Ein Fuzzy-Auswertungs- und Krigingsystem für raumbezogene Daten

Diplomarbeit

vorgelegt von
Frank Bartels

Betreuer:
Dr. A. Salski
Prof. Dr. P. Kandzia

Februar 1997
Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität zu Kiel

Ein Fuzzy-Auswertungs- und Krigingsystem
für raumbezogene Daten

Inhaltsverzeichnis

Definitionen und Sätze

[1] Diese Definitionen sind nicht explizit mit '**Definition:**' gekennzeichnet.

Abbildungen

1 Einleitung, Vorauswahl der Methoden

1.1 Ziel der Diplomarbeit, Darstellung des Problems

In der Geostatistik gibt es verschiedene Interpolationsverfahren für raumbezogene Daten. Auf der Grundlage von bekannten Werten an einigen Stellen liefern diese Verfahren einen geschätzten Wert für eine beliebige Stelle (diese sollte sich allerdings innerhalb des Bereichs der Stellen mit bekannten Werten befinden). Der Begriff *Kriging* bezeichnet dabei die Gruppe von Interpolationsverfahren, die auf einer statistischen Analyse mit Hilfe sogenannter Variogramme beruhen. Diese Verfahren, insbesondere Normales Kriging (Ordinary Kriging), werden häufig eingesetzt.

Ein wesentliches Ziel der Diplomarbeit ist die Umsetzung einer Erweiterung des Normalen Kriging auf unscharfe Zahlen[1], genannt Fuzzy Kriging. Die verschiedenen Möglichkeiten dazu werden in Kapitel 1.4 genauer ausgeführt (Fuzzy Kriging Typ 1 bis 3).

Eine mögliche Motivation für den Einsatz von unscharfen Zahlen als Eingabe für Kriging-Verfahren ist, unpräzise Informationen, wie beispielsweise Schätzungen eines Experten, miteinbeziehen zu können. Dies kann in Gebieten von Interesse sein, in denen zu wenig Stützwerte exakt bekannt sind, um ausreichend zuverlässige Interpolationen durchzuführen.

Eine andere Motivation ist, daß man abschätzbare Ungenauigkeiten der Eingabewerte, die ohnehin bekannt sind, nutzen kann, um für das Kriging-Ergebnis, in dem sich diese Ungenauigkeiten bei Fuzzy Kriging niederschlagen, eine realistischere Beurteilung zu ermöglichen.

Eine zusätzliche Ergänzung zur räumlichen Interpolation, die *Aggregation* verschiedener Eingabeparameter, ist weiteres Ziel dieser Diplomarbeit. Insbesondere soll auch hierfür die Nutzung von unscharfen Zahlen als Eingabeparameter ermöglicht werden. Die Art von Funktionen, die hier zur Aggregation dienen können, wird in Kapitel 1.3 näher beschrieben.

Ein Beispiel für den Einsatz einer Aggregationsfunktion stellt die Abschätzung der Bodenqualität zum Anbau etwa von Mais dar. Als Eingabeparameter kämen in diesem Fall diverse Bodenparameter in Betracht, z.B. der Tongehalt der Erde. Ein anderes Anwendungsbeispiel, das ebenfalls auf Bodenparametern basiert, wäre die Unterstützung zur Beurteilung eines Standorts als Mülldeponie.

[1] Unscharfe Zahlen (fuzzy numbers) werden in Kapitel 2.1.1 definiert.

Zusammensetzen kann man die Krigingfunktionen und die Aggregationsfunktion nun auf zwei verschiede Weisen, um aus allen bekannten Eingabewerten und der Koordinate *x* einen aggregierten Wert an der Stelle *x* zu ermitteln:

(*KA*) Zunächst wird die Interpolation für alle Einzelparameter an der Stelle *x* durchgeführt und dann die Aggregationsfunktion auf diese Resultate angewendet.

(*AK*) Zunächst wird die Aggregation für die Stellen, an denen alle Werte gegeben sind, durchgeführt, und dann wird ein Wert an der Stelle *x* auf Grundlage dieser Aggregationsresultate interpoliert.

Bemerkung: Falls *x* eine der Koordinaten ist, an dennen die Werte für die Einzelparameter gegeben sind, dann ergibt sich in beiden Fällen das gleiche Resultat, falls man normales Kriging oder Fuzzy Kriging zur Interpolation verwendet.

Um die Entscheidung zwischen den obengenannten Varianten zu erläutern, folgt hier eine Liste von Vor- und Nachteilen der Verfahren:

- Wenn man die Interpolationen zuerst durchführt (*KA*), kann man die Zwischenergebnisse leicht als Karte darstellen. Da das Endergebnis auf diesen Zwischenergebnissen beruht, kann man die Berechnung bei *KA* etwas besser veranschaulichen als bei *AK*.
  Bei *AK* auf der anderen Seite fallen die Karten der Eingabeparameter nicht als Zwischenergebnisse an, und man muß sie gesondert berechnen, falls deren Darstellung erwünscht wird.
- Kriging zuerst für alle Eingabeparameter durchzuführen (*KA*) ist aufwendig, da man die dem Kriging zugrundeliegende statistische Analyse in Form von Variogrammen für jeden der Eingabeparameter durchführen muß. Dies erfordert sowohl mehr Zutun des Anwenders als auch mehr Rechenzeit.
- Wenn man das Kriging zum Schluß durchführt (*AK*), ist es leicht möglich die Kriging-Varianz zu ermitteln (eine Schätzvarianz zu schätzen - *zu schätzen*, da diese Schätzvarianz ebenso wie das Kriging-Ergebnis auf derselben statistischen Analyse in Form von einem Variogramm beruht). Die Kriging-Varianz zeigt, an welchen Stellen die Schätzungen weniger genau sind.
  Bei *KA* ist es theoretisch auch möglich, diese Kriging-Varianz für das Endergebnis zu berechnen; dies ist jedoch auf theoretischer Seite sehr aufwendig, da die Kriging-Varianzen der Krigingoperationen für jeden Eingabeparameter der Aggregationsfunktion entsprechend verknüpft werden müßten.

Da die Angabe der Kriging-Varianz des Endergebnisses sehr wichtig sein kann und da der erste Punkt dadurch entschärft wird, daß die Berechnung der Karten der Eingabeparameter immerhin *möglich* ist, soll als zusammengesetzte Funktion *AK* benutzt werden.

1.2 Überblick

In Kapitel 1.3 und 1.4 werden Fuzzy Aggregation und Fuzzy Kriging schon einmal in grober Form vorgestellt.

In Kapitel 2 wird die Theorie zu diesen Vorgehensweisen und Verfahren vorgestellt. Zum einen werden allgemeine (aber hierfür notwendige) theoretische Grundlagen in Kapitel 2.1 dargestellt; zum anderen werden Fuzzy Aggregation und Fuzzy Kriging (vom Typ 1) in den Kapiteln 2.2 und 2.3 detailliert erläutert. In diesen Kapiteln werden auch weitere Voraussetzungen für eine Implementierung geschaffen.

Kapitel 3 zeigt dann als Bestandteile der Implementierung sowohl die Benutzerschnittstelle (Kapitel 3.1) als auch die notwendigen Datenstrukturen (Kapitel 3.2) sowie die gewählte Programmstruktur und einige ausgewählte Algorithmen (Kapitel 3.3).

Zur Diplomarbeit wurde ein System erstellt, in dem die beiden Methoden Fuzzy Aggregation und Fuzzy Kriging (vom Typ 1) realisiert sind. Es heißt FUZZEKS (Fuzzy Evaluation and Kriging System), läuft unter Windows 3.1 und beinhaltet einen ausführlichen Windows-Hilfetext, in dem der Umgang mit dem System erläutert wird (ein Hypertext-Dokument, das auch Abbildungen und ein Register enthält).

Der Fuzzy Kriging-Teil aus dem System wurde schon für Anwendungen mit hydrogeologischen Daten genutzt. Nachzulesen ist dies in mehreren Veröffentlichungen [Piotrowski/Bartels/Salski/Schmidt 1994 bis 1996], bei denen der Autor dieser Diplomarbeit selbst der zweite Autor ist, sowie in der Diplomarbeit [Schmidt 1994] und der Doktorarbeit [Rumohr, 1996]. Die Anwendung des Gesamtkonzepts ist aber auch geplant.

1.3 Fuzzy Aggregation von Eingabeparametern

Das Ziel ist, die Parameter aggregieren zu können. Dafür ist zunächst einmal von Bedeutung, welche Funktionen man zur Aggregation benutzen kann. Prinzipiell soll die Aggregationsfunktion vom Anwender festgelegt werden.

Für diese Diplomarbeit wird dafür folgendes Konzept benutzt: Die Aggregationsfunktion soll aus bestimmten Aggregationsoperatoren zusammengesetzt werden. Damit eine günstige Auswahl von Standardoperatoren zur Verfügung gestellt werden kann, ist es zweckmäßig, die Eingabeparameter zunächst zu transformieren, um sie auf eine gemeinsame Skala zu bringen.

Man stelle sich beispielsweise vor, das Endergebnis der Aggregationsfunktion möge der Zugehörigkeitswert zu "Gesamtbeurteilung des geplanten Deponiestandortes positiv" sein. Zu diesem Aggregationsergebnis liegt es inhaltlich nahe, z.B. zum Parameter "Tongehalt" (des Bodens in %) eine Zugehörigkeitsfunktion zur Eigenschaft "Tongehalt hoch" zu definieren, da dies von Vorteil für eine Deponie ist. Solche Zugehörigkeitsfunktionen sollen als Transformationsfunktionen für die Parameter benutzt werden; die transformierten Werte sind dann Zugehörigkeitswerte aus dem Intervall [0,1] (der gemeinsamen Skala).

Bemerkung: In Kapitel 2.2.1 wird ein Zusammenhang zwischen diesen Zugehörigkeitsfunktionen und dem Begriff der linguistischen Variablen hergestellt.

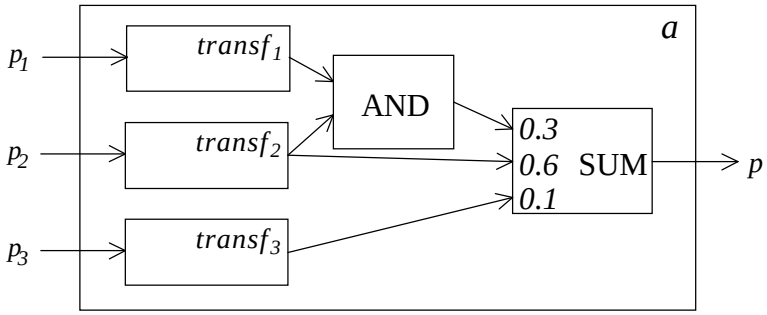


Abbildung 1.3: Beispielstruktur einer Aggregationsfunktion $a(p_1,p_2,p_3) = 0.3 \cdot (transf_1(p_1)$ AND $transf_2(p_2)) + 0.6 \cdot transf_2(p_2) + 0.1 \cdot transf_3(p_3)$

Die Auswahl der Aggregationsoperatoren im Rahmen dieser Diplomarbeit:

- AND: Die Und-Verknüpfung für unscharfe Mengen, implementiert als Minimum der Eingabe-Zugehörigkeitswerte.
- OR: Die Oder-Verknüpfung für unscharfe Mengen, implementiert als Maximum der Eingabe-Zugehörigkeitswerte.
- SUM: Die gewichtete Summe der Eingabe-Zugehörigkeitswerte. Die Gewichte kann der Anwender festlegen; sie müssen jedoch größer als 0 sein; die Summe der Gewichte muß 1 ergeben. Dieser Aggregationsoperator wird in [Burrough 1989] vorgestellt.

Die Aggregationsfunktion setzt sich also aus den Transformationsfunktionen und obigen arithmetischen und logischen Operatoren zusammen. Laut Kapitel 1.1 soll sie auch unscharfe Zahlen abbilden können; Resultat ist dann eine unscharfe Zahl (oder eine scharfe Zahl als Sonderfall unscharfer Zahlen, siehe dazu auch Kapitel 2.1.1).

1.4 Fuzzy Kriging als Methode zur räumlichen Interpolation

Wie in Kapitel 1.1 schon erwähnt, bezeichnet der Begriff Kriging in der Geostatistik übliche Interpolationsverfahren, die auf einer statistischen Analyse in Form von Variogrammen beruhen. Variogramme beschreiben die Abhängigkeit zwischen den Werten eines Parameters bezüglich ihrer räumlichen Anordnung (die Parameter werden auch als sog. *regionalisierte Variable* bezeichnet - Variable, die die Werte einer Größe in Abhängigkeit vom Ort angeben).

Das *Variogramm* γ zu einem Parameter *p* (auch Semivariogramm genannt) ist für Punkte x_1 und x_2 folgendermaßen definert (VAR(*X*) ist die Bezeichnung der Varianz von *X*):

$$\gamma(x_1,x_2) := \mathrm{VAR}(p(x_1) - p(x_2)) / 2$$

Die statistische Analyse beginnt man mit der Berechnung des sog. *experimentellen Variogramms*, bei dem zu tatsächlich bei den Eingabedaten vorkommenden Abständen oder Klassen solcher Abstände jeweils die halbe Varianz ermittelt wird. Dies wird i.a. in einer zweidimensionalen Graphik dargestellt (auf der x-Achse die Abstände und auf der y-Achse die Varianzen aufgetragen). Für den Fall, daß nicht nur die Abstände zur Untersuchung herangezogen werden sollen, sondern auch die Richtung des Differenzvektors zwischen den Koordinaten, werden solche Graphiken für jede Richtung (oder jede Klasse von Richtungen) erstellt.

Das sogenannte *theoretische Variogram* ist eine vom Anwender definierte Funktion, die als Basis für die Kriginginterpolation dient. Die Kurve für diese Funktion wird in FUZZEKS, dem System zu dieser Diplomarbeit, in derselben Graphik dargestellt, wie das experimentelle Variogramm. Der Anwender legt das theoretische Variogramm so fest, daß es eine möglichst gute Anpassung an die durch das experimentelle Variogramm vorgegebenen Punkte darstellt. Zur Definition des theoretischen Variogramms werden Standardfunktionen mit bestimmten Parametern zur Auswahl gestellt, auch eine Linearkombination solcher Funktionen ist möglich.

Krigingverfahren beruhen weiterhin auf bestimmten Modellannahmen, insbesondere einer *Hauptkriginggleichung*, mit der man den Parameter zu beschreiben versucht. Beim *normalen Kriging* (*Ordinary Kriging*) ist die Hauptkriginggleichung:

$$K_\gamma(p(x_1), ..., p(x_n), x) = \sum_{i=1}^{n} \delta_i(x) * p(x_i)$$

Die $\delta_i(x)$ werden ermittelt, indem man eine auf dem theoretischen Variogramm γ basierende Gleichung für die *Kriging-Varianz* (*Schätzvarianz*) aufstellt (mit Hilfe zusätzlicher Modellannahmen), und dann die $\delta_i(x)$ ermittelt, bei denen die Schätzvarianz minimal ist (siehe Kapitel 2.3.1.2).

In Abbildung 1.4 wird die logische Struktur des eben beschriebenen Vorgehens zur besseren Übersicht noch einmal in Form eines Diagramms dargestellt. Die Kennzeichnung der unscharfen Werte gilt allerdings nur für Fuzzy Kriging vom Typ 1 (siehe unten).

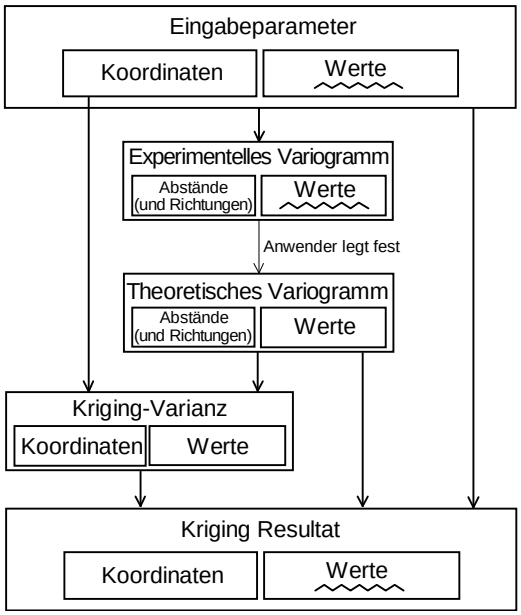


Abbildung 1.4: Struktur von Fuzzy Kriging vom Typ 1
(unscharfe Werte sind unterstrichen)

Fuzzy Kriging ist eine Erweiterung von normalem Kriging. Es gibt verschiedene Möglichkeiten unscharfe Zahlen in die oben beschriebene Vorgehensweise einzubeziehen ([Bardossy 1989]):

	Eingabedaten	Theoretisches Variogramm
Fuzzy Kriging, Typ 1	fuzzy	non-fuzzy
Fuzzy Kriging, Typ 2	non-fuzzy	fuzzy
Fuzzy Kriging, Typ 3	fuzzy	fuzzy

Bemerkung: Bei allen Typen erhält man unscharfe Ausgabedaten.

Ein Unscharfes theoretisches Variogramm kann man nutzen, um die Anpassung des theoretischen Variogramms an das experimentelle Variogramm zu erleichtern oder zu verbessern.

Was im Rahmen dieser Diplomarbeit allerdings hauptsächlich interessiert, ist die Möglichkeit, unscharfe Eingabedaten nutzen zu können. Damit wird also die Entscheidung für Fuzzy Kriging vom Typ 1 getroffen, was im folgenden auch einfach *Fuzzy Kriging* genannt wird.

Von der Unterscheidung in Typ 1 bis Typ 3 unberührt bleibt die Möglichkeit, das experimentelle Variogramm unscharf zu berechnen. Das erleichtert oder verbessert dann aufgrund zusätzlicher Informationen auch die Variogrammanpassung.

2 Theoretische Grundlagen

In Kapitel 2.1 werden zunächst die unscharfen Zahlen (fuzzy numbers) definiert, da sie Grundlage der weiteren Betrachtungen sind. Außerdem werden die Grundlagen für die Vorgehensweisen, die in den folgenden Kapiteln benutzt werden, dargestellt.

In Kapitel 2.2 wird definiert, welche Aggregationsfunktionen verwendet werden können und wie die Funktionen auf unscharfe Zahlen zu erweitern sind. Kapitel 2.3 zeigt entsprechendes für die Kriging-Interpolation.

2.1 Unscharfe Zahlen, ihre Darstellung und Verknüpfung

Ziel der Definition von unscharfen Zahlen ist, ungenau gegebene Daten, wie z.B. "ungefähr 10", als Eingabe nutzen zu können. Dazu muß die Ungenauigkeit (also das "ungefähr" aus "ungefähr 10") präzisiert werden. Die Fuzzy-Set-Theorie stellt dafür Formalisierungsmöglichkeiten, insbesondere den Begriff der unscharfen Zahl (fuzzy number), zur Verfügung.

Funktionen, deren Argumente reelle Zahlen sind, sollen so erweitert werden, daß auch unscharfe Zahlen berücksichtigt werden können. In Kapitel 2.1.3 werden die dafür üblichen Methoden dargestellt.

Um unscharfe Zahlen verwenden zu können, ist es erforderlich, sich mit ihrer Darstellung zu beschäftigen. Am Ende von Kapitel 2.1.4 werden verschiedene Darstellungsmöglichkeiten angesprochen. Es zeigt sich, daß die sogenannte Schnittdarstellung eine besonders günstige Basis dazu bietet. In den Kapiteln 2.1.2 und 2.1.4 werden die Schnittdarstellung und ihre Auswirkungen auf die Erweiterung von Funktionen auf unscharfe Zahlen gezeigt. In Kapitel 3.2.1 wird dieser Ansatz aufgegriffen, um zu einer Implementierung von unscharfen Zahlen zu gelangen.

2.1.1 Definition von unscharfen Zahlen

Als Grundlage zur Definition von unscharfen Zahlen werden zunächst unscharfe Mengen definiert. Unscharfe Mengen sind eine Verallgemeinerung des (scharfen) Mengenbegriffs.

Definition (unscharfe Menge, fuzzy set): Gegeben sei ein Grundbereich X. Eine unscharfe Menge A über X wird durch Angabe ihrer sog. Zugehörigkeitsfunktion $m_A: X \to [0,1]$ definiert.

$m_A(x)$ wird als Zugehörigkeitswert von x zur Menge A bezeichnet.

$\boldsymbol{F}(X)$ bezeichne die Menge aller unscharfen Mengen über X.

Null als Zugehörigkeitswert von x zu einer unscharfen Menge A soll bedeuten, daß x kein Element der unscharfen Menge ist. Eins als Zugehörigkeitswert hingegen soll bedeuten, daß x Element der unscharfen Menge ist.

Die Werte dazwischen geben eine graduelle Zugehörigkeit an; insbesondere soll x als eher (oder "mehr") zur Menge A gehörig als y angesehen werden, wenn der Zugehörigkeitsgrad von x zu A größer als der von y zu A ist ($m_A(x)>m_A(y)$).

Da bei (gewöhnlichen, scharfen) Teilmengen M von X ein Element des Grundbereichs X nur vollständig zu M gehören kann oder gar nicht, bettet man die Menge der (scharfen) Teilmengen M von X in die Menge der unscharfen Mengen über folgende Festlegung ein:

$$m_M(x) := \begin{cases} 1 & x \in M \\ 0 & \text{sonst} \end{cases} .$$

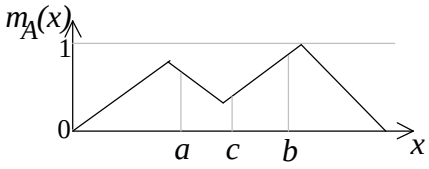
m_M ist dann die Zugehörigkeitsfunktion zu der als unscharfe Menge aufgefaßten Teilmenge M von X.

Zur Definition der unscharfen Zahl soll zunächst noch der Begriff 'konvex' eingeführt werden.

Definition (konvex): Eine unscharfe Menge A über $\mathbb{R}$ nennt man konvex, wenn ihre Zugehörigkeitsfunktion m_A keine Nebenmaxima hat, d.h. für alle $a,b,c \in \mathbb{R}$ muß gelten:

$$c \in [a,b] \quad \Rightarrow \quad m_A(c) \geq \min(m_A(a), m_A(b))$$

Beispiel: Folgende unscharfe Menge A über $\mathbb{R}$ ist *nicht* konvex, da z.B. bei den eingezeichneten a, b und c die Konvexitätsbedingung verletzt wird:



Definition (unscharfe Zahl, fuzzy number): Eine unscharfe Menge A über $\mathbb{R}$ nennt man unscharfe Zahl, wenn sie konvex ist und *genau* ein $x \in X$ existiert mit $m_A(x)=1$.

FN bezeichne die Menge aller unscharfen Zahlen.

Die Menge der (scharfen) reellen Zahlen bettet man in die Menge der unscharfen Zahlen über folgende Festlegung der Zugehörigkeitsfunktion für reelle Zahlen r ein:

$$m_r(x) := \begin{cases} 1 & x = r \\ 0 & \text{sonst} \end{cases}$$

Eine weitere, etwas anschaulichere Charakterisierung von unscharfen Zahlen erfolgt im nächsten Kapitel mit Hilfe des Begriffs 'α-Schnitt' $A^{>\alpha}$.

2.1.2 Schnittdarstellung von unscharfen Zahlen

Definition (α-Schnitt $A^{>\alpha}$, scharfer α-Schnitt $A^{\geq\alpha}$): Zu einer unscharfen Menge A über X und $\alpha\in[0,1]$ wird definiert:

$\underline{\alpha\text{-Schnitt}}$: $\quad A^{>\alpha} := \{\, x\in X \mid m_A(x)>\alpha \,\}$

$\underline{\text{scharfer } \alpha\text{-Schnitt}}$: $\quad A^{\geq\alpha} := \{\, x\in X \mid m_A(x)\geq\alpha \,\}$

Bemerkung (Zusammenhang mit unscharfen Zahlen): Die Konvexitätsbedingung für unscharfe Zahlen läßt sich nun mit Hilfe von folgendem Satz dadurch ersetzen, daß man fordert, daß die α-Schnitte für $\alpha\in[0,1)$ oder die scharfen α-Schnitte für $\alpha\in(0,1]$ Intervalle sind (jede der beiden Seiten dieser Intervalle können beim Schnitt und beim scharfen Schnitt offen oder abgeschlossen sein, da keine Stetigkeitsvoraussetzungen für die Zugehörigkeitsfunktion gemacht wurden). Dies ergibt eine etwas anschaulichere Definition für unscharfe Zahlen.

Eine Teilmenge I von $\mathbb{R}$ wird im folgenden als $\underline{\text{Intervall}}$ bezeichnet, wenn gilt: $I=\emptyset$ oder $\forall\, c\in\mathbb{R}$ mit $\inf(I)<c<\sup(I)$: $c\in I$ (*Bem.*: Infimum und Supremum sind in diesem Zusammenhang Funktionen inf, sup: $2^{\mathbb{R}}\setminus\{\emptyset\} \rightarrow \mathbb{R}\cup\{-\infty,\infty\}$). Diese Definition von Intervallen umfaßt alle offenen, halboffenen, abgeschlossenen und uneigentlichen Intervalle, sowie die leere Menge.

Satz 1: Sei A eine unscharfe Menge. Dann sind äquivalent:

(1) A ist konvex

(2) $A^{>\alpha}$ ist für alle $\alpha\in[0,1)$ ein Intervall

(3) $A^{\geq\alpha}$ ist für alle $\alpha\in(0,1]$ ein Intervall

Beweis: (Bemerkung: Die Markierungen $^{(*)}$ zeigen die Stellen, an denen sich (1)$\Rightarrow$(3) von (1)$\Rightarrow$(2) unterscheidet.)

(1)$\Rightarrow$(2): Sei A konvex und $\alpha\in[0,1)$.
Fall $A^{>\alpha}=\emptyset$: $A^{>\alpha}$ ist das leere Intervall.
Fall $A^{>\alpha}\neq\emptyset$: Zu zeigen: $\forall\, c\in\mathbb{R}$ mit $\inf(A^{>\alpha}) < c < \sup(A^{>\alpha})$: $c\in A^{>\alpha}$.
Falls $\inf(A^{>\alpha}) = \sup(A^{>\alpha})$, dann ist nichts zu zeigen (Es handelt sich um das einpunktige, abgeschlossene Intervall $[\inf(A^{>\alpha}),\inf(A^{>\alpha})]$).
Sonst sei $c\in\mathbb{R}$ mit $\inf(A^{>\alpha}) < c < \sup(A^{>\alpha})$.
Dann gibt es $a,b\in\mathbb{R}$ mit $\inf(A^{>\alpha}) \leq a < c < b \leq \sup(A^{>\alpha})$ und $a,b\in A^{>\alpha}$ (Eigenschaft von inf und sup).
Dann gilt, weil A konvex ist, $m_A(c)\geq\min(m_A(a),m_A(b))$. Da $m_A(a)$ und $m_A(b)$ größer$^{(*)}$ als α sind ($a,b\in A^{>\alpha}$), gilt zudem$^{(*)}$ $\min(m_A(a),m_A(b))>\alpha$. Zusammengenommen also folgt$^{(*)}$ $m_A(c) > \alpha$, also $c\in A^{>\alpha}$.

(1)$\Rightarrow$(3): Ist analog (1)$\Rightarrow$(2) zeigbar. An den Stellen $^{(*)}$ gibt es den Unterschied, daß "$\geq$" anstatt ">" benutzt werden muß. Anstatt "$A^{>\cdots}$" muß natürlich immer "$A^{\geq\cdots}$" eingesetzt werden. Dies bewirkt aber jeweils keine Änderung der Folgerungen.

(2)⇒(1): Gelte für alle $\alpha\in[0,1)$, daß $A^{>\alpha}$ ein Intervall ist (also falls $A^{>\alpha}\neq\emptyset$: Für alle $c\in\mathbb{R}$ mit $\inf(A^{>\alpha}) < c < \sup(A^{>\alpha})$ gilt: $c\in A^{>\alpha}$).
Seien $a,b,c\in\mathbb{R}$ mit $c\in[a,b]$.
Sei $\beta:=\min(m_A(a),m_A(b))$.
Falls $\beta=0$ oder $c=a$ oder $c=b$ folgt $\boldsymbol{m_A(c) \geq \min(m_A(a),m_A(b))}$ trivialerweise.
Sonst gilt $\beta>0$ und (laut Definition von β und γ-Schnitt):
$\forall\gamma \in [0,\beta) : a \in A^{>\gamma} \wedge b \in A^{>\gamma}$, also $A^{>\gamma}\neq\emptyset$.
Für jedes $\gamma\in[0,\beta)$ ist laut Voraussetzung $A^{>\gamma}$ ein Intervall, also, da $A^{>\gamma}\neq\emptyset$, folgt mit $\inf(A^{>\gamma}) \leq a < c < b \leq \sup(A^{>\gamma})$, daß $c\in A^{>\gamma}$ gilt.
Insgesamt gilt also $\forall\gamma \in [0,\beta) : c \in A^{>\gamma}$.
Daraus folgt $m_A(c)\geq\beta$ (denn wenn $m_A(c)<\beta$ wäre, dann würde für jene γ mit $m_A(c)<\gamma<\beta$ gelten: $\gamma\in[0,\beta)$ und $m_A(c)<\gamma$ (also $c \notin A^{>\gamma}$), was im Widerspruch zu obigem Ergebnis steht).
Laut Definition von β bedeutet das $\boldsymbol{m_A(c) \geq \min(m_A(a),m_A(b))}$.

(3)⇒(1): Gelte für alle $\alpha\in[0,1]$, daß $A^{\geq\alpha}$ ein Intervall ist (also falls $A^{\geq\alpha}\neq\emptyset$: Für alle $c\in\mathbb{R}$ mit $\inf(A^{\geq\alpha}) < c < \sup(A^{\geq\alpha})$ gilt: $c\in A^{\geq\alpha}$) ($A^{\geq 0}$ ist immer ein Intervall).
Seien $a,b,c\in\mathbb{R}$ mit $c\in[a,b]$.
Falls $c=a$ oder $c=b$ folgt $\boldsymbol{m_A(c) \geq \min(m_A(a),m_A(b))}$ trivialerweise.
Sei also $a<c<b$. Sei $\beta:=\min(m_A(a),m_A(b))$.
Da in dieser Situation $A^{\geq\beta}\neq\emptyset$, gilt dann:
$\inf(A^{\geq\beta}) \leq \inf(A^{\geq m_A(a)}) \leq a < c < b \leq \sup(A^{\geq m_A(b)}) \leq \sup(A^{\geq\beta})$.
In dieser Situation kann man die Voraussetzung ausnutzen, daß $A^{\geq\beta}$ ein Intervall ist (da $\beta\in[0,1]$); es ergibt sich $c\in A^{\geq\beta}$. Daraus folgt $m_A(c) \geq \beta$, also $\boldsymbol{m_A(c) \geq \min(m_A(a),m_A(b))}$. □

Bemerkung (Eigenschaften von α-Schnitten für unscharfe Zahlen): (Scharfe) Schnitte für kleineres α enthalten diejenigen für größeres α als Teilmenge; Die Intervalle werden gewissermaßen nach oben hin kleiner. Für den scharfen 0-Schnitt gilt: $A^{\geq 0} = \mathbb{R}$. Der scharfe 1-Schnitt enthält genau ein Element.

Der folgende Darstellungssatz (siehe auch [Bandemer/Gottwald], [Kruse 1993], [Kandzia 1995]) liefert die Möglichkeit, die Zugehörigkeitsfunktion einer unscharfen Menge aus den α-Schnitten (für $\alpha\in[0,1)$) bzw. aus den scharfen α-Schnitten (für $\alpha\in(0,1]$) zurückzugewinnen. Dies bedeutet, daß sich unscharfe Mengen (und damit auch unscharfe Zahlen) als Alternative zur Zugehörigkeitsfunktion m_A auch mit einer Menge von Schnitten (also gewöhnlichen Mengen) darstellen lassen.

Darstellungssatz: Sei A eine unscharfe Menge über X und sei $x\in X$. Dann gilt (χ_M ist für Mengen M die charakteristische Funktion $\chi_M: M\to\{0,1\}$; $\chi_M(x) := \begin{cases} 1 & \text{falls } x \in M \\ 0 & \text{falls } x \notin M \end{cases}$):

$$m_A(x) = \sup_{\alpha\in[0,1)} (\alpha * \chi_{A^{>\alpha}}(x)) = \sup_{\alpha\in[0,1)} (\min\{\alpha, \chi_{A^{>\alpha}}(x)\})$$

und entspechendes für scharfe Schnitte:

$$m_A(x) = \sup_{\alpha\in(0,1]} (\alpha * \chi_{A^{\geq\alpha}}(x)) = \sup_{\alpha\in(0,1]} (\min\{\alpha, \chi_{A^{\geq\alpha}}(x)\}).$$

2.1.3 Das Erweiterungsprinzip

Das Ziel ist, eine Funktion $g: X^n \to Y$ zu einer unscharfen Funktion $\hat{g}: \boldsymbol{F}(X)^n \to \boldsymbol{F}(Y)$ zu erweitern, die scharfe Mengen (als Spezialfall von unscharfen Mengen) genauso abbildet wie g und andere unscharfe Mengen in einer natürlichen Art und Weise g entsprechend abbildet. Für unscharfe Zahlen (als Spezialfall von unscharfen Mengen, $X=\mathbb{R}$) kann dieselbe Erweiterung von $g:\mathbb{R}^n \to \mathbb{R}$ benutzt werden, jedoch muß man beachten, daß nicht jede so erweiterte Funktion unscharfe Zahlen immer auf unscharfe Zahlen abbildet ($\hat{g}: \boldsymbol{FN}^n \to \boldsymbol{F}(\mathbb{R})$).

Die übliche Methode zur Erweiterung von Funktionen auf unscharfe Funktionen (mit obengenannten Eigenschaften) ist das Erweiterungsprinzip:

Definition (Erweiterungsprinzip): Sei $n \in \mathbb{N}_0$ und $g: X^n \to Y$ gegeben. Dann wird $\hat{g}: \boldsymbol{F}(X)^n \to \boldsymbol{F}(Y)$ folgendermaßen für alle $y \in Y$ und $A_1,...,A_n \in \boldsymbol{F}(X)$ (mit Setzung von $\sup \varnothing := 0$ und $\min \varnothing := 1$) definiert:

$$m_{\hat{g}(A_1,...,A_n)}(y) := \sup_{\substack{(x_1,...,x_n)\in X^n \\ y=g(x_1,...,x_n)}} \min\{m_{A_1}(x_1), ..., m_{A_n}(x_n)\}$$

Die Fortsetzung von sup auf leere Mengen ist erforderlich, um Funktionen, die nicht alle Werte aus Y treffen, erweitern zu können. Für $y \in Y$, die nicht als Funktionswert von g vorkommen, ergibt sich nämlich $m_{\hat{g}(A_1,...,A_n)}(y) := \sup \varnothing$.

Die Fortsetzung von min auf leere Mengen stellt sicher, daß die konstante Funktion (und damit Konstanten aus Y) natürlich erweitert werden.

Bemerkung: Man beachte, daß diese Definition nur für Funktionen g mit einem Urbildbereich X^n (und nicht etwa beliebigen Teilmengen davon) etwas definiert. Man kann die Definition zwar für $X_1 \times \cdots \times X_n$ anstelle von X^n durchführen (was hier nicht benötigt wird), wenn man sie jedoch für Mengen wie z.B. {(0,0), (0,1), (1,0)} (bei $X=\{0,1\}$ und $n=2$) anstelle von X^n durchführen würde, ergäben sich Unterschiede in der weiteren Verwendung. Diese Problematik wird in Kapitel 2.2.3 aufgegriffen.

Daß das Erweiterungsprinzip keine ganz willkürliche Festlegung ist, kann man mit Hilfe des Begriffs "Akzeptanzgrad" ([Kruse 1993], [Kandzia 1995]) zeigen.

Definition (Akzeptanzgrad): Die Akzeptanz *acc* ist eine Abbildung von einer Menge von Aussagen P in das reelle Intervall [0,1] ($acc: P \to [0,1]$). Prinzipiell ordnet man folgendermaßen Bildwerte zu ($acc(p)$ wird als Akzeptanzgrad der Aussage p bezeichnet):

$$acc(p) := \begin{cases} 0 & \text{falls } p \text{ definitiv falsch} \\ 1 & \text{falls } p \text{ definitiv wahr} \\ \text{zwischen 0 und 1} & \text{falls } p \text{ graduell wahr} \end{cases}$$

Die exakte Zuordnung läßt sich induktiv definieren, falls die Menge der möglichen Aussagen induktiv mit Hilfe von Verknüpfungen definiert wurde und zu den einzelnen Verknüpfungen ein Akzeptanzgrad definiert ist.

Um die Analogie zur Komplexfunktion (s.u.) zu zeigen, sei P folgendermaßen induktiv definiert:

- Wenn x ein Variablenbezeichner für eine Variable mit Werten aus X ist, und A der Bezeichner einer unscharfen Menge, dann ist "$x \in A$" Element von P.
- Wenn p und q Elemente von P sind, dann sind auch "$(p \wedge q)$", "$(p \vee q)$" und "$(\neg p)$" Elemente von P.
- Wenn $p(x)$ Element von P ist, das x als Bezeichner für eine freie (nicht an $\forall$ oder $\exists$ gebundene) Variable mit Werten aus X enthält, dann sind auch "$(\forall\, x \in X: p(x))$" und "$(\exists\, x \in X: p(x))$" Element von P.
- Nur nach obigen Regeln entstandene Ausdrücke sind in P enthalten.

Dazu wähle man als Akzeptanzgrad zu den Elementen aus P (siehe auch [Kruse 1993]):

- Wenn x ein Variablenbezeichner für eine Variable mit Werten aus X ist, und A der Bezeichner einer unscharfen Menge:
  $acc("x \in A") := m_A(x)$
- Wenn p und q Elemente von P sind:
  $acc("(p \wedge q)") := \min(acc("p"), acc("q"))$
  $acc("(p \vee q)") := \max(acc("p"), acc("q"))$
  $acc("(\neg p)") := 1\text{-}acc("p")$
  $acc("(p)") := acc("p")$
- Wenn $p(x)$ Element von P ist, das x als Bezeichner für eine freie (nicht an $\forall$ oder $\exists$ gebundene) Variable mit Werten aus X enthält:
  $acc("(\forall\, x \in X: p(x))") := \inf \{acc(p(x)) \mid x \in X\}$
  $acc("(\exists\, x \in X: p(x))") := \sup \{acc(p(x)) \mid x \in X\}$

Außerdem seien folgende Kurzschreibweisen definiert:

- Man darf Klammerpaare ganz außen und laut folgenden Prioritäten weglassen (von höchster zu niedrigster Priorität): $\neg$, $\wedge$, $\vee$
- Man darf "$(\exists\, (x_1,...,x_n) \in X^n: p(x_1,...,x_n))$" anstelle von "$(\exists\, x_1 \in X: \ldots (\exists\, x_n \in X: p(x_1,...,x_n)) \ldots)$" schreiben (für $\forall$ entsprechend).

Nun läßt sich Zugehörigkeitsgrad, der im Erweiterungsprinzip definiert wird, mit Hilfe obiger Definition von P als Akzeptanzgrad einer Aussage schreiben: $m_{\hat{g}(A_1,...,A_n)}(y) = acc("\exists (x_1, ..., x_n) \in X^n : y \doteq g(x_1, ..., x_n) \wedge x_1 \in A_1 \wedge \cdots \wedge x_n \in A_n")$, wobei zusätzlich der Akzeptanzgrad von "$y \doteq g(x_1, ..., x_n)$" folgendermaßen definiert ist: $acc("y \doteq g(x_1, ..., x_n)") := \begin{cases} 1 & \text{falls } y = g(x_1, ..., x_n) \\ 0 & \text{sonst} \end{cases}$

Es liegt nahe *diese* Aussage zu wählen, wenn man die Analogie zur in der Mathematik üblichen Komplexfunktion zu g betrachtet:

Definition (Erweiterung zur Komplexfunktion): Sei $n \in \mathbb{N}_0$ und $g: X^n \to Y$ gegeben. Dann wird die Komplexfunktion $\bar{g} : (2^X)^n \to 2^Y$ zu g für $M_1,...,M_n \subseteq X$ (also für konventionelle Teilmengen von X anstelle unscharfer Mengen wie beim Erweiterungsprinzip) folgendermaßen für $n>0$ definiert:
$\bar{g}(M_1, ..., M_n) := \{y \in Y \mid \exists (x_1, ..., x_n) \in X^n : y = g(x_1, ..., x_n) \wedge x_1 \in M_1 \wedge \cdots \wedge x_n \in M_n\}$
Für $n=0$ sei $\bar{g}( ) := \{y \in Y \mid y = g( )\} = \{g( )\}$.

2.1.4 Realisierungsmöglichkeiten von Verknüpfungen bei Schnittdarstellung

Man kann das Erweiterungsprinzip (siehe 2.1.3) auch mit Hilfe der Schnitte (siehe 2.1.2) formulieren (siehe auch [Bandemer/Gottwald], [Kruse 1993], [Kandzia 1995]). Wenn die unscharfen Mengen in Schnittdarstellung gegeben sind, liefert diese Formulierung eine i.a. sehr einfache (und sehr effizient implementierbare) Möglichkeit, die unscharfe Funktion zu berechnen.

Satz 1 (Schnittweise Formulierung des Erweiterungsprinzips): Sei $n \in \mathbb{N}_0$ und $g: X^n \to Y$ gegeben und sei $\hat{g}: \boldsymbol{F}(X)^n \to \boldsymbol{F}(Y)$ und $A_1,...,A_n \in \boldsymbol{F}(X)$. $\hat{g}$ sei mit Hilfe des Erweiterungsprinzips definiert.
Dann gilt für alle $\alpha \in [0, 1]$:
$(\hat{g}(A_1, ..., A_n))^{>\alpha} = \bar{g}(A_1^{>\alpha}, ..., A_n^{>\alpha})$ (schnittweise Formulierung).
(Dabei ist $\bar{g}$ die übliche Komplexfunktion (siehe Ende von 2.1.3); sie wird oft auch einfach g geschrieben.)

Bemerkung: Laut dem Darstellungssatz aus Kapitel 2.1.2 ist durch die Festlegung der Schnitte auch die Zugehörigkeitsfunktion festgelegt. Daher ist mit Satz 1 die schnittweise Formulierung des Erweiterungsprinzips *äquivalent* zur direkten Festlegung der Zugehörigkeitswerte.

Beweis: Seien die Voraussetzungen wie im Satz gegeben. Z.z.: Schnittweise Formulierung.

$$(\hat{g}(A_1, ..., A_n))^{>\alpha}$$
$$=^{(1)} \{y \in Y \mid (\sup_{\substack{(x_1,...,x_n)\in X^n \\ y=g(x_1,...,x_n)}} \min\{m_{A_1}(x_1), ..., m_{A_n}(x_n)\}) > \alpha\}$$
$$=^{(2)} \{y \in Y \mid \exists (x_1, ..., x_n) \in X^n \text{ mit } y = g(x_1, ..., x_n) : \min\{m_{A_1}(x_1), ..., m_{A_n}(x_n)\} > \alpha\}$$
$$=^{(3)} \{y \in Y \mid \exists (x_1, ..., x_n) \in X^n : y = g(x_1, ..., x_n) \wedge \min\{m_{A_1}(x_1), ..., m_{A_n}(x_n)\} > \alpha\}$$
$$=^{(4)} \{y \in Y \mid \exists (x_1, ..., x_n) \in X^n : y = g(x_1, ..., x_n) \wedge m_{A_1}(x_1) > \alpha \wedge ... \wedge m_{A_n}(x_n) > \alpha\}$$
$$=^{(5)} \{y \in Y \mid \exists (x_1, ..., x_n) \in X^n : y = g(x_1, ..., x_n) \wedge x_1 \in A_1^{>\alpha} \wedge ... \wedge x_n \in A_n^{>\alpha}\}$$
$$=^{(6)} \bar{g}(A_1^{>\alpha}, ..., A_n^{>\alpha})$$

Erläuterungen:

(1) Definition von $\hat{g}$ mit dem Erweiterungsprinzip und Einsetzen in die Definition des α-Schnitts.

(2) Sei $Z := \{(x_1, ..., x_n) \in X^n \mid y = g(x_1, ..., x_n)\}$
und $p((x_1, ..., x_n)) := \min\{m_{A_1}(x_1), ..., m_{A_n}(x_n)\}$.
Dann läßt sich die Umformung so schreiben:
$(\sup_{z\in Z} p(z)) > \alpha \quad \Leftrightarrow \quad \exists z \in Z : (p(z) > \alpha)$.
Sei $\beta := (\sup_{z\in Z} p(z))$. Da $\beta > \alpha$ und *p(z)* Werte beliebig nahe an β annimmt, gibt es auch Werte von *p(z)*, die größer als α sind. Die Rückrichtung ist trivial.

Bemerkung: Diese Äquivalenz gilt i.a. nicht, wenn $\geq \alpha$ anstelle $> \alpha$ auf beiden Seiten vorkommt, denn das Supremum muß bei den *p(z)* nicht vorkommen. Allerdings gibt es Situationen, in denen aufgrund zusätzlicher Annahmen die Äquivalenz trotzdem gültig ist, insbesondere in der Situation, daß die $A_1,...,A_n$ unscharfe Zahlen sind, *y* einen bestimmten Wert hat und $\alpha = 1$ ist, denn die Definition von unscharfen Zahlen verlangt, daß der Zugehörigkeitswert 1 bei den Zugehörigkeitsfunktionen $m_{A_1}, ..., m_{A_n}$ genau einmal vorkommt, man muß also nur *y* dort wählen, dann gibt es eine Kombination der $x_1,...,x_n$, bei denen alle $m_{A_1}(x_1), ..., m_{A_n}(x_n)$ gleich eins sind (somit kommt das Supremum bei den *p(z)* vor). Also stimmt die Gleichheit (2) auch für ≥ 1, also gilt (da alle anderen Umformungen mit $\geq \alpha$ genauso funktionieren, wie mit $> \alpha$) $(\hat{g}(A_1, ..., A_n))^{\geq 1} = \bar{g}(A_1^{\geq 1}, ..., A_n^{\geq 1})$.

(3) Schreibweise geändert.

(4) Wenn das Minimum der $m_{A_1}(x_1), ..., m_{A_n}(x_n)$ größer als α ist, dann müssen alle einzelnen der $m_{A_1}(x_1), ..., m_{A_n}(x_n)$ größer als α sein und umgekehrt.

(5) Definition des α-Schnitts ausgenutzt.

(6) Definition der Komplexfunktion ausgenutzt. □

Die Berechnung der abgebildeten Schnitte reduziert sich laut dem Satz also auf die Berechnung der Komplexfunktion. Dies ist i.a. jedoch sehr einfach, wie man an folgendem Beipiel sieht:

Beispiel (Berechnung der Addition von zwei unscharfen Zahlen in Schnittdarstellung): Die Schnitte von unscharfen Zahlen sind Intervalle, also beispielsweise seien [1,2] und [3,6] die 0-Schnitte. Dann gilt:
$[1,2]+[3,6]=\{y \in \mathbb{R} \mid \exists x_1 \in [1,2], x_2 \in [3,6] : x_1 + x_2 = y\}=[4,8]$. Offensichtlich muß man also nur nach linken und rechten Intervallgrenzen getrennt addieren.

Der Grund für die offensichtliche Einfachheit liegt darin, daß die Addition eine monoton steigende und stetige Funktion ist. Folgende Sätze zeigen den genauen Zusammenhang.

Bemerkung: Sei $n \in \mathbb{N}_0$ und $f:\mathbb{R}^n \to \mathbb{R}$ eine Funktion. Dann ist *f* monoton steigend $:\Leftrightarrow \forall x_1,...,x_n,y_1,...,y_n \in \mathbb{R}$ mit $y_i \geq x_i$ f.a. $i \in \{1..n\}$: $f(y_1,...,y_n) \geq f(x_1,...,x_n)$).

Untersuchungen zur Abbildung von abgeschlossenen Intervallen sind auch Thema der Intervallarithmetik (siehe z.B. [Bauch 1987], [Neumaier 1990]).

Der folgende Satz zeigt zunächst ein Kriterium für Funktionen das sicherstellt, daß Intervalle auf Intervalle abgebildet werden, was nicht selbstverständlich ist. Dieser Satz kann für (beliebige) Intervalle fomuliert werden.

Satz 2 (Eigenschaft stetiger Funktionen): Sei $n \in \mathbb{N}_0$ und $f:\mathbb{R}^n \to \mathbb{R}$ eine stetige Funktion. Dann bildet $\bar{f}$ Intervalle (bzw. nichts bei n=0) auf Intervalle ab.

Beweis: Seien die Voraussetzungen aus dem Satz gültig und seien $I_1,...,I_n$ Intervalle aus $\mathbb{R}$.
Zu zeigen: $\forall c \in (\inf(\bar{f}(I_i,...,I_n)), \sup(\bar{f}(I_i,...,I_n)) : c \in \bar{f}(I_i,...,I_n)$.
Sei also $c \in (\inf(\bar{f}(I_i,...,I_n)), \sup(\bar{f}(I_i,...,I_n))$.
Wähle $a,b \in \mathbb{R}$ mit $\inf(\bar{f}(I_i,...,I_n)) \le a < c < b \le \sup(\bar{f}(I_i,...,I_n)$ und mit der Eigenschaft, daß $a_i \in I_i$ f.a. $i = 1,...,n$ existieren, so daß $a = f(a_1,...,a_n)$ und für b entsprechendes gilt. Dies ist möglich, da entweder das Infimum angenommen wird (a kann dann dieses Infimum sein) oder Werte aus den Intervallen existieren, deren Funktionswerte beliebig nahe am Infimum liegen (Eigenschaft des Infimums).
Da f eingeschränkt auf E ($f|_E:E \to \mathbb{R}$) auf dem zusammenhängenden $E := [a_1,b_1] \times ... \times [a_n,b_n]$ (die a_i und b_i von oben) stetig ist, $a<b$ gilt und $a,b \in \bar{f}([a_1,b_1] \times ... \times [a_n,b_n])$, ist der Satz von Bolzano (manchmal auch Zwischenwertsatz genannt) anwendbar und ergibt für c mit $a<c<b$, daß es $(x_1,...,x_n) \in E$ gibt mit $f(x_1,...,x_n) = c$.
Da $E \subseteq I_1 \times ... \times I_n$, folgt $c \in \bar{f}(I_i,...,I_n)$. □

Folgerung: Sei $f:\mathbb{R}^n \to \mathbb{R}$ eine stetige Funktion. Laut Satz 2 werden von $\hat{f}$, die gemäß dem Erweiterungsprinzip definierte Erweiterung von f, dann auch unscharfe Zahlen auf unscharfe Zahlen abgebildet (zu zeigen unter Ausnutzung der Schnittdarstellung der unscharfen Zahlen: Mit Satz 2 sind die Schnitte der Resultate wieder Intervalle; daß der scharfe 1-Schnitt (als einpunktiges Intervall) wieder auf ein einpunktiges Intervall abgebildet wird, ist ohnehin klar und garantiert, daß die Zugehörigkeitsfunktion an genau einer Stelle den Wert 1 annimmt).

Satz 3 (Verknüpfung abgeschlossener Intervalle bei monotonen und stetigen Funktionen mit Urbildbereich $\mathbb{R}^n$): Sei $n \in \mathbb{N}_0$ und $g:\mathbb{R}^n \to \mathbb{R}$ eine monoton steigende (fallende) und stetige Funktion und $a_1,...,a_n,b_1,...,b_n \in \mathbb{R}$ mit $a_i \le b_i$. Dann bildet die Komplexfunktion von g abgeschlossene Intervalle $[a_1,b_1],...,[a_n,b_n]$ folgendermaßen ab:

$\bar{g}([a_1,b_1],...,[a_n,b_n]) = [g(a_1,...,a_n), g(b_1,...,b_n)]$ (falls g monoton steigend)
$\bar{g}([a_1,b_1],...,[a_n,b_n]) = [g(b_1,...,b_n), g(a_1,...,a_n)]$ (falls g monoton fallend)

Beweis (nur für monoton steigende Funktionen): Seien die Voraussetzungen aus dem Satz gültig (mit monoton steigender Funktion). Laut Definition gilt $\bar{g}([a_1,b_1],...,[a_n,b_n]) = \{y \in \mathbb{R} | \exists x_i \in [a_i,b_i] \text{ f.a. } i \in \{1...n\} : g(x_1,...,x_n) = y\}$.

1. Das Minimum der Elemente der Menge ist $g(a_1,...,a_n)$, da g monoton steigt (somit also die Bildwerte von g auf keinen Fall kleiner werden können, als wenn man die kleinstmöglichen Werte aus den Intervallen einsetzt).
2. Das Maximum ist entsprechend $g(b_1,...,b_n)$.
3. Daß es tatsächlich ein Intervall ist (also daß die Werte zwischen $g(a_1,...,a_n)$ und $g(b_1,...,b_n)$ vorkommen), zeigt Satz 2 (da f stetig ist). □

Bemerkungen zur Anwendung der Sätze in Bezug auf unscharfe Zahlen:

Wie schon angedeutet, kann man diese Sätze auf die Abbildung von unscharfen Zahlen in Schnittdarstellung anwenden, wenn nur abgeschlossene Intervalle unter den α-Schnitten mit $\alpha \in [0,1)$ vorkommen. Leer können diese α-Schnitte nicht sein, da bei unscharfen Zahlen der Zugehörigkeitswert 1 vorkommt.

Wenn man einen der Sätze auf eine Funktion anwenden kann, dann ist auch klar, daß die (durch das Erweiterungsprinzip erweiterte) Funktion unscharfe Zahlen, deren α-Schnitte mit $\alpha \in [0,1)$ abgeschlosse Intervalle sind, auf unscharfe Zahlen *mit derselben Eigenschaft* abbildet.

Daß man mit Satz 3 nur abgeschlossene Intervalle behandeln kann, scheint auf den ersten Blick sehr unangenehm zu sein.

Die Repräsentation von unscharfen Zahlen und die Anwendung dieser Sätze ist aber hauptsächlich dann von Interesse, wenn es um die *Implementierung* von mit dem Erweiterungsprinzip erweiterten Funktionen geht. Für die *Definition* dieser unscharfen Funktionen würde auf theoretischer Seite das Erweiterungsprinzip vollkommen ausreichen.

Grundlegende Implementierungsalternativen: Um zu einer günstigen Implementierung zu gelangen, muß man sich zunächst fragen, was dazu wichtig ist:

- Die Repräsentation der unscharfen Zahlen ist wichtig: Man möchte die Zugehörigkeitsfunktionen möglichst exakt repräsentieren oder zumindest ihre wichtigsten Merkmale berücksichtigen.
- Die Implementierung der unscharfen Funktionen sollte möglichst einfach sein.
- Effizienz ist natürlich immer von Vorteil (wenig Speicherplatzverbrauch und hohe Verarbeitungsgeschwindigkeit).

Drei grundlegende Alternativen wären für eine Darstellung der unscharfen Zahlen möglich:

Erstens könnte man die Zugehörigkeitsfunktionen als Funktionen (z.B. aus bestimmten Grundtypen zusammengesetzt) repräsentieren, was jedoch zu dem Problem führt, daß bei Anwendung einer unscharfen Funktion neue Zugehörigkeitsfunktionstypen zur Repräsentation des Resultats notwendig werden können, was i.a. dazu führt, daß man manche Funktionen doch nur näherungsweise implementiert oder zu extrem komplizierten Implementierungen gelangt.

Zweitens könnte man die Werteachse diskretisieren, also nur die Zugehörigkeitswerte zu bestimmten Werten des Grundbereichs zur Darstellung heranziehen. Damit die wichtigsten Merkmale der Zugehörigkeitsfunktionen (wie z.B. der Wert aus dem Grundbereich, bei dem der Zugehörigkeitswert 1 ist) ausreichend berücksichtigt werden können, muß die Teilmenge der Werte aus dem Grundbereich bei jeder Abbildung verändert werden (da sonst z.B. der Zugehörigkeitswert 1 evtl. gar nicht mehr vorkäme). Wenn man bei der Repräsentation keine oder wenig Kompromisse machen möchte, dann muß man sie also bei der Effizienz machen, da bei den meisten Abbildungen die Teilmenge der Werte aus dem Grundbereich größer werden muß.

Drittens könnte man die Zugehörigkeitsachse diskretisieren. Der Darstellungssatz liefert eine Rekonstruktion der Zugehörigkeitsfunktion aus den Schnitten zwar nur, wenn die α-Schnitte für alle $\alpha \in [0,1)$ zur Verfügung stehen, jedoch können die wichtigsten Merkmale der Zugehörigkeitsfunktion auch schon mit wenigen Schnitten repräsentiert werden. Zur Implementierung sollen hier der 0-Schnitt, der scharfe 1-Schnitt und eine vom Anwender definierbare Anzahl von α-Schnitten für $\alpha \in (0,1)$ benutzt werden (das bringt nur Sinn, wenn man die Bemerkung im Beweis von Satz 1 miteinbezieht, nämlich die Aussage von Satz 1 mit "≥ 1" anstelle von "$>\alpha$"). Bei Anwendbarkeit von Satz 3 ist die Implementierung von Funktionen sehr einfach und die Effizienz (Speicherplatzverbrauch und Geschwindigkeit) ist sehr gut, da man sich auf die wesentlichen Merkmale der Zugehörigkeitsfunktionen beschränkt und die Abbildungsvorschrift so einfach wie möglich ist.
Bemerkung: Daß man zur Repräsentation von unscharfen Mengen nicht unbedingt eine Menge ausschließlich von Schnitten oder ausschließlich von scharfen Schnitten nehmen muß, wird auch durch die Betrachtung von sogenannten Mengen-Repräsentationen ([Kruse 1993], [Kandzia 1995]) verdeutlicht.

Darstellungs- und Rechenungenauigkeiten: Alle der obigen Methoden beruhen grundsätzlich auch auf der Darstellung reeller Zahlen, wozu man aufgrund Einfachheit und Effizienz normalerweise vorhandene Hardwarestrukturen nutzen möchte. Der Nachteil dabei ist eine (in den meisten Fällen geringe) Darstellungs- und Rechenungenauigkeit.

Aufgrund dieser Ungenauigkeiten ist eine Unterscheidung zwischen *offenen* und *abgeschlossenen* Intervallen in der praktischen Implementierung, außer für unbeschränkte Intervalle, nicht sinnvoll, denn der Unterschied ist unendlich klein. Die Rechenungenauigkeiten treten hingegen in Größenordnungen auf, die man ohne weiteres angeben kann.

Wenn man sich auf beschränkte Intervalle einschränkt, bedeutet es also keinen nennenswerten Verlust für Implementierungen, nur abgeschlossene Intervalle zu behandeln. Unbeschränkte Intervalle braucht man nur, wenn unbeschränkte 0-Schnitte möglich sein sollen, was bei unscharfen Zahlen i.a. nicht benötigt wird. Um beschränkte offene Intervalle durch abgeschlossene Intervalle zu repräsentieren, muß man nur einen unendlich kleinen Fehler in Kauf nehmen.

In der Theorie ist es sehr bequem, wenn man nur abgeschlossene Intervalle behandeln muß, da man dann oft Satz 3 anwenden kann. Wenn man *nur* Funktionen nutzt, die abgeschlossene Intervalle auf abgeschlossene Intervalle abbilden (was im Rahmen dieser Diplomarbeit der Fall sein wird), kann man die theoretischen Vorbereitungen zur Implementierung auch ohne obigen Blick auf die Darstellungs- und Rechenungenauigkeiten machen, da man ja in der Menge der abgeschlossenen Intervalle bleiben kann.

In der Intervallarithmetik ([Bauch 1987], [Neumaier 1990]), in dem Abbildungen von Intervallen behandelt werden, was meist dazu dient, Rechenfehler abzuschätzen, werden i.a. auch nur abgeschlossene Intervalle betrachtet.

Es wäre übrigens keine brauchbare Alternative, nur *offene* Intervalle zu benutzen, denn manche der Funktionen in folgenden Kapiteln bilden viele offene Intervalle auf halboffene Intervalle ab.

2.2 Fuzzy Aggregation von Eingabeparametern

Wie in Kapitel 1.3 beschrieben, sollen die Aggregationsfunktionen aus bestimmten Aggregationsoperatoren zusammengesetzt werden können; damit eine günstige Auswahl von Aggregationsoperatoren zur Verfügung gestellt werden kann, ist es von Vorteil, wenn als Vorbereitung dazu die Eingabeparameter auf eine gemeinsame Skala transformiert werden (siehe auch [Burrough 1989]).

Kapitel 2.2.1 zeigt, welche Transformationen möglich sein sollen und wie sie mit Hilfe des Erweiterungsprinzips zu unscharfen Funktionen erweitert werden. In Kapitel 2.2.2 werden die Aggregationsoperatoren vorgestellt, es werden bestimmte Eigenschaften gezeigt, und erneut wird das Erweiterungsprinzip genutzt, um aus den einzelnen Operatoren unscharfe Funktionen zu machen.

Da das Erweiterungsprinzip nur für Funktionen mit Urbildbereich X^n definiert war, ergibt sich die Schwierigkeit, die Gesamtfunktion aus den Transformationen und insbesondere den Aggregationsoperatoren zu einer unscharfen Gesamtfunktion zu erweitern. Dieses Problem wird in Kapitel 2.2.3 verdeutlicht und eine Lösung wird präsentiert, die auch günstig zu implementieren ist.

2.2.1 Transformation auf eine gemeinsame Skala

Die gemeinsame Skala, die angestrebt wird, sind Akzeptanzwerte aus dem Intervall [0,1] zu Aussagen über die Eingabeparameter; die jeweilige Aussage kann der Anwender festlegen. Beispielsweise könnte "Cl-Konzentration gering" die gewählte Aussage zum Bodenparameter "Cl-Konzentration" sein; die Akzeptanzwerte dazu müssen durch eine Zugehörigkeitsfunktion festgelegt werden. Z.B. für den Parameter "Cl-Konzentration" legt man die Akzeptanz durch eine vom Anwender definierte Zugehörigkeitsfunktion (zu einer unscharfen Menge mit dem Namen "Cl-Konzentration gering") folgendermaßen fest: acc("Cl-Konzentration gering") := $m_{\text{"Cl-Konzentration gering"}}$(Cl-Konzentration).

Man kann dieses Vorgehen auch mit Hilfe des Begriffs der *linguistischen Variablen* erklären: Wenn man den Eingabeparameter als linguistische Variable betrachtet, dann legt man für die Zuordnung linguistischer Werte zu den Werten des Eingabeparameters zunächst eine Menge von linguistischen Werten mit ihren Zugehörigkeitsfunktionen fest. Die Menge muß in diesem Fall genügend linguistische Werte enthalten, um den ganzen eigentlichen Grundbereich (den Bereich der Werte, die zumindest theoretisch vorkommen können) abzudecken. Obige Transformationsfunktion ist eine solche Zugehörigkeitsfunktion, die einen einzelnen Wert der linguistischen Variable definiert. Im Rahmen des Konzepts dieser Diplomarbeit kann nur ein einziger Wert einer solchen linguistischen Variable praktisch festgelegt werden. Der Grund für diese Einschränkung wird in Kapitel 2.2.3 offenbar (die Anwendung von Satz 1 aus Kapitel 2.2.3 wäre sonst am Ende des Kapitels 2.2.3 (vor den Bemerkungen) nicht mehr möglich, was dort auch erwähnt wird).

Die typischerweise auszudrückenden Eigenschaften sind zum Eingabeparameter E:
"E gering", "E hoch" und "E ungefähr vom Wert ...". Folglich werden dementsprechende Zugehörigkeitsfunktionen zur Auswahl gestellt, und zwar zwei verschiedene Alternativen zu jedem eben erwähnten Grundtyp.

Die eine Alternative sind die von Burrough ([Burrough 1989]) vorgeschlagenen Funktionen. Sie enthalten noch zwei Parameter: *s* (eine reelle Zahl größer als 0) legt die Steilheit des Übergangs zum Zugehörigkeitswert 1 fest; *v* (eine reelle Zahl) legt eine Verschiebung auf der Parameterachse fest. Die Funktion nimmt das Maximum 1 bei *v* an, fällt nach beiden Seiten ab und konvergiert für Parameterwerte gegen $\pm\infty$ gegen 0. Die Funktion ist folgendermaßen definiert (für reelle Zahlen *p*) (in Abbildung 2.2.1a ist ein Beispiel dargestellt):

$$transf_{Burrough\text{-}max\text{-}o,s,v}(p) := \frac{1}{1+s(p-v)^2}$$

Um die Variante für "gering" (bzw. "hoch") zu erhalten, setzt man die Funktionswerte unterhalb (bzw. oberhalb) von *v* auf 1:

$$transf_{Burrough\text{-}fallend,s,v}(p) := \begin{cases} 1 & \text{falls } p < v \\ \frac{1}{1+s(p-v)^2} & \text{sonst} \end{cases}$$

$$transf_{Burrough\text{-}steigend,s,v}(p) := \begin{cases} 1 & \text{falls } p > v \\ \frac{1}{1+s(p-v)^2} & \text{sonst} \end{cases}$$

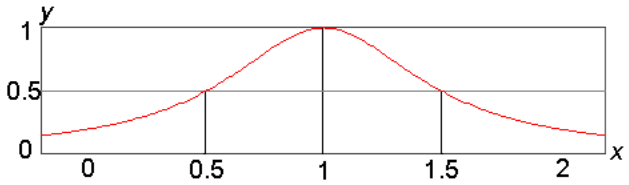


Abbildung 2.2.1a: $y=transf_{Burrough\text{-}max\text{-}o,4,1}(x)$

Folgende Funktion ist eine kleine Erweiterung dieser Möglichkeiten:

$$transf_{Burrough\text{-}max,s1,s2,v1,v2}(p) := \begin{cases} \frac{1}{1+s1(p-v1)^2} & \text{falls } p < v1 \\ 1 & \text{falls } v1 \le p \le v2 \\ \frac{1}{1+s2(p-v2)^2} & \text{falls } v2 < p \end{cases}$$

Sie erlaubt durch die Fallunterscheidung in linken Teil, mittleren Teil und rechten Teil, die Funktion asymmetrisch zu gestalten; im mittleren Teil wird konstant 1 geliefert. $transf_{Burrough\text{-}max\text{-}o,s,v}$ ist ein Spezialfall dieser Funktion, denn es gilt
$transf_{Burrough\text{-}max\text{-}o,s,v} = transf_{Burrough\text{-}max,s,s,v,v}$.

Die zweite Alternative bietet Funktionen, die nicht gegen 0 *konvergieren*, sondern ab einem bestimmten *x*-Wert 0 liefern. Solche Funktionen wurden z.B. in [Huajun 1991] im gleichen Zusammenhang benutzt und mit "S-membership function" bezeichnet. Sie können mit Hilfe folgender Hilfsfunktion definiert werden:

$$std(x_0,y_0,x_1,y_1,gradient,x) := y_0 + dy \cdot \left(\frac{x-x_0}{dx}\right)^{gradient*dx/dy}$$

(*dx* bezeichne $(x_1\text{-}x_0)$ und *dy* bezeichne $(y_1\text{-}y_0)$)

Diese Hilfsfunktion liefert für $x=x_0$ den Wert y_0 und für $x=x_1$ den Wert y_1. Die Steigung beträgt 0 bei x_0 und *gradient* bei x_1.

Aus dieser Hilfsfunktion werden dann, den Burrough-Varianten entsprechend, Zugehörigkeitsfunktionen mit *Maximum* (bei p_1 bis p_2), *steigend* (von p_0 bis p_1) bzw. *fallend* (von p_2 nach p_3) gebildet:

Seien $p_0,p_1,p_2,p_3 \in \mathbb{R}$, $p_0<p_1\leq p_2<p_3$, $p_{m1}:=(p_0+p_1)/2$, $p_{m2}:=(p_2+p_3)/2$, $gr_{m1}:=2/(p_1-p_0)$, $gr_{m2}:=2/(p_2-p_3)$.

$$transf_{Std\text{-}steigend,p0,p1}(p) := \begin{cases} 0 & p < p_0 \\ std(p_0, 0, p_{m1}, 0.5, gr_{m1}, p) & p_0 \leq p < p_{m1} \\ std(p_1, 1, p_{m1}, 0.5, gr_{m1}, p) & p_{m1} \leq p < p_1 \\ 1 & p_1 \leq p \end{cases}$$

$$transf_{Std\text{-}fallend,p2,p3}(p) := \begin{cases} 1 & p < p_2 \\ std(p_2, 1, p_{m2}, 0.5, gr_{m2}, p) & p_2 \leq p < p_{m2} \\ std(p_3, 0, p_{m2}, 0.5, gr_{m2}, p) & p_{m2} \leq p < p_3 \\ 0 & p_3 \leq p \end{cases}$$

$$transf_{Std\text{-}max,p0,p1,p2,p3}(p) := \begin{cases} transf_{Std-steigend,p_0,p_1}(p) & p \leq p_1 \\ 1 & p_1 < p \leq p_2 \\ transf_{Std-fallend,p_2,p_3}(p) & p_2 < p \end{cases}$$

Diese Funktionen haben bei p_0, p_1, p_2 und p_3 die Steigung 0; bei p_{m1} und ist die Steigung maximal (bei $p_1-p_0=1$ nämlich 2) und bei p_{m2} ist die Steigung minimal (bei $p_2-p_3=-1$ nämlich -2). Wenn man den Abstand von p_0 und p_1 (bzw. p_2 und p_3) vergrößert, wird die Funktion praktisch maßstäblich vergrößert (entlang der p-Achse). In Abbildung 2.2.1b ist ein Beispiel dargestellt.

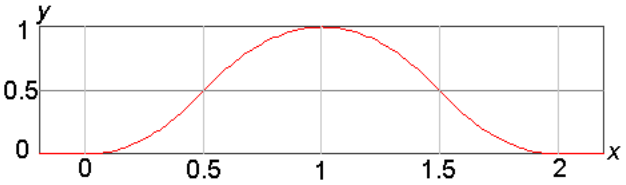


Abbildung 2.2.1b: $y=transf_{Std\text{-}max,0,1,1,2}(x)$; Die Grenzen zwischen den Fallunterscheidungen sind als vertikale graue Linien zu sehen.

Diese Funktionen sollen nun mit dem Erweiterungsprinzip zu Funktionen erweitert werden, die unscharfe Zahlen abbilden. Wie in Kapitel 2.1.4 beschrieben, reicht es bei Schnittdarstellung, wenn man sich überlegt, wie Schnitte - bei unscharfen Zahlen also Intervalle - abzubilden sind. Wie in Kapitel 2.1.4 außerdem erläutert, ist es als Vorbereitung zur Implementierung ausreichend, nur abgeschlossene Intervalle zu betrachten.

Satz 3 aus Kapitel 2.1.4 zeigt, wie man auf besonders einfache Art und Weise die Abbildung der abgeschlossenen Intervalle durchführen kann, wenn die Funktionen monoton und stetig sind. Stetig sind alle obigen Funktionen, aber damit man den Satz anwenden kann, muß man die Funktionen, die nicht monoton sind (die ein Maximum besitzen), in dem Fall, daß das Maximum in dem abzubildenden Intervall auftritt, zerlegen. Dazu also folgende Überlegung:

Sei f eine stetige Funktion, die ein Maximum bei v hat und auf beiden Seiten des Maximums monoton ist (links monoton steigend, rechts monoton fallend). Dann werden die abgeschlossenen Intervalle $[a,b]$ folgendermaßen abgebildet:

$$\bar{f}([a,b]) = \begin{cases} [f(a), f(b)] & \text{falls } b \le v \ (f \text{ monoton steigend}) \\ [\min(f(a), f(b)),\ f(v)] & \text{falls } a < v < b \text{ (Fall siehe unten)} \\ [f(b), f(a)] & \text{falls } v \le a \ (f \text{ monoton fallend}) \end{cases}$$

Der erste und dritte Fall ergeben sich aus Anwendung von Satz 3 aus Kapitel 2.1.4. Im Fall $a<v<b$ muß die Zerlegung stattfinden:

$$\begin{aligned} &\bar{f}([a,b]) \\ &=^{(1)} \{\, y \in \mathbb{R} \mid \exists x \in [a,b] : f(x)=y \,\} \\ &=^{(2)} \{\, y \in \mathbb{R} \mid \exists x \in [a,v] : f(x)=y \,\} \cup \{\, y \in \mathbb{R} \mid \exists x \in [v,b] : f(x)=y \,\} \\ &=^{(3)} [f(a), f(v)] \cup [f(b), f(v)] \\ &=^{(4)} [min(f(a),f(b)), f(v)] \end{aligned}$$

Erläuterungen:

(1) Laut Definition.

(2) $\exists x \in [a,b] : f(x)=y \ \Leftrightarrow\ (\exists x \in [a,v] : f(x)=y) \vee (\exists x \in [v,b] : f(x)=y)$.

(3) Satz 3 aus Kapitel 2.1.4.

(4) Falls $f(a)<f(b)$ gilt, dann $[f(b), f(v)] \subseteq [f(a), f(v)]$, also trägt $[f(b), f(v)]$ nichts zu der Vereinigung bei, sondern nur das größere Intervall mit der kleineren unteren Grenze (der umgekehrte Fall ist entsprechend).

Laut dieser Überlegung lassen sich die beiden Zugehörigkeitsfunktionen mit einem Maximum auch einfach behandeln (die Behandlung der anderen Funktionen ist ohnehin klar, da Satz 3 aus Kapitel 2.1.4 direkt anwendbar ist). Als Beispiel hier das Resultat der Behandlung von $transf_{Burrough\text{-}max\text{-}o,s,v}$ für Intervalle $[a,b]$:

$$\overline{transf_{Burrough-\max-o,s,v}}([a,b]) = \begin{cases} \left[\frac{1}{1+s(a-v)^2}, \frac{1}{1+s(b-v)^2}\right] & \text{falls } b \le v \\ \left[\min(\frac{1}{1+s(a-v)^2}, \frac{1}{1+s(b-v)^2}), 1\right] & \text{falls } a < v < b \\ \left[\frac{1}{1+s(b-v)^2}, \frac{1}{1+s(a-v)^2}\right] & \text{falls } v \le a \end{cases}$$

Die auf unscharfe Zahlen erweiterten obigen Funktionen bilden unscharfe Zahlen auf unscharfe Zahlen ab, wie in der Folgerung zu Satz 2 aus Kapitel 2.1.4 gezeigt (dort wurde nur die Stetigkeit der Funktionen vorausgesetzt).

Bemerkung: In [Dong/Shah 1987] wird auch eine allgemeinere Methode angegeben, die die Behandlung von Extrema erlaubt (Stetigkeit wird auch dort vorausgesetzt).

2.2.2 Arithmetische und logische Verknüpfungen (Aggregationsoperatoren)

Nachdem die Werte der Eingabeparameter nun auf die gemeinsame Skala der Zugehörigkeitswerte (aus dem Intervall [0,1]) transformiert wurden, sollen Aggregationsoperatoren zur Auswahl gestellt werden, die diese Werte wiederum in Zugehörigkeitswerte (aus dem Intervall [0,1]) abbilden. Der Anwender soll mit deren Hilfe in die Lage versetzt werden, eine Gesamt-Aggregationsfunktion aus diesen Aggregationsoperatoren zusammenzusetzen, die Zugehörigkeitswerte zu einer für ihn interessanten aggregierten Eigenschaft der Eingabeparameter liefert.

Die Problematik des Zusammensetzens zur Gesamt-Aggregationsfunktion aus den Aggregationsoperatoren wird erst in Kapitel 2.2.3 behandelt.

In diesem Kapitel hier geht es nur um die Definition der im Rahmen dieser Diplomarbeit betrachteten Aggregationsoperatoren und erneut um die Frage, wie sie zu unscharfen Operatoren (unscharfen Funktionen) erweitert werden können. Dazu sei jeweils $n \in \mathbb{N}$.

Zur Verfügung gestellt werden zwei logische und ein arithmetischer Operator:

- AND: $[0,1]^n \to [0,1]$; $\text{AND}(m_1,...,m_n) := \min(m_1,...,m_n)$
  AND ist die übliche Und-Verknüpfung für unscharfen Mengen; sie ist eine Erweiterung der gewöhnlichen Und-Verknüpfung, wenn man 'falsch' mit 0 und 'wahr' mit 1 identifiziert (also folgende Funktionalität für die gewöhnliche Und-Verknüpfung betrachtet: und: $\{0,1\}^n \to \{0,1\}$).
- OR: $[0,1]^n \to [0,1]$; $\text{OR}(m_1,...,m_n) := \max(m_1,...,m_n)$
  OR ist die übliche Oder-Verknüpfung für unscharfen Mengen; sie ist genauso wie bei AND eine Erweiterung der gewöhnlichen Oder-Verknüpfung.
- Seien $w_1,...,w_n \in \mathbb{R}$ mit $w_1,...,w_n \geq 0$ und $w_1+...+w_n=1$.
  $\text{SUM}_{w_1,...,w_n}: [0,1]^n \to [0,1]$; $\text{SUM}_{w_1,...,w_n}(m_1,...,m_n) := w_1 \cdot m_1+...+w_n \cdot m_n$
  SUM ist die gewichtete Summe (mit Gewichten $w_1,...,w_n$) und dient dazu, ausdrücken zu können, daß jeder Eingangswert einen seinem Gewicht entsprechenden Teil zum Resultat beiträgt (siehe auch [Burrough 1989]).

Alle vorgestellten Operatoren haben die Eigenschaften, stetig und monoton steigend für jedes Argument zu sein. Diese Voraussetzung ermöglicht die Anwendung von Satz 3 aus Kapitel 2.1.4: Um unscharfe Zahlen abbilden zu können, wird wiederum das Erweiterungsprinzip angewendet. Bei Betrachtung von unscharfen Zahlen in Schnittdarstellung soll wieder die Abbildung von Intervallen (hier nur abgeschlossenen, wie in Kapitel 2.1.4 begründet) angegeben werden (***ARI*** bezeichne die Menge alle abgeschlossenen reellen Intervalle):

- $\overline{\text{AND}}: \boldsymbol{ARI}^n \to \boldsymbol{ARI}$;
  $\overline{\text{AND}}([a_1,b_1],...,[a_n,b_n]) = [\min(a_1,...,a_n), \min(b_1,...,b_n)]$
- $\overline{\text{OR}}: \boldsymbol{ARI}^n \to \boldsymbol{ARI}$;
  $\overline{\text{OR}}([a_1,b_1],...,[a_n,b_n]) = [\max(a_1,...,a_n), \max(b_1,...,b_n)]$
- $\overline{\text{SUM}}_{w_1,...,w_n}: \boldsymbol{ARI}^n \to \boldsymbol{ARI}$;
  $\overline{\text{SUM}}_{w_1,...,w_n}([a_1,b_1],...,[a_n,b_n]) = [w_1 \cdot a_1+...+w_n \cdot a_n,\ w_1 \cdot b_1+...+w_n \cdot b_n]$

2.2.3 Bildung der Gesamtfunktion aus den Verknüpfungen, Kompositionsproblem

Wie schon am Anfang von Kapitel 2.2 erwähnt, ist die Erweiterung der Gesamtfunktion, die sich aus Transformationsfunktionen und Aggregationsoperatoren zusammensetzt, mit dem Erweiterungsprinzip nicht unproblematisch. Zunächst einmal soll hier das Problem, vor dem man steht, verdeutlicht werden.

Beispiel: Sei g: $\mathbb{R}\rightarrow\mathbb{R}$; $g(a):=a+(-a)$ mit dem Erweiterungsprinzip zu $\hat{g}:\boldsymbol{FN}\rightarrow\boldsymbol{FN}$ zu erweitern (***FN*** wurde in Kapitel 2.1.1 als die Menge aller unscharfen Zahlen definiert). Dann werden abgeschlossene Intervalle folgendermaßen abgebildet:

$\bar{g}([a_l,a_r]) = \{y\in\mathbb{R} \mid \exists\, a\in[a_l,a_r]: a+(-a)=y\} = \{y\in\mathbb{R} \mid \exists\, a\in[a_l,a_r]: 0=y\} = [0,0]$

Wenn man die zwei Funktionen + (2 reelle Argumente) und - (1 reelles Argument) zu unscharfen Funktionen erweitert, erhält man mit Hilfe der Sätze 2 und 3 aus Kapitel 2.1.4 für abgeschlossene Intervalle:

$[a_l,a_r]+[b_l,b_r]=[a_l+b_l,\ a_r+b_r]$ und $-[a_l,a_r]=[-a_r,-a_l]$.

Diese beiden unscharfen Funktionen durch Einsetzung zusammenzusetzen, ergibt etwas ganz anderes als $\bar{g}$, nämlich $\bar{f}$, für abgeschlossene Intervalle also:

$\bar{f}([a_l,a_r])=[a_l-a_r,\ a_r-a_l]$.

Wie man am Beispiel sieht, kann man nicht einfach Operatoren zu unscharfen Funktionen erweitern und diese dann durch Einsetzung zusammensetzen, um die Funktion zu erhalten, die sich aus Anwendung des Erweiterungsprinzips auf die zusammengesetzte scharfe Funktion ergibt.

Das Problem besteht darin, daß man diese Art von Einsetzung nicht benutzen kann, da das scharfe + als Funktion $+:\mathbb{R}^2\rightarrow\mathbb{R}$ mit dem Erweiterungsprinzip zum unscharfen + erweitert wurde, also nicht als Funktion $+:M\rightarrow\mathbb{R}$, wobei M eine Teilmenge von $\mathbb{R}^2$ ist. Bei der Einsetzung ergibt sich nämlich, daß gar nicht mehr alle Werte aus $\mathbb{R}^2$ vorkommen *können*, sondern nur die aus $M=\{(a,b)\in\mathbb{R}^2 \mid a=-b\}$ sind möglich.

Die Abhängigkeiten zwischen den Variablen, die in eine Funktion eingesetzt werden, ergeben also das Problem. Diese Abhängigkeit wurde oben mit der Menge M charakterisiert, mit der man den Grundbereich (gegenüber $\mathbb{R}^2$) einschränken müßte.

Man kann ein, um einschränkende Grundbereiche erweitertes, alternatives Erweiterungsprinzip definieren, um mit diesen Problemen weiter umzugehen. Dies wird hier aber nicht erforderlich, da bei den Funktionen aus Kapitel 2.2.2 mit Hilfe ihrer Eigenschaften und der Schnitt-Darstellung der unscharfen Zahlen auch auf relativ einfache Art gezeigt werden kann, daß die Einsetzung von den unscharfen Funktionen ineinander hier *nicht* zu Problemen führt.

Eine weitere Methode, mit diesen Problemen umzugehen, wird in [Dubois/Prade 1987] angesprochen: Bei obigem Beispiel ist es möglich, die Funktion zunächst umzuformen bis jede Variable höchstens einmal vorkommt ($a+(-a)=0$), und dann die geeignete Repräsentation (also hier 0) operatorweise zu erweitern (bei dem Term 0 ist hier gar nichts mehr zu tun). Bei der reellen Funktion $g(a,b,c):=ab+ac$ z.B. führt diese Vorgehensweise zum Erfolg (Ausnutzung der Distributivität im Reellen liefert $ab+ac=a(b+c)$). Es ist allerdings nicht für jede Funktion möglich, durch Umformung eine Repräsentation zu erhalten, bei der obige Probleme nicht auftreten (etwa die reelle Funktion $g(a,b,c):=ab+bc+ca$ läßt sich nicht so umformen, daß jede Variable nur einmal vorkommt und nur die Operatoren der Grundrechenarten vorkommen).

Ziel soll im folgenden sein, zu zeigen, daß bei den Funktionen aus Kapitel 2.2.1 und 2.2.2 *keine* Probleme bei einfachem Ineinandersetzen der auf Intervalle erweiterten Funktionen auftreten. Insbesondere wird auch (wieder) ausgenutzt werden, daß die Operatoren stetig und monoton steigend sind.

Die folgenden Sätze charakterisieren Mengen von Funktionen, bei denen keine Probleme beim einfachen Ineinandereinsetzen auftauchen. Satz 1 zeigt, daß keine Probleme auftauchen, wenn jede Variable nur einmal vorkommt, und wird für die Transformationsfunktionen benötigt, was aber erst zum Schluß des Kapitels ausgenutzt wird. Die Folgerung aus dem folgenden Satz 2 kann entsprechend auf die Komposition der Aggregationsoperatoren angewendet werden.

Satz 1: Sei $k\in\mathbb{N}_0$. Seien $f_i:X\to Y$ für $i\in\{1...k\}$ und $g:Y^k\to Z$ beliebige Funktionen. Sei $h:X^k\to Z$ folgendermaßen für alle $x_1,...,x_k\in X$ definiert:
$h(x_1,...,x_k) := g(f_1(x_1),...,f_k(x_k))$.
Dann gilt $\bar{g}(\bar{f_1}(I_1),...,\bar{f_n}(I_k)) = \bar{h}(I_1,...,I_k)$ für alle $I_1,...,I_k\subseteq X$.
(d.h. wenn jede Variable nur einmal vorkommt, kann man $h:X^k\to Z$ zur Komplexfunktion erweitern, indem man die Teilfunktionen $g,f_1,...,f_k$ zunächst zur Komplexfunktion erweitert und sie dann ineinander einsetzt).

Beweis: Die Voraussetzungen aus dem Satz seien gültig. Dann folgt für $I_1,...,I_k\subseteq X$ und $k>0$ (für $k=0$ ist die Aussage trivial):

$$\begin{aligned}
&\bar{g}(\bar{f_1}(I_1),...,\bar{f_k}(I_k))\\
&=^{(1)} \{z\in Z \mid \exists y_i \in \bar{f_i}(I_i) \text{ für alle } i\in\{1,...,k\} : g(y_1,...,y_k)=z\}\\
&=^{(2)} \{z\in Z \mid \exists y_i \in \{w_i\in Y \mid \exists x_i\in I_i : f_i(x_i)=w_i\} \text{ f.a. } i\in\{1,...,k\} : g(y_1,...,y_k)=z\}\\
&=^{(3)} \{z\in Z \mid \exists y_i \in Y \text{ mit } (\exists x_i\in I_i : f_i(x_i)=y_i) \text{ f.a. } i\in\{1,...,k\} : g(y_1,...,y_k)=z\}\\
&=^{(4)} \{z\in Z \mid \exists y_i \in Y \text{ und } x_i\in I_i \text{ mit } f_i(x_i)=y_i \text{ f.a. } i\in\{1,...,k\} : g(y_1,...,y_k)=z\}\\
&=^{(5)} \{z\in Z \mid \exists x_i \in I_i \text{ f.a. } i\in\{1,...,k\} : g(f_1(x_1),...,f_k(x_k))=z\}\\
&=^{(6)} \{z\in Z \mid \exists x_i \in I_i \text{ f.a. } i\in\{1,...,k\} : h(x_1,...,x_k)=z\}\\
&=^{(7)} \bar{h}(I_1,...,I_k).
\end{aligned}$$

Bemerkungen dazu:

(1), (2) und (7) Definition der Komplexfunktion.

(3) und (4) Andere Schreibweise zur Vorbereitung von (5).

(5) Da $y_j\in Y$ überflüssig ist (weil $f_i:X\to Y$), kann $f_i(x_i)$ für y_i eingesetzt werden.

(6) Definition von h. □

Satz 2: Seien $k,n \in \mathbb{N}_0$ und $f_i:\mathbb{R}^n \to \mathbb{R}$ für $i=1...k$ und $g:\mathbb{R}^k \to \mathbb{R}$ stetige und monoton steigende Funktionen. Ferner sei für reelle $x_1,...,x_n$ die Funktion $h:\mathbb{R}^n \to \mathbb{R}$ folgendermaßen definiert: $h(x_1,...,x_n) := g(f_1(x_1,...,x_n), ... , f_k(x_1,...,x_n))$.
Für Intervalle $I_1,...,I_n$ gilt dann: $\bar{g}(\bar{f}_1(I_1, ..., I_n), ..., \bar{f}_k(I_1, ..., I_n)) = \bar{h}(I_1, ..., I_n)$.

Beweis: Die Voraussetzungen aus dem Satz seien gültig. Dann folgt für Intervalle $I_1,...,I_k$ und $k,n>0$ (für $k=0$ bzw. $n=0$ ist die Aussage jeweils trivial):

$\bar{g}(\bar{f}_1(I_1, ..., I_n), ..., \bar{f}_k(I_1, ..., I_n))$

$=^{(1)}$ $\{z \in \mathbb{R} \mid \exists y_j \in \bar{f}_j(I_1, ..., I_n)$ für alle $j \in \{1, ..., k\} : g(y_1, ..., y_k) = z\}$

$=^{(2)}$ $\{z \in \mathbb{R} \mid \exists y_j \in \{w_j \in \mathbb{R} \mid \exists x_{ji} \in I_i$ f.a. $i \in \{1,...,n\}: f_j(x_{j1},...,x_{jn})=w_j\}$ f.a. $j \in \{1,...,k\}:$ $g(y_1,...,y_k) = z\}$

$=^{(3)}$ $\{z \in \mathbb{R} \mid \exists y_j \in \mathbb{R}$ mit $(\exists x_{ji} \in I_i$ f.a. $i \in \{1,...,n\}: f_j(x_{j1},...,x_{jn})=y_j)$ f.a. $j \in \{1,...,k\}:$ $g(y_1,...,y_k) = z\}$

$=^{(4)}$ $\{z \in \mathbb{R} \mid \exists y_j \in \mathbb{R}$ und $(x_{ji} \in I_i$ f.a. $i \in \{1,...,n\})$ mit $f_j(x_{j1},...,x_{jn})=y_j$ f.a. $j \in \{1,...,k\}:$ $g(y_1,...,y_k) = z\}$

$=^{(5)}$ $\{z \in \mathbb{R} \mid \exists x_{1i},...,x_{ki} \in I_i$ f.a. $i \in \{1,...,n\}: g(f_1(x_{11},...,x_{1n}),...,f_k(x_{k1},...,x_{kn})) = z\}$

$=^{(6)}$ $\{z \in \mathbb{R} \mid \exists \; x_i \in I_i$ f.a. $i \in \{1,...,n\}: g(f_1(x_1, ..., x_n),...,f_k(x_1, ..., x_n)) = z\}$

$=^{(7)}$ $\{z \in \mathbb{R} \mid \exists \; x_i \in I_i$ f.a. $i \in \{1,...,n\}: h(x_1, ..., x_n) = z\}$

$=^{(8)}$ $\bar{h}(I_1, ..., I_n)$

Bemerkungen dazu:

(1), (2) und (8) Definition der Komplexfunktion.

(3) und (4) Andere Schreibweise zur Vorbereitung von (5).

(5) Da $y_j \in \mathbb{R}$ überflüssig ist (weil $f_j:\mathbb{R}^n \to \mathbb{R}$), kann $f_j(x_{j1},...,x_{jn})$ für y_j eingesetzt werden.

(7) Definition von h.

(6) stellt das eigentliche Problem dar. Im folgenden sei die linke Seite mit J_l und die rechte Seite mit J_r bezeichnet. Daß $J_l \supseteq J_r$ gilt, ist trivial (denn zu $z \in J_r$ existieren $x_i \in I_i$ f.a. $i \in \{1,...,n\}$ mit $g(f_1(x_1,...,x_n),...,f_k(x_1,...,x_n))=z$; wähle $x_{1i}=...=x_{ki}=x_i \in I_i$ f.a. $i \in \{1,...,n\}$ dann gilt also $g(f_1(x_{11},...,x_{1n}),...,f_k(x_{k1},...,x_{kn}))=z$; somit ist $z \in J_l$).

Zu zeigen ist also $J_l \subseteq J_r$.

Sei also $z \in J_l$. Dazu existieren $x_{1i},...,x_{ki} \in I_i$ f.a. $i \in \{1,...,n\}$ mit $g(f_1(x_{11},...,x_{1n}),...,f_k(x_{k1},...,x_{kn}))=z$.

Wähle $x_i=\min(x_{1i},...,x_{ki}) \in I_i$ f.a. $i \in \{1,...,n\}$. Dies zeigt, daß $z_{\min} := g(f_1(x_1,...,x_n),...,f_k(x_1,...,x_n))$ in J_r liegt.

Wähle zusätzlich $v_i=\max(x_{1i},...,x_{ki}) \in I_i$ f.a. $i \in \{1,...,n\}$. Dies zeigt, daß $z_{\max} := g(f_1(v_1,...,v_n),...,f_k(v_1,...,v_n))$ in J_r liegt.

Da laut Satz 2 aus Kapitel 2.1.4 (h ist als Komposition von **stetig**en Funktionen stetig) J_r ein Intervall ist und sowohl $z_{\min}$ als auch $z_{\max}$ darin liegen, enthält J_r auch alle Elemente aus $[z_{\min},z_{\max}]$. Es genügt also zu zeigen, daß $z \in [z_{\min},z_{\max}]$. Aufgrund der Eigenschaft von $g(f_1,...,f_k)$ als Funktion von $k*n$ Argumenten monoton steigend zu sein (da alle Einzelfunktionen **monoton steigend** sind) gilt $z \geq z_{\min}$ (wegen $x_i=\min(x_{1i},...,x_{ki})$) und $z \leq z_{\max}$ (wegen $v_i=\max(x_{1i},...,x_{ki})$). □

Folgerung (aus Satz 2): Sei $n \in \mathbb{N}_0$. Sei eine Gesamtaggregationsfunktion $G:\mathbb{R}^n \rightarrow \mathbb{R}$ aus stetigen und monoton steigenden Funktionen (z.B. den obigen Operatoren MIN, MAX und $\text{SUM}_{w_1,...,w_m}$) zusammengesetzt und enthalte die Variablen $x_1,...,x_n \in \mathbb{R}$. PR_i seien die i-ten Projektionen; sie sind auch stetig und monoton steigend. Sei G' die Funktion, die aus G ensteht durch Einsetzung von $\text{PR}_1(x_1,...,x_n),...,\text{PR}_n(x_1,...,x_n)$ für $x_1,...,x_n$. Offensichtlich gilt $G'(x_1,...,x_n)=G(x_1,...,x_n)$ für alle $x_1,...,x_n \in \mathbb{R}$.
Dann gilt: Wenn man die Operatoren zu Operatoren auf Intervallen erweitert und diese erweiterten Operatoren laut G' ineinander einsetzt, dann ergibt sich dasselbe Resultat wie bei Erweiterung von G' (und somit auch von G) auf Intervalle.

Zum Nachweis der Folgerung nehme man an, daß dieVoraussetzungen aus dem Satz gültig seien. Man schreibe zunächst $G'(x_1,...,x_n)$ in der Form $g(f_1(x_1,...,x_n),...,f_k(x_1,...,x_n))$; g muß dabei eine der oben erwähnten stetigen und monoton steigenden Funktionen sein.
Bemerkung: Falls G die Identität ist (also $G(x_1):=x_1$), hat G' die Form $id(\text{PR}_1(x_1))$ und $g=id$); falls G konstant ist, betrachte man g als konstante Funktion ohne Argumente ($k=0$)).
Dann setze man auf Intervalle erweiterte Funktionen $\overline{f_1}, ..., \overline{f_k}$ in die Erweiterung des Operators g auf Intervalle (also $\overline{g}$) ein. Laut Satz 2 erhält man eine Funktion, die Intervalle genauso abbildet, wie die Funktion, die man erhält, wenn man G' zur Komplexfunktion erweitert.
Zum Schluß wende man dasselbe Verfahren rekursiv auf die $\overline{f_1}, ..., \overline{f_k}$ (jeweils anstelle $\overline{G'}$) an, sofern sie nicht schon Projektionsoperatoren oder Konstanten (Funktionen mit 0 Parametern) sind.
Damit ergibt sich insgesamt eine Erweiterung von G' auf Intervalle, die ausschließlich aus der Einsetzung von auf Intervalle erweiterte Funktionen ineinander (laut G') zustandekommt.

Mit der Folgerung aus Satz 2 läßt sich die Erweiterung von der Gesamtaggregationsfunktion (ohne die Transformationsfunktionen aus Kapitel 2.2.1) auf Intervalle nun sehr einfach durchführen. Anstelle der Parameter dieser erweiterten Gesamtaggregationsfunktion setze man nun die auf Intervalle erweiterten Transformationsfunktionen aus Kapitel 2.2.1 ein. Diese Einsetzung ergibt eine Erweiterung von obiger Gesamtaggregationsfunktion auf Intervalle, weil jeder Parameter der Transformationsfunktionen genau einmal vorkommt und damit Satz 1 anwendbar ist. (Die Anwendung von Satz 1 wäre nicht möglich, wenn man zu jedem Eingabeparameter mehrere Zugehörigkeitsfunktionen festlegen könnte.)

Insgesamt läßt sich die Erweiterung von der Gesamtaggregationsfunktion inklusive der Transformationsfunktionen aus Kapitel 2.2.1 auf abgeschlossene Intervalle also sehr einfach (nämlich durch Ineinander-Einsetzen der auf Intervalle erweiterten Funktionen und Operatoren aus Kapitel 2.2.1 und 2.2.2) durchführen.

Bemerkung 1: In Kapitel 2.1.4 wurde gezeigt, daß die Schnitte von unscharfen Zahlen (nämlich Intervalle) eine zur Zugehörigkeitsfunktion äquivalente Darstellung von unscharfen Zahlen ergeben. Daher gelten die Sätze dieses Kapitels auch für unscharfe Zahlen.

Bemerkung 2: Obige Sätze sind nicht in gleicher Art und Weise für monoton fallende Funktionen konstruierbar. Falls man auf die Möglichkeit Wert legt, auch Gesamtfunktionen, die stetig sind und nur kein Extremum haben, zu unscharfen Funktionen korrekt zu erweitern, müßte man eine gänzlich andere Methode anwenden, um die unscharfe Gesamtfunktion zu berechnen. Dies wäre mit der in [Dong/Shah 1987] beschriebenen Methode möglich; jedoch besteht in diesem Fall für den Anwender nicht mehr die Möglichkeit, die Zwischenergebnisse (sie Resultate der Teilterme) zu betrachten (weil das Resultat der Gesamtfunktion nicht aus solchen Zwischenergebnissen konstruiert werden kann). Die Zwischenergebnisse betrachten zu können, ist aber bei der Festlegung einer Gesamtaggregationsfunktion sehr nützlich (dies wird bei Anwendbarkeit obiger Sätze möglich und wurde bei dieser Diplomarbeit auch implementiert).

Bemerkung 3: Im Beweis von Satz 2 wird für $\overline{g}(\overline{f}_1(I_1,...,I_n),...,\overline{f}_k(I_1,...,I_n)) \supseteq \overline{h}(I_1,...,I_n)$ weder Stetigkeit noch Monotonie benötigt. Wenn man die Methode zum Nachweis der Folgerung aus Satz 2 entsprechend auf dieses Ergebnis anwendet, ergibt sich, daß durch eigentlich unerlaubte Einsetzung (wie im Beispiel durchgeführt) die Intervalle nur größer werden können, in dem Sinne, daß sie immer die Intervalle enthalten, die aus der direkten Anwendung des Erweiterungsprinzips resultieren würden (anschaulich betrachtet nimmt die Unschärfe zu).

Bemerkung 4: Wenn jede Variable nur einmal vorkommt, ergibt Satz 1 zusammen mit der Methode zum Nachweis der Folgerung aus Satz 2, daß einfaches Einsetzen der auf Intervalle erweiterten Funktionen ineinander dasselbe ergibt wie die direkte Anwendung des Erweiterungsprinzips auf die Gesamtfunktion.

2.3 Fuzzy Kriging

In Kapitel 1.4 wurde ein Überblick über die Vorgehensweise bei Kriging-Verfahren zur räumlichen Interpolation und die verschiedenen Möglichkeiten, unscharfe Zahlen miteinzubeziehen, gegeben.

In diesem Kapitel wird konventionelles (scharfes) Kriging detaillierter dargestellt; dies findet in Kapitel 2.3.1 statt. In Kapitel 2.3.2 wird dann Fuzzy Kriging (vom Typ 1) anhand der Unterschiede zum konventionellen Kriging dargestellt. Wichtige Eigenschaften dieser Verfahren werden jeweils in einem eigenen Unterkapitel behandelt.

2.3.1 Konventionelles, scharfes Kriging

Das in diesem Kapitel beschriebene Verfahren kann man in Geostatistik-Büchern und anderen Geostatistik-Quellen finden, unter anderem auch in [Heinrich 1992], [Zirschky 1984], [Akin/Siemes 1988], [Cressie].

Wie in Kapitel 1.4 dargestellt, dienen Kriging-Verfahren zur räumlichen Interpolation und beruhen auf einer statistischen Analyse der Eingabedaten in Form von Variogrammen. In Kapitel 2.3.1.1 werden Variogramme definiert und der Umgang mit ihnen wird beschrieben.

Kapitel 2.3.1.2 behandelt dann ein konventionelles, häufig benutztes Kriging-Verfahren das Normale Kriging (Ordinary Kriging). Kapitel 2.3.1.3 geht dann auf Eigenschaften dieses Verfahrens ein.

Kapitel 2.3.1.4 schließlich geht auf eine mögliche, häufig benötigte Transformation (die logarithmische) ein, die man benutzen kann, damit sich die Eingabedaten für den Krigingprozeß besser eignen.

2.3.1.1 Variogramme

Variogramme sind ein in der Geostatistik übliches Mittel zur statistischen Analyse räumlicher Daten (von regionalisierten Variablen) und können zur Beschreibung von Eigenschaften eines Parameters benutzt werden, was für Kriging-Verfahren wichtig ist. Sie beschreiben die Abhängigkeit zwischen den Werten eines Parameters in Abhängigkeit von der räumlichen Anordnung.

Das Variogramm γ zu einem Parameter p (auch Semivariogramm genannt) ist für Punkte x_1 und x_2 folgendermaßen definert (VAR(X) ist die Bezeichnung der Varianz von X):

$$\gamma(x_1,x_2) := \mathrm{VAR}(p(x_1) - p(x_2)) / 2$$

Zur statistischen Analyse der Eingabedaten zum Parameter *p* dient das sog. experimentelle Variogramm γ^*. Seien $x_1,...,x_n$ Punkte, bei denen der Wert des Parameters *p* bekannt ist. Um für diese Eingabedaten das experimentelle Variogramm zu berechnen, werden Paare von Koordinaten (zu jeweils zwei bekannten Werten des Parameters) zunächst in Klassen eingeteilt. Prinzipiell teilt man in Klassen von Punkten mit gleichem Differenzvektor (h) ein, was zu folgender Formulierung führt ($N(h)$ sei die Anzahl der Paare in solch einer Klasse):

$$\gamma^*(h) := \frac{1}{2N(h)} \sum_{\substack{\text{Eingabekoordinaten } x_j,x_i \text{ mit} \\ x_j - x_i = h}} (p(x_i) - p(x_i + h))^2$$

Tatsächlich gibt es i.a. zu wenige Eingabepunkte, um tatsächlich diese Klasseneinteilungen zu benutzen. Deshalb faßt man diese Klassen nochmals zu größeren Klassen zusammen, und zwar faßt man Abstandsvektoren zusammen, die gleich lang (oder in etwa gleich lang) sind und deren Richtung (z.B. in Grad im 2-dimensionalen Fall) in einem bestimmten Bereich (im Intervall [a,b] im 2-dimensionalen Fall) liegt. Die Festlegung dieser Bereiche bzw. Intervalle kann i.a. der Anwender beeinflussen oder man läßt die Richtung ohne Beachtung (dann ist γ^* nur noch eine partielle Funktion von *Abständen*, also reellen Zahlen). In Bezug auf die Abstände teilt man i.a. in äquidistante (und gleichgroße) Klassen ein, um in etwa gleich lange Vektoren zusammenzufassen. Für den Fall, daß man nur die Abstände betrachtet und auch die Abstände in Klassen eingeteilt hat, ist ein Beispiel in Abbildung 2.3.1.1a zu sehen.

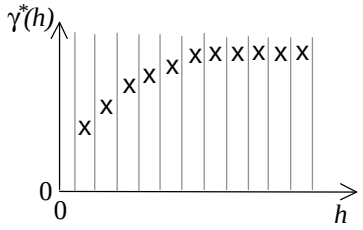


Abbildung 2.3.1.1a: Beispiel eines experimentellen Variogramms
(die vertikalen Linien zeigen die Klasseneinteilung)

Das experimentelle Variogramm liefert im allgemeinen nicht ausreichend Informationen, um damit Kriging-Verfahren durchzuführen, denn die Klassen, in denen (ausreichend) Werte vorkommen, decken normalerweise nicht alle Bereiche ab, in denen man eine Schätzung benötigt. Insbesondere fehlen meist Variogrammwerte für sehr kleine Abstandsvektoren (bzw. Abstände) *h*, denn man verfolgt meist in erster Linie das Ziel, das Proberaster so regelmäßig wie möglich zu machen, was natürlich dazu führt, daß unterhalb eines bestimmten Abstandes gar keine Wertepaare mehr auftauchen.

Um trotzdem Kriging-Verfahren durchführen zu können, wird das sog. theoretische Variogramm vom Anwender festgelegt; es ist eine total definierte Funktion γ, die von der Menge der Abstandsvektoren aus $\mathbb{R}^k$ (bzw. der Menge der Abstände aus $\mathbb{R}_0^+$) nach $\mathbb{R}_0^+$ abbildet und bietet damit alle für das Kriging notwendigen Informationen. Das theoretische Variogramm kann auch (wie in der Definition vom Variogramm) als Funktion von zwei Parametern aufgefaßt werden (mit folgender Festlegung für $x_1,x_2 \in \mathbb{R}^k$: $\gamma(x_1,x_2) := \gamma(x_1-x_2)$).

Zusätzliches Expertenwissen kann bei der Anpassung des theoretischen Variogramms an das experimentelle Variogramm helfen, um es ausreichend realitätsnah festzulegen. Es gibt allerdings auch Methoden, die dabei helfen können, die Qualität eines theoretischen Variogramms abzuschätzen. Dazu zählt z.B. die Kreuzvalidierung; bei ihr wird jeweils einer der Eingabepunkte weggelassen, an derselben Stelle ein Wert mit Hilfe des Krigings geschätzt und dann mit dem weggelassenen Wert verglichen; die Differenz dieser Werte wird als *Fehler der Schätzung* bezeichnet (i.a. wird die Summe der Quadrate dieser Fehler als Gütekriterium der Schätzung benutzt [Heinrich 1992]). Zusätzlich kann ein Experte möglicherweise beurteilen, wie realistisch Kriging-Schätzungen sind. Bei unscharfen Eingabedaten kann man auch das experimentelle Variogramm unscharf berechnen und die Unschärfe als zusätzliche Entscheidungshilfe benutzen; dazu aber in Kapitel 2.3.2.1 Genaueres.

Um ein theoretisches Variogramm festzulegen, bieten Programme i.a. Standardfunktionen zur Auswahl an, die über zusätzliche Parameter verfügen und meist ist möglich, eine Linearkombination aus diesen Standardfunktionen zu benutzen. (Für eine Auswahl häufig benutzter Standardfunktionen siehe den Hilfetext zu FUZZEKS.)

Um die zusätzlichen Parameter der Standardfunktionen möglichst anschaulich festzulegen, benutzt man für die Anwendersicht vorrangig die üblichen Kenngrößen solcher Variogramme:

- Schwellenwert (sill): Das Supremum der Funktionswerte, falls es existiert.
- Aussageweite (range): Die Entfernung, ab der das Supremum angenommen wird, bzw. ab der 95% des Schwellenwertes erreicht wird, falls es nicht angenommen wird.
- Nuggeteffekt (nugget-effect): $\gamma(0)$. Ein hoher Nuggeteffekt (relativ zum Schwellenwert, etwa ab 50%) zeugt von einer geringen (oder bei sehr hohem Nuggeteffekt sogar gar nicht für die Anwendung von Kriging-Verfahren ausreichenden) Korrelation in der Stichprobe (bei den Eingabedaten). Ein hoher Nuggeteffekt kann auf Meßfehler oder hohe Mikrovariabilität, die unterhalb des Probenabstandes liegt, hindeuten ([Heinrich 1992]); entweder sollte durch zusätzliche Messungen die Stichprobe verbessert werden (falls der Probenabstand zu groß war) oder es sollten Meßfehler als Ursache ermittelt werden oder Kriging-Methoden können gar nicht eingesetzt werden (falls es bei dem Parameter keine oder nur sehr geringe räumliche Abhängigkeiten gibt).

In Abbildung 2.3.1.1b wird ein Beispiel einer theoretischen Variogrammkurve gezeigt, in der diese drei Größen eingetragen sind.

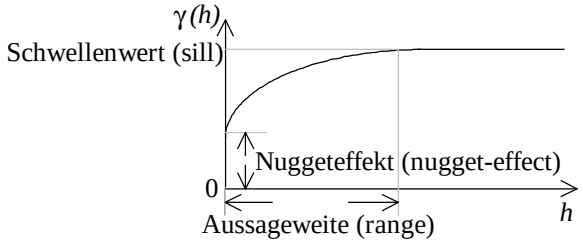


Abbildung 2.3.1.1b: Beispiel für theoretische Variogrammkurve

2.3.1.2 Normales Kriging (Ordinary Kriging)

Für Kriging-Verfahren zur räumlichen Interpolation benötigt man neben dem theoretischen Variogramm γ noch zusätzliche Modellannahmen. In diesem Kapitel werden zunächst die Modellannahmen für normales Kriging dargestellt und dann wird beschrieben, wie man daraus Schätzungen für den Parameter an beliebigen Stellen ableitet.

Zunächst einmal macht man folgende Voraussetzungen in Bezug auf den Parameter:

(V1) $E(p(x)) = m$ für alle Punkte x, d.h. der Erwartungswert von $p(x)$ hängt nicht von x ab.

(V2) $VAR(p(x+h)-p(x)) = 2\gamma(h)$, d.h. die Differenz $p(x+h)-p(x)$ hat endliche Varianz und die Varianz hängt nicht von x ab.

Außerdem legt man eine Formel fest, die beschreibt, wie sich aus den bekannten Werten die Schätzwerte ergeben, und man definiert Eigenschaften, die diese Schätzungen aufweisen sollen.

Die Hauptkriginggleichung legt fest, daß sich der Schätzwert $p^*(x)$ für die Stelle x aus einer Linearkombination der bekannten Werte ergibt (die $\delta_i(x)$ werden später mit Hilfe aller Modellannahmen bestimmt):

$$p^*(x) = \sum_{i=1}^{n} \delta_i(x) * p(x_i)$$

Folgende Eigenschaften sollen Schätzungen aufweisen:

(E1) $E(p(x) - p^*(x)) = 0$, d.h. im Durchschnitt weichen die Schätzungen nicht vom wahren Wert ab.

(E2) $\sigma^2(x) := E((p(x) - p^*(x))^2)$, also die Schätzvarianz oder Kriging-Varianz, ist minimal.

Aus den Voraussetzungen und Modellannahmen ergeben sich nun folgende Gleichungen:

Aus der Hauptkriginggleichung, zusammen mit (E1) und (V1) ergibt sich

(G1) $\sum_{i=1}^{n} \delta_i(x) = 1.$

Wiederum aus der Hauptkriginggleichung, in diesem Fall zusammen mit (E2) und (V2) ergibt sich für die Schätzvarianz

(G2) $\sigma(x)^2 = 2\sum_{i=1}^{n} \delta_i(x)\gamma(x, x_i) - \gamma(x, x) - \sum_{j=1}^{n}\sum_{i=1}^{n} \delta_j(x)\delta_i(x)\gamma(x_j, x_i).$

Laut (E2) soll die Schätzvarianz minimal sein. Das ist genau für jene $\delta_1(x), ..., \delta_n(x)$ der Fall, bei denen die n Ableitungen der Schätzvarianzformel (G2) (nach $\delta_k(x)$ für $k \in \{1,...,n\}$) null sind. Da zusätzlich noch die Bedingung (G1) eingehalten werden soll, hat man somit (n+1) Gleichungen und n Unbekannte. Unter Zuhilfenahme des Lagrange-Prinzips erhält man ein lineares Gleichungssystem mit (n+1) Gleichungen und (n+1) Unbekannten (wegen der zusätzlichen Unbekannten, nämlich dem Lagrange-Faktor $\lambda(x)$):

$$\left(\sum_{j=1}^{n} \delta_j(x)\gamma(x_j, x_i)\right) + \lambda(x) = \gamma(x, x_i) \quad \text{für } i \in \{1,...,n\} \quad \text{und} \quad \sum_{i=1}^{n} \delta_i(x) = 1.$$

In Matritzenschreibweise lautet dieses Gleichungssystem

$$\begin{pmatrix} 0 & 1 & \cdots & 1 \\ 1 & \gamma(x_1, x_1) & \cdots & \gamma(x_1, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \gamma(x_n, x_1) & \cdots & \gamma(x_n, x_n) \end{pmatrix} \begin{pmatrix} \lambda(x) \\ \delta_1(x) \\ \vdots \\ \delta_n(x) \end{pmatrix} = \begin{pmatrix} 1 \\ \gamma(x, x_1) \\ \vdots \\ \gamma(x, x_n) \end{pmatrix}.$$

Dieses Gleichungssystem kann man nun mit den üblichen Mitteln (z.B. [Stoer 1983]) lösen, sofern das Gleichungssystem eine eindeutige Lösung hat (bzw. die Matrix invertierbar ist). Daß das der Fall ist, kann man schon mit den Auswahlmöglichkeiten für die theoretischen Variogramme sicherstellen (man benutzt i.a. nur Variogrammtypen, bei denen das Gleichungssystem immer eindeutig lösbar ist).

Bemerkung (Einfaches Kriging (Simple Kriging)): Bei bekanntem Mittelwert (des Parameters) m kann auch eine leicht modifizierte Variante des normalen Krigings eingesetzt werden, nämlich das einfache Kriging (siehe auch [Akin/Siemes 1988], [Cressie]): Hierbei wird die Bedingung (G1) weggelassen und die Hauptkriginggleichung abgeändert zu

$$p^*(x) = \left(\sum_{i=1}^{n} \delta_i(x) * p(x_i)\right) + (1 - \sum_{i=1}^{n} \delta_i(x)) * m.$$

Der Term $(1 - \sum_{i=1}^{n} \delta_i(x))$ ist gewissermaßen der Koeffizient zu einem virtuellen Meßpunkt mit dem Wert m (ohne Koordinatenangabe) und fängt das Wegfallen der Bedingung (G1) auf (denn es gilt $\left(\sum_{i=1}^{n} \delta_i(x)\right) + (1 - \sum_{i=1}^{n} \delta_i(x)) = 1$). Bei diesem Verfahren entsteht dann nur ein Gleichungssystem mit n Gleichungen und n Unbekannten.

Schreibweise: Unter Zuhilfenahme der Bezeichnungen aus diesem Kapitel soll mit $\bar{p}:=(p(x_1), ..., p(x_n))$ und $\bar{x}:=(x_1, ..., x_n)$ die Kriging-Funktion (zu normalem Kriging) im folgenden mit Hilfe der Definition $K(\gamma, \bar{x}, \bar{p})(x) := p^*(x)$ ($p^*(x)$ siehe Hauptkriginggleichung weiter oben) geschrieben werden in der Form

$$K(\gamma, \bar{x}, \bar{p})$$

und mit Hilfe der Definition $V(\gamma, \bar{x}, \bar{p})(x) := \sigma(x)^2$ soll die Kriging-Varianz geschrieben werden in der Form

$$V(\gamma, \bar{x}, \bar{p}).$$

2.3.1.3 Eigenschaften von Normalem Kriging

Unter Zuhilfenahme der Schreibweisen für die Kriging-Funktion und die Kriging-Varianz am Ende vom vorigen Kapitel lassen sich die Eigenschaften übersichtlich formulieren. Außerdem sei $\overline{q}$ analog $\overline{p}$ definiert, nur für eine andere regionalisierte Variable q anstelle p (unter der Annahme, daß dasselbe theoretische Variogramm, das p beschreibt, auch q beschreibt und Werte für den Parameter q an denselben Stellen $x_1, ..., x_n$ bekannt sind wie p).

Eine ganz elementare Eigenschaft ist, daß die Kriging-Funktion K für die Punkte, an denen der Wert bekannt ist, auch genau diesen Wert liefert.

(0) $K(\gamma, \overline{x}, \overline{p})(x_i) = p(x_i)$ für alle $i \in \{1,...,n\}$

Dies ist leicht nachzuweisen, wenn man sich überlegt, daß die Kriging-Varianz (G2) an der Stelle x_i jeweils 0 ist, wenn $\delta_i(x)=1$ und $\delta_j(x)=0$ für alle $j \in \{1,...,n\}\backslash\{i\}$ gilt. Da die Kriging-Varianz nicht kleiner als Null sein kann, ist sie in genau diesem Fall minimal.

Die nächsten beiden Eigenschaften zeigen, daß die Kriging-Funktion K bezüglich der Eingabewerte eine lineare Funktion ist, weshalb man auch von *linearem Kriging* spricht.

(1) $K(\gamma, \overline{x}, c\cdot\overline{p}) = c\cdot K(\gamma, \overline{x}, \overline{p})$ für alle $c \in \mathbb{R}$

(2) $K(\gamma, \overline{x}, \overline{p}+\overline{q}) = K(\gamma, \overline{x}, \overline{p}) + K(\gamma, \overline{x}, \overline{q})$ für alle q (wie oben beschrieben)

Außerdem gilt für die Eingabewerte noch, daß man die Addition einer Konstanten herausziehen kann (3).

(3) $K(\gamma, \overline{x}, c+\overline{p}) = c+K(\gamma, \overline{x}, \overline{p})$ für alle $c \in \mathbb{R}$

Für den Variogramm-Parameter gelten folgende zwei Eigenschaften, die zeigen, daß bei Änderung des Variogramms durch Addieren oder Multiplizieren einer reellen Konstante sich die Kriging-Schätzungen gar nicht verändern.

(4) $K(c\cdot\gamma, \overline{x}, \overline{p}) = K(\gamma, \overline{x}, \overline{p})$ für alle $c \in \mathbb{R}^{\neq 0}$

(5) $K(c+\gamma, \overline{x}, \overline{p}) = K(\gamma, \overline{x}, \overline{p})$ für alle $c \in \mathbb{R}$

Dies bedeutet übrigens auch, daß ein Programm die Interpolationsergebnisse nicht neu zu berechnen braucht, wenn nur die Parameter Schwellenwert oder Nuggeteffekt des theoretischen Variogramms verändert werden (eine Veränderung der Aussageweite allerdings bewirkt i.a. Veränderungen der Interpolationsergebnisse).

Die nächsten drei hier erwähnten Eigenschaften beziehen sich auf die Kriging-Varianz V. Die Multiplikation einer Konstanten an das theoretische Variogramm kann man herausziehen(6), wohingegen die Addition einer Konstanten die Kriging-Varianz unverändert läßt (7). Die Kriging-Varianz ist von den Eingabewerten nicht direkt abhängig (8), somit kann man den dritten Parameter hier auch weglassen.

(6) $V(c\cdot\gamma, \overline{x}, \overline{p}) = c\cdot V(\gamma, \overline{x}, \overline{p})$ für alle $c\in\mathbb{R}^{\neq 0}$

(7) $V(c+\gamma, \overline{x}, \overline{p}) = V(\gamma, \overline{x}, \overline{p})$ für alle $c\in\mathbb{R}$

(8) $V(\gamma, \overline{x}, \overline{p}) = V(\gamma, \overline{x}, \overline{q})$ für alle q (wie oben beschrieben)

Aufgrund von (6) muß ein Programm die Kriging-Varianz i.a. auch neu berechnen, wenn der Anwender nur Schwellenwert oder Nuggeteffekt des theoretischen Variogramms ändert (weil auch eine Änderung nur des Nuggeteffekts sowohl die Addition einer Konstanten als auch die Multiplikation einer Konstanten erforderlich macht, um den Schwellenwert unbeeinflußt zu lassen).

Bemerkung: Zu zeigen sind alle diese Eigenschaften durch simples Anwenden der Definitionen von K bzw. V und einfache Umformungen (teilweise unter Anwendung von (G1) aus dem vorigen Kapitel).
(1) bis (3) lassen sich aber auch direkt an der Hauptkriginggleichung ablesen (da die $\delta_i(x)$ nicht direkt von den $p(x_1)$, ..., $p(x_n)$ abhängen, sondern nur über das theoretische Variogramm).
(4) und (5) zeigt man, indem man die Ableitungen der Kriging-Varianzformel betrachtet oder man zeigt zuerst (6) und (7); aus diesen kann man (4) und (5) direkt ableiten (man muß sich dazu nur überlegen, daß die Multiplikation einer Konstanten keine Veränderung beim Minimierungsergebnis (siehe voriges Kapitel) mit sich bringt).
Für (6) und (7) muß man in die Kriging-Varianzformel $c\cdot\gamma$ bzw. $c+\gamma$ anstelle von γ einsetzen und umformen, bis das Gewünschte herauskommt (für (7) benötigt man (G1) aus dem vorigen Kapitel).
(8) zu zeigen ist trivial, denn (8) dient nur als Hinweis, daß die Eingabewerte in der Kriging-Varianzformel gar nicht vorkommen.

Zum Schluß sollen noch einige Eigenschaften betrachtet werden, die zur weiteren Veranschaulichung des Normalen Krigings dienen können.

Zunächst gilt, daß zum zu schätzenden Punkt naheliegende bekannte Werte ein höheres Gewicht ($\delta_i(x)$) haben als die weiter entfernten.

Je näher man sich bei einem bekannten Wert befindet, desto geringer ist die Kriging-Varianz (an den bekannten Punkten ist sie, wie bei (0) gezeigt, sogar 0). Die Kriging-Varianz sinkt auch, je mehr bekannte Werte zur Schätzung herangezogen werden.

Punkte mit bekannten Werten, die relativ zum zu schätzenden Punkt hinter einem anderen liegen, haben ein niedrigeres Gewicht, als es ihrer Entfernung entspräche (hier können sogar negative Gewichte vorkommen); diese Punkte werden gewissermaßen durch den näherliegenden Punkt abgeschirmt (dieser Effekt wird auch "screen-effect" genannt). Ein Beispiel, das auch diesen Abschirmeffekt zeigt, ist in Kapitel 2.3.2.3 (als Gegenbeispiel zu (3')) zu finden.

2.3.1.4 Logarithmische Transformation

Häufig treten bei Anwendungen *lognormalverteilte* Parameter auf; dies sind Parameter p mit der Eigenschaft, daß $\log(p)$ normalverteilt ist. Da eine Normalverteilung des Parameters sehr günstig für die Anwendung von normalem Kriging ist, und da das normale Kriging eine lineare Funktion ist (siehe voriges Kapitel), somit also den nichtlinearen Charakter der Parameterwerte nicht brauchbar berücksichtigen kann, sollte eine Modifikation des Kriging-Verfahrens für diesen Fall eingesetzt werden.

Dazu kann der Parameter vor Anwendung von normalem Kriging zunächst mit Hilfe der Logarithmusfunktion transformiert und bei Ausgabe der Krigingresultate entsprechend mit der Exponentialfunktion zurücktransformiert werden. Es ist zu beachten, daß die Voraussetzungen für normales Kriging dabei natürlich für den logarithmisch transformierten Parameter erfüllt sein müssen.

Bemerkung: Solange das experimentelle Variogramm nur zur Bestimmung des theoretischen Variogramms benötigt wird, ist es naheliegend, diese Variogramme auf der logartithmischen Skala zu belassen. Es ist aber zu berücksichtigen, daß das experimentelle Variogramm dann nicht mehr eine Variogrammanalyse des nichttransformierten Parameters darstellt.

2.3.2 Fuzzy Kriging

In Kapitel 2.3.1 und den dazugehörigen Unterkapiteln wurde konventionelles normales Kriging dargestellt. In Kapitel 1.4 wurde schon dargelegt, welche Variante von Fuzzy Kriging (nämlich jene vom Typ 1) hier von besonderem Interesse ist. Bei dieser Variante kann man unscharfe Parameterwerte verarbeiten, nutzt dafür jedoch nur scharfe theoretische Variogramme. Fuzzy Kriging vom Typ 1 (bis 3) wird in [Bardossy 1989] behandelt.

In Kapitel 2.3.2.1 wird dargestellt, wozu unscharf berechnete experimentelle Variogramme nützlich sind und wie sie näherungsweise berechnet werden können.

Wenn man unscharfe Parameterwerte anstelle von scharfen als Eingabe für das normale Kriging zulassen möchte, muß man das Kriging-Verfahren (siehe Kapitel 2.3.1.2) entsprechend überarbeiten. Dies geschieht in Kapitel 2.3.2.2. Daraufhin muß dann auch untersucht werden, inwieweit sich die Eigenschaften des normalen Krigings übertragen bzw. verändern, was in Kapitel 2.3.2.3 untersucht wird. Die logarithmische Transformation, die man dem Kriging vorschalten kann (siehe Kapitel 2.3.1.4), muß natürlich auch zu einer unscharfen Funktion erweitert werden, was in Kapitel 2.3.2.4 durchgeführt wird.

2.3.2.1 Fuzzy Variogramme

In Kapitel 2.3.1.1 (Variogramme) wurde im Zusammenhang mit den Schwierigkeiten bei der Anpassung von theoretischem Variogramm an das experimentelle Variogramm schon erwähnt, daß man dabei die zusätzliche Information die die Unschärfe mit sich bringt, berücksichtigen könnte; "zusätzlich" ist in Bezug auf entsprechende scharfe Eingabewerte zu sehen (nämlich die Werte, die den Zugehörigkeitsgrad 1 bei den Zugehörigkeitsfunktionen für die unscharfen Werte liefern). Der Vorteil bei der Anpassung mit unscharfem experimentellen Variogramm liegt darin begründet, daß in Bereichen des experimentellen Variogramms, in denen eine große Unschärfe bekannt ist, augenscheinlich die Anpassung an den am ehesten möglichen Wert nicht mehr ganz so streng stattfinden muß. Man kann sich damit auf Bereiche konzentrieren, in denen die Notwendigkeiten genauer bekannt sind.

Leider stellt sich eine exakte Berechnung des unscharfen experimentellen Variogramms als schwierig und zeitaufwendig heraus, da in der Formel zum experimentellen Variogramm Variablen mehrfach auftreten und die Formel auch nicht nur aus monoton steigenden Teilfunktionen zusammengesetzt ist (Differenz, Quadrieren), so daß die Sätze aus Kapitel 2.2.3 nicht angewendet werden können. Für eine exakte Berechnung würde man die folgende Methode benutzen (siehe [Bardossy 1989]).

Da die Formel immerhin stetig ist, steht zumindest fest, daß Intervalle wieder auf Intervalle abgebildet werden (Satz 2 aus Kapitel 2.1.4). Da α-Schnitte (oder auch scharfe α-Schnitte) der unscharfen Parameterwerte $I_1,...,I_n$ fogendermaßen abgebildet werden

$$\overline{\gamma^*}(h, I_1, ..., I_n) = \{y \in \mathbb{R} \mid \exists (x_1, ..., x_n) \in (I_1, ..., I_n) : y = \frac{1}{2N(h)} \sum_{\substack{\text{Eing.koord.} x_j, x_i \\ \text{mit } x_j - x_i = h}} (p(x_i) - p(x_i + h))^2\},$$

können sowohl die linke als auch die rechte Grenze des resultierenden Intervalls durch eine Optimierung berechnet werden ([Bardossy 1989]), was jedoch recht aufwendig ist.

Als Ausweichmöglichkeit bietet sich an, das unscharfe experimentelle Variogramm nur näherungsweise zu berechnen. Da das Ziel, das experimentelle Variogramm unscharf zu berechnen, nur eine von mehreren Entscheidungshilfen beim Anpassen des theoretischen Variogramms an das experimentelle Variogramm darstellt (siehe Kapitel 2.3.1.1), soll die näherungsweise Berechnung im Rahmen dieser Diplomarbeit genügen.

Als Methode zur näherungsweisen Berechnung bietet sich natürlich an, zunächst die Teilfunktionen zu unscharfen Funktionen zu erweitern und dann zusammenzusetzen, obwohl dies dann nicht die unscharfe Funktion ergibt, die man bei Erweiterung von γ^* mit Hilfe des Erweiterungsprinzips (nicht angewendet auf den Parameter *h*, sondern nur auf die Parameterwerte) erhält. Laut Bemerkung 3 am Ende von Kapitel 2.2.3 können bei schnittweiser Berechnung die Intervalle dabei zu groß werden, sie enthalten aber in jedem Fall die Intervalle, die die korrekten Ergebnisschnitte darstellen. Anschaulich gesehen kann also die Unschärfe überschätzt, aber nicht unterschätzt werden.

Was zur Anwendung dieser Methode nur noch fehlt, ist die Erweiterung der Funktion $f: \mathbb{R} \to \mathbb{R}$; $f(x) := x^2$, denn die anderen Teilfunktionen sind monotone und stetige Funktionen und können also mit Satz 3 aus Kapitel 2.1.4 (für abgeschlossene Intervalle) einfach behandelt werden (dazu muß man nur die Subtraktion noch in eine Negation und eine Addition zerlegen).

Für obengenanntes *f* ergibt sich folgende Abbildung abgeschlossener Schnitte

$$\bar{f}([a,b]) = \begin{cases} [b^2, a^2] & \text{falls } b \le 0 \ (f \text{ monoton fallend}) \\ [0, \max(a^2, b^2)] & \text{falls } a < 0 < b \ (\text{Fall siehe unten}) \\ [a^2, b^2] & \text{falls } 0 \le a \ (f \text{ monoton steigend}) \end{cases},$$

denn im Fall $a<0<b$ gilt

$\bar{f}([a,b])$
$=^{(1)} \{ y \in \mathbb{R} \mid \exists x \in [a,b]: y = x^2 \}$
$=^{(2)} \{ y \in \mathbb{R} \mid \exists x \in [a,0]: y = x^2 \} \cup \{ y \in \mathbb{R} \mid \exists x \in [0,b]: y = x^2 \}$
$=^{(3)} [0^2, a^2] \cup [0^2, b^2]$
$=^{(4)} [0, \max(a^2, b^2)].$

Erläuterungen:

(1) Laut Definition.

(2) $\exists x \in [a,b]: y = x^2 \Leftrightarrow (\exists x \in [a,0]: y = x^2) \vee (\exists x \in [0,b]: y = x^2)$.

(3) Satz 3 aus Kapitel 2.1.4 (mit monoton fallend bzw. monoton steigend).

(4) Falls $a^2<b^2$ gilt, dann $[0, a^2] \subseteq [0, b^2]$, also trägt $[0, a^2]$ nichts zu der Vereinigung bei, sondern nur das größere Intervall mit der größeren oberen Grenze (der umgekehrte Fall ist entsprechend).

Damit ergibt sich insgesamt folgende näherungsweise Abbildung der abgeschlossenen Intervalle:

$$\overline{\gamma^*}(h,[a_1,b_1],...,[a_n,b_n]) \approx [a,b] \quad \text{mit}$$

$$a \equiv \frac{1}{2N(h)} \sum_{\substack{\text{Eing.koord.}x_j,x_i \\ \text{mit } x_j-x_i=h}} \begin{cases} (b_i-a_j)^2 & \text{falls} \quad b_i-a_j \le 0 \\ 0 & \text{falls} \quad a_i-b_j < 0 < b_i-a_j \\ (a_i-b_j)^2 & \text{falls} \quad 0 \le a_i-b_j \end{cases} \quad \text{und}$$

$$b \equiv \frac{1}{2N(h)} \sum_{\substack{\text{Eing.koord.}x_j,x_i \\ \text{mit } x_j-x_i=h}} \begin{cases} (a_i-b_j)^2 & \text{falls} \quad b_i-a_j \le 0 \\ \max((a_i-b_j)^2,(b_i-a_j)^2) & \text{falls} \quad a_i-b_j < 0 < b_i-a_j \\ (b_i-a_j)^2 & \text{falls} \quad 0 \le a_i-b_j \end{cases} .$$

Die theoretischen Variogramme werden bei Fuzzy Kriging vom Typ 1 nur in ihrer scharfen Version benötigt, also sind hier keine weiteren Betrachtungen nötig, als in Kapitel 2.3.1.1 schon gemacht wurden (bei den anderen Typen von Fuzzy Kriging müßte der Anwender ein unscharfes theoretisches Variogramm an das unscharfe experimentelle Variogramm anpassen - von theoretischer Seite aus gäbe es dazu dann allerdings auch nichts weiter vorzubereiten).

2.3.2.2 Fuzzy Hauptkriginggleichung

Die Voraussetzungen vom normalen Kriging werden für Fuzzy Kriging vom Typ 1 einfach übernommen, denn sie sagen etwas über den Parameter im allgemeinen aus. Fuzzy Kriging vom Typ 1 ist eine Erweiterung der scharfen Kriging-Funktion $K(\gamma, \bar{x}, \bar{p})$ unter Anwendung des Erweiterungsprinzips nur auf die Parameterwerte $\bar{p}=(p(x_1),..., p(x_n))$. Für die Berechnung der $\delta_i(x)$ waren in Kapitel 2.3.1.2 die Parameterwerte selbst gar nicht vonnöten. Somit beschränken sich die notwendigen Änderungen auf eine Erweiterung der Hauptkriginggleichung mit Hilfe des Erweiterungsprinzips, das in diesem Fall nur auf die Parameterwerte $p(x_1),..., p(x_n)$ angewendet werden soll. Hier wird erneut nur die Abbildung von Schnitten (oder scharfen Schnitten) untersucht, wobei wiederum nur abgeschlossene Intervalle betrachtet werden sollen (wie am Ende von Kapitel 2.1.4 erläutert).

Seien also $\hat{p}(x_1), ..., \hat{p}(x_n)$ die unscharfen Parameterwerte, die anstelle von den scharfen Parameterwerten $p(x_1), ..., p(x_n)$ eingesetzt werden sollen. Als Hauptkriginggleichung ergibt sich nun also

$$\hat{p}^*_\gamma(x, \hat{p}(x_1), ..., \hat{p}(x_n)) = \sum_{i=1}^{n} \delta_i(x) * \hat{p}(x_i)$$

Die abgeschlossenen Intervalle, deren Abbildung nun genauer betrachtet werden soll, seien mit $[a_1,b_1],...,[a_n,b_n]$ bezeichnet. Für sie ergibt sich einfach

$$\overline{p^*_\gamma}(x,[a_1,b_1],...,[a_n,b_n]) = [a,b] \quad \text{mit}$$

$$a \equiv \sum_{i=1}^{n} \begin{cases} \delta_i(x) \cdot a_i & \text{falls } \delta_i(x) \ge 0 \\ \delta_i(x) \cdot b_i & \text{falls } \delta_i(x) < 0 \end{cases} \quad \text{und}$$

$$b \equiv \sum_{i=1}^{n} \begin{cases} \delta_i(x) \cdot b_i & \text{falls } \delta_i(x) \ge 0 \\ \delta_i(x) \cdot a_i & \text{falls } \delta_i(x) < 0 \end{cases} \quad \text{, denn:}$$

1. Das Produkt mit einer Konstanten ist stetig und monoton steigend (falls die Konstante $\delta_i(x)$ positiv ist) bzw. monoton fallend (falls die Konstante $\delta_i(x)$ negativ ist), so daß Satz 3 aus Kapitel 2.1.4 dafür angewendet werden kann. Für die Summenbildung paßt der Satz auch, da sie monoton steigend ist.

2. Die Summe dieser Produkte kann man laut Satz 1 aus Kapitel 2.2.3 (die Summe wäre dort *f* und die Produkte im *i*-ten Summenglied wären die g_i, und die $p(x_i)$ kommen jeweils nur in g_i vor) zur Komplexfunktion erweitern, indem man einfach die zur Komplexfunktion erweiterten Teilfunktionen ineinander einsetzt.

Bemerkung (zum Kriging-Varianz): Da die Parameterwerte $p(x_1),..., p(x_n)$ in der Kriging-Varianzformel gar nicht vorkommen, ist bei der Berechnung der Kriging-Varianz nichts gegenüber dem scharfen Fall zu ändern.

2.3.2.3 Eigenschaften von Fuzzy Kriging (vom Typ 1)

Zunächst einmal soll aber eine Aussage über den Zusammenhang von normalem Kriging und Fuzzy Kriging vom Typ 1 gemacht werden.

(Zusammenahng) Falls $\hat{p}(x_1), ..., \hat{p}(x_n)$ scharfe Zahlen sind, dann liefern normales Kriging und Fuzzy Kriging vom Typ 1 dieselben Werte (dieselben scharfen Zahlen).

Dies ist bei schnittweiser Betrachtung der unscharfen Zahlen sofort klar, denn alle α-Schnitte sind einpunktige Intervalle, und einzelne Werte werden mit der Komplexfunktion von K (das ist auch die Erweiterung, die man zur schnittweisen Berechnung von FK benötigt) genauso abgebildet wie mit K.

Im folgenden wird untersucht, ob sich die Eigenschaften (0) bis (8) aus Kapitel 2.3.1.3 auf den Fall von Fuzzy Kriging vom Typ 1 übertragen, d.h. es wird nun $\bar{p}$ als der Vektor der *unscharfen* Parameterwerten $\hat{p}(x_1), ..., \hat{p}(x_n)$ betrachtet. Die Eigenschaften (0) und (4) bis (8) übertragen sich trivialerweise, denn die Parameterwerte $\hat{p}(x_1), ..., \hat{p}(x_n)$ kommen in der Varianzformel, die jeweils zur Begründung diente, gar nicht vor.

Interessant ist also das Nachprüfen der Eigenschaften (1) bis (3). Dazu wird im folgenden die Fuzzy Kriging-Funktion mit FK anstelle von K bezeichnet.

Eigenschaft (1) besagt nun

(1) $FK(\gamma, \bar{x}, c\cdot\bar{p}) = c\cdot FK(\gamma, \bar{x}, \bar{p})$ für alle $c\in\mathbb{R}$.

Man kann sich das leicht überlegen, indem man Bemerkung 4 aus Kapitel 2.2.3 anwendet. Man betrachte dazu die Funktion

$f(x,p(x_1),..., p(x_n)) := \sum_{i=1}^{n} \delta_i(x) \cdot (c \cdot p(x_i))$.

Wie Bemerkung 4 aus Kapitel 2.2.3 zeigt, ist die Erweiterung von *f* zur unscharfen Funktion mit Hilfe des Erweiterungsprinzips (angewandt auf die Parameter $p(x_1),..., p(x_n)$) möglich, indem man die zu unscharfen Funktionen erweiterten Teilfunktionen ineinander einsetzt, denn die $p(x_i)$ kommen ja jeweils in nur einem Summenterm einmal vor. Insbesondere ist es egal, ob man $\bar{q}:=c\cdot\bar{p}$ vorher berechnet und dann FK(γ, $\bar{x}$, $\bar{q}$)(*x*) ausrechnet, oder ob man die scharfe Gesamtformel *f* direkt mit dem Erweiterungsprinzip behandelt. Somit ergibt sich FK(γ, $\bar{x}$, $c\cdot\bar{p}$)(*x*) = $\bar{f}(x, \hat{p}(x_1), ..., \hat{p}(x_n))$ für alle Koordinaten *x*.

Wenn man des weiteren die Funktion $g(x,p(x_1),..., p(x_n)) := c \cdot \sum_{i=1}^{n} \delta_i(x) \cdot p(x_i)$ betrachtet, ergibt sich, daß $\bar{g}(x, \hat{p}(x_1), ..., \hat{p}(x_n)) = \bar{f}(x, \hat{p}(x_1), ..., \hat{p}(x_n))$ gilt, denn es gilt $g(x, p(x_1), ..., p(x_n)) = f(x, p(x_1), ..., p(x_n))$ (jeweils für alle Koordinaten *x*).
Da obige Argumentation mit Bemerkung 4 aus Kapitel 2.2.3 auch für *g* gilt, ist es egal, ob man erst FK(γ, $\bar{x}$, $\bar{p}$)(*x*) ausrechnet und diesen Wert dann mit *c* multipliziert, oder $c \cdot \sum_{i=1}^{n} \delta_i(x) \cdot p(x_i)$, also $g(x,p(x_1),..., p(x_n))$ direkt mit dem Erweiterungsprinzip behandelt; somit gilt $\bar{g}(x, \hat{p}(x_1), ..., \hat{p}(x_n)) = c\cdot$FK(γ, $\bar{x}$, $\bar{p}$)(*x*) für alle Koordinaten *x*.
Insgesamt ergibt sich also (1).

Interessant ist aber auch die Eigenschaft (1'):

(1') I.a. gilt ***nicht***: FK(γ, $\bar{x}$, $\hat{c}\cdot\bar{p}$) = $\hat{c}\cdot$FK(γ, $\bar{x}$, $\bar{p}$) für alle $\hat{c}\in$ ***FN***.

Wenn man die Formel für die linke Seite betrachtet, stellt man fest, daß die unscharfe Zahl $\hat{c}$ in jedem Summenterm einmal vorkommt, also *n*-mal, sodaß die Anwendung von Bemerkung 4 aus Kapitel 2.2.3 nicht mehr möglich ist. Tatsächlich gilt die Gleichheit auch nicht, was man mit einem einfachen Gegenbeispiel zeigen kann:
Gegenbeispiel: Sei *n*=2, x_1=-1, x_2=1, *x*=0 (also $\delta_1(x)=\delta_2(x)=0.5$), $\hat{p}(x_1)$=1 (die scharfe 1), $\hat{p}(x_2)$=-1 (die scharfe -1) und $\hat{c}$ sei eine unscharfe 0, nämlich

$$m_{\hat{c}}(y) := \begin{cases} 0 & \text{falls} \quad y < -1 \\ 0.5 & \text{falls} \quad -1 \le y < 0 \\ 1 & \text{falls} \quad y = 0 \\ 0.5 & \text{falls} \quad 0 < y \le 1 \\ 0 & \text{falls} \quad 1 < y \end{cases} \quad ; \text{der 0-Schnitt ist dabei } I := \hat{c}^{>0} = [-1, 1].$$

Die Formeln für linke und rechte Seite ergeben also nicht mehr dasselbe (gezeigt hier nur für den 0-Schnitt):

FK(γ, $\bar{x}$, $((\hat{c}^{>0} \cdot \hat{p}(x_1)^{>0}), (\hat{c}^{>0} \cdot \hat{p}(x_2)^{>0}))$)$(x)^{>0}$= 0.5·(*I*·[1,1]) + 0.5·(*I*·[-1,-1]) = [-1,1]

$\neq$

$\hat{c}^{>0}$·FK(γ, $\bar{x}$, $(\hat{p}(x_1)^{>0}, \hat{p}(x_2)^{>0})$)$(x)^{>0}$ = *I* · (0.5·[1,1]+0.5·[-1,-1]) = [0,0]

Bemerkung: Da diese Ungleichheit auch ohne den Faktor 0.5 gilt, also (*I*·[1,1]) + (*I*·[-1,-1]) ≠ *I* · ([1,1]+[-1,-1]), zeigt sich hier auch, daß das Distributivgesetz nicht für unscharfe Zahlen gilt, denn dann müßte Gleichheit gelten.

Eigenschaft (2) gilt auch für Fuzzy Kriging vom Typ 1.

(2) $FK(\gamma, \bar{x}, \bar{p}+\bar{q}) = FK(\gamma, \bar{x}, \bar{p}) + FK(\gamma, \bar{x}, \bar{q})$ für alle unscharfen $q,\bar{q}$ (entsprechend den in Kapitel 2.3.1.3 beschriebenen scharfen $q,\bar{q}$)

Man kann sich das wieder leicht überlegen, indem man Bemerkung 4 aus Kapitel 2.2.3 anwendet, was möglich ist, da in den beiden Funktionen

$f(x,p(x_1),..., p(x_n),q(x_1),..., q(x_n)) := \sum_{i=1}^{n} \delta_i(x) * (p(x_i) + q(x_i))$ und

$g(x,p(x_1),..., p(x_n),q(x_1),..., q(x_n)) := \sum_{i=1}^{n} \delta_i(x) * p(x_i) + \sum_{i=1}^{n} \delta_i(x) * q(x_i)$

die Parameterwerte $p(x_1),..., p(x_n),q(x_1),..., q(x_n)$ wieder nur jeweils einmal vorkommen. Die restliche Argumentation analog zu (1).

Für die Addition einer Konstanten gilt folgendes:

(3) $FK(\gamma, \bar{x}, c+\bar{p}) = c+FK(\gamma, \bar{x}, \bar{p})$ für alle $c \in \mathbb{R}$

(3') I.a. gilt ***nicht***: $FK(\gamma, \bar{x}, \hat{c}+\bar{p}) = \hat{c}+FK(\gamma, \bar{x}, \bar{p})$ für alle $\hat{c} \in \boldsymbol{FN}$.

Man kann (3) genau so zeigen, wie (1) gezeigt wurde ($f=g$ gilt in diesem Fall, da $\sum_{i=1}^{n} \delta_i(x) = 1$, also $\sum_{i=1}^{n} \delta_i(x) \cdot c = c$).

Für (3') ist wieder ein Gegenbeispiel anzugeben:

Sei $n=2$, $x_1=1.2$, $x_2=0$, $x=-0.9$ (also mit $\gamma(y_1,y_2):=|y_2-y_1|$ folgt $\delta_1(x)=-0.1$ und $\delta_2(x)=1.1$), $\hat{p}(x_1)=\hat{p}(x_2)=0$ (die scharfe 0) und $\hat{c}$ sei die unscharfe 0 aus dem Gegenbeispiel zu (1') mit dem 0-Schnitt $I := \hat{c}^{>0} = [-1, 1]$.

Die Formeln für linke und rechte Seite ergeben also nicht mehr dasselbe (gezeigt hier nur für den 0-Schnitt):

$FK(\gamma, \bar{x}, ((\hat{c}^{>0} + \hat{p}(x_1)^{>0}), (\hat{c}^{>0} + \hat{p}(x_2)^{>0})))(x)^{>0} = -0.1 \cdot (I+[0,0]) + 1.1 \cdot (I+[0,0]) = [-1.2,1.2]$

$\neq$

$\hat{c}^{>0}+FK(\gamma, \bar{x}, (\hat{p}(x_1)^{>0}, \hat{p}(x_2)^{>0}))(x)^{>0} = I + (0.5 \cdot [0,0]+0.5 \cdot [0,0]) = [-1,1]$

2.3.2.4 Logarithmische Transformation

Auch die scharfe logarithmische Transformation aus Kapitel 2.3.1.4 und die Rücktransformation müssen mit dem Erweiterungsprinzip behandelt werden, um unscharfe Zahlen abbilden zu können. Die Abbildung abgeschlossener Intervalle ist mit Hilfe von Satz 3 aus Kapitel 2.1.4 möglich, da sowohl die Funktion $\log_{10}$ als auch die Exponentialfunktion stetig und monoton steigend sind. Abgeschlossene Intervalle $[a,b]$ werden also folgendermaßen abgebildet:

$$\log_{10}([a,b]) = [\log_{10}(a), \log_{10}(b)]$$

und

$$10^{[a,b]} = [10^a, 10^b]$$

3 Implementierung

Zur Diplomarbeit gehört die Implementierung der unscharfen Aggregation (Kapitel 2.2) und des Fuzzy Krigings vom Typ 1 (Kapitel 2.3). Die unscharfen experimentellen Variogramme sollen, wie in Kapitel 2.3.2.1 dargestellt, nur näherungsweise berechnet werden.

Das entwickelte Programm hat den Namen FUZZEKS erhalten, was für "Fuzzy Evaluation and Kriging System" steht. Neben der Implementierung obiger Verfahren enthält es auch eine Oberfläche, die dem Anwender in übersichtlicher Art und Weise die Anwendung der Verfahren ermöglicht. Insbesondere benötigt man natürlich eine Ein- und Ausgabeschnittstelle für die unscharfen Daten; beides ist mit Hilfe von ASCII-Dateien möglich, der Ausgabe wurde jedoch große Aufmerksmkeit zugewandt, um die Resultate auch übersichtlich und aussagekräftig mit Hilfe von Graphiken präsentieren zu können.

Die Diplomarbeit wurde beim Projektzentrum Ökosystemforschung erstellt. Als Betriebssystem für die Implementierung wurde Windows 3.1 gewählt, da dies dort an jedem Arbeitsplatz verfügbar ist und über Fenstertechnik verfügt, die von FUZZEKS auch intensiv genutzt wird.

In Kapitel 3.1 wird zunächst die Benutzerschnittstelle beschrieben. Aus dieser und aus den Notwendigkeiten, die sich schon aufgrund der theoretischen Grundlagen aus Kapitel 2 ergeben (z.B. eine Repräsentation unscharfer Zahlen implementieren zu müssen), entstehen die Forderungen an die Datenstrukturen, die dann in Kapitel 3.2 dargestellt werden. Kapitel 3.3 beschreibt dann Programmstrukturen und Algorithmen, die auf der Grundlage dieser Datenstrukturen entstanden sind und alle wesentlichen Aufgaben innerhalb des Programms übernehmen. Eine Übersicht über die Programmstruktur liefert Abbildung 3.3 am Anfang von Kapitel 3.3.

3.1 Die Benutzerschnittstelle

Das grundlegende Konzept für die Implementierung besteht darin, die Inhalte zu einzelnen Themenbereichen jeweils mit Hilfe eines Fensters sichtbar und manipulierbar zu machen. Folgende Liste zeigt die Aufteilung der Themen in Fenster.

- Das **Hintergrundfenster** erlaubt grundlegende organisatorische Aufgaben zu erledigen, wie z.B. das Einlesen von ASCII-Dateien, Festlegen von bestimmten Verzeichnissen im Dateisystem und das Verlassen des Programms.

- Das **Managementfenster** (siehe Abbildung 3.1a) bildet die Aggregationsmöglichkeiten ab: Den Transformationsfunktionen und den Verknüpfungsoperatoren sind jeweils eigene Symbole zugeordnet; die Verbindungen zwischen ihnen sind mit Hilfe von Verbindungslinien dargestellt. Das Fenster erlaubt es, Verknüpfungsoperatoren zu plazieren und Verbindungen mit Hilfe der Maus zu erzeugen, zu verändern und zu löschen. Einige Details dazu wurden jedoch in Unterfenster verlagert; sie sind zu erreichen, indem man in den entsprechenden Bereich eines Symboles des Managementfensters mit der Maus klickt. Außerdem ist es auf die gleiche Art und Weise möglich, Krigingfenster zu öffnen.
  Bemerkung: Der Mauscursor erscheint hier immer in Form eines Symbols, das andeutet, was ein Mausklick an der aktuellen Stelle bewirkt (eine vollständige Liste dieser Symbole ist im Hilfetext zu finden).
  Hier aber zunächst eine Liste der Unterfenster des Managementfensters:
  - **Zugehörigkeitsfunktionsfenster** (siehe Abbildung 3.1b) erlauben die Definition jeweils einer Transformationsfunktion laut Kapitel 2.2.1. Der Name des Fensters rührt von der Tatsache her, daß die Transformationsfunktionen jeweils mit Hilfe einer Zugehörigkeitsfunktion definiert werden.
  - Die Gewichte der SUM-Operatoren laut Kapitel 2.2.2 können in einem entsprechenden Dialogfenster geändert werden.
  - Kommentare zu Aggregationsoperatoren können auch in einem entsprechenden Dialogfenster geändert werden.

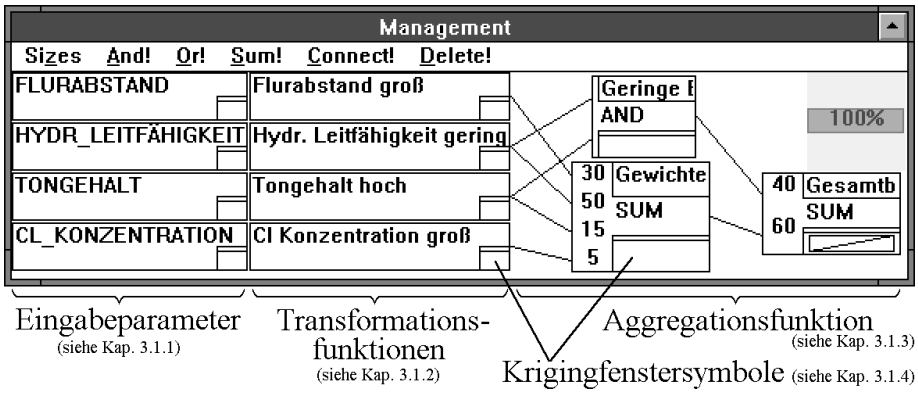


Abbildung 3.1a: Das Managementfenster

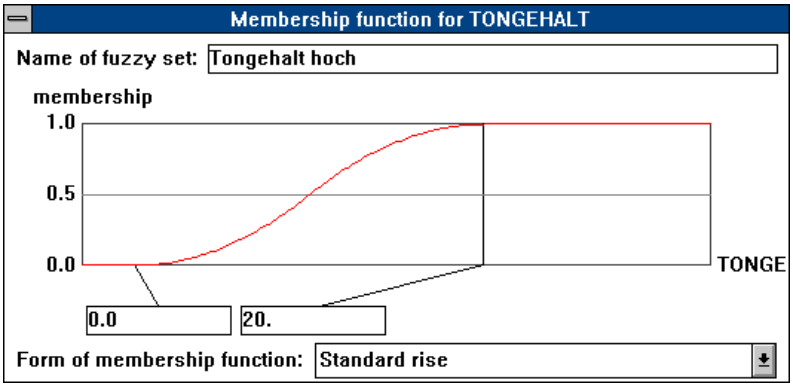


Abbildung 3.1b: Ein Zugehörigkeitsfunktionsfenster

- **Krigingfenster** (siehe Abbildung 3.1c) erlauben die Handhabung von Fuzzy Kriging (vom Typ 1). Hier werden sowohl Variogramme als auch Kriging-Ergebnisse dargestellt, das theoretische Variogramm kann hier angepaßt werden, und es können über verschiedene Dialogfenster diverse Parameter eingestellt werden: zur Berechnung und Abbildung von experimentellem Variogramm und Krigingresultat. Außerdem können hier Ergebnisdaten (experimentelles Variogramm und Krigingresultat) in Form von ASCII-Dateien exportiert werden.
  Um die Resultate noch detaillierter graphisch darzustellen, kann man für einzelne Punkte und für Schnitte in einem eigenen Unterfenster darstellen:
  - Im **Punktfenster** (siehe Abbildung 3.1d) wird zu einer einzelnen Koordinate die Zugehörigkeitsfunktion des unscharfen Krigingresultats angezeigt.
  - Im **Schnittfenster** (siehe Abbildung 3.1e) werden zu einer geraden Linie zwischen zwei Koordinaten die unscharfen Krigingresultatwerte dargestellt, indem für die α-Schnitte, für die die Krigingresultate berechnet wurden, jeweils die untere und obere Grenze des Intervalls abgebildet wird.

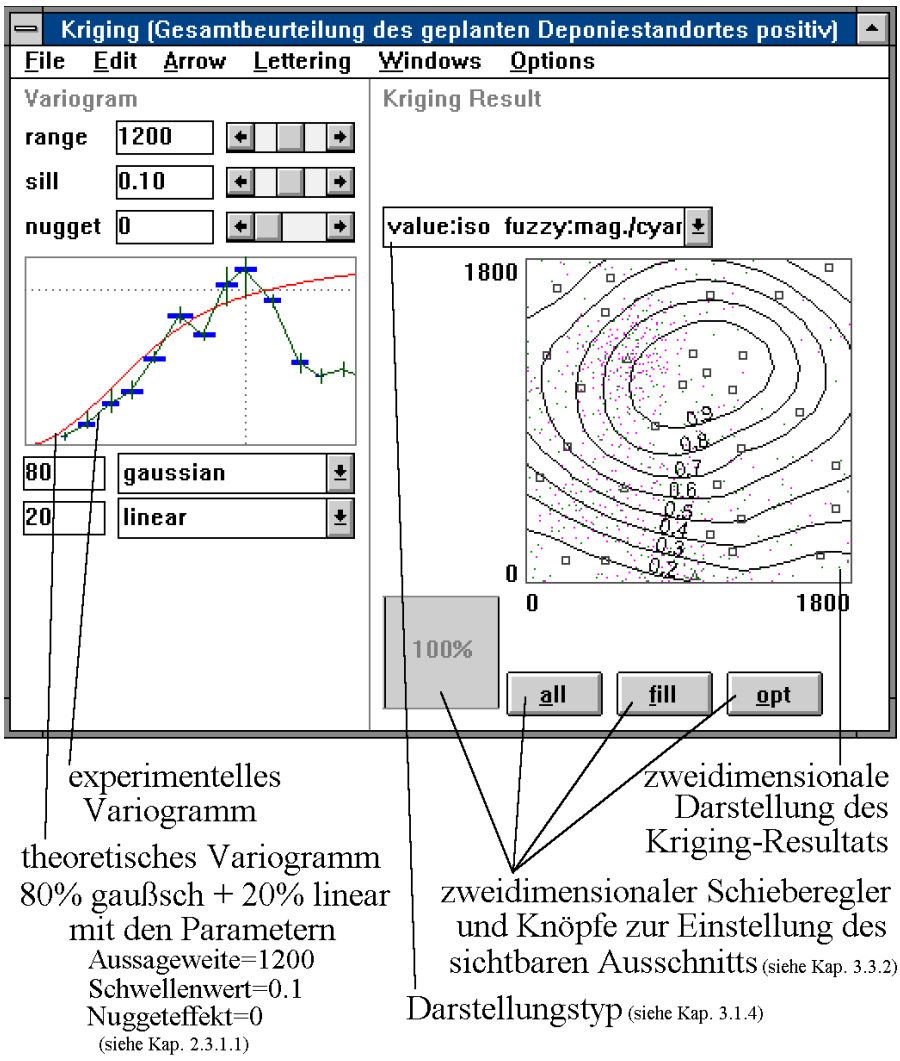


Abbildung 3.1c: Ein Krigingfenster

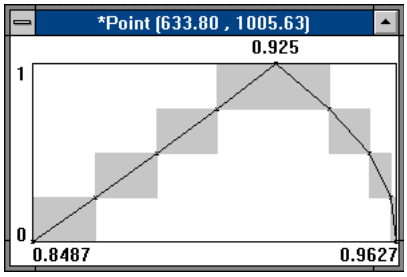


Abbildung 3.1d: Ein Punktfenster

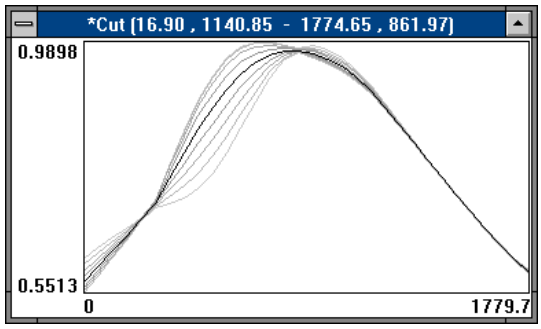


Abbildung 3.1e: Ein Schnittfenster

In folgenden Unterkapiteln wird der Umgang mit dem Programm vom Laden einer Eingabedatei bis zum Exportieren von Resultatdaten nachgezeichnet.

3.1.1 Festlegen der betrachteten Parameter, Eingabeformat von ASCII-Dateien

Durch das Einlesen einer Eingabedatei kann man neue Parameter zu einer aktuellen Liste von Parametern ergänzen. Falls die Namen der eingelesenen Parameter schon vorhanden sind, werden die Daten aktualisiert. Die aktuelle Liste von Parametern wird im Managementfenster angezeigt. Dies dient als Grundlage dafür, mit den Eingabedaten, wie in folgenden Kapiteln beschrieben, weiter zu verfahren.

Die Eingabedateien müssen in dem speziell für diesen Zweck erstellten sog. FDF-Format vorliegen. FDF kürzt dabei "fuzzy data file" ab und wird als Dateiendung für die Eingabedateien verwendet. Im folgenden wird die Grammatik dieses ASCII-Dateiformats mit ihren Besonderheiten vorgestellt.

Die Grammatik ist mit Hilfe von Produktionen in der üblichen Form aufgeschrieben. "{*x*}" bedeutet hier, daß *x* beliebig oft (auch null mal) vorkommen darf. "fuzzy_data_file" ist das Axiom. Nach der Grammatik selbst folgen genauere Erläuterungen und Bemerkungen zur Semantik sowie ein Beispiel.

```
fuzzy_data_file ::= region cuts params { table } end

region          ::= [decisepdef] "region" leftbottom rightbottom lefttop [area]
leftbottom      ::= [ newline ] number number
rightbottom     ::= [ newline ] number number
lefttop         ::= [ newline ] number number newline
area            ::= { number number [newline] } newline

cuts            ::= [ decisepdef ] "cuts" [ newline ] { number [ newline ] }

params          ::= [ decisepdef ] "parameters" [ newline ] { param }
param           ::= idf [ ":" "variable" ] { "(" param_type ")" } newline
param_type      ::= "+" [ number ] "/" "-" number  |  "log10"

table           ::= [ tabledefs ] "table" { newline } attrib_line { fvalue_line }
attrib_line     ::= attrib { delimiter attrib } newline { newline }
attrib          ::= "x"  |  "y"  |  idf
fvalue_line     ::= fvalue { delimiter fvalue } newline { newline }
fvalue          ::= (  ( hcut | number ) [ {"/" hcut} | {"/" cut} ]  )  |  undef
hcut            ::= number ":" number "-" number
cut             ::= number "-" number

tabledefs       ::= [ undefdef ] [ delimiterdef ] [ decisepdef ]
undefdef        ::= "undefstring"          "is" undef      newline
delimiterdef    ::= "delimiterstring"      "is" delimiter  newline
decisepdef      ::= "decisepcharacter"     "is" decisep    newline
end             ::= "end"
number          ::= ["+"|"-"] digitstr
```

Als atomare Ausdrücke, die vom Scanner erkannt, und dem Parser geliefert werden, treten hier auf:

- "idf" und durch Anführungsstriche gekennzeichneter Text sind Zeichenketten. Klein- oder Großschreibung ist dabei egal.
- "digitstr" beschreibt numerische Ausdrücke ohne Vorzeichen; Exponentialschreibweise ist erlaubt. Für "digitstr" gilt folgende Scanner-Teilsyntax:

```
digitstr ::= ( digit{digit} [decisep {digit}]       ['e'['+'|'-']digit{digit}] )
           | (              decisep digit{digit} ['e'['+'|'-']digit{digit}] )
digit    ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

- "newline" beschreibt das Zeilenende. Dies ist entweder CR/LF (Carriage Return, Code 13 und Line Feed, Code 10) oder nur LF.
- "undef" steht für eine spezielle Zeichenkette (bzw. ein Sonderzeichen), deren Bedeutung "Wert unbekannt" sein soll; welche Zeichenkette (bzw. welches Sonderzeichen) für den undefinierten Wert stehen soll, kann durch spezielle Anweisungen in der Eingabedatei festgelegt werden.
- "delimiter": Trennt die Spalten im "table"-Teil voneinander ab.

Die Grammatik teilt Eingabedateien zunächst grob in folgende Teile auf:

- Im "region"-Teil beschreibt man, für welches Gebiet man sich interessiert, d.h. in welchem Bereich man später Werte mit Hilfe von Fuzzy Kriging interpolieren möchte. Dazu dienen drei Koordinaten, die ein beliebig gedrehtes Rechteck beschreiben: Linke untere, rechte untere und linke obere Ecke. Außerdem kann man noch eine Umrandung in Form eines Polygons festlegen, außerhalb dessen Werte nicht dargestellt werden sollen.
- Im "cuts"-Teil wird festgelegt, welche α-Schnitte zur Darstellung der unscharfen Zahlen benutzt werden sollen. Sie müssen absteigend sortiert sein, und 1 sowie 0 müssen enthalten sein. Für $\alpha \in [0,1)$ sind α-Schnitte gemeint; für $\alpha=1$ ist der scharfe 1-Schnitt gemeint. Warum dies so sein soll, wird in Kapitel 3.2.1 (Datenstrukturen / Schnittdarstellung von unscharfen Zahlen) genauer erläutert.
- Im "params"-Teil werden die Namen der Parameter, die in dieser Datei vorkommen sollen, aufgelistet. Zusätzlich kann eine Vorgabeunschärfe festgelegt werden (z.B. "+/-2" bedeutet: Wenn bei diesem Parameter im "table"-Teil nur "number" x vorkommt und keine weiteren Schnitte beschrieben sind, die Eingabe als eine dreieckige unscharfe Zahl mit dem 0-Schnitt $[x-2,x+2]$ interpretiert wird), oder man kann "log10" angeben, um die logarithmische Transformation (Kapitel 2.3.2.4) zu bewirken.

- In "table"-Teilen werden dann die eigentlichen unscharfen Parameterwerte in Form von Tabellen angegeben. Zunächst beschreibt eine Attributezeile, in welcher Spalte der Tabelle welcher Parameterwert zu finden ist, wobei "x" und "y" für die Koordinatenangaben vorkommen müssen; andere Bezeichner müssen aus dem "params"-Teil stammen. In den folgenden Zeilen werden dann die Koordinaten bzw. unscharfen Parameterwerte selbst (die von der Form "fvalue" sein müssen) aufgelistet.
  Die Produktion "fvalue" (Kurzform für "fuzzy value") legt fest, wie man unscharfe Zahlen schreibt: Sie besteht aus der Angabe der α-Schnitte, wobei man neben Intervallen [*a*,*b*] (geschrieben α ":" *a* "-" *b*) auch Intervalle [*a*,*a*] in Kurzform darstellen darf (geschrieben *a* anstatt *a* "-" *a*) und α ":" weglassen darf, falls man die Schnitte in der Reihenfolge, wie im "cuts"-Teil beschrieben, auflistet. Die Schnitte für verschiedene α werden mit "/" voneinander abgegrenzt. Der erste angegebene Schnitt muß immer der scharfe 1-Schnitt sein und kann in der Form "number" geschrieben werden, da er bei unscharfen Zahlen immer nur einen Wert enthält. Die α-Schnitte müssen absteigend nach α sortiert sein.
  Falls man nicht die im "cuts"-Teil angegebenen Schnitte festlegt, werden aus den angegebenen Schnitten die Schnitte, die der Repräsentation dienen, (ggfs. nach der logarithmisierung) linear interpoliert.
  Auch ein undefinierter Wert ist als "fvalue" erlaubt.
  Beispiel: Der Ausdruck "5.4/0:4.4-6.4" legt fest, daß der scharfe 1-Schnitt das Intervall [5.4,5.4] und der 0-Schnitt das Intervall [4.4,6.4] sei. Falls im "cuts"-Teil auch der 0.5-Schnitt angegeben war, so wird als 0.5-Schnitt [4.9,5.9] angenommen (wenn es sich um einen Wert zu einem nicht zu logarithmisierenden Parameter handelt).
- Das syntaktische Ende der Datei wird mit "end" gekennzeichnet.

Damit man schon vorhandene Dateien besser an diese Syntax anpassen kann, ist es möglich, bestimmte syntaktische Festlegungen an bestimmten Stellen der Paramterdatei (nämlich jeweils vor den oben aufgelisteten Teilen außer "end") umzudefinieren:

- "undef": Darstellung von undefinierten Werten.
  Voreinstellung: "undef".
  Nicht erlaubt: delimiter, decisep, newline, "undefdefstring", "delimiterstring", "decisepcharacter", "table", "end".
- "delimiter": Trennt die Spalten im "table"-Teil voneinander ab; auch die leere Zeichenkette ist dafür erlaubt (dann muß nur mindestens eine Leerstelle zwischen zwei Einträgen vorkommen).
  Voreinstellung: "".
  Nicht erlaubt: decisep, undef, newline, "/", ":", number, "undefdefstring", "delimiterstring", "decisepcharacter", "table", "end".
- "decisep": Trennt die Vorkommastellen von den Nachkommastellen bei den dezimalen Zahlendarstellungen. Nur ein Sonderzeichen ist erlaubt. Diese Einstellung wird vom Scanner beachtet, und zwar bei "digitstr".
  Voreinstellung: ".".
  Nicht erlaubt: delimiter, undef, "", alphanumerische Zeichen, mehr als ein Zeichen lange Zeichenketten.

Folgende Beispieleingabedatei dient auch als Grundlage für die Beispiele in den folgenden Kapiteln (und entspricht der Eingabedatei, die im Hilfetext für das dort angegebene Beispiel verwendet wird).

Diese Beispieleingabedatei soll die Grundlage für einen aggregierten Parameter bilden, der etwas über die Eignung eines jeweiligen Ortes als Mülldeponie-Standort aussagt.

```
region
          0       0
       1800       0
          0    1800
cuts
    1   0.75   0.5   0.25   0
parameters
    Cl_Konzentration
    Tongehalt
    Hydr_Leitfähigkeit  (log10)
    Flurabstand
table
 x     y       Flurabstand    Hydr_Leitfähigkeit          Tongehalt      Cl_Konzentration

 165 1630    3.9            0.000003000                 12             20
 460 1690    4.0            0.000001000                 13             30
 435 1500    4.5            0.000000600                 17             40
 105 1265    4.5            0.000001000                 15             40
 300 1080    5.0            0.000000400                 17             75
 560 1240    5.4/0:4.4-6.4  0.7e-7/0:0.7e-8-0.7e-6      22/0:17-27     80/0:50-130
1010 1600    3.8            0.000000100                 14             65
1400 1600    3.4            0.000000200                 13             85
1200 1260    4.8            0.000000009                 19            130
1510  950    5.2            0.000000300                 14            175
1710  650    4.9            0.000003000                 10            150
 540  530    4.2/0:3.6-4.9  0.000000700                 10            145/0:95-195
 225  755    4.3            0.000000800                 12             90
  70  590    3.6            0.000004000                  8             70
 210  130    2.8            0.000010000                  3             65
 435  130    3.0            0.000009000                  4            100
 930   45    3.3/0:2.3-4.3  0.000010000                  4            105/0:55-155
1010  275    4.1            0.000004000                  8            160
1135  175    3.9            0.000007000                  7            130
1185  360    4.5            0.000003000                  9            170
1705  415    4.4            0.000007000                  8            125
1625  160    3.8            0.000020000                  5             85
 915 1270    5.4            0.000000008                 22            101
1000 1165    5.7            0.000000002                 21            130
 860 1100    6.2            0.000000007                 21            130
1670 1745    2.9            0.000001000                 10             85
1135 1070    5.8            0.000000020                 18            155
 705  870    6.1            0.000000060                 16            145
1190  750    6.0            0.000000400                 14            220
1055  545    4.8            0.000000700                 12            210
end
```

Nun folgen noch einige technische Anmerkungen zur Implementierung.

Der Scanner zerlegt den eingelesenen Text nach folgenden Regeln in die atomaren Ausdrücke (Token), die schon weiter oben aufgezählt wurden.

- Ein nicht-alphanumerisches Zeichen (ein Sonderzeichen) wird als komplettes Token interpretiert.
- Aufeinanderfolgende Leerstellen werden als ein Trenner zwischen Token interpretiert.
- Ausnahme zu obigen Regeln ist "digitstr", dessen Syntax schon weiter oben beschrieben wurde, denn hierfür werden mehrere Token laut obigen zwei Regeln zu einem Token zusammengefaßt.

Die Implementierung des Parsers wurde in Form eines "Recursive-descent" Parsers vorgenommen, bei dem eine Funktion für jede Produktion geschrieben wird, die den Wahrheitswert TRUE (wahr) liefert, falls das jeweils aus der Eingabe Folgende von der entsprechenden Form ist, wobei in diesem Fall als Seiteneffekt auch die entsprechenden Zeichen aus der Eingabewarteschlange entfernt werden. Nach geringfügigen äquivalenten Umformungen konnte die Syntax mit Lookahead eins geparst werden (d.h., daß man immer nach Betrachtung des jeweils nächsten Tokens schon erkennen kann, welche Produktion als nächstes anzuwenden ist). Der Grund für die andere Darstellung der Syntax ist, daß oben die Syntax in einer besser lesbaren Form angegeben werden konnte.

3.1.2 Festlegen der Transformationsfunktion

Nachdem man eine Eingabedatei eingelesen hat, erscheint im Managementfenster am linken Rand für jeden eingelesenen Parameter ein Symbol, das mit seinem Namen gekennzeichnet ist. Rechts daneben erscheint zusätzlich jeweils ein Symbol für die Transformationsfunktion. Wenn der Anwender darauf mit der Maus klickt, öffnet sich das Fenster zum Editieren der Transformationsfunktion (siehe Abbildung 3.1b).

Die Transformationsfunktion wird festgelegt, indem man zunächst einen der Grundtypen aus dem Auswahlfeld auswählt. Daraufhin werden unter der graphischen Darstellung der Transformationsfunktion entsprechende Eingabefelder für die weiteren Parameter angezeigt, die dann natürlich auszufüllen sind.

Zusätzlich kann man die Aussage, deren Akzeptanz die Transformationsfunktion darstellt (siehe Kapitel 2.2.1), in ein entsprechendes Eingabefeld als Kommentar eintragen. Dies kann auch als Bezeichnung der unscharfen Menge, deren Zugehörigkeitsfunktion die Akzeptanz definiert, gesehen werden. Obendrein kann dies alternativ als die Bezeichnung des linguistischen Wertes aufgefaßt werden, dessen Zugehörigkeitsfunktion die Transformationsfunktion festlegt, falls man den Parameter als linguistische Variable betrachtet. Siehe zu diesem Thema auch den Anfang von Kapitel 2.2.1.

3.1.3 Festlegen der Aggregationsfunktion

Die Aggregationsfunktion soll die transformierten Parameterwerte verknüpfen und muß sich aus den Aggregationsoperatoren, die in Kapitel 2.2.2 angegeben wurden (zwei logische, nämlich AND und OR, und ein arithmetischer, nämlich die gewichtete Summe SUM), zusammengesetzt werden.

Die Aggregationsfunktion wird, wie schon in in Abbildung 1.3 aus Kapitel 1.3 gezeigt, in graphischer Form eingegeben und dargestellt und nicht in Form eines Terms. Natürlich ist diese Darstellung aber äquivalent zur Darstellung als Term. Sie hat allerdings den Vorteil, daß man gleiche Teilterme nicht mehrfach eingeben muß, sondern einmal eingegebene Teilterme mehrfach verwenden kann, was allerdings nur als abkürzende Darstellung für die wiederholte Schreibweise zu betrachten ist. Dies macht jedoch hier keinen Unterschied, da, wie in Kapitel 2.2.3 deutlich gemacht, die gesamte Aggregationsfunktion zur unscharfen Funktion mit Hilfe des Erweiterungsprinzips erweitert wird, was auch bedeutet, daß die Erweiterung zur unscharfen Funktion nicht abhängig von der Darstellung des Terms ist (vergleiche mit Kapitel 2.2.3).

Die praktische Vorgehensweise beim Eingeben der Aggregationsfunktion in graphischer Form besteht darin, zunächst Aggregationsoperatoren auszuwählen und zu plazieren. Dann zieht man Verbindungslinien zwischen den Ausgängen von Transformationsfunktionen oder anderen Operatoren und den Eingängen (Parametern) der Operatoren. Dies findet alles im Managementfenster (siehe Abbildung 3.1a) statt.

Man wählt die Aggregationsoperatoren über das Menü des Fensters, woraufhin sich der Mauszeiger in das entsprechende Operatorsymbol wandelt. Dieses Symbol bewegt man dann mit der Maus einfach an die gewünschte Stelle und drückt die Maustaste, um es dort abzulegen.

Die Verbindungen legt man fest, indem man zunächst aus dem Menü des Fensters die Verbindungsfunktion auswählt, dann das darauf erscheinende verbinde-von-Symbol (Mauscursor) auf das Tranformationsfunktions- oder Operatorsymbol bewegt, dessen Ausgang benutzt werden soll und die Maustaste drückt; daraufhin erscheint das verbinde-nach-Symbol, das man auf das Symbol des Operators bewegt, dessen Eingang festgelegt werden soll, und man drückt dort erneut die Maustaste.

Außerdem gibt es natürlich die Möglichkeit, bestehende Aggregationsfunktionen zu verändern: Man kann die Operatorsymbole verschieben, löschen oder neue einfügen, und man kann Verbindungen löschen oder neue festlegen. Die Lage der Verbindungen kann man nur über die Lage der Operatorsymbole beeinflussen; das Programm legt die Anordnung der Eingänge eines Operatorsymbols selbständig so fest, daß sich möglichst wenige Verbindungslinien überschneiden.

Große Vorteile der graphischen Darstellung sind, daß man die Zwischenergebnisse mit Kommentaren bezeichnen kann, was der Übersicht dient, und daß man sich die Zwischenergebnisse anzeigen lassen kann (zur Vorgehensweise dazu siehe folgendes Kapitel).

3.1.4 Umgang mit Fuzzy Kriging vom Typ 1

In jedem der Symbole im Managementfenster befindet sich in der unteren rechten Ecke ein Krigingfenstersymbol (siehe Abbildung 3.1a). Indem man mit der Maus darauf klickt, öffnet man das Krigingfenster (siehe Abbildung 3.1c), das erlaubt, die Werte zu diesem Parameter mit Hilfe von Fuzzy Kriging als zweidimensionale Graphik darzustellen. Wenn das Krigingfenster schon offen war, wird es nur in den Vordergrund gebracht. Wenn man also z.B. das Endergebnis der Aggregationsfunktion räumlich darstellen möchte, muß man in die rechte untere Ecke des Symbols für den Aggregationsoperator klicken, dessen Ausgang unbenutzt ist, der also das Endergebnis repräsentiert. Ein Vorteil dieser Vorgehensweise ist, daß man Krigingfenster, die man sucht (z.B. weil sie in den Hintergrund von anderen Fenstern geraten sind), leicht finden kann, indem man (per Tastendruck) das Managementfenster in den Vordergrund bringt und dort einfach auf das entsprechende Symbol klickt.

Wenn man dieses Krigingfenster noch nicht zuvor geöffnet hat oder die räumlich darzustellenden Werte inzwischen verändert wurden, muß man zunächst das theoretische Variogramm an das experimentelle anpassen (Dabei geht man prinzipiell vor wie in Kapitel 2.3.1.1 und Kapitel 2.3.2.1 sowie im Hilfetext beschrieben). Praktisch führt man dies durch, indem man zunächst einen – oder zwei – der zur Auswahl gebotenen Funktionstypen für das theoretische Variogramm aus den Listenauswahlfeldern auswählt – bei zweien muß man noch die prozentuale Verteilung in den Eingabefeldern links daneben angeben – und dann die Parameter Aussageweite (range), Schwellenwert (sill) und Nuggeteffekt (nugget-effect) festlegt. Eventuell sollte man die Parameter zur Berechnung und Anzeige des experimentellen Variogramms dafür zuvor abändern (über das Dialogfenster, das sich nach Auswahl des Menüpunkts Optionen/Variogramme öffnet); damit kann man z.B. eine andere Anzahl von Klassen vorgeben, in die die Abstände zwischen Punkten eingeteilt werden (siehe dazu Kapitel 2.3.1.1).

Bemerkung: Klassen für verschiedene Richtungen (also richtungsabhängige Variogramme) sind in FUZZEKS nicht vorgesehen.

Nun ist nur noch die geeignete Darstellung des Kriging-Ergebnisses auszuwählen. Der Bereich, für den Interpolationen durchgeführt werden sollen, wurde in der Eingabedatei festgelegt. Der Anwender hat bei der Darstellung des Krigingresultats in zwei Dimensionen nun vielfältige Auswahlmöglichkeiten, die die Darstellung beeinflussen. Zunächst einmal wird man einen Darstellungstyp aus einer Liste von vorgegebenen Darstellungstypen aus dem Listenauswahlfeld, das in Abbildung 3.1.4 in seiner aufgeklappten Form zu sehen ist, auswählen. Dieses Listenauswahlfeld ist Bestandteil des Kriging-Fensters (siehe Abbildung 3.1c). Diese Darstellungstypen setzen sich aus ein bis zwei Unterdarstellungstypen zusammen; diese legen jeweils fest, *was*, nämlich welcher Teil des unscharfen Resultats bzw. der Kriging-Varianz, dargestellt wird und *wie* er dargestellt wird.

Abbildung 3.1.4: Listenauswahlfeld für Darstellungstypen zur zweidimensionalen Darstellung des Krigingresultats

Folgende Teile des unscharfen Resultats bzw. der Kriging-Varianz kommen bei den Darstellungstypen vor:

- "value": Die 'möglichsten' Werte, d.h. für jede dargestellte Stelle der Wert, bei dem der Zugehörigkeitswert der Zugehörigkeitsfunktion der interpolierten unscharfen Zahl an dieser Stelle eins ist. In dem Fall, daß alle Eingaben hierfür scharfe Zahlen sind, ist dies genau das scharfe Kriging-Resultat.
- "value(lo)" bzw. "value(hi)": Für jede dargestellte Stelle die untere bzw. obere Grenze des Intervalls, das sich als 0-Schnitt der interpolierten unscharfen Zahl an dieser Stelle ergibt.
- "fuzzy(lo)" bzw. "fuzzy(hi)": Der Wert ergibt sich jeweils aus obigen Werten: ("value-value(lo)") bzw. ("value(hi)-value"). Dies beschreibt die möglichen Abweichungen vom am ehesten möglichen Wert nach unten bzw. oben (der Darstellungstyp "fuzzy" steht für das Wertepaar ("fuzzy(lo)", "fuzzy(hi)")).
- "vari.": Die Kriging-Varianz.

Folgende Methoden zur Darstellung können benutzt werden, um obiges darzustellen:

- "iso": Es werden Isolinien benutzt. Die Isolinien können beschriftet werden, indem man mit der Maus eine orange Linie durch das Isolinienbild zieht und dann über das Menü die Funktion zum Beschriften der Isolinien auswählt. Diese Funktion bewirkt, daß an jedem Schnittpunkt zwischen der orangen Linie und einer Isolinie der Wert zur Isolinie als Beschriftung angebracht wird. Die Ausrichtung der Beschriftung wird tangential an der Isolinie orientiert. Dafür gibt es zwei Möglichkeiten; welche davon ausgewählt wird, hängt davon ab, in welcher Richtung die orange Linie gezogen wird.
  Mit einem zusätzlichen Dialogfenster kann man die Größe der Beschriftungen und die gewünschte Zahlendarstellung festlegen.
- "dark": Grauwerte werden zur Darstellung benutzt, wobei dunklere Graustufen höhere Werte darstellen. Die Skalierung kann über ein zusätzliches Dialogfenster eingestellt werden.

- "brightness": Es werden ebenfalls Graustufen zur Darstellung benutzt, jedoch geben hier hellere Graustufen höhere Werte an. Die Skalierung findet automatisch statt, und zwar, indem der kleinste vorkommende Wert schwarz und der größte vorkommende Wert weiß dargestellt wird.
- "mag./cyan": Wird nur für "fuzzy" benutzt. Magenta- bzw. Cyanfärbung wird für fuzzy(lo) bzw. fuzzy(hi) zur Darstellung benutzt. Stärkere Färbung gibt dabei höhere Werte an. Die Skalierung kann über ein zusätzliches Dialogfenster eingestellt werden.
- "magenta-lo, cyan-hi": Wird nur für "value" benutzt. Für diese Darstellungsmethode muß man in einem Eingabefeld, das nur speziell für diese Darstellungsmethode erscheint, einen scharfen Grenzwert eintragen. Eine Isolinie wird nur für diesen Wert angezeigt. Zusätzlich werden die Flächen, in denen die größtmöglichen Werte kleiner sind als der vorgegebene Wert, magenta, und die Flächen, in denen die kleinstmöglichen Werte größer sind als der vorgegebene Wert, cyan eingefärbt.

Es gibt eine weitere Methode, das Kriging-Resultat darzustellen, und zwar, indem man mit der Maus in der oben beschriebenen zweidimensionalen Darstellung eine orange Linie von Punkt P_1 nach Punkt P_2 zieht, und dann über ein Menü die Funktion zur Darstellung der Werte entlang dieser Linie auswählt. Ist $P_1=P_2$, so wird ein Punktfenster geöffnet, um die Zugehörigkeitsfunktion des unscharfen Wertes an dieser Stelle anzuzeigen. Ansonsten wird ein Schnittfenster geöffnet, um die berechneten α-Schnittintervalle entlang dieser Linie in Form von Kurven für die unteren und oberen Grenzen der Intervalle darzustellen (diese beiden Darstellungstypen wurden schon kurz in Kapitel 3.1 beschrieben).

3.1.5 Datenexport, Ausgabeformate

Die Datenexportfunktionen befinden sich im Menü der Krigingfenster. Es gibt drei solcher Funktionen:

- Eine ASCII-Ausgabe der experimentellen (und theoretischen) Variogrammwerte. Ein Dialogfenster erlaubt, die Attribute auszuwählen, die man benötigt. Da das experimentelle Variogramm nur für die am ehesten möglichen Werte (den scharfen 1-Schnitt) und den 0-Schnitt berechnet werden, ist eine Darstellung von unscharfen Zahlen hier nicht vonnöten. Die Variogrammwerte werden in Form einer Tabelle ausgegeben. Für jedes im Dialogfenster ausgewählte Attribut wird eine Spalte (mit dem Attribut als Beschriftung) erzeugt; je Entfernungsklasse, in der mindestens ein Wert vorkam, wird eine Zeile mit Werten zu den gewählten Attributen erzeugt. Üblicherweise wählt man mindestens folgende Attribute: Eins für den Mittelwert der Abstände in der jeweiligen Klasse, eins für den experimentellen Variogrammwert für diese Klasse und eins für die Anzahl der Werte in der Klasse.
- Eine ASCII-Ausgabe des Kriging-Resultats in Form von Werten für die Mittelpunkte von Zellen eines Rasters, das mit Hilfe einer zusätzlichen ASCII-Eingabedatei (in einem speziell darauf ausgerichteten Format), anstelle mit Hilfe eines Dialogfensters beschrieben wird. Diese Beschreibung macht man in Form einer Liste der Zellenbreiten - einmal in *x*-Richtung und einmal in *y*-Richtung. Die absolute Positionierung findet an der linken unteren Ecke des im "region"-Teil der Eingabedatei festgelegten Rechtecks statt.
  Damit die unscharfen Ausgabedaten möglichst leicht von anderen Programmen einlesbar sind, kann man über ein Dialogfenster auswählen, ob man unscharfe Zahlen (in der Form, die auch in der Eingabedatei erlaubt ist) für jede oben beschriebene Position ausgibt oder ob man die scharfe Zahl für eine untere bzw. obere Grenze eines α-Schnitts der unscharfen Zahl an der jeweiligen Stelle ausgibt. Zusätzlich kann man noch die Angabe von Koordinaten an- und abschalten (falls abgeschaltet, werden die Werte für gleiche *y*-Koordinaten in jeweils einer Zeile dargestellt), und man kann die Sortierung der *y*-Koordinaten (aufsteigend oder absteigend) festlegen.
- Die gerade angezeigte zweidimensionale Darstellung des Krigingresultats im Krigingfenster kann man in beliebiger (oder zumindest nicht durch das Fenster begrenzter) Auflösung als Graphik in die Windows-Zwischenablage kopieren. Von dort aus können z.B. Graphik- und Textverarbeitungsprogramme darauf zugreifen, die dann auch über diesen Weg das Drucken solch einer Graphik möglich machen. Über ein Dialogfenster kann man die Auflösung und Liniendicke der Linien (z.B. der Isolinien) für diese Graphik vorgeben. Die Beschriftungen der Isolinien werden relativ zum Abstand der Isolinien genauso groß gemacht, wie in der Anzeige im Krigingfenster gerade zu sehen ist.

3.2 Datenstrukturen

Die wesentlichen Anforderungen an die Datenstrukturen ergeben sich schon aus den theoretischen Kapiteln. Zunächst ist zu berücksichtigen, daß man regionalisierte unscharfe Variable darstellen muß, d.h. in diesem Fall Listen von unscharfen Zahlen mit Koordinatenangabe. Um dies zu realisieren, muß man insbesondere die Darstellung von unscharfen Zahlen festlegen, was allerdings schon in Kapitel 2.1.4 vorbereitet wurde. In Kapitel 3.2.1 wird dies durchgeführt, und es werden dort auch gleich Listen von unscharfen Zahlen mit Koordinatenangabe implementiert.

In Kapitel 3.2.2 wird eine Verwaltungs-Datenstruktur vorgestellt, die die Struktur der Aggregationsfunktion (inklusive den Transformationsfunktionen) und die interne Struktur des Fuzzy Kriging abbilden kann. In Kapitel 3.3 wird dann gezeigt, wie ein spezieller Teil des Programms (die Managementkomponente) damit umgeht, um immer automatisch alle Daten konsistent zu halten, d.h. wenn beispielsweise der Anwender eine Eingabe ändert, wird die Managementkomponente davon unterrichtet und ergreift alle notwendigen Maßnahmen (löst also z.B. die notwendigen Berechnungen aus, sorgt für das Abspeichern in Dateien, etc.).

In Kapitel 3.2.3 werden dann noch weitere Datenstrukturen aufgezählt, die relativ wichtig sind.

Zum Schluß wird in Kapitel 3.2.4 noch die Speichersegmentierung von Windows 3.1 diskutiert, da sie Auswirkungen auf Details der Datenstrukturen hat.

Grundsätzlich ist zur Implementierung zu bemerken, daß alle in diesem Kapitel beschriebenen Datenstrukturen in C definiert wurden, obwohl C++ die Implementierungssprache ist. Dies ist natürlich kein Problem, da die Sprache C (im Großen und Ganzen) eine Untermenge von der Sprache C++ darstellt. Die objektorientierten Möglichkeiten von C++ wurden nur zur Programmierung der Oberfläche genutzt, weil dafür eine relativ gute Objektbibliothek zur Verfügung stand. Die in C definierten Datenstrukturen werden in sehr einfache und transparente Strukturen vom Compiler übersetzt (dies ist eine Eigenschaft, die sogar zur Definition der Sprache C gehört). Dies wurde genutzt, um einige Teile der Datenstrukturen besonders einfach als Dateien abspeichern bzw. lesen zu können.

Damit die Managementkomponente bequem mit verschiedenen Datenstrukturen umgehen kann, und damit das Abspeichern des gesamten Zustandes möglichst einfach ist, haben alle wichtigen Datenstrukturen einen Vorspann, der Verwaltungsdaten enthält. Dieser Vorspann ist folgendermaßen definiert:

```
typedef struct
{ int4      length;          // Größe im Speicher - für das Abspeichern
  int4      arraylength;     // -1, bzw. Länge eines Arrays (falls vorhanden)
  npoint    number;          // Referenzen laufen über diese Nummern
  HGLOBAL   handle;          // Für Speicherverwaltung notwendig
  HGLOBAL   handle_checksum; // (Prüfsumme, dient der Speicherverwaltung)
  boolean   tosave;          // Ob abzuspeichern (für Managementkomponente)
  boolean   saved;           // Ob Abgespeichertes aktuell ( " )
  thing_type type;           // Eigentlicher Typ der Datenstruktur, von dem
} thing_header;              //                          dies der Vorspann ist
```

Bemerkungen dazu:

"int4" ist ein Ganzzahldatentyp mit 4 Byte Darstellungslänge.

"arraylength" bezieht sich auf das Array in der Datenstruktur, das eine zur Laufzeit bestimmbare Länge hat (falls vorhanden). Davon kann man in C eins pro Datenstruktur benutzen, wenn man es an das Ende dieser Struktur legt und die Speicherverwaltung selbst übernimmt (d.h. man muß dann die jeweils notwendige Menge von Speicher bestimmen und vom System anfordern, um eine Instanz zu erzeugen).

"number" stellt eine interne Referenz der aktuellen Instanz der entsprechenden Datenstruktur dar; diese Referenz ist nicht als Zeiger implementiert (sondern als ganze Zahl - der Datentyp "npoint" ist mit Hilfe der folgenden Definition festgelegt: "typedef int2 npoint;"). Dies hat den Vorteil, daß man die Referenzen zwischen den Instanzen der Datenstrukturen selbst verwalten kann und bei einem neuen Programmlauf wieder die gleichen Referenzen verwendbar sind, was z.B. beim Abspeichern und Wiedereinlesen des aktuellen Zustands nützlich ist.

Ein "handle" wird für die Windows 3.1-Speicherverwaltungsfunktionen benötigt. Ohne dieses Handle (was nicht identisch mit dem Zeiger auf den Anfang des Speicherblocks ist) könnte man einen Speicherblock nicht wieder freigeben.

"tosave" und "saved" dienen zur Unterstützung der Methode, mit der der aktuelle Zusand abgespeichert werden kann. Das Konzept ist dazu folgendes: Der Anwender gibt ein Verzeichnis an, in dem das Programm seine Zustandsdaten fortan speichern darf. Das Programm hält die Dateien in diesem Verzeichnis dann automatisch (mit einem bestimmten Zeitversatz in manchen Fällen) auf dem neuesten Stand, so daß der Anwender dann keine Anweisung zum Speichern mehr geben muß. "tosave" zeigt, ob die betreffende Instanz überhaupt abgespeichert werden muß, und "saved" zeigt, ob die betreffende Instanz schon abgespeichert wurde.

3.2.1 Schnittdarstellung von unscharfen Zahlen, regionalisierte Variable

In Kapitel 2.1.4 wurden schon grundsätzlich verschiedene Methoden diskutiert, die der Implementierung von unscharfen Zahlen dienen können. Für FUZZEKS soll die Methode benutzt werden, α-Schnitte, also Intervalle abzuspeichern. Der Anwender legt, wie in Kapitel 3.1.1 beschrieben, die α-Werte, für die Schnitte gespeichert (und berechnet) werden sollen, im "cuts"-Teil der Eingabedatei fest.

Aus praktischen Gründen kann man nicht unendlich viele Schnitte abspeichern. Tatsächlich ist man aber auch vor allem an der Repräsentation bestimmter Merkmale der Zugehörigkeitsfunktion interessiert. Eins davon wird bei einer endlichen Liste von α-Schnitten aber nicht ausreichend berücksichtigt, nämlich der Wert, bei dem die Zugehörigkeitsfunktion eins ist. Dieses Merkmal kann man mit Hilfe des scharfen 1-Schnitts repräsentieren.

Da Satz 1 aus Kapitel 2.1.4 auch für den scharfen 1-Schnitt gilt, was im Beweis zu diesem Satz bemerkt wurde, sind alle Verknüpfungen für Schnitte auch für diesen scharfen Schnitt anwendbar. Deshalb wird der scharfe 1-Schnitt zu der Liste von Schnitten hinzugenommen.

Auch der 0-Schnitt wurde, da er ein ganz wesentliches Merkmal der Zugehörigkeitsfunktion darstellt (er wird auch *Träger* genannt), vorgeschrieben.

Zusammenfassung: In FUZZEKS werden unscharfe Zahlen mit Hilfe von einer endlichen Menge von α-Schnitten für $\alpha \in [0,1)$, wobei $\alpha=0$ vorkommen *muß*, und dem scharfen 1-Schnitt näherungsweise beschreiben.

Da die Schnitte laut Satz 1 aus Kapitel 2.1.2 Intervalle sind, liegt es nahe, zur Darstellung der Intervalle die Grenzen der Intervalle zu speichern.

Verloren gehen bei dieser Methode erstens die Möglichkeit, unbeschränkte Intervalle zu repräsentieren, und zweitens die Möglichkeit zwischen offenen und abgeschlossenen Intervallen zu unterscheiden. Letzteres ist, wie am Ende von Kapitel 2.1.4 schon erläutert, nicht mehr von besonderem Interesse, wenn man es im Lichte von Darstellungsungenauigkeiten betrachtet, denn diese sind viel größer als die Unterschiede zwischen offenen und abgeschlossenen Intervallen.

Unbeschränkte Intervalle könnte man trotzdem darstellen, wenn man zu den reellen Wertebereichen für die Grenzen der Intervalle noch zusätzlich die Werte $-\infty$ und $+\infty$ hinzunehmen würde. Da dies jedoch einen zusätzlichen Aufwand bei jeder Verknüpfung darstellt und sich eine besondere Notwendigkeit dafür nicht gezeigt hat, wurde dies in FUZZEKS nicht implementiert.

Auf theoretischer Seite erwies es sich als günstig, die Intervalle immer als abgeschlossene Intervalle zu betrachten, da alle hier zu implementierenden Funktionen abgeschlossene Intervalle auf abgeschlossene Intervalle abbilden und diese Abbildungen, wie in den Theorie-Kapiteln vorbereitet, einfach zu implementieren sind. Nur für die Darstellung von unbeschränkten Intervallen wäre die Repräsentation offener Intervalle von Bedeutung. In FUZZEKS wurde nur die Repräsentation von abgeschlossenen Intervallen implementiert.

Leere Intervalle können übrigens als Schnitte bei obiger Darstellung von unscharfen Zahlen nicht vorkommen, da die Zugehörigkeitsfunktion von unscharfen Zahlen immer an genau einer Stelle den Wert 1 annimmt.

Bemerkung: Da laut Kapitel 2.1.1 scharfe Zahlen als Spezialfall von unscharfen Zahlen betrachtet werden können, wurde keine Ausnahme für den Fall von scharfen Zahlen implementiert.

Bei Ausgaben des Programms schlägt sich die Wahl der Schnitte (die im "cuts"-Teil der Eingabedatei vorgenommen wurde) natürlich nieder; sehr wenige Schnitte zu benutzen, führt zu einer relativ ungenauen Bestimmung der Zugehörigkeitsfunktion der auszugebenden unscharfen Zahl. Sehr viele Schnitte zu benutzen, verlängert dagegen die Rechenzeiten.

Die ausgewählte Liste von Schnitten muß natürlich auch mit Hilfe einer Datenstruktur repräsentiert werden. Die C-Strukturen dafür lauten bei FUZZEKS folgendermaßen (zunächst wird die Beschreibung einzelner Schnitte durchgeführt, dann ein Array aus solchen Schnittbeschreibungen gebildet):

```
typedef struct
{ t_real          height;   // Höhe des Schnitts im Bereich von 0.0 bis 1.0
} t_cutdescr;

typedef struct
{ thing_header  header;
  t_cutdescr    descr[];  // Array von Schnitthöhen (mit zur Laufzeit bestimmter
                          //                                Länge) , absteigend sortiert
} t_cuts;
```

Nun folgen die eigentlichen Datenstrukturen für die Darstellung von unscharfen Zahlen. Da überall im Programm Arrays von unscharfen Zahlen mit Koordinatenangabe benötigt werden, folgen auch die dafür notwendigen Ergänzungen.

Zunächst soll eine Datenstruktur für Koordinaten definiert werden:

```
typedef struct
{ t_real   x,y;
} t_coord;
```

Des weiteren benötigt man die Darstellung eines Schnittes, was laut obigen Betrachtungen hier auf je eine reelle Zahl für die linke (untere) und rechte (obere) Grenze hinausläuft:

```
typedef struct
{ t_real   l,r;
} t_cut;
```

Eine unscharfe Zahl mit Koordinatenangabe wird dann mit Hilfe folgender Datenstruktur repräsentiert:

```
typedef struct
{ t_coord    coord;      // Die Koordinatenangabe
  boolean    undef;      // Erlaubt die Information "Wert unbekannt" darzustellen
  t_cut      cut[];      // Das Array von Schnittbeschreibungen (Die Länge des
                         //                          Arrays wird zur Laufzeit festgelegt)
} t_fuzzy_nc;
```

Folgende Datenstruktur ergibt das Array von unscharfen Zahlen mit Koordinatenangabe. Zusätzlich können noch einige nützliche Randinformationen gespeichert werden.

```
typedef struct
{ thing_header  header;
  int4          defined_length;  // Anzahl der nicht undefinierten Werte
  int           line_length;     // Länge der Zeilen, falls 2-dimensionale Struktur;
                                 //              0 bedeutet 1-dimensionale Struktur
  int           transformation;  // Ob und wie die Daten transformiert sind
  int           interpretation;  // (Dient ggfs. dem Verteilen des Ergebnisses)
  npoint        cutinfo;         // Verweis auf das zugehörige Array von Schnitthöhen
  int2          low_address;     // (Intern, zur Speicherverwaltung)
  npoint        next;            // X    (Die drei mit X gekennzeichneten Einträge
  t_real        weight;          // X    dienen nur dem Zweck, in manchen Situationen
  int           index;           // X    bequem Argumentlisten bilden zu können)
  p_fuzzy_nc    fuzzy_nc[];      // Das Array unscharfer Zahlen mit Koordinatenangabe
} t_fuzzyarray;
```

"line_length" erlaubt, das Array als zweidimensionales Array zu betrachten, wobei "line_length" dann die eine Dimension angibt (die andere ergibt sich aus header.arraylength / line_length). Solche zweidimensionalen Arrays werden für das Kriging-Ergebnis benötigt, das i.a. ein zweidimensionales Array von geschätzten Werten ist.

"transformation" gibt an, ob die Daten logarithmisch transformiert wurden (siehe auch Kapitel 2.3.1.4 und 2.3.2.4). Daraus ergibt sich dann bei Ausgabe der Daten die Forderung zur Rücktransformation.

"cutinfo" enthält die Referenz auf die weiter oben gezeigte Datenstruktur, die die Schnitthöhen beschreibt (t_cuts).

"fuzzy_nc" schließlich ist ein array von C-Zeigern auf Instanzen vom Typ t_fuzzy_nc; die Länge dieses Arrays wird zur Laufzeit festgelegt. Die Funktionen zur grundlegenden Behandlung von "t_fuzzyarray" sorgen dafür, daß passende Zeiger eingetragen werden, nämlich Zeiger auf den Bereich nach dem "fuzzy_nc"-Array, so daß das gesamte Array von unscharfen Zahlen mit Koordinatenangabe (bis auf die Schnitthöhen) einen zusammenhängenden Datenblock ergibt. Es wäre nicht möglich gewesen, ein Array vom Typ "t_fuzzy_nc ...[]" zu benutzen, denn dies würde bedeuten, daß die Länge der einzelnen Arrayelemente zur Übersetzungszeit nicht bekannt ist, so daß der Compiler den Code dafür nicht erzeugen könnte.
Die Implementierung dieses Arrays ist bzgl. der Effizienz (Geschwindigkeit) bei der Nutzung der Datenstruktur relativ günstig, denn zum Auffinden eines Arrayelements brauchen keine arithmetischen Operationen mehr durchgeführt werden, da die Adresse jedes Arrayelements unmittelbar gegeben ist.
Im Kapitel 3.2.4 (Speichersegmentierung) werden zusätzliche Probleme im Umgang mit diesem Array unter Windows 3.1 und deren Lösung gezeigt.

3.2.2 Verwaltungsstrukturen

Die Strukturen, die verwaltet werden sollen, sind sowohl die Struktur der Aggregationsfunktion (inklusive der Transformationsfunktionen) als auch die interne Struktur des Fuzzy Kriging. Begleitend zu diesen Strukturen treten noch Benutzereinstellungen, Zwischenergebnisse und Ergebnisse auf.

Ein spezieller Teil des Programms, die Managementkomponente, benutzt die folgenden Datenstrukturen, um die ebengenannten Strukturen zu repräsentieren, und um immer automatisch alle Daten konsistent zu halten.

Im einzelnen läuft das folgendermaßen ab: Wenn der Anwender über die Oberfläche ein Datum ändert, dann teilen die Oberflächenfunktionen die Änderungen der Managementkomponente mit. Diese berücksichtigt alle Änderungen, die sich daraus ergeben (insbesondere werden Zwischenergebnisse und Ergebnisse, die nun nicht mehr aktuell sind, gelöscht, was wiederum den entsprechenden Fenstern mitgeteilt wird, die dies dann bei der Darstellung berücksichtigen können). Außerdem veranlaßt sie die Ausführung entsprechender Berechnungsalgorithmen, die dann im Hintergrund ablaufen und ihre Ergebnisse der Managementkomponente zurückgeben. Die Ergebnisse werden dann, soweit dafür relevant, an die entsprechenden Fenster zur Darstellung übermittelt.

Zusätzlich zu obigen Aufgaben veranlaßt die Managementkomponente auch das Abspeichern in Dateien, wenn dies nötig wird (das Konzept dazu wurde am Ende von Kapitel 3.2 schon kurz beschrieben).

Das Konzept, die Verwaltungsaufgaben einer Managementkomponente zu übertragen, hat den Vorteil, daß sich eine Modularisierung des Programms auf ganz natürliche Art und Weise ergibt.

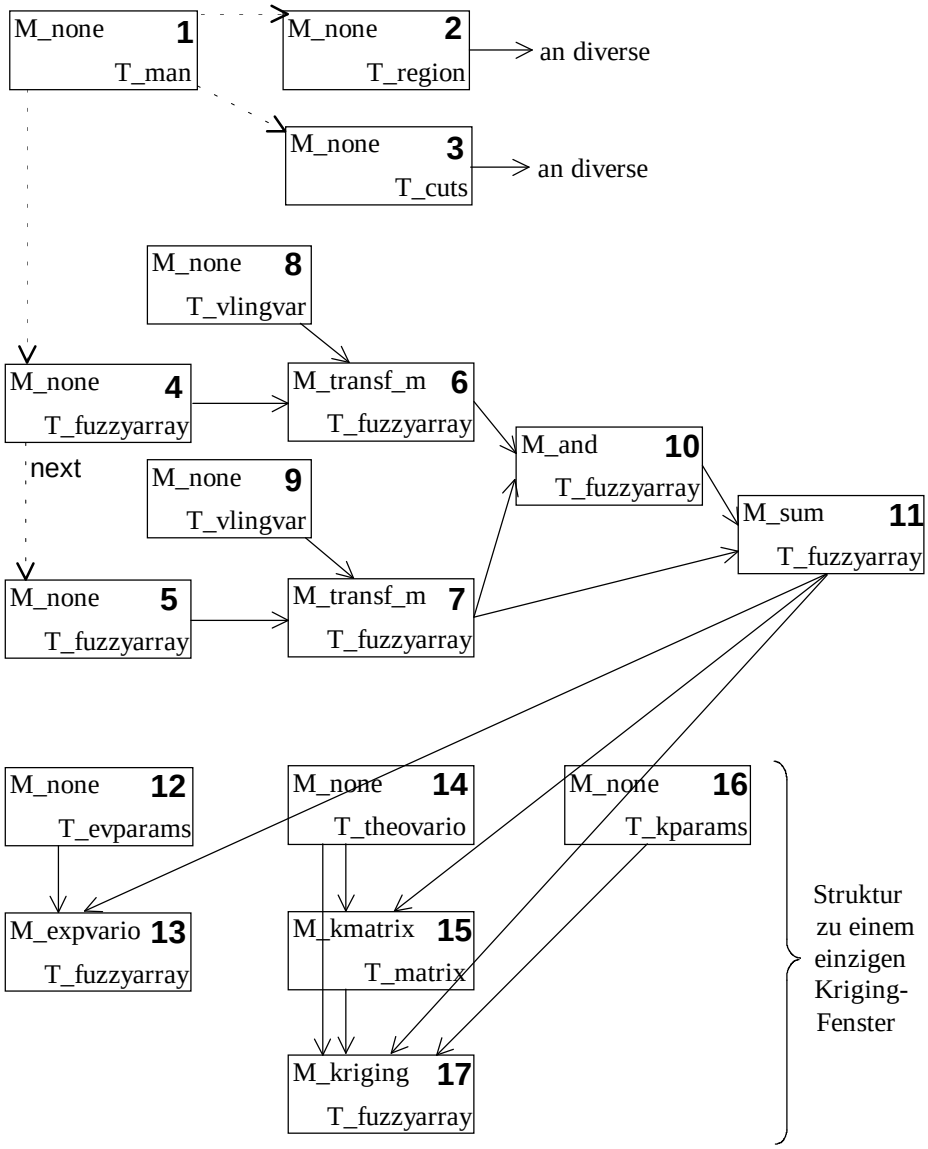


Abbildung 3.2.2: Beispiel einer Gesamtstruktur
(In der linken oberen Ecke ist jeweils die Methode verzeichnet, die zur Berechnung der Daten zu diesem Knoten dient, in der rechten unteren Ecke ist der Typ der Daten - sieheKapitel 3.2.1 und 3.2.3 - verzeichnet, die an solch einem Knoten hängen.)

Zunächst einmal sollen hier die Strukturen, die in FUZZEKS verwaltet werden, etwas genauer umrissen werden. Man betrachte dazu die als Graph dargestellte Beispielstruktur aus Abbildung 3.2.2. Knoten 1 dient als globaler Aufhänger für die gesamte Struktur. Knoten 4 und 5 entsprechen den Parametern aus der Eingabedatei. Die Knoten 6 bis 11 entsprechen einer Aggregationsfunktion (inkl. der Transformationsfunktionen 6 und 7). Die Aggregationsoperatoren sind dabei die Knoten 10 (AND) und 11 (SUM). Knoten 4 bis 7 und 10 bis 11 entsprechen übrigens den Symbolen aus dem Managementfenster. Knoten 8 und 9 entsprechen einem Zugehörigkeitsfunktionsfenster.

Knoten 12 bis 17 dienen einem Krigingfenster zur Darstellung des Aggregationsresultats von Knoten 11. Auf der einen Seite muß dazu das experimentelle Variogramm berechnet werden (Knoten 12 und 13); auf der anderen Seite ist das Krigingresultat zu berechnen (Knoten 17), was auf der Grundlage des theoretischen Variogramms (Knoten 14) und über den Zwischenschritt, eine Matrix zu berechnen, geschieht (nämlich die invertierte Matrix von der aus Kapitel 2.3.1.2, Knoten 15).

Im oben dargestellten Beispiel kann man schon zum Teil die tatsächliche Implementierung der Struktur erkennen. Sie basiert auf einer Datenstruktur für die Knoten der Struktur (siehe Beispielabbildung). Diese ist folgendermaßen definiert:

```
typedef struct
{ npoint        in;             // Referenz auf Knoten, der am Eingang hängt
  t_real        in_weight;      // Gewicht des Eingangs im Fall eines SUM-Knotens
  npoint        out;            // Referenz auf Knoten, der am Ausgang hängt
} t_npoint_in_and_out;

typedef struct
{ thing_header  header;
  int           method;     // Die Methode, die zur Berechnung benutzt wird
  boolean       recalc;     // Ob die Informationen neu berechnet werden müssen
  boolean       inactive;   // Ob Berechnungen nicht notwendig sind
  int           temp;       // Ob der Knoten nach erfolgreicher Berechnung
                            // gelöscht werden soll
  int           prio;       // Wichtigkeit des Knotens, beeinflußt Joblistenbildung
  npoint        next;       // Für Listenbildung (der Eingabeparameter)
  boolean       visited;    // (Flag für Joblisten-Algorithmen)
  npoint        nextjob;    // Verkettung für Jobliste
  boolean       calc_in_progress; // Zeigt, ob dieser Knoten gerade neu berechnet wird
  int           type;       // Typ der Daten
  npoint        data;       // Die Daten, für die dieser Knoten steht
  npoint        data2;      // zusätzliche Daten wie etwa Varianz
  CNodeWindow   *window;    // 1. assoziiertes Fenster (Krigingfenster)
  CNodeWindow   *window2;   // 2. assoziiertes Fenster (Zugehörigkeitsfunktionsfenster)
  npoint        next_with_out0; // Ergibt Liste von Knoten ohne Nachfolger
  int           x,y;        // Position, falls Aggregationsoperator
  char          comment[MAX_COMMENT+1]; // Kommentar zum Knoten
  boolean       sum_is_1;
  int                 in,out; // Anzahl der Ein- bzw. Ausgänge
  t_npoint_in_and_out net[];  // Die Referenzen auf die Ein- und Ausgänge
} t_node;
```

Die wichtigen Teile, die schon in der Beispielabbildung zu sehen waren, sind fett gedruckt. Die eigentlichen Daten zu dem Knoten sind (falls vorhanden, weil z.B. schon berechnet) jeweils über die Referenz "data" zu erreichen. Die Methode zur Berechnung dieser Daten wird bei "method" vermerkt; falls es sich um Benutzereingaben oder Parameter aus der Eingabedatei handelt, wird keine Methode vermerkt ("M_none").

Sehr wichtig sind die Verbindungen (Kanten) zwischen den Knoten, die den Datenfluß bestimmen; sie sind in der Beispielabbildung als durchgezogene Pfeile dargestellt. Damit die Managementkomponente alle Abhängigkeiten und die Berechnungsreihenfolgen effizient ermitteln kann, sind alle diese Verbindungen als doppelte Verkettungen (Verweise in beide Richtungen) ausgeführt. Deshalb muß die Datenstruktur für die Knoten beliebig viele Ein- und Ausgänge erlauben. Dazu wurde wieder die Technik verwendet, mit deren Hilfe man Arrays mit zur Laufzeit bestimmter Länge realisieren kann. Diese Technik erlaubt jedoch nur *ein* Array dieser Art pro C-struct. Deshalb wurde ein Array von Ein- *und* Ausgängen verwandt ("net"). Die Einträge "in" und "out" (von "t_node") geben an, wieviel Ein- bzw. Ausgänge tatsächlich im Array zu finden sind. (Die Länge des Arrays selbst steht wie in allen vorangehenden Datenstrukturen in "header.arraylength".) Es wurden Funktionen zur Verwaltung dieser Strukturen implementiert, die diese Details beim Benutzen der Datenstruktur verstecken.

Die meisten anderen Einträge dienen der Managementkomponente zur Erfüllung der weiter oben erwähnten Aufgaben. Auch Informationen zur Darstellung von Knoten im Managementfenster (insbesondere ihre Position) sind hier zu finden.

3.2.3 Sonstige Datenstrukturen

Wie im vorigen Kapitel in der Beispielabbildung schon zu sehen war, gibt es noch weitere Datenstrukturen in FUZZEKS, die noch nicht genauer vorgestellt wurden. Dazu gehört u.a. auch der Datentyp "t_matrix", der technisch gesehen im Prinzip genauso wie bei "t_fuzzyarray" ein Array von Arrays implementiert; deswegen soll hier nicht noch einmal diese Technik erklärt werden. Laut folgender Definition ist "t_matrix" ein Array (der Länge "header.arraylength") von (Zeigern auf) Vektoren von reellen Zahlen (der Länge "elements"). "p_vector" ist dabei als Zeiger auf Instanzen von "t_vector" definiert.

```
typedef struct
{ t_real        element[];      // Ein Array von reellen Matrixelementen
} t_vector;

typedef struct
{ thing_header  header;
  int           elements;       // Größe der Matrix (zusammen mit header.arraylength)
  int2          low_address;    // (Technisches Hilfsmittel zur Speicherverwaltung)
  p_vector      vector[];       // Das Array von reellen Vektoren
} t_matrix;
```

Datenstrukturen vom Typ "t_man" enthalten nur einige organisatorische Angaben für die Managementkomponente, die auch abgespeichert werden müssen. Dazu gehören insbesondere der Aufhänger für die Liste der Eingabeparameter (siehe auch "nin_root" in der Beispielabbildung aus dem letzten Kapitel).

Die Datenstruktur vom Typ "t_region" enthält die Informationen aus dem "region"-Teil der Eingabedatei.

Andere wesentliche Datenstrukturen, deren Definition nicht angegeben wurde, beschreiben Benutzereingaben:

- "t_vlingvar": Die Zugehörigkeitsfunktionen (zu jeweils einem Wert einer linguistischen Variablen), die als Transformationsfunktionen genutzt werden.
- "t_evparams": Parameter zur Berechnung experimenteller Variogramme.
- "t_theovario": Theoretische Variogramme.
- "t_kparams": Parameter zur Berechnung des Kriging-Resultats; insbesondere die Anordnung der Punkte, für die jeweils ein Wert interpoliert werden soll.

3.2.4 Die Speichersegmentierung von Windows 3.1

Unter Windows 3.1 ist der Speicher in Segmente aufgeteilt, deren maximale Größe 64 KByte beträgt. Dies bedeutet, wenn man auf eine Adresse aus dem gesamten Speicher zugreifen möchte, muß man eine Segmentnummer in ein geeignetes Segmentregister laden und kann dann über eine 16-Bit-Adresse auf die Daten zugreifen. Um Arrays zu implementieren, die größer sind als 64 KByte, ist ein Compiler gezwungen, eine sehr aufwendige Adreßarithmetik zu implementieren, die leicht eine (ca.) Verfünffachung der Programmlaufzeit bewirken kann. Der Grund für den enormen Zeitaufwand liegt hauptsächlich darin, daß die Segmentnummern für aufeinanderfolgende Segmente nicht fortlaufend sind, sondern sich um jeweils 8 erhöhen.

Um nicht immer diese langsame Adreßarithmetik benutzen zu müssen, wurden für Windows 3.1 von den Compilerherstellern Erweiterungen zur Sprache C definiert, nämlich Modifizierer der Zeigerdatentypen. Unter Windows gibt es folgende Zeigertyp-Varianten:

- "near": 16-Bit Zeiger in festgelegte, maximal 64 KByte große, Segmente.
- "far": 32-Bit Zeiger (16 Bit Segmentnummer und 16 Bit Adresse). Arithmetik ist nur auf den unteren 16 Bit, also nur innerhalb eines 64 KByte-Segments, möglich.
- "huge": 32-Bit Zeiger (16 Bit Segmentnummer und 16 Bit Adresse). Arithmetik ist auf allen 32 Bit möglich, jedoch, wie oben beschrieben, sehr zeitaufwendig.

Beispiel: Anstelle der Deklaration eines Zeigers auf int "int *ip;" muß man also "int huge *ip;" schreiben, wenn man im Umgang mit dem Zeiger "ip" 32-Bit-Adreßarithmetik benötigt.

Wenn ein Programm unter Windows 3.1 also nicht extrem langsam sein soll, muß man "huge"-Zeiger vermeiden. Bei den Arrays für "t_fuzzyarray" und "t_matrix" stellt das ein Problem dar, da diese leicht größer als 64 KByte werden können.

Schon aus anderen Gründen (nämlich um Arrays von zur Laufzeit bestimmter Länge zu erhalten) wurde bei "t_fuzzyarray" und "t_matrix" anstelle eines Arrays von Elementen ein Array von Zeigern auf die Elemente implementiert. Normalerweise würde man die Elemente, auf die gezeigt werden soll, im Speicher jeweils hintereinander ablegen, was aber bedeutet, daß manche Elemente auf Segmentgrenzen liegen könnten. Innerhalb dieser Elemente könnte man also Zugriffe nur mit Hilfe von "huge"-Zeigern machen (wobei man aber sogar noch selbst dafür Sorge tragen muß, daß z.B. die 8-Byte-Blöcke, die zur Darstellung einer reellen Zahl (im double-Format) dienen, niemals über den Segmentgrenzen liegen, da der Prozessor, der diese 8-Byte-Blöcke in einem Befehl lädt, die Segmentierung *nicht* berücksichtigen kann).

Damit man innerhalb der einzelnen Elemente mit "far"-Zeigern zugreifen kann, kann man die Elemente anders im Speicher anordnen, falls sie höchstens 64 KByte groß sind. Falls die Elemente höchstens 32 KByte groß sind, verschwendet man dabei allerdings maximal ca. $(\frac{E}{64\text{K}-E} \cdot 100)$ % Speicher (E sei die Größe der Elemente in Bytes). Bei den Matrizen, die aus anderen Gründen (s.u.) nur eine Größe von 1300*1300 reellen Elementen (8 Byte pro Element) haben können, ergibt sich also eine maximal mögliche Speicherverschwendung von ca. 19 %. Um derartig viel Speicher bei den Instanzen vom Typ "t_fuzzyarray" zu verschwenden, müßte man ca. 650 α-Schnitte zur Darstellung der unscharfen Zahlen verlangen (was aber unwahrscheinlich ist). Da diese Speicherverschwendung i.a. leichter zu tolerieren ist, als eine Verfünffachung der Rechenzeit, wurde diese Methode gewählt.

Bemerkung: Die Begrenzung auf ca. 1300*1300 Elemente bei den Matrizen ist auf eine weitere Speicher-Problematik von Windows zurückzuführen: Man erhält nämlich immer nur maximal 16 MByte Speicher vom System in einem Stück. Obige Lösung des Speichersegmentierungs-Problems hat diese Begrenzung natürlich noch verschlimmert. Auf die Arbeit, mehrere solche 16 MByte-Stücke zusammenzusetzen, wurde bei FUZZEKS verzichtet, da bis jetzt bei keiner Anwendung 1300 Eingabewerte pro Parameter vorkamen. Wenn dies geschehen würde, müßte man sich ohnehin mit sehr langen Rechenzeiten für das Invertieren der Matrix und für die Durchführung vom Rest des Fuzzy Kriging-Verfahrens abfinden. Für solche Anwendungen sollte man deshalb das Kriging-Verfahren ändern (es macht i.a. keinen großen Unterschied, ob man alle bekannten Werte zur Schätzung des Wertes an einer Stelle heranzieht, oder ob man nur die etwa 25 nächsten Punkte benutzt).

3.3 Programmstruktur und Algorithmen

Kapitel 3.1 und Kapitel 3.2 legen schon eine grobe Struktur nahe. Diese besteht (siehe auch Abbildung 3.3) in der Trennung zwischen erstens einer Benutzeroberfläche, zweitens einer Managementkomponente zur Verwaltung und Organisation, und drittens den Berechnungsalgorithmen, die die zum Teil unscharfen Funktionen berechnen, die in dem Theorie-Teil angegeben wurden. Außerdem ist das Abspeichern der Dateien so aufwendig, daß dafür eine eigene Komponente angebracht erscheint. Hinzu kommt noch eine Ebene niedrigeren Abstraktionsniveaus, die aus diversen grundlegenden Funktionen zur Nutzung der Datenstrukturen besteht. Es existieren natürlich Kommunikationswege zwischen der Managementkomponente und den anderen Programmteilen (dargestellt durch die Doppelpfeile in der Abbildung).

Der Programmteil zur Implementierung der Benutzeroberfläche untergliedert sich auf natürliche Art und Weise in Programmteile für jeden einzelnen Fenstertyp. Bei den Berechnungsalgorithmen ergibt sich eine Untergliederung in die Aufgaben, die sich aus den theoretischen Kapiteln ableiten, also grob in Algorithmen, die der Aggregation bzw. dem Kriging dienen. In der Abbildung sind die Fenster und Algorithmen mit Hilfe der gestrichelten Linien grob nach Aggregation bzw. Kriging untergliedert.

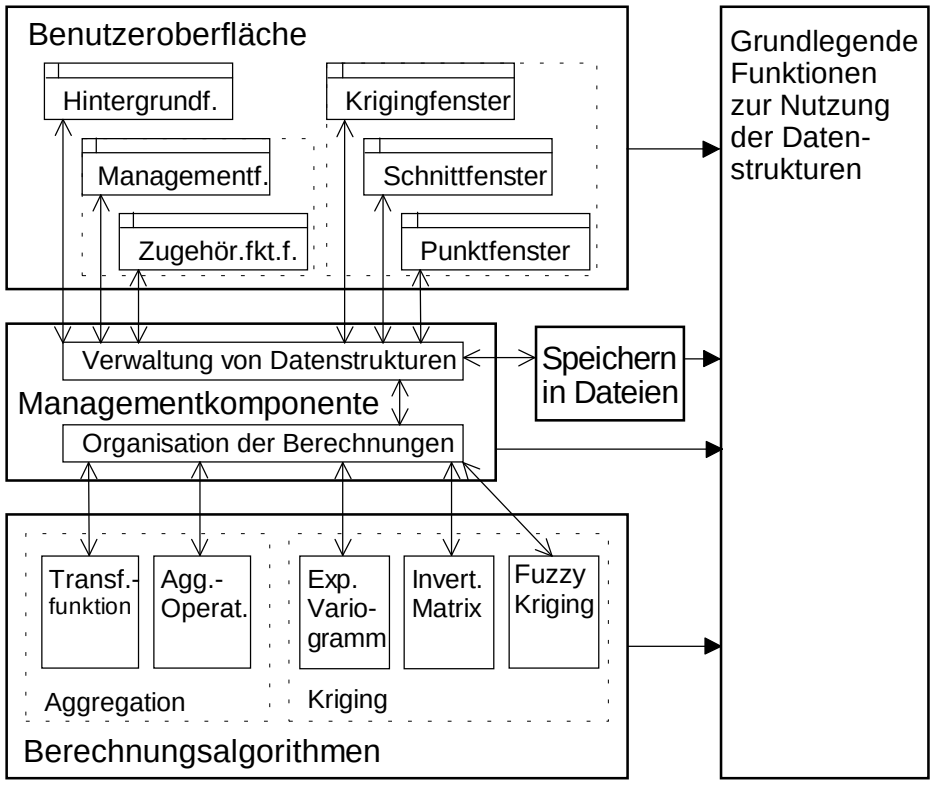


Abbildung 3.3: Programmstruktur von FUZZEKS

Wie schon in Kapitel 3.2 erwähnt, wurde die Implementierung der Oberfläche in C++ und die Implementierung des Restes in C geschrieben. Als Compiler wurde der Microsoft Visual C/C++ Compiler in der Version 1.5 eingesetzt. Für die Oberflächenprogrammierung wurde die zum Compiler mitgelieferte Klassenbibliothek MFC ("Microsoft Foundation Classes") benutzt.

Die Kommunikationswege wurden durch (teilweise) gegenseitige Funktionsaufrufe implementiert, d.h. ein Fenster ruft beispielsweise eine entsprechende Funktion der Managementkomponente auf, um ihr eine Veränderung mitzuteilen; die Managementkomponente ihrerseits ruft die Methode eines Fensters auf, um ihm neuere Informationen mitzuteilen. In ganz wenigen Fällen werden auch globale Variablen zur Kommunikation eingesetzt.

Obwohl unter Windows 3.1 kein Multitasking vorgesehen ist, wurde ein quasigleichzeitiger Ablauf von jeweils einem Berechnungsalgorithmus und dem Rest des Programms (und den anderen Windows-Programmen) implementiert, was in Kapitel 3.3.5 dargestellt wird. Dort zeigt sich auch, wie die Kommunikation mit Hilfe von Funktionsaufrufen dabei berücksichtigt werden muß.

Kapitel 3.3.1 bis 3.3.4 behandeln jeweils das, was als ein grober Block in Abbildung 3.3 dargestellt wurde, außer den grundlegenden Funktionen zur Nutzung der Datenstrukturen. Diese implementieren nur relativ triviale Details, wie z.B. Anforderungen von Instanzen der Datenstrukturen (was auf Anforderungen von Speicher vom Betriebssystem abgebildet wird), das entsprechende Wiederfreigeben, und grundlegende Zugriffsmechanismen für die Datenstrukturen (z.B. die Verwaltung der Verbindungen zwischen Knoten; dafür gibt es u.a. Funktionen zum Anlegen und Löschen der Verbindungen, die intern als doppelte Verkettungen implementiert sind).

3.3.1 Die Managementkomponente

Wie schon in Abbildung 3.3 zu erkennen ist, spielt in FUZZEKS die Managementkomponente eine zentrale Rolle. Außerdem kann man die grobe Einteilung in die Verwaltung von Datenstrukturen (genauer gesagt von Instanzen der in Kapitel 3.2 angegebenen Datenstrukturen) und die Organisation der eigentlichen Berechnungen sehen.

Um die Aufgaben der Managementkomponente nachzuvollziehen, muß man das Zusammenspiel der Managementkomponente mit den restlichen Programmteilen betrachten. Dies soll anhand von typischen Beispielabläufen der Ereignisse gezeigt werden:

- Angenommen, der Anwender klickt mit der Maus in dem Managementfenster auf ein Symbol zum Öffnen eines Krigingfensters. Dies teilt das Managementfenster der Managementkomponente mit, woraufhin diese die passenden Datenstrukturen für ein Krigingfenster (siehe Abbildung 3.2.2) und eine Instanz der Krigingfensterklasse anlegt. Außerdem müssen evtl. Neuberechnungen vorgenommen werden; siehe dazu aber das nächste Beispiel.

- Angenommen, der Anwender verändert das theoretische Variogramm in einem Krigingfenster. Das Krigingfenster teilt der Managementkomponente das neue theoretische Variogramm mit, woraufhin die Managementkomponente es an den passenden Knoten hängt (siehe Abbildung 3.2.2, dort kam dafür nur Knoten 14 in Frage). In Abbildung 3.2.2 sind die Abhängigkeiten zwischen den Daten mit (durchgezogenen) Pfeilen dargestellt. Aufgrund der Abhängigkeiten (auch der mittelbaren) müssen jetzt also die alte Matrix und das alte Kriging-Ergebnis gelöscht werden (falls vorhanden); dies wird natürlich auch den entsprechenden Fenstern (in diesem Fall nur demselben Krigingfenster) mitgeteilt, damit dies die Darstellungen, die jetzt nicht mehr aktuell sind (in diesem Fall die Darstellung des Kriging-Ergebnisses) löschen kann.
  Nun ergeben sich auch neue Notwendigkeiten für Berechnungen, und die Komponente zur Organisation der Berechnungen kommt ins Spiel. Diese stellt eine neue Liste von noch notwendigen Berechnungen (in passender Reihenfolge) auf und startet den ersten Berechnungsalgorithmus in dieser Liste mit den passenden Argumenten (dies wäre in diesem Beispiel, falls zuvor keine Berechnungen notwendig waren, der Algorithmus zur Berechnung der invertierten Matrix von jener aus Kapitel 2.3.1.2).
  Wenn der Algorithmus terminiert, liefert er sein Ergebnis an die Managementkomponente zurück, woraufhin diese gegebenenfalls das Ergebnis einem Fenster mitteilt, es abspeichert und den nächsten Berechnungsalgorithmus startet.

Aus den Beispielen ergeben sich schon die wesentlichen Aufgaben der Managementkomponente:

- Anlegen oder Löschen von Datenstrukturen; insbesondere der Struktur, die mit den Knoten, die in Kapitel 3.2.2 dargestellt sind, beschrieben wird. Zusätzlich werden gegebenenfalls auch mit Vorgabewerten gefüllte Instanzen von Strukturen, die z.B. Benutzereingaben repräsentieren, angelegt und an der passenden Stelle an Knoten angehängt (das "data"-Element der "t_node"-Struktur ist der Aufhänger).
- Mitteilungen über die Veränderung von Daten annehmen und auswerten. Neue Daten können durch Benutzereingaben oder das Terminieren von Berechnungsalgorithmen entstehen. Die durch die Knoten-Struktur repräsentierten Abhängigkeiten müssen ausgenutzt werden, um zu erkennen, welche anderen Daten nun nicht mehr zum Zustand passen und daher gelöscht werden müssen. Danach wird die Komponente zum Abspeichern der Daten angestoßen.
- Die Organisation der Berechnungen: Aufstellen einer neuen Liste von noch notwendigen Berechnungen in passender Reihenfolge (im folgenden kurz "Jobliste" genannt). Außerdem muß geprüft werden, ob die gerade laufende Berechnung überhaupt auf Daten basiert, die noch Gültigkeit haben, da andernfalls die Berechnung unterbrochen werden soll. Entsprechend der Jobliste muß immer der nächste Berechnungsalgorithmus gestartet werden, wenn kein Berechnungsalgorithmus mehr läuft.
  Zur Parallelität von obiger Arbeit und den Berechnungsalgorithmen siehe Kapitel 3.3.5.

Interessant sind hauptsächlich die zwei Algorithmen, die die Abhängigkeiten, die durch die Knoten-Struktur dargestellt wird, berücksichtigen.

Als erstes wird kurz der Algorithmus "man_new()" vorgestellt, der nach Veränderung eines Datums "data" an einem Knoten "node" aufgerufen wird. Hierbei müssen die Daten von allen unmittelbaren und mittelbaren Nachfolgern als ungültig verworfen werden, denn ihre Berechnung hängt direkt oder indirekt vom Datum am Knoten "node" ab.

```
void man_new(npoint node, npoint data)
{   int i;
    save_later();                          // Für das Abspeichern sorgen
    if (NODE(node)->data!=NIL)             // Falls Daten am Knoten hängen...
    {   free_thing(NODE(node)->data);      // ...werden sie verworfen und...
        CNODE(node)->data=NIL;             // ...der Knoten ändert sich.
    }
    CNODE(node)->data=copy_thing(data);    // Die neuen Daten an den Knoten hängen
    if (NODE(node)->window!=NULL)          // Falls ein Fenster an diesem Knoten..
        NODE(node)->window->
            new_anything(copy_thing(data), // ...dann teile es dem Fenster mit.
                         NODE(node)->type,
                         NODE(node)->method);
    for (i=0; i<(NODE(node)->out); i++)    // Für jeden Nachfolger...
        man_recalc(NODE(node)->net[i].out); // ...rufe man_recalc() auf.
    man_exec();                            // Stoße den zweiten Algorithmus an
}

void man_recalc(npoint node)
{   int i;
    if (NODE(node)->data!=NIL)             // (Wie oben)
    {   free_thing(NODE(node)->data);
        CNODE(node)->data=NIL;
    }
    if (NODE(node)->window!=NULL)          // (Kommentar wie oben)
        NODE(node)->window->
            new_anything(NIL,
                NODE(node)->type,
                NODE(node)->method);
    if (!NODE(node)->recalc)               // Falls sich an diesem Knoten etwas...
        CNODE(node);                       // ...ändert, dann vermerke Veränderung.
    NODE(node)->recalc=TRUE;               // Vermerke, daß neuzuberechnen.
    for (i=0; i<(NODE(node)->out); i++)    // Wie oben, Rekursion weiter, bis...
        man_recalc(NODE(node)->net[i].out); // ...ein Knoten keine Nachfolger hat.
}
```

Bemerkung: Die Funktionen (auch die folgenden) werden hier gegenüber den tatsächlich implementierten Funktionen geringfügig verkürzt dargestellt.

Erläuterungen:

NODE und CNODE sind Makros, die das Dereferenzieren der "npoint"-Referenzen (siehe Kapitel 3.2, Stichwort "number") implementieren. Bei CNODE wird zusätzlich in der Variable "header.saved" des Knotens FALSE vermerkt.

"save_later()" stellt einen Timer, der nach kurzer Zeit eine Nachricht vom System auslöst, die das Programm daran erinnert, daß es noch abspeichern soll. Diese Nachricht erreicht das Programm aber nicht, bevor die aktuell zu bearbeitende Nachricht abgearbeitet ist. Möglicherweise wird "save_later()" aber mehrmals ausgeführt (nicht durch die angegebenen Funktionen), was dann trotzdem nur zu einer Abspeicher-Operation führt.

"man_exec()" ist mit Hilfe derselben Timer-Technik erstellt und sorgt für den Aufruf vom zweiten Algorithmus, also der Funktion "man_exec_procedure()".

Als zweites wird der Algorithmus vorgestellt, der die Jobliste neu aufstellt und ausführt. "joblist" und "end_of_joblist" sind globale Variablen vom Typ "npoint".

```
void man_exec_procedure()
{   npoint  node, d, job, joblist;
    int     maxprio, nextmaxprio;

    ...-Hier weggelassen-... : // "visited" aller Knoten auf FALSE setzen.
    ...-Hier weggelassen-... : // "maxprio=" Max. Priorität der nachfolgerlosen Knoten.
    joblist=NIL;                                  // Jobliste sei zunächst leer.
    do
    {   nextmaxprio=-1;
        for ( node=list_of_nodes_with_out0;       // Die Liste von Knoten ohne...
              node!=NIL;                          // ...Nachfolger durchgehen.
              node=NODE(node)->next_with_out0 )
        {   if (NODE(node)->prio > nextmaxprio && NODE(node)->prio < maxprio)
              nextmaxprio=NODE(node)->prio;       // Die Priorität für den nächsten...
                                                  // ...Durchgang ermitteln.
            if (NODE(node)->prio==maxprio)        // Diese Priorität ist dran.
              man_ins_joblist(node);              // Alle Vorgänger sollen in die Jobliste.
        }
        maxprio=nextmaxprio;              // Max. Priorität unter den restlichen Knoten
    } while (nextmaxprio>=0);             // Bis alle Prioritäten abgearbeitet sind

    for ( job=joblist;        // Jobliste durchgehen zum Ausführen der einzelnen Jobs
          job!=NIL;
          job=NODE(job)->nextjob )
    {   job=man_call(job);          // Eigentlicher Aufruf des Berechnungsalgorithmus
        ...-Hier weggelassen-... : // Weitere Verarbeitung (u.a. Knoten zum Abspeichern
                                   // markieren, Ergebnis dem entsprechenden Fenster
                                   //  mitteilen, "recalc" des Knotens auf FALSE setzen).
  }
}
```

```
void man_ins_joblist(npoint target)
{  int  i;

   if (!NODE(target)->recalc ||                  // Rekursion beenden, falls der
      (NODE(target)->visited)) return;           // ...Knoten nicht neuzuberechnen...
                                                 // ...oder schon abgehakt ist.
   NODE(target)->visited=TRUE;                   // Knoten abhaken.
   for (i=0; i<(NODE(node)->in); i++)            // Rekursion zu den Eingängen...
      man_ins_joblist(NODE(node)->net[i].in);// ..., bis ein Knoten keine...
                                                 // ...Vorgänger mehr hat.
   if (joblist==NIL)                             // Falls Liste noch leer, dann...
      joblist=target;                            // ...erstes Element der Liste,...
   else                                          // ...sonst...
      NODE(end_of_joblist)->nextjob=target;  // ...an das aktuelle Ende der...
   NODE(target)->nextjob=NIL;                    // ...Jobliste anhängen.
   end_of_joblist=target;                        // Letztes Element ist es in jedem Fall.
}
```

Erläuterungen:

"man_exec_procedure()" geht die Ziele nach ihrer Priorität geordnet durch (zuerst die höchste, dann absteigend). Prioritäten sind Elemente von $\mathbb{N}^{\geq 0}$; größere Werte geben höhere Ausführungspriorität an.

"man_call()" ruft den eigentlichen Berechnungsalgorithmus auf und terminiert erst, wenn dieser terminiert oder abgebrochen wird.

"man_ins_joblist()" macht eine Rekursion zu den Vorgängern, die jeweils an einem Ziel hängen (also genau die umgekehrte Richtung von "man_recalc()").

Das Abhaken der Knoten mit Hilfe von "visited" stellt in erster Linie sicher, daß kein Knoten mehr als einmal berechnet wird. Die Funktionen der Oberfläche zum Verbinden der Transformationsfunktionen und Aggregationsoperatoren sorgen dafür, daß keine Zykel in der Knotenstruktur entstehen können (entsprechende Versuche, Zykel zu erzeugen, werden mit einer Fehlermeldung abgeblockt). Trotzdem wird "visited" (vom aktuellen Ziel "target") *vor* der Rekursion auf TRUE gesetzt, obwohl dies eigentlich nicht notwendig wäre. Mit dieser Reihenfolge ist das Programm fehlertoleranter, denn ein Zykel würde so nicht zum Absturz führen.

"man_ins_joblist()" fügt die Knoten immer erst dann in die Jobliste ein, wenn die Rekursion stattgefunden hat. Dies stellt sicher, daß alle Argumente berechnet sind, wenn der Job zu diesem Knoten ausgeführt werden soll.

"man_exec_procedure()" geht beim Abarbeiten die Jobliste in derselben Reihenfolge durch, wie die Knoten in die Liste eingefügt wurden.

In obiger Darstellung von "man_exec_procedure()" wurden alle Datails weggelassen, die damit zu tun haben, daß "man_exec_procedure()" während der Ausführung noch einmal aufgerufen werden könnte. Die Vorgehensweise für diesen Fall ist, daß die zweite Ausführung eine Nachricht an die erste schickt (mit Hilfe einer globalen Variablen) und dann sofort wieder terminiert. Die erste stellt dann die Jobliste neu auf und startet dann die Jobs aus der neuen Jobliste.

Bemerkung zu "man_call()": Als Argumente der Berechnungsalgorithmen treten die Datenelemente der Vorgängerknoten auf (siehe dazu auch Abbildung 3.2.2). Diese Argumente werden mit Hilfe der "call by value"-Methode übergeben; dies kann jedoch nicht mit dem "call by value"-Mechanismus von C geschehen, da dies die Referenzen auf die Datenelemente nicht selbständig weiterverfolgt. Von den Argumentwerten wird also explizit eine Kopie angelegt, die bei Beenden des Berechnungsalgorithmus wieder freigegeben wird. Der Grund für diese Vorgehensweise ist, daß die Managementkomponente quasi-gleichzeitig zum Berechnungsalgorithmus ausgeführt wird; dadurch ergibt sich die Forderung, daß jede Ausführungsinstanz ihre eigenen Variablen benötigt. Im Fall *eines* Arguments wurde von diesem Vorgehen aber eine Ausnahme gemacht. Es betrifft die Matrix, die zum Fuzzy Kriging benötigt wird: Das Problem besteht darin, daß die Matrix sehr viel Speicher verbrauchen kann (bis zu 16 MByte in FUZZEKS). Die meisten Windows-Rechner haben nicht genügend Speicher, um davon eine Kopie anzulegen, ohne daß die Mechanismen zur virtuellen Speicherverwaltung von Windows 3.1 ausgelöst werden, was zu gravierenden Geschwindigkeitseinbußen führen kann. Die Matrix wird also nicht mit der "call by value"-Methode übergeben, sondern verliehen, d.h. sie wird dem Berechnungsalgorithmus übergeben, im Knoten zu der Matrix wird dafür aber die Referenz auf die Matrix entfernt, so daß die Managementkomponente während der Ausführung des Berechnungsalgorithmus keinen Zugriff auf die Matrix hat (nach Beenden des Berechnungsalgorithmus wird sie wieder zurückgegeben).

3.3.2 Fenster und Benutzerinteraktionen

Die Fensterklassen für die FUZZEKS-Fenster wurden von der entsprechenden Klasse aus der MFC-Klassenbibliothek (nämlich "CFrameWnd") abgeleitet. Die Behandlung der diversen Nachrichten vom Betriebssystem wurde in dieser Klasse schon vorbereitet. Man muß nur noch Methoden zur Behandlung jener Nachrichten schreiben, die für die eigenen Belange von Interesse sind (i.a. sind folgende Nachrichten von Interesse: Maus-, Tastatur- und Menüeingaben, Veränderung des Inhalts von Eingabefeldern, Veränderung der Fenstergröße und die Notwendigkeit zur Neudarstellung des Fensterinhalts).

Außerdem bietet die MFC-Klassenbibliothek noch die für Ausgaben notwendigen Klassen. Man benötigt zur Ausgabe in ein Fenster z.B. einen "Gerätekontext" und "Koordinaten", möglicherweise auch "Farben", "Pinsel", "Rechtecke", und anderes. Der Gerätekontext bietet Methoden zum Ziehen von Linen und anderem; er bildet die Ausgabewünsche des Programms auf die Funktionen der Treiber ab (in diesem Fall auf die Treiber-Funktionen der Graphikkarte).

Die Fenster haben i.a. jeweils eigene Kopien der Daten, die die Managementkomponente verwaltet, aber es gibt natürlich auch zusätzliche Instanzvariablen, die hauptsächlich den Zustand der Benutzerinteraktion wiederspiegeln. Als Beispiel dazu sei das Ziehen einer Linie in der Darstellung des Krigingresultats mit Hilfe der Maus beschrieben (zum Zweck dieser Linie siehe die Beschreibung von "iso" in Kapitel 3.1.4 und auch das Darstellen in einem Schnittfenster am Ende desselben Kapitels):

Wenn die Nachricht "Maustaste im Fenster gedrückt" an ein Krigingfenster geht, löst diese Nachricht vom Betriebssystem den Aufruf einer entsprechenden Methode des entspechenden Fenster-Objekts aus, die zunächst noch die Mausposition im Fenster überprüft. Falls sich der Mauspfeil gerade in der Darstellung des Krigingresultats befindet, wird diese Startkoordinate und der Fakt, daß die Maustaste noch gedrückt ist, in einer Instanzvariablen abgespeichert. Später kommen i.a. Nachrichten, die neue Mauspositionen anzeigen hinzu, und auch diese werden abgespeichert; zusätzlich wird die aktuelle Linie angezeigt. Wenn dann die Nachricht kommt, daß die Maustaste losgelassen wurde, wird in den Instanzvariablen vermerkt, daß die Maustaste nicht mehr gedrückt ist. Die Linie kann nun benutzt werden, indem z.B. über das Menü eine entsprechende Nachricht ausgelöst wird, die zum Aufruf der Methode zum Erzeugen eines Schnittfensters führt.

Erwähnenswert sind noch die Implementierungen der folgenden Funktionen, die mit der Benutzeroberfläche zu tun haben.

Die ersten beiden Funktionen haben mit der Darstellung des Krigingresultats in zwei Dimensionen zu tun. Zunächst einmal muß der davon sichtbare Ausschnitt vom Anwender eingestellt werden können, denn wenn der Bereich, in dem Werte geschätzt werden sollen, z.B. sehr lang relativ zu seiner Breite ist, dann würde man bei vollständiger Darstellung des Resultats für diesen Bereich keine Details mehr erkennen können. Um dieses Problem zu lösen, bieten die meisten Programme einen Zoomfaktor und Schieberegler ("scroll-bars") zur Einstellung der absoluten Position des sichtbaren Bereichs an. Die Schieberegler sind aber für zweidimensionale Bereiche, bei denen es auf eine vernünftige Darstellung des Längen/Breiten-Verhältnisses ankommt, nicht besonders gut geeignet; insbesondere können die von Windows 3.1 zur Verfügung gestellten Regler nicht einmal anzeigen, wieviel man denn in *einer* Dimension vom ganzen Bereich sieht, da der Knopf zum Manipulieren der Position immer gleich groß ist.

Aus diesem Grund wurde neben einem Dialogfenster, in dem man den Bereich numerisch angeben kann, und zwei Knöpfen, die Standardeinstellungen bewirken, ein einfacher zweidimensionaler 'Schieberegler' erstellt. In diesem wird der aktuell dargestellte Bereich als hellgraues Rechteck angezeigt, und zwar auf dem dunkelgrauen Rechteck, das den gesamten Bereich repräsentiert. Wenn man mit der Maus in den Bereich des dunkelgrauen Rechtecks klickt, legt dies die Mitte des zukünftig dargestellten Bereichs fest. Falls alles dargestellt werden kann, erscheint der Hinweis "100%" (siehe Abbildung 3.1c), da in diesem Fall das dunkelgraue Rechteck vollständig vom hellgrauen überdeckt wird.

Die zweite Funktion, die mit der Darstellung des Krigingresultats in zwei Dimensionen zu tun hat, sorgt für die Darstellung der Isolinien. Das Konzept zur Darstellung der Isolinien ist wie folgt: Das Fuzzy Kriging wird für ein regelmäßiges Raster von Koordinaten durchgeführt. Dann muß nur noch jeweils innerhalb eines der Rechtecke, bei denen die interpolierten Werte für die Eckpunkte bekannt sind, eine angenäherte Darstellung der Isolinien gemacht werden. Eine höhere Auflösung bei dem regelmäßigen Raster einzustellen erlaubt eine genauere Darstellung (erfordert aber längere Rechenzeiten).

Die Darstellung der Isolinien innerhalb obengenannter Rechtecke wird wie folgt vorgenommen: Zunächst einmal werden auf den vier Rändern des Rechtecks die Stellen durch lineare Interpolation bestimmt, durch die Isolinien verlaufen; dazu werden dann immer auch die interpolierten Werte der Isolinien abgespeichert. Danach werden die eben gefundenen Positionen miteinander durch eine gerade Linie verbunden, falls deren interpolierte Werte gleich sind. Dies geht allerdings schief, wenn vier anstatt zwei Stellen den gleichen Wert aufweisen, denn dann gibt es zwei verschiedene Möglichkeiten, jeweils zwei zu verbinden. Die "richtige" davon kann man nicht bestimmen (ohne weitere Informationen als nur den Werten bei den vier Eckpunkten). Außerdem können (aufgrund von Rechenungenauigkeiten) auch *drei* Stellen mit gleichen Werten auftauchen. Deswegen wurden in diesen Fällen alle jene Stellen mit ihrem gemeinsamen Schwerpunkt verbunden. Falls sich daraufhin die so gezeichneten "Isolinien" mit anderen überschneiden, muß der Anwender die Auflösung des Rasters erhöhen, was *spätestens* dann zum Erfolg (keine Überschneidungen mehr) führt, wenn jedes Rechteck nur noch von Isolinien eines Wertes durchlaufen wird. Solche Überschneidungsprobleme treten in der Praxis jedoch nur sehr selten auf.

Die dritte Funktion, die hier noch erwähnt werden soll, liefert, ob sich ein Punkt auf einer geraden Linie befindet oder einen Abstand von weniger als 3 Einheiten von dieser Linie hat (die Koordinaten müssen alle als ganze Zahlen angegeben werden). Dies wird benötigt, um die Verbindungen zwischen den Aggregationsoperatoren mit der Maus selektierbar zu machen, was wiederum beim Löschen der Verbindungen gebraucht wird. Die Einheiten sind in diesem Fall Pixel.

Diese Funktion ist wie folgt implementiert: Zunächst einmal wird geprüft, ob der Punkt in dem Rechteck liegt, das sich ergibt, wenn man Anfangs- und Endpunkt der Linie als Eckpunkte eines Rechtecks betrachtet, und dann dieses Rechteck noch auf allen Seiten um 2 Einheiten vergrößert (nur 2 Einheiten, da die Koordinaten in ganzen Zahlen beschrieben werden). Falls der Punkt nicht in diesem Rechteck liegt, wird FALSE als Resultat geliefert.

Sonst wird geprüft, ob der Abstand zwischen Anfangs- und Endpunkt kleiner als 1 (gleich 0) ist. Dann ist die 'Linie' nur ein Punkt und es wird TRUE als Resultat geliefert, da dann alle möglichen Abstände des Punkts von der 'Linie' (in obigem Rechteck) kleiner als 3 Einheiten sind.

Wenn dies nicht der Fall ist, wird der Startpunkt der Linie (als Vektor betrachtet) von allen Koordinaten subtrahiert, d.h. die Koordinaten werden so parallelverschoben, daß der Startpunkt in den Ursprung kommt. Der Endpunkt habe dann die Koordinate (e_x, e_y) und der zu prüfende Punkt (p_x, p_y). Zum Vektor (e_x, e_y) wird eine Senkrechte gleicher Länge gebildet und mit (s_x, s_y) bezeichnet (z.B. $(s_x, s_y):=(-e_y, e_x)$). Der Punkt (p_x, p_y) soll sich nun als Linearkombination von (s_x, s_y) und (e_x, e_y) ergeben. Dies ist immer möglich, da weder (s_x, s_y) noch (e_x, e_y) null sind (dies wurde oben abgefangen), und die beiden Vektoren senkrecht aufeinander stehen. Es ergibt sich folgendes lineares Gleichungssystem (a und b seien reelle Zahlen):

$$p_x = a \cdot s_x + b \cdot e_x$$

$$p_y = a \cdot s_y + b \cdot e_y$$

Daraus bestimmt man a, und $|a| \cdot |(s_x, s_y)|$ $(= |a| \cdot \sqrt{s_x^2 + s_y^2})$ ist dann der Abstand des Punktes von der Linie. Wenn dieser kleiner als 3 ist, wird TRUE als Resultat geliefert, sonst FALSE.

3.3.3 Berechnungsalgorithmen

Die logische Gruppierung der Berechnungsalgorithmen in jene, die der Aggregation dienen, und in jene, die Bestandteil des Kriging-Verfahrens sind, wurde schon in Abbildung 3.3 dargestellt. Die folgenden zwei Unterkapitel beschäftigen sich mit jeweils einer dieser Gruppen.

Als Argumente der Berechnungsalgorithmen treten die Datenelemente der Vorgängerknoten auf (siehe dazu auch Abbildung 3.2.2). Besonderheiten der Parameterübergabe werden in der Bemerkung am Ende von Kapitel 3.3.1 behandelt.

Während der Ausführung der Berechnungsalgorithmen wird der Fortschritt angezeigt (relativ zur vollständigen Ausführung dieses Berechnungsalgorithmus). Die Implementierung dazu erfordert, daß an geeigneten Stellen der Berechnungsalgorithmen dieser relative Fortschritt abgeschätzt und angezeigt wird. Dies wurde aus praktischen Gründen mit der Implementierung des Multitasking (siehe Kapitel 3.3.5) kombiniert.

Bemerkung (zur Abschätzung von Zeitkomplexitäten in folgenden Kapiteln): Diese sind nur dann richtig angegeben, wenn man dem Betriebssystem das Auslagern von Speicherbereichen in sogenannte Swap-Dateien verbietet, da sonst erhöhte Komplexitäten beim Zugriff auf die Variablen denkbar wären.

3.3.3.1 Aggregation

Die Aggregation besteht aus den Transformationsfunktionen und den Aggregationsoperatoren, die in der Managementstruktur vorkommen. In Kapitel 2.2.3 wurde gezeigt, daß die einzelnen unscharfen Funktionen voneinander unabhängig berechnet werden können und sich trotzdem das korrekte unscharfe Gesamtergebnis ergibt.

Die Implementierung der Transformationsfunktionen besteht in der Umsetzung der auf Intervalle erweiterten Funktionen aus Kapitel 2.2.1. Der Berechnungsalgorithmus soll alle Werte einer regionalisierten Variablen transformieren. Dazu wird nur (um die Funktionen aus Kapitel 2.2.1 herum) eine Schleife über die unscharfen Zahlen von einem Parameter (von den verschiedenen vorgegebenen Stellen) benötigt. Außerdem wird natürlich eine Schleife über die Liste der Schnitte benötigt. Undefinierte Werte werden einfach wieder auf den undefinierten Wert abgebildet.

Die Implementierung der Aggregationsoperatoren besteht in der Umsetzung der auf Intervalle erweiterten Funktionen aus Kapitel 2.2.2. In diesem Fall gibt es aber eine *Liste* von Parametern, die verknüpft werden müssen. Genauer gesagt müssen diejenigen Werte verknüpft werden, die die gleiche Positionsangabe aufweisen. Diese müssen zunächst herausgesucht werden, da sie nicht unbedingt alle dieselbe Position in den Arrays der unscharfen Zahlen mit Koordinatenangabe haben, denn der Anwender könnte die verschiedenen Eingabeparameter aus mehreren Tabellen innerhalb einer Eingabedatei (siehe Kapitel 3.1.1) oder aus verschiedenen Eingabedateien gelesen haben.

Der Algorithmus dazu arbeitet folgendermaßen:

```
Initialisiere n mit dem kleinstmöglichen Index eines Wertes im Ergebnis.
Schleife über die nicht-undefinierten Werte vom ersten Parameter (mit Index i)
{  Schleife über 2. bis letzten Parameter (mit Index p)
   {  Suche p-ten Parameter nach Koordinate durch, die gleich der von Parameter...
         ... i ist; den Index dieser Koordiante im p-ten Parameter speichere in k[p].
      Falls keine solche Koordinate gefunden wurde oder...
         ...der Wert mit dem Index k[p] vom p-ten Parameter undefiniert ist,...
         ...dann springe zur Marke next_i (also schreibe die äußerste Schleife fort).
   }
   Schleife über die Liste der Schnitte, d.h. Intervalle (mit Index j)
   {  Initialisiere einen Akkumulator a mit dem j-ten Intervall vom i-ten Wert...
         ...des ersten Parameters (im Falle der SUM-Verknüpfung ist zunächst...
         ...noch mit dem Gewicht des ersten Eingangs zu multiplizieren).
      Schleife über 2. bis letzten Parameter (mit Index p)
      {  Verknüpfe das Intervall aus a und...
            ...das j-te Intervall vom k[p]-ten Wert des p-ten Parameters...
            ...mit der gewünschten Verknüpfung (bei SUM muß das letzte Argument...
            ...mit dem entsprechenden Gewicht multipliziert werden) und...
            ... speichere das Ergebnis in a.
      }
      Speichere das Intervall a als j-ten Schnitt beim n-ten Wert des Ergebnisses.
   }
   Speichere die Koordinate des i-ten Wertes vom ersten Parameter...
      ...als Koordinate beim n-ten Wert des Ergebnisses.
   Erhöhe n um 1.
   next_i (Marke)
}
```

Bemerkungen:

Im Ergebnis tauchen bei diesem Algorithmus niemals undefinierte Werte auf.

Die implementierten Verknüpfungen entsprechen denen aus Kapitel 2.2.2, nur daß die Berechnung der linken und rechten Grenze des Ergebnisintervalls gewissermaßen parallel ablaufen. Dies ist möglich, obwohl der Akkumulator **a** einen Schnitt speichert, da die linke Grenze sich immer aus den linken Grenzen ergibt und die rechte aus den rechten.

Zur Abschätzung der Rechenzeitkomplexität: Sei AP die Anzahl der Parameter, AS die Anzahl der Schnitte und MW die maximale Anzahl von Werten der Parameter. Dann ist die Rechenzeitkomplexität mit
$O(MW \cdot \max(AP \cdot MW, AS \cdot AP))$ abschätzbar. Da AP i.a. relativ klein und für einen gegebenen Aggregationsoperator konstant ist, kann man sie auch mit **$O(\max(MW^2, MW \cdot AS))$** abschätzen. Versteckte Komplexitäten, etwa beim Zugriff auf Variablen, gibt es nicht (d.h. sie sind alle O(1)).

3.3.3.2 Fuzzy Kriging

Zur Realisierung des Fuzzy Kriging wurden, wie schon in Abbildung 3.3 gezeigt, drei Algorithmen implementiert: Die Berechnung des experimentellen Variogramms, die Berechnung der invertierten Matrix (zu derjenigen Matrix aus Kapitel 2.3.1.2) und die Berechnung des Fuzzy Kriging-Resultats mit Hilfe der invertierten Matrix. Die Berechnung einzelner theoretischer Variogrammwerte muß man eher bei den grundlegenden Funktionen zur Nutzung der Datenstrukturen einordnen, da sie sowohl vom Krigingfenster zur Darstellung des theoretischen Variogramms als auch für die zwei letztgenannten Algorithmen benötigt wird und im Grunde nur das darstellt, was mit der Datenstruktur "t_theovario" beschrieben wird.

Zur Berechnung des experimentellen Variogramms:

Dieser Algorithmus hat zwei Argumente: Erstens die regionalisierte Variable, von der ein experimentelles Variogramm berechnet werden soll (vom Typ "t_fuzzyarray") und zweitens die Parameter zur Berechnung (vom Typ "t_evparams", siehe auch Bild 3.2.2); zu diesen gehört z.B. auch die gewünschte Anzahl von Klassen *k*.

Als erster Schritt wird abgeschätzt, in welchem Bereich von Abständen es sinnvoll ist, Werte zu schätzen. Dazu wird das Minimum und Maximum der Abstände zwischen jeweils zwei Punkten brechnet.

Dieser Bereich von Minimum bis Maximum wird dann in *k* gleichlange Klassen unterteilt. Eine Schleife über diese *k* Klassen umfaßt den Rest dieses Algorithmus.

In dieser werden dann nur noch die experimentelle Variogrammformel aus Kapitel 2.3.1.1 für den scharfen 1-Schnitt (denn dieser besteht nur aus einem Wert) und die näherungsweise Abbildung des experimentellen Variogramms für den 0-Schnitt (die am Ende von Kapitel 2.3.2.1 angegeben wurde) berechnet. Dazu benötigt man für die jeweilige Summenbildung eine Schleife über die Wertepaare, die in der aktuellen Klasse liegen.

Zur Berechnung der invertierten Matrix:

Dieser Algorithmus hat auch zwei Argumente: Erstens wieder die regionalisierte Variable und zweitens das theoretische Variogramm.

Zunächst wird die $(n+1)\cdot(n+1)$-Matrix von scharfen Werten aus Kapitel 2.3.1.2 erzeugt und in eine Variable vom Typ "t_matrix" geschrieben. Dann wird die Matrix mit Hilfe des Gauß-Jordan-Algorithmus ([Stoer 1983]) invertiert (Zeitkomplexität $O((n+1)^3)$ $(=O(n^3))$.

Zur Berechnung des Fuzzy Kriging-Resultats (mit Hilfe der invertierten Matrix):

Dieser Algorithmus hat fünf Argumente: Erstens die regionalisierte Variable, zweitens das theoretische Variogramm, drittens die invertierte Matrix, viertens die "region"-Angaben aus der (zuletzt gelesenen) Eingabedatei (am Knoten 2 in Abbildung 3.2.2) und viertens zusätzliche Krigingparameter, die z.B. die Positionen festlegen, für die jeweils ein Wert berechnet werden soll (siehe auch Abbildung 3.2.2).

Außerdem hat dieser Algorithmus zwei Ergebnisse: Das Krigingresultat, das den Typ "t_fuzzyarray" hat (mit der in Kapitel 3.2.1 beschriebenen möglichen zweidimensionalen Struktur unter Nutzung des "line_length"-Elements der Struktur "t_fuzzyarray"), und, falls dies vom Anwender gewünscht wurde (einer der zusätzlichen Krigingparameter), auch die Kriging-Varianz.

Eine Schleife über die Koordinaten, für die ein Wert zu schätzen ist (deren Anzahl sei AK), umfaßt den Rest dieses Algorithmus.

Für die Kriging-Varianz ist die Formel (G2) aus Kapitel 2.3.1.2 zu berechnen (Zeitkomplexität $O(n^2)$ pro Koordinate).

Für das Krigingresultat ist zunächst der Vektor von Variogrammwerten ($(1, \gamma(x, x_1), ..., \gamma(x, x_n))^t$, siehe Kapitel 2.3.1.2) zu berechnen. Dann ist die invertierte Matrix mit diesem Variogrammvektor zu multiplizieren, um die Faktoren der Hauptkriginggleichung $\delta_i(x)$ für i von 1 bis n herauszubekommen (Zeitkomplexität $O(n^2)$ pro Koordinate, falls n>0). Zum Schluß wird die Hauptkriginggleichung für abgeschlossene Intervalle (siehe Kapitel 2.3.2.2) mit diesen Faktoren auf die Schnitte angewendet (Zeitkomplexität O(AS*n) pro Koordinate; AS sei dabei die Anzahl der Schnitte).

Insgesamt ergibt sich eine Zeitkomplexität von O (AK * max(n^2, AS*n)), falls n>0. Versteckte Komplexitäten, etwa beim Zugriff auf Variablen oder bei der Berechnung der theoretischen Variogrammwerte, gibt es auch hier nicht. Dieser Zeitaufwand ist oft größer als bei der Invertierung der Matrix, denn AK ist bei der Berechnung der Darstellung des Krigingresultates in zwei Dimensionen i.a. größer als n (denn das Ergebnis kann bei zu groben Auflösungen nicht mehr brauchbar dargestellt werden). Wenn man für jede Koordinate die Matrix invertieren müßte, würde sich die Komplexität um den Faktor n erhöhen. Aufgrund dieser vollständigen Abtrennung kann in manchen Situationen ganz auf eine Neuberechnung der invertierten Matrix verzichtet werden, wenn nämlich nur die zusätzlichen Krigingparameter vom Anwender verändert werden (insbesondere ist dies der Fall, wenn die Werte entlang eines Schnittes interpoliert werden sollen oder gar nur der Wert für einen einzelnen Punkt, denn dann ist AK im ersten Fall meist nur größer als $\sqrt{n}$, in letzteren Fall ist AK=1).

3.3.4 Speichern des Zustands in Dateien

Wie in Kapitel 3.2 bei der Beschreibung der Datenstruktur "thing_header" schon kurz beschrieben wurde, basiert das Konzept zum Abspeichern des aktuellen Zustands in Dateien darauf, daß der Anwender ein Verzeichnis angibt, in dem das Programm seine Zustandsdaten fortan speichern darf. Das Programm soll die Dateien in diesem Verzeichnis dann automatisch immer auf dem neuesten Stand bringen.

Dabei stellt sich zunächst die Frage, welche Daten jeweils zusammen in einer Datei abgespeichet werden sollten.

Dazu sollen zunächst einige Vorteile des Zusammenfassens von Daten zu ganzen Kategorien, die in einer einzigen Datei gespeichert werden sollen, angegeben werden:

- Bei vielen Änderungen geht es relativ schnell, weil nur wenige Dateien geöffnet werden müssen.
- Es müssen weniger Dateien angelegt werden, wodurch man Speicher im Dateisystem spart.
- Die Daten, die gelöscht wurden, brauchen einfach nur nicht neu geschrieben zu werden. Bei Einzeldateien muß das Löschen von Daten im Zustand auch das Löschen von Dateien bewirken.

Dagegen stehen die Vorteile für einzelne Dateien je Instanz der Datenstrukturen:

- Bei wenigen Änderungen oder sehr großen Einheiten geht es sehr viel schneller.
- Man könnte bei Hauptspeicherplatzmangel die Dateien bequem als Auslagerungsdateien benutzen. Dies wurde in FUZZEKS jedoch nicht durchgeführt.

Aufgrund dieser Überlegungen wurde die Entscheidung getroffen, daß Zwischenergebnisse in einzelnen Dateien und andere Daten möglichst in gemeinsamen Dateien abgespeichert werden sollen, denn die Zwischenergebnisse sind immer Matrizen oder Arrays von unscharfen Zahlen mit Koordinatenangabe, die i.a. recht groß sind. Die Eingabedaten aus den Eingabedateien könnten auch ganz gut in einer gemeinsamen Datei abgespeichert werden; um die Implementierung aber nicht zu kompliziert zu machen, wurde enschieden, daß *alle* Matrizen und Arrays von unscharfen Zahlen mit Koordinatenangabe in einzelnen Dateien gespeichert werden sollen. Die restlichen Daten werden in nur zwei Gruppen für gemeinschaftliche Dateien unterteilt, nämlich zum einen die Daten vom Typ "t_node" und zum anderen die restlichen.

Eine weitere Entscheidung, die man treffen muß, ist, ob man Daten einfach als Abbild von Hauptspeicherstücken oder in Form einer ASCII-Repräsentation in die Dateien schreiben möchte. Erstere Methode hat den Vorteil, insbesondere bei größeren Dateien sehr platz- und zeiteffizient zu sein. Außerdem ist sie sehr einfach zu implementieren. Die Vorteile von ASCII-Repräsentationen sind zum einen, daß man Datenstrukturänderungen zwischen verschiedenen Programmversionen leichter implementieren kann, und zum anderen, daß man die Dateien leichter auch ohne spezielles Programm lesen kann (z.B. zur Fehlersuche, um sich ein Bild vom Zustand zu verschaffen).

Die Art, in der die Referenzen zwischen den Instanzen der Datenstrukturen realisiert sind (siehe Kapitel 5.2, Datenstruktur "thing_header", Kommentar zu "number"), erlaubt, die Referenzen (implementiert als natürliche Zahlen) ohne weitere Überarbeitung abzuspeichern.

Es wurde die Entscheidung getroffen, daß die Matrizen und Arrays von unscharfen Zahlen mit Koordinatenangabe aufgrund ihrer Länge als Abbild von Hauptspeicherstücken geschrieben werden sollen (bis auf die "thing_header"-Informationen) und alles andere immer in Form von ASCII-Repräsentationen.

Als Dateinamen werden gewählt:

- *n*.1T für das Speichern der Datenstruktur mit der Referenz *n* ($n \in \mathbb{N}$, s.o.).
- NODES.MT für das Speichern der Knoteninformationen.
- ELSE.MT für das Speichern aller restlichen Informationen.

Zur Realisierung der ASCII-Repräsentationen soll die Syntax grob beschrieben werden. Am Anfang jeder Zeile wird ein (ein Zeichen langer) Bezeichner angegeben (gefolgt von ':'), der auf die Bedeutung dieser Zeile hinweist.

In der ersten Zeile einer Datei steht der Programmname und die Versionsinformation des Programms, mit der sie geschrieben wurde (z.B. "V:FUZZEKS 1 0").

Danach folgen die Instanzdaten zu allen Datenstrukturen, die in dieser Datei gespeichert werden sollen, in folgendem Format.

Die "thing_header"-Informationen werden wie folgt gespeichert: Die erste Zeile enthält die Nummer der Instanz (Bezeichner: '#'), die zweite den Typ der Instanz (Bezeichner: '*') und die dritte die Länge des Arrays in der Datenstruktur, falls vorhanden, sonst 0 (Bezeichner: 'L').

Danach folgen die eigentlichen Daten dieser Instanz. Im Fall des direkten Speicherabbilds kommt zunächst noch eine Zeile in ASCII, nämlich die Angabe der Länge des Speicherabbilds in Byte (Bezeichner: 'B'); das Speicherabbild folgt unmittelbar auf diese Zeile. In den anderen Fällen folgen datentypspezifische Zeilen nach demselben Muster, bis die Instanz vollständig beschrieben ist.

Die Bezeichner zu jeder Zeile zu benutzen, hat den Vorteil, daß für Versionsänderungen weitere Bezeichner eingefügt werden können; damit könnten neuere Programmversionen Dateien älterer Programmversionen mit denselben Funktionen lesen.

Zur Implementierung wurden in Kapitel 3.2 schon zur Datenstruktur "thing_header" die Elemente "tosave" und "saved" kurz beschrieben, in denen festgehalten wird, ob die entsprechende Instanz überhaupt abgespeichert werden soll und, falls dies so ist, ob sie nach ihrer letzten Veränderung schon abgespeichert wurde. Diese Informationen müssen natürlich bei jeder Veränderung entsprechend festgelegt werden. In Kapitel 3.3.1 wurden schon kurz die Makros erwähnt, die zum Dereferenzieren der Referenzen der Instanzen dienen, nämlich NODE() und CNODE(). CNODE() setzt, im Unterschied zu NODE(), das "header.saved"-Element als Seiteneffekt auf FALSE, und muß immer bei schreibenden Zugriffen auf die Instanzen verwendet werden, sofern sich durch die betreffende Veränderung die Notwendigkeit zum Abspeichern ergibt.

Die Programmkomponente zum Speichern der Dateien enthält hauptsächlich zwei Funktionen: Eine zum Speichern des aktuellen Zustands und eine zum Laden eines abgespeicherten Zustands. Erstere wird automatisch (aber meistens verzögert) ausgelöst, wenn Daten verändert wurden.

Die Funktion, die den aktuellen Zustand speichert, geht folgendermaßen vor: Als erstes werden veraltete Dateien gelöscht. Ob eine Datei veraltet ist, wird in einem Array gespeichert, das für jede mögliche Referenz auf eine Instanz genau diese Information enthält. Sie wird beim Löschen einer Instanz im Speicher gesetzt und nach dem Löschen der Datei wieder gelöscht.

Dann wird geprüft, ob sich in den gemeinschaftlichen Dateien jeweils auch nur eine Instanz verändert hat, hinzugekommen ist, oder gelöscht wurde. Dann wird sie entsprechend neu geschrieben.

Zuletzt wird für jede Instanz, die eine Matrix oder ein Array von unscharfen Zahlen mit Koordinatenangabe ist, eine eigene Datei geschrieben, falls dafür noch keine aktuelle Datei vorliegt.

Zum Laden eines Zustands werden zunächst alle Instanzen freigegeben, und dann werden die Dateien, die den alten Zustand beschreiben, im vom Anwender angegebenen Verzeichnis gelesen. Zuerst werden die Knoten- und sonstigen Informationen gelesen, woraufhin untersucht werden kann, welche Instanzen man noch laden muß, denn für alle Matrizen oder Arrays von unscharfen Zahlen mit Koordinatenangabe existiert eine Referenz in einem Knoten. Das Freigeben aller Instanzen vorweg ist notwendig, damit die Instanzen des einzulesenden Zustands ihre alten Referenz-Nummern behalten können.

3.3.5 Multitasking und Multithreading unter Windows 3.1

Zwei Arten von quasi-parallelen Programmausführungen wurden in FUZZEKS implementiert. Zum einen sollen andere Programme auf demselben Rechner auch dann noch weiterlaufen, d.h. Nachrichten vom Betriebssystem empfangen und verarbeiten können, wenn FUZZEKS mit längerfristigen Berechnungen beschäftigt ist. Zum anderen sollen auch von FUZZEKS selbst während dieser Zeit noch Nachrichten angenommen und verarbeitet werden, d.h. die Benutzeroberfläche soll aktiv bleiben.

Unter Windows 3.1 wird die erstere Art oben beschriebenen Verhaltens der Programme nicht vom Betriebssystem sichergestellt. Auch für die zweite wurden keine Vorkehrungen getroffen. In den Handbüchern zum C/C++ Compiler der Version 7.0 zumindest ([Microsoft, 1991]) steht auch kein Hinweis, wie man so etwas realisieren könnte. Es wird nur darauf hingewiesen, daß die Bearbeitung einer Nachricht möglichst kurz geschehen sollte; falls dies nicht möglich sei, so möge man solange den Mauscursor in Form einer Eieruhr anzeigen, um dem Benutzer anzudeuten, daß während dieser Zeit keine Eingaben zu unmittelbaren Reaktionen der Programme führen können (es werden zwar noch Nachrichten erzeugt, aber die Programme bekommen sie während dieser Zeit noch nicht).

Tatsächlich hat Windows 3.1 aber grundlegende Eigenschaften, die die quasiparallele Ausführung von mehreren Programmen gleichzeitig möglich machen. Das wichtigste ist die weitgehende Trennung der Datenbereiche für verschiedene Programme, insbesondere hat jedes Programm seinen eigenen Stack, seine eigenen globalen Datenbereiche und seine eigene Eingabewarteschlange für die Nachrichten. Weil ein Programm hier übrigens nur *einen* Stack haben kann, gilt, daß jedem Programm genau ein Prozeß zugeordnet ist. Was fehlt, ist daß das Betriebssystem aktiv die Umschaltung auf andere Prozesse erzwingt, wenn ein Prozeß mehr als eine bestimmte Zeit lang läuft (dies wäre dann sogenanntes "preemptives" Multitasking). Man kann allerdings bestimmte Funktionen des Betriebssystems aktiv aufrufen, die als Seiteneffekt haben, daß das Betriebssystem auf andere Prozesse umschaltet, falls andere Prozesse etwas zu tun haben. Diese Methode wird üblicherweise "kooperatives" Multitasking genannt.

Ohne diese Möglichkeit müßten in einem Programm die Berechnungen, die zu lange dauern, in kurze aufgespalten werden. Das Programm müßte dann das jeweils nächste Stückchen der Berechnung auslösen, indem es, falls die Berechnung noch nicht komplett abgeschlossen ist, kurz vor Beenden der Bearbeitung der aktuellen Nachricht, eine Nachricht an sich selbst schickt. Diese Methode zu verwirklichen ist außerordentlich schwierig, denn das Aufspalten der zu langen Berechnungen in ausreichend kurze Stückchen ist wahrscheinlich nur in seltenen Fällen auf einfache Weise möglich. Man müßte zum Teil Schleifenkonstruktionen simulieren, anstatt diejenigen der Programmiersprache nutzen zu können; auch lokale Variablen zu einer Funktion müßten jeweils anders implementiert werden.

Eine der Funktionen, die als Seiteneffekt hat, daß das System auf andere Prozesse umschalten kann, ist die Funktion, die das System nach der nächsten Nachricht fragt, ohne diese allerdings aus der Nachrichtenwarteschlange zu entfernen, falls eine vorhanden ist. Falls keine Nachricht vorhanden ist, wartet diese Funktion nicht ab, bis dies der Fall ist, sondern liefert als Resultat die Information, daß zur Zeit keine Nachricht anliegt. Genau diese Funktion kann man auch gleichzeitig sehr gut verwenden, um während einer längeren Berechnung Nachrichten an den eigenen Prozeß zu empfangen, was man als eine Form des Multithreading bezeichnen könnte (nur daß hier nicht zwei Stacks vorhanden sind). Wenn der Aufruf der Funktion also ergab, daß eine Nachricht anliegt, muß man die entsprechende Funktion zur Bearbeitung der Nachricht aufrufen, wobei diese Funktion dann allerdings wirklich nur relativ kurze Zeit zur Ausführung benötigen darf, denn solange sie nicht terminiert, kann die ursprüngliche Berechnung nicht fortgesetzt werden.

Der letzte Kommentar zu "man_exec_procedure()" in Kapitel 3.3.1 beschäftigte sich schon mit dem Thema, was passiert, wenn "man_exec_procedure()" nochmals aufgerufen wird, wenn es schon läuft. Dort wurde das Vorgehen beschrieben, das zwar die dort gewünschte Funktion hat, aber trotzdem den erneuten Aufruf nach nur wenigen Befehlen terminieren läßt, so daß obige Forderung in FUZZEKS erfüllt werden kann. Die Darstellungsalgorithmen der Fenster, die bei sehr hohen Auflösungen die Bearbeitung der Nachricht "Bitte Fenster neu darstellen!" ein paar Sekunden dauern lassen können, während derer nicht weiterberechnet werden kann, lassen (wieder mit Hilfe obiger Methode) zumindest andere Prozesse zwischendurch zur Ausführung gelangen.

Diese Vorgehensweise ist immer noch etwas mühsam zu implementieren, da man an zeitlich möglichst regelmäßig verteilten Stellen bei den entsprechenden Algorithmen die Abfrage der Nachrichtensituation einbauen muß. Dies wurde durch die Implementierung einer Funktion erleichtert, die nur bei jedem *k*-ten Aufruf tatsächlich die Abfrage der Nachrichtensituation (die sonst nennenswerte Zeit kostet) ausführt. In den Algorithmen kann also dieser Aufruf in den fast innersten Schleifen geschehen, nur muß man vor diesen Schleifen die Berechnung eines geeigneten *k* einbauen. Dieses *k* ergibt sich aus der Zeit, die solch ein Schleifendurchlauf benötigt, i.a. kann sie einfach mit Hilfe der Problemgrößen abschätzen. Wenn man diese Aufrufe nicht in die innerste Schleife aufnehmen muß (was tatsächlich in FUZZEKS nie der Fall war), fällt die zusätzliche Rechenzeit, die für dieses Vorgehen benötigt wird, gering aus. In Kapitel 3.3.3 wurde schon erwähnt, daß man diese Stellen auch nutzen kann, um den Fortschritt der Berechnungen zu ermitteln und anzuzeigen.

Bemerkung: Es ist übrigens von Vorteil, daß der Algorithmus zur Darstellung des Fensters die gerade laufende Berechnung unterbricht, denn die Alternative wäre, daß beide Algorithmen quasi-gleichzeitig ablaufen würden, d.h. daß die Fensterdarstellung erst später fertig werden könnte; aber auch die Berechnung hätte dabei keinen Zeitvorteil, außer wenn sie noch während der Darstellung beendet werden könnte, was aber nur sehr selten der Fall wäre, da die Darstellung viel weniger Rechenzeit verbraucht als die Berechnungen. Bei Mehrprozessorsystemen kann eine stärkere Parallelisierung allerdings aufgrund der *echten* Gleichzeitigkeit von Vorteil sein. Auf die Parallelisierung voneinander unabhängiger Berechnungen sind obige Überlegungen auch anwendbar.

Literaturverzeichnis

Akin, H. , Siemes, H., 1988
Praktische Geostatistik - Eine Einführung für den Bergbau und die Geowissenschaften
Springer-Verlag Berlin Heidelberg 1988

Bandemer, H., Gottwald, S., 1993
Einführung in Fuzzy-Methoden
Akademie Verlag, Berlin, 1993

Bardossy, A., Bogardi, I., Kelly, W.E., 1988
Imprecise (Fuzzy) Information in Geostatistics
Mathematical Geology, Vol. 20, No. 4, 1989, S. 287-309

Bardossy, A., Bogardi, I., Kelly, W.E., 1989
Geostatistics Utilizing Imprecise (Fuzzy) Information
Fuzzy Sets and Systems 31, 1989, S. 311-328

Bardossy, A., Bogardi, I., Kelly, W.E., 1990
Kriging with Imprecise (Fuzzy) Variograms. I: Theory
Mathematical Geology, Vol. 22, No. 1, 1990, S. 63-79

Bardossy, A., Bogardi, I., Kelly, W.E., 1990
Kriging with Imprecise (Fuzzy) Variograms. II: Application
Mathematical Geology, Vol. 22, No. 1, 1990, S. 81-94

Bauch, H., Jahn, K.-U., Oelschlägel, D., Süße, H., Wiebigke, V., 1987
Intervallarithmetik
BSB B.G. Teubner Verlagsgesellschaft, Leipzig, 1987

Burrough, P.A., 1989
Fuzzy mathematical methods for soil survey and land evaluation
Journal of Soil Science, 1989, 40, S. 477-492

Cressie, N.A.C., 1993
Statistics for Spatial Data
Wiley-Interscience, 1993

W. Dong, H.C. Shah, 1987
Vertex Method for Computing Functions of Fuzzy Variables
Fuzzy Sets and Systems 24, 1987, S. 65-78

Dubois, D., Prade, H., 1987
Fuzzy Numbers: An Overview
In Analysis of Fuzzy Information, Volume I (Mathematics and Logic), Ed. J.C. Bezdek, 1987, S. 3-39

Heinrich, U., 1992
Zur Methodik der räumlichen Interpolation
mit geostatistischen Verfahren
Deutscher Universitäts Verlag, 1992
(Zugl.: Kiel, Univ., Diss. 1990, u.d.T.: Heinrich, Uwe:
Untersuchungen zur Validität flächenhafter Schätzungen
diskreter Messungen kontinuierlicher raumzeitlicher Prozesse)

Huajun, T., Debaveye, J., Da, R., van Ranst, E., 1991
Land Suiatbility Classification Based on Fuzzy Set Theory
Pedologie, XLI-3, Ghent, 1991, S. 277-290

Kandzia, P., 1995
Fuzzy-Methoden
Vorlesungsmitschrift, Institut für Informatik,
Christian-Albrechts-Universität zu Kiel, 1995

Kruse, R., Gebhardt, J., Klawonn, F., 1993
Fuzzy-Systeme
B.G. Teubner Stuttgart 1993

Microsoft, 1991
Handbücher (25 Stück) zum C/C++-Compiler der Version 7.0
Microsoft 1991

Neumaier, A., 1990
Interval methods for systems of equations
Cambridge University Press 1990

Piotrowski, J. A., Bartels, F., Salski, A., Schmidt, G., 1994
Fuzzy Kriging of Imprecise Hydrogeological Data
International Association for Mathematical Geology, Annual
Conference Mont Tremblant, Quebec, Canada,
October 1-5, 1994; Proceedings, S. 282-288

Piotrowski, J. A., Bartels, F., Salski, A., Schmidt, G., 1995
Fuzzy logic in hydrogeology - closer to nature?
9 Int. Conf. on the state of the Art of Ecological Modelling
(ISEM'95), 11-15 August 1995, Beijing, China; bstracts, S. 91

Piotrowski, J. A., Bartels, F., Salski, A., Schmidt, G., 1995
Geostatistische Regionalisierung hydrogeologischer Parameter
mit Fuzzy Kriging
62. Tagung der Arbeitsgem. Nordwestdeutscher Geologen,
Hamburg-Bergedorf, Tagungsband, 12-19, S. 12-19

Piotrowski, J. A., Bartels, F., Salski, A., Schmidt, G., 1996
Fuzzy-Kriging-Regionalisierung hydrogeologischer Parameter
In: Merkel, B., Dietrich, P.G., Struckmeier, W. & L Löhnert, E.P. (Hrsg.): Grundwasser und Rohstoffgewinnung. GeoCongress 2, Verlag Sven von Loga, Köln, S. 400-405.

Piotrowski, J. A., Bartels, F., Salski, A., Schmidt, G., 1996
Geostatistical regionalization of glacial aquitard thickness in northwestern Germany, based on fuzzy kriging
Mathematical Geology 28(4), S. 437-452

Piotrowski, J. A., Bartels, F., Salski, A., Schmidt, G., 1996
Estimation of hydrogeological parameters for groundwater modelling with fuzzy geostatistics: closer to nature?
In: Kovar, K. & van der Heijde, P. (eds.) Calibration and Reliability in Groundwater modelling; Int. Conf.ModelCARE'96, Golden (Colorado), USA, 24.-26. Sept. 1996, Proceedings, S. 511-520

Rumohr, S., 1996
Die Grundwasserdynamik zwischen Bornhöveder Seenkette und dem großen Plöner See
Doktorarbeit an der Mathematisch-Naturwissenschaftlichen Fakultät (Geologosches Institut) der Christian-Albrechts-Universität zu Kiel, 1994

Schmidt, G., 1994
Regionalisierung mit Fuzzy-Kriging und Modellierung der subglazialen Grundwasserdynamik pleistozäner Sedimente im Großraum Kiel
Diplomarbeit an der Mathematisch-Naturwissenschaftlichen Fakultät (Geologosches Institut) der Christian-Albrechts-Universität zu Kiel, 1994

Stoer, J., 1983
Einführung in die Numerische Mathematik I
Springer-Verlag 1983

Zadeh, L.A., 1965
Fuzzy Sets
Information and Control 8, 1965, S. 338-353

Zimmermann, H.-J., 1991
Fuzzy Set Theory - and Its Applications
Kluwer Academic Press, Second Edition, 1991

Zirschky, J.H., 1984
Spatial Analysis of Hazardous Waste Data using Geostatistics
Dissertation, Graduate School of Clemson University, 1984

Erklärung

Hiermit erkläre ich, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 4.2.1997