Nächste–Nachbar basierte Methoden in der nichtlinearen Zeitreihenanalyse

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von
Christian Merkwirth
aus
Kassel

Göttingen 2000

D7
Referent: PD Dr. Ulrich Parlitz
Korreferent: Prof. Dr. Dirk Ronneberger
Tag der mündlichen Prüfung: 2. Nov. 2000

Zusammenfassung

Das Thema dieser Arbeit ist die Anwendung der Nächste–Nachbar–Suche in Verfahren der nichtlinearen Zeitreihenanalyse (NLZ).

Unter dem Begriff der Nächsten–Nachbarn–Suche in einem metrischen Raum versteht man die Bestimmung derjenigen Punkte eines Datensatzes, die zu einem gegebenen Anfragepunkt, der oft dem gleichen Datensatz entnommen ist, einen minimalen Abstand aufweisen. Da die meisten Abstandsmaße (Metriken) die *Ähnlichkeit* zweier Punkte in Hinblick auf gewisse Kriterien bewerten, spricht man bei der Nächsten–Nachbar–Suche auch von der Suche nach ähnlichen Punkten bzw. im Rahmen der NLZ auch von ähnlichen Zuständen.

Ein Haupteinsatzgebiet der Nächsten–Nachbar–Suche in der NLZ ist die Modellierung und Vorhersage nichtlinearer dynamischer Systeme. Dazu werden meist skalare Zeitreihen dieser Systeme durch die Technik der Zeitverzögerungsrekonstruktion in einen mehrdimensionalen Zustandsraum eingebettet. Diese Verfahren werden im zweiten Teil der Arbeit zusammen mit einer Methode zur Validierung der so gewonnenen Modelle vorgestellt, ohne daß dort darauf eingegangen wird, wie die verwendeten Nächsten–Nachbarn effizient numerisch bestimmt werden können.

Als ein zentrales Ergebnis dieser Arbeit wird daher im dritten Teil ein effizienter Algorithmus zur Nächsten–Nachbar–Suche, der sogenannte ATRIA (Advanced Triangle Inequality Algorithm), vorgestellt. Dieser zeichnet sich sowohl durch flexible Wahl der zur Distanzberechnung verwendeten Metrik als auch durch die Möglichkeit aus, die Laufzeit des Algorithmus durch eine abgeschwächte Variante der Suche, bei der sogenannte approximative Nächste–Nachbarn bestimmt werden, weiter zu verringern.

Im vierten Teil der Arbeit werden verschiedene Methoden der Bestimmung des Spektrums der fraktalen Dimensionen eines dynamischer Systems vorgestellt und verglichen. Abgeschlossen wird das Kapitel von numerischen Untersuchungen zur Ermittlung des Zusammenhangs zwischen Laufzeit des ATRIA und der fraktalen Dimension des Datensatzes, in dem die Nachbarsuche stattfindet.

Im fünften Teil der Arbeit wird ein Verfahren zur lokalen Modellierung raum–zeitlicher dynamischer Systeme vorgestellt. Dieses Verfahren, bei dem Nächste–Nachbarn in einem vergleichsweise hochdimensionalen Raum bestimmt werden müssen, profitiert deutlich von der Verwendung eines Algorithmus, dessen Laufzeit wie die des im dritten Kapitel eingeführten ATRIA nur unkritisch von der formalen Dimension des Datensatzes abhängt.

Der sechste Teil der Arbeit geht kurz auf das Programmpaket TSTOOL ein, das verschiedene Methoden der nichtlinearen Zeitreihenanalyse unter einer einheitlichen Bedienoberfläche vereint. Alle numerischen Experimente dieser Arbeit wurden mit Hilfe dieses Programmpaketes durchgeführt.

Inhaltsverzeichnis

1. Einleitung **1**

2. Modellierung dynamischer Systeme **4**

2.1. Grundbegriffe 4

2.1.1. Dynamische Systeme 4

2.1.2. Rekonstruktion des Attraktors 6

2.2. Modellierung nichtlinearer deterministischer Systeme 8

2.2.1. Lokale Modellierung im Rekonstruktionsraum 9

2.2.2. Iterierte Vorhersage 10

2.2.3. Funktionsapproximation mit lokal konstantem Modell 11

2.3. Modellvalidierung 13

2.3.1. Cross–Validation 14

2.3.2. Leave–One–Out Cross–Validation für lokale Modelle 15

2.3.3. Erweiterungen der Leave–One–Out Cross–Validation 16

2.4. Beispiele lokaler Modellierung zur Vorhersage skalarer Zeitreihen niedrigdimensionaler dynamischer Systeme 17

2.4.1. Untersuchte Systeme 17

2.4.2. Verwendete Metrik 19

2.4.3. Verwendetes Fehlermaß 20
2.4.4. Parametervorgaben 20
2.4.5. Vorhersagen 21
2.5. Zusammenfassung 22

3. Effiziente Nächste-Nachbar Suche **24**

3.1. Einführung 24
3.1.1. Begriffsbildungen 24
3.1.1.1. Metrischer Raum 24
3.1.1.2. Nächste–Nachbarn 25
3.1.1.3. Approximative Nächste–Nachbarn 25
3.2. ATRIA - Ein auf der Dreiecksungleichung basierender Algorithmus 26
3.3. Vorverarbeitung 27
3.3.1. Hierarchische Überdeckung 27
3.3.2. Zuordungsstrategie 28
3.3.3. Implementation der Datenstrukturen und Verwaltung der Punkte . . . 29
3.3.3.1. Verwaltung der Punkte des Datensatzes 29
3.3.3.2. Repräsentation eines Clusters 31
3.4. Suchphase 32
3.4.1. Eingrenzung des Suchraums 32
3.4.2. Berechnung von $\hat{d}_{min}$ 34
3.4.3. Prioritätswarteschlangen kontrollierte Suchreihenfolge 35
3.4.4. Bearbeitung einer Suchanfrage 35
3.4.4.1. Exakte Suchen 36
3.4.4.2. Approximative Suchen 37

3.4.4.3. Bereichssuche 38

3.5. Verringerung der Rechenzeit bei approximativen Suchen mit verschiedenem ϵ 39

3.6. Anwendung von Kernfunktionen zur Berechnung von Distanzen in hochdimensionalen Merkmalsräumen 40

3.7. Zusammenfassung 41

4. Dimensionen **44**

4.1. Begriffsbildungen 44

4.1.1. Verallgemeinerte Dimensionen 44

4.1.2. Das invariante Maß 44

4.1.3. Die Hausdorffdimension 45

4.1.4. Die Kapazitätsdimension 46

4.1.5. Die Rényi–Dimensionen 46

4.2. Numerische Berechnung der verallgemeinerten Dimensionen 48

4.2.1. Box–Counting basierte Ansätze 48

4.2.2. Korrelationssummen basierte Ansätze 51

4.2.3. Nächste–Nachbar basierte Ansätze 52

4.2.4. Numerische Bestimmung des Dimensionsspektrums dynamischer Systeme aus deren Zeitreihen 55

4.3. Einfluß des Dimensionsspektrums auf die Effizienz der Nächsten–Nachbar Suche 57

4.3.1. Einfluß der fraktalen Dimension auf die Selektivität der Nachbarsuche 59

4.4. Zusammenfassung 62

5. Lokale Modellierung raum–zeitlicher dynamischer Systeme **64**

5.1. Motivation 64

5.2. Vorarbeiten . . . 65
5.3. Rekonstruktion lokaler Zustände . . . 66
5.3.1. Randbedingungen . . . 66
5.4. Lokale Modellierung der Dynamik lokaler Zustände . . . 68
5.4.1. Iterierte Vorhersage mit lokalen Zuständen . . . 69
5.4.2. Validierung . . . 70
5.5. Beispiele lokaler Modellierung zur Vorhersage raum–zeitlicher Zeitreihen . . . 71
5.5.1. Untersuchte Systeme . . . 71
5.5.2. Parametervorgaben . . . 71
5.5.3. Vorhersagen . . . 72

6. Das Programmpaket TSTOOL **76**

7. Ausblick **80**

A. Anhang **82**

A.1. Beweise . . . 82
A.1.1. Ausschlußregel . . . 82
A.1.2. Abschätzung von $\hat{d}_{min}$ . . . 83
A.1.3. Ausschlußregel für Punkte innerhalb eines endständigen Knotens . . . 84
A.1.4. Maximale Interpunktdistanz in einer endlichen Punktmenge . . . 85
A.2. Die Gammafunktion . . . 85
A.3. Die Poissonverteilung . . . 85
A.4. Quelltexte . . . 87
A.4.1. Beispielimplementation des ATRIA . . . 87
A.4.2. Implementation einer k–Nächsten–Nachbar–Suche in L_2–Metrik . . . 97

A.4.3. Implementation einer k–Nächsten–Nachbar Suche in L_∞–Metrik, wobei die Anfragepunkte aus dem Datensatz selbst stammen 98

A.4.4. Beispielimplementation der schnellen Korrelationssummenberechnung in L_∞–Metrik 99

A.4.5. Beispielimplementation eines ternären Suchbaumes für Box–Counting Verfahren 101

Literaturverzeichnis **105**

1. Einleitung

Technische Zeitreihenanalyse

Häufig liefern Messungen bei wissenschaftlichen Experimenten oder bei der Datenerfassung an technischen Systemen irregulär oszillierende Zeitreihen, die analysiert oder in ihrem weiteren Verlauf vorhergesagt werden sollen. Neben stochastischen Einflüssen sind Nichtlinearitäten eine mögliche Ursache für aperiodisches Verhalten und können auch bei Systemen mit wenigen Freiheitsgraden zu großer Empfindlichkeit gegenüber kleinen Störungen führen. Die Methoden der linearen Zeitreihenanalyse, obwohl sehr ausgereift und vielseitig einsetzbar, bieten oft keine adäquate Beschreibung solcher Systeme. Aus diesem Grund wurden speziell für Zeitreihen, denen eine derartige deterministische chaotische Dynamik zugrunde liegt, in den letzten Jahren zahlreiche Analysemethoden entwickelt, die in entsprechend modifizierter Form auch zur Auswertung technisch-industrieller Daten geeignet sind.

Dort hat sich in vielen Fällen eine Kombination linearer und nichtlinearer Konzepte als vorteilhaft erwiesen. Die linearen Methoden übernehmen in diesem Zusammenhang häufig die Rolle einer Vorverarbeitung (z.B. Filterung), während die nachgeschalteten nichtlinearen Auswertungsverfahren die eigentliche Merkmalsextraktion liefern. Dabei handelt es sich bei irregulären technischen Daten, die z.B. an (defekten) Maschinen aufgenommen werden, in der Regel nicht um chaotische Signale im Sinne der nichtlinearen Dynamik. Trotzdem können die zunächst für diese Systeme entwickelten Methoden hier angewandt werden, wenn die Interpretation der Ergebnisse den neuen Voraussetzungen angepaßt wird.

So war es möglich, mit dem im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Projektes „Analyse von Maschinen– und Prozeßzuständen mit Methoden der Nichtlinearen Dynamik“ (Fördernummer 13N7038/9) erstellten Programmpaket namens *TSTOOL* (vgl. Kapitel 6) zur nichtlinearen Zeitreihenanalyse, das verschiedene

Methoden des Arbeitsgebietes unter einer einheitlichen Bedienoberfläche zur Verfügung stellen und somit die Erprobung und den Einsatz dieser neuen Methoden wirksam unterstützen soll, ein Problem der Qualitätssicherung bei der Produktion von Lüftermotoren zu lösen, bei dem anhand des Laufgeräusches eines Lüftermotors automatisch eine Klassifizierung in defekt bzw. nicht defekt vorzunehmen war.

Bei der Konzeption des TSTOOL Programmpaketes wurde die bestehende Literatur kritisch durchsucht, um eine Auswahl erprobter, relevanter oder praxisnaher Methoden zu finden, die dann implementiert wurden. Schnell zeigte sich dabei, daß ein Großteil der bislang vorgeschlagenen Methoden lokal in einem z.B. durch Zeitverzögerungskoordinaten erzeugten Zustandsraum operieren, weshalb in Kapitel 2 eine Darstellung der Techniken zur lokalen Modellierung der Dynamik nichtlinearer Systeme im Zustandsraum anhand skalarer Zeitreihen gegeben wird.

Effiziente Algorithmen

Obwohl sich lokal im Zustandsraum operierende Methoden meist durch eine vergleichsweise einfache mathematische Formulierung auszeichnen, kann die notwendige Suche der Nächsten–Nachbarn in großen Datensätzen bei naiver Implementation einen enormen numerischen Aufwand darstellen. Die Bereitstellung eines effizienten Algorithmus zur Nächste–Nachbarn–Suche erwies sich als essentiell für die Praxistauglichkeit und Verwendbarkeit des oben vorgestellten Programmpaketes. Daher wurde ein bereits zuvor zur Vorhersage skalarer Zeitreihen verwendeter Algorithmus implementiert und weiterentwickelt. Dieser Algorithmus, der sich in seiner jetzigen Version sowohl durch gute Effizienz als auch durch die Fähigkeit, mit beliebigen Metriken zu arbeiten, auszeichnet, wird im Kapitel 3 vorgestellt. Im Anhang wird zusätzlich eine Beispielimplementation des Algorithmus in der Programmiersprache C++ gegeben, die durch Verwendung moderner hochsprachlicher Abstraktionsmöglichkeiten (u.a. Templates) hohe Portabilität und leichte Adaption für spezifische Anwendungsfälle ermöglicht.

Aber auch bei Methoden, die nicht direkt auf der Berechnung der Nächsten–Nachbarn beruhen, wurde besonderes Augenmerk auf die Verwendung *schneller* bzw. *effizienter* Algorithmen gelegt. So wird z.B. in Kapitel 4 die Datenstruktur des *ternären Baumes* zur Implementation von *Box–Counting* Methoden vorgestellt, deren Speicheraufwand im schlechtesten Fall lediglich linear mit der Anzahl der Punkte eines Datensatzes wächst und die dennoch ein

schnelles Auffinden einzelner Partitionen ermöglicht. Im gleichen Kapitel werden auch andere Methoden zur Berechnung des Spektrums der fraktalen Dimensionen vorgestellt. Dabei wird der enge Zusammenhang von fraktaler Dimension und der Effizienz des Algorithmus zur Nächsten–Nachbar–Suche aufgezeigt.

Raum–zeitliche Systeme

Die Modellierung und Vorhersage von Zeitreihen raum–zeitlicher Systeme bringt, speziell bei globalen Ansätzen, einen oft prohibitiv hohen numerischen Aufwand mit sich. Daher wird in Kapitel 5 die Methode der Rekonstruktion lokaler Zustände für raum–zeitliche Zeitreihen vorgestellt. Diese erschließt die in Kapitel 2 eingeführten lokalen Modellierungsverfahren für die Analyse und Vorhersage der Dynamik raum–zeitlicher Systeme. In Verbindung mit der Fähigkeit des vorgestellten Algorithmus zur Nächsten–Nachbar–Suche, unter bestimmten Voraussetzungen auch in formal hochdimensionalen Datenräumen schnell arbeiten zu können, erlaubt dies die effiziente numerische Behandlung der Aufgabenstellung.

2. Modellierung dynamischer Systeme

2.1. Grundbegriffe

2.1.1. Dynamische Systeme

Gemeinhin wird unter einem dynamischen System ein deterministisches System unter zeitlicher Entwicklung verstanden. Im weiteren Verlauf dieses Kapitels stehen Systeme im Vordergrund, deren Zustand eindeutig und vollständig durch einen Punkt $\mathbf{y} \in \mathbb{R}^d$ in einem endlichdimensionalen reellen Vektorraum, dem sog. Zustandsraum, dargestellt werden kann. Im kontinuierlichen Fall wird die Zeitentwicklung eines solchen dynamischen Systems meist durch einen Satz gewöhnlicher Differentialgleichungen beschrieben:

$$\dot{\mathbf{y}} = F(\mathbf{y}), \tag{2.1}$$

dabei soll in einem Gebiet $G \subset \mathbb{R}^d$ gelten

$$F \in C^1(G),$$

wodurch für alle $\mathbf{y} \in G$ die Existenz und Eindeutigkeit einer Lösung von Gleichung 2.1 gesichert ist [22].

Die zeitliche Entwicklung eines Punktes $\mathbf{y}_0(t)$ nennt man die *Trajektorie* oder Bahnkurve von $\mathbf{y}_0$. Die Gesamtheit aller Trajektorien nennt man das Phasenbild des Systems.

Im zeitdiskreten Fall wird ein dynamisches System durch Differenzengleichungen beschrieben:

$$\mathbf{y}_n = f(\mathbf{y}_{n-1}), \tag{2.2}$$

wobei $f \in C^1(G)$ gelten soll.

Beide Fälle definieren einen *Fluß* im Phasenraum, der im allgemeinen durch eine stetig differenzierbare Abbildung gegeben wird:

$$\begin{aligned} \varphi : \mathbb{R}^d \times \mathbb{K} &\rightarrow \mathbb{R}^d \\ (\mathbf{y}, t) &\mapsto \varphi(\mathbf{y}, t) \end{aligned} \tag{2.3}$$

mit den Eigenschaften

$$\begin{aligned} \varphi(\mathbf{y}, t=0) &= \mathbf{y} \\ \varphi(\varphi(\mathbf{y}, t), u) &= \varphi(\mathbf{y}, u+t) \qquad \forall t, u \in \mathbb{K} \;\; \forall \mathbf{y} \in \mathbb{R}^d. \end{aligned} \tag{2.4}$$

Für kontinuierliche dynamische Systeme ist $\mathbb{K} = \mathbb{R}$, für diskrete Systeme gilt $\mathbb{K} = \mathbb{Z}$, wobei der Fluß durch die Abbildung 2.2 selbst gegeben ist.

Den Zusammenhang zwischen Flüssen und Differentialgleichungen erhält man, indem man $\varphi(t, \mathbf{y}_0)$ als Lösung von Gleichung (2.1) interpretiert, die zur Zeit $t = 0$ durch den Punkt $\mathbf{y}_0$ geht. Auf diese Weise korrespondiert jeder Fluß mit einer Differentialgleichung, allerdings gilt die die Umkehrung im allgemeinen nicht [4].

Ein System nennt man *dissipativ*, falls sich das d-dimensionale Volumen eines gegebenen Phasenraumgebietes unter Einwirkung des Flusses im zeitlichen Mittel kontrahiert. Als Folgerung aus dem Liouvilleschen Satz der klassischen Mechanik ergibt sich somit

$$\mathrm{div} F(\mathbf{y}) \;\; < \;\; 0$$

als Bedingung für Dissipation [1, 2].

Von besonderem Interesse sind dabei Gebiete des Phasenraumes, die unter der Wirkung des Flusses invariant bleiben. Auf ihnen spielt sich das Langzeitverhalten eines dissipativen Systems ab, nachdem die Transienten abgeklungen sind [25].

Im folgenden seien U und V zwei offene Mengen des Phasenraums. Eine Menge $A \subset \mathbb{R}^d$ nennt man *Attraktor*[1] mit der *fundamentalen Nachbarschaft* U, wenn folgende Bedingungen erfüllt sind:

[1] Für diesen fundamentalen Begriff gibt es leider keine allgemein anerkannte Definition, in diesem Fall ist die von D. Ruelle vorgeschlagene Version verwendet worden [37].

- A ist eine kompakte Menge

- A ist invariant bzgl. des Flusses

$$\forall t \in \mathbb{R} \ : \ \varphi(A,t) = A$$

- A ist eine *anziehende Menge*

$$\forall V \supset A \ \ \exists t \in \mathbb{R} \ : \ \varphi(U,t) \subset V$$

- Die Menge A läßt sich nicht in zwei nichtleere invariante Mengen zerlegen

Das *Bassin* des Attraktors A wird durch alle Punkte des Phasenraumes definiert, für die A eine anziehende Menge ist [37].

2.1.2. Rekonstruktion des Attraktors

Führt man an einem dynamischen System eine Messung durch, so mißt man normalerweise eine einzige Systemgröße durch zeitdiskrete Abtastung (im Englischen *sampling*) mit einer konstanten Abtastzeit $\triangle T$. Die erhaltene Liste der Meßwerte $\{s^t\} \quad t \in \mathbb{Z}$ nennt man eine *Zeitreihe* des Systems. Die Einbettungstheoreme von Takens [44] und von Sauer et al. [38] zeigen eine Möglichkeit auf, den Attraktor eines Systems aus einer gemessenen Zeitreihe zu rekonstruieren. Beide Theoreme sind Verallgemeinerungen des Whitneyschen Einbettungssatzes der Differentialtopologie, der besagt, daß sich jede d-dimensionale differenzierbare Mannigfaltigkeit abgeschlossen in den $\mathbb{R}^{2d+1}$ einbetten läßt [10].

Sei φ der Fluß eines dynamischen Systems auf der Mannigfaltigkeit M und $\tau \in \mathbb{R}$. Die Messung einer Systemgröße soll durch eine stetig differenzierbare Funktion $h : M \rightarrow \mathbb{R}$ beschrieben werden:

$$s^t \ = \ h\left(\varphi(\mathbf{y}, t\triangle T)\right)$$

Die Idee bei der Einführung sog. Delay-Koordinaten (auch Zeitverzögerungskoordinaten genannt) ist die Konstruktion eines Vektors aus jeweils um τ zeitverschobenen Werten der Zeitreihe. In der Praxis muß dazu natürlich τ ein ganzzahliges Vielfaches L der Abtastzeit

$\triangle T$ sein, prinzipiell kann die Abtastzeit jedoch beliebig klein gewählt werden, um eine gute Feineinstellung der Delay–Zeit zu ermöglichen. Die D-dimensionale Delay–Koordinaten–Abbildung $Z(h, \varphi, \tau) : M \to \mathbb{R}^D$ mit der Delay–Zeit τ wird dann definiert durch

$$\begin{aligned} Z(h,\varphi,\tau)(\mathbf{y}) &= (h(\mathbf{y}), h(\varphi(\mathbf{y},-\tau)), \ldots, h(\varphi(\mathbf{y},-(D-1)\tau))) \\ \mathbf{x} &= (s^0, s^{-L}, \ldots, s^{-(D-1)L}) \end{aligned}$$

Die Delay–Koordinaten–Abbildung Z wird nacheinander auf alle Abtastwerte s^t angewandt, wodurch man einen Satz von Zeitverzögerungsvektoren erhält:

$$\mathbf{x}^t = (s^t, s^{t-L}, \ldots, s^{t-(D-1)L}) \quad t \in \mathbb{Z}$$

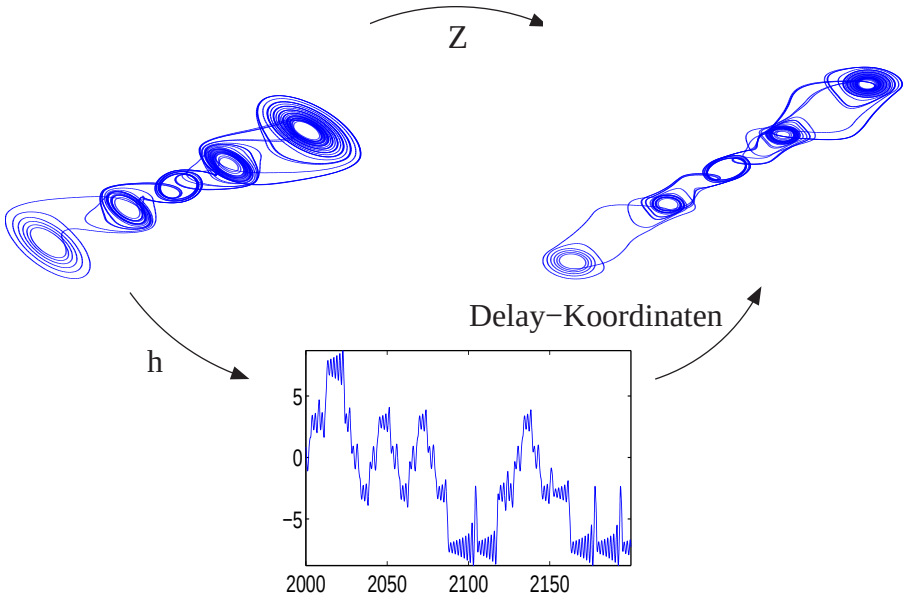


Abbildung 2.1.: Schema der Delay-Einbettung

Die Verwendung der Systemvergangenheit zur Beschreibung des aktuellen Zustandes geschieht in Anlehnung an das *Windowing* der linearen Zeitreihenanalyse, wo sich dieses Verfahren seit langem bewährt hat [11].

Sauer et al. [38] konnten die Ergebnisse von Takens verallgemeinern. Sie zeigten, daß auch unter weniger starken Voraussetzungen als beim Einbettungstheorem von Takens die Delay-Rekonstruktion des Attraktors diffeomorph zum Original-Attraktor im Phasenraum ist.

Für eine offene Menge $U \subset M$ und eine kompakte (φ-invariante) Menge $A \subset U$ mit der Kapazitätsdimension $\dim_K(A) = d$, in der $\varphi(\mathbf{y}, t)$ keine periodischen Orbits der Periode τ und 2τ, nur endlich viele Fixpunkte und periodische Orbits der Periode $3\tau, 4\tau, \ldots, n\tau$ besitzt und ferner die Linearisierung des Flusses entlang der periodischen Orbits verschiedene Eigenwerte hat, ist für fast jede stetig differenzierbare Funktion $h : U \to \mathbb{R}$ die Delay-Koordinaten-Abbildung $Z(h, \varphi, \tau) : U \to \mathbb{R}^D$ eine eindeutig umkehrbare Abbildung und für jede kompakte Teilmenge einer in A enthaltenen glatten Mannigfaltigkeit auch eine Einbettung, falls $D > 2d$ gilt.

Die Möglichkeit, ein diffeomorphes Abbild der Dynamik im Phasenraum durch Messung einer (oder weniger) Systemgrößen zu erhalten, erlaubt insbesondere die Bestimmung wichtiger C^1-Invarianten des Systems (z.B. Rényi–Dimensionen, Lyapunovexponenten, Entropien). Darüberhinaus bietet dieses Verfahren auch die Möglichkeit der Visualisierung verschiedener Attraktoren aus gemessenen Zeitreihen.

2.2. Modellierung nichtlinearer deterministischer Systeme

Oftmals werden Systeme beobachtet, für die noch keine konsistente Theorie vorliegt, die Aufschluß über die inneren Zusammenhänge der beobachtbaren Größen liefern könnte, oder die so komplex sind, daß sie keine Reduktion auf „Laborbedingungen“ erlauben (ökonomische Daten oder medizinische Meßreihen, wie z.B. EKG).

Seit der erfolgreichen Einführung von nichtlinearen deterministischen Modellen (Farmer und Sidorovich (1987) [18], Crutchfield und McNamara (1987) [15] und Casdagli (1989) [13]) vermutete man in vielen Systemen mit irregulärem Zeitverhalten „deterministisches Chaos“ auf niedrigdimensionalen Attraktoren, wobei sich die Hoffnung, einen strengen Determinismus zu finden, oft als trügerisch erwies.

In vielen Fällen erhebt sich die Frage, ob eine gemessene Zeitreihe überhaupt von einem deterministischen System stammt, oder ob dem zugrundeliegenden Determinismus noch eine stochastische Komponente überlagert ist (z.B. Meßfehler oder Rauschen). Verschiedene statistische Verfahren wurden daher vorgeschlagen, um nichtlineare deterministische Systeme von stochastischen Systemen zu trennen. Bei der von Theiler et al. vorgeschlagenen Methode der

Surrogatdaten [46], die in das Programmpaket TSTOOL integriert wurde, formuliert man eine Hypothese über den möglichen Ursprung des beobachteten Prozesses (z.B.: „Es handelt sich um weißes Rauschen“) und generiert nun mehrere, im Hinblick auf die Hypothese gleichwertige Datensätze, die sogenannten Surrogatdaten (in diesem Beispiel also Zeitreihen mit dem Mittelwert und der Varianz der gemessenen Daten). Dann berechnet man für die verschiedenen Datensätze einige charakteristische Größen, z.B. Lyapunovexponenten, Korrelationsdimension, Vorhersagefehler eines lokal linearen Modells usw., und vergleicht die jeweiligen Werte der Surrogatdaten mit denen der Originaldaten. Treten dabei signifikante Abweichungen zwischen den Surrogatdaten und den Originaldaten auf, so kann man die Hypothese auf einem vorgegebenen Signifikanzniveau verwerfen.

2.2.1. Lokale Modellierung im Rekonstruktionsraum

Von einem *lokalen Modell* spricht man, wenn nur Zustände in einem begrenzten Gebiet des Rekonstruktionsraums zur Approximation des Flusses ψ herangezogen werden. Die ersten lokalen Modelle in Zusammenhang mit der nichtlinearen Zeitreihenanalyse wurden 1987 von Farmer und Sidorovich [18] vorgeschlagen, bei dem Versuch die Zeitreihen niedrigdimensionaler chaotischer Systeme (Rayleigh-Bénard-Konvektion und Taylor-Couette-Strömung) vorherzusagen.

Voraussetzung für diese Art der Modellierung ist die Stetigkeit des Flusses, aus der folgt, daß sich die Entwicklung eines Systemzustandes für einen kleinen Zeitraum nicht wesentlich von der Entwicklung eng benachbarter Zustände im Phasenraum (und somit auch im Rekonstruktionsraum) unterscheidet. Man versucht deshalb eine Abbildung zu finden, die den Fluß in der Nachbarschaft eines Zustandes $\mathbf{x}$ approximiert[2], um mit Hilfe dieser Abbildung die zukünftigen Systemzustände $\tilde{\mathbf{x}}_{n+i}, \quad i = 1, 2, \ldots$ vorauszusagen.

Man unterscheidet dabei zwischen der *direkten* Vorhersage, bei der, ausgehend von einem bekannten Systemzustand $\mathbf{x}$, der Fluß $\psi(\mathbf{x}, i\triangle T)$ für $i = 1, 2, \ldots$ approximiert wird, und der *iterierten* Ein–Schritt–Vorhersage, bei der lediglich $\psi(\mathbf{x}, \triangle T)$ modelliert wird und die Prädiktion dann iterativ ausgeführt wird (siehe Abschnitt 2.2.2). In beiden Fällen werden dazu die k Nächsten–Nachbarn $\{\mathbf{x}^{nn_j}\}_{j=1,\ldots,k}$ des Zustandes $\mathbf{x}$ bezüglich einer geeigneten Metrik bestimmt und deren (bekannte) Bilder $\{\mathbf{x}^{nn_j+i}\}_{j=1,\ldots,k}$, die aus diesen unter Wirkung des Flus-

[2] Diese Abbildung wird bei M. Casdagli *„local predictor“* genannt, und die Approximation des Flusses setzt sich dann aus diesen lokalen Funktionen zusammen [13].

ses $\psi(\mathbf{x}^{nn_j}, i\triangle T)$ hervorgegangen sind, zur Berechnung der (unbekannten) Zeitentwicklung von $\mathbf{x}$ herangezogen. Die Approximation des gesamten Flusses setzt sich somit jeweils aus vielen einzelnen lokalen Approximationen zusammen. Dies macht es unmöglich, die Approximationen in einer „geschlossenen Form“ anzugeben.

In der Literatur wurden die Vor- und Nachteil beider Verfahren kontrovers diskutiert. Ein Problem der direkten Vorhersage ist, daß der zu approximierende Fluß mit zunehmender Schrittweite gerade bei chaotischen Systemen immer komplizierter wird. Auch muß für jede Schrittweite prinzipiell ein neues Modell konstruiert bzw. optimiert werden. Andererseits akkumuliert sich bei der iterativen Methode der Fehler der einzelnen Ein–Schritt–Vorhersagen, ohne daß dies im Verfahren explizit berücksichtigt wird. Darüberhinaus muß bei der iterativen Vorhersage bei jedem Iterationsschritt ein neuer Satz Nächster–Nachbarn bestimmt werden, was bei langen Zeitreihen und naiver Implementierung der Nachbarsuche zu einem hohen Rechenaufwand führt. In Verbindung mit dem im Kapitel 3 vorgestellten Algorithmus zur effizienten Nächsten–Nachbar Suche stellt die lokale Modellierung mit iterierter Ein–Schritt–Vorhersage jedoch ein mächtiges Werkzeug zur Prädiktion von Zeitreihen nichtlinearer deterministischer Systeme dar (vgl.[43] sowie [28]).

Dabei ist zu beobachten, daß bei Verwendung längerer Zeitreihen eine bessere Approximation des Flusses erreicht wird, da sich bei Hinzunahme neuer Datenpunkte im Mittel die Abstände der Punkte im Rekonstruktionsraum verringert, was einen kleineren Fehler in der Vorhersage des nächsten Systemzustandes zur Folge hat [18]. Allerdings bedeutet dies jedoch umgekehrt, daß lokale Modelle bei sehr kurzen Zeitreihen ihre Vorhersagefähigkeit einbüßen können[3].

2.2.2. Iterierte Vorhersage

Die Problemstellung der Approximation des Flusses $\psi(\mathbf{x}, \triangle T)$ im Rekonstruktionsraum läßt sich auf das Problem der Approximation der skalaren Funktion $f(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathbb{R}, \mathbf{x} \mapsto \tilde{s}$ zurückführen:

$$\begin{aligned} \tilde{\psi}(\mathbf{x}^t, \triangle T) &= \tilde{\mathbf{x}}^{t+1} = (\tilde{s}^{t+1}, s^{t+1-L}, \ldots, s^{t+1-(D-1)L}) \\ &= (f(\mathbf{x}^t), s^{t+1-L}, \ldots, s^{t+1-(D-1)L}) \\ \Rightarrow \tilde{s}^{t+1} &= f(\mathbf{x}^t) = f((s^t, s^{t-L}, \ldots, s^{t-(D-1)L})). \end{aligned} \tag{2.5}$$

[3] Auch die Vorhersagefähigkeit globaler Modelle profitiert von langen Trainingszeitreihen.

Beginnt man mit der Vorhersage bei dem zeitlich neuesten Rekonstruktionsvektor $\mathbf{x}^N$, kann der prädizierte Zeitreihenwert $\tilde{s}^{N+1}$ zur Fortsetzung der gemessenen Zeitreihe $\{s^1, s^2, \ldots s^N\}$ verwendet werden, was wiederum die Konstruktion des $N+1$sten Rekonstruktionsvektors $\tilde{\mathbf{x}}^{N+1}$ und somit eine iterative Vorhersage der Zeitreihe erlaubt.

Allerdings führt die schon beschriebene Akkumulation der Ein–Schritt–Vorhersagefehler nach Überschreiten eines gewissen Prädiktionshorizontes dazu, daß schon der in den nächsten Iterationsschritt eingehende prädizierte Zustand keine gültige Rekonstruktion des wahren Systemzustandes zu diesem Zeitpunkt mehr darstellt.

Der bei diesem Verfahren auftretende Prädiktionshorizont läßt sich jedoch nicht nur auf mangelnde Modellgenauigkeit oder gar Unzulänglichkeit des verwendeten Verfahrens zurückführen, sondern auch auf das Vorhandensein eines oder mehrerer positiver Lyapunovexponenten, die bei Zeitreihen chaotischer Systeme bereits *prinzipiell* die Vorhersagefähigkeit begrenzen.

2.2.3. Funktionsapproximation mit lokal konstantem Modell

Zur Approximation der skalaren Funktion $f(\mathbf{x})$ wird in dieser Arbeit durchgehend eine lokal konstante Modellierung, basierend auf den k Nächsten–Nachbarn $\mathbf{x}^{nn_j}, j = 1, \ldots, k$ eines Zustandes $\mathbf{x}^t$, verwendet. Bei der **absoluten** Mittelung wird dazu ein evtl. gewichteter Mittelwert der Zeitentwicklung der Nachbarn gebildet:

$$\tilde{s}^{t+1} \quad = \quad \tilde{f}(\mathbf{x}^t) = \frac{1}{\sum_{j=1}^{k} w_j} \sum_{j=1}^{k} w_j \, s^{nn_j+1} \tag{2.6}$$

Ein Problem lokal konstanter Modelle, die mangelhafte Fähigkeit zur Extrapolation außerhalb des von den Nachbarn eingeschlossenen Gebietes, kann durch Verwendung der sog. **integrierten** Mittelung gemildert werden. Dazu wird nicht der absolute Wert des zukünftigen Zeitreihenwertes $\tilde{s}^{t+1}$ vorhergesagt, sondern nur dessen Änderung (siehe Abbildung 2.2) im Vergleich zum aktuellen Wert s^t:

$$\tilde{s}^{t+1} \quad = \quad \tilde{f}(\mathbf{x}^t) = s^t + \frac{1}{\sum_{j=1}^{k} w_j} \sum_{j=1}^{k} w_j \, (s^{nn_j+1} - s^{nn_j}) \tag{2.7}$$

Ein grundsätzliches Problem beider Ansätze ist die Diskontinuität der lokalen Modellierung (vgl. McNames [28]). Kleine Änderung des Eingangszustands können dazu führen, daß

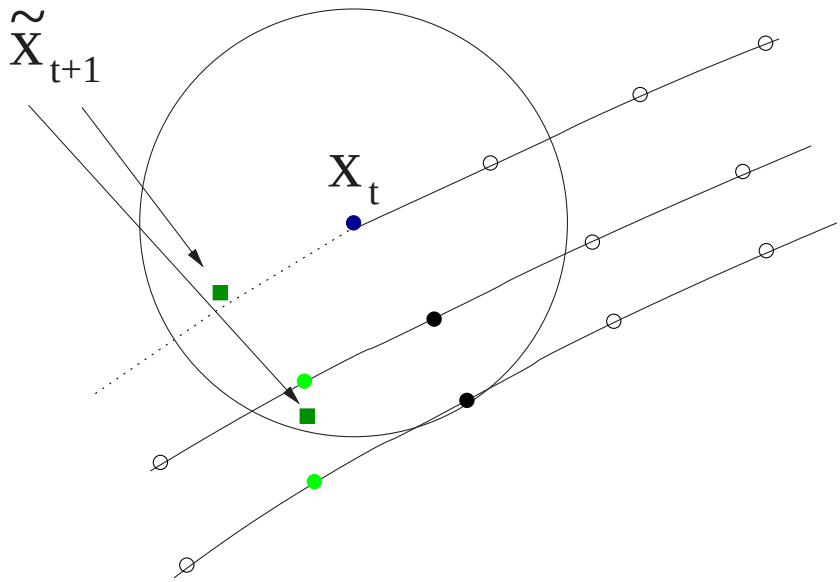


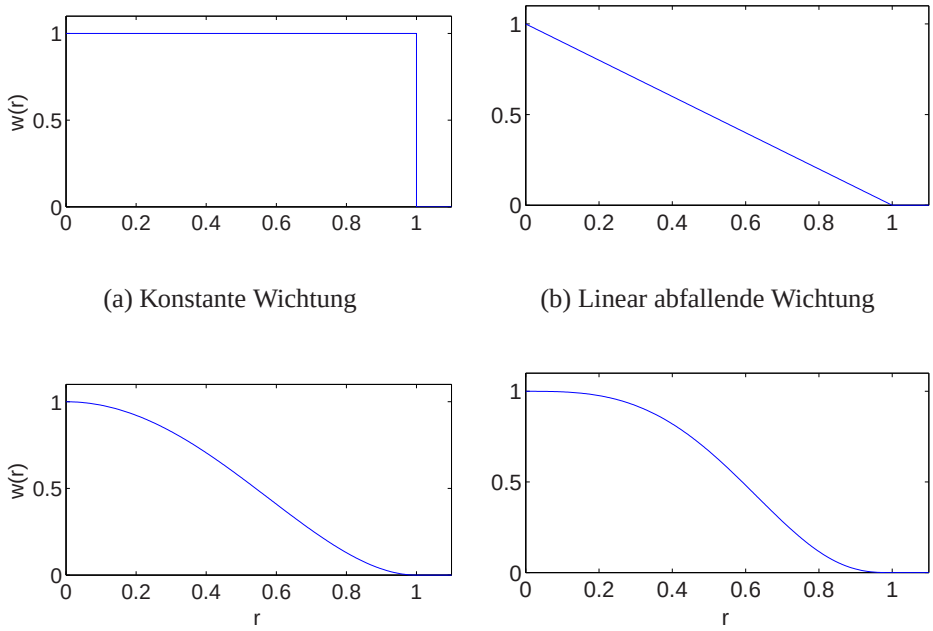
Abbildung 2.2.: Integrierte versus absolute Mittelung. Drei Trajektorien des Systems kommen jeweils von rechts oben, wobei die Zeitentwicklung des Zustandes x_t (blauer ausgefüllter Kreis) vorausgesagt werden soll. Während bei absoluter Mittelung der prädizierte Wert $\tilde{\mathrm{x}}$ (unteres blaues Quadrat) aus der Linearkombination der Zeitentwicklungen (grüne ausgefüllte Kreise) der Nächsten–Nachbarn (schwarze ausgefüllte Kreise) gewonnenen wird und somit keine Extrapolation erlaubt, kann die integrierte Mittelung die tatsächliche Zeitentwicklung (gepunktete Linie) in diesem Beispiel deutlich besser vorhersagen.

sich die Reihenfolge der gefundenen Nachbarn ändert oder sogar andere Nachbarn gefunden werden, wodurch sich ein durchaus unterschiedliches Modell ergeben kann. Daher erlauben sowohl direkte als auch integrierte Mittelung die Benutzung distanzabhängiger Gewichte $w_j = w(r_j), j = 1, \ldots, k$ zur Berechnung des Funktionswertes $\tilde{f}$. Um der unterschiedlichen Dichte der Nachbarn um einen Referenzpunkt $\mathbf{x}^t$ je nach dessen Position auf dem Attraktor gerecht zu werden, geht nicht der absolute Abstand $d_j := d(\mathbf{x}^t, \mathbf{x}^{nn_j})$ des j-ten Nachbarn ein, sondern nur der relative Abstand $r_j := \frac{d_j}{d_{k+1}}$ in die Berechnung[4] der Gewichte $w_j = w(r_j)$ ein.

Obwohl prinzipiell eine fast unbegrenzte Anzahl möglicher Gewichtsfunktionen $w(r)$ verwendet werden kann, beschränken wir uns in dieser Arbeit auf Gewichtsfunktionen des Typs

[4]Daher werden bei der gewichteten Mittelung auch, wenn nominal k Nachbarn angegeben sind, tatsächlich $k+1$ Nachbarn zu jedem Anfragepunkt bestimmt, auch wenn der $k+1$ste Nachbar nur zur Normierung der Distanzen der anderen herangezogen wird. Die implizite Sortierung der Nachbarn in Reihenfolge zunehmender Distanz zum Anfragepunkt und die Positivität der Distanz garantiert, daß sich r_j für $j = 1, \ldots, k$ im Intervall $[0, 1]$ bewegt.

$w^n(r) = (1-r^n)^n, n = 0, 1, 2, 3$, die sich durch Robustheit und einfache Berechenbarkeit auszeichnen. Für $n = 0$ ergibt sich eine konstante Gewichtung, für alle anderen Fälle fällt $w^n(r)$ von $w^n(0) = 1$ stetig auf $w^n(1) = 0$ ab, wobei jede Funktion $n - 1$ mal stetig differenzierbar ist[5] (siehe Abbildung 2.3).



(a) Konstante Wichtung

(b) Linear abfallende Wichtung

(c) Biquadratische Wichtung

(d) Trikubische Wichtung

Abbildung 2.3.: Zur Glättung der zur Funktionsapproximation verwendeten lokalen Mittelung wird eine Auswahl von distanzabhängigen Gewichtsfunktionen verwendet. Um die Modellierung unabhängig von der absoluten Skala der verwendeten Metrik zu machen, werden statt der absoluten Distanzen der k Nächsten–Nachbarn die auf den Abstand des $k + 1$sten Nachbarn normierten relativen Distanzen r benutzt.

2.3. Modellvalidierung

Nahezu alle nichtlinearen Modellierungsverfahren, seien sie lokal oder global, habe freie Parameter, die vor oder während der Konstruktion des Modelles gewählt werden müssen. Eines der grundsätzlichen Probleme bei der Modellbildung ist daher die Validierung des konstruier-

[5] Die Fälle $n = 2$ und $n = 3$ werden auch als *biquadratische* respektive *trikubische* Wichtung bezeichnet.

ten Modells bzw. dessen Parameter. Idealerweise werden diese freien Parameter so bestimmt, daß sie ein bestimmtes Maß für die Genauigkeit des Modells maximieren. Doch selbst angesichts der ständig ansteigenden verfügbaren Rechenleistung ist eine globale Optimierung aller Modellparameter so gut wie nie möglich. Das Mittel der Wahl, die nichtlineare Optimierung, kann zwar oft dazu benutzt werden, einen Teil der Parameter zu bestimmen, aber auch in Fällen, in denen das verwendete Fehlermaß konvex in einigen der Modellparametern ist, tut sich immer noch das Problem lokaler (Neben–)Maxima auf.

Trotzdem gibt es oft Methoden, die es erlauben, günstige Werte oder zumindest Wertebereiche[6] für einen Teil der Parameter anzugeben. Bei einer pragmatischen Vorgehensweise wird daher oft versucht, durch Hinzuziehen aller verfügbaren Information über das System eine Vorauswahl von möglichen Parametereinstellungen zu gewinnen und für diese dann die Genauigkeit des Modells zu ermitteln. Limitiert wird ein solcher Ansatz natürlich von der zur Verfügung stehenden Rechenleistung und Geduld des Experimentators.

2.3.1. Cross–Validation

Ein Modell sollte allerdings nicht nur die Daten, anhand derer man das Modell konstruiert bzw. trainiert hat, möglichst gut beschreiben[7], sondern auch Datensätze gleichen Ursprungs, die aber zum Zeitpunkt der Konstruktion des Modells noch nicht bekannt waren oder vorlagen, vorhersagen können, also die Fähigkeit zur Generalisierung besitzen. In der Praxis sind dies oft zwei gegenläufige Forderungen an das Modell, so daß ein genaues Abwägen seitens des Experimentators erforderlich ist.

Eine Methode, sowohl Genauigkeit als auch Generalisierungsvermögen eines Modells zu testen, wenn nur ein Datensatz des zu modellierenden Systems vorliegt, ist die sog. *Cross–Validation*[8]. Bei diesem Verfahren wird der Datensatz in zwei Teile aufgeteilt:

1. Einen Trainingsdatensatz, auf dessen Grundlage das Modell erstellt bzw. trainiert wird.

[6] So lassen sich z.B. hinreichend gute Werte für die Einbettungsdimension bei der Methode der Zeitverzögerungskoordinaten mittels Verfahren wie *False–Nearest–Neighbors* [12] oder dem Grassberger–Procaccia Algorithmus zur Dimensionsschätzung (vgl Abschnitt 4.2.2) bestimmen.

[7] Dies wäre im Extremfall gleichbedeutend mit einer *Kodierung* der Eingangsdaten, was zwar eine perfekte Rekonstruktion dieser zuließe, aber einem solchem Modell jegliches Generalisierungsvermögen nehmen würde.

[8] In der Statistik werden derartige Verfahren als *Bootstrap-Methoden* bezeichnet [24].

2. Einen Testdatensatz, an welchem man den Vorhersagefehler unter Verwendung verschiedener Parametereinstellungen bestimmt bzw. minimiert.

Die (zufällige) Einteilung des gesamten Datensatzes in zwei Teile wird normalerweise wiederholt ausgeführt und der Fehler gemittelt, um aussagekräftige Ergebnisse zu erhalten. Obwohl prinzipiell ein beliebiges Fehlermaß verwendet werden kann, finden hauptsächlich mittlerer quadratischer Fehler oder absoluter Fehler Verwendung. Wesentlich bei der Methode der Cross–Validation ist, keinerlei Information über den Testdatensatz in die Modellbildung eingehen zu lassen, um zu verhindern, daß auch dieser in das Modell kodiert wird und somit das Ziel der Bestimmung des Generalisierungsvermögens unterminiert.

2.3.2. Leave–One–Out Cross–Validation für lokale Modelle

Eine „extreme“ Variante der Cross–Validation ist die sog. Leave–One–Out Cross–Validation. Dabei wird ein Datensatz der Größe N wiederholt in einen Trainingsdatensatz der Größe $N-1$ und einen Testdatensatz der Größe 1 aufgeteilt und der Fehler (*one–step ahead cross–validation error*, kurz OSCV) bei der Vorhersage des ausgelassenen Punktes bzw. dessen Nachfolgers mittels des aus dem Trainingsdatensatz konstruierten Modelles bestimmt.

Während die Leave–One–Out Cross–Validation bei umfangreichen Datensätzen für globale Modelle zu enorm hohem Rechenaufwand führen kann, bietet sie sich für Validierung lokaler Modelle an, wo sie einfach durch Vernächlässigen des auszulassenden Punktes bei der k–Nächsten–Nachbarsuche realisiert wird. Diese Tatsache läßt diese Art der Modellvalidierung auch für die im Kapitel 5 vorgestellte Anwendung besonders geeignet erscheinen.

Der OSCV bei Verwendung des mittleren quadratischen Fehlers ergibt sich zu:

$$\mathrm{MSE}_1 \quad = \quad \frac{1}{|T_{\mathrm{ref}}|} \sum_{t \in T_{\mathrm{ref}}} \left(s^{t+1} - \tilde{f}_t^t(\mathbf{x}^t) \right)^2, \tag{2.8}$$

wobei $\tilde{f}_{t_1}^{t_2}(\mathbf{x}^t)$ die gemäß Gleichung 2.6 bzw. Gleichung 2.7 konstruierte Ein–Schritt–Vohersagefunktion ist, bei der alle Rekonstruktionsvektoren mit Zeitindex von t_1 bis t_2 (jeweils einschließlich) von der Nachbarsuche ausgeschlossen sind. T_{ref} bezeichnet eine hinreichend große, zufällig gewählte Stichprobe möglicher Zeitindizes, an denen die Vorhersage validiert wird[9].

[9] Dies entspricht der typischen Vorgehensweise in der Praxis, wo die Cross–Validierung bei umfangreichen Da-

Wird der OSCV nach Gleichung 2.8 für sehr fein abgetastete Zeitreihen berechnet[10], tritt das Problem auf, daß zeitlich eng benachtbarte Zustände $(\ldots, \mathbf{x}^{t-2}, \mathbf{x}^{t-1}, \mathbf{x}^{t+1}, \mathbf{x}^{t+2}, \ldots)$ des ausgelassenen Zustandes $\mathbf{x}^t$ u.U. dichter an $\mathbf{x}^t$ liegen als Zustände, die nicht auf dem gleichen Trajektoriensegment liegen (siehe Abbildung 2.2, nicht ausgefüllter Kreis rechts oberhalb des Zustandes x_t). Da diese zeitlich eng benachtbarten Zustände wenig zur Vorhersage beitragen, bietet es sich an, statt nur den aktuellen Zustand $\mathbf{x}^t$ von der Nachbarsuche auszuschließen, ein ganzes Trajektoriensegment $(\mathbf{x}^{t-c}, \ldots, \mathbf{x}^t, \ldots, \mathbf{x}^{t+c})$ bei der Suche zu ignorieren:

$$\mathrm{MSE}_1^c = \frac{1}{|T_{\text{ref}}|} \sum_{t \in T_{\text{ref}}} \left(s^{t+1} - \tilde{f}_{t-c}^{t+c}(\mathbf{x}^t) \right)^2 . \tag{2.9}$$

Ein Vorteil der Verwendung des OSCV zur Validierung ist dessen vergleichsweise kleiner Rechenaufwand. So muß für jeden Zeitindex nur eine (rechenintensive) Suche der $k+1$ Nächsten–Nachbarn ausgeführt werden, dieses Ergebnis läßt sich dann sowohl für Modelle mit kleinerer Anzahl Nachbarn als auch für alle beschriebenen Varianten der Funktionsapproximation verwenden.

2.3.3. Erweiterungen der Leave–One–Out Cross–Validation

Ein Nachteil bei der Verwendung des OSCV ist die fehlende Berücksichtigung der Empfindlichkeit des Modells im Bezug auf die akkumulierten Fehler im Eingangsvektor bei iterierter Prädiktion. Ein Parametersatz, der den OSCV optimiert, muß nicht zwangsläufig auch optimal bei der iterierten Vorhersage der Länge p sein, was jedoch das Ziel der Validierung ist. Dies läßt die Verwendung des *multi–step ahead cross–validation error* (MSCV), der den mittleren Fehler bei der iterierten Vorhersage über p Schritte erfaßt, als besser geeignet erscheinen. Bei Verwendung der mittleren quadratischen Abweichung als Fehlermaß ergibt sich der MSCV zu:

$$\begin{aligned} \mathrm{MSE}_{1,p}^c = \frac{1}{p|T_{\text{ref}}|} \sum_{t \in T_{\text{ref}}} \Bigg(& \left(s^{t+1} - \tilde{f}_{t-c}^{t+c}(\mathbf{x}^t) \right)^2 \\ & + \sum_{i=1}^{p-1} \left(s^{t+i+1} - \tilde{f}_{t+i-c}^{t+i+c}(\tilde{\mathbf{x}}^{t+i}) \right)^2 \Bigg) , \end{aligned} \tag{2.10}$$

tensätzen meistens nicht für alle möglichen Zeitindizes ausgeführt wird, um Rechenzeit zu sparen. Prinzipiell kann diese Auswahl natürlich alle Zeitindizes umfassen.

[10] Die Verwendung fein abgetasteter Zeitreihen bzw. die artifizielle Interpolation gemessener Zeitreihen auf ein Mehrfaches der Nyquist–Frequenz wird u.A. von McNames empfohlen, um die Vorhersagegenauigkeit zu steigern [28].

wobei die iterierte Prädiktion nach Einsetzen eines anfänglich bekannten Rekonstruktionsvektors $\mathbf{x}^t$ nur noch auf den vorhergesagten Zuständen $\tilde{\mathbf{x}}^{t+i}$ weiterläuft.

Leider ist die numerische Berechnung des MSCV deutlich aufwendiger als die Berechnung des OSCV, da hier die Nachbarsuche für jede Modellvariante (in Anzahl der Nachbarn und Methode der Funktionsapproximation) erneut ausgeführt werden muß. Zwar sind Nachbarn des aus der Trainingszeitreihe gewonnenen Zustandes $\mathbf{x}^t$ für alle Varianten gleich, doch werden nach dem ersten Prädiktionsschritt dann Nachbarn der rekonstruierten Zustände $\tilde{\mathbf{x}}^{t+i}$ gesucht, die sich i.A. von Variante zu Variante unterscheiden werden.

Die Anzahl der Iterationsschritte p darf nicht zu hoch gewählt werden, da bei Überschreiten des systembedingten Prädiktionshorizontes kein aussagekräftiges Minimum in der Fehlerlandschaft des MSCV aufzufinden ist.

2.4. Beispiele lokaler Modellierung zur Vorhersage skalarer Zeitreihen niedrigdimensionaler dynamischer Systeme

Die bereits diskutierten Methoden zur Cross–Validierung wurden auf vier Zeitreihen verschiedener in der Literatur beschriebener chaotischer Systeme angewandt. Mit den daraus gewonnenen optimalen Parametersätzen wurde anschließend eine iterierte Vorhersage der Zeitentwicklung des Systems über das Ende der Trainingszeitreihe hinaus vorgenommen (siehe Abbildungen 2.4). In den folgenden Abschnitten wird die genaue Vorgehensweise beschrieben:

2.4.1. Untersuchte Systeme

Zeitreihen zu je 10000 Abtastwerten wurden durch numerische Integration der folgenden Systeme gewöhnlicher Differentialgleichungssysteme (DGLS) generiert:

(a) Baier–Sahle DGLS:
Dabei handelt es sich um eine hyperchaotische Verallgemeinerung des Rössler–

Systems, die von Baier und Sahle eingeführt wurde [7]:

$$\begin{aligned} \dot{x}_1 &= -x_2 + ax_1 \\ \dot{x}_i &= x_{i-1} - x_{i+1} \quad i = 2, \ldots, M-1 \\ \dot{x}_M &= \epsilon + bx_M(x_{M-1} - d) \end{aligned}$$

mit den Parametern $a = 0.28$, $b = 4$, $d = 2$, $\epsilon = 0.1$. Zwei Datensätze mit $M = 3$ sowie mit $M = 5$ wurden erzeugt[11]. Die Systeme wurden jeweils von $T = 0$ bis $T = 4000$ integriert, wobei ihr Zustand alle $\triangle T = 0.2$ abgetastet wurde. Die ersten 10000 Sample wurden als transient verworfen. Die folgenden 10000 Werte der ersten Variable x_1 wurden dann als skalare Zeitreihe aufgefaßt. Die Informationsdimension der Systeme wurde mit der in Abschnitt 4.2.3 vorgestellten Methode für den Fall $M = 3$ zu $D_1 = 2.01$, für den Fall $M = 5$ zu $D_1 = 4.26$. ermittelt.

(b) Chua Oszillator:

$$\begin{aligned} \dot{x}_1 &= \alpha(x_2 - h(x_1)) \\ \dot{x}_2 &= x_1 - x_2 + x_3 \\ \dot{x}_3 &= -\beta x_2 \\ h(y) &= ym_2 + 0.5(m_2 - m_1)(|y + c_0| - |y - c_0|) \end{aligned}$$

mit den Parametern $\alpha = 9$, $\beta = 14.286$, $m_1 = -\frac{1}{7}$, $m_2 = \frac{2}{7}$ sowie $c_0 = 1$. Das System wurde von $T = 0$ bis $T = 2000$ integriert, wobei der Zustand alle $\triangle T = 0.1$ abgetastet wurde. Die ersten 10000 Sample wurden als transient verworfen. Die folgenden 10000 Werte der ersten Variable x_1 wurden dann als skalare Zeitreihe aufgefaßt. Als Informationsdimension ergab sich ein Wert von $D_1 = 2.30$.

(c) Lorenz DGLS:

$$\begin{aligned} \dot{x}_1 &= \sigma(x_1 - x_2) \\ \dot{x}_2 &= rx_1 - x_2 - x_1x_3 \\ \dot{x}_3 &= x_1x_2 - bx_3 \end{aligned}$$

[11] Anm.: Die Systemdimension M darf nur ungeradzahlige Werte annehmen. Da das System erst für $M > 3$ hyperchaotisch wird, ist 5 die kleinste Systemdimension, ab der von einem hyperchaotischen System gesprochen werden kann.

mit den Parametern $\sigma = -10$, $b = \frac{8}{3}$ sowie $r = 28$. Das System wurde von $T = 0$ bis $T = 600$ integriert, wobei der Zustand alle $\triangle T = 0.03$ abgetastet wurde. Die ersten 10000 Sample wurden als transient verworfen. Die folgenden 10000 Werte der ersten Variable x_1 wurden dann als skalare Zeitreihe aufgefaßt. Als Informationsdimension ergab sich ein Wert von $D_1 = 2.15$.

(d) Rössler DGLS:

$$\begin{aligned} \dot{x}_1 &= -x_2 - x_3 \\ \dot{x}_2 &= x_1 + a x_2 \\ \dot{x}_3 &= b + x_3(x_1 - c) \end{aligned}$$

mit den Parametern $a = 0.45$, $b = 2$ sowie $c = 4$. Das System wurde von $T = 0$ bis $T = 4000$ integriert, wobei der Zustand alle $\triangle T = 0.2$ abgetastet wurde. Die ersten 10000 Sample wurden als transient verworfen. Die folgenden 10000 Werte der ersten Variable x_1 wurden dann als skalare Zeitreihe aufgefaßt. Als Informationsdimension ergab sich ein Wert von $D_1 = 1.97$.

2.4.2. Verwendete Metrik

Zur Berechnung der Distanzen im Rekonstruktionsraum wurde eine exponentiell gewichtete euklidische Metrik verwendet [28]:

$$d_\lambda(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{D} \lambda^{i-1}(x_i - y_i)^2}$$

bzw. im Fall der Time–Delay–Vektoren

$$d_\lambda(\mathbf{x}^{t_1}, \mathbf{x}^{t_2}) = \sqrt{\sum_{i=0}^{D-1} \lambda^i (s^{t_1 - iL} - s^{t_2 - iL})^2}.$$

Die für $0 < \lambda \leq 1$ exponentiell abfallende Gewichtung betont die aktuellen Zeitreihenwerte stärker als die weiter in der Vergangenheit liegenden, um deren abklingendem Einfluß auf den aktuellen Zustand Rechnung zu tragen. Die exponentiell gewichtete euklidische Metrik schließt für den Fall $\lambda = 1$ die gewöhnliche euklidische Metrik ein, kann somit die Modellgenauigkeit höchstens verbessern.

2.4.3. Verwendetes Fehlermaß

Zur besseren Vergleichbarkeit der Ergebnisse wurde als Maß des Fehlers der iterierten Vorhersage eine mit der mittleren quadratischen Abweichung der Zeitreihe normierte Version des bereits vorgestellten $\mathrm{MSE}^c_{1,p}$ (vgl. Gleichung 2.10) verwendet:

$$\begin{aligned}\mathrm{NMSE}^c_{1,p} &= \frac{N}{p|T_{\text{ref}}|\sum_{t=1}^{N}(s^t-\bar{s})^2}\sum_{t\in T_{\text{ref}}}\left(\left(s^{t+1}-\tilde{f}^{t+c}_{t-c}(\mathbf{x}^t)\right)^2\right.\\ &\quad\left.+\sum_{i=1}^{p-1}\left(s^{t+i+1}-\tilde{f}^{t+i+c}_{t+i-c}(\tilde{\mathbf{x}}^{t+i})\right)^2\right). \qquad (2.11)\end{aligned}$$

2.4.4. Parametervorgaben

Zur Wahl des Parameters c, der den Ausschluß zeitlich benachtbarter Zustände des vorherzusagenden Punktes steuert, wird hier die *mittlere Wiederkehrzeit* des Systems verwendet. Dazu wird die Entwicklung des Abstandes $d_i = d(\mathbf{x}^r, \mathbf{x}^r + i), i = 0, 1, \ldots$ zwischen zufällig gewählten Referenzpunkten $\mathbf{x}^r$ und deren zeitlichen Nachfolgern mit wachsendem i betrachtet. Anfangs wird d_i zunehmen. Der erste Zeitpunkt i_u, zu dem $d_{i_u} < d_{i_u-1}$ gilt, wird als halbe Wiederkehrzeit gewertet.

Statt nun Rekonstruktionsdimension D und Delay L unabhängig voneinander als freie Parameter aufzufassen, wurde L konstant zu 1 gewählt. In diesem Fall unterscheiden sich zwei zeitlich aufeinanderfolgende Zeitverzögerungsvektoren $\mathbf{x}^t$ und $\mathbf{x}^{t+1}$ lediglich durch einen neu hinzugekommenen Zeitreihenwert s^{t+1} (sowie die Stellung der anderen Elemente)

$$\begin{aligned}\mathbf{x}^t &= (s^t, s^{t-1}, \ldots, s^{t-(D-1)})\\ \mathbf{x}^{t+1} &= (s^{t+1}, s^t, s^{t-1}, \ldots, s^{t-(D-2)}),\end{aligned}$$

wodurch Gleichung 2.5 besonders einfach wird:

$$\begin{aligned}\tilde{\psi}(\mathbf{x}^t, \triangle T) &= \tilde{\mathbf{x}}^{t+1} = (\tilde{s}^{t+1}, s^t, s^{t-1}, \ldots, s^{t-(D-2)})\\ &= (f(\mathbf{x}^t), s^t, s^{t-1}, \ldots, s^{t-(D-2)})\\ \tilde{s}^{t+1} &= f(\mathbf{x}^t) = f((s^t, s^{t-1}, \ldots, s^{t-(D-1)})).\end{aligned}$$

Die vergleichsweise hohen Dimensionen des Rekonstruktionsraums, in dem die Nachbarsuche stattfindet, erhöht die Laufzeit der Validierung nur unwesentlich, da die Effizienz des

System	p	D	k	λ	n	Modus	$\mathrm{NMSE}^c_{1,p}$	$\frac{\mathrm{NMSE}_{\max}}{\mathrm{NMSE}_{\min}}$
Baier–Sahle, $M = 3$	80	12	6	1.0	1	Int.	0.007919	49.9
Baier–Sahle, $M = 5$	40	40	8	1.0	1	Int.	0.050045	27.9
Chua	50	30	5	1.0	3	Int.	0.053058	14.6
Lorenz	50	40	1	0.5	0	Int.	0.084055	5.2
Rössler	80	40	1	0.7	0	Abs.	0.005964	20.7

Tabelle 2.1.: Ergebnisse der Crossvalidierung. Jede Zeile der Tabelle zeigt den Parametersatz für das jeweilige System, der das verwendete Fehlermaß (vorletzte Spalte) minimiert. Die letzte Spalte zeigt das Verhältnis des Fehlers des schlechtesten Parametersatzes (der hier nicht angegeben ist) zum optimalen.

hier verwendeten ATRIA Algorithmus wesentlich empfindlicher von der in diesen Fällen vergleichsweise niedrigen fraktalen Dimensionen des Datensatzes als von der „formalen“ Rekonstruktionsdimension D abhängt (siehe Abschnitt 4.3).

Bei der Validierung wurde der $\mathrm{NMSE}^c_{1,p}$ (siehe Tabelle 2.1) für alle Kombinationen aus folgenden Parametervorgaben bzw. Approximationsmethoden ermittelt:

- Rekonstruktionsdimension $D \in \{4, 8, 12, 16, 20, 25, 30, 40\}$
- Exponent der gewichteten Metrik $\lambda \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
- Anzahl verwendeter Nachbarn $k \in \{1, \ldots, 8\}$
- Konstante, linear abfallende, biquadratische oder trikubische Wichtung w^n, $n \in \{1, \ldots, 8\}$
- Absolute oder intergrierte Mittelung

Insgesamt wurde dabei jede Zeitreihe 1000 mal zufällig in Test- und Trainingsdatensatz aufgeteilt.

2.4.5. Vorhersagen

Abbildung 2.4 zeigt die Ergebnisse der freilaufenden, iterierten Vorhersage für die generierten Zeitreihen. Für die Modellierung jeder Zeitreihe wurde der bei der Crossvalidierung ermittelte optimale Parametersatz verwendet. Zur Beurteilung der Güte der Vorhersage wurde jeweils der weitere Verlauf der Originalzeitreihe als gestrichelte Linie mit dargestellt, dieser ist weder

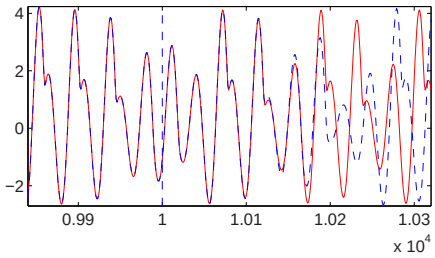
in die Crossvalidierung noch in den Trainingsdatensatz der Prädiktion eingegangen. Die Länge der Vorhersage beträgt jeweils das Vierfache der Anzahl p der zur Bestimmung des Fehlers $\mathrm{NMSE}^{c}_{1,p}$ verwendeten Iterationen (siehe Tabelle 2.1).

Der Vergleich der Güte der Modellierung anhand der dargestellten Vorhersagen gestaltet sich als schwierig. Zwar läßt sich ein gewisser Zusammenhang zwischen der Länge des übereinstimmenden Verlaufes von Vorhersage und Original und dem Fehler der Crossvalidierung ausmachen, doch erlaubt dieser keine definitive Aussage über den zu erwartenden Prädiktionshorizont im Einzelfall, da dieser selbst bei Zeitreihen des gleichen Systems nicht invariant bezüglich des Startzeitpunktes der Vorhersage ist. So scheint die besonders gute Vorhersage des Rössler Systems, die erst am Rande des in Abbildung 2.4(e) wiedergegebenen Bereiches von dem tatsächlichen Verlauf der Originalzeitreihe abzuweichen beginnt, zu einem besonders günstigen Startzeitpunkt begonnen zu haben.

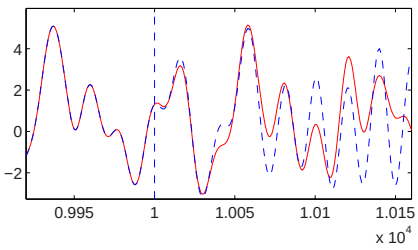
2.5. Zusammenfassung

Die vorgestellte Methode der Zeitverzögerungsrekonstruktion zusammen mit der Verwendung lokaler Modellierung zur Funktionsapproximation stellt ein mächtiges Werkzeug zur iterierten Vorhersage von Zeitreihen deterministischer nichtlinearer Systeme dar. Zugleich erlaubt die lokale Funktionsapproximation unter Verwendung benachbarter Zustände die effiziente Validierung der gewonnenen Modelle mit der Technik der Cross–Validation, bei der ein einziger Datensatz wiederholt in Test- und Trainingsdatensatz eingeteilt wird. Eine Erweiterung für den Fall der iterierten Prädiktion stellt der MSCV (multi–step ahead cross–validation error) dar, der zur Ermittlung der optimalen Modellparameter für die untersuchten Zeitreihen benutzt wird. Die damit erzeugten Vorhersagen können zwar die Zeitentwicklung auf kurzen Zeitskalen reproduzieren, doch stoßen auch diese Verfahren an die prinzipiellen Grenzen der Vorhersagbarkeit chaotischer Systeme.

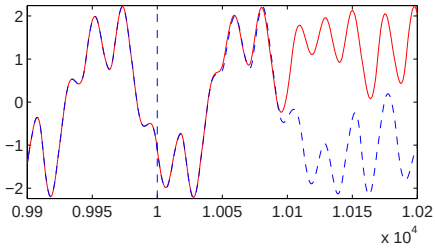
Bislang wurde noch nicht darauf eingegangen, wie die Nächste–Nachbar–Suche, die als Kernkomponente der lokalen Modellierungsverfahren gerade im Hinblick auf deren numerische Praktikabilität von Bedeutung ist, effizient ausgeführt werden kann. Dies geschieht im folgenden Kapitel.



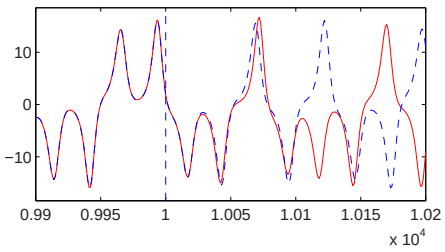
(a) Baier–Sahle, $M = 3$



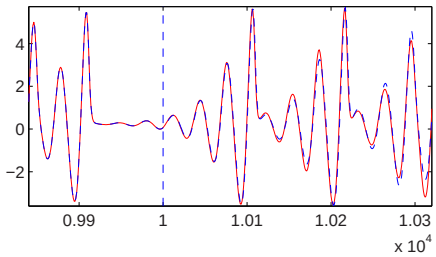
(b) Baier–Sahle, $M = 5$



(c) Chua



(d) Lorenz



(e) Rössler

Abbildung 2.4.: Freilaufende Vorhersagen basierend auf den Ergebnissen der Crossvalidierung für Zeitreihen verschiedener niedrigdimensionaler dynamischer Systeme. Von jedem System wurden je 10000 Zeitreihenwerte als Trainingsdatensatz verwendet. Die gestrichelte Linie gibt den Verlauf der ursprünglichen Zeitreihe an. Die durchgezogene Linie gibt den Verlauf der ab Sample 10001 freilaufenden, iterierten Vorhersage basierend auf dem Trainingsdatensatz und den während der Crossvalidierung gefundenen optimalen Parametereinstellungen an. Zur besseren Visualisierung wurde ein Teil der Trainingszeitreihe mit dargestellt. Der Beginn der freilaufenden Prädiktion ist jeweils mit einer gestrichelten vertikalen Linie gekennzeichnet. Die verwendeten Parametereinstellungen sind in Tabelle 2.1 aufgeführt.

3. Effiziente Nächste-Nachbar Suche

Der im diesem Kapitel vorgestellte Algorithmus zur Nächsten–Nachbar Suche ist ein Kernteil der vorgelegten Arbeit. Um diesen herum gruppieren sich die in den anderen Kapiteln dargestellten Anwendungen. Eine Beispielimplementation in C++ ist im Anhang A.4.1 zu finden.

3.1. Einführung

3.1.1. Begriffsbildungen

3.1.1.1. Metrischer Raum

Definition 1 *des metrischen Raums*
Eine Menge $\mathbf{Y}$ *von Elementen* $x, y, ...$ *heißt metrischer Raum, wenn jedem Elementenpaar* $x, y \in \mathbf{Y}$ *eine reelle Zahl* $d(x, y)$ *mit folgenden Eigenschaften zugeordnet ist:*

(1) $d(x, y) \geq 0$*, wobei* $d(x, y) = 0$ *genau dann, wenn* $x = y$

(2) $d(x, y) = d(y, x)$

(3) Für drei beliebige Elemente $x, y, z \in M$ *gilt die* Dreiecksungleichung $d(x, z) \leq d(x, y) + d(y, z)$

Man nennt $d(x, y)$ *den* Abstand *zwischen* x *und* y*. Die Elemente von* $\mathbf{Y}$ *heißen auch Punkte.*

Die Punkte eines metrischen Raumes müssen nicht notwendigerweise aus einem Vektorraum stammen. Hier sind, speziell im Bereich der Informatik, auch Datenstrukturen oder Daten-

bankeinträge denkbar. Entsprechend weit gefaßt ist auch der Begriff der Metrik oder Abstandsfunktion, der über die häufig verwendete euklidische Metrik oder die Maximumsmetrik hinausgeht, auch wenn in dieser Arbeit letztendlich meistens auf diese zurückgegriffen wird. Trotzdem muß betont werden, daß der hier vorgestellte Algorithmus zur Nachbarsuche sowohl theoretisch als auch in der Realisierung als Programmcode mit jeder Metrik, die den oben genannten Bedingungen genügt, einsetzbar ist.

3.1.1.2. Nächste–Nachbarn

Definition 2 *Definition der k–Nächsten–Nachbarn:*
Sei $\mathbf{Y}$ ein metrischer Raum mit Abstandsfunktion $d(x, y)$. $\mathbf{X}$ sei eine endliche Teilmenge von $\mathbf{Y}$ mit Mächtigkeit $|\mathbf{X}| = N$ (endlicher Datensatz). Weiter sei q ein Punkt aus $\mathbf{Y}$, der nicht notwendigerweise auch Element von $\mathbf{X}$ sein muß (Anfragepunkt). Die Punkte $n_i^0 \in X \quad i = 1, \ldots, k$ nennt man die k–Nächsten–Nachbarn zum Anfragepunkt q falls gilt:

$$0 \leq d(n_i^0, q) \leq d(n_{i+1}^0, q) \quad i = 1, \ldots, k-1 \tag{3.1}$$

$$d(n_k^0, q) \leq d(x, q) \, \forall x \in X \backslash \{n_1^0, \ldots, n_k^0\} \tag{3.2}$$

Die k–Nächsten–Nachbarn sind eindeutig bis auf Permutationen innerhalb von Gruppen von Nachbarn mit gleicher Distanz zum Anfragepunkt. Speziell kann die Menge der k–Nächsten–Nachbarn nicht eindeutig sein, falls es mehrere Punkte $x \in \mathbf{X}$ gibt, deren Abstand zu q gleich $d(n_k^0, q)$ ist. Falls erforderlich, läßt sich die Eindeutigkeit wieder herstellen, indem man die Punkte in X durchnummeriert (indiziert) und fordert, bei Gruppen von Nachbarn gleicher Distanz diese in Reihenfolge aufsteigender Indizes anzuordnen.

3.1.1.3. Approximative Nächste–Nachbarn

Die zur Bezeichnung der Nächsten–Nachbarn verwendete Notation n_i^0 mit hochgestellter 0 soll auf die anschließend eingeführte Unterscheidung zwischen *exakten* und *approximativen* Nächsten–Nachbarn vorbereiten [5]. Der in Definition 2 vorgestellte klassische Nachbar–Begriff entspricht dabei den exakten Nächsten–Nachbarn.

Definition 3 *Definition der ϵ–approximativen k–Nächsten–Nachbarn:*
Ausgehend von den in Def. 2 beschriebenen k–Nächsten–Nachbarn, nennt man jeden Punkt n_i^ϵ einer in Reihenfolge zunehmender Distanz zum Anfragepunkt q sortierten Teilmenge $A \subset X$, $A = \{n_i^\epsilon \in X \quad i = 1, \ldots, k\}$ ϵ-approximativen Nachbar, wenn gilt:

$$d(n_i^\epsilon, q) \leq (1 + \epsilon)\, d(n_i^0, q) \qquad i = 1, \ldots, k \tag{3.3}$$

Damit erfüllen die exakten k–Nächsten–Nachbarn für jedes $\epsilon \geq 0$ die Definition der ϵ–approximativen k–Nächsten–Nachbarn. Generell darf ein Algorithmus bei der ϵ–approximativen Suche jedoch Punkte des Datensatzes als Ergebnis zurückliefern, die eine bis zu $(1 + \epsilon)$mal größere Distanz zu q aufweisen als der jeweils entsprechende exakte Nächste–Nachbar[1]. Damit schwächen approximative Nächste–Nachbarn die Forderungen des exakten Problems in einer Weise ab, die bei nur minimaler Erweiterung des Algorithmus eine deutliche Verringerung des Rechenaufwandes zuläßt (siehe 3.4.4.2).

3.2. ATRIA - Ein auf der Dreiecksungleichung basierender Algorithmus

Der ATRIA (**A**dvanced **Tr**iangle **I**nequality **A**lgorithm) arbeitet in zwei getrennten Phasen:

Vorverarbeitungsphase Während dieser Phase wird eine Datenstruktur angelegt, die das Lokalisieren von benachbarten Punkten beschleunigt. Dabei wird der Datensatz in einer Art *divide and conquer* Verfahren rekursiv in sog. *Cluster* zerlegt, die jeweils einen Teil der Punkte des Datensatzes repräsentieren. Die Cluster bilden eine hierarchische Folge von Überdeckungen des Datenraumes (siehe Abbildung 3.1) und werden daher zu einem (Such)Baum angeordnet. Diese Phase des Algorithmus hat eine durchschnittliche Zeit–Komplexität von $O(N \log N)$ und einen Speicherbedarf von $O(N)$.

Suchphase Während der Suchphase werden die in der Vorverarbeitungsphase angelegten Datenstrukturen benutzt, um eine beliebige Anzahl von Suchanfragen (*requests* oder

[1] Daher wird im folgenden auch von einem relativen Fehler der zurückgelieferten Distanzen bis zu ϵ gesprochen. Dies darf nicht so verstanden werden, daß tatsächlich ein Fehler bei der Berechnung des numerischen Wertes der Distanz gemacht wird. Vielmehr liefert der Algorithmus bei der Suche nach approximativen Nachbarn einen Satz an Punkten zurück, die nicht notwendigerweise identisch zu den tatsächlichen Nächsten–Nachbarn sind.

queries) zu beantworten. Da die Komplexität einer einzelnen Suchanfrage $O(\log N)$ beträgt, amortisiert sich der Aufwand der Vorverarbeitung erst nach $O(N)$ Suchanfragen.

Die in der Vorverarbeitungsphase angelegten Datenstrukturen sind unabhängig von der Art der Anfragen (exakte Nächste–Nachbarn, approximative Nächste–Nachbarn oder Bereichssuche), die während der Suchphase gestellt werden, und müssen daher nur einmal für einen Datensatz ausgeführt werden. Lediglich bei Änderung der zur Distanzberechnung verwendeten Metrik oder Einfügen bzw. Entfernen von Punkten in bzw. aus dem Datensatz muß eine erneute Vorverarbeitung erfolgen.

3.3. Vorverarbeitung

3.3.1. Hierarchische Überdeckung

Die im ATRIA verwendete Datenstruktur des Clusters dient der geometrischen Charakterisierung einer Teilmenge der Punkte des Datensatzes. Dazu beinhaltet ein Cluster neben einer Liste der zu ihm gehörenden Punkte auch noch Informationen über deren Anordnung in Bezug zu einem ausgezeichneten Punkt des Clusters, dem sog. *charakteristischen Punkt*. Dieser wird sowohl bei der Division eines Clusters in dessen Abkömmlinge als auch bei der Bearbeitung einer Suchanfrage benutzt. Obwohl im folgenden bei der Beschreibung eines Cluster von charakteristischem Punkt c und Radius R gesprochen wird, darf man sich die Punkte eines Clusters weder unbedingt besonders dicht noch kugelförmig um c herum gruppiert vorstellen, vielmehr bestimmen Metrik und Zuordungsstrategie (siehe 3.3.2) die tatsächliche Geometrie der Punkte des Clusters.

Ausgehend vom *Rootcluster*, der alle Punkte des Datensatzes umfaßt, wird ein binärer Baum aufgebaut, dessen Knoten Cluster sind. Folgende Eigenschaften kennzeichnen die hierarchische Zerlegung des Datenraumes in Cluster:

- Die Cluster einer Ebene des Baumes enthalten zusammen alle Punkte des Datensatzes[2].

[2] Es wurden auch Varianten des Algorithmus implementiert, in der die charakteristischen Punkte nicht weiter in die Rekursion eingingen. Dies hat den Vorteil, daß ein Punkt des Datensatzes höchstens einmal als charakteristischer Punkt verwendet werden kann und somit keine redundanten Distanzberechnungen stattfinden. Die empirischen Laufzeiten beider Varianten unterscheiden sich allerdings nur unwesentlich.

- Die Cluster einer Ebene des Baumes sind paarweise disjunkt.
- Die Abkömmlinge eines Clusters enthalten zusammen alle Punkte ihres Elternclusters.

3.3.2. Zuordungsstrategie

Im diesem Abschnitt wird die rekursive Einteilung des Datensatzes als Pseudo–Programmablauf dargestellt:

- Wähle zwei Punkte des aktuellen Clusters aus. Diese Punkte werden jeweils zum charakteristischen Punkt des linken und des rechten Abkömmlings des aktuellen Clusters. Obwohl verschiedene Auswahlstrategien denkbar sind, wird im ATRIA–Algorithmus eine vergleichsweise einfache verwendet:
  - Falls es sich beim aktuellen Cluster um den Rootcluster handelt, wähle zuerst einen beliebigen Punkt a aus. Ansonsten setze a gleich dem charakteristischen Punkt des aktuellen Clusters.
  - Bestimme denjenigen Punkt c_r des aktuellen Clusters, der die größtmögliche Entfernung zu a hat. Dieser wird der charakteristische Punkt des ersten (rechten[3]) Abkömmlings.
  - Der charakteristische Punkt c_l des zweiten (linken) Abkömmlings wird nun derjenige Punkt, der den größten Abstand zu c_r aufweist.
- Ordne jeden Punkt des aktuellen Clusters einem der beiden Abkömmlinge zu. Als Kriterium wird die Distanz zu den beiden charakteristischen Punkten verwendet: Die Zuweisung erfolgt jeweils zu dem Cluster mit dem näheren charakteristischen Punkt.
- Berechne und speichere die maximale Distanz R vom charakteristischen Punkt zu den zugeordeneten Punkten für jeden Abkömmling.

[3] Der Einfachheit halber werden hier zur Unterscheidung der beiden Abkömmlinge eines Knotens des Baumes die Begriffe *links* und *rechts* verwendet, ohne daß diese einen direkten Bezug zur Lage der durch sie repräsentierten Punkte hätten.

- Die beschriebenen Schritte werden rekursiv auf die entstandenen Abkömmlinge angewandt, sofern diese noch mehr als L Punkte enthalten ($L \approx 30, ..., 200$)[4]. Ansonsten wird der Abkömmling ein endständiger Cluster[5] des Baumes. Für einen endständigen Knoten werden die Distanzen vom charakteristischen Punkt zu allen Punkten des Clusters berechnet und gespeichert.

3.3.3. Implementation der Datenstrukturen und Verwaltung der Punkte

3.3.3.1. Verwaltung der Punkte des Datensatzes

Die Punkte eines Datensatzes werden über Indizes von 1 bis N addressiert. Der Algorithmus legt zur Verwaltung der Punkte ein lineares Array A mit Länge N an, das zu jedem Zeitpunkt der Vorverarbeitungsphase eine Permutation der Indizes enthält. Zu Beginn wird A mit 1 ... N initialisiert. Die Indizes der Punkte, die zu gleichen Cluster gehören, sind in einem zusammenhängenden Abschnitt von A zu finden. Dies erlaubt die Anwendung eines *divide and conquer* Schemas bei der rekursiven Einteilung der Punkte in Cluster, bei dem zur weiteren Unterteilung eines Clusters in Abkömmlinge lediglich Permutationen innerhalb des Abschnittes ausgeführt werden, auf dem sich die Indizes der Punkte des aktuellen Clusters befinden (siehe Abbildung 3.2).

[4] Die genaue Wahl dieses Parameters L ist im Hinblick auf die gesamte Rechenzeit des Algorithmus nicht besonders kritisch. Zwar sinkt die durchschnittliche Anzahl der Distanzberechnungen pro Anfrage, doch wird dies durch eine längere Vorverabeitungszeit und einen erhöhten Aufwand für die Durchquerung des Baumes bei der Suche wieder kompensiert. Die Lage des Optimums hängt dabei sowohl von der voraussichtlichen Anzahl der Suchanfragen sowie Anzahl und Verteilung der Daten- und Anfragepunkte ab.

[5] Ein endständiger Knoten wird auch als Blatt bezeichnet.

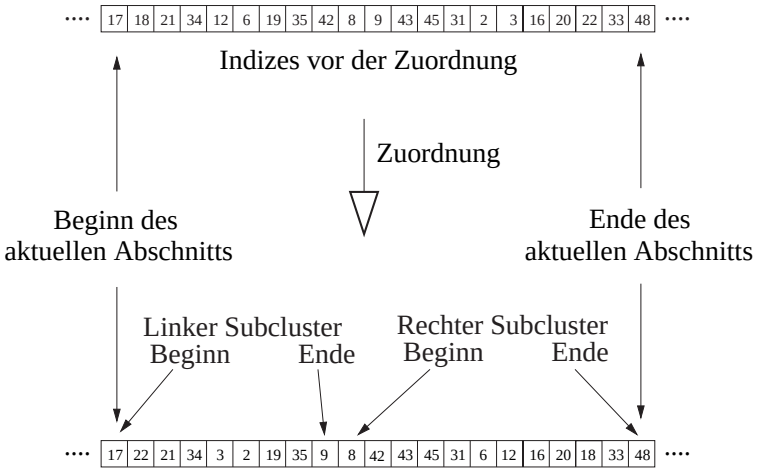


Abbildung 3.2.: Unterteilung des aktuellen Clusters in dessen Abkömmlinge (Subcluster). Die Grenzen des Abschnittes auf A, in dem die Indizes der zum aktuellen Cluster gehörenden Punkte abgelegt sind, bleiben während der Einteilung unverändert. Die Indizes werden nur innerhalb dieses Abschnittes vertauscht, um die zum charakteristischen Punkt jeweils des linken bzw. rechten Abkömmlings gehörenden Punkte voneinander zu separieren.

Diese Permutationen können auf elegante Weise mittels eines Quicksort–ähnlichen [41] Algorithmus vorgenommen werden (siehe Abbildung 3.3).

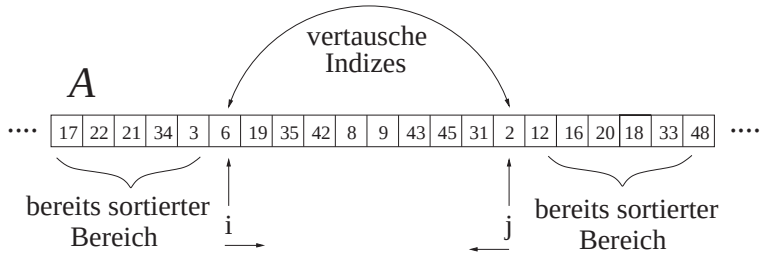


Abbildung 3.3.: Vertauschen der Indizes der Punkte. Diejenigen Punkte, die (geometrisch) näher zum charakteristischen Punkt des linken bzw. rechten Abkömmlings des aktuellen Clusters liegen, werden durch Vertauschen in den linken bzw. rechten Rand des aktuellen Abschnittes verschoben. Während dieses Sortiervorgangs laufen zwei Zeiger, i und j, jeweils vom linken und rechten Rand des Abschnittes aus zur Mitte hin, wobei jeweils Punktpaare ermittelt werden, deren Indizes vertauscht werden müssen. Die Stelle, an der sich i und j überkreuzen, legt den rechten bzw. linken Rand der beiden neu erzeugten Abkömmlinge (Subcluster oder auch Untercluster) fest. Dieses Verfahren hat den Vorteil, daß Separation der Indizes beider Subcluster und Festlegen der neuen Abschnittsgrenzen in einem einzigen Lauf ohne Anlegen eines Zwischenspeichers ausgeführt werden können.

Zusammen mit den Indizes der zu einem endständigen Cluster gehörenden Punkte wird auch deren Distanz zum charakteristischen Punkt des Clusters gespeichert. Auch dazu genügt ein Array der Länge N, dessen Einträge sinnvollerweise in der gleichen Reihenfolge wie die Indizes im Array A abgelegt sind. Es bietet sich daher an, von vornherein ein einziges Array zu verwenden, dessen Einträge Tupel, bestehend aus Index und Distanz, sind. Während der Konstruktion des Baumes läßt sich dieses Array dazu einsetzen, überflüssige Berechnungen bereits bekannter Distanzen zu vermeiden (siehe Quelltext A.4.1).

3.3.3.2. Repräsentation eines Clusters

Zur Repräsentation eines Clusters C wird folgende Datenstruktur verwendet:

c Index des charakteristischen Punkt des Clusters.

R Die maximale Distanz vom charakteristischen Punkt zu einem beliebigen Punkt des Clu-

sters.

$$R = \max_{x \in C} d(c, x)$$

g Während der Zuordnung der Punkte zu einem der beiden Abkömmlinge eines Clusters werden die Distanzen zu beiden charakteristischen Punkten berechnet. Da die Zuordnung zu dem Cluster mit dem näheren charakteristischen Punkt erfolgt, sind für die Punkte eines Abkömmlings die Distanzen zum charakteristischen Punkt seines Geschwisterclusters immer größer oder gleich den Distanzen zum eigenen charakteristischen Punkt. Das Minimum der Differenzen aus beiden Distanzen kann im Zuge der Zuordnung ohne zusätzliche Distanzberechnungen ermittelt und in die Datenstruktur eingetragen werden.

$$g = \min_{x \in C} d(c_{sister}, x) - d(c, x)$$

start Der Beginn des Abschnittes auf A, in dem die Indizes der Punkte dieses Clusters abgelegt sind.

end Die Länge des Abschnittes auf A, in dem die Indizes der Punkte dieses Clusters abgelegt sind.

left, right Verweise (*pointer*) auf den linken und rechten Abkömmling (nicht bei endständigen Knoten).

3.4. Suchphase

3.4.1. Eingrenzung des Suchraums

Das Lokalisieren eines oder mehrerer nächster Nachbarn innerhalb einer endlichen Menge X, $|X| = N$ kann trivialerweise mit einer Komplexität von $O(N)$ durchgeführt werden [6]. Um diese Komplexität zu reduzieren, muß eine Suchstrategie gefunden werden, die nur jene Teile des Suchbaums durchforstet, in denen überhaupt Nachbarn vorkommen könnnen.

Ein Ansatz zur Realisierung einer solchen Suchstrategie erfordert die Bereitstellung einer nach Distanzen geordneten Tabelle der vorläufigen Nachbarn $m_1, \ldots m_k$. Im Verlauf der Suche

[6] Bei der Suche nach mehreren Nachbarn ergibt sich ein zusätzlicher Aufwand für das Verwalten einer geordneten Tabelle der vorläufigen Kandidaten, der aber von dem verwendeten Komplexitätsmaß nicht berücksichtigt wird (siehe 4.3.1) und auch in der Praxis bei im Vergleich zur Datensatzgröße kleinen Nachbarzahlen nicht ins Gewicht fällt.

werden diese vorläufigen Nachbarn solange bei Bedarf durch bessere (d.h. nähere) Kandidaten ersetzt, bis feststeht, daß keine besseren Nachbarn mehr gefunden werden könnnen und der Algorithmus die Tabelle als Ergebnis zurückgeben kann.

Sobald k Kandidaten in die Tabelle eingetragen wurden, ist die Distanz zum k–ten vorläufigen Nachbarn $d(m_k, q)$, die eine obere Schranke für die Distanz des tatsächlichen k–ten Nachbarn bildet, bekannt und kann zur Eingrenzung des Suchraums verwendet werden.

Im Rahmen des vorgestellten Algorithmus findet $d(m_k, q)$ Verwendung in drei Regeln zur Reduzierung des Suchaufwandes:

(1) Schließe Cluster i von der Suche aus, falls

$$d(m_k, q) < \hat{d}_{min}(i), \tag{3.4}$$

wobei $\hat{d}_{min}(i)$ eine untere Schranke für die Distanzen vom Anfragepunkt q zu jedem Punkt im Cluster i darstellt (siehe Abschnitt 3.4.2).

(2) Falls Cluster i ein Blatt des Suchbaumes ist, schließe jeden Punkt x dieses Clusters von der Suche aus, falls

$$d(m_k, q) < |d(c_i, q) - d(c_i, x)|.$$

Die Berechnung der Distanzen $d(c_i, x)$ erfolgte während der Vorverarbeitungsphase (siehe Abschnitt 3.3.3 und Appendix, Beweis 4).

(3) Partielle Distanzberechnung:

Um einen Punkt x als möglichen nächsten Nachbarn zu qualifizieren, muß seine Distanz $d(x, q)$ zum Anfragepunkt kleiner als der momentane Schwellwert $d(m_k, q)$ sein, ansonsten ist der genaue Wert $d(x, q)$ nicht relevant. Dies erlaubt es, die Berechnung von $d(x, q)$ dann abzubrechen, sobald die partielle Distanz (Details variieren mit der Art der verwendeten Metrik) den Schwellwert $d(m_k, q)$ überschreitet [14].

Sorgfältig implementiert, kann partielle Distanzberechnung problemlos in den Algorithmus integriert werden und die Rechenzeit merklich verkürzen, obwohl dies in die hier verwendete Komplexitätsnotation nicht eingeht. Wie in der zuvor erwähnten Regel wird die partielle Distanzberechnung nur während des Durchsuchens eines endständigen Knotens des Baumes angewendet.

3.4.2. Berechnung von $\hat{d}_{min}$

Um während einer Suche festzustellen, ob der Inhalt eines Cluster C_i durchsucht werden muß, benötigt der Algorithmus eine Abschätzung $\hat{d}_{min}(i)$ für die (unbekannte) minimale Distanz $d_{min}(i) = \min_{x \in C_i} d(x, q)$ vom Anfragepunkt zu einem beliebigen Punkt aus C_i so daß $0 \leq \hat{d}_{min}(i) \leq d_{min}(i)$.

Drei untere Schranken für d_{min} können aus der während der Vorverarbeitungsphase angelegten Datenstruktur und Informationen, die im bisherigen Verlauf der Suche angefallen sind, berechnet werden:

(1) $d(c_i, q) - R_i$
Hier bezeichnet $d(c_i, q)$ die Distanz zwischen dem Anfragepunkt q und dem charakteristischen Punkt c von C_i, während R_i dessen Radius angibt (siehe Appendix, Beweis 2).

(2) $\frac{1}{2}(d(c_i, q) - d(q, c_{sister}) + g_i)$
wobei $d(q, c_{sister})$ die Distanz vom Anfragepunkt zum charakteristischen Punkt vom Geschwistercluster zu C_i darstellt (siehe Appendix, Beweis 3).

(3) Aus der rekursiven Einteilung des Datensatzes in Cluster folgt, daß das Minimum d_{min} eines Clusters C_i nicht kleiner sein kann als die Abschätzung $\hat{d}_{min}$ seines Elternclusters.

Da alle drei Werte untere Schranken für den tatsächliche Minimalabstand darstellen, wird ihr Maximum als Schätzwert $\hat{d}_{min}(i)$ verwendet.

Es kann kaum überbetont werden, daß eine genaue Abschätzung von d_{min} eines der Kernprobleme dieses und anderer Nachbar-Algorithmen darstellt. Hätte man eine Berechnungsvorschrift zur exakten Bestimmung von d_{min}, könnten selektiv nur jene terminalen Cluster des Baumes ausgewählt werden, die tatsächlich auch einen oder mehrere nächste Nachbarn enthielten (insgesamt maximal k Stück). Unglücklicherweise scheint es, daß es nicht möglich ist, d_{min} exakt zu berechnen, ohne explizit alle Distanzen $d(x, q)\, \forall x$ eines gegebenen Clusters C auszurechnen. Dann allerdings ist d_{min} wertlos, da der Rechenaufwand, den es eigentlich zu vermeiden galt, bereits aufgebracht wurde.

3.4.3. Prioritätswarteschlangen kontrollierte Suchreihenfolge

Die Reihenfolge, in der die Knoten des Baumes durchsucht werden, spielt eine entscheidende Rolle bei der Minimierung der Anzahl der Distanzberechnungen. Um den Suchraum effektiv gemäß Gleichung 3.4 eingrenzen zu können, ist es zusätzlich zur möglichst guten Abschätzung von d_{min} notwendig, auch möglichst früh im Verlauf der Suche dem späteren Endwert von $d(m_k, q)$ (vgl. 3.4.1) nahe zu kommen. Dazu wird im ATRIA statt einer Tiefensuche (im Engl. *depth first*) eine Prioritätswarteschlange benutzt (vgl. auch Berchtold [9]). Die Prioritätswarteschlange [41] enthält als Einträge die noch zu durchsuchenden Cluster, geordnet nach $\hat{d}_{min}$, und ermöglicht, jeweils den Eintrag mit dem aktuell kleinsten $\hat{d}_{min}$ effizient zu extrahieren. Dies bringt mehrere Vorteile mit sich:

(1) Cluster mit kleinem $\hat{d}_{min}$ enthalten mit hoher Trefferrate nächste Nachbarn. Schon nach vergleichsweise wenigen durchsuchten Clustern hat $d(m_k, q)$ fast den endgültigen Wert erreicht (vgl. Abbildung 3.4).

(2) Cluster mit hohem $\hat{d}_{min}$ werden, wenn überhaupt, erst sehr spät durchsucht.

3.4.4. Bearbeitung einer Suchanfrage

Die drei folgenden Abschnitte skizzieren den Ablauf einer Suche sowie die Abbruchbedingungen für exakte und approximative Suchanfragen. Dazu wird zuerst die Kernschleife des Suchalgorithmus dargestellt:

(1) Initialisiere eine leere Tabelle der vorläufigen Nachbarn.

(2) Füge den Rootcluster in die Prioritätswarteschlange (PW) ein.

(3) Iteriere:

- Extrahiere den Clustern C mit kleinstem $\hat{d}_{min}$ aus der PW.
- Falls C ein endständiger Knoten ist:
  - Berechne unter Berücksichtigung von Regel 2 explizit die Distanzen zu allen Punkten des Clusters.

– Falls Distanzen kleiner $d(m_k, q)$ gefunden werden, ist die Tabelle der vorläufigen Nachbarn entsprechend zu aktualisieren.

Ansonsten füge die Abkömmlinge von C in die PW ein.

3.4.4.1. Exakte Suchen

Sobald alle Cluster, die folgende Bedingung

$$\hat{d}_{min} \leq d(m_k, q) \tag{3.5}$$

erfüllen, durchsucht wurden, kann kein Punkt gefunden werden, der näher am Anfragepunkt q liegt als der momentane k–te vorläufige Nachbar m_k. Damit steht fest, daß die in der Tabelle vorhandenen vorläufigen Nachbarn die gesuchten exakten sind. Der Algorithmus kann die Hauptschleife abbrechen und die Tabelle als Resultat der Suche zurückliefern.

Exemplarisch wird dies in Abbildung 3.4 illustriert, in der der dynamische Verlauf sowohl der Distanz $d(m_k, q)$ zum k–ten vorläufigen Nachbarn als auch der der Abschätzung $\hat{d}_{min}$ der Mindestentfernung zu den Punkten des jeweiligen Clusters dargestellt ist. Die Suche terminiert sobald $d(m_k, q)$ unter $\hat{d}_{min}$ fällt.

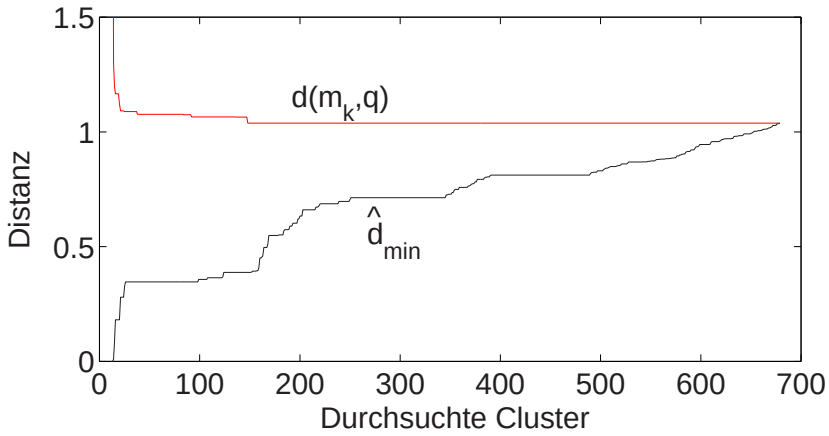


Abbildung 3.4.: Typischer Verlauf von $d(m_k, q)$ und $\hat{d}_{min}$ für eine exakte Suche. Der Baum bestand aus mehr als 2300 Knoten (Cluster). Die Abbruchbedingung wird erreicht, sobald $d(m_k, q)$ unter $\hat{d}_{min}$ fällt. Der lange flache Teil des $d(m_k, q)$ Graphen zeigt, daß nach dem 147. Cluster keine weiteren Nachbarn mehr gefunden werden, obwohl das Abbruchkriterium erst beim 679. Cluster erreicht wird. Dies ist eine Konsequenz der Unfähigkeit, d_{min} exakt abzuschätzen.

3.4.4.2. Approximative Suchen

Approximative Suchen stellen eine Abschwächung des exakten Suchproblems dar. Statt vom Algorithmus zu fordern, die exakten Nächsten–Nachbarn eines Anfragepunktes q zu liefern, gibt man sich damit zufrieden, daß der Algorithmus Nachbarn zurückliefert, die im Hinblick auf ihre Distanz zu q einen vorgegebenen relativen Fehler ϵ gegenüber den exakten Nachbarn nicht überschreiten [5] (siehe Def. 3.3). Für $\epsilon \to 0$ geht eine approximative in die entsprechende exakte Suche über.

Im ATRIA werden approximative Suchen durch die vorzeitige Beendigung der Hauptschleife gemäß folgendem Abbruchkriterium realisiert:

$$\hat{d}_{min} > \frac{d(m_k, q)}{1+\epsilon}. \tag{3.6}$$

Dieses Kriterium hat zur Folge, daß nicht alle Cluster durchsucht werden, die exakte nächste Nachbarn enthalten könnten, wodurch der Suchaufwand reduziert wird.

Folgender Beweis zeigt, daß der Algorithmus bei Verwendung dieses Kriteriums ϵ–approximative Nachbarn liefert:

Beweis 1 *Seien $n_i^0 \in X \quad i = 1, \ldots, k$ die k exakten Nächsten–Nachbarn zum Anfragepunkt q so daß:*

$$0 \leq d(n_i^0, q) \leq d(n_{i+1}^0, q) \quad i = 1, \ldots, k-1$$
$$d(n_k^0, q) \leq d(x, q)\, \forall x \in X \backslash \{n_1^0, \ldots, n_k^0\}$$

Daher gilt für jede nach Distanzen geordnete Auswahl an vorläufigen Nachbarn $d(m_i, q)$:

$$d(n_i^0, q) \leq d(m_i, q) \quad i = 1, \ldots, k$$

Wenn der Suchalgorithmus bei Erreichen folgenden Kriteriums

$$\hat{d}_{min} > \frac{d(m_k, q)}{1+\epsilon}$$

terminiert, wissen wir, daß alle vorläufigen Nachbarn mit $d(m_i, q) \leq d(m_k, q)/(1+\epsilon)$ sogar exakte sind und es auch keine weiteren exakten Nachbarn geben kann, deren Abstand zu q kleiner als dieser Schwellwert ist, da alle in Frage kommenden Cluster erschöpfend durchsucht

wurden. Sei j der höchste Index, für den obige Bedingung zutrifft, anderenfalls setzen wir j zu 0. Für $i = 1, \ldots, j$ gilt $n_i^0 = m_i$ und damit ist Gleichung 3.6 für diese trivialerweise erfüllt. Weiter ist aus der Anordnung der vorläufigen Nachbarn bekannt daß:

$$\frac{d(m_k, q)}{1+\epsilon} \leq d(m_{j+1}, q) \leq d(m_{j+2}, q) \leq \cdots \leq d(m_k, q) \tag{3.7}$$

Aus dem Argument des erschöpfenden Durchsuchens und der Anordnung der exakten Nächsten–Nachbarn folgt analog:

$$\begin{aligned} &\frac{d(m_k, q)}{1+\epsilon} \leq d(n_{j+1}^0, q) \leq d(n_{j+2}^0, q) \leq \cdots \leq d(m_k, q) \\ &\Rightarrow d(m_k, q) \leq (1+\epsilon) d(n_{j+1}^0, q) \leq (1+\epsilon) d(n_{j+2}^0, q) \end{aligned} \tag{3.8}$$

Kombiniert man 3.7 und 3.8, so erhält man:

$$d(m_{j+1}, q) \leq d(m_{j+2}, q) \leq \cdots \leq d(m_k, q) \leq (1+\epsilon) d(n_{j+1}^0, q)$$

Damit erfüllen die vorläufigen Nachbarn auch für $i = j+1, \ldots, k$ die Bedingung 3.3 für approximative Nachbarn.

3.4.4.3. Bereichssuche

Auch eine weitere Variante des Nächste–Nachbarn Problems, die sog. Bereichssuche (im Englischen *range search*), läßt sich ohne aufwendige Modifikationen mit dem beschriebenen Algorithmus behandeln. Bei der Problemstellung der Bereichssuche wird nicht nach einer festen Anzahl Nachbarn zu einem Anfragepunkt q gesucht (*fixed mass approach*), statt dessen sollen alle Punkte x mit Abstand $d(x, q) \leq r$ (*fixed size approach*) als Ergebnis zurückgeliefert werden (siehe 4.2.2).

Folgende Modifikationen sind erforderlich:

- Das dynamische Abbruchkriterium 3.5 kann durch folgendes statisches Kriterium ersetzt werden:
$$\hat{d}_{min} \leq r \tag{3.9}$$
- Durch die Verwendung des statischen Kriteriums bringt es keine Vorteile, die Cluster geordnet nach $\hat{d}_{min}$ zu durchsuchen. Daher kann die Prioritätswarteschlange durch einen einfachen Stapel (*stack*) mit geringerem Verwaltungsaufwand ersetzt werden.

3.5. Verringerung der Rechenzeit bei approximativen Suchen mit verschiedenem ϵ

Zur Bestimmung der Beschleunigung der Suche und der damit einhergehenden Fehler in den Distanzen der gefundenen Nachbarn bei der approximativen Suche wurden Suchläufe mit zunehmendem ϵ auf Datensätzen mit je 50000 Punkten durchgeführt. Die Datensätze wurden durch Iteration einer D–dimensionalen Verallgemeinerung der Hénon Abbildung [6] generiert

$$(x_1)_{n+1} = a - (x_{D-1})_n^2 - b(x_D)_n$$
$$(x_i)_{n+1} = (x_{i-1})_n \quad i = 2, \ldots, D$$

mit $a = 1.76$, $b = 0.1$ sowie $D = 8$. Ausgehend von zufälligen Anfangswerten im Intervall $[0 \ldots 0.01]$, wurden die ersten 5000 Iterationen verworfen und die nachfolgenden 200000 als eigentlicher Datensatz genutzt. Die Aufgabenstellung jedes Suchlaufs war es, acht Nächste–Nachbarn zu je 10000 zufällig aus dem Datensatz gewählten Anfragepunkten zu finden, wobei Selbsttreffer ausgeschlossen wurden.

Während die durchgezogene Linie in Abbildung 3.5 die mittlere Beschleunigung der approximativen gegenüber der exakten Suche für zunehmendes ϵ wiedergibt[7], stellt die gestrichelte Linie den relativen Fehler in den Distanzen der gefundenen Nachbarn dar, gemittelt über alle Nachbarn und Anfragepunkte. Die Beschleunigung ergibt sich dabei aus dem Verhältnis der zur Bestimmung der exakten Nachbarn benötigten Zeit zu der Ausführungzeit der entsprechenden approximativen Suche[8]. Die gepunktete Linie zeigt den maximalen relativen Fehler der Nachbardistanzen.

Der mittlere relative Fehler in den Nachbardistanzen überschreitet 10 Prozent selbst für die hohe Wahl von $\epsilon = 7$ nicht. Dies ist vergleichsweise wenig, erlaubt doch $\epsilon = 7$ einen relativen Fehler von bis zu 700 Prozent. Wieder scheint diese Diskrepanz eine Folge des Unvermögens zu sein, d_{min} exakt zu schätzen.

[7] Für $\epsilon \to 0$ geht die approximative Suche in die entsprechende exakte über.

[8] Da die Vorverarbeitung unabhängig von der Art der Suchanfrage ist, mußte sie nur einmal für den verwendeten Datensatz ausgeführt werden. Diese Zeit ging nicht in die dargestellten Messungen ein.

3.6. Anwendung von Kernfunktionen zur Berechnung von Distanzen in hochdimensionalen Merkmalsräumen

Der vorgestellte Algorithmus zur Berechnung der Nächsten–Nachbarn bezieht Informationen über die Geometrie des Datensatzes sowohl während der Vorverabeitungs- als auch während der Suchphase ausschließlich aus den im Datensatz auftretenden Interpunkt–Distanzen sowie aus Distanzen zwischen dem Anfragepunkt q und ausgewählten Punkten des Datensatzes. Dies erlaubt die Anwendung des Algorithmus auch auf Problemstellungen, die über die gewöhnliche Darstellung von Punkten als Vektoren des $\mathbb{R}^d$ und L_∞ bzw. L_2 Metrik hinausgehen. Beispiele solcher weitergehenden Problemstellungen sind u.a. die Suche in einer Datenbank von Automodellen oder die Bestimmung von Nachbarn in einer Menge von Strings variabler Länge [9].

Eine andere Möglichkeit, die sich im Zusammenhang damit bietet, ist es, Vektoren aus einem niedrigdimensionalen Ausgangsraum I (im Englischen *input space*) über eine (nichtlineare) Funktion Φ in einen höherdimensionalen Merkmalsraum F (im Englischen *feature space*) abzubilden und Nachbarn in diesem Raum zu suchen. Sinnvollerweise beschränkt man sich dabei auf Abbildungen Φ, die es erlauben, das Skalarprodukt zweier Bilder $\mathbf{X} \cdot \mathbf{Y} = \Phi(\mathbf{x}) \cdot \Phi(\mathbf{y})$ über eine sog. Kernfunktion $k(\mathbf{x}, \mathbf{y})$ der Urbilder $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ effizient auszurechnen:

$$\begin{array}{rccl} \Phi: & I & \longrightarrow & F \\ & \mathbf{x} & \longrightarrow & \Phi(\mathbf{x}) = \mathbf{X} \end{array}$$

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}) = k(\mathbf{x}, \mathbf{y}) \tag{3.10}$$

Zur Distanzberechnung in F wird die gewöhnliche euklidische Metrik verwendet:

$$\begin{aligned} d(\mathbf{x}, \mathbf{y}) &= ((\mathbf{X} - \mathbf{Y}) \cdot (\mathbf{X} - \mathbf{Y}))^{\frac{1}{2}} \\ &= (\mathbf{X} \cdot \mathbf{X} - 2\mathbf{X} \cdot \mathbf{Y} + \mathbf{Y} \cdot \mathbf{Y})^{\frac{1}{2}} \\ &= (\Phi(\mathbf{x}) \cdot \Phi(\mathbf{x}) - 2\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}) + \Phi(\mathbf{y}) \cdot \Phi(\mathbf{y}))^{\frac{1}{2}} \\ &= (k(\mathbf{x}, \mathbf{x}) - 2k(\mathbf{x}, \mathbf{y}) + k(\mathbf{y}, \mathbf{y}))^{\frac{1}{2}} \end{aligned} \tag{3.11}$$

[9] Zur Distanzberechnung zwischen Strings ungleicher Länge kann dabei beispielsweise die Editier–Distanz eingesetzt werden. Ähnlich der Berechnung der Hamming–Distanz, wird dabei die Anzahl elementarer Editier–Operationen (Löschen, Austauschen und Einfügen), evtl. gewichtet, aufsummiert.

Eine mögliche Kernfunktion, die dem Skalarprodukt im Raum aller Monome aus den einzelnen Komponenten der Inputvektoren entspricht, soll hier als Beispiel erwähnt werden:

$$k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^d = \left(\sum_{i=1}^{d} x_i y_i + 1 \right)^d \tag{3.12}$$

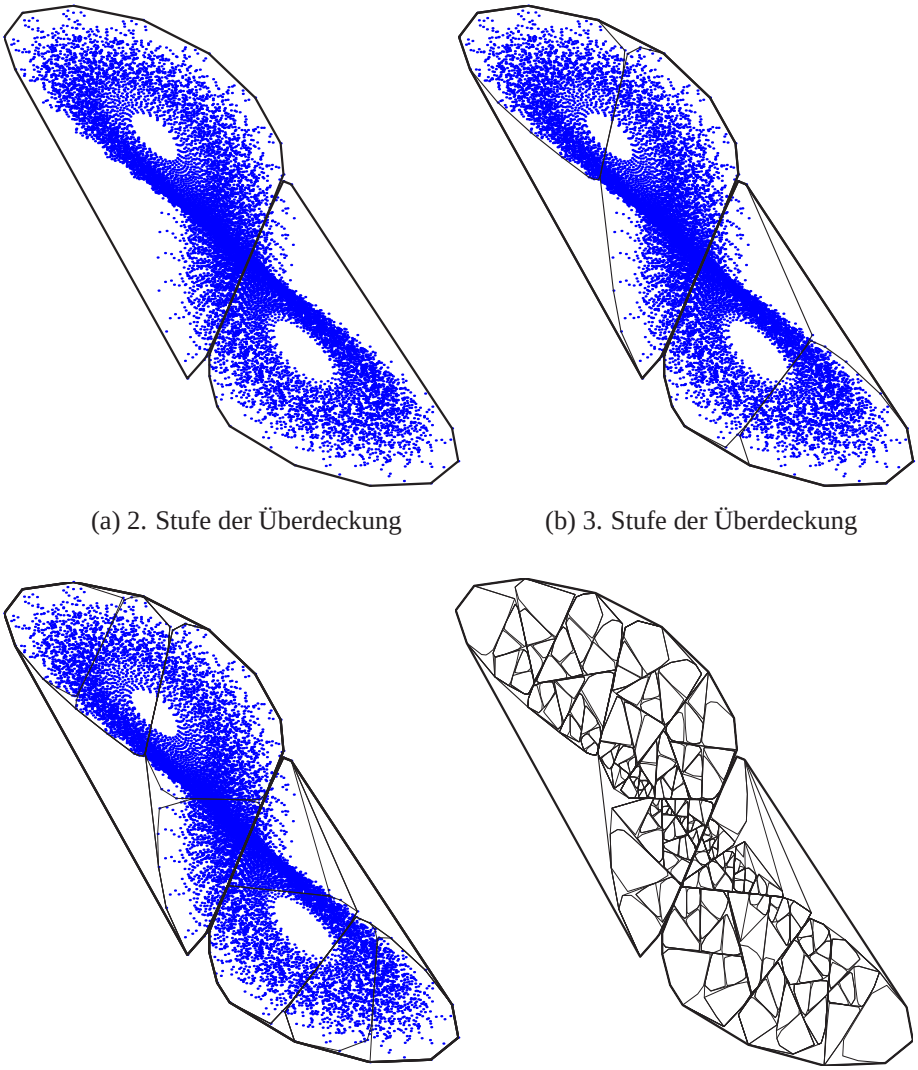
Der Zusammenhang zwischen möglichen Kernfunktion, Abbildung Φ und Merkmalsraum wird durch das Theorem von Mercer gegeben [48].

3.7. Zusammenfassung

Der vorgestellte effiziente Algorithmus zur Nächste–Nachbar–Suche zeichnet sich sowohl durch die Möglichkeit der flexiblen Wahl der Metrik als auch durch eine gute Anpassung der während der Vorverabeitungsphase angelegten Baumstruktur an die Geometrie des Datensatzes aus. Letztere Eigenschaft erlaubt besonders dann eine schnelle Bearbeitung der Suchanfrage, wenn die Punkte eines Datensatzes in einem niedrigdimensionalen, aber nicht notwendigerweise linearenen Unterraum des Datenraumes liegen. Nicht–adaptive Verfahren wie die Box–Assisted Nächste–Nachbar–Suche [39] sind hier im Nachteil, der allerdings bei einfachen Problemstellungen durch die schnellere Vorverabeitung dieser Verfahren wieder kompensiert werden kann. Vergleiche der Laufzeit des ATRIA mit anderen Algorithmen zur Nachbarsuche finden sich in [29].

Die Möglichkeit, statt der exakten approximative Nächste–Nachbarn zu suchen, bietet den Vorteil einer zusätzlichen deutlichen Verringerung der Rechenzeit. Allerdings findet diese Option in der nichtlinearen Zeitreihenanalyse wenig Verwendungsmöglichkeit, interessieren hier doch meist nur die exakten Nächsten–Nachbarn.

Der Einfluß der Verteilung der Punkte des Datensatzes auf das Laufzeitverhalten von Algorithmen zur Nächsten–Nachbar–Suche wurde bislang bis auf wenige Ausnahmen (z.B. [8]) nur ungenügend untersucht. Wichtig in diesem Zusammenhang ist die Verwendung geeigneter Größen zur Charakterisierung der Verteilung der Punkte. Dazu werden im folgenden Kapitel verschiedene Dimensionsbegriffe und Methoden zu deren numerischen Berechnung vorgestellt.



(a) 2. Stufe der Überdeckung (b) 3. Stufe der Überdeckung

(c) 4. Stufe der Überdeckung (d) Vollständige Zerlegung für $L = 128$

Abbildung 3.1.: In der Vorverarbeitungsphase angelegte Segmentierung eines Datensatzes, bestehend aus 15000 Punkten, der durch Projektion der Trajektorie eines Lorenz–Systems auf 2 Dimensionen erzeugt wurde. Eingezeichnet sind jeweils die Polygone der konvexen Hülle der Punkte eines Clusters. Die Linienstärke nimmt mit zunehmender Tiefe des Baumes ab. Deutlich zu erkennen sind die beiden direkten Abkömmlinge des Rootclusters, der selbst nicht eingezeichnet ist. Endständige Cluster sind daran zu erkennen, daß sie keine weiteren Polygone enthalten. In der tiefsten Stufe der Segmentierung (Bild d) mit Parameter $L = 128$ befinden sich im Mittel rund 79 Punkte in jedem der 191 endständigen Cluster, minimal sind es 5, maximal 128, was die Obergrenze der Punktzahl darstellt, bei der ein Cluster nicht weiter unterteilt wird.

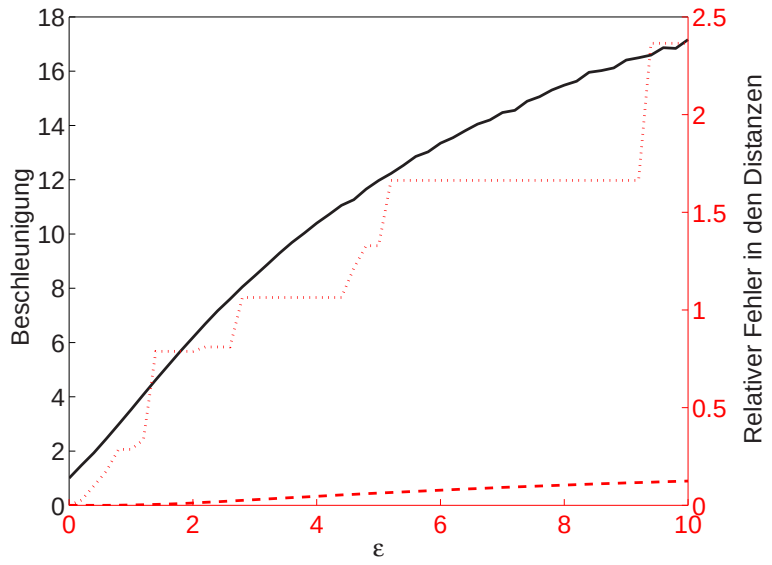


Abbildung 3.5.: Verringerung der Suchzeit und des relativen Fehlers in den Nachbardistanzen, aufgetragen gegen den Toleranzparameter ϵ für einen Datensatz, der von einer $D = 8$–dimensionalen Verallgemeinerung der Hénon Abbildung erzeugt wurde mit Informationsdimension $D_1 \approx 5.9$. Die durchgezogene Linie gibt das Verhältnis der Suchzeiten exakt zu approximativ an. Die gestrichelte Linie zeigt den durchschnittlichen relativen Fehler in den Distanzen an. Der maximal aufgetretene Fehler ist als gepunktete Linie dargestellt.

4. Dimensionen

4.1. Begriffsbildungen

4.1.1. Verallgemeinerte Dimensionen

Der klassische Dimensionsbegriff hat seinen Ursprung in der euklidischen Geometrie und deren Weiterentwicklung zur analytischen Geometrie mit der Einführung eines Koordinatensystems durch René Descartes (1596-1650). Die erste Verallgemeinerung der auf diese Weise geschaffenen Objekte führt zum Begriff des Vektorraums. Unter der Dimension D eines Vektorraums versteht man die Mächtigkeit seiner Basis.

Die von Brouwer entdeckte Invarianz der Dimension unter homöomorphen Abbildungen erlaubte eine Verallgemeinerung des Dimensionsbegriffes für beliebige, zum euklidischen Raum lokal homöomorphe Mengen. Um jedoch die zerklüftete (fraktale) Struktur bestimmter Attraktoren quantitativ zu beschreiben, wurden dem klassischen Dimensionsbegriff Konzepte verallgemeinerter Dimensionen beiseitegestellt, die invariant gegenüber stetig differenzierbaren, invertierbaren Koordinatentransformationen sind.

4.1.2. Das invariante Maß

Ein wichtiges Hilfsmittel bei der Untersuchung chaotischer Systeme ist das *invariante natürliche Maß*. Wenn langfristige Vorhersagen des Systemverhaltens nicht mehr möglich sind, kann man mit Hilfe des invarianten Maßes immerhin statistische Aussagen über den möglichen Zustand des Systems machen. Allerdings kann das invariante Maß selten explizit angegeben werden, und man ist auf Näherungsverfahren angewiesen [19, 31].

Wenn nach dem Abklingen der Einschwingvorgänge die Bewegung des Systems im Pha-

senraum auf dem Attraktor A verläuft, gibt es ein Wahrscheinlichkeitsmaß μ auf dem d-dimensionalen Borelschen Meßraum $(\mathbb{R}^d, \mathcal{B}^d)$ mit den Eigenschaften

$$\mu(A) = 1 \tag{4.1}$$

$$\forall B \subseteq A;\ \forall t > 0\ : \mu(B) = \mu(\varphi(B, -t)). \tag{4.2}$$

Ein solches Maß heißt *φ-invariantes Maß*, und die Existenz folgt aus der Kompaktheit des Trägers und der Stetigkeit des Flusses [49].

Da es für ein System mehrere invariante Maße geben kann, ist es üblich, das von Kolmogorov vorgeschlagene *natürliche invariante* Maß zu betrachten. Dieses ist das letzte invariante Maß, das übrig bleibt, wenn das System gestört wird [16].

Ein invariantes Maß heißt *ergodisch*, wenn sich der Phasenraum des Systems nicht in zwei disjunkte, meßbare und jeweils flußinvariante Teilmengen mit positivem Maß zerlegen läßt [3]. Systeme mit einem ergodischen Maß werden *ergodische Systeme* genannt. In ergodischen Systemen kann man das Zeitmittel einer Funktion durch das Phasenraummittel ersetzen.
Sei $\mathrm{g} : \mathbb{R}^{\mathrm{n}} \to \mathbb{R}$ eine μ-meßbare Funktion. Dann wird das *Zeitmittel* von g definiert durch

$$\langle \mathrm{g} \rangle_{\mathrm{t}} = \lim_{\mathrm{T}\to\infty} \frac{1}{\mathrm{T}} \int_0^{\mathrm{T}} \mathrm{g}(\varphi(\mathbf{y}_0, \mathrm{t}))\, \mathrm{dt} \tag{4.3}$$

und das *Phasenraummittel* durch

$$\langle \mathrm{g} \rangle_{\mu} = \int_{\mathrm{A}} \mathrm{g}(\mathbf{y})\, \mathrm{d}\mu(\mathbf{y}). \tag{4.4}$$

In einem ergodischen System gilt dann für fast alle Startwerte $\mathbf{y}_0 \in A$ (d.h. bis auf eine Nullmenge bzgl. μ)

$$\langle \mathrm{g} \rangle_{\mathrm{t}} = \langle \mathrm{g} \rangle_{\mu}. \tag{4.5}$$

Für einige diskrete dynamische Systeme, wie z.B. die Zeltabbildung [40], läßt sich Ergodizität explizit nachweisen. In den meisten Fällen bleibt die Ergodizität des Systems jedoch eine unbewiesene Hypothese.

4.1.3. Die Hausdorffdimension

Hausdorff untersuchte im Jahre 1919 die Cantormenge im Hinblick auf das 1914 von Carathéodory eingeführte äußere Maß und fand für diese Menge einen nicht-ganzzahligen Wert des äußeren Maßes [17].

Seitdem ist Hausdorffs Name mit diesem Maß und dem weiterführenden Dimensionsbegriff verbunden, der später von Mandelbrot als fraktale Dimension bezeichnet wurde [26]. Als „fraktale Mengen“ (auch kurz „Fraktale“) bezeichnet Mandelbrot dabei solche Mengen mit gebrochener Hausdorffdimension.

Für eine Menge $U \subseteq \mathbb{R}^d$ mit $U \neq \emptyset$ ist der Durchmesser von U definiert als

$$d(U) := \sup\{\|x - y\| \text{ mit } x, y \in U\}.$$

Sei $A \subset \bigcup_i U_i$ und $0 < d(U_i) \leq r$ für alle i, dann heißt $\{U_i\}_i$ r-Überdeckung von A.

Als äußeres α-Hausdorffmaß einer Menge $A \subseteq \mathbb{R}^d$, $\alpha \geq 0$ bezeichnet man nun den Grenzwert

$$H^\alpha(A) := \lim_{r \to 0} \inf \left\{ \sum_i d(U_i)^\alpha \mid d(U_i) \leq r \text{ und } \{U_i\}_i \text{ } r\text{-Überdeckung von } A \right\}$$

und als *Hausdorffdimension* von A den eindeutigen Wert

$$D_H(A) := \inf\{\alpha | H^\alpha(A) = 0\} = \sup\{\alpha | H^\alpha(A) = \infty\} \ .$$

4.1.4. Die Kapazitätsdimension

Sei $N(r, A)$ die Anzahl der offenen Kugeln mit dem Radius r, die benötigt wird, um die Menge A zu überdecken. Die *Kapazitätsdimension* $\dim_K(A)$ wird dann definiert durch

$$D_K(A) \quad := \quad \lim_{r \to 0} \sup \frac{\log(N(r, A))}{\log(1/r)}. \tag{4.6}$$

Für die klassischen Objekte der euklidischen Geometrie (z.B. die platonischen Körper) nehmen Hausdorffdimension und Kapazitätsdimension nur ganzzahlige Werte an, d.h., sie stimmen mit der oben erwähnten klassischen Dimension überein [17].

4.1.5. Die Rényi–Dimensionen

Dieser Dimensionsbegriff wurde von Rényi in die Informationstheorie eingeführt [35].
Sei $\{U_i\}_i$ eine r-Überdeckung von A und $p_i = \mu(U_i)$ die Aufenthaltswahrscheinlichkeit des

Systems in der Menge U_i des Attraktors A. Dann definiert man daß Maß q-ter Ordnung der Information als

$$I_q(r,\mu) \;:=\; \frac{1}{1-q}\log_2(\sum_i p_i^q) \qquad q \in \mathbb{R} \tag{4.7}$$

und die verallgemeinerte *Rényi–Dimension q-ter Ordnung* als den Grenzwert

$$\begin{aligned} D_q(A,\mu) \;&:=\; \lim_{r\to 0}\frac{I_q(r,\mu)}{\log_2(1/r)} \\ &=\; \lim_{r\to 0}\frac{1}{q-1}\frac{\log_2(\sum p_i^q)}{\log_2 r}. \end{aligned} \tag{4.8}$$

Die Verwendung des Logarithmus zur Basis 2 anstelle des natürlichen Logarithmus ist in der Informationstheorie üblich. Die verallgemeinerte Rényi-Dimension q-ter Ordnung ist jedoch von der Wahl der Basis des Logarithmus unabhängig. Der Grenzübergang $r \to 0$ dient dazu, das Skalierungsverhalten der Größe $\log_2(\sum p_i^q)$ gegen $\log_2 r$ auf solchen Skalen zu ermitteln, bei denen die Grobstruktur des Attraktors keinen Einfluß mehr auf die p_i hat. Numerisch ist dieser Grenzübergang jedoch aufgrund der finiten Größe der Datensätze nicht möglich, daher wird in der Praxis die Größe

$$\frac{1}{q-1}\frac{d\left(\log_2(\sum p_i^q)\right)}{d\left(\log_2 r\right)}$$

aus der Steigung der doppeltlogarithmischen Auftragung von $\sum p_i^q$ gegen r in einem Bereich linearer Skalierung bestimmt.

Im Grenzfall $q \to 0$ erhält man die Kapazitätsdimension. Für $q \to 1$ entspricht (4.7) der Shannon-Information [42] $I_1(r,\mu) = -\sum p_i \log_2(p_i)$, daher nennt man die zugehörige verallgemeinerte Dimension auch *Informationsdimension*[25]. Darüberhinaus wird D_q für den Fall $q = 2$ oft mit der *Korrelationsdimension* gleichgesetzt (vgl. Abschnitt 4.2.2), obwohl die Gleichheit beider Größen nicht im strengen Sinne bewiesen ist.

Allgemein erfüllt das Spektrum der Rényi-Dimensionen folgendende Ungleichung

$$D_q(A,\mu) \le D_p(A,\mu) \qquad \text{für} \quad q > p\;. \tag{4.9}$$

4.2. Numerische Berechnung der verallgemeinerten Dimensionen

Zur numerischen Berechnung der verallgemeinerten Dimensionen wurden in den vergangenen Jahren eine Vielzahl unterschiedlicher Ansätze und Verfahrensweisen, die teilweise aufeinander aufbauen, vorgeschlagen. Diese lassen sich in mehrere Klassen einteilen:

(1) Ansätze basierend auf der Einteilung des Phasenraums in Hyberkuben und Auszählung der in diese fallenden Punkte. Im Englischen werden diese Verfahren auch als *box–counting* Verfahren bezeichnet.

(2) Korrelationssummenbasierte Verfahren. Die Korrelationssumme $C(r)$ zählt die relative Anzahl Punktpaare des Datensatzes innerhalb eines gegebenen Abstandes r. Aus der doppeltlogarithmischen Auftragung von $C(r)$ gegen den Abstand r läßt sich D_2 näherungsweise bestimmen.

(3) Verfahren, die auf den Momenten der Nächsten–Nachbar–Distanzen innerhalb des Datensatzes beruhen. Diese erlauben es, das Spektrum D_q in einem relativen großen Bereich von q zu bestimmen.

Diese Klassen sollen in den folgenden Abschnitten im Einzelnen näher erläutert werden.

4.2.1. Box–Counting basierte Ansätze

Bei den Box–Counting basierten Ansätzen wird, in Anlehnung an die Definition der Rényi–Information (4.7), der Phasenraum in Hyberkuben eingeteilt und die relative Häufigkeit p_i durch Auszählen der in die i-te Box fallenden Punkte bestimmt. Aus dem Skalierungsverhalten der auf diese Weise gewonnenen Schätzung der Rényi–Information q-ter Ordnung gegen die Abmessungen r (z.B. Kantenlänge) der Partitionen in logarithmischer Auftragung läßt sich D_q näherungsweise bestimmen.

Eine naive Implementation der Box–Counting basierten Verfahren, bei der ein D dimensionales Array im Speicher angelegt wird, wobei jede Zelle des Arrays einer Partition bzw. einem Hyperkubus des Phasenraums entspricht, scheitert sehr schnell bei Zunahme der Anzahl b der Unterteilungen pro Dimension aufgrund des enormen Speicheraufwandes $O(b^D)$.

Ein sehr effizienter Algorithmus basiert auf der Datenstruktur des *ternären Suchbaumes*, die ursprünglich von R. Sedgewick [41] zur schnellen Sortierung von Strings variabler Länge konzipiert wurde. Der ternäre Baum erweitert den binären Baum, mit dem man effizient eindimensionale Punkte (also Skalare) sortieren kann, dahingehend, daß jeder Knoten dieses Baumes neben den Zeigern auf den Abkömmling mit kleinerem Schlüsselwert bzw. Abkömmling mit größerem Schlüsselwert noch einen dritten Abkömmling hat, der die Wurzel eines Unterbaumes darstellt, dessen Schlüsselwerte aus der jeweils nächsten Koordinate der zu kodierenden Punkte gewonnen werden. Die Schlüsselwerte des ersten Baumes werden durch Einteilung der jeweils ersten Koordinate der zu sortierenden Vektoren in diskrete Unterteilungen $1, \ldots, b$ erzeugt, entsprechendes gilt für die weiteren Koordinaten (siehe Abbildung 4.1).

Ein Knoten eines ternären Baumes enthält folgende Einträge:

level Dimension ($\in 1, \ldots, D$), aus deren Koordinate der aktuelle Schlüsselwert gewonnen wurde.

key Aktueller Schlüsselwert dieses Knotens. Dieser wird durch Einteilung der **level**–ten Koordinate des Punktes auf die diskrete Unterteilung ($\in 1, \ldots, b$) gewonnen.

lokid Verweis auf den Abkömmling mit kleinerem Schlüsselwert als **key**.

hikid Verweis auf den Abkömmling mit größerem Schlüsselwert als **key**.

eqkid Verweis auf den Wurzelknoten des Unterbaumes, dessen Schlüsselwerte aus der Dimension **level**$+1$ gewonnen werden.

count Anzahl der Punkte des Datensatzes, die von diesem Knoten kodiert werden.

Knoten mit **level** $= D$ enthalten zwar keine **eqkid**–Verweise, deren **count** Eintrag entspricht jedoch der Anzahl der Punkte des Datensatzes, die in die durch die Schlüsselwerte des zu diesem Knoten führenden Pfades adressierte Partition fallen würden. Die Gesamtzahl der Knoten kann das Produkt aus Datensatzgröße N und Dimension des Datenraumes D nicht übersteigen. Trotzdem läßt sich eine Partition (mithin ein Knoten mit **level** $= D$) durch D binäre Suchen[1] auf den Unterbäumen der Level 1 bis D effizient lokalisieren. Nach der

[1] Leider sind die jeweiligen Unterbäume bei zufälligem Einfügen von Knoten nicht notwendigerweise balanciert, so daß im schlimmsten Falle der Zeitaufwand der binären Suche linear in der Größe des Unterbaums wachsen kann. Bei Anwendung auf reale Daten tritt dies glücklicherweise in den seltensten Fällen auf. Der Algorithmus könnte jedoch dahingehend erweitert werden, daß Knoten randomisiert eingefügt werden oder daß die einzelnen Unterbäume dynamisch balanciert werden.

Konstruktion des Baumes lassen sich alle Knoten eines Levels leicht auffinden und auf diese Weise die Summation in Gleichung (4.7) leicht ausführen.

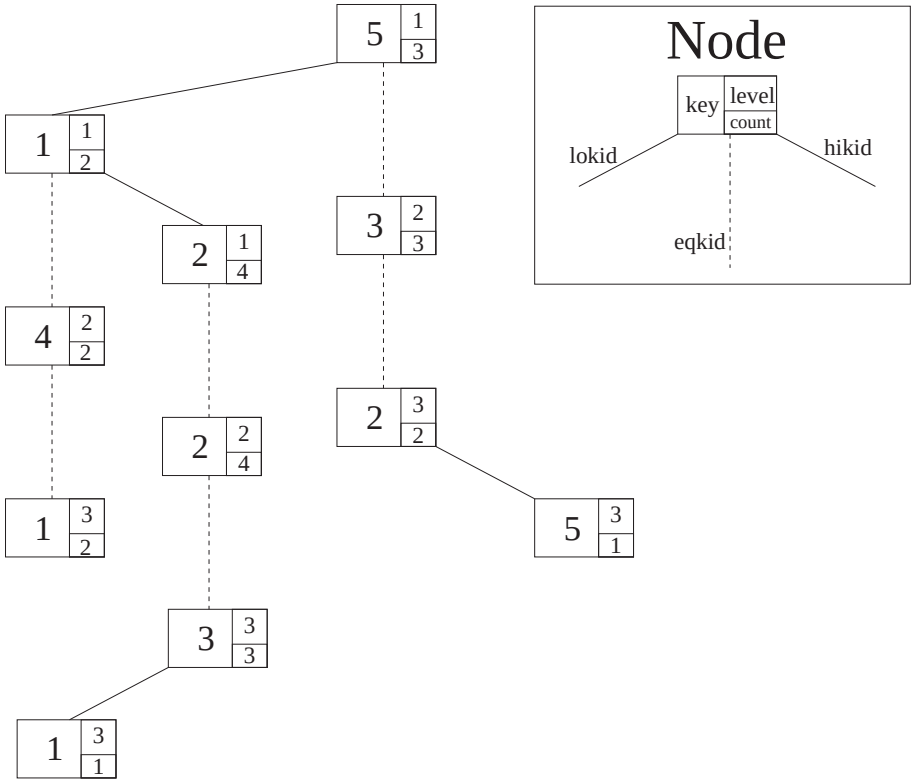


Abbildung 4.1.: Beispiel eines ternären Baums, in dem die folgenden Punkte $\{(5,3,2), (1,4,1), (5,3,2), (2,2,3), (5,3,5), (1,4,1), (2,2,3), (2,2,1), (2,2,3)\}$ in der angegebenen Reihenfolge eingefügt wurden.

Leider stellt die Bestimmung der verallgemeinerten Dimensionen mittels Box–Counting basierten Methoden, auch wenn diese effizient implementiert sind, für statistisch haltbare Aussagen über D_q, besonders für $q < 0$ bzw. $q > 1$, enorm hohe Anforderungen an die Größe der verwendeten Datensätze. Da diese oft nicht gegeben sind, stellen Box–Counting basierte Methoden eine schlechte Wahl zur Berechnung der verallgemeinerten Dimensionen dar. Nichtsdestotrotz findet sich ein auf ternären Suchbäumen basierender Algorithmus im Modul boxcount des Programmpaketes TSTOOL, der die Schätzung von D_0, D_1 sowie D_2 erlaubt.

4.2.2. Korrelationssummen basierte Ansätze

Eine deutliche Verbesserung sowohl im Hinblick auf die Fähigkeit, mit kleineren Datensätzen brauchbare Schätzungen zu erreichen, als auch in deren Genauigkeit bringt die Verwendung der *Korrelationssumme*:

$$C(r) = \lim_{N\to\infty} \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{j=1(i\neq j)}^{N} \Theta(r - d(x^i, x^j)), \tag{4.10}$$

wobei $\Theta(r)$ die Heaviside-sche Sprungfunktion darstellt:

$$\Theta(r) = \begin{cases} 0 & : \quad r \leq 0 \\ 1 & : \quad r > 0 \end{cases}$$

Der von Grassberger vorgeschlagene Algorithmus [20] erlaubt die Berechnung der Korrelationsdimension durch Bestimmung der Steigung der doppeltlogarithmischen Auftragung der Korrelationssumme gegen die Skalengröße r und ist in der Literatur als klassischer Grassberger–Procaccia–Algorithmus bekannt.

Von den vielen verschiedenen Korrekturen zur Verbesserung der Abschätzung von D_2 für spezielle Systeme oder bei kurzen Zeitreihen soll hier nur der von Theiler [45] vorgeschlagene Ausschluß zeitlich zum jeweiligen Referenzpunkt dicht benachbarter Vorgänger bzw. Nachfolgerpunkte erwähnt werden:

$$C^c(r) = \lim_{N\to\infty} \frac{1}{N_{\text{pairs}}} \sum_{i=1}^{N} \sum_{j=1(|i-j|>c)}^{N} \Theta(r - d(x^i, x^j)), \tag{4.11}$$

der Gleichung 4.10 für den Fall $c = 0$ mit einschließt. N_{pairs} gibt hier die Anzahl insgesamt betrachteter Punktpaare an.

Die Berechnung der Korrelationssumme $C(r)$ profitiert von der Verwendung eines effizienten Nächsten–Nachbar–Algorithmus, wobei jedoch statt der k–Nächsten–Nachbarn alle Punkte innerhalb eines gegebenen Radius gesucht werden (Bereichssuche, vgl. 3.4.4.3). Der im vorhergehenden Kapitel vorgestellte ATRIA Algorithmus stellt eine solche Bereichssuchfunktion zur Verfügung, die auf der gleichen Vorverarbeitungsstruktur basiert wie die eigentliche Nächste–Nachbar Suche.

Neben der Verwendung eines effizienten Algorithmus zur Bereichssuche kann der Rechenaufwand durch die Beschränkung der Auswertung der Korrelationssumme nur an zufällig aus dem

Datensatz gewählten Referenzpunkten reduziert werden, wobei eine zu starke Reduzierung der Anzahl der Referenzpunkte zu einer schlechten Statistik führt und den Skalierungsbereich der Korrelationssumme stark einschränkt. Da die Korrelationssumme in der numerischen Praxis nur für eine Reihe, meist exponentiell angeordneter Skalen $r_i = r_0 l^i$ ausgewertet wird, und gerade die im Vergleich zum Attraktordurchmesser bereits großen Skalen, bedingt durch die hohe Anzahl von Nachbarn, die dabei gefunden werden, einen hohen Rechenaufwand verursachen, sollte in einem weiteren Schritt die Anzahl der Referenzpunkte für jede Skala r_i einzeln gewählt werden. Bei großen Skalen r, die durch die hohe Anzahl Treffer meist schon bei Verwendung nur weniger Referenzpunkte bereits eine gute Statistik aufweisen, kann daher die Anzahl der Referenzpunkte deutlich kleiner als bei den kleinsten Skalen gewählt werden. Bei der Implementation der Korrelationssummenberechnung im Programmpaket TSTOOL wurde daher ein Algorithmus eingesetzt, der die Anzahl Referenzpunkte für jede Skala r_i innerhalb gegebener Unter- und Obergrenze so wählt, daß, wenn möglich, eine spezifizierte Mindestanzahl von Punktpaaren, die die Bedingung $d(x^i, x^j) < r_i$ erfüllen, gefunden wird (vgl. Quelltext A.4.4).

4.2.3. Nächste–Nachbar basierte Ansätze

Bei beiden bisher betrachteten Methoden zur Berechnung der Dimensionen wurden Korrelation oder empirische Häufigkeiten in Abhängigkeit eines Abstandes oder einer charakteristischen Abmessung r betrachtet. Da dieser Abstand bei der Berechnung fest vorgegeben wird, werden solche Methoden im Englischen auch als *fixed–size* Verfahren bezeichnet. Diese Verfahren sind für niedrigdimensionale Systeme, vorausgesetzt es stehen genügend Meßwerte zur Verfügung, ausreichend brauchbar. Für hochdimensionale Systeme ergibt sich jedoch die Schwierigkeit, daß die Verteilung der Interpunktdistanzen generell immer schmaler wird [34] (vgl. Abbildung 4.3, die Korrelationssumme variiert um mehr als den Faktor 2^{30}, während die r lediglich um den Faktor 2^8 variiert.). Dies macht das Bestimmen eines Bereiches, in dem z.B. die Korrelationssumme in der doppeltlogarithmischen Auftragung linear skaliert, oft recht schwierig.

Alternativ wurden daher Ansätze entwickelt, die die Skalierung der mittleren Abstände der Nächste–Nachbarn in Abhängigkeit der Nachbarordnung k betrachten. Badii und Politi 1984 sowie Somorjai 1986 nutzen zuerst und unabhängig voneinander Information über die Verteilung der Nachbardistanzen zur Schätzung der fraktalen Dimensionen.

Schram und van de Water [47] leiten unter Verwendung des multifraktalen Formalismus und der Annahme, daß die Wahrscheinlichkeit, mehrere Punkte in einer Partition zu finden, auf kleinen Skalen einer Poisson–Verteilung genügt (siehe A.3), als Resultat einen Zusammenhang zwischen den Momenten der Nachbardistanzen $\delta^\gamma(k,n)$ und dem Spektrum der fraktalen Dimensionen her:

$$\delta^\gamma(k,n) \quad = \quad \langle l^\gamma \rangle^{\frac{1}{\gamma}} \sim n^{-\frac{1}{D'(\gamma)}} \left(\frac{\Gamma(k+\frac{\gamma}{D'(\gamma)})}{\Gamma(k)} \right)^{\frac{1}{\gamma}}. \tag{4.12}$$

Die neu eingeführte Größe $D'(\gamma)$ steht mit den Rényi–Dimension D_q in folgendem Zusammenhang:

$$D'(\gamma = (1-q)D_q) \quad = \quad D_q. \tag{4.13}$$

Für den Fall $\gamma = 0$ ergibt sich folgende Formulierung, die die direkte Bestimmung der Informationsdimension D_1 ermöglicht:

$$\delta^0(k,n) \quad = \quad e^{\langle \ln l \rangle} \sim n^{-\frac{1}{D_1}} \quad e^{\frac{1}{D_1}\frac{d \ln \Gamma(k)}{dk}}. \tag{4.14}$$

Auch wenn Gleichung 4.12 und 4.14 prinzipiell die Bestimmung von D' aus der Skalierung der Nachbardistanzen mit zunehmender Nachbarordnung k oder mit zunehmender Datensatzgröße n oder beidem erlaubt, wurde im Verlauf dieser Arbeit nur die Skalierung mit k benutzt, da diese Methode bei Verwendung des bereits beschriebenen ATRIA zur Nachbarsuche ohne erneute Vorverabeitung ausgeführt werden kann.

Praktisch geschieht die Bestimmung von D' durch Anfitten des theoretischen Ausdrucks

$$\begin{aligned} z(k,\gamma) \quad &= \quad \left(\frac{\Gamma(k+\frac{\gamma}{D'(\gamma)})}{\Gamma(k)} \right)^{\frac{1}{\gamma}} \quad \text{bzw.} \\ z(k,0) \quad &= \quad e^{\frac{1}{D_1}\frac{d \ln \Gamma(k)}{dk}} \end{aligned} \tag{4.15}$$

gegen die numerisch bestimmten Momente der Nachbardistanzen

$$\begin{aligned} \delta^\gamma(k,n) \quad &\approx m(k,\gamma) = \quad \left(\frac{1}{|N_{\text{ref}}|} \sum_{j \in N_{\text{ref}}} d(j, nn_k(j))^\gamma \right)^{\frac{1}{\gamma}} \quad \text{bzw.} \\ \delta^0(k,n) \quad &\approx m(k,0) = \quad e^{\frac{1}{|N_{\text{ref}}|} \sum_{j \in N_{\text{ref}}} \ln(d(j, nn_k(j)))}, \end{aligned}$$

die aus dem Datensatz der vollen Größe n durch Suche der k_{max} Nächsten Nachbarn zu einer Menge zufällig gewählter Referenzpunkte N_{ref} bestimmt werden.

Anschließend wird folgende Fehlerfunktion mittels eines üblichen mehrdimensionalen Optimierungsverfahren minimiert:

$$\begin{aligned} E(D', \alpha) &= \sum_{k \in K} \ln(1 + e_k^2) \quad mit \\ e_k &= \frac{z(k) - \alpha m(k)}{z(k)}, \end{aligned}$$

wobei lediglich die Lage des Minimums bezüglich D' interessiert, die absolute Größenskala α wird verworfen. Dies hat den Vorteil, daß die Berechnung von Gleichung 4.15 durch Ausnutzen der Funktionalgleichung der Gammafunktion (vgl. Gleichung A.6 in Abschnitt A.2) zu einer effizient zu berechnenden Rekursion vereinfacht werden kann:

$$\begin{aligned} z(k, \gamma) &= u(k, \gamma)^{\frac{1}{\gamma}} \\ u(1, \gamma) &= 1 \\ u(k+1, \gamma) &= u(k, \gamma) \frac{k + \frac{\gamma}{D'(\gamma)}}{k} \end{aligned}$$

Die Verwendung einer logarithmischen Fehlerfunktion dient einer robusten Schätzung, bei der einzelne, stark abweichende Ausreißer das Ergebnis nicht zu stark beeinflussen können.

Die Auswahl des Bereichs der Nachbarordnungen $K = \{k_{\text{min}}, \ldots, k_{\text{max}}\}$ erfordert einige Überlegungen, die jedoch über den Rahmen dieser Arbeit hinausgehen. Daher sei an dieser Stelle auf den Artikel von Schram und van de Water [47] verwiesen.

Nachteilig bei diesem Nachbar–basierten Ansatz ist lediglich, daß es nicht möglich ist, die Dimensionen für ein vorgegebenes q direkt zu berechnen[2], da der Zusammenhang zwischen q und γ leider die vor der Berechnung noch unbekannte Größe D_q beinhaltet (vgl. Gleichung 4.13). Diesem Problem kann auf zwei Arten begegnet werden:

(1) Die Berechnung wird wiederholt ausgeführt, wobei das anfänglich gewählte γ vor jeder neuen Iteration so eingestellt wird, daß das effektiv berechnete $q_{\text{eff}} = 1 - \frac{\gamma}{D'(\gamma)}$ gegen das

[2] Mit Ausnahme von $q = 1$, das durch die Wahl von $\gamma = 0$ direkt berechnet werden kann (vgl. Gleichung 4.14).

vorgegebene q_0 konvergiert. Diese Methode hat allerdings den Nachteil, daß zur Bestimmung der Rényi–Dimensionen zu einem vorgegebenen q_0 mehrere Optimierungsläufe ausgeführt werden müssen.

(2) Die Optimierung wird für einen vorgegebenen Satz von γ–Stützstellen ausgeführt, anschließend wird aus dem daraus gewonnenen Verlauf von $D(q)$ auf ein vorgegebenes q_0 interpoliert[3] (siehe Abbildung 4.2). Dies hat neben dem geringeren Rechenaufwand noch den Vorteil, daß man von vornherein zu stark negative oder zu stark positive γ–Werte ausschließen kann, da deren verläßliche Schätzung enorm große Datensätze voraussetzen würde.

4.2.4. Numerische Bestimmung des Dimensionsspektrums dynamischer Systeme aus deren Zeitreihen

Zur verläßlichen Schätzung des Dimensionsspektrums dynamischer Systeme ist es neben der Verwendung einer ausreichend großen Datenmenge[4] wichtig, den systematischen Fehler, bedingt durch Autokorrelationseffekte der Zeitreihe, auszuschalten. Diese führen zur Unterschätzung der tatsächlichen Dimension. Gerade bei sehr fein abgetasteten Zeitreihen „sehen“ die meisten Methoden auf kleinen Skalen bzw. bei kleinen Nachbarordnungen die Dimension der Trajektorie, auf der sich der Systemzustand entwickelt.

Zwei Ansätze existieren zur Lösung des Problems:

(1) Ausschluß aller Punkte mit Indizes $t_{\text{ref}} - c \leq t_i \leq t_{\text{ref}} + c, c > 0$ um den jeweiligen Referenzpunkt t_{ref}. Diese ursprünglich von Theiler [45] vorgeschlagene Methode findet in ähnlicher Form bereits bei der Modellierung und Vorhersage dynamischer Systeme Anwendung (vgl. Abschnitt 2.3.2). Diese Methode ist allerdings bei den box–counting basierten Ansätzen nicht anwendbar, da dort keine Referenzpunkte verwendet werden.

(2) Wenn genügend Datensatzpunkte zur Verfügung stehen, kann daraus eine zufällige Stichprobe mit deutlich kleinerer Stückzahl gezogen werden. Diese Methode ist u.a. dann anwendbar, wenn die Datensätze numerisch generiert werden und in kurzer Zeit

[3] Dies ist allerdings nur möglich, sofern das Intervall, aus dem die γ–Stützstellen anfänglich gewählt wurden, groß genug war.

[4] Leider schwanken die Angaben zur Mindestanzahl der Punkte, die zur Schätzung einer gewissen Dimension erforderlich ist, von Autor zu Autor teilweise beträchtlich.

große Datenmengen produziert werden können. Bei diesem Ansatz müssen dann sowohl bei den korrelationssummenbasierten Ansätzen als auch bei den Ansätzen basierend auf der Skalierung der Momente der Nächste–Nachbardistanzen lediglich Selbsttreffer bei der Bereichssuche bzw. der Nachbarsuche ausgeschlossen werden ($c = 0$). Diese Methode wurde dann auch für die nachfolgend beschriebenen Berechnungen verwendet.

Zur Generierung der Testdatensätze wurde das bereits in Abschnitt 2.4.1 vorgestellte Baier–Sahle Differentialgleichungssystem mit den angegebenen Parameterwerten sowie Systemdimension $M = 5$ verwendet. Gemäß des zweiten Ansatzes wurde das System 100 mal von $T = 0$ bis $T = 26000$ mit Schrittweite $\triangle T = 0.5$ integriert, die ersten 2000 Sample wurden jeweils als transient verworfen. Aus den verbleibenden 50000 Samplen wurden dann je 2000 zufällig gezogen und dem ersten Testdatensatz zugefügt.

Der zweite Testdatensatz wurde durch Normierung jeder der fünf Komponenten des ersten Datensatzes auf den Wertebereich der ersten Komponente gewonnen.

Zur Generierung des dritten Testdatensatzes wurde eine Zeitverzögerungsrekonstruktion der Werte der ersten Komponente der originalen Zustandsvektoren mit Rekonstruktionsdimension $D = 10$ und Versatzzeit $L = 5$ durchgeführt. Die Versatzzeit wurde aus dem ersten Nulldurchgang der Autokorrelationsfunktion bestimmt. Die Rekonstruktionsdimension ergab sich aus der Tatsache, daß eine Einbettung fast sicher möglich ist, falls D größer als die zweifache Kapazitätsdimension, die hier zwar nicht bekannt ist, aber nach oben durch die Systemdimension $M = 5$ abgeschätzt werden kann, gewählt wird (vgl. Abschnitt 2.1.2). Darüberhinaus konnte nach der Methode von Cao [12] eine minimale Rekonstruktionsdimension zwischen 8 und 9 ermittelt werden, was sich nachträglich als konsistent mit den berechneten Dimensionsspektren herausstellen wird. Auch von diesen Zeitverzögerungsvektoren wurden dann jeweils 2000 Stück zufällig ausgewählt und dem Testdatensatz zugefügt. Alle Datensätze bestanden somit aus 200000 Punkten.

Die Ergebnisse der Dimensionsbestimmung für die rekonstruierten Zustandsvektoren unterscheiden sich nur unwesentlich bezüglich der verwendeten Metrik. Auch der theoretisch geforderte monoton fallende Verlauf ist gegeben (siehe Abbildung 4.2). Deutlicher fällt dagegen der Unterschied zu den aus den Originalzuständen geschätzten Dimensionsspektren aus. Diese zeigen nicht nur Abweichungen zu denen des rekonstruierten Datensatzes, sondern differieren auch für $q < 1$ zwischen L_2 und L_∞–Norm sowie zwischen normiert und nicht–normiert deutlich. Zudem weisen alle aus den Originalzuständen geschätzten Spektren nicht das theo-

retisch geforderte monotone Abfallen mit zunehmendem q auf. Allerdings stimmen die zum Vergleich in Tabelle 4.1 aufgeführten Werte der Korrelationsdimension jeweils mit den aus den Nächsten–Nachbardistanzen geschätzten Werten für D_2 verhältnismäßig gut überein.

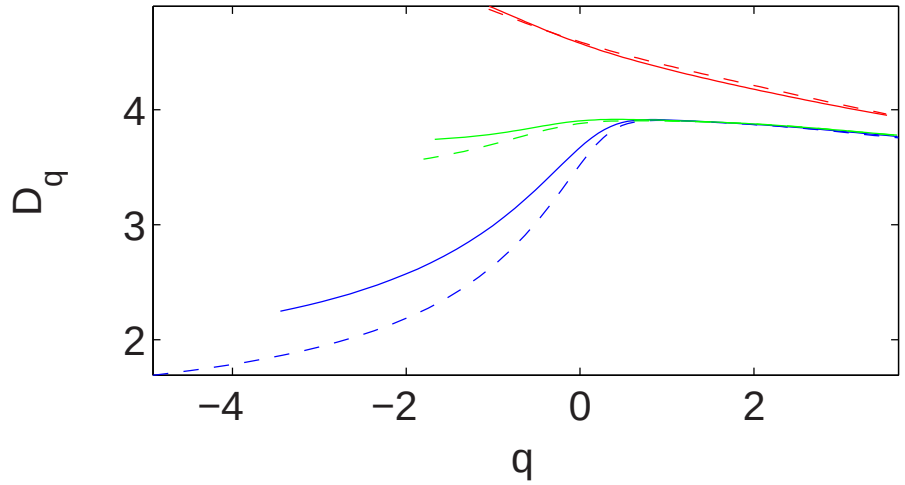


Abbildung 4.2.: Ergebnisse der Dimensionsbestimmung mittels des in Abschnitt 4.2.3 beschriebenen Verfahrens basierend auf den Verteilungen der Nächsten–Nachbardistanzen für die in Abschnitt 4.2.4 beschriebenen Datensätze. Das rote Linienpaar gibt das geschätzte Spektrum der Rényi–Dimensionen für den mittels Zeitverzögerungsrekonstruktion erzeugten Datensatz an. Das blaue Linienpaar gibt D_q für den Datensatz der originalen Zustandsvektoren an. Das grüne Linienpaar gibt D_q für den Datensatz der normierten Zustandsvektoren an. Die durchgezogene Linie steht jeweils für die L_2–Metrik, die gestrichelte für L_∞–Metrik.

4.3. Einfluß des Dimensionsspektrums auf die Effizienz der Nächsten–Nachbar Suche

Zur empirischen Bestimmung des Einflusses des Spektrums der fraktalen Dimensionen auf die Effizienz und damit Laufzeit des im Kapitel 3 vorgestellten Algorithmus zur Nächsten–Nachbar–Suche werden im folgenden Abschnitt die Ergebnisse einiger numerischer Experimente dargestellt. Alle Messungen wurden auf einem Arbeitsplatzrechner vom Typ Silicon Graphics Indigo2 unter dem Betriebssystem IRIX 6.5 (eine Unix–Variante) ausgeführt. Der

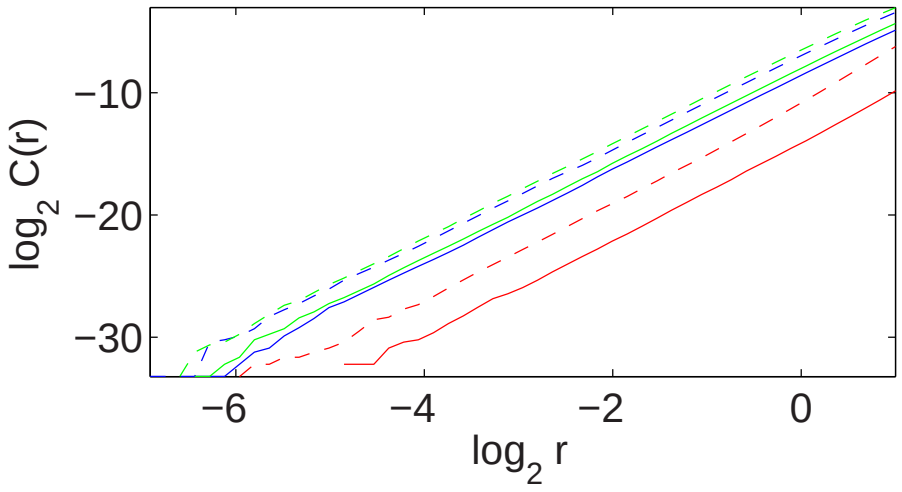


Abbildung 4.3.: Doppeltlogarithmische Auftragungen der in Abschnitt 4.2.2 beschriebenen Korrelationssumme $C^0(r)$ gegen den Radius r für die in Abschnitt 4.2.4 beschriebenen Datensätze. Das blaue Linienpaar gibt $C^0(r)$ für den Datensatz der originalen Zustandsvektoren an. Das grüne Linienpaar gibt $C^0(r)$ für den Datensatz der normierten Zustandsvektoren an. Das untere Linienpaar gibt $C^0(r)$ für den mittels Zeitverzögerungsrekonstruktion erzeugten Datensatz an. Die durchgezogene Linie steht jeweils für die L_2–Metrik, die gestrichelte für L_∞–Metrik. Die zugehörigen Steigungen sind in Tabelle 4.1 angegeben.

Rechner war mit einer 175 MHz R10000 CPU, 256 MB Hauptspeicher sowie 1 MB Secondary Level Cache ausgestattet. Es wurde stets darauf geachtet, die Hintergrundauslastung des Systems während der Tests so konstant und minimal zu halten, wie auf einem Mehrbenutzersystem möglich. Wenn nicht anders angegeben, wurden Distanzen in euklidischer Metrik berechnet. Die für die Tests benutzten Datensätze entstammen zwei verschiedenen dynamischen Systemen:

Datensätze des Typs **A** wurden durch Iteration der bereits in Abschnitt 3.5 vorgestellten D–dimensionalen Verallgemeinerung der Hénon Abbildung generiert [6].

Datensätze des Typs **B** wurden durch Integration des in Abschnitt 2.4.1 vorgestellten hyperchaotischen Baier–Sahle Systems mit den Parametern $a = 0.28$, $b = 4$, $d = 2$ und $\epsilon = 0.1$ erzeugt. Die Integration von $T = 0$ bis $T = 21200$ wurde mit zufälligen Anfangsbedingungen aus dem Intervall $[0 \ldots 0.01]$ gestartet, wobei der Systemzustand alle $\triangle T = 0.1$ abgetastet wurde. Die ersten 10000 Abtastwerte wurden wieder als

Typ	Metrik	D_2
Original	L_∞	3.82
Original	L_2	3.81
Original normiert	L_∞	3.86
Original normiert	L_2	3.92
Rekonstruiert	L_∞	3.96
Rekonstruiert	L_2	4.00

Tabelle 4.1.: Ergebnisse der Bestimmung der Korrelationsdimension mittels Auswertung der Steigung der Korrelationssumme in der doppeltlogarithmischen Auftragung (siehe Abbildung 4.3) für die in Abschnitt 4.2.4 beschriebenen Datensätze.

transient verworfen. Dann wurde x_1 zur Verzögerungskoordinatenrekonstruktion mit Einbettungsdimension $D = 24$ und Delay $L = 9\triangle T$ benutzt. Die Datensätze wurden jeweils auf 200000 Rekonstruktionsvektoren gekürzt.

4.3.1. Einfluß der fraktalen Dimension auf die Selektivität der Nachbarsuche

Als maschinenunabhängiges Maß der Effektivität eines Nächsten–Nachbar–Algorithmus wird im folgenden das Verhältnis f_k aus der Anzahl Distanzberechnungen, die notwendig sind, um k Nächste–Nachbarn zu finden, zu der Anzahl der Punkte im Datensatz verwendet. Da dieser Wert vom jeweiligen Anfragepunkt q abhängt, wird f_k immer durch Mittelung über eine große Anzahl verschiedener Anfragepunkte gewonnen. Da die Anfragepunkte in den nachfolgenden Rechnungen aus dem Datensatz selbst stammen, ist ihre Verteilung prinzipiell identisch zu der des Datensatzes. Dies macht es allerdings notwendig, Selbsttreffer bei der Suche auszuschließen, da das Lokalisieren eines im Datensatz vorkommendes Punktes mit fast konstantem Aufwand unabhängig von Datensatzdimension und fraktaler Dimension erledigt werden kann wie in Abbildung 4.4 dargestellt.

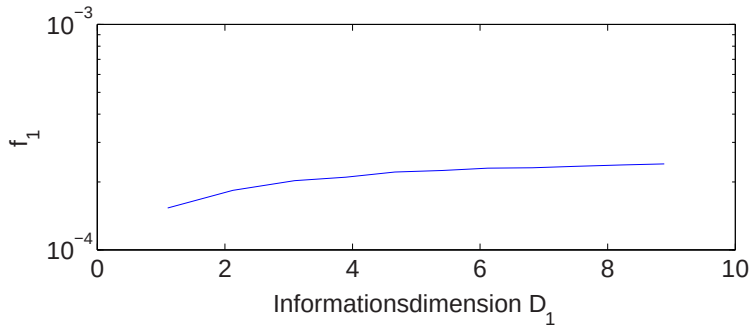


Abbildung 4.4.: Relative Anzahl der zur Lokalisierung eines Punktes notwendigen Distanzberechnungen aufgetragen gegen die Informationsdimension D_1. Der Graph stellt f_1 für Datensätze vom Typ A mit D zwischen 2 und 12 dar. Für dieses System wächst die Informationsdimension D_1 monoton mit zunehmendem D an. Jeder Datensatz bestand aus 200000 Punkten, aus denen die Anfragepunkte zufällig ausgewählt wurden. Selbsttreffer waren ausdrücklich erlaubt.

Werden die Nächsten–Nachbarn durch vollständiges Durchsuchen des Datensatzes ermittelt (im Englischen auch *brute force method* oder *exhaustive search*), ist f_k identisch 1. Ein effizienter Algorithmus sollte jedoch, zumindest für $k \ll N$, durch selektives Auswählen von Clustern, die mit hoher Wahrscheinlichkeit Nachbarn enthalten, wesentlich kleinere Werte erreichen.

Abbildung 4.5 zeigt f_1 und f_{128} jeweils für Datensätze des Typs A und des Typs B. Während für erstere die formale Dimension D des Datenraumes zwischen 2 und 12 variiert, wird D für die Datensätze des Typs B konstant bei 24 gehalten, dafür aber die Systemdimension M in Zweierschritten von 3 auf 11 erhöht, um verschiedene Werte der fraktalen Dimension D_q zu erzielen. Dabei wird, mit allem Vorbehalt, die Informationsdimension D_1 als Repräsentant für die fraktale Dimension verwendet [21].

Aufgabe der Experimente war es, die 128 exakten Nächsten–Nachbarn für jeweils 5000 zufällig aus dem jeweiligen Datensatz ausgewählte Anfragepunkte zu finden. Man beachte die logarithmische Auftragung der f_k Achse! Die Anzahl der Distanzberechnungen wächst anfänglich fast exponentiell mit der Informationsdimension D_1. Bei höheren Werten von D_1 macht sich eine Art Sättigung bemerkbar, da sich die Anzahl Distanzberechnungen der Zahl der Punkte im Datensatz annähert. Spätestens dann allerdings übersteigt die Rechenzeit des vorgestellten Algorithmus aufgrund des zusätzlichen Verwaltungsaufwands die einer naiven

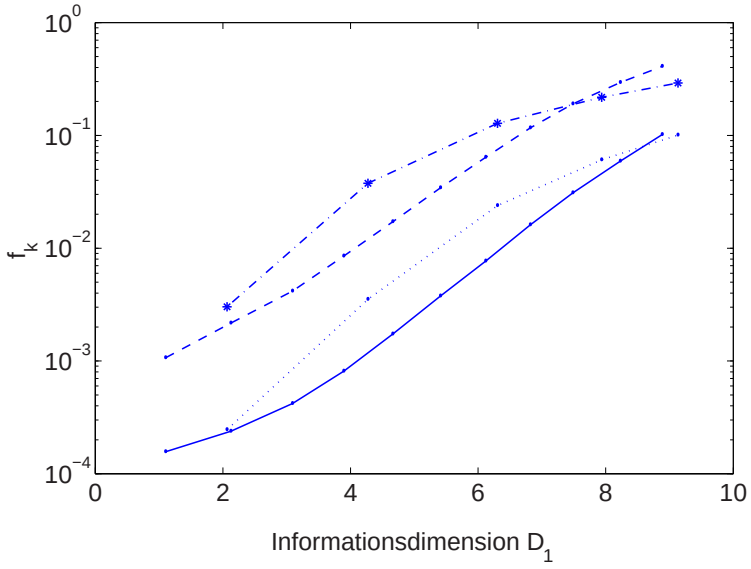


Abbildung 4.5.: Relative Anzahl der Distanzberechnungen aufgetragen gegen die Informationsdimension D_1 des Datensatzes. Der durchgezogene Graph zeigt f_1 für Datensätze vom Typ A mit D zwischen 2 und 12, die gestrichelte Kurve gibt f_{128} wieder. Für dieses System wächst die Informationsdimension D_1 monoton mit D an. Der gepunktete bzw. der gestrichelt–gepunktete Graph stellen f_1 bzw. f_{128} für Datensätze vom Typ B mit konstanter Einbettungsdimension $D = 24$ dar. Für diese wird der Systemparameter M in Zweierschritten von 3 auf 11 erhöht. Alle verwendeten Datensätze bestanden aus 200000 Punkten.

Implementation, die den Datensatz einfach vollständig durchsucht.

Trotz der deutlich unterschiedlichen formalen Dimension D der Datenräume von A und B zeigen die jeweiligen Graphen f_1 sowie f_{128} für beide Typen Datensätze einen ähnlichen Verlauf. Dennoch beträgt beispielsweise die formale Dimension D des Datensatzes A am ersten Schnittpunkt beider f_1 Graphen, an dem sowohl Effizienz der Suche als auch Informationsdimension D_1 beider Datensätze nahezu übereinstimmen, 3, während die des anderen Datensatzes 24 ist. Dies zeigt deutlich, daß die Dimension des Datenraumes für die Selektivität des Suchalgorithmus kaum von Bedeutung ist. Wesentlich entscheidender ist dagegen die fraktale Dimension des Datensatzes, in dem die Nächsten–Nachbarn gesucht werden. Diese determiniert stark die Selektivität des Suchverfahrens, wodurch auch in formal hochdimensionalen Räumen bei niedriger fraktaler Dimension der Daten eine gute Selektivität erreicht werden

kann. Allerdings nimmt die tatsächliche Laufzeit der Suche selbst bei konstanter Selektivität schon allein deshalb mit der formalen Dimension zu, da die Anzahl der zur Berechnung einer Distanz notwendigen Rechenschritte zunimmt.

Die Verwendung der Informationsdimension D_1 zur Charakterisierung der Verteilung der Punkte eines Datensatzes ist sicherlich eine willkürliche Entscheidung und wurde hauptsächlich durch den Vorteil der einfachen und robusten Berechenbarkeit motiviert (vgl. 4.2.3). Dabei muß erwähnt werden, daß D_1 die Verteilung der Punkte des Datensatzes auf keinen Fall vollständig beschreibt und dieser Wert daher nur grobe Aussagen über die zu erwartende Effizienz des vorgestellten Algorithmus zur Berechnung der Nächsten–Nachbarn geben kann.

4.4. Zusammenfassung

Die Entwicklung des Dimensionsbegriffes von den klassischen, ganzzahligen Dimension der euklidischen Geometrie bis zu dem Dimensionsbegriff von Rényi zeigt eine zunehmende Verfeinerung des Konzeptes des von einer Menge von Punkten abgedeckten Raumes. Wird aus dieser Punktmenge nur eine endliche Stichprobe gezogen (z.B. in Form eines numerisch generierten Datensatzes), bedarf es spezieller numerischer Verfahren, daraus eine Teilmenge des Dimensionensspektrums zu schätzen. Drei Klassen solcher Verfahren wurden vorgestellt:

- Box–Counting basierte Verfahren.
- Korrelationssummen basierte Verfahren
- Nächste–Nachbar basierte Verfahren

Während der Zusammenhang der beiden zuletzt genannten mit der Bestimmung der Nächsten–Nachbarn bzw. der Bereichssuche sofort einsichtig ist, fällt die enge Verwandschaft des ersten zu den Box–Assissted Verfahren zur Nachbarsuche [39] erst auf den zweiten Blick auf. Dieses ist auch das am wenigsten geeignete Verfahren zur verläßlichen Bestimmung fraktaler Dimensionen aus kleinen Datensätzen und wurde daher nicht weiter betrachtet.

Die Anwendung der anderen Verfahren auf einen Datensatz eines hyperchaotischen Systems zeigt die Einsatzgebiete und Grenzen dieser Ansätze. Die unbefriedigenden Resultate bei der Bestimmung des Spektrums der Rényi–Dimensionen für negative q aus den Originaldaten

des untersuchten Systems machen es wünschenswert, Verfahren zu entwickeln, die zusätzlich Fehlergrenzen zurückliefern.

Die Untersuchungen über den Zusammenhang zwischen Effizienz des ATRIA und der fraktalen Dimension des Datensatzes zeigen, daß die Rechenzeit des Algorithmus zwar kritisch von der fraktalen Dimension abhängt, nicht jedoch von der *formalen* Dimension des Raumes, in dem die Datenpunkte liegen. Dies macht den Algorithmus gut geeignet für den Einsatz in der nichtlinearen Zeitreihenanalyse, da die fraktale Dimension vieler Datensätze, speziell die der durch Zeitverzögerungsrekonstruktion erzeugten, oft deutlich geringer als deren formale ist.

5. Lokale Modellierung raum–zeitlicher dynamischer Systeme

5.1. Motivation

Der Ausgangspunkt für viele Analysemethoden, die im Rahmen der nichtlinearen Zeitreihenanalyse angewendet werden, ist die Messung einer einzelnen, skalaren Observablen des zu untersuchenden Systems. Dagegen sind viele interessante dynamische Systeme vom Aufbau her räumlich ausgedehnt, wodurch deren Beschreibung lediglich anhand einiger weniger globalen Observablen das System nicht komplett beschreiben kann. Andererseits sind heutzutage Vorrichtungen zur Aufnahme, Speicherung und Auswertung größerer raum–zeitlicher Datensätze[1] in vielen Fällen problemlos verfügbar. Dies erlaubt es, die aus dem Bereich der Zeitreihenanalyse skalarer Observablen bekannten datengetriebenen Modellierung– und Vorhersageverfahren auf räumlich ausgedehnte Systeme auszuweiten.

Inspiriert von der Tatsache, daß die Dynamik vieler physikalischer, technischer und chemischer Systeme durch lokal formulierte Gleichungen beschrieben werden kann, wird in dieser Arbeit ein Ansatz zur datengetrieben Modellierung raum–zeitlicher Dynamik propagiert, bei dem analog zur Zustandsrekonstruktion in der skalaren Zeitreihenanalyse eine Rekonstruktion des Zustandes des dynamischen Systems erzeugt wird, der allerdings nur für ein vergleichsweise eng umschriebenes Gebiet des räumlich ausgedehnten Systems Aussagekraft hat. Das vorgestellte Rekonstruktionsverfahren bedient sich hierbei der Information aus der zeitlichen und räumlichen Umgebung (Nachbarschaft) des Gebietes der Zeitreihe, dessen Zustand rekonstruiert werden soll. Allerdings können dabei die räumlichen Dimensionen (wovon es eine

[1] U.a. in Form von CCD–Kameras nebst Bildwandlerkarte und angeschlossenem Rechner.

oder mehrere geben kann) und die zeitliche aus Gründen der Kausalität nicht gleich behandelt werden, analog zum skalaren Fall gehen nur Information über gegenwärtige und vergangene Meßpunkte in die Rekonstruktion ein.

Als Voraussetzung für das Verfahren wird eine homogene Dynamik an verschiedenen Punkten (ausgenommen Gebiete am Rand) des Systems angenommen. Diese Annahme ist sicherlich in der Realität nie ganz zu erfüllen, mag aber in guter Näherung in vielen Fällen gegeben sein. Eine Erweiterung auf Systeme mit nicht–homogener Dynamik ist allerdings möglich und wird im Abschnitt 5.3.1 kurz skizziert.

5.2. Vorarbeiten

Prinzipiell kann die Rekonstruktion solcher *lokalen* Zustände auf verschiedene Weisen bewerkstelligt werden. Die grundsätzliche Idee wurde nach bestem Wissen des Autors von K. Kaneko [23] 1989 vorgeschlagen. Später wurde von Rubin [36] ein ähnlicher Ansatz zur Charakterisierung dynamischer und statischer Muster verwendet. Orstavik und Stark [30] benutzten raum–zeitliche Einbettungstechniken zur Kreuzvorhersage gekoppelter Abbildungsgitter (coupled map lattices). Dabei betrachteten sie den zu untersuchenden räumlich lokalen Ausschnitt als ein niedrigdimensionales System, das mit dem gesamten Gitter nur über den Rand interagiert. In dieser Sichtweise ist das hochdimensionale Gesamtsystem ein stochastischer Antrieb des niedrigdimensionalen lokalen Systems und akzeptable Vorhersagen können nur dann gemacht werden, wenn die Stärke der Kopplung und damit die stochastische Komponente klein ist.

Im Gegensatz dazu wird bei der im folgenden beschriebenen Vorgehensweise davon ausgegangen, daß ein lokaler Zustand in einem eindeutigen und deterministischen Sinne existiert, der zumindest prinzipiell exakte Vorhersagen erlauben würde (vgl. [33]). Vorhersagefehler lassen sich in dieser Sichtweise also nur auf unzureichende Modellierungstechniken, nicht jedoch auf eine stochastische Komponente der Systemdynamik zurückführen.

Dieser spekulative Ansatz wird durch die Klasse der gekoppelten Abbildungsgitter motiviert, bei der die gemachten Annahmen leicht verifiziert werden können. Numerische Experimente mit gekoppelten Oszillatoren sowie Systemen von partiellen Differentialgleichungen (PDE) weisen jedoch darauf hin, daß die Annahmen auch bei anderen Klassen von Systemen zutreffen könnten. Es muß betont werden, daß letzteres eine notwendige Bedingung dafür darstellt,

daß die im folgenden Anschnitt genauer beschriebene Rekonstruktionstechnik auch auf Datensätze von technischen oder natürlichen Systemen erfolgreich angewendet werden kann, bei denen von vornherein nicht unbedingt feststeht, zu welcher Klasse von Systemen sie gehören.

5.3. Rekonstruktion lokaler Zustände

Im folgenden Abschnitt wird die Rekonstruktion lokaler Zustände anhand einer räumlich eindimensionalen Zeitreihe diskutiert. Die Erweiterung des Verfahrens auf mehrere räumliche Dimensionen ist problemlos möglich.

Die gesamte raum–zeitliche Zeitreihe (im Englischen **s***patio–***t***emporal* **t***ime* **s***eries*, was die Abkürzung *STTS* motiviert), sei als $N \times M$-Matrix S, wie in Abbildung 5.1 dargestellt, gegeben. Der lokale Zustand des Systems an räumlicher Position m zum Zeitpunkt n wird analog zur Methode der Zeitverzögerungskoordinaten (vgl 2.1.2) aus dem aktuellen Sample s_m^n nebst räumlich benachbarten Werten zu beiden Seiten des zentralen Elements sowie den korrespondierenden Samples aus der zeitlichen Vergangenheit gebildet. Zusammen formen diese Abtastwerte den sogenannten *lokalen Zustandsvektor*:

$$\begin{aligned} \mathbf{x}_m^n \quad = \quad & (s_{m-IK}^n, \ldots, s_m^n, \ldots, s_{m+IK}^n \\ & s_{m-IK}^{n-L}, \ldots, s_m^{n-L}, \ldots, s_{m+IK}^{n-L}), \ldots, \\ & s_{m-IK}^{n-JL}, \ldots, s_m^{n-JL}, \ldots, s_{m+IK}^{n-JL}) \end{aligned} \tag{5.1}$$

wobei I die Anzahl räumlicher Nachbarn, J die Anzahl vergangener Werte, K die räumliche Verzögerung und L die aus der Zeitverzögerungsrekonstruktion skalarer Zeitreihen bekannte zeitliche Verzögerung darstellt. Das Rekonstruktionsschema für $I = 1$, $J = 3$, $K = 2$ und $L = 2$ ist exemplarisch in Abbildung 5.1 skizziert. Die Dimension d des Zustandsvektors $\mathbf{x}_m^n \in R^d$ ist durch $d = (J + 1)(1 + 2I)$ gegeben.

5.3.1. Randbedingungen

Bei dem im vorigen Abschnitt vorgestellten Verfahren treten zwei Effekte bei der Rekonstruktion lokaler Zustände am (räumlichen) Rand der Zeitreihe auf, die besondere Beachtung verdienen:

(1) Am Rande des Gebietes, von dem die Meßapparatur Daten aufgenommenen hat, kommt es zu Problemen bei der Konstruktion der lokalen Zustände, da schlicht und einfach Meßwerte fehlen, um den lokalen Zustandsvektor zu formen. Dieses Problem läßt sich durch Anfügen künstlicher Meßwerte $-c$, $-2c$, $-3c$, ...am linken und c, $2c$, $3c$ am rechten Rand der Zeitreihe begegnen. Die Konstante c wird dabei deutlich größer als das Maximum der Absolutwerte aller Zeitreihenwerte gewählt. Durch dieses Vorgehen befinden sich alle randnahen Zustände mit gleicher Entfernung zum Rand in jeweils separaten Unterräumen des globalen Rekonstruktionsraumes R^d und werden dementsprechend auch bei nachfolgenden Verarbeitungsschritten seperat gehandhabt.

Da sich die effektive Dimension der rekonstruierten Zustände am Rand von der im nicht von Rand beeinflußten Teil befindlichen unterscheidet, könnte für diese prinzipiell ein

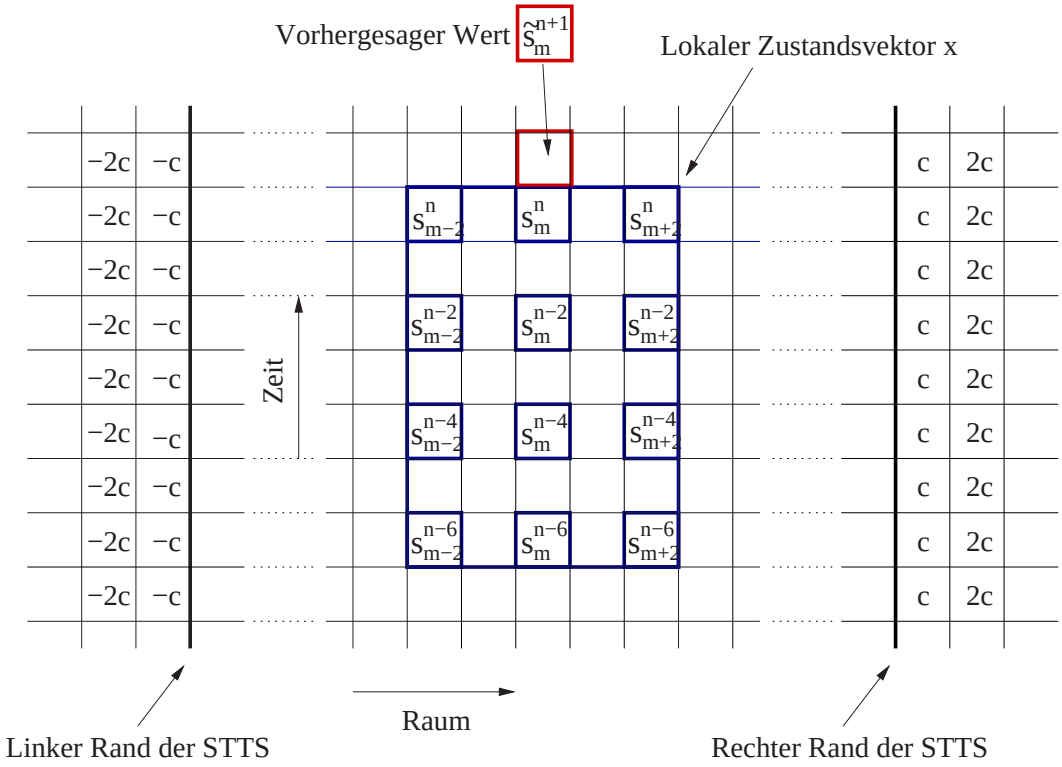


Abbildung 5.1.: Rekonstruktion lokaler Zustände aus den Meßwerten einer raum–zeitlichen Zeitreihe $S = \{s_m^n\}_{m=1,\ldots,M}^{n=1,\ldots,N}$. Die Konstanten $-2c$, $-c$, ...auf der linken sowie c, $2c$, ...auf der rechten Seite der Abbildung weisen auf die in Abschnitt 5.3.1 beschriebene Erweiterung der Zeitreihe über deren eigentliche Begrenzung hinaus hin.

anderer Satz Parameter I, J, K und L verwendet werden, was jedoch aufgrund der unterschiedlichen Dimension der Rekonstruktionsvektoren Schwierigkeiten bei der nachfolgend beschriebenen lokalen Modellierung bereiten würde.

(2) Dem Einfluß der physikalischen Randbedingungen des Systems kann man begegnen, indem man, statt die Zeitreihe an den Rändern zu erweitern, die lokalen Zustandsvektoren mit einer weitere Komponente bzw. Koordinate versieht, der sogenannten *Penalty*–Komponente $p(\tilde{m})$. Diese ist eine Funktion des räumlichen Ortes m des lokalen Zustandes:

$$p(\tilde{m}) = a\tilde{m}^b \tag{5.2}$$

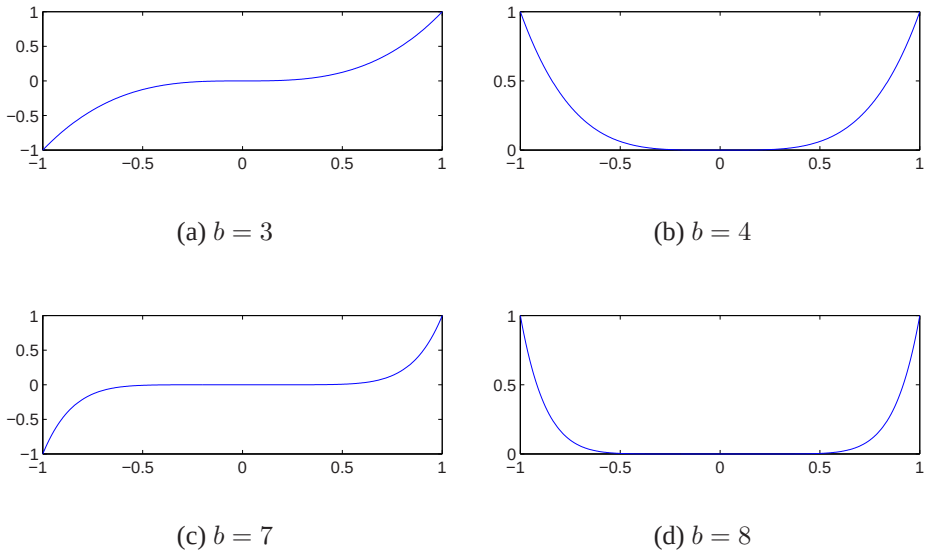
wobei $-1 \leq \tilde{m} \leq 1$ die aus m gewonnene normalisierte Position des Zustandes darstellt. Die Parameter $a \in \mathbb{R}^+$ und $b \in \mathbb{N}$ dienen zur Feinabstimmung der Penalty–Komponente (siehe Abbildung 5.2). Die Motivation dahinter ist die Annahme, daß lokale Zustände mit gleicher Position bezüglich der Ränder des Systems der gleichen Dynamik unterliegen. Durch die Penalty–Komponente wird daher Information über die Position des lokalen Zustandes in diesen eingebracht. Gleichzeitig erlaubt diese Erweiterung des Zustandsvektors auch den Umgang mit Inhomogenitäten der Dynamik, die nicht nur am Rand des Systems in Erscheinung treten.

Natürlich können der Rand des von der Meßapparatur aufgenommenen Gebietes und der physikalische Rand des Systems auch übereinstimmen. Bei den in den folgenden Abschnitten verwendeten numerisch generierten Zeitreihen ist dies der Fall. Daher kommen zur Rekonstruktion der lokalen Zustandsvektoren im folgenden beide Methoden zum Einsatz:

$$\begin{aligned} x_m^n \;=\; & \big(s_{m-IK}^n, \ldots, s_m^n, \ldots, s_{m+IK}^n \\ & s_{m-IK}^{n-JL}, \ldots, s_m^{n-JL}, \ldots, s_{m+IK}^{n-JL}, p(\tilde{m})\big) \end{aligned} \tag{5.3}$$

5.4. Lokale Modellierung der Dynamik lokaler Zustände

Die in den Abschnitten 2.2.1 bis 2.4.5 bereits diskutierten Methoden zur Modellierung, Validierung und Vorhersage können ohne größere Modifikationen in Verbindung mit der vorgestellten Rekonstruktion lokaler Zustände angewandt werden, um die Dynamik raum–zeitlicher



(a) $b = 3$ (b) $b = 4$

(c) $b = 7$ (d) $b = 8$

Abbildung 5.2.: Verlauf der Penalty–Funktion $p(\tilde{m})$ für verschiedene Exponenten b. Der Parameter a wurde konstant zu 1 gesetzt. Durch Wahl geradzahliger bzw. ungeradzahliger Exponenten b läßt sich symmetrischer bzw. asymmetrischer Verlauf der Penalty–Komponente einstellen.

Systeme zu modellieren. Einzig die Behandlung der zeitlichen und räumlichen Indizes erfordert genauere Betrachtung. Während im skalaren Fall die zeitlichen Indizes der Punkte zur Identifizierung eines Zustandsvektors ausreichen, läßt sich in diesem Fall ein Zustandsvektor erst durch einen Zeitindex n und einen Raumindex m lokalisieren. Dies erfordert eine Modifikationen des Nächsten–Nachbar–Algorithmus, der zu jedem gefundenen Nachbarn sowohl dessen Zeitindex nn_j^n als auch dessen Raumindex nn_j^m zurückliefern muß[2].

5.4.1. Iterierte Vorhersage mit lokalen Zuständen

Die iterierte Vorhersage raum–zeitlicher Zeitreihe kann prinzipiell analog zur Vorhersage skalarer Zeitreihen (vgl. 2.2.2) betrieben werden. Allerdings müssen jetzt zunächst die Ein–Schritt–Vorhersagen $\tilde{s}_m^{n+1} = \tilde{f}(x_m^n)$ für alle räumlichen Positionen $m = 1, \ldots, M$ berechnet werden, bevor zum nächsten Zeitschritt $n + 1$ übergegangen werden kann, da erst dann die

[2]Dies läßt sich natürlich schon dadurch erreichen, daß für die N mal M Punkte des Datensatzes ein neuer, eindeutiger Index $i = n * M + m$ bzw. $i = n + m * N$ zur Addressierung verwendet wird, so daß der eigentliche Nachbar–Algorithmus auf diesem linearen Index i arbeiten kann. Danach werden die Indizes nn_j der gefundenen Nachbarn wieder in räumlichen nn_j^m und zeitlichen Part nn_j^n umgerechnet.

alle lokalen Zustände $\tilde{\mathbf{x}}_m^{n+1}$ gebildet werden können.

Wieder kann sowohl absolute Mittelung:

$$\tilde{s}_m^{n+1} = \tilde{f}(x_m^n) = \frac{1}{\sum_{j=1}^{k} w_j} \sum_{j=1}^{k} w_j \, s_{nn_j^m}^{nn_j^n+1} \tag{5.4}$$

oder integrierte Mittelung:

$$\tilde{s}_m^{n+1} = \tilde{f}(x_m^n) = s_m^n + \frac{1}{\sum_{j=1}^{k} w_j} \sum_{j=1}^{k} w_j \, (s_{nn_j^m}^{nn_j^n+1} - s_{nn_j^m}^{nn_j^n}) \tag{5.5}$$

zum Einsatz kommen.

5.4.2. Validierung

Die bereits in den Abschnitten 2.3 bis 2.3.3 eingeführten Methoden der Validierung können dank ihrer Generalität gut auf die Problemstellung der Modellierung raum–zeitlicher Zeitreihen übertragen werden. Dabei kommt die normierte Variante des MSCV (vgl. Gleichung 2.11)

$$\begin{aligned} \mathrm{NMSE}_{1,p}^{c} = {} & \frac{NM}{p|T_{\text{ref}}| \sum_{n=1,m=1}^{N,M} (s_m^n - \bar{s})^2} \sum_{n \in N_{\text{ref}}} \sum_{m=1}^{M} \Big(\big(s_m^{n+1} - \\ & \tilde{f}_{t-c}^{t+c}(x_m^n) \big)^2 + \sum_{i=1}^{p-1} \Big(s_m^{n+i+1} - \tilde{f}_{t+i-c}^{t+i+c}(\tilde{x}_m^{n+i}) \Big)^2 \Big) . \end{aligned} \tag{5.6}$$

in einer angepassten Formulierung zum Einsatz. Lediglich das Auslassen einzelner Zustandsvektoren des Trainingsdatensatzes bei der Nachbarsuche (vgl. 2.3.2) erfordert genauere Betrachtung. So werden bei der Berechnung der Approximation $\tilde{f}_{t_1}^{t_2}(x_m^n)$ nicht nur alle lokalen Zustandsvektoren $x_m^{t_1}, \ldots, x_m^{t_2}$ mit gleichem m von der Suche ausgeschlossen, sondern auch diejenigen mit beliebigen $m = 1, \ldots, M$, um keine Information aus dem Testdatensatz in das Ergebnis der Vorhersage (und damit das Ergebnis der Validierung) mit eingehen zu lassen.

5.5. Beispiele lokaler Modellierung zur Vorhersage raum–zeitlicher Zeitreihen

5.5.1. Untersuchte Systeme

Die in den numerischen Experimenten benutzten STTS stammen von zwei verschiedenen Systemen:

(a) Der erste Datensatz wurde durch Integration der Kuramoto–Sivashinsky partiellen Differentialgleichung (PDE) [27]:

$$u_t = -2uu_x - u_{xx} - u_{xxxx} \tag{5.7}$$

im Intervall $[0, L]$ mit den Randbedingungen $u = u_x = 0$ am linken $x = 0$ und am rechten Rand $x = L = 200$ erzeugt. Die raum–zeitliche Dynamik des Systems wird von einem hyperchaotischen Attraktor mit einer Lyapunov–Dimension von $D_L \approx 43$ bestimmt. Die räumliche Abtastrate beträgt 2, so daß insgesamt $M = 400$ Abtastwerte pro Zeitschritt zur Verfügung stehen. 500 Zeitschritte gehen in die Validierung ein.

(b) Dieser Datensatz wurde durch die Iteration einer Kette von gekoppelte logistischen Abbildungen

$$\begin{aligned} s_m^{n+1} &= f\left(0.5s_m^n + 0.25(s_{m-1}^n + s_{m+1}^n)\right) \quad \text{mit} \\ f(x) &= ax(1-x) \end{aligned} \tag{5.8}$$

generiert, wobei der Parameter a afu 4 gesetzt wurde, weiter wurden feste Randbedingungen $x_1^n = 0.5$ für alle Zeitschritte $n = 1, \ldots, 200$ angenommen.

5.5.2. Parametervorgaben

Wie bereits in Abschnitt 2.3 diskutiert, erfordert es meist das Geschick oder die Erfahrung des Experimentators, um eine geeignete Vorauswahl der Parameter zu treffen, die einerseits generell genug ist, um das globale Optimum des eingesetzten Modellierungsverfahrens einzuschließen, andererseits aber den Rechenaufwand der Validierung in realistischen Grenzen hält. Oft versucht man daher, zusätzliche Information über das untersuchte System in die

Entscheidungsfindung einzubeziehen. So wurde hier für beide Systeme versucht, die Anzahl zeitlich vergangener Werte J im Vorfeld abzuschätzen. Dabei zeigte sich, daß die Hinzunahme vergangener Werte nicht zur Verbesserung der Vorhersage führte, daher wurden sie von vornherein weggelassen.

Zur Validierung wurde der $\text{NMSE}^c_{1,p}$ (siehe Tabelle 5.1) für alle Kombination aus folgenden Parametervorgaben bzw. Approximationsmethoden ermittelt:

- Die Anzahl der räumlichen Nachbarwerte I wurde für den KS–Datensatz aus der Menge $\{4, 5, 6, 7, 8, 9, 10, 12\}$ gewählt, für den Datensatz der gekoppelten logistischen Abbildungen aus $\{1, 2, 3\}$. Die Anzahl der zeitlichen Nachbarwerte wurde auf 0 gesetzt, K sowie L auf 1.
- Für den Datensatz der gekoppelten logistischen Abbildungen wurde zur Behandlung der Randbedingungen lediglich c auf 2 gesetzt, es wurde hier keine Penalty–Komponente verwendet. Dagegen kamen bei dem KS–Datensatz beide Methoden der Behandlung von Randbedingungen zum Einsatz mit Parametern $a = 1.0$, $b = 7$ und $c = 7.43$.
- Anzahl verwendeter Nachbarn $k \in \{1, \ldots, 5\}$
- Konstante, linear abfallende, biquadratische oder trikubische Wichtung
- Absolute oder intergrierte Mittelung

Insgesamt wurde dabei jede Zeitreihe 100 mal zufällig in Test- und Trainingsdatensatz aufgeteilt. Die Nachbarsuche profitiert aufgrund der vergleichsweise hohen Dimension der lokalen Zustandsvektoren deutlich von der Verwendung der partiellen Distanzberechnung (vgl. Abschnitt 3.4.1). Zur Berechnung der Distanzen im Rekonstruktionsraum wurde die euklidische Metrik verwendet.

5.5.3. Vorhersagen

Abbildungen 5.3 und 5.4 zeigen die Ergebnisse der freilaufenden, iterierten Vorhersage für beide raum–zeitliche Datensätze. Für die Modellierung beider Zeitreihe wurde der bei der Crossvalidierung ermittelte optimale Parametersatz verwendet. Zum Vergleich der Güte der Modellierung wurde jeweils der Absolutwert der Differenz zwischen Originalzeitreihe und

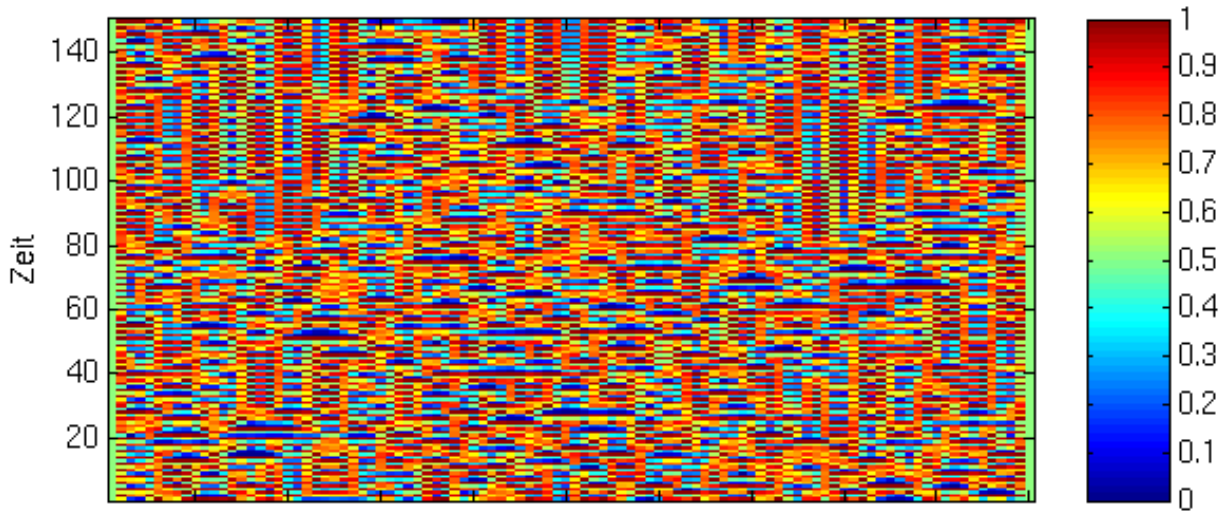
System	p	I	k	n	Modus	$\mathrm{NMSE}^{c}_{1,p}$	$\frac{\mathrm{NMSE}_{\max}}{\mathrm{NMSE}_{\min}}$
Logistische Abbildungen	4	1	5	1	Abs.	0.0026	122
Kuramoto–Sivashinsky	8	9	5	1	Int.	0.0701	7.5

Tabelle 5.1.: Ergebnisse der Validierung der STTS–Datensätze. Jede Zeile der Tabelle zeigt den Parametersatz für das jeweilige System, der das verwendete Fehlermaß (vorletzte Spalte) minimiert. Die letzte Spalte zeigt das Verhältnis des Fehlers des schlechtesten Parametersatzes (der hier nicht angegeben ist) zum optimalen.

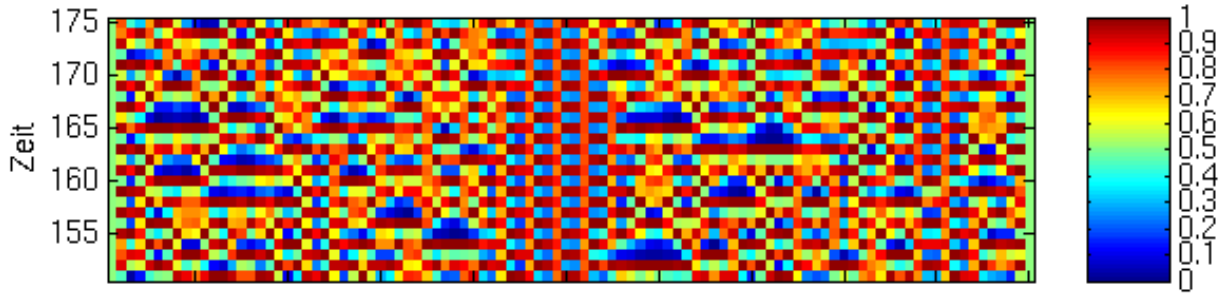
Vorhersage dargestellt, wobei der vorherzusagende Teil der Originalzeitreihe nicht in den Trainingsdatensatz der Prädiktion einging. Die Länge der Vorhersage beträgt für beide Systeme 25 Zeitschritte.

Das an sich triviale Beispiel der gekoppelten logistischen Abbildungen zeigt, daß die Crossvalidierung die zur Beschaffenheit der Systemgleichungen richtigen Parameter mit nur $I = 1$ einem räumlichen Nachbarn findet. Trotzdem zeigt die Vorhersage nur auf den ersten 10 Zeitschritten eine befriedigende Übereinstimmung mit der Originalzeitreihe. Interessant ist der räumlich variierende Prädiktionshorizont, der sich in dem eher periodischen Segment in der räumlichen Mitte der Zeitreihe über alle Zeitschritte der Vorhersage erstreckt.

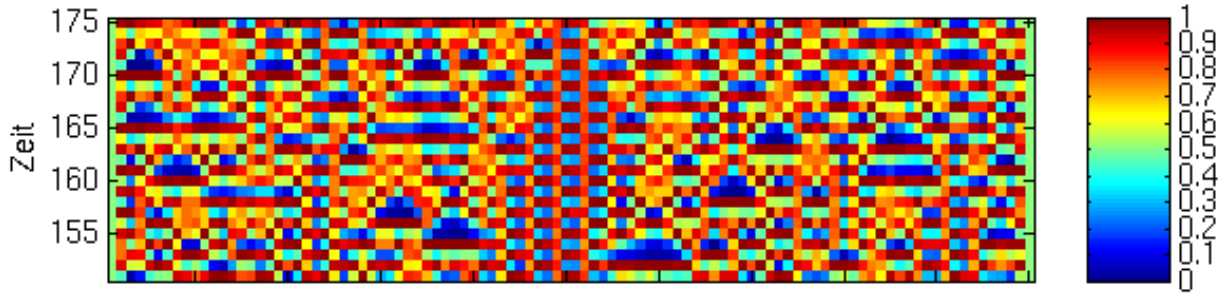
Während die optimalen Parameter zur Rekonstruktion lokaler Zustände im ersten Beispiel offensichtlich sind, können sie im Fall der Kuramoto–Sivashinsky PDE nicht direkt der Systemgleichung entnommen werden. Trotzdem ist auch hier eine Vorhersage über mindestens 10 Zeitschritte hinweg möglich. Interessant ist hier, daß die qualitative Struktur der Dynamik erstaunlich gut reproduziert wird, es jedoch schwierig ist, den räumlichen Verlauf einer Störung über längere Zeit präzise vorherzusagen.



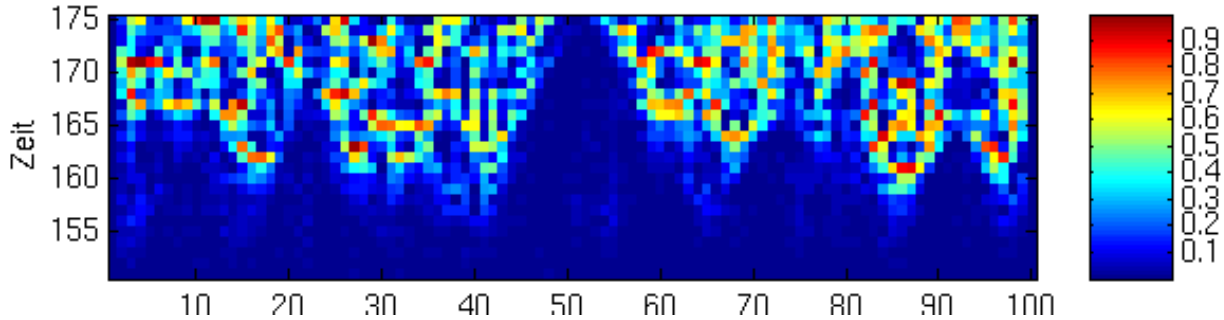
(a) Trainingsdatensatz der Länge 175



(b) Vorherzusagende Originalzeitreihe der Länge 25

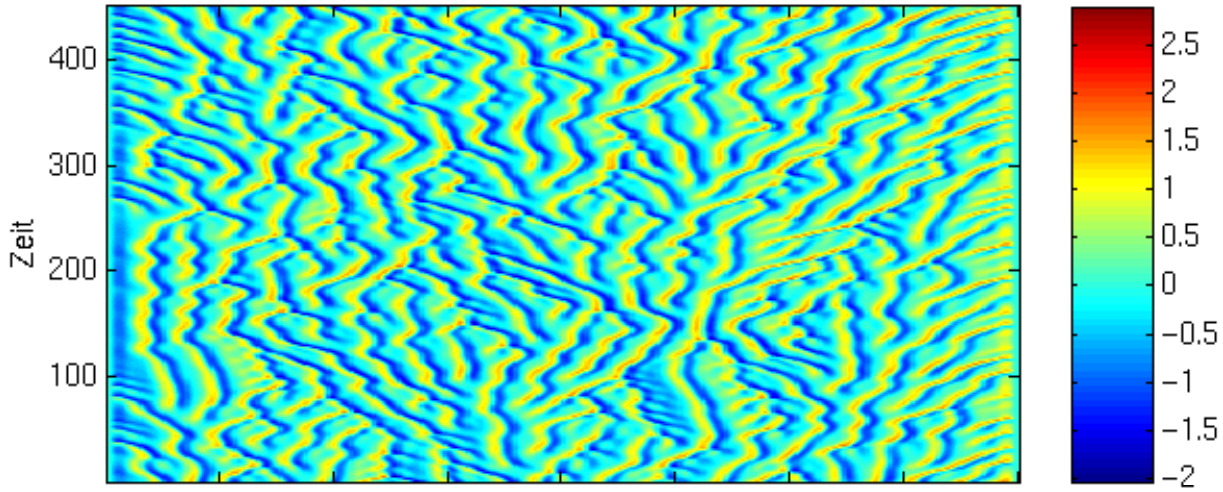


(c) Vorhersage

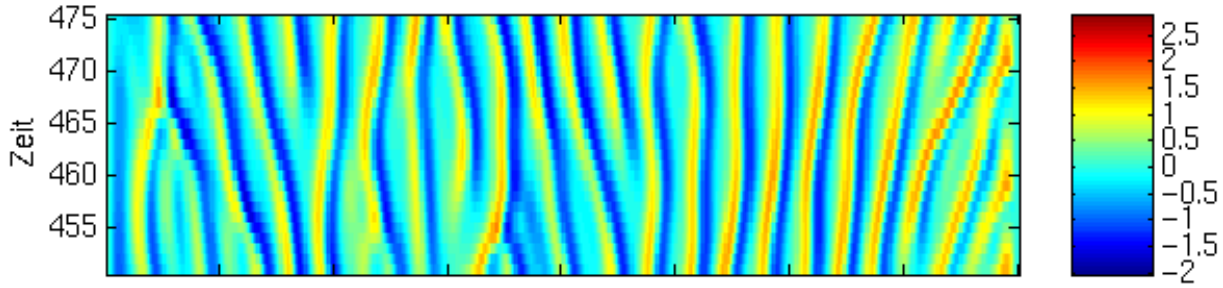


(d) Absolutwerte der Differenz zwischen Original und Vorhersage

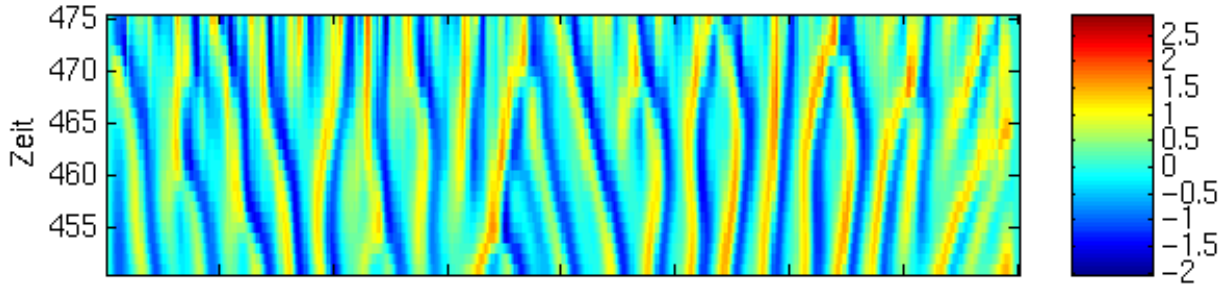
Abbildung 5.3.: Vorhersage des Datensatzes der gekoppelten logistischen Abbildungen unter Verwendung des während der Crossvalidierung ermittelten optimalen Parametersatzes (vgl. Tabelle 5.1). Unterabbildung (a) zeigt den Trainingsdatensatz mit 150 Zeitschritten, der von den 25 Zeitschritten des vorherzusagenden Testdatensatzes (b) fortgesetzt wird. Unterabbildung (c) stellt die Vorhersage mittels des in Abschnitt 5.4.1 beschriebenen Verfahrens dar. Der Absolutwert der Differenz zwischen Vorhersage und Testdatensatz ist in (d) wiedergegeben.



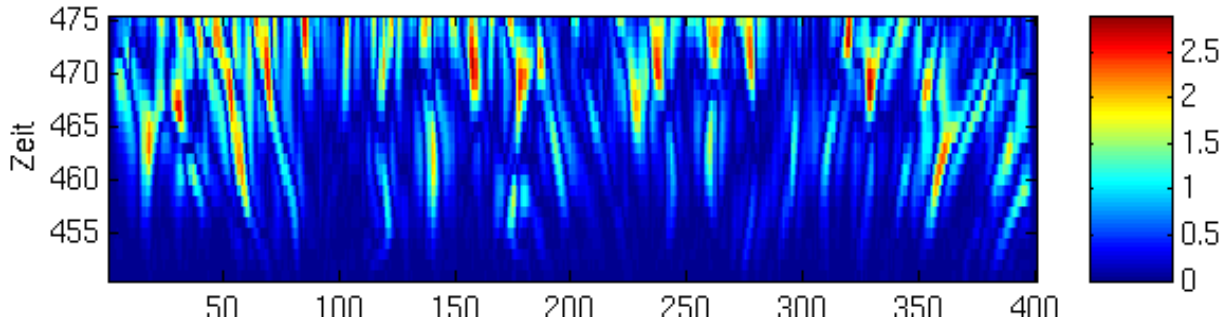
(a) Trainingsdatensatz der Länge 450



(b) Vorherzusagende Originalzeitreihe der Länge 25



(c) Vorhersage



(d) Absolutwerte der Differenz zwischen Original und Vorhersage

Abbildung 5.4.: Vorhersage des Kuramoto–Sivashinsky Datensatzes unter Verwendung des während der Crossvalidierung ermittelten optimalen Parametersatzes (vgl. Tabelle 5.1). Unterabbildung (a) zeigt den Trainingsdatensatz mit 450 Zeitschritten, der von den 25 Zeitschritten des vorherzusagenden Testdatensatzes (b) fortgesetzt wird. Unterabbildung (c) stellt die Vorhersage mittels des in Abschnitt 5.4.1 beschriebenen Verfahrens dar. Der Absolutwert der Differenz zwischen Vorhersage und Testdatensatz ist in (d) wiedergegeben.

6. Das Programmpaket TSTOOL

TSTOOL ist ein Programmpaket zur nichtlinearen Zeitreihenanalyse. Die Implementation erfolgte hauptsächlich in Matlab[1], allerdings wurden einige zeitkritische Methoden in der Programmiersprache C++ realisiert, diese stellen sich jedoch dem Anwender in gleicher Weise dar wie die rein in Matlab implementierten.

Das Aufbau des Programmpaketes läßt sich in drei Schichten einteilen:

(1) Die Ebene der grafischen Benutzeroberfläche (GUI). Diese bietet dem Benutzer die einfache Anwendung der bereitgestellten Methoden sowie komfortable Ein- und Ausgabe der Daten, Visualisierung derselben und die Möglichkeit, wiederholte Arbeitsfolgen mittels Macros zu automatisieren. Die Benutzeroberfläche selbst ist ein direkter Aufsatz auf die zweite Ebene:

(2) Die objekt–orientierte implementierte Ebene der Signalverarbeitung. Auf dieser wird die Programmierumgebung Matlab um den Datentyp des Signals erweitert (@*signal*), der ein physikalisches Signal nebst beschreibender Information modelliert. Diese Ebene ist direkt von der Matlab–Kommandozeile aus zugänglich und läßt sich daher in eigene Matlab–Programme oder Skripte einbinden. Obwohl die meisten Analysemethoden des TSTOOL in einer klassen–spezifischen Version für die Signal–Klasse zur Verfügung stehen (siehe Tabelle 6.1), rufen diese doch immer die entsprechenden Funktionen der dritten Ebene auf:

(3) Die Ebene der numerischen Verfahren. Nur auf dieser Ebene finden sich neben den rein in Matlab implemetierten Methoden auch in der Programmiersprache C++ realisierte

[1] Matlab ist eine kommerzielle, mathematische Programmierumgebung, die eine umfangreiche Sammlung mathematischer Routinen auf Skalaren, Vektoren und Matrizen sowie vielfältige Visualisierungsmöglichkeiten beinhaltet.

acf	Berechnung der normierten Autokorrelationsfunktion
amutual	Berechnung der Mutual–Information einer Zeitreihe gegen ihr zeitverschobenes Selbst
boxdim	Berechnung der Kapazitätsdimension D_0 mittels eines Box–counting Verfahrens
cao	Berechnung der Mutual–Information einer Zeitreihe gegen ihr zeitverschobenes Selbst
corrdim	Berechnung der Korrelationsdimension D_2 mittels eines Box–counting Verfahrens
corrsum	Berechnung der Korrelationssumme mit vorgegebener Anzahl Referenzpunkte
corrsum2	Berechnung der Korrelationssumme mit adaptiver Wahl der Referenzpunkte
embed	Rekonstruktion mit Verzögerungskoordinaten
fracdims	Schätzung des Spektrums der Rényi–Dimensionen aus der Skalierung der Nachbardistanzen
infodim	Berechnung der Informationsdimension D_1 mittels eines Box–counting Verfahrens
infodim2	Berechnung der Informationsdimension D_1 aus der Skalierung der Nachbardistanzen
largelyap	Bestimmung des größten Lyapunovexponenten
localdensity	Bestimmung der lokalen Dichte
nearneigh	Nächste–Nachbar Statistik
pca	Hauptachsentransformation für vektorielle Zeitreihen
poincare	Poincareschnitt
predict	Lokale Modellierung und Vorhersage für multivariate Zeitreihen
predict2	Lokale Modellierung und Vorhersage für skalare Zeitreihen
rang	Transformation einer skalaren Zeitreihe auf Rangzahlen
return_time	Bestimmung des Histogrammes der Wiederkehrzeiten für multivariate Zeitreihen
surrogate*	Erzeugen von Surrogatdaten
takens_estimator	Takens's Schätzer der Korrelationsdimension D_2
upsample	Erhöhung der Abtastrate durch Interpolation der Zeitreihe

Tabelle 6.1.: Auswahl der Methoden der Signal–Klasse des TSTOOL

(siehe Tabelle 6.2). Die Methoden dieser Ebene lassen sich bei Bedarf auch unabhängig von den darüberliegenden Ebenen verwenden. Übereinstimmende Einträge in den Tabellen 6.1 und 6.2 sind darauf zurückzuführen, daß eine Methode der zweiten Ebene die Daten eines Signal–Objektes zur numerischen Bearbeitung meist an die entsprechende Methode der dritten Ebene weiterreicht. Auf diese Weise soll eine systematische Trennung der eher verwaltenden Funktionalität der Signal–Klasse von der Implementation der numerischen Algorithmen erreicht werden.

akimaspline	Interpolation mittels Akima-Splines
amutual	Berechnung der Mutual–Information einer Zeitreihe gegen ihr zeitverschobenes Selbst
baker	Berechnung der iterierten Baker–Abbildung
boxcount	Berechnung von D_0, D_1 und D_2 mittels eines Box–counting Verfahrens
cao	Bestimmung der minimalen Einbettungsdimension mittels der Methode von Cao
chaosys	ODE–Löser für einige häufig verwendete chaotische Systeme, u.A. Lorenz, Chua, Rössler etc.
corrsum	Berechnung der Korrelationssumme $C(r)$ für eine vorgegebene Menge von Referenzpunkten
corrsum2	Berechnung der Korrelationssumme $C(r)$ mit adaptiver Auswahl der Referenzpunkten
fnearneigh	Berechnung der k–Nächsten–Nachbarn mittels des ATRIA–Algorithmus
gendimest	Schätzung des Spektrums der Rényi–Dimensionen aus den Distanzen der Nächsten–Nachbarn
henon	Berechnung der iterierten Henon–Abbildung
largelyap	Bestimmung des größten Lyapunovexponenten
nn_prepare	Ausgabe der Vorverarbeitungstruktur des ATRIA–Algorithmus
nn_search	Berechnung der k–Nächsten–Nachbarn, benötigt die von nn_prepare erzeugte Vorverarbeitungstruktur
predict	Lokale Modellierung und Vorhersage für Zustandsvektoren
predict2	Lokale Modellierung und Vorhersage für skalare Zeitreihen mittels Methode der Zeitverzögerungskoordinaten
range_search	Bereichssuche, benötigt die von nn_prepare erzeugte Vorverarbeitungstruktur
return_time	Bestimmung des Histogrammes der Wiederkehrzeiten für Zustandsvektoren
takens_estimator	Takens's Schätzer der Korrelationsdimension D_2
tentmap	Berechnung der iterierten Zeltabbildung

Tabelle 6.2.: In C++ implementierte Methoden des TSTOOL

7. Ausblick

Nächste–Nachbar basierte Methoden zeigen ein erstaunlich vielfältiges Anwendungsspektrum in der nichtlinearen Zeitreihenanalyse. Dort dienen sie zum einen zum Lernen eines in Form gemessener Daten gegebenen Urbild–Bild Zusammenhangs, zum anderen zur Bestimmung charakterischer Größen der zugrundeliegenden dynamischen Systeme wie z.B. Lyapunov–Exponenten [32] oder, wie in dieser Arbeit vorgestellt, dem Spektrum der fraktalen Dimensionen.

Lokale Modellierung

Die lokale Modellierung unter Verwendung der Nächsten–Nachbarn bietet dabei noch weiteren Raum für Verbesserungen. Zu kann z.B. eine Vorauswahl der Komponenten der Zustandsvektoren mit dem Ziel erfolgen, nur diejenigen Komponenten zu selektieren, die zu einer Verbesserung des Vorhersagefehlers beitragen. Auch ein Auswahlverfahren, das ein Maß der gegenseitigen Information zwischen Bild und Komponente des Urbildes optimiert, ist denkbar. Weiter kann sowohl die Anzahl der Nachbarn als auch die verwendete Metrik an die jeweilige Aufgabenstellung angepaßt werden.

Lokale Modellierung ist nicht auf die Verwendung lokal konstanter Modelle angewiesen. Lokal lineare Modelle erfordern nur geringfügig höheren Rechenaufwand. Aufwendiger, aber vielversprechend ist eine Verbindung der lokalen Modellierung mit Support–Vector–Methoden.

Nächste–Nachbar–Suche

Obwohl der vorgestellte ATRIA durchaus gute Ergebnisse beim Einsatz in der NLZ zeigt, sind weitere Verbesserungen z.B. dadurch möglich, daß Nächste–Nachbar–Algorithmen speziell für den Einsatz mit einer vorgegebenen Metrik entworfen werden. Auf diese Weise können

dichtere untere Schranken der Distanz berechnet werden, so daß solche Algorithmen Nachbarn mit geringerem Suchaufwand bestimmen können.

Modellierung raum–zeitlicher dynamischer Systeme

Die Ergebnisse des Abschnittes über die Modellierung raum–zeitlicher dynamischer Systeme zeigen, daß die vorgestellte Methode der Rekonstruktion lokaler Zustände einen vielversprechenden Ansatz zur Modellierung raum–zeitlicher Systeme darstellt, mit dem die Dynamik raum–zeitlicher System auf kurzen Zeitskalen durchaus zufriedenstellend vorhergesagt werden kann. Dabei ist die Methode nicht auf die Verwendung der Nächsten–Nachbarn zur Modellierung der Dynamik festgelegt, auch wenn diese Art der Funktionsapproximation gerade im Hinblick auf die Modellvalidierung Vorteile aufweist. So könnte die Modellierung auch mit Hilfe von Polynommodellen, radialen Basisfunktionen oder neuronalen Netzwerken erfolgen.

Eine Verallgemeinerung des besprochenen Verfahrens kann in verschiedene Richtungen erfolgen. So kann zur weiteren Verringerung des Rechenaufwandes die Dimension der rekonstruierten Zustände weiter reduziert werden, falls die STTS von einem System stammt, von dem bekannt ist, daß es zusätzliche räumliche Symmetrien aufweist. Im eindimensionalen Fall kann dies z.B. bei Systemen, deren Dynamik invariant bezüglich räumlicher Spiegelung ist, ausgenutzt werden. Auch kann für raum–zeitliche Systeme mit mehreren räumlichen Dimension, die einer inhomogenen Dynamik unterliegen, die Anzahl der Penalty–Komponente an die Zahl räumlicher Dimensionen angeglichen werden. Allerdings erhöht sich dabei gleichzeitig auch die benötigte Anzahl Trainingsdatenpunkte. Weiter muß die Region, aus der die Komponenten der lokalen Zustände gebildet werden, nicht unbedingt ein rechteckiges Gitter bilden. Motiviert durch die endliche Geschwindigkeit der Informationsausbreitung, kann eine spezifische Auswahl der Komponenten der STTS derart erfolgen, daß nur solche in den Zustandsvektor aufgenommen werden, die einen deterministischen Einfluß auf den vorherzusagenden Wert aufweisen. Dabei könnte der Vorhersagefehler, ähnlich zu der Prädiktion der skalaren Zeitreihen in Abschnitt 2.4.2, durch Wahl einer angepassten Metrik noch weiter reduziert werden.

Obwohl eine rigorose theoretische Fundierung des Verfahrens zur Rekonstruktion lokaler Zustände noch aussteht, kann die vorgestellte Methode als Ausgangspunkt für eine weiterreichende mathematische Behandlung solcher Systeme dienen.

A. Anhang

A.1. Beweise

A.1.1. Ausschlußregel

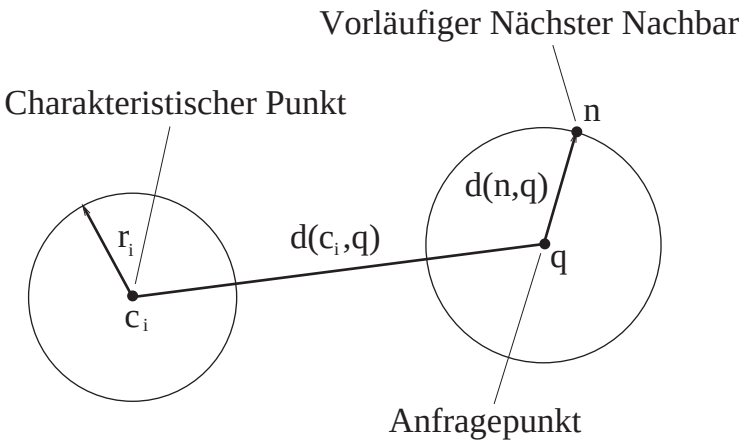


Abbildung A.1.: Illustration für Beweis 2

Beweis 2 *Aus der Konstruktion ist bekannt, daß* $\forall x$ *des aktuellen Clusters* C_i *(siehe Abb. A.1) gilt:*

$$d(c_i, x) \leq R_i$$

Aus der Dreiecksungleichung folgt:

$$d(q, c_i) \leq d(c_i, x) + d(q, x)$$

$$d(q, c_i) \leq R_i + d(q, x) \Rightarrow d(q, c_i) - R_i \leq d(q, x)$$

A.1.2. Abschätzung von $\hat{d}_{min}$

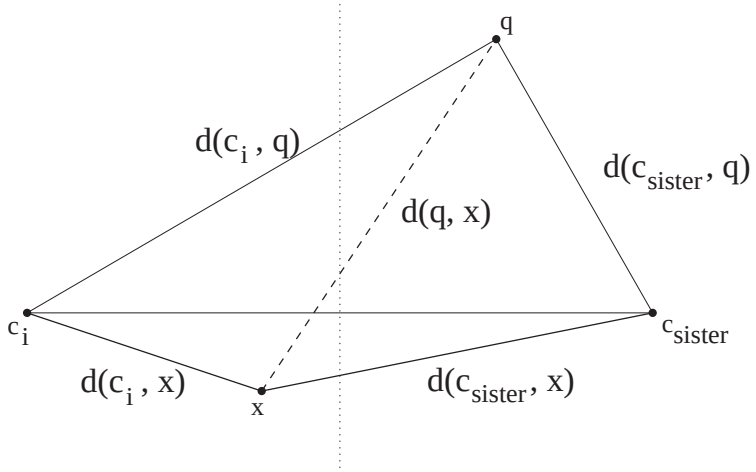


Abbildung A.2.: Illustration für Beweis 3

Beweis 3 *Aus der Konstruktion des Baumes ist bekannt, daß* $\forall x$ *des aktuellen Clusters* C_i *(siehe Abb. A.2) gilt:*

$$d(c_i, x) + g_i \leq d(c_{sister}, x) \tag{A.1}$$

Aus der Dreiecksungleichung folgt:

$$d(c_{sister}, x) \leq d(q, c_{sister}) + d(q, x) \tag{A.2}$$

$$d(c_i, q) \leq d(c_i, x) + d(q, x) \tag{A.3}$$

$$(A.1) + (A.2) \Rightarrow d(c_i, x) + g_i \leq d(q, c_{sister}) + d(q, x) \tag{A.4}$$

$$(A.3) \Rightarrow d(c_i, q) - d(q, x) \leq d(c_i, x) \tag{A.5}$$

$$(A.4) + (A.5) \Rightarrow d(c_i, q) - d(q, c_{sister}) + g_i \leq 2d(q, x)$$

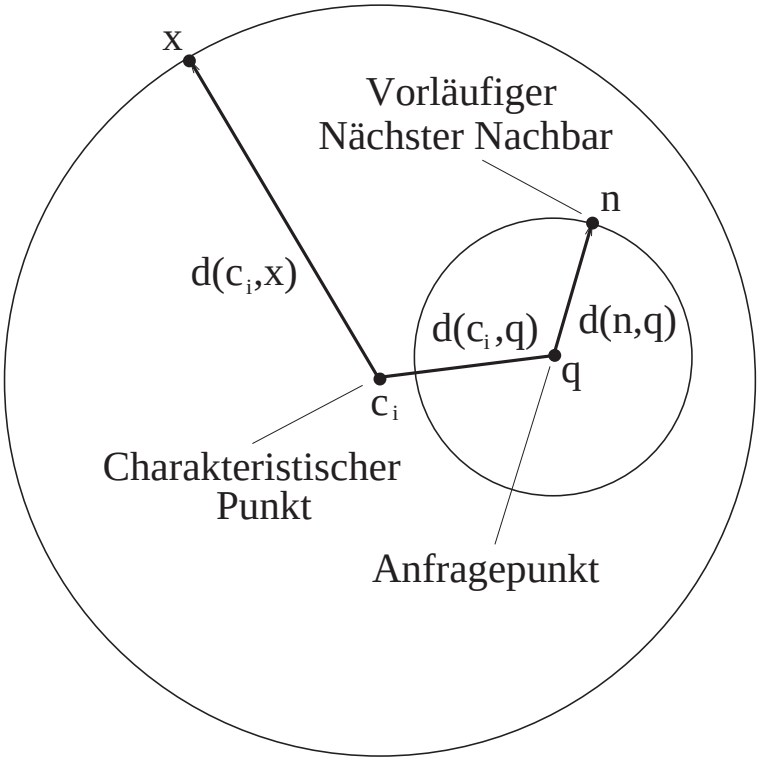


Abbildung A.3.: Illustration für Beweis 4

A.1.3. Ausschlußregel für Punkte innerhalb eines endständigen Knotens

Beweis 4 *Aus der Dreiecksungleichung ist bekannt, daß $\forall x$ des aktuellen Clusters C_i (siehe Abb. A.3) gilt:*

$$d(c_i, x) \leq d(c_i, q) + d(q, x)$$
$$d(c_i, q) \leq d(c_i, x) + d(q, x)$$

$$\Rightarrow |d(c_i, q) - d(c_i, x)| \leq d(q, x)$$

A.1.4. Maximale Interpunktdistanz in einer endlichen Punktmenge

Sei X eine nicht-leere Menge von Punkten. Für einen bestimmten Punkt $a \in X$ sei bekannt : $d(a,x) \leq R \quad \forall x \in X$. Dann folgt daraus in jedem metrischen Raum, daß für zwei beliebige Punkte $b, c \in X$ gilt : $d(b,c) \leq 2R$.

Beweis 5

$$b \in X \Rightarrow d(a,b) \leq R$$
$$c \in X \Rightarrow d(a,c) \leq R$$

Aus der Dreiecksungleichung folgt :

$$d(a,b) + d(a,c) \geq d(b,c)$$
$$\Rightarrow R + R \geq d(b,c)$$

A.2. Die Gammafunktion

Die Gammafunktion $\Gamma : (0,\infty) \to \mathbb{R}$ zählt zu den wichtigeren der höheren Funktionen. Sie ist wie folgt definiert:

$$\Gamma(x) := \int_0^\infty e^{-t} t^{x-1}\, dt$$

Aus der *Funktionalgleichung* der Gammafunktion

$$\Gamma(x+1) = x\Gamma(x) \quad \text{für} \quad x > 0, \tag{A.6}$$

folgt mit $\Gamma(1) = \int_0^\infty e^{-t}\, dt = 1$ für alle $n \in \mathbb{N}$:

$$\Gamma(n+1) = n\,\Gamma(n) = n(n-1)\,\Gamma(n-1) = n!\,\Gamma(1) = n!$$

A.3. Die Poissonverteilung

Definition 4 *Eine Zufallsgröße heißt poissonverteilt mit dem Parameter λ, wenn sie abzählbar unendlich vielen mögliche Werte $k = 0, 1, 2, \ldots$ mit den Wahrscheinlichkeiten*

$$P_k(\lambda) = \frac{\lambda^k}{k!} e^{-k}$$

annimmt.

Es gilt:

$$\begin{aligned} \sum_{k=0}^{\infty} P_k(\lambda) &= 1 \\ \sum_{k=0}^{\infty} k P_k(\lambda) &= \lambda \end{aligned}$$

Der Parameter λ, der gleichzeitig den Erwartungswert der Verteilung darstellt (vgl. Gleichung A.7), kann als Produkt einer großen Anzahl n von (gleichartigen) Zufallsexperimenten mit kleiner Erfolgswahrscheinlichkeit p aufgefaßt werden: $\lambda = np$. In diesem Fall kann die Poissonverteilung als gute Näherung der Binomialverteilung

$$P_k(\lambda = np) \approx \binom{n}{k} p^k (1-p)^{n-k} \quad k = 0, 1, \ldots, n$$

benutzt werden.

A.4. Quelltexte

Obwohl das Einfügen von Quelltexten in eine wissenschaftliche Arbeit oft als Lückenbüßer dient und daher manchmal mit entsprechendem Vorbehalt betrachtet wird, ist es in diesem Fall doch für das Nachvollziehen des im Kapitel 3 beschriebenen Algorithmus von großem Nutzen, läßt doch die Beschreibung eines Algorithmus in Worten unterschiedliche Implementation zu, die sich leider in den tatsächlichen Laufzeiten deutlich unterscheiden können. Die wiedergegebenen Quelltexte wurden dem Softwarepaket TSTOOL (siehe Kapitel 6) entnommen, wobei dem Verständnis überflüssige oder ablenkende, eher technisch motivierte Passagen (z.B. Fehlerbehandlung, Objektpersistenz, Profiling etc.) entfernt wurden. Nichtsdestotrotz lassen sich die Quelltexte in der angegebenen Form mit einem ANSI–C++ konformen Compiler (z.B. gcc 2.95.2) übersetzen und bieten die volle Funktionalität der im Text beschriebenen Algorithmen an. Obowhl der Nächste–Nachbar Algorithmus an einer Vielzahl verschiedenster Datensätze bei Verwendung unterschiedlicher Metriken getestet wurde, kann hier keine Garantie für Vollständigkeit oder Korrektheit der dargestellten Implementation geleistet werden.

A.4.1. Beispielimplementation des ATRIA

```
#include <math.h>
#include <iostream.h>
#include <stack.h>
#include <vector.h>
#include <algo.h>

// INFINITY is needed as an upper bound for every
// distance that might be encountered during search
// So be carefull : if the data set has a very unusual scaling,
// actual distances might exceed this value, which would lead
// to wrong results.

#define INFINITY DBL_MAX

// This header file defines templated function objects (functors) for distance
// calculations. They work on  any reasonable container class by using forward
// iterators. For good performance, no bound checking is done,  so it's the
// programmer's resposibility to have everything properly allocated.
//
// To support PARTIAL DISTANCE CALCULATION during the search phase (not the
// preprocessing phase!), operator()  is overloaded. operator() with three arguments is
// the standard distance calculation and returns the exact distance. operator() with
// four arguments is the tresholded distance calculation with terminates as soon as the
// partial distance exceeds the given threshold value (but otherwise returns the exact
// distance). Carefull implementation can speed up the search considerably. When
// threshold is exceeded, INFINITY is returned to prevent the search functions from
// considering this value as an exact distance.

#define PARTIAL_SEARCH

// Parameters for the ATRIA nearest neighbor search:
// A cluster will not be further subdivided if it contains
// less than ATRIAMINPOINTS points. A smaller value might
// accelerate search time, but increase preprocessing time.

#define ATRIAMINPOINTS 64

class euclidian_distance {
  public:
    euclidian_distance() {};
```

```
    template <class ForwardIterator1, class ForwardIterator2>
    double operator() (ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2) const
    {
      const double y = (*first1 - *first2);
      double dist = y*y;
        for (++first1, ++first2 ; first1 != last1; ++first1, ++first2) {
        const double x = (*first1 - *first2);
        dist += x * x;
      }
        return sqrt(dist);
    }

    // support partial search : if partial distance exceeds thresh,
    // stop computing of distance
    template <class ForwardIterator1, class ForwardIterator2>
    double operator() (ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, const double thresh) const
    {
      const double t = thresh * thresh;
      const double y = (*first1 - *first2);
      double dist = y*y;

      if (dist > t)
        return INFINITY;

        for (++first1, ++first2 ; first1 != last1; ++first1, ++first2) {
        const double x = (*first1 - *first2);
        dist += x * x;
        if (dist > t)
          return INFINITY;
      }
        return sqrt(dist);
    }
};

class maximum_distance {
  public:
    maximum_distance() {};
    template <class ForwardIterator1, class ForwardIterator2>
    double operator() (ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2) const
    {
      double dist = fabs(*first1 - *first2);
        for (++first1, ++first2 ; first1 != last1; ++first1, ++first2) {
        const double x = fabs(*first1 - *first2);
        if (x > dist) dist = x;
      }
        return dist;
    }
    // support partial search
    template <class ForwardIterator1, class ForwardIterator2>
    double operator() (ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, const double thresh) const
    {
      double dist = fabs(*first1 - *first2);
      if (dist > thresh) {
        return INFINITY;
      }
        for (++first1, ++first2 ; first1 != last1; ++first1, ++first2) {
        const double x = fabs(*first1 - *first2);
        if (x > dist) {
          if (x > thresh) {
            return INFINITY;
          }
          dist = x;
        }
      }
        return dist;
    }
};

// class interleaved_pointer  : a smart pointer that iterates with an interleave
// of #inc elements over an array
// May be used to iterate over rows or columns of a dense matrix of type T
template<class T>
class interleaved_pointer
{
  private:
    const T* ptr;
    const long increment;
  public:
    typedef interleaved_pointer self;
    interleaved_pointer(const T* const p, const long inc)
      : ptr(p), increment(inc) {};
```

```
    T operator*() const { return *ptr; }
    self& operator++() {
        ptr+=increment;
        return *this;
      }
      self& operator--(){
        ptr-=increment;
        return *this;
      }
    bool operator==(const interleaved_pointer& x) {
      return ptr == x.ptr;
    }
    bool operator!=(const interleaved_pointer& x) {
      return ptr != x.ptr;
    }
};

// The next class models a set of points. Points can be accessed by an
// index, ranging from zero to N-1. The implementation of the points is not important
// as long as the class provides the possibility to calculate distances between two
// points of the point set and the distance between a point from the data set and an
// externally given point. Class point_set is parametrized by the METRIC that is used
// to compute distances. METRIC must be a class having an operator(). For possible
// implementations of a METRIC, see above. This point_set implementation is the
// standard tstool way of handling matlab matrix data
// The particular implementation can be used for use with Matlab mex-files, where
// a point set is given as a Fortran style matrix (column major). The coordinates of
// a point is represented by one row of this matrix
template <class METRIC>
class point_set  {
  protected:
    const long N;    // number of points
    const long D;    // formal dimension of points

    // points are stored as row vectors of a N by D fortran style matrix
    const double* const matrix_ptr;

    // a function object that calculates distances
    const METRIC Distance;

  public:
    typedef METRIC Metric;  // export metric type

    point_set(const long n, const long d, const double* mat)
      : N(n), D(d), matrix_ptr(mat), Distance() {};
    point_set(const long n, const long d, const double* mat, const METRIC& metr)
      : N(n), D(d), matrix_ptr(mat), Distance(metr) {};

    ~point_set() {};
    long dimension() const { return D; }
    long size() const { return N; }

    // a smart pointer that iterates over the elements of one point
    // in this point_set (points are row vectors)
    typedef interleaved_pointer<double> row_iterator;

    row_iterator point_begin(const long n) const {
      return row_iterator(matrix_ptr + n, N);
    }

    row_iterator point_end(const long n) const {
      return row_iterator(matrix_ptr + n + N*D, N);
    } // STL like past-the-end iterator

    // indices may vary between 0 and N-1, or 0 and D-1
    // vec2 must be a double vector of length D, vec1[0] ... vec1[D-1]
    double coordinate(const long n, const long d) const {
      return matrix_ptr[n + N*d];
    }

    template<class ForwardIterator>
    double distance(const long index1, ForwardIterator vec2) const
    {
      return Distance(point_begin(index1), point_end(index1), vec2);
    }

#ifdef PARTIAL_SEARCH
    template<class ForwardIterator>
    double distance(const long index1, ForwardIterator vec2, const double thresh) const
    {
      return Distance(point_begin(index1), point_end(index1), vec2, thresh);
    }
#endif

    double distance(const long index1, const long index2) const
```

```
    {
      return Distance(point_begin(index1), point_end(index1),
        point_begin(index2));
    }
};

// A "neighbor" is described by its index (into the point set) and
// a distance value. This class is also used for the permutation table
// of class ATRIA, where it does not exactly specify a neighbor, so
// the name "neighbor" may be a little bit misleading.
class neighbor
{
  protected:
    long i;      // index of neighbor (relative to point set)
    double d;    // distance
  public:
    neighbor() {};
    neighbor(const long I, const double D) : i(I), d(D) {};
    long index() const { return i; };
    double dist() const { return d; };
    long& index() { return i; }
    double& dist() { return d; }
    bool operator<(const neighbor& x) const { return d < x.d; }
};

// function object to sort neighbors by distance
class neighborCompare : public binary_function<neighbor, neighbor, bool> {
  public:
      bool operator()(const neighbor& x, const neighbor& y) const {
      return x.dist() < y.dist();
    }
};

// The next class models the table of preliminary neighbors m_1, ..., m_k
class SortedNeighborTable {
  protected:
    long K;    // number of neighbors to be searched
    double hd;     // cache highest distance

    priority_queue<neighbor, vector<neighbor> , neighborCompare > pq;

  public:
    SortedNeighborTable() : K(1), hd(INFINITY) {};
    SortedNeighborTable(const long k) : K(k), hd(INFINITY) {};
   ~SortedNeighborTable() {};

    double highdist() const { return hd; };
    void insert(const neighbor& x);

    void init_search(const long k) { K = k; hd = INFINITY; }
    long finish_search(vector<neighbor>& v);
};

class cluster {
  public:
    long charac_point;  // index of characteristic point for this cluster
              // (points direcly into the point set)

    double Rmax;     // Rmax is by definition >= 0. So we use its
              // sign as a flag to indicate whether we have
              // a terminal node or not. If Rmax <= 0 we have
              // a terminal node. See member functions
              // is_terminal() and R_max() below

    double g_min;

    union {
      cluster* left;  // used in case of a non-terminal node
      long start;   // used in case of a terminal node, points
              // into the permutation_table
    };
    union {
      cluster* right;   // used in case of a non-terminal node
      long length;    // used in case of a terminal node
    };

    cluster()
      : charac_point(0), Rmax(INFINITY), g_min(0), start(0), length(0)  {};
    cluster(const long c)
      : charac_point(c), Rmax(INFINITY), g_min(0), start(0), length(0) {};
    cluster(const long s, const long l, const long c = 0)
      : charac_point(c), Rmax(INFINITY), g_min(0), start(s), length(l)  {};

   ~cluster() {};
```

```
    int is_terminal() const { return (Rmax <= 0); };
    double R_max() const { return fabs(Rmax); };
};

// During k-nearest neighbor search, clusters are inserted in form of "searchitems"
// into the priority queue or stack.
class searchitem {
  protected:
    const cluster* c;   // pointer to cluster object
    double d;      // distance from query point to the cluster's charac_point
    double dbrother;  // distance from query point to brother cluster's charac_point

    double dmin;     // estimate of miniumum distance from query point
              // to any point inside  cluster c (\hat{d}_{min})

    double dmax;     // estimate of maximal distance from query point to
              // any point  nside cluster
  public:
    searchitem() {};

    searchitem(const cluster* C, const double D)
    : c(C), d(D), dbrother(INFINITY), dmin(D - C->R_max()), dmax(D+C->R_max()) {};

    searchitem(const cluster* C, const double D, const double Dbrother,
          const searchitem& parent)
    : c(C), d(D), dbrother(Dbrother),
      dmin(max(max(0.0, 0.5*(D-Dbrother+c->g_min)), max(D - C->R_max(), parent.dmin))),
      dmax(min(parent.dmax, D + C->R_max())) {};

    const cluster* clusterp() const { return c; };
    double dist() const { return d; };
    double dist_brother() const { return dbrother; };

    double d_min() const { return dmin; };
    double d_max() const { return dmax; };
};

// function object to sort searchitems by dmin
class searchitemCompare : public binary_function<searchitem, searchitem, bool> {
  public:
    bool operator()(const searchitem& x, const searchitem& y) const {
      if (x.d_min() == y.d_min())
        return x.d_max() > y.d_max();
      else
        return x.d_min() > y.d_min();
    };
};

// base class for the nearest neihbor search which defines a common interface
template<class POINT_SET>
class nearneigh_searcher {
  protected :
    const POINT_SET& points;
    const long Nused;        // number of points of the data set actually used

    SortedNeighborTable table;

    template<class ForwardIterator>
    void test(const long index, ForwardIterator qp, const double thresh) {
#ifdef PARTIAL_SEARCH
      const double d = points.distance(index, qp, thresh);
#else
      const double d = points.distance(index, qp);
#endif
      if (d < thresh)
        table.insert(neighbor(index,d));
    }

  public :
    typedef POINT_SET point_set;  // export type

    // prepare searching for a point set points
    // excl gives the number of points that should be
    // excluded from searching (from the end of the point set)
    nearneigh_searcher(const POINT_SET& p, const long excl = 0);
    ~nearneigh_searcher() {};
};


// ATRIA : A class that implements an advanced triangle inequality algorithm.
template<class POINT_SET>
class ATRIA : public nearneigh_searcher<POINT_SET> {
  protected :
    const long MINPOINTS;
    cluster root;
```

```
    neighbor* const permutation_table;  // this table contains the indices
                     // of all points in POINT_SET, as well
                     // as distances the charac_point of the
                     // cluster where each point falls into

    typedef typename POINT_SET::Metric METRIC;
    typedef searchitem SearchItem;

    priority_queue<SearchItem, vector<SearchItem>, searchitemCompare> search_queue;
    stack<SearchItem, vector<SearchItem> > SearchStack;

    void create_tree();
    void destroy_tree();

    pair<long, long> find_child_cluster_charac_points(const cluster* const c,
            neighbor* const Section, const long c_length);

    long assign_points_to_charac_points(neighbor* const Section, const long c_length,
             pair<cluster*, cluster*> childs);

    template<class T>
    static void swap(T* a, const long i, const long j) {
      T tmp = a[i]; a[i] = a[j]; a[j] = tmp;
    }
  public:
    ATRIA(const POINT_SET& p, const long excl = 0, const long minpts = ATRIAMINPOINTS);
    ~ATRIA();

    // Search for k nearest neighbors of the point query_point,
    // excluding all points with indices between first and last from the search.
    // Returns number of nearest neighbor found and a sorted vector of neighbors
    // (by reference). An error in search_k_neighbors() will not result
    //   in an errorstate for the searcher ( see geterr() )
    template<class ForwardIterator>
    long search(vector<neighbor>& v, const long k, ForwardIterator query_point,
        const long first = -1, const long last = -1, const double epsilon = 0);

    // search (and count) number of points within distance 'radius' from the
    // query point. An unsorted vector v of neigbors is returned.
    template<class ForwardIterator>
    long search_range(vector<neighbor>& v, const double radius, ForwardIterator query_point,
            const long first = -1, const long last = -1);
};


long SortedNeighborTable::finish_search(vector<neighbor>& v)
{
  v.reserve(pq.size());

  while (!pq.empty()) {
    v.push_back(pq.top());
    pq.pop();
  }

  reverse(v.begin(),v.end());
  return v.size();
}

void SortedNeighborTable::insert(const neighbor& x)
{
  pq.push(x);

  while(pq.size() > K) {
    pq.pop();
  }

  if (pq.size() < K) {
    hd = INFINITY;
  }
  else {
    hd = pq.top().dist();
  }
}

template<class POINT_SET>
nearneigh_searcher<POINT_SET>::nearneigh_searcher(const POINT_SET& p, const long excl)
  : points(p), Nused(points.size() - excl)
{
  if ((Nused < 1) || (excl < 0))
  {
    cerr << "Wrong parameters for nearest neighbour search" << endl;
    return;
  }
}
```

A. Anhang

```
template<class POINT_SET>
ATRIA<POINT_SET>::ATRIA(const POINT_SET& p, const long excl, const long minpts)
  : nearneigh_searcher<POINT_SET>(p, excl), root(1,Nused-1), MINPOINTS(minpts),
    permutation_table(new neighbor[Nused])
{
  create_tree();
}

template<class POINT_SET>
ATRIA<POINT_SET>::~ATRIA()
{
  destroy_tree();
  delete[] permutation_table;
}

template<class POINT_SET>
pair<long, long> ATRIA<POINT_SET>::find_child_cluster_charac_points(const cluster* const c,
        neighbor* const Section, const long length)
{
  pair<long, long> charac_points(-1, -1);

  if (c->Rmax == 0) {    // if all data points seem to be identical, indicate
    return charac_points;   // that there's no need to further divide this data set
  }

  long index = 0;
  long charac_point_right = Section[index].index();
  double dist = Section[index].dist();

  // Compute right charac_point, the point that is farthest away from the c->charac_point

  for (long i=1; i < length; i++) {
    const double d = Section[i].dist();
    if (d>dist) {
      dist = d;
      charac_point_right = Section[i].index();
      index = i;
    }
  }

  charac_points.second = charac_point_right;

  // move this charac_point the the last (rightmost) element of this Section
  swap(Section, index, length-1);

  // next compute left charac_point, the point that is farthest away
  // from the charac_point_right

  index = 0;
  long charac_point_left = Section[index].index();
  dist = points.distance(charac_point_right, Section[index].index());
  Section[index].dist() = dist;

  for (long i=1; i < length-1; i++) {
    const double d = points.distance(charac_point_right, Section[i].index());
    Section[i].dist() = d;
    if (d > dist) {
      dist = d;
      charac_point_left = Section[i].index();
      index = i;
    }
  }

  // move this charac_point the the first (leftmost) element of this Section
  swap(Section, index, 0);

  charac_points.first = charac_point_left;

  return charac_points;
}

template<class POINT_SET>
void ATRIA<POINT_SET>::create_tree() {
  stack<cluster*, vector< cluster* > > Stack;

  permutation_table[0] = neighbor(root.charac_point = 0, 0);

  root.Rmax = 0;
  for (long k=1; k < Nused; k++) {
    const double d = points.distance(k, root.charac_point);
    permutation_table[k] = neighbor(k, d);
    if (d > root.Rmax)
      root.Rmax = d;
  }
```

A. Anhang

```
  // now create the tree

  Stack.push(&root);  // push root cluster on Stack

  while(!Stack.empty())
  {
    cluster* const c = Stack.top(); Stack.pop();

    const long c_start = c->start;
    const long c_length = c->length;

    neighbor* const Section = permutation_table + c_start;

    if (c->length > MINPOINTS) {     // Further divide this cluster ?
      pair<long, long> new_child_charac_points = find_child_cluster_charac_points((const cluster*) c,
                          Section, c_length);

      if ((new_child_charac_points.first == -1) || (new_child_charac_points.second == -1)) {
        // cluster could not be divided further and has already been
        // marked as terminal cluster/node (see find_child_cluster_charac_points())
        continue;
      }
      c->left = new cluster(new_child_charac_points.first);
      c->right = new cluster(new_child_charac_points.second);

      // create two subclusters and set properties
      const long j = assign_points_to_charac_points(Section, c_length,
              pair<cluster*, cluster*>(c->left,c->right));

      c->left->start = c_start+1;   // leave charac_points out
      c->left->length = j-1;

      c->right->start = c_start + j;
      c->right->length = c_length - j - 1;

      // process new subclusters
      Stack.push(c->right);
      Stack.push(c->left);
    }
    else {  // this is going to be a terminal node
      c->Rmax = - c->Rmax;  // a Rmax value <= 0 marks this cluster as a terminal node of the search tree
    }
  }
}

// assign each point to the nearest charac_point, using a kind of quicksort like sorting procedure
template<class POINT_SET>
long ATRIA<POINT_SET>::assign_points_to_charac_points(neighbor* const Section, const long c_length,
            pair<cluster*, cluster*> childs)
{
  const long charac_point_left  = childs.first->charac_point;

  long i = 0;
  long j = c_length-1;

  // maximal distance fron one cluster's charac_point to points belonging to this cluster
  double Rmax_left = 0;
  double Rmax_right = 0;

  // minimal gap in distances to both cluster charac_points
  double g_min_left = INFINITY;
  double g_min_right = INFINITY;

  while(1) {
    short i_belongs_to_left = 1;
    short j_belongs_to_right = 1;

    while(i+1 < j) {
      i++;
      const double dl = points.distance(charac_point_left, Section[i].index());
      const double dr = Section[i].dist();

      if (dl > dr) {
        // point belongs to the right corner
        const double diff = dl - dr;

        Section[i].dist() = dr;
        i_belongs_to_left = 0;

        g_min_right = min(g_min_right, diff);
        Rmax_right = max(Rmax_right, dr);
        break;
      }

      // point belongs to the left corner
```

```
      const double diff = dr - dl;

      Section[i].dist() = dl;

      g_min_left = min(g_min_left, diff);
      Rmax_left = max(Rmax_left, dl);
    }

    while(j-1 > i) {  // either we reached the start of the array
            // or this element was already checked by the i loop
      --j;

      const double dr = Section[j].dist();
      const double dl = points.distance(charac_point_left, Section[j].index());

      if (dr >= dl) {
        // point belongs to the left corner
        const double diff = dr - dl;

        Section[j].dist() = dl;
        j_belongs_to_right = 0;

        g_min_left = min(g_min_left, diff);
        Rmax_left = max(Rmax_left, dl);
        break;
      }

      // point belongs to the right corner
      const double diff = dl - dr;

      Section[j].dist() = dr;

      g_min_right = min(g_min_right, diff);
      Rmax_right = max(Rmax_right, dr);
    }

    if (i == j-1){
      if ((!i_belongs_to_left) && (!j_belongs_to_right)) {
        swap(Section, i,j);
      } else if (!i_belongs_to_left) {
        i--;j--;
      } else if (!j_belongs_to_right) {
        i++; j++;
      }
      break;  // finished walking through the array
    } else {
      swap(Section, i,j);
    }
  }

  childs.first->g_min = g_min_left;
  childs.first->Rmax = Rmax_left;

  childs.second->g_min = g_min_right;
  childs.second->Rmax = Rmax_right;

  return j;
}

template<class POINT_SET>
void ATRIA<POINT_SET>::destroy_tree()
{
  stack<cluster*, vector< cluster* > > Stack;

  if (!root.is_terminal()) {
    Stack.push(root.left);
    Stack.push(root.right);
  }

  while (!Stack.empty()) {
    cluster* c = Stack.top(); Stack.pop();

    if (c != 0) {
      if (!c->is_terminal()) {
        Stack.push(c->left);
        Stack.push(c->right);
      }
      delete c;
    }
  }
}

template<class POINT_SET>
template<class ForwardIterator>
long ATRIA<POINT_SET>::search(vector<neighbor>& v, const long k, ForwardIterator query_point,
```

A. Anhang

```
            const long first, const long last, const double epsilon)
{
  const double root_dist = points.distance(root.charac_point, query_point);

  table.init_search(k);

  // clear search queue
  while(!search_queue.empty()) search_queue.pop();

  // push root cluster as search item into the queue
  search_queue.push(SearchItem(&root, root_dist));

  while(!search_queue.empty())
  {
    const SearchItem si = search_queue.top(); search_queue.pop();
    const cluster* const c = si.clusterp();

    if ((table.highdist() > si.dist()) &&
      ((c->charac_point < first) || (c->charac_point > last)))
      table.insert(neighbor(c->charac_point, si.dist()));

    if (table.highdist() >= si.d_min() * (1.0 + epsilon)) {
      if (c->is_terminal())  {
        const neighbor* const Section = permutation_table + c->start;

        if (c->Rmax == 0.0) {
          for (long i=0; i < c->length; i++) {
            const long j = Section[i].index();

            if (table.highdist() <= si.dist())
              break;

            if ((j < first) || (j > last))
              table.insert(neighbor(j,si.dist()));
          }
        } else {
          for (long i=0; i < c->length; i++) {
            const long j = Section[i].index();

            if ((j < first) || (j > last)) {
              if (table.highdist() > fabs(si.dist() - Section[i].dist()))
                test(j, query_point, table.highdist());
            }
          }
        }
      }
      else {        // this is an internal node
        const double dl = points.distance(c->left->charac_point, query_point);
        const double dr = points.distance(c->right->charac_point, query_point);

        // create child cluster search items
        SearchItem si_left = SearchItem(c->left, dl, dr, si);
        SearchItem si_right = SearchItem(c->right, dr, dl, si);

        // priority queue based search
        search_queue.push(si_right);
        search_queue.push(si_left);
      }
    }
  }

  // append table items to v, v should be empty, afterwards table is empty
  return table.finish_search(v);
}

template<class POINT_SET> template<class ForwardIterator>
long ATRIA<POINT_SET>::search_range(vector<neighbor>& v, const double radius,
        ForwardIterator query_point, const long first, const long last)
{
  long count = 0;   // number of points found

  while (!SearchStack.empty()) SearchStack.pop();   // make shure stack is empty

  SearchStack.push(SearchItem(&root, points.distance(root.charac_point, query_point)));

  while (!SearchStack.empty())
  {
    const SearchItem si = SearchStack.top();
    SearchStack.pop();

    if (radius >= si.d_min()) {
      const cluster* const c = si.clusterp();

      if (((c->charac_point < first) || (c->charac_point > last))
        && (si.dist() <= radius)) {
```

```
        v.push_back(neighbor(c->charac_point, si.dist()));
        count++;
      }

      if (c->is_terminal()) {  // this is a terminal node
        const neighbor* const Section = permutation_table + c->start;

        if (c->Rmax == 0.0) {
          // cluster has zero radius, so all points inside will have the
          // same distance to q
          if (radius >= si.dist()) {
            for (long i=0; i < c->length; i++) {
              const long j = Section[i].index();

              if ((j < first) || (j > last)) {
                v.push_back(neighbor(j,si.dist()));
                count++;
              }
            }
          }
        } else {
          for (long i=0; i < c->length; i++) {
            const long j = Section[i].index();

            if (((j < first) || (j > last)) &&
              (radius >= fabs(si.dist() - Section[i].dist())) )
            {
#ifdef PARTIAL_SEARCH
              const double d = points.distance(j, query_point, radius);
#else
              const double d = points.distance(j, query_point);
#endif
              if (d <= radius) {
                v.push_back(neighbor(j,d));
                count++;
              }
            }
          }
        }
      }
      else {        // this is an internal node
        const double dl = points.distance(c->left->charac_point, query_point);
        const double dr = points.distance(c->right->charac_point, query_point);

        const SearchItem x = SearchItem(c->left, dl, dr, si);
        const SearchItem y = SearchItem(c->right, dr, dl, si);

        SearchStack.push(x);
        SearchStack.push(y);
      }
    }
  }

  return count;
}
```

A.4.2. Implementation einer k–Nächsten–Nachbar–Suche in L_2–Metrik

```
// Sample driver routine to demonstrate
// usage of the searcher class. Reads
// data set points and query points from
// stdin, then searches for k nearest neighbors
// using Euclidian metric.
//
// Compile with: g++ nnsearch.cpp -o nnsearch -O2 -lm

#define PARTIAL_SEARCH

#include "atria.h"

int main(int argc, char** argv)
{
  if (argc < 5) {
    cerr << "Arguments: N D R k"  << endl;
    return -1;
  }

  const long N = atol(argv[1]);
  const long D = atol(argv[2]);
  const long R = atol(argv[3]);
```

```
  const long k = atol(argv[4]);

  double* const p = new double[N*D];
  double* const query_p = new double[R*D];

  for (long n=0; n < N; n++) {
    for (long d=0; d < D; d++) {
      cin >> p[n + d*N];  // points are read from stdin
    }
  }
  for (long r=0; r < R; r++) {
    for (long d=0; d < D; d++) {
      cin >> query_p[r + d*R];  // query points are also read from stdin
    }
  }

  point_set<euclidian_distance> points(N, D, p);
  ATRIA<point_set <euclidian_distance> > searcher(points);
  point_set<euclidian_distance> query_points(R, D, query_p);

  for (long r=0; r < R; r++) {
    vector<neighbor> v;

    searcher.search(v, k, query_points.point_begin(r), -1, -1, 0);
    for (vector<neighbor>::iterator i=v.begin(); i < v.end(); ++i) {
      cout << (*i).index()+1 << " " << (*i).dist() << "  ";
    }
    cout << endl;
  }

  delete[] query_p;
  delete[] p;
  return 0;
}
```

A.4.3. Implementation einer k–Nächsten–Nachbar Suche in L_∞–Metrik, wobei die Anfragepunkte aus dem Datensatz selbst stammen

```
// Sample driver routine to demonstrate
// usage of the searcher class. Reads
// data set points from stdin and searches for
// the k nearest neighbors to each data point
// using maximum distance. Selft matches are
// excluded.
//
// Compile with: g++ nnsearch2.cpp -o nnsearch2 -O2 -lm

#define PARTIAL_SEARCH

#include "atria.h"

int main(int argc, char** argv)
{
  if (argc < 4) {
    cerr << "Arguments: N D k"  << endl;
    return -1;
  }

  const long N = atol(argv[1]);
  const long D = atol(argv[2]);
  const long k = atol(argv[3]);

  double* const p = new double[N*D];

  for (long n=0; n < N; n++) {
    for (long d=0; d < D; d++) {
      cin >> p[n + d*N];  // points are read from stdin
    }
  }

  point_set<maximum_distance> points(N, D, p);
  ATRIA<point_set <maximum_distance> > searcher(points);

  for (long r=0; r < N; r++) {
    vector<neighbor> v;

    searcher.search(v, k, points.point_begin(r), r, r, 0);
    for (vector<neighbor>::iterator i=v.begin(); i < v.end(); ++i) {
      cout << (*i).index()+1 << " " << (*i).dist() << "  ";
```

```
    }
    cout << endl;
  }

  delete[] p;
  return 0;
}
```

A.4.4. Beispielimplementation der schnellen Korrelationssummenberechnung in L_∞–Metrik

```
// Data set points are read from stdin
//
// Compile with: g++ -I. corrsum.cpp -o corrsum -O2 -lm

#define PARTIAL_SEARCH

#include <stdio.h>
#include <math.h>
#include "atria.h"

inline long lmax(const long a, const long b) { return a >= b ? a : b; }
inline long lmin(const long a, const long b) { return a <= b ? a : b; }

long random_permutation(long* ref, const long length, const long pos) {
  const long index = (long) floor((((double)rand())/(RAND_MAX+1))*(length-pos));
  const long r = ref[pos+index];
  ref[pos+index] = ref[pos];  // swap elements
  ref[pos] = r;
  return r;
}

int main(int argc, char** argv)
{
  long bins = 32;   // number of bins

  if (argc < 9) {
    cerr << "Arguments: N D Npairs Nref_min Nref_max eps_min eps_max past bins opt_flag"  << endl;
    return -1;
  }

  const long N = atol(argv[1]);
  const long D = atol(argv[2]);
  const long Npairs = atol(argv[3]); // number of pairs
  long Nref_min = atol(argv[4]);
  long Nref_max = atol(argv[5]);
  const double eps_min = atof(argv[6]);
  const double eps_max = atof(argv[7]);
  const long past = atol(argv[8]);

  if (argc > 9)
    bins = atol(argv[9]);

  if (Nref_min < 1) Nref_min = 1;
  if (Nref_max > N) Nref_max = N;

  double* const p =  new double[N*D];

  for (long n=0; n < N; n++) {
    for (long d=0; d < D; d++) {
      cin >> p[n + d*N];  // points are read from stdin
    }
  }

  point_set<maximum_distance> points(N,D, p);
  ATRIA< point_set<maximum_distance> > searcher(points);

  double scale_factor = pow(eps_max/eps_min, 1.0/(bins-1));

  printf("Number of data set points              : %d\n", N);
  printf("Number of pairs to find                : %d\n", Npairs);
  printf("Miniumum number of reference points    : %d\n", Nref_min);
  printf("Maximum number of reference points     : %d\n", Nref_max);
  printf("Number of partitions used              : %d\n", bins);
  printf("Time window to exclude from search     : %d\n", past);
  printf("Minimal length scale                   : %f\n", eps_min);
  printf("Starting at maximal length scale       : %f\n", eps_max);

  long* const ref = new long[N];         // needed to create random reference indices
  long* const total_pairs = new long[bins]; // number of total pairs is not equal for all bins
  long* const pairs_found = new long[bins]; // number of pairs found within distance dist
```

```
  double* const dists = new double[bins];

  double x = eps_min;

  // initialize vectors
  // pairs_found[i] counts the number of points/distances (real) smaller than dists[i]
  for  (long bin=0; bin < bins; bin++) {
    pairs_found[bin] = 0;
    total_pairs[bin] = 0;
    dists[bin] = x;
    x *= scale_factor;
  }

  for (long r=0; r < N; r++)
    ref[r] = r;

  long R = 0;      // number of reference points actually used
  long bin = bins-1;  // the current highest bin (and length scale) that needs to be filled over Npairs

  while(((R < Nref_min) || (pairs_found[bin] < Npairs)) && (R < Nref_max)) {
    vector<neighbor> v;

    // all points with indices i so that first <= i <= last are excluded(!) from search
    long first, last;
    long pairs = 0;

    // choose random index from 0...N-1, without reoccurences
    const long actual = random_permutation(ref, N, R++);

    if (past >= 0)
      first = actual-past;
    else
      first = actual;

    last = N;
    pairs = lmin(first,last); // don't search points from [actual-past .. N-1]

    if (pairs <= 0) {
      continue;
    }

    searcher.search_range(v, dists[bin], points.point_begin(actual), first, last);

    if (v.size() > 0) {
      for (vector<neighbor>::iterator i = v.begin(); i < v.end(); i++) { // v is unsorted (!!!)
        const double d = (*i).dist();

        for (long n = bin; n >= 0; n--){
          if (d > dists[n])
            break;
        }
      }
      for (long n = 0; n <= bin; n++) {
        total_pairs[n] += pairs;
      }

      // see if we can reduce length scale
      int bins_changed = 0;
      while((pairs_found[bin] >= Npairs) && (bin > 0) && (R >= Nref_min)) {
        bin--;
        bins_changed = 1;
      }
      if (bins_changed) {
        printf("Reference points used so far          : %d\n", R);
        printf("Switching to length scale             : %f\n", dists[bin]);
      }
    }

    printf("Number of reference points used       : %d\n", R);

    for  (long bin=0; bin < bins; bin++) {
      cout << dists[bin] << "  "
      << ((double) pairs_found[bin])/((double) total_pairs[bin]) << endl;
    }

    delete[] p;
    delete[] total_pairs;
    delete[] pairs_found;
    delete[] ref;
    delete[] dists;
}
```

A.4.5. Beispielimplementation eines ternären Suchbaumes für Box–Counting Verfahren

```
// Box counting for a data set of points (row vectors) for
// different partition sizes (2, 3, 4, ...), (equidistant partitioning)
//
// Every value of input point set must be scaled to be within [0,1]
// Returns boxcounting, information and correlation dimension (I_0, I_1 and I_2) (using log2)
//
// A fast algorithm based on ternary search trees to store nonempty boxes is used

// g++ boxcount.cpp -o boxcount -O2 -lm

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#define TST_BUFFERS 128 // number of buffers
#define TST_BUFSIZE 512 // inital buffer size

template<class KEY>
struct Tnode {
    KEY splitkey;
  int level;        /* level of tree */
  unsigned long long count; /* count how often the key leading to this node exists */

  Tnode* lokid;
  Tnode* eqkid;
  Tnode* hikid;
};

// class ternary_search_tree stores multikey-data with fixed key length

template<class KEY, class Evaluater>
class ternary_search_tree
{
  protected:
    typedef Tnode<KEY>* Tptr;

    // these four variables are used to accelerate allocation of Tnode objects

    Tptr buf;      // pointer to current buffer
    long next_buf_size; // size of the buffer that will be allocated next
    long bufn;       // number of next buffer
    long freen;          // number of free nodes in current buffer

    Tptr freearr[TST_BUFFERS];

    const long len;    // key length
    Tptr root;       // tree root node

  public:
    ternary_search_tree(const long keylength);
   ~ternary_search_tree();
    int insert(const KEY* const key);    // insert key vector
    long total_nodes() const { return (long) (TST_BUFSIZE * (pow(2.0, (double) bufn) - 1) - freen); }

    void traverse(Evaluater& eval);
};

template<class KEY, class Evaluater>
ternary_search_tree<KEY, Evaluater>::ternary_search_tree(const long keylength) : bufn(0), freen(0),
  root(0), next_buf_size(TST_BUFSIZE), len(keylength) {}


// return 0 on SUCCESS
template<class KEY, class Evaluater>
int ternary_search_tree<KEY, Evaluater>::insert(const KEY* const key)
{
  KEY d;
  Tptr pp;

    Tptr* p = &root;
  long level = 0;      // level goes up to len-1

    while(pp = *p) {     // as long as we encounter already exisiting nodes, we stay inside this while loop
    if ((d = key[level] - pp->splitkey) == 0) {    // go to next tree level
      pp->count++;
            p = &(pp->eqkid);
      if ((++level) == len) return 0;    // SUCCESS
        } else if (d < 0) {
            p = &(pp->lokid); /* move left in the current level */
```

```
    }
        else {
            p = &(pp->hikid); /* move right in the current level */
    }
    }
    for (;;) {  /* once we find a node that is not allocated (==0), we must create every next node */
    if (freen-- == 0){
      if (bufn == TST_BUFFERS) {
        //mexErrMsgTxt("Ran out of available buffers for tree nodes");
        return -1;     // FAILURE
      }
      buf = new Tnode<KEY>[next_buf_size];
      freearr[bufn++] = buf;
      freen = next_buf_size-1;
      next_buf_size *= 2;   // double size of the next buffer (this keeps overall number of allocations small)
    }
    *p = buf++;
        pp = *p;
        pp->splitkey = key[level];
    pp->count = 1;          // this node is newly created, so count is set to one
    pp->level = level;
        pp->lokid = pp->eqkid = pp->hikid = 0;
        if ((++level) == len) return 0;
        p = &(pp->eqkid);
    }
}

// traverse tree in an arbitrary order, execute the given function object on every node that is not empty
template<class KEY, class Evaluater>
void ternary_search_tree<KEY, Evaluater>::traverse(Evaluater& eval)
{
  long buf_size = TST_BUFSIZE;     // the actual size of the buffer is doubling each iteration

  // traverse through all buffers that are completely filled
  for (long i = 0; i < bufn; i++){
    long number_nodes = buf_size;
    const Tptr b = (Tptr) freearr[i];

    if (i == bufn - 1)     // last buffer may not be completely filled
      number_nodes = buf_size-freen;

    buf_size *= 2;

    for (long j = 0; j < number_nodes; j++){
      const Tptr p = b + j;
      eval(p->count, p->level);
    }
  }
}

template<class KEY, class Evaluater>
ternary_search_tree<KEY, Evaluater>::~ternary_search_tree()
{
    for (long i = 0; i < bufn; i++)
        delete[] freearr[i];
}

#define PARTITIONMAX 16384
#define MY_EPS 1e-6
#define LOGARITHM_OF_2 0.69314718055995
#define MAXDIM 512

inline double log_2(const double x) { return (log(x)/LOGARITHM_OF_2); }
inline double max(double a,double b) {return ((a >= b) ? a : b);}
inline double min(double a,double b) {return ((a <= b) ? a : b);}

class dim_estimator {
  protected:
    long* boxes;  /* scaling of number of boxes for dimensions from 1 to dim */
    double* in;   /* scaling of information for dimensions from 1 to dim */
    double* co;   /* scaling of correlation for dimensions from 1 to dim */

    const long dim;
    const long N;
    long total_points;

  public:
    dim_estimator(const long n, const long Dim) : N(n), dim(Dim), total_points(0),
      boxes(0), in(0), co(0) {
      boxes = new long[dim];
      in = new double[dim];
      co = new double[dim];

      for (long d=0; d < dim; d++){  // initialize arrays to zero
        boxes[d] = 0;
```

```
        in[d] = co[d] = 0;
      }
    }
    ~dim_estimator() {
      delete[] co;
      delete[] in;
      delete[] boxes;
    };

    void operator()(const long mass, const int level) {
      // mass is the absolute frequency of points falling into that box at level level of the tree

      const double p_i = (((double)mass) / (double)N);     // relative frequency

        total_points += mass;                // Check we got all points
        boxes[level]++;                   // D0 = Boxen zaehlen (capacity dimension)
        in[level] += (p_i * log_2(p_i));       // D1 information dimension
        co[level] += p_i * p_i;              // D2 correlation dimension
    }

    long get_total_points() const { return total_points; }
    double boxd(const long d) const { return  -log_2((double) boxes[d]); }
    double infod(const long d) const { return  in[d]; }
    double corrd(const long d) const { return  log_2(co[d]); }
};


int main(int argc, char** argv)
{
  if (argc < 3) {
    cerr << "Arguments: N D Npartitions"  << endl;
    return -1;
  }

  const long N = atol(argv[1]);
  const long dim = atol(argv[2]);
  const long Npartitions = atol(argv[3]); // number of partitions

  if (N < 1) {
    cerr << "Data set must consist of at least two points (row vectors)" << endl;
    return -1;
  }
  if (dim < 1) {
    cerr << "Data points must be at least of dimension one" << endl;
    return -1;
  }
  if (dim > MAXDIM) {
    cerr << "Maximal dimension exceeded"  << endl;
    return -1;
  }


  double* const p =  new double[N*dim];

  for (long n=0; n < N; n++) {
    for (long d=0; d < dim; d++) {
      cin >> p[n + d*N];  // points are read from stdin
    }
  }

  double minimum = p[0];
  double maximum = p[0];

  for (long i=1; i < N * dim; i++) {
    if (minimum > p[i])
      minimum = p[i];
    if (maximum < p[i])
      maximum = p[i];
  }

  printf("Minimum and maximum of input data :: %lf and %lf \n", minimum, maximum);

  if (minimum == maximum) {
    printf("Data values are indentical, nothing to compute\n");
    return -1;
  } else {
    int key[MAXDIM];

    dim_estimator estimator(N, dim);
    ternary_search_tree<int, dim_estimator> tree(dim);

    const double a = (((double) Npartitions)-MY_EPS) / (maximum - minimum);
    const double b = a * minimum;

    printf("Number of partitions per axis   : %d\n", Npartitions);
```

A. Anhang

```
    // Tree fuellen
    for (long index=0; index < N; index++) {
      for (long d=0; d < dim; d++) {
        //const int x = floor((partitions-MY_EPS) * (p[index + d*N] - minimum) / (maximum- minimum));
        const int x = (int) floor(a*p[index + d*N] - b);

        if ((x < 0) || (x >= Npartitions)) {
          cerr << "Data values seem not to be in range [0,1]" << endl;
          return -1;
        }

        key[d] = x;
      }
      if (tree.insert(key)) {
        cerr << "Ran out of memory" << endl;
        return -1;
      }
    }

    // Now that the tree is constructed, compute the output
    tree.traverse(estimator);

    printf("Total nodes allocated   : %d\n", tree.total_nodes());
    printf("Total memory allocated  : %d bytes\n", tree.total_nodes() * sizeof(Tnode<int>));

    if (estimator.get_total_points() != dim * N) {
      cerr << "Internal error, tree did not contain all points" << endl;
      return -1;
    }

    for (long d=0; d < dim; d++) {
      printf("%ld : %lf %lf %lf\n", (d+1), estimator.boxd(d), estimator.infod(d), estimator.corrd(d));
    }
  }

  return 0;
}
```

Literaturverzeichnis

[1] J. Argyris, G. Faust, und M. Haase. *Die Erforschung des Chaos*. Vieweg, 1994.

[2] V.I. Arnold. *Mathematical Methods of classical Mechanics*. Springer Verlag, 1978.

[3] V.I. Arnold und A. Avez. *Problémes Ergodiques de la Mècanique Classique*. Gauthier-Villars, 1967.

[4] D.K. Arrowsmith und C.M. Place. *Dynamische Systeme*. Spektrum Lehrbuch. Spektrum Akademischer Verlag, Heidelberg, 1990.

[5] S. Arya und D. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, Seiten 271–280, 1993.

[6] G. Baier und M. Klein. Chaotic attractors derived from spheroidal dynamics. *Phys. Lett. A*, 151:281 ff., 1990.

[7] G. Baier und S. Sahle. Design of hyperchaotic flows. *Phys. Rev. E*, 51(4):R2712–R2714, 1995.

[8] A. Belussi und C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In Umeshwar Dayal, Peter M. D. Gray, und Shojiro Nishio, Herausgeber, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zürich, Switzerland*, Seiten 299–310. Morgan Kaufmann, 1995.

[9] S. Berchtold, D.A. Böhm, C.and Keim, und H.P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, Seiten 78–86. ACM Press, 1997. ISBN 0-89791-910-6.

[10] T. Bröcker und K. Jänich. *Einführung in die Differentialtopologie*. Springer Verlag, 1973.

[11] P.J. Brockwell und R.A. Davis. *Time Series : Theory and Methods*. Springer Series in Statistics. Springer Verlag, New York, Inc., zweite Auflage, 1991.

[12] L Cao. Practial method for determining the mininum embedding dimension of a scalar time series. *Physica D*, 110:43–50, 1997.

[13] M. Casdagli. Nonlinear prediction of chaotic time series. *Physica D*, 35:335 – 356, 1989.

[14] D.Y. Cheng, A. Gersho, B. Ramamurthi, und Y. Shoham. Fast search algorithms for vector quantization and pattern matching. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, Seiten 9.11.1–9.11.4. IEEE Press, Piscataway NJ, March 1984.

[15] J.P. Crutchfield und B.S. McNamara. Equations of motion from a data series. *Complex Systems*, 1:417 – 452, 1987.

[16] J.-P. Eckmann und D. Ruelle. Ergodic theory of chaos and strange attractors. *Rev. Mod. Phys.*, 57(3):617 – 656, 1985.

[17] K.J. Falconer. *The geometry of fractal sets*. Cambridge University Press, 1985.

[18] J.D. Farmer und J.J. Sidorowich. Predicting chaotic time series. *Phys. Rev. Lett.*, 59(8):845 – 848, 1987.

[19] G. Froyland, K. Judd, A.I. Mees, D. Watson, und K. Murao. Constructing invariant measures from data. *Int. J. Bif. Chaos*, 5(4):1181–1192, 1995.

[20] P. Grassberger. Generalized dimensions of strange attractors. *Phys. Lett. A*, 1983.

[21] T.C Halsey, M.H. Jensen, L.P. Kadanoff, I. Procaccia, und B.I. Shraiman. Fractal measures and their singularities: The characterization of strange sets. *Phys. Rev. A*, 33(2):1141–1151, 1986.

[22] H. Heuser. *Gewöhnliche Differentialgleichungen*. Teubner, Stuttgart, 1989.

[23] K. Kaneko. Nonstationary chaos revisited from large deviation theory. *Prog. Theor. Phys. Suppl.*, 99:149–164, 1989.

[24] U. Krengel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik*. Vieweg Verlag, 1991.

[25] R.W. Leven, B.P. Koch, und B. Pompe. *Chaos in dissipativen Systemen*. Akademie Verlag, Berlin, 1994.

[26] B. Mandelbrot. *Die fraktale Geometrie der Natur*. Birkhäuser Verlag, Basel, 1987.

[27] P. Manneville. *Lecture Notes in Physics*, Band 230. Springer, New York, 1985.

[28] J. McNames. A nearest trajectory strategy for time series prediction. In J.A.K. Suykens und J. Vandewalle, Herausgeber, *Proceedings of the International Workshop on Advanced Black-Box Techniques for Nonlinear Modeling*, Seiten 112–128. Katholieke Universiteit Leuven Belgium, July 8-10 1998.

[29] C. Merkwirth, U. Parlitz, und W. Lauterborn. Fast exact and approximate nearest neighbor searching for nonlinear signal processing. *Phys. Rev. E*, 62(2):2089–2097, 2000.

[30] S. Ørstavik und J. Stark. Reconstruction and cross-prediction in coupled map lattices using spatio-temporal embedding techniques. *Phys. Lett. A*, 247:145–160, 1998.

[31] G.J. Ortega. Invariant measures as lagrangian varaibles: Their application to time series analysis. *Phys. Rev. Lett.*, 77(2):259–261, 1996.

[32] U. Parlitz. *Nonlinear Modeling - Advanced Black–Box Techniques*, Kapitel Nonlinear Time–Series Analysis, Seiten 209–239. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[33] U. Parlitz und C. Merkwirth. Prediction of spatiotemporal time series based on reconstructed local states. *Phys. Rev. Lett.*, 84(9):1890–1893, 2000.

[34] V Pestov. On the geometry of similarity search: Dimensionality curse and concentration of measure. *Information Processing Letters*, 73 (1-2):47–51, 2000.

[35] A. Rényi. *Wahrscheinlichkeitsrechnung mit einem Anhang über Informationstheorie*. VEB Deutscher Verlag der Wissenschaften, 1962.

[36] D.M. Rubin. Use of forecasting signatures to help distinguish periodicity, randomness, and chaos in ripples and other spatial patterns. *Chaos*, 2:525–535, 1992.

[37] D. Ruelle. *Chaotic evolution and strange attractors*. Cambridge University Press, 1989.

[38] T. Sauer, J.A. Yorke, und M. Casdagli. Embedology. *J.Stat.Phys.*, 65, 1991.

[39] T. Schreiber. Efficient neighbor searching in nonlinear time–series analysis. *Int. J. Bifurcation and Chaos*, 5:349–358, 1996.

[40] H.G. Schuster. *Deterministic Chaos*. VCH Verlagsgesellschaft mbH, Weinheim, 1988.

[41] R. Sedgewick. *Algorithms in C++, Parts 1-4*. Addison–Wesley, Reading,Massachusetts, dritte Auflage, 1998.

[42] C.E. Shannon und W. Weaver. *The mathematical theory of communication*. University of Ill. Press, Urbana, 1949.

[43] J.A.K Suykens und J. Vandewalle. *Nonlinear Modeling - Advanced Black–Box Techniques*, Kapitel The K.U. Leuven Time Series Prediction Competition, Seiten 241–253. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[44] F. Takens. Detecting strange attractors in turbulence. In *Dynamical Systems and Turbulence*, Lect. Notes Math. Sringer-Verlag, Berlin, 1981.

[45] J Theiler. Spurious dimensions from correlation algorithms applied to limited time–series data. *Phys. Rev. A*, 34:2427–2431, 1986.

[46] J. Theiler, S. Eubank, A. Longtin, B. Galdrikian, und J.D. Farmer. Testing for nonlinearity in time series: the method of surrogate data. *Physica D*, 58:77 – 94, 1992.

[47] W. van de Water und P. Schram. Generalized dimensions from near–neighbor information. *Phys. Rev. A*, 37(8):3118–3125, 1988.

[48] V.N. Vapnik. *Statistical Learning Theory*. John Wiley & Son, New York, 1998.

[49] P. Walters. *An introduction to ergodic theory*. Springer-Verlag, 1982.

Danksagung

Herzlich bedanken möchte ich mich bei PD Dr. Ulrich Parlitz und Prof. Dr. Werner Lauterborn, die die vorgelegte Arbeit ermöglicht, gefördert und betreut haben.

Neben diesen haben aber auch viele andere meinen wissenschaftlichen Werdegang beeinflußt. Eine Person, die ich leider nur flüchtig persönlich kennengelernt habe, aber deren systematische Arbeit mir oft als Leitfaden gedient hat, ist sicherlich James McNames, der jetzt an der Portland State University beschäftigt ist. Johann Suykens von der Katholieke Universiteit Leuven, der mich mit seinen Ideen zur Verbindung von lokaler Modellierung und Support Vector Verfahren inspiriert hat sowie Dr. Mark Keßeböhmer, mit dem ich interessante Diskussionen zur Berechnung der Rényi–Dimensionen führen konnte, und nicht zuletzt die Mitglieder unserer Arbeitsgruppe, unter diesen speziell Jörg Daniel Wichard, Jochen Bröcker, Martin Wiesenfeldt, Claus-Dieter Ohl, Stefan Luther, Lutz Junge und Alexander Freiherr von Schenck zu Schweinsberg, müssen hier ausdrücklich erwähnt werden.

Auch den vielen unterstützenden Händen, ohne die das wissenschaftliche Arbeiten im Institut kaum möglich, geschweige denn angenehm gewesen wäre, bin ich sehr zu Dank verpflichtet. Außerdem möchte ich mich bei all den hier nicht Genannten bedanken, die mir mit fachmännischem Rat und tatkräftiger Unterstützung beim Erstellen dieser Arbeit geholfen haben. Allen Mitgliedern des Institutes danke ich für die freundliche, offene und kooperative Atmosphäre im Haus.

Teile der vorgelegten Arbeit beruhen auf Ergebnissen, die im Rahmen des vom Bundesministeriums für Bildung und Forschung (BMBF) geförderten Projektes „Analyse von Maschinen– und Prozeßzuständen mit Methoden der Nichtlinearen Dynamik“ (Fördernummer 13N7038/9) entstanden.

Meinen Eltern möchte ich für ihre ausdauernde Unterstützung während meines gesamten Stu-

diums danken.

Lebenslauf

Persönliche Daten

Name	Christian Merkwirth
Geburtsdatum	19.06.1972
Geburtsort	Kassel
Staatsangehörigkeit	deutsch
Familienstand	ledig

Schulausbildung

1978 – 1982	Grundschule Balhorn
1982 – 1984	Förderstufe Naumburg
1984 – 1988	Gesamtschule Bad Emstal
1988 – 1989	Gymnas. Oberstufe Oberzwehren
1989 – 1991	Gymnas. Oberstufe der Wilhelm-Filchner Gesamtschule Wolfhagen
06/1991	Allgemeine Hochschulreife an der Gymnas. Oberstufe der Wilhelm-Filchner Gesamtschule Wolfhagen

Wissenschaftliche Ausbildung

WS 1991/92 – SS 1993	Grundstudium Physik an der Georg-August-Universität zu Göttingen
10/1993	Physikvordiplom, Gesamtnote: Sehr gut
WS 1993/94 – WS 1996/97	Hauptstudium Physik an der Georg-August-Universität
02/1997	Diplomprüfung Physik Thema der Diplomarbeit: Untersuchung parametrisch angeregter Oberflächenwellen
02/1997 – 02/2000	Wissenschaftlicher Mitarbeiter am Dritten Physikalischen Institut der Universität Göttingen
SS 1997 – SS 2000	Aufbaustudiengang Physik an der Georg-August-Universität zu Göttingen