cons_dim)
{
  #ifndef ASSERT_OFF
    Tracer tr("FRuleSugeno::FRuleSugeno(int cons_dim)");
  #endif
  if (cons_dim>0)
   { SLList<FClause> premise;
     ColumnVector cons(cons_dim);
     cons[0]=0.0;
     for(int k=1; k<cons_dim; k++)
       cons[k]=0.0;
     consequence = cons;
     cons[0] = 0.0;
     grad_cons = cons;
     old_grad_cons = cons;
     for(k=0; k<cons_dim; k++)
       cons[k]=delta_0_cons;
     delta_cons = cons;
   }
  else
   { cerr << "FRuleSugeno::FRuleSugeno error: consequence dimension must be
              greater than 0" << endl;
     exit(-1);
   }
}

FRuleSugeno::~FRuleSugeno()
{
  #ifndef ASSERT_OFF
    Tracer tr("FRuleSugeno::~FRuleSugeno()");
  #endif
  premise.clear();
}

FRuleSugeno& FRuleSugeno::operator = (FRuleSugeno& frule)
{
  #ifndef ASSERT_OFF
    Tracer tr("FRuleSugeno& FRuleSugeno::operator = (FRuleSugeno& frule)");
  #endif
  if (this != &frule)
    { if (! premise.empty())
        premise.clear();
      if (! frule.premise.empty())
        { Pix p = frule.premise.first();
```

```
          while (p != NULL)
           { premise.append(frule.premise(p));
             frule.premise.next(p);
           }
        }
      int cons_dim = frule.consequence.Nrows();
      if (consequence.Nrows() != cons_dim)
        { cerr << "FRuleSugeno::operator =  error: different consequence
                   dimensions in rule assignement" << endl;
          cerr << "left side: " << consequence.Nrows() << " right side: "
               << cons_dim << endl;
          exit(-1);
        }
      for(int k=0; k<cons_dim; k++)
        { consequence[k] = frule.consequence[k];
          grad_cons[k] = frule.grad_cons[k];
          old_grad_cons[k] = frule.old_grad_cons[k];
          delta_cons[k] = frule.delta_cons[k];
        }

    } ///// end if (this != &frule)
  return *this;
}

FClause& FRuleSugeno::get_prem_fclause(int i)
{
  #ifndef ASSERT_OFF
    Tracer tr("FClause& FRuleSugeno::get_prem_fclause(int i)");
  #endif
  if (i>get_n_prem_fclauses())
    { cerr << "FRuleSugeno::get_prem_fclause error: access to nonexistant FClause"
           << endl;
      exit(-1);
    }
  Pix plist = premise.first();
  for (int k=0; k<i; k++)
    premise.next(plist);
  if (plist != NULL)
    return premise(plist);
  else
    { cerr << "FRuleSugeno::get_prem_fclause error: access to nonexistant FClause"
           << endl;
      exit(-1);
    }
}

Real FRuleSugeno::premise_value(const FSetBase& fsetbase, const RowVector& invect)
{
  #ifndef ASSERT_OFF
    Tracer tr("Real FRuleSugeno::premise_value(const FSetBase& fsetbase, ...)");
  #endif
  Real premvalue = 1.0;
  Pix plist = premise.first();
  for(int k=0; k<premise.length(); k++)
   { premvalue = t_norm(premvalue,
      fsetbase[premise(plist).get_fset()].get_membership(invect[premise(plist).get_fvar()])
                         );
     premise.next(plist);
   }
  return premvalue;
}

Real FRuleSugeno::consequence_value(const RowVector& invect)
{
  #ifndef ASSERT_OFF
```

```
    Tracer tr("Real FRuleSugeno::consequence_value(const RowVector& invect)");
  #endif
  Real sum = consequence[0];
  for (int k=1; k<get_cons_dim(); k++)
   sum += consequence[k] * invect[k];
  return sum;
}

void FRuleSugeno::print(ostream& strm, const FSetBase& fsetbase)
{
  #ifndef ASSERT_OFF
    Tracer tr("void FRuleSugeno::print(ostream& strm, const FSetBase& fsetbase)");
  #endif
  int k;
  if (get_n_prem_fclauses() == 0)
    strm << "if TRUE ";
  else
    { strm << "if ";
      get_prem_fclause(0).print(strm, fsetbase);
    }
  for(k=1; k<get_n_prem_fclauses(); k++)
    { strm << " and ";
      get_prem_fclause(k).print(strm, fsetbase);
    }
  strm << " then Y = " << consequence[0];
  for(k=1; k<get_cons_dim(); k++)
    strm << " + " << consequence[k] << "*X" << k;
}

ostream& operator << (ostream& strm, FRuleSugeno& frule)
{
  #ifndef ASSERT_OFF
    Tracer tr("ostream& operator << (ostream& strm, FRuleSugeno& frule)");
  #endif
  int k;
  if (frule.get_n_prem_fclauses() == 0)
    strm << "if TRUE ";
  else
    strm << "if " << frule.get_prem_fclause(0);
  for(k=1; k<frule.get_n_prem_fclauses(); k++)
    strm << "and " << frule.get_prem_fclause(k);
  strm << "then Y = " << frule.consequence[0];
  for(k=1; k<frule.get_cons_dim(); k++)
    strm << " + " << frule.consequence[k] << "*X" << k;
  return strm;
}

/*
 * ********** FRuleBaseSugeno
 */

FRuleBaseSugeno::FRuleBaseSugeno(int maxnfrules, int consdim)
{
  #ifndef ASSERT_OFF
    Tracer tr("FRuleBaseSugeno::FRuleBaseSugeno(int maxnfrules, int consdim)");
  #endif
  if (maxnfrules < 1)
    { cerr << "FRuleBaseSugeno::FRuleBaseSugeno error: max_n_frules < 1"
           << endl;
      exit(-1);
    }
  max_n_frules = maxnfrules;
  start = new FRuleSugeno[max_n_frules];
  for(int k=0; k<maxnfrules; k++)
    start[k].ReDimension(consdim);
```

```
  n_frules = 0;
}

FRuleBaseSugeno::~FRuleBaseSugeno()
{
  #ifndef ASSERT_OFF
    Tracer tr("FRuleBaseSugeno::~FRuleBaseSugeno()");
  #endif
  delete[] start;
}

FRuleBaseSugeno& FRuleBaseSugeno::operator = (const FRuleBaseSugeno& frulebase)
{
  #ifndef ASSERT_OFF
    Tracer tr("FRuleBaseSugeno& FRuleBaseSugeno::operator = (const ...)");
  #endif
  if (this != &frulebase)
    { if (max_n_frules != frulebase.max_n_frules)
        { delete[] start;
          max_n_frules = frulebase.max_n_frules;
          start = new FRuleSugeno[max_n_frules];
        }
      n_frules  = frulebase.n_frules;
      for(int i=0; i<n_frules; i++)
        start[i] = frulebase.start[i];
    }
  return *this;
}

void FRuleBaseSugeno::add_frule(FRuleSugeno& frule)
{
  #ifndef ASSERT_OFF
    Tracer tr("void FRuleBaseSugeno::add_frule(FRuleSugeno& frule)");
  #endif
  if (n_frules < max_n_frules)
   { start[n_frules] = frule;
     n_frules++;
   }
  else
   { cerr << "FRuleBaseSugeno::add_frule error: Couldn't append new fuzzy rule:
             Maximum number of rules reached" << endl;
     exit(-1);
   }
}

void FRuleBaseSugeno::fit_cons_param_LMS(const FSetBase& fsetbase,
                                         const Matrix& X, const Matrix& y)
{
  #ifndef ASSERT_OFF
    Tracer tr("void FRuleBaseSugeno::fit_cons_param_LMS(const ...)");
  #endif
  if (n_frules < 1)
    { cerr << "FRuleBaseSugeno::fit_cons_param_LMS error: cannot fit
              parameters of empty rule base";
      cerr << endl;
      exit(-1);
    }
  register int r, i, j;
  register int cons_dim=start[0].get_cons_dim();
  RowVector w(n_frules);       ///// n_frules = R, X.Nrows = M
  Real sum_w_s;
  Real v;
  Matrix A(X.Nrows(), cons_dim * n_frules );
  Matrix V;
  DiagonalMatrix D;
```

```
  for(j=0; j<X.Nrows(); j++)   ///// create Matrix A
   { sum_w_s = 0.0;
     for(r=0; r<n_frules; r++)
       { w[r] = start[r].premise_value(fsetbase, X.Row(j+1));
         sum_w_s += w[r];
       }
     if (sum_w_s == 0.0)
      { cerr << "FRuleBaseSugeno::fit_cons_param_LMS error: Division by zero";
        cerr << endl;
        exit(-1);
      }
     for(r=0; r<n_frules; r++)
       { v = w[r]/sum_w_s;
         for(i=0; i<cons_dim; i++)
           A[j][r*cons_dim+i] = v*X[j][i];
       }
   }
  SVD(A, D, A, V);   ///// claculation of U "in place"
  ColumnVector p = V * (D.i() * (A.t() * y));
  for(r=0; r<n_frules; r++)
    for(i=0; i<cons_dim; i++)
      start[r].set_cons_param(i, p[r*cons_dim+i]);
}

Real FRuleBaseSugeno::inference_all(const FSetBase& fsetbase, const RowVector&
                                    invect, int info_level, int& exception)
{
  #ifndef ASSERT_OFF
    Tracer tr("Real FRuleBaseSugeno::inference_all(const FSetBase& ...)");
  #endif
  register Real sum_w_s = 0.0;
  register Real numerator = 0.0;
  register Real premvalue;
  for(register int k=0; k<get_n_frules(); k++)
   { premvalue = start[k].premise_value(fsetbase, invect);
     sum_w_s += premvalue;
     numerator += inference( premvalue, start[k].consequence_value(invect) );
   }
 if (sum_w_s <= 0.0)
  { if (info_level > 2)
      { cerr << "FRuleBaseSugeno::inference_all warning: Sum_w_s is zero" << endl;
        PRINT(invect);
      }
    exception = 1;
    return 0.0;
  }
 return (numerator / sum_w_s);
}

Real FRuleBaseSugeno::RPROP(FSetBase& fsetbase, Matrix& U, const Matrix& y,
                            FSetSigmoid& dummy, int info_level, int& exception)
{
  #ifndef ASSERT_OFF
    Tracer tr("Real FRuleBaseSugeno::RPROP(...)");
  #endif
  RowVector premvalue(get_n_frules());
  RowVector consvalue(get_n_frules());
  RowVector U_i;
  Real y_hat;
  Real y_i;
  Real sum_w_s;
  Real error = 0.0;
  Real error_i;
  register int j;
  int i, r;
```

```
for(i=1; i<=U.Nrows(); i++) ///// for all examples
{ U_i = U.Row(i);
  y_i = y(i, 1);
  y_hat = 0.0;
  sum_w_s = 0.0;
  error_i = 0.0;
  for(r=0; r<get_n_frules(); r++)
    { premvalue[r] = start[r].premise_value(fsetbase, U_i);
      sum_w_s += premvalue[r];
      consvalue[r] = start[r].consequence_value(U_i);
      y_hat += inference( premvalue[r], consvalue[r] );
    }
  if (sum_w_s == 0.0)
    { if (info_level > 0)
        { cerr << "FRuleBaseSugeno::RPROP warning: Sum_w_s is zero when
                   calculating RPROP for sigmoidal fuzzyset";
          cerr << endl;
        }
      exception = 3;
      return FLT_MAX;
    }
  else
    { y_hat /= sum_w_s;
      error_i = (y_hat - y_i);
      error += error_i * error_i;
      Real const_factor = 2 * error_i / sum_w_s;
      for(r=0; r<get_n_frules(); r++) ///// for all rules
        { ///// calculate consequence parameter gradients
          Real const_factor_cons_r = 2 * error_i * premvalue[r];
          for(j=0; j<start[r].get_cons_dim(); j++)
            start[r].set_grad_cons(j, start[r].get_grad_cons(j) + const_factor_cons_r * U_i[j]);
          ///// calculate premise paramter gradients
          Pix ppremise = start[r].get_p_first_premise();
          Real const_factor_r = const_factor * (consvalue[r] - y_hat) * premvalue[r];
          for(j=0; j<start[r].get_n_prem_fclauses(); j++) ///// for all clauses in rule r
            { ///// this part is critical for the calculation time
              ///// that's why it is optimized with help variables, registers, etc.
              register int fset_index = start[r].get_premise_fclause(ppremise).get_fset();
              register Real u_ji = U_i[ start[r].get_premise_fclause(ppremise).get_fvar() ];
              register Real const_factor_jr = const_factor_r
                                       * (1 - fsetbase[fset_index].get_membership(u_ji));
              register Real temp = fsetbase[fset_index].get_grad_xm();
              temp += fsetbase[fset_index].get_sigma() * const_factor_jr;
              fsetbase[fset_index].set_grad_xm( temp );
              temp = fsetbase[fset_index].get_grad_sigma();
              temp += (fsetbase[fset_index].get_xm() - u_ji) * const_factor_jr;
              fsetbase[fset_index].set_grad_sigma( temp );
              start[r].get_p_next_premise(ppremise);
            }
        }
    }
} ///// end for all examples
Pix ppremise;
PremiseFSet fset_jr;
FClause fclause;
for(r=0; r<get_n_frules(); r++) ///// for all rules
  { ///// fit consequence parameters with RPROP algorithm
    for(j=0; j<start[r].get_cons_dim(); j++)
      { if (start[r].get_old_grad_cons(j) * start[r].get_grad_cons(j) > 0.0)
          { start[r].set_delta_cons(j, start[r].get_delta_cons(j) * eta_plus);
            ///// update parameters
            if (start[r].get_grad_cons(j) > 0.0)
              start[r].set_cons_param(j, start[r].get_cons_param(j)
                                          - start[r].get_delta_cons(j));
             else
```

```
              start[r].set_cons_param(j, start[r].get_cons_param(j)
                                      + start[r].get_delta_cons(j));
          }
        else if (start[r].get_old_grad_cons(j) * start[r].get_grad_cons(j) < 0.0)
          { ///// last step was wrong: go back, make step smaller
            start[r].set_cons_param(j, start[r].get_cons_param(j) - start[r].get_delta_cons(j));
            start[r].set_delta_cons(j, start[r].get_delta_cons(j) * eta_minus);
          }
        start[r].set_old_grad_cons(j, start[r].get_grad_cons(j));
        start[r].set_grad_cons(j, 0.0);
      }
    ///// fit premise parameters with RPROP algorithm
    ppremise = start[r].get_p_first_premise();
    for(j=0; j<start[r].get_n_prem_fclauses(); j++)
      { fclause = start[r].get_premise_fclause(ppremise);
        fset_jr = fsetbase[ fclause.get_fset() ];
        if ((fset_jr.get_old_grad_xm() * fset_jr.get_grad_xm()) > 0.0)
          { fset_jr.set_delta_xm(fset_jr.get_delta_xm() * eta_plus);
            if (fset_jr.get_grad_xm() > 0.0)
              fset_jr.set_xm(fset_jr.get_xm() - fset_jr.get_delta_xm());
            else
              fset_jr.set_xm(fset_jr.get_xm() + fset_jr.get_delta_xm());
          }
        else if ((fset_jr.get_old_grad_xm() * fset_jr.get_grad_xm()) < 0.0)
          { fset_jr.set_xm(fset_jr.get_xm() - fset_jr.get_delta_xm());
            fset_jr.set_delta_xm(fset_jr.get_delta_xm() * eta_minus);
          }
        if ((fset_jr.get_old_grad_sigma() * fset_jr.get_grad_sigma()) > 0.0)
          { fset_jr.set_delta_sigma(fset_jr.get_delta_sigma() * eta_plus);
            if (fset_jr.get_grad_sigma() > 0.0)
              fset_jr.set_sigma(fset_jr.get_sigma() - fset_jr.get_delta_sigma());
            else
              fset_jr.set_sigma(fset_jr.get_sigma() + fset_jr.get_delta_sigma());
          }
        else if ((fset_jr.get_old_grad_sigma() * fset_jr.get_grad_sigma()) < 0.0)
          { fset_jr.set_sigma(fset_jr.get_sigma() - fset_jr.get_delta_sigma());
            fset_jr.set_delta_sigma(fset_jr.get_delta_sigma() * eta_minus);
          }
        fset_jr.set_old_grad_xm( fset_jr.get_grad_xm() );
        fset_jr.set_old_grad_sigma( fset_jr.get_grad_sigma() );
        fset_jr.set_grad_xm( 0.0 );
        fset_jr.set_grad_sigma( 0.0 );
        fsetbase[fclause.get_fset()] = fset_jr;
        start[r].get_p_next_premise(ppremise);
      }
  }
return sqrt(error);
}

void FRuleBaseSugeno::save_estimation(FSetBase& fsetbase, Matrix& X, char* outfilename,
                                      int output_format, int newlines, int info_level)
{
  #ifndef ASSERT_OFF
    Tracer tr("void FRuleBaseSugeno::save_estimation_t(...)");
  #endif
  FILE *outputfilep;
  int k;
  int exception = 0;
  Real y_hat;
  outputfilep = fopen(outfilename, "w");
  if (outputfilep == NULL)
    { cerr << "FRuleBaseSugeno::save_estimation error: could not open outputfile "
           << outfilename;
      exit(-1);
    }
```

```
  for(k = 0; k < X.Nrows(); k++)
    { y_hat = inference_all(fsetbase, X.Row(k+1), 0, exception);
      if (exception)
        { if (info_level > 0)
            cerr << "FRuleBaseSugeno::save_estimation: model output not defined for row "
                 << k+1 << ": " << X.Row(k+1) << endl;
          y_hat = 0.0;
          exception = 0;
        }
      if (output_format == GNUPLOT1)
        fprintf(outputfilep, "%f\n", y_hat);
      else if (output_format == GNUPLOT2)
        { fprintf(outputfilep, "%f %f %f\n", X[k][1], X[k][2], y_hat);
          if (newlines > 0)
            if ((k+1) % newlines == 0)
              fprintf(outputfilep, "\n");
        }
      else if (output_format == MATLAB1)
        fprintf(outputfilep, "%f\n", y_hat);
      else if (output_format == MATLAB2)
        { fprintf(outputfilep, "%f ", y_hat);
          if (newlines > 0)
            if ((k+1) % newlines == 0)
              fprintf(outputfilep, "\n");
        }
    }
  fclose(outputfilep);
}

///// load and parse .fzy file
void FRuleBaseSugeno::load(FSetBase& fsetbase, const char* filename, Matrix& X)
{
  #ifndef ASSERT_OFF
    Tracer tr("void load(FSetBase& fsetbase, const char* filename, Matrix& X)");
  #endif
  FILE *inputfilep;
  char line[1000];
  Boolean parse_error = FALSE;
  if ( (inputfilep=fopen(filename, "r"))==NULL )
   { cerr << "FRuleBaseSugeno::load error: cannot open input file ";
     cerr << filename << endl;
     exit(-1);
   }
  ///// first 6 strings: dimensions
  fscanf(inputfilep, "%s", &line); fscanf(inputfilep, "%s", &line);
  fscanf(inputfilep, "%s", &line); fscanf(inputfilep, "%s", &line);
  fscanf(inputfilep, "%s", &line); fscanf(inputfilep, "%s", &line);
  while (fscanf(inputfilep, "%s", &line)==1)
   { if (! strncmp(line, "R", 1)) ///// load fuzzy rule
       { FRule frule(get_rule_cons_dim());
         fscanf(inputfilep, "%s", &line);
         if (! strncmp(line, "TRUE", 4))
           { fscanf(inputfilep, "%s", &line);
           }
         else while (! strncmp(line, "X", 1)) ///// load fuzzy fclause
           { int fvar = -1;
             fvar = atoi(line+1);
             fscanf(inputfilep, "%s", &line);
             if (strncmp(line, "is", 2))
               parse_error = TRUE;
             fscanf(inputfilep, "%s", &line);
             int fset = -1;
             fset = atoi(line+1);
             FClause fclause(X, fsetbase, fvar, fset);
             frule.add_prem_fclause(fclause);
```

```
            fscanf(inputfilep, "%s", &line);
          }
        frule.set_cons_param(0, atof(line));
        fscanf(inputfilep, "%s", &line);
        frule.set_grad_cons(0, atof(line));
        fscanf(inputfilep, "%s", &line);
        frule.set_old_grad_cons(0, atof(line));
        fscanf(inputfilep, "%s", &line);
        frule.set_delta_cons(0, atof(line));
        ///// load consequence
        for(int i = 1; i < start[0].get_cons_dim(); i++)
          { fscanf(inputfilep, "%s", &line);
            frule.set_cons_param(i, atof(line));
            fscanf(inputfilep, "%s", &line);
            frule.set_grad_cons(i, atof(line));
            fscanf(inputfilep, "%s", &line);
            frule.set_old_grad_cons(i, atof(line));
            fscanf(inputfilep, "%s", &line);
            frule.set_delta_cons(i, atof(line));
          }
        add_frule(frule);
      }
    else if (! strncmp(line, "F", 1)) ///// load fuzzy set
      { PremiseFSet fset;
        fscanf(inputfilep, "%s", &line);
        fset.set_xm(atof(line));
        fscanf(inputfilep, "%s", &line);
        fset.set_sigma(atof(line));
        fscanf(inputfilep, "%s", &line);
        fset.set_grad_xm(atof(line));
        fscanf(inputfilep, "%s", &line);
        fset.set_grad_sigma(atof(line));
        fscanf(inputfilep, "%s", &line);
        fset.set_old_grad_xm(atof(line));
        fscanf(inputfilep, "%s", &line);
        fset.set_old_grad_sigma(atof(line));
        fscanf(inputfilep, "%s", &line);
        fset.set_delta_xm(atof(line));
        fscanf(inputfilep, "%s", &line);
        fset.set_delta_sigma(atof(line));
        fsetbase.add_fset(fset);
      }
    else
      parse_error = TRUE;
  }
 fclose(inputfilep);
 if (parse_error)
   cerr << "FRuleBaseSugeno::load warning: parse error occured while parsing file "
        << filename << endl;
}

ostream& operator << (ostream& strm, const FRuleBaseSugeno& frulebase)
{
  #ifndef ASSERT_OFF
    Tracer tr("ostream& operator << (ostream& strm, const FRuleBaseSugeno& frulebase)");
  #endif
  for(int k=0; k<frulebase.get_n_frules(); k++)
    strm << "R" << k+1 << ": " << frulebase[k] << endl;
  return strm;
}
```

B.4 fzyident.cc

```
/*
 * fzyident.cc
 *  - fuzzy modelisation/identification
 *  - saveing/loading of fuzzy models in/from .fzy files
 *  - prediction & simulation
 *
 * Diplomarbeit, Manfred Maennle, April 95
 * last change: July 18th 95
 *
 */

extern "C"
{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
}

#include <iostream.h>
#include <newmat.h>
#include <newmatap.h>
#include <newmatio.h>
#include "fuzzsets.h"

const int N_FRULE_LIMIT = 100;

enum mode { UNDEFINED, ESTIMATION, IDENTIFICATION, MODELISATION, SIMULATION };
enum crit { UNDEF, UC, R2 }; /// valid criterions

void copyright( void );

void modelisation_r2(char* file1, char* file2, int n_examples, int x_dim, int cons_dim,
Real impr, int max_n_fr, int min_opt_it, int max_opt_it, int opt_best, int opt_last,
int output_format, int info_level, char* n_ext, int newlines);

void modelisation_uc(char* file1, char* file2, int n_examples, int x_dim, int cons_dim,
Real impr, int max_n_fr, int min_opt_it, int max_opt_it, int opt_best, int opt_last,
int output_format, int info_level, char* n_ext, int newlines);

void identification_r2(char* file1, char* file2, char* file3, int n_examples, int x_dim,
int min_opt_it, int max_opt_it, int output_format, int info_level, char* n_ext,
int newlines);

void identification_uc(char* file1, char* file2, char* file3, int n_examples, int x_dim,
int min_opt_it,
int max_opt_it, int output_format, int info_level, char* n_ext, int newlines);

void estimation(char* file1, char* file2, int n_examples, int x_dim, int output_format,
int info_level, char* name_extension, int newlines);

void simulation(int order, char* file1, char* file2, int n_examples,
int x_dim, int output_format, int info_level, char* name_extension, int newlines);

void itoa(int n, char s[]);

void load_datafile(Matrix& X, Matrix& y, char* in_filename);

void create_new_candidate(FRuleBase& frulebase, FSetBase& fsetbase, Matrix& XA,
int cons_dim, int rule, int var);

void optimize_parameters_r2(FRuleBase& frulebase, FSetBase& fsetbase, Matrix& XA,
```

```
Matrix& ya,  Matrix& XB, Matrix& yb, PremiseFSet& fset_type, int min_opt_iterations,
int max_opt_iterations, Real& cand_crit, int info_level, int& exception);

inline Real calculate_r2(FRuleBase& frulebase, FSetBase& fsetbase, Matrix& XA,
Matrix& ya, Matrix& XB, Matrix& yb, int info_level, int& exception);


int main( int argc, char* argv[] )
{
{
Tracer tr("main: init");

///// define all options (with default values)
///// for more information see the help message
///// or the report 'Diplomarbeit', chapter A.2.2
int   actual_mode = UNDEFINED;
char* file1 = NULL;
char* file2 = NULL;
char* file3 = NULL;
int   n_examples = 0;
int   x_dim = 0;
int   cons_dim = 0;
Real  improvement_rate = 5.0; /// in %
int   max_number_frules = 10;
int   min_opt_iterations = 5;
int   max_opt_iterations = 200;
int   opt_best = 0;
int   opt_last = 0;
int   criterion = R2;
int   output_format = MATLAB1;
int   info_level = 2;
char* name_extension = NULL;
int   newlines = 0;
int   order = 0;

///// check main(...) parameters
tr.ReName("main: check main(...) parameters");
Boolean parse_error = FALSE;
int i = 1; /// begin command line parsing
while ( argv[i] )
  { if (! strcmp(argv[i], "-estim"))       { actual_mode = ESTIMATION; }
    else if (! strcmp(argv[i], "-ident"))   { actual_mode = IDENTIFICATION; }
    else if (! strcmp(argv[i], "-sim"))     { actual_mode = SIMULATION; }
    else if (! strcmp(argv[i], "-model"))   { actual_mode = MODELISATION; }
    else if (! strcmp(argv[i], "-f1"))      { file1 = argv[++i]; }
    else if (! strcmp(argv[i], "-f2"))      { file2 = argv[++i]; }
    else if (! strcmp(argv[i], "-f3"))      { file3 = argv[++i]; }
    else if (! strcmp(argv[i], "-ex"))      { n_examples = atoi(argv[++i]); }
    else if (! strcmp(argv[i], "-xdim"))    { x_dim = atoi(argv[++i]);
                                              cons_dim = x_dim+1; }
    else if (! strcmp(argv[i], "-cdim"))    { cons_dim = atoi(argv[++i]); }
    else if (! strcmp(argv[i], "-imp"))     { improvement_rate = (Real) atof(argv[++i]); }
    else if (! strcmp(argv[i], "-frules"))  { max_number_frules = atoi(argv[++i]); }
    else if (! strcmp(argv[i], "-minopt"))  { min_opt_iterations = atoi(argv[++i]); }
    else if (! strcmp(argv[i], "-maxopt"))  { max_opt_iterations = atoi(argv[++i]); }
    else if (! strcmp(argv[i], "-optbest")) { opt_best = atoi(argv[++i]); }
    else if (! strcmp(argv[i], "-uc"))      { criterion = UC; }
    else if (! strcmp(argv[i], "-r2"))      { criterion = R2; }
    else if (! strcmp(argv[i], "-gnuplot1")){ output_format = GNUPLOT1; }
    else if (! strcmp(argv[i], "-gnuplot2")){ output_format = GNUPLOT2; }
    else if (! strcmp(argv[i], "-matlab1")) { output_format = MATLAB1; }
    else if (! strcmp(argv[i], "-matlab2")) { output_format = MATLAB2; }
    else if (! strcmp(argv[i], "-info"))    { info_level = atoi(argv[++i]); }
    else if (! strcmp(argv[i], "-n_ext"))   { name_extension = argv[++i]; }
    else if (! strcmp(argv[i], "-nlines"))  { newlines = atoi(argv[++i]); }
```

```
    else if (! strcmp(argv[i], "-ord"))     { order = atoi(argv[++i]); }
    else { cerr << endl << "invalid parameter: " << argv[i] << endl << endl;
           parse_error = TRUE;
         }
    i++;
  }
if (    (actual_mode == UNDEFINED)
     || (file1 == NULL)
     || (file2 == NULL)
     || (n_examples < 1)
     || (x_dim < 1)
     || (cons_dim<1 || cons_dim>x_dim+1)
     || (improvement_rate>100.0)
     || (max_number_frules<1 || max_number_frules>N_FRULE_LIMIT)
     || (min_opt_iterations < 1)
     || (max_opt_iterations < 1)
     || (opt_best < 0)
     || (opt_last < 0)
     || (info_level<0 || info_level>3)
     || (newlines < 0)
   )
  parse_error = TRUE;
if ( (actual_mode == IDENTIFICATION) && (file3 == NULL) )
  parse_error = TRUE;
if ( (actual_mode == SIMULATION) && (order == 0) )
  { cerr << "error: order = 0 at simulation mode" << endl;
    parse_error = TRUE;
  }

if (argc > 1)
  { cout << endl << argv[0] << " - fuzzy identification of a Sugeno type fuzzy model"
         << endl;
    cout << "Copyright (C) by Manfred Maennle, 1995" << endl;
    cout << "This software comes WITHOUT ANY WARRANTY." << endl << endl;
  }

if ((info_level > 1) && (argc > 1))
  {
    cout << "chosen parameters:" << endl;
    if (actual_mode == ESTIMATION)
      cout << "-estim" << endl;
    else if (actual_mode == SIMULATION)
      cout << "-sim" << endl;
    else if (actual_mode == IDENTIFICATION)
      cout << "-ident" << endl;
    else if (actual_mode == MODELISATION)
      cout << "-model" << endl;
    if (file1 != NULL)
      cout << "<file_a> " << file1 << endl;
    else
      cout << "<file_a> " << endl;
    if (file2 != NULL)
      cout << "<file_b> " << file2 << endl;
    else
      cout << "<file_b> " << endl;
    if (file3 != NULL)
      cout << "<file_c> " << file3 << endl;
    else
      cout << "<file_c> " << endl;
    cout << "-ex      " << n_examples << endl;
    cout << "-xdim    " << x_dim << endl;
    cout << "-cdim    " << cons_dim << endl;
    cout << "-imp     " << improvement_rate << endl;
    cout << "-frules  " << max_number_frules << endl;
    cout << "-minopt  " << min_opt_iterations << endl;
```

```
    cout << "-maxopt  " << max_opt_iterations << endl;
    cout << "-optbest " << opt_best << endl;
    cout << "-optlast " << opt_last << endl;
    if (actual_mode == SIMULATION)
      cout << "-order   " << order << endl;
    if (criterion == R2)
      cout << "-r2" << endl;
    else if (criterion == UC)
      cout << "-uc" << endl;
    if (output_format == GNUPLOT1)
      cout << "-gnuplot1" << endl;
    else if (output_format == GNUPLOT2)
      cout << "-gnuplot2" << endl;
    else if (output_format == MATLAB1)
      cout << "-matlab1" << endl;
    else if (output_format == MATLAB2)
      cout << "-matlab2" << endl;
    cout << "-info    " << info_level << endl;
    if (name_extension != NULL)
      cout << "-n_ext   " << name_extension << endl;
    else
      cout << "-n_ext   " << endl;
    cout << "-nlines  " << newlines << endl;
  }
if (parse_error)
  { /// help message

      ...

  }
///// end check main() parameters

if (actual_mode == MODELISATION)
  if (criterion == UC)
    modelisation_uc(file1, file2, n_examples, x_dim, cons_dim, improvement_rate,
                    max_number_frules, min_opt_iterations, max_opt_iterations,
                    opt_best, opt_last, output_format, info_level, name_extension,
                    newlines);
  else
    modelisation_r2(file1, file2, n_examples, x_dim, cons_dim, improvement_rate,
                    max_number_frules, min_opt_iterations, max_opt_iterations,
                    opt_best, opt_last, output_format, info_level, name_extension,
                    newlines);

if (actual_mode == IDENTIFICATION)
  if (criterion == UC)
    { cerr << "error: identification with UC not yet available." << endl;
      exit(-1);
    }
  else
    identification_r2(file1, file2, file3, n_examples, x_dim, min_opt_iterations,
                      max_opt_iterations, output_format, info_level, name_extension,
                      newlines);

if (actual_mode == ESTIMATION)
  estimation(file1, file2, n_examples, x_dim, output_format, info_level,
             name_extension, newlines);

if (actual_mode == SIMULATION)
  simulation(order, file1, file2, n_examples, x_dim, output_format, info_level,
              name_extension, newlines);

tr.ReName("main: end");
}
#ifndef ASSERT_OFF
```

```
  FreeCheck::Status(); /// check deletion of all objects (newmat member function)
#endif
return 0;
} ///// end of main()

////////////////////////////////////////////////////////////////////////////////

void modelisation_r2(char* file1, char* file2, int n_examples, int x_dim, int cons_dim,
                     Real improvement_rate, int max_number_frules,
                     int min_opt_iterations, int max_opt_iterations, int opt_best,
                     int opt_last, int output_format, int info_level,
                     char* name_extension, int newlines)
{
Tracer tr("void modelisation_r2(char* file1, char* file2, int n_examples, ...)");
const int MAXNFSETS = 2 * max_number_frules;
int exception = 0;
PremiseFSet fset_type;
char out_filename[100];
char fzy_filename[100];
char r2_filename[100];
char save_number[10];

Matrix XA(n_examples, x_dim + 1); /// system input matrix, first column is 1.0
Matrix XB(n_examples, x_dim + 1);
Matrix ya(n_examples, 1);         /// system output vector
Matrix yb(n_examples, 1);
load_datafile(XA, ya, file1);
load_datafile(XB, yb, file2);

FSetBase  old_fsetbase(MAXNFSETS);       /// global best model
FSetBase  best_fsetbase(MAXNFSETS);      /// best model within actual epoch
FSetBase  candidate_fsetbase(MAXNFSETS); /// candidate model
FRuleBase old_frulebase(max_number_frules, cons_dim);
FRuleBase best_frulebase(max_number_frules, cons_dim);
FRuleBase candidate_frulebase(max_number_frules, cons_dim);
Real      old_criterion;
Real      best_criterion;
Real      candidate_criterion;
int   best_rule_index;
int    best_variable_index;

///// initialize best_frulebase
{ FRule frule(cons_dim); best_frulebase.add_frule(frule); }

strcpy(r2_filename, "s_"); /// create filename
if (name_extension != NULL)
  strcat(r2_filename, name_extension);
strcat(r2_filename, ".r2");
///// estimation of initial model ( == 0.0 )
best_frulebase.save_R2(best_fsetbase, XB, yb, r2_filename, info_level);

best_frulebase.fit_cons_param_LMS(best_fsetbase, XA, ya);
best_criterion = calculate_r2(best_frulebase, best_fsetbase, XA, ya, XB, yb,
                              info_level, exception);

if (info_level > 1)
  { cout << endl << "initial candidate:" << endl;
    best_frulebase.print(cout, best_fsetbase);
    PRINT(best_criterion); cout << endl;
  }
strcpy(out_filename, "s_");
if (name_extension != NULL)
  strcat(out_filename, name_extension);
strcpy(fzy_filename, out_filename);
strcat(fzy_filename, "1.fzy");
```

```
strcat(out_filename, "1.out");
best_frulebase.save_estimation(best_fsetbase, XA, out_filename, output_format,
                               newlines, info_level);
best_frulebase.save_R2(best_fsetbase, XB, yb, r2_filename, info_level);
best_frulebase.dump(best_fsetbase, fzy_filename);

///// Sugeno's heuristical model search
do
{ tr.ReName("main: loop init");
  old_frulebase = best_frulebase;
  old_fsetbase = best_fsetbase;
  old_criterion = best_criterion;
  best_criterion = FLT_MAX;
  ///// forall rules
  for(int rule_index = 0; rule_index < old_frulebase.get_n_frules(); rule_index++)
    ///// forall variables
    for(int variable_index = 1; variable_index <= x_dim; variable_index++)
      { ///// create new candidate
        tr.ReName("modelisation_r2: create new candidate");
        candidate_frulebase = old_frulebase;
        candidate_fsetbase = old_fsetbase;
        if (info_level > 0)
          cout << "new candidate, splitting X" << variable_index << " in R"
               << rule_index+1 << endl;
        create_new_candidate(candidate_frulebase, candidate_fsetbase, XA, cons_dim,
                             rule_index, variable_index);
        if (info_level > 2)
          candidate_frulebase.print(cout, candidate_fsetbase);
        ///// optimize new candidate
        tr.ReName("modelisation_r2: optimize new candidate");
        optimize_parameters_r2(candidate_frulebase, candidate_fsetbase, XA, ya, XB,
                               yb, fset_type, min_opt_iterations, max_opt_iterations,
                               candidate_criterion, info_level, exception);
        if (exception)
          { tr.ReName("modelisation_r2: exception has occured");
            candidate_frulebase.RPROP_backstep(candidate_fsetbase, fset_type,
                                               exception);
            if (info_level > 1)
              { cout << "an exception has occured when optimizing the parameters, ";
                cout << "optimization interrupted" << endl;
                exception = 0;
                candidate_criterion = calculate_r2(best_frulebase, best_fsetbase, XA,
                                                   ya, XB, yb, info_level, exception);
              }
            exception = 0;
          }
        else if (candidate_criterion < best_criterion) /// candidate is better
          { tr.ReName("modelisation_r2: copy best candidate");
            best_frulebase = candidate_frulebase;
            best_fsetbase = candidate_fsetbase;
            best_criterion = candidate_criterion;
            best_rule_index = rule_index;
            best_variable_index = variable_index;
          }
        if (info_level > 2)
          { cout << "optimized candidate: " << endl;
            candidate_frulebase.print(cout, candidate_fsetbase);
          }
        if (info_level > 1)
          PRINT(candidate_criterion);
      }//end forall variables and forall rules
  ///// additional training of best candidate within fixed number of rules
  if (opt_best > 0)
    { tr.ReName("modelisation_r2: optimize best within fixed number of rules
                 with LMS");
```

```
      best_frulebase.fit_cons_param_LMS(best_fsetbase, XA, ya);
      tr.ReName("modelisation_r2: optimize best within fixed number of rules
                 with RPROP");
      Real error;
      int k=0;
      while ((k<opt_best) && (! exception))
        { k++;
          error = best_frulebase.RPROP(best_fsetbase, XA, ya, fset_type,
                                       info_level, exception);
        }
      if (exception)
        { best_frulebase.RPROP_backstep(best_fsetbase, fset_type, exception);
          exception = 0;
        }
      best_criterion = calculate_r2(best_frulebase, best_fsetbase, XA, ya, XB, yb,
                                    info_level, exception);
    }
  ///// print best candidate within fixed number of rules and save its estimation
  if (info_level > 1)
    { tr.ReName("modelisation_r2: print best candidate within fixed number of rules");
      cout << "*** best candidate with " << best_frulebase.get_n_frules()
           << " rules:" << endl;
      cout << "split X" << best_variable_index << " in rule R"
           << best_rule_index+1 << endl;
      best_frulebase.print(cout, best_fsetbase);
      PRINT(best_criterion);
      cout << "improvement: " << (old_criterion-best_criterion)/old_criterion*100.0
           << " %" << endl << endl;
    }
  strcpy(out_filename, "s_");
  if (name_extension != NULL)
    strcat(out_filename, name_extension);
  itoa(best_frulebase.get_n_frules(), save_number);
  strcat(out_filename, save_number);
  strcpy(fzy_filename, out_filename);
  strcat(fzy_filename, ".fzy");
  strcat(out_filename, ".out");
  tr.ReName("modelisation_r2: save estimation of best candidate within
             fixed number of rules");
  best_frulebase.save_estimation(best_fsetbase, XA, out_filename, output_format,
                                 newlines, info_level);
  best_frulebase.save_R2(best_fsetbase, XB, yb, r2_filename, info_level);
  best_frulebase.dump(best_fsetbase, fzy_filename);

} while ( (best_criterion < (100-improvement_rate)/100.0*old_criterion)
          && (best_frulebase.get_n_frules() < max_number_frules) );
///// end of Sugeno's heuristic search

if (best_criterion < (100-improvement_rate)/100.0*old_criterion)
  { old_frulebase = best_frulebase;
    old_fsetbase = best_fsetbase;
    old_criterion = best_criterion;
  }
///// optimize best model afterwards
if (opt_last > 0)
  { tr.ReName("modelisation_r2: optimize afterwards with LMS");
    old_frulebase.fit_cons_param_LMS(old_fsetbase, XA, ya);
    tr.ReName("modelisation_r2: optimize afterwards with RPROP");
    Real error;
    int k=0;
    while ((k<opt_last) && (! exception))
      { k++;
        error = old_frulebase.RPROP(old_fsetbase, XA, ya, fset_type,
                                    info_level, exception);
      }
```

```
    if (exception)
      { old_frulebase.RPROP_backstep(old_fsetbase, fset_type, exception);
        exception = 0;
      }
  }
///// print optimal model and save its estimation
if (info_level > 0)
  { tr.ReName("modelisation_r2: save model");
    cout << endl << "best model:" << endl;
    old_frulebase.print(cout, old_fsetbase);
  }
tr.ReName("modelisation_r2: save estimation");
strcpy(out_filename, "s_");
if (name_extension != NULL)
  { strcat(out_filename, name_extension);
    strcat(out_filename, "_");
  }
strcpy(fzy_filename, out_filename);
strcat(fzy_filename, "opt.fzy");
old_frulebase.dump(old_fsetbase, fzy_filename);
strcat(out_filename, "a.out");
old_frulebase.save_estimation(old_fsetbase, XA, out_filename, output_format,
                              newlines, info_level);
old_frulebase.save_R2(old_fsetbase, XB, yb, r2_filename, info_level);
}

////////////////////////////////////////////////////////////////////////////////

Real calculate_error(FRuleBase& frulebase, FSetBase& fsetbase, Matrix& X, Matrix& y,
                     int info_level, int& exception)
{
  Real error = 0.0;
  Real y_hat;
  for(int k=1; k<=X.Nrows(); k++)
    { y_hat = frulebase.inference_all(fsetbase, X.Row(k), info_level, exception);
      error += (y(k, 1) - y_hat) * (y(k, 1) - y_hat);
    }
  return sqrt(error);
}

////////////////////////////////////////////////////////////////////////////////

void itoa(int n, char s[]) ///// transform an integer to a string
{
  #ifndef ASSERT_OFF
    Tracer tr("void itoa(int n, char s[])");
  #endif
  int i, k, sign;
  char c;
  if ((sign =  n) < 0)
    n = -n;
  i=0;
  do {
    s[i++] = n % 10 + '0';
  } while ((n /= 10) > 0);
  if (sign < 0)
     s[i++] = '-';
  s[i] = '\0';
  for (k=0; k<i/2; k++)
    { c = s[k];
      s[k] = s[i-k-1];
      s[i-k-1] = c;
    }
}
```

Literaturverzeichnis

[AL95] Abe, S. and M. Lan: *Fuzzy rules extraction directly from numerical data for function approximation.* IEEE Transactions on Systems, Man, and Cybernetics, 25(1):119–129, 1995.

[Arr93] Arroyo, F.: *Modélisation de processus industriels par traitement statistique de données. Application à la réalisation d'un outil d'aide à la conduite d'une machine à papier.* Thèse, Université de Nancy I, Avril 1993.

[BG93] Bandemer, H. und S. Gottwald: *Einführung in Fuzzy-Methoden.* Akademie Verlag, 4. bearbeitete und erweiterte Auflage, 1993.

[BJ76] Box, G. and G. Jenkins: *Time Series Analysis, Forecasting and Control.* Holden-Day, 6th edition, 1976.

[Bö94] Böhler, A.: *Lösung von Fuzzy-Relationen-Gleichungen.* Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, Institut für Prozeßrechentechnik und Robotik, Oktober 1994.

[BR92] Braun, H. and M. Riedmiller: *Rprop: A fast adaptive learning algorithm.* In *Proceedings of the Int. Symposium on Computer and Information Science VII*, 1992.

[BR93] Braun, H. and M. Riedmiller: *Rprop: A fast and robust backpropagation learning strategy.* In *Proceedings of the ACNN*, 1993.

[Bre94] Breymann, U.: *C++ Eine Einführung.* Carl Hanser Verlag, 1. bearbeitete Auflage, 1994.

[Che95] Chen, S.: *Modelling of paper machines for control: Theory and practice.* Pulp & Paper Canada, 96(1):44–48, 1995.

[CLC94] Chen, J., J. Lu, and L. Chen: *An on-line identification algorithm for fuzzy systems.* Fuzzy Sets and Systems, 64(1):63–72, 1994.

[CLR92] Cormen, B., C. Leiserson, and R. Rivest: *Introduction to Algorithms.* MIT Press, 1992.

[Dav95] Davies, R.: *Documentation for Newmat08, a Matrix Library in C++*, 1995. ftp://tahi.isor.vuw.ac.nz/pub/newmat08.

[Fis91] Fishwick, P.: *Fuzzy simulation: Specifying and identifying qualitative models.* International Journal of General Systems, 19:295–316, 1991.

[Föl92] Föllinger, O.: *Regelungstechnik.* Hüthig Verlag, 7. Auflage, 1992.

[Fra95] Frank, I.: *Modern nonlinear regression methods.* Chemometrics and Intelligent Laboratory Systems, 27:1–19, 1995.

[Goo93] Goos, G.: *Skript zur Vorlesung „Unscharfe Mengen“.* Universität Karlsruhe, WS 1992/93.

[GRW92] Guariso, G., A. Rizzoli, and H. Werthner: *Identification of model structure via qualitative simulation.* IEEE Transactions on Systems, Man, and Cybernetics, 22(5):1075–1086, 1992.

[Hih93] Hihi, J.: *Evaluation de méthodes d'identification de systèmes non linéaires en régime permanent : Méthode de Traitement des Données par Groupes ; Identification floue.* Thèse, Université de Nancy I, Mars 1993.

[HM94] Hagan, M. and M. Menhaj: *Training feedforward networks with the marquardt algorithm.* IEEE Transactions on Neural Networks, 5(6):989–993, 1994.

[INT93] Ishibuchi, H., K. Nozaki, and H. Tanaka: *Efficient fuzzy partition of pattern space for classification problems.* Fuzzy Sets and Systems, 59(3):295–304, 1993.

[JS95] Jang, J. and C. Sun: *Neuro-fuzzy modeling and control.* In *Proceedings of the IEEE*, volume 83(3), pages 378–405, 1995.

[KGK93] Kruse, J., J Gebhardt und F. Klawonn: *Fuzzy-Systeme.* Teubner, Stuttgart, 1993.

[KL80] Klema, V. and A. Laub: *The singular value decomposition: Its computation and some applications.* IEEE Transactions on Automatic Control, AC-25(2):164–176, 1980.

[KR90] Kernigham, B. und D. Ritchie: *Programmieren in C.* Carl Hanser Verlag, 2. Auflage, 1990.

[LC94] LAI, Y. and S. CHANG: *A fuzzy approach for multiresponse optimization: An off-line quality engineering problem.* Fuzzy Sets and Systems, 63(2):117–129, 1994.

[Lea94] LEA, D.: *User's Guide to the GNU C++ Library.* 675 Mass Ave, Cambridge, MA 02139, USA, 1994. libg++ version 2.5.3.

[Mä95] MÄNNLE, M.: *Une Modélisation à Base de Règles Floues.* Rapport intern, CRAN, Université de Nancy I, mai 1995.

[Men95] MENDEL, J.: *Fuzzy logic systems for engineering: A tutorial.* In *Proceedings of the IEEE*, volume 83(3), pages 345–377, 1995.

[Men94] MENZEL, W.: *Skript zur Vorlesung „Theorie der Neuronalen Netze“.* Universität Karlsruhe, WS 1993/94.

[NC92] NEGOITA, G. and V. CANCIU: *Fuzzy identification of systems: A new cybernetic method.* Kybernetes, 21(4):67–79, 1992.

[NP90] NARENDA, K. and K. PARTHASARATHY: *Identification and control of dynamical systems using neural networks.* IEEE Transactions on Neural Networks, 1(1):4–27, 1990.

[NR94] NAKAMORI, Y. and M. RYOKE: *Identification of fuzzy prediction models through hyperellipsoidal clustering.* IEEE Transactions on Systems, Man, and Cybernetics, 24(8):1153–1173, 1994.

[RW94] REDDEN, D. and W. WOODALL: *Properties of certain fuzzy linear regression methods.* Fuzzy Sets and Systems, 64:361–375, 1994.

[SK88] SUGENO, M. and G. KANG: *Structure identification of fuzzy model.* Fuzzy Sets and Systems, 26(1):15–33, 1988.

[SL93] SHEN, Q. and R. LEITCH: *Fuzzy qualitative simulation.* IEEE Transactions on Systems, Man, and Cybernetics, 23(4):1038–1061, 1993.

[Sta94] STALLMANN, R.: *Using and Porting GNU CC.* 675 Mass Ave, Cambridge, MA 02139, USA, 1994. GCC version 2.6.3.

[Str92] STROUSTRUP, B.: *Die C++ Programmiersprache.* Addison-Wesley, 2. Auflage, 1992.

[SY93] SUGENO, M. and T. YASUKAWA: *A fuzzy-logic-based approach to qualitative modeling.* IEEE Transactions on Fuzzy Systems, 1(1):7–31, 1993.

[TS85] TAKAGI, T. and M. SUGENO: *Fuzzy identification of systems and its application to modeling and control.* IEEE Transactions on Systems, Man, and Cybernetics, 15(1):116–132, 1985.

[TYT94] TANAKA, M., J. YE, and T. TANINO: *Identification of nonlinear systems using fuzzy logic and genetic algorithms.* SYSID, 1:301–306, 1994.

[XM92] XIZHAO, W. and H. MINGHU: *Fuzzy linear regression analysis.* Fuzzy Sets and Systems, 51:179–188, 1992.

[YF93] YAGER, R. and D. FILEV: *Unified structure and parameter identification of fuzzy models.* IEEE Transactions on Systems, Man, and Cybernetics, 23(4):1198–1205, 1993.

[Zad65] ZADEH, L.: *Fuzzy sets.* Information and Control, 8:338–353, 1965.

[Zad94] ZADEH, L.: *The role of fuzzy logic in modeling, identification and control.* Modeling, Identification, and Control, 15(3):191–203, 1994.

[ZMS+94] ZELL, A., N. MACHE, T. SOMMER, and OTHERS: *Stuttgart Neural Network Simulator, Users Manual, Version 3.2.* University of Stuttgart, june 1994.