Regler zur Robotersteuerung

Studienarbeit im Fach Informatik

vorgelegt
von

Michael Holzheu

geboren am [redacted]

Angefertigt am

Lehrstuhl für Mustererkennung (Informatik 5)
Institut für Mathematische Maschinen und Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

[redacted]

Beginn der Arbeit: 01.09.1995

Abgabe der Arbeit: 26.03.1996

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien des Lehrstuhls für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts in Punkt 2.3.

Erlangen, den ....

Übersicht

In der vorliegenden Arbeit werden unterschiedliche aus der Regelungstechnik bekannte Regler zur Steuerung von Kameras in einem Objektverfolgungssytem untersucht. Es werden hierzu eine auf der Hand eines Roboters montierte Kamera sowie eine Schwenk-/Kipp-Kamera verwendet. Zielsetzung dieser Studienarbeit ist es, eine Klassenhierarchie für die verschiedenen Reglerklassen zu entwerfen und in der Programmiersprache C++ zu implementieren. Dabei wird ausführlich auf die zugrundeliegende Theorie der Regelungstechnik eingegangen. Es werden sowohl die klassischen linearen Regler wie P,PI,PID usw. als auch Fuzzy-Regler behandelt. Die verschiedenen Ansätze werden durch theoretische Berechnungen und in realen Experimenten auf ihre Eignung für die Objektverfolgung untersucht. Dabei werden sie in Bezug auf Regelgüte, Speicherbedarf und Laufzeitverhalten verglichen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Mustererkennung und Regelungstechnik sind zwei wissenschaftliche Disziplinen aus dem Bereich der Informatik und des Ingenieurwesens. Meist haben ihre Anwendungsgebiete nur wenig miteinander zu tun. In dem Teilbereich der Objektverfolgung jedoch treffen diese beiden Wissenschaften aufeinander. Die Zielsetzung hierbei ist es, ein sich fortbewegendes Objekt mit Hilfe einer beweglichen Kamera zu verfolgen. Dabei besteht die Aufgabe der Mustererkennung darin, das Objekt zu erfassen und seine Position, Geschwindigkeit usw. zu bestimmen. Diese Informationen verwendet der regelungstechnische Teil, um mit Hilfe von Reglern die Kamera so nachzuführen, daß das Objekt eine gewünschte Position im Kamerabild einnimmt.

Es gibt eine Vielzahl potentiell möglicher Anwendungen der Objektverfolgung. Eine davon wäre z.B. ein Bildtelefonsystem, bei dem sich die kommunizierenden Menschen weitgehend frei im Raum bewegen können, und eine Kamera das Gesicht des jeweiligen Gesprächspartners im Zentrum des Bildes hält. Auch für überwachungstechnische Aufgaben, z.B. der nächtlichen Sicherung von Grundstücken, könnten Kamerasysteme mit Objektverfolgung sinnvoll sein.

Auf Seiten der Regelungstechnik wurde in den letzten Jahren Objektverfolgung vor allem im Bereich der Robotik verstärkt eingesetzt (siehe [PKK93], [WSN87], [FLM91]). Dabei fungiert die Kamera als visueller Sensor, der fortlaufend die Position eines Werkstücks erfassen kann und diese an Regler weiterleitet, die den Roboter dann den verschiedenen Aufgabenstellungen entsprechend steuern. Dabei entsteht ein rückgekoppeltes System, bei dem die Objektposition die Roboterbewegung, und diese wiederum die relative Position des Objekts im Kamerabild beeinflußt. Es entfällt der Zwang, in hochstrukturierten Umgebungen arbeiten zu müssen, in denen die Bewegungen des Roboters mit den Objektbewegungen schon im voraus exakt abgestimmt werden müssen [FL90]. Der Roboter ist somit durch die visuelle Information in der Lage, sich einer veränderlichen Umwelt besser anzupassen.

In der Literatur finden sich verschiedene Arbeiten, die sich mit dem Einsatz von Reg-

lern in Verbindung mit Objektverfolgung befassen: Einer der ersten Ansätze stammt von Coulon und Nougaret [CN83]. Hierbei werden eindimensionale Objektbewegungen betrachtet, wobei für die Regelung ein PI–Regler verwendet wird. Pananikolopoulos et al. [PKK93] setzen für die Verfolgung von Objekten, die sich mit unbekannter Geschwindigkeit in einer Ebene bewegen, PI– und LQR–Regler mit Kalmanfilter ein und vergleichen die einzelnen Ansätze durch Simulationen und Experimente miteinander. In [WSN87] wird der Einsatz von adaptiven Reglern in Kombination mit visueller Rückkopplung zur Verbesserung der Positioniergenauigkeit von Robotern und zum Ausgleichen von Modellungenauigkeiten untersucht. Dabei wird vor allem der Geschwindigkeitsvorteil der dynamischen Rückkopplung gegenüber statischen Systemen herausgestellt, bei welchen die Bestimmung der Objektposition und die Roboterbewegung unabhänging voneinander und in sequentieller Reihenfolge stattfinden. In [BCS92] wird für die Steuerung einer Schwenk–/Kipp–Kamera für jeden Freiheitsgrad einzeln ein PID–Regler sowie ein α–β–γ–Filter verwendet. Getestet wird das System anhand von kreisförmigen Objektbewegungen in der Ebene. Danilidis et al. schließlich verwenden zur Objektverfolgung einen TRC–Stereo–Kopf und setzen für die Regelung einen Zustandsregler (LQR) mit Kalmanfilter zur Zustandsschätzung und Prädiktion ein [DHKS95].

Auch am Lehrstuhl für Mustererkennung (LME) an der Friedrich–Alexander–Universität in Erlangen wurde ein System zur Echtzeitobjektverfolgung implementiert [DN95]. Es arbeitet mit aktiven Konturen. In der Testumgebung fährt eine Modelleisenbahn auf einem Rundkurs (siehe Bild 1.1) und kann wahlweise von einer Canon–Kamera oder einer auf der Hand eines Roboters befestigten Kamera verfolgt werden. Dabei arbeiten die Objektverfolgung und die Kamerasteuerung parallel auf einem Workstation–Cluster, wobei die Kommunikation der einzelnen Module über `pvm` [GBD*93] erfolgt. Um das Objekt im Bildmittelpunkt zu halten, bekommt der Regler von der Objektverfolgung jeweils die Position des Schwerpunkts der Eisenbahn im aktuellen Kamerabild übermittelt. Dabei wurde für die Ansteuerung der Kameras bisher ein einfacher linearer P–Regler verwendet.

Ziel dieser Arbeit ist es, in einer Klassenhierarchie unter der Programmiersprache C++ verschiedene aus der Regelungstechnik bekannte Regler zu realisieren und miteinander zu vergleichen. Es werden dabei die linearen Standardregler P, PI, PID usw. sowie Ansätze aus der Fuzzy–Logik implementiert und am bestehenden System getestet.

Zunächst soll dem Leser eine kurze Einführung in die elementaren Grundlagen der Regelungstechnik gegeben werden (Kapitel 2). Anschließend folgen zwei Kapitel, die auf die Besonderheiten der linearen (Kapitel 3) und der Fuzzy–Regler (Kapitel 4) eingehen. Kapitel 5 befaßt sich mit der Implementierung der Regler in C++ unter dem am LME existierenden Bildverarbeitungssystem HIPPOS. Dabei werden die verschiedenen Klassenhierarchien, sowie die wichtigsten Implementierungdetails vorgestellt. Im darauffolgenden

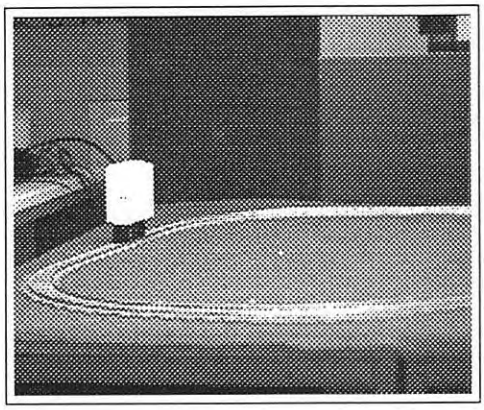


Bild 1.1: Rundkurs der Eisenbahn

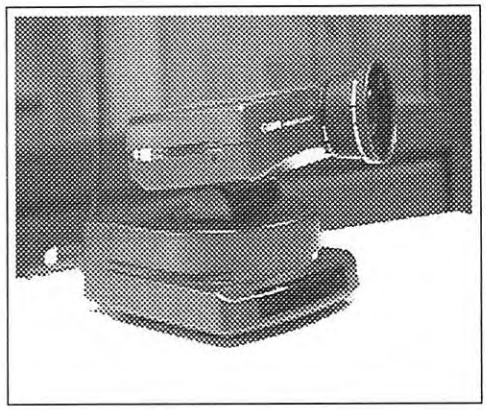


Bild 1.2: Verwendete Canon–Kamera

Kapitel werden dann die Ergebnisse der verschiedenen Ansätze zusammengefaßt, wobei sowohl simulierte als auch real mit der Canon–Kamera bzw. der Roboter–Kamera gemessene Ergebnisse vorgestellt werden. Abschließend folgt ein kurzer Ausblick sowie eine Zusammenfassung der Arbeit.

Kapitel 2

Grundlagen der Regelungstechnik

In diesem Kapitel werden grundlegende Konzepte und Definitionen der Regelungstechnik vorgestellt, die für das Verständnis der weiteren Ausführungen benötigt werden. Hierbei wird besonders auf [Unb87] und [Leo92] Bezug genommen.

2.1 Struktur eines Regelkreises

Ziel der Regelungstechnik ist es, Werkzeuge zur Verfügung zu stellen, um Prozesse automatisch regulieren zu können, die zuvor der Mensch von Hand geregelt hat, oder die zuvor überhaupt nicht regelbar waren. Dabei beruht das Prinzip der Regelung darauf, daß in einem geschlossenen Wirkungsablauf den Störungen, die auf ein dynamisches System Einfluß haben, entgegengewirkt wird. Man spricht hier von einer sog. negativen Rückkopplung. Die geschlossene Wirkungskette wird als *Regelkreis* bezeichnet.

In [Unb87] ist ein dynamisches System definiert als Funktionseinheit zur Verarbeitung von Signalen, wobei die Systemeingangsgrößen als Ursache und die Systemausgangsgrößen als deren zeitliche Auswirkung in Relation zueinander gebracht werden. Ein dynamisches System besitzt also eine gewisse Anzahl von Eingangsgrößen, die zeitlich versetzt Auswirkung auf die Ausgänge des Systems haben. Man unterscheidet zwischen einfachen Eingrößensystemen mit jeweils nur einem Ein- und Ausgang und komplexen Mehrgrößensystemen, die mehrere Ein- und Ausgänge besitzen können. Dynamische Systeme können auch als *Übertragungsglieder* oder *Übertragungssysteme* bezeichnet werden. Jedes Übertragungsglied besitzt mindestens einen Eingang und einen Ausgang, sowie eine eindeutige Wirkungsrichtung. Übertragungsglieder werden meist durch Blockschaltbilder visualisiert. In Bild 2.1 ist der schematische Aufbau eines Regelkreises im Blockschaltbild dargestellt. Die vier Hauptelemente eines Regelkreises bilden Regler, Stellglied, Regelstrecke sowie das Meßglied.

Die Signale zwischen den Gliedern werden durch die international üblichen Buchstaben [Unb87] bezeichnet:

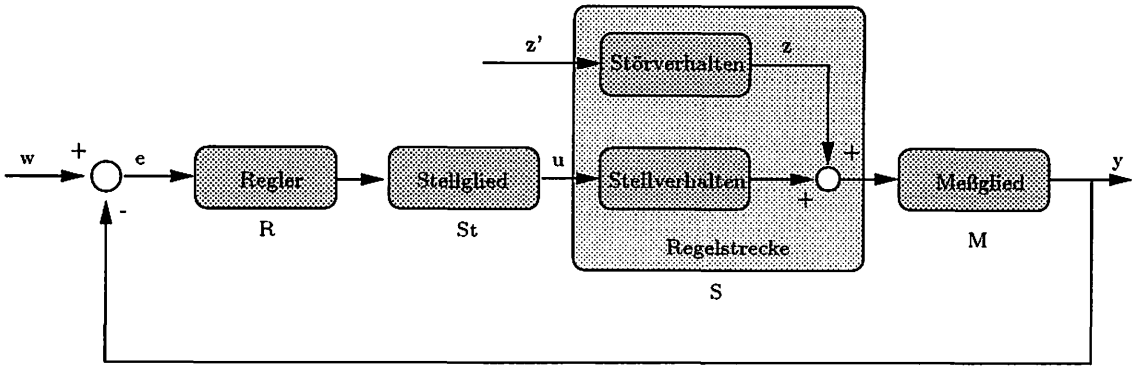


Bild 2.1: Blockschaltbild eines Regelkreises

- y: Regelgröße
- w: Führungsgröße
- e: Regelabweichung
- u: Stellgröße
- z: Störgröße

Die Regelgröße y beschreibt die gemessenen Istwerte der Regelstrecke, d.h. den aktuellen Zustand. Die Führungsgröße w stellt den Wunschzustand dar, also den Sollwert, den die Regelstrecke erreichen muß (z.B. eine bestimmte Umdrehungszahl eines Motors). Die Regelabweichung e wird durch die Differenz zwischen der Führungsgröße w und der Regelgröße y berechnet. Sie bildet den Eingangswert für den Regler R. Dieser erzeugt daraus mit Hilfe des Stellgliedes St die Stellgröße u, die den Störgrößen z entgegenwirken soll. Durch das Meßglied M werden die aktuellen Werte der Regelstrecke gewonnen. Die Aufgabe des Reglers besteht nun darin, die Regelabweichung möglichst schnell auf Null zu zwingen oder zumindest sehr klein zu halten.

Man unterscheidet grundsätzlich zwischen zwei Regelungsarten: Bei der *Festwertregelung* (auch *Störgrößenregelung* genannt) ist ein konstanter Sollwert vorgegeben, und Ziel der Regelung ist es, Störeinflüsse, die auf den Prozeß wirken, zu kompensieren und den vorgegebenen Sollwert möglichst gut zu erreichen. Die zweite Regelungsart stellt die sog. *Folgeregelung* oder *Nachlaufregelung* dar. Hier ist der Sollwert variabel und die Regelgrößen des Prozesses müssen den sich ändernden Sollwerten möglichst gut nachgeführt werden. Der Sollwert wird hierbei auch als *Führungsgröße* bezeichnet.

2.2 Statisches und dynamisches Verhalten

Bei Systemen wird zwischen ihrem statischen und dynamischen Verhalten unterschieden. Das dynamische Verhalten beschreibt den zeitlichen Verlauf der Ausgangsgröße $x_a(t)$ des Systems bei einer gegebenen Eingangsgröße $x_e(t)$. Häufig wird bei der Betrachtung des dynamischen Verhaltens eine sprungförmige Veränderung der Eingangsgröße angenommen (Bild 2.2 nach [Unb87]).

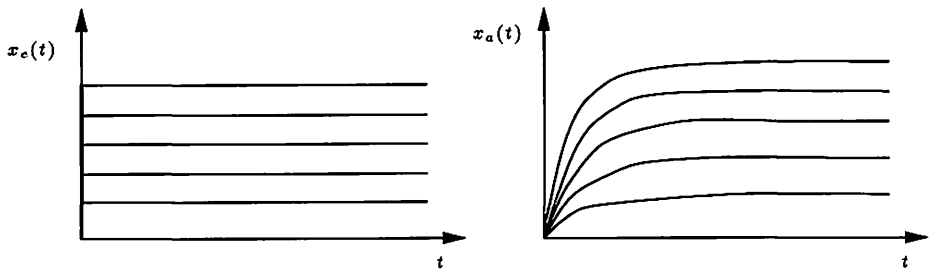


Bild 2.2: dynamisches Verhalten eines Systems

Betrachtet man das statische Verhalten eines Systems, so wird die Ausgangsgröße $x_a(t)$ bei einem bestimmten Eingangssignal $x_e(t)$ für $t \to \infty$ ermittelt. Verwendet man nun als Eingangssignal z.B. die Sprungfunktion mit unterschiedlicher Sprunghöhe und trägt jeweils $x_{a,s} = x_a(\infty)$ gegen die Sprunghöhe $x_{e,s}$ auf, so ergibt sich eine Kennlinie, die das statische Verhalten des Systems beschreibt (Bild 2.3). Durch die statische Kennlinie wird das Ruheverhalten des Systems charakterisiert.

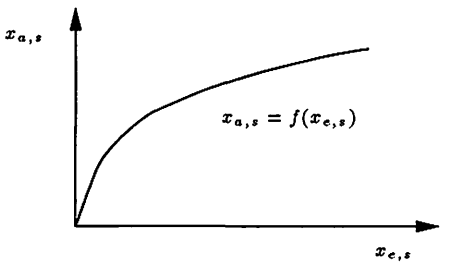


Bild 2.3: statisches Verhalten eines Systems

2.3 Kontinuierliche und diskrete Systeme

In der klassischen Regelungstechnik wird meist mit kontinuierlichen Signalen gearbeitet. Dies bedeutet, daß das Signal zu jedem beliebigen Zeitpunkt gegeben ist und in gewissen Grenzen eine stetige Amplitude besitzt. Man spricht dann von einem *kontinuierlichen* Signalverlauf. Setzt man nun Digitalrechner für Regelungssysteme ein, so muß das Signal diskret abgetastet werden. Es ergibt sich dann ein Signalverlauf, der sowohl nach der

Amplitude als auch nach der Zeit quantisiert ist. Solche Regelungssysteme werden als *Abtastsysteme* bezeichnet. In dieser Arbeit wird bei der Realisierung der Regler auf derartige Systeme eingegangen, da die Implementierung auf einem Digitalrechner erfolgen soll.

Kapitel 3

Lineare Regler

Im folgenden sollen die für die Theorie linearer Systeme wichtigen Aspekte kurz zusammengefaßt werden. Anschließend werden die einzelnen Reglertypen beschrieben.

3.1 Lineare Systeme

Die klassische Regelungstechnik beruht weitgehend auf der Theorie von linearen Systemen. Dieses Gebiet ist mittlerweile sehr gründlich erforscht, und es existiert eine Vielzahl von Methoden zur Behandlung dieser Systeme.

Das dynamische Verhalten eines linearen Systems kann durch einen Operator T dargestellt werden, der den Zusammenhang zwischen Eingangsgrößen u und Ausgangsgrößen y festgelegt. Das System wird genau dann als linear bezeichnet, wenn unter Verwendung des Operators T das sog. *Superpositionsprinzip* gilt [Unb87]:

$$\begin{aligned} y(t) &= T[u(t)] \\ \sum_{i=1}^{n} k_i y_i(t) &= T\left[\sum_{i=1}^{n} k_i u_i(t)\right] \end{aligned} \tag{3.1}$$

3.1.1 Grundlegende Definitionen

Hier werden nun einige zum Verständnis der folgenden Punkte wichtigen grundlegende Definitionen festgelegt.

Definition 1 *Die Sprungfunktion s(t) ist definiert durch*

$$s(t) = \begin{cases} 1 & für \quad t > 0 \\ \frac{1}{2} & für \quad t = 0 \\ 0 & für \quad t < 0. \end{cases}$$

Definition 2 *Der Einheitsimpuls (oder Dirac–Impuls) $\delta(t)$ ist die Ableitung der Sprungfunktion $s(t)$ und ist definiert durch*

$$\delta(t) = \begin{cases} \infty & für \quad t = 0 \\ 0 & für \quad t \neq 0. \end{cases}$$

3.1.2 Die Laplace–Transformation

Die Laplace–Transformation ist eine eindeutig umkehrbare Transformation von einer Klasse von Originalfunktionen $f(t)$ nach einer Klasse von Bildfunktionen $F(s)$. Sie ist folgendermaßen definiert [Unb87]:

$$F(s) = \int_0^\infty f(t)e^{-st}dt = \mathcal{L}\{f(t)\} \tag{3.2}$$

Dabei ist s eine komplexe Größe. Laplace–transformierte Funktionen werden im allgemeinen mit Großbuchstaben bezeichnet, ihre Originalfunktionen mit Kleinbuchstaben. In dieser Arbeit wird insbesondere der Endwertsatz der Laplace–Transformation verwendet. Will man den Grenzwert einer Funktion $f(t)$ für $t \to \infty$ bestimmen, so gilt:

$$f(\infty) = \lim_{t\to\infty} f(t) = \lim_{s\to 0} sF(s). \tag{3.3}$$

Dieser Satz gilt nur unter der Voraussetzung, daß $f(t)$ für $t \to \infty$ auch wirklich konvergiert. Hat man also die Beschreibung eines linearen Systems in Form einer s–Übertragungsfunktion gegeben, so ist es sehr einfach, durch Anwendung des Endwertsatzes, sein statisches Verhalten (Kapitel 2.2) zu bestimmen.

3.1.3 Beschreibungsmöglichkeiten linearer Systeme

Man kann lineare Systeme auf unterschiedliche Weise beschreiben. Durch jede der folgenden Beschreibungsmöglichkeiten ist ein lineares System vollständig bestimmt:

- Übergangsfunktion (Sprungantwort) $h(t)$
- Gewichtsfunktion (Impulsantwort) $g(t)$
- s–Übertragungsfunktion $G(s)$
- z–Übertragungsfunktion $G(z)$

Im Zeitbereich kann ein lineares System u.a. durch die *Übergangsfunktion* oder die *Gewichtsfunktion* dargestellt werden. Die Übergangsfunktion $h(t)$ ist die auf die Sprunghöhe $\hat{u}$ bezogene Reaktion des Systems auf die Sprungfunktion $s(t)$:

$$\begin{aligned} u(t) &= \hat{u}s(t), \qquad \hat{u} = const. \\ h(t) &= \frac{1}{\hat{u}}y(t) \end{aligned} \tag{3.4}$$

Dabei ist $y(t)$ die nichtnormierte Sprungantwort des Systems. Durch Differenzieren der Übergangsfunktion erhält man die Gewichtsfunktion $g(t)$. Diese ist die Reaktion des linearen Systems auf einen Einheitsimpuls $\delta(t)$.

Im Frequenzbereich werden lineare Übertragungsglieder im allgemeinen durch ihre *Übertragungsfunktionen* beschrieben. Im kontinuierlichen Fall verwendet man dazu die sog. s–Übertragungsfunktion $G(s)$, welche die Laplace–transformierte der Gewichtsfunktion $g(t)$ ist (siehe [Unb87] und Kapitel 3.1.2). Man kann die s–Übertragungsfunktion auch durch das Verhältnis des Laplace–transformierten Ausgangssignals $y(t)$ und des ebenfalls Laplace–transformierten Eingangssignals $u(t)$ definieren:

$$G(s) = \frac{Y(s)}{U(s)} \tag{3.5}$$

Im diskreten Fall, also bei Abtastsystemen, wird die z–Übertragungsfunktion G(z) verwendet. Diese ist definiert durch den Quotienten der z–transformierten Größen $y(t)$ und $u(t)$. Für Einzelheiten der z–Transformation sei hier auf [Ise88] verwiesen.

$$G(z) = \frac{y(z)}{u(z)} \tag{3.6}$$

Bei den weiteren Ausführungen wird die z–Transformation für die Implementierung der linearen Regler benötigt. Die Laplace–Transformation wird in Kapitel 6 dazu verwendet, um die Regelstrecke zu beschreiben und die linearen Regler dafür zu entwerfen.

3.1.4 Stabilität

Jeder geschlossene Regelkreis sollte die Minimalanforderung erfüllen, daß er stabil ist. Um Aussagen über die Stabilität eines linearen Systems zu machen, werden meist die Pole der s–Übertragungsfunktion, also die Nullstellen ihres Nenners, betrachtet. Ein System wird als (asymptotisch) stabil bezeichnet, wenn seine Gewichtsfunktion $g(t)$ für $t \to \infty$ gegen Null strebt.

$$\lim_{t \to \infty} g(t) = 0 \tag{3.7}$$

Dies ist gerade dann der Fall, wenn alle Pole der s–Übertragungsfunktion in der linken komplexen s–Halbebene liegen [Unb87]. Liegt ein Pol in der rechten Halbebene, oder es existiert ein mehrfacher Pol (Vielfachheit ≥ 2) auf der imaginären Achse, so ist das

System instabil. Das bedeutet, daß die Gewichtsfunktion betragsmäßig für $t \rightarrow \infty$ über alle Maßen wächst. Als Spezialfall gibt es noch die Möglichkeit der Grenzstabilität. Diese ist dann gegeben, wenn kein Pol in der rechten s–Halbebene liegt, und kein mehrfacher Pol auf der Imaginär–Achse auftritt, aber mindestens ein einfacher Pol auf der imaginären Achse vorliegt. Für die Gewichtsfunktion heißt das, daß sie einen endlichen Wert nicht über– bzw. unterschreitet.

3.1.5 Zustandsgrößendarstellung

In Kapitel 3.1.3 wurde die Übertragungsfunktion als Beschreibungsmöglichkeit eines linearen Systems vorgestellt. In der modernen Systemtheorie und bei der Beschreibung von Mehrgrößensystemen wird oft statt der Übertragungsfunktion, die nur das Eingangs–/Ausgangsverhalten eines Systems beschreibt, die Zustandsgrößendarstellung verwendet. Dabei werden die internen Wirkungszusammenhänge durch Zustandsgrößen beschrieben. Diese Darstellungsform bringt den Vorteil, daß sich das System schon am Anfang in einem definierten Startzustand befinden kann. Außerdem können relativ einfach vektorielle Größen verarbeitet werden, wie es bei Mehrgrößensystemen erforderlich ist. Im diskreten Fall wird ein lineares System durch eine Zustands–Differenzengleichung und eine Ausgangsgleichung charakterisiert.

$$\boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \tag{3.8}$$

$$\boldsymbol{y}(k) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \tag{3.9}$$

Mit Hilfe der Zustands–Differenzengleichung (3.8) werden die internen Zustände rekursiv durch die Zustände im vorhergehenden Zeitschritt sowie durch die Eingabegrößen $\boldsymbol{u}$ bestimmt. Dabei werden die Zustände $\boldsymbol{x}(k)$ mit der Systemmatrix $\boldsymbol{A}$ und die Eingabegrößen mit der Steuermatrix $\boldsymbol{B}$ multipliziert. Die Ausgangsgleichung (3.9) beschreibt den Zusammenhang der Zustände $\boldsymbol{x}(k)$ mit der Ausgangsgröße $\boldsymbol{y}$. Der Einfluß von $\boldsymbol{u}$ auf $\boldsymbol{y}$ wird durch die Matrix $\boldsymbol{D}$ festgelegt. $\boldsymbol{D}$ ist immer dann Null, wenn $\boldsymbol{u}$ nicht direkt auf den Ausgang durchgeschaltet ist. Die Ausgangsmatrix $\boldsymbol{C}$ gibt an, wie die Zustände des Systems auf den Ausgang wirken.

3.2 Die linearen E/A–Regler

Die Eingangs–/Ausgangs–Regler sind dadurch charakterisiert, daß die Ausgabegrößen der Regler nur durch ihre Eingabewerte sowie durch die Parameter der Übertragungsfunktion, bestimmt sind. Die allgemeine Übertragungsfunktion dieser Regler lautet folgendermaßen

[Ise88]:

$$G_R(z) = \frac{u(z)}{e(z)} = \frac{Q(z^{-1})}{P(z^{-1})} = \frac{q_0 + q_1 z^{-1} + \ldots + q_\nu z^{-\nu}}{1 + p_1 z^{-1} + \ldots + p_\mu z^{-\mu}} \tag{3.10}$$

Dabei steht e für die Regelabweichung, die als Eingabegröße für den Regler dient. Die Stellgröße u ist die Ausgangsgröße des Reglers. Aus (3.10) ergibt sich mit Hilfe des Rechtsverschiebungssatzes der z–Transformation (siehe [Ise88]) die dazugehörige Differenzengleichung des linearen Systems:

$$u(k) + p_1 u(k-1) + \ldots + p_\mu (k-\mu) = q_0 e(k) + q_1 e(k-1) + \ldots + q_\nu e(k-\nu) \tag{3.11}$$

Durch Auflösen auf $u(k)$ erhält man nun auf einfache Weise den für die Implementierung des Reglers nötigen rekursiven Regelalgorithmus:

$$u(k) = -p_1 u(k-1) - p_2 u(k-2) - \ldots - p_\mu u(k-\mu) + q_0 e(k) + q_1 e(k-1) + \ldots + q_\nu e(k-\nu) \tag{3.12}$$

3.2.1 Diskreter PID–Regler

Ein Standardregler der klassischen analogen Regelungstechnik ist der PID–Regler. Man hat für die Handhabung dieses Reglertyps viele Verfahren und Methoden entwickelt. Um dies auch für digitale Regelungssysteme einsetzen zu können, wurde der PID–Regler für diskrete Abtastsysteme umgesetzt.

Die idealisierte Gleichung eines PID–Reglers ist durch

$$u(t) = K \left[e(t) + \frac{1}{T_I} \int_0^t e(\tau) d\tau + T_D \frac{de(t)}{dt} \right] \tag{3.13}$$

bestimmt [Ise88]. Dabei werden die einzelnen Parameter folgendermaßen bezeichnet:

- K Verstärkungsfaktor
- T_I Integrierzeit (Nachstellzeit)
- T_D Differenzierzeit (Vorhaltezeit)
- $e(t)$ Regelabweichung zum Zeitpunkt t

Hat man nun sehr kleine Abtastzeiten T_0, so kann diese Gleichung direkt in eine Differenzengleichung umgesetzt werden. Man ersetzt hier einfach den Differentialquotienten durch eine Differenz und das Integral durch eine Summe. Bei Annäherung der kontinuierlichen Integration durch Rechteckintegration ergibt sich folgende Berechnungsvorschrift für den PID–Regler:

$$u(k) = K \left[e(k) + \frac{T_0}{T_I} \sum_{i=0}^{k-1} e(i) + \frac{T_D}{T_0} (e(k) - e(k-1)) \right] \tag{3.14}$$

Bildet man die Differenz von $u(k)$ und $u(k-1)$, so erhält man aus (3.14) den PID–Regelalgorithmus in rekursiver Form:

$$u(k) = u(k-1) + q_0 e(k) + q_1 e(k-1) + q_2 e(k-2) \tag{3.15}$$

mit

$$\begin{aligned} q_0 &= K\left(1 + \tfrac{T_D}{T_0}\right) \\ q_1 &= -K\left(1 + 2\tfrac{T_D}{T_0} - \tfrac{T_0}{T_I}\right) \\ q_2 &= K\tfrac{T_D}{T_0} \end{aligned}$$

Auf das gleiche Ergebnis kommt man auch unter Verwendung der z–Übertragungsfunktion des PID–Reglers (Formel 3.16), wenn man nach der Methode, die am Anfang von Kapitel 3.2 beschrieben ist, vorgeht.

$$G_{PID}(z) = \frac{q_0 + q_1 z^{-1} + q_2 z^{-2}}{1 - z^{-1}} \tag{3.16}$$

Der oben vorgestellte PID–Regler kann auch durch parallelgeschaltete Übertragungsglieder P,I und D dargestellt werden (siehe Bild 3.1). Dazu muß die z–Übertragungsfunktion (Formel 3.16) nur entsprechend umgeformt werden [Ise88], um eine Summe der einzelnen Glieder zu erhalten:

$$G_{PID}(z) = K\left[1 + \frac{T_0}{T_I}\frac{z^{-1}}{1 - z^{-1}} + \frac{T_D}{T_0}(1 - z^{-1})\right] \tag{3.17}$$

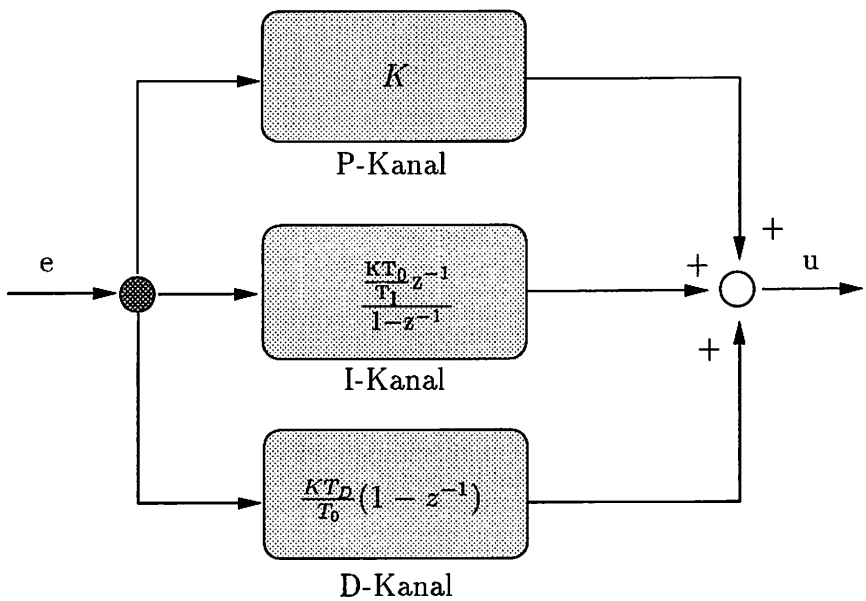


Bild 3.1: Blockschaltbild des PID–Regelalgorithmus

Durch Wahl von K, T_I und T_D können zum Einstellen des diskreten Reglers weitge-

hend die herkömmlichen Verfahren für die analogen Regler verwendet werden, sofern die Abtastzeiten klein genug sind.

3.2.2 P–Glied

Durch das P–Glied (Proportional–Glied) wird ein rein proportionaler Zusammenhang zwischen der Ein– und Ausgangsgröße beschrieben. Die z–Übertragungsfunktion des P–Gliedes lautet folgendermaßen:

$$G_P(z) = K \tag{3.18}$$

Für den dazugehörigen Regelalgorithmus ergibt sich dann:

$$u_P(k) = Ke(k) \tag{3.19}$$

Dabei ist K eine der Regelstrecke anzupassende Konstante. Sie wird als Übertragungsbeiwert oder als *Verstärkungsfaktor* bezeichnet. Der P–Regelalgorithmus ist nicht rekursiv, da nur die momentane Regelabweichung berücksichtigt wird. Die s–Übertragungsfunktion $G_P(s)$ des P–Gliedes entspricht der z–Übertragungsfunktion:

$$G_P(s) = K \tag{3.20}$$

3.2.3 I–Glied

Das I–Glied (Integral–Glied) führt eine Integration der Eingangsgröße durch. Daraus ergibt sich im Diskreten eine Summation der Regelabweichungen über die Zeit. Die z–Übertragungsfunktion dieses Übertragungsgliedes lautet:

$$G_I(z) = \frac{KT_0z^{-1}}{T_I(1-z^{-1})} \tag{3.21}$$

Daraus ergibt sich wie in Kapitel 3.2 der Regelalgorithmus für das Integral–Glied:

$$u(k) = u(k-1) + K\frac{T_0}{T_I}e(k-1) \tag{3.22}$$

Hierbei sind T_0 und T_I Zeitkonstanten. T_0 stellt die Abtastzeit des Reglers dar, T_I wird als Integrierzeit bzw. *Nachstellzeit* bezeichnet. Die Sprungantwort des Reglers steigt nach einer Zeit T_I, bzw. nach $\frac{T_I}{T_0}$ Abtastschritten um den Wert K. Der Wert der Stellgröße des I–Reglers entspricht also nach einer Zeit T_I der des P–Reglers. Die s–Übertragungsfunktion des I–Gliedes ist definiert durch

$$G_I(s) = \frac{1}{T_Is}. \tag{3.23}$$

Will man das diskrete Glied wie das analoge einstellen, so muß für K der Wert 1 gewählt werden.

3.2.4 D–Glied

Das D–Glied (differenzierendes Glied) führt eine Differentiation der Eingangsgröße des Reglers durch. Die z–Übertragungsfunktion dieses Übertragungsgliedes lautet folgendermaßen:

$$G_D(z) = \frac{KT_D}{T_0}(1 - z^{-1}) \tag{3.24}$$

Analog zu den P– und I–Gliedern erhält man den Regelalgorithmus:

$$u(k) = \frac{KT_D}{T_0}[e(k) - e(k-1)] \tag{3.25}$$

T_D wird als Differenzierzeit oder als *Vorhaltezeit* bezeichnet. Nimmt man ein rampenförmiges Eingangssignal $e(k) = k$ an, so erreicht die Stellgröße des P–Gliedes zur Zeit $kT_0 = T_D$ bzw. nach $k = \frac{T_D}{T_0}$ Abtastungen den gleichen Wert wie das D–Glied. Die s–Übertragungsfunktion des D–Gliedes ist

$$G_D(s) = sT_D. \tag{3.26}$$

Wie schon beim I–Glied gilt, daß, wenn T_D für das analoge Glied vorliegt, man beim diskreten Glied für K den Wert 1 wählen muß.

Kapitel 4

Fuzzy–Regler

Die Fuzzy–Theorie wurde bereits in den sechziger Jahren von Prof. Lofti A. Zadeh entwickelt [Pro95], erfuhr aber in den ersten 20 Jahren nur sehr wenig Beachtung. In den 90'er Jahren entwickelte sie sich dann, besonders infolge ihres verstärkten Einsatzes durch die Japaner [Alt95], zu einer anerkannten Technologie im Ingenieurwesen. Sie stellt eine Erweiterung der klassischen Mengentheorie dar und ermöglicht auch unter Verwendung von Mikro–Computern den Umgang mit vagen Aussagen, Ereignissen und Werten. Somit bietet sie eine praktikable Alternative zur binären Logik. Sie erlaubt es, mit unscharfen Begriffen wie z.B. „warm" und „kalt" umzugehen, die der Mensch täglich anwendet, welche aber kaum mit binärer Logik zu erfassen sind. Für die Fuzzy–Logik gibt es mittlerweile ein sehr breites Anwendungsspektrum. Fuzzy–Methoden werden z.B. in Expertensystemen, Robotik, Bilderkennung, Prozeßregelung und vielen anderen Gebieten erfolgreich eingesetzt. Besonders in der Regelungstechnik haben sich Fuzzy–Regler als Alternative zu den konventionellen Reglern (PID, usw.) als ein weiteres Werkzeug bereits etabliert. Ein großer Vorteil dieser neueren Entwicklung ist, daß für eine Regelstrecke keine komplizierten mathematischen Modelle entwickelt werden müssen, was die Entwicklungszeit und v.a. die Kosten erheblich senkt. Es können aber kaum Stabilitätsberechnungen für Fuzzy–Regler angestellt werden, was oftmals als großer Nachteil dieser Technik angesehen wird. Andererseits muß man jedoch bedenken, daß die mathematischen Modelle für komplizierte Regelstrecken manchmal mit der Realität nur noch wenig zu tun haben, und somit der Nachweis für die Stabilität eines Reglers in solchen Fällen zumindest zweifelhaft erscheint.

Im folgenden wird auf die zugrundeliegende Theorie eines Fuzzy–Reglers eingegangen. Dazu werden zunächst die nötigen Grundlagen der Fuzzy–Theorie kurz umrissen. Es werden dabei die Konzepte der Fuzzy–Mengen, linguistischen Variablen und der Fuzzy–Operatoren vorgestellt. Kapitel 4.2 erläutert den Aufbau und die allgemeine Arbeitsweise eines auf unscharfen Mengen basierenden Reglers.

4.1 Die Fuzzy–Theorie

Die Fuzzy–Mengen–Theorie ist eine Verallgemeinerung der klassischen (scharfen) Mengentheorie. Der zentrale Begriff ist hierbei der Zugehörigkeitsgrad von Elementen zu einer Menge.

4.1.1 Fuzzy–Mengen

Fuzzy–Mengen (auch vage oder unscharfe Mengen genannt) stellen die Grundlage für alle Fuzzy–Anwendungen dar und erweitern das Konzept der bekannten scharfen Mengen.
Eine klassische Menge A ist definiert durch die in ihr enthaltenen Elemente u aus einem Universum U. Man kann eine charakteristische Funktion $\chi_A(u)$ angeben, die 1 liefert, falls u ein Element von A ist, und die 0 wird, falls u nicht zu A gehört. Es gibt also für ein Element u nur diese zwei Möglichkeiten der Zugehörigkeit [HMB93].

$$\chi_A : U \rightarrow \{0,1\}$$

$$\chi_A(u) = \begin{cases} 1 & : \quad wenn \quad u \in A \\ 0 & : \quad sonst \end{cases} \tag{4.1}$$

In einer unscharfen Menge dagegen haben die Elemente Zugehörigkeitswerte, die im Bereich [0,1] liegen. Durch die charakteristische Funktion μ erhält jedes Element u aus U eine Zugehörigkeit $\mu_A(u)$ zu der Menge A. Also kann eine Fuzzy–Menge durch eine Menge von geordneten Paaren u und $\mu(u)$ beschrieben werden [HMB93]:

$$A = \{(u, \mu_A(u)) \text{ mit } u \in U\}$$

Für diskrete endliche Fuzzy–Mengen hat sich folgende Summenschreibweise eingebürgert:

$$A = \mu_A(u_1)/u_1 + \ldots + \mu_A(u_n)/u_n = \sum_{i=1}^{n} \mu_A(u_i)/u_i \tag{4.2}$$

Wenn u kontinuierlich ist, kann man eine Fuzzy–Menge auch folgendermaßen schreiben:

$$A = \int_u \mu_A(u)/u \tag{4.3}$$

Dabei haben die Symbole „/" und „+" natürlich nicht ihre sonst übliche mathematische Bedeutung. Das Plus–Symbol steht hier für die Vereinigung, und das Geteilt–Symbol deutet auf die Zusammengehörigkeit von u und $\mu(u)$ hin. Somit wird z.B eine Fuzzy–

Menge A, die bei 10 die Zugehörigkeit 0.5 und bei 20 die Zugehörigkeit 0.7 besitzt, als $A = 0.5/10 + 0.7/20$ geschrieben.

Der Vorteil der unscharfen Mengen gegenüber den klassischen Mengen besteht nun darin, daß man „schwammige" Begriffe wie z.B. „warm" und „kalt", oder „klein" und „groß" modellieren kann.

Beispiel:

Es sollen die Mengen *dumm* und *schlau* modelliert werden. In der scharfen Logik könnte man z.B. Menschen mit IQ 100–120 der Menge *schlau* zuordnen, die mit IQ 80–99 der Menge *dumm*. Somit würde eine Person mit einem Intelligenzquotienten von 99 als dumm bezeichnet, und eine andere mit IQ 100 als schlau bezeichnet werden (Bild 4.1). Man sieht, daß hier das Konzept der scharfen Mengen nicht anzuwenden ist. Nun schaffen die unscharfen Mengen Abhilfe. Dort kann nämlich ein Mensch mit einem IQ von 100 mit einer Zugehörigkeit von 50% zur Menge *dumm* und mit 50% zur Menge *schlau* gehören (Bild 4.1). Er läge also irgendwo dazwischen, was der menschlichen Logik doch wesentlich näher kommt.

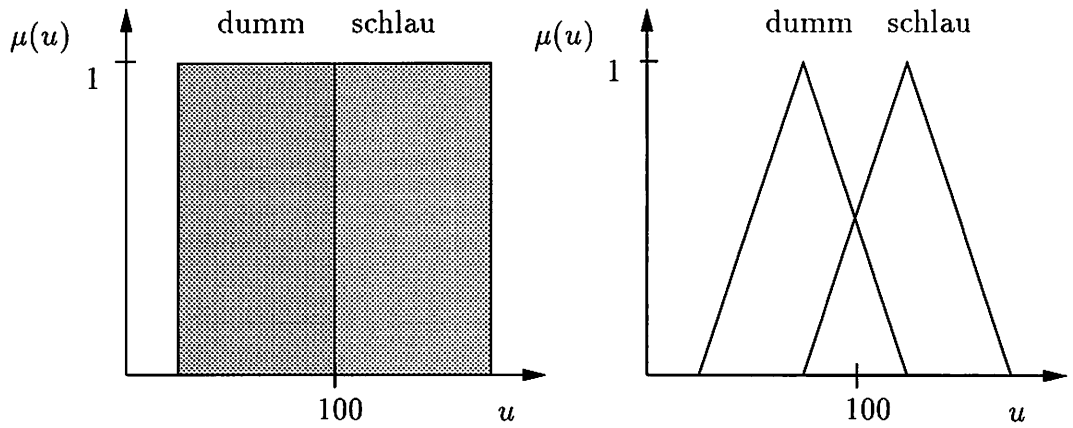


Bild 4.1: Scharfe und unscharfe Mengen

Im folgenden werden einige Begriffe, die in der Fuzzy–Theorie gebräuchlich sind, definiert [Zim92, HMB93, Tra94, Pro95]:

Definition 3 *Eine Fuzzy–Menge A heißt normalisiert, wenn gilt:*

$$sup_x \mu_A(x) = 1 \tag{4.4}$$

Anderenfalls nennt man die Fuzzy–Menge subnormal.

Definition 4 *Die Toleranz $T(A)$ einer Fuzzy–Menge A besteht aus allen Elementen x_i, deren Zugehörigkeitsgrad $\mu_A(x_i)$ gleich Eins ist:*

$$x_i \in T(A) \leftrightarrow \mu_A(x_i) = 1; \tag{4.5}$$

Definition 5 *Der Support (oder Träger) $S(A)$ einer Fuzzy-Menge A besteht aus allen Elementen x_i, deren Zugehörigkeitsgrad $\mu_A(x_i)$ größer als Null ist:*

$$x_i \in S(A) \leftrightarrow \mu_A(x_i) > 0 \tag{4.6}$$

Man spricht von einem kompakten Support, wenn dieser endlich ist.

Definition 6 *Die Höhe hgt einer Fuzzy-Menge* A *ist definiert als das Supremum ihrer Zugehörigkeitsgrade:*

$$hgt(A) = sup_x \mu_A(x) \tag{4.7}$$

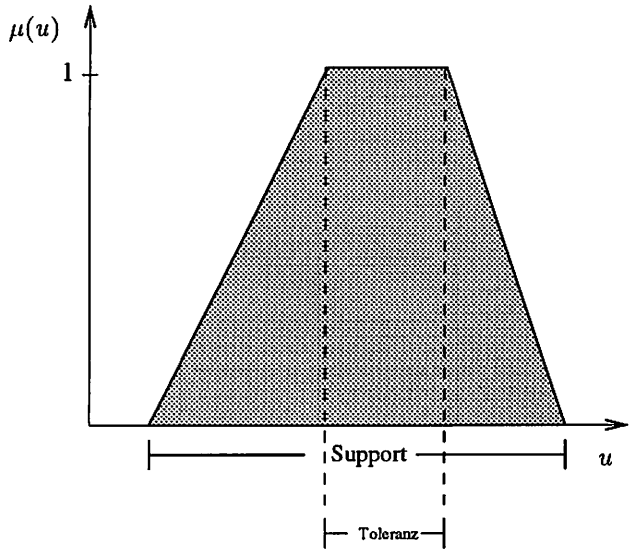


Bild 4.2: Toleranz und Support einer unscharfen Menge

Definition 7 *Der Betrag einer diskreten endlichen Fuzzy-Menge A ist die Summe aller Zugehörigkeitsgrade.*

$$|A| = \sum_{i=0}^{n} \mu_A(x_i) \tag{4.8}$$

Es gibt viele verschiedene Vorschläge für die Form von Fuzzy-Mengen (Bild 4.3). In der Praxis werden allerdings meist dreiecks- und trapezförmige Mengen verwendet.

Eine sehr allgemeine Klasse von Fuzzy-Mengen sind die sog. LR-Fuzzy-Mengen. Diese sind weitgehend durch zwei Funktionen L und R charakterisiert, wobei L die linke und R die rechte Flanke der Fuzzy-Menge beschreibt. Dabei müssen diese Funktionen folgende Bedingungen erfüllen:

- $F(0) = 1$
- $F(1) = 0$

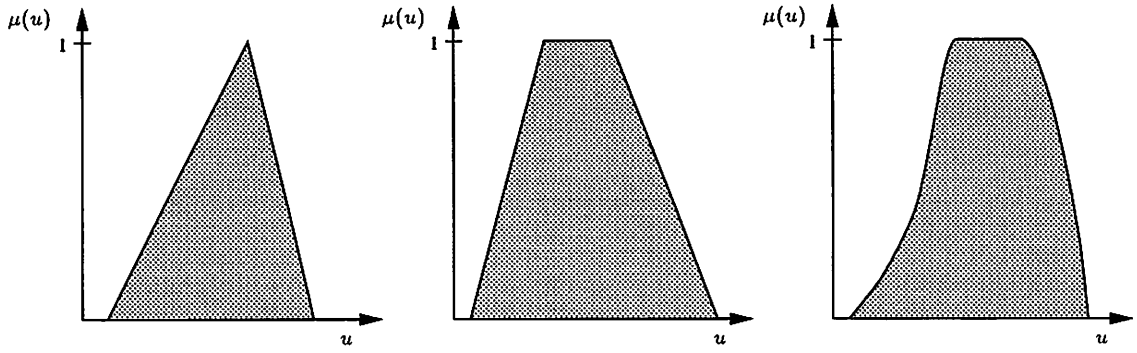


Bild 4.3: Verschiedene vorgeschlagene Formen von Fuzzy–Mengen

- $F(x) < 1$ falls $x > 0$
- $F(x) > 0$ falls $x < 1$

4.1.2 Linguistische Variablen

Linguistische Variablen können als Verallgemeinerung von Fuzzy–Mengen aufgefaßt werden [Alt95]. Während eine unscharfe Menge nur eine einzige Ausprägung einer Eigenschaft (z.B. schlau) beschreibt, beinhaltet eine linguistische Variable verschiedene Ausprägungen dieser Eigenschaft. Der Wertebereich einer linguistischen Variablen besteht aus sprachlichen Begriffen, die durch Fuzzy–Mengen repräsentiert werden. In diesem Zusammenhang werden die Fuzzy–Mengen auch als Terme bezeichnet. So wäre z.B. *Intelligenz* eine linguistische Variable mit den Termen *dumm*, *durchschnittlich intelligent*, und *schlau*. Um eine linguistische Variable zu visualisieren, werden die Zugehörigkeitsfunktionen der einzelnen Terme meist in ein Koordinatensystem eingetragen (Bild 4.4).

4.1.3 Fuzzy–Operatoren

Um mit Fuzzy–Mengen und linguistischen Variablen arbeiten zu können, benötigt man Operatoren, mit denen man diese verknüpfen kann. Schon Zadeh hat 1965 Operatoren zur Verknüpfung von unscharfen Mengen vorgeschlagen. Dies war der Minimum– und der Maximum–Operator. In [Tra94], [HMB93] und [Til92] werden verschiedene mögliche Operatoren für Fuzzy–Mengen vorgestellt. Davon sollen nun im folgenden einige erläutert werden.

Eine wichtige Gruppe von Fuzzy–Operatoren stellen die T–Normen und Co–T–Normen (oder S–Normen) dar [HMB93]. Dabei können die T–Normen als UND–Verknüpfung aufgefaßt und zur Durchschnittsbildung von Fuzzy–Mengen herangezogen werden. Die S–Normen sind ODER–Verknüpfungen und werden zur Vereinigung von unscharfen Mengen verwendet.

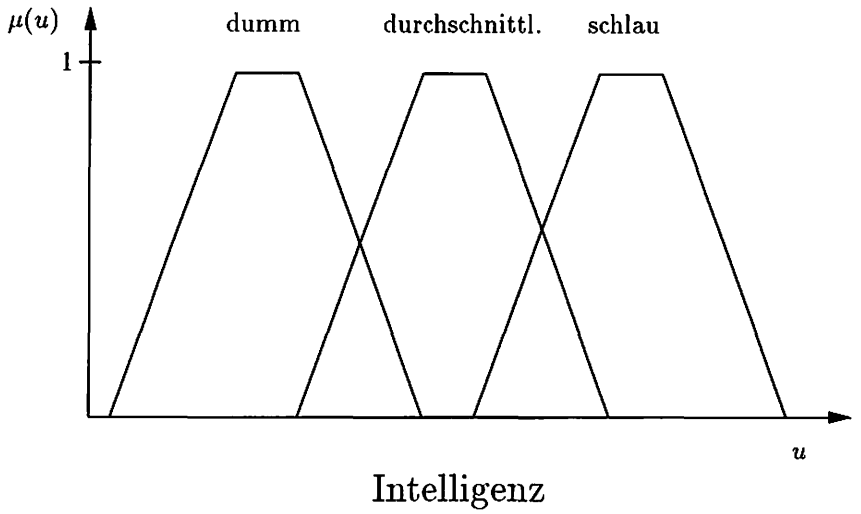


Bild 4.4: Linguistische Variable „Intelligenz"

T–Normen:

T–Normen sind zweiwertige Funktionen von $[0,1]\times[0,1]$ nach $[0,1]$ und erfüllen für $\mu(u) \in [0,1]$ die folgenden Bedingungen:

$t(\mu_A(u), t(\mu_B(u), \mu_C(u))) = t(t(\mu_A(u), \mu_B(u)), \mu_C(u))$ (Assoziativität)

$t(\mu_A(u), \mu_B(u)) = t(\mu_B(u), \mu_A(u))$ (Kommutativität)

$t(0,0) = 0; \quad t(\mu_A(u), 1) = t(1, \mu_A(u)) = \mu_A(u)$ (Randbedingungen)

$t(\mu_A(u), \mu_B(u)) \leq t(\mu_C(u), \mu_D(u))$ falls $\mu_A(u) \leq \mu_D(u)$ und $\mu_B(u) \leq \mu_D(u)$ (Monotonie)

Die Eigenschaft der Assoziativität ermöglicht es, UND–Verknüpfungen von mehr als zwei Mengen rekursiv durchzuführen. Es wurden mittlerweile ein ganze Reihe von T–Normen entwickelt [Til92]. Nachfolgend sind zwei Vertreter dieser Operatoren aufgeführt:

- Der Minimum–Operator (Zadeh 1965):

$$t_{min}(\mu_A(u), \mu_B(u)) = \mu_A(u) \wedge \mu_B(u) = min(\mu_A(u), \mu_B(u))$$

- Das Einstein–Produkt:

$$t_{1.5}(\mu_A(u), \mu_B(u)) = \frac{\mu_A(u)\mu_B(u)}{2 - (\mu_A(u) + \mu_B(u) - \mu_A(u)\mu_B(u))}$$

S–Normen

S–Normen bilden den Bereich $[0,1] \times [0,1]$ auf $[0,1]$ ab und haben die folgenden Eigenschaften:

$s(\mu_A(u), s(\mu_B(u), \mu_C(u))) = s(s(\mu_A(u), \mu_B(u)), \mu_C(u))$ (Assoziativität)

$s(\mu_A(u), \mu_B(u)) = s(\mu_B(u), \mu_A(u))$ (Kommutativität)

$s(1,1) = 1; \quad s(\mu_A(u), 0) = s(0, \mu_A(u)) = \mu_A(u)$ (Randbedingungen)

$s(\mu_A(u), \mu_B(u)) \leq t(\mu_C(u), \mu_D(u))$ falls $\mu_A(u) \leq \mu_D(u)$ und $\mu_B(u) \leq \mu_D(u)$ (Monotonie)

Zwei typische Vertreter dieser Operatorenklasse sind der Maximum–Operator und die Einstein–Summe:

- Der Maximum–Operator (Zadeh 1965):

$$s_{max}(\mu_A(u), \mu_B(u)) = \mu_A(u) \vee \mu_B(u) = max(\mu_A(u), \mu_B(u))$$

- Die Einstein–Summe:

$$s_{1.5}(\mu_A(u), \mu_B(u)) = \frac{\mu_A(u) + \mu_B(u)}{1 + \mu_A(u)\mu_B(u)}$$

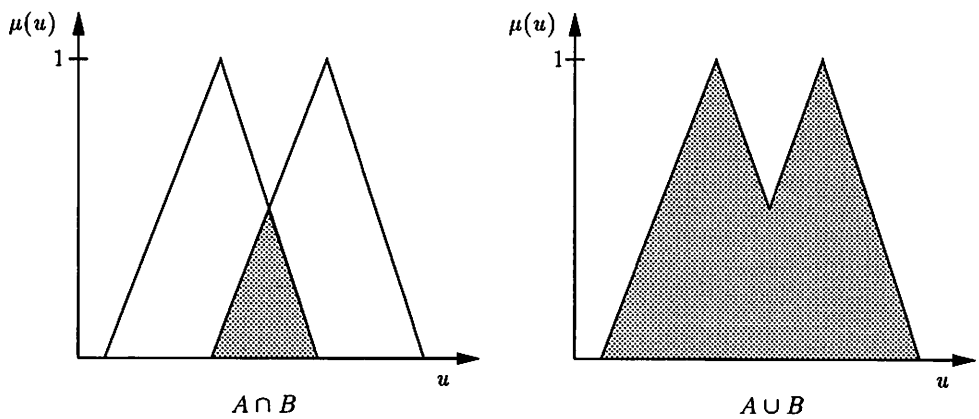


Bild 4.5: Minimum– und Maximum–Operator

Der Minimum–Operator hat keine kompensatorischen Eigenschaften. Das heißt, daß niedrige Werte nicht durch hohe Werte ausgeglichen werden können. Es ist immer der niedrigste Wert ausschlaggebend. Im Gegensatz dazu ist der Maximum–Operator vollkommen kompensatorisch. Ein großer Wert kompensiert hier immer einen kleinen. Die menschliche

Logik liegt meist irgendwo zwischen diesen zwei Extremen. Ein Operator, mit dem man Zwischenstufen von t_{min} und s_{max} realisieren kann, ist der sogenannte *Gamma-Operator*, der von Zimmermann und Zysno 1980 entwickelt wurde [Zim92]:

$$\mu(u) = \left[\prod_{i=1}^{m} \mu_i(u)\right]^{1-\gamma} \left[1 - \prod_{i=1}^{m}(1 - \mu_i(u))\right]^{\gamma}, \quad 0 \leq \gamma \leq 1$$

Für $\gamma = 1$ erhält man einen ODER-Operator mit voller Kompensation. Wählt man für γ den Wert 0, so entsteht ein UND-Operator, der keine kompensatorischen Eigenschaften besitzt. Für Werte im Bereich [0,1] ergeben sich Mischformen mit teilweiser Kompensation.
Es existieren noch eine ganze Reihe weiterer Operatoren für die Verknüpfung von Fuzzy-Mengen [HMB93]. Dabei sollte noch angemerkt werden, daß es keinen „Universal-Operator" gibt, der für alle Probleme gleich gut geeignet ist. Es muß von Fall zu Fall entschieden werden, welcher Operator am günstigsten erscheint. In der Praxis wird sehr häufig der Minimum-Operator in Kombination mit dem Maximum-Operator verwendet.

4.1.4 Approximatives Schließen

In diesem Abschnitt wird auf eine einfache Form des approximativen Schließens (auch Fuzzy-Inferenz genannt) eingegangen. In Fuzzy-Systemen wird Wissen meist in Form von Wenn-Dann-Regeln, oder sog. linguistischer Implikation dargestellt [HMB93].

$$A(u) \Rightarrow B(v) \quad oder \quad „WENN \;\; A(u) \;\; DANN \;\; B(v)"$$

Es wird also eine Relation zwischen dem Bedingungs-Teil $A(u)$ und der Konsequenz $B(v)$ festgelegt, die über die charakteristischen Funktionen $\mu(u)/u \in U$ und $\mu(v)/v \in V$ definiert ist. Diese Relation kann durch das kartesische Produkt dargestellt werden:

$$R = A \times B = \int_{U \times V} t(\mu(u), \mu(v))/(u, v) \tag{4.9}$$

Dabei ist t eine T-Norm (siehe Kapitel 4.1.3), und $U \times V$ die Menge von geordneten Paaren u und v. Das Integral ist im Sinne von Ausdruck (4.3) zu verstehen.
Hat man z.B. die Fuzzy-Mengen *kalt* und *heizen* mit $\mu_{kalt}(u)/u = 1/0 + 0.6/10 + 0.2/20 + 0/30$ und $\mu_{heizen}(u)/u = 0/0 + 0.2/10 + 0.6/20 + 1/30$, so ergibt der Schluß „WENN kalt DANN heizen" mit dem Minimum-Operator (siehe Kapitel 4.1.3) folgende Relation:

$k \Rightarrow h$	0	10	20	30
0	0	0.2	0.6	1
10	0	0.2	0.6	0.6
20	0	0.2	0.2	0.2
30	0	0	0	0

Wenn mehrere Regeln angewendet werden sollen, so schreibt man diese i.a. durch eine ODER-Verknüpfung.

$$\begin{array}{lllllll} R1: & & WENN & A_1(u) & DANN & B_1(v) \\ R2: & ODER & WENN & A_2(u) & DANN & B_2(v) \\ . & \ldots & \ldots & . & \ldots & . \\ Rn: & ODER & WENN & A_n(u) & DANN & B_n(v) \end{array}$$

Es ergibt sich also:

$$R = \int s(t(\mu_{A_i}(u), \mu_{B_i}(v)))/(u,v), \quad i = 1, \ldots, N \tag{4.10}$$

Verwendet man für die S-Norm s den Maximum-Operator und für die T-Norm t den Minimum-Operator, so erhält man:

$$R = \int \max_i(\min(\mu_{A_i}(u), \mu_{B_i}(v)))/(u,v), \quad i = 1, \ldots, N \tag{4.11}$$

Diese Relation stellt das Gedächtnis des Reglers dar. In ihr müssen alle Regeln gespeichert sein, die für die Regelung erforderlich sind. Da die Werte von u durch die Eingabe festgelegt sind, erhält man mit $R(u_0, v)$ eine Fuzzy-Menge, die von v abhängt, und das Inferenzergebnis zu einer Eingabe u_0 beschreibt.

4.2 Funktionsweise eines Fuzzy–Reglers

Die Aufgabe eines Fuzzy-Reglers besteht darin, aus einem scharfen Eingabevektor von Eingangssignalen eine passende, der Regelungsaufgabe angemessene scharfe Ausgabe zu erzeugen. Die interne Verarbeitung allerdings basiert auf der Theorie der unscharfen Mengen.

Ein Fuzzy-Regler besteht grundsätzlich aus drei Hauptelementen, die in gegenseitiger Beziehung stehen. Für die Eingabe besitzt er eine Menge von linguistischen Variablen. Normalerweise wird für jeden scharfen Eingang eine linguistische Variable bereitgestellt. Das zweite Element ist die Inferenzmaschine mit der Regelbasis, die für die Schlußfolgerungen verantwortlich ist. Schließlich gibt es noch einen Satz linguistischer Variablen für den Ausgang. Genauso wie beim Eingang ist jedem scharfen Ausgang eine linguistische

Variable zugeordnet. Oft gibt es allerdings nur einen einzigen Ausgang. In Bild 4.6 ist der prinzipielle Aufbau eines Fuzzy–Reglers dargestellt.

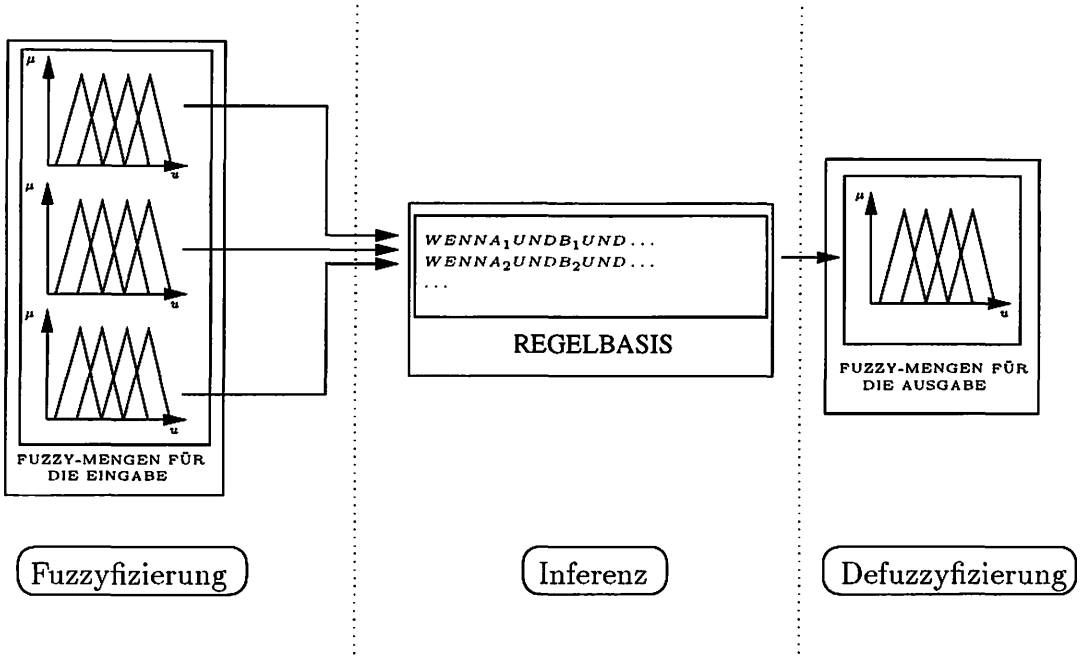


Bild 4.6: Schematischer Aufbau eines Fuzzy–Reglers

Die Verarbeitungsweise geht nun folgendermaßen vor sich: Zuerst müssen die scharfen Eingabewerte, die i.a. durch Messungen an der Regelstrecke gewonnen werden, in das Konzept der Fuzzy–Mengen transformiert werden. Dies geschieht durch die sog. Fuzzyfizierung. Nach der internen Verarbeitung müssen dann aus den sich ergebenden Fuzzy–Mengen wieder scharfe Ausgangswerte (Regelgrößen) erzeugt werden. Diesen Vorgang nennt man Defuzzyfizierung. Im folgenden werden die einzelnen Verarbeitungsschritte etwas beleuchtet.

4.2.1 Fuzzyfizierung

Man habe nun eine Reihe von Regeln in der folgenden Form:

$$\begin{array}{rlllll} & WENN & A_{11} & UND & A_{21} & UND \ \ldots \ DANN \ B_1 \\ ODER & WENN & A_{12} & UND & A_{22} & UND \ \ldots \ DANN \ B_2 \\ ODER & WENN & A_{13} & \ldots & & \end{array}$$

Da die Eingangswerte eines Reglers i.a. Meßgrößen sind, also scharfe Werte, müssen diese zunächst in das sprachliche Konzept der linguistischen Variablen umgesetzt werden. Dazu werden die Zugehörigkeitswerte (Kompatibilitätsmaße) aller Terme zu den aktuellen Eingangsgrößen berechnet (Bild 4.7). Man erhält also in jeder Regel j zu einer gegebenen

scharfen Eingabe e die Kompatibilitätsmaße a_{ij}:

$$\begin{aligned} a_{1j} &= komp(A_{1j}, e) \\ a_{2j} &= komp(A_{2j}, e) \\ &\vdots \\ a_{nj} &= komp(A_{nj}, e) \end{aligned}$$

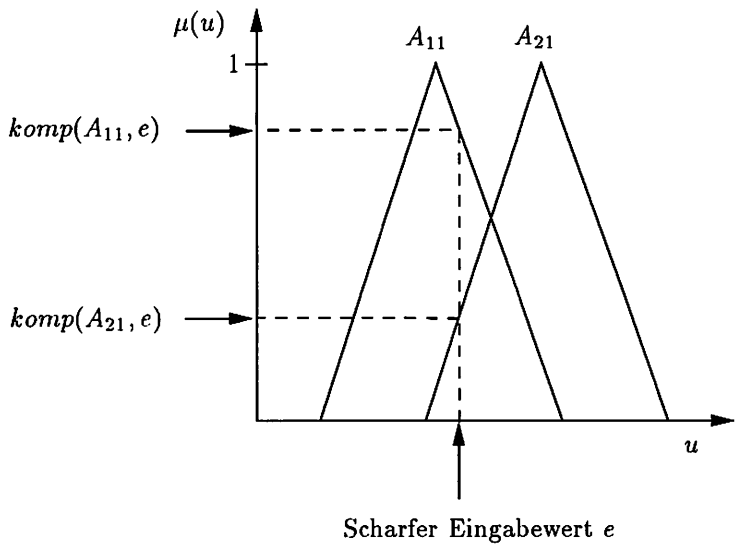


Bild 4.7: Vorgang der Fuzzyfizierung

4.2.2 Inferenz–Maschine

Die eigentliche Arbeit in einem Fuzzy–Regler wird durch die Inferenz–Maschine geleistet. Ihre Aufgabe ist es, aus gegebenen Fakten (Eingabedaten) die entsprechenden Schlußfolgerungen für die Ausgabe zu erzeugen. In ihr ist die gesamte Regelbasis gespeichert, die das Verhalten des Regelungsvorgangs bestimmt. Die Inferenz–Strategie führt nun zu folgender Vorgehensweise:

Nach der Fuzzyfizierung der scharfen Eingangswerte liegen die Kompatibilitätsmaße a_{ij} zu allen Fuzzy–Mengen der Eingabe vor. Mit diesen werden dann alle gespeicherten Regeln ausgewertet. Zunächst muß die linke Seite einer Regel j durch einen Durchschnitts–Operator *aggOp* zu einem Gesamtkompatibilitätsmaß a_j aggregiert werden. Für den Operator wird i.a. eine T–Norm verwendet, da eine UND–Verknüpfung vorliegt. Das Gesamtkompatibilitätsmaß a_j drückt den Erfüllungsgrad der Vorbedingung einer Regel j aus.

$$a_j = aggOp(a_{1j}, a_{2j}, \ldots, a_{nj}) \tag{4.12}$$

Führen dann mehrere Regeln zur gleichen Schlußfolgerung B_k, so muß noch eine Komposition der entsprechenden a_j mit Hilfe eines Vereinigungs–Operators stattfinden:

$$a_k^* = kompOp(a_j) \quad \text{für alle} \quad a_j, \quad \text{deren Regel die Schlußfolgerung} \quad B_k \quad \text{hat} \tag{4.13}$$

Es kann nun argumentiert werden, daß die Schlußfolgerungen von Regeln höchstens in dem Maße erfüllt sein sollten, wie die Vorbedingungen gelten. Die Fuzzy–Menge der Schlußfolgerung erhält man mit Hilfe eines Inferenz–Operators *infOp*:

$$B_k^* = infOp(a_k^*, B_k) \quad \text{mit} \quad hgt(B_k^*) < a_k^* \tag{4.14}$$

Es darf also die Höhe der resultierenden Fuzzy–Menge den Gesamtkompatibilitätsgrad a_k^* nicht übersteigen. Als Inferenz–Operatoren werden meist nicht–kompensatorische T–Normen verwendet. Wird etwa der Minimum–Operator (siehe Kapitel 4.1.3) eingesetzt, so wird die Fuzzy–Menge für die Schlußfolgerung einfach auf Höhe a_k abgeschnitten.

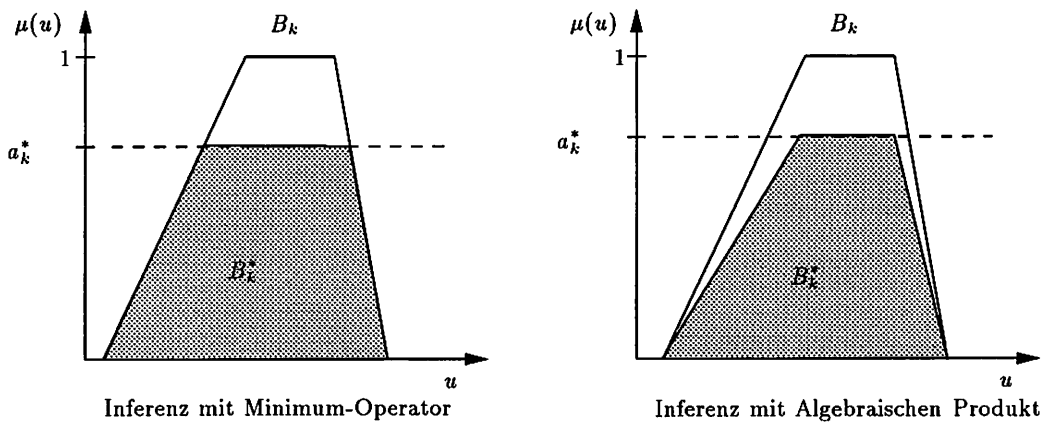


Bild 4.8: Verschiedene Inferenz–Operatoren

Die letzte Operation, die die Inferenzmaschine durchführen muß, ist die Akkumulation der erhaltenen Schlußfolgerungen B_k^*. Da die Regeln alternativen Charakter besitzen (siehe auch Kapitel 4.1.4), sollte man zur Akkumulation eine S–Norm verwenden. Die Ergebnis–Fuzzy–Menge B^* des Inferenzprozesses ergibt sich also aus:

$$B^* = accOp(B_1^*, B_2^*, \ldots, B_m^*) \tag{4.15}$$

Bekannte Inferenzschemata sind z.B. die *Max–Min–Inferenz* oder die *Max–Prod–Inferenz*, bei denen für die Akkumulations–Operation der Maximum–Operator und für den Inferenz–Operator der Minimum– bzw. Produkt–Operator verwendet wird [Alt95].

4.2.3 Defuzzyfizierung

Um aus den Fuzzy-Mengen, die sich aus der Inferenz ergeben haben, wieder scharfe Ausgangswerte zu erzeugen, gibt es eine Reihe von verschiedenen Methoden. Davon sollen nun zwei näher erläutert werden.

Mittelwert der Maxima (Mean of Maximum)

Die wohl einfachste Methode ist es, wenn man den Wert v_0 innerhalb des Definitionsbereichs V als Repräsentant für die Fuzzy-Menge wählt, an dem sie ihr globales Maximum hat:

$$v_0 = argmax(\mu_v(v)) \qquad \text{mit} \quad v \in V \tag{4.16}$$

Diese Methode ist allerdings nicht eindeutig, falls es mehrere Maxima mit gleichem Wert gibt. Nimmt man den Mittelwert der v-Werte aller dieser Maximalstellen, so tritt dieses Problem nicht auf:

$$v_0 = \sum_{j=1}^{n} \frac{v_j}{n} \tag{4.17}$$

Dabei ist n die Anzahl der Maximalstellen und v_j sind ihre Abszissenwerte.
Die obigen Verfahren zur Defuzzyfizierung beschreiben die sog. „Plausibelste Lösung", weil der Term bestimmend ist, der die höchste Zugehörigkeit besitzt, also am plausibelsten erscheint. Mit obigen Methoden wird die Information über die Form der Fuzzy-Mengen nicht verwendet, da nur die Position der Extremstellen entscheidend ist [HMB93].

Schwerpunkt-Methode (Center of Area)

Die klassische und wohl am weitesten verbreitete Defuzzyfizierungsmethode ist die Schwerpunkt-Methode. Dabei wird der Quotient des ersten Moments von $\mu_v(v)$ und der Fläche der Fuzzy-Menge gebildet:

$$v_o = \frac{\int_V v\mu_v(v)dv}{\int_V \mu_v(v)dv} \tag{4.18}$$

Somit erhält man den Abszissenwert des Flächenschwerpunktes der betrachteten Fuzzy-Menge. Durch diese Vorgehensweise wird der sog. „Beste Kompromiß" gebildet, da hier die gesamten Ausgabe-Fuzzy-Mengen auf das scharfe Endresultat Einfluß haben.

Man muß nun von Fall zu Fall entscheiden, welche Art der Defuzzyfizierung verwendet werden soll. Bild 4.10 zeigt eine Situation, in der eine Methode der „Plausibelsten Lösung" dem „Besten Kompromiß" vorzuziehen wäre:

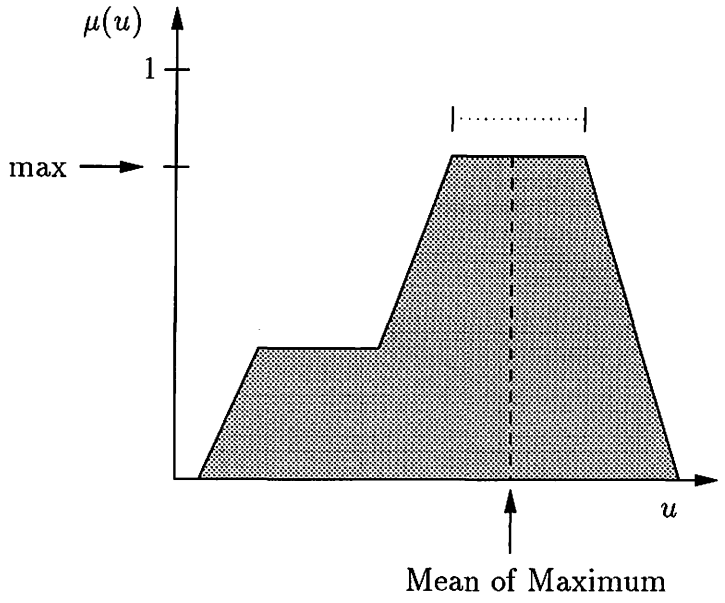


Bild 4.9: Defuzzyfizierung mit der „Mean of Maximum"-Methode

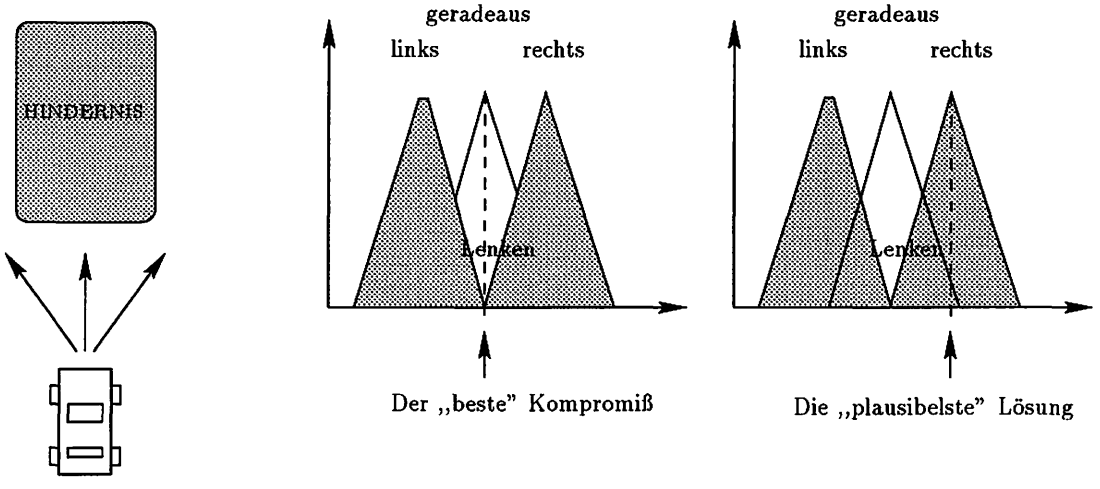


Bild 4.10: Problem beim Umfahren eines Hindernisses: Wird hier zur Defuzzyfizierung eine Methode des „Besten Kompromisses" gewählt, so wird das Auto geradeaus weiterfahren [Alt95]. Durch die „plausibelste Lösung" würde man sich hier entweder für die linke, oder die rechte Seite entscheiden.

Die Schwerpunkt-Methode erzeugt im Gegensatz zur „Mean of Maximum"-Methode glatte Verläufe der Ausgangsgröße. Mit ihr lassen sich ähnliche Resultate erzielen, wie mit den linearen Reglern (nach [HMB93, Alt95]).

Kapitel 5

Implementierung der Klassenhierarchien

Die Implementierung der in den bisherigen Ausführungen behandelten Regler wird nun in diesem Kapitel erläutert. Dabei werden die Klassenhierarchien, die zur Lösung der gestellten Aufgabe entwickelt wurden, eingehend beschrieben.

5.1 Allgemeine Anforderungen

Die Implementierung der Klassen wird in der Programmiersprache C++ vorgenommen. Dabei werden sie unter dem am LME bestehenden System PUMA [Pau91] entwickelt. Es wird die Klassenbibliothek NIHCL [KG91] als Ausgangsbasis verwendet. Die Klasse *Object* aus dieser Bibliothek dient als Wurzel für alle Klassenhierarchien. Dies bringt den Vorteil mit sich, daß die entwickelten Klassen dadurch kompatibel zu allen NIHCL–Klassen werden, und daß so die Mechanismen dieser Bibliothek, wie z.B. Container–Klassen oder Methoden zur Speicherung auf Festplatte, effektiv genutzt werden können. Deshalb sind auch in allen implementierten Klassen die von NIHCL geforderten Methoden realisiert.

Durch den objektorientierten Ansatz sollte infolge einer intelligenten Klassenhierarchie mehrfacher Programmcode weitgehend vermieden werden. Durch Kapselung in den Klassen wird eine erhöhte Sicherheit erzielt. Die Software sollte möglichst offen gestaltet werden, so daß sich spätere Erweiterungen des Systems leicht durchführen lassen. Die Schnittstellen müssen daher klar und exakt definiert sein. In C++ wird dies durch den Einsatz von *pure virtual functions* unterstützt. Durch eine einheitliche Schnittstelle für alle Regler können diese theoretisch untereinander beliebig ausgetauscht werden.

5.2 Die Regler–Klassenhierarchie

In Bild 5.1 ist die gesamte Klassenhierarchie der implementierten Regler zu sehen. Als Wurzel für den Ableitungsbaum dient die Klasse *RsRegler*, die von der NIHCL–Klasse *Object* abgeleitet ist. Alle Subklassen von *RsRegler* müssen die virtuelle Methode *Regle_()* implementiert haben, die den jeweiligen Regelalgorithmus ausführt und als Argumente einen Eingabe– und einen Ausgabevektor von reellen Zahlen erwartet. *Regle_()* wird von der in *RsRegler* implementierten Methode *Regle()* aufgerufen, die überprüft, ob die resultierenden Stellgrößen noch innerhalb des gültigen Wertebereichs liegen.

Von der Basisklasse für die linearen Regler *RsLinRegler* ist der Eingangs–/Ausgangs–Regler *RsEARegler*, sowie der Zustandsregler *RsZustRegler* abgeleitet. Die linearen Regler *RsPRegler*, *RsIRegler* usw. besitzen Methoden, um die aus der analogen Regelungstechnik bekannten Parameter für die Basisklasse *RsEARegler* umzusetzen.

Weitere Subklassen von *RsRegler* sind *RsFuzzyRegler*, *RsSerie* und *RsParallel*. In den nachfolgenden Kapiteln wird nun auf die Implementierung aller verwendeten Klassen eingegangen.

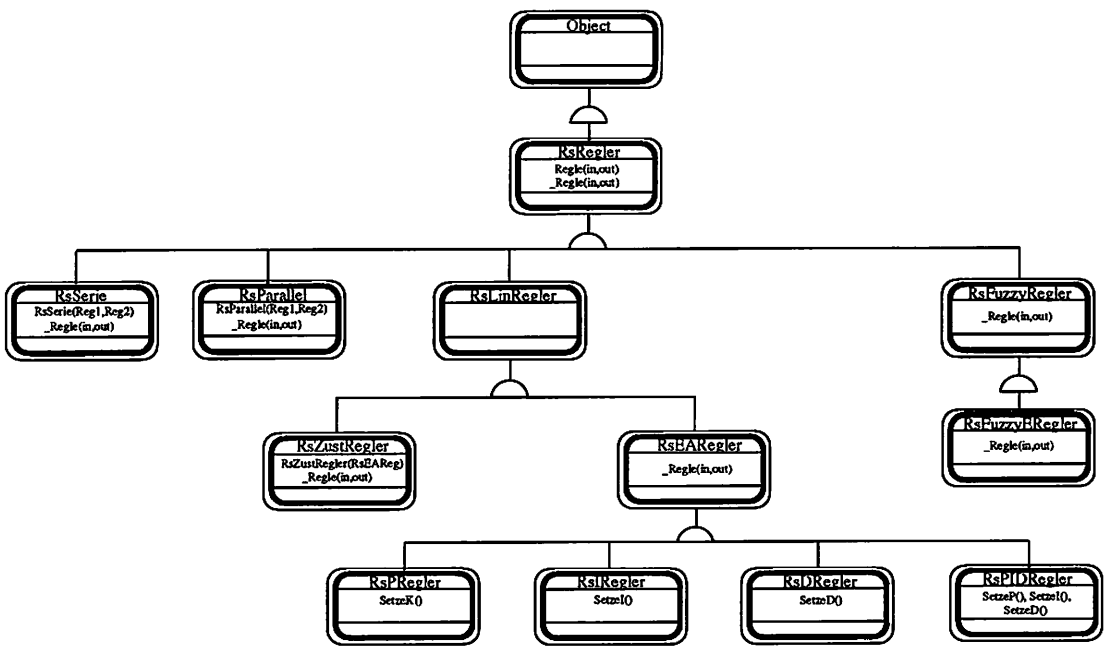


Bild 5.1: Klassenhierarchie der Regler

5.3 Die linearen Regler

5.3.1 E/A–Regler

Als Basisklasse aller Eingangs–/Ausgangs–Regler (siehe Kapitel 3.2) wurde die Klasse *RsEARegler* implementiert, die von *RsLinRegler* abgeleitet ist. Sie kann alle linearen Regler darstellen, die durch die z–Übertragungsfunktion (3.10) beschrieben sind. Durch die überladene Methode *Regle_()* erzeugt sie die Stellgröße $u(k)$ mit Hilfe des in Kapitel 3.2 vorgestellten rekursiven Regelalgorithmus. Es kann darüber hinaus ein Intervall für den Fehler $e(k)$ angegeben werden, in dem die Stellgröße den vorhergehenden Wert erhält. Dadurch ist es möglich, bei kleinen Regelabweichungen das Stellglied nicht unnötig zu belasten, was bei manchen Anwendungen sinnvoll sein kann. Alle E/A–Regler, die von *RsEARegler* abgeleitet sind, werden als Eingrößensysteme konzipiert, sie erzeugen also aus einer eindimensionalen Eingangsgröße $e(k)$ eine ebenfalls eindimensionale Stellgröße $u(k)$.

5.3.2 PID–Regler

Wie schon in Kapitel 3.2.1 vorgestellt, kann ein PID–Regler auch durch Parallelschaltung der Übertragungselemente P,I und D erzeugt werden. Wegen der häufigen Anwendung dieses Reglertyps und einer etwas effizienteren Implementierung wurde dieser Regler jedoch separat implementiert. *RsPIDRegler* wird von der Basisklasse *RsEARegler* abgeleitet, und bietet für kleine Abtastzeiten (im Vergleich zum Verhalten der Regelstrecke) die Funktionalität des verbreiteten analogen PID–Reglers. Dabei werden die bekannten Reglerparameter K, T_I, T_D, sowie die Abtastzeit T_0 in die entsprechenden Parameter der Basisklasse umgerechnet.

5.3.3 P–,I– und D–Regler

Die Klassen *RsPRegler*, *RsIRegler* und *RsDRegler* sind ebenfalls von der Basisklasse *RsEARegler* abgeleitet und verwenden die in Kapitel 3.2 vorgeschlagenen Regelalgorithmen. Durch Parallelschalten dieser Übertragungsglieder können dann PI–,PD–, und PID–Regler realisiert werden. Diese haben für genügend kleine Abtastzeiten das gleiche Verhalten, wie die bekannten analogen Regler, und es können deshalb weitgehend dieselben Methoden zum Einstellen dieser Regler verwendet werden.

5.3.4 Zustandsregler

Die Klasse *RsZustRegler* ist von der Basisklasse *RsLinRegler* abgeleitet und repräsentiert einen linearen Zustandsregler (siehe Kapitel 3.1.5). Im Gegensatz zu den

Eingangs-/Ausgangs-Reglern ist es damit möglich auch Mehrgrößenregelungen zu realisieren. Es existiert in dieser Klasse ein Konstruktor, der es erlaubt, jeden beliebigen von *RsEARegler* abgeleiteten Regler in einen Zustandsregler umzuwandeln.

5.4 Die Fuzzy–Klassen

Wie in Kapitel 4.2 beschrieben, werden für einen Fuzzy–Regler verschiedene Elemente benötigt. Im folgenden wird ein kurzer Überblick über die Implementierung der Bestandteile eines solchen Reglers, also z.B. Fuzzy–Mengen, linguistischer Variablen, Regelbasis, usw., gegeben.

5.4.1 Fuzzy–Mengen

Die Klassenhierarchie der Fuzzy–Mengen wurde weitgehend von [Til92] übernommen. Allen Fuzzy–Mengen gemeinsam ist eine Methode, die zu einem gegebenen reellen Wert seinen Zugehörigkeitsgrad zu der Menge liefert. Außerdem muß jede von *RsFuzzyMenge* abgeleitete Klasse eine Funktion implementiert haben, die den Schwerpunkt der Menge zurückliefert. Dies wird für die Defuzzyfizierung (siehe Kapitel 4.2.3) benötigt.

Die Subklasse *RsFuzzyDreieck* ist eine Spezialisierung der Klasse *RsFuzzyTrapez*, da hier ein Trapez vorliegt, bei dem die oberen zwei Eckpunkte zusammenfallen. Eine trapezförmige Fuzzy–Menge ihrerseits ist eine Spezialisierung einer LR–Fuzzy–Menge (siehe Kapitel 4.1.1), wobei die beiden Flanken durch lineare Funktionen beschrieben werden. Die Klasse *RsFuzzyParabel* besitzt als Funktionen für die Flanken Parabeln. *RsFuzzyFeld* schließlich ist eine Fuzzy–Menge, die ein Array von reellen Zahlen enthält, in dem die Zugehörigkeitsfunktion diskret gespeichert ist. Bild 5.2 zeigt den Ableitungsbaum der implementierten Fuzzy–Mengen.

5.4.2 Linguistische Variable

Die Klasse *RsFuzzyLingV* repräsentiert eine linguistische Variable, wie sie in Kapitel 4.1.2 beschrieben ist. In ihr können alle Fuzzy–Mengen gespeichert werden, die von *RsFuzzyMenge* abgeleitet sind. Sie besitzt die Methode *Berechne()*, die als Argument eine reelle Zahl erwartet und in allen enthaltenen Fuzzy–Mengen die Zugehörigkeiten zu diesem Wert berechnet (siehe Fuzzyfizierung Kapitel 4.2.1). Diese Zugehörigkeitswerte sind danach in den Fuzzy–Mengen gespeichert. Die Methode *Inferenz()* erzeugt zu einem gegebenen reellen Eingabevektor, der die einzelnen Zugehörigkeitsgrade der Fuzzy–Mengen enthalten muß, mit zwei Operatoren zur Aggregation und Komposition (siehe Kapitel 4.1.4) eine Ergebnis–Fuzzy–Menge. Diese stellt die aktuelle Schlußfolgerung zu dem übergebenen Zugehörigkeitsvektor dar.

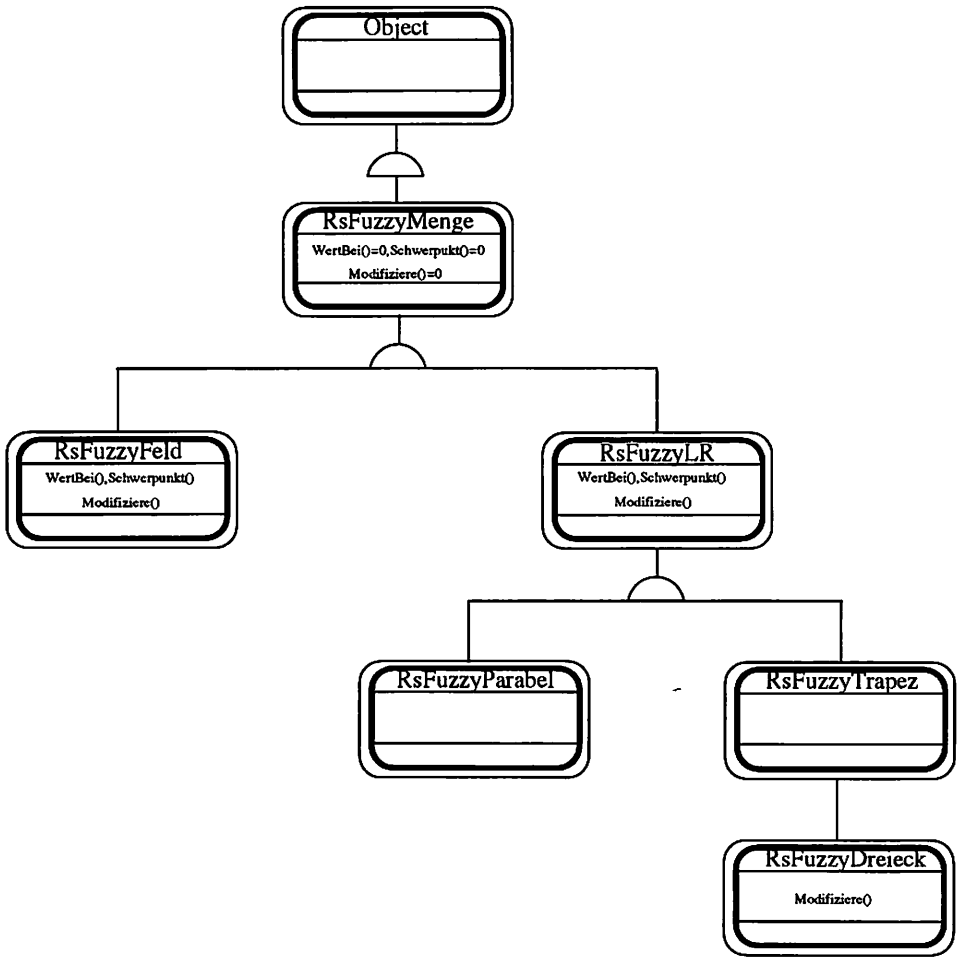


Bild 5.2: Klassenhierarchie für Fuzzy–Mengen

5.4.3 Fuzzy–Operatoren

Die Klassenhierarchie für Fuzzy–Operatoren vereinigt eine Reihe von T– und S–Normen (siehe Kapitel 4.1.3). Bild 5.3 zeigt die Klassenhierarchie für die T–Normen. Der entsprechende Baum für S–Normen sieht weitgehend gleich aus, und wird deshalb hier nicht angegeben. Für alle Fuzzy–Operatoren existiert eine gemeinsame Basisklasse *RsFuzzyOperator*. Durch Verwendung von *pure virtual functions* wird eine einheitliche Schnittstelle für alle Operatoren geschaffen. Jeder Operator muß eine Methode implementiert haben, die bei der Verknüpfung von Fuzzy–Mengen verwendet wird und zwei reelle Zahlen als Argumente erwartet. Somit können alle Klassen, die diese Operatoren verwenden, auf die gleiche Schnittstelle zurückgreifen, und dadurch mit den verschiedensten Fuzzy–Operatoren zusammenarbeiten. Durch diese Konstruktion wird also ein sehr hohes Maß an Flexibilität und Erweiterbarkeit erreicht.

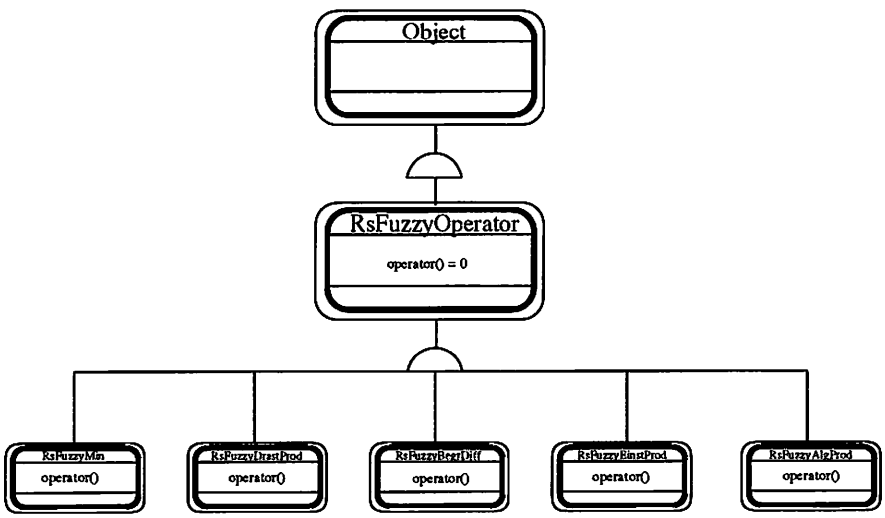


Bild 5.3: Klassenhierarchie für Fuzzy–T–Normen

Die Operatoren, die in Tabelle 5.1 aufgeführt sind, wurden für diese Arbeit implementiert. Es gibt noch ein Vielzahl weiterer Operatoren, aber für die Masse der Anwendungen wird die implementierte Auswahl genügen.

5.4.4 Regelbasis

Die Klasse *RsFuzzyRegelB* repräsentiert die Regelbasis eines Fuzzy–Reglers. Sie besitzt Methoden zum Hinzufügen von Regeln und zum Auswerten der Regelbasis. Diese Klasse könnte, eventuell durch Bildung einer Subklasse, auch für unscharfe Expertensysteme verwendet werden.

T–Normen	S–Normen
Minimum–Operator	Maximum–Operator
drastisches Produkt	drastische Summe
begrenzte Differenz	begrenzte Summe
Einstein–Produkt	Einstein–Summe
Algebraisches Produkt	Algebraische Summe
Hamacher–Produkt	Hamacher–Summe

Tabelle 5.1: Implementierte Fuzzy–Operatoren

5.4.5 Fuzzy–Regler

Der Fuzzy–Regler ist Bestandteil der Reglerhierarchie (siehe Kapitel 5.2) und ist von der Basisklasse *RsRegler* abgeleitet. In *RsFuzzyRegler* werden nun alle oben beschriebenen Bausteine vereint. Es wird ein Teil der Schnittstelle von *RsFuzzyRegelB* nach außen geführt, wie z.B. eine Methode zum Hinzufügen von neuen Regeln. Der zunächst geplante Ansatz, Mehrfachvererbung zu verwenden und den Fuzzy–Regler als Subklasse von *RsRegler* und *RsFuzzyRegelB* zu bilden, mußte aus folgenden Gründen wieder fallengelassen werden: Die Verwendung von Mehrfachvererbung und die damit verbundene Verwendung von virtuellen Basisklassen bringt einen Overhead von ca. Faktor 5 [KG91] mit sich, der gerade bei zeitkritischen Regelungsanwendungen nicht vertretbar erscheint. Außerdem hat sich gezeigt, daß es programmiertechnisch nicht unkritisch ist, Mehrfachvererbung zusammen mit NIHCL und PUMA einzusetzen.

Der Fuzzy–Regler wurde so entworfen, daß er sehr flexibel zu konfigurieren ist. Durch Übergabe von vier verschiedenen Operatoren (siehe Kapitel 5.4.3) kann der Inferenzprozeß (siehe Kapitel 4.2.2) individuell eingestellt werden. Darüber hinaus können die verschiedensten Fuzzy–Mengen, die von *RsFuzzyMenge* abgeleitet sind, in den linguistischen Variablen für die Ein– und Ausgabe verwendet werden. Der Fuzzy–Regler ist für n scharfe Eingänge und einen scharfen Ausgang konzipiert. Sollten mehrere Ausgänge benötigt werden, so müssen verschiedene Fuzzy–Regler parallel betrieben werden.

5.4.6 Fuzzy–Einstell–Regler

Die Klasse *RsFuzzyERegler* ist vorgesehen, um die Einstellung eines Fuzzy–Reglers für eine gegebene Regelstrecke zu erleichtern. Sie ist von *RsFuzzyRegler* abgeleitet und erbt die gesamte Schnittstelle der Basisklasse. Mit Hilfe der Methode *StartKonf()* wird der Regler in den Einstellmodus versetzt. Es sind zwei unterschiedliche Modi zum Einstellen des Reglers implementiert:

Der sog. STACK–Modus bewirkt, daß in festgelegten Abständen (bestimmte Anzahl

von Regelungen) nacheinander verschiedene Einstellungen des Reglers aus einer Konfigurationsdatei gelesen werden, und der Regler dann anschließend mit den neuen Werten betrieben wird. Es besteht hiermit also die Möglichkeit, die Parameter der im Regler enthaltenen linguistischen Variablen zu ändern.

Der zweite Modus ist der sog. ONLINE–Modus. Dabei wird aus der Konfigurationsdatei während des Betriebs des Reglers in bestimmten Zeitabständen jeweils nur die erste Konfiguration eingelesen. Somit kann der Benutzer immer wieder Änderungen an der Einstellung des Reglers vornehmen, indem er die Konfigurationsdatei abändert. Es kann dann sofort die Auswirkung auf das Regelverhalten im laufenden System beobachtet werden. Dieser Modus ist insbesondere für die Feineinstellung des Reglers gedacht.

5.5 Sonstige Reglerklassen

Um neue Regler durch Parallelschaltung oder Serienschaltung von einfachen oder bereits zusammengesetzten Reglern zu erzeugen, sind die Klassen *RsParallel* und *RsSerie* vorgesehen. Sie sind von *RsRegler* abgeleitet und können deshalb wie herkömmliche Regler verwendet werden. Ein Beispiel für den Einsatz dieser Klassen wäre die Erzeugung eines PI–Reglers durch Parallelschaltung eines P– und I–Gliedes.

Kapitel 6

Ergebnisse

In diesem Kapitel wird ein Modell für die Regelstrecken entworfen, das zur Einstellung der linearen Regler dient. Es wird beschrieben, wie die Parameter der verschiedenen Regler ermittelt werden. Ihr Verhalten im geschlossenen Regelkreis wird anhand von Simulationen und Experimenten untersucht. Anschließend erfolgt ein Vergleich der Ergebnisse.

6.1 Modellbildung

Um der Bewegung des Objektes zu folgen, werden eine Canon–Kamera sowie eine Roboter–Kamera verwendet. Beide können Schwenk– und Kipp–Bewegungen ausführen, d.h. daß Rotationen um zwei Raumachsen möglich sind. Damit nun für die gestellte regelungstechnische Aufgabe die Parameter der Regler richtig eingestellt werden können, wäre es hilfreich, wenn ein mathematisches Modell der Regelstrecke zur Verfügung stehen würde, mit dem man auf theoretischem Wege eine geeignete Einstellung finden könnte. Da auch lineare Regler verwendet werden, sollte dieses Modell ebenfalls linear sein. Es werden nun horizontale und vertikale Bewegungen gesondert betrachtet, so daß man sich auf den eindimensionalen Fall beschränken kann. Die weiteren Untersuchungen können also sowohl auf die Schwenk–, als auch auf die Kipp–Bewegung übertragen werden.

Zunächst werden die Ein– und Ausgänge der Regelstrecke charakterisiert. Die Eingangsgröße der Strecke ist ein Wert, der die Winkelgeschwindigkeit der Kamera beeinflußt. Sie wirkt also direkt auf die Drehzahl des Motors, die die Geschwindigkeit der Bewegung bestimmt. Somit erhält man als erstes Übertragungsglied G_M der Strecke die Umsetzung von angelegter Sollgeschwindigkeit auf die Drehzahl des Motors, der die Kamera bewegt. Die Struktur dieses Gliedes muß noch ermittelt werden. Am Ausgang der Strecke wird der Winkel der Kamerastellung α_{Kam} bezüglich eines festen Koordinatensystems gemessen. Der Winkel entsteht durch Integration der Motordrehzahl, die wiederum proportional zur Winkelgeschwindigkeit der Kamera ist. Hat man also eine konstante Umdrehungszahl des Motors, so erhält man am Ausgang einen linearen Verlauf der Kameraposition. Es ist somit

das zweite Übertragungsglied der Regelstrecke durch einen Integrator bestimmt, der die Umsetzung von Motordrehzahl auf Raumwinkel (absoluter Position der Kamera) leistet. Nachfolgende Formel (6.1) beschreibt die s–Übertragungsfunktion eines Integralgliedes.

$$G_I(s) = \frac{1}{sT_{I_S}} \tag{6.1}$$

Man sieht, daß hier nur ein einziger Parameter T_{I_S} zu bestimmen ist. Dies ist gerade die Zeit, die das I–Glied bei einer Sprungfunktion am Eingang benötigt, um den gleichen Wert wie ein P–Glied am Ausgang zu erreichen (siehe Kapitel 3.2.3). T_{I_S} hängt also von der Geschwindigkeit der Kamerasteuerung bei Eingangsgröße $u = 1$ ab. Die gesamte Regelstrecke kann somit durch eine Hintereinanderschaltung des Übertragungsgliedes für die Motoransteuerung und des I–Gliedes modelliert werden.

Das zu verfolgende Objekt besitzt ebenfalls eine absolute Position, die durch den Winkel α_{Obj} (siehe Bild 6.1) bestimmt ist. Durch Differenzbildung der Winkel α_{Kam} und α_{Obj} ergibt sich die Abweichung $\Delta\alpha$ der Kameraposition von der Objektposition.

Das Regelungssystem wird nun als Festwertregelung (siehe Kapitel 2) ausgelegt, bei der als Störgröße die Position des Objekts am Ausgang der Regelstrecke eingreift. Die Führungsgröße w ist dabei konstant Null, weil ein Verschwinden der Positionsabweichung $\Delta\alpha$ angestrebt wird. $\Delta\alpha$ wird mit positiven Vorzeichen zum Eingang des Reglers (mit Übertragungsfunktion $G_R(s)$) zurückgeführt, wodurch der geschlossene Regelkreis entsteht (siehe Bild 6.2).

Befindet sich das Objekt jetzt beispielsweise rechts von der Kameraposition, so ergibt sich eine positive Regelabweichung, weil das Objekt eine größere absolute Position besitzt als die Kamera. Dies bedingt dann wiederum eine positive Stellgröße.

Obige Annahmen setzten voraus, daß die Abweichung des Winkels α_{Obj} von α_{Kam} gemessen werden kann. Da jedoch das Meßglied in Form einer Kamera gegeben ist, wird die relative Position Δx_p des Objekts zum Bildmittelpunkt gemessen. Verwendet man das Lochkameramodell (siehe Bild 6.1), ergibt sich jedoch eine einfache Beziehung zwischen $\Delta\alpha$ und Δx_p, nämlich

$$\Delta x_p = \tan(\Delta\alpha f_l) \qquad \textit{und} \qquad \Delta\alpha = \arctan\left(\frac{\Delta x_p}{f_l}\right). \tag{6.2}$$

Hierbei ist f_l die Brennweite der Kamera.

Man erhält also das erweiterte Modell 2 (Bild 6.3), bei dem die Winkelabweichung $\Delta\alpha$ zunächst in die beobachtbare relative Position Δx_p in der Bildebene umgesetzt wird. Die gemessene Position muß dann zur Rückführung wieder in den Winkel $\Delta\alpha$ umgewandelt werden.

Modell 1 und 2 sind vom dynamischen und statischen Verhalten identisch, weil die

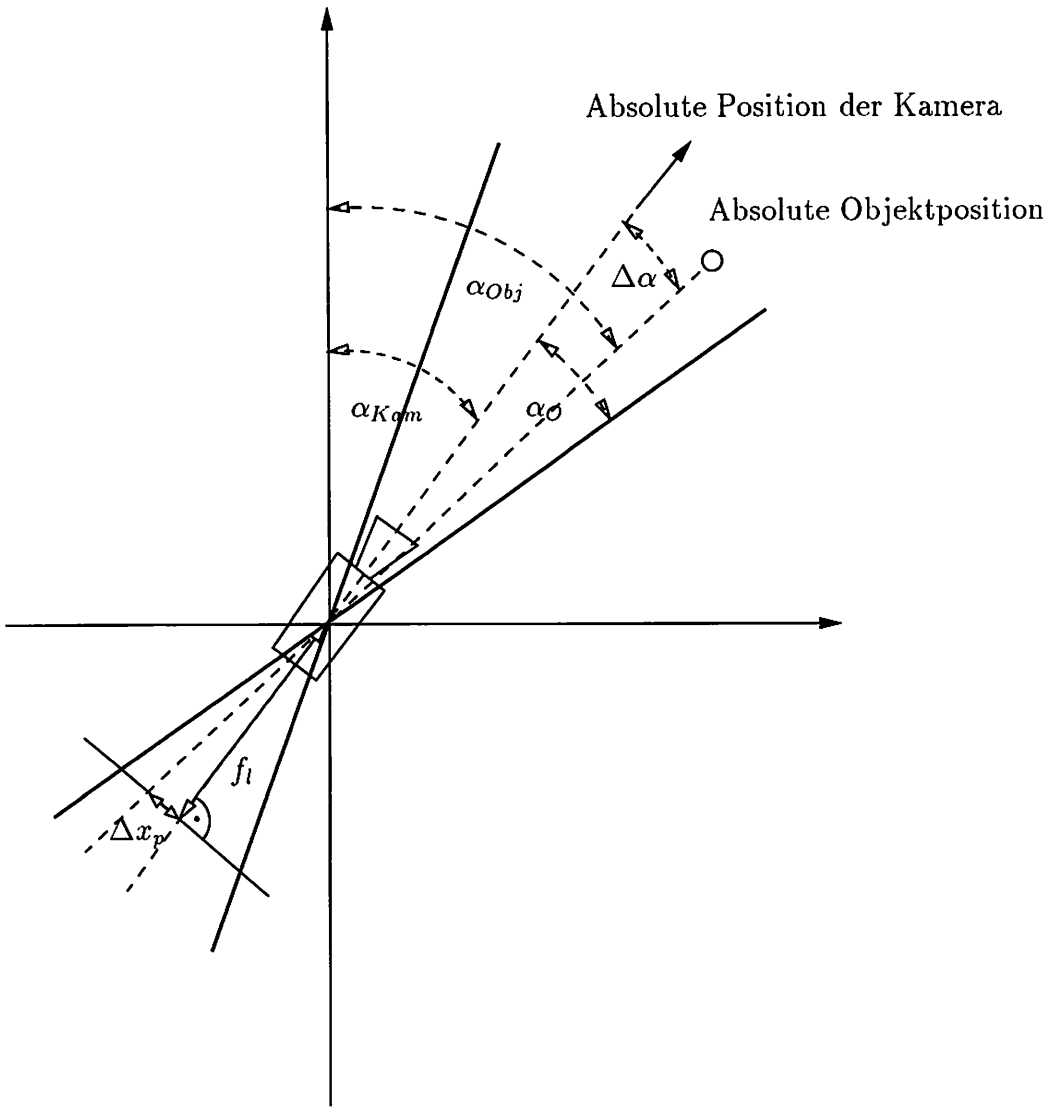


Bild 6.1: Lochkamera–Modell

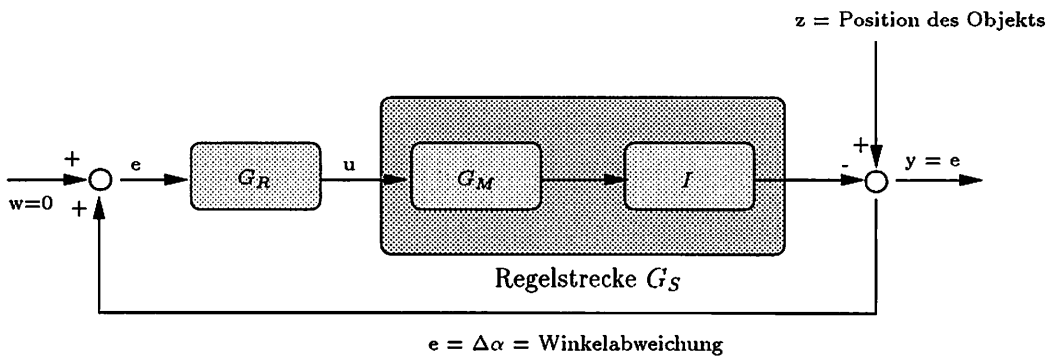


Bild 6.2: Modell 1 der Testumgebung

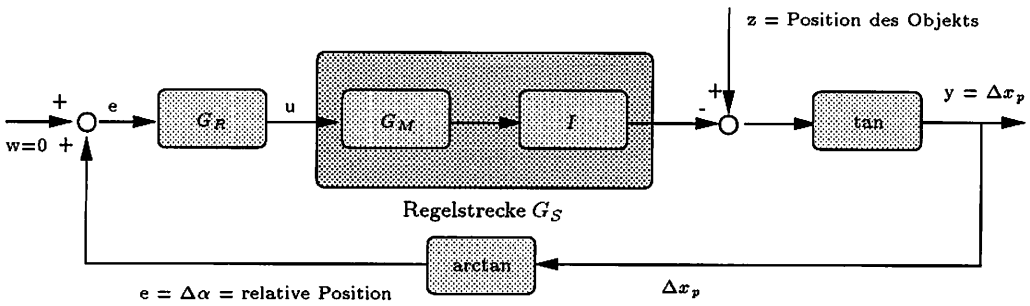


Bild 6.3: Modell 2 des geschlossenen Regelkreises

Rückführung beim zweiten Modell gerade so gewählt wurde, daß genau der gleiche Winkel $\Delta\alpha$ wie im Modell 1 als Eingangsgröße für den Regler dient. Im folgenden soll das erste Modell dazu verwendet werden, um geeignete Regler zu entwerfen, damit die Regelabweichung möglichst schnell eliminiert wird, wobei das Objekt nicht verloren werden sollte. Dazu muß nun zunächst die Übertragungsfunktion des geschlossenen Regelkreises ermittelt werden. Für $Y(s)$ ergibt sich direkt aus der Struktur des Regelkreises folgende Lösung:

$$\begin{aligned} Y(s) &= Z(s) - [W(s) + Y(s)]\, G_R(s) G_S(s) \\ Y(s) &= \frac{1}{1 + G_R(s) G_S(s)} Z(s) \end{aligned} \tag{6.3}$$

Die Laplace–transformierten Signale sind durch Großbuchstaben gekennzeichnet. Hieraus erhält man mit $G_0(s) = G_R(s)G_S(s)$ für die Störungsübertragungsfunktion:

$$G_Z(s) = \frac{Y(s)}{Z(s)} = \frac{1}{1 + G_0(s)} \tag{6.4}$$

6.2 Normierung

Um die verwendeten Größen theoretisch besser handhaben zu können, sollten die realen Werte normiert, also dimensionslos gemacht werden. Damit können Ergebnisse leichter miteinander verglichen werden. Die erste Normierung muß bei der in Pixeln gemessenen Abweichung vom Nullpunkt vorgenommen werden. Üblicherweise wird als normierter Wertebereich [-1..1] verwendet. Somit soll auch die gemessene Abweichung in x–Richtung (Δx_{Pixel}) und in y–Richtung (Δy_{Pixel}) auf diesen Bereich abgebildet werden.

$$\Delta x_{norm} = \frac{\Delta x_{Pixel}}{\frac{X_{Res}}{2}} \qquad \text{und} \qquad \Delta y_{norm} = \frac{\Delta y_{Pixel}}{\frac{Y_{Res}}{2}} \tag{6.5}$$

Hier sind X_{Res} und Y_{Res} die Auflösungen der Kamera in x– und y–Richtung, und der Wertebereich von Δx_{Pixel} und Δy_{Pixel} wird mit $[-\frac{X_{Res}}{2}..\frac{X_{Res}}{2}]$ bzw. $[-\frac{Y_{Res}}{2}..\frac{Y_{Res}}{2}]$ angenommen.

Für die Winkelabweichung $\Delta\alpha$ ist es ebenfalls zweckmäßig, als normierten Bereich das Intervall [-1..1] zu verwenden, wobei in den Fällen $\Delta\alpha_{norm} = 0$ und $\Delta\alpha_{norm} = \pm 1$ die Beziehung $\Delta\alpha_{norm} = \Delta x_{norm}$ gelten soll. Mit Bild 6.1 ergibt sich dann für $\Delta\alpha_{norm}$ der Ausdruck

$$\Delta\alpha_{norm} = \frac{1}{\alpha_O} \arctan\left(\frac{\Delta x_{norm}}{f_{norm}}\right) \qquad \text{mit} \qquad f_{norm} = \frac{1}{\tan(\alpha_O)}. \tag{6.6}$$

Dabei ist f_{norm} die skalierte Brennweite, die durch die Normierung von Δx_{Pixel} festgelegt ist. Für den die vertikale Bewegung betreffenden Winkel $\Delta\beta_{norm}$ ergibt sich die Normierung analog zu $\Delta\alpha_{norm}$.

Als letzte zu normierende Größe bleibt die Stellgröße, die ebenfalls wieder auf den Bereich [-1..1] abgebildet wird. Dabei bewirkt eine Stellgröße mit Wert $U_{x_{max}}$ bzw. $U_{y_{max}}$ die maximale Geschwindigkeit der Kamera in die jeweilige Richtung. Die Regler erzeugen die Größen $u_{x_{norm}}$ und $u_{y_{norm}}$, die dann durch Multiplikation mit $U_{x_{max}}$ bzw. $U_{y_{max}}$ für die Ansteuerung der Kamera umgerechnet werden.

$$u_{x_{real}} = u_{x_{norm}} U_{x_{max}} \qquad \text{und} \qquad u_{y_{real}} = u_{y_{norm}} U_{y_{max}} \tag{6.7}$$

6.3 Die Regelstrecken

Die Objektverfolgung und die Regelung arbeiten parallel auf zwei verschiedenen Rechnern (siehe [DN95]). Für die Prozeßkommunikation wird die pvm–Bibliothek [GBD*93] verwendet, wodurch für den Benutzer die physikalischen Maschinen verborgen werden. Die Objektverfolgung sendet etwa alle $T_0 = 0.14$ Sekunden neue Positionsangaben an den Regler. Dieser spricht über eine RS232–Schnittstelle die Canon– bzw. die Roboter–Kamera an.

6.3.1 Canon–Kamera

Die verwendete Canon–Kamera vom Typ *VC–C1* ist eine pan–/tilt–Kamera, bietet also die Möglichkeit, Schwenk– und Kippbewegungen auszuführen. Die Zeitkonstanten des Motors sind im Vergleich zur Abtastzeit der Objektposition sehr klein, so daß man das Übertragungsglied, das die Umsetzung von angegebener Wunschgeschwindigkeit auf die Drehzahl des Motors beschreibt, näherungsweise als reines P–Glied modellieren kann. Dies bedeutet, daß nach Angabe einer Sollgeschwindigkeit diese bis zum folgenden Abtastschritt bereits erreicht ist. Der Einfachheit halber wird als Verstärkungsfaktor Eins gewählt, was beim folgenden Übertragungsglied berücksichtigt wird. Somit ist die Übertragungsfunktion der gesamten Regelstrecke ein einfaches I–Glied, da ein P–Glied mit

Verstärkungsfaktor Eins keine Auswirkung auf das System besitzt. Die Übertragungsfunktion des geschlossenen Regelkreises lautet dann mit (6.4):

$$G_Z(s) = \frac{1}{1 + G_R \frac{1}{T_{I_S} s}} \tag{6.8}$$

Der Öffnungswinkel der Kamera α_O wurde bei einer festen Zoom–Einstellung horizontal mit 11.3 Grad und vertikal mit 8.5 Grad bestimmt. Diese Winkel entsprechen 150 bzw. 120 Schritten des Schrittmotors. Um die Zeitkonstante T_{I_S} für die Canon–Kamera näherungsweise zu ermitteln, mußte der Zusammenhang zwischen Eingangsgröße und Kamerageschwindigkeit für die Schwenk– und Kipp–Bewegung gemessen werden. Bild 6.4 und 6.5 zeigen die daraus resultierenden Kennlinien, wobei jeweils die Winkelgeschwindigkeit (Schritte/Sekunde) gegen die Eingangsgröße aufgetragen ist.

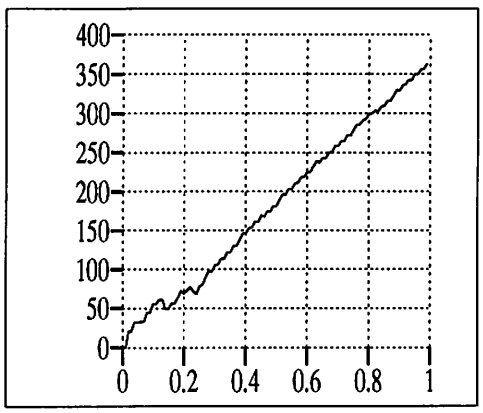


Bild 6.4: Kennlinie für horizontale Bewegung

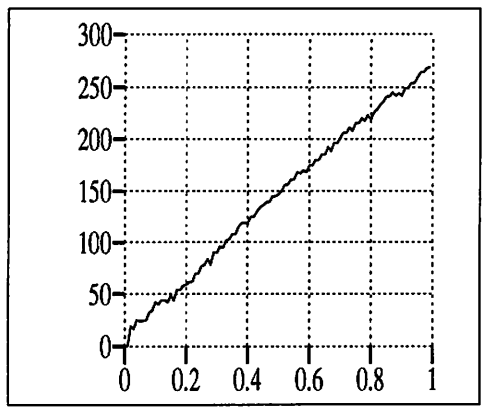


Bild 6.5: Kennlinie für vertikale Bewegung

Die Kennlinien sollen näherungsweise als linear angenommen werden. Damit ergibt sich für die horizontale Bewegung eine Steigung von 360, was bedeutet, daß bei Eingangsgröße Eins die Kamera eine Winkelgeschwindigkeit von 360 Schritten pro Sekunde erreicht. T_{I_S} ist die Zeit, die das I–Glied benötigt, um bei der Sprungfunktion als Eingabe am Ausgang den Wert Eins zu erzeugen, was nach der Normierung gerade α_O, also 150 Schritten entspricht. Es ergibt sich daher für T_{I_S} der Wert $\frac{150}{360} \approx 0.42$ Sekunden. Analog erhält man bei der Kipp–Bewegung für T_{I_S} den Wert $\frac{120}{270} \approx 0.44$ Sekunden.

6.3.2 Roboter–Kamera

Die Roboter–Kamera ist auf dem letzten Glied eines Roboters vom Typ SCORBOT–ER VII montiert (Bild 6.6). Es werden nur die Achsen 4 und 5 zur Steuerung verwendet. Dabei erreicht man genau, wie bei der Canon–Kamera eine Schwenk–/Kipp–Funktionalität. Die Ansteuerung des Roboters über die RS232-Schnittstelle kann wegen

der beschränkten Bandbreite nur jeweils in Abständen von 0.27 Sekunden erfolgen. Dies bedeutet, daß zur Regelung nicht alle von der Objektverfolgung übermittelten Positionsangaben verwendet werden. Die Öffnungswinkel α_O der Roboter–Kamera betragen horizontal 11.0 Grad und vertikal 8.3 Grad. Wie auch in Kapitel 6.3.1 werden für die Motoren P–Übertragungsglieder angenommen. Die Ansteuerung des Roboters wurde so eingestellt, daß die Konstanten T_{I_S} der Roboter–Kamera gerade denen der Canon–Kamera entsprechen. Somit schwenkt die Kamera bei Eingabegröße Eins in $T_{I_S} \approx 0.42$ Sekunden um eine halbe Bildbreite in horizontaler Richtung. Für die vertikale Richtung wurde entsprechend der Canon–Kamera $T_{I_S} \approx 0.44$ gewählt. Da somit die Zeitkonstanten der Canon– und der Roboter–Kamera identisch sind, können für beide Systeme die gleichen Einstellungen der Regler verwendet werden.

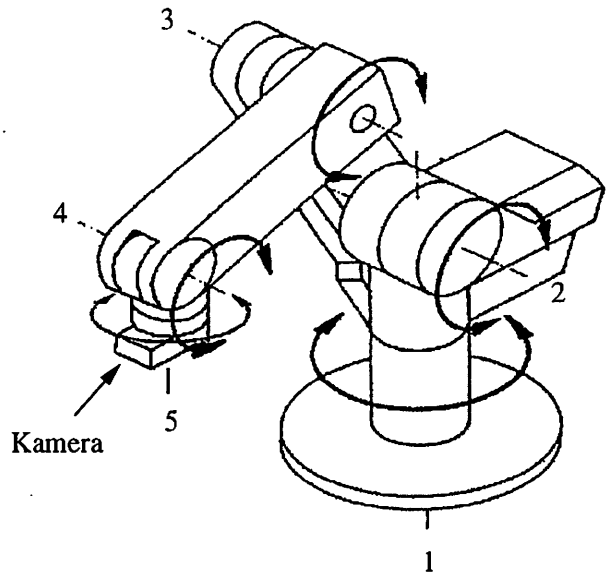


Bild 6.6: Roboter SCORBOT–ER VII

6.4 Einstellen der linearen Regler

Im folgenden werden nun auf theoretischem Wege lineare Regler für die in Kapitel 6.1 beschriebene Regelstrecke entworfen, die dann anschließend mit den ermittelten Parametern getestet werden. Wenn nicht anders angegeben, wird bei Simulationen und Messungen grundsätzlich die horizontale Bewegung betrachtet.

6.4.1 P–Regler

Bei einem P–Regler wird die Regelabweichung durch einen Verstärkungsfaktor K_R gewichtet zur Regelstrecke zurückgeführt (siehe Kapitel 3.2.2). Somit muß nur ein Parameter optimiert werden. Die s–Übertragungsfunktion eines P–Reglers lautet:

$$G_R(s) = K_R \tag{6.9}$$

Daraus läßt sich nun mit (6.8) die Übertragungsfunktion des geschlossenen Regelkreises mit P–Regler berechnen:

$$G_Z(s) = \frac{1}{1 + K_R \frac{1}{T_{I_S} s}} = \frac{T_{I_S} s}{T_{I_S} s + K_R} \tag{6.10}$$

Zunächst wird das statische Verhalten dieses Regelkreises näher untersucht, d.h es wird der sich für $t \to \infty$ einstellende Wert $e(\infty)$ der Objektposition relativ zum Nullpunkt betrachtet. Als Eingangssignale werden die Sprungfunktion sowie die Rampenfunktion verwendet. Die Sprungfunktion bedeutet anschaulich, daß sich die absolute Position des Objekts von einem Abtastschritt zum anderen um eine Einheit ändert, was nach der Normierung gerade einer halben Bildbreite entspricht. In der Realität kann dieser Fall dann auftreten, wenn das momentan verfolgte Objekt verloren, und an anderer Stelle ein neues Objekt gefunden wird. Die Rampenfunktion beschreibt eine Bewegung des Objekts mit konstanter Winkelgeschwindigkeit. Sie ist dazu geeignet, das Verhalten des Regelkreises bei einer kontinuierlichen Bewegung des Objektes zu untersuchen.

Mit Hilfe des Endwertsatzes der Laplace–Transformation (siehe Kapitel 3.1.2) ist es nun leicht möglich, die bleibende Regelabweichung für $t \to \infty$ zu bestimmen. Dazu werden die Laplace–transformierten Eingangssignale benötigt (Tabelle 6.1).

Eingangssignal	**Zeitbereich**	**Laplace–Transformierte**
Sprung	1	$\frac{1}{s}$
Rampe	t	$\frac{1}{s^2}$

Tabelle 6.1: Verschiedene Eingangssignale

Mit (6.3) ergibt sich für den Fehler $E_{Sprung}(s)$ des geschlossenen Regelkreises mit der Sprungfunktion $Z(s) = \frac{1}{s}$ folgender Ausdruck:

$$E_{Sprung}(s) = W(s) + Y(s) = 0 + \frac{T_{I_S} s}{T_{I_S} s + K_R} \frac{1}{s} = \frac{T_{I_S}}{T_{I_S} s + K_R} \tag{6.11}$$

Den statischen Wert für $e_{Sprung}(\infty)$ erhält man durch die Anwendung des Endwertsatzes:

$$\lim_{t\to\infty}(e_{Sprung}(t)) = \lim_{s\to 0}(sE_{Sprung}(s)) = \lim_{s\to 0}\left[\frac{T_{I_S}s}{T_{I_S}s + K_R}\right] = 0 \tag{6.12}$$

Es entsteht also mit Einsatz eines P–Reglers bei einer sprungförmigen Bewegung des Objekts keine bleibende relative Abweichung vom Nullpunkt; das Objekt kann also wieder ins Zentrum der Kamera gerückt werden. Anders sieht es für die in der Praxis wohl wichtigere Rampenfunktion aus, die eine kontinuierliche Bewegung des Objekts beschreibt:

$$\lim_{t\to\infty}(e_{Rampe}(t)) = \lim_{s\to 0}(sE_{Rampe}(s)) = \lim_{s\to 0}\left[\frac{T_{I_S}s}{T_{I_S}s^2 + K_R s}\right] = \frac{T_{I_S}}{K_R} \tag{6.13}$$

Es ergibt sich also eine bleibende Regelabweichung, die von den Konstanten T_{I_S} und K_R abhängt. Durch Erhöhen von K_R könnte diese Abweichung jedoch theoretisch sehr klein gehalten werden. Leider ist es in der Praxis nicht möglich, den Verstärkungsfaktor K_R beliebig zu erhöhen, weil zum einen die Stellgröße nur endlich groß werden kann, und zum anderen ein Abtastsystem vorliegt.

In [Ise88] wird darauf hingewiesen, daß die Ergebnisse eines analogen Reglerentwurfs auch auf digitale Regelungssysteme übertragen werden können, wenn mindestens gilt:

$$\frac{T_0}{T_{95}} \approx \frac{1}{15} \cdots \frac{1}{4} \tag{6.14}$$

Hier sind T_0 die Abtastzeit und T_{95} die Einschwingdauer der Übergangsfunktion bis auf 95% ihres Endwertes. Allgemein wird im weiteren die Zeit T_{100-p} als die Zeit bezeichnet, bis $(100 - p)\%$ des Endwertes bei sprungförmiger Anregung erreicht ist. Um die Übergangsfunktion zu erhalten, muß die inverse Laplace–Transformation für $E_{Sprung}(s)$ berechnet werden. Man erhält so folgende Lösung[1]:

$$\mathcal{L}^{-1}\{G_{Sprung}(s)\} = \mathcal{L}^{-1}\left\{\frac{T_{I_S}}{T_{I_S}s + K_R}\right\} = e_{Sprung}(t) = e^{-\frac{K_R t}{T_{I_S}}} \tag{6.15}$$

Es soll nun K_R so gewählt werden, daß $e_{Sprung}(T_{100-p}) = p/100$ gilt, wobei p die Abweichung in Prozent nach T_{100-p} Sekunden ist. Nach K_R aufgelöst erhält man dann:

$$K_R = -\frac{ln(p/100)T_{I_S}}{T_{100-p}} \tag{6.16}$$

Aus (6.14) und (6.16) wird ersichtlich, daß K_R von T_0, T_{I_S} sowie der gewünschten Anzahl von Abtastungen bis zur Zeit T_{100-p} abhängt. Praktisch erhält man also K_R dadurch, daß

[1] Verwendeter Maple-Aufruf siehe Anhang B

man T_{100-p} festlegt, da T_0 und T_{I_S} durch die Regelstrecke selbst bestimmt sind.

Zum Abschluß soll nun noch kurz auf die Stabilität des geschlossenen Regelkreises mit P–Regler eingegangen werden (siehe Kapitel 3.1.4). Aus (6.10) ergibt sich ein reeller Pol bei $s = -\frac{K_R}{T_{I_S}}$. Für positive K_R befindet er sich also in der linken s–Halbebene. Somit ist der geschlossene Regelkreis für $K_R > 0$ stabil.

Einstellwerte für die P–Regler

In Versuchen hat sich herausgestellt, daß das Verhältnis von T_0 zu T_{95} noch kleiner gewählt werden sollte, als von [Ise88] vorgeschlagen. Erst für $\frac{T_0}{T_{95}} \leq \frac{1}{30}$ konnten zufriedenstellende Ergebnisse erzielt werden, was wohl damit zusammenhängt, daß die Objektverfolgung auf zu große Sprünge zwischen zwei Bildern sehr sensibel reagiert. Schließlich wurde für $\frac{T_0}{T_{95}}$ das Verhältnis $\frac{1}{35}$ gewählt. Daraus resultiert zusammen mit den in Kapitel 6.3.1 ermittelten Zeitkonstanten $T_{I_S} = 0.42$ und $T_0 = 0.14$ unter Verwendung von Gleichung (6.16) der Verstärkungsfaktor K_R für die Schwenk–Bewegung:

$$K_R = -\frac{ln(0.05)T_{I_S}}{35T_0} \approx 0.26 \tag{6.17}$$

Analog erhält man bei der vertikalen Regelung für K_R den Wert 0.27.

Die kontinuierliche Bewegung des Objekts mit festen Winkelgeschwindigkeiten wird nun durch rampenförmige Eingangssignale mit unterschiedlichen Steigungen simuliert. Dabei bedeutet eine Geschwindigkeit mit Wert Eins, daß sich das Objekt in einer Sekunde um den Winkel α_O (siehe Bild 6.1) bewegt. In der Testumgebung erreicht die Lok bei Maximalgeschwindigkeit auf der hinteren Geraden der Eisenbahnstrecke eine Winkelgeschwindigkeit von etwa $\frac{1}{24}$, bewegt sich also in 24 Sekunden um den Winkel α_O. Auf der vorderen Geraden wird eine Geschwindigkeit von ca. $\frac{1}{13}$ erreicht. Für die bleibende Regelabweichung in Schwenk–Richtung bei rampenförmiger Störung ergeben sich mit den Objektgeschwindigkeiten von $\frac{1}{24}$ bzw. $\frac{1}{13}$ die Werte 0.067 und 0.124, indem die mit Formel (6.13) berechneten Ergebnisse noch durch 24 bzw. 13 dividiert werden. Bild 6.7 und Bild 6.8 zeigen die simulierten[2] Verläufe der Regelabweichung für sprungförmige bzw. rampenförmige Störung. Man sieht hier, daß beim Sprung 95% der Regelabweichung gerade nach $T_{95} = 35T_0 = 4.9$ Sekunden beseitigt werden. Bei der rampenförmigen Anregung kann die oben ermittelte bleibende Regelabweichung beobachtet werden.

Betrachtet man nun das reale Verhalten der Regelabweichung für die horizontale Bewegung während zweier Umläufe der Eisenbahn (Bild 6.9), so stellt sich auch tatsächlich annähernd der theoretisch berechnete Fehler ein. In Bild 6.9 ist an der x-Achse die normierte Regelabweichung und an der y-Achse die Zeit in Sekunden angetragen. Die größere

[2]Verwendete Matlab-Aufrufe siehe Anhang B

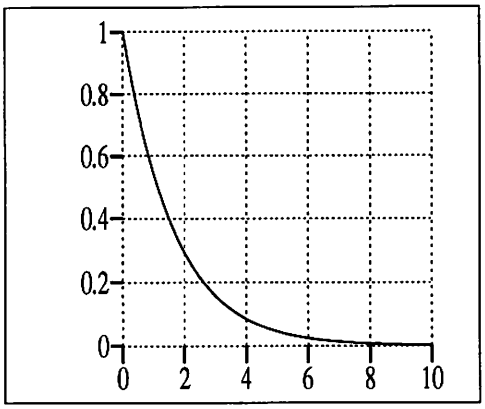


Bild 6.7: Regelabweichung mit P–Regler bei Sprungfunktion mit Höhe 1

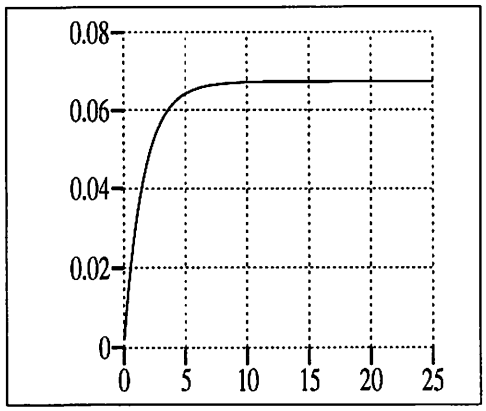


Bild 6.8: Regelabweichung mit P–Regler bei Rampe mit $\frac{1}{24}$ Steigung

Regelabweichung tritt dann auf, wenn sich die Lok auf der vorderen Geraden der Eisenbahnstrecke, also näher an der Kamera befindet, da dort ihre Winkelgeschwindigkeit höher ist als in weiterer Entfernung.

Variiert man den Verstärkungsfaktor K_R, so sieht man, daß die bleibende Regelabweichung indirekt proportional zu K_R ist (Tabelle 6.2). Die beiden obigen Beobachtungen können als Indiz dafür gewertet werden, daß das gewählte theoretische Modell mit der tatsächlichen realen Regelstrecke weitgehend übereinstimmt.

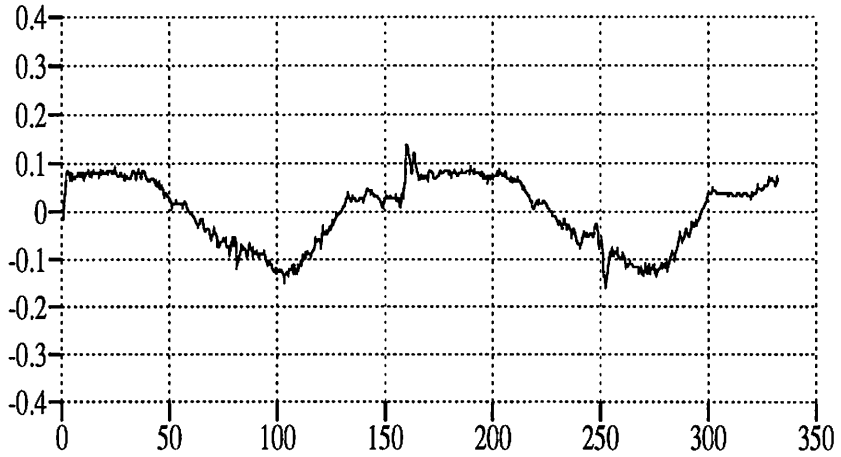


Bild 6.9: Regelabweichung mit Roboter–Kamera (P–Regler) während zweier Umläufe

Bild 6.10 zeigt den gemessenen und den simulierten Verlauf der Regelabweichung mit $K_R = 0.2$ und $T_{I_S} = 0.42$ unter Verwendung der Canon–Kamera. Der reale Verlauf konnte ermittelt werden, indem die Objektverfolgung gestartet wurde, als sich die Eisenbahn etwa $0.9X_{Res}$ Pixel rechts vom Bildmittelpunkt befand, und sich mit Geschwindigkeit $\frac{1}{24}$ nach rechts bewegte. Diese Bewegung wurde näherungsweise durch einen rampenförmigen Verlauf der Objektposition mit Startwert 0.9 und Steigung $\frac{1}{24}$ beschrieben. Für die initiale

K_R	**0.05**	**0.1**	**0.2**	**0.3**
Berechnet: $\frac{1}{24}$	0.35	0.18	0.09	0.06
Gemessen: $\frac{1}{24}$	0.39	0.20	0.10	0.07
Berechnet: $\frac{1}{13}$	0.65	0.32	0.16	0.11
Gemessen: $\frac{1}{13}$	0.79	0.38	0.19	0.12

Tabelle 6.2: Abweichungen in Schwenk–Richtung mit Canon–Kamera

Kameraposition wurde der Wert Null angenommen.

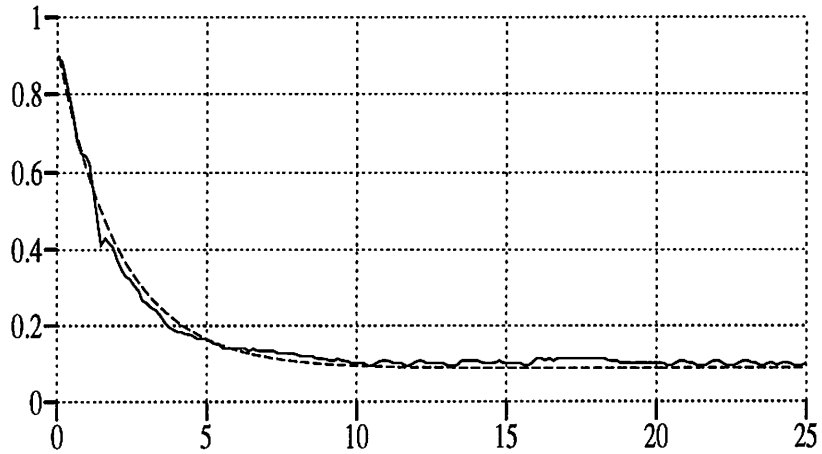


Bild 6.10: Realer und simulierter Verlauf der Regelabweichung mit P–Regler

Wie man sieht, entsprechen sich der gemessene Verlauf der Regelabweichung und die Simulation mit dem angenommen Modell weitgehend, was ein weiterer Beweis für die Korrektheit des Modells ist.

Der geschlossene Regelkreis ist für die gewählten $K_R = 0.26$ bzw. $K_R = 0.27$ stabil, da sich jeweils ein negativer reeller Pol bei $s \approx -0.6$ ergibt.

6.4.2 D– und I–Regler

Verwendet man statt des P–Reglers nun einen reinen D– oder I–Regler, erreicht man kein zufriedenstellendes Ergebnis. Da ein D–Glied eine Differentiation der Eingangsgröße vornimmt, reagiert es nur auf Änderungen am Reglereingang. Also gleicht ein D–Regler einen konstanten Fehler nicht aus, womit er für den Einsatz in dem gegebenen Regelkreis offensichtlich nicht geeignet ist und daher auch nicht weiter untersucht werden soll.

Um das Verhalten der Regelstrecke mit I–Regler zu analysieren, wird nun wieder die

Übertragungsfunktion dieses Gliedes betrachtet:

$$G_R(s) = \frac{1}{sT_{I_R}} \tag{6.18}$$

Für den geschlossenen Regelkreis erhält man dann mit (6.8):

$$G_Z(s) = \frac{s^2 T_{I_S} T_{I_R}}{s^2 T_{I_S} T_{I_R} + 1} \tag{6.19}$$

Man sieht leicht, daß das System zwei Pole auf der imaginären Achse aufweist, nämlich bei $\pm i \frac{1}{\sqrt{T_{I_S} T_{I_R}}}$. Ein Regelkreis mit Polen auf der imaginären Achse wird als grenzstabil bezeichnet (Kapitel 3.1.4). Dabei ergeben sich sowohl bei sprungförmiger, als auch bei rampenförmiger Störgröße Dauerschwingungen. Je kleiner die Zeitkonstante T_{I_R} gewählt wird, desto höher ist die Frequenz der Schwingung und desto kleiner ist bei rampenförmiger Erregung ihre Amplitude. Natürlich kann ein solches Verhalten des Regelkreises nicht als akzeptabel betrachtet werden, wodurch sich auch der Einsatz eines reinen I–Reglers verbietet. Die Abbildungen 6.11 und 6.12 zeigen für zwei verschiedene T_{I_R} den simulierten Verlauf der Regelabweichung bei rampenförmiger Störung.

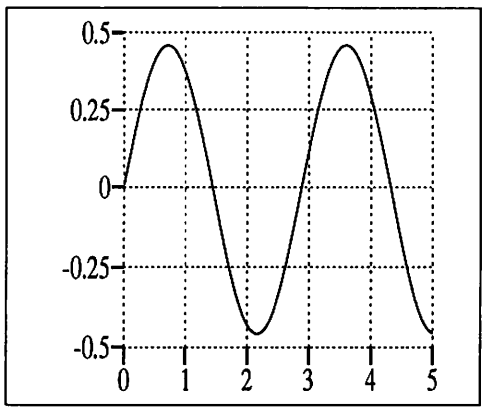


Bild 6.11: $T_{T_S} = 0.42$ und $T_{I_R} = 0.5$

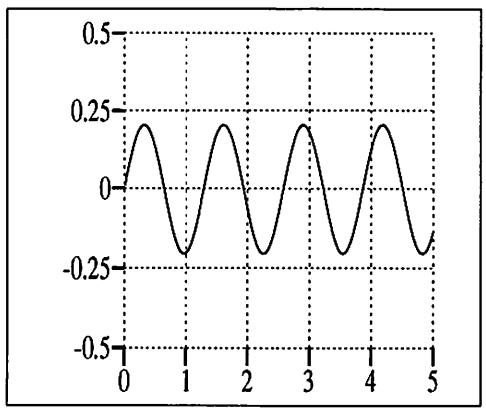


Bild 6.12: $T_{I_S} = 0.42$ und $T_{I_R} = 0.1$

6.4.3 PI–Regler

Zusätzlich zum Verstärkungsfaktor K_R besitzt ein PI–Regler noch ein Integrationsglied mit dem Parameter T_{I_R}, das die laufende Regelabweichung integriert (siehe Kapitel 3.2.3). Die s–Übertragungsfunktion des PI–Reglers lautet

$$G_R(s) = K_R \frac{1 + sT_{I_R}}{sT_{I_R}}. \tag{6.20}$$

Daraus ergibt sich analog zu Kapitel 6.4.1 die Übertragungsfunktion des geschlossenen Regelkreises:

$$G_Z(s) = \frac{1}{1 + \frac{1}{T_{I_S}s} K_R \frac{1+sT_{I_R}}{sT_{I_R}}} = \frac{s^2 T_{I_S}}{s^2 T_{I_S} + K_R s + \frac{K_R}{T_{I_R}}} \tag{6.21}$$

Das statische Verhalten der Regelstrecke mit einem PI–Regler bei einer Sprungfunktion als Störung entspricht dem mit einem P–Regler. Es entsteht keine bleibende Regelabweichung:

$$\lim_{t\to\infty} e_{Sprung}(t) = \lim_{s\to 0}(sE_{Sprung}(s)) = \lim_{s\to 0}\left[\frac{s^2 T_{I_S}}{s^2 T_{I_S} + K_R s + \frac{K_R}{T_{I_R}}}\right] = 0 \tag{6.22}$$

Im Gegensatz zum P–Regler schafft es der PI–Regler darüber hinaus auch, die Regelabweichung bei einem rampenförmigen Eingangssignal zum Verschwinden zu bringen:

$$\lim_{t\to\infty}(e_{Rampe}(t)) = \lim_{s\to 0}(sE_{Rampe}(s)) = \lim_{s\to 0}\left[\frac{sT_{I_S}}{s^2 T_{I_S} + K_R s + \frac{K_R}{T_{I_R}}}\right] = 0 \tag{6.23}$$

Diese Erkenntnis läßt vermuten, daß sich der PI–Regler gut für den Einsatz in der gegebenen Regelstrecke eignet. Für die späteren Stabilitätsbetrachtungen müssen noch die Pole von $G_Z(s)$ ermittelt werden:

$$s_{1,2} = -\frac{1}{2}\left(\frac{K_R}{T_{I_S}} \pm \sqrt{\frac{{K_R}^2}{{T_{I_S}}^2} - 4\frac{K_R}{T_{I_R}T_{I_S}}}\right) \tag{6.24}$$

Einstellwerte für die PI–Regler

Mit Hilfe von Simulationen und durch Testen der realen Regelstrecke mit den dadurch gewonnenen Parametern haben sich als Einstellungen der PI–Regler die Werte in Tabelle 6.3 als akzeptabel erwiesen. Dabei wurde für die Kipp–Bewegung ein größerer Wert für T_{I_R} gewählt, da in der Testumgebung nur relativ wenig vertikale Lageveränderungen des Objekts stattfinden, und somit ein weniger schnelles I–Glied ausreichend ist.

	K_R	T_{I_R}
pan:	0.15	7
tilt:	0.14	10

Tabelle 6.3: Einstellwerte für den PI–Regler

Die Bilder 6.13 und 6.14 zeigen die simulierten Ergebnisse für diese Einstellung des PI–

Reglers bei sprung- und rampenförmiger Erregung. Dem Bild 6.14 kann man entnehmen, daß mit Einsatz des PI-Reglers keine bleibende Regelabweichung bei rampenförmiger Eingangsgröße entsteht. Man sieht auch, daß der Regelkreis zu leichtem Überschwingen neigt, was durch größeres T_{I_R} vermieden werden könnte. Jedoch nimmt mit großen Werten von T_{I_R} vor allem bei rampenförmiger Störung auch die Zeit zu, bis die Regelabweichung beseitigt wird. Die gewählte Einstellung stellt damit einen Kompromiß dar, der ein gewisses Überschwingen bei möglichst schneller Beseitigung der Regelabweichung noch toleriert.

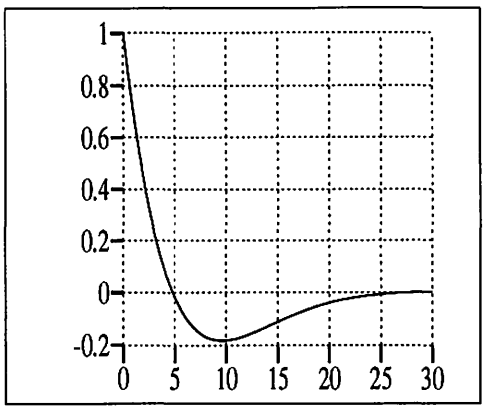


Bild 6.13: Regelabweichung mit PI-Regler bei Sprungfunktion mit Höhe 1

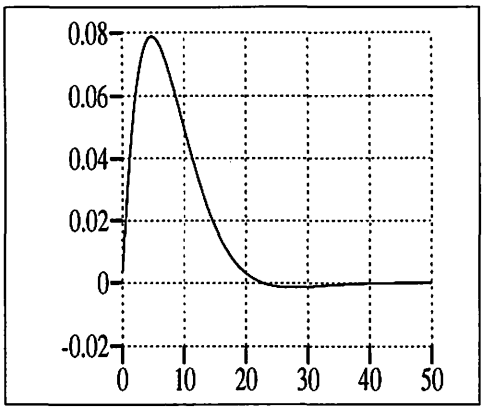


Bild 6.14: Regelabweichung mit PI-Regler bei Rampe mit $\frac{1}{24}$ Steigung

Das reale Verhalten der Regelstrecke mit PI-Regler bestätigt die simulierten Ergebnisse. In Bild 6.15 ist wieder der Verlauf der Regelabweichung während zweier Umläufe der Eisenbahn zu sehen. Man erkennt, daß die Abweichung vom Nullpunkt durch den Regler weitgehend kompensiert werden kann. Die großen Regelabweichungen nach etwa 60 und 240 Sekunden kommen zustande, weil das Objekt kurzzeitig von der Objektverfolgung verloren wurde. Des weiteren kann beobachtet werden, daß in den Kurven, wenn sich die Winkelgeschwindigkeit der Lok auf Null verringert, eine höhere Regelabweichung entsteht (im Bild nach ca. 60, 130, 240 und 300 Sekunden). Dies liegt an der Trägheit des I-Gliedes im PI-Regler, welches eine gewisse Zeit benötigt, um seinen Integralwert entsprechend der neuen Objektgeschwindigkeit anzupassen.

Der geschlossene Regelkreis mit dem verwendeten PI-Regler ist stabil, da wieder sämtliche Pole auf der linken s-Halbebene liegen. Mit Formel (6.24) ergeben sich für die Pole des geschlossenen Regelkreises die Werte $-0.18 \pm 0.14i$ (horizontale Bewegung) bzw. $-0.16 \pm 0.08i$ (vertikale Bewegung).

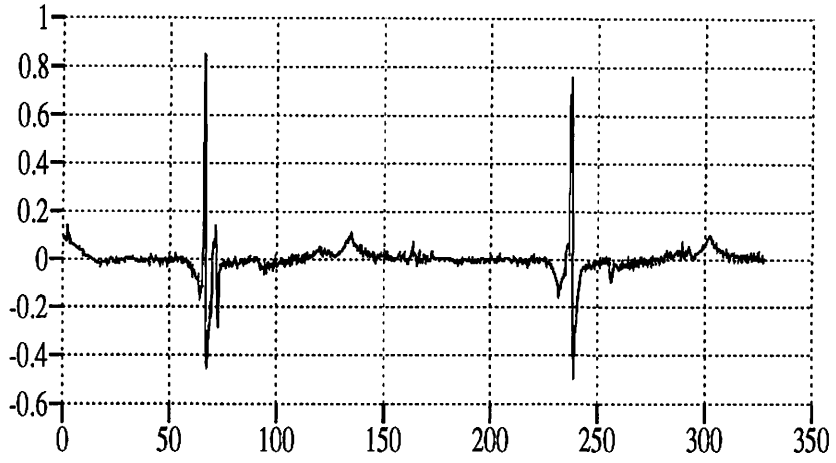


Bild 6.15: Regelabweichung mit Canon–Kamera (PI–Regler) während zweier Umläufe

6.4.4 PID–Regler

Als letzter linearer Regler wird der PID–Regler untersucht. Die Übertragungsfunktion dieses Reglers lautet:

$$G_R(s) = \frac{T_{D_R} K_R s^2 + K_R s + \frac{K_R}{T_{I_R}}}{s} \tag{6.25}$$

Für die Übertragungsfunktion des geschlossenen Regelkreises ergibt sich:

$$G_Z(s) = \frac{T_{I_S} s^2}{s^2(T_{I_S} + T_{D_R} K_R) + K_R s + \frac{K_R}{T_{I_R}}} \tag{6.26}$$

Wie schon der PI–Regler ist auch der PID–Regler in der Lage, die Regelabweichung bei sprungförmiger und trapezförmiger Anregung zu eliminieren. Darüber hinaus bietet sich noch die Möglichkeit, das Verhalten des Regelkreises bei Änderungen der Eingangswerte durch den Parameter T_{D_R} zu beeinflussen. Allerdings hat sich herausgestellt, daß T_{D_R} eher klein gewählt werden sollte, da sonst die Stellgrößen bei sprungförmiger Bewegung des Objekts zu groß werden. Für die Pole des geschlossenen Regelkreises mit PID–Regler ergibt sich:

$$\frac{-K_R \pm \sqrt{K_R^2 - 4K_R \frac{T_{I_S} + T_D K_R}{T_{I_R}}}}{2(T_{I_S} + T_D K_R)} \tag{6.27}$$

Einstellwerte für die PID–Regler

Als praktikable Werte für die Parameter der PID–Regler wurden die in Tabelle 6.4 aufgeführten Konstanten gefunden.

Aufgrund des niedrigen Wertes von T_{D_R} weicht das reale Verhalten des geschlossenen Regelkreises mit PID–Regler nicht sehr von dem mit einem PI–Regler ab. Es kann deshalb

	K_R	T_{I_R}	T_{D_R}
pan:	0.15	10	0.05
tilt:	0.13	12	0.05

Tabelle 6.4: Einstellwerte für die PID–Regler

auch ein sehr ähnlicher realer Verlauf der Regelabweichung (Bild 6.16) beobachtet werden.

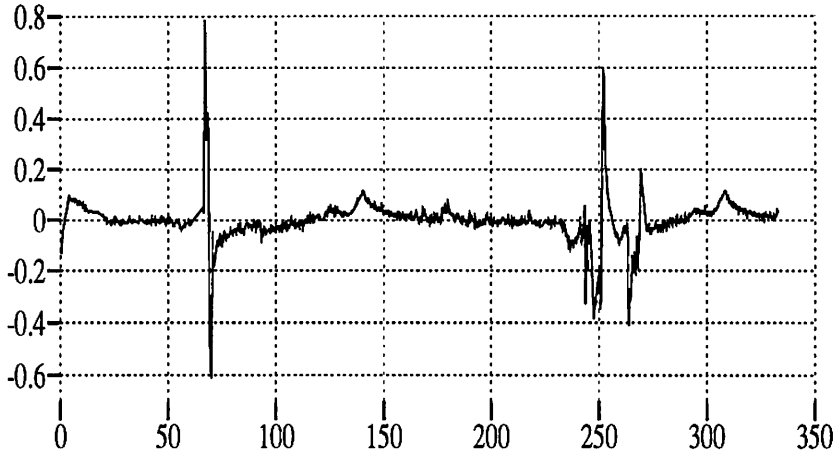


Bild 6.16: Regelabweichung mit Canon–Kamera (PID–Regler) während zweier Umläufe

Bei einer Sprungfunktion als Eingangsgröße ist beim PID–Regler ein minimal geringeres Überschwingen als beim vorher eingestellten PI–Regler zu beobachten, da hier das D–Glied zu Beginn eine hohe Stellgröße bewirkt. Dagegen entsteht bei rampenförmiger Anregung in den ersten Sekunden eine etwas größere Regelabweichung. Für die gewählten Einstell–Werte ergeben sich mit Formel (6.27) Pole bei $-0.18 \pm 0.07i$ (pan) und bei $-0.15 \pm 0.06i$ (tilt). Der Regelkreis weist also auch mit PID–Regler stabiles Verhalten auf.

An dieser Stelle soll noch kurz auf den implementierten Zustandsregler *RsZustRegler* (siehe 5.3.4) eingegangen werden. Verwendet man die Zustandsraumdarstellung des PID–Reglers in Regelungsnormalform (siehe [Ise88] und Kapitel 3.1.5), so erhält man folgende Matrizen:

$$\boldsymbol{A}_{pan} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}, \quad \boldsymbol{B}_{pan} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \boldsymbol{C}_{pan} = \begin{bmatrix} 0.05357 & -0.05147 \end{bmatrix}, \quad \boldsymbol{D}_{pan} = 0.20357$$

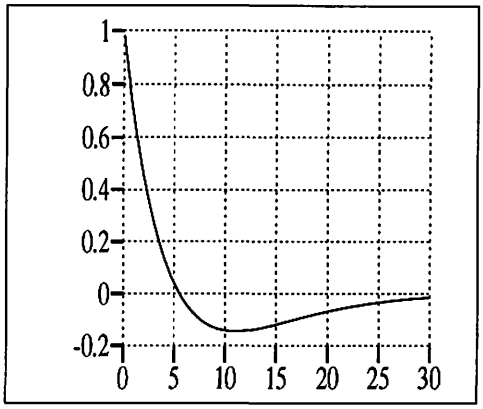


Bild 6.17: Regelabweichung mit PID–Regler bei Sprung mit Höhe 1

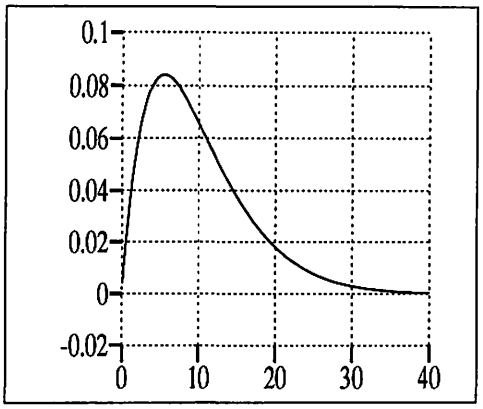


Bild 6.18: Regelabweichung mit PID–Regler bei Rampe mit $\frac{1}{24}$ Steigung

$$A_{tilt} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}, \quad B_{tilt} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad C_{tilt} = \begin{bmatrix} 0.04642 & -0.04491 \end{bmatrix}, \quad D_{tilt} = 0.17642$$

Das Regelverhalten dieses Zustandsreglers ist dabei vollkommen identisch mit dem des oben vorgestellten Eingangs–/Ausgangs–Reglers (PID–Regler).

6.5 Einstellen des Fuzzy–Reglers

Im Gegensatz zu den linearen Reglern kann bei der Einstellung des Fuzzy–Reglers kein mathematischer Ansatz gewählt werden, da das Übertragungsverhalten von Fuzzy–Reglern nicht geschlossen mathematisch darstellbar ist.

Es wird wie in den vorhergehenden Kapiteln jeweils ein Regler für die Schwenk– und einer für die Kipp–Bewegung der Kameras verwendet. Für die Eingänge der Regler werden zwei linguistische Variablen gewählt, nämlich die auf den Bereich [-1..1] normierte Abweichung vom Bildmittelpunkt sowie die Summe der durch einen konstanten Faktor gewichteten vergangenen Abweichungen. Durch entsprechende Wahl des Faktors kann der Einfluß der vorhergehenden Abweichungen auf die Stellgröße erhöht bzw. erniedrigt werden. Der Ausgang des Fuzzy–Reglers besitzt eine linguistische Variable mit Wertebereich [-1..1] zur Ansteuerung der Kamera. Durch diese Anordnung wird ein ähnliches Verhalten, wie beim PI–Regler erreicht. Die linguistischen Variablen für die Eingabe sollen im folgenden mit *Fehler* bzw. *Fehler–Summe* bezeichnet werden; für die Ausgabe wird die Bezeichnung *Geschwindigkeit* verwendet.

Es werden ausschließlich normalisierte dreiecks– und trapezförmige Fuzzy–Mengen

eingesetzt. Für die jeweils in den linguistischen Variablen außenliegenden Fuzzy-Mengen wird eine Trapezform, für die innenliegenden eine Dreiecksform gewählt. Die linguistische Variable *Fehler* besitzt 7, die *Fehler-Summe* 5 Fuzzy-Mengen. Für die *Geschwindigkeit* werden 7 Fuzzy-Mengen verwendet. Der Inferenzprozeß (siehe Kapitel 4.2.2) arbeitet mit den in Tabelle 6.5 aufgeführten Fuzzy-Operatoren.

aggOp	kompOp	infOp	accOp
Algebr. Prod.	Algebr. Summe	Minimum-Operator	Maximum-Operator

Tabelle 6.5: Verwendete Fuzzy-Operatoren

Tabelle 6.6 zeigt die für den Fuzzy-Regler verwendete Regelbasis, wobei die Abkürzungen NB (Negative Big), NM (Negative Medium), NS (Negative Small), ZO (Zero), PS (Positive Small) usw. benutzt werden. Es wird hier für beide Bewegungsrichtungen der Kamera die gleiche Regelbasis eingesetzt. Nach rechts ist dabei der *Fehler* und nach unten die *Fehler-Summe* angetragen. Die Werte in der Tabelle zeigen die Schlußfolgerungen für die *Geschwindigkeit* bei bestimmtem *Fehler* und bestimmter *Fehler-Summe*. Eine Schlußfolgerung wäre zum Beispiel: „**WENN** *Fehler* = NM **UND** *Fehler-Summe* = NB **DANN** *Geschwindigkeit* = NB". Die Aufstellung der Regeln erfolgte nach rein empirischen Gesichtspunkten.

	NB	**NM**	**NS**	**ZO**	**PS**	**PM**	**PB**
NB	NB	NB	NM	NS	PS	PS	PM
NS	NB	NM	NS	NS	PS	PS	PM
ZO	NB	NM	NS	ZO	PS	PM	PB
PS	NM	NS	NS	PS	PS	PM	PB
PB	NM	NS	NS	PS	PM	PB	PB

Tabelle 6.6: Regelbasis des Fuzzy-Reglers

Unter Verwendung der Klasse *RsFuzzyERegler* wurden durch systematisches Verändern von Support, Toleranz und Neigung der Fuzzy-Mengen in den einzelnen linguistischen Variablen geeignete Einstellungen für den Regler gefunden (Bild 6.19).

Für die Schwenkrichtung ergibt sich das in Bild 6.20 dargestellte Übertragungsverhalten des Reglers. Die Funktionswerte repräsentieren die Kameraschwindigkeit bei bestimmtem *Fehler* und bestimmter *Fehler-Summe*. Man sieht, daß die Steigung der Funktion für niedrige Werte von *Fehler* oder *Fehler-Summe* größer ist als für weiter außerhalb liegende Werte. Dadurch entsteht für kleine Regelabweichungen eine höhere Stellgröße

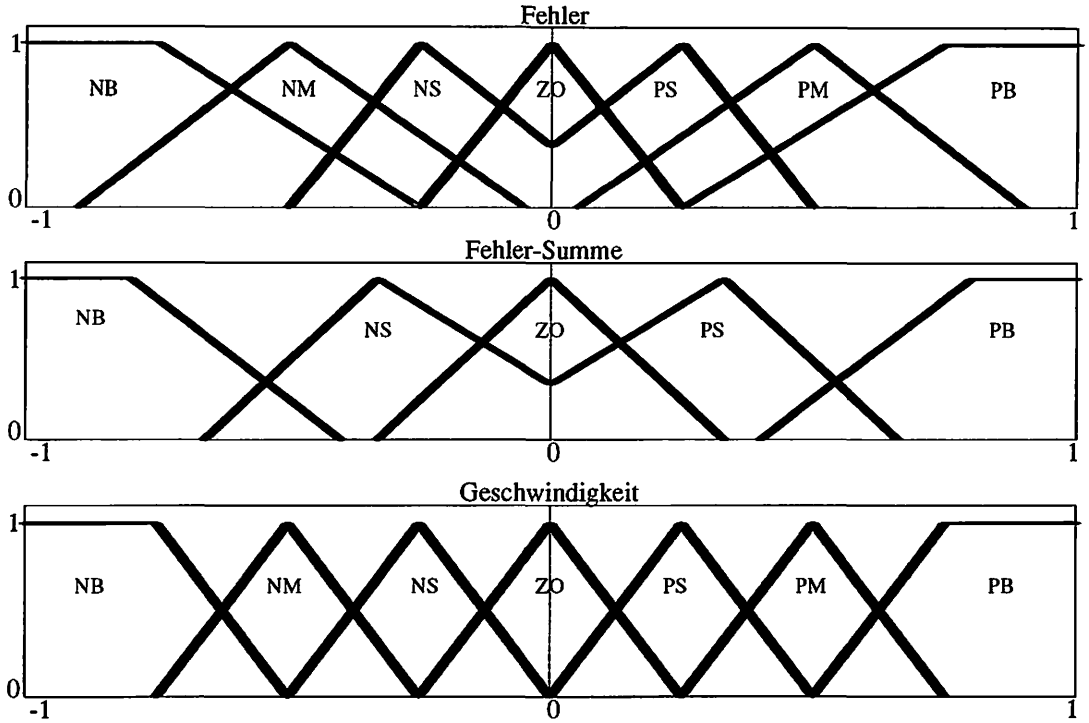


Bild 6.19: Linguistische Variablen für Fuzzy–Regler (Schwenkrichtung)

als im linearen Fall, wodurch der Regler schneller auf Regelabweichungen reagiert und entsprechend höhere Stellgrößen erzeugt. Die Einstellung des Reglers für die vertikale Bewegung ist ähnlich zu der bei der Schwenkrichtung, wobei allerdings die oben angesprochenen Steigungen etwas geringer ausfallen.

In Bild 6.21 ist der Verlauf der Regelabweichung für die Schwenkbewegung der Canon–Kamera während zweier Umläufe der Eisenbahn zu sehen. Für den konstanten Faktor, mit dem die vergangenen Regelabweichungen bei Bildung der *Fehler–Summe* multipliziert werden, wird dabei für beide Bewegungsrichtungen der Wert 0.15 verwendet. Es kann wie auch schon beim linearen PI–Regler (siehe Kapitel 6.4.3) die Trägheit des Reglers an den Wendepunkten der Objektbewegung beobachtet werden. Dieser Effekt ist besonders gut nach etwa 60 bzw. 130 Sekunden zu sehen. Über die Stabilität des Regelkreises mit Fuzzy–Regler kann keine mathematische Aussage getroffen werden, da, wie schon oben erwähnt, kein entsprechendes Modell für den Regler vorliegt.

6.6 Speicher– und Laufzeitverhalten der Regler

Der Speicherbedarf und das Laufzeitverhalten der verschiedenen Regler wurde auf einem Rechner von *Hewlett Packard* vom Typ *HP700* gemessen. Dieser Computer besitzt eine Rechenleistung von 124 Mips. Zum Compilieren der Klassen und der Testprogramme wurde ein C++–Compiler von AT&T verwendet. In Tabelle 6.7 sieht man die CPU–Zeiten

der einzelnen Regler für 1000 Regelungen, sowie den Speicherverbrauch eines Programms mit nur einem Regler–Objekt. Die Laufzeiten der linearen Regler sind sehr gering und fallen praktisch überhaupt nicht ins Gewicht. Auch die Regelzeiten des in Kapitel 6.5 vorgestellten Fuzzy–Reglers liegen weit unter den in der Objektverfolgung erreichten Abtastzeiten. Der Speicherbedarf aller Regler ist konstant und im Verhältnis zu den anderen Komponenten wie z.B. den `pvm`–Funktionen vernachlässigbar gering. Die etwas höheren Werte bei Regelzeit und Speicherbedarf des PI–Reglers im Vergleich zum PID–Regler kommen daher, daß das PI–Glied durch Parallelschaltung eines P– und eines I–Reglers mit Hilfe der Klasse *RsParallel* erzeugt wurde. Der PID–Zustandsregler benötigt aufgrund der verwendeten Matrixklassen etwas mehr Ressourcen als die übrigen linearen Regler.

	P	**PI**	**PID**	**PID–Zust.**	**Fuzzy–R.**
CPU–Zeit (1000 Regel.)	2.5ms	38ms	16ms	191ms	5.34s
Speicherbedarf	392KB	400KB	392KB	400KB	532KB

Tabelle 6.7: Speicher– und Laufzeitverhalten der Regler

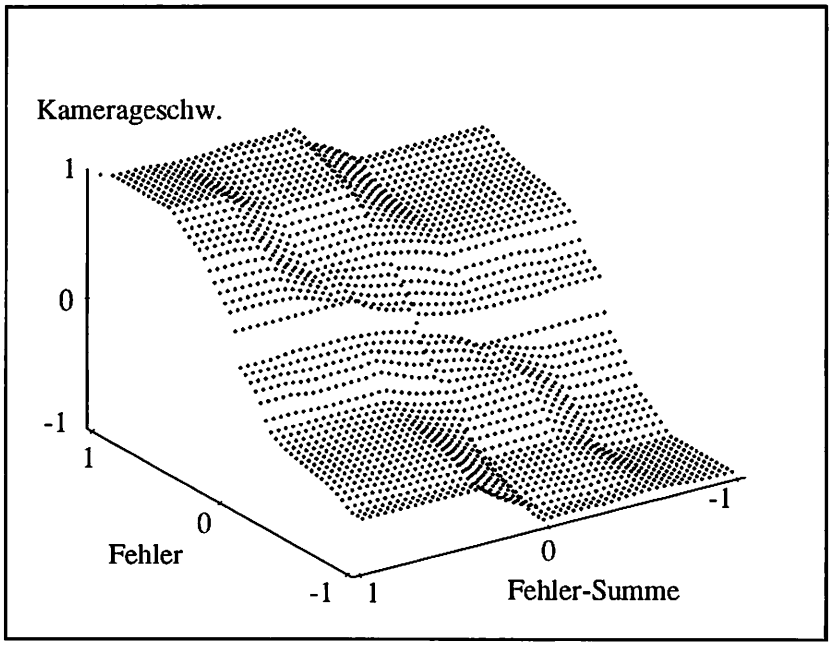


Bild 6.20: Übertragungsverhalten des Fuzzy–Reglers für Schwenk–Bewegung

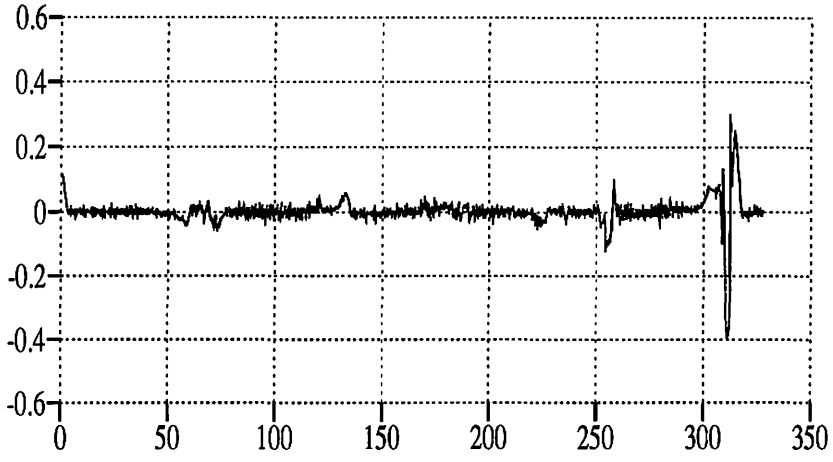


Bild 6.21: Regelabweichung mit Canon–Kamera (Fuzzy–Regler) während zweier Umläufe

6.7 Vergleich der Regler

Bei den linearen Reglern konnte mit dem PID–Regler das beste Regelergebnis erzielt werden. Durch den Integral–Anteil ist dieser Reglertyp in der Lage, bei Objekten, die sich mit konstanter Geschwindigkeit bewegen, die Fehlerabweichung zu eliminieren. Hierbei könnte man von einer impliziten Prädiktion sprechen, da sich der Integralwert so anpaßt, daß die Kamera auf die zu erwartende Objektposition im nächsten Bild bewegt wird, sofern sich die Winkelgeschwindigkeit des Objekts nicht ändert. Der D–Anteil bewirkt, daß der PID–Regler schneller auf Geschwindigkeitsänderungen des Objekts reagiert als ein PI–Regler, was sich vor allem bei höheren Objektgeschwindigkeiten positiv auswirkt. Allerdings fällt aufgrund der sehr klein gewählten Konstante T_D der Unterschied im Regelverhalten zwischen PI–Regler und PID–Regler relativ gering aus.

Der P–Regler besitzt den Nachteil, daß er bei kontinuierlicher Bewegung des Objekts die Regelabweichung nicht eliminieren kann. Er eignet sich deshalb weniger gut für den Einsatz in der Objektverfolgung. D– und I–Regler sind gänzlich ungeeignet für die gegebene Regelstrecke. Mit ihnen kann kein zufriedenstellendes Regelverhalten erreicht werden.

Der verwendete Fuzzy–Regler weist ähnliches Verhalten wie der PI–Regler auf, reagiert allerdings bei kleinen Regelabweichungen etwas rascher als der lineare Regler. Das beim PI– und PID–Regler zu beobachtende Überschwingen bei Änderungen der Bewegungsrichtung des Objekts in den Kurven ist beim Fuzzy–Regler etwas geringer. Der Entwurf des Fuzzy–Reglers basiert auf rein empirischem Vorgehen. Aufgrund des fehlenden mathematischen Modells kann kein analytischer Stabilitätsnachweis für den Regelkreis mit Fuzzy–Regler geführt werden. In der realen Regelstrecke erwies sich die Regelung aber als stabil.

Kapitel 7

Ausblick

Die in dieser Studienarbeit realisierte Klassenhierarchie für Regler kann aufgrund der Eigenschaften der verwendeten objektorientierten Programmiersprache C++ leicht um weitere Reglerklassen erweitert werden.

In einer Fortsetzung dieser Arbeit könnte beim betrachteten Fuzzy–Regler der Einsatz von andersgearteten Fuzzy–Mengen, wie z.B. parabelförmigen Mengen, untersucht werden. Darüber hinaus besteht die Möglichkeit, den Inferenzprozeß durch eine andere Wahl von Fuzzy–Operatoren zu beeinflussen, um dadurch unter Umständen ein verbessertes Regelverhalten zu erzielen. Allerdings sind obige Vorschläge aufgrund der notwendigen Versuche an der realen Regelstrecke nur recht zeitaufwendig zu realisieren. Außerdem ist zu erwarten, daß dabei das Regelverhalten nur wenig verbessert werden kann. Erfolgversprechender erscheint es, den Fuzzy–Regler mit mehr Eingangsgrößen zu betreiben. So ist es z.B. denkbar, als zusätzliche Eingänge die aktuelle Objektgeschwindigkeit oder die Objektbeschleunigung zu verwenden. Die Regelbasis wird bei mehr als drei Eingängen jedoch sehr unübersichtlich, wodurch die Aufstellung der Regeln stark erschwert wird.

Von den linearen Eingangs–/Ausgangs–Reglern wurden nur Regler bis zur Ordnung Zwei betrachtet. In weiterführenden Untersuchungen könnten auch lineare Regler höherer Ordnung getestet werden.

Zur Ansteuerung der Kameras wurden in dieser Arbeit jeweils zwei getrennte Regler verwendet. Einer für die Schwenk– und einer für die Kipp–Bewegung. Es wurde also immer nur der eindimensionale Fall einer Objektbewegung betrachtet. Durch Einsatz von Mehrgrößenregelung könnte die Steuerung der Kamera auf den zweidimensionalen Fall erweitert werden. Für diese Aufgabe würde sich die Verwendung von linearen Zustandsreglern anbieten. Darüber hinaus gibt es noch die Möglichkeit durch Einsatz von stochastischer Regelung, z.B. mit Hilfe eines Kalmanfilters durch Schätzung der linearen Modellzustände, die Prädiktion der Objektposition zu verbessern. Dabei wird durch eine gewichtete Mittelwertbildung zwischen der aktuellen Messung und des aufgrund der vorhergehenden Messungen ermittelten Zustandes der neue Zustand so geschätzt, daß die

Varianzen des Schätzfehlers minimiert werden. Durch entsprechende Gewichtung kann mehr oder weniger Vertrauen in die aktuelle Messung oder die Prädiktion gesetzt werden. Ansätze hierzu sind z.B. in [RG92] und [Ise87] nachzulesen.

Kapitel 8

Zusammenfassung

Bei der Objektverfolgung werden sowohl Methoden der Mustererkennung als auch der Regelungstechnik eingesetzt. Dabei besteht die Aufgabe des regelungstechnischen Teils darin, aufgrund von Informationen über die Objektbewegung eine Kamera möglichst optimal in Bezug auf ein anwendungsspezifisches Gütekriterium zu steuern. Für diesen Zweck werden Regler benötigt.

Für die Objektverfolgung existieren eine Reihe von möglichen Anwendungsgebieten, wie z.B. in Bildtelefonsystemen oder bei überwachungstechnischen Aufgaben. In den letzten Jahren wurde vor allem im Bereich der Robotik verstärkt Objektverfolgung eingesetzt, wodurch die Koordination von sich bewegenden Werkstücken und der Roboterbewegung verbessert werden konnte (siehe [FL90]).

Der Artikel [DN95] beschreibt ein System zur Echtzeit–Objektverfolgung, das am Lehrstuhl für Mustererkennung der Universität Erlangen–Nürnberg entwickelt wurde. Hierbei wurde bisher für die Ansteuerung der Kamera ein einfacher P–Regler verwendet. Zielsetzung dieser Studienarbeit ist es, die Kamerasteuerung zu erweitern und dafür verschiedene aus der Regelungstechnik bekannte Regler in einer Klassenhierarchie zu implementieren und miteinander zu vergleichen. Dabei soll die Kamera einem sich bewegenden Objekt so nachgeführt werden, daß dieses im Bildzentrum gehalten wird. Es werden hierzu eine Roboter–Kamera sowie eine Schwenk–/Kipp–Kamera verwendet. Die Abweichung des zu verfolgenden Objekts vom Bildmittelpunkt soll dabei minimiert werden.

Im Zentrum des Interesses bei der Regelungstechnik steht die Regelstrecke. Diese ist in der Realität meist ein technischer Prozeß oder eine Maschine. Die Struktur eines Regelkreises ist im wesentlichen durch diese Regelstrecke sowie durch den verwendeten Regler bestimmt. Durch die Rückführung von Ausgangsgrößen der Regelstrecke zum Regler wird der geschlossene Regelkreis erzeugt. Bei dem dadurch entstehenden System unterscheidet man zwischen seinem statischen und dynamischen Verhalten. Zudem wird zwischen kontinuierlichen und diskreten Systemen unterschieden. Dabei entstehen diskrete Systeme durch den Einsatz von Digitalrechnern zur Regelung.

Für lineare Systeme gibt es eine Reihe von Beschreibungsmöglichkeiten im Zeit- und Frequenzbereich. Zur Beschreibung im Zeitbereich kann die Gewichtsfunktion oder die Übergangsfunktion dienen. Im Frequenzbereich verwendet man zur Modellierung von linearen analogen Regelungssystemen meist s–Übertragungsfunktionen. Digitale Systeme können durch z–Übertragungsfunktionen beschrieben werden. Liegt das Modell eines Regelkreises in Form einer Übertragungsfunktion vor, so können durch Bestimmung von Lage und Art ihrer Polstellen Aussagen über seine Stabilität gemacht werden.

Bei linearen Eingangs–/Ausgangs–Reglern hängen die Ausgangsgrößen nur von den Eingangsgrößen sowie den Reglerparametern ab. Aus ihren z–Übertragungsfunktionen können direkt rekursive Regelalgorithmen abgeleitet werden, die zur Implementierung dieser Regler notwendig sind. Innerhalb dieser Arbeit wurde dabei der allgemeine Fall des Eingangs–/Ausgangs–Reglers sowie im speziellen die aus der analogen Regelungstechnik bekannten Regler wie P, I, D, PID usw. betrachtet. Dabei wurden die Parameter der analogen Regler, wie z.B. die Nachstellzeit T_I des I–Reglers, nach [Ise88] für den diskreten Fall umgerechnet.

Als nichtlinearer Regler wurde der Fuzzy–Regler betrachtet. Dieser basiert auf der Theorie der Fuzzy–Logik. Als Vorteil dieses Reglers gilt, daß zu seiner Einstellung für eine vorgegebene Regelstrecke kein mathematisches Modell notwendig ist, weshalb der Einstellaufwand im Vergleich zu den linearen Reglern meist geringer ausfällt. Aufgrund des fehlenden mathematischen Modells ist dafür aber kein analytischer Stabilitätsnachweis für den geschlossenen Regelkreis möglich. Die Hauptelemente eines Fuzzy–Reglers sind linguistische Variablen und die Regelbasis. Eine linguistische Variable besteht aus mehreren Fuzzy–Mengen. Diese sind definiert durch ihre charakteristischen Funktionen, die im Vergleich zur klassischen Mengentheorie auch andere Zugehörigkeitswerte außer Null oder Eins annehmen können. In der Literatur ([Tra94], [Til92], [HMB93]) finden sich verschiedene Operatoren, mit denen Fuzzy–Mengen miteinander verknüpft werden können. Dabei wurde in dieser Arbeit besonders auf T– und S–Normen eingegangen, die die beiden wichtigsten Gruppen von Fuzzy–Operatoren repräsentieren. Die Arbeitsweise eines Fuzzy–Reglers erfolgt in drei Schritten: Durch die Fuzzyfizierung werden scharfe Eingangswerte in Zugehörigkeitswerte für Fuzzy–Mengen umgewandelt. Diese werden dann im Inferenz–Prozeß zur Auswertung der Regelbasis verwendet, wobei eine Ergebnis–Fuzzy–Menge ausgegeben wird. Die Defuzzyfizierung erzeugt daraus wieder einen scharfen Wert, der die Stellgröße des Reglers ergibt.

Die Implementierung der Klassen erfolgte in der Programmiersprache C++, wobei die Klassenbibliothek NIHCL verwendet wurde. Als Wurzel aller realisierten Klassenhierarchien diente die Basisklasse von NIHCL. In der Regler–Klassenhierarchie wurden die verschiedenen Ansätze aus der linearen Regelungstechnik sowie aus der Fuzzy–Logik in einen Ableitungsbaum integriert. Alle dort implementierten Klassen besitzen eine Methode, die

den jeweiligen spezifischen Regelalgorithmus ausführt. Von den linearen Reglern wurden der allgemeine Eingangs-/Ausgangs-Regler, die P-, I-, D- und PID-Regler sowie ein Zustandsregler realisiert. Daneben existieren ein Fuzzy-Regler und Klassen, mit denen eine Parallel- oder Serienschaltung von Reglern erzeugt werden kann. Für die Bestandteile des Fuzzy-Reglers mußten weitere Klassenhierarchien von Fuzzy-Mengen, Fuzzy-Operatoren, linguistischen Variablen usw. implementiert werden.

Um geeignete Parameter der linearen Regler zu erhalten, wurde ein analytisches Modell für die Regelstrecke entworfen. Als Beschreibungsmöglichkeit der verschiedenen Systeme wurden dabei s-Übertragungsfunktionen verwendet. Durch das Modellieren der Motoren der Canon-Kamera und des Roboters als reine P-Glieder, konnte die gesamte Regelstrecke als I-Glied dargestellt werden. Der geschlossene Regelkreis wurde als Festwertregelung ausgelegt, wobei als Störgröße die Objektposition am Ausgang der Regelstrecke eingreift. Die Führungsgröße in Form der Abweichung des Objekts vom Bildmittelpunkt wurde dabei auf konstant Null gesetzt, da als Ziel das Verschwinden der Abweichung angestrebt wird. Geeignetes Einstellen der Kamerasteuerung von Canon- und Roboter-Kamera erlaubte es, für beide Systeme die gleichen Modellparameter zu verwenden. Dadurch mußten die Reglerparameter jeweils nur einmal ermittelt werden.

Für jeden Freiheitsgrad der Kameras, also der horizontalen Schwenk- und der vertikalen Kippbewegung, wurde jeweils getrennt ein Regler verwendet. Es wurde also immer nur der eindimensionale Fall einer Objektbewegung betrachtet. Das mathematische Modell der Regelstrecke ermöglichte es, für die verschiedenen linearen Regler Simulationen mit dem Software-Paket *Matlab* sowie Berechnungen für das statische Verhalten der verschiedenen Systeme durchzuführen. Bei der Betrachtung des P-Reglers im geschlossenen Regelkreis ergab sich unter Verwendung des Endwertsatzes der Laplace-Transformation, daß dieser Regler bei konstanten Objektgeschwindigkeiten die Abweichung des Objekts vom Bildmittelpunkt nicht eliminieren kann. Dieses Verhalten konnte auch in realen Experimenten nachgewiesen werden. Um die Abweichung gering zu halten, mußte ein möglichst großer Wert für den Verstärkungsfaktor des P-Reglers gewählt werden, wobei jedoch die Höhe des Wertes durch Forderungen in Bezug auf die Abtastraten nach oben begrenzt wurde.

Die Analyse des geschlossenen Regelkreises mit D- und I-Regler zeigte, daß diese Reglertypen völlig ungeeignet für den Einsatz in der untersuchten Regelstrecke sind. Bei der Verwendung eines I-Reglers entsteht ein grenzstabiles System, das zu Dauerschwingungen neigt.

Im Gegensatz zum P-Regler ist der PI-Regler dazu in der Lage, die Regelabweichung bei konstanten Objektgeschwindigkeiten zu beseitigen. Dies konnte sowohl theoretisch berechnet, als auch beim Einsatz in der realen Regelstrecke gezeigt werden. Durch die Trägheit des I-Gliedes im PI-Regler muß jedoch ein leichtes Überschwingen bei Geschwindigkeitsänderungen des Objekts hingenommen werden. Der PID-Regler liefert aufgrund

der Parameterwahl ähnliche Ergebnisse wie der PI–Regler, reagiert allerdings wegen des enthaltenen D–Gliedes etwas rascher auf Geschwindigkeitsänderungen des Objekts.

Bei der Einstellung des Fuzzy–Reglers wurde nach rein empirischen Gesichtspunkten vorgegangen, weil kein analytisches Modell für diesen Reglertyp existiert. Da für die Eingangsgrößen des Reglers die Abweichung vom Bildmittelpunkt sowie die Summe der vergangenen Abweichungen gewählt wurde, ergab sich ein ähnliches Regelverhalten wie beim PI–Regler. Allerdings reagiert der Fuzzy–Regler aufgrund seines nichtlinearen Übertragungsverhaltens etwas schneller auf kleine Abweichungen als sein lineares Gegenstück.

Die Laufzeiten der implementierten Regler sind konstant und genügen leicht den Anforderungen in der Echtzeit–Objektverfolgung. Ebenso verhält es sich mit dem Speicherbedarf.

Literaturverzeichnis

[Alt95] Altrock, C.: *Fuzzy Logic - Technologie*, Oldenbourg Verlag GmbH, München, 2. Ausg., 1995.

[BCS92] Brown, C.; Coombs, D.; Soong, J.: *Real-time Smooth Pursuit Tracking*, in Blake, A.; Yuille, A. (Hrsg.): *Computer Vision*, 1992, S. 132–137.

[CN83] Coulan, P.; Nougaret, M.: *Use of a TV camera system in closed-loop position control mechanisms*, in Pugh, A. (Hrsg.): *Robot Vision*, 1983, S. 115–127.

[DHKS95] Daniilidis, K.; Hansen, M.; Krauss, C.; Sommer, G.: *Auf dem Weg zum künstlichen aktiven Sehen: Modellfreie Bewegungsverfolgung durch Kameranachführung*, in *DAGM 1995, Bielefeld*, 1995, S. 277–284.

[DN95] Denzler, J.; Niemann, H.: *Combination of Simple Vision Modules for Robust Real-Time Motion Tracking, European Transactions on Telecommunications*, Bd. 5, Nr. 3, 1995, S. 275–286.

[FL90] Feddema, J.; Lee, C.: *Adaptive Image Feature Prediction and Control for Visual Tracking with a Hand-Eye Coordinated Camera, IEEE Transactions on Systems, Man and Cybernetics*, 1990, S. 1172–1183.

[FLM91] Feddema, J.; Lee, C.; Mitchell, O.: *Weighted Selection of Image Features for Resolved Rate Visual Feedback Control, IEEE Transactions on Robotics and Automation*, 1991, S. 31–47.

[GBD*93] Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V.: *PVM 3.0 User's Guide and Reference Manual*, ORNL/TM-12187, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Tennessee, 1993.

[HMB93] Harris, C.; Moore, C.; Brown, M.: *Intelligent Control - Aspects of Fuzzy Logic and Neural Nets*, World Scientific Publishing Co. Pte. Ltd., Singapore, New Jersey, London, Hong Kong, 1. Ausg., 1993.

[Ise87] Isermann, R.: *Digitale Regelsysteme Band II*, Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 2. Ausg., 1987.

[Ise88] Isermann, R.: *Digitale Regelsysteme Band I*, Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 2. Ausg., 1988.

[KG91] K.E. Gorlen, P. P., S.M. Orlow: *Data Abstraction and Object-Oriented Programming in C++*, Teubner-Verlag, Stuttgart, 1. Ausg., 1991.

[Leo92] Leonhard, W.: *Einführung in die Regelungstechnik*, Vieweg-Verlag, Braunschweig/Wiesbaden, 6. Ausg., 1992.

[Pau91] Paulus, D.: *Das System Puma*, Erlangen, 1991.

[Ped95] Pedrycz, W.: *Fuzzy Sets Engeneering*, CRC Press, Inc., 1. Ausg., 1995.

[PKK93] Papanikolopoulos, N.; Khosla, P.; Kanade, T.: *Visual Tracking of a Moving Target by a Camera Mounted on a Robot: A Combination of Control and Vision*, *IEEE Transactions on Robotics and Automation*, Bd. 9, Nr. 1, 1993, S. 14–34.

[Pro95] Protzel, P.: *Neuronale Netze und Fuzzy Logik*, FORWISS, Erlangen, 1995.

[RG92] R. Groover Brown, P. H.: *Introduction to random signals and applied Kalman filtering*, John Wiley & Sons, INC., New York, Chichester, Brisbane, Toronto, Singapore, 2. Ausg., 1992.

[Til92] Tilli, T.: *Fuzzy-Logik*, Franzis Verlag GmbH & Co. KG, München, 2. Ausg., 1992.

[Tra94] Traeger, D.: *Einführung in die Fuzzy-Logik*, Teubner Verlag, Stuttgart, 2. Ausg., 1994.

[Unb87] Unbehauen, H.: *Regelungstechnik I*, Vieweg-Verlag, Braunschweig/Wiesbaden, 5. Ausg., 1987.

[WSN87] Weiss, L.; Sanderson, A.; Neuman, C.: *Dynamic Sensor–Based Control of Robots with Visual Feedback*, *IEEE Journal of Robotics and Automation*, 1987, S. 404–417.

[Zim92] Zimmermann, H.: *Fuzzy Set Theorie and its applications*, Kluwer Academic Publishers, Norwell, Massachusetts USA, 2. Ausg., 1992.

Anhang A

Klassendokumentation

Es folgt die Dokumentation zu den in der Programmiersprache C++ implementierten Klassen. Die Schnittstellen aller Klassen werden eingehend beschrieben. Dabei werden die realisierten Konstruktoren, Methoden und Operatoren erläutert. Alle vorliegenden Klassenbeschreibungen wurden automatisch aus den Header–Files extrahiert.

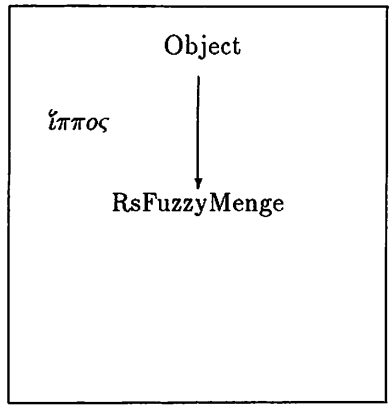
RsFuzzyMenge Basis-Fuzzy-Menge

Klassenname: RsFuzzyMenge

Version: 1.6

Basisklasse: Object

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyMenge* ist die abstrakte Basisklasse für alle Fuzzy-Mengen.

Methoden:

RsFuzzyMenge: `(real xMin, real xMax)` ⟶ `(Cons)`
Konstruktor für RsFuzzyMenge:
Parameter:

- **xMin** : Wertebereich-Minimum
- **xMax** : Wertebereich-Maximum

RsFuzzyMenge: `()` ⟶ `(Cons)`
Default-Konstruktor: setzt *xMin*, *xMax* auf 0

RsFuzzyMenge: `(const RsFuzzyMenge& m)` ⟶ `(Cons)`
Copy-Konstruktor

WertBei: `(real xWert) = 0` ⟶ `real`
Liefert Zugehörigkeit bei *xWert*

Schwerpunkt: `() const = 0` ⟶ `real`
Liefert x-Wert des Schwerpunkts der Menge

Modifiziere: `(ifstream& s) = 0` ⟶ `void`
Methode für den Einstell-Regler: Liest Konfigurationsfile und setzt die angegebenen Parameter.

HoleXMin: `() const` ⟶ `real`
Liefert Wertebereich-Minimum

SetzeXMin: `(real xMin)` ⟶ `void`
Setzt Wertebereich-Minimum auf *xMin*

HoleXMax: `() const ⟶ real`
Liefert Wertebereich-Maximum

SetzeXMax: `(real xMax) ⟶ void`
Setzt Wertebereich-Maximum auf *xMax*

HoleAltenWert: `() const ⟶ real`
Liefert die zuletzt berechnete Zugehörigkeit zur Menge

ErzeugeGnuPlot: `(int anzahlWerte, ostream& strm) ⟶ void`
Erzeugt Gnu-Plot der Menge mit *anzahlWerte* Abtastungen

VarOperator: `(KommOp op, ...) = 0 ⟶ void`
Verknüpft Fuzzy-Mengen über operator *op*. Nach *op* werden Zeiger auf *RsFuzzyMengen* erwartet. Am Ende muß ein 0-Zeiger stehen! Das Ergebnis der Verknüpfung wird in *this* gespeichert.

Operatoren:

`=` `⟶ RsFuzzyMenge&`
Zuweisungsoperator

`==` `⟶ int`
Prüft alle Membervariablen auf Gleichheit.

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyMenge.h

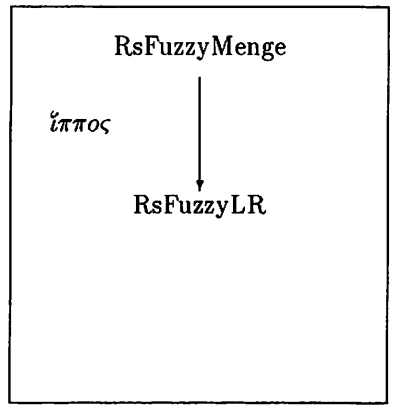
RsFuzzyLR LR-Fuzzy-Menge

Klassenname: RsFuzzyLR

Version: 1.7

Basisklasse: RsFuzzyMenge

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyLR* repräsentiert Fuzzy-Mengen, bei denen die linke und rechte Flanke durch Funktionen beschrieben wird.

Methoden:

RsFuzzyLR: `(real xMin, real xMax, real yMin, real yMax, real alpha, real betha, real m1, real m2, LRFunktion LFunktion, LRFunktion RFunktion)` ⟶ `(Cons)`
Konstruktor für *RsFuzzyLR*:
Parameter:

- **xMin/xMax** : x-Wertebereich der Menge
- **yMin/yMax** : minimale/maximale Zugehörigkeit
- **alpha** : linke Aufspreizung
- **betha** : rechte Aufspreizung
- **m1,m2** : Im Bereich [m1,m2] besitzt die Menge volle Zugehörigkeit *yMax*.
- **L/RFunktion**: Funktionen für linke/rechte Flanke der Menge

RsFuzzyLR: `(const RsFuzzyLR& m)` ⟶ `(Cons)`
Copy-Konstruktor

WertBei: `(real xWert)` ⟶ `real`
Liefert Zugehörigkeit bei *xWert*

Schwerpunkt: `() const ⟶ real`
Nicht implementiert

VarOperator: `(KommOp op, ...) ⟶ void`
Nicht implementiert

Modifiziere: `(ifstream& s) ⟶ void`
Methode, die vom Einstell-Regler verwendet wird, um die Parameter zu setzen. Zuerst wird das Schlüsselwort für die einzustellenden Parameter gelesen, dann der Wert, der zu setzen ist. Ist die Eingabe nicht gültig, so wird das Programm mit einer Fehlermeldung beendet.

HoleParameter: `(real &yMin, real &yMax, real &alpha, real &betha, real &m1, real &m2) const ⟶ void`
Liefert Parameter der Menge

SetzeParameter: `(real yMin, real yMax, real alpha, real betha, real m1, real m2) ⟶ int`
Zum Einstellen der Parameter: Rückgabewert 1, wenn Werte ok, sonst 0.

HoleAlpha: `() ⟶ real`
Liefert alpha-Wert

HoleBetha: `() ⟶ real`
Liefert betha-Wert

HoleM1: `() ⟶ real`
Liefert m1

HoleM2: `() ⟶ real`
Liefert m2

SetzeAlpha: `(real alpha) ⟶ void`

SetzeBetha: `(real betha) ⟶ void`

SetzeM1: `(real m1) ⟶ void`

SetzeM2: `(real m2) ⟶ void`

Operatoren:

`=` `⟶ RsFuzzyLR&`
Zuweisungsoperator

== $\longrightarrow$ `int`

Prüft jede Membervariable auf Gleichheit

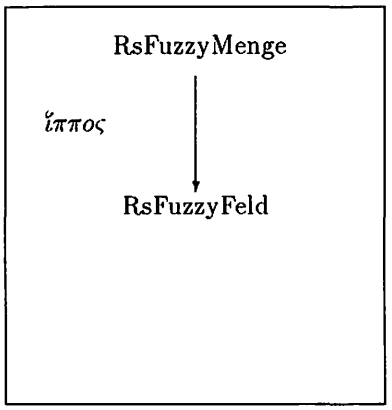
Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyLR.h

Klassenname: RsFuzzyFeld

Version: 1.6

Basisklasse: RsFuzzyMenge

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyFeld* repräsentiert Fuzzy-Mengen, die durch diskrete Werte definiert sind. Die Zugehörigkeiten werden dabei in einem Vektor abgespeichert.

Methoden:

RsFuzzyFeld: `(real xMin, real xMax, int feldLaenge)` ⟶ `(Cons)`
Initialisiert Feld mit 0
Parameter:

- **xMin/xMax** : x-Wertebereich der Menge
- **feldLaenge** : Anzahl der Abtastungen des Feldes (Auflösung)

RsFuzzyFeld: `(real xMin, real xMax, real* fuzzyFeld, int feldLaenge)` ⟶ `(Cons)`
Initialisiert Feld mit *fuzzyFeld*

RsFuzzyFeld: `(const RsFuzzyFeld& m)` ⟶ `(Cons)`
Copy-Konstruktor: Kopiert Feld vollständig!

RsFuzzyFeld: `()` ⟶ `(Cons)`
Default-Konstruktor: Belegt alles mit 0

WertBei: `(real xWert)` ⟶ `real`
Liefert Zugehörigkeit bei *xWert*

Schwerpunkt: `() const` ⟶ `real`
Liefert *x-Wert* des Schwerpukts der Menge

VarOperator: `(KommOp op, ...)` ⟶ `void`
Verknüpft mehrere FuzzyMengen über *op*

Modifiziere: `(ifstream& s)` ⟶ `void`
Nicht implementiert

HoleFeldLaenge: `() const` ⟶ `int`

Operatoren:

`[]` ⟶ `real`
Const Zugriff auf *fuzzyFeld*

`[]` ⟶ `real&`
Normaler Zugriff auf *fuzzyFeld*

`=` ⟶ `RsFuzzyFeld&`
Zuweisungoperator: Löscht altes Feld und kopiert Neues!

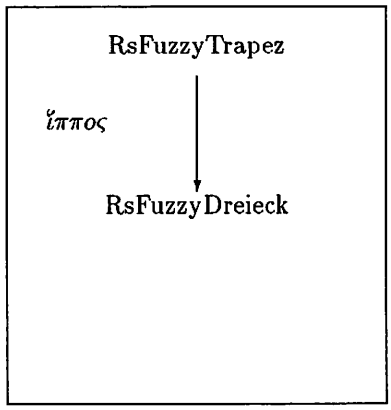
Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyFeld.h

Klassenname: RsFuzzyDreieck

Version: 1.6

Basisklasse: RsFuzzyTrapez

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyDreieck* repräsentiert Fuzzy-Mengen, die ein dreieckförmiges Aussehen besitzen.

Methoden:

RsFuzzyDreieck: `(real xMin, real xMax, real yMin, real yMax, real alpha, real betha, real xSpitze)` ⟶ `(Cons)`
Konstruktor für RsFuzzyDreieck:
Parameter

- **xMin/xMax** : x-Wertebereich der Menge
- **yMin/yMax** : Minimale/maximale Zugehörigkeit
- **alpha** : Linke Aufspreizung
- **betha** : rechte Aufspreizung
- **xSpitze** : x-Wert mit maximaler Zugehörigkeit (Spitze)

RsFuzzyDreieck: `(const RsFuzzyDreieck& m)` ⟶ `(Cons)`
Copy-Konstruktor

SetzeParameter: `(real yMin, real yMax, real alpha, real betha, real xSpitze)` ⟶ `int`
Zum Einstellen der Parameter. Rückgabewert 1, wenn Wert ok, sonst 0.

HoleParameter: `(real &yMin, real &yMax, real &alpha, real &betha, real &xSpitze) const` ⟶ `void`
Liefert Parameter der Dreieck-Fuzzy-Menge

Modifiziere: `(ifstream& s)` $\longrightarrow$ `void`

Methode, die vom Einstell-Regler verwendet wird, um die Parameter zu setzen. Zuerst wird das Schlüsselwort für die einzustellenden Parameter gelesen, dann der Wert, der zu setzen ist. Ist die Eingabe nicht gültig, so wird das Programm mit einer Fehlermeldung beendet.

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyDrei.h

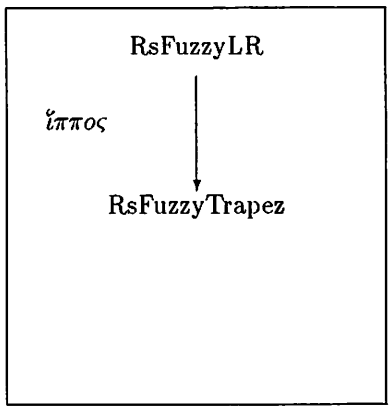
Trapez-Fuzzy-Menge **RsFuzzyTrapez**

Klassenname: RsFuzzyTrapez

Version: 1.6

Basisklasse: RsFuzzyLR

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyTrapez* repräsentiert Fuzzy-Mengen, die ein trapezförmiges Aussehen besitzen.

Methoden:

RsFuzzyTrapez: `(real xMin, real xMax, real yMin, real yMax, real alpha, real betha, real m1, real m2)` ⟶ `(Cons)`
Konstruktor für RsFuzzyTrapez:
Parameter:

- **xMin/xMax** : x-Wertebereich der Menge
- **yMin/yMax** : minimale/maximale Zugehörigkeit
- **alpha** : linke Aufspreizung
- **betha** : rechte Aufspreizung
- **m1/m2** : x-Bereich mit maximaler Zugehörigkeit *yMax*

RsFuzzyTrapez: `(const RsFuzzyTrapez& m)` ⟶ `(Cons)`
Copy-Konstruktor

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyTrap.h

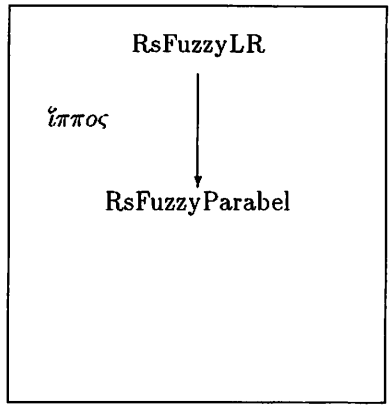
RsFuzzyParabel Parabel-Fuzzy-Menge

Klassenname: RsFuzzyParabel

Version: 1.4

Basisklasse: RsFuzzyLR

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyParabel* repräsentiert Fuzzy-Mengen, die ein parabelförmiges Aussehen besitzen.

Methoden:

RsFuzzyParabel: `(real xMin, real xMax, real yMin, real yMax, real alpha, real betha, real m1, real m2)` ⟶ `(Cons)`
Konstruktor für RsFuzzyParabel:
Parameter:

- **xMin/xMax** : x-Wertebereich der Menge
- **yMin/yMax** : minimale/maximale Zugehörigkeit
- **alpha** : linke Aufspreizung
- **betha** : rechte Aufspreizung
- **m1/m2** : x-Bereich mit maximaler Zugehörigkeit *yMax*

RsFuzzyParabel: `(const RsFuzzyParabel& m)` ⟶ `(Cons)`
Copy-Konstruktor

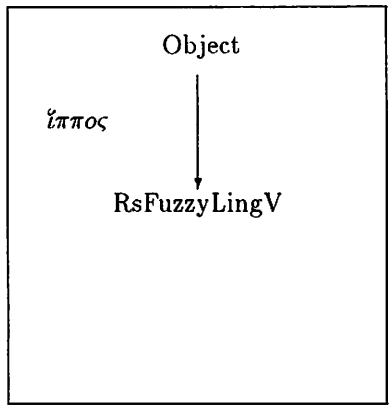
Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyParab.h

Klassenname: RsFuzzyLingV

Version: 1.8

Basisklasse: Object

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyLingV* repräsentiert eine linguistische Variable im Sinne der Fuzzy-Theorie. Als Wertebereich besitzt diese Variable Fuzzy-Mengen. Ein Objekt der Klasse *RsFuzzyLingV* kann alle Fuzzy-Mengen speichern, die von *RsFuzzyMenge* abgeleitet sind! Dabei werden die Fuzzy-Mengen **vollständig** kopiert!

Methoden:

RsFuzzyLingV: `()` ⟶ `(Cons)`
Default Konstruktor: Alles wird mit 0 initialisiert.

RsFuzzyLingV: `(const OrderedCltn&)` ⟶ `(Cons)`
Erzeugt linguistische Variable mit Wertemenge `OrderedCltn<RsFuzzyMenge*>` . Alle Objekte hinter Zeigern werden kopiert!

RsFuzzyLingV: `(const RsFuzzyLingV& v)` ⟶ `(Cons)`
Copy-Konstruktor: Alle Objekte hinter Zeigern werden kopiert!

NeueMenge: `(RsFuzzyMenge* m)` ⟶ `void`
Fügt Fuzzy-Menge *m* zur linguistischen Variablen hinzu!

Berechne: `(real x)` ⟶ `void`
Berechnet alle Zugehörigkeiten der Fuzzy-Mengen für x. Die Ergebnisse werden in den Fuzzy-Mengen gespeichert.

Inferenz: `(RsFuzzyFeld& f, Vector<real> v, RsFuzzyOperator& opAcc, RsFuzzyOperator& opInf) ⟶ void`
Berechnet f bei Eingabedaten v:
Parameter:

- **f** : hier wird das Ergebnis abgelegt
- **v** : Vektor mit Eingabedaten für Inferenzbildung
- **opAcc** : Operator zur Akkumulation der einzelnen durch Inferenz entstehenden Fuzzy-Mengen (z.B. max)
- **opInf** : Operator zur Inferenzbildung von Wert in v und entsprechender Fuzzy-Menge von *RsFuzzyLingV* (z.B. min)

ErzeugeGnuPlot: `(int aufl, ostream& strm) const ⟶ void`
Gibt auf *strm* die linguistische Variable mit *aufl* Abtastungen im gnuplot-Format aus

AnzahlMengen: `() const ⟶ int`

HoleXMin: `() const ⟶ real`
Liefert minimalen x-Wert der linguistischen Variablen

HoleXMax: `() const ⟶ real`
Liefert maximalen x-Wert der linguistischen Variablen

Operatoren:

`[]` `⟶ RsFuzzyMenge*`
Gibt i 'te Fuzzy-Menge zurück

`==` `⟶ int`
Prüft alle Membervariablen auf Gleichheit

`=` `⟶ RsFuzzyLingV&`
Zuweisungsoperator: Alle Objekte hinter Zeigern werden kopiert!

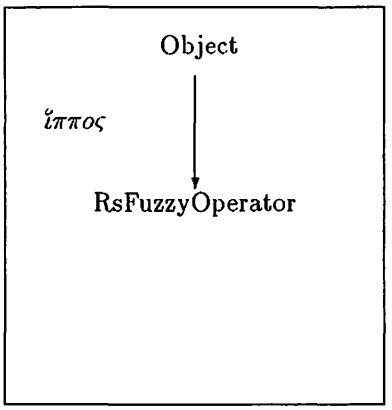
Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyLingV.h

Klassenname: RsFuzzyOperator

Version: 1.6

Basisklasse: Object

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyOperator* ist die abstrakte Basisklasse für alle Operatoren, die in *RsFuzzyRegelB* oder *RsFuzzyRegler* verwendet werden können.

Bisher verfügbare Operatoren:

1. **T-Normen:**
   - *RsFuzzyMin*
   - *RsFuzzyDrastProd*
   - *RsFuzzyBegrDiff*
   - *RsFuzzyEinstProd*
   - *RsFuzzyAlgProd*
   - *RsFuzzyHamacherProd*
2. **S-Normen:**
   - *RsFuzzyMax*
   - *RsFuzzyDrastSumm*
   - *RsFuzzyBegrSumm*
   - *RsFuzzyEinstSumm*
   - *RsFuzzyAlgSumm*
   - *RsFuzzyHamacherSumm*

In Header-Files nachschauen!

Methoden:

RsFuzzyOperator:

() ⟶ (Cons)

Operatoren:

⟶ real

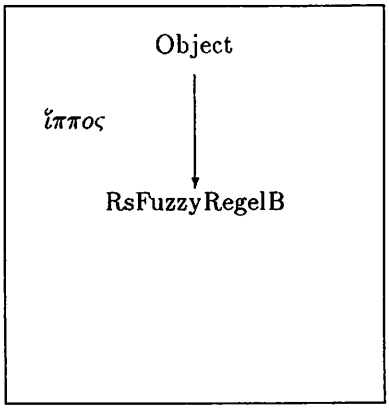
Funktion, die in den Subklassen implementiert werden muß!
Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyOperat.h

Klassenname: RsFuzzyRegelB

Version: 1.11

Basisklasse: Object

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyRegelB* führt nach Eingabe eines scharfen Eigabevektors *(n x 1)* zunächst eine Fuzzyfizierung der Eingabewerte durch. Mit der gespeicherten Regelbasis und anschließender Defuzzyfizierung kann dann wieder ein scharfer Ausgangswert ermittelt werden. Diese Klasse bildet die Grundlage zur Erstellung eines Fuzzy-Reglers, kann aber möglicherweise auch z.B. zum Bau eines Expertensystems herangezogen werden.

Methoden:

RsFuzzyRegelB: `(const RsFuzzyLingV& l, const RsFuzzyOperator& op1, const RsFuzzyOperator& op2, const RsFuzzyOperator& op3, const RsFuzzyOperator& op4)` ⟶ `(Cons)`

Parameter:

- l : linguistische Variable für den Ausgang
- **op1** : Operator zur Verknüpfung der Fuzzy-Mengen auf den rechten Seiten der Regeln. (meist min)
- **op2** : Operator zur Akkumulation von Regelergebnissen, falls für eine Ausgabe-Fuzzy-Menge mehrere Regeln „feuern"
- **op3** : Operator zur Inferenzbildung (meist min zwischen) Ausgabewert der Regelbasis und jeweiliger Fuzzy-Menge
- **op4** : Operator zur Akkumulation der durch Inferenzbildung entstandenen Fuzzy-Mengen (meist max)

RsFuzzyRegelB: `(const RsFuzzyRegelB& r)` ⟶ `(Cons)`

RsFuzzyRegelB: `()` ⟶ `(Cons)`

NeueRegel: `(int nr, ...)` ⟶ `void`

Fügt für die Ausgabe-Fuzzy-Menge mit Nummer *nr* eine neue Regel zur Regelbasis hinzu. *nr* ist die Nummer der entsprechenden Fuzzy-Menge in der linguistischen Variable für die Ausgabe. Nach *nr* können eine beliebige Anzahl von Zeigern *int** als Argumente angegeben werden. Dabei entspricht *int* Fuzzy-Menge[i][j]* der *j* 'ten FuzzyMenge in der *i* 'ten Eingabe-Linguistischen Variablen .Die Argumentliste muß IMMER Nullterminiert sein! Die angegebenen Fuzzy-Mengen werden dann bei der Regelauswertung mit *op1* (siehe Konstruktor) verknüpft. Die Regel lautet dann also:
WENN Fuzzy-Menge1 **op1** Fuzzy-Menge2 **op1** ... **DANN** Ausgabe-Fuzzy-Menge[*nr*]
Beispiel:
int fehler_n[2] = 0,0;
RegelBasis.NeueRegel(0, fehler_n, 0)

NeueRegel: `(int nr,const Vector<FuzzyMengenRef>& v)` ⟶ `void`

Erzeugt neue Regel: s.o.

NeueEingabeLingV:

`(const RsFuzzyLingV& v)` ⟶ `void`

Fuegt neue Linguistische Variable für den Eingang hinzu

Berechne: `(const realArray& v, RsFuzzyFeld& f)` ⟶ `real`

Berechnet einen scharfen Ausgabewert für den scharfen Eingabevektor *v*.
Parameter:

- **v** : Eingabevektor mit scharfen Werten.
- **f** : Referenz auf *RsFuzzyFeld*. Dort wird die Ergebnis-Fuzzy-Menge gespeichert.
- **Rückgabewert** : Schwerpunkt von *v*

OutputLingVar: `()` ⟶ `RsFuzzyLingV&`

Liefert Referenz auf linguistische Variable für Ausgabe zurück.

InputLingVar: `(int n)` ⟶ `RsFuzzyLingV&`

Liefert *n* 'te linguistische Variable für die Eingabe zurück

HoleAnzahlEingLingVar:

`() const ⟶ int`

Liefert Anzahl der linguistischen Variablen für den Eingang zurück

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyRegelB.h

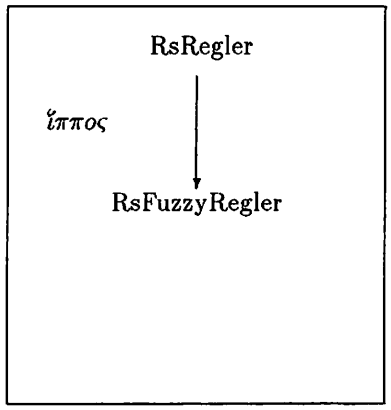
RsFuzzyRegler Der Fuzzy-Regler

Klassenname: RsFuzzyRegler

Version: 1.11

Basisklasse: RsRegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Mit der Klasse RsFuzzyReger ist es möglich Fuzzy-Regler zu erzeugen, die beliebig viele Eingänge und einen Ausgang besitzen. Durch verschiedenartige Fuzzy-Mengen und unterschiedliche Verknüpfungsoperatoren für diese kann der Regler sehr variabel eingestellt werden.
Nach dem Aufruf des Konstruktors müssen die linguistischen Variablen für die Eingabe mit *„NeueEingabeLingV(...)"* angegeben werden. Danach muß die **vollständige** Regelmatrix mit *„NeueRegel(...)"* eingegeben werden. Nach dieser Prozedur ist der Regler fertig konfiguriert.

Methoden:

RsFuzzyRegler: `(int aufl, int anzEing, const RsFuzzyLingV& l, const RsFuzzyOperator& op1, const RsFuzzyOperator& op2, const RsFuzzyOperator& op3, const RsFuzzyOperator& op4)` ⟶ `(Cons)`
Konstruktor von *RsFuzzyRegler* :
Parameter:

- **aufl:** Anzahl der Abtastungen bei Inferenzbildung.
  Je höher dieser Wert ist, desto genauer
  arbeitet der Regler. (Normal: 200 - 400)
- **anzEing:** Anzahl der Eingänge des Reglers
- **l:** linguistische Variable für den Ausgang
- **op1:** Operator zur Verknüpfung der Fuzzy-Mengen auf den rechten Seiten der Regeln. (meist min)
- **op2:** Operator zur Akkumulation von Regelergebnissen, falls für eine Ausgabe-Fuzzy-Menge mehrere Regeln „feuern"
- **op3:** Operator zur Inferenzbildung (meist min zwischen) Ausgabewert der Regelbasis und jeweiliger Fuzzy-Menge
- **op4:** Operator zur Akkumulation der durch Inferenzbildung entstandenen Fuzzy-Mengen (meist max)

RsFuzzyRegler: `(const RsFuzzyRegler& r)` ⟶ `(Cons)`

RsFuzzyRegler: `()` ⟶ `(Cons)`

Regle_: `(const realArray& input, realArray& output)` ⟶ `int`
Erzeugt aus Führungsgrößen *input* die Stellgrößen *output*. Die Stellgrößen werden nach *output* geschrieben.

NeueRegel: `(int nr, ...)` ⟶ `void`

Fügt für die Ausgabe-Fuzzy-Menge mit Nummer *nr* eine neue Regel zur Regelbasis hinzu. *nr* ist die Nummer der entsprechenden Fuzzy-Menge in der linguistischen Variable für die Ausgabe. Nach *nr* können eine beliebige Anzahl von Zeigern *int** als Argumente angegeben werden. Dabei entspricht *int* Fuzzy-Menge[i][j]* der *j* 'ten FuzzyMenge in der *i* 'ten Eingabe-Linguistischen Variablen . Die Argumentliste muß IMMER Nullterminiert sein! Die angegebenen Fuzzy-Mengen werden dann bei der Regelauswertung mit *op1* (siehe Konstruktor) verknüpft. Die Regel lautet dann also:

WENN Fuzzy-Menge1 **op1** Fuzzy-Menge2 **op1** ... **DANN** Ausgabe-Fuzzy-Menge[*nr*]

Beispiel:

int fehler_n[2] = 0,0;

FuzzyRegeler.NeueRegel(0, fehler_n, 0)

NeueEingabeLingV:

`(const RsFuzzyLingV& v)` ⟶ `void`

Fügt neue Linguistische Variable für Eingang hinzu. Sie werden von 0 an aufwärts durchnummeriert. *(für NeueRegel())*

HoleRegelBasis: `()` ⟶ `RsFuzzyRegelB&`

Liefert Referenz auf interne Regelbasis zurück

ErzeugeXmaplePlot:

`(ofstream& strm, int punkte)` ⟶ `void`

Erzeugt einen Plot des Übertragungsverhaltens des Reglers für Xmaple. *punkte* sind die Anzahl der Abtastungen pro Dimension. Der Plot muß in xmaple mit *include* geladen werden. Im Moment ist nur Plotten von Reglern mit zwei linguistischen Variablen für die Eingabe implementiert.

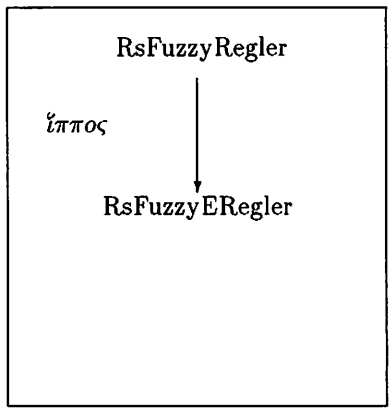
Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyRegler.h

Klassenname: RsFuzzyERegler

Version: 1.6

Basisklasse: RsFuzzyRegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsFuzzyERegler* ist vorgesehen, um den Fuzzy-Regler für eine gegebene Regelstrecke richtig einzustellen. Es sind zwei Modi zur Einstellung implementiert: Der ONLINE- und der STACK-Modus. Beim STACK-Modus muß zunächst ein Konfigurationsfile erzeugt werden, in dem alle Konfigurationen des Reglers aufgeführt sind, die getestet werden sollen. Während des Testlaufs mit der Regelstrecke werden dann vom Einstell-Regler in vom Benutzer vorgegebenen Intervallen (Anzahl der Regelungen) die einzelnen Konfigurationen eingelesen, und durch verschiedene trace-files (Gnu-Plot) dokumentiert.
Beim ONLINE-Modus wird immer nur eine Konfiguration gelesen, die vom Benutzer ständig geändert werden kann. Somit kann man unmittelbar durch Ändern des Konfigurationsfiles das Reglerverhalten beeinflussen und somit sofort die Auswirkungen an der Regelstrecke beobachten. Der ONLINE-Modus ist eher für das Fine-Tuning des Reglers gedacht, nachdem man ihn bereits grob eingestellt hat. Das Konfigurationsfile könnte z.B folgendermaßen aussehen:

ANFANG_KONFIG
E 0 0 M2 -0.5
A 1 ALPHA 0.5
ENDE_KONFIG

Dies bewirkt, daß im Regler der Parameter *m2* in der *0*'ten Fuzzy-Menge der *0*'ten linguistischen Variablen für die Eingabe auf *-0.5* gesetzt wird. Die erste Fuzzy-Menge der linuistischen Variablen wird auf den Wert *alpha* = 0.5 gesetzt. Alle anderen Werte des Reglers bleiben unverändert. Aktiviert werden die Einstell-Modi durch Aufruf der Methode *StarteKonf()*. Vor diesem Aufruf verhält sich der Einstellregler genauso wie ein „normaler" *RsFuzzyRegler*. Folgende trace-files können ausgegeben werden:

- **Eingabe:** Abfolge der scharfen Eingabegrößen
- **Ausgabe:** Abfolge der vom Regler bestimmten Stellgröße
- **Eingabe-Ling-Var:** Aktuelle Linguistische Variablen für die Eingabe
- **Ausgabe-Ling-Var:** Aktuelle Linguistische Variable für die Ausgabe
- **Gesamt-Eingabe:** Aufaddierte Beträge der Eingangswerte für jede Konfiguration.

Defaultmäßig werden die Eingabe-Files und die Files für die linguistische Variablen der Eingabe erzeugt. Alle trace-files werden im Gnu-Plot Format ausgegeben.

Methoden:

RsFuzzyERegler:`(int aufl, int anzEing, const RsFuzzyLingV& l, const RsFuzzyOperator& op1, const RsFuzzyOperator& op2, const RsFuzzyOperator& op3, const RsFuzzyOperator& op4)` ⟶ `(Cons)`

Konstruktor von *RsFuzzyERegler* :

Parameter:

- **aufl:** Anzahl der Abtastungen bei Inferenzbildung.
  Je höher dieser Wert ist, desto genauer
  arbeitet der Regler. (Normal: 200 - 400)
- **anzEing:** Anzahl der Eingänge des Reglers
- **l:** linguistische Variable für den Ausgang
- **op1:** Operator zur Verknüpfung der Fuzzy-Mengen auf den rechten Seiten der Regeln. (meist min)
- **op2:** Operator zur Akkumulation von Regelergebnissen, falls für eine Ausgabe-Fuzzy-Menge mehrere Regeln „feuern"
- **op3:** Operator zur Inferenzbildung (meist min zwischen) Ausgabewert der Regelbasis und jeweiliger Fuzzy-Menge
- **op4:** Operator zur Akkumulation der durch Inferenzbildung entstandenen Fuzzy-Mengen (meist max)

RsFuzzyERegler:`(const RsFuzzyERegler& r)` ⟶ `(Cons)`

RsFuzzyERegler:`()` ⟶ `(Cons)`

Regle_: `(const realArray& input, realArray& output) ⟶ int`

Erzeugt aus Führungsgrößen *input* die Stellgrößen *output*. Die Stellgrößen werden nach *output* geschrieben. Wurde vorher *StarteKonf()* aufgerufen, so befindet sich der Regler in einem der zwei Einstell-Modi und kann mit Hilfe des Konfigurationsfiles eingestellt werden.

StarteKonf: `(long anzahlRegelungen, const char* verzeichnis, const char* konfigFile, Modus = STACK, int IkonfNr = 0) ⟶ void`

Startet den Einstell-Vorgang:
Modus gibt an, welcher Modus des Einstellregler benutzt werden soll. Es stehen hier STACK (default) und ONLINE zur Verfügung (s.o). Beim STACK-Modus wird jede Konfiguration aus *konfigFile anzahlRegelungen* Regelungsdurchläufe als Reglerkonfiguration betrieben. Beim ONLINE-Modus wird alle *anzahlRegelungen* Regelungen das Konfigurationsfile *konfigFile* neu gelesen. Die trace-files werden im Verzeichnis *verzeichnis* abgelegt.

EntferneAusgabeFile:

`(AusgabeFiles f) ⟶ void`

Angegebenes trace-file *f* wird nicht erzeugt. Mögliche Angaben für *f*:

- **AUSGANG_LING** : Linguistische Variable für den Ausgang
- **EINGANG_LING** : Linguistische Variablen für den Eingang
- **AUSGANG** : Trace-file für den Ausgang
- **EINGANG** : Trace-file für den Eingang
- **EINGANG_GESAMT** : Beträge der Eingangswerte aufaddiert

NeuesAusgabeFile:

`(AusgabeFiles f) ⟶ void`

Angegebenes trace-file *f* wird erzeugt. Mögliche Angaben für *f*: s.o.!

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsFuzzyERegler.h

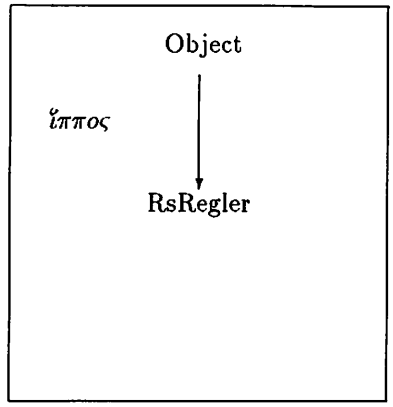
RsRegler Regler-Basisklasse

Klassenname: RsRegler

Version: 1.7

Basisklasse: Object

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsRegler* ist die abstrakte Basisklasse für alle Reglerklassen.

Methoden:

RsRegler: `(int anzEing, int anzAusg)` ⟶ `(Cons)`
Konstruktor für *RsRegler* :
Parameter:

- **anzEing** : Anzahl der Eingangsgrößen
- **anzAusg** : Anzahl der Ausgangsgrößen

RsRegler: `(const RsRegler& r)` ⟶ `(Cons)`

RsRegler: `()` ⟶ `(Cons)`

Regle_: `(const realArray& input, realArray& output) = 0` ⟶ `int`
Erzeugt aus Führungsgrößen *input* die Stellgrößen *output*. Die Stellgrößen werden in *output* geschrieben.
Implementierung in den Sub-Klassen!

Regle: `(const realArray& input, realArray& output)` ⟶ `int`
Erzeugt aus Führungsgrößen *input* die Stellgrößen *output*. Die Stellgrößen werden in *output* geschrieben. Ruft virtuelle Methode *Regle_* der Subklasse auf und überprüft *output*, ob die Maximalwerte überschritten wurden. Ist dies der Fall, so werden die Maximalwerte nach output geschrieben. („*sicherer Regler!*")

SetzeNMinAusg: `(int n, real min) ⟶ void`

Setzt den minimal zulässigen Ausgangspegel für Ausgang n

HoleNMinAusg: `(int n) const ⟶ real`

Liefert den minimal zulässigen Ausgangspegel für Ausgang n

SetzeNMaxAusg: `(int n, real max) ⟶ void`

Setzt den maximal zulässigen Ausgangspegel für Ausgang n

HoleNMaxAusg: `(int n) const ⟶ real`

Liefert den maximal zulässigen Ausgangspegel für Ausgang n

HoleAnzahlAusg: `() const ⟶ int`

Liefert die Anzahl der vorhandenen Ausgänge des Reglers.

HoleAnzahlEing: `() const ⟶ int`

Liefert die Anzahl der vorhandenen Eingänge des Reglers.

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsRegler.h

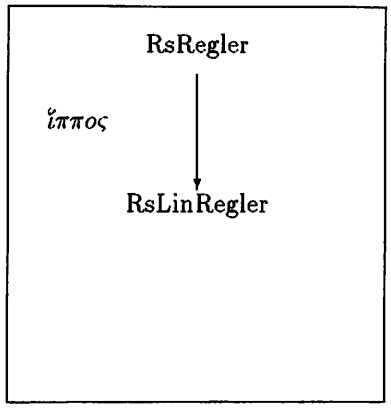
RsLinRegler Basisklasse für lineare Regler

Klassenname: RsLinRegler

Version: 1.3

Basisklasse: RsRegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsLinRegler* ist die abstrakte Basisklasse für alle linearen Reglerklassen.

Methoden:

RsLinRegler: `(int n, int m)` ⟶ `(Cons)`
Konstruktor für linearen Regler:

- **n:** Anzahl der Eingangsgrößen
- **m:** Anzahl der Ausgangsgrößen

RsLinRegler: `()` ⟶ `(Cons)`
Default-Konstruktor

RsLinRegler: `(const RsLinRegler& r)` ⟶ `(Cons)`
Copy-Konstruktor

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsLinRegler.h

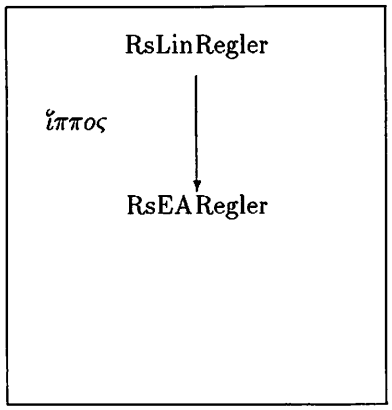
Basisklasse für linearen Ein/Ausgabe-Regler **RsEARegler**

Klassenname: RsEARegler

Version: 1.1

Basisklasse: RsLinRegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsEARegler* ist die abstrakte Basisklasse für alle linearen Reglerklassen mit einem Regelalgorithmus der Form:

$$u(k) = -p[0]u(k-1) - \ldots - p[m-1]u(k-m) + q[0]e(k) + q[1]e(k-1) + \ldots + q[n]e(k-n)$$

Dabei wird die Stellgröße $u(k)$ aus den vorhergehenden Stellgrößen, den Reglerparametern $q[i]$ und $p[j]$, sowie den vergangenen Regelabweichungen $e(k-i)$ berechnet. Obiger Regelalgorithmus gehört zu der folgenden Übertragungsfunktion des linearen Reglers:

$$G(z) = \frac{q[0] + q[1]z^{-1} + \ldots + q[n]z^{-n}}{1 + p[0]z^{-1} + \ldots + p[m-1]z^{-m}}$$

Methoden:

RsEARegler: `(int n, int m)` $\longrightarrow$ `(Cons)`
Konstruktor für linearen Regler: Alle Parameter p und q werden auf Null gesetzt!

- **n:** Grad des Zählerpolynoms der Übertragungsfunktion
- **m:** Grad des Nennerpolynoms der Übertragungsfunktion

RsEARegler: `()` ⟶ `(Cons)`
Default-Konstruktor

RsEARegler: `(const RsEARegler& r)` ⟶ `(Cons)`
Copy-Konstruktor

Regle_: `(const realArray& e, realArray& u)` ⟶ `int`
Erzeugt aus der Regelabweichung e die Stellgröße u nach obigen Regelalgorithmus.

Reset: `()` ⟶ `void`
Setzt Fehler-Vektor und Stellgrößenvektor auf Null zurück.

SetzeParameterNq:
`(int n, real wert)` ⟶ `void`
Der n'te Parameter q (begonnen bei 0) wird auf *wert* gesetzt.

SetzeParameterNp:
`(int n, real wert)` ⟶ `void`
Der n'te Parameter p (begonnen bei 0) wird auf *wert* gesetzt.

HoleParameterNq:
`(int n) const` ⟶ `real`
Liefert *q[n]* (ab 0) des Reglers zurück.

HoleParameterNp:
`(int n) const` ⟶ `real`
Liefert *p[n]* (ab 0) des Reglers zurück.

HoleGradNenner:
`() const` ⟶ `int`
Liefert den Grad des Nenners der Übertragungsfunktion

HoleGradZaehler:
`() const` ⟶ `int`
Liefert den Grad des Zählers der Übertragungsfunktion

SetzeTollFehler: `(real tolE)` ⟶ `void`
Setzt Schranke für tolerierbaren Fehler:
$u(k) = u(k-1) \quad wenn |e(k)| < tolE$

HoleTollFehler: `() const` ⟶ `real`
Liefert Schranke für tolerierbaren Fehler

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsEARegler.h

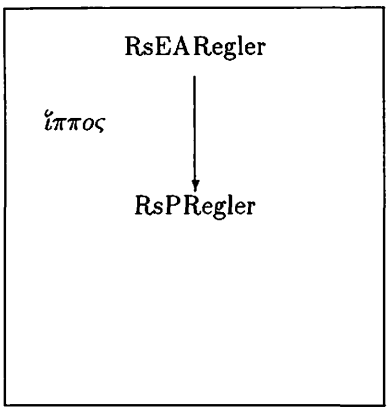
P-Regler **RsPRegler**

Klassenname: RsPRegler

Version: 1.2

Basisklasse: RsEARegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsPRegler* repräsentiert einen P-Regler, mit dem Verstärkungsfaktor K.

Methoden:

RsPRegler: `(real K)` ⟶ `(Cons)`
Konstruktor für P-Regler: Verstärkungsfaktor K wird gesetzt.

RsPRegler: `()` ⟶ `(Cons)`
Default-Konstruktor

RsPRegler: `(const RsPRegler& r)` ⟶ `(Cons)`
Copy-Konstruktor

SetzeK: `(real K)` ⟶ `void`
Setzt Verstärkungsfaktor K

HoleK: `()` ⟶ `real`
Liefert Verstärkungsfaktor

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsPRegler.h

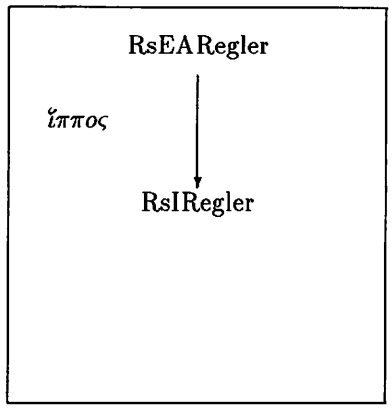
RsIRegler I-Regler

Klassenname: RsIRegler

Version: 1.2

Basisklasse: RsEARegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsIRegler* repräsentiert einen I-Regler mit den Parametern *K*, *TI* und *T0*.

- *K* Verstärkungsfaktor
- *TI* Integrierzeit (Nachstellzeit)
- *T0* Abtastzeit

Methoden:

RsIRegler: `(real K, real TI, real T0)` ⟶ `(Cons)`
Konstruktor für I-Regler

RsIRegler: `()` ⟶ `(Cons)`
Default-Konstruktor

RsIRegler: `(const RsIRegler& r)` ⟶ `(Cons)`
Copy-Konstruktor

SetzeK: `(real K)` ⟶ `void`
Setzt Verstärkungsfaktor

SetzeTI: `(real TI)` ⟶ `void`
Setzt Integrierzeit

SetzeT0: `(real T0)` ⟶ `void`
Setzt Abtastzeit

HoleK: `()` ⟶ `real`
Liefert Verstärkungsfaktor

HoleTI: `()` ⟶ `real`
Liefert Integrierzeit

HoleT0: `()` $\longrightarrow$ `real`

Liefert Abtastzeit

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsIRegler.h

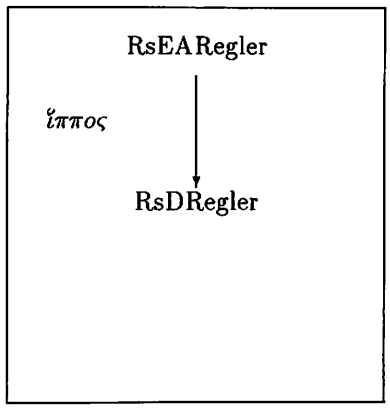
RsDRegler D-Regler

Klassenname: RsDRegler

Version: 1.2

Basisklasse: RsEARegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsDRegler* repräsentiert einen D-Regler mit den Parametern *K*, *TD*, *TI* und *T0*.

- *K* Verstärkungsfaktor
- *TD* Differenzierzeit (Vorhaltezeit)
- *T0* Abtastzeit

Methoden:

RsDRegler: `(real K, real TD, real T0)` ⟶ `(Cons)`
Konstruktor für D-Regler

RsDRegler: `()` ⟶ `(Cons)`
Default-Konstruktor

RsDRegler: `(const RsDRegler& r)` ⟶ `(Cons)`
Copy-Konstruktor

SetzeK: `(real K)` ⟶ `void`
Setzt Verstärkungsfaktor

SetzeTD: `(real TD)` ⟶ `void`
Setzt Differnzierzeit

SetzeT0: `(real T0)` ⟶ `void`
Setzt Abtastzeit

HoleK: `()` ⟶ `real`
Liefert Verstärkungsfaktor

HoleTD: `()` ⟶ `real`
Liefert Differnzierzeit

HoleT0: `()` $\longrightarrow$ `real`
Liefert Abtastzeit

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsDRegler.h

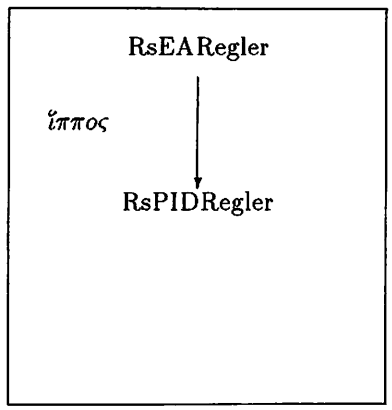
RsPIDRegler PID-Regler

Klassenname: RsPIDRegler

Version: 1.3

Basisklasse: RsEARegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsPIDRegler* repräsentiert einen PID-Regler mit den Parametern *K*, *TD*, *TI* und *T0*.

- *K* Verstärkungsfaktor
- *TI* Integrierzeit (Nachstellzeit)
- *TD* Differenzierzeit (Vorhaltezeit)
- *T0* Abtastzeit

Methoden:

RsPIDRegler: `(real K, real TI, real TD, real T0)` ⟶ `(Cons)`
Konstruktor für PID-Regler

RsPIDRegler: `()` ⟶ `(Cons)`
Default-Konstruktor

RsPIDRegler: `(const RsPIDRegler& r)` ⟶ `(Cons)`
Copy-Konstruktor

SetzeK: `(real K)` ⟶ `void`
Setzt Verstärkungsfaktor

SetzeTI: `(real TI)` ⟶ `void`
Setzt Integrierzeit

SetzeTD: `(real TD)` ⟶ `void`
Setzt Differnzierzeit

SetzeT0: `(real T0)` ⟶ `void`
Setzt Abtastzeit

HoleK: `()` $\longrightarrow$ `real`
Liefert Verstärkungsfaktor

HoleTI: `()` $\longrightarrow$ `real`
Liefert Integrierzeit

HoleTD: `()` $\longrightarrow$ `real`
Liefert Differnzierzeit

HoleT0: `()` $\longrightarrow$ `real`
Liefert Abtastzeit

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsPIDRegler.h

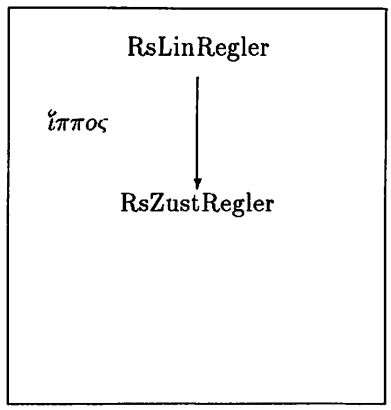
RsZustRegler Zustandsregler

Klassenname: RsZustRegler

Version: 1.3

Basisklasse: RsLinRegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsZustRegler* repräsentiert alle linearen Regler, die durch eine Zustandsbeschreibung gegeben sind. Dabei können sowohl Eingrößensysteme als auch Mehrgrößenregelungen realisiert werden. Die Gleichungen des Zustandsreglers sind gegeben durch

$$x(k+1) = A(k)x(k) + B(k)e(k)$$

und

$$u(k) = C(k)x(k) + D(k)e(k)$$

Es wird also in jedem Zeitschritt der Zustandsvektor x mit Hilfe der Systemmatrix A und der Steuermatrix B aktualisiert. Als Eingabegröße für den Regler dient die Regelabweichung *e(k)*. Die Ausgabegröße *u(k)* wird durch Gewichtung des Zustandes x mit der Ausgabematrix C, sowie der durch D gewichteten Durchspeisung von e erzeugt.

Methoden:

RsZustRegler: `(int dimX, int dimE, int dimU)` ⟶ `(Cons)`
Konstruktor für linearen Zustandsregler mit:

- **dimX** dimensionalem Zustandsvektor
- **dimE** dimensionalem Eingangsvektor
- **dimU** dimensionalem Ausgangsvektor

Alle Matrizen werden mit Null initialisiert.

RsZustRegler: `(const RsZustRegler& r)` ⟶ `(Cons)`
Copy-Konstruktor

RsZustRegler: `(const RsEARegler& r)` ⟶ `(Cons)`
Erzeugt Zustandsregler in Regelungsnormalform aus E/A-Regler

Regle_: `(const realArray& e, realArray& u)` ⟶ `int`
Erzeugt aus der Eingabegröße e und dem aktuellen Zustand x die Stellgröße u.

SetzeA: `(const realArray2d& A)` ⟶ `void`
Setzt Systemmatrix auf A. ACHTUNG: Bei realArray2d ist 1. Parameter die Zeile!

SetzeB: `(const realArray2d& B)` ⟶ `void`
Setzt Steuermatrix auf B. ACHTUNG: Bei realArray2d ist 1. Parameter die Zeile!

SetzeC: `(const realArray2d& C)` ⟶ `void`
Setzt Ausgabematrix auf C. ACHTUNG: Bei realArray2d ist 1. Parameter die Zeile!

SetzeD: `(const realArray2d& D)` ⟶ `void`
Setzt Durchspeisungsmatrix auf D. ACHTUNG: Bei realArray2d ist 1. Parameter die Zeile!

SetzeX: `(const realArray2d& X)` ⟶ `void`
Setzt Zustandsvektor auf X. ACHTUNG: Bei realArray2d ist 1. Parameter die Zeile!

HoleA: `() const` ⟶ `realArray2d`
Liefert Systemmatrix A

HoleB: `() const` ⟶ `realArray2d`
Liefert Steuermatrix B

HoleC: `() const` ⟶ `realArray2d`
Liefert Ausgabematrix C

HoleD: `() const` ⟶ `realArray2d`
Liefert Durchspeisungsmatrix D

HoleX: `() const` ⟶ `realArray2d`
Liefert Zustandsvektor X

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsZustRegler.h

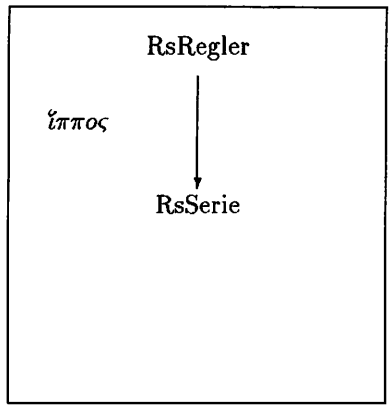
RsSerie Klasse für Serienschaltung von Reglern

Klassenname: RsSerie

Version: 1.2

Basisklasse: RsRegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsSerie* dient zum Hintereinanderschalten von Übertragungsgliedern (bzw. Reglern), die von *RsRegler* abgeleitet sind. Bei den Konstruktoren werden jeweils nur die Zeiger kopiert. Dies bringt den Vorteil mit sich, daß die Parameter der Regler an den bestehenden Instanzen verändert werden können und sofort Einfluß auf den durch Serienschaltung erzeugten Regler haben. Will man die hintereinandergeschalteten Regler jedoch komplett kopieren, so muß nach dem Konstuktor die Methode *deepenShallowCopy()* an der Instanz aufgerufen werden.

Methoden:

RsSerie: `(RsRegler* a,RsRegler* b)` ⟶ `(Cons)`
Konstruktor für in Serie geschaltene Regler:

- **a:** Regler 1
- **b:** Regler 2

RsSerie: `(const RsSerie& r)` ⟶ `(Cons)`
Copy-Konstruktor

Regle_: `(const realArray& e, realArray& u)` ⟶ `int`
Erzeugt aus der Regelabweichung e die Stellgröße u durch Aufruf der Methode*Regle_()* an Regler 1 und 2. Das Ergebnis von Regler 1 dient Regler 2 als Eingabe.

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsSerie.h

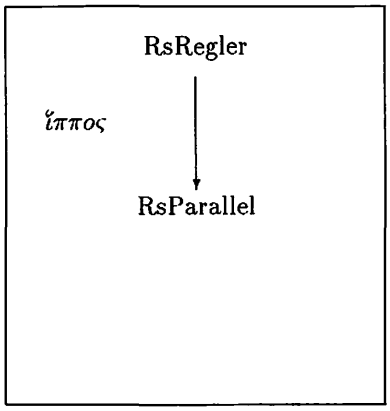
Klasse zum Parallelschalten von Reglern **RsParallel**

Klassenname: RsParallel

Version: 1.2

Basisklasse: RsRegler

Autor: Michael Holzheu, (mlholzhe)



Beschreibung:

Die Klasse *RsParallel* dient zum Parallelschalten von Übertragungsgliedern (bzw. Reglern), die von *RsRegler* abgeleitet sind. Bei den Konstruktoren werden jeweils nur die Zeiger kopiert. Dies bringt den Vorteil mit sich, daß die Parameter der Regler an den bestehenden Instanzen verändert werden können und sofort Einfluß auf den durch Parallelschaltung erzeugten Regler haben. Will man die parallelgeschalteten Regler jedoch komplett kopieren, so muß nach dem Konstuktor die Methode *deepenShallowCopy()* an der Instanz aufgerufen werden.

Methoden:

RsParallel: `(RsRegler* a,RsRegler* b)` ⟶ `(Cons)`
Konstruktor für parallelgeschaltete Regler:

- **a:** Regler 1
- **b:** Regler 2

RsParallel: `(const RsParallel& r)` ⟶ `(Cons)`
Copy-Konstruktor

Regle_: `(const realArray& e, realArray& u)` ⟶ `int`
Erzeugt aus der Regelabweichung e die Stellgröße u durch Aufruf der Methode*Regle_()* an Regler 1 und 2. Die Ergebnisse werden dann addiert.

Automatisch extrahiert aus der Datei /home/mlholzhe/STUDIENARBEIT/include/RsParallel.h

Anhang B

Aufrufe von Matlab und Maple

Hier werden kurz die in dieser Arbeit verwendeten Funktionsaufrufe für die Software–Pakete *Matlab* und *Maple* zusammengestellt.

B.1 Matlab

Die verwendeten Aufrufe stammen alle aus der *Control–Toolbox* von *Matlab*, die zusätzlich zum Basis–Paket erhältlich ist.

Eine s–Übertragungsfunktion im Kontinuierlichen bzw. eine z–Übertragungsfunktion im Diskreten wird durch Definition des Nenners und des Zählers bestimmt. So definiert man z.B die s–Übertragungsfunktion des geschlossenen Regelkreises mit PI–Regler (siehe Kapitel 6.4.3) durch:

```
num = [0.42  0  0];
den = [0.42 0.15 0.15/7];
```

Hat man nun die Übertragungsfunktion des Systems definiert, stehen viele Funktionen zur Verfügung, um Informationen über das System zu gewinnen. Folgende Funktionen wurden im Zusammenhang mit dieser Arbeit verwendet:

- `rlocus(num,den)`: Bestimmung der Pol– und Nullstellen des Systems
- `[d]step(num,den)`: Sprungantwort des Systems
- `[d]lsim(num,den,U,T)`: Simulation des Systems bei vorgegebenem Eingangsverlauf
  - `U`: Vektor mit Eingangswerten
  - `T`: Dazugehöriger Zeitvektor
- `[d]bode(num,den)`: Bode–Plot des Systems

Die Funktionen `dstep`, `dlsim` und `dbode` erwarten als Argumente die z–Übertragungsfunktion des Systems. Für `step`, `lsim` und `bode` muß die s–Übertragungsfunktion angegeben werden. Alle obigen Aufrufe funktionieren auch mit der Zustandsraumdarstellung des Systems.

Um z.B. den in Bild 6.10 dargestellten simulierten Verlauf der Regelabweichung zu erhalten, kann folgendes kurzes Programm verwendet werden:

```
num = [0.41 0];      % Zaehler von Gz
den = [0.41 0.2];    % Nenner  von Gz

U = eye(1,500);      % (1xn)-Vektor
T = eye(1,500);      % (1xn)-Vektor
zeit = 25;           % 25 Sekunden-Simulation
startSprung = 0.9;   % initialer Sprung auf 0.9
TL = 24;             % Lok braucht 24 Sekunden fuer halbe Bildbreite!

% Belegung der Vektoren:
% ======================

for i=1:500,
  T(1,i)  = (zeit*i)/500;
  U(1,i) = startSprung + (zeit*i)/(TL*500);
end

% Start der Simulation:
% =====================

lsim(num,den,U,T);
```

B.2 Maple

In dieser Arbeit wurden die Aufrufe für die Laplace–Transformtion sowie die inverse Laplace–Transformation verwendet:

- Laplace–Transformation: `laplace(f(t), t, s);`
- Inverse Laplace–Transformation: `readlib(laplace):invlaplace(F(s),s,t);`