UNIVERSITÄT LEIPZIG

FAKULTÄT FÜR MATHEMATIK UND INFORMATIK

INSTITUT FÜR INFORMATIK

DIPLOMARBEIT

ADAPTIVE MATHEMATISCHE MODELLIERUNG UND ENTWICKLUNG EINES ECHTZEITSYSTEMS ZUR OPTIMALEN KONTROLLE EINES CHEMISCHEN REAKTORS

VORGELEGT VON OLAF LUMMER

AM 26. AUGUST 1997

BETREUER: PRIV.-DOZ. DR. RALF DER

Danksagung

Einer Diplomarbeit sieht man die Mühe, die mit ihrer Fertigstellung verbunden war, nur selten an. Ebensowenig sieht man meist, wie viele Menschen letztendlich zum Gelingen beigetragen haben. Es ist mir deshalb eine angenehme Aufgabe, diesen Menschen hier meinen Dank auszusprechen:

An vorderster Stelle möchte ich meinen Betreuer, Herrn Priv.-Doz. Dr. Der nennen, der mich auf organisatorischem wie wissenschaftlichem Gebiet hervorragend unterstützte. Sein persönliches Engagement und seine große Bereitschaft, auf meine Probleme einzugehen, waren maßgeblich für den Erfolg dieser Arbeit.

Der Firma Wacker Chemie GmbH in Burghausen danke ich für die finanzielle Unterstützung sowie für die Möglichkeit, meine Diplomarbeit über ein Thema aus der betrieblichen Praxis durchführen zu dürfen. Innerhalb dieser Firma gibt es zwei Mitarbeiter, denen ich zu besonderem Dank verpflichtet bin:

> Herrn Dipl.-Ing. Th. List, auf dessen Vorschlag diese Arbeit zurückgeht, und der viel seiner kostbaren Zeit opferte, um dieses Projekt in die Realität umzusetzen, sowie
>
> Herrn Dr. Dempf, Betriebsleiter und unter anderem für den ACAC-Reaktor verantwortlich, der keine Mühe scheute, mir die Praxis „seines“ Reaktors nahezubringen.

Zum Abschluß möchte ich – stellvertretend für alle, die mich während dieser Arbeit mit Vorschlägen und Korrekturen unterstützt haben – Herrn Jan Leipold danken, der die (hoffentlich) letzten grammatikalischen Fehler in großen Teilen dieser Arbeit aufspürte.

Leipzig, im August 1997

Inhaltsverzeichnis

Symbolverzeichnis

$\mathbf{I}$	Eingangsvariablen (gemessene Werte)
$\mathbf{x}$	Zustandsvariablen (gemessene Werte)
$\mathbf{O}$	Ausgangsvariablen (gemessene Werte)
$\mathbf{a}$	Systemparameter
N	Anzahl der Zustandsvariablen des Systems
NI	Anzahl der Inputvariablen des Systems
NA	Anzahl der Systemparameter
f	allg. Funktion, rechte Seite der expliziten Bestimmungsgleichg. des dyn. Systems
$\mathbf{x}_n$	Vektor der Zustandsvariablen zum Zeitpunkt t_n
$\mathbf{I}_n$	Vektor der Inputvariablen zum Zeitpunkt t_n
$\mathbf{I}', \mathbf{x}', \mathbf{a}'$	wahre Werte der Eingangs- und Zustandsvariablen sowie der Systemparameter
$\hat{\mathbf{I}}', \hat{\mathbf{x}}', \hat{\mathbf{a}}'$	Schätzwerte der Eingangs- und Zustandsvariablen sowie der Systemparameter
$\mathrm{v}(\hat{x}_n)$	Vorwärtsschätzung ausgehend von $\hat{x}_n$
$\mathrm{v}_i(\hat{\mathbf{x}}_n)$	partielle Ableitung der Vorwärtsschätzung von $\hat{x}_n$ nach $\hat{x}_{n,i}$
$\mathrm{v}_{a\,i}(\hat{\mathbf{x}}_n, \hat{\mathbf{a}})$	partielle Ableitung der Vorwärtsschätzung von $\hat{x}_n$ nach a_i
$\mathrm{r}(\hat{x}_n)$	Rückwärtsschätzung ausgehend von $\hat{x}_n$
E	Fehlerfunktion für den Gradientenabstieg
ε	Anpaßrate (Schrittweite des Gradientenabstiegs)
ε_i	dynamische Anpaßrate für die Position i im Zeitfenster
K	Gewichtung für zusätzlichen Einfluß der Meßwerte auf den Gradientenabstieg
$u_{n,i}$	Updatewert für die Veränderung von $\hat{x}_{n,i}$
$\mathbf{L}_{n,i}$	Zwischenterme im Runge-Kutta-Verfahren
h	Schrittweite für Approximationsterme
W	Breite des Zeitfensters
M	festgelegte Anzahl von Iterationen je Fensterschritt
$\mathbf{b}$	systematische Fehler (Bias) der Messungen der Zustandsvariablen
f_i^b	modifizierte Systemfunktionen mit Bias als zusätzlichem Parameter
x_i^b	Meßwerte ohne Einfluß der zufälligen Fehler (nur biasbehaftet)
z	Zeitaufwand

Einführung

Die vorliegende Arbeit befaßt sich mit der Vorstellung eines neuen Ansatzes auf dem Gebiet der adaptiven mathematischen Modellierung und seiner Anwendung auf ein praktisches Problem aus der Tätigkeit der Wacker Chemie GmbH.

Für Zwecke der optimalen Steuerung eines Reaktors zur Erzeugung von Azetylazeton war dort ein Simulationssystem auf der Grundlage eines mathematischen Modells erstellt worden. Beim Einsatz dieses Systems zeigten sich jedoch größere quantitative Abweichungen zwischen dem vom Simulationssystem vorhergesagten und dem tatsächlichen Verhalten des Reaktors. Als Ursache dieses Problems vermutete man sowohl systematische Abweichungen bei den Messun gen am Reaktor als auch Fehler bei der Identifikation gewisser Parameter des Modells.

Ziel dieser Arbeit sollte es daher sein, die Grundlagen für ein Pascal-basiertes On-line-System zu schaffen, das durch Korrektur der Meßwerte und Anpassung der Modellparameter mit Mitteln der Data Reconciliation die Qualität der Simulation (und somit auch die Effektivität der Steue rung) erhöhen soll.

Ursprünglich war zu diesem Zweck eine Implementierung des von Liebman et al. (1992) vorgestellten NDDR-Algorithmus[1] im Gespräch, aufgrund eines Vorschlags von Dr. Der kam jedoch der erwähnte neue Algorithmus zum Einsatz. Durch diese Entscheidung ergaben sich zwei wesent liche Aufgabenbereiche für die Diplomarbeit:

- Test und Weiterentwicklung des neuen Algorithmus
- Implementierung des Algorithmus im Rahmen des Simulationssystems

Bei der Darstellung der Ergebnisse in letzterem Aufgabenbereich mußte jedoch Rücksicht auf den Schutz von Betriebsgeheimnissen der Wacker Chemie GmbH genommen werden, eine entsprechende Verpflichtung war Teil meines Diplomandenvertrages.

Aus diesem Grund legte ich bei der Niederschrift dieser Arbeit die Betonung mehr auf die erste Teilaufgabe, was sich auch in der Gliederung widerspiegelt:

Das *Kapitel 1: Grundlagen* stellt wichtige Begriffe und Konventionen bereit.

Das *Kapitel 2: Ein neuer Algorithmus zur Data Reconciliation* befaßt sich mit der Darstellung des er wähnten neuen Algorithmus. Zunächst wird seine Funktionsweise für einfache Anwendungs fälle erläutert, anschließend werden einige Modifikationen vorgestellt, die es erlauben, auch komplexere Probleme zu lösen.

Das *Kapitel 3: Diskussion des Algorithmus* faßt die wichtigsten Eigenschaften und Leistungen des Algorithmus zusammen.

---

[1] NDDR = nonlinear dynamic data reconciliation

Das *Kapitel 4: Praktisches Vorgehen* ist in zwei Abschnitte unterteilt, die je sich jeweils mit einem der genannten Aufgabenbereiche beschäftigen:

Der *Abschnitt 4.1: Test und Verbesserung des Verfahrens* beschreibt die Experimente, die ich zur Erprobung und Verbesserung des Algorithmus durchgeführt habe und gibt so auch einen Überblick über die ständige Weiterentwicklung des Verfahrens.

Der *Abschnitt 4.2: Data Reconciliation am ACAC-Reaktor* schildert die Aufgaben und Probleme beim Einsatz des Algorithmus am Modell des Reaktors sowie ihre Lösung.

Das *Kapitel 5: Zusammenfassung* gibt in kurzer Form einen Überblick über die Ergebnisse dieser Diplomarbeit.

Im *Anhang* schließlich finden sich mathematische Herleitungen, Programmbeschreibungen und Quelltexte, die im Hauptteil der Arbeit keinen Platz fanden.

1 Grundlagen

1.1 Data Reconciliation

Im Bereich des technischen Einsatzes von Computern und computergestützten Systemen kommt der Verarbeitung von Meßwerten eine große Bedeutung zu. Ob für Zwecke der Überwachung, Steuerung oder Analyse, stets ist es notwendig, durch Messungen ein möglichst genaues Bild über den Zustand und das Verhalten eines real existierenden Systems (des *Objektsystems*) zu erhalten. Solche Messungen sind jedoch stets mit Fehlern behaftet:

- grobe Fehler
- systematische Fehler
- zufällige Fehler

Ziel der *Data Reconciliation* ist nun das Erkennen und Beseitigen derartiger Fehler (auf grobe Fehler wird jedoch im Zusammenhang dieser Arbeit nicht näher eingegangen). Dabei werden Abhängigkeiten zwischen den Ergebnissen verschiedener Messungen ausgenutzt.

Der einfachste Fall solcher Abhängigkeiten liegt bei redundanten Messungen vor, beispielsweise bei physikalischen Experimenten durch häufige Wiederholung eines Versuchs. Mit Methoden der Ausgleichsrechnung kann hier zumindest der Einfluß zufälliger Fehler auf das Ergebnis erkannt und vermindert werden.

In technischen Anwendungen dagegen sind solche redundanten Messungen meist nicht durchführbar. Die betrachteten Systeme sind in der Regel offene dynamische Systeme, deren Verhalten darüber hinaus häufig nichtlinear ist. Messungen können hier nur in gewissen Zeitabständen wiederholt werden, wobei sich der Zustand des Objektsystems zwischen den Meßzeitpunkten ändern kann.

Zwar existieren auch in diesem Fall Abhängigkeiten zwischen den Ergebnissen verschiedener Messungen, jedoch sind diese meist komplexer Natur, weshalb hier vor allem der Einsatz von modellbasierten Verfahren erfolgversprechend ist.

Ein solches modellbasiertes Verfahren, welches nicht nur in der Lage ist, mit Hilfe eines Systemmodells zufällige und systematische Fehler in den Meßwerten zu verringern, sondern auch das Modell selbst dem Objektsystem anzupassen (Systemidentifikation), ist Gegenstand dieser Arbeit und wird im Kapitel 2 näher erläutert.

1.2 Struktur des Modells

Es erscheint sinnvoll, die Menge der betrachteten Systeme einigen Einschränkungen zu unterwerfen. Nur endlichdimensionale zeitkontinuierliche dynamische Systeme sollen betrachtet werden. Ein solches System besitzt drei Typen von Attributen (nach Möller 1992):

$\mathbf{I}'$ *Eingangs- (Input-)variablen*
sind in Betrag und zeitlichem Verlauf unabhängig vom betrachteten System

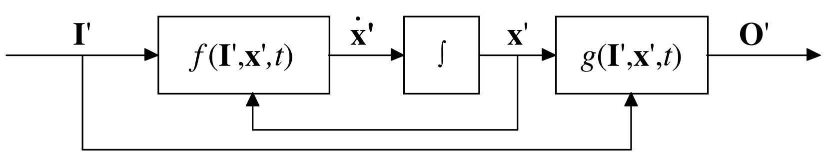
$\mathbf{x}'$ *Zustands- (State-)variablen*
beschreiben vollständig den aktuellen Zustand des Systems

$\mathbf{O}'$ *Ausgangs- (Output-)variablen*
werden durch das System erzeugt, üben aber keinen Einfluß auf das System aus

Die Beziehungen zwischen diesen Attributen werden durch das allgemeine mathematische Zustandsmodell beschrieben:

$$\begin{aligned} O_i'(t) &= g_i(\mathbf{I}', \mathbf{x}', t) \\ \frac{\partial x_i'}{\partial t} &= f_i(\mathbf{I}', \mathbf{x}', t) \end{aligned} \tag{1.1}$$

Damit läßt sich die Struktur dieses Modells so darstellen:



In den folgenden Kapiteln werde ich auf die gesonderte Betrachtung der Ausgangsvariablen verzichten, da ihnen in den untersuchten Systemen praktisch keine Bedeutung zukommt. Die gezeigten Verfahren lassen sich jedoch ohne weiteres auch auf Systeme anwenden, in denen die Modellierung solcher Ausgangsvariablen erforderlich ist.

1.3 Simulation

Wie erwähnt, werden nur rein dynamische Systeme betrachtet. Daher ist es möglich, aus der Kenntnis der Zustandsvariablen an einem Zeitpunkt t_0 und der Kenntnis der Eingangsvariablen im Intervall $[t_0, t_1]$ den Zustand des Systems zum Zeitpunkt t_1 vorherzusagen. Dies kann in seltenen Fällen analytisch erfolgen, meist wird jedoch eine Simulation eingesetzt.

In praktischen Anwendungen gibt es hierbei aber eine Reihe von Problemen:

– eventuell können nicht alle Zustandsvariablen gemessen werden
– eventuell können bestimmte Eingangsvariablen nicht oder nicht zu jedem Zeitpunkt gemessen werden
– alle Messungen sind mit Fehlern behaftet
– das Modell beschreibt das reale System nicht vollständig oder nur mit begrenzter Genauigkeit
– numerische Ungenauigkeiten

Diese Faktoren führen dazu, daß sich das Verhalten des Modells von dem des Objektsystems unterscheidet. Um diesen Effekt zu vermindern, können Techniken der Data Reconciliation eingesetzt werden.

1.4 Modellierung von Fehlern

Um die Leistungsfähigkeit der eingesetzten Algorithmen testen zu können, war es notwendig, diese mit Meßergebnissen zu konfrontieren, die mit *bekannten* Fehlern behaftet waren. Bei realen Meßergebnissen ist das kaum zu verwirklichen, so daß auch das Auftreten von Fehlern modelliert werden mußte.

Dazu wurde die Simulation eines geeigneten mathematischen Modells als Objektsystem eingesetzt. Den so ermittelten Werten der Input- und Zustandsvariablen wurden dann Fehler hinzugefügt:

$$x_i(t) = x_i'(t) + b_i' + \varepsilon_i'(t) \qquad \textbf{(1.2)}$$

x_i Meßwert der Zustandsvariablen

x_i' wahrer Wert der Zustandsvariablen

b_i' konstanter Bias der Messung (systematischer Fehler)

ε_i' (weißes) Rauschen in der Messung (zufälliger Fehler)

(analog erfolgt die Formulierung für die Inputvariablen)

Die Darstellung von systematischen Fehlern als einfacher Bias erscheint zunächst simpel, ist aber den Bedingungen der Praxis durchaus angemessen und wird auch in anderen Arbeiten (Liebman et al. 1992) ähnlich gehandhabt.

2 Ein neuer Algorithmus zur Data Reconciliation

In diesem Kapitel soll ein Algorithmus vorgestellt werden, der Probleme der Data Reconciliation und der Systemidentifikation auch unter Beachtung von Echtzeitforderungen sehr effizient lösen kann. Dieser Algorithmus wurde vom Betreuer der Arbeit, Herrn Dr. Der, am Institut für Infor matik der Universität Leipzig entwickelt und im Rahmen dieser Diplomarbeit erstmals getestet und weiter verbessert.

2.1 Anschauliche Darstellung des Verfahrens

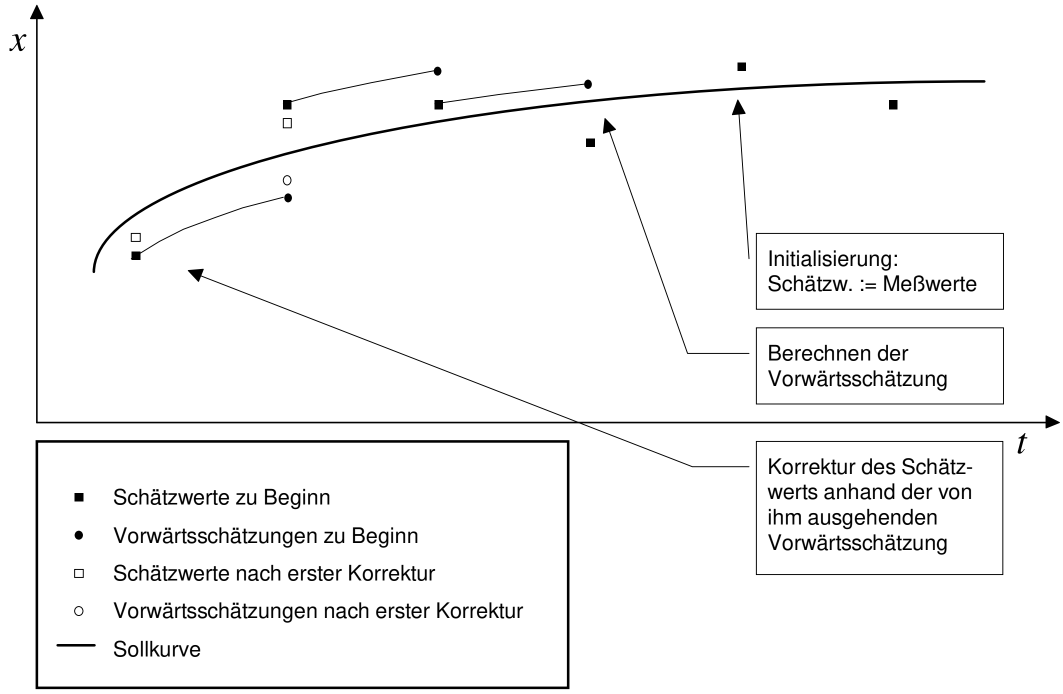
Da der verwendete Algorithmus recht komplex ist, möchte ich zunächst die Wirkungsweise des Grundkonzeptes anschaulich beschreiben und erst in den folgenden Kapiteln zur mathe mati-schen Formulierung übergehen. Dabei werde ich mich anfangs auf die Behandlung eines Systems mit nur einer Zustandsvariablen x beschränken, deren Messungen ein weißes (gaußverteiltes) Rau schen überlagert ist. Ein geeignetes mathematisches Modell für das System sei bekannt.

In einem x-t-Diagramm kann der Zustand des Systems nun durch eine Kurve dargestellt werden (nachfolgend als Sollkurve bezeichnet). Dem Beobachter, der die Messungen durchführt und aus wertet, ist diese Kurve jedoch unbekannt. Er kann nur in regelmäßigen Abständen (Δt) Meß wer te x_n auf nehmen, die gaußverteilt um diese Kurve streuen. Das Ziel des Algorithmus besteht nun da rin, zu jedem Meßwert einen geeigneten *Schätzwert* $\hat{x}_n$ zu finden. Dieser soll möglichst näher an der Soll kurve liegen und so wesentlich genauer als der Meßwert die wahre Größe der gemessenen Vari ablen zum Zeitpunkt der Messung beschreiben.

Der Algorithmus ist iterativer Natur: Zunächst werden die Schätzwerte mit den Meß werten initialisiert. Dann sollen die Schätzwerte schrittweise verbessert werden, indem sie in jedem Schritt ein wenig in Richtung der Sollkurve verschoben werden. Die Richtung dieser Ver schie-bung wird durch Ein satz des Modells bestimmt:

Man initialisiere x jeweils mit einem der Schätzwerte und bestimme mit Hilfe des Modells (1.1) den Wert von x nach der Zeitspanne Δt. Den so erhaltenen Wert bezeichne man als *Vorwärts schät zung* $\mathrm{v}(\hat{x}_n)$. Im einfachsten Fall (alle Schätzwerte liegen auf der Soll kurve) wird diese Vorwärts schät zung gleich dem Schätzwert $\hat{x}_{n+1}$ der nächsten Messung sein, ansonsten ist eine Korrek tur der Schätzwerte erforderlich: [2]

[2] Die Differenz zwischen Vorwärtsschätzung und nächstem Meßwert wird auch als *Ausgangsfehler* der Simulation bezeichnet, Methoden zur Systemidentifikation, die auf der Minimie rung dieses Fehlers beruhen, bezeichnet man als *Ausgangsfehlermethoden* .



Jeder Schätzwert $\hat{x}_n$ wird ein wenig so bewegt, daß möglichst eine bessere Übereinstimmung

- des Schätzwertes $\hat{x}_n$ mit der Vorwärtsschätzung $v(\hat{x}_{n-1})$
- der Vorwärtsschätzung $v(\hat{x}_n)$ mit dem nächsten Schätzwert $\hat{x}_{n+1}$

erreicht wird. Die Berechnung der Vorwärtsschätzungen und die Korrektur der Schätzwerte wird solange wiederholt, bis nur noch minimale Korrekturen auftreten.

Eine anschauliche Vorstellung des Algorithmus kann man erhalten, wenn man sich vorstellt, daß die kurzen Simulationsläufe von einem Meßzeitpunkt zum nächsten jeweils kurze Kurvenstücke im x-t-Diagramm beschreiben, [3] deren Anfangspunkt der betrachtete Schätzwert bildet und deren Endpunkt die zugehörige Vorwärtsschätzung liefert. Zwischen diesen Kurvenstücken gibt es Unstetigkeiten, die vom Algorithmus durch Bewegen der Anfangspunkte (der Schätzwerte) verringert werden. Aufgrund der anfänglich idealen Gaußverteilung der Schätzwerte um die Sollkurve treffen sich die Kurvenstücke schließlich auf dieser Sollkurve – die Unstetigkeiten sind beseitigt.

[3] Bei jedem dieser Kurvenstücke handelt es sich um ein Teilstück einer Kurve aus der Schar der Lösungen der Differentialgleichung für x.

2.2 Grundlegende Funktionsweise: Behandlung zufälliger Fehler bei der Messung von Zustandsvariablen

Zunächst soll der Algorithmus in seiner einfachsten Form dargestellt werden. Es werde angenommen, daß bis auf zufällige Fehler in den Messungen der Zustandsvariablen keine Fehler auf treten und das Modell vollständig u nd exakt sei. Ferner gebe es keine Input-Variablen. Alle Zu standsvariablen werden in regelmäßigen Intervallen (Δt) gleichzeitig gemessen:

$$t_n = t_0 + n\Delta t$$
$$x_{n,i} = x_i(t_n)$$

Das Ziel des Algorithmus besteht nun darin, aus den verrauschten Meßwerten $x_{n,i}$ Schätzwerte $\hat{x}_{n,i}$ zu gewinnen, die möglichst nahe an den wahren Werten $x'_{n,i}$ der Zustandsvariablen liegen. Dabei wird die Tatsache aus genutzt, daß die wahren Werte alle derselben Lösung des Differential gleichungs systems (1.1) angehören müssen, also im Falle der exakten Lösung des Problems:

$$\hat{x}_{n,i} = x'_{n,i} = \mathrm{v}\left(\hat{x}_{n-1,1}, \ldots, \hat{x}_{n-1,N}\right)_i \qquad \textbf{(2.1)}$$

sein muß, wobei v die Lösung des Anfangswertproblems der Differentialgleichung (1.1) darstellt:

$$\mathrm{v}\left(\hat{x}_{n-1,1}, \ldots, \hat{x}_{n-1,N}\right)_i = y_i(t_n) \text{ mit den Anfangswerten: } \quad y_j(t_{n-1}) = \hat{x}_j(t_{n-1}), \quad \frac{\partial y_j}{\partial t} = f_j(\mathbf{y}, t)$$

Deshalb wird $\mathrm{v}\left(\hat{x}_{n-1,1}, \ldots, \hat{x}_{n-1,N}\right)$ auch *Vorwärtsschätzung ausgehend von* $\hat{\mathrm{x}}_{n-1}$ genannt und mit $\mathrm{v}\left(\hat{\mathbf{x}}\left(t_{n-1}\right)\right)$ oder mit $\mathrm{v}\left(\hat{\mathbf{x}}_{n-1}\right)$ bezeichnet. [4]

Somit läßt sich eine Fehlerfunktion E definieren, die nichtnegativ ist und ein globales Minimum (Null) annimmt, wenn die Schätzwerte identisch mit den unverrauschten Meßwerten sind:

$$E = \tfrac{1}{2} \sum_n \sum_i \left(\mathrm{v}\left(\hat{\mathbf{x}}_{n-1}\right)_i - \hat{x}_{n,i}\right)^2 \qquad \textbf{(2.2)}$$

Der Algorithmus ist nun ein Gradientenabstiegsverfahren auf dieser Fehlerfunktion, die deshalb auch als *Energiefunktion* bezeichnet wird.

---

[4] Letztere Kurzschreibweise ist zwar mathematisch fragwürdig (die Bezeichnungen $\hat{x}_i$ und $\hat{\mathrm{x}}_n$ sind hier nicht völlig konsistent), vereinfacht aber die Darstellung erheblich. Um die Über sicht zu wahren, dienen in dieser Arbeit die Indizes i und j stets zur Unterscheidung der Systemvariablen, n und m zur Unterscheidung der Meßzeitpunkte.

Schritt 1: **(Initialisierung)**

für alle i,n: $\hat{x}_{n,i} := x_{n,i}$

Schritt 2: **(Verbesserung der Schätzwerte)**

für alle i,n: $\Delta \hat{x}_{n,i} := -\varepsilon \dfrac{\partial E}{\partial \hat{x}_{n,i}}$

Schritt 3: **(Wiederholung)**

wiederhole Schritt 2, bis keine nennenswerten Änderungen mehr auftreten [5]

Jede Ausführung des Schrittes 2 heiße im folgenden eine *Iteration* des Algorithmus.

Die Konstante ε in Schritt 2 stellt die Schrittweite des Gradientenabstiegs dar ($0 < \varepsilon < 1$) und soll künftig als Anpaßrate bezeichnet werden. Bei hinreichend kleinem Epsilon konvergiert das Verfahren des Gradientenabstiegs gegen ein Minimum der Energiefunktion.

Problematisch hierbei ist jedoch, daß die Energiefunktion (2.2) unendlich viele globale Minima hat: Sie nimmt den Wert Null an, sobald alle $\hat{x}_{n,i}$ zur selben Lösung des Differentialgleichungssystems gehören. Die Meßwerte haben also hier keinen Einfluß auf die Lage der Minima, sie werden lediglich als Startpunkt für den Gradientenabstieg verwendet. Es scheint also fraglich, ob der Algorithmus immer gegen das richtige Minimum der Energiefunktion (die unverrauschten Meßwerte) konvergiert. Um dem zu begegnen, wird die Annahme ausgenutzt, daß die Meßwerte einer Variablen jeweils gaußverteilt um die unverrauschten Werte streuen. Die Energiefunktion wird leicht verändert:

$$E = \tfrac{1}{2}\sum_n \sum_i \left(\left(\mathrm{v}\left(\hat{\mathbf{x}}_{n-1}\right)_i - \hat{x}_{n,i} \right)^2 + \tfrac{1}{K}\left(x_{n,i} - \hat{x}_{n,i} \right)^2 \right) \quad \textbf{mit} \quad K \in \mathrm{R} \text{ konst.}; K \geq 1 \qquad \textbf{(2.3)}$$

So werden unter den Minima der Energiefunktion jene bevorzugt, die nahe an den Meßwerten liegen. Allerdings wird (besonders bei kleinem K) die Lage der Minima geändert: Die Schätzwerte bei Erreichen eines Minimums stellen nicht mehr notwendigerweise (stetige) Lösungen des Differentialgleichungssystems dar. Mit sinkendem K liegen die Minima immer näher an den verrauschten Meßwerten, die „Unstetigkeit" der Ergebniskurven des Algorithmus nimmt zu.

Für $K \to \infty$ erhält man wieder die ursprüngliche Energiefunktion. Nachfolgend wird daher immer mit der erweiterten Energiefunktion (2.3) gearbeitet, die Wahl von K wird im Kapitel 4.1 genauer betrachtet.

Bei den von mir durchgeführten Tests mit diesem Algorithmus zeigte sich ferner, daß es bei großem Epsilon in Schritt 2 auch von Bedeutung ist, in welcher Reihenfolge die Updates der

[5] Alternative Abbruchkriterien sind in Abschnitt 2.4.1 beschrieben.

Schätz werte berechnet und angewendet werden. Die E inführung eines synchronen Updates an stelle dieses Schrittes brachte Verbesserungen im Ergebnis:

Schritt 2a: **(Ermittlung der Updates)**

Bestimme alle $u_{n,i}$ durch: $u_{n,i} := \frac{\partial E}{\partial \hat{x}_{n,i}}$

Schritt 2b: **(Anwendung der Updates)**

Verändere alle Schätzwerte: $\hat{x}_{n,i} := \hat{x}_{n,i} - \varepsilon\, u_{n,i}$

2.3 Berechnung des Updateterms

Bei der Darstellung des Algorithmus im vorigen Kapitel wurde eine Formel für die Para me ter-updates genannt:

$$\Delta \hat{x}_{n,i} = -\varepsilon \frac{\partial E}{\partial \hat{x}_{n,i}}$$

Die Auflösung gestaltet sich etwas langwierig:

$$\begin{aligned}
\Delta \hat{x}_{n,i} &= -\varepsilon \frac{\partial}{\partial \hat{x}_{n,i}} \left(\tfrac{1}{2} \sum_m \sum_j \left(\left(\mathrm{v}(\hat{\mathbf{x}}_{m-1})_j - \hat{x}_{m,j} \right)^2 + \tfrac{1}{K} \left(x_{m,j} - \hat{x}_{m,j} \right)^2 \right) \right) \\
&= -\varepsilon \sum_m \sum_j \tfrac{1}{2} \left(\frac{\partial}{\partial \hat{x}_{n,i}} \left(\left(\mathrm{v}(\hat{\mathbf{x}}_{m-1})_j - \hat{x}_{m,j} \right)^2 + \tfrac{1}{K} \left(x_{m,j} - \hat{x}_{m,j} \right)^2 \right) \right) \\
&= -\varepsilon \sum_m \sum_j \tfrac{1}{2} \left(\frac{\partial}{\partial \hat{x}_{n,i}} \left(\left(\mathrm{v}(\hat{\mathbf{x}}_{m-1})_j - \hat{x}_{m,j} \right)^2 \right) + \tfrac{2}{K} \left(x_{m,j} - \hat{x}_{m,j} \right)(-1)\delta_{ij}\delta_{nm} \right) \\
&= -\varepsilon \left(\tfrac{-1}{K} \left(x_{n,i} - \hat{x}_{n,i} \right) + \sum_m \sum_j \tfrac{1}{2} \left[\frac{\partial}{\partial \hat{x}_{n,i}} \left(\mathrm{v}(\hat{\mathbf{x}}_{m-1})_j - \hat{x}_{m,j} \right)^2 \right] \right) \\
&= -\varepsilon \left(\tfrac{-1}{K} \left(x_{n,i} - \hat{x}_{n,i} \right) + \sum_m \sum_j \left[\left(\mathrm{v}(\hat{\mathbf{x}}_{m-1})_j - \hat{x}_{m,j} \right) \left(\frac{\partial \mathrm{v}(\hat{\mathbf{x}}_{m-1})_j}{\partial \hat{x}_{n,i}} - \delta_{ij}\delta_{nm} \right) \right] \right) \\
&= -\varepsilon \left(\tfrac{-1}{K} \left(x_{n,i} - \hat{x}_{n,i} \right) - \left(\mathrm{v}(\hat{\mathbf{x}}_{n-1})_i - \hat{x}_{n,i} \right) + \sum_j \left(\mathrm{v}(\hat{\mathbf{x}}_n)_j - \hat{x}_{n+1,j} \right) \frac{\partial \mathrm{v}(\hat{\mathbf{x}}_n)_j}{\partial \hat{x}_{n,i}} \right)
\end{aligned}$$

Die Schreibweise der partiellen Ableitungen der Vorwärtsschätzungen werde im folgenden ab-gekürzt:

$$\mathrm{v}_i(\hat{x}_n)_j := \frac{\partial \mathrm{v}(\hat{\mathbf{x}}_n)_j}{\partial \hat{x}_{n,i}}$$

Damit läßt sich die Updateregel etwas übersichtlicher formulieren:

$$\Delta\,\hat{x}_{n,i} = -\varepsilon\left(\tfrac{-1}{K}\left(x_{n,i} - \hat{x}_{n,i}\right) - \left(\mathrm{v}\left(\hat{\mathbf{x}}_{n-1}\right)_i - \hat{x}_{n,i}\right) + \sum_j \left(\mathrm{v}\left(\hat{\mathbf{x}}_n\right)_j - \hat{x}_{n+1,j}\right)\mathrm{v}_i\left(\hat{\mathbf{x}}_n\right)_j \right) \qquad \textbf{(2.4)}$$

Noch besteht allerdings das Problem der Bestimmung der Vorwärtsschätzungen und ihrer partiellen Ableitungen, das nur in Ausnahmefällen analytisch gelöst werden kann. Zur numerischen Lösung kann beispielsweise (für hinreichend kleine Δt)[6] das Runge-Kutta-Verfahren[7] eingesetzt werden:

$$\begin{aligned}
&\mathrm{v}(\hat{\mathbf{x}}_n) = \hat{\mathbf{x}}_n + \frac{\Delta t}{6}\left(\mathbf{L}_{n,1} + 2\mathbf{L}_{n,2} + 2\mathbf{L}_{n,3} + \mathbf{L}_{n,4}\right) \\
&\mathbf{L}_{n,1} = f(\hat{\mathbf{x}}_n, t_n) \\
&\mathbf{L}_{n,2} = f(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} L_{n,1}, t_n + \tfrac{\Delta t}{2}) \\
&\mathbf{L}_{n,3} = f(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} L_{n,2}, t_n + \tfrac{\Delta t}{2}) \\
&\mathbf{L}_{n,4} = f(\hat{\mathbf{x}}_n + \Delta t\, L_{n,3}, t_n + \Delta t)
\end{aligned}$$

Die Formeln zur numerischen Bestimmung der partiellen Ableitungen können hieraus hergeleitet werden, was im Anhang A-1 näher dargestellt wird. Dabei ergeben sich jedoch zwei wichtige Probleme:

- die Kenntnis der partiellen Ableitungen der Funktionen f ist notwendig[8]
- die Berechnungen erfordern hohen Zeitaufwand

Daher habe ich an einfachen Modellen die Auswirkung numerischer Approximationen für die partiellen Ableitungen auf die Leistungsfähigkeit des Algorithmus untersucht und habe festgestellt, daß es problemlos möglich ist, auch einfache Approximationen der Form

---

[6] Wird Δt zu groß, muß das Runge-Kutta-Verfahren als Mehrschrittverfahren eingesetzt werden. Die Vorwärtsschätzung wird dann durch mehrere kleine Integrationsschritte bestimmt. Die partiellen Ableitungen können in diesem Fall analog zum Anhang A-1 durch Anwendung der allgemeinen Kettenregel bestimmt werden.

[7] Einfachere Verfahren (Polygonzug) lieferten in den Versuchen keine zufriedenstellenden Ergebnisse.

[8] Bei einfachen „akademischen“ Modellen stellt dies kaum ein Problem dar, wohl aber beim Einsatz in der Praxis, da dort erheblich komplexere Modelle der Regelfall sind.

$$
\begin{aligned}
&\frac{\partial \mathrm{v}(\hat{\mathbf{x}}_n)_j}{\partial \hat{x}_{n,i}} \approx \frac{\mathrm{v}(\hat{x}_{n,1},\cdots,\hat{x}_{n,i}+h,\cdots,\hat{x}_{n,N})_j - \mathrm{v}(\hat{x}_{n,1},\cdots,\hat{x}_{n,i}-h,\cdots,\hat{x}_{n,N})_j}{2h} \\
&\text{oder} \\
&\frac{\partial \mathrm{v}(\hat{\mathbf{x}}_n)_j}{\partial \hat{x}_{n,i}} \approx \frac{\mathrm{v}(\hat{x}_{n,1},\cdots,\hat{x}_{n,i}+h,\cdots,\hat{x}_{n,N})_j - \mathrm{v}(\hat{x}_{n,1},\cdots,\hat{x}_{n,i},\cdots,\hat{x}_{n,N})_j}{h}
\end{aligned}
\tag{2.5}
$$

zu verwenden. Der Approximationsfehler ist hierbei (näherungsweise) abhängig von h^2 bzw. h. Durch verwenden eines geeigneten Zahlenformates[9] bei der Berechnung kann h kleiner als $10^{-10} * \hat{x}_{n,i}$ gewählt werden, was hinreichend klein erscheint.

2.4 Echtzeitfähigkeit

2.4.1 Zeitfenstertechnik

In der bisherigen Betrachtung des Algorithmus spielten Forderungen der Echtzeitfähigkeit noch keine Rolle – es wurde davon ausgegangen, daß stets die Meßwerte zu allen vergangenen Meß zeitpunkten t_n berücksichtigt und die entsprechenden Schätzwerte verbessert werden.

In technischen Anwendungen jedoch werden Systeme meist über einen sehr langen Zeitraum beobachtet, so daß sich die aufgezeichnete „Geschichte“ des Systems über Hunderte oder Tausende Meßzeitpunkte erstreckt. Alle diese Daten einzubeziehen, bedeutet einen enormen Verbrauch an Speicher und Rechenzeit und ist somit höchstens für Zwecke der Off-line-Analyse von Daten geeignet.

Um den Algorithmus dennoch echtzeitfähig zu gestalten, wurde auf eine Zeitfenstertechnik zu rück gegriffen, die der im Artikel von Liebman et al. (1992) geschilderten nachempfunden ist:

Zunächst wird die Breite $W \in \mathbf{N}$ des Zeitfensters festgelegt,[10] die auch als *Zeithorizont* bezeichnet wird. Der Start des Algorithmus erfolge nun wie bisher, allerdings werden nur die Messungen zu den Zeitpunkten t_0 bis t_{W-1} betrachtet. Die Verbesserung der Schätzwerte wird wiederholt, bis ein bestimmtes Abbruchkriterium (s. u.) eintrifft. Dann erfolgt ein soge nannter *Fensterschritt*:

Die nächste Messung (t_W) wird in die Berechnung mit einbezogen, während die bis dahin letzte betrachtete Messung (t_0) aus dem Zeitfenster fällt. Die zu t_0 gehörenden Schätzwerte werden somit nicht weiter verbessert, die zu t_W gehörenden Schätzwerte zunächst mit den Meßwerten

---

[9] z. B. 64-Bit-Darstellung gemäß IEEE-Standard 754 („doppelt genaue“ Gleitkommadarstellung)

[10] Typischerweise liegt W zwischen 5 und 30.

initialisiert. Die Verbesserung der Schätzwerte wird daraufhin wieder bis zum Eintreten des Abbruchkriteriums fortgesetzt.

Durch Anreihung dieser Fensterschritte wandert das Zeitfenster entlang der t-Achse. Der Rechenzeitaufwand je Fensterschritt kann dabei durch die Wahl der Breite des Zeitfensters gesteuert werden: Kleinere Zeitfenster senken den Aufwand, allerdings schränken sehr kleine Zeitfenster auch die mögliche Qualität des Ergebnisses deutlich ein (Details zur Wahl von W siehe Abschnitt 4.1.4).

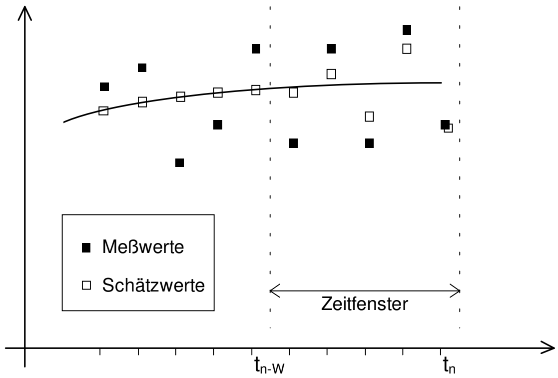


Abbildung 2-1: Momentaufnahme des Zeitfensters

Für die Wahl des erwähnten Abbruchkriteriums gibt es verschiedene Alternativen:

a) Abbruch bei Erreichen einer festgesetzten Anzahl M Iterationen je Fensterschritt
b) Abbruch bei Unterschreiten eines Schwellwertes für E
c) Abbruch bei Eintreffen der nächsten Messung (im On-line-Einsatz)

Kriterium c) gewährleistet die Echtzeitfähigkeit des Algorithmus:

- Der Algorithmus paßt sich an die Rechenzeit an, die zwischen dem Eintreffen zweier Messungen zur Verfügung steht. Auch Schwankungen werden dabei toleriert. Allerdings ist die Qualität der Ergebnisse von der zur Verfügung stehenden Rechenzeit abhängig.
- Etwa zum Zeitpunkt t_{n+W} wird das Update des Schätzwertes $\hat{x}_{n,i}$ beendet. Sollte es aber erforderlich sein (z. B. für Zwecke der On-line-Steuerung), kann der Schätzwert auch zu einem beliebigen früheren Zeitpunkt angefordert werden. Dieser hat bis dahin nur einen Teil der vorgesehenen Iterationen durchlaufen, aber Beobachtungen zeigen, daß die Schätzwerte gerade während der ersten Iterationen besonders starke Verbesserungen erfahren. Beim Abruf solcher „unvollständigen“ Schätzwerte sollte jedoch bis zum Ende der aktuellen Iteration gewartet werden. Die dadurch entstehende Latenzzeit ist nach oben beschränkt [11] (garantierte Antwortzeit).

---

[11] Diese Latenzzeit kann vom Entwickler auch entsprechend den Anforderungen beeinflußt werden. Details siehe Kapitel 3.1.

2.4.2 *Modifikation: dynamische Anpaßrate*

Die eben dargestellte Zeitfenstertechnik bildete die Grundlage einer weiterführenden Überlegung, die sich jedoch nur als bedingt erfolgreich herausstellen sollte: Betrachtet man wie in Abbildung 2-1 die Meß- und Schätzwerte des Fensters zu einem beliebigen Zeitpunkt, wird in der Regel der Schätzwert der zuletzt eingetroffenen Messung („jüngster Schätzwert“) besonders weit von der Lösung abweichen, da er weit weniger Updates erfahren hat als alle anderen Schätzwerte im Zeitfenster. Es scheint also angebracht, diesen (noch besonders fehlerhaften) Schätzwert stärker zu korrigieren als die übrigen (schon besser angepaßten) Schätzwerte. Gleichzeitig sollten die dem jüngsten Schätzwert benachbarten Schätzwerte nur sehr wenig korrigiert werden, um den Einfluß des noch sehr fehlerhaften jüngsten Schätzwertes auf diese zu unterdrücken.

Solche Unterschiede in der Korrekturintensität lassen sich sehr einfach umsetzen, indem die Anpaßrate ε individuell für jeden Schätzwert im Zeitfenster festgelegt wird: ε_1 für den jüngsten Schätzwert, ε_W für den letzten Schätzwert im Zeitfenster. Für die Festlegung der einzelnen ε_i in Übereinstimmung mit diesen Überlegungen wählte ich folgende Formel:

$$\varepsilon_i = \varepsilon \tfrac{1}{2}\left(1 - e^{-4u(i)} + 2e^{-100u(i)}\right) \quad \text{mit:} \quad u(i) = \left(\frac{i-1}{W-1}\right)^2$$

Damit ergibt sich folgender Verlauf der ε_i im Zeitfenster:

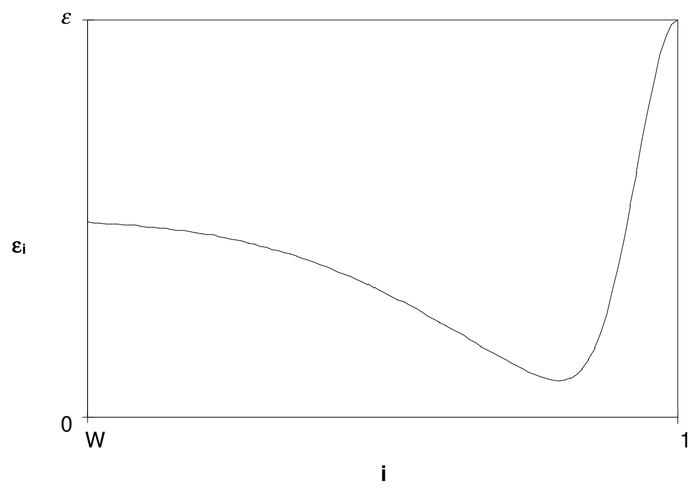


Abbildung 2-2: Verlauf der dynamischen Anpaßrate im Zeitfenster

Diese Modifikation des Verfahrens war in der Tat in einigen Fällen in der Lage, die Qualität der Ergebnisse deutlich zu verbessern und insbesondere „langwellige“ Störungen in den Meßwerten gut zu unterdrücken. Dabei wurde jedoch der Einfluß der Meßwerte deutlich geringer – der Anfangszustand des Modells war von großer Bedeutung. Eine Verbesserung der Ergebnisse durch den Einsatz dieser dynamischen Anpaßrate konnte nur dann erzielt werden, wenn der Anfangszustand des Modells mit dem des Objektsystems nahezu übereinstimmte (siehe dazu auch Abschnitt 4.1.7).

Da eine solche Übereinstimmung der Anfangszustände in der Praxis nur selten realisierbar sein wird, kann die Verwendung der dynamischen Anpaßrate kaum empfohlen werden.

2.5 Einbeziehen von Systemparametern

Während bei der Beschreibung der Grundlagen des Algorithmus noch davon ausgegangen wurde, daß das Modell ein vollständiges und exaktes Bild des Objektsystems liefert, kann diese Annahme in der Praxis selten gehalten werden.

Eines der häufigsten Probleme in diesem Zusammenhang ist die korrekte Parametrisierung mathematischer Modelle (Systemidentifikation). Es ist oft möglich, für die Abläufe im Objektsystem die wichtigsten physikalischen (und/oder chemischen) Gesetzmäßigkeiten zu erkennen und auf dieser Grundlage ein Modell zu formulieren, das die entscheidenden Zusammenhänge qualitativ richtig wiedergibt. Für eine auch quantitativ exakte Modellierung ist aber in der Regel die genaue Kenntnis zahlreicher physikalischer Eigenschaften des Systems (Maße, Massen, thermodynamische Eigenschaften usw. verschiedener Komponenten) erforderlich. Es kann nicht davon ausgegangen werden, daß diese Eigenschaften stets mit der notwendigen Genauigkeit ermittelt werden können,[12] das Verhalten des Modells wird also möglicherweise deutliche Abweichungen vom Verhalten des Objektsystems zeigen.

Da es sich bei der vorgestellten Methode zur Data Reconciliation um ein modellbasiertes Verfahren handelt, können derartige Abweichungen einen starken negativen Einfluß auf die Qualität der Ergebnisse des Algorithmus haben. Es erscheint also angeraten, einen Automatismus vorzusehen, der anhand der Meßdaten Fehler bei in der Einstellung von Systemparametern erkennen und/oder beseitigen kann.

Ein solcher Automatismus läßt sich konsistent zu dem bisher beschriebenen Verfahren hinzufügen. Dazu werden zunächst die Differentialgleichungen aus der grundlegenden Formulierung der Modellstruktur (1.1) erweitert, um den Einfluß der konstanten Systemparameter a'_i darzustellen (auf Inputvariablen werde wieder verzichtet):

$$\frac{\partial x'_j}{\partial t} = f_j(x', a', t) \qquad \textbf{(2.6)}$$

Auch hier sind die wahren Parameter a'_i nicht bekannt, nur der „angenommene Wert“ a_i jedes Parameters. Es sollen wiederum Schätzwerte $\hat{a}_i$ ermittelt werden, die zunächst mit den „angenommenen Werten“ initialisiert werden. Im Verlauf der Meßwertkorrektur sollen diese

[12] Als Beispiel seien Dicke und spezifische Wärmekapazität einer Ablagerungsschicht an den Wänden eines Reaktionsrohres genannt.

Schätzwerte schließlich schrittweise verbessert werden. Auch die Vorwärtsschätzung wird so erweitert zu $\mathrm{v}(\hat{\mathbf{x}}_n, \hat{\mathbf{a}})$.

Erneut wird das Gradientenabstiegsverfahren eingesetzt, wobei wieder die Energiefunktion (2.3) zum Einsatz kommt:

$$\begin{aligned}
\Delta \hat{a}_i &= -\varepsilon \frac{\partial E}{\partial \hat{a}_i} \\
&= -\varepsilon \frac{\partial}{\partial \hat{a}_i} \left(\tfrac{1}{2} \sum_m \sum_j \left(\left(\mathrm{v}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j - \hat{x}_{m,j} \right)^2 + \tfrac{1}{K} \left(x_{m,j} - \hat{x}_{m,j} \right)^2 \right) \right) \\
&= -\varepsilon \sum_m \sum_j \tfrac{1}{2} \left(\frac{\partial}{\partial \hat{a}_i} \left(\left(\mathrm{v}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j - \hat{x}_{m,j} \right)^2 + \tfrac{1}{K} \left(x_{m,j} - \hat{x}_{m,j} \right)^2 \right) \right) \\
&= -\varepsilon \sum_m \sum_j \tfrac{1}{2} \left(\frac{\partial}{\partial \hat{a}_i} \left(\mathrm{v}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j - \hat{x}_{m,j} \right)^2 \right) \\
&= -\varepsilon \sum_m \sum_j \left(\mathrm{v}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j - \hat{x}_{m,j} \right) \frac{\partial \mathrm{v}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j}{\partial \hat{a}_i}
\end{aligned}$$

Die Schreibweise wird wieder vereinfacht:

$$\Delta \hat{a}_i = -\varepsilon \sum_m \sum_j \left(\mathrm{v}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j - \hat{x}_{m,j} \right) \mathrm{v}_{a\,i}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j \qquad \textbf{(2.7)}$$

Die Berechnung der partiellen Ableitungen erfolgt wieder wie im Anhang dargelegt oder durch Approximation (siehe Abschnitt 2.3).

Vergleicht man die eben gewonnene Formel mit der Formel für die Veränderung der Schätzwerte der Zustandsvariablen (2.4), liegt die Vermutung nahe, daß die Updates der Systemparameter wesentlich höhere Beträge annehmen werden. Dies bestätigte sich in der Praxis, was zu unliebsamen Konsequenzen führte:

- Die (betragsmäßig) großen Updates erzwingen häufig die Wahl einer niedrigeren Anpaßrate, um die Konvergenz des Algorithmus sicherzustellen. Das machte die manuelle Einstellung der Anpaßrate erheblich schwieriger. Außerdem bedeuten niedrigere Anpaßraten auch einen Performanceverlust.
- Unter Verwendung dieser Formel wurden die Systemparameter sehr schnell (bereits in wenigen Fensterschritten) angepaßt. Dieses Verhalten führte häufig dazu, daß die Systemparameter auf „langwellige“ Rauschanteile stark ansprachen.[13] (Lagen z.B.

---

[13] Ein ähnliches Verhalten des NDDR ist bei Liebman et al. (1992) beschrieben. Dort wird das Problem gelöst, indem ein Term zur Energiefunktion hinzugefügt wird, der Änderungen der Schätzwerte der Systemparameter bestraft. Es ist dabei notwendig, einen Skalierungsfaktor

alle Messungen einer Zustandsvariablen innerhalb des Zeitfensters unterhalb der wahren Werte, führte das häufig zu einer deutlichen Fehlanpassung der Systemparameter in diesem Fensterschritt.)

Diese Probleme legten es nahe die (im Vergleich zu (2.4) „zusätzliche“) Summation über m durch einen geeigneten Vorfaktor auszugleichen:

$$\Delta \hat{a}_i = -\varepsilon \frac{1}{W} \sum_m \sum_j \left(\mathrm{v}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j - \hat{x}_{m,j} \right) \mathrm{v}_{a\,i}(\hat{\mathbf{x}}_{m-1}, \hat{\mathbf{a}})_j \qquad \textbf{(2.8)}$$

Es konnte gezeigt werden, daß sich mit dieser Modifikation des Verfahrens Systemparameter ein fach und sicher bestimmen lassen. Details dazu sind Ab schnitt 4.1.6 nachzulesen.

2.6 Einbeziehen von Input-Variablen

Wurde in der grundlegenden Erläuterung des Algorithmus noch auf Inputvariablen verzichtet, kommt diese Einschränkung für den praktischen Einsatz kaum noch in Frage, da fast alle realen Sy steme in irgendeiner Form auf Veränderungen der Umwelt reagieren. In technischen Sy ste men findet sich meist eine Anzahl Regelgrößen, deren Betrachtung für die Data Reconciliation unbedingt notwendig ist.

Für die folgenden Betrachtungen werde vorausgesetzt, daß die Dynamik der Eingangsgrößen relativ gering ist, also z. B. innerhalb eines Zeitfensters nur geringe Änderungen stattfinden.

Ein bei Liebman et al. (1992) verwendeter Ansatz sieht vor, die Eingangsgrößen als innerhalb eines Zeitfensters konstant anzunehmen und deshalb dieselben Korrekturmechanismen wie für Kon stanten (Systemparameter) einzusetzen. In Tests konnte ich jedoch feststellen, daß eine Über nahme dieser Vorgehensweise dazu führte, daß die Leistungen des Algorithmus bei stär keren Schwankungen der Eingangsgrößen sehr nachließen.

Da im praktischen Einsatz solche plötzlichen Schwankungen aber kaum auszuschließen sind, war eine Methode erforderlich, die sich gegenüber derartigen Abweichungen toleranter verhielt:

Man beschreibt den Verlauf der Input-Größen in einer Form, die keine Abhängigkeiten vom Sy stem zeigt: $I_i'(t) = h_i(t)$

und verwendet die weitgehende Konstanz innerhalb des Zeitfensters: $\frac{\partial I_i'}{\partial t} \approx 0$

---

für diesen Strafterm anzugeben, was diese Lösung in meinen Augen etwas willkürlicher als die hier vorgestellte Variante erscheinen läßt.

So kann man für die Reconciliation jede Inputvariable I_i' als zusätzliche Zustandsvariable x_{N+i}' darstellen:

$$\frac{\partial x'_{N+i}}{\partial t} = f_{N+i}(\mathbf{x}', t) := 0$$

Auf diese Weise können zur Korrektur des Schätzwertes von Input-Variablen dieselben Formeln (2.4) wie für Zustandsvariablen herangezogen werden. Bei der Berechnung von Updates können dabei einige besondere Eigenschaften (für $j > N$) ausgenutzt werden:

$$v(\hat{\mathbf{x}}_n)_j = \hat{x}_{n,j} \quad \text{sowie} \quad \frac{\partial v(\hat{\mathbf{x}}_n)_j}{\partial \hat{x}_{n,i}} = \delta_{i,j}$$

Allerdings konnte der Algorithmus auch mit dieser Modifikation noch nicht zufriedenstellend auf starke Schwankungen der Inputvariablen reagieren. Deshalb nahm ich eine geringfügige Änderung am Algorithmus vor: In regelmäßige Abständen (zu Beginn jedes Fensterschrittes) wurde ein „Kaltstart" der Schätzungen für die Inputvariablen im Zeitfenster durchgeführt:

$$\hat{x}_{n,j} := x_{n,j} \quad \text{für alle} \quad j > N$$

Details zur praktischen Realisierung können im Abschnitt 4.1.9 nachgelesen werden.

2.7 Einbeziehen von systematischen Fehlern

Wie bereits im Abschnitt 1.4 erwähnt, werden in dieser Arbeit systematische Fehler lediglich in Form eines Bias (konstanter, additiver Fehler) bei der Messung von Zustandsvariablen betrachtet: [14]

$$x_i(t) = x_i'(t) - b_i' + \varepsilon_i'(t)$$

Bei geeigneter Formulierung des Problems läßt sich auch die Rekonstruktion des Bias einer Messung mit den bereits bekannten Mitteln erledigen (gezeigt wird hier nur die Behandlung des Bias von Zustandsvariablen, die Erweiterung auf Inputvariablen ist jedoch unschwierig):

2.7.1 Erweiterung der Differentialgleichungen

Die einfachste Variante, um den Bias in die Überlegungen einzubeziehen, ist die Darstellung der b_i als Parameter des Differentialgleichungssystems durch Erweiterung der Gleichungen:

[14] Im Vergleich zu Formel (1.2) wurde das Vorzeichen des Bias getauscht. Dies kann ohne Beschränkung der Allgemeinheit geschehen: b_i' kann sowohl positiv als auch negativ sein.

$$\frac{\partial x_i'}{\partial t} = f_i(\mathbf{I}', \mathbf{x}', t)$$
$$= f_i(\mathbf{I}', (x_1^b + b_1, x_2^b + b_2, \ldots, x_N^b + b_N), t)$$
$$= f_i^b(\mathbf{I}', (x_1^b, x_2^b, \ldots, x_N^b), b_1, \ldots, b_N, t)$$

Nun kann man die bisher beschriebenen Verfahren einsetzen, um zu Schätzwerten dieser Vari ablen zu gelangen:

$$\hat{x}_{i,n} \approx x_i^b(t_n) = x_i'(t_n) - b_i \quad \text{sowie} \quad \hat{b}_i \approx b_i$$

Diese Erweiterung des Differentialgleichungssystems führt jedoch mit jedem betrachteten Bias zu einem erhöhten Bedarf an Rechenzeit.

2.7.2 Eine schnellere Alternative

Die eben beschriebene Variante der Erweiterung des Differentialgleichungssystems besitzt noch einen entscheidenden Nachteil: Jeder betrachtete Bias wird einen deutlichen Anstieg des Rechen zeitbedarfs bewirken. Das kann jedoch umgangen werden, wie die folgende Überlegung zeigt.

Zunächst werde die Energiefunktion unter Berücksichtigung der $\hat{b}$ erweitert:

$$E = \tfrac{1}{2}\sum_n \sum_i \left(\left(\mathrm{v}\left(\hat{x}_{n-1,1} + \hat{b}_1, \ldots, \hat{x}_{n-1,N} + \hat{b}_N\right)_i - \left(\hat{x}_{n,i} + \hat{b}_i\right) \right)^2 + \tfrac{1}{K}\left(x_{n,i} - \hat{x}_{n,i}\right)^2 \right)$$
$$= \tfrac{1}{2}\sum_n \sum_i \left(\left(\mathrm{v}\left(\hat{\mathbf{x}}_{n-1} + \hat{\mathbf{b}}\right)_i - \left(\hat{x}_{n,i} + \hat{b}_i\right) \right)^2 + \tfrac{1}{K}\left(x_{n,i} - \hat{x}_{n,i}\right)^2 \right)$$

Zur Bestimmung der Updates für die Schätzungen der Zustandsvariablen ermittelt man nun analog zu Abschnitt 2.3:

$$\frac{\partial E}{\partial \hat{x}_{n,i}} = \tfrac{1}{2}\sum_m \sum_j \frac{\partial}{\partial \hat{x}_{n,i}}\left(\left(\mathrm{v}\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{m,j}+\hat{b}_j\right)\right)^2 + \tfrac{1}{K}\left(x_{m,j}-\hat{x}_{m,j}\right)^2\right)$$

$$= \tfrac{1}{2}\sum_j \left[\frac{\partial}{\partial \hat{x}_{n,i}}\left(\mathrm{v}\left(\hat{\mathbf{x}}_{n}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{n+1,j}+\hat{b}_j\right)\right)^2 + \frac{\partial}{\partial \hat{x}_{n,i}}\left(\mathrm{v}\left(\hat{\mathbf{x}}_{n-1}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{n,j}+\hat{b}_j\right)\right)^2\right]$$

$$-\tfrac{1}{K}\left(x_{n,i}-\hat{x}_{n,i}\right)$$

$$= \sum_j \left[\frac{\partial\, \mathrm{v}\left(\hat{\mathbf{x}}_{n}+\hat{\mathbf{b}}\right)_j}{\partial \hat{x}_{n,i}}\left(\mathrm{v}\left(\hat{\mathbf{x}}_{n}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{n+1,j}+\hat{b}_j\right)\right)\right] - \left(\mathrm{v}\left(\hat{\mathbf{x}}_{n-1}+\hat{\mathbf{b}}\right)_i - \left(\hat{x}_{n,i}+\hat{b}_i\right)\right)$$

$$-\tfrac{1}{K}\left(x_{n,i}-\hat{x}_{n,i}\right)$$

Anwendung der allgemeinen Kettenregel:

$$= \sum_j \left[\mathrm{v}_i\left(\hat{\mathbf{x}}_{n}+\hat{\mathbf{b}}\right)_j * \left(\mathrm{v}\left(\hat{\mathbf{x}}_{n}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{n+1,j}+\hat{b}_j\right)\right)\right] - \left(\mathrm{v}\left(\hat{\mathbf{x}}_{n-1}+\hat{\mathbf{b}}\right)_i - \left(\hat{x}_{n,i}+\hat{b}_i\right)\right)$$

$$-\tfrac{1}{K}\left(x_{n,i}-\hat{x}_{n,i}\right)$$

und für die Schätzungen der Bias-Variablen:

$$\frac{\partial E}{\partial \hat{b}_i} = \tfrac{1}{2}\sum_m \sum_j \frac{\partial}{\partial \hat{b}_i}\left(\left(\mathrm{v}\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{m,j}+\hat{b}_j\right)\right)^2 + \tfrac{1}{K}\left(x_{m,j}-\hat{x}_{m,j}\right)^2\right)$$

$$= \tfrac{1}{2}\sum_m \sum_j \left[2\left(\mathrm{v}\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{m,j}+\hat{b}_j\right)\right) * \left(\frac{\partial \mathrm{v}\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j}{\partial \hat{b}_i} - \delta_{ij}\right)\right]$$

Anwendung der allgemeinen Kettenregel:

$$= \sum_m \sum_j \left[\left(\mathrm{v}\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{m,j}+\hat{b}_j\right)\right) * \left(\mathrm{v}_i\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j - \delta_{ij}\right)\right]$$

$$= \sum_m \left[\sum_j \left[\left(\mathrm{v}\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j - \left(\hat{x}_{m,j}+\hat{b}_j\right)\right) * \mathrm{v}_i\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_j\right] - \left(\mathrm{v}\left(\hat{\mathbf{x}}_{m-1}+\hat{\mathbf{b}}\right)_i - \left(\hat{x}_{m,i}+\hat{b}_i\right)\right)\right]$$

Für $K \to \infty$ und bei geeigneter Behandlung der „Ränder“ des Zeitfensters ist damit (vgl. Kapitel 2.5):

$$\frac{\partial E}{\partial \hat{b}_i} \approx \sum_m \frac{\partial E}{\partial \hat{x}_{m,i}} \quad \text{und somit:} \quad \Delta\hat{b}_i \approx \frac{1}{W}\sum_m \Delta\hat{x}_{m,i}$$

Damit beschränkt sich der Rechenaufwand in jeder Iteration, der zusätzlich nötig ist, um die systematischen Fehler zu bestimmen, auf die Additionen zur Bestimmung der $\hat{\mathbf{x}} + \hat{\mathbf{b}}$ sowie auf je W Additionen und eine Multiplikation je betrachtetem Bias. Dieser Aufwand ist verglichen mit dem Aufwand zur Bestimmung der Vorwärtsschätzungen oder deren partieller Ableitungen jedoch so klein, daß man sagen kann, mit dieser Modifikation ließen sich die systematischen Fehler **nahezu ohne zusätzlichen Aufwand** bestimmen.

2.8 Einstellen der Anpaßrate

Wie bereits in Abschnitt 2.2 ausgeführt, ist die richtige Wahl der Anpaßrate entscheidend für die Konvergenz des Verfahrens: Wählt man ε zu groß, wird unter Umständen kein Minimum der Energiefunktion erreicht. Andererseits ist aber auch eine zu niedrige Einstellung von ε verhängnisvoll: Die Anzahl notwendiger Iterationen, um ein bestimmtes Ergebnis zu erreichen, kann so drastisch ansteigen (im Extremfall indirekt proportional zu ε).

Eine gute Einstellung der Anpaßrate ist also recht schwierig zu finden und stark abhängig vom betrachteten System. Um die einfache Handhabung des Algorithmus zu gewährleisten, soll hier eine Möglichkeit der automatischen Einstellung der Anpaßrate angegeben werden (auf die Behandlung von Systemparametern und Inputvariablen wird wieder verzichtet):

Zunächst werden die Updates $u_{n,i}$ der Schätzwerte berechnet und gespeichert. Anschließend werden diese mit einer relativ großen Anpaßrate ε skaliert und zu den Schätzwerten addiert. Nun wird der Wert der Energiefunktion E bestimmt. Anschließend wird geprüft, wie sich eine Verringerung der Anpaßrate auf den Betrag von E auswirkt: Die Anpaßrate wird solange verkleinert, wie sich dadurch eine Verringerung von E erreichen läßt.

Der dazu eingesetzte Algorithmus (er ersetzt Schritt 2b aus dem Kapitel 2.2) sei hier in pascalähnlichem Pseudocode angegeben:

```
const EpsStart=0.8; {größter einzusetzender Wert für Epsilon}
const EpsScale=0.5; {Faktor f. Anpassung von Epsilon; >0 und <=0.5}
const EpsMin=EpsStart/1000; {kleinstes Epsilon (Abbruchkriterium)}
var Epsilon,LastE :real;
...
{Berechnung der u[n,i]}
...
LastE:=GetE();
Epsilon:=EpsStart;
ApplyUpdates(Epsilon); {zunächst mit größtmöglichem Epsilon}

while (GetE() < LastE) and (Epsilon > EpsMin) do begin
  LastE:=GetE();
  ApplyUpdates(-Epsilon);     {Updates wieder zurücknehmen}
  Epsilon:=Epsilon*EpsScale; {Epsilon verkleinern...}
  ApplyUpdates(Epsilon);      {...mit kleinerem Epsilon versuchen}
end;
{wenn letztes Update keine Verbesserung brachte, zurücknehmen}
if GetE()>LastE then AddUpdates(-Epsilon);
...
```

Für den Algorithmus sind zwei zusätzliche Prozeduren erforderlich:

```
function GetE:real; {liefert aktuellen Wert der Energiefunktion E}
procedure ApplyUpdates(d:Real); {für alle n,i: x̂_{n,i}:= x̂_{n,i} - d * u_{n,i} }
```

Anmerkungen:

- Die Konstanten **`EpsStart`**, **`EpsScale`** und **`EpsMin`** können auch anders gewählt werden, eine Anpassung dieser Werte an das betrachtete System ist jedoch nicht notwendig.
- Der obige Pseudocde ist nur als Demonstration gedacht und ist daher nicht besonders effizient. Für den praktischen Einsatz kann der Algorithmus so formuliert werden, daß je Iteration der **`while`**-Schleife nur je ein Aufruf von **`ApplyUpdates`** und **`GetE`** nötig ist.

3 Diskussion des Algorithmus

3.1 Bedarf an Rechenzeit

Ein wichtiges Kriterium für die Anwendbarkeit des Algorithmus in der Praxis (insbesondere in Echtzeit-Anwendungen) stellt der Bedarf an Rechenzeit dar. Aus diesem Grunde sei hier eine Abschätzung der Komplexität des Algorithmus gegeben:

Betrachtet werde die Rechenzeit z_{it}, die während einer Iteration benötigt wird. Diese bestimmt sich aus der Rechenzeit zur Ermittlung der Updates und der Rechenzeit zur Durchführung der Updates:

$$z_{it} = z_{ue} + z_{ua}$$

Bestimmend für diese Rechenzeiten ist jeweils der Zeitaufwand für die Vorwärtsschätzungen. Verglichen damit ist der Zeitaufwand für die übrigen Operationen vernachlässigbar. Definiert man nun:

z_{vor}	Zeitaufwand einer Vorwärtsschätzung $v(\hat{x}_n)_i$ für alle Variablen i
N	Anzahl der Zustandsvariablen
NI	Anzahl der Inputvariablen
NA	Anzahl zu korrigierender Systemparameter
S	Anzahl der Schleifendurchläufe der automatischen Anpaßwertbestimmung (siehe Abschnitt 2.8)
M	Anzahl der Iterationen je Fensterschritt

so erhält man bei Verwendung der einfachsten Approximation der partiellen Ableitungen (Formel (2.5) [15])

$$z_{ue} \approx (W\text{-}1)\, z_{vor} + N(W\text{-}1) z_{vor} + NI(W\text{-}1) z_{vor} + NA(W\text{-}1) z_{vor} = (1+N+NI+NA)\cdot(W\text{-}1)\cdot z_{vor}$$

Der Rechenzeitbedarf der Bestimmung der Anpaßrate bestimmt sich im wesentlichen durch die Berechnung der Energiefunktion. Der Zeitaufwand von **`ApplyUpdates`** (siehe Abschnitt 2.8) kann dagegen vernachlässigt werden:

$$z_{ua} \approx S \cdot (W\text{-}1) \cdot z_{vor}$$

---

[15] Dabei werde die Tatsache ausgenutzt, daß eine der in dieser Formel vorkommenden Vorwärtsschätzungen nicht nochmals berechnet werden muß, da sie auch explizit in der Updateformel erscheint – eine der beiden Berechnungen kann eingespart werden. Somit ist der Zeitaufwand der partiellen Ableitung der Vorwärtsschätzung nach einer Variablen gleich dem Zeitaufwand für die betreffende Vorwärtsschätzung.

Damit erhält man:

$$z_{it} \approx (1+S+N+NA+NI) \cdot (W\text{-}1) \cdot z_{vor}$$

Somit läßt sich also im praktischen Einsatz eine obere Grenze für die Dauer der in Kapitel 2.4.1 erwähnten Latenzzeit einhalten: W sowie obere Grenzen für S und M können vom Entwickler vorgegeben werden, alle anderen vorkommenden Größen hängen nur von der Struktur der Aufgabe ab und sind in der Praxis Konstanten.

Noch zu bestimmen bleibt der Zeitaufwand der Vorwärtsschätzung. In der Regel wird für die Vorwärtsschätzungen das Runge-Kutta-Verfahren eingesetzt. Betrachtet werden hier nur der Zeitaufwand für Multiplikationen z_{mul} sowie der durchschnittliche Aufwand z_f für die Bestimmung des Funktionswertes einer der Funktionen f_i:

$$z_{vor} \approx P \cdot (4 \cdot N \cdot z_f + 5 \cdot N \cdot z_{mul})$$

Bei nicht-trivialen Systemen ist meist $z_{mul} \ll z_f$ und somit:

$$z_{vor} \approx 4 \cdot P \cdot N \cdot z_f$$
$$z_{it} \approx 4 \cdot (1+S+N+NA+NI) \cdot (W\text{-}1) \cdot P \cdot N \cdot z_f$$

und für die Gesamtzeit eines Fensterschrittes:

$$z_{ges} = M \cdot z_{it} \approx 4 \cdot (1+S+N+NA+NI) \cdot (W\text{-}1) \cdot M \cdot P \cdot N \cdot z_f$$

Das heißt, der Algorithmus ist (bezüglich der Anzahl von Zustandsvariablen) von der Ordnung $O(N^2)$. Bezogen auf die übrigen der genannten Einflußgrößen ist der Algorithmus von linearer Ordnung.

Weiterhin wäre auch die Frage von großem Interesse, wie es um den Zeitverbrauch des Algorithmus verglichen mit anderen Algorithmen zur Data Reconciliation (insbesondere dem NDDR-Algorithmus) steht. Leider ist jedoch ein direkter Vergleich zwischen den von mir beobachteten Rechenzeiten und den von Liebman et al. (1992, S. 978) für das NDDR-Verfahren publizierten Werten kaum möglich, da sich die Hardwareplattformen (VaxStation und IBM-kompatibler PC) zu stark voneinander unterscheiden.[16]

Somit kann ein aussagekräftiger Performance-Vergleich nur durch Implementierung der beiden Algorithmen auf derselben Hardwareplattform durchgeführt werden, was aber den Umfang dieser Diplomarbeit gesprengt hätte. Dennoch glaube ich, daß der hier vorgestellte Algo-

---

[16] Verschiedene Benchmarks, die auf beiden Plattformen durchgeführt wurden, kommen zu sehr unterschiedlichen Ergebnissen beim Vergleich zwischen VaxStation und PC. Als Beispiel für solche Unterschiede seien die Veröffentlichungen von Dongarra (1997) und Tomiak (1997) genannt.

rithmus als dem NDDR-Algorithmus in dieser Beziehung mindestens gleichwertig eingeschätzt werden kann.

3.2 Leistungsvergleich mit dem NDDR-Algorithmus

Von großem Interesse bei den Tests des neuen Algorithmus war natürlich die Frage, in welchem Verhältnis seine Ergebnisqualität zu der des NDDR-Verfahrens steht. Bei den letzen Versuchen wurden daher die Experimente reproduziert, deren Resultate von Liebman et al. (a. a. O.) publiziert wurden. Es zeigte sich dabei, daß der Algorithmus in der Lage war, die Resultate des NDDR in gleicher oder besserer Qualität nachzuvollziehen.

Als Beispiel seien hier die Ergebnisse eines Experiments mit der CSTR-Simulation aus Kapitel 4.1.7 vorgestellt. Die Versuchsbedingungen sind bei Liebman et al. (1992, S. 976ff.) näher beschrieben. (Einstellungen für den neuen Algorithmus: M=5, W=5, K=100.000, Epsilon automatisch eingestellt)

	NDDR			neuer Algorithmus					
Bias Fenstergröße Rechenzeit (s)	ohne Bias [17] 5 1,14 ([18])			ohne Bias 5 0,085 ([19])			A_0 mit Bias 5 0,085 ([19])		
Kriterium	MSE vor	MSE nach	MSE Verringerung	MSE vor	MSE nach	MSE Verringerung	MSE vor	MSE nach	MSE Verringerung
A	0,0547	0,0192	64,9%	0,0443	0,0138	68,9%	0,1033	0,0197	80,9%
T	0,0545	0,0160	70,6%	0,0540	0,0148	72,6%	0,0495	0,0117	76,4%
A_0	0,0496	0,4550	-817,3%	0,0495	0,2924	-490,7%	0,0501	0,2825	-463,9%
T_0	0,0537	0,0252	53,1%	0,0551	0,0271	50,8%	0,0466	0,0244	47,6%

[17] Ein entsprechendes Experiment **mit** Bias bei der Messung von A_0 wird bei Liebman et al. (1992, S. 979ff.) zwar beschrieben, konkrete Zahlen zur Ergebnisqualität oder der verbrauchten CPU-Zeit werden dort jedoch nicht aufgeführt.

[18] durchschnittliche CPU-Zeit je Fensterschritt auf VAXStation 3200

[19] durchschnittliche CPU-Zeit je Fensterschritt auf IBM-kompatiblem PC (Prozessor AMD 486DX/2 80MHz)

Die in der Tabelle erkennbare deutliche Verschlechterung des MSE bei der Inputvariablen A_0 ist darauf zurückzuführen, daß die Testbedingungen sprunghafte Änderungen dieser Größe vorsahen – eine Aufgabe, die beiden Algorithmen Probleme bereitet. (siehe auch Kapitel 4.1.9)

3.3 Praxistauglichkeit

Bei der Entwicklung des vorliegenden Algorithmus wurde besonderer Wert auf eine möglichst gute Praxistauglichkeit gelegt. Dies äußert sich in einer Reihe von positiven Merkmalen:

Flexibilität
Mit Hilfe der verschiedenen in Kapitel 2 dargestellten Modifikationen kann der Algorithmus nicht nur zur Reconciliation von Zustands- und Eingangsvariablen, sondern auch zur Identifikation von systematischen Meßfehlern und sogar zur Identifikation von Systemparametern eingesetzt werden. Dies eröffnet eine große Palette von Einsatzmöglichkeiten.

Modellinvarianz
Sofern für die Berechnung der partiellen Ableitungen die Approximationen (2.5) zum Einsatz kommen, tauchen die Differentialgleichungen des Modells in den Updateformeln nicht mehr explizit, sondern nur noch implizit durch die Vorwärtsschätzungen auf. Das Modell kann so zu einer „ Black Box “ werden, lediglich die Möglichkeit der Berechnung von Vorwärtsschätzungen muß in irgendeiner Form gegeben sein. Dies aber trifft auf alle Systeme zu, für die eine dynamische Simulation möglich ist.

einfache Handhabung
Die einfache Handhabung des Algorithmus ergibt sich neben seiner einfachen Gesamtstruktur auch durch die sehr kleine Anzahl an Tuningparametern, für deren Wahl darüber hinaus auch Faustregeln angegeben werden können:

Parameter	Erläuterung	Wahl des Parameters	s. Kapitel
M	Anzahl Iterationen je Fensterschritt	automatisch je nach verfügbarer Rechenzeit	2.4.1 , 4.1.4
W	Fensterbreite	möglichst groß unter Beachtung der zur verfügbaren Rechenzeit	4.1.3 , 4.1.4
K	(inverses) Gewicht für Einfluß der Meßwerte	möglichst groß	4.1.5
ε	Anpaßrate	automatisch	2.8

gute Skalierungseigenschaften
Wie im Abschnitt 3.1 beschrieben, ist der Algorithmus von quadratischer Ordnung bezogen auf die Anzahl der Zustandsvariablen des Systems. Diese Tatsache in Verbindung mit den bei der CSTR-Simulation beobachteten Rechenzeiten läßt die Vermutung zu, daß sich auch praktische

Pro bleme, bei denen eine Vielzahl solcher Variablen auftritt, mit diesem Algorithmus in angemessener Zeit bearbeiten lassen.

Echtzeittauglichkeit

Die Tauglichkeit des Algorithmus für Echtzeitanwendungen wurde bereits im Kapitel 2.4 diskutiert. Als beson de rer Vorteil zeigte sich dabei, daß der Algorithmus die Schätzwerte ite rativ verbessert und so mit prak tisch jederzeit „vorläufige" Schätzwerte bereitstehen, die für Zwecke der On-line-Steuerung verwendet werden können. Außerdem kann der laufende Fen sterschritt so mit kurzer Latenzzeit unterbrochen werden, um auf das Eintreffen einer neuen Messung zu reagieren.

Der wohl eindrucksvollste Beleg für die Praxistauglichkeit des neuen Algorithmus aber ist sicher seine erfolgreiche Anwendung auf die Meßwertkorrektur am ACAC-Reaktor, wie sie im Kapitel 4.2 beschrieben wird. Verglichen beispielsweise mit der von McBrayer (1997) vor gestellten Problemlösung unter Anwendung des NDDR-Verfahrens handelt es sich hier um ein sehr komplexes Objektsystem, was die Leistungsfähigkeit des neuen Algorithmus deutlich unterstreicht.

4 Praktisches Vorgehen

4.1 Test und Verbesserung des Verfahrens

4.1.1 Untersuchung einer Grundidee

Die erste Aufgabe der Diplomarbeit bestand in der Erprobung der Grundidee des von Dr. Der vorgeschlagenen Verfahrens an einfachen Modellen. Diese Grundidee wich allerdings von der unter 2.2 beschriebenen Variante noch in wesentlichen Punkten ab:

- betrachtet wurden nur Systeme mit genau einer Zustandsvariablen x, ohne Inputvariablen oder fehlerhafte Parameter
- es wurden nicht nur Vorwärtsschätzungen $\mathrm{v}(\hat{\mathrm{x}})$, sondern auch Rückwärtsschätzungen $\mathrm{r}(\hat{\mathrm{x}})$ eingesetzt, als Update-Term wurde verwendet: $\Delta\,\hat{x}_{n,i} = \varepsilon\left(\mathrm{v}(\hat{\mathbf{x}}_{n-1})_i + \mathrm{r}(\hat{\mathbf{x}}_{n+1})_i - 2\hat{x}_{n,i}\right)$
- die Vorwärts- und Rückwärtsschätzungen wurden nicht mit dem Runge-Kutta-Verfahren, sondern mit der Eulerschen Methode ermittelt, wodurch sich die Updates äußerst einfach berechnen ließen.

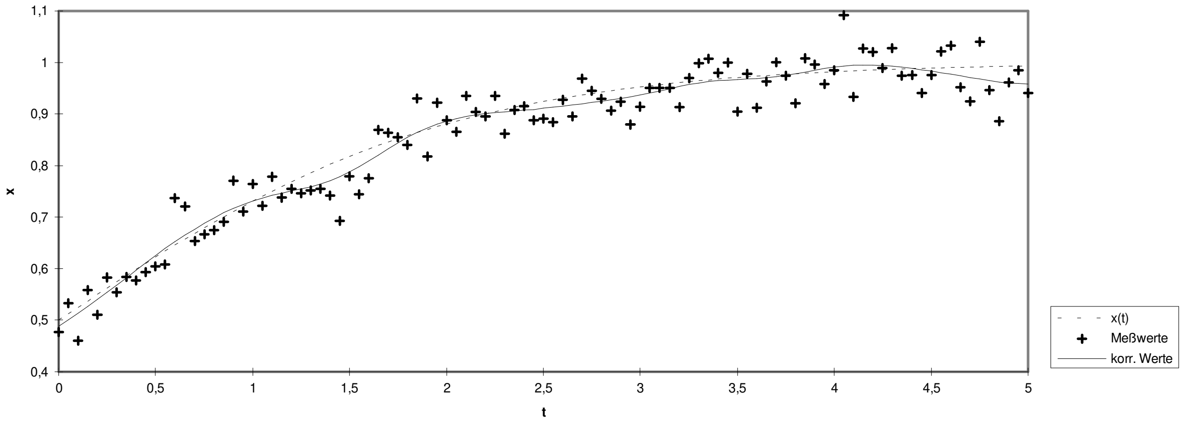
Als betrachtetes System (d. h. sowohl als Objektsystem wie auch als dessen Modell) wurde eine einfache Differentialgleichung eingesetzt, die sich auch analytisch einfach lösen läßt:

$$\frac{\partial x}{\partial t} = x - x^2$$

Für die ersten Experimente setzte ich Microsoft Excel ein, da mit minimalem Programmieraufwand (ca. 60 Zeilen Code und etwa zehn Zellenformeln) komplette Testläufe und grafische Auswertungen möglich waren. Die geringe Performance dieser Lösung[20] stellte zunächst kein Problem dar, weil die Datenmengen klein und die Updateformeln äußerst einfach waren.

Die Ergebnisse dieser Experimente zeigten die prinzipielle Funktionsfähigkeit des vorläufigen Ansatzes. Als Beispiel sei die Korrektur von 100 Meßwerten mit 50 bzw. 500 Iterationen des Algorithmus (Einstellung für Epsilon: 0,2) gezeigt:

[20] Die Erstellung von Programmen unter Excel erfolgt in einem Basic-Dialekt, der zur Laufzeit interpretiert wird. Dies führt zu einem großen Overhead beim Programmlauf.



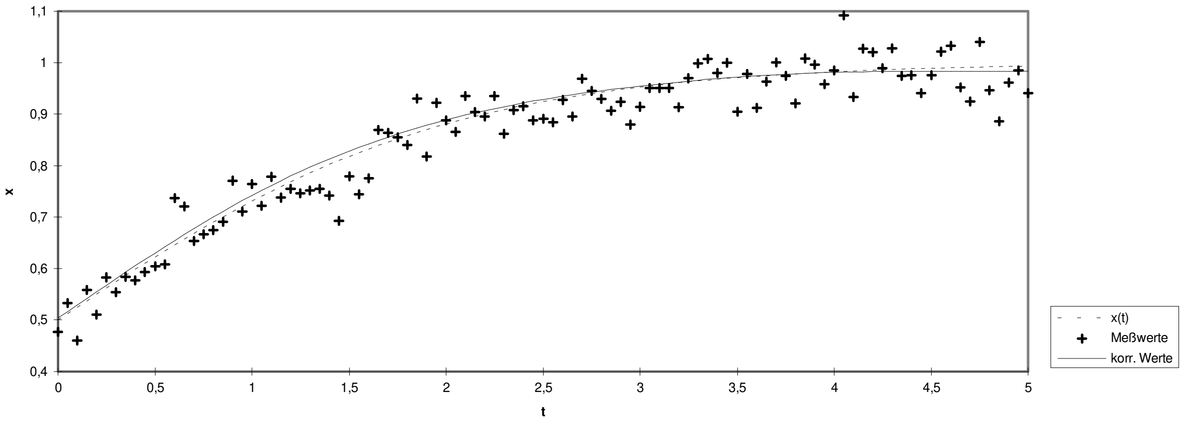


Abbildung 4-1: Korrektur verrauschter Meßwerte nach 50 bzw. 500 Schritten

Gleichzeitig wurden jedoch erste Probleme sichtbar:

Eines dieser Probleme lag darin, daß als Ergebnis nicht immer die Kurve der unverrauschten Meßwerte erreicht wurde, in wenigen Fällen konvergierten die Schätzwerte auch gegen andere Kurven aus der Schar der Lösungen der Differentialgleichung.

Ein weiteres Problem lag in der mangelhaften numerischen Stabilität des Algorithmus, die sich besonders zeigte, wenn die Meßwerte nicht fehlerbehaftet waren. Der Algorithmus hätte die Schätzwerte in diesem Falle nicht verändern dürfen, sollten diese doch einen Fixpunkt in seinem Konvergenzverhalten darstellen. Statt dessen schaukelten sich jedoch (besonders bei $\varepsilon > 0{,}4$) numerische Fehler auf:

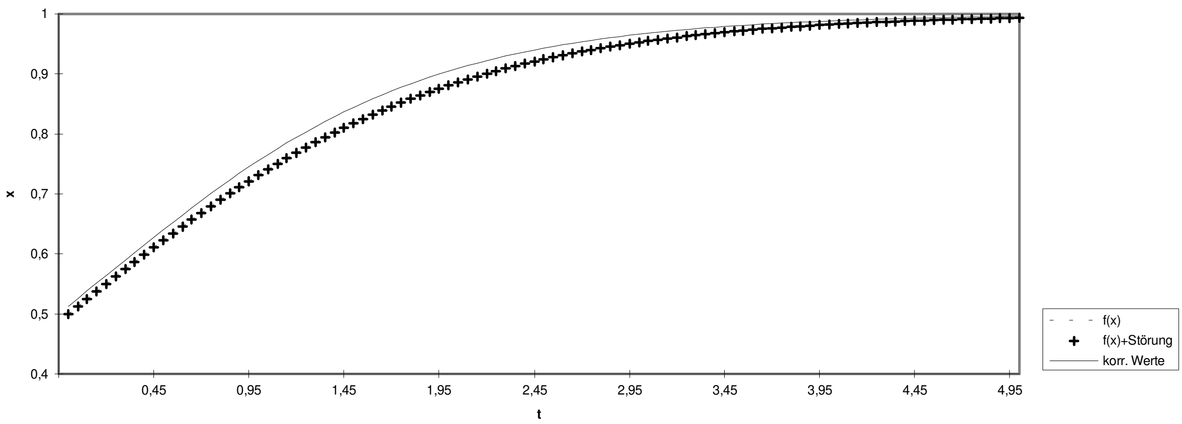


Abbildung 4-2: Korrektur unverrauschter Meßwerte nach 250 Schritten ($\varepsilon = 0{,}5$)

Aufgrund dieser Instabilitäten wurde schließlich das Runge-Kutta-Verfahren eingesetzt, um die Vorwärts- und Rückwärtsschätzungen zu bestimmen. Dies erwies sich als sehr wirkungsvoll, wenn gleich die Rechengeschwindigkeit der Excel-Makros stark abnahm.

4.1.2 Überarbeitung des Verfahrens

Auf Grundlage der Ergebnisse dieser Tests überarbeitete Dr. Der das Verfahren vollständig und for mulierte es sc hließlich so wie im Kapitel 2.2 beschrieben. Wegen der oben beschriebenen numerischen Probleme in den vorangegangenen Tests schlug ich die Verwendung des Runge-Kutta-Verfahrens für die Vorwärtsschätzungen vor und bestimmte die nötigen par tiellen Ablei tungen (siehe Anhang A-1).

Anschließend mußte noch ein Testprogramm für den Algorithmus entwickelt werden. Eine nähere Beschreibung dieser Software findet sich im Anhang.

Da erste Tests mit diesem Programm und dem überarbeiteten Verfahren erfolgreich verliefen (die Ergebnisse der Experimente mit dem ursprünglichen Verfahren konnten in gleicher oder besse rer Qualität reproduziert werden), wurde dieser Ansatz weiterverfolgt.

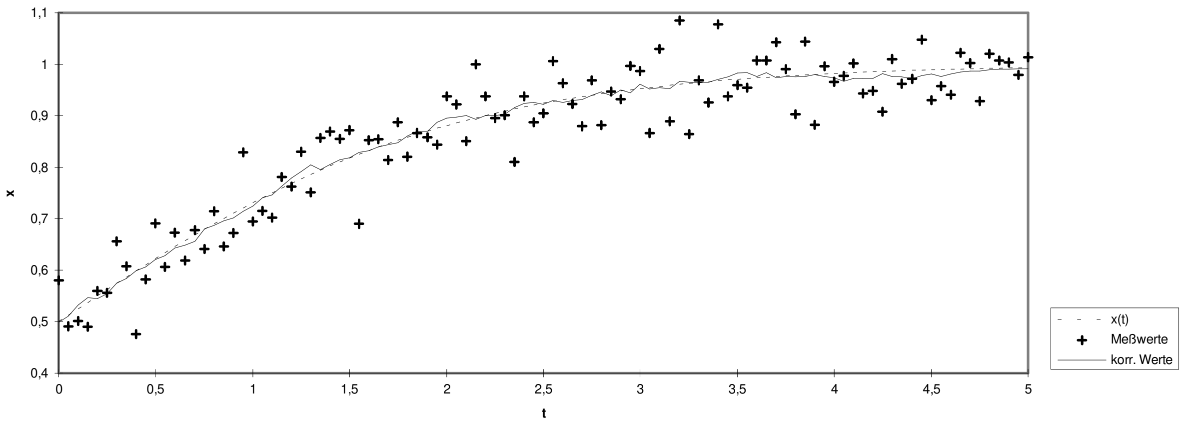
Die nächsten Versuche befaßten sich vor allem mit der Einführung der Zeitfenstertechnik wie im Abschnitt 2.4, allerdings wurden noch einige Einschränkungen vorgenommen:

- Auf das synchrone Ausführen der Updates wie in Kapitel 2.2 wurde verzichtet.
- Die Versuche wurden off-line durchgeführt, die Meßwerte wurden aus einer Datei ent nommen. Daher wurde das Zeitfenster nicht beim Eintreffen einer neuen Messung nach vorn geschoben, sondern jeweils nach Durchführung von M Iterationen. Die Wahl der Konstanten M ist jeweils angegeben.

Dabei konnte eine Reihe interessanter Erkenntnisse gewonnen werden:

4.1.3 *Einfluß der Fensterbreite W*

Die Breite des Zeitfensters übt einen entscheidenden Einfluß auf die erreichbare Qualität der Ergebnisse aus. Wählt man M ausreichend groß (s. o.), läßt sich beobachten, daß der Algorithmus bei kleinem Zeitfenster langwellige Störungen in den Meßdaten deutlich schlechter ausgleicht: (M=100; Epsilon=0,2; K=100.000; W=5 bzw. W=20)



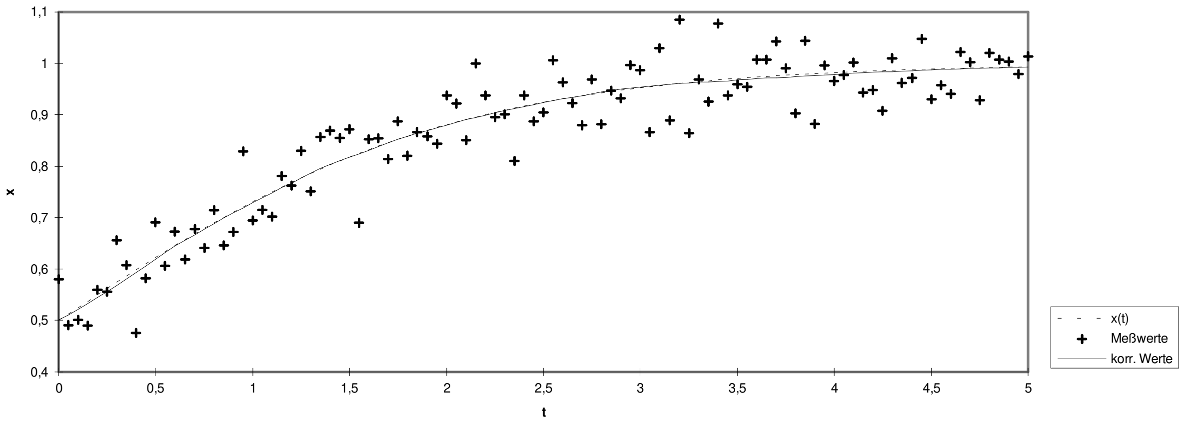


Abbildung 4-3: Korrektur verrauschter Meßwerte mit W=5 und W=20

4.1.4 *Fensterbreite und Iterationsanzahl*

Die Ergebnisse der Anpassung mit verschiedenen Fenstergrößen legen nahe, ein möglichst großes Zeitfenster zu verwenden. Dies hat jedoch auch einen hohen Zeitaufwand bei den Rechnungen jedes einzelnen Fensterschritts zur Folge. Aus den Berechnungen im Abschnitt 3.1 geht hervor, daß Zeitaufwand in jedem Fensterschritt etwa proportional zu W·M ist. Da im On-line-Einsatz die zwischen zwei Messungen zur Verfügung stehende Rechenzeit aber eine feste Vorgabe darstellt, ist in diesem Fall auch W·M konstant vorgegeben. Wie sollen nun aber W und M gewählt werden? (Beispiele mit K=100.000; Epsilon=0,2; W·M=100):

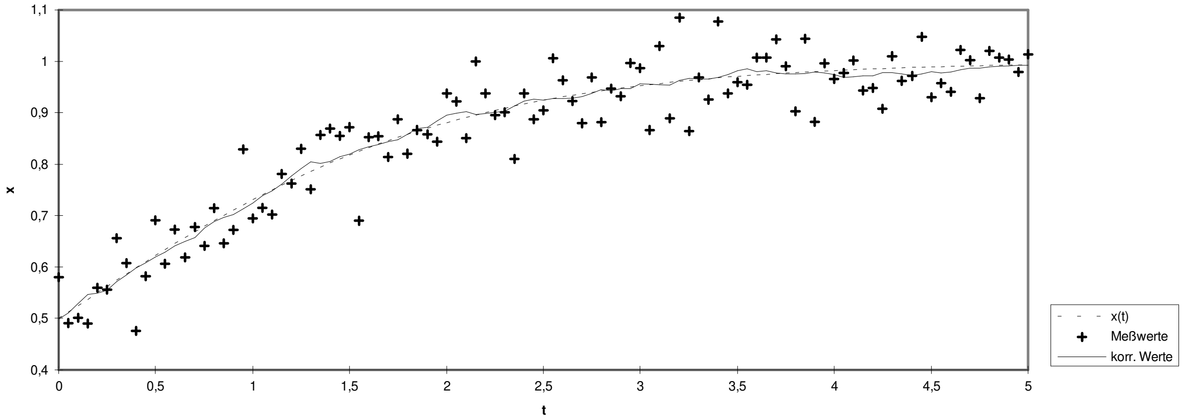


Abbildung 4-4: Korrektur verrauschter Meßwerte mit W=5; M=20

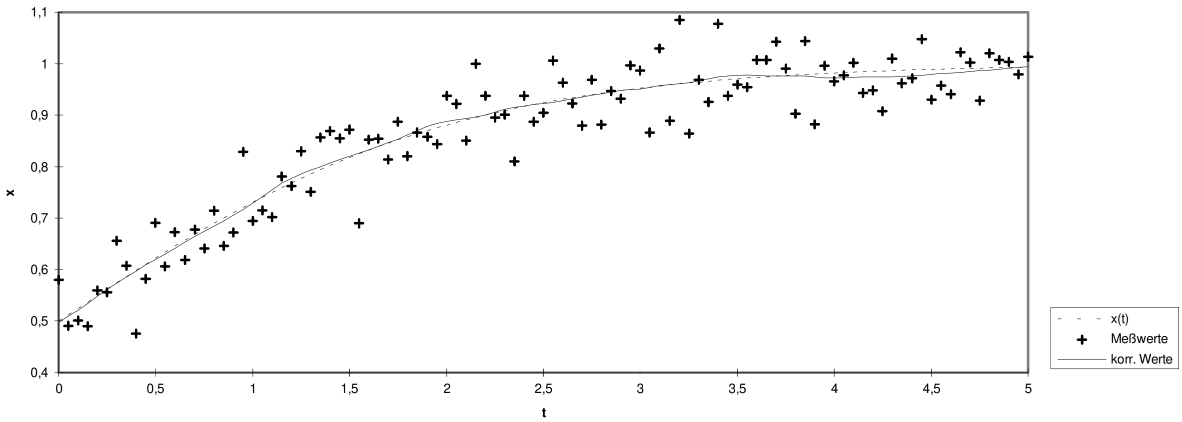


Abbildung 4-5: Korrektur verrauschter Meßwerte mit W=10; M=10

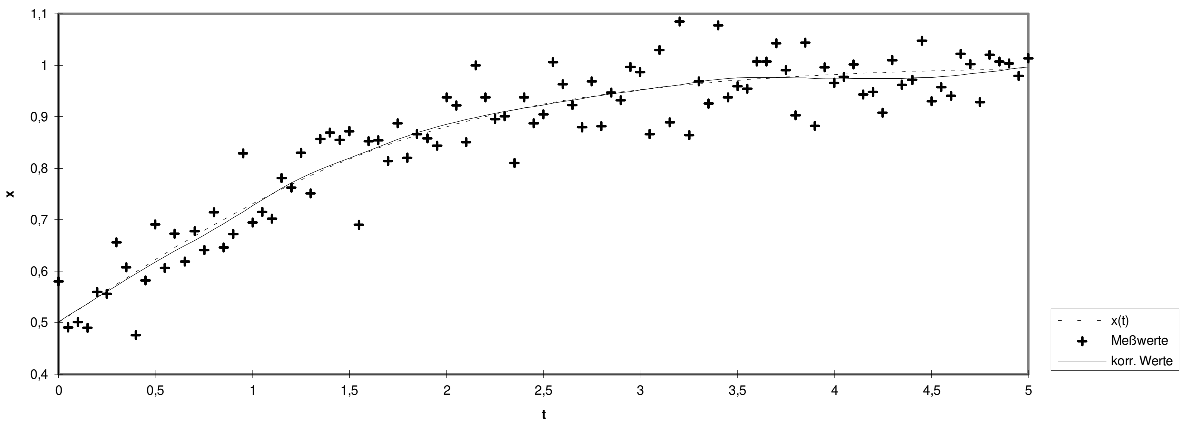


Abbildung 4-6: Korrektur verrauschter Meßwerte mit W=20; M=5

Die Ergebnisse der Tests legten nahe, W möglichst groß zu wählen – M ergibt sich dann aus der zur Verfügung stehenden Rechenzeit. Aus praktischen Gründen (z. B. Latenzzeit im On-line-Betrieb) empfiehlt es sich jedoch, M nicht kleiner als 5 werden zu lassen.

Hierbei bleibt jedoch ein Aspekt unberücksichtigt, der für den Einsatz des Algorithmus in Anwendungen zur On-line-Kontrolle bedeutsam sein könnte: In derartigen Anwendungen kann es wichtig sein, möglichst schnell einen guten Schätzwert für die jüngste Messung zu erhalten. Bei sehr klein gewähltem M aber hat der Schätzwert dieser Messung beim Eintreffen der nächsten Messung nur wenige Updates und daher auch nur eine geringe Verbesserung erfahren. Obwohl zu dieser Thematik keine Versuchsreihen durchgeführt wurden, empfiehlt sich in solchen Fällen nach meiner Meinung zumindest die Einhaltung der Faustregel:

$$M \geq \frac{2}{\varepsilon}$$

4.1.5 Einfluß von K

Die Einführung der erweiterten Energiefunktion (2.3) und damit der Konstanten K ging auf die in den ersten Experimenten gemachten Erfahrungen zurück, daß in einigen seltenen Fällen auch eine andere als die beabsichtigte Lösung der Differentialgleichung als Ergebnis des Verfahrens gefunden wurde. Der sogenannte „K-Term“ in der erweiterten Energiefunktion sollte dieses „Weglaufen“ der Lösung verhindern, wobei abzusehen war, daß sich die Wahl eines kleinen K (< 20) deutlich negativ auf die Ergebnisqualität auswirken würde. Dies konnte auch in den Versuchen beobachtet werden (vgl. Abbildung 4-7). K wurde deshalb in fast allen weiteren Experimenten sehr hoch (>10.000) angesetzt. Negative Auswirkungen, wie das befürchtete „Einrasten“ der Schätzwerte auf einer anderen Lösung der Differentialgleichung (siehe Kapitel 2.2) , konnten in keinem Fall beobachtet werden. Daraus ergibt sich die Empfehlung, auch im praktischen Einsatz des Algorithmus K ähnlich hoch anzusetzen.

(Beispiele mit W=10; M=10; Epsilon=0,2)

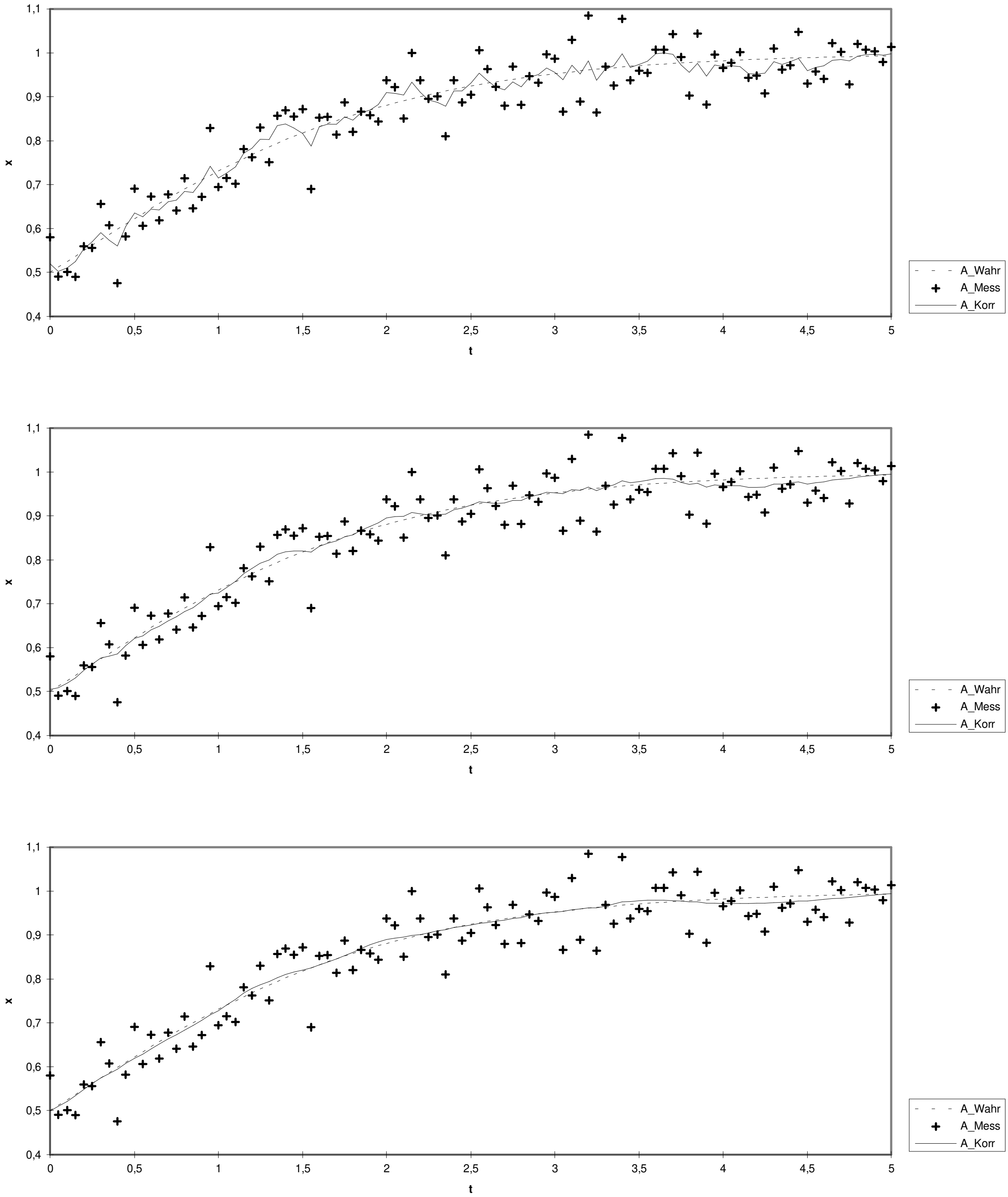


Abbildung 4-7: Korrektur verrauschter Meßwerte mit K=4, K=20 und K=100

4.1.6 Einbeziehen von Systemparametern

Der nächste Schritt bei der Entwicklung des Algorithmus bestand im Hinzufügen der Identifikation von Systemparametern wie im Abschnitt 2.5 beschrieben. Die Verwendung der Updateformel ohne Vorfaktor erwies sich vor allem wegen der dort erwähnten Probleme bei der manuellen Wahl der Anpaßrate als kaum praktikabel. Die Tests wurden also mit der modifizierten Form (2.8) der Updateformel für die Systemparameter durchgeführt. Die Variante ohne Vorfaktor (2.7) konnte erst nach Einführung der automatischen Wahl der Anpaßrate verwendet werden (siehe 4.1.10).

Zum Test dieses Teils des Algorithmus war natürlich eine Differentialgleichung notwendig, die mindestens einen zu identifizierenden Parameter enthält:

$$\frac{\partial x}{\partial t} = ax - x^2$$

Die Versuche damit zeigten schnell Erfolge. In der folgenden Grafik (Abbildung 4-8) ist die Korrektur des Parameters a zu sehen: Das Objektsystem verwendete $a' = 1$, die Schätzung dagegen wurde mit der Vermutung $a = 0$ gestartet (W=10; M=10; Epsilon=0,2; K=100.000). Bereits nach der Verarbeitung von 100 Messungen läßt sich die Tendenz zur Anpassung des Parameters gut erkennen.[21]

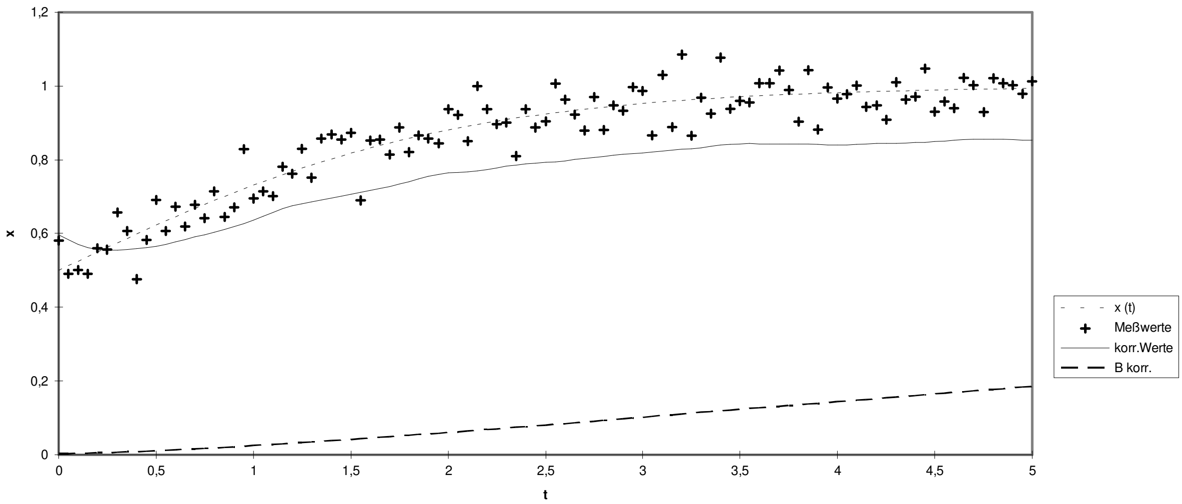


Abbildung 4-8: Anpassung eines Systemparameters

Setzt man dieses Beispiel mit der Betrachtung zusätzlicher Meßwerte fort, kann eine weitere Annäherung der Parameter beobachtet werden ($\hat{a}$=0,9 nach 750 Messungen).

[21] Die in der Grafik erkennbare schlechte Korrektur der Schätzwerte $\hat{x}_n$ ist eine Folge des noch sehr fehlerhaften Parameters b. In dem Maße, in dem dieser Parameter verbessert wird, steigt auch die Qualität der Schätzwerte.

4.1.7 Andere Objektsysteme

Die bisher beschriebenen Versuche wurden jeweils auf der Grundlage derselben (bzw. für die Parameteridentifikation nur geringfügig veränderten) Differentialgleichung durchgeführt. So war zwar die Vergleichbarkeit der Ergebnisse verschiedener Versuchsreihen gegeben, aber noch war nicht gesichert, daß sich der neue Algorithmus auch auf andere Systeme erfolgreich anwenden ließe. Von besonderem Interesse waren in diesem Zusammenhang Systeme, die über mehrere Zustandsvariablen verfügen. Die Wahl fiel auf zwei Systeme:

Als erstes System mit mehreren Variablen diente das Modell eines einfachen Räuber-Beute-Systems:

$$\frac{\partial x}{\partial t} = ax - bxy$$

$$\frac{\partial y}{\partial t} = cx + dxy$$

Als zweites System wurde die Simulation eines CSTR[22] nach Liebman et al. (1992) als Objektsystem eingesetzt, um den Algorithmus in einer „praxisnahen" Anwendung zu testen:

$$\frac{\partial A}{\partial t} = \frac{q}{V}\left(A_0 - A\right) - \alpha_d kA$$

$$\frac{\partial T}{\partial t} = \frac{q}{V}\left(T_0 - T\right) + \alpha_d \frac{-\Delta H_r A_r}{pC_p T_r} kA - \frac{UA_r}{pC_p V}\left(T - T_c\right)$$

$$k = K_0 \exp\left(\frac{-E_A}{TT_r}\right)$$

In diesem System sind die Zulauftemperatur T_0 und die Zulaufkonzentration A_0 Inputvariablen, die Tanktemperatur T und die Tankkonzentration A Zustandsvariablen. Messungen dieser Größen werden jeweils im Abstand von 2,5 s vorgenommen. Die übrigen Größen in den Gleichungen beschreiben die chemisch-physikalischen Eigenschaften des Reaktionssystems:

[22] continuos-flow stirred tank reactor = kontinuierlich betriebener Rührkesselreaktor

Parameter	typ. Wert	Einheit
q	10,0	$cm^3 \cdot s^{-1}$
V	1000,0	cm^3
ΔH_r	-27.000,0	$cal \cdot gmol^{-1}$
p	0,001	$g \cdot cm^{-3}$
C_p	1,0	$cal \cdot (g \cdot K)^{-1}$
A_r	10,0	cm^2
T_c	340,0	K
U	$5 \cdot 10^{-4}$	$cal \cdot (cm^2 \cdot s \cdot K)^{-1}$
k_0	$7{,}86 \cdot 10^{12}$	s^{-1}
E_A	14.090,0	K
α_d	1,0	–

Experimente mit diesen beiden Objektsystemen konnten zeigen, daß der Algorithmus auch Systeme mit mehreren Zustandsvariablen problemlos bearbeiten kann. Für das in der folgenden Grafik dargestellte Experiment mit dem Räuber-Beute-System wurden folgende Parameter gewählt:

Startwerte der Zustandsvariablen:	$x' = 0{,}5;\ y' = 1$
Werte nicht zu identifizierender Systemparameter:	$b = b' = 1,\ d = d' = 1$
wahre Werte zu identifizierender Systemparameter:	$a' = 0{,}5;\ c' = 3$
geschätzte Werte der Systemparameter:	$a = 1;\ c = 1$
Abstand zweier Messungen auf der t-Achse:	$\Delta t = 0{,}1$
weitere Einstellungen:	W=20; M=20; K=20; Epsilon=0,3

Außerdem wurden (wie auch für alle folgenden Experimente) die Updates der Schätzwerte synchron ausgeführt, wie am Ende des Abschnitts 2.2 beschrieben.

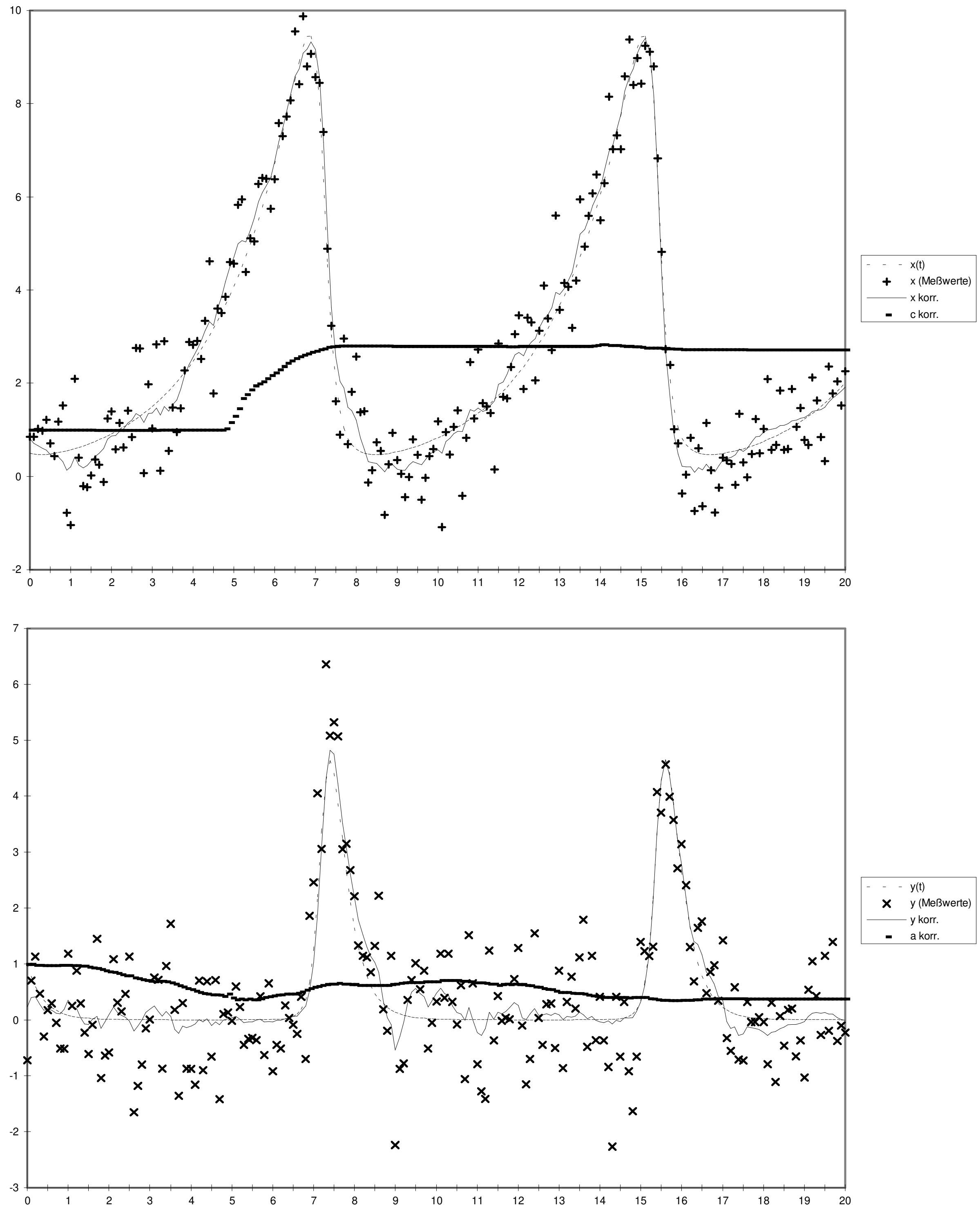
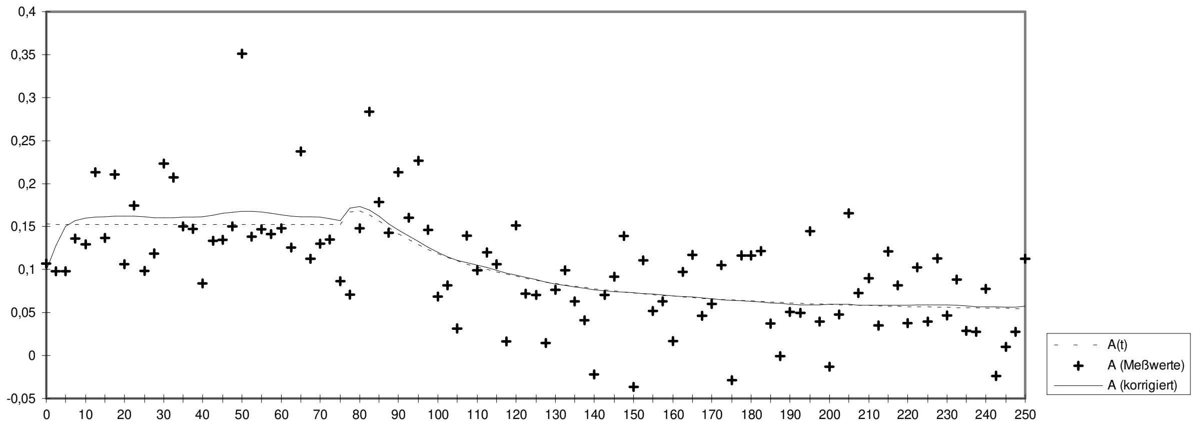


Abbildung 4-9: Korrektur gestörter Meßwerte des Räuber-Beute-Systems

Nach der Bearbeitung von 200 Meßwert-Paaren waren in diesem Experiment auch die System parameter gut approximiert: $\hat{a} = 0{,}372;\ \hat{c} = 2{,}713$

Bei den Versuchen mit der CSTR-Simulation tauchten erstmals Input-Variablen auf. Um erste Tests mit diesem System durchführen zu können, ohne vorher Untersuchungen zur Korrektur von Input-Variablen durchzuführen, wurden zunächst fehlerhafte Messungen für die Inputs nur durch Bildung des gleitenden Mittels entrauscht.[23] Dennoch konnten bereits mit dieser Notlösung gute Resultate bei Reconciliation der Zustandsvariablen erreicht werden. (Einstellungen für den gezeigten Versuch: W=5; M=30; K=100.000; Epsilon=0,4 (dynamische Anpaßrate, siehe 2.4.2), synchrones Update)



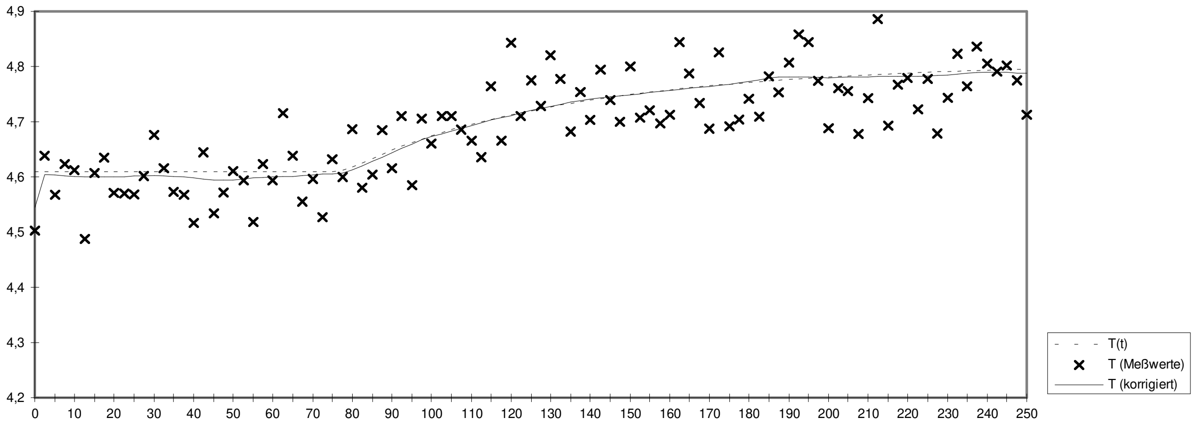


Abbildung 4-10 : Korrektur verrauschter Meßwerte aus der CSTR-Simulation

Im Rahmen der eben beschriebenen Versuche wurde wiederum eine Anzahl von Modifikationen des ursprünglichen Verfahrens getestet. Die wichtigsten dieser Modifikationen seien

[23] Das führte natürlich zu schlechten Resultaten bei der Korrektur der Input-Variablen. Allerdings waren diese Resultate akzeptabel für die Vergleiche mit dem NDDR-Algorithmus, da die von Liebman et al. publizierten Leistungen dieses Verfahrens bei der Korrektur der Input-Messungen ebenfalls mangelhaft waren. Der Grund lag in sprunghaften Änderungen der Input-Variablen bei diesem Experiment, für die das NDDR-Verfahren in der publizierten Form kaum geeignet ist (vgl. Liebman et al. 1992, S. 974 ff.).

nachfolgend kurz aufgeführt. Als Maß für den Erfolg jeder Verbesserung diente die Varianz der korrigierten Meßwerte (auch MSE = medium square error).

synchrones Update: Die synchrone Ausführung der Updates aller Schätzwerte im Zeitfenster, wie am Ende des Abschnitts 2.2 beschrieben, konnte die Qualität der Ergebnisse deutlich verbessern (10% bis 30% Abnahme des MSE). Alle folgenden Versuche wurden daher mit synchronem Update ausgeführt.

dynamische Anpaßrate: Diese Modifikation (siehe Abschnitt 2.4.2) konnte die Resultate dann entscheidend verbessern, wenn keine Fehler in den Systemparametern vorlagen und der Anfangszustand von Objektsystem und Simulation möglichst gut übereinstimmte (in diesen Fällen Reduktion des MSE um bis zu 30%, insbesondere durch verbesserte Unterdrückung „langwelliger" Störungen), bei Verletzung dieser Bedingungen erwies sich diese Modifikation aber als kontraproduktiv, so daß ich von ihrem praktischen Einsatz abrate.

veränderte Updateregel: Es wurde auch mit veränderten, „symmetrischen" Updateregeln experimentiert, die neben Vorwärts- auch Rückwärtsschätzungen enthielten. Allerdings zeigte sich, daß bei der CSTR-Simulation die Rückwärtsschätzungen hochgradig instabil waren und ausgehend von fehlerbehafteten Schätzwerten häufig unbrauchbare Ergebnisse lieferten.

4.1.8 Einbeziehen systematischer Meßfehler

In den nächsten Versuchen wurde die Aufmerksamkeit auf die Erkennung systematischer Meßfehler (Bias) gerichtet. Der im Kapitel 2.7 beschriebene Ansatz zur Darstellung des Bias als zusätzlichem Systemparameter zeigte schnell Erfolge. Als Beispiel seien hier die Ergebnisse der Reproduktion eines bei Liebman et al. (1992, S. 980 f.) beschriebenen Experimentes zur Bias-Identifikikation am CSTR gezeigt: Die Messung von A wurde mit einem Bias von 0,1 belegt, als anfängliche Schätzung für den Bias wurde Null angenommen. Das Ergebnis hat nach 100 Messungen noch nicht die Qualität der Resultate von Liebman erreicht,[24] wird aber mit der Verarbeitung weiterer Messungen noch verbessert. (Einstellung des Algorithmus: W=5; M=30; K=100.000; Epsilon=0,4)

---

[24] Die von Liebman publizierten Schätzwerte für die Bias-Korrektur schwanken zwischen 0,9 und 1,1. Diese Qualität der Näherung wird schon nach zehn Messungen erreicht, verbessert sich dann jedoch nicht mehr, unabhängig von der Zahl der verarbeiteten Messungen.

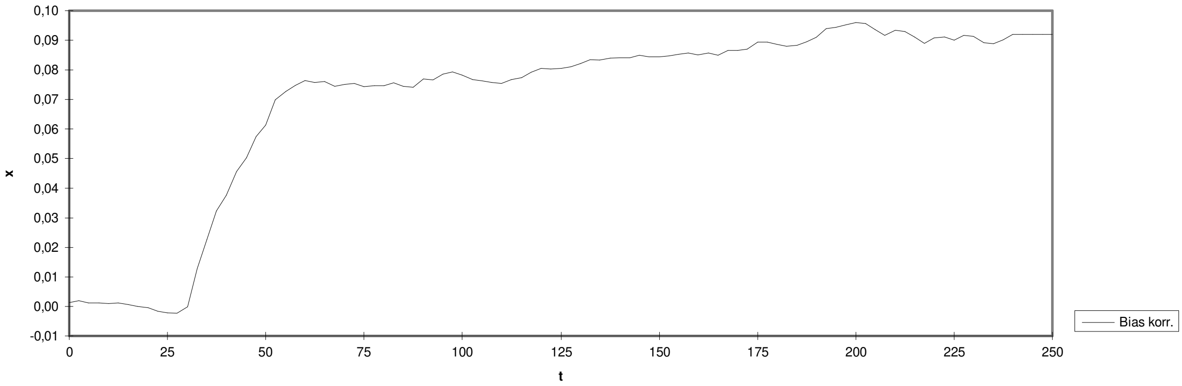


Abbildung 4-11: Identifikation des Bias der Variablen A am CSTR [25]

4.1.9 Einbeziehen von Input-Variablen

In den vorigen Experimenten mit der Simulation des CSTR wurden die Input-Schätzwerte nur auf eine sehr einfache Weise korrigiert. Die Bemühungen um eine verbesserte Korrektur orientierten sich zunächst an dem von Liebman et al. angewandten Vorgehen (siehe 2.6). Die Resultate dieser Versuche waren zunächst enttäuschend, wenn die Inputvariablen nicht konstant blieben. Die in Abschnitt 2.6 beschriebene Variante, die Inputvariablen als zusätzliche Zustandsvariablen zu behandeln, zeigte etwas bessere Erfolge, allerdings auch immer noch große Schwierigkeiten bei sprunghaften Änderungen der Input-Variablen: (Einstellungen für das folgende Beispiel: W=5; M=30; Epsilon=0,4; K=1000 bzw. K=100)

[25] Die in der Grafik erkennbare Konstanz der Biasschätzung für die letzten fünf Fensterschritte ist auf den Abbruch des Algorithmus zurückzuführen.

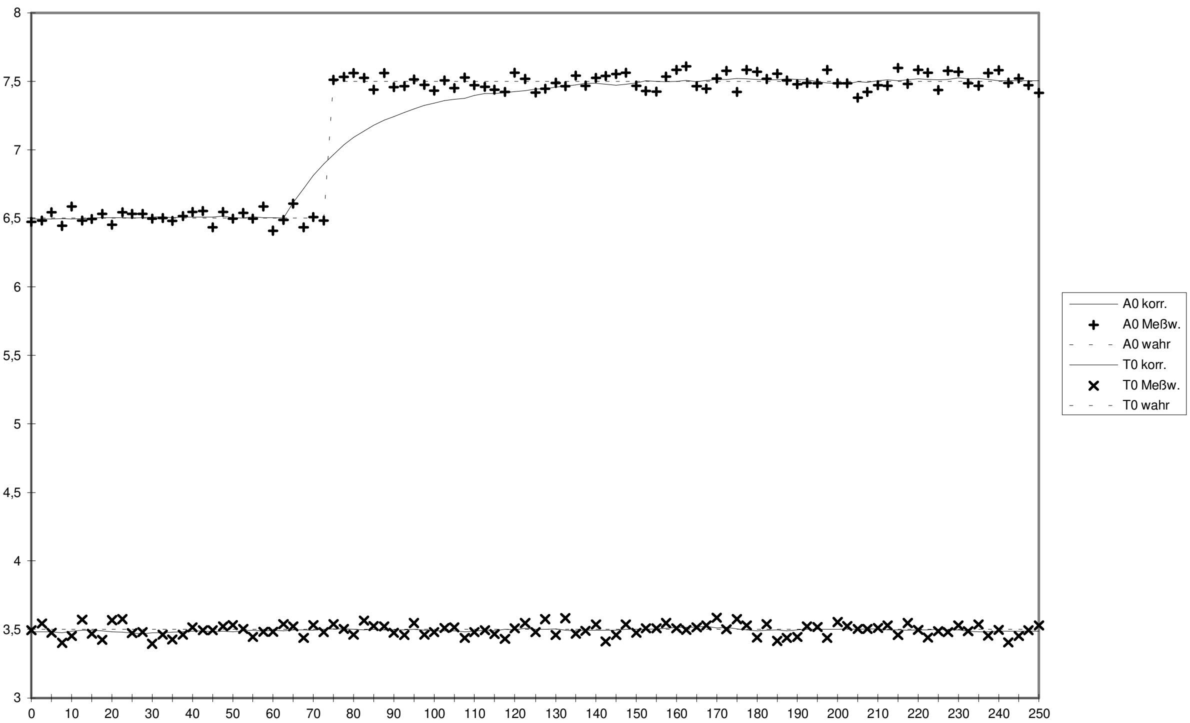


Abbildung 4-12 : Korrektur verrauschter Input-Messungen mit K=1000

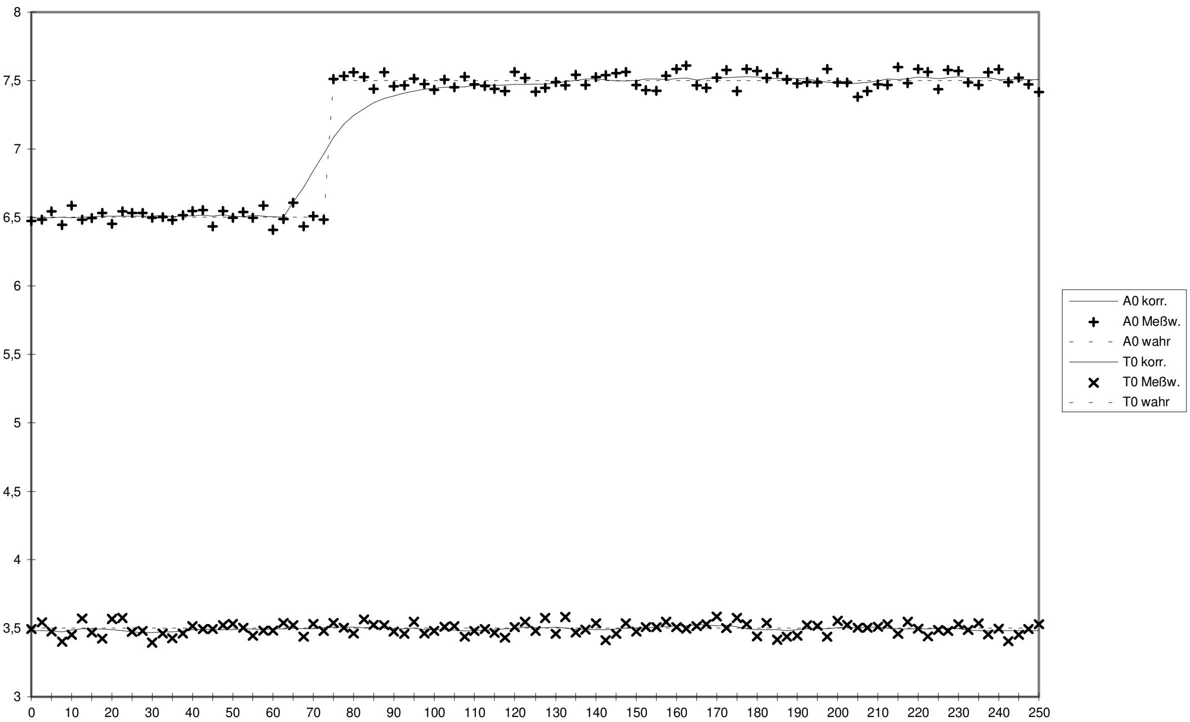


Abbildung 4-13 : Korrektur verrauschter Input-Messungen mit K=100

Wie in diesem Beispiel erkennbar, kann die Senkung von K positiven Einfluß auf die Erkennung derartiger sprunghafter Änderungen in den Inputs haben. Dabei sind jedoch die bereits beschriebenen negativen Auswirkungen einer Senkung von K zu berücksichtigen.

Um auch ohne ein Absenken von K brauchbare Ergebnisse bei der Reaktion der Input-Schätzungen auf starke Veränderungen in den Inputvariablen zu erhalten, setzte ich die im Abschnitt 2.6 beschriebene Modifikation des Algorithmus durch Kaltstart der Inputschätzungen ein, was sich als praktikable Notlösung erwies: (Einstellungen wie für das vorige Beispiel, jedoch K=100.000)

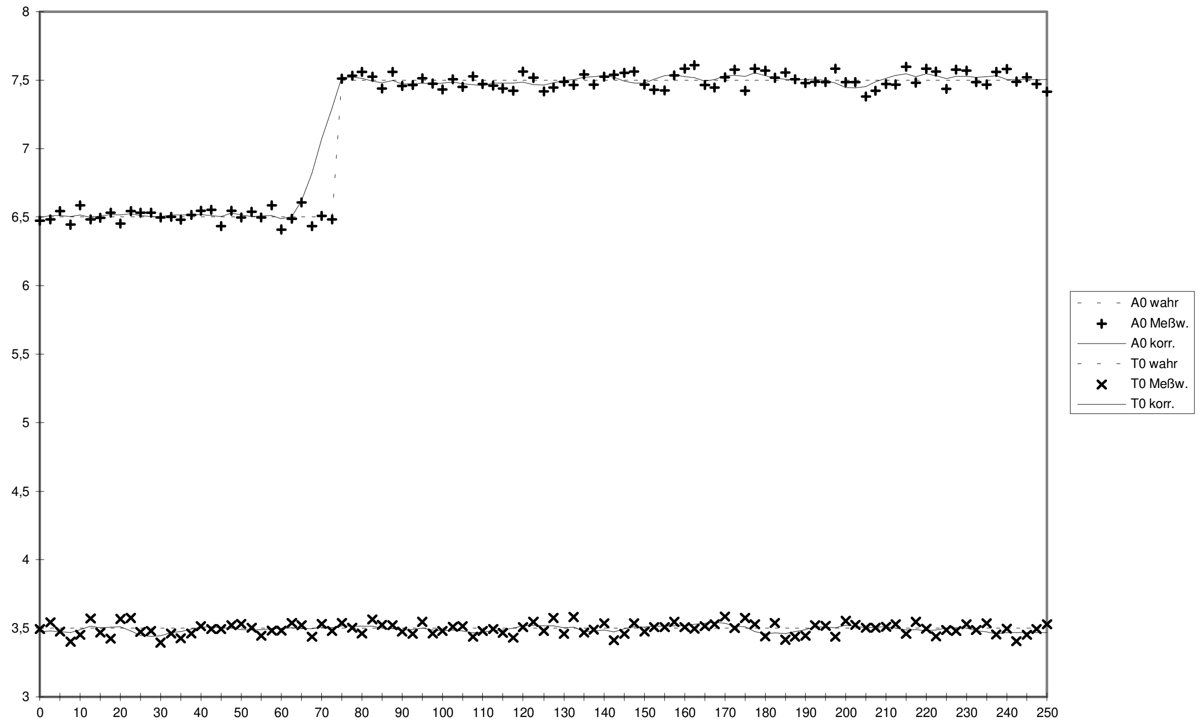


Abbildung 4-14: Korrektur verrauschter Input-Messungen mit Kaltstart

Zusammenfassend muß aber gesagt werden, daß das Problem der flexiblen Behandlung von Messungen der Inputvariablen bislang nicht endgültig gelöst werden konnte.

4.1.10 Abschließende Experimente

Obwohl mit dem bisher beschriebenen Stand der Versuche die Funktionsfähigkeit des Algorithmus in allen wichtigen Belangen nachgewiesen werden konnte, waren die Möglichkeiten noch längst nicht „ausgereizt". In den folgenden Wochen ergaben sich noch einige gute Ideen, die Leistungsfähigkeit und Praxistauglichkeit des Algorithmus nochmals deutlich verbessern konnten. Einen wichtigen Maßstab zur Bewertung dieser Verbesserungen bildeten die von Liebman et al. publizierten Leistungen des NDDR-Algorithmus. Im Abschnitt 3.2 finden sich Details zu den Ergebnissen dieses Vergleichs.

Approximation der Ableitungen der Vorwärtsschätzung: Wie im Kapitel 2.3 beschrieben, wurden statt der „komplizierten" numerischen Näherungen für die partiellen Ableitungen der

Vorwärtsschätzung einfache Approximationen der Form (2.5) eingesetzt. Dies verbesserte die Performance deutlich, ohne nennenswerte Einflüsse auf die Qualität des Ergebnisses zu zeigen.

Einstellung der Anpaßrate: Der in Abschnitt 2.8 beschriebene Automatismus zum Einstellen der Anpaßrate wurde implementiert. Dies hatte zwar kaum unmittelbare Auswirkungen auf die Qualität der Ergebnisse, wohl aber auf die Arbeitsgeschwindigkeit des Algorithmus. War vorher die Performance entscheidend von der richtigen Wahl von Epsilon abhängig (bei kleinem Epsilon wurden wesentlich mehr Iterationen benötigt, bei zu großem Epsilon verschlechterte sich die Konvergenz), wurde Epsilon nun stets richtig und für jede Iteration individuell eingestellt. Das führte dazu, daß nun auch ohne langwierige Tests sofort die optimale Perfomance bei jedem Problem erzielt werden konnte. Der so erreichte Wegfall des Tuningparameters Epsilon erleichtert außerdem den Einsatz in der Praxis erheblich.

Identifikation von Systemparametern: Die automatische Einstellung der Anpaßrate ermöglichte nun auch Experimente mit der Updateformel (2.7), die ohne den Vorfaktor $\frac{1}{W}$ auskommt. Dabei traten allerdings die im Abschnitt 2.5 beschriebenen negativen Auswirkungen der zu schnellen Anpassung der Systemparameter zu Tage, so daß diese Berechnungsmöglichkeit für die Schätzwerte der Systemparameter wieder verworfen wurde.

Identifikation systematischer Fehler: Neben dem eben beschriebenen Weglassen des Vorfaktors in der Updateformel (das dieselben negativen Effekte zeigte wie bei den Systemparametern beschrieben) wendete ich auch erstmals die vereinfachte Berechnung der Bias-Schätzwerte an, die im Kapitel 2.7.2 beschrieben wird. Das Ergebnis war überzeugend: Die Qualität der Resultate blieb gleich, jedoch wurde die Geschwindigkeit stark gesteigert – bei gleichbleibendem M und W war der zusätzliche Zeitverbrauch für die Ermittlung der Bias-Schätzungen kaum nachzuweisen.

4.2 Data Reconciliation am ACAC-Reaktor

4.2.1 Der ACAC-Reaktor

Um das Verständnis der folgenden Kapitel zu erleichtern, soll an dieser Stelle ein kurzer Einblick in den Aufbau des ACAC-Reaktors gegeben werden.

Der Reaktor, dessen Simulation hier näher untersucht wird, ist einer von insgesamt drei Reaktoren unterschiedlicher Bauart, die im Werk Burghausen der Wacker Chemie GmbH zur Produktion von Acetylaceton (kurz ACAC, Strukturformel $CH_3{-}CO{-}CH_2{-}CO{-}CH_3$) eingesetzt werden.

Der Reaktor besteht aus einer massiven Hülle, die den Rektorinnenraum umschließt. In diesem Innenraum befinden sich ein Reaktionsrohr sowie Heizvorrichtungen, die auf dieses Reaktionsrohr wirken. Während des Betriebes des Reaktors durchströmt ein Stoffgemisch das Reaktionsrohr. Dabei laufen verschiedene chemische Reaktionen ab, die unter anderem zur Bildung von ACAC führen. Am Ende des Reaktionsrohres verläßt das Stoffgemisch den Reaktor und wird einer weiteren Anlage zugeführt, die zur Trennung des ACAC aus dem Stoffgemisch dient. Temperatur und Druck im Reaktionsrohr sind so gewählt, daß alle beteiligten Substanzen (Ausgangsstoffe, Zwischen- und Endprodukte) gasförmig sind.[26]

Die Konzentration der einzelnen Komponenten am Stoffeintrag ist dabei praktisch konstant, lediglich während der Aufwärmphase bei der Inbetriebnahme des Reaktors wird der Stoffeintrag mit reinem Stickstoff versorgt.

Der Ablauf der einzelnen Reaktionen (und somit die Bildung von ACAC) ist vor allem vom Temperaturverlauf im Reaktionsrohr abhängig. Entscheidend hierfür sind zwei Einflußmöglichkeiten: die Steuerung der Heizeinrichtungen und der Durchflußrate des Stoffeintrags (letztere beeinflußt die Strömungsgeschwindigkeit des Stoffgemisches im Reaktor und so dessen Wärmeaustausch mit dem Reaktionsrohr).

Am Reaktor wird eine Reihe von Meßgrößen erfaßt, unter anderem:

- Temperatur des Gasgemisches an verschiedenen Positionen des Reaktionsrohres
- Wandtemperatur des Reaktionsrohres an verschiedenen Positionen
- Durchflußmenge am Stoffeintrag

Derzeit wird ein Prozeßleitsystem am Reaktor installiert, das die Meß- und Stellgrößen der computergestützten Archivierung und der On-line-Verarbeitung zugänglich machen soll. Da

[26] Ein Nebenprodukt der Reaktionen bildet hierbei eine Ausnahme. Es kondensiert bei gewissen Betriebsbedingungen als Ablagerungsschicht an der Innenwand des Reaktionsrohres.

sich die Inbetriebnahme dieses Systems mehrfach verzögerte, war es bis zum Abschluß dieser Arbeit nicht möglich, die entwickelte Software auf die Zusammenarbeit mit dem Prozeßleitsystem abzustimmen und den Algorithmus so an realen Daten aus dem Betrieb des Reaktors zu testen.

4.2.2 Weiterentwicklung des Simulationsprogramms

Bevor der Algorithmus auf die Simulation des ACAC-Reaktors angewendet werden konnte, waren zunächst einige Vorarbeiten notwendig. Zuerst war eine Portierung des existierenden Simulationssystems notwendig. Der Grund dafür lag darin, daß das dieses System in Fortran-77 programmiert worden war, während für die Meßwertkorrektur eine delphi-basierte Lösung erstellt werden sollte, die auch die Simulation beinhalten sollte.

Der Umfang des Simulationssystems betrug etwa 2000 Zeilen Fortran-Code, der nur spärlich oder gar nicht kommentiert war. Eine Dokumentation zu den Quelltexten gab es nicht, lediglich einige firmeninterne Berichte (Quellenangabe ist Geschäftsgeheimnis), die sich vorwiegend mit den physikalisch-chemischen Grundlagen des Modells und der Bedienung des Simulationssystems befaßten.

Da erste Versuche der Übertragung von Fortran-Code mit automatischen Konvertieren katastrophale Ergebnisse bezüglich Erfolgsquote und Lesbarkeit des erzeugten Codes lieferten, entschloß ich mich, die Übertragung der Quelltexte nach Pascal zum größten Teil manuell durchzuführen.

Um den Arbeitsaufwand dafür in Grenzen zu halten, setzte ich mit Microsoft Word 6.0 für Windows ein Textverarbeitungssystem ein, das umfangreiche Möglichkeiten der Makroprogrammierung durch einen Basic-Dialekt bietet.

Die Übertragung von Fortran nach Pascal erfolgte dabei nicht sequentiell (in der Reihenfolge der Anweisungen im Quelltext), sondern getrennt nach Anweisungstypen. So konnten einige sehr einfache Code-Elemente (Wertzuweisungen, CALLs, Kommentare...) mit simplen Makros vollautomatisch übersetzt werden.

Bei einer Anzahl weiterer Fortran-Konstrukte (IF-Anweisungen, Schleifen) konnten zumindest gewisse Teile jeder Anweisung automatisch übersetzt werden (beispielsweise die Vergleichsoperatoren in IF-Konstrukten), während eine komplette Übertragung dieser Anweisungen ohne ein gewisses Parsing der Quelltexte nicht möglich war. Dieses zu implementieren hätte jedoch mehr Zeitaufwand bedeutet, als für eine manuelle Übertragung nötig war, so daß diese Anweisungen „halbautomatisch“ übertragen wurden: die Syntaxanalyse wurde manuell ausgeführt, für bestimmte Anweisungsteile kamen dann Makros zum Einsatz.

Eine dritte Gruppe von Anweisungen konnte dagegen auch mit dieser Taktik nicht übersetzt werden, da es für sie keine direkte Entsprechung in Pascal gibt. Dies betrifft beispielsweise die Statements ASSIGN, FORMAT und COMMON, die nur manuell übersetzt werden konnten.

Schließlich war noch die Ablaufsteuerung des Programms umzustellen. Waren im Fortran-Code vor allem Labels und GOTOs zur Ablaufsteuerung eingesetzt, mußten diese Konstrukte jetzt in die pascal-typischen Blöcke und Schleifen übertragen werden, was teilweise nur unter Schwierigkeiten möglich war.

Ein weiteres Problem stellten die fortran-typischen dynamischen Arrays dar. Besonders arbeitsaufwendig waren Prozedurdeklarationen, die solche Arrays als Parameter enthielten. Hier war jeweils eine Analyse aller Aufrufe dieser Prozedur nötig, um in Pascal einen Array-Typ geeigneter Größe definieren zu können.[27]

Schließlich waren alle Fortran-Anweisungen in Pascal-Konstrukte übersetzt und so eine lauffähige Konsolenapplikation in Pascal entstanden, die dem ursprünglichen Fortran-Programm sehr genau entsprach.

Den nächste Schritt bildete ein (moderates) Reengineering des entstandenen Pascal-Codes. Teilweise durch die Beschränkungen von Fortran, teilweise durch die sehr „mechanistische“ Portierung gab es eine Reihe von Konstruktionen, die mit den Mitteln von Pascal effizienter und/oder übersichtlicher formuliert werden konnten.

Ein Beispiel dafür ist die große Anzahl (ca. 100) globaler Variablen, die im Pascal-Programm durch die Übersetzung der COMMON-Statements entstanden war. In Verbindung mit der Beschränkung auf Variablennamen mit maximal sechs Zeichen Länge stellte sie ein ernsthaftes Hindernis beim Verständnis des Programmcodes dar. Daher wurden die Variablen entsprechend ihrer Bedeutung zu Records oder Objekten zusammengefaßt, wobei sich herausstellte, daß eine ganze Anzahl Variablen im Programm niemals verwendet wurde und darum entfernt werden konnte. Außerdem wurden einige ausgewählte Variablen mit aussagekräftigeren Namen versehen.

Anschließend begann ich mit der Umsetzung des Kommandozeilenprogramms in eine Applikation mit MDI-Interface, die gegenüber dem ursprünglichen Programm bereits wichtige Erweiterungen aufwies:

- schematische Darstellung des Reaktors am Bildschirm mit allen aktuellen Meßergebnissen
- Speicherung dieser Meßergebnisse in verschiedenen Formaten (u. a. dBase IV)
- Bedienung des simulierten Reaktors über grafische Steuerelemente (bis dahin über Einträge in einer Konfigurationsdatei)
- Darstellung der örtlichen und zeitlichen Verläufe wichtiger Größen in Diagrammen
- Speichern und Wiederherstellen des Zustandes des Reaktors

---

[27] Die Nachbildung dynamischer Arrays mit den Mitteln von Delphi wäre mit großem Aufwand und Performanceverlusten verbunden gewesen, zumal es sich häufig um mehrdimensionale Arrays handelte.

Im Zuge dieser Arbeit wurden einige neue Konzepte und Objekte eingeführt. Als Beispiel sei hier das Konzept des Meßgeräts dargestellt:

Eine Basisklasse **`TMeter`** stellt die grundlegende Methoden und Eigenschaften von Meßgeräten bereit (Meßwert, physikalische Einheit des Meßwertes, Genauigkeit, zufällige und systematische Störung des Meßwertes...). Davon abgeleitet werden schließlich die verschiedenen Klassen von Meßgeräten (**`TWallTMeter`**, **`TGasTMeter`**, **`TFlowMeter`**...), jedes Meßgerät am Reaktor wird schließlich durch eine Instanz einer solchen Klasse repräsentiert.

So konnten die Darstellung und Speicherung von Daten aus verschiedenen Quellen in der Simulation vereinheitlicht werden. Außerdem vereinfachte sich so die Erzeugung von gestörten Meßwerten für den Test des Algorithmus.

4.2.3 Anwendung des Algorithmus auf das ACAC-Reaktormodell

Nachdem die Übersetzung des Simulationsprogramms für den ACAC-Reaktor nach Pascal und die damit verbundene gründliche Analyse der Struktur der Simulation weit genug fortgeschritten war, zeigte sich, daß die Simulation aus drei Teilsimulationen bestand:

a) Produkttrennung
Die Anlage zur Separation des ACAC wurde in einem größtenteils isolierten Abschnitt der Simulation modelliert. Da an dieser Teilanlage bislang keine relevanten Meßgeräte installiert waren und die Simulation dieser Komponente durch Herrn List als „weitestgehend korrekt“ eingestuft wurde, war es nicht nötig, diesen Teil des Systems in die Data Reconciliation einzubeziehen.

b) Verteilung der Komponenten im Reaktionsrohr
In dieser Teilsimulation wurde der Verlauf der chemischen Reaktionen und der Temperatur im strömenden Gasgemisch betrachtet. Radiale Unterschiede in Strömungsgeschwindigkeit, Temperatur und Konzentration der einzelnen Bestandteile wurden bei der Modellierung vernachlässigt. Ferner wurde davon ausgegangen, daß während jedes Simulationsschrittes die Durchflußrate des Gasgemisches und die Wandtemperaturen konstant bleiben. Die auf dieser Grundlage formulierten Differentialgleichungen für Gastemperatur T_g und Reaktionsfortschritt ξ und die abgegebene Wärme Q – insgesamt 14 Gleichungen – waren Differentialgleichungen des Ortes: [28]

$$\frac{\mathrm{d}T_g}{\mathrm{d}x}=\dots \quad \frac{\mathrm{d}\xi_i}{\mathrm{d}x}=\dots \quad \frac{\mathrm{d}Q}{\mathrm{d}x}=\dots \tag{4.1}$$

[28] Die Ortskoordinate ist hier eindimensional, da radiale Unterschiede wie erwähnt vernachlässigt wurden.

Besonders problematisch in diesem Zusammenhang ist die Tatsache, daß sich diese Teil simulation, wie sie in den von der Wacker Chemie GmbH zur Verfügung gestellten Quelltexten realisiert war, als unvollständig erwies, da nicht die gesamte Länge der Reaktionsröhre einbezogen war.

c) Thermodynamische Austauschprozesse

Ein weiterer Bereich der Simulation modellierte die Wärmeaustauschprozesse zwischen Heizung, Reaktionsrohr, Reaktorwand und Umgebung. Die Wand des Reaktors wurde dabei in mehrere Schichten unterteilt, das Reaktionsrohr in mehrere Sektionen gleicher Länge. Die Zustandsvariablen dieser Teilsimulation sind:

- Wandtemperatur jeder Sektion k des Reaktionsrohres T_{tk} (ca. 20 Variablen)
- Temperatur jeder Schicht k der Reaktorwand T_{wk} (ca. 5 Variablen)

Das Verhalten dieser Zustandsgrößen wird durch Differentialgleichungen der Zeit beschrieben. Dabei tauchen einerseits die T_{tk} auch in den Differentialgleichungen (4.1) auf, andererseits findet sich Q in den Gleichungen für die T_{tk}. So wird die Verbindung zwischen den Modellen der Teil simu lationen b und c hergestellt.

Die Simulation des Gesamtsystems erfolgte nun durch abwechselnde Ausführung der Teilsimulationen b und c. Jeder Zeitschritt der Teilsimulation b umfaßte dabei 20 Sekunden.

Diese Struktur von Modell und Simulation brachte einige Schwierigkeiten mit sich:

- Beide Teilsimulationen b und c enthalten eine große Anzahl von Zustandsvariablen, von denen nur wenige gemessen werden. (In Teilsimulation b wird eine von 14 Variablen, in Teilsimulation c werden drei von 29 Variablen gemessen.)
- Die Gleichungen in Teilsimulation b sind Differentialgleichungen des Ortes, bei der Entwicklung des Algorithmus waren aber nur Differentialgleichungen der Zeit und dadurch entstehende Abhängigkeiten zwischen Messungen zu verschiedenen Zeitpunk ten berücksichtigt worden.
- Die verwendeten Differentialgleichungen sind größtenteils sehr komplex, was einen hohen Rechenaufwand bei der Reconciliation erfordert.

Die Summe dieser Probleme erforderte Zugeständnisse und kreative Lösungsansätze bei der Anwendung des Algorithmus zur Data Reconciliation:

Um Performanceprobleme zu vermeiden, beschränkte ich die Reconciliation auf die Teil simulation b und innerhalb dieser Teilsimulation auf die Zustandsvariable T_g. Der Grund für diese Wahl lag einerseits darin, daß im früheren Einsatz des Simulationssystems hier besonders große Abweichungen zwischen Prognose und Realität auftraten, andererseits hat die Gastemperatur einen besonders großen Einfluß auf die ξ_i und somit auf die Bildung von ACAC im Reaktor.

Der Algorithmus zur Data Reconciliation mußte umformuliert werden. An die Stelle der Zeitabhängigkeit der Zustandsvariablen trat dabei die Abhängigkeit von der eindimensionalen

Orts koordinate. Statt den Messungen zu verschiedenen Zeitpunkten wurden nun die Messungen von T_g an verschiedenen Orten zum selben Zeitpunkt betrachtet.

Schließlich weitete ich die Reconciliation auf die Betrachtung der Wandtemperatur T_w aus, die ja in den Gleichungen (4.1) als Parameter vorkommt. Dabei ist zu beachten, daß T_w an unterschiedlichen Orten entlang des Reaktionsrohres unterschiedliche Werte annimmt. In der Simulation wird das durch die Unterteilung des Reaktionsrohres in Sektionen mit jeweils konstanter Wandtemperatur T_{wk} modelliert. Die Anpassung all dieser T_{wk} (mehr als 20 Variablen!) in der Reconciliation als Systemparameter hätte einen kaum zu bewältigenden Aufwand an Rechen zeit zur Folge gehabt. Gelöst wurde dieses Problem, indem die T_{wk} zu Gruppen zusammen gefaßt wurden. Die partiellen Ableitungen – und somit die Updates – wurden für jede Gruppe gemein sam bestimmt. Dies geschah durch Modifikation der Approximationsformel für die par tiellen Ableitungen (2.5):

$$\frac{\partial \mathrm{v}(T_g,T_w)}{\partial T_{w,k}} \approx \begin{array}{l} \frac{1}{h}\mathrm{v}(T_g,T_{w,1},\ldots,T_{w,i-1},T_{w,i}+h,\ldots,T_{w,j}+h,T_{w,j+1},\ldots,T_{w,N}) \\ \qquad -\frac{1}{h}\mathrm{v}(T_g,T_{w,1},\ldots,T_{w,i},\ldots,T_{w,j},\ldots,T_{w,N}) \end{array} \quad \text{für} \quad i \le k \le j$$

Die Eingangskonzentrationen wurden ebenso wie die Durchflußrate des Stoffeintrags nicht der Reconciliation unterworfen, da die Konzentrationen als konstant angenommen werden kön nen und die Durchflußrate mit einem sehr genauen Meßgerät (1% Toleranz) gemessen wird.

Die systematischen Abweichungen der Meßgeräte für Wand- und Gastemperatur wurden bestimmt, wie im Abschnitt 2.7.2 beschrieben. Außerdem wurde eine Möglichkeit geschaffen, die von der Simu lation verwendeten Systemparameter einer Anpassung wie in Abschnitt 2.5 zu unter werfen. Dabei ist jedoch zu beachten, daß im On-line-Eins atz nur wenige dieser Systempara meter gleich zeitig betrachtet werden können, da der Rechenaufwand sonst zu groß wird.

Es bleibt festzustellen, daß große Teile der Teilsimulation c von der Reconciliation aus genommen bleiben, da für diese Teile zur Zeit keine relevanten Meßgeräte installiert sind. Die se Konstellation stellt hohe Ansprüche an die Korrektheit der Teilsimulation c, so daß Tests dieser Konstruktion mit realen Daten unerläßlich bleiben.

4.2.4 Test des Systems

Ein Test des so entwickelten Systems an realen Meßwerten vom ACAC-Reaktor wäre sicher von großem Interesse gewesen. Leider war jedoch bis zum Ende dieser Arbeit das Prozeß leitsystem am Reaktor, daß die Meßwerte on-line zur Verfügung stellen und computer gestützt archi vieren sollte, nicht vollständig betriebsbereit. Eine weitere Schwierigkeit stellte die Tatsache dar, daß in den vorliegenden Fortran-Quelltexten die Teilsimulation b wie er wähnt nicht vollständig war. Um die Software dennoch testen zu können, implementierte ich in der Simulation des Reaktors einige zusätzliche Funktionen:

Mit dem Simulationsteil des Programms können nun Meßwerte für alle acht Meßgeräte am betrachteten Teil des Reaktors erzeugt werden. Diese Messungen lassen sich auch mit systematischen und zufälligen Fehlern (Bias bzw. gaußverteiltes Rauschen) versehen. Die so modifizierten Werte können in eine Datei gespeichert und/oder in eine Named Pipe gesendet werden.

Der Reconciliation-Teil des Programms ist so gestaltet, daß die Meßwerte entweder (off-line) aus einer Datei oder (on-line) aus der Named Pipe gelesen und verarbeitet werden können. Auf diese Weise konnte zumindest die grundlegende Funktionalität von Programm und Algorithmus geprüft werden.

Die unter diesen Voraussetzungen erreichten Ergebnisse bei der Erkennung systematischer und zufälliger Fehler in den Temperaturmessungen demonstrieren die Leistungsfähigkeit des Algorithmus. Als Beispiel seien hier die Ergebnisse einer Testreihe zur Erkennung systematischer Fehler vorgestellt (alle Angaben in K, Ergebnisse nach der Auswertung von 272 Meßwerttupeln):

Meßgerät	Standardabweichung	Bias wahr	Bias ermittelt
A	2,0	0,0	0,2
B	2,0	5,0	6,4
C	2,0	-7,0	-7,0
D	2,0	3,0	5,0
E	2,0	-3,0	-0,8
F	2,0	10,0	9,3
G	2,0	5,0	1,3

Es ist erkennbar, daß an allen Meßgeräten das Vorhandensein, das Vorzeichen und teilweise sogar der Betrag der systematischen Fehler zuverlässig erkannt wurden.

Gleichzeitig wurde beobachtet, wie ein (dem System fehlerfrei vorgegebener) Systemparameter verändert wurde (alle Angaben in bar):

Parameter	wahrer Wert	vorgegeben	ermittelt
A	1,055	1,055	1,34

Hier ist eine deutliche Fehlanpassung zu verzeichnen, deren Ursache in späteren Tests erhellt werden konnte: Die Sensitivität der Simulation gegenüber dem ausgewählten Parameter war relativ gering, d.h. auch mit dem fehlerhaften Parameter konnten noch qualitativ gute Simulationsergebnisse gewonnen werden. Somit ist für den Algorithmus der Zwang, diesen Parameter korrekt einzustellen, nicht besonders stark, was zu deutlichen Schwankungen bei der Bestimmung dieses Parameters führte. Hier sind noch weitere Anstrengungen nötig, um zu klären, welche Qualität die Anpassung von Parametern erreicht, auf deren Änderung die Simulation besonders empfindlich reagiert.

Ein weiteres Ergebnis dieser Tests betrifft die Performance des Programms: Bei dem beschriebenen Experiment betrug der Mittelwert der verbrauchten CPU-Zeit je Meßwerttupel 3,65 Sekunden auf einem unterdurchschnittlich ausgestatteten PC.[29] In diesem Experiment wurden sieben Meßgeräte und ein Systemparameter berücksichtigt. Geht man ferner davon aus, daß das Prozeßleitsystem des Reaktors ein Meßwerttupel aller 20 Sekunden liefert, verfügt das Programm über genügende Leistungsreserven für den On-line-Betrieb, auch wenn mehr Systemparameter oder zusätzliche Meßgeräte in die Betrachtung einbezogen werden sollen.

[29] IBM-kompatibler PC mit 486-DX/2-CPU (80 MHz)

5 Zusammenfassung

Neben der Vorstellung eines neuen Algorithmus zur modellbasierten Entfernung zufälliger Fehler aus Messungen an dynamischen Systemen war es Ziel dieser Arbeit, den Algorithmus zu erweitern und an ausgewählten Problemstellungen zu testen.

Die vorgestellten Erweiterungen betreffen:

- die Identifikation von Parametern des dynamischen Systems
- die Einführung einer Zeitfenstertechnik zur Gewährleistung der Echtzeitfähigkeit
- die Identifikation von systematischen Fehlern (in Form eines Bias)
- die Minimierung von Meßfehlern bei Eingangsgrößen des Systems
- die automatische Adaption eines Parameters des Algorithmus (Anpaßrate)

Nach der Entwicklung eines Testprogrammes konnte durch Tests mit den Modellen dreier verschiedener dynamischer Systeme die Leistungsfähigkeit des Algorithmus und seiner Modifikationen gezeigt werden. Gleichzeitig wurden wichtige Erfahrungen für den praktischen Einsatz gewonnen, die teilweise zu einer weiteren Verbesserung des Algorithmus führten. Bei diesen Modellen handelte es sich um:

- eine einfache Differentialgleichung: $\dot{x}(t) = ax - x^2$
- das Modell eines Räuber-Beute-Systems
- das Modell eines einfachen Rührkesselreaktors (CSTR)

Tests mit dem Modell des CSTR wurden herangezogen, um Vergleiche mit dem NDDR-Algorithmus – einem etablierten Verfahren auf dem Gebiet der Data Reconciliation – durchzuführen. Die Ergebnisse dieses Vergleichs zeigen mindestens eine Gleichwertigkeit des neuen Algorithmus bezüglich der Reduzierung zufälliger Fehler. Gleichzeitig konnten die wichtigsten Vorteile des neuen Algorithmus herausgearbeitet werden:

- bessere Eignung für Echtzeitanwendungen
- einfache Implementierung durch Modellinvarianz und geringe Anzahl von Tuningparametern
- sehr gute Performance

Der letzte Teil der Arbeit demonstrierte die Flexibilität und Leistungsfähigkeit des Verfahrens durch die Lösung eines komplexen Problems aus der Tätigkeit der Wacker Chemie GmbH (Burghausen/Bayern): Zur Simulation und optimalen Kontrolle eines chemischen Reaktors wurde aus einem bestehenden Simulationssystem eine interaktive, echtzeitfähige Software zur On-line-Korrektur von Meßdaten dieses Reaktors entwickelt. Dieses Programm konnte zwar nicht im Praxiseinsatz getestet werden, es wurden jedoch Versuche mit zufällig und systematisch gestörten Meßdaten durchgeführt, die aus einer Simulation des Reaktors stammen. Die Ergebnisse dieser Versuche zeigten die Fähigkeit der Softwarelösung, systematische und zufällige Fehler in den Messungen zu erkennen und effektiv zu verringern.

Anhang

A-1 Partielle Ableitungen der Vorwärtsschätzung

An dieser Stelle sollen einige der komplexeren Formeln aufgeführt werden, die zur Näherung der partiellen Ab leitungen der Vorwärtsschätzungen verwendet wurden. Diese Formeln wurden in vielen Tests eingesetzt, später aber durch einfachere Näherungen (2.5) verdrängt, um Rechenzeit zu sparen.

Zunächst die partielle Ableitung der Vorwärtsschätzung nach einer Zustandsvariablen:

$$\mathrm{v}_i\left(\hat{x}_n\right)_j := \frac{\partial \mathrm{v}\left(\hat{\mathbf{x}}_n\right)_j}{\partial \hat{x}_{n,i}}$$

Zur Ermittlung von v kommt das Runge-Kutta-Verfahren zum Einsatz:

$$\begin{aligned}
&\mathrm{v}(\hat{\mathbf{x}}_n) = \hat{\mathbf{x}}_n + \frac{\Delta t}{6}\left(\mathbf{L}_{n,1} + 2\mathbf{L}_{n,2} + 2\mathbf{L}_{n,3} + \mathbf{L}_{n,4}\right) \\
&\mathbf{L}_{n,1} = f(\hat{\mathbf{x}}_n, t_n) \\
&\mathbf{L}_{n,2} = f(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} L_{n,1}, t_n + \tfrac{\Delta t}{2}) \\
&\mathbf{L}_{n,3} = f(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} L_{n,2}, t_n + \tfrac{\Delta t}{2}) \\
&\mathbf{L}_{n,4} = f(\hat{\mathbf{x}}_n + \Delta t\, L_{n,3}, t_n + \Delta t)
\end{aligned}$$

Damit ergibt sich für die partielle Ableitung:

$$\begin{aligned}
\frac{\partial \mathrm{v}\left(\hat{\mathbf{x}}_n\right)_j}{\partial \hat{x}_{n,i}} &= \frac{\partial}{\partial \hat{x}_{n,i}}\left(\hat{\mathbf{x}}_n + \frac{\Delta t}{6}\left(\mathbf{L}_{n,1} + 2\mathbf{L}_{n,2} + 2\mathbf{L}_{n,3} + \mathbf{L}_{n,4}\right)\right)_j \\
&= \delta_{ij} + \frac{\Delta t}{6}\left(\frac{\partial}{\partial \hat{x}_{n,i}}\mathbf{L}_{n,1} + 2\frac{\partial}{\partial \hat{x}_{n,i}}\mathbf{L}_{n,2} + 2\frac{\partial}{\partial \hat{x}_{n,i}}\mathbf{L}_{n,3} + \frac{\partial}{\partial \hat{x}_{n,i}}\mathbf{L}_{n,4}\right)_j
\end{aligned}$$

Die Ableitungen der L_n bestimmt man nun mit der allgemeinen Kettenregel (man beachte die vereinfachte Schreibweise für die Ableitungen der f):

$$\frac{\partial}{\partial \hat{x}_{n,i}} \mathbf{L}_{n,1} = \sum_k \frac{\partial f(\hat{\mathbf{x}}_n, t_n)}{\partial \hat{x}_{n,k}} \cdot \frac{\partial \hat{x}_{n,k}}{\partial \hat{x}_{n,i}} = \frac{\partial f(\hat{\mathbf{x}}_n, t_n)}{\partial \hat{x}_{n,i}} \mathrel{\hat{=}} f_{(i)}(\hat{\mathbf{x}}_n, t_n)$$

$$\frac{\partial}{\partial \hat{x}_{n,i}} \mathbf{L}_{n,2} = \sum_k \left[\frac{\partial f(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,1}, t_n + \tfrac{\Delta t}{2})}{\partial \left(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,1} \right)_k} \cdot \frac{\partial \left(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,1} \right)_k}{\partial \hat{x}_{n,i}} \right]$$

$$= \sum_k \left[f_{(k)}(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,1}, t_n + \tfrac{\Delta t}{2}) \cdot \left(\delta_{ik} + \tfrac{\Delta t}{2} \left(\frac{\partial \mathbf{L}_{n,1}}{\partial \hat{x}_{n,i}} \right)_k \right) \right]$$

$$\frac{\partial}{\partial \hat{x}_{n,i}} \mathbf{L}_{n,3} = \sum_k \left[\frac{\partial f(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,2}, t_n + \tfrac{\Delta t}{2})}{\partial \left(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,2} \right)_k} \cdot \frac{\partial \left(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,2} \right)_k}{\partial \hat{x}_{n,i}} \right]$$

$$= \sum_k \left[f_{(k)}(\hat{\mathbf{x}}_n + \tfrac{\Delta t}{2} \mathbf{L}_{n,2}, t_n + \tfrac{\Delta t}{2}) \cdot \left(\delta_{ik} + \tfrac{\Delta t}{2} \left(\frac{\partial \mathbf{L}_{n,2}}{\partial \hat{x}_{n,i}} \right)_k \right) \right]$$

$$\frac{\partial}{\partial \hat{x}_{n,i}} \mathbf{L}_{n,4} = \sum_k \left[\frac{\partial f(\hat{\mathbf{x}}_n + \Delta t \, \mathbf{L}_{n,3}, t_n + \Delta t)}{\partial \left(\hat{\mathbf{x}}_n + \Delta t \, \mathbf{L}_{n,3} \right)_k} \cdot \frac{\partial \left(\hat{\mathbf{x}}_n + \Delta t \, \mathbf{L}_{n,3} \right)_k}{\partial \hat{x}_{n,i}} \right]$$

$$= \sum_k \left[f_{(k)}(\hat{\mathbf{x}}_n + \Delta t \, \mathbf{L}_{n,3}, t_n + \Delta t) \cdot \left(\delta_{ik} + \Delta t \left(\frac{\partial \mathbf{L}_{n,3}}{\partial \hat{x}_{n,i}} \right)_k \right) \right]$$

Analog lassen sich auch die partiellen Ableitungen der Vorwärtsschätzung nach System para-metern

$$\frac{\partial \mathrm{v}(\hat{\mathbf{x}}_m, \hat{\mathbf{a}})_j}{\partial \hat{a}_i}$$

ermitteln. Auf eine ausführliche Darstellung der verwendeten Formeln möchte ich hier jedoch verzichten, da sie sehr ähnlich zu den oben aufgeführten sind.

A-2 Beschreibung der Testsoftware

Da Test und Weiterentwicklung des Algorithmus einen hohen Arbeitsaufwand mit sich brachten, erschien es lohnenswert, Software zu entwickeln, mit deren Hilfe die Testreihen möglichst komfortabel abgewickelt werden konnten. Außerdem sollte zu Demonstrationszwecken die Wirkung des Algorithmus grafisch darstellbar sein.

Die Software verfügt über folgende Leistungsmerkmale:

- Ein- und Ausgabe von Meß- und Schätzwerten in verschiedenen Datenbankformaten (dies erleichterte u. a. statistische Auswertungen unter Zuhilfenahme von Standardsoftware)
- interaktive Erzeugung von fehlerbehafteten Meßreihen (systematische Fehler, zufällige Fehler, falsch bestimmte Systemparameter)
- Darstellung beliebiger Schätzwerte, Meßwerte und wahrer Werte im x-t-Diagramm
- einfache Auswertungsfunktionen für die Qualität des Ergebnisses (MSE)
- alle Parameter des Algorithmus (Anpaßrate usw.) sind interaktiv wählbar
- Reconciliation im Hintergrund möglich durch Multi-Thread-Architektur

Allerdings bestanden auch einige Einschränkungen:

- On-line-Reconciliation ist nicht vorgesehen
- aus Performancegründen ist das mathematische Modell in einigen Quelltext-Routinen festgelegt, um ein anderes System zu betrachten, ist also eine Änderung des Quelltextes notwendig

Entsprechend den Wünschen der Wacker Chemie GmbH verwendete ich zur Programmierung das Pascal-Entwicklungswerkzeug Delphi 2. Da der Gesamtumfang des Programms etwa 3900 Zeilen beträgt, sind nur die Quelltexte der wichtigsten Module des Programms im nächsten Anhang wiedergegeben. Um das Verständnis dieser Quelltexte zu erleichtern, sollen hier einige Grundlagen des Programms erörtert werden.

Die wichtigsten Module

`sysdef.pas` – In dieser Unit werden die grundlegenden Eigenschaften des betrachteten Systems definiert (Anzahl und Art der Zustandsvariablen, Parameter und Inputs, die Differentialgleichungen des dynamischen Systems usw.). Für jede der Variablen des Systems wird eine Instanz der Klasse **`TVariableDescription`** generiert. Um Änderungen am Differentialgleichungssystem vorzunehmen, muß der Quelltext dieses Moduls geändert werden. Eine genauere Beschreibung dieses Vorgehens folgt weiter unten.

`testdef.pas` – Diese Unit enthält lediglich die Funktion **`xt`**, die den zeitlichen Verlauf der Inputvariablen beschreibt. Dies ist für die Erzeugung von Meßreihen mit Hilfe des Modells notwendig.

mathutil.pas – Diese Unit stellt neben elementaren Vektoroperationen (Addition, Skalarmultiplikation...) auch Funktionen und Prozeduren für das Runge-Kutta-Verfahren (**RungeKutta**, **RungeKuttaMulti**) und die Ermittlung der partiellen Ableitungen der Vorwärts schätzungen (**RungeKuttaDa**, **RKDaApprox**) bereit.

dataclas.pas – In dieser Unit wird die Klasse **TMeasurement** definiert, die zur Speicherung der Meßwerte, Schätzwerte und wahren Werte innerhalb des Programms dient. Ferner wird eine Klasse **TMeasurementList** für dynamische Listen aus **TMeasurement**s definiert, die unter anderem auch Methoden für das Erzeugen von Meßreihen und für das Laden und Speichern der Daten besitzt.

glaetthr.pas – Diese Unit enthält die Thread-Klasse **TGlaettThread**, in deren **Execute**-Methode die Reconciliation abläuft. Wichtige Methoden für die Reconciliation stellen **ComputeUpdates** und **DoUpdate** dar. Von Interesse sind weiterhin die Klasse **TTimeWindow** (eine dynamische Liste, die die Zeitfenstertechnik realisiert) sowie das Symbol **SlowDeriv**, das durch bedingte Compilierung entscheidet, welches Verfahren zur Bestimmung der partiellen Ableitungen eingesetzt wird.

Die Systemdefinition

Im Modul **sysdef.pas** ist das Differentialgleichungssystem „hart codiert“, so daß nur durch Modifikation des Quelltextes andere Systeme betrachtet werden können. Dieses Vorgehen sei an dieser Stelle näher erläutert.

Zunächst ist es notwendig, die Variablen des Systems zu definieren. Als „Variable“ in diesem Zusammenhang werden angesehen:

- Zustandsvariablen
- Inputvariablen
- anzupassende Systemparameter

Für jede derartige Variable ist dem Aufzählungstyp **TSysVars** ein Bezeichner hinzuzufügen, mit dem diese Variable in Zukunft referenziert wird. Im Initialisierungsteil der Unit ist anschließend für jede Variable einmal die Prozedur **AddSysVar** aufzurufen, die folgende Parameter erwartet:

- den Bezeichner der Variablen (s. o.)
- einen String (max. 7 Zeichen, keine Leerzeichen), der als Name der Variablen im Programm verwendet wird
- den Typ der Variablen (eine der Konstanten **tvtState**, **tvtInput** oder **tvtParameter**)
- einen booleschen Wert, der angibt, ob Meßwerte für diese Variable vorhanden sein sollen
- eine positive Gleitkommazahl, die einen Bezugswert für die Variable definiert. Dieser Wert dient dazu, die unterschiedlichen Größenordnungen der einzelnen Variablen zu

berücksichtigen. Auf diesen Referenzwert beziehen sich beispielsweise die Angaben für die Standardabweichung bei der Erzeugung verrauschter Meßwerte.

- einen booleschen Wert, der angibt, ob systematische Fehler (Bias) dieser Variablen betrachtet werden sollen

Nun ist die Funktion **`fx`** zu modifizieren, um die Funktionen f aus Formel (1.1) zu implementieren. Als Beispiel kann die Implementierung der Funktionen der CSTR-Simulation dienen, wie sie in den Quelltexten des Anhangs A-3 realisiert ist.

In ähnlicher Art und Weise ist die Funktion **`DfxDa`** zu modifizieren, so daß diese die partiellen Ableitungen der f nach der Variablen mit dem Index a zurückgibt. Die Implementierung dieser Funktion ist nur erforderlich, wenn das Symbol **`SlowUpdates`** in der Unit **`GlaetThr`** definiert wird. (Dies hat zur Folge, daß die Formeln aus Anhang A-1 statt der einfacheren Approximationen eingesetzt werden.)

Anschließend ist der Konstanten **`SystemName`** ein sinnvoller Name für das System zuzuweisen.

Gegebenenfalls ist noch die Funktion **`xt`** der Unit **`TestDef`** (s. o.) für die Testdatengenerierung zu ändern.

A-3 Ausgewählte Quelltexte der Testsoftware

```
unit SysDef;

{*************************************************************************}
{*                                                                       *}
{* Datei:          SysDef.PAS                                            *}
{* Typ:            Unit                                                  *}
{* Beschreibung:   definiert das zu verwendende DGL-System               *}
{*                 ist vom Nutzer zu ändern, um andere DGL-Systeme zu    *}
{*                 betrachten, die Änderungen dabei sollten sich nur auf *}
{*                 solche Teile beziehen, die mit (!) gekennzeichnet sind *}
{*                                                                       *}
{*                 aufbauend auf diesen Definitionen wird der Datentyp   *}
{*                 TXArray erzeugt                                       *}
{*                                                                       *}
{* Klassen:        TVariableDescription                                  *}
{*                                                                       *}
{* Autor:          Olaf Lummer, Leipzig                                  *}
{*                                                                       *}
{* Letzte Änderung: 10.05.97 von O.L.                                    *}
{*                                                                       *}
{*************************************************************************}


    interface
    {*** Definition der Genauigkeit für das gesamte Programm}
    Type TStdZahl=Double;


    {Zur Benennung der "Variablen" des Systems:
     STATE-VARIABLE = Variable, die den Zustand des Systems verkörpert
                      Differentialgleichung xPunkt=fx(x,t) ist bekannt.
                      Wird gemessen.
     PARAMETER      = Variable, die das Systemverhalten beeinflußt.
                      Ist zeitlich konstant.
                      Ist als Schätzung bekannt.
     INPUT-GROESSE  = Variable, die das Systemverhalten beeinflußt.
                      Ändert sich nicht vorhersehbar.
                      Ändert sich mit geringer Dynamik,
                             daher als konstant im Zeitfenster betrachtet
                      Wird gemessen.}


    type TVariableType=(tvtState,tvtParameter,tvtInput);
    type TVariableDescription=class(TObject)
             protected
               FVarName :string;
               FVarType :TVariableType;
               FMeasured:Boolean;
               FEstimateBias:Boolean;
               FSignalLevel:TStdZahl;
               function RIsState:Boolean;
               function RIsInput:Boolean;
               function RIsParameter:Boolean;
               procedure SetEstimateBias(Value:Boolean);
             Public
               constructor Create(const aName:String; aType:TVariableType; aMeasured:Boolean;
                                  aSignalLevel:TStdZahl; aEstimateBias:Boolean);
               property VarName:string read FVarName;
               property VarType:TVariableType read FVarType;
               property Measured:Boolean read FMeasured;
               property EstimateBias:Boolean read FEstimateBias write SetEstimateBias;
               property IsState:Boolean read RIsState;
               property IsInput:Boolean read RIsInput;
               property IsParameter:Boolean read RIsParameter;
               property SignalLevel:TStdZahl read FSignalLevel write FSignalLevel;
                      {gibt den durchschittlichen Betrag der Vaiablen an, zu Skalierungszwecken}
          end;

    {******  Definition des zu approximierenden DGL-Systems ******}

    {Anzahl und Benennung der Variablen}
{!} Const SystemName='System: Liebman et al.';
{!} type  TSysVars=(VarA,VarT,VarA0,VarT0);

    Const NumVars    =ord(High(TSysVars))+1;    {die Gesamtanzahl der Variablen}
    Const MaxVarIndex=ord(High(TSysVars));      {der größte Variablenindex}

    Type TXArray           =array [0..MaxVarIndex] of TStdZahl;
    Type TSysVarIndexArray =array [Low(TSysVars)..High(TSysVars)] of TStdZahl;
    Type PXArray            =^TXArray;

    var  SysVariables:array [0..MaxVarIndex] of TVariableDescription;


    {die beteiligten Differentialgleichungen}
    function fx(t:TStdZahl;const x:TXArray):TXArray;
             {die (nur für Zustandsvariablen bekannte) DGL 1.Ordnung: x° (t)=fx(xt(t))}
```

```
    function DfxDa(const Aindex:Byte; t:TStdZahl; x:TXArray):TXArray;
             {Partielle Ableitung von fx[] nach der Variablen A,
              wird nur für die langsameren partiellen Ableitungen der Vorw.-Schätz. benötigt}

    implementation
    Uses SysUtils,Math;

    {Fixe Systemparameter}
{!} Const q=10;
{!}       V=1000;
{!}       dHt=-27000;
{!}       p=0.001;
{!}       Cp=1;
{!}       U=5E-4;
{!}       Ar=10;
{!}       Tc=3.40;
{!}       k0=7.86E12;
{!}       Ea=14090;
{!}       AlphaD=1;
{!}
{!}      {Skalierung mit Referenzwerten}
{!}       At=1E-6;
{!}       Tt=100;


    function fx(t:TStdZahl;const x:TXArray):TXArray;
    var Xi:TSysVarIndexArray absolute x;
    var fxi:TSysVarIndexArray absolute Result;

{!} var k:TStdZahl;
{!} begin
{!} k:=k0*exp(-Ea/(xi[VarT]*Tt));
{!}   fxi[VarA]:=(q/V)*(xi[VarA0]-xi[VarA])-AlphaD*k*xi[VarA];
{!}   fxi[VarT]:=(q/V)*(xi[VarT0]-xi[VarT])+AlphaD*k*xi[VarA]*(-dHt)*At/(p*Cp*Tt)
{!}                -U*Ar*(xi[VarT]-Tc)/(p*Cp*V);
{!}   fxi[VarA0]:=0;
{!}   fxi[VarT0]:=0;
{!} end; {fx}

    function DfxDa(const Aindex:Byte; t:TStdZahl; x:TXArray):TXArray;
             {Partielle Ableitung von fx[] nach der Variablen A}
    var Xi:TSysVarIndexArray absolute X;
    var dfxi:TSysVarIndexArray absolute Result;

{!} var k,kdt:TStdZahl;
{!} begin
{!}  {raise EInvalidOp.Create('dFXdX NICHT DEFINIERT');}
{!}  dfxi[VarA0]:=0;
{!}  dfxi[VarT0]:=0;
{!}
{!}  case TSysVars(AIndex) of
{!}       VarA  : begin
{!}                 k:=k0*exp(-Ea/(xi[VarT]*Tt));
{!}                 dfxi[VarA]:=-q/V-AlphaD*k;                 {df[A] nach dA }
{!}                 dfxi[VarT]:=AlphaD*k*(-dHt)*At/(p*Cp*Tt); {df[T] nach dA}
{!}               end;
{!}       VarT  : begin
{!}                 k:=k0*exp(-Ea/(xi[VarT]*Tt));
{!}                 kdt:=Ea/(sqr(xi[VarT])*Tt);
{!}                 dfxi[VarA]:=-AlphaD*xi[VarA]*k*kdT;{df[A] nach dT }
{!}                 dfxi[VarT]:=-q/V+AlphaD*k*kdT*xi[VarA]*(-dHt)*At/(p*Cp*Tt)
{!}                                 -U*Ar/(p*Cp*V);    {df[T] nach dT}
{!}               end;
{!}       VarA0 : begin
{!}                 dfxi[VarA]:=q/V;  {df[A] nach dA0}
{!}                 dfxi[VarT]:=0;    {df[T] nach dA0}
{!}               end;
{!}       VarT0 : begin
{!}                 dfxi[VarA]:=0;    {df[A] nach dT0}
{!}                 dfxi[VarT]:=q/V;  {df[T] nach dT0}
{!}               end;
{!}  else raise EInvalidArgument.Create('ungültiger Index beim Aufruf von dfxda');
{!}  end;
{!} end; {DfxDa}


    {*** TVariableDescription ***}

    constructor TVariableDescription.Create(const aName:String; aType:TVariableType; aMeasured:Boolean;
                                        aSignalLevel:TStdZahl; aEstimateBias:Boolean);
    begin
      if (aType=tvtParameter)and aMeasured
      then raise EInvalidArgument.Create('Versuch, Parameter als gemessen zu deklarieren');

      inherited Create;
      FVarName:=aName;
      FVarType:=aType;
      FMeasured:=aMeasured;

      if (aSignalLevel<=0)
      then raise EInvalidArgument.Create('Versuch, negativen SignalLevel zu setzen');

      FSignalLevel:=aSignalLevel;
      EstimateBias:=aEstimateBias;
```

```
    end;

    function TVariableDescription.RIsState:Boolean;
      begin Result:=(FVarType=tvtState) end;

    function TVariableDescription.RIsInput:Boolean;
      begin Result:=(FVarType=tvtInput) end;

    function TVariableDescription.RIsParameter:Boolean;
      begin Result:=(FVarType=tvtParameter) end;

    procedure TVariableDescription.SetEstimateBias(Value:Boolean);
    begin
      if Value and RIsParameter
      then EInvalidArgument.Create('Für Parameter kann kein Bias ermittelt werden.');
      FEstimateBias:=Value;
    end;

    {*** Vereinbarung der Systemvariablen ***}
    procedure AddSysVar(Index:TSySVars;const Name:String; aType:TVariableType; Measured:Boolean;
                        SignalLevel:TStdZahl; EstimateBias:Boolean);
    begin
    SysVariables[ord(Index)]:=TVariableDescription.Create(Name,aType,Measured,SignalLevel,EstimateBias);
    end;


    begin
{!}   AddSysVar(VarA ,'A'  ,tvtState    ,True  ,1, False);
{!}   AddSysVar(VarT ,'T'  ,tvtState    ,True  ,1, False);
{!}   AddSysVar(VarA0,'A0' ,tvtInput    ,True  ,1, False);
{!}   AddSysVar(VarT0,'T0' ,tvtInput    ,True  ,1, False);
    end.
```

```
unit TestDef;

{***************************************************************************}
{*                                                                         *}
{* Datei:          TestDef.PAS                                             *}
{* Typ:            Unit                                                    *}
{* Beschreibung:   enthält Funktionen und Konstanten zur Generierung der   *}
{*                 Testdaten                                               *}
{*                                                                         *}
{*                 ist vom Nutzer zu ändern, um den Verlauf der            *}
{*                 Inputvariablen zu beeinflussen                          *}
{*                                                                         *}
{* Klassen:        ---                                                     *}
{*                                                                         *}
{* Autor:          Olaf Lummer, Leipzig                                    *}
{*                                                                         *}
{* Letzte Änderung: 10.06.97 von O.L.                                      *}
{***************************************************************************}

interface
Uses SysDef;

function xt(const index:byte; t:TStdZahl):TStdZahl;
         {die ungestörte Funktion, ist nur zur Erzeugung der Testdaten
          und nur für Eingangsgroesssen notwendig}

implementation
uses SysUtils, Math;

function xt(const index:byte; t:TStdZahl):TStdZahl;
begin
  case TSysVars(index) of
     VarA0:if t<100 then xt:=6.5
                    else if t<150 then xt:=8.5
                                  else xt:=5.5;
     VarT0:xt:=3.5;
  else raise EInvalidOp.Create('xt für Variable angefordert, die keine Eingangsgroesse ist')
  end;
end; {xt}


end.
```

```
unit MathUtil;

{**************************************************************************}
{*                                                                        *}
{* Datei:          MathUtil.PAS                                           *}
{* Typ:            Unit                                                   *}
{* Beschreibung:   verschiedene mathematische Funktionen:                 *}
{*                 - Grundfunktionen (Min,Max,MaxAbs,Kronecker)           *}
{*                 - Vektorfunktionen                                     *}
{*                 - Erzeugung StdNormalverteilter Zufallswerte           *}
{*                 auf das in SysDef beschriebene System bezogene Funktionen:*}
{*                 - Runge-Kutta-Näherungen                               *}
{*                                                                        *}
{* Klassen:        ---                                                    *}
{*                                                                        *}
{* Autor:          Olaf Lummer, Leipzig                                   *}
{*                                                                        *}
{* Letzte Änderung: 29.5.97 von O.L.                                      *}
{**************************************************************************}


interface
Uses Sysdef;

{grundlegende Funktionen}

function Min(x1,x2:TStdZahl):TStdZahl;
function Max(x1,x2:TStdZahl):TStdZahl;
function MaxAbs(x1,x2:TStdZahl):TStdZahl;
function Kronecker(a,b:integer):integer;

{einfache Vektor-Funktionen}
procedure FillArray(var Ar:array of TStdZahl; FillValue:TStdZahl);
          {füllt das Array mit dem FillValue}
procedure MulArrayScalar(var Ar:array of TStdZahl; Scalar:TStdZahl);
          {Multipliziert das Array mit einem Skalarwert}
procedure MulArrayArray(var A1:Array of TStdZahl;const A2:Array of TStdZahl);
          {Multipliziert jedes Element in A1 mit dem entsprechenden Element in A2}
procedure AddArrayScalar(var Ar:array of TStdZahl; Scalar:TStdZahl);
          {Addiert den Skalarwert zu jedem Element}
procedure AddArrayArray(var A1:Array of TStdZahl;const A2:Array of TStdZahl);
          {Addiert A2 zu A1}
procedure SubArrayArray(var A1:Array of TStdZahl;const A2:Array of TStdZahl);
          {Subtrahiert A2 von A1}
procedure InvArray(var Ar:array of TStdZahl);
          {bildet 1/x für jedes Element im Array}

function ElementSumArray(const Ar:array of TStdZahl):TStdZahl;
          {bildet die Summe aller Elemente des Arrays}
function SqrSumArray(const Ar:array of TStdZahl):TStdZahl;
          {bildet die Summe der Quadrate aller Elemente des Arrays}
function MaxAbsArray(const Ar:array of TStdZahl):TStdZahl;
          {ermittelt den größten Absolutwert des Arrays}
procedure LimitArray(var Ar:array of TStdZahl; MaxValue:TStdZahl);
          {begrenzt die Zahlen im Array auf (-MaxValue..Maxvalue)}


{Runge-Kutta-Funktionen}

function RungeKutta(X:TXarray;t0,DeltaT:TStdZahl;Steps:Integer):TXArray;
         {macht eine Runge-Kutta-Näherung in (Steps) Schritten ausgehend
          von X zum Zielpunkt (t0+DeltaT)}
function RungeKuttaDa(X:TXarray;t,DeltaT:TStdZahl;a:byte):TXArray;
         {gibt die partielle Ableitung der Runge-Kutta-Näherung nach der Systemvariablen a zurück}
procedure RungeKuttaMulti(X:TXarray;var XN, XNDa:TXarray;t,DeltaT:TStdZahl;a,Steps:byte);
          {gibt die Runge-Kutta-Näherung und deren Ableitung über mehrere Schritte an}
procedure RKDaApprox(X:TXarray;var XN, XNDa:TXarray;t,DeltaT:TStdZahl;a,Steps:byte;DeltaX:TStdZahl);
          {gibt die Runge-Kutta-Näherung und deren Ableitung über mehrere Schritte an (approxmativ)}

{Zufallsfunktionen}

function StdNormVertRandom:TStdZahl;
         {liefert einen standardnormalverteilten Zufallswert}
function NormVertRandom(const EW:TStdZahl;const StdAbw:TStdZahl):TStdZahl;
         {liefert einen normalverteilten Zufallswert mit
          vorgegebenem Erwartungswert und Standardabweichung}

var EmptyXArray:TXArray; {ein komplett mit 0 gefülltes TXArray}

implementation
uses SysUtils;

...
```

Die Implementierung zahlreicher einfacher Funktionen wird an dieser Stelle weggelassen, da sie zum Verständnis nicht erforderlich ist

```
...

{*** Runge-Kutta-Näherungen ***}

function RungeKutta(X:TXarray;t0,DeltaT:TStdZahl;Steps:Integer):TXArray;
{macht eine Runge-Kutta-Näherung in (Steps) Schritten ausgehend von X zum Zielpunkt (t0+DeltaT)
 nur Zustandsvariablen werden neu berechnet, alle anderen werden 'durchgeschleift'}
var i,j:integer;
    dt:TStdZahl;{Länge eines RK-Schrittes}
```

```
    t:TStdZahl;{Zeit am Beginn des akt RK-Schrittes}
    L1,L2,L3,L4,xTemp:TXArray;
begin
  dt:=DeltaT/Steps;
  for i:=0 to Steps-1 do begin

    t:=t0+(i/Steps)*DeltaT;       {Bestimmung von L1}
    L1:=fx(t,X);

    xTemp:=L1;                    {Bestimmung von L2}
    MulArrayScalar(xTemp,0.5*dt);
    AddArrayArray(xTemp,x);
    L2:=fx(T+0.5*dt,xTemp);

    xTemp:=L2;                    {Bestimmung von L3}
    MulArrayScalar(xTemp,0.5*dt);
    AddArrayArray(xTemp,x);
    L3:=fx(T+0.5*dt,xTemp);

    xTemp:=L3;                    {Bestimmung von L4}
    MulArrayScalar(xTemp,dt);
    AddArrayArray(xTemp,x);
    L4:=fx(t+dt,xTemp);

    for j:=0 to MaxVarIndex do
     if SysVariables[j].IsState then x[j]:=x[j]+(dt/6)*(L1[j]+L4[j]+2*(L2[j]+L3[j]));
  end;
  RungeKutta:=X;
end;


function RungeKuttaDa(X:TXarray;t,DeltaT:TStdZahl;a:byte):TXArray;
        {gibt die partielle Ableitung der Runge-Kutta-Näherung nach der Systemvariablen a zurück}
        {zu den verwendeten Formeln siehe Anhang der Diplomarbeit}
var i,j,k:integer;
    xOrt,xTemp,xTemp2:TXArray;
    L1,L2,L3,L4:TXArray;
    L1Da,L2Da,L3Da,L4Da:TXArray; {die nach der Variablen a differenzierten L1..L4}
begin
    L1:=fx(t,X);                  {Bestimmung von L1}
    L1Da:=DfxDa(a,t,X);           {Bestimmung von L1Da}

    xOrt:=L1;                           {Bestimmung von L2}
    MulArrayScalar(xOrt,0.5*DeltaT);
    AddArrayArray(xOrt,x);
    L2:=fx(T+0.5*DeltaT,xOrt);

    xTemp:=L1Da;                        {Bestimmung von L2Da}
    MulArrayScalar(xTemp,0.5*DeltaT);
    xTemp[a]:=xTemp[a]+1;
    L2Da:=EmptyXArray;
    for k:=0 to MaxVarIndex do begin
       xTemp2:=DfxDa(k,T+0.5*DeltaT,xOrt);
       for j:=0 to MaxVarIndex do L2Da[j]:=L2Da[j]+xTemp[k]*xTemp2[j];
    end;

    xOrt:=L2;                          {Bestimmung von L3}
    MulArrayScalar(xOrt,0.5*DeltaT);
    AddArrayArray(xOrt,x);
    L3:=fx(T+0.5*DeltaT,xOrt);

    xTemp:=L2Da;                        {Bestimmung von L3Da}
    MulArrayScalar(xTemp,0.5*DeltaT);
    xTemp[a]:=xTemp[a]+1;
    L3Da:=EmptyXArray;
    for k:=0 to MaxVarIndex do begin
      xTemp2:=DfxDa(k,T+0.5*DeltaT,xOrt);
      for j:=0 to MaxVarIndex do L3Da[j]:=L3Da[j]+xTemp[k]*xTemp2[j];
    end;

    xOrt:=L3;                          {Bestimmung von L4}
    MulArrayScalar(xOrt,DeltaT);
    AddArrayArray(xOrt,x);
    L4:=fx(t+DeltaT,xOrt);

    xTemp:=L3Da;                        {Bestimmung von L4Da}
    MulArrayScalar(xTemp,DeltaT);
    xTemp[a]:=xTemp[a]+1;
    L4Da:=EmptyXArray;
    for k:=0 to MaxVarIndex do begin
      xTemp2:=DfxDa(k,T+DeltaT,xOrt);
      for j:=0 to MaxVarIndex do L4Da[j]:=L4Da[j]+xTemp[k]*xTemp2[j];
    end;

    for j:=0 to MaxVarIndex do
     if SysVariables[j].IsState or SysVariables[j].IsInput
        then Result[j]:=Kronecker(a,j)+(DeltaT/6)*(L1da[j]+L4da[j]+2*(L2da[j]+L3da[j]))
        else Result[j]:=Kronecker(a,j);
end;


procedure RungeKuttaMulti(X:TXarray;var XN, XNDa:TXarray;t,DeltaT:TStdZahl;a,Steps:byte);
var ts:TStdZahl;
    i,k:integer;
```

```
    xTemp,kSum:TXArray;
begin
  ts:=DeltaT/Steps;
  XN:=RungeKutta(X,t,ts,1);
  XNDa:=RungeKuttaDa(X,t,ts,a);

  For i:=1 to Steps-1 do begin
    kSum:=EmptyXArray;
    for k:=0 to MaxVarIndex do begin
      xTemp:=RungeKuttaDa(XN,t+i*ts,ts,k);
      MulArrayScalar(xTemp,XNDa[k]);
      AddArrayArray(kSum,xTemp);
    end;
    XN:=RungeKutta(XN,t+i*ts,ts,1);
    XNDa:=kSum;
  end;
end;

procedure RKDaApprox(X:TXarray;var XN, XNDa:TXarray;t,DeltaT:TStdZahl;a,Steps:byte;DeltaX:TStdZahl);
var XNLeft,XNRight:TXArray;
begin
  XN:=RungeKutta(X,t,DeltaT,Steps);
  X[a]:=X[a]-DeltaX;
  XNLeft:=RungeKutta(X,t,DeltaT,Steps);
  X[a]:=X[a]+2*DeltaX;
  XNRight:=RungeKutta(X,t,DeltaT,Steps);
  XNDa:=XNRight;
  SubArrayArray(XNDa,XNLeft);
  MulArrayScalar(XNDa,1/(2*DeltaX));
end;



{*** Generierung normalverteilter Zufallszahlen ***}

...
diese Prozeduren sind für die Reconcilation nicht relevant und daher entfernt
...


{*** Unit-Main ***}
begin
  FillArray(EmptyXArray,0);
  randomize;
end.
```

```
unit DataClas;
{*************************************************************************}
{* Datei:          DataClas.PAS                                          *}
{* Typ:            Unit                                                  *}
{* Beschreibung:   enthält Klassen für die Ablage von Zeitreihen für Meß-    *}
{*                 und Schätzwerte                                       *}
{*                                                                       *}
{* Klassen:        TMeasurement, TMeasurementList                        *}
{*                                                                       *}
{* Autor:          Olaf Lummer, Leipzig                                  *}
{*                                                                       *}
{* Letzte Änderung: 12.06.97 von O.L.                                    *}
{*************************************************************************}

interface
uses Classes, DBTables, SysDef, TestDef, MathUtil;

Type TMeasurement=class(TObject)
     Protected    {Basisfunktionen für die Properties}
        function  FieldValueGet(Index:integer):TStdZahl;
        procedure FieldValuePut(Index: Integer; Item: TStdZahl);
        function  AsStringGet(Index:integer):string;
     Public
        t:TStdZahl;             {Meßzeitpunkt}
        TrueX     :TXArray;     {wahre Werte der Variablen}
        MeasuredX :TXArray;     {Meßwerte}
        CorrectedX:TXArray;     {Schätzwerte}
        TrueBias  :TXArray;     {wahrer Wert des Bias}
        CorrectedBias:TXArray;  {derzeitige Schätzung für den Bias}
        GuessedBias  :TXArray;  {Anfangsschätzung für den Bias}
        property FieldValue[Index: Integer]:TStdZahl read FieldValueGet write FieldValuePut;
                 {gibt über Index Zugriff auf alle relevanten Felder aller Variablen des Objekts}
        property AsString[Index: Integer]:string read AsStringGet;
                 {gibt zum Index eine Kurzbezeichnung des von FieldValue gelieferten Feldes}
     end;

type TMeasurementFillParams=record
       LBound  :Single;   {t(Start)}
       UBound  :Single;   {t(Ende)}
       Count   :Integer;  {Anzahl der Intervalle zwischen L und UBound}
       RKSteps :Integer;  {Anzahl der Runge-Kutta-Schritte in jedem diser Intervalle}
       StdError:Single;   {Standardabweichung der Meßwerte (in Prozent des Bezugswertes)}
       StartValues   :TXArray; {die Anfangswerte der Variablen und Parameter zur
                                Erzeugung der ungestörten Funktion}
       GuessedParams :TXArray; {die Schätzungen für die Parameter des Systems}
       TrueBias      :TXArray; {die wahren Beträge der Bias-Werte}
       GuessedBias   :TXArray; {Anfangsschätzungen für die Bias-Werte}
     end;

type TMeasurementList=class(TList) {Listenklasse über TMeasurement}
      Protected   {Basisfunktionen und -Variablen für die Properties}
        FFreeWhenDelete:Boolean;
        FLastFileName:string;
        FDirty:Boolean;
        function  ElementGet(Index:integer):TMeasurement;
        procedure ElementPut(Index: Integer; Item: TMeasurement);
      Public
        property  Element[Index: Integer]:TMeasurement read ElementGet write ElementPut; default;
                  {Zugriff auf die Listenelemente}
        property  FreeWhenDelete:Boolean read FFreeWhenDelete write FFreeWhenDelete;
                  {wenn ja, wird das von Delete gelöschte Element automatisch freigegeben}
        property  LastFileName:String read FLastFileName;
                  {Name der Datei vom letzten ReadTable oder WriteTable}
        property  Dirty:Boolean read FDirty write FDirty;
                  {sind daten seit dem letzten ReadTable oder WriteTable geändert ?}
        constructor Create;
        destructor  Destroy; override;

        procedure FillNew(Params:TMeasurementFillParams);    {neue Meßwerte erzeugen}
        procedure ReadTable(aPath:String);                   {aus Datenbank lesen}
        procedure WriteTable(aPath:String;aType:TTableType); {in Datenbank schreiben}
        procedure ResetData;                                 {Schätzwerte auf Meßwerte zurücksetzen}
        function  MSE:TStdZahl;         {Varianz der Schätzwerte}
        function  SDMeasured:TXArray;   {Standardabweichung der Meßwerte}
        function  SDCorrected:TXArray;  {Standardabweichung der Schätzwerte}
        procedure Delete(Index: Integer);        {überladene Standardfunktion}
        function  Remove(Item: Pointer):Integer; {überladene Standardfunktion}
        procedure Clear;                         {überladene Standardfunktion}
     end;
...
```
die übrigen Teile dieser Unit sind für das Verständnis der Reconciliation nicht notwendig und daher weggelassen

```
unit GlaetThr;

{***************************************************************************}
{*                                                                         *}
{* Datei:          GlaetThr.PAS                                            *}
{* Typ:            Unit                                                    *}
{* Beschreibung:   Thread-Klasse für den Reconciliation-Vorgang            *}
{*                                                                         *}
{* Klassen:        TTimeWindow, TUpdateList, TGlaettThread                 *}
{*                                                                         *}
{* Autor:          Olaf Lummer, Leipzig                                    *}
{*                                                                         *}
{* Letzte Änderung: 16.06.97 von O.L.                                      *}
{*                                                                         *}
{***************************************************************************}

interface

uses Classes, Forms, Windows, Math, SysUtils,
     SysDef, MathUtil, DataClas, LogForm;

Type TGlaettInputs=record       {Typ für die Übergabe von Einstellungen an TGlaettThread}
        WinWidth:Integer;       {Breite W des Zeitfensters}
        Steps:Integer;          {Maximalzahl Iterationen je Fensterschritt}
        FirstSteps:Integer;     {Maximalzahl Iterationen im 1. Fensterschritt}
        Epsilon:Single;         {Anpaßrate (=1, wenn automatisch adaptiert)}
        EpsilonDynamic:Boolean; {dynamische Anpaßrate verwenden?}
        K:TStdZahl;             {Parameter K des Algorithmus}
        RKSteps:Integer;        {Anzahl Zwischenschritte bei Runge-Kutta}
        ResetCorrected:Boolean; {beim Laden aus der Tabelle Corrected:=Measured setzen}
     end;
Type TTimeWindow = class(TMeasurementList) {Klasse zur Darstellung eines Zeitfensters
                                            aus chronologisch geordneten Meßwerttupeln}
         constructor Create(aMasterList:TMeasurementList;aWindowWidth:Integer;aResetCorrected:boolean);
      Protected
         MasterList:TMeasurementList;{Liste mit der gesamten Meßreihe}
         ReadRecordNumber:Longint;   {nächstes zu lesendes Element in der MasterList}
         WriteRecordNumber:Longint;  {nächstes zu schreibendes Element in der MasterList}
         WindowWidth:Integer;        {Breite W des Zeitfensters}
         ResetCorrected:boolean;     {beim Laden aus der MasterList Corrected:=Measured setzen? }
      Public
         Procedure AddSingleElement;
         procedure WriteSingleElement(Data:TMeasurement);
         procedure Fill;        {füllt das Zeitfenster für den ersten Iterationsschritt}
         procedure StepForward; {bewegt das Zeitfenster einen Datensatz nach vorn}
         procedure Flush;       {leert Zeitfenster, alle Daten werden geschrieben}
     end;

Type PSingle=^Single;

Type TUpdateList=class(TList)   {Listenklasse vom Typ PXArray
                                 mit Properties zum Zugriff auf einzelne Array-Elemente}
        constructor Create(aCount:integer);
        destructor Destroy; override;
     protected
        function  ElementGet(IndexR,IndexA:Integer):TStdZahl;
        procedure ElementPut(IndexR,IndexA:Integer; Item: TStdZahl);
        function  ArrayGet(Index:Integer):PXArray;
     public
        property  Element[IndexR,IndexA:Integer]:TStdZahl read ElementGet write ElementPut; default;
        property  ArrayPtr[Index:Integer]:PXArray read ArrayGet;
        procedure Reset;        {setzt alle Elemente auf Null}
     end;

Type
  TGlaettThread = class(TThread) {Thread-Klasse für die Reconciliation}
  public
    constructor Create(InputRec:TGlaettInputs;aBaseForm:TForm;aMasterList:TMeasurementList);
    destructor  Destroy; override;
  protected
    { Eingabe-Variablen }
    Inputs:TGlaettInputs;
    { Verbindung mit der Datenreihe}
    BaseForm:TForm;         {das Form, von dem aus der Thread gestartet wurde,
                             wird in abgeleiteter Klasse benötigt}
    RecCount:Integer;       {Anzahl der zu bearbeitenden Meßwerttupel}
    { Status-Variablen}
    MaxIter:Integer;
    CurrIter:Integer;
    HadException:boolean;   {gibt an, ob eine Exception aufgetreten war, wird
                             in abgeleiteter Klasse benötigt}
    { Variablen der Berechung}
    TimeWindow:TTimeWindow; {dieses Zeitfenster enthält die gerade betrachteten Meß- und Schätzwerte}
    EpsilonDyn:TList;       {enthält einen spezifischen Epsilon-Wert für jede Position des Zeitfensters}
    Updates:TUpdateList;    {Updates für alle Schätzwerte im Zeitfenster}
    BiasUpdates:TXArray;    {Updates für die Bias-Schätzung}
    EStart:TStdZahl;        {Wert der Energiefunktion vor dem Ausführen der Updates}

    procedure Execute; override;
    procedure ComputeUpdates;     {errechnet die Updates für Schätzwerte und Bias-Schätzungen}
    function  DoUpdate:Boolean;   {wendet die errechneten Updates 'intelligent' an}
    function  SimpleUpdate:Boolean;{wendet die errechneten Updates mit fester Skalierung an}
    procedure ColdStartInput;     {setzt Input-Schätzungen auf die Meßwerte zurück}
    function  E:TStdZahl;         {der aktuelle Wert der Energiefunktion}
```

```
    procedure InitEpsilonConstant;{initialisiert EpsilonDyn mit konstantem Epsilon für alle Punkte im
Zeitfenster}
    procedure InitEpsilonDynamic; {initialisiert EpsilonDyn mit den dynamischen Epsilon-Werten}
    procedure ClearEpsilon;       {löscht den Inhalt von EpsilonDyn}

    procedure SendStatus; virtual; abstract; {stellt Fortschritt im Mutterfenster dar}
  end;

implementation

{um statt der einfachen Approximationen die genaueren Formeln für die Bestimmung
 der partiellen Ableitungen der Vorwärtsschätzung zu verwenden, folgende Definition aktivieren}
{*** $DEFINE SlowDeriv}


{allgemeine Prozeduren}
procedure CopyUnmeasured(var dest:TMeasurement;const src:TMeasurement);
var i:integer;
begin
  for i:=0 to MaxVarIndex do if not SysVariables[i].Measured then dest.CorrectedX[i]:=src.CorrectedX[i];
  dest.CorrectedBias:=src.CorrectedBias;
end;


{ TUpdateList }
constructor TUpdateList.Create(aCount:integer);
var i:Integer;
    p:PXArray;
begin
  inherited Create;
  Capacity:=aCount;
  for i:=1 to aCount do begin
    new(p);
    p^:=EmptyXArray;
    Add(p);
  end;
end;

function  TUpdateList.ElementGet(IndexR,IndexA:Integer):TStdZahl;
begin
  Result:=PXArray(Get(IndexR))^[IndexA];
end;

procedure TUpdateList.ElementPut(IndexR,IndexA:Integer; Item: TStdZahl);
begin
  PXArray(Get(IndexR))^[IndexA]:=Item;
end;

function  TUpdateList.ArrayGet(Index:Integer):PXArray;
begin
  Result:=PXArray(Get(Index));
end;

procedure TUpdateList.Reset;
var i:integer;
begin
  for i:=Count-1 downto 0 do PXArray(Get(i))^:=EmptyXArray;
end;

destructor  TUpdateList.Destroy;
var i:integer;
    p:pXArray;
begin
  for i:=0 to Count-1 do
    try
      dispose(Get(i));
    except
      on Exception do {nothing};
  end;
  try
    Clear;
  finally
    inherited Destroy;
  end;
end;


{ TTimeWindow }

Constructor
TTimeWindow.Create(aMasterList:TMeasurementList;aWindowWidth:Integer;aResetCorrected:boolean);
begin
  inherited Create;
  FreeWhenDelete   :=False;
  MasterList       :=aMasterList;
  ReadRecordNumber :=0;
  WriteRecordNumber:=0;
  WindowWidth      :=aWindowWidth;
  Capacity         :=WindowWidth;
  ResetCorrected   :=aResetCorrected;
end;
```

```
procedure TTimeWindow.AddSingleElement;
var Data:Tmeasurement;
begin
  Data:=MasterList[ReadRecordNumber];
  inc(ReadRecordNumber);
  if ResetCorrected then Data.CorrectedX:=Data.MeasuredX;
  Add(Data);
end;

procedure TTimeWindow.WriteSingleElement(Data:TMeasurement);
begin
  MasterList.Dirty:=true;
  inc(WriteRecordNumber);
end;

procedure TTimeWindow.Fill;  {füllt das Zeitfenster für den ersten Iterationsschritt}
var i,a:integer;
    PrevRec,LastRec:TMeasurement;
begin
  Clear;
  AddSingleElement;   {erstes Element}
  PrevRec:=Last;

  for i:=1 to WindowWidth-1 do begin  {alle weiteren Elemente}
      AddSingleElement;
      LastRec:=Last;
      CopyUnmeasured(LastRec,PrevRec); {ungemessene Werte fortschreiben}
      PrevRec:=LastRec;
  end;
end;


procedure TTimeWindow.StepForward; {bewegt das Zeitfenster einen Datensatz nach vorn}
var PrevRec,LastRec,FirstRec:TMeasurement;
    a:integer;
begin
  FirstRec:=First;
  PrevRec :=Last;
  WriteSingleElement(Element[0]);   {ältesten Datensatz entfernen}
  Delete(0);
  AddSingleElement;
  LastRec:=Last;                    {ungemessene Werte fortschreiben}
  CopyUnmeasured(LastRec,PrevRec);
end;

procedure TTimeWindow.Flush;       {leert Zeitfenster, alle Daten werden geschrieben}
var i:Integer;
begin
  for i:=0 to WindowWidth-1 do WriteSingleElement(Element[i]);
  Clear;
end;

{ TGlaettThread }
constructor TGlaettThread.Create(InputRec:TGlaettInputs;aBaseForm:TForm;aMasterList:TMeasurementList);
begin
  inherited Create(false);
  TimeWindow:=TTimeWindow.Create(aMasterList,InputRec.WinWidth,InputRec.ResetCorrected);
  EpsilonDyn:=TList.Create;
  Updates   :=TUpdateList.Create(InputRec.WinWidth);
  Inputs    :=InputRec;
  BaseForm  :=aBaseForm;
  RecCount  :=aMasterList.Count;
  MaxIter   :=RecCount-Inputs.WinWidth+1;
  Priority  :=tpNormal;
  FreeOnTerminate:=True;
  HadException:=False;
end;

destructor TGlaettThread.Destroy;
begin
 try
   if Assigned(EpsilonDyn) then ClearEpsilon;
   EpsilonDyn.Free;
   TimeWindow.Free;
   Updates.Free;
 finally
   inherited Destroy;
 end;
end;

procedure TGlaettThread.InitEpsilonConstant;
var i:integer;
    p:PSingle;
begin
  ClearEpsilon;
  for i:=1 to Inputs.WinWidth do begin
    New(p);
    p^:=Inputs.Epsilon;
    EpsilonDyn.Add(p);
  end;
end;

procedure TGlaettThread.InitEpsilonDynamic;
var i:integer;
```

```
    p:PSingle;
    tq:Single;
begin
  ClearEpsilon;
  for i:=0 to Inputs.WinWidth-1 do begin
    tq:=sqr(i/(Inputs.WinWidth-1));
    New(p);
    {p^:=Inputs.Epsilon*(0.5-0.5*exp(-4*tq)+exp(-100*tq));}
    p^:=Inputs.Epsilon/(I+1);
    EpsilonDyn.Insert(0,p); {weil das ERSTE Listenelement den ÄLTESTEN Datensatz darstellt}
  end;
end;

procedure TGlaettThread.ClearEpsilon;
var i:integer;
begin
  for i:=0 to EpsilonDyn.Count-1 do dispose(EpsilonDyn[i]);
  EpsilonDyn.Clear;
end;

function TGlaettThread.E:TStdZahl;
var i:integer;
    Diff,BCX:TXArray;
begin
  try
    Result:=0;
    for i:=0 to Inputs.WinWidth-2 do begin
      with TimeWindow[i] do begin
        BCX:=CorrectedX;
        AddArrayArray(BCX,CorrectedBias);
        Diff:=RungeKutta(BCX,t,TimeWindow[i+1].t-t,Inputs.RKSteps);
      end;
      SubArrayArray(Diff,TimeWindow[i+1].CorrectedX);
      SubArrayArray(Diff,TimeWindow[i+1].CorrectedBias);
      Result:=Result+SqrSumArray(Diff);
    end;
  except
    on EMathError do Result:=1e1000;
  end;
end;


procedure TGlaettThread.ComputeUpdates;
{berechnet einen Updateschritt für alle Punkte im Zeitfenster}
var i,j:integer;         {Laufvariablen}
    tCurr,tNx:TStdZahl;  {t(i),t(i+1)}
    MX,CX:TXArray;       {Zwischenspeicher für gemessenes und CorrectedX}
    BX,BCX:TXArray;      {Zwischenspeicher für Bias und Bias+CorrectedX}
    DXNx:TXArray;        {Differenz von korrigiertem und geschätztem Wert für die nächste Messung}
    SchaetzVorw:TXArray; {Schätzung des aktuellen Punktes aus den Daten des vorherigen Punktes}
    RKDA:TXArray;        {partielle Ableitung der Schätzung des nächsten Punktes}
    ParamUpdates:TXArray;{gemitteltes Update für die Parameter}
    Dummy:TXArray;       {der Name sagt alles...}
    EinsDurchK:TStdZahl; {der Name sagt alles...}
    SumRKDA:TStdZahl;    {Summe der Elemente von RKDA}
    Epsilon:TStdZahl;    {Zwischenspeicher für Epsilon für aktuellen Punkte}
begin

  EinsDurchK:=1/Inputs.K;
  EStart:=0; {die Berechnung von EStart fällt hier nebenbei mit ab}

  for i:=0 to Inputs.WinWidth-1 do begin

    {häufig benötigte Werte zwischenspeichern}
    with TimeWindow[i] do begin
      tCurr:=t;
      MX:=MeasuredX;
      CX:=CorrectedX;
      BX:=CorrectedBias;
    end;
    BCX:=CX;
    AddArrayArray(BCX,BX);
    Epsilon:=PSingle(EpsilonDyn[i])^;

    {Daten bestimmen, die sich auf die Nachbarzeitpunkte beziehen}
    if i>0 then begin
       with TimeWindow[i-1] do begin
         Dummy:=CorrectedX;
         AddArrayArray(Dummy,CorrectedBias);
         SchaetzVorw:=RungeKutta(Dummy,t,tCurr-t,Inputs.RKSteps);
       end;

       Dummy:=BCX; {Bestimmung von EStart}
       SubArrayArray(Dummy,SchaetzVorw);
       EStart:=EStart+SqrSumArray(Dummy);
    end
    else SchaetzVorw:=BCX;

    if i<Inputs.WinWidth-1 then begin
       tNx:=TimeWindow[i+1].t;
       DXNx:=RungeKutta(BCX,tCurr,tNx-tCurr,Inputs.RKSteps);
       SubArrayArray(DXNx,TimeWindow[i+1].CorrectedX);
       SubArrayArray(DXNX,TimeWindow[i+1].CorrectedBias);
    end
```

```
    else begin    {in diesem Falle werden DXNx und RKDa später nicht berechnet}
      tNx:=tCurr;
      DXNx:=EmptyXArray;
      RKDA:=EmptyXArray;
    end;

    {Variablen des aktuellen Punktes korrigieren}
    for j:=0 to MaxVarIndex do begin

      if i<Inputs.WinWidth-1 then begin

       {$IFDEF SlowDeriv}
        RungeKuttaMulti(BCX,Dummy,RKDA,tCurr,tNx-tCurr,j,Inputs.RKSteps);
       {$ELSE}
        RKDaApprox(BCX,Dummy,RKDA,tCurr,tNx-tCurr,j,Inputs.RKSteps,SysVariables[j].SignalLevel*(1E-7));
       {$ENDIF}

        MulArrayArray(RKDA,DXNx);
        SumRKDA:=ElementSumArray(RKDA);
      end
      else SumRKDA:=(BCX[j]-SchaetzVorw[j]);

      case SysVariables[j].VarType of
        tvtState: Updates[i,j]:=Epsilon*(((SchaetzVorw[j]-BCX[j])-SumRKDA)+(MX[j]-CX[j])*EinsDurchK);
        tvtInput: Updates[i,j]:=Epsilon*(((SchaetzVorw[j]-BCX[j])-SumRKDA)+(MX[j]-CX[j])*EinsDurchK);
        tvtParameter: Updates[i,j]:=Inputs.Epsilon *(-SumRKDA);
      end;{case}
    end; {for j}
  end; {for i}

  {Parameter- und Biasupdates durch ihre Mittelwerte über das Fenster ersetzen}

  {Updates summieren}
  ParamUpdates:=EmptyXArray;
  for i:=0 to Inputs.WinWidth-2 do begin
    AddArrayArray(ParamUpdates,Updates.ArrayPtr[i]^);
  end;

  {Durchschnitt bilden}
  for j:=0 to MaxVarIndex do {if not Inputs.EpsilonDynamic then}
      if SysVariables[j].isParameter
      then ParamUpdates[j]:=ParamUpdates[j]/(Inputs.WinWidth-1) ;


  {Biasupdates übernehmen}
  for j:=0 to MaxVarIndex do
    if SysVariables[j].EstimateBias
    then BiasUpdates[j]:=(ParamUpdates[j]+Updates[Inputs.WinWidth-1,j])/(Inputs.WinWidth)
    else BiasUpdates[j]:=0;

  {Parameterupdates überall einsetzen}
  for i:=0 to Inputs.WinWidth-1 do
    for j:=0 to MaxVarIndex do
      if SysVariables[j].isParameter
      then Updates[i,j]:=ParamUpdates[j];


end; {ComputeUpdates}

function TGlaettThread.SimpleUpdate:Boolean;
var k:integer;
    Bias:TXArray;
begin
  Bias:=TimeWindow[0].CorrectedBias;
  AddArrayArray(Bias,BiasUpdates);
  for k:=0 to Inputs.WinWidth-1 do begin
    AddArrayArray(TimeWindow[k].CorrectedX,Updates.ArrayPtr[k]^);
    TimeWindow[k].CorrectedBias:=Bias;
  end;
  Result:=true;
end;

function TGlaettThread.DoUpdate:Boolean;
{ zum Verständnis dieser Prozedur siehe Diplomarbeit,
  Abschnitt "Einstellen der Anpaßrate" }
const EpsStart=0.8;
      EpsScale=0.5;
      EpsMin  =EpsStart/100;
var EpsOld,EpsCurrent:TStdZahl;
    EOld,ECurrent:TStdZahl;
    EThreshold:TStdZahl;   {unterschreitet E diese Schwelle, wird die Verbesserung der
                            Schätzwerte abgebrochen, weil nicht mehr lohnenswert}
    i:integer;
```

```
  procedure ApplyUpdates(Eps:Double);
  var k:integer;
      Up,Bias:TXArray;
  begin
    Bias:=BiasUpdates;
    MulArrayScalar(Bias,Eps);
    AddArrayArray(Bias,TimeWindow[0].CorrectedBias);
    for k:=0 to Inputs.WinWidth-1 do begin {Updates übernehmen}
       Up:=Updates.ArrayPtr[k]^;
       if Eps<>1 then MulArrayScalar(Up,Eps);
       AddArrayArray(TimeWindow[k].CorrectedX,Up);
       TimeWindow[k].CorrectedBias:=Bias;
    end;
  end;

begin {DoUpdate}
  if Inputs.Epsilon<1 then Result:=SimpleUpdate
  else begin
    EOld:=EStart;
    EpsOld:=0;
    EpsCurrent:=EpsStart;

    ApplyUpdates(EpsCurrent);
    ECurrent:=e;

    repeat
      EOld:=ECurrent;
      EpsOld:=EpsCurrent;
      EpsCurrent:=EpsCurrent*EpsScale;
      ApplyUpdates(EpsCurrent-EpsOld);
      ECurrent:=E;
    until ((ECurrent>EOld)) or (EpsCurrent<EpsMin);

    if (ECurrent>EStart) and (EOld>EStart) then begin
      {alles zurücknehmen}
      ApplyUpdates(-EpsCurrent);
      Result:=False;
    end
    else begin
      Result:=true;
      if ECurrent>EOld then begin
        {letzten Schritt zurück}
        ApplyUpdates(EpsOld-EpsCurrent);
      end; {else}
    end; {if}

    {wird E kleiner als (heuristischer) Schwellwert, lohnt keine weitere Verbesserung}
    EThreshold:=0;
    for i:=0 to MaxVarIndex do EThreshold:=EThreshold+sqr(SysVariables[i].SignalLevel*0.005);
    if min(ECurrent,EOld)<EThreshold*(Inputs.WinWidth-1) then Result:=false;
  end; {else}
end;  {DoUpdate}

procedure TGlaettThread.ColdStartInput;
var i,j:integer;
begin
  for i:=0 to TimeWindow.Count-1 do
    for j:=0 to MaxVarIndex do
      if SysVariables[j].IsInput
      then with TimeWindow[i] do CorrectedX[j]:=MeasuredX[j];
end;

procedure TGlaettThread.Execute;
var i,j:Integer;

    procedure Iterate(StepCount:Integer);
    begin
      repeat
        Inc(StepCount,-1);
        Updates.Reset;
        ComputeUpdates;
      until (not DoUpdate) or (StepCount=0);
    end;

begin
  {Zeitfenster aufbauen}
  CurrIter:=1;
  TimeWindow.Fill;

  try
    {Im Startfenster iterieren}
    Synchronize(SendStatus);
    InitEpsilonConstant;
    Iterate(Inputs.FirstSteps);
```

```
    {Zeitschritt}
    If Inputs.EpsilonDynamic then InitEpsilonDynamic;
    for i:=1 to RecCount-Inputs.WinWidth do begin
      if Terminated then Break;
      CurrIter:=i+1;
      Synchronize(SendStatus);

      {Zeitfenster einen Schritt nach vorn und dann iterieren}
      TimeWindow.StepForward;
      ColdStartInput;
      Iterate(Inputs.Steps);
    end; {for}

  except
    on Ex:EMathError do begin
       HadException:=true;
       Log('Math-Exception in TGlaetThread.Execute!  (TimeWin:'+ IntToStr(CurrIter) +')');
       Log('   '+Ex.Message);
       Log('  ');
    end;
  end;

  {TimeWindow leeren}
  TimeWindow.Flush;

  {Wenn ohne Abbruch abgearbeitet, Signal geben}
  if not Terminated
  then if HadException
       then MessageBeep(MB_ICONHAND)
       else MessageBeep(MB_OK);
end; {Execute}

end.
```

Inhalt der beiliegenden Diskette

Da nicht alle Quelltexte im Anhang dieser Arbeit erscheinen konnten und außerdem das im Rahmen dieser Diplomarbeit entstandene Testprogramm sicher für die weitergehende Beschäftigung mit dem vorgestellten Algorithmus interessant ist, habe ich mich entschlossen, eine Diskette beizulegen.

Auf der Diskette sind enthalten:

- die Quelltexte des Testprogramms
- eine kurze Bedienungsanleitung
- Diagramme und Tabellen für einige Beispiele
- eine fertig kompilierte Version des Testprogramms mit der CSTR-Simulation

Bitte beachten Sie die Voraussetzungen für die Lauffähigkeit:

- für die Übersetzung der Quelltexte ist Borland Delphi 2.0 (Developer) erforderlich
- für den Ablauf des Programms muß die Borland Database Engine (BDE) 3.0 (32bit) installiert sein
- das Programm ist unter Windows NT 3.51 bzw. Windows 95 oder höher lauffähig

Nicht auf der Diskette enthalten sind:

- Quelltexte oder Programmteile des Reconciliation-Systems für den ACAC-Reaktor
- die Borland Database Engine (Platzmangel)

Weitere Informationen finden Sie in der Datei **`readme.txt`** auf der Diskette.

Literaturliste

(Internet-Quellen: Stand Juli 1997)

Brammer, Karl; Siffling, Gerhard: Kalman-Bucy-Filter; München: Oldenbourg 1994

Braun, Martin: Differentialgleichungen und ihre Anwendungen; 3. Auflage, Berlin et al.: Springer 1994

Bronstein, Ilja Nikolaevich et al.: Taschenbuch der Mathematik; Leipzig: BSB Teubner, 1989

Der, Ralf: Ein dynamisches Prinzip zur Generierung der fehlerminimierenden Lösung der Differentialgleichung; Universität Leipzig 1996 (Arbeitsunterlage)

Dongarra, Jack: Linpack Benchmark; http://performance.netlib.org/performance/html/linpack.data.col0.html

Emons, Hans-Heinz (Hg.): Lehrbuch der technischen Chemie; Leipzig: Verlag für Grundstoffindustrie, 1988

Liebman, M. J.; Edgar, T. F.; Ladson L. S.: Efficient Data Reconciliation and Estimation for Dynamic Processes using Nonlinear Programming Techniques; in: Computers in Chemical Engineering, Vol. 16(1992), No. 10/11

McBrayer, Karen F.; Edgar, T. F.: The Application of Nonlinear Dynamic Data Reconciliation to Plant Data; http://www.emsl.pnl.gov:2080/docs/incinc/data_rect/KFMdoc.html

Möller, Dietmar: Modellbildung, Simulation und Identifikation dynamischer Systeme; Berlin et al.: Springer 1992

Müller, Roderich; Piotrowski, Anton: Einführung in die Elektrotechnik und Elektronik; München, Wien: Oldenbourg 1994

Müller-Erlwein, Erwin: Computeranwendungen in der chemischen Reaktionstechnik; Weinheim et al.: VCH, 1991

Naeve, Peter: Stochastik für Informatik; Wien: Oldenbourg 1995

Rudnyi, E. B.: Combined Processing of Experimental Series with Systematic Errors – non-linear Pysico-chemical Model with Linear Error Model; http://www.emsl.pnl.gov:2080/docs/incinc/data_rect/EBRdoc.html

Stahmer, Eckehart: FORTRAN; Berlin: Die Wirtschaft 1987

Tomiak, Andreas; Böttger, Christian: Dhrystone 1.1; http://island.isl.uni-bremen.de/~boettger/bench2/section3_1.html

Vauck, Wilhelm R.; Müller, Hermann A.: Grundoperationen chemischer Verfahrenstechnik; Leipzig: Verlag für Grundstoffindustrie, 1989

verschiedene interne Unterlagen der Wacker Chemie GmbH (Quellenangabe Geschäfts-geheimnis)

Selbständigkeitserklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegeben Quellen und Hilfsmittel angefertigt habe.

Leipzig, den 25. August 1997