FAKULTÄT FÜR !NFORMATIK

FuzzyData

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christian Moerz
Matrikelnummer 9825832

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.-Prof. Dr.-Ing. Gerhard H. Schildt

Wien, 02. Juli 2009

_____ (Unterschrift Verfasser/in)

_____ (Unterschrift Betreuer/in)

Abstract

Eines der vorrangigen Probleme in der Messtechnik ist die bei jeder Form von Messung üblicherweise auftretende Unschärfe. Die mit Mengen arbeitende Fuzzy-Logik, die daran anknüpfenden unscharfen Zahlen und die jeweils darauf aufbauenden Methoden machen sich diese Eigenschaft zu Nutze.

Im Rahmen dieser Arbeit wird anfangs eine Übersicht über die Grundlagen unscharfer Zahlen gegeben und die dafür existierenden statistischen Auswertungsmethoden beleuchtet. Danach wird das Softwareprogramm „FuzzyData“ vorgestellt, welches als praktisches Anwendungsbeispiel der definierten statistischen Methoden dient. Bei der Implementierung von FuzzyData lag ein starker Fokus auf einer effizienten Speicher- und Ressourcennutzung; daher wurde – auch wegen der relativ guten Portabilität – als Implementierungssprache C/C++ gewählt.

Das Programm bietet die typischen statistischen Auswertungsmethoden wie z.B. Berechnung von

- Minimum,
- Durchschnitt,
- Maximum,
- Empirische Verteilungsfunktion
- Streuung,
- Varianz und
- Histogramm-Aufbereitung einer Stichprobe zwecks 3D-Visualisierung

Zusätzlich gestattet das Programm die Erzeugung von auf Zufallszahlen beruhenden, normalverteilten Stichproben.

Danksagung

Ich möchte mich herzlich bei allen, die mich in den Jahren meines Studiums unterstützt haben, bedanken. Ohne deren Hilfe hätte diese Arbeit nicht entstehen können. Dies inkludiert meine Eltern und Großeltern ebenso wie meinen Bruder.

Allen voran gilt mein Dank besonders Univ.-Prof. Dr.-Ing. Gerhard H. Schildt, der mich über all die Jahre, die ich an der TU Wien studierte, konstant unterstützte und motivierte.

Inhalt

1 Einführung und Grundlagen

Eines der vorrangigen Probleme in der Messtechnik ist die bei jeder Form von Messung üblicherweise auftretende Unschärfe. Es existieren unterschiedliche Ansätze um diesem – nicht rein der Messtechnik vorbehaltenen – Phänomen entgegenzuwirken. In diversen Wissenschaftszweigen wird Ungenauigkeit einfach durch eine (meist zu vernachlässigende) Größe bzw. Variable mit in Berechnungen einbezogen. Kaum ein Ansatz hat jedoch sowohl die Computertechnik als auch die Mathematik in Kombination so stark beeinflusst wie die Fuzzy-Theorie.

Prinzipiell besitzt die Fuzzy-Theorie viele Einsatzbereiche: Neben der bereits beschriebenen Messtechnik findet sie zum Beispiel auch in der Regelungs- und sogar in der Datenbanktechnik Verwendung. Die theoretischen Verwendungsmöglichkeiten sind vielfältig: im Architektur- und Baustatik-Bereich, bei Datamining und Wissensrepräsentation, in den Gebieten der Geophysik und Meteorologie etc. Die eigentliche Überlegung, die den Fuzzy-Zahlen zugrunde liegt, ist nun schon mehrere Jahrzehnte alt; dennoch besitzt sie nach wie vor eine erstaunlicher Aktualität. Moderne Prozessoren und der Preisverfall von Speicherbausteinen erlauben heute mehr denn je einen Einsatz von Computersystemen zur Berechnung und Analyse von komplexen unscharfen Daten.

1.1 Ursachen der Unschärfe

In gewissen Situationen ist es nicht effektiv, einen einzelnen distinkten Wert als das Ergebnis einer Messung oder einer Datenauswertung anzugeben. Dies trifft vor allem auf Messungen und Datenerhebungen zu, die in Umfeldern durchgeführt werden, in denen eine präzise Datenerhebung nur erschwert oder gar unmöglich ist. Die Unschärfe der vorliegenden Daten kann dabei aufgrund unterschiedlicher Faktoren hervorgerufen werden:

1. **Messung des gewünschten Wertes nicht möglich:** dies tritt vorrangig bei Messungen von Parametern auf, die sich aufgrund ablaufender physikalischer oder chemischer Prozesse ständig verändern (z.B. Temperatur, Druck, etc.). Für gewöhnlich kann nur eine daraus abgeleitete Größe gemessen werden.
2. **Messinstrumente verfügen nur über eine eingeschränkte Genauigkeit:** In der Medizin werden beispielsweise beim Aufzeichnen von Langzeit-EKGs oft tragbare Mini-Computer eingesetzt, die über Sensoren den Herzschlag des Trägers aufzeichnen. Diese Geräte können jedoch nicht eine konstante Kurve über den Herzschlag aufzeichnen sondern lediglich eine digitalisierte Variante davon, welche die Kurve mit Hilfe von an Abtastpunkten genommenen Werten repräsentiert.
3. **Modellunsicherheit:** Ein konkretes Beispiel für Unschärfe durch Modellunsicherheit findet man in der Finanzwelt. Die komplexen Zusammenspiele und Abhängigkeiten der verfügbaren Finanzinstrumente und wie sich diese gegenseitig beeinflussen wird von Banken oft in komplexen Modellen abgebildet. Dennoch können diese, aufgrund der Komplexität des realen Systems nur bis zu einem gewissen Grad die Realität widerspiegeln. Dadurch entsteht eine gewisse Unsicherheit und in Folge Unschärfe bei der Verwendung der Modelle (1).
4. **Daten können nicht genau angegeben werden:** Dieses Problem lässt sich etwa mit der Dauer der Wirkung eines Medikaments veranschaulichen. Sie kann nicht in Form einer einzelnen Zahl dargestellt werden, weil Medikamente einen schleichenden Übergang zwischen „nicht wirken“ und „wirken“ durchlaufen. Ähnlich stellt sich das Problem zum Beispiel dar, wenn man versucht, den konkreten Marktpreis eines Handelsartikels zu bestimmen. Die unterschiedlichen Anbieter des Artikels verlangen meist unterschiedliche Preise. Somit kann man nicht konkret von „dem“ Preis eines Artikels reden.

5. **Fehlende objektive Messskala:** Dieses Problem tritt etwa auf, wenn man versucht, eine menschliche Bewertung vorzunehmen. Versucht man die Temperatur von Wasser zu klassifizieren – ob dieses nun „heiß“, „warm“, „lauwarm“ oder gar „kalt“ ist – wird man keine konkret objektive Messskala dafür angeben können, da die Bewertung von subjektiven Faktoren abhängt.

Im Fall der ersten drei Punkte wird die Unschärfe meistens durch den Messvorgang verursacht; umgekehrt liegt den letzten beiden Punkten generell eine Unschärfe zugrunde. Im Bereich der Quantenphysik kommt zusätzlich die Heisenbergsche Unschärferelation (43) zu tragen: in diesem Fall ist es nicht immer möglich zwei Messgrößen eines Teilchens gleichzeitig beliebig genau zu bestimmen.

Sehr oft können Meßergebnisse allgemein nur mit einer oberen und unteren Schranke abgegrenzt werden. Solche als „Intervallzahlen“ bezeichneten Daten sind ein Spezialfall von „unscharfen Zahlen“.

Bei der Messung von eindimensionalen Daten treten generell vier Arten von Ungewissheit auf:

- Zufälligkeit
- Unschärfe
- Messfehler
- Modellunsicherheiten

Für gewöhnlich behilft man sich bei der Beschreibung der Ungewissheit mit auf Wahrscheinlichkeiten beruhenden stochastischen Modellen. Dies bezieht jedoch nur die Variabilität in die Modellbildung mit ein, während andere Arten der Ungewissheit außer Acht gelassen werden. Besonders in Bezug auf die Betrachtung der Unschärfe einer Beobachtung ist dies unzulänglich, da diese nicht von der stochastischen Komponente erfasst wird. Sie kann allerdings mit Hilfe von unscharfen Zahlen wieder in die Modellbildung eingebracht werden.

1.2 Definition unscharfer Mengen

In der diskreten Mengenlehre kann jede Zahl eindeutig einer Menge zugeordnet werden. Dies erfolgt allgemein für eine Menge I mit Hilfe einer Indikatorfunktion $I_M: \rightarrow \{0,1\}$. Für eine Menge A in der Zahlenmenge $\mathbb{R}$ lässt sich I etwa definieren als

$$I_A(x) = \begin{cases} 1 \text{ für } x \in A \\ 0 \text{ für } x \notin A \end{cases} \quad \forall x \in \mathbb{R}$$

Für alle zugehörigen Werte gibt die Indikatorfunktion den Wert 1, für alle nicht zugehörigen Werte das Ergebnis 0 zurück.

Beispiel. Die Menge aller reeller Zahlen zwischen 0 und 5 sei definiert als:

$$\mathrm{M} = \{\mathrm{x} | 0 < x < 5, x \in \mathbb{R}\} =]0; 5[$$

Für die Zahlen 2 und 6 ergibt sich somit folgendes Ergebnis beim Einsetzen in die Indikatorfunktion:

$$2 \in \mathrm{M} \Longrightarrow \mathrm{I_M}(2) = 1$$

$$6 \notin \mathrm{M} \Longrightarrow \mathrm{I}_{\mathrm{M}}(6) = 0$$

Somit ist in der klassischen Mengenlehre jede Zahl eindeutig entweder zu einer Menge zugehörig oder eindeutig nicht zugehörig.

Im Jahre 1951 publizierte Karl Menger (1902-1985) eine Erweiterung und Verallgemeinerung dieses Mengenkonzeptes (3) und schuf so die Idee der unscharfen Mengen (4). Er bezeichnete diese als „ensemble flous“. Der generelle Gedankengang dahinter war, die Zuordnung zu einer Menge nicht mehr durch ein distinktes „ja“ oder „nein“ zu gestalten sondern eine Abstufung zwischen den beiden zu ermöglichen. Er verallgemeinerte somit die Indikatorfunktion und dessen Idee wurde 14 Jahre später von L. A. Zadeh (*1921) aufgegriffen (5) (6). Er gab dieser Form von Teilmengen den Namen „unscharfe Mengen“ und bezeichnete die Verallgemeinerung der Indikatorfunktion als „Zugehörigkeitsfunktion“. Folglich kann jeder Wert nicht mehr nur mit 0 oder 1 als zugehörig oder nicht zugehörig klassifiziert werden, sondern durch einen belieben Wert aus dem Intervall [0,1].

Der daraus entwickelte Begriff „Fuzzy-Logik“ entstand, da mit Hilfe dieser Fuzzymengen Aussagen im Sinne einer mehrwertigen Logik möglich wurden.

1.3 Definition unscharfer Zahlen

Die Definition einer unscharfen Zahl x^* erfolgt durch die Angabe ihrer charakterisierenden Funktion $\xi_{x^*}(\cdot)$. Diese weist jedem Wert aus $\mathbb{R}$ einen eindeutigen Wert im Intervall von [0,1] zu (7), (4).

Für $\xi_{x^*}(\cdot)$ gilt:

1. $\xi_{x^*}(\cdot)\colon \mathbb{R} \to [0,1]$
2. $\forall\, \delta \in (0,1]$ ist der δ-Schnitt („Deltaschnitt“) definiert als endliches, nichtleeres und abgeschlossenes Intervall

$$C_\delta(x^*) \coloneqq \{x \in \mathbb{R}\colon \xi_{x^*}(x) \geq \delta\} = [a_\delta, b_\delta]$$

Die Menge der unscharfen Zahlen x^* bezeichnet man mit $\mathcal{F}(\mathbb{R})$. Der Träger der charakterisierenden Funktion $\xi_{x^*}(\cdot)$ besteht aus allen Elementen des Merkmalraums mit Zugehörigkeitswert $\xi_{x^*}(x) > 0$ und definiert sich als

$$Tr(x^*) \coloneqq \{x \in \mathbb{R}\colon \xi_{x^*}(x) > 0\}$$

1.1 Deltaschnitt

Die charakterisierende Funktion $\xi_{x^*}(\cdot)$ einer unscharfen Zahl x^* kann eindeutig durch ihre „Deltaschnitte” definiert werden (8). Diese sind die Menge der Schnittpunkte einzelner Deltaebenen mit der charakterisierenden Funktion. Die Familie der Deltaschnitte $(C_\delta(x^*); \delta \in (0,1])$ steht somit direkt im Zusammenhang mit der charakterisierenden Funktion:

$$\xi_{x^*}(\cdot) = \max_{\delta \in (0,1]} \delta I_{C_\delta(x^*)}(x)$$

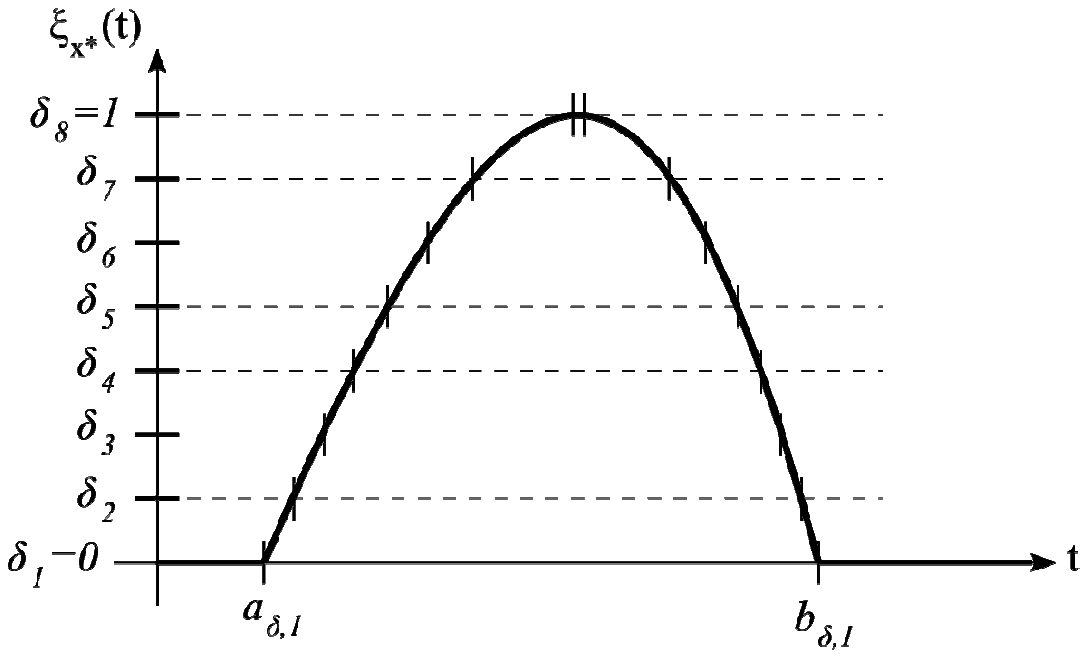

Abbildung 1.1: Beispiel für Deltaschnitt-Intervalle einer beliebigen unscharfen Zahl

1.2 Arten unscharfer Zahlen

Mit Hilfe von charakterisierenden Funktionen können nun die vier unterschiedlichen Klassen von unscharfen Zahlen generieren:

1. Intervallzahlen
2. Dreieckszahlen
3. Trapezzahlen und
4. Polygonzahlen

1.2.1 Intervallzahlen

Intervallzahlen sind durch ein Intervall zwischen zwei reellen Werten definiert, in dem die charakterisierende Funktion 1 ist. Es gilt für die Intervallzahl $x^* = [x_1, x_2]$

$$\xi_{x^*}(x) = \begin{cases} 0 & \forall x < x_1 \\ 1 & \forall x_1 \leq x \leq x_2 \\ 0 & \forall x_2 < x \end{cases}$$

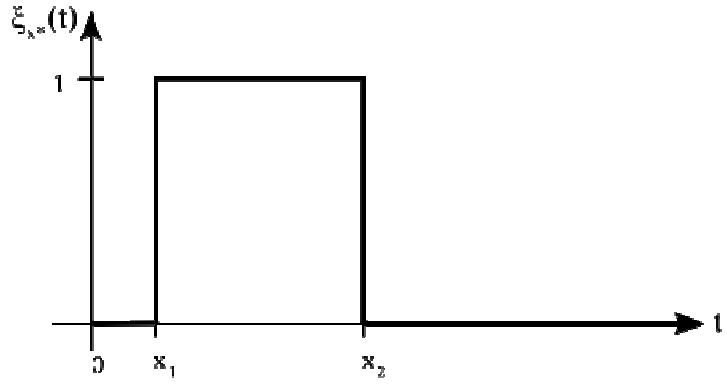

Abbildung 1.2: Intervallzahl

1.2.2 Dreieckszahlen

Dreieckszahlen, auch LR-Zahlen genannt (9), definieren sich durch eine charakterisierende Funktion in der Form eines Dreiecks. Für die Dreieckszahl $x^* = (x_1, x_2, x_3)$ gilt

$$\xi_{x^*}(x) = \begin{cases} \dfrac{x - x_1}{x_2 - x_1} & \forall x_1 \leq x \leq x_2 \\ 1 - \dfrac{x - x_2}{x_3 - x_2} & \forall x_2 < x \leq x_3 \\ 0 & \text{sonst} \end{cases}$$

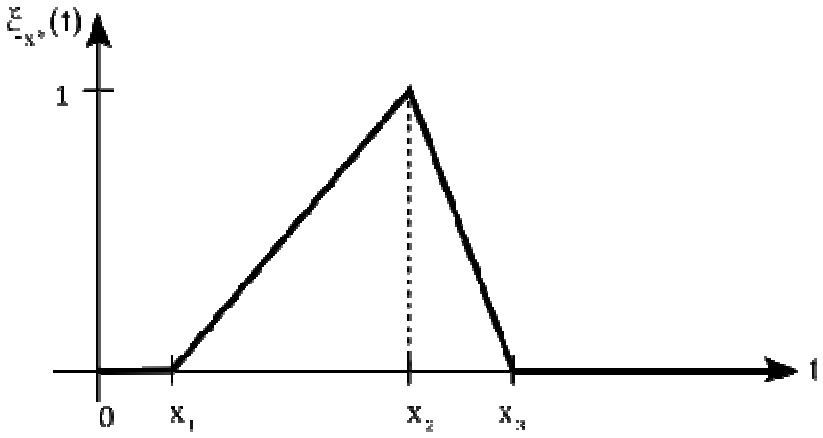

Abbildung 1.3: Dreieckszahl

1.2.3 Trapezzahlen

Eine Trapezzahl $x^* = (x_1, x_2, x_3, x_4)$ definiert sich anhand ihrer charakterisierenden Funktion

$$\xi_{x^*}(x) = \begin{cases} \dfrac{x - x_1}{x_2 - x_1} & x_1 \leq x < x_2 \\ 1 & x_2 \leq x \leq x_3 \\ 1 - \dfrac{x - x_3}{x_4 - x_3} & x_3 < x \leq x_4 \\ 0 & \text{sonst} \end{cases}$$

Diese ist in Abbildung 1.4 veranschaulicht.

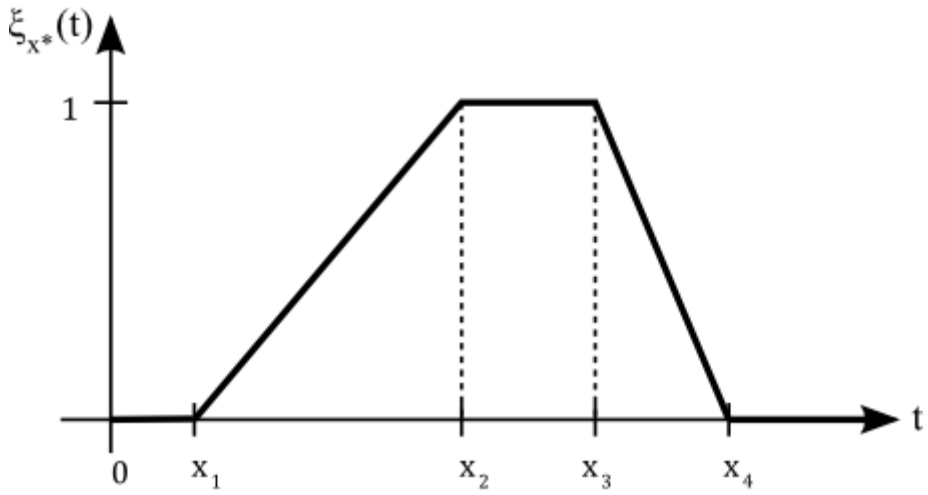

Abbildung 1.4: Trapezzahl

Prinzipiell kann man auch Intervall- und Dreieckszahlen in Form einer Trapezzahl definieren:

- Intervallzahl $x^* = [x_1, x_4] = (x_1, x_1 = x_2, x_3 = x_4, x_4)$
- Dreieckszahl $x^* = (x_1, x_3, x_4) = (x_1, x_2 = x_3, x_3, x_4)$

1.2.4 Polygonzahlen

Eine Polygonzahl ist eine polygonförmige, unscharfe Zahl, die sich aus einer Stützstellen-Punktmenge definiert. D.h., eine Polygonzahl x^* definiert sich als

$$x^* = \{(x_1, y_1 = \xi_{x^*}(x_1) = 0), (x_2, y_2 = \xi_{x^*}(x_2)), (x_3, y_3), \dots, (x_n, y_n = \xi_{x^*}(x_n) = 0)\}$$

Die charakterisierende Funktion $\xi_{x^*}(x)$ wächst bzw. fällt linear zwischen den Punkten

$$(x_i, y_i) \text{ und } (x_{i+1}, y_{i+1}) \text{ wobei } i = 1(1)n$$

und es gilt

$$\xi_{x^*}(x) = 0 \begin{cases} \forall\, x < x_1 \\ \\ \forall x > x_n \end{cases}$$

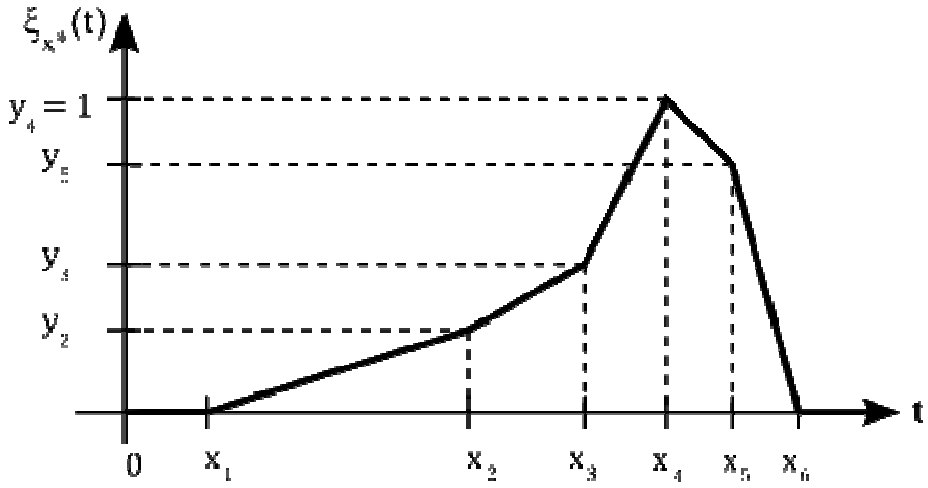

Abbildung 1.5: Polygonzahl

Vergleichbar zum Zusammenhang zwischen Intervall-, Dreiecks- und Trapezzahlen, können erstere auch in Form einer Polygonzahl definiert werden:

- Intervallzahl $x^* = [x_1, x_2] = \{(x_1, 0), (x_1, 1)(x_2, 1), (x_2, 0)\}$
- Dreieckszahl $x^* = (x_1, x_2, x_3) = \{(x_1, 0), (x_2, 1), (x_3, 0)\}$
- Trapezzahl $x^* = (x_1, x_2, x_3, x_4) = \{(x_1, 0), (x_2, 1), (x_3, 1), (x_4, 0)\}$

1.3 Funktionen auf unscharfen Zahlen

Das sogenannte „Erweiterungsprinzip“, das ursprünglich aus der Theorie unscharfer Mengen stammt, stellt die Grundlage für die Definition von Funktionen auf unscharfen Zahlen dar (7). Dieses besagt, dass für eine reelle Funktion $f: \mathbb{R}^n \to \mathbb{R}$ die Zugehörigkeitsfunktion $\xi_{y^*}(\cdot)$ des unscharfen Funktionswerts $y^* = f(x^*)$ für einen unscharfen Vektor x^* mit vektorcharakterisierender Funktion $\xi_{x^*}(\cdot, \dots, \cdot)$ definiert ist durch

$$\xi_{x^*}(y) := \begin{Bmatrix} \sup\{\xi_{x^*}(x): f(x) = y\} & \text{falls } \exists\, x \in \mathbb{R}^n : f(x) = y \\ \\ 0 & \text{falls } \nexists x \in \mathbb{R}^n : f(x) = y \end{Bmatrix} \quad \forall y \in \mathbb{R}$$

Für stetige Funktionen $f: \mathbb{R}^n \to \mathbb{R}$ ist der Funktionswert $y^* = f(x^*)$ eine unscharfe Zahl. Es gilt damit:

1. $y^* = f(x^*)$ ist eine unscharfe Zahl
2. Die Deltaschnitte von y^* sind definiert durch

$$C_\delta(y^*) = \left[\min_{x \in C_\delta(x^*)} f(x)\,,\ \max_{x \in C_\delta(x^*)} f(x) \right] \quad \forall \delta \in (0,1]$$

Auf Basis dieser Feststellung können nun Funktionen auf unscharfe Zahlen definiert werden.

1.3.1 Addition zweier unscharfer Zahlen

Die Addition $x_1^* \oplus x_2^*$ zweier unscharfer Zahlen wird über die stetige Funktion $f(x_1, x_2) = x_1 + x_2$ definiert. Die charakterisierende Funktion $\xi_{x^*}(\cdot)$ der verallgemeinerten Summe $x^* = x_1^* \oplus x_2^*$ wird definiert mit

$$\begin{aligned}\xi_{x^*}(x) &= \xi_{x_1^* \oplus x_2^*}(x) \\ &= \sup\left\{\min \xi_{x_1^*}(x_1), \xi_{x_2^*}(x_2) \,\middle|\, (x_1, x_2) \in \mathbb{R}^2 \text{ und } x_1 + x_2 = x\right\} \\ &= \sup\left\{\min \xi_{x_1^*}(y), \xi_{x_2^*}(x - y) \,\middle|\, y \in \mathbb{R}\right\} \;\forall x \in \mathbb{R}\end{aligned}$$

Der Delta-Schnitt ist definiert als

$$\begin{aligned}C_\delta(x^*) &= \left[\min_{(x_1,x_2) \in C_\delta(x_1^*) \times C_\delta(x_2^*)} x_1 + x_2 \,,\; \max_{(x_1,x_2) \in C_\delta(x_1^*) \times C_\delta(x_2^*)} x_1 + x_2\right] \\ &= \left[\underline{x}_{1,\delta} + \underline{x}_{2,\delta}, \overline{x}_{1,\delta} + \overline{x}_{2,\delta}\right] \qquad \forall \delta \in (0,1]\end{aligned}$$

Die Werte $\underline{x}_{1,\delta}, \underline{x}_{2,\delta}, \overline{x}_{1,\delta}$ und $\overline{x}_{2,\delta}$ kennzeichnen die jeweiligen Unter- und Obergrenzen der Deltaschnitte (siehe Abbildung 1.6).

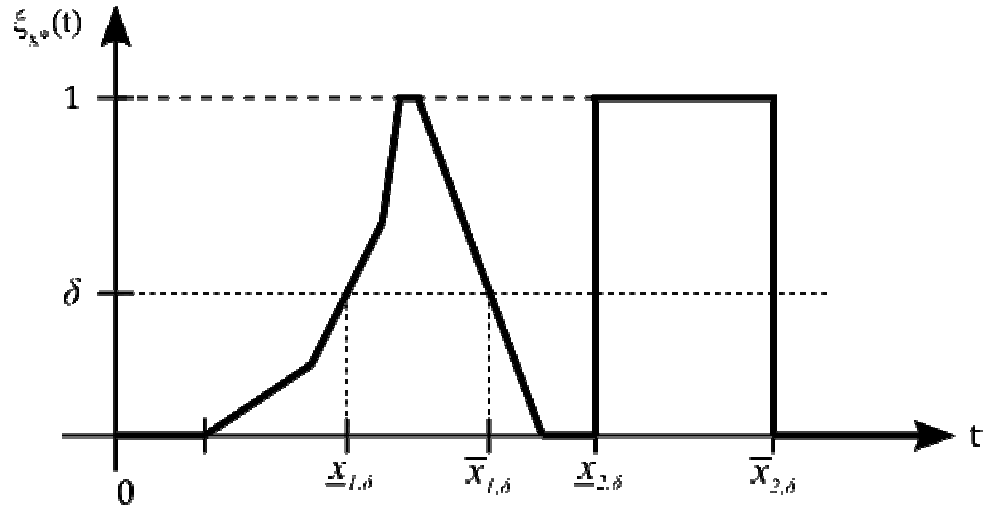

Abbildung 1.6: Ober- und Untergrenzen eines Deltaschnitts für zwei beliebige unscharfe Zahlen

1.3.2 Multiplikation zweier unscharfer Zahlen

Die Multiplikation $x_1^* \odot x_2^*$ zweier unscharfer Zahlen wird über die stetige Funktion $f(x_1, x_2) = x_1 \cdot x_2$ definiert (7):

$$\begin{aligned}\xi_{x^*}(x) &= \xi_{x_1^* \odot x_2^*}(x) \\ &= \sup\left\{\min\left\{\xi_{x_1^*}(x_1), \xi_{x_2^*}(x_2)\right\} \,\middle|\, (x_1, x_2) \in \mathbb{R}^2 \text{ und } x_1 \cdot x_2 = x\right\}\end{aligned}$$

Eine Skalarmultiplikation $\lambda \odot x^*$ einer unscharfen Zahl x^* mit einem Skalar $\lambda \in \mathbb{R}$ wird definiert über die charakterisierende Funktion

$$\xi_{y^*}(y) = \xi_{\lambda \odot x^*}(y)$$
$$= \sup\{\xi_{x^*}(x) | x \in \mathbb{R} \text{ und } \lambda \cdot x = y\} = \begin{cases} \xi_{x^*}(\lambda^{-1} y) & \text{für } \lambda \neq 0 \\ I_{\{0\}}(y) & \text{für } \lambda = 0 \end{cases}$$
$$\forall y \in \mathbb{R}$$

Die Skalarmultiplikation wird vereinfacht ausgedrückt Deltaschnitt für Deltaschnitt aufgebaut: für jeden Deltaschnitt wird die untere und obere Intervallgrenze mit dem Skalar multipliziert und ergibt somit die Unter- und Obergrenze des Deltaschnitts des Ergebnis.

Für die Multiplikation zweier unscharfer Zahlen sind die Deltaschnitte definiert als

$$C_\delta(x^*) = \left[\min_{(x_1,x_2) \in C_\delta(x_1^*) \times C_\delta(x_2^*)} x_1 \cdot x_2 \, , \max_{(x_1,x_2) \in C_\delta(x_1^*) \times C_\delta(x_2^*)} x_1 \cdot x_2 \right]$$

Für die konkrete Berechnung werden folgende Fälle für $C_\delta(x_1^*) = \left[\underline{x}_{1,\delta}, \overline{x}_{1,\delta}\right]$ und $C_\delta(x_2^*) = \left[\underline{x}_{2,\delta}, \overline{x}_{2,\delta}\right]$ unterschieden (7):

1. Es liegen die Deltaschnitte beider unscharfer Zahlen im positiven Bereich mit $\underline{x}_{1,\delta} \geq 0$ und $\underline{x}_{2,\delta} \geq 0$, so berechnet sich der Deltaschnitt von x^* durch
$$C_\delta(x^*) = \left[\underline{x}_{1,\delta} \cdot \underline{x}_{2,\delta}, \overline{x}_{1,\delta} \cdot \overline{x}_{2,\delta}\right]$$
2. Liegt der Deltaschnitt einer unscharfen Zahl vollständig im positiven und der Deltaschnitt der anderen Zahl vollständig im negativen Bereich, d.h., $x_{1,\delta} \geq 0$ und $x_{2,\delta} \leq 0$, so gilt
$$C_\delta(x^*) = \left[\overline{x}_{1,\delta} \cdot \underline{x}_{2,\delta}, \underline{x}_{1,\delta} \cdot \overline{x}_{2,\delta}\right]$$
3. Befinden sich die Deltaschnitte beider unscharfer Zahlen vollständig im negativen Bereich, d.h., $\overline{x}_{1,\delta} \leq 0$ und $\overline{x}_{2,\delta} \leq 0$, so berechnet sich der δ-Schnitt von x^* durch
$$C_\delta(x^*) = \left[\overline{x}_{1,\delta} \cdot \overline{x}_{2,\delta}, \underline{x}_{1,\delta} \cdot \underline{x}_{2,\delta}\right]$$
4. Liegen die δ-Schnitte beider unscharfer Zahlen sowohl im positiven als auch negativen Bereich, d.h., $\underline{x}_{1,\delta} \leq 0$, $\overline{x}_{1,\delta} \geq 0$, $\underline{x}_{2,\delta} \leq 0$ und $\overline{x}_{2,\delta} \geq 0$, so gilt
$$C_\delta(x^*) = \left[\min\left\{\underline{x}_{1,\delta} \cdot \overline{x}_{2,\delta}, \overline{x}_{1,\delta} \cdot \underline{x}_{2,\delta}\right\}, \max\left\{\underline{x}_{1,\delta} \cdot \underline{x}_{2,\delta}, \overline{x}_{1,\delta} \cdot \overline{x}_{2,\delta}\right\}\right]$$
5. Fällt der Deltaschnitt einer unscharfen Zahl vollständig in den positiven Bereich und der Deltaschnitt der zweiten unscharfen Zahl sowohl in den positiven, als auch in den negativen Bereich, dann berechnet sich der Deltaschnitt $C_\delta(x^*)$ als
$$C_\delta(x^*) = \left[\overline{x}_{1,\delta} \cdot \underline{x}_{2,\delta}, \overline{x}_{1,\delta} \cdot \overline{x}_{2,\delta}\right]$$
6. Fällt der Deltaschnitt einer unscharfen Zahl vollständig in den negativen Bereich und der Deltaschnitt der zweiten unscharfen Zahl sowohl in den positiven, als auch in den negativen Bereich, dann berechnet sich der Deltaschnitt $C_\delta(x^*)$ als

$$C_\delta(x^*) = \left[\underline{x}_{1,\delta} \cdot \overline{x}_{2,\delta}, \underline{x}_{1,\delta} \cdot \underline{x}_{2,\delta}\right]$$

1.3.3 Subtraktion zweier unscharfer Zahlen

Mit Hilfe der zuvor definierten Operationen Addition und Skalarmultiplikation kann nun die Subtraktion zweier unscharfer Zahlen definiert werden als:

$$-x^* = (-1) \odot x^* \implies \xi_{-x^*}(x) = \xi_{x^*}(-x)\ \forall x \in \mathbb{R}$$

$$x_1^* \ominus x_2^* := x_1^* \oplus (-x_2^*)$$

1.3.4 Arithmetische Mittelwert

Der arithmetische Mittelwert y^* von n unscharfen Zahlen $x_1^*, x_2^*, \dots, x_n^*$ berechnet sich aus den Deltaschnitten $C_\delta(x_k^*) = \left[\underline{x}_{k,\delta}, \overline{x}_{k,\delta}\right]$ der unscharfen Beobachtungen (7) mit Hilfe der Additions-Operation $\oplus$ für unscharfe Zahlen:

$$C_\delta(y^*) = C_\delta\left(\frac{1}{n} \odot \left(\bigoplus_{k=1}^{n} x_k^*\right)\right) = \left[\frac{1}{n}\sum_{k=1}^{n} \underline{x}_{k,\delta}, \frac{1}{n}\sum_{k=1}^{n} \overline{x}_{k,\delta}\right]$$

Hierbei wird für jeden Deltaschnitt die Unter- und Obergrenze aller Beobachtungen errechnet; danach wird gesondert für die Unter- und Obergrenzen das arithmetische Mittel berechnet. Die beiden Ergebnisse werden dann als Unter- und Obergrenze des Ergebnis-Deltaschnitts angenommen. Die Menge aller Deltaschnitte definiert nach Abschluss der Berechnung das Ergebnis als unscharfe Zahl.

1.4 Statistische Auswertung unscharfer Daten

1.4.1 Häufigkeitsverteilung und Histogramm

Die Häufigkeitsverteilung einer Stichprobe wird im Allgemeinen durch ein Histogramm dargestellt. Hierzu wird der Merkmalraum M, d.h., der Raum der möglichen Versuchsausgänge bzw. Beobachtungen, in k disjunkte Klassen $K_1, K_2, \dots, K_k$ eingeteilt. Es gilt also

$$K_i \cap K_j = \emptyset \quad \forall 1 \le i < j \le k$$

$$\bigcup_{i=1}^{k} K_i = M$$

Die Breiten der einzelnen Klassen müssen nicht zwingend gleich groß sein und werden daher getrennt als $b_1, b_2, \dots, b_k$ bezeichnet. Im Normalfall werden jedoch gleichbreite, d.h., „äquidistante“ Klassen eingesetzt.

Erstellt man ein Histogramm für eine Stichprobe mit reellen Zahlen $x_1, x_2, \dots, x_n$, wird nun für jede der Klassen die relative Häufigkeit $\tilde{h}_n(K_i), i = 1(1)k$ der jeweils in ihr vorliegenden Stichprobenelemente berechnet. Es gilt also

$$\tilde{h}_n(K_i) = \frac{\{x_i | x_i \in K_i\}}{n}\ \forall i: 1 \le i \le k$$

Im Histogramm werden dann – wie in Abbildung 1.7 veranschaulicht – die durch die Klassenbreiten dividierten relativen Häufigkeiten

$$h_n(K_i) = \frac{\tilde{h}_n(K_i)}{b_i}$$

aufgetragen.

Liegt eine Stichprobe mit unscharfen Zahlen $x_1^*, x_2^*, \dots, x_n^*$ vor, wird ebenso die relative Häufigkeit berechnet. Diese wird jedoch über ihre Deltaschnitte definiert und errechnet (7):

$$C_\delta\left(h_n^*(K_i)\right) = \left[\underline{h}_{n,\delta}(K_i), \overline{h}_{n,\delta}(K_i)\right]$$

$$\underline{h}_{n,\delta}(K_i) = \frac{\left|\left\{x_j^* \middle| C_\delta\left(x_j^*\right) \subseteq K_i\right\}\right|}{n}$$

$$\overline{h}_{n,\delta}(K_i) = \frac{\left|\left\{x_j^* \middle| C_\delta\left(x_j^*\right) \cap K_i \neq \emptyset\right\}\right|}{n}$$

$\underline{h}_{n,\delta}(K_i)$ errechnet die Anzahl aller Beobachtungen, die vollständig in das Klassenintervall fallen, d.h., für den betrachteten Deltaschnitt alle jene Beobachtungen, deren Unter- und Obergrenze innerhalb des Klassenintervalls sind.

$\overline{h}_{n,\delta}(K_i)$ errechnet die Anzahl aller Beobachtungen, die entweder vollständig oder teilweise in das Klassenintervall fallen. Dies umfasst somit jene Deltaschnitte von Beobachtungen, die mit dem Klassenintervall einen nicht-leeren Durchschnitt bilden.

Möchte man das Ergebnis der Berechnungen in Form eines Histogramms zeichnen, muss man zuerst – im Fall von ungleichen Klassenbreiten – wie im diskreten Fall zuerst mit Hilfe der Skalarmultiplikation $\odot$ durch die jeweilige Klassenbreite dividieren.

$$\frac{1}{b} \odot h_n^*(K_i)$$

Das Ergebnis ist ein dreidimensionales Histogramm, auf dem die X-Achse die Klassen wiedergibt, die Y-Achse relative Häufigkeit und die Z-Achse den Grad der Unschärfe.

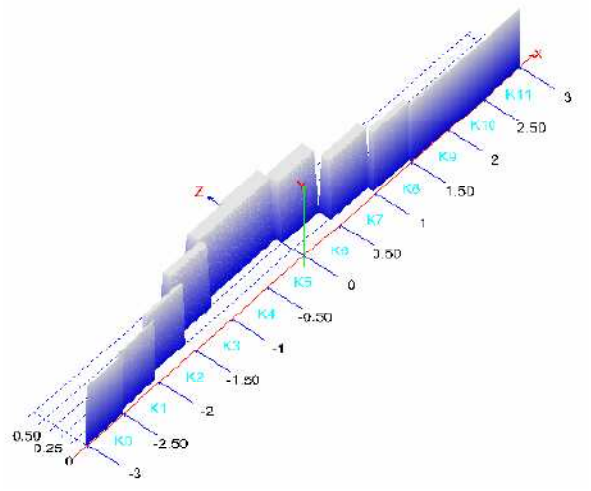

Abbildung 1.7: Dreidimensionales Histogramm einer Stichprobe unscharfer Zahlen

1.4.2 k-ten Momente von Verteilungen

Für einen Merkmalraum $M = \{a_1, a_2, \dots\}$ sei die Punktwahrscheinlichkeit für dessen Elemente definiert als die Wahrscheinlichkeit für ein konkretes Element des Merkmalraums:

$$p(a_i) = W(\{a_i\}) \; \forall a_i \in M$$

Der Erwartungswert, auch „Mittelwert“ genannt, einer diskret verteilten stochastischen Größe X mit Merkmalraum M_X und Punktwahrscheinlichkeiten $p(x)$ ist definiert durch (10)

$$\mathbb{E}\mathrm{X} = \sum_{\mathrm{x} \in \mathrm{M_X}} \mathrm{x} \cdot \mathrm{p(x)}$$

Für eine stochastisgche Größe X ergibt sich daraus das k-te Moment m^k als Erwartungswert von X^k. Für eine stetige Zufallsvariable X mit Dichte $f(\cdot)$ wird das k-te Moment folgend definiert (7):

$$m^k = \mathbb{E}(X^k) = \int_{\mathbb{R}} x^k f(x) dx$$

Und für eine diskrete Zufallsvariable X (7)

$$m^k = \mathbb{E}(X^k) = \sum_{x \in M_X} x^k p(x) dx$$

Das k-te empirische Moment $\hat{m}^k$ von n unscharfen Beobachtungen $x_1^*, x_2^*, \dots, x_n^*$ wird anhand seiner Deltaschnitte berechnet (7):

$$C_\delta((\hat{m}^k)^*) = \left[\frac{1}{n}\sum_{i=1}^{n} \min\left\{\max\left\{\underline{x}_{i,\delta}^k, 0\right\}, \overline{x}_{i,\delta}^k\right\}, \sum_{i=1}^{n} \max\left\{\underline{x}_{i,\delta}^k, \overline{x}_{i,\delta}^k\right\}\right]$$

Ebenso wie beim arithmetischen Mittel wird hier das Ergebnis Deltaschnitt für Deltaschnitt berechnet: für jeden Deltaschnitt wird für die Unter- und Obergrenze die jeweilige Summe gebildet. Die Gesamtmenge aller Deltaschnitt-Intervalle definiert zum Schluss das Endergebnis $\left(\widehat{m}^k\right)^*$.

1.4.3 Unscharfe Stichprobenvarianz

Die unscharfe Stichprobenvarianz, auch „zweites zentrales empirisches Moment“ genannt, berechnet sich durch

$$s_n^2(x_1, \dots, x_n) = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\overline{x}_n^2\right)$$

und ist über dessen Delta-Schnitte definierbar als (7)

$$C_\delta(s_n^{2\,*}) = \left[\min_{x \in C_\delta(x_1^*) \times \dots \times C_\delta(x_n^*)} s_n^2(x)\,,\ \max_{x \in C_\delta(x_1^*) \times \dots \times C_\delta(x_n^*)} s_n^2(x)\right]$$

Die Stichprobenvarianz s_n^2 der n Beobachtungen gestattet, die Stichprobenvarianz der ersten n-1 Beobachtungen rekursiv zu berechnen:

$$s_{n-1}^2 = \frac{n-1}{n-2}\left(s_n^2 - \frac{(\overline{x}_{n-1} - x_n)^2}{n}\right)$$

Das Stichprobenmittel $\overline{x}_{n-1}$ wird dabei berechnet durch

$$\overline{x}_{n-1} = \frac{n\overline{x}_n - x_n}{n-1}$$

Ein effektiver Algorithmus zur Berechnung der Stichprobenvarianz ist z.B. in (7) gegeben und trennt dabei in zwei grobe Schritte: die Berechnung der Unter- und Obergrenze der Deltaschnitte. Der konkrete Berechnungsweg ist im Folgenden angegeben.

1.4.3.1 Berechnung der Untergrenze

Zur Berechnung der Untergrenze werden die folgenden Schritte durchlaufen

1. Berechnung des Vektors, der aus den unteren Grenzen der Deltaschnitte der beteiligten einzelnen Beobachtungen bestimmt wird:

$$x_{\min} = \left(x_{\min,1}, \dots x_{\min,n}\right) = \left(\underline{x}_{1,\delta}, \dots, \underline{x}_{n,\delta}\right)$$

2. Es wird der Mittelwert $\overline{\boldsymbol{x}}_{\min}$ aus den einzelnen Vektorkomponenten berechnet.
3. Abhängig von der Lage der einzelnen Deltaschnitte zum berechneten Mittelwert, wird ein neuer Vektor $\hat{x}_{min}$ erstellt mit dem Ziel, einen möglichst nahe am Mittelwert liegenden Punkt aus den vorhandenden Deltaschnitten zu wählen. Für jede Beobachtung wird nun untersucht:
   a. Liegt $\overline{\boldsymbol{x}}_{\min,i}$ innerhalb eines Deltaschnitts einer unscharfen Beobachtung, so wird für die entsprechende Stelle $\hat{x}_{\min,i} = \overline{\boldsymbol{x}}_{\min,i}$ gewählt.
   b. Liegt $\overline{\boldsymbol{x}}_{\min,i}$ vollständig oberhalb bzw. rechts vom Deltaschnitt $C_\delta(x_i^*) = (\underline{x}_{i,\delta}, \overline{x}_{i,\delta})$ von einer unscharfen Beobachtung x_i^*, wird für $\hat{x}_{\min,i} = \overline{x}_{i,\delta}$ gewählt.

c. Liegt $\overline{\boldsymbol{x}}_{\min,i}$ vollständig unterhalb bzw. links vom Deltaschnitt $C_\delta(x_i^*) = (\underline{x}_{i,\delta}, \overline{x}_{i,\delta})$ von einer unscharfen Beobachtung x_i^*, wird für $\hat{x}_{\min,i} = \underline{x}_{i,\delta}$ gewählt.

4. $\overline{\boldsymbol{x}}_{min}$ wird gleich $\hat{x}_{min}$ gesetzt und es wird erneut bei Schritt 3 angefangen, solange bis sich keine wesentliche Veränderung mehr zeigt. Es kann z.B. abgebrochen wenn

$$\left|\hat{x}_{\min,i} - \overline{\boldsymbol{x}}_{\min,i}\right| < 10^{-7}$$

1.4.3.2 Berechnung der Obergrenze

Zur Berechnung der Untergrenze werden die folgenden Schritte durchlaufen

1. Berechnung des Vektors, der aus den unteren Grenzen der Deltaschnitte der beteiligten einzelnen Beobachtungen bestimmt wird:

$$x_{\max} = \left(x_{\max,1}, \dots x_{\max,n}\right) = \left(\overline{\mathrm{x}}_{1,\delta}, \dots, \overline{\mathrm{x}}_{n,\delta}\right)$$

2. Es wird der Mittelwert $\overline{\boldsymbol{x}}_{\max}$ aus den einzelnen Vektorkomponenten berechnet.
3. Jedes Vektorelement wird nun untersucht, ob es angepasst werden muss:
   a. Es wird der Mittelwert für $x_{\max,\mathrm{i}} = \overline{x}_{i,\delta}$ berechnet

$$\overline{x}_n = \frac{(n-1)\overline{x}_{n-1} + \overline{x}_{i,\delta}}{n}$$

   b. Danach wird die Varianz berechnet
   c. Es folgt die Berechnung des Mittelwerts für $x_{\max,\mathrm{i}} = \underline{x}_{i,\delta}$
   d. Und die Varianz
   e. Ist $\overline{s}_n^2 > \underline{s}_n^2$, so wird $x_{\max,i} = \overline{x}_{i,\delta}$ gesetzt, sonst $x_{\max,i} = \underline{x}_{i,\delta}$
4. $\overline{\boldsymbol{x}}_{\min}$ wird gleich $\hat{x}_{\min}$ gesetzt, und es wird erneut bei Schritt 3 angefangen, solange bis die Varianz nach einem vollständigen Durchlauf nicht mehr grösser wird.

1.4.4 Maximum

Das Maximum unscharfer Beobachtungen $x_1^*, \dots, x_n^*$ ist definiert durch seine δ-Schnitte (7):

$$C_\delta(x_{max}^*) = \left[\max_{i=1(1)n} \underline{x}_{i,\delta}, \max_{i=1(1)n} \overline{x}_{i,\delta}\right]$$

1.4.5 Minimum

Das Minimum unscharfer Beobachtungen $x_1^*, \dots, x_n^*$ ist gegeben durch seine Deltaschnitte (7)

$$C_\delta(x_{min}^*) = \left[\min_{i=1(1)n} \underline{x}_{i,\delta}, \min_{i=1(1)n} \overline{x}_{i,\delta}\right]$$

1.4.6 Geglättete empirische Verteilungsfunktion

Die empirische Verteilungsfunktion gibt für einen Wert x an, wie hoch die relative Häufigkeit aller Beobachtungen ist, die kleiner oder gleich x sind. Dies bedeutet jedoch, dass im Allgemeinen diese Funktion eine treppenförmige, nicht stetige Funktion ist.

Die geglättete empirische Verteilungsfunktion überbrückt diese Treppensprünge und ersetzt diese durch stetige Übergänge.

Liegen nun $x_1^*, \dots, x_n^*$ unscharfe Beobachtungen vor, mit zugehörigen charakterisierenden Funktionen $\xi_{x_1^*}(\cdot), \dots, \xi_{x_n^*}(\cdot)$, definiert sich die geglättete empirische Verteilungsfunktion mit (7)

$$\hat{F}_n^*(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{\int_{-\infty}^{x} \xi_{x_i^*}(t)dt}{\int_{-\infty}^{\infty} \xi_{x_i^*}(t)dt}$$

Handelt es sich um eine gemischte Menge aus l unscharfen Beobachtungen $x_1^*, \dots, x_l^*$ und k reellen Beobachtungen $y_1, \dots y_k$, wird zur Berechnung zuerst eine Unterteilung in scharfe und unscharfe Beobachtungen vorgenommen:

$$\hat{F}_n^*(x) = \frac{1}{n}\sum_{i=1}^{k} I_{(-\infty,x]}(y_i) + \frac{1}{n}\sum_{i=1}^{l} \frac{\int_{-\infty}^{x} \xi_{x_i^*}(t)dt}{\int_{-\infty}^{\infty} \xi_{x_i^*}(t)dt} \quad \forall x \in \mathbb{R}$$

Die Berechnung der uneigentlichen Integrale in diesen Formeln kann sehr leicht durchgeführt werden, da diese über die charakterisierenden Funktionen der Stichprobe durchgeführt werden. Die charakterisierenden Funktionen sind hierbei nur innerhalb eines endlichen Intervalls >0 und sonst überall 0. Damit reduziert sich die Berechnung auf eine Flächenberechnung der jeweiligen charaktierisierenden Funktionen, die selbst im Fall einer polygonförmigen Zahl mit linearem Aufwand berechnet werden kann.

1.4.7 Summenkurve

Die Konstruktion einer Summenkurve beruht prinzipiell auf demselben Schema wie die Konstruktion eines Histogramms: der gesamte Merkmalraum wird anfangs in l paarweise disjunkte Klassen $K_1, \dots, K_l$ eingeteilt, wobei die Klassen nicht notwendigerweise die gleichen Breiten besitzen müssen.

Der Wert der Summenkurve an der Stelle x ergibt sich somit für eine Stichprobe $x_1, \dots, x_n$ als die relative Häufigkeit aller Beobachtungen, die kleiner oder gleich x sind. Sind die Klassen $K_1, \dots, K_l$ definiert durch die Intervallgrenzen $a_1 = \underline{k}_1, a_2 = \overline{k}_1 = \underline{k}_2, \dots, a_{l+1} = \overline{k}_l$ gilt somit

$$S_n(a_i) = \frac{\left|\{x_j | x_j \le a_i\}\right|}{n} \quad \forall i = 1(1)l+1$$

Für eine Menge n unscharfer Beobachtungen $x_1^*, \dots, x_n^*$ mit zugehörigen charakterisierenden Funktionen $\xi_{x_1^*}(\cdot), \dots, \xi_{x_n^*}(\cdot)$ definiert sich die Summenkurve $S_n^*(\cdot)$ wie in (7) angegeben als

$$S_n^*(x) = \frac{\sum_{i=1}^{n} \int_{-\infty}^{x} \xi_{x_i^*}(t)dt}{\sum_{i=1}^{n} \int_{-\infty}^{\infty} \xi_{x_i^*}(t)dt}$$

Dabei ist zu beachten, dass dies nur gilt, wenn tatsächlich nur unscharfe Beobachtungen vorliegen. Enthält die Stichprobe auch reellwertige Beobachtungen, muss die Summenkurven-Berechnung auf die geglättete empirische Verteilungsfunktion umgelegt werden.

1.4.8 Empirische Fraktile der geglätteten empirischen Verteilungsfunktion

Für eine Verteilungsfunktion $F(\cdot)$ einer Wahrscheinlichkeitsverteilung auf $(\mathbb{R}, \mathcal{B})$ nennt man die reelle Zahl $x_p \forall p \in [0,1]$ ein p-Fraktiles oder p-Quantil wenn gilt (10)

$$F(x_p) = p$$

Somit ist x_p jener reeller Wert, an dem die geglättete Verteilungsfunktion den Wert p erreicht. Im Falle einer unscharfen empirischen Verteilungsfunktion lässt sich ein Fraktil über die invertierte empirische Verteilungsfunktion $\hat{F}^{-1}(\cdot;\cdot, \dots, \cdot)$ definieren

$$q_p^* = \left(\hat{F}_n^{-1}\right)^*(p) = \left(\hat{F}_n^{-1}\right)^*(p; x_1^*, \dots, x_n^*), \;\; p \in [0,1]$$

Für reelle Beobachtungen $x_1, \dots, x_n$ ist die invertierte empirische Verteilungsfunktion für alle $p \in [0,1]$ definiert als

$$\hat{F}_n^{-1}(p) = \hat{F}_n^{-1}(p; x_1, \dots, x_n) = \min\{z \in \mathbb{R}: \hat{F}_n(z) \geq p\}$$

Für unscharfe Beobachtungen kann diese über deren Deltaschnitte berechnet werden (7):

$$C_\delta(q_p^*) = \left[\min_{x \in C_\delta(x^*)} \hat{F}_n^{-1}(p; x)\,, \max_{x \in C_\delta(x^*)} \hat{F}_n^{-1}(p; x)\right]$$

Ein einfacher Weg, diese zu errechnen, ist für jeden Deltaschnitt die Unter- und Obergrenzen der Deltaschnitt-Intervalle der Beobachtungen zu errechnen und diese in zwei geordnete, aufsteigend sortierte Mengen einzuteilen: es wird also aus $x_1^*, \dots, x_n^*$ zu jeder Beobachtung $C_\delta(x_k^*) = \left[\underline{x}_k, \overline{x}_k\right], 1 \leq k \leq n$ errechnet und dann für gegebenes Delta die Mengen

$$Left_\delta = \left\{\underline{x}_{k,\delta} \middle| 1 \leq k \leq n \text{ und } \underline{x}_{k,\delta} < \underline{x}_{k+1.\delta} \forall 1 \leq k < n\right\}$$

$$Right_\delta = \left\{\overline{x}_{k,\delta} \middle| 1 \leq k \leq n \text{ und } \overline{x}_{k,\delta} < \overline{x}_{k+1,\delta} \forall 1 \leq k < n\right\}$$

zur Berechnung herangezogen. Es gilt nach (7), dass für das betrachtete δ für die Schnitt-Untergrenze des Fraktils jenes $\underline{x}_{k,\delta}$ aus $Left_\delta$ gewählt werden kann, das den kleinsten Wert k besitzt, für den gilt $k > np$. Das Gleiche gilt für die Auswahl der Schnitt-Obergrenze, wobei hier jenes $\overline{x}_{k,\delta}$ aus $Right_\delta$ ausgewählt wird, das den kleinsten Wert k besitzt, für den $k > np$ gilt.

2 Computerbasierte Unterstützung

Computerunterstützung eröffnet im Bereich der Fuzzy-Logik – ebenso wie in vielen anderen Wissenschaftszweigen – eine Vielzahl an neuen Möglichkeiten. Es mangelt nicht an Ideen und Konzepten: allerdings finden sich zumeist nur sehr spezialisierte und begrenzt verwertbare Implementierungen. Es wurden eine Vielzahl an Arbeiten veröffentlicht über Basisbereiche wie fuzzy-basiertes Schließen (11) (12) und unscharfe Logik, genauso wie Papers zu komplexen Themen zu Fuzzy Logik in Verbindung mit neuronalen Netzen (13) und Bild-Verarbeitung (14) (15) (16). Die Implementierung dieser Ideen ist vielversprechend – beispielsweise die Definition eines fuzzy-basierten Thesaurus (17); dennoch hinkt die effektive Entwicklung, vor allem aber auch die Adoption von generell einsetzbaren Werkzeugen und Bibliotheken hinterher.

Die folgenden Unterkapitel sollen einen rudimentären Überblick über aktuell verfügbare Werkzeuge im Bereich der Fuzzy-Logik geben; der Hauptaugenmerk liegt hierbei auf allgemein zugänglichen Softwarepaketen – vorzugsweise im Open Source Bereich. Die Liste erhebt keinen Anspruch auf Vollständigkeit.

2.1 FLIP++

Die „FLIP++“ Bibliothek wurde am DBAI Institut an der TU Wien entwickelt und ist Teil einer größeren Sammlung von Fuzzy-Logik Codekomponenten namens „StarFLIP++“. Die interaktiven, grafischen Werkzeuge im Paket sind für eine X11-Umgebung ausgelegt und können daher auf regulären Unix-Systemen betrieben werden; eine unter MS Windows lauffähige Version ist ebenso im Paket enthalten (18).

Das Gesamtpaket besteht aus folgenden Teilen:

- FLIP++: Fuzzy-Logik Inferenz-Bibliothek
- ConFLIP++: erlaubt die Definition von Fuzzy-Einschränkungen
- DynaFLIP++: Generierung von dynamischen Einschränkungen
- DomFLIP++: Repräsentation von Domänenwissen
- OptiFLIP++: heuritische Optimierungen/Algorithmen (aktuell Tabu-Seach und genetische Algorithmen)
- InterFLIP++: Graphisches User Interface für X Windows und MS Windows
- DocuFLIP++: HTML Dokumentation (Endbenutzer/Knowledge Engineer/Programmierer)
- CheckFLIP++: Konsistenz-Checker für Daten-Veränderungen
- TestFLIP++: Versionskontrolle und Testumgebung für das Gesamtpaket
- The simulation toolkit, SimFLIP++: Simulations-Toolkit
- The reactive optimizer, ReaFLIP++: „reaktives” Optimierungstool
- NeuroFLIP++: Erweiterung für neuronale Netze zwecks automatischem Feintuning von Zugehörigkeitsfunktionen

Der Code für StarFLIP++ (ausgenommen NeuroFLIP++, ReaFLIP++, SimFLIP++, TestFLIP++ und CheckFLIP++) steht auf der Projektwebseite (18) zum Download zur Verfügung. Dieser wurde unter der GPL veröffentlicht und kann entsprechend als Open Source Software weiterentwickelt bzw. zur Entwicklung neuer Software herangezogen werden. Die Bibliothek wurde jedoch seit 1997 nicht mehr weiter entwickelt.

```
http://www.dbai.tuwien.ac.at/proj/StarFLIP/
```

```
http://www.dbai.tuwien.ac.at/
```

2.2 Fool & Fox

Die Abteilung für „Process Control and Robotics“ an der Universität Oldenburg entwickelte 1999 „Fool & Fox“, das aus zwei Elementen besteht. Das erstere „Fool” (kurz für “Fuzzy Organizer Oldenburg”) getaufte Werkzeug ist ein graphisches Werkzeug zur Erstellung von Fuzzy-Regelbasen. Man kann damit eine Datenbank erstellen und warten, welche das Verhalten eines Fuzzycontrollers beschreibt. Das zweite Werkzeug „Fox” arbeitet mit der von Fool erstellten Datenbank, liest Daten von der Kommandozeile und schreibt neue Kontrollwerte zurück auf die Shell. Dies gestattet den Einsatz von Fuzzy-Routinen in nahezu jedem beliebigen Programm. Ebenso wie StarFLIP++ wurde Fool & Fox seit geraumer Zeit nicht mehr weiter entwickelt. Die letzte Aktualisierung fand 2002 statt (19).

```
http://sourceforge.net/projects/fool
http://www.Uni-Oldenburg.DE/
```

2.3 FuzzyJ and FuzzyJess

Das kanadische National Research Council des Institute for Information Technology entwickelte das NRC FuzzyJ Toolkit. Mit dessen Hilfe können unter Java generelle Fuzzy-Konzepte umgesetzt werden, wie etwa einfache Fuzzy Logik und Fuzzy Reasoning. Die Entwicklung basiert auf einer zuvor entwickelten CLIPS-Erweiterung namens FuzzyCLIPS.

Die entwickelte FuzzyJ Bibliothek kann sowohl alleinstehend eingesetzt werden als auch in Verbindung mit der von Sandia National Laboratories entwickelten Experten-Shell „Jess“.

Die letzte Version 1.10a wurde 2006 online gestellt. (20)

```
http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit2.html
```

2.4 Neuro-fuzzy Identification and Data Analysis Tool for Matlab

Gianluca Bontempi und Mauro Birattari entwickelten an der ULB Brüssel (Belgien) ein Matlab Software-Werkzeug zur fuzzy-basierten Analyse und Identifikation von Daten. Das Tool wurde 1999 zuletzt weiterentwickelt und ist auf die nicht mehr aktuelle Matlab Version 4.2 ausgerichtet (21).

```
http://iridia.ulb.ac.be/~gbonte/software/Local/FIS.html

http://iridia.ulb.ac.be/
```

2.5 Type-2 Fuzzy Logic Software

Jerry M. Mendel entwickelte an der University of Southern California (USA) zu seinem Buch „Uncertain Rule-Based Fuzzy Logic Systems“ ein Softwarepaket für Matlab, welches die im Buch beschriebenen Fuzzy-Methoden und –Konzepte umsetzt und veranschaulicht. Die letzte Überarbeitung wurde 2007 durchgeführt (22).

```
http://sipi.usc.edu/~mendel/software/

http://sipi.usc.edu/
```

2.6 Xfuzzy

XFuzzy wurde am Instituto de Microelectrónica de Sevilla (Spanien) entwickelt und ist eine Designumgebung für Fuzzy-Logiksysteme. Es bietet eine graphische Benutzeroberfläche, die eine Fuzzy-basierte Entwicklung und einfache Definition und Verifikation von Fuzzy-Daten gestatten.

XFuzzy nutzt dazu eine eigene textbasierte Sprache namens XFL; diese kann zur Spezifikation und Evaluierung genutzt werden. Weitere Elemente des Software-Pakets sind xfsim, mit dessen Hilfe das Verhalten von Fuzzy-Systemen verifiziert werden kann, und xfbpa, das die Konfiguration der Parameter einer XFL Definition erlaubt (23).

```
http://www.imse.cnm.es/Xfuzzy/

http://www.imse.cnm.es/
```

2.7 Fuzzy Sets For Ada 4

2005 implementierte Dmitry A. Kazakov die Bibliothek „Fuzzy Sets“ für Ada 4. Die nur für Windows verfügbare Bibliothek gestattet sowohl einen Einsatz von Fuzzy Logik als auch unscharfer 2- oder mehr-dimensionaler Zahlen und ist zum Download verfügbar, wird jedoch nicht mehr weiterentwickelt (24).

```
http://downloads.zdnet.com/abstract.aspx?docid=727799
```

2.8 „R“ und das Modul „fuzzOP“

R ist eine unter der GPL publizierte Softwareumgebung für statistische Berechnungen und Grafikdarstellungen und an das weit bekannte ursprünglich von Bell Laboratories entwickelte kommerzielle Werkzeug namens „S“ angelehnt. R bietet eine eigene Programmiersprache, um komplexe Modelle und Programme zu erstellen, die für unterschiedliche Zwecke eingesetzt werden können. Es bietet zusätzlich auch Erweiterungsmöglichkeiten, indem Module angefügt werden können; auch der Aufruf von C, C++ und Fortran-Code ist für prozessor-intensive Aufgaben möglich (25).

Eines der zum Download bereit stehenden Module ist „fuzzyOP“, welches eine Definition und Verwendung von unscharfen Zahlen in Berechnungen und Modellen innerhalb R´s Umgebung gestattet; eine konkrete Weiterentwicklung des Moduls ist nicht ersichtlich (25) (26).

```
http://cran.r-project.org/web/packages/
```

3 Design eines Tools zur Analyse unscharfer Zahlen und Stichproben

In den folgenden Unterkapiteln wird die Implementierung des Tools „FuzzyData“ für die statistische Analyse von unscharfen Zahlendaten und Stichproben beleuchtet. Im Interesse einer möglichst Plattform-unabhängigen Implementierung, wurde für die Erstellung des Programms die Programmiersprache C bzw. C++ herangezogen.

Die einzelnen Funktionalitäten des Programms wurden modular implementiert, um Erweiterbarkeits- und Konfigurationsmöglichkeiten zu bieten. Das Programm wurde zusätzlich zur einfacheren Handhabung mit einer grafischen Benutzeroberfläche ausgestattet; diese wurde mit Hilfe der WIN32 API unter Windows XP umgesetzt.

Die folgenden Kapitel geben einen Überblick über die Funktionen, die FuzzyData bietet und wie diese konkret implementiert wurden. Dabei wurde – besonders bei Wiedergabe von Sourcecode-Segmenten – vorrangig auf Lesbarkeit geachtet, weshalb die generellen Algorithmen in Pseudocode angegeben wurden. Auf diesem Weg kann auch ein Leser, der nicht mit C/C++ - jedoch mit generellen Programmier-Mechanismen – vertraut ist, die Algorithmen des Programms verstehen.

3.1 Repräsentation unscharfer Zahlen und Stichproben

FuzzyData benötigt für die Durchführung statistischer Auswertungen Eingabedaten; diese werden in Form von Datenfiles zur Verfügung gestellt, deren Ursprung einer Vielfalt an Quellen entstammen können: sowohl geologische Messdaten als auch zum Beispiel Temperaturmessdaten sind potentielle Inputs. Zusätzlich bietet FuzzyData allerdings auch zu Testzwecken eine autonome Funktion zum Generieren einer normalverteilten Menge an Zufallsdaten.

Diese Eingabedaten entsprechen somit einer Mischung aus diskreten und den wie in Kapitel 1.4 beschriebenen unscharfen Beobachtungen.

Die Eingabedaten für FuzzyData wurden als Binärdatenfiles definiert, um einen Performance-Verlust durch sonst erforderliche Parsing-Methoden zu vermeiden. Die als „Dataset“ bezeichneten Binärdateien besitzen einen standardisierten Header, der durch folgenden C-Code definiert ist:

```
typedef struct DATASETHEADER {
   FLOAT VersionNumber;
   CHAR szGenerator[255];
   DOUBLE dLowerBorder;
   DOUBLE dUpperBorder;
} __DATASETHEADER;
```

Diesem folgen die Zahlendaten nach Zahlentyp sortiert, d.h., diskrete Zahlen, Intervallzahlen usw. sind gruppiert nacheinander in der Datei angeordnet. Die konkrete Reihenfolge ist:

1. Diskrete Fließkommazahlen
2. Intervallzahlen

3. Dreieckszahlen
4. Trapezzahlen
5. Polygonzahlen

Die explizite Umsetzung der Zahlentypen wird in den folgenden Unterkapiteln erläutert. Dabei zeigt sich ein konkreter Vorteil des Binärdatenformats: die Zahlendaten haben sowohl in persistentem Format als auch im Programmspeicher die gleiche Formatierung und benötigen daher keinerlei zusätzlicher Nachbearbeitung beim Speichern am und Lesen vom Datenträger.

Um die Arbeit mit den Datenfiles zu erleichtern, wurden die gesamten Zugriffsfunktionen in einer eigenen Dynamic Link Library namens SETAC.DLL untergebracht, die auch für andere Tools eingesetzt werden kann. Die Library bietet sowohl reguläre C-Funktionen als auch eine komfortable C++ Helper-Klasse names CDatasetInterface, welche die zur Verfügung stehenden Funktionen kapselt.

FuzzyData setzt programmintern ergänzend die folgende DATASET_DESCRIPTION Struktur ein, um Referenzinformationen über ein Dataset im Speicher zu behalten:

```
typedef struct DATASET_DESCRIPTION {

      CHAR        szFilename[MAX_PATH];
      BOOL        bHeaderCheck;

      INT         iFloats;
      INT         iIntervals;
      INT         iTriangles;
      INT         iTrapezoids;
      INT         iPolygons;
      INT         iSamplingPoints;

      // <summary> when analyzing data set file, this array is
      // filled with the offset adresses of the polygon
      // numbers</summary>

      DWORD       *pdwSamplingOffsets;

      // <description>represents lower border of dataset
      // sample</description>

      DOUBLE      dLowerBorder;

      // <description>represents upper boundary of dataset
      // sample</description>

      DOUBLE      dUpperBorder;

} __DATASET_DESCRIPTION;
```

Auf diese Weise werden öfter benötigte Informationen, wie z.B. die Unter- und Obergrenze einer Stichprobe, zwischengespeichert, um eine mehrmalige Berechnung zu vermeiden.

Zwecks besserer Lesbarkeit des Quellcodes werden Operationen auf den unscharfen Zahlendaten generell über den folgend definierten Typ FUZZYNUMBER abgewickelt:

```
enum DefNumType {

      NUMFLOAT          =     0,
      NUMINTERVAL       =     1,
      NUMTRIANGLE       =     2,
      NUMTRAPEZOID            =     3,
      NUMPOLYGONIAL           =     4

};

typedef struct FUZZYNUMBER {

      enum DefNumType         NumType;

      // <summary>if num type is POLYMOMIAL this flag
      // contains the number of
      // sampling points</summary>
      DWORD                   dwFlag;

      union {

            FLOAT             flNumber;
            INTERVALNUMBER    fiNumber;
            TRIANGLENUMBER    triNumber;
            TRAPEZOIDNUMBER   traNumber;
            POLYGONNUMBER     *pPolNumber;

      };

} __FUZZYNUMBER;
```

Der „Enum“-Typ DefNumType gibt dabei an, um welchen Zahlentyp es sich konkret handelt, der im union des structs gespeichert ist. Auf diesem Weg wird ein genereller Datentyp verwendet, der einen möglichst geringen Speicherverbrauch aufweist.

3.1.1 Intervallzahlen

Intervallzahlen werden durch einen Struct INTERVALNUMBER definiert:

```
typedef struct INTERVALNUMBER {

      FLOAT             fLower;
      FLOAT             fUpper;

} __INTERVALNUMBER;
```

Eine Intervallzahl benötigt damit in Kombination mit dem FUZZYNUMBER Struct 17 Bytes.

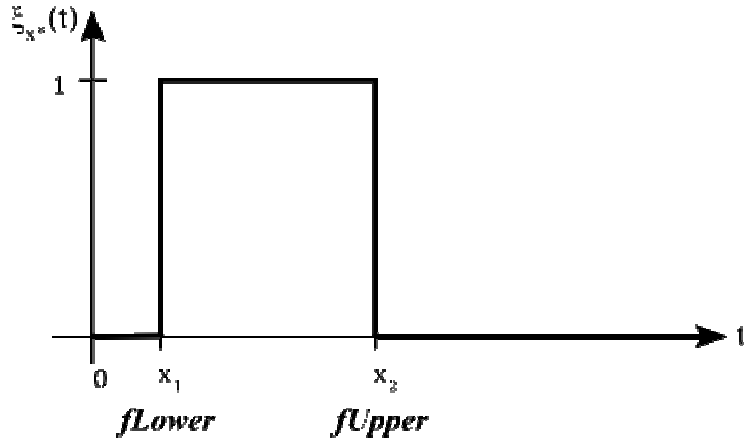

Abbildung 3.1: INTERVALNUMBER Repräsentation einer Intervallzahl

3.1.2 Dreieckszahlen

Dreieckszahlen werden durch einen struct TRIANGLENUMBER definiert:

```
typedef struct TRIANGLENUMBER {
      FLOAT             fMiddle;
      FLOAT             fLeft;
      FLOAT             fRight;
} __TRIANGLENUMBER;
```

Eine Dreieckszahl benötigt somit in Kombination mit dem FUZZYNUMBER Struct 21 Bytes.

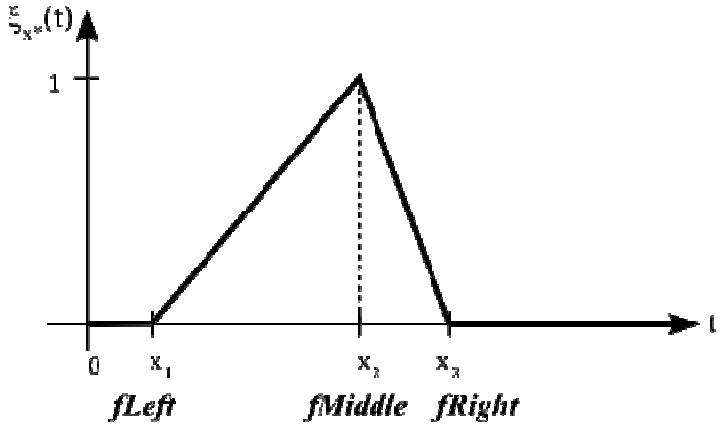

Abbildung 3.2: TRIANGLENUMBER Repräsentation einer Dreieckszahl

3.1.3 Trapezförmige Zahlen

Trapezzahlen werden durch einen Struct TRAPEZOIDNUMBER definiert:

```
typedef struct TRAPEZOIDNUMBER {
      FLOAT             fStart;
      FLOAT             fWidth;
      FLOAT             fLeft;
      FLOAT             fRight;
} __TRAPEZOIDNUMBER;
```

Eine Trapezzahl benötigt daher in Kombination mit dem FUZZYNUMBER Struct 25 Bytes.

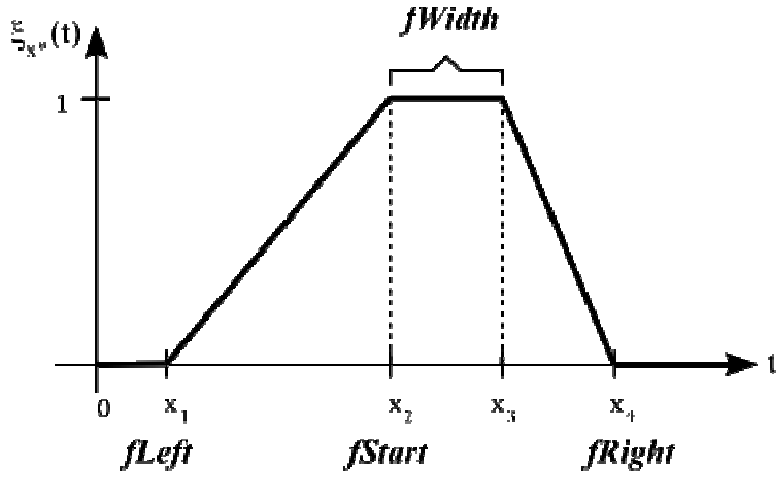

Abbildung 3.3: TRAPEZOIDNUMBER Repräsentation einer Trapezzahl

3.1.4 Polygonzahlen

Eine Polygonzahl wird mit Hilfe der Menge ihrer Stützstellen definiert. Eine Stützstelle wird hierbei mit einem POLYGONNUMBER struct abgebildet. Die einzelnen Structs werden dann im FUZZYNUMBER Struct gesammelt als Array gespeichert, wobei das Array nach aufsteigenden x-Werten sortiert ist und die y-Werte zuerst stetig steigend und nach Erreichen des Werts 1 stetig fallend sind.

```
typedef struct POLYGONNUMBER {
      FLOAT             fX;
      FLOAT             fY;
} __POLYGONNUMBER;
```

Die Sortierung innerhalb des Arrays muss erhalten bleiben; sie ist Voraussetzung für die Funktionstüchtigkeit der in den folgenden Kapiteln beschriebenen Algorithmen und Methoden. D.h., Änderungen am Array – z.B. Entfernen oder Hinzufügen von Stützstellen, darf diese Eigenschaft nicht zerstören.

Zur Vereinfachung verwendet FuzzyData in den meisten Modulen eine C++ Klasse „CPolygon“. Diese bietet die generellen Rechenoperationen „+“ und „*“ und vermeidet auf diesem Weg die relative

aufwändige Array-Manipulation, die sonst etwa bei einer Operation mit zwei FUZZYNUMBER Structs erforderlich wäre.

3.1.5 Visualisierung von Stichprobendaten und Berechnungsergebnissen

Die in einem Dataset enthaltenen Stichproben-Files können mit Hilfe des von Eduard Hirsch erstellten Programms „FuzzyVisualisation“ [42] graphisch dargestellt werden. Ein Beispiel für eine solche Darstellung kann man in Abbildung 3.4 sehen, in der eine Stichprobenmenge aus reellen und unscharfen Beobachtungen visualisiert wird.

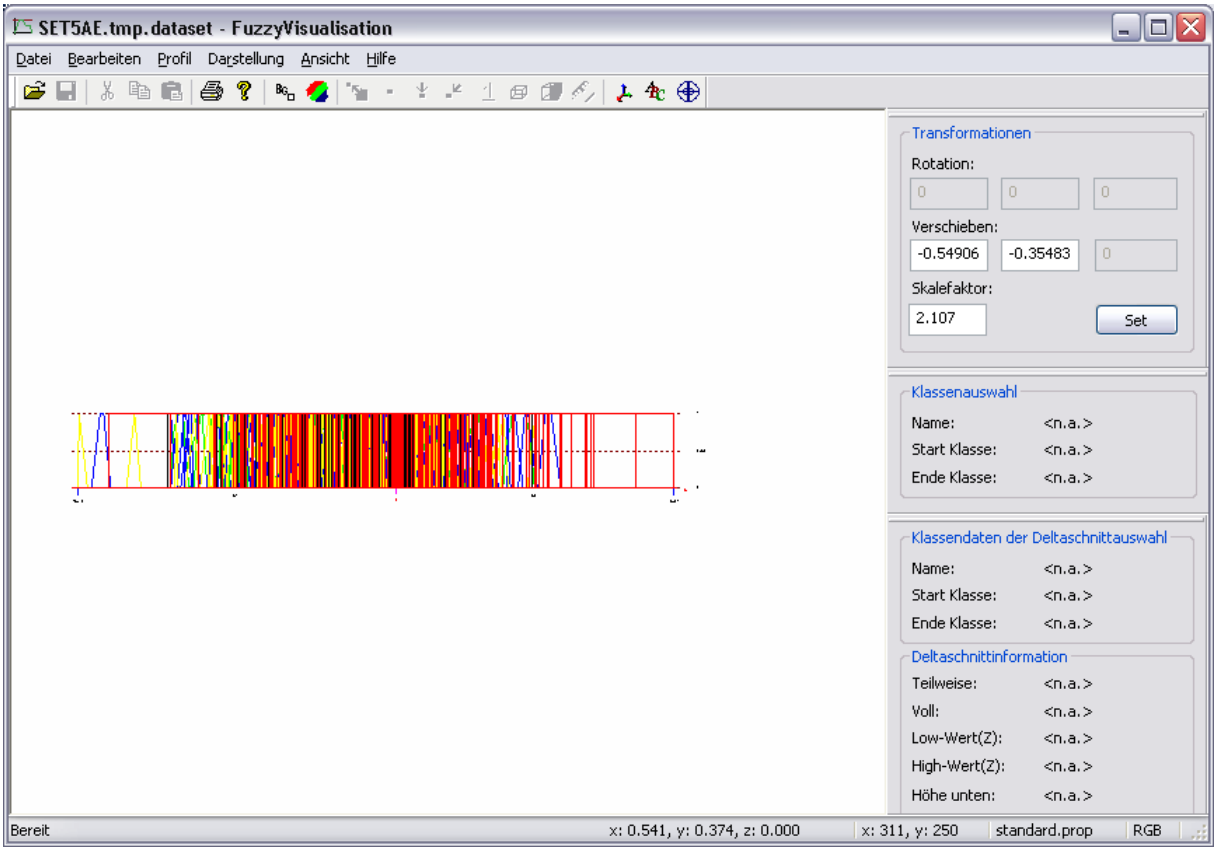

Abbildung 3.4: Visualisierung einer Stichprobe

3.2 Erzeugen von Zufallszahlen und Stichproben

Da FuzzyData eine „Proof-of-Concept“ Applikation ist, für die aktuell keine Anbindung von Messgeräten etc. existiert, bedarf es für den Einsatz der zur Verfügung stehenden Funktionen und Methoden einer Konverter-Applikation, um praxisbezogene Eingabedaten in ein für FuzzyData verständliches Format zu übersetzen. Für eine vereinfachte Verwendung der Applikation und effektiveren Präsentation seines Funktionsumfangs, bietet FuzzyData jedoch auch eine Prozedur zur Erstellung einer unscharfen Stichprobe.

Der Algorithmus für die Erzeugung durchläuft eine Schleife, die in einem oder mehreren Durchläufen nacheinander eine zufällige Anzahl an diskreten Zahlen, Intervallzahlen, Dreieckszahlen, Trapezzahlen und zuletzt Polygonzahlen erzeugt. Sind nach Abschluss der Schleife noch nicht genug Zahlen erzeugt, beginnt der Durchlauf noch einmal von vorn. In Pseudo-Code sieht der Algorithmus folgendermaßen aus:

```
While (numbersDone < requiredNumbers ) do begin
```

```
      discreteNumbers = round(rand( requiredNumbers – numbersDone ));
      createDiscreteNumbers( discreteNumbers );
      numbersDone = numbersDone + discreteNumbers;

      intervalNumbers = round(rand( requiredNumbers – numbersDone ));
      createIntervalNumbers( intervalNumbers );
      numbersDone = numbersDone + intervalNumbers;

      …

      polygonNumbers = round(rand( requiredNumbers – numbersDone ));
      createPolygonNumbers( polygonNumbers );
      numbersDone = numbersDone + polygonNumbers;

End
```

Die mit Hilfe der C-Funktion `rand()` generierten Pseudo-Zufallszahlen helfen, die Verteilung der erzeugten Zahlendaten zwischen den unterschiedlichen Zahlentypen möglichst zufällig vorzunehmen.

3.2.1 Basisalgorithmus

Als Basis für die Generierung der einzelnen Zahlentypen wird der folgende in Pseudo-Code angegebene Algorithmus zur Erzeugung einer standard-normalverteilten Zufallszahl eingesetzt:

```
DOUBLE                  u1, u2;
DOUBLE                  v1, v2;
DOUBLE                  s = 2.0f;

v1 = v2 = 0;

while ( (s > 1.0f) ) do begin

      while( (!v1) && (!v2) ) {

            u1 = rand( 1 );
            u2 = rand( 1 );

            v1 = 2*u1-1;
            v2 = 2*u2-1;

      }

      s = v1*v1 + v2*v2;

End

x = sqrt( -2 * ln( s ) / s ) * v1;
y = sqrt ( -2 * ln( s ) / s ) * v2;

if ( x > y ) then begin

      s = y;
      y = x;
      x = s;

End

Return [x,y];
```

Die Methode erzeugt anfangs in u1 und u2 zwei auf [0,1] gleichverteilte Pseudo-Zufallszahlen. Die Hilfsvariablen v1 und v2 werden zur Berechnung der Entscheidungsvariable s herangezogen; der Algorithmus stellt sicher, dass stets 0<s<=1 gilt. s wird somit für die Berechnung der Werte

$$x = \left(\sqrt{\frac{-2\ln(s)}{s}}\right) v_1$$

und

$$y = \left(\sqrt{\frac{-2\ln(s)}{s}}\right) v_2$$

verwendet.

3.2.2 Diskrete Zufallszahl

Für die Erstellung einer diskreten Zufallszahl wird einfach die Variable x aus dem zuvor vorgestellten Algorithmus übernommen.

3.2.3 Intervallzahl

Zur Erstellung einer Zufalls-Intervallzahl wird x als die untere Intervallgrenze angenommen; die obere Grenze wird durch folgende Pseudocode-Formel errechnet:

```
fUpper = fLower + y*0.1;
```

3.2.4 Dreiecksförmige unscharfe Zahl

Die Erstellung einer Dreieckszahl ist folgendermaßen implementiert:

- Der Dreiecksmittelpunkt wird mit x festgelegt
- Die Längen des linken und rechten Bereichs davon werden durch zwei Pseudo-Zufallszahlen im Bereich [0,0.1] ermittelt.

```
fMiddle = x;
fLeft = fMiddle – rand(0.1);
fRight = fMiddle + rand(0.1);
```

3.2.5 Trapezförmige unscharfe Zahl

Als Mitte des 1-Schnitts wird x angenommen; die Breite des 1-Schnitts wird als Pseudo-Zufallszahl im Intervall [0,0.1] angenommen. Ebenso wird die Breite des linken und rechten Bereichs als Pseudo-Zufallszahl im Intervall [0,0.1] berechnet.

```
fWidth = rand(0.1);
fStart = x – fWidth/2;
fLeft = fStart - rand(0.1);
```

```
fRight = fStart + fWidth + rand(0.1);
```

3.2.6 Polygonförmige unscharfe Zahl

Für die Erstellung einer Polygonzahl wird folgender in Pseudocode beschriebener Algorithmus eingesetzt. Es werden die folgenden Variablen eingesetzt:

```
POLYGONNUMBER     Stuetzstellen[];
Double            length1Schnitt;
Double            lengthLeft;
Double            lengthRight;
int               amountLeft;
int               amountRight;
```

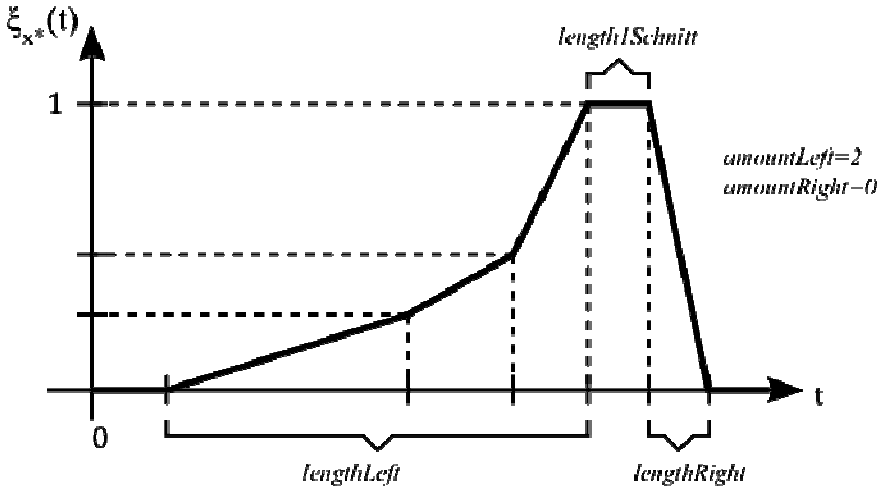

Abbildung 3.5: Helpervariablen Beispiel für die Erstellung einer Polygonzahl

Die Variable “Stuetzstellen” wird in Folge mit der Menge der Stützstellen der polygonförmigen unscharfen Zahl befüllt. Die Koordinaten werden dabei mit Hilfe des im Unterkapitel 3.1.4 definierten Struct POLYGONNUMBER gespeichert. „length1Schnitt“ wird mit einer im Intervall [0,0.1] gleichverteilten Pseudo-Zufallszahl befüllt, die darüber entscheidet, wie lang der 1-Schnitt der Polygonzahl ist. Die „left“ und „right“ Variablen werden ebenso mit Pseudo-Zufallszahlen im Intervall [0,0.2] befüllt und entscheiden über die Länge des linken aufsteigenden und rechten abfallenden Bereichs. Die „amount“ Variablen entscheiden über die durch Pseudo-Zufallszahl zwischen 0 und 10 gewählte Anzahl an Stützstellen im linken und rechten Bereich.

```
length1Schnitt = rand( 0.1 );

lengthLeft = rand( 0.2 );

lengthRight = rand( 0.2 );

amountLeft = round( rand( 10 ));

amountRight = round( rand( 10 ));

Stuetzstellen = new array[amountLeft+amountRight+2];

Stuetzstellen[amountLeft].x = x;
Stuetzstellen[amountLeft].y = 1.0;

Stuetzstellen[amountLeft+1].x = x+length1Schnitt;
Stuetzstellen[amountLeft+1].y = 1.0;
```

```
Stuetzstellen[0].x = x – lengthLeft;
Stuetzstellen[0].y = 0;

Stuetzstellen[amountLeft+amountRight+1].x =
      x+length1Schnitt+lengthRight;
Stuetzstellen[amountLeft+amountRight+1].y = 0;
```

Dann wird für den linken Arm die durch „amountLeft“ angegebene Anzahl an Stützstellen erzeugt:

```
for ( count = 1; count < amountLeft; count++ ) do begin

      Stuetzstellen[count].x =
            Stuetzstellen[count-1].x + lengthLeft/amountLeft;

      Stuetzstellen[count].y =
            Stuetzstellen[count-1].y +
            rand(1-Stuetzstellen[count-1].y);

End
```

Nach dem gleichen Schema werden danach die Stützstellen für den rechten Arm erzeugt.

3.3 Glätten polygonförmiger unscharfer Zahlen

Wie man aus der zuvor aufgeschlüsselten Definition von polygonförmigen Zahlen erkennen kann, ist es möglich, dass diese relativ komplexe Formen annehmen. Vor allem nach der Durchführung von Rechenoperationen kann sich die Menge der einzelnen Stützstellen stark vervielfachen mit dem Effekt, dass einerseits sehr viel Speicher verbraucht wird, weitere Rechenoperationen länger benötigen (da ggfs. alle Stützstellen berücksichtigt werden müssen) und eine Vielzahl an redundanten Stützstellen (z.B. Punkte auf einer Geraden) gespeichert sind.

Im Interesse von Performance und auch einer besseren Lesbarkeit der in den unscharfen Zahlen enthaltenen Informationen können polygonförmige Zahlen „geglättet“ werden – in diesem Prozess werden

- überflüssige Stützstellen (d.h., redundante Punkte auf einer Geraden) entfernt und
- zusätzlich können auch Punkte, die bis auf ein geringes Delta von einer Geraden entfernt liegen, aus der Stützstellen-Liste genommen werden.

Wie groß dieses Delta maximal sein darf, hängt davon ab, welche Genauigkeit sich der Benutzer bei der Berechnung wünscht. Generell wird $\delta < 10^{-3}$ vermutlich eine ausreichend kleine Schwelle sein.

Für die Berechnung, ob eine Stützstelle entfernt werden kann, wird anfangs – wie in Abbildung 3.6 dargestellt – der y-Punkt für die Gerade zwischen zwei Stützstellen eines Punkte-Tripels untersucht. Die dafür verwendete Formel lautet

$$\hat{y}_2 = \frac{y_3 - y_1}{x_3 - x_1}(x_2 - x_1) + y_1$$

Der berechnete $\hat{y}$-Wert wird dem tatsächlichen y-Wert gegenübergestellt; ist die Differenz kleiner als die gesetzte Fehlerschwelle ϵ, kann der Punkt entfernt werden.

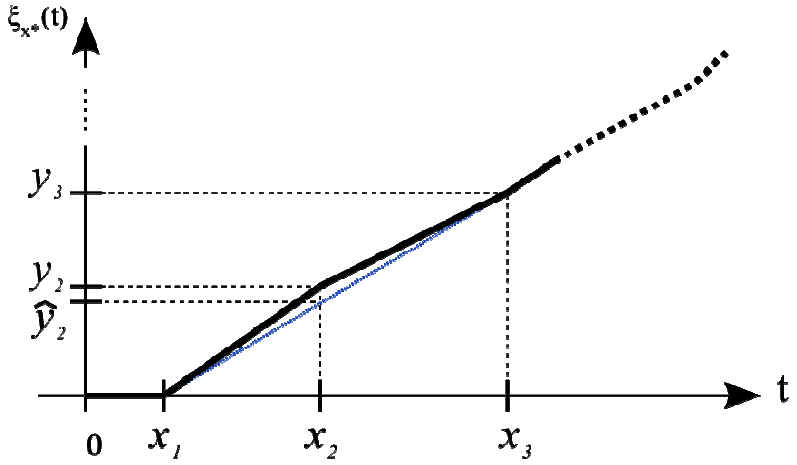

Abbildung 3.6: Glätten einer polygonförmige Zahl

Der in FuzzyData eingesetzte Algorithmus betrachtet stets nur polygonförmige Zahlen mit mindestens 5 Stützstellen und durchläuft dabei folgende in Pseudo-Code gehaltene Prozedur. Die dabei verwendete LineHelper-Klasse ist eine einfache Hilfsklasse, die über die Funktion „GetYByX“ den Y-Wert für einen gegebenen X-Wert berechnet, welcher auf der Geraden zwischen zwei Punkten liegt.

1. Notwendige Variablen werden definiert.

```
iterator    count, compar;
iterator    start, end;
LineHelper  line;
DOUBLE      dSample;
```

2. Wenn weniger als 5 Stützstellen vorhanden sind, wird keine Glättung vorgenommen.

```
if ( points.size() < 5 ) then
      return TRUE;
```

3. Für den Fall, dass die Zahl im Anfangsbereich mehrere Stützstellen mit y=0 aufweist, werden zuerst alle bis auf die letzte dieser Stützstellen entfernt. Das gleiche erfolgt danach invertiert für den rechten Arm der polygonförmigen Zahl. Existieren mehrere Zahlen im rechten Arm der unscharfen Zahl, deren y=0 ist, werden bis auf die erste Stützstelle alle weiteren redundanten Stützstellen aus dem Punkte-Array entfernt.

```
// solange die ersten zwei stützstellen Y=0 sind, den ersten
// punkt entfernen
while ( ! (points.begin()+1).Y ) do
      points.erase( points.begin() );

// solange die letzten zwei stützstellen Y=0 sind, den letzten
// punkt entfernen
while ( ! (points.end()-2).Y ) do
      points.erase( points.end()-1 );
```

4. Darauf folgt eine Iteration über alle verfügbaren Punkte bis zur vorletzten verfügbaren Stützstelle.

```
for ( count = points.begin(); (count < (points.end()-1)); count++ )
do
Begin

      start = NULL; end = NULL;
      line.SetPoint1 ( count.X, count.Y );
      dSample = (count+1).Y;
```

5. Jede der betrachteten Stützstellen wird nun mit den jeweils übernächsten und folgenden Stützstelle mit einer Geraden verbunden. Befinden sich alle Stützstellen dazwischen bis auf ein vernachlässigbar kleines Epsilon auf oder in der Nähe der Geraden, können die Stützstellen dazwischen wegoptimiert werden. Es wird so lange versucht, eine Stützstelle weiter hinten als zweiten Punkt der Geraden zu wählen, bis die Punkte dazwischen nicht mehr auf einer Geraden liegen. Danach wird die effektive Glättung vorgenommen.

```
      for ( compar = count+2; compar < points.end();
            compar++ ) do
      Begin

            line.SetPoint2 ( (compar).dX, (compar).dY );

            if ( abs(dSample – line.GetYByX((count+1).dX)) <= ϵ ) Then
                  if ( start == NULL ) Then
                        start = end = count+1;
                  else
                        end = compar-1;
            Else
                  break;

      End

      if ( (start!=NULL) && (end!=NULL) ) Then
            if ( start == end ) Then
                  points.erase(start);
            else
                  points.erase(start,end);


End
```

3.4 Funktionen unscharfer Zahlen

3.4.1 Grundlagen

Wie in Kapitel 1.3 gezeigt, definieren sich Funktionen auf unscharfen Zahlen über die Deltaschnitte der einzelnen Beobachtungen. Daher ist es erforderlich, zuerst ein computerverständliches Format zu definieren, mit dessen Hilfe Deltaschnitte repräsentiert und gespeichert werden können.

Für die Berechnung von Operationsergebnissen verwendet FuzzyData ein Array aus Struct POLYGONNUMBER, das eine unscharfe Zahl über deren Koordinaten der einzelnen Stützstellen definiert. Dazu werden zuerst alle nicht-polygonförmigen Zahlentypen, die an der Operation beteiligt sind, in Polygonform umgewandelt.

```
typedef struct FUZZYNUMBER {

      enum DefNumType         NumType;

      //<summary>if num type is POLYMOMIAL this flag contains
      // the number of sampling points</summary>
      DWORD                   dwFlag;

      union {

            FLOAT             flNumber;
            INTERVALNUMBER    fiNumber;
            TRIANGLENUMBER    triNumber;
            TRAPEZOIDNUMBER   traNumber;
            POLYGONNUMBER     *pPolNumber;

      };

} __FUZZYNUMBER;

typedef struct POLYGONNUMBER {

      FLOAT             fX;

      FLOAT             fY;

} __POLYGONNUMBER;
```

D.h., es wird zum Beispiel eine Dreieckszahl, die über

```
typedef struct TRIANGLENUMBER {

      FLOAT             fMiddle;

      FLOAT             fLeft;

      FLOAT             fRight;

} __TRIANGLENUMBER;
```

definiert wird und für unseren Beispielfall, die Werte fMiddle=0.5, fLeft=0.2 und fRight=1.3 besitzt, in ein POLYGONNUMBER Array {[x=0.2,y=0],[x=0.5,y=1.0],[x=1.3,y=0]} umgewandelt.

Die Hilfsklasse CDeltacutPolygon gestattet es, danach die aus den Stützstellen implizit vorhandenen Deltaebenen auf allen an der Operation beteiligten unscharfen Zahlen zu vereinheitlichen. Es wird also eine Liste aller vorhandenen y-Werte aus den resultierenden Polygonzahlen extrahiert und danach bei allen unscharfen Zahlen sichergestellt, dass für jeden vorhandenen y-Wert die beiden passenden unteren und oberen δ-Schnitt-Intervallgrenzen gespeichert sind. Falls ein Deltaschnitt-Intervall fehlt, werden die Werte berechnet und ins Array eingefügt. CDeltacutPolygon ist speziell darauf zugeschnitten, Polygone auf eine Deltaschnitt-Repräsentation umzuformen und eine Manipulation der Datenpunkte auf Deltaschnitt-Ebene zu gestatten. Die Klasse bietet etwa folgende Methoden

- GetDeltaCount(), GetDeltaList(): liefert die Anzahl der Detalschnitt-Ebenen und ein Array von DOUBLE-Werten, welche die einzelnen Deltaebenen angeben.

- CacheDelta(DOUBLE delta): überprüft, ob ein Deltaschnitt für den angegebenen delta-Wert existiert; wenn keiner vorhanden ist, werden für den übergebenen Wert die untere und obere Intervallgrenze berechnet und das Intervall in die Deltaliste eingefügt.

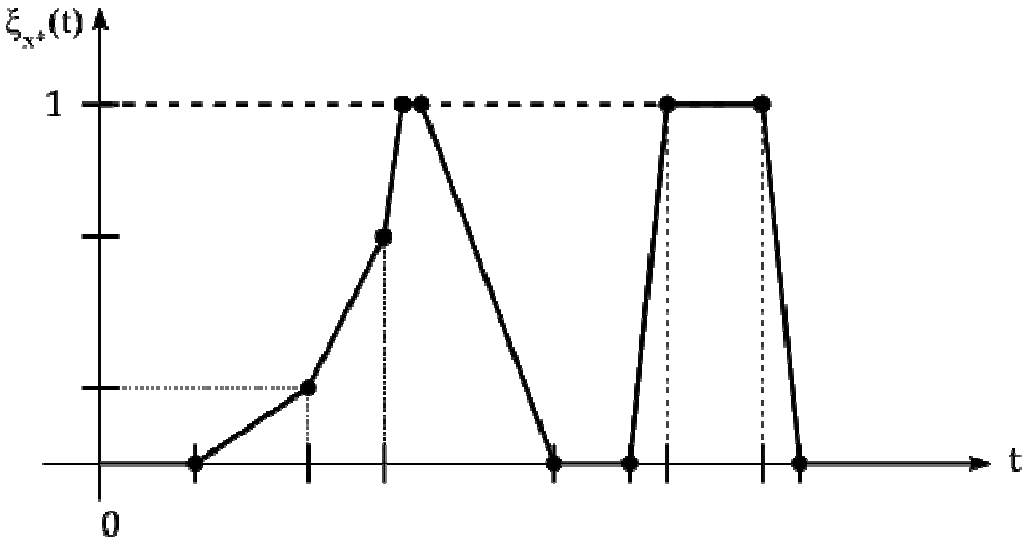

Abbildung 3.7: Stützstellen vor der Vereinheitlichung der Deltaschnitt-Ebenen

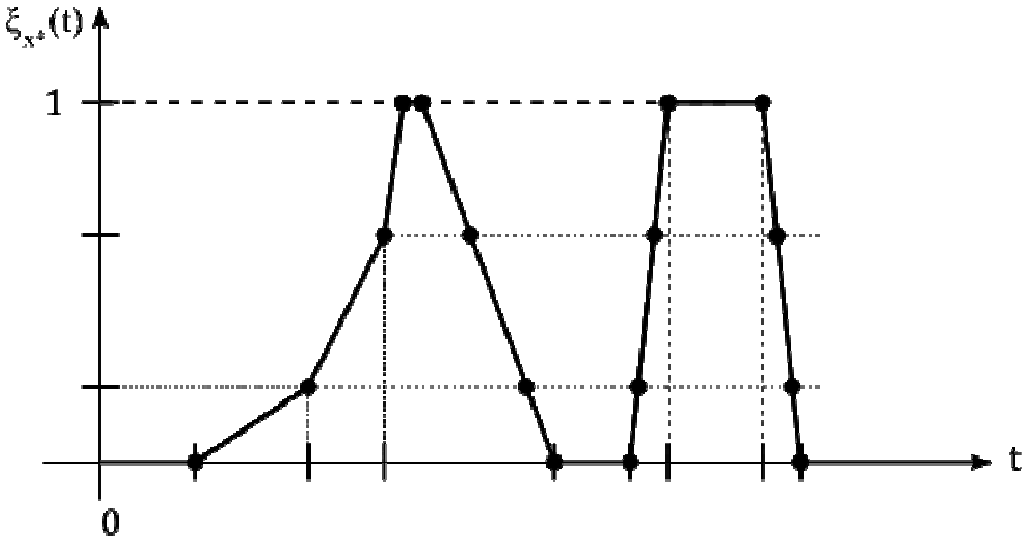

Abbildung 3.8: Stützstellen nach der Vereinheitlichung der Deltaschnitt-Ebenen

3.4.2 Berechnung von Delta-Schnitten

Die Berechnung von Deltaschnitt-Intervallen führt FuzzyData über die beiden Funktionen GetXByY und GetXByY_Inverted der Klasse CPolygon durch. GetXByY berechnet, vom ersten Stützpunkt ausgehend und sequentiell ans Ende suchend, die erste Stelle im aufsteigenden Ast der polygonförmigen Zahl, an welcher der gegebene Y-Wert erreicht ist. GetXByY_Inverted führt prinzipiell dieselbe Methode durch, arbeitet jedoch von der letzten Stützstelle rückwärts zum erste Stützpunkt nach vorne, d.h., sie arbeitet in umgekehrter Reihenfolge und operiert daher auf dem absteigenden Ast der polygonförmigen Zahl.

Im Fall, dass GetXByY und GetYByX innerhalb eines in Schleifenform aufgebauten Algorithmus eingesetzt wird, gestatten beide Methoden die Angabe eines Start-Index Wertes. Dieser indexwert gibt an, ab welcher Arrayposition die Suche begonnen werden soll. Auf diesem Weg können durch

bereits zuvor ausgeschlossene Bereiche einer unscharfen Zahl von der Suche ausgenommen werden, was effektiv den Algorithmus stark beschleunigen kann. Die Methode GetXByY ist im Folgenden in Pseudo-Code angegeben:

6. Definition erforderliche Variablen: es wird ein Iterator count zum Iterieren über die verfügbare Stützstellen und eine Hilfsvariable dHeight definiert.

```
SP_LIST::iterator       count;
DOUBLE                  dHeight;
```

7. Ist der Y-Wert der ersten Stützstelle nicht gleich 0, bricht der Algorithmus ab. In diesem Fall handelt es sich nicht um eine gültige unscharfe Zahl.

```
if( !Y )
      Return points.at(0).X;

for ( count = points.begin(); count < points.end()-1; count++ ) do
Begin
```

8. Sollte der Y-Wert sowohl der betrachteten als auch der nächsten Zahl exakt mit dem gewünschten Y-Wert übereinstimmen, wird der X-Wert der aktuell betrachteten Zahl zurückgegeben.

```
      if (( count.Y == Y ) && (( count+1 ).Y == Y ))
            return count.X;
```

9. Sollte der gewünschte Y-Wert genau zwischen den Y-Werten der betrachteten und der nächsten Zahl liegen, so wird der Schnittpunkt auf der Geraden zwischen den beiden Stützstellen mit der Geraden Y berechnet und zurückgegeben.

```
      if (( count.Y <= Y ) && (( count+1 ).Y > Y )) then

            // up-flank intersection
            return count.X + (( Y - count.Y ) *
                        (( count+1 ).X-count.X) /
                        (( count+1 ).Y-count.Y ) );

      else if (( count.Y >= Y ) && (( count+1 ).Y <= Y )) then

            Begin

                  // down-flank intersection

                  dHeight = count.Y - ( count+1 ).Y;

                  return count.X + (( count+1 ).X - count.X ) *
                        ( dHeight - ( Y - ( count+1 ).Y )) / dHeight;

            End

End

throw new CNoIntersectionException();

return 0.0f;
```

3.4.3 Summe unscharfer Zahlen

Die Addition zweier unscharfer Zahlen implementiert FuzzyData im überladenen „+“-Operator in der Klasse CPolygon. Die Methode durchläuft sequentiell die Stützstellen der beiden beteiligten Polygone. Der prinzipielle Algorithmus sieht wie folgt aus:

1. Definieren der erforderlichen Hilfsvariablen:

```
CPolygon                *pNew = new CPolygon();
DWORD                   dwIndex[2] = { 0, 0 };
DWORD                   dwDown[2] = { 0, 0 };
DOUBLE                  dFoundX;
DOUBLE                  dCurrentY;
BOOL                    bUpFlank = TRUE;
DOUBLE                  dDelimiter = 0.0f;
DWORD                   dwMarker[2] = {0, 0};
```

2. Da beide polygonförmige Zahlen in der ersten Stützstelle einen y-Wert = 0 besitzen, kann die erste Stützstelle der addierten Polygonzahl automatisch mit der Addition der beiden X-Werte definiert werden. GetXByIndex(n) liefert hierbei den X-Wert der n-ten Stützstelle (im gegebenen Fall n=0).

```
// starting left, adding both left side start point values...
pNew.AddSamplingPoint (
      this.GetXByIndex(dwIndex[0]) +
      poly.GetXByIndex(dwIndex[1]), 0.0f );
```

3. Danach werden beide Stützstellen-Indizes inkrementiert.

```
dwIndex[0]++; dwIndex[1]++;

dwMarker[0] = dwIndex[0]; dwMarker[1] = dwIndex[1];
```

4. Der kleinere Y-Wert der nächsten Stützstelle in den beiden unscharfen Zahlen entscheidet, auf welcher Deltaebene weitergearbeitet wird.

```
dDelimiter = min ( this.GetYByIndex(dwIndex[0]),
      poly.GetYByIndex(dwIndex[1]) );

while ( dDelimiter < 1.0f ) do
Begin
```

5. Solange der 1-Schnitt nicht erreicht ist und man sich auf dem stetig steigenden Ast der polygonförmigen Zahl befindet, wird die folgende Schleife durchlaufen.

```
if(this.GetYByIndex( dwIndex[0] ) < poly.GetYByIndex( dwIndex[1] ))
then begin
```

6. Wenn die erste (in „this“ enthaltene) polygonförmige unscharfe Zahl in der nächsten betrachteten Stützstelle den kleineren Y-Wert aufweist, wird der zugehörige X-Wert genommen und mit dem interpolierten X-Wert aus dem zweiten Polygon addiert, um die nächste Stützstelle zu bilden.

```
      pNew.AddSamplingPoint ( this.GetXByIndex(dwIndex[0]) +
            ( dFoundX = poly.GetXByY( ( dCurrentY =
                  this.GetYByIndex(dwIndex[0]) ), dwMarker[1]) ),
                  dCurrentY );

      if ( (dwIndex[0]+1) < this.GetSamplingCount() ) then
      Begin
```

7. Für den Fall, dass die betrachtete Stützstelle auf einem “Plateau“ liegt, wird eine Marker-Variable gesetzt.

```
            if ( this.GetYByIndex(dwIndex[0]+1) ==
                  this.GetYByIndex(dwIndex[0]) ) then
                        dwMarker[0] = dwIndex[0]+1;
      End

      dwIndex[0]++;

End else begin
```

8. Gleiches Vorgehen für den Fall, dass die zweite (in „poly“ enthaltene) Polygonzahl den kleineren Y-Wert in der nächsten Stützstelle aufweist.

```
      pNew.AddSamplingPoint ( poly.GetXByIndex(dwIndex[1]) +
            ( dFoundX = this.GetXByY ( ( dCurrentY =
            poly.GetYByIndex ( dwIndex[1] ) ), dwMarker[0] ) ),
            dCurrentY );

      if ( (dwIndex[1]+1) < poly.GetSamplingCount() ) then
      Begin

            if ( poly.GetYByIndex(dwIndex[1]+1) ==
                  poly.GetYByIndex(dwIndex[1]) ) then
                  dwMarker[1] = dwIndex[1]+1;

      end

      dwIndex[1]++;

end
```

9. Berechnung des nächsten zu bearbeitenden Y-Werts aus dem Minimum der Y-Werte der nächsten beiden verfügbaren Stützstellen.

```
dDelimiter = min ( this.GetYByIndex(dwIndex[0]),
                  poly.GetYByIndex(dwIndex[1]) );
```

10. Ende der Schleife für den aufsteigenden Ast

```
End
```

11. Für beide unscharfe Zahlen ist nun der 1-Schnitt erreicht; es wird die nächste Stützstelle für die addierte unscharfe Zahl berechnet durch Addition der X-Werte der Stützstellen aus „this“ und „poly“.

```
pNew.AddSamplingPoint ( this.GetXByIndex ( dwIndex[0] ) +
      poly.GetXByIndex ( dwIndex[1] ), 1.0f );
```

12. Danach werden für beide unscharfe Zahlen die Stützstellen-Indizes solange inkrementiert, bis die nächste Stützstelle im Array bereits auf dem stetig fallenden Seitenarm liegt. Mit den resultierenden Indizes wird anschließend eine eventuell zweite vorliegende Stützstelle am 1-Schnitt berechnet.

```
if ( this.GetYByIndex(dwIndex[0]+1 ) == 1.0f )
      dwIndex[0]++;

if ( poly.GetYByIndex(dwIndex[1]+1 ) == 1.0f )
      dwIndex[1]++;

pNew.AddSamplingPoint( this.GetXByIndex ( dwIndex[0] ) +
      poly.GetXByIndex ( dwIndex[1] ), 1.0f );

dwIndex[0]++;
dwIndex[1]++;
```

13. Das Vorgehen für den rechten Seitenarm wird in umgekehrter Reihenfolge von der letzten Stützstelle bis hinauf zum 1-Schnitt durchlaufen.
14. Zuletzt werden überflüssige bzw. redundante Stützstellen in der resultierenden unscharfen polygonförmigen Zahl entfernt. Das Vorgehen hierfür wurde bereits in Kapitel 3.3 beschrieben. Danach wird die neue resultierende unscharfe Zahl zurückgeliefert.

```
pNew.Optimize();

return pNew;
```

3.4.4 Multiplikation unscharfer Zahlen

Die Multiplikation zweier unscharfer Zahlen wird im überladenen Multiplikations-Operator * der Klasse CPolygon implementiert. FuzzyData konvertiert zum Multiplizieren zweier unscharfer Zahlen diese zunächst in polygonförmige Repräsentation und nutzt dann den implementierten Operator.

Der Algorithmus der Multiplikation ist im Folgenden in Pseudocode angegeben:

1. Es werden zunächst Hilfsvariablen definiert: pNew ist eine neu instantiierte polygonförmige unscharfe Zahl, welche mit dem Ergebnis befüllt wird.

```
CPolygon                pNew = new CPolygon();
CPolygon                pExec[], pQuery[];
DWORD                   dwIndex[2] = { 0, 0 };
DWORD                   pdwExec[], pdwQuery[];
DOUBLE                  dBorders[4];
DOUBLE                  dCurrentY;
```

2. Zunächst wird auf beiden an der Operation beteiligten unscharfen Zahlen eine Glättung vorgenommen, wie in Kapitel 3.3 beschrieben:

```
this.Optimize();
poly.Optimize();


pExec = pQuery = NULL;
dCurrentY = 0.0f;
```

3. dCurrentY hält für den weiteren Verlauf des Algorithmus den jeweils gerade berechneten Deltschnitt-Wert im Intervall [0,1].

```
while ( dCurrentY <= 1.0f ) do
begin
```

4. Zunächst wird ermittelt, welche der beiden unscharfen Zahlen in der nächsten betrachteten Stützstelle den kleineren Y-Wert aufweist. Der Index und Array-Pointer der Zahl mit dem kleineren Index wird in pExec/pdwExec und die andere Zahl in pQuery/pdwQuery übernommen.

```
if( this.GetYByIndex(dwIndex[0]) < poly.GetYByIndex(dwIndex[1]) )
then
begin
      pExec = this; pdwExec = dwIndex[0];
      pQuery = poly; pdwQuery = dwIndex[1];
else
      pExec = poly; pdwExec = dwIndex[1];
      pQuery = this; pdwQuery = dwIndex[0];
end
```

5. Die Schnittpunkte auf beiden unscharfen Zahlen für den gewünschten Y-Wert werden errechnet und in das Eregbnispolygon übertragen.

```
dCurrentY = pExec.GetYByIndex ( pdwExec );

dBorders[0] = pExec.GetXByIndex ( pdwExec );

dBorders[1] = pExec.GetXByY_Inverted ( dCurrentY );

dBorders[2] = pQuery.GetXByY ( dCurrentY );

dBorders[3] = pQuery.GetXByY_Inverted ( dCurrentY );


pNew.AddSamplingPoint ( min(min(min(dBorders[0]*dBorders[2],
      dBorders[0]*dBorders[3]), dBorders[1]*dBorders[2]),
      dBorders[1]*dBorders[3]), dCurrentY );

pNew.AddSamplingPoint ( max(max(max(dBorders[0]*dBorders[2],
      dBorders[0]*dBorders[3]), dBorders[1]*dBorders[2]),
      dBorders[1]*dBorders[3]), dCurrentY );

end;

return pNew;
```

3.4.5 Flächenintegration auf polygonförmigen Zahlen

CPolygon bietet für die Integration einer polygonförmigen Zahl die Methode IntegrateArea(dX), welche etwa bei der Berechnung der Summenkurve (siehe Kapitel 1.4.7) zum Tragen kommt. Die Fläche unterhalb der polygonförmigen Zahl wird hierzu in eine Menge von rechteckigen und dreieckigen Flächen zerlegt und deren Flächensummen berechnet.

Der Algorithmus hierzu ist im Folgenden in Pseudocode angegeben.

1. Hilfsvariablen werden definiert

```
DOUBLE                  dArea = 0.0f;

SP_LIST::iterator       count;

DOUBLE                  dWidth;

DOUBLE                  dY;

DOUBLE                  dDiff;
```

2. Wenn der übergebene Integrations-Parameter dX kleiner ist als das X der ersten Stützstelle, wird sofort 0 zurückgegeben.

```
if ( dX <= points.at(0).X ) then
      return 0.0f;
```

3. Der Algorithmus iteriert von der 2.ten bis zur letzten Stützstelle der polygonförmigen unscharfen Zahl.

```
for ( count = points.begin()+1;
      count < points.end(); count++ ) do

begin

      if ((( count-1 ).X < dX ) && (( count ).X > dX )) then
            break;

      dWidth = abs(count.X – (count-1).X);
```

4. Die bisher berechnete Fläche wird addiert mit der Fläche zwischen der aktuell betrachteten und letzten Stützstelle.

```
      dArea += abs( dWidth * abs(( count ).Y -
            ( count-1 ).Y )) / 2 ) +
            min( ( count ).Y, (( count-1 )).Y ) * dWidth;
```

5. Wenn die Integrationsgrenze dX dem X-Wert der aktuell betrachteten Stützstelle entspricht, wird die bisher berechnete Fläche zurückgegeben.

```
      if ( ( count ).X == dX ) then
            return dArea;

end

if ( dX > (points.end()-1).dX )
      return dArea;

// if dX is located between to sampling sections, we need
// to calculate the dY at the specified point and
// calculate with that value

dY = ( count).dX;

dY = ( count-1 ).dX;



dY = ( count ).dY - ( count-1 ).dY;

dDiff = abs ( ( count-1 ).dX - ( count ).dX );

dY /= dDiff;

dY *= (dWidth = abs (dX - ( count-1 ).dX));

dY += ( count-1 ).dY;

dArea += ( dWidth * abs(dY - ( count-1 ).dY ) ) / 2 +
      min(dY, ( count-1 ).dY) * dWidth;

return dArea;
```

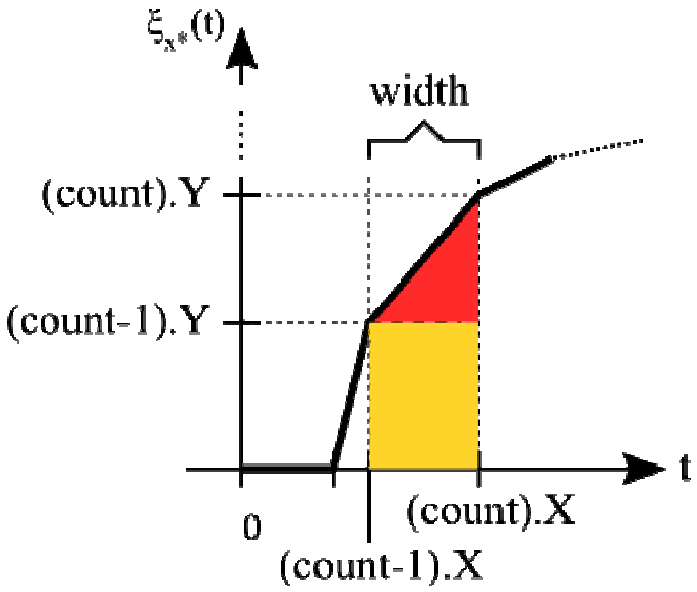

Abbildung 3.9: Zerlegen der Polygonfläche in Dreiecke und Rechtecke

3.5 Auswertung einer unscharfen Stichprobe

3.5.1 Häufigkeitsverteilung

Wie in Kapitel 1.4.1 erläutert, wird für die Berechnung der Häufigkeitsverteilung eine Klasseneinteilung des Merkmalraums vorgenommen. Im Fall einer unscharfen Stichprobe wird zusätzlich eine Menge an Deltaschnitten definiert, über welche das dreidimensionale Histogramm generiert wird.

Für jeden Deltaschnitt $C_\delta(x_i^*) = \left[\underline{x}_{i,\delta}^*, \overline{x}_{i,\delta}^*\right]$ wird überprüft, wie dieser in Relation zur jeweils betrachteten Klasse $K_i = \left[\underline{k}_i, \overline{k}_i\right]$ liegt. Dabei gibt es konkret drei Fälle:

1. Der Deltaschnitt liegt vollständig im Klassenintervall, d.h., es gilt

$$\left(\underline{x}_{i,\delta}^* > \underline{k}_i\right) \wedge \left(\overline{x}_{i,\delta}^* < \overline{k}_i\right)$$

2. Der Deltaschnitt liegt vollständig außerhalb des Klassenintervalls, d.h., es gilt

$$\left(\underline{x}_{i,\delta}^* > \overline{k}_i\right) \vee \left(\overline{x}_{i,\delta}^* < \underline{k}_i\right)$$

3. Trifft weder (1) noch (2) zu, liegt der Deltaschnitt teilweise im Klassenintervall.

Für jedes δ und jede Klasse K_i wird in Folge ein Zähler $\text{count}_{full,\delta}(K_i)$ geführt, der die Anzahl der vollständig im Klassenintervall liegenden Deltaschnitte enthält und ein weiterer Zähler $\text{count}_{partial,\delta}(K_i)$, der die Anzahl der teilweise im Klassenintervall liegenden Deltaschnitte speichert.

3.5.1.1 OP_Analyze

Die Implementierung des Algorithmus zur Histogramm-Erstellung implementiert FuzzyData in der Dynamic Link Library „OP_Analyze.dll“. Diese analysiert ein vom Benutzer ausgewähltes Dataset und generiert daraus ein sogenanntes „Workset2“-File, dessen Inhalt die Ergebnisse der zuvor aufgeschlüsselten Berechnungen widerspiegelt.

Es werden somit alle im Dataset enthaltenen unscharfen und reellen Zahlendaten pro Deltaebene auf deren Lage gegenüber den einzelnen Klassen untersucht und pro Klasse die Anzahl der exakt, teilweise innerhalb und außerhalb liegenden Zahlen errechnet. Es kann also – wie zum Beispiel in Abbildung 3.10 ersichtlich – somit auch der Fall eintreten, dass eine unscharfe Zahl auf einem Deltaschnitt vollständig innerhalb einer Klasse liegt, auf einem anderen Deltaschnitt aber nur teilweise.

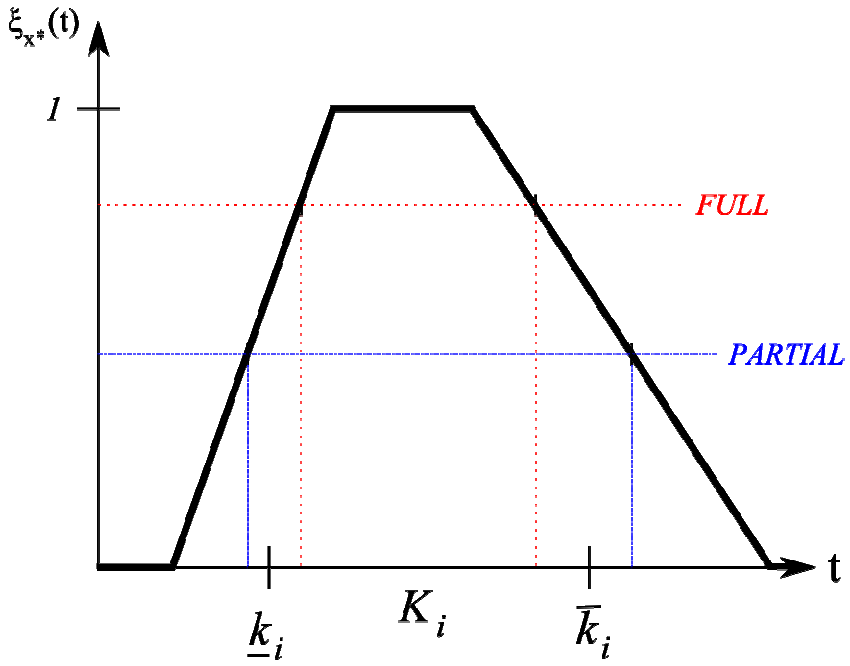

Abbildung 3.10 Änderung der Beziehung zur Klasse über Deltaschnitte hinweg

3.5.1.2 Workset2 Files

Workset2 Dateien speichern die von OP_Analyze errechneten Ergebnisse in einem Binärformat, welches in Folge als dreidimensionales Histogramm visualisiert werden kann. Das Dateiformat selbst ist relativ simpel aus einem Header- und Datenblock aufgebaut.

Der Dateiheader sieht zwei Felder zur Überprüfung des Dateiformats vor: cbSize und dVersion. Das Feld „cbSize“ enthält die Bytegröße des ANALYZEDDATA2_HEADER, um sicherzustellen, dass die erzeugte Datei tatsächlich mit der vorliegenden Variante des structs arbeitete. Die Variable dVersion ist generell eine von OP_Analyze vorgegebene Variable, welche mit der Programmversion des Analysealgorithmus versehen ist. Stimmen die beiden Werte nicht mit den erwarteten Werten überein, kann daher davon ausgegangen werden, dass die vorliegende Datei entweder defekt oder kein tatsächliches Workset2-File ist.

```
typedef struct ANALYZEDDATA2_HEADER {

      // <description>size of structure</description>
      DWORD       cbSize;

      // <description>version number, should be consistent with
      // ANALYZERVERSION</description>
      DOUBLE      dVersion;

      DWORD       dwClassCount;
      DWORD       dwDeltaCount;
```

```
} __ANALYZEDDATA2_HEADER;
```

Dem Dateiheader-Struct folgt danach aneinander gereiht die Klassengrenzen und darauf folgend die in der Häufigkeitsverteilungs-Analyse durchgeführten Deltaschnittwerte als DOUBLE Werte. Wie in Abbildung 3.11 erkennbar, finden sich somit bei n Klassen $n+2$ DOUBLE Werte, da sowohl die unterste Untergrenze, d.h., $\underline{k}_1$ und die oberste Obergrenze $\overline{k}_n$ gespeichert werden muss.

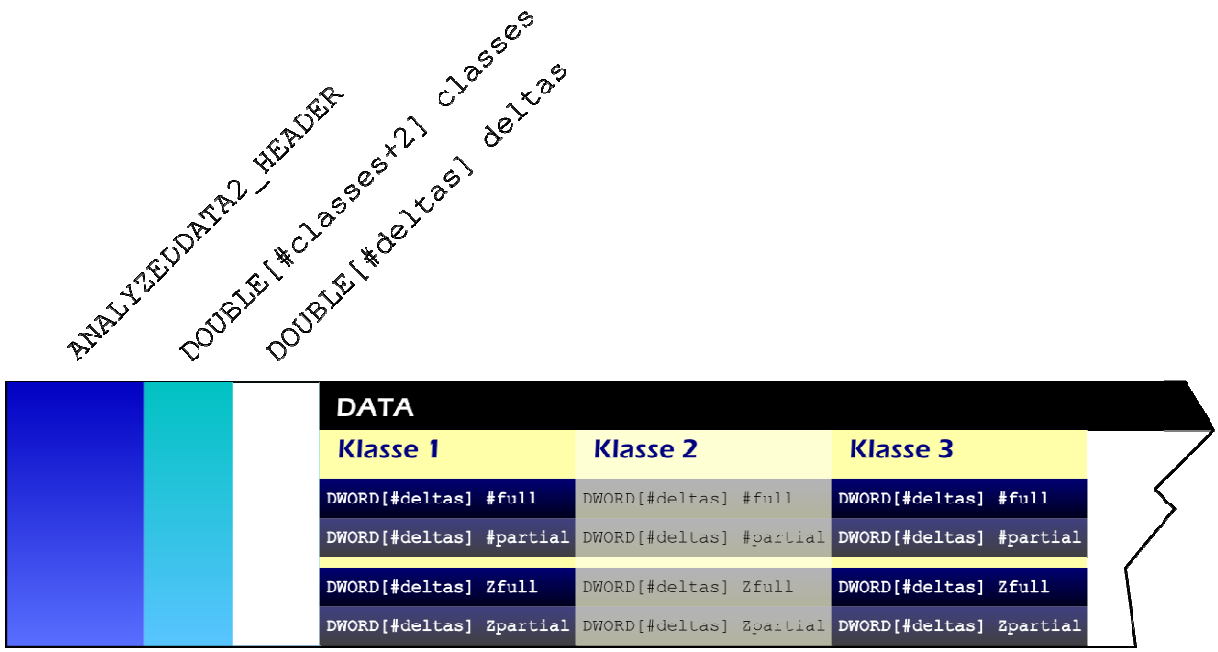

Abbildung 3.11: Struktur einer Workset2 Dateiheaders

Im Datenblock finden sich pro Klasse die einzelnen Berechnungsergebnisse:

- Für jeden Deltaschnitt die Anzahl der voll in der Klasse liegenden Beobachtungen
- Für jeden Deltaschnitt die Anzahl der teilweise in der Klasse liegenden Beobachtungen
- Für jeden Deltaschnitt der Z-Wert, welcher die voll in die Klasse fallenden Beobachtungen repräsentiert
- Für jeden Deltaschnitt der Z-Wert, welcher die teilweise in die Klasse fallenden Beobachtungen repräsentiert

Somit ergibt sich pro Klasse und Deltaschnitt die Information, wieviele Beobachtungen vollständig in die Klasse fallen und wieviele teilweise. Ergänzend findet man diese Werte auch bereits durch die Größe der Stichprobe und Klassenbreite dividiert um sie zum Beispiel als dreidimensionales Histogramm auf der Z-Achse auftragen zu können.

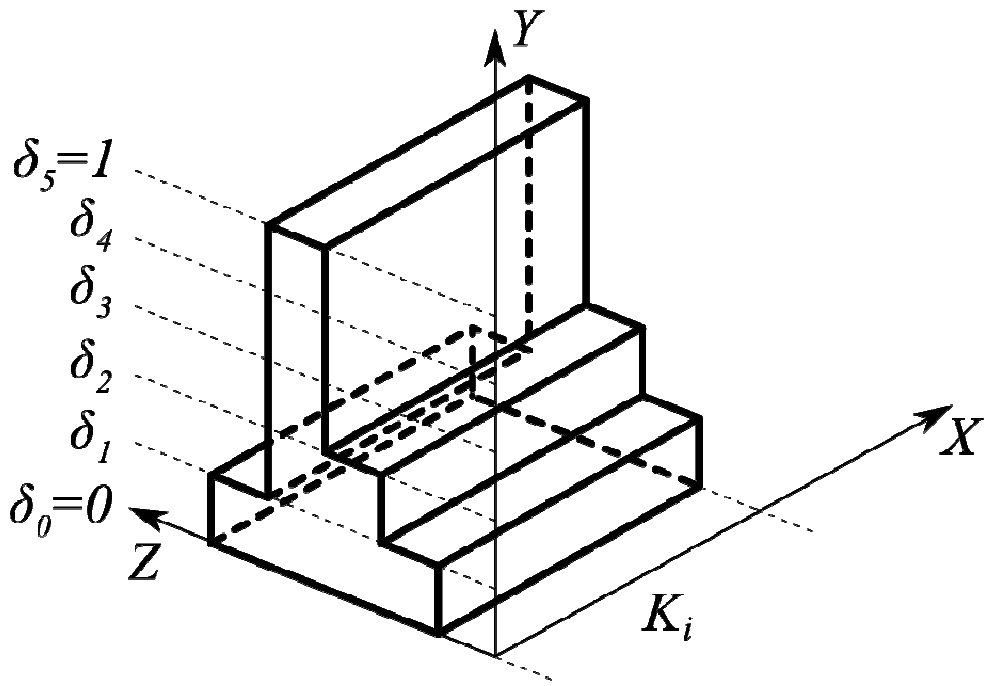

Abbildung 3.12: Dreidimensionale Darstellung der Histogrammdaten einer Klasse

3.5.1.3 Algorithmus für Schnittprüfung

Wie zuvor beschrieben, werden die einzelnen Deltaschnitte auf deren Lage zu den Klassengrenzen hin untersucht. FuzzyData nutzt hierfür die Methode „CutCheck".

```
enum CUTTYPE CutCheck (
      FLOAT fBorderLeft, FLOAT fBorderRight,
      FLOAT fCut1, FLOAT fCut2
)
```

Diese nimmt die untere (fBorderLeft) und obere (fBorderRight) Klassengrenze, die untere (fCut1) und obere (fCut2) Deltschnittgrenzen als Parameter und liefert als CUTTYPE enum entweder FULL, PARTIAL oder NONE zurück.

```
if ( fCut1 >= fBorderLeft ) then begin
      // untere Deltaschnittgrenze über unterer Klassengrenze

      if ( fCut1 < fBorderRight ) then

            if ( fCut2 < fBorderRight ) then
                  return FULL;
            else
                  return PARTIAL;

      else

            return NONE;


else begin
            // untere Deltaschnittgrenze über der
            // unteren Klassengrenze
```

```
      if ( fCut2 < fBorderRight ) then begin
            // obere Deltaschnittgrenze unterhalb
            // oberer Klassengrenze

            if ( fCut2 < fBorderLeft ) then
                  return NONE;
            else
                  return PARTIAL;

      else

            return PARTIAL;


end

return NONE;
```

3.5.1.4 Visualisierung

Die dreidimensionale Darstellung des Histogramms kann nach Durchführung der Berechnungen z.B. mit Hilfe des von Eduard Hirsch erstellten Programms „FuzzyVisualisation“ (siehe Screenshot in Abbildung 3.13) vorgenommen werden. Dieses gestattet zum Beispiel auch das Histogramm beliebig im dreidimensionalen Raum zu rotieren und Details via Zoomfunktion zu vergrößern.

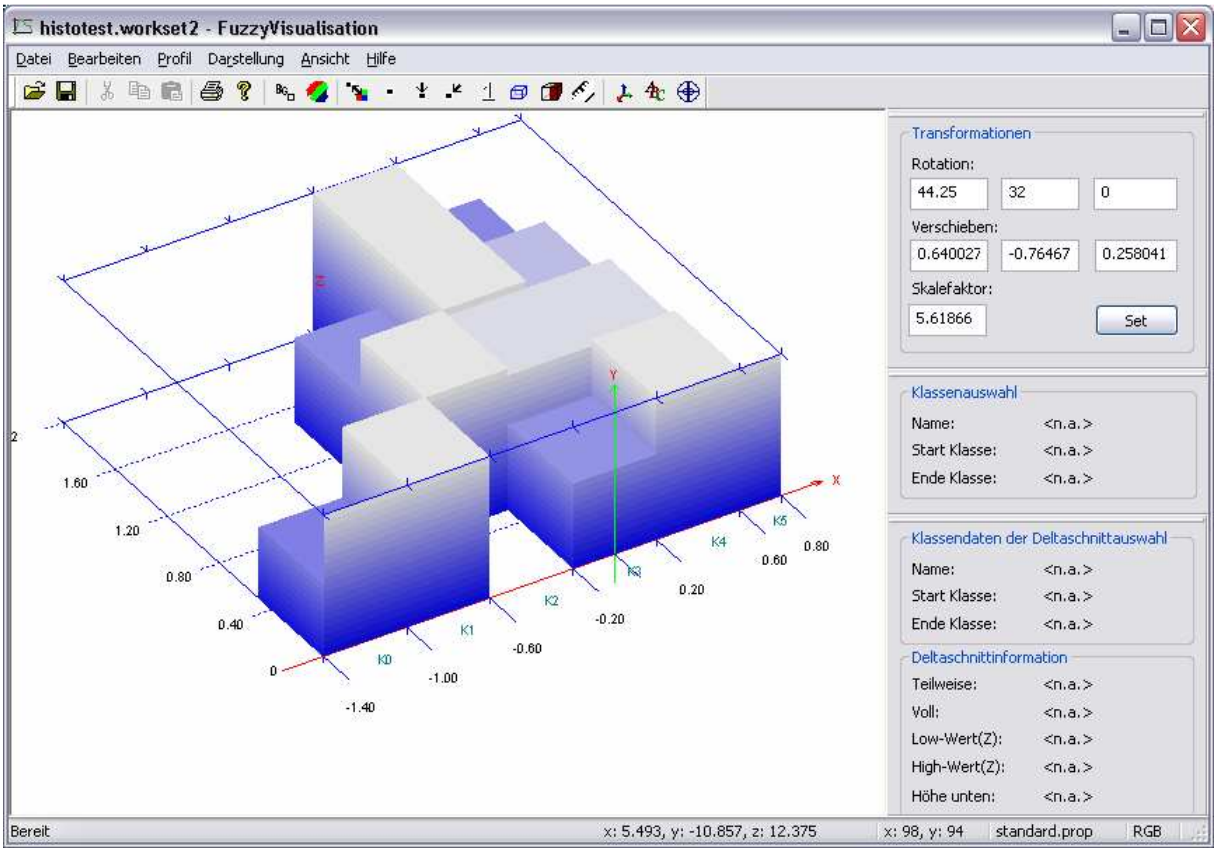

Abbildung 3.13: Beispiel für ein dreidimensionales Histogramm in FuzzyVisualisation

3.5.2 k-Fraktil

Das k-Fraktil wird vom Pluginmodul „OP_Fractile.dll“ berechnet. Dieses errechnet die Deltaschnitte des k-Fraktils wie in Kapitel 1.4.8 angegeben. Die resultierende unscharfe Zahl wird dabei nach und nach aus einzelnen Deltaschnitten in 0.01 großen Intervallen aufgebaut.

Vor dem effektiven Berechnungsalgorithmus konvertiert das Modul alle Beobachtungen in eine Polygon-Repräsentation in Form von CPolygon-Klassen. Auf diese Weise kann auf die einheitliche Schnittprüfungsmethoden GetXByY und GetXByY_Inverted zurückgegriffen werden. Somit finden sich alle Beobachtungen der Stichprobe in der Variable `polygons`.

Während der Berechnung des k-Fraktils wird über Aufruf der qsort-Funktion der Quicksort-Algorithmus eingesetzt. Die folgende Vergleichsfunktion nimmt dabei den Größenvergleich der Array-Elemente vor:

```
int sortHelper( DOUBLE dOne, DOUBLE dTwo )

begin

      if ( (dOne) < (dTwo) ) {

            return -1;

      } else if ( (dOne) == (dTwo) ) {

            return 0;

      }

      return 1;

end
```

Der konkrete Berechnungs-Algorithmus ist im Folgenden in Pseudocode angegeben:

1. Benötigte Variablen werden definiert

```
DOUBLE                        dVal, dDelta;
POLYGONLIST::iterator         polcount;
CNoDuplicateDoubleList        pNDList;
DWORD                         dwItem;
DOUBLE                        dLeft, dRight;
DWORD                         dwStep, dwMax;
DOUBLE                        pXLower[], pXUpper[];
DOUBLE                        pXLeft[], pXRight[];
int                           k;
```

2. Eine Liste der Deltaschnitte wird erzeugt im Intervall [0,1] mit Schrittgröße 0.01

```
pNDList = new CNoDuplicateDoubleList ();

for ( dVal = 0.0f; dVal < 1.0f; dVal += 0.01f ) do

      pNDList.AddValue(dVal);

pNDList.AddValue(1.0f);

pNDList.PurgeValueDif ();
```

3. Hilfsvariablen werden initialisiert:
    a. pXLower wird verwendet, um die unteren Intervallgrenzen der Deltaschnitte des Ergebniswerts zu speichern

b. pXUpper wird verwendet, um die oberen Intervallgrenzen der Deltaschnitte des Ergebniswerts zu speichern
c. pXLeft wird verwendet, um die Untergrenzen der Deltaschnitte der einzelnen Beobachtungen in einem Array zu speichern und zu sortieren
d. pXRight wird verwendet, um die Obergrenzen der Deltaschnitte der einzelnen Beobachtungen in einem Array zu speichern und zu sortieren

```
pXLower = new DOUBLE[polygons.size()];
pXUpper = new DOUBLE[polygons.size()];
pXLeft = new DOUBLE[pNDList.size()];
pXRight = new DOUBLE[pNDList.size()];


pResult = new CPolygon();
dwMax = pNDList.size();
```

4. Anschließend wird über die einzelnen Deltaschnitte iteriert.

```
for( dwItem = 0; dwItem < dwMax; dwItem++ ) do begin
      dDelta = pNDList.GetValue( dwItem );


      for( polcount = polygons.begin(), dwStep = 0;
            polcount < polygons.end(); polcount++, dwStep++ ) do
      begin
```

5. Für jede Beobachtung wird die untere (dLeft) und obere (dRight) Deltaschnitt-Intervallgrenze berechnet und dann in das Array pXLower und pXUpper eingetragen.

```
            dLeft = (polcount).GetXByY( dDelta );
            dRight = (polcount).GetXByY_Inverted( dDelta );

            pXLower[dwStep] = dLeft;
            pXUpper[dwStep] = dRight;

      end
```

6. Die beiden Arrays pXLower und pXUpper werden anschließend aufsteigend sortiert.

```
      qsort( pXLower, polygons.size(), sizeof(DOUBLE), sortHelper );
      qsort( pXUpper, polygons.size(), sizeof(DOUBLE), sortHelper );
```

7. Wie in Kapitel 1.4.8 beschrieben, wird das kleinste k ermittelt, das größer als np ist - in unserem Fall p (dFractileParam) mulitipliziert mit der Anzahl vorhandener Beobachtungen polgyons.size().

```
      k = ceil( dFractileParam * polygons.size() );
```

8. Die jeweiligen Werte werden danach in das Deltaschnitt-Array für den Ergebniswert übertragen.

```
      pXLeft[dwItem] = pXLower[k];

      pXRight[dwMax-dwItem-1] = pXUpper[k];


end
```

9. Die zuvor gesammelten Deltaschnittintervalle werden als Stützstellen in die CPolygon-Klasse in pResult übertragen.

```
for( dwStep = 0; dwStep < dwMax; dwStep++ ) do begin

      pResult.AddSamplingPoint( pXLeft[dwStep],
            pNDList.GetValue( dwStep ) );

end

for( dwStep = 0; dwStep < dwMax; dwStep++ ) do begin

      pResult.AddSamplingPoint( pXRight[dwStep],
            pNDList.GetValue( dwMax-dwStep-1 ) );

end


return pResult;
```

3.5.3 Stichprobenvarianz

Die Berechnung der Stichprobenvarianz ist im Modul OP_Varianz.dll implementiert. Das Modul greift zur Berechnung auf den in (7) vorgeschlagenen Algorithmus zur Berechnung der Unter- und Obergrenzen der Stichprobenvarianz zurück, um den Vorgang zu beschleunigen und zu vereinfachen.

Der implementierte Algorithmus nutzt zum Zwischenspeichern die beiden Strukturen VARIANCECUT und VARIANCEPOLYDEF, um den Aufbau einer polygonförmigen unscharfen Zahl über dessen Deltaschnitte zwischen zu speichern. Zur Vereinfachung des Algorithmus konvertiert das Modul zuerst alle Beobachtungen in der Stichprobendatei in eine polygonförmige Repräsentation, um ein einheitliche Zugriffs- und Schnittmethoden benutzen zu können. Die resultierenden polygonförmigen Zahlen befinden sich danach in der Listen-Variable „polygons“.

```
typedef struct VARIANCECUT {

      DOUBLE            dLeft;

      DOUBLE            dRight;

} __VARIANCECUT;
```

```
typedef struct VARIANCEPOLYDEF {

      VARIANCECUT       interval;

      DOUBLE            dY;

} __VARIANCEPOLYDEF;

typedef vector<VARIANCECUT>         VARIANCECUTLIST;
```

Der eigentliche Berechnungsalgorithmus ist im Folgenden in Pseudocode beschrieben:

1. Definition erforderlicher Hilfsvariablen.

```
CNoDuplicateDoubleList       pNDList;
POLYGONLIST::iterator        count;
DWORD                        dwItem = 0;
VARIANCECUTLIST              cutlist;
VARIANCECUTLIST              borders;
WORD                         dwCounter = 0, dwRoundCount = 0;
DOUBLE                       dY;
VARIANCEPOLYDEF              polycut;
VARIANCECUT                  cut;
DOUBLE                       dMean = 0.0f, dPreviousMean = 0.0f;
DOUBLE                       dPointVariance = 0.0f;
DOUBLE                       dStepMean = 0.0f, dStepVar = 0.0f;
DOUBLE                       dLowerVar = 0.0f, dUpperVar = 0.0f;
DOUBLE                       dPreviousVariance = 0.0f;
VARIANCECUTLIST::iterator    cutcount;
CPolygon                     pResult = new CPolygon();

pNDList = new CNoDuplicateDoubleList ();
```

2. Sammeln aller Delta- bzw. Y-Werte aus allen Beobachtungen der Stichprobe.

```
for ( count = polygons.begin(); count < polygons.end(); count++ ) do
begin

      for ( dwItem = 0; dwItem < (count).size(); dwItem++ ) do
      begin

            pNDList.add ( (count).GetPointByIndex(dwItem).dY );

      end

end
```

3. Wenn weniger als 10 Deltaschnitte aus der Stichprobe gesammelt wurden, wird die Liste der Deltaschnitte um alle Deltaschnitte in 0.01 Intervallschritten zwischen 0 und 1 ergänzt.

```
if ( pNDList.GetItemCount() < 10 ) then begin

      dY = 0.0f;

      for ( dwItem = 100; dwItem >0; dwItem-- ) do begin

            pNDList.AddValue(((dY += 0.01)>1.0f?(dY=1.0f):dY));

      end
```

```
end
```

4. Anschließend wird über alle in der Deltaliste gesammelten Deltas iteriert.

```
for ( dwCounter = 0; dwCounter < pNDList.size(); dwCounter++ ) do
begin
      // reset result cut information
      ZeroMemory ( &polycut, sizeof(VARIANCEPOLYDEF) );
      dY = polycut.dY = pNDList.GetValue(dwCounter);

      cutlist.clear();
      borders.clear();

      for ( count = polygons.begin(); count < polygons.end();
            count++ ) do begin
```

5. Zu jeder Beobachtung aus der Stichprobe werden für das betrachtete Delta die entsprechenden Intervallgrenzen des Deltaschnitts berechnet. Die Ergebnisse werden anschließend in die Listen cutlist und borders hinzugefügt.

```
            cut.dLeft = (count).GetXByY ( polycut.dY );
            cut.dRight = (count).GetXByY_Inverted ( polycut.dY );

            cutlist.push_back ( cut );
            borders.push_back ( cut );
      end

      dPreviousMean = 0.0f;
      dwRoundCount = 0;
```

6. Zunächst wird die Untergrenze des Deltaschnitts der Stichprobenvarianz berechnet. Die Schleife bricht nach einer vorgegebenen Anzahl an Iterationen ab um eine Endlosschleife zu verhindern, falls die gewünschte Abbruchbedingung – ein Auffinden der Untergrenze bis auf einen geringen Fehler von 10^{-7} genau – nicht erreicht wird. Diese Sicherheitsabfrage kann ggf. individuell angepasst werden um einen früheren Abbruchzeitpunkt zu erreichen.

```
      Do begin
            dMean = 0.0f;
            for ( cutcount = borders.begin();
                  cutcount < borders.end(); cutcount++ ) do begin
```

```
                dMean += (cutcount).dLeft;

            end

            dMean /= borders.size();

            if ( dwRoundCount ) then
                if ( fabs ( dMean - dPreviousMean ) <
                    0.00000001 ) then
                        break;



            dPreviousMean = dMean;
```

7. Zunächst wird der Durschnitt aus den zuvor errechneten Untergrenzen der Deltaschnitte berechnet und in dMean gespeichert; ist das berechnete dMean nicht einmal um 10^{-7} größer oder kleiner als ein zuvor berechneter Durchschnitt, wird die Schleife abgebrochen.

```
            for ( cutcount = borders.begin();
                cutcount < borders.end(); cutcount++ ) do

            begin

                (cutcount).dLeft =
                    max ( cutlist.at(cutcount -
                        borders.begin()).dLeft,

                    min ( cutlist.at(cutcount -
                        borders.begin()).dRight, dMean )
                    );
```

8. Für jede Beobachtung aus der Stichprobe wird nun falls erforderlich für die Untergrenze eine Korrektur vorgenommen. Ziel ist es, die Untergrenze möglichst nahe an den zuvor errechneten Mittelwert zu bewegen.
    a. Liegt der zuvor errechnete Mittelwert dMean innerhalb der Unter- (cutlist[].dLeft) und Obergrenze (cutlist[].dRight), wird als neue Untergrenze dMean angenommen.
    b. Liegt der zuvor errechnete Mittelwert dMean unterhalb der Untergrenze, wird die Untergrenze bei dLeft belassen.
    c. Liegt der zuvor errechnete Mittelwert dMean überhalb der Obergrenze dRight, so wird dLeft=dRight gesetzt.

```
            }

            dwRoundCount++;

        end while ( dwRoundCount < 10000000000000000 );
```

9. Das Ergebnis kann anschließend bereits in die Ergebnisvariable übertragen werden.

```
        pResult.AddSamplingPoint( dMean, dY );

        dwRoundCount = 0;

        dPreviousVariance = 0.0f;
```

10. Der gleiche Vorgang wird nun für die Obergrenzen fortgesetzt. Es wird zunächst der Mittelwert der ursprünglichen Untergrenzen berechnet und in dMean gespeichert.

```
dMean = 0.0f;

for ( cutcount = borders.begin(); cutcount < borders.end();
       cutcount++ ) do begin

      dMean += ( (cutcount).dRight =
            cutlist.at(cutcount-borders.begin()).dLeft );

end

dMean /= borders.size();
```

11. Dann wird die Stichprobenvarianz aus den ursprünglichen Untergrenzen der Deltaschnitte berechnet.

```
dPointVariance = 0.0f;

for ( cutcount = borders.begin(); cutcount < borders.end();
       cutcount++ ) do begin

      dPointVariance += pow ( (cutcount).dRight, 2.0f );

end

dPointVariance -= borders.size() * pow ( dMean, 2.0f );

dPointVariance /= borders.size() - 1;


do begin

      for ( cutcount = borders.begin();
            cutcount < borders.end(); cutcount++ ) do begin
```

12. Für jede Beobachtung in der Stichprobe wird anschließend deren Einfluss auf die Stichprobenvarianz heraus gerechnet. Dazu wird zuerst der Mittelwert des Vektors der Deltaschnitt-Untergrenzen und dann dessen Stichprobenvarianz ohne die gerade betrachtete Beobachtung errechnet über die Formel

$$\overline{x}_{n-1} = \frac{n\overline{x}_n - x_{max,i}}{n-1}$$

$$s_{n-1}^2 = \frac{n-1}{n-2}\left(s_n^2 - \frac{\left(\overline{x}_{n-1} - x_{max,i}\right)^2}{n}\right)$$

```
            dStepMean = (borders.size() * dMean -
                  (cutcount).dRight) /
                        (borders.size()-1);
```

```
                dStepVar = (borders.size()-1)/(borders.size()-2) *
                       ( dPointVariance -
                      ( pow ( ( dStepMean – (cutcount).dRight ),
                            2.0f ) / borders.size() ) );
```

13. Dann werden zwei unterschiedliche Varianten der Stichprobenvarianz ermittelt: im ersten Fall wird für die betrachtete Beobachtung die Obergrenze des ursprünglichen Deltaschnitts ausgewählt und das Ergebnis in dUpperVar gespeichert. Im zweiten wird für die betrachtete Beobachtung die Untergrenze des ursprünglichen Deltaschnitts ausgewählt und das Ergebnis in dLowerVar gespeichert. Danach werden die beiden errechneten Varianz-Ergebnisse miteinander verglichen.

```
                dUpperVar = (borders.size()-2)/(borders.size()-
                                1)*dStepVar +
                                ( pow ( dStepMean -
                                cutlist.at(cutcount-
                                borders.begin()).dRight, 2.0f ) )/
                                borders.size();

                dLowerVar = (borders.size()-2)/(borders.size()-
                                1)*dStepVar +
                                ( pow ( dStepMean -
                                cutlist.at(cutcount-
                                borders.begin()).dLeft, 2.0f ) ) /
                                borders.size();
```

14. Ist dUpperVar größer als dLowerVar, wird für die betrachtete Beobachtung die ursprüngliche Obergrenze des Deltaschnitts übernommen, ansonsten die Untergrenze.

```
                (cutcount).dRight =
                      (dUpperVar > dLowerVar)?
                      cutlist.at(cutcount-borders.begin()).dRight:
                      cutlist.at(cutcount-borders.begin()).dLeft;

                dPointVariance = max ( dUpperVar, dLowerVar );

                dMean = ( dStepMean * ( borders.size()-1 ) +
                       (cutcount).dRight ) / borders.size();
```

15. Die Schleife wird solange durchlaufen, bis sich nach Durchlauf aller Beobachtungen die berechnete Stichprobenvarianz im Vergleich zur zuletzt berechneten Stichprobenvarianz nicht mehr vergrößert hat.

```
            end


            dMean = 0.0f;

            for ( cutcount = borders.begin();
                  cutcount < borders.end(); cutcount++ ) do

            begin

                  dMean += (cutcount).dRight;

            end
```

```
            dMean /= borders.size();

            if ( dwRoundCount )
                  if ( fabs ( dPointVariance - dPreviousVariance ) <
                        0.00000001f )
                              break;

            dPreviousVariance = dPointVariance;



            dwRoundCount++;

      end while ( dwRoundCount < 10000000000000000 );
```

16. Das Ergebnis kann wiederum in die Ergebnisvariable pResult übertragen werden.

```
      pResult.AddSamplingPoint( dPointVariance, dY );

end

delete pNDList;
```

17. Die zuvor erstellte polygonförmige Ergebniszahl wird zurückgegeben.

```
return pResult;
```

3.5.4 Streuung

Wie in (27) definiert, ist die Standardabweichung definiert als die Quadratwurzel der Varianz. Das Modul „OP_Streuung“, das die Standardabweichung einer Stichprobendatei berechnet, greift daher zunächst auf das Modul „OP_Varianz“ zurück, um die Varianz der Stichprobe zu errechnen. Aus dem Ergebnis zieht OP_Streuung anschließend die Wurzel und gibt diese als Berechnungsergebnis zurück.

Der Algorithmus ist im Folgenden in Pseudo-Code angegeben:

1. Es werden Hilfsvariablen definiert

```
BOOL        bNegative;

DWORD       dwCount;

DOUBLE      dValue;

BOOL        bSuccess;
```

2. Die Varianz wird berechnet. Schlägt die Berechnung fehl, wird der Algorithmus abgebrochen.

```
if ( ! ( bSuccess = CVariance::Initialize() ) ) then

      return FALSE;
```

3. Für jede Stützstelle bzw. über die vorhandenen Deltaschnitte hinweg wird die Wurzel der Varianz errechnet.

```
// prep up the result by sqrt-ing
```

```
for ( dwCount = 0; dwCount < result.dwFlag; dwCount++ ) do
begin
      bNegative = ((dValue = result.pPolNumber[dwCount].fX) < 0);
      result.pPolNumber[dwCount].fX = sqrt(fabs(dValue)) *
                                          (bNegative?-1:1);
end
return result;
```

3.5.5 Maximum

Das Modul „OP_Minimum“ gestattet die Berechnung des Stichprobenminimums. Dazu konvertiert dieses anfangs alle Beobachtungen in der Stichprobe in eine polygonförmige Repräsentation in der Variable „polygons“.

Der konkrete Algorithmus zur Berechnung ist im Folgenden in Pseudocode angegeben:

1. Es werden Hilfsvariablen deklariert

```
CNoDuplicateDoubleList       pNDList;
POLYGONLIST::iterator        count;
DWORD                        dwItem;
DWORD                        dwCounter, dwRoundCount;
DOUBLE                       dY;
VARIANCEPOLYDEF              polycut;
VARIANCECUT                  cut;
DOUBLE                       dMean, dPreviousMean;
DOUBLE                       dPointVariance;
DOUBLE                       dStepMean, dStepVar;
DOUBLE                       dLowerVar, dUpperVar;
DOUBLE                       dPreviousVariance;
VARIANCECUTLIST::iterator    cutcount;
CPolygon                     pResult = new CPolygon();
```

2. Es wird eine Liste aller Deltaschnitte der in der Stichprobe enthaltenen Beobachtungen aufgestellt. pNDList enthält somit alle Delta-Werte, für die mindestens eine Stützstelle in der Stichprobe existiert.

```
pNDList = new CNoDuplicateDoubleList ();
for ( count = polygons.begin(); count < polygons.end(); count++ ) do
begin
```

```
      for ( dwItem = 0; dwItem < (count).GetSamplingCount();
            dwItem++ ) do begin

            pNDList.AddValue((count).GetPointByIndex(dwItem).dY );

      end

end


for ( dwCounter = 0; dwCounter < pNDList.size(); dwCounter++ ) do

begin
```

3. Es wird über jedes dieser Deltawerte iteriert und das Deltaschnitt-Intervall für jede Beobachtung berechnet. Ist entweder die Unter- oder Obergrenze des Deltaschnitt-Intervalls größer als jede bisher betrachtete Unter- oder Obergrenze, so wird dieser jeweils als die Unter- bzw. Obergrenze des Maximumwerts übernommen.

```
      polycut.interval.dLeft = polygons.at(0).GetXByY (
            ( dY = pNDList.GetValue(dwCounter) ) );

      polycut.interval.dRight = polygons.at(0).GetXByY_Inverted(dY);

      for( count = polygons.begin(); count < polygons.end();
            count++ ) do begin

            polycut.interval.dLeft = max ( polycut.interval.dLeft,
             (count).GetXByY ( dY ) );

            polycut.interval.dRight = max ( polycut.interval.dRight,
                  (count).GetXByY_Inverted ( dY ) );

      end

      pResult.AddSamplingPoint( polycut.interval.dLeft, dY );

      pResult.AddSamplingPoint( polycut.interval.dRight, dY );
```

4. Nach Durchlauf der Schleife enthält polycut somit die größte Unter- und Obergrenze aller betrachteten Deltaschnitt-Intervalle. Dieses Intervall wird in die Ergebnisvariable pResult als Stützstellen übertragen.

```
end
```

5. Das Ergebnis wird zurückgeliefert.

```
return pResult;
```

3.5.6 Minimum

Äquivalent zur Berechnung des Stichproben-Maximums berechnet das Modul „OP_Minimum“ das Stichprobenminimum. Der Algorithmus ist prinzipiell der gleiche, übernimmt in der Hauptschleife jedoch die kleinsten Unter- und Obergrenzen der einzelnen Deltaschnitt-Intervalle der Beobachtungen.

Der Algorithmus ist im Folgenden in Pseudocode angegeben.

1. Hilfsvariablen werden definiert.

```
DWORD                   dwCount, dwBarrier = GetNumberCount();
DWORD                   dwDeltas, dwDeltaCount;
CONST DOUBLE            pDeltas[];
DOUBLE                  dLeft, dRight;
CPolygon                pPolygon;

dwDeltas = GetDeltaCount();
pDeltas = GetDeltaList();

pResult = new CPolygon();

for ( dwDeltaCount = 0; dwDeltaCount < dwDeltas; dwDeltaCount++ ) do
begin
```

2. Es wird über alle Deltawerte iteriert.

```
      for ( dwCount = 0; dwCount < dwBarrier; dwCount++ ) do
      begin
```

3. Jede der in der Stichprobe vorhandenen Beobachtungen werden betrachtet und deren untere und obere Deltaschnitt-Intervallgrenze berechnet. Wenn die aktuell betrachteten Intervallgrenzen kleiner sind als alle bisher betrachteten Werte, wird entsprechend der neue Wert übernommen und in dLeft bzw. dRight gespeichert.

```
            if ( ! dwCount ) then begin
                  dLeft = GetDeltacutPolygon(dwCount).
                              GetLowerValue(dwDeltaCount);

                  dRight = GetDeltacutPolygon(dwCount).
                              GetUpperValue(dwDeltaCount);

            end else begin
                  dLeft = min ( dLeft, GetDeltacutPolygon(dwCount).
                                    GetLowerValue(dwDeltaCount) );

                  dRight = min ( dRight, GetDeltacutPolygon(dwCount).
                                    GetUpperValue(dwDeltaCount) );

            end

      end
```

4. Die gefundenen kleinsten Intervallgrenzen werden als Stützstellen in das Ergebnis-Polygon übertragen.

```
      pResult.AddSamplingPoint( pLeft, pDeltas[dwDeltaCount] );
      pResult.AddSamplingPoint( pRight, pDeltas[dwDeltaCount] );
end
```

5. Das Ergebnis wird zurückgeliefert.

```
return pResult;
```

3.5.7 Geglättete empirische Verteilungsfunktion

Die geglättete empirische Verteilungsfunktion wird vom Modul „OP_SmoothEmpDist.dll“ berechnet. Zur Vereinfachung der Berechnung transformiert das Modul vor der Berechnung alle Zahlendaten in eine polygonförmige Repräsentation und speichert die Daten in der Variable „polygons“. Vor der Berechnung wird zusätzlich ein Dateiname für eine Ausgabedatei in der Variable „strOutname“ erfasst; die Ausgabe des Algorithmus erfolgt als workset2-Datenfile, welches mit Hilfe von FuzzyVisualisation als 3-dimensionale Kurve betrachtet werden kann.

Der Algorithmus für die Berechnung ist im Folgenden in Pseudo-Code aufgeschlüsselt.

1. Es werden Hilfsvariablen definiert

```
DOUBLE                      dxMin, dxMax;
DOUBLE                      dVal, dDelta;
POLYGONLIST::iterator       polcount;
CNoDuplicateDoubleList      pNDList;
DWORD                       dwItem;
DOUBLE                      dLeft, dRight;
DWORD                       dwLowerMatch, dwUpperMatch;
DWORD                       pLower[], pUpper[];
DOUBLE                      pZLower[], pZUpper[];
HANDLE                      hFile;
DWORD                       dwClassCount;
DOUBLE                      pClasses[], pDeltas[];
DWORD                       dwStep;
```

2. Eine Liste an Deltaschnitt-Werten mit Intervallgröße 0.01 wird erstellt.

```
pNDList = new CNoDuplicateDoubleList ();
```

```
for ( dVal = 0.0f; dVal < 1.0f; dVal += 0.01f ) do

      pNDList.AddValue(dVal);

pNDList.AddValue(1.0f);
```

3. Es werden die Unter- und Obergrenzen der Stichprobe übernommen.

```
dxMin = polygons.minimumX();

dxMax = polygons.maximumX();
```

4. Innerhalb des Intervalls zwischen dxMin und xMax wird eine Klasseneinteilung vorgenommen. Die Klassenbreite wird aus der Konstante SAMPLERATE definiert. Die resultierenden Klassengrenzen werden im Array pClasses festgehalten.

```
pClasses = new DOUBLE[(dwClassCount=((dxMax-dxMin)/SAMPLERATE];

for ( dwItem = 0, dVal = dxMin; dVal < dxMax;
      dVal += SAMPLERATE, dwItem++ ) do begin

      pClasses[dwItem] = dVal;

end
```

5. Es wird ein Outputfile erzeugt, in das die Ergebniswerte während der Berechnung geschrieben werden. Als Parameter werden übergeben: ein Dateiname, die Anzahl der Klassen, die Anzahl der Deltaschnitte, die Klassengrenzen und Deltawerte.

```
hFile = CreateAnalyzedSet2 (strOutname, dwClassCount, pNDList.size(),
            pClasses, pNDList.values() );
```

6. Speicher-Arrays werden allokiert, um die Ergebniswerte während des Durchlaufs zu speichern.

```
pLower = new DWORD[pNDList.GetItemCount()+1];
pUpper = new DWORD[pNDList.GetItemCount()+1];
pZLower = new DOUBLE[pNDList.GetItemCount()+1];
pZUpper = new DOUBLE[pNDList.GetItemCount()+1];



for ( dwStep = 0, dVal = dxMin; dwStep < dwClassCount;
      dVal += SAMPLERATE, dwStep++ ) do

begin
```

7. Der Algorithmus iteriert nun über die einzelnen Klassenintervalle

```
      for ( dwItem = 0; dwItem<pNDList.GetItemCount(); dwItem++ ) do

      begin
```

8. und in jedem Klassenintervall über die einzelnen Deltaschnittwerte

```
            dDelta = pNDList.GetValue(dwItem);
            dwLowerMatch = dwUpperMatch = 0;


            for ( polcount = polygons.begin();
                  polcount < polygons.end(); polcount++ ) do begin
```

9. Für jede Beobachtung wird nun jeweils dessen Deltaschnitt-Intervall [dLeft,dRight] berechnet für das Delta dDelta.

```
                  dLeft = (polcount).GetXByY(dDelta);
                  dRight = (polcount).GetXByY_Inverted(dDelta);
```

10. Je nach Lage des Deltaschnittintervalls, wird der Deltaschnitt als teilweise oder vollständig in der Klasse liegend angerechnet.

```
                  if ( dRight <= dVal ) then
                        dwLowerMatch ++;


                  if ( (dLeft <= dVal) ) then
                        dwUpperMatch ++;
            end


            pLower[dwItem] = dwLowerMatch;
            pUpper[dwItem] = dwUpperMatch;
            pZLower[dwItem] = dwLowerMatch) / polygons.size();
            pZUpper[dwItem] = dwUpperMatch) / polygons.size();
      end
```

11. Die Ergebnisse werden anschließend pro Klasse in das Ergebnisfile geschrieben.

```
      WriteAnalyzedSet2_Class  (  hFile,  "",  pNDList.GetItemCount(),
            pLower, pUpper, pZLower, pZUpper );
end
```

12. Zuletzt wird die Datei geschlossen und damit die Berechnung beendet.

```
CloseHandle ( hFile );
return TRUE;
```

3.5.8 Summenkurve

Die Summenkurve wird nach dem im Abschnitt 1.4.7 beschriebenen Prinzip durch das Modul „OP_Sumcurve.dll“ berechnet. Das Modul konvertiert zur Vereinfachung der Berechnung zunächst alle in der Stichprobe enthaltenen Beobachtungen in eine polygonförmige Zahl und speichert diese in der Listenvariable „polygons“.

Der Algorithmus zur Summenkurvenberechnung ist folgend in Pseucode angegeben:

1. Hilfsvariablen werden definiert.

```
DOUBLE                  dxMin, dxMax, dVal;
POLYGONLIST::iterator   polcount;
DOUBLE                  dX;
DOUBLE                  dY;
DWORD                   dwIndex;
```

2. Es wird das Stichprobenminimum und –maximum übernommen.

```
dxMin = polygons.minimumX();
dxMax = polygons.maximum();


pResult = new CPolygon();
for ( dX = dxMin; dX <= (dxMax+0.5f); dX += SAMPLERATE ) do
begin
      dY = 0.0f; dVal = 0.0f;
```

3. Es werden SAMPLERATE große Klassen zwischen dem Stichprobeniminimum und –maximum erzeugt und durchlaufen. Für jede der SAMPLERATE großen Intervalle werden die Beobachtungen näher betrachtet:

```
      for ( polcount = polygons.begin();
            polcount < polygons.end(); polcount++ ) do begin
            if ( (polcount).GetSamplingCount() == 3 ) then
            begin
                  if ( (polcount).GetXByIndex(0) ==
                        (polcount).GetXByIndex(2) ) then begin
```

4. Handelt es sich bei der Beobachtung um eine reelle Zahl, wird sowohl dY als auch dVal inkrementiert und mit der nächsten Beobachtung fortgesetzt.

```
                        if ( (polcount).GetXByIndex(0) <= dX ) then
                        begin
```

```
                              dY += 1.0f; dVal += 1.0f;
                        end
                        continue;
                  end
            end
```

5. dVal wird um die Gesamtfläche der charakterisierenden Funktion der Beobachtung erhöht.

```
            dVal += (polcount).IntegrateArea(
                  (polcount).GetXByIndex(
                  (polcount).GetSamplingCount()-1 ) +1.0f );
```

6. dY wird inkrementiert um das Integral der charakterisierenden Funktion bis zur Intervallobergrenze.

```
            dY += ( (polcount).IntegrateArea(dX)  );
      end
```

7. Die addierten Integrale werden durch die Gesamtflächen dividiert und als Stützstelle in das Ergebnispolygon eingefügt.

```
      dY /= dVal;
      pResult.AddSamplingPoint(dX, dY);
end
```

8. Das Ergebnis wird zurückgegeben.

```
return pResult;
```

4 Zusammenfassung

Bei jeder Messung tritt üblicherweise ein gewisser Grad an Unschärfe auf. Die mit Mengen arbeitende Fuzzy-Logik, die daran anknüpfenden unscharfen Zahlen sowie die jeweils darauf aufbauenden Methoden machen sich diese Eigenschaft zu Nutze.

Im Rahmen dieser Arbeit wurde anfangs eine Übersicht über die Grundlagen unscharfer Zahlen gegeben und die dafür existierenden statistischen Auswertungsmethoden beleuchtet. Danach wurde das Softwareprogramm „FuzzyData“ vorgestellt.

FuzzyData ist effektiv eine „Proof-of-Concept“ Applikation und gestattet Stichproben aus reellen und scharfen Beobachtungen statistisch auszuwerten. Das Programm bietet zusätzlich die Möglichkeit, auf Pseudo-Zufallszahlen beruhende Stichproben zu generieren und so die statistischen Operationen zu veranschaulichen. Die Stichprobendaten können auf Wunsch auch manuell nachbearbeitet werden.

Die Applikation wurde in C/C++ mit Hilfe von Microsoft Visual Studio 2003 und Visual Studio 2005 entwickelt, um ein Maximum an Portabilität zu gewährleisten. Die mathematischen Operationen und Dateioperationen wurden in eigenen Dynamic Linked Libraries (DLLs) implementiert, um die Nutzung der Algorithmen auch durch externe Werkzeuge zu ermöglichen.

Die von FuzzyData eingesetzten Methoden und Algorithmen sind in dieser Arbeit in Pseudo-Code wiedergegeben, um einerseits das Verständnis des Programms als auch ein eventuelles Aufgreifen der Methoden zu gestatten.

Der Einsatz von FuzzyData und den damit verbunden Operations-Modulen gestattet dem Benutzer somit, unscharfe Stichprobendaten zu analysieren und zu Berechnungen zu verwenden ohne typische Verluste durch Abbildung auf reelle Zahlen.

Ausblick

Die von FuzzyData implementierten Methoden ermöglichen nun, die Methoden und Operationen auf unscharfe Zahlen weiter fortzuführen: auf der einen Seite ist eine technologische Weiterentwicklung – zum Beispiel auf andere Plattformen und Sprachen (etwa. C# oder Java) – denkbar, auf der anderen Seite bietet sich die Implementierung zusätzlicher, komplexerer statistischer Methoden auf Basis der vorhandenen Codestrukturen, an.

Ein weiterer Bereich, in dem FuzzyData erweitert werden könnte, ist im Embedded- bzw. Hardware-Bereich. Durch Implementierung von Schnittstellen zu konkreten Messgeräten wäre es möglich, unscharfe Stichprobendaten aus der Praxis (z.B. geologische Messungen) zu erhalten und z.B. durch eine entsprechende Studie die Effektivität der unscharfen statistischen Methoden im Praxisfall auf die Probe zu stellen.

5 Abbildungsverzeichnis

6 Formelzeichenindex

Formel	Beschreibung
$I_A(x)$	Indikatorfunktion der Menge A
x^*	Unscharfe Zahl
$\xi_{x^*}(\cdot)$	Charakterisierende Funktion für die unscharfe Zahl x^*
$C_\delta(x^*)$	Deltaschnitt-Intervall für die unscharfe Zahl x^* und das gewählte δ
$\mathbb{E}X$	Erwartungswert der stochastischen Größe X
$\hat{m}^k$	k-tes empirisches Moment
$s_n^2(x_1, \dots, x_n)$	Stichprobenvarianz
$\underline{x}_\delta$	Untere Deltaschnitt-Intervallgrenze der unscharfen Zahl x^* für gewähltes Delta δ
$\overline{x}_\delta$	Obere Deltaschnitt-Intervallgrenze der unscharfen Zahl x^* für gewähltes Delta δ
$\hat{F}_n^*(x)$	Unscharfe geglättete empirische Verteilungsfunktion
$F^{-1}(\cdot;\cdots;\cdot)$	Unscharfe invertierte empirische Verteilungsfunktion
$S_n^*(x)$	Unscharfe Summenkurve
q_p^*	Unscharfes p-Quantil für unscharfe Verteilungsfunktion

7 Literaturverzeichnis

1. **Merk:** Finanz Lexikon: Modellunsicherheit. *www.finanz-lexikon.net.* [Online] 2007. [Zitat vom: 02. Februar 2009.] http://www.finanz-lexikon.net/Modellunsicherheit_2155.html.

2. **Menger, K.:** *Ensembles flous et fonctions aleatoires.* Paris : Comptes Rendus Acad. Sci., 1951.

3. **Voß, W.; Buttler, G.:** *Taschenbuch der Statistik.* s.l. : Carl Hanser Verlag München, 2004. ISBN 3-446-22605-2.

4. **Zadeh, L.:** Fuzzy sets. *Information and Control.* 8, page 338-353, 1965.

5. **Zadeh, L. A.:** *In: Fox J, editor. System Theory.* Brooklyn, NY : Polytechnic Press, 1965.

6. **Viertl, R.; Hareter, D.:** *Beschreibung und Analyse unscharfer Information.* Wien : Springer, 2005. ISBN 978-3-211-23877-6

7. **Viertl, R.:** *Statistical Methods for Non-Precise Data.* Boca Raton, Florida : CRC Press, 1996.

8. **Knosala, R.; Breiing, A.:** *Bewerten technischer Systeme.* s.l. : Springer Berlin Heidelberg New York, 1997. ISBN 3-540-61086-3.

9. **Viertl, R.:** *Einführung in die Stochastik, 2. Auflage.* s.l. : Springer Wien New-York, 1997. ISBN 3-211-83027-8.

10. **Cai, K.; Zhang, L.:** Fuzzy Reasoning as a Control Problem. *IEEE TRANSACTIONS ON FUZZY SYSTEMS.* 3, 2008, Bd. 16.

11. **Gaines, B. R.:** *Fuzzy Reasoning and the logics of uncertainty, Proceedings of the 6th international symposium on Multiple-valued logic.* Logan, Utah, United States : IEEE Computer Society Press, 1976.

12. **Zadeh, L. A.:** Fuzzy logic, neural networks, and soft computing. *Communications of the ACM.* Issue 3, 1994, Bd. Volume 37, March 1994.

13. **Bloch, I.:** *Fuzzy sets in image processing, Proceedings of the 1994 ACM symposium on Applied computing.* Phoenix, Arizona, United States : ACM, 1994.

14. **Bloch, I.; Maitre, H.:** *Fuzzy distances and image processing, Proceedings of the 1995 ACM symposium on Applied computing.* Nashville, Tennessee, United States : ACM, 1995.

15. **Nie, Y.; Barner, K. E.** The Fuzzy Transformation and Its Applications in Image Processing. *IEEE TRANSACTIONS ON IMAGE PROCESSING.* 4, 2006, Bd. 15.

16. **Intan, R.; Makaidono, M.:** *A proposal of fuzzy thesaurus generated by fuzzy covering, NAFIPS 2003, 22nd International Conference of the North American Fuzzy Information Processing Society.* Surabaya, Indonesia; Kawasaka-shi, Kanagawa-ken, Japan : IEEE, 2003. ISBN 0-7803-7918-7/03.

17. **TU Wien, Database and Expert Systems Group.** DBAI: *StarFLIP.* [Online] 7. August 1997. [Zitat vom: 11. Februar 2009.] http://www.dbai.tuwien.ac.at/proj/StarFLIP/.

18. **Hartwig, R.:** Fool & Fox Homepage. *FOOL & FOX: Fuzzy system development tools.* [Online] 16. Mai 2005. [Zitat vom: 11. Februar 2009.] http://www.rhaug.de/fool/.

19. **Orchard, B.:** FuzzyJ Toolkit & FuzzyJess. [Online] 5. September 2006. [Zitat vom: 23. Februar 2009.] http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit2.html.

20. **Bontempi, G.; Birattari, M.:** MATLAB SOFTWARE TOOL FOR NEURO-FUZZY IDENTIFICATION AND DATA ANALYSIS. [Online] 5. April 1999. [Zitat vom: 23. Februar 2009.] http://www.ulb.ac.be/di/map/gbonte/software/Local/FIS.html.

21. **Mendel, J. M.:** Uncertain Rule-Based Fuzzy Logic Systems. *Introduction and New Directions.* [Online] [Zitat vom: 23. Februar 2009.] http://sipi.usc.edu/~mendel/book/.

22. **IMSE-CNM:** Xfuzzy 3.0. [Online] [Zitat vom: 23. Februar 2009.] http://www.imse.cnm.es/Xfuzzy/Xfuzzy_3.0/index.html.

23. **ZDnet.com:** Fuzzy Sets for Ada 4. [Online] December 2005. [Zitat vom: 07. July 2009.] http://downloads.zdnet.com/abstract.aspx?docid=727799.

24. **R project:** R Project. [Online] [Zitat vom: 06. July 2009.] http://www.r-project.org/.

25. **Aklan, S., et al.:** CRAN Package Site. *fuzzyOP.* [Online] [Zitat vom: 07. July 2009.] http://cran.r-project.org/web/packages/.

26. **Munk, K.:** Regressionsrechnung mit unscharfen Daten. *Diplomarbeit.* Wien, Österreich : Institut für Statistik und Wahrscheinlichkeitsrechnung, TU Wien, 1998.

27. **Weiss, M. A.:** *Data structures & algorithm analysis in C++, 2nd edition.* s.l. : Addison-Wesley, 1999.

28. **Clarkson:** Bell-Labs. *CIS677, Notes for lecture 3.* [Online] Dezember 1997. [Zitat vom: 10. Februar 2009.] http://cm.bell-labs.com/who/clarkson/cis677/lecture/3.

29. **Process Control and Robotics Unit, Univ. Oldenburg, Deutschland:** Fool & Fox. *Fuzzy Organizer Oldenburg.* [Online] 19. März 2002. [Zitat vom: 11. Februar 2009.] http://sourceforge.net/projects/fool.

30. **Viertl, R.:** *Einführung in die Stochastik. Mit Elementen der Bayes-Statistik und der Analyse unscharfer Information, 3. Auflage.* Wien : Springer, 2003.

31. **Dubois, D.; Prade, H.:** *Fundamentals of Fuzzy Sets.* Boston : Kluwer, 2000.

32. **Klement, E.; Mesiar, R.; Pap, E.:** *Triangular Norms.* Dordrecht, Boston, London : Kluwer Acad. Publ., 2000.

33. **Bandemer, H.; Gottwald, S.:** *Einführung in Fuzzy-Methoden, 4. Auflage.* Berlin : Akademie Verlag, 1993.

34. **Krätschmer, V.:** Some complete metrics on spaces of fuzzy subsets. *ACM.* Volume 130 Issue 3, 2002, pp. 357-365.

35. **Viertl, R.; Hareter, D.:** *Generalized Bayes' theorem for non-precise a-priori distribution. Metrika, Vol.59, No. 3,* Wien : Springer, 2004.

36. **Besset, D. H.:** *Object-Oriented Implementation of Numerical Methods.* San Diego, CA, USA : Morgan Kaufmann Publishers, 2001. ISBN 1-55860-679-3.

37. **Hoos, H. H.; Stützle, T.:** *Stochastic Local Search, Foundations and Applications.* San Francisco, CA, USA : Morgan Kaufmann Publishers, 2005. ISBN 1-55860-872-9.

38. **Knuth, D. E.:** *The Art of Computer Programming, 3rd edition.* s.l. : Addison Wesley Longman, 1998. ISBN 0-201-89684-2.

39. **Mitzenmacher, M.; Upfal, E.:** *Probability and Computing.* s.l. : Cambridge University Press, 2005. ISBN 0-521-83540-2.

40. **Sedgewick, R.:** *Algorithmen, 2. Auflage.* Reading, MA, USA : Addison-Wesley Deutschland, 1997. ISBN 3-89319-402-9.

41. **Weiss, M. A.:** *Data Structures & Algorithms, 2nd edition.* s.l. : Addison-Wesley Longman Inc., 1999. ISBN 0-201-36122-1.

42. **Hirsch, E.:** Fuzzy Visualisation [Zitat vom: 01. September 2009] http://www.auto.tuwien.ac.at/~edi/fuzzyvis/index.html

43. **Robertson, H. P.** The Uncertainty Principle. *Physical Review.* 34, 1929, Bd. 1.

8 Web-Referenzen

#	URL	Beschreibung	Zugriff
1	http://www.finanz-lexikon.net/Modellunsicherheit_2155.html	Merk: Finanz Lexikon: Modellunsicherheit	02. Feb 2009
17	http://www.dbai.tuwien.ac.at/proj/StarFLIP/	TU Wien DBAI: StarFLIP	11. Feb 2009
18	http://www.rhaug.de/fool/	Fool & Fox Homepage	11. Feb 2009
19	http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit2.html	FuzzyJ Toolkit	23. Feb 2009
20	http://www.ulb.ac.be/di/map/gbonte/software/Local/FIS.html	Matlab software tool for neuro-fuzzy identification and data analysis	23. Feb 2009
21	http://sipi.usc.edu/~mendel/book/	Uncertain Rule-Based Fuzzy Logic Systems	23. Feb 2009
22	http://www.imse.cnm.es/Xfuzzy/Xfuzzy_3.0/index.html	XFuzzy 3.0	23. Feb 2009
23	http://downloads.zdnet.com/abstract.aspx?docid=727799	Fuzzy sets for Ada 4	07. Jul 2009
24	http://www.r-project.org/	R Project	06. Jul 2009
25	http://cran.r-project.org/web/packages	CRAN FuzzyOP	07. Jul 2009
28	http://cm.bell-labs.com/who/clarkson/cis677/lecture/3	Bell Labs CIS677 Notes for lecture 3	10. Feb 2009
29	http://sourceforge.net/projects/fool	Fool & Fox Sourceforge Project Site	11. Feb 2009
30	http://www.auto.tuwien.ac.at/~edi/fuzzyvis/indei.html	Fuzzy Visualisation Homepage	01. Sep 2009