Optimale Approximation uni- oder multivariater Funktionen in mehreren Auflösungen

Oliver Kreylos

30. November 1998

Erklärung

Ich versichere hiermit, daß ich zur Anfertigung dieser Diplomarbeit keine Quellen außer den in der Literaturliste angegebenen verwendet habe; weiterhin habe ich diese Arbeit alleine und ohne fremde Hilfe angefertigt.

Oliver Kreylos

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

In vielen Anwendungen beschäftigt man sich mit der Repräsentation komplexer Geometrien oder komplexer physikalischer Phänomene in mehreren Auflösungsstufen. Im Zusammenhang mit Computergrafik und wissenschaftlicher Visualisierung haben sogenannte *multiresolution Methoden* entscheidende Bedeutung für die Analyse sehr großer numerischer Datensätze (siehe [1], [2], [3], [4] und [5]). Beispiele sind unter anderem hochauflösende Geländedaten (digitale Höhenkarten) und hochauflösende, dreidimensionale bildgebende Verfahren wie Magnetic Resonance Imaging oder Computer Aided Tomography.

In dieser Arbeit präsentieren wir eine Methode zur Konstruktion von Darstellungen sehr großer Streudatensätze in mehreren Auflösungen unter Verwendung des Prinzips des *Simulated Annealing* (siehe [12], [15] und [16]). Unser Ziel ist die Bestimmung mehrerer linearer Splines, welche die gegebenen Streudaten optimal approximieren.

Hierbei nehmen wir an, daß die gegebenen Datensätze Abtastmengen einer reell- oder vektorwertigen Funktion einer oder zweier Veränderlicher[1] sind, wobei die Abtaststellen zufällig über den Definitionsbereich der Funktion verteilt sind. Wir treffen keine weiteren Annahmen über das Verhalten der abgetasteten Funktion zwischen den Abtaststellen, insbesondere wird nichts über Stetigkeit oder sogar Differenzierbarkeit der Funktionen vorausgesetzt. Wir benutzen also nur den „kleinsten gemeinsamen Nenner“ aller möglichen Funktionen, um das Verfahren auf möglichst viele Streudatenquellen anset-

[1] Das dargestellte Approximationsverfahren funktioniert für beliebig viele Veränderliche, aber im Rahmen dieser Arbeit wird nur auf die uni- und bivariaten Fälle im Detail eingegangen.

zen zu können.

Jeder einzelne approximierende lineare Spline ist eindeutig definiert durch die Menge seiner Kontrollpunkte und die Art, wie diese Kontrollpunkte zu einem Simplexnetz verbunden sind. Im univariaten Fall formen die Kontrollpunkte einen Streckenzug, im bivariaten Fall eine Triangulierung. Im Falle linearer Splines wird der Funktionswert des Splines in einem Punkt innerhalb eines Simplex durch lineare Interpolation der Funktionswerte in den Eckpunkten des Simplex bestimmt.

Wenn ein hochauflösender Datensatz durch einen linearen Spline niedriger Auflösung, sprich durch einen linearen Spline mit nur wenigen Kontrollpunkten approximiert wird, ist die Plazierung der Kontrollpunkte und ihre Verbindung zu einem Simplexnetz von entscheidender Bedeutung für die Güte der Approximation (siehe Abbildungen 1.1 und 1.2).

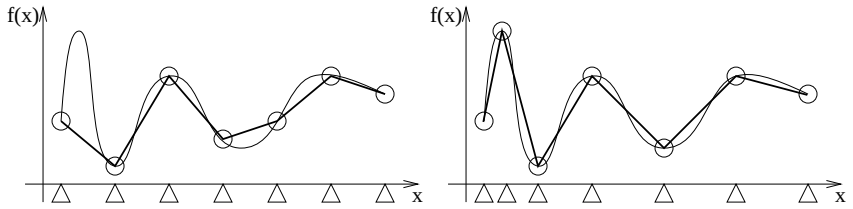

Abbildung 1.1: Uniforme und optimale Kontrollpunktplazierung im univariaten Fall. Links: Uniforme Plazierung, rechts: optimale Plazierung.

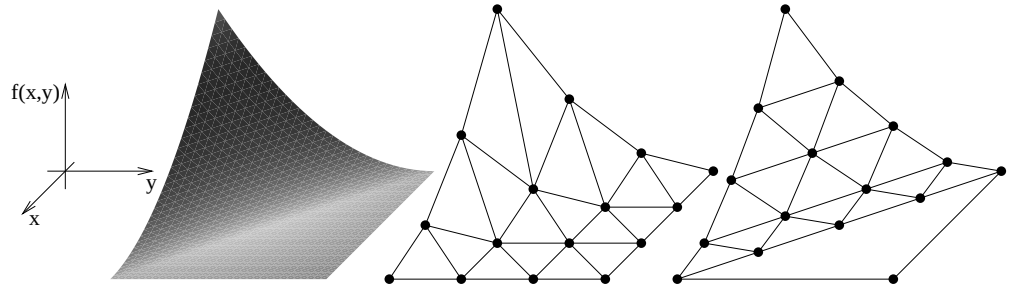

Abbildung 1.2: Uniforme und optimale Kontrollpunktplazierung im bivariaten Fall. Links: zu approximierende Funktion, mitte: uniforme Plazierung, rechts: optimale Plazierung.

Um den optimalen approximierenden linearen Spline für eine gegebene Streudatenmenge und eine festgelegte Zahl von Kontrollpunkten zu finden,

benutzen wir einen iterativen Optimierungsalgorithmus, der die „Qualität“ einer initialen Approximation – gemessen mit einem geeigneten Abstandsmaß (siehe 2.2) – durch Bewegung der Kontrollpunkte und Änderung der Struktur des Simplexnetzes zu verbessern sucht (siehe Abbildung 1.3). Da alle zu approximierenden Funktionen nur durch ihre Werte an zufällig verteilten Stellen ihrer Definitionsbereiche gegeben sind, beschränken wir die Plazierung von Kontrollpunkten auf die folgende Art und Weise: Wir plazieren Kontrollpunkte nur an solchen Stellen, die im Streudatensatz enthalten sind; und wir benutzen als Funktionswerte der Kontrollpunkte nur die gegebenen Werte aus dem Streudatensatz.

Der Hauptvorteil dieser Herangehensweise ist die Tatsache, daß keine zusätzlichen Informationen über die Geometrie des Problems oder über die Funktion bekannt sein müssen. Ließe man die Plazierung von Kontrollpunkten an beliebigen Stellen zu, müsste der Wert der Funktion an diesen Stellen zuerst durch ein geeignetes Interpolationsverfahren bestimmt werden; und zur permanenten Speicherung einer Approximation müssten alle Stellen und Funktionswerte aller Kontrollpunkte gespeichert werden. Durch die Einschränkung auf originale Stellen und Funktionswerte wird das Problem der Kontrollpunktplazierung auf die Auswahl einer Untermenge des Streudatensatzes beschränkt, und zur permanenten Speicherung brauchen nur Indizes in die Streudatenmenge gespeichert zu werden. Weiterhin wird das Problem der „Bewegung eines Kontrollpunktes“ auf die Entfernung eines Elements aus der Untermenge und die Einfügung eines anderen Elements reduziert. Mit anderen Worten, Kontrollpunkte „bewegen“ sich nicht, sie „springen“ von Stelle zu Stelle.

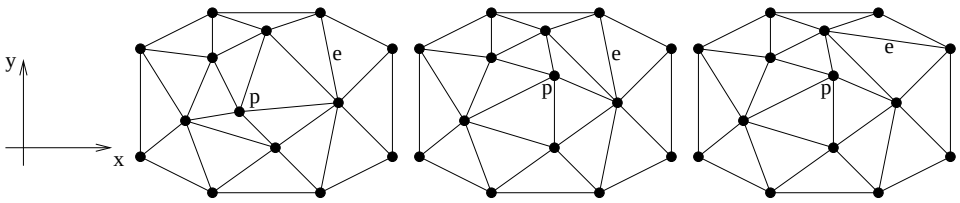

Abbildung 1.3: Änderung eines linearen Splines durch zwei Iterationsschritte. Links: initialer Zustand, mitte: Änderung der Kontrollpunktplazierung durch „Bewegung“ des Kontrollpunktes p, rechts: Änderung des Simplexnetzes durch anschließende „Rotation“ der Kante e.

Der Verlauf dieser Iteration wird durch den Simulated-Annealing-Algo-

rithmus gesteuert, eine Optimierungstechnik, die sehr geeignet ist für hochdimensionale Optimierungsprobleme, in denen das gesuchte globale Minimum unter sehr vielen, schlechteren, lokalen Minima verborgen ist (siehe 2.1). Simulated Annealing ist ein probabilistisches Optimierungsverfahren. Es ist also schwierig bis unmöglich, Zeitschranken für den Optimierungsprozeß bis zur Erreichung eines Minimums anzugeben, und es ist immer möglich, daß der Algorithmus trotz seiner Auslegung nicht das gewünschte globale Minimum findet. Weiterhin verwendet der Algorithmus nur minimales Wissen über das jeweilige Optimierungsproblem, und ist daher in „klassischen" Fällen den „klassischen" Verfahren wie Gradientenabstieg oder Simplexverfahren (siehe [15]) stets unterlegen. Das wir zur Lösung unseres Optimierungsproblems trotzdem den Simulated-Annealing-Algorithmus verwenden, ist durch zwei Besonderheiten unserer Klasse von Problemen begründet, die den Einsatz klassischer Verfahren vereiteln:

1. Unsere Approximierungsprobleme sind extrem hochdimensional. Nehmen wir an, unser Ziel sei die Approximation einer bivariaten, reellwertigen Funktion (gegeben als Streudatensatz) durch einen linearen Spline mit $n = 1.000$ Kontrollpunkten (eine durchaus üblicher Fall, wie Kapitel 4 zeigen wird). Da wir als Funktionswerte der Kontrollpunkte des Splines nur Funktionswerte des Streudatensatzes verwenden, ist die Position jedes Kontrollpunktes im Definitionsbereich der Funktion durch zwei Variablen gegeben. Weiterhin enthält das Simplexnetz des Splines bei n Kontrollpunkten ungefähr $3n$ Kanten, jede Kante ist durch zwei Indizes in die Kontrollpunktmenge definiert. Das heißt, ein linearer Spline mit $n = 1.000$ Kontrollpunkten ist durch $n \cdot 2 + 3n \cdot 2 = 6n = 6.000$ Variablen definiert. Da der Definitionsbereich des zu optimierenden Qualitätsmaßes in unserem Fall genau der Raum aller linearen Splines mit n Kontrollpunkten ist, ist die Dimension des Optimierungsproblems $d = 6.000$. Klassische Optimierungsverfahren wie Gradientenabstieg müssen in jedem Schritt den Gradienten der Zielfunktion bestimmen oder schätzen, das bedeutete in diesem Fall die Berechnung der Qualitätsmaße von 6.000 linearen Splines in jedem Schritt. Auch die Benutzung des Simplexverfahrens scheitert: Obwohl hier in der Regel nur eine Auswertung der Zielfunktion pro Schritt nötig ist, müssen doch Darstellungen von 6.000 linearen Splines zur Iteration vorgehalten werden.

2. Unsere Approximierungsprobleme enthalten extrem viele lokale Mini-

ma. Ein 6.000-dimensionaler Raum ist schwer zu visualisieren, aber eine genauere Untersuchung der Probleme hat ergeben, daß jeder eindimensionale Schnitt durch den Graph der Zielfunktion nicht, wie in klassischen Problemen, „glatt“ ist, sondern „gewellt“. Für dieses Aussehen der Zielfunktionen haben wir den Begriff *Gänsehaut* geprägt (siehe Abbildung 1.4). Ein Raffke-Optimierungsverfahren (alle klassischen Verfahren gehören in diese Klasse) kann von jedem Punkt der Zielfunktion stets nur zu einem anderen Punkt gehen, wenn dieser einen geringeren Funktionswert aufweist. Ein Raffke-Verfahren müsste also langsam, in kleinen Schritten, einen Weg um die „Hügel“ der Zielfunktion herum finden, während Simulated Annealing die Gänsehaut einfach überspringt.

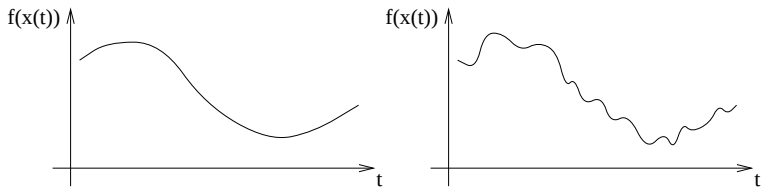

Abbildung 1.4: Eindimensionale Schnitte durch den Graph der Zielfunktion für „klassische“ Probleme (links) und für Lineare-Spline-Approximationen (rechts). Das Aussehen der Funktion im rechten Bild bezeichnen wir als Gänsehaut. In beiden Fällen ist $x(t) := x_0 + t \cdot \Delta x$ eine Gerade im d-dimensionalen Raum.

1.1 Grundlegende Definitionen

Um die Diskussion der verwendeten Algorithmen und Datenstrukturen zu vereinfachen, und um die einheitliche Beschreibung der uni- und bivariaten Fälle zu ermöglichen, definieren wir spezielle Begriffe. Einige entstammen dem Feld der Computational Geometry, andere sind Definitionen wohlbekannter mathematischer Begriffe, die hier zur Vermeidung von Mißverständnissen noch einmal präzisiert werden.

- Eine *Stelle* ist ein Punkt innerhalb des Definitionsbereichs einer Funktion. Für eine Funktion von n (reellen) Veränderlichen ist eine Stelle

definiert als ein Element $s := (x_1, \ldots, x_n) \in \mathbf{R}^n$ des n-dimensionalen Standardvektorraums.

- Die *konvexe Hülle* einer Menge $S \subset \mathbf{R}^n$ von Stellen ist die kleinste konvexe Menge $CH(S) \subset \mathbf{R}^n$, die alle Stellen in S enthält, für die also $S \subset CH(S)$ gilt. Mit anderen Worten, $CH(S)$ ist die Schnittmenge aller konvexen Obermengen von S,

$$CH(S) := \bigcap_{\substack{C \text{ ist konvex} \\ S \subset C}} C \quad . \tag{1.1}$$

  Für jede Dimension n ist eine konvexe Hülle ein konvexes n-dimensionales „Hyperpolyeder“, also die Schnittmenge einer endlichen Zahl von Halbräumen des $\mathbf{R}^n$. Im besonderen sind die Eckpunkte dieses Hyperpolyeders stets Elemente von S. Wir bezeichnen eine Stelle s als *innere Stelle*, wenn s kein Eckpunkt des Hyperpolyeders von $CH(S)$ ist, oder mit anderen Worten, wenn das Entfernen von s aus S die konvexe Hülle $CH(S)$ invariant liesse, $CH\big(S \setminus \{s\}\big) = CH(S)$ (siehe Abbildung 1.5).

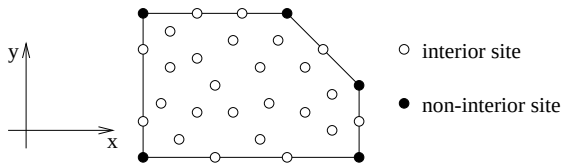

Abbildung 1.5: Innere und äußere Stellen im bivariaten Fall.

- Ein *Vertex* ist ein Paar bestehend aus einer Stelle s und dem Wert $f(s)$ einer Funktion f an der Stelle s. Für eine Funktion $f\colon \mathbf{R}^n \to \mathbf{R}^m$ von n Veränderlichen und m Funktionswerten ist ein Vertex ein $(n+m)$-Tupel $v := \big(s, f(s)\big) = \Big((x_1, \ldots, x_n), \big(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)\big)\Big) \in \mathbf{R}^n \times \mathbf{R}^m$. Wir bezeichnen einen Vertex als *inneren Vertex*, wenn seine Stelle $v.s$ eine innere Stelle ist. Im Zusammenhang mit linearen Splines benutzen wir den Begriff „Vertex“ auch als handliche Bezeichnung für die Kontrollpunkte des Splines.

- Eine *Streudatenmenge* ist die Darstellung einer Funktion $f\colon \mathbf{R}^n \to \mathbf{R}^m$ von n Veränderlichen und m Funktionswerten als eine endliche Menge von Vertizes, wobei die Vertizes paarweise verschiedene Stellen haben.

Wir gehen generell davon aus, daß die Stellen einer Streudatenmenge zufällig über den Definitionsbereich der Funktion f verteilt sind; die im Kapitel 3 erklärten Algorithmen funktionieren aber auch für den Spezialfall gleichmäßig angeordneter Stellen, resultierend zum Beispiel aus der Abtastung von f über einem rectilinearen Gitter.

- Eine *Vertexplazierung* ist ein Tupel von Vertizes, wobei die Vertizes paarweise verschiedene Stellen haben. Für N Vertizes und eine Funktion $f: \mathbf{R}^n \rightarrow \mathbf{R}^m$ ist die zugehörige Vertexplazierung ein N-Tupel $V := \Big(\big(s_1, f(s_1)\big), \ldots, \big(s_N, f(s_N)\big)\Big) \in (\mathbf{R}^n \times \mathbf{R}^m)^N$. In unserer Anwendung fungiert eine Vertexplazierung als die Kontrollpunktmenge eines linearen Spline.

- Eine *Verbindungsmenge* oder ein *Simplexnetz* ist die Menge aller n-dimensionalen Simplizes, welche durch alle $(n+1)$-Tupel benachbarter Stellen einer Vertexplazierung definiert werden. Dies bedeutet, daß jeder Punkt innerhalb der konvexen Hülle der Stellen der Vertexplazierung, der in „allgemeiner Lage“ zu diesen Stellen liegt, von genau einem Simplex der Verbindungsmenge überdeckt wird. Punkte auf der Grenze zwischen zwei benachbarten Simplizes werden von beiden überdeckt, und die Stellen der Vertizes selbst werden von allen Simplizes überdeckt, die durch den jeweiligen Vertex definiert sind. Im univariaten Fall ist die Verbindungsmenge die Menge aller Intervalle, die durch je zwei benachbarte Vertizes begrenzt werden; im bivariaten Fall ist die Verbindungsmenge eine Triangulierung der konvexen Hülle der Stellen; für drei Variablen ist die Verbindungsmenge analog ein Tetraedernetz. Generell bezeichnen wir die Menge aller n-dimensionalen Simplizes als CS_n.

  Die $(n-1)$-dimensionalen Simplizes, die benachbarte Simplizes in der Verbindungsmenge trennen, werden ohne Berücksichtigung der Dimension n stets *Kanten* genannt. Generell bezeichnen wir die Menge aller Kanten aller n-dimensionalen Verbindungsmengen als CE_n. Das *Platelet* eines Vertex ist die Vereinigung aller Simplizes, die die Stelle dieses Vertex überdecken. Im univariaten Fall ist ein Platelet stets ein Intervall, im bivariaten Fall ist ein Platelet das innere (und der Rand) eines Sternpolygons, in dessen Kern die Stelle des Vertex liegt. Der Zusammenhang zwischen Stellen, Vertizes und Platelets wird in Abbildung 1.6 illustriert.

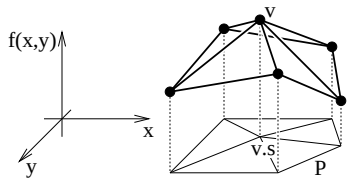

Abbildung 1.6: Ein Vertex v, seine Stelle $v.s$ und die Begrenzung P seines Platelets im bivariaten Fall.

- Eine *Konfiguration* $C = (V, SM)$ ist das Paar aus Vertexplazierung V und dazugehörigem Simplexnetz SM. Eine Konfiguration definiert genau einen linearen Spline, und die Menge aller Konfigurationen mit einer festgelegten Zahl N von Vertizes ist der Definitionsbereich für die Zielfunktion des Optimierungsproblems der Linearen-Spline-Approximation. Generell bezeichnen wir die Menge aller linearen Splines von n Veränderlichen und m Funktionswerten als $LS_{n,m}$, die Untermenge aller linearen Splines mit N Vertizes nennen wir $LS_{n,m}(N)$.

1.2 Visualisierung großer Datensätze

Das Hauptproblem bei der Visualisierung von Funktionen, die durch große Streudatensätze definiert sind, ist die Erzeugung von Bildern dieser Funktionen in Echtzeit. Besonders bei der Visualisierung großer dreidimensionaler Datensätze aus medizinischen bildgebenden Verfahren ist die Möglichkeit, den Blickwinkel der Darstellung in Echtzeit ändern zu können, von großer Bedeutung für die Auswertung der Datensätze durch den Benutzer. Eine Echtzeitdarstellung kann jedoch nur erfolgen, wenn der umfangreiche Datensatz durch eine genügend kleine Zahl von *grafischen Primitiven*, wie z.B. Strecken oder Dreiecken, approximiert werden kann. Weiterhin ist es wünschenswert, die Qualität der Darstellung durch Auswahl einer angemessen feinen Approximation in weiten Grenzen bestimmen zu können. So wäre es möglich, einen interessierenden Ausschnitt eines großen Datensatzes bei geringer Bildqualität schnell aufzufinden, und dann unter größerem Zeitaufwand ein qualitativ hochwertiges Bild des gefundenen Ausschnitts darzustellen. Zu diesem Zweck ist die Verfügbarkeit einer Hierarchie von Approximationen mit steigenden Anzahlen grafischer Primitive wichtig.

Um all diesen Anforderungen gerecht zu werden, erzeugen wir eine n-

stufige Hierarchie von Approximationen; und für jede Hierarchieebene $k \in \{1, \ldots, n\}$ wählen wir N_k Vertizes aus der gegebenen Streudatenmenge, d.h. wir wählen nur Stellen, die in der Streudatenmenge enthalten sind, und wir verwenden nur die an diesen Stellen gegebenen Funktionswerte. Weiterhin stellen wir sicher, daß die in allen Stufen $j < k$ ausgewählten Vertexmengen in der Menge der Stufe k enthalten sind. Für jede Stufe k wird dann aus den N_k ausgewählten Vertizes durch ein geeignet gewähltes Simplexnetz ein linearer Spline definiert. Die so gebildeten n linearen Splines stellen die von uns gewünschte Approximationshierarchie an die gegebene Streudatenmenge dar (siehe Abbildung 1.7).

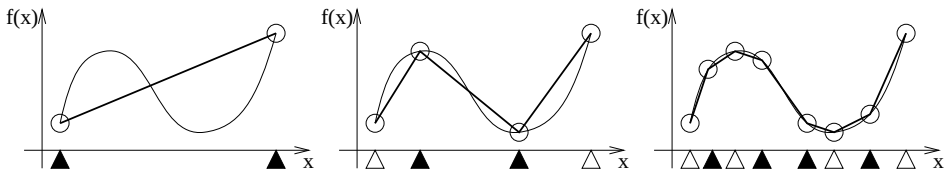

Abbildung 1.7: Eine Hierarchie von approximierenden linearen Splines an eine gegebene Funktion im univariaten Fall. Die Stellen in einer Hierarchieebene neu eingefügter Vertizes sind durch ausgefüllte Dreiecke markiert.

Wenn die Anzahl N_k von Vertizes in einer Hierarchieebene festgelegt ist, müssen noch zwei Probleme gelöst werden:

1. Welche Vertizes sollen für die Approximation ausgewählt werden, d.h. wie soll die Vertexplazierung erzeugt werden?

2. Wie sollen die ausgewählten Vertizes durch Simplizes verbunden werden, d.h. wie soll die Verbindungsmenge erzeugt werden?

Im Spezialfall von Funktionen nur einer Veränderlicher brauchen wir nur das erste Problem anzugreifen, da in diesem Fall das Simplexnetz durch die Vertexplazierung eindeutig festgelegt ist. Der einzige Weg der Verbindung ist die Sortierung der Vertices nach der numerischen Ordnung ihrer Stellen, und die anschließende Verbindung benachbarter Vertices durch Strecken.

1.3 Optimale Approximation

Unser Verfahren zur Bestimmung einer optimalen Linearen-Spline-Approximation mit einer gegebenen, festen Anzahl von Kontrollpunkten N_k basiert

auf einem iterativen Optimierungsverfahren. Zuerst erzeugen wir eine initiale Konfiguration, d.h. eine initiale Vertexplazierung und ein initiales Simplexnetz; dann suchen wir diese initiale Konfiguration durch eine Folge von Schritten zu verbessern, wobei in jedem Schritt Vertexplazierung und Simplexnetz geändert werden. Da dieses Optimierungsproblem hochdimensional ist und lokale Minima im Überfluß aufweist, ist der Simulated-Annealing-Algorithmus das Verfahren der Wahl zur Steuerung der Iteration und zur Erzeugung „guter" Linearer-Spline-Approximationen (siehe Abschnitt 2.1).

Während des Optimierungsprozesses beschränken wir unseren Algorithmus in der Art, daß nur Vertizes zur Definition von linearen Splines herangezogen werden, die in der gegebenen Streudatenmenge vorkommen. Diese Entscheidung bringt zwei Hauptvorteile mit sich:

1. Da die zu approximierende Funktion nur an den abgetasteten Stellen bekannt ist, müssten wir den Funktionswert an nicht in der Streudatenmenge enthaltenen Stellen abschätzen, d.h. wir müssten einen geeigneten Streudateninterpolations- oder -approximationsalgorithmus entwickeln. Die Verwendung eines solchen Algorithmus setzte aber Annahmen über die zu approximierenden Funktionen (wie Stetigkeit oder Differenzierbarkeit) voraus, die die allgemeine Verwendbarkeit des Optimierungsalgorithmus nur künstlich einschränkten.

2. Durch die alleinige Verwendung definierter Stellen und Funktionswerte ist die Vertexmenge jeder Approximation stets eine Untermenge der gegebenen Streudatenmenge. Um also eine Approximation darzustellen, genügt es, eine Menge von Indizes in die Streudatenmenge zu unterhalten, anstelle einer Menge von Stellen und assoziierten Funktionswerten. Zur Repräsentation einer Hierarchie von Approximationen genügt es sogar, für jeden gegebenen Vertex eine (kleine) Ganzzahl zu speichern, die angibt, ab welcher Hierarchieebene der Vertex Teil der Vertexmenge der Approximation wird. Durch unsere Entscheidung, die Vertexmenge jeder Hierarchieebene Teilmenge aller „höheren" Vertexmengen sein zu lassen, ist die Eindeutigkeit dieser Darstellung gewährleistet.

Wie auch immer, im Falle zweier oder mehrerer Veränderlicher hängt die Qualität einer Konfiguration nicht nur von der Vertexplazierung, sondern auch vom Simplexnetz ab (siehe Abschnitt 2.3.2). Auch hier gibt es zwei mögliche Vorgehensweisen:

1. Man kann das Problem des optimalen Simplexnetzes einfach ignorieren und eine Funktion von n Veränderlichen wie eine univariate Funktion behandeln, indem man die Verwendung einer festgelegten Klasse von Verbindungsmengen während des gesamten Optimierungsprozesses vorschreibt; ein offensichtlicher Kandidat im bivariaten Fall ist die Klasse der Delaunay-Triangulierungen (siehe Abschnitt 2.3.1). Unter dieser Einschränkung ist ein Simplexnetz durch eine Vertexplazierung impliziert (zumindest bis auf vernachlässigbare Zweideutigkeiten[2]), und der Iterationsalgorithmus kann genau wie im univariaten Fall fortschreiten.

   Nach Beendigung der Iteration, wenn eine optimale Vertexplazierung für die verwendete Klasse von Verbindungsmengen gefunden ist, kann dieses Ergebnis als Eingabe eines *datenabhängigen Triangulierungsalgorithmus* verwendet werden, der eine optimale Verbindungsmenge für die nun festgelegte Vertexplazierung zu finden sucht (siehe 2.3.2). Der Nachteil dieses zweiteiligen Algorithmus ist, daß die endgültige Vertexplazierung nicht optimal für die endgültige Verbindungsmenge ist, da beide Teile der Konfiguration getrennt optimiert wurden.

2. Man kann versuchen, beide Teile der Konfiguration, also Vertexplazierung und Simplexnetz, parallel zu optimieren. Zum Beispiel könnte der Iterationsalgorithmus vor jedem Schritt zufällig entscheiden, welcher Teil in diesem Schritt geändert werden soll, und dann entweder einen Vertex bewegen oder eine Kante rotieren. Die Wahrscheinlichkeit der Entscheidung, welcher Teil der Konfiguration geändert werden soll, könnte entweder während der gesamten Iteration konstant bleiben, oder sie könnte sich ändern, um Änderungen der Verbindungsmenge in späteren Stadien der Iteration zu favorisieren.

   Wie auch immer, obwohl dieser Algorithmus in dem Sinne „besser“ ist, daß ein besseres Minimum des Abstandsmaßes gefunden werden kann, hat er doch zwei Nachteile: Zum einen muß nun der univariate Fall getrennt von den multivariaten Fällen behandelt werden, und durch die weitere Erhöhung der Dimension der Zielfunktion wird der Optimierungsprozeß noch schwieriger handhabbar und benötigt noch mehr Zeit, um ein Minimum zu finden.

[2]Diese Zweideutigkeiten treten auf, wenn eine Menge von Stellen nicht in *allgemeiner Lage* oder *degeneriert* ist, siehe [9] und [10].

Im Rahmen dieser Arbeit werden wir beide Möglichkeiten behandeln, da die entsprechenden Algorithmen sich nur in Details unterscheiden; und wir werden die Resultate beider Verfahren im Kapitel 4 vergleichen.

Kapitel 2

Grundlagen und verwandte Arbeiten

In diesem Kapitel beschreiben wir grundlegende Prinzipien, die von unserem Verfahren verwendet werden, und verwandte Arbeiten aus dem Bereich der Funktionsapproximation.

2.1 Das Simulated-Annealing-Verfahren

Simulated Annealing ist eine iterative Minimierungsmethode, die gut für Optimierungsprobleme großen Maßstabs geeignet ist, besonders solche, bei denen das gewünschte globale Minimum von sehr vielen, schlechteren lokalen Minima umgeben ist. Im Gegensatz zu Raffke-Optimierungsverfahren, welche stets den lokal besten Schritt von jeder Konfiguration aus wählen, kann Simulated Annealing auch einen Schritt nehmen, der den Wert der Zielfunktion erhöht (siehe Abbildung 2.1). Dies ermöglicht dem Algorithmus, aus dem Einzugsbereich eines lokalen Minimums zu entkommen und eventuell ein besseres lokales Minimum oder sogar das globale Minimum zu finden.

2.1.1 Das Erstarren einer Metallschmelze

Der Simulated-Annealing-Algorithmus ist die Übertragung eines chemisch-physikalischen Prozesses auf den Bereich der Optimierungen. Dieser Prozeß ist der Übergang einer metallischen Schmelze zurück in den Kristallzustand, während die Temperatur der Umgebung langsam verringert wird. Obwohl

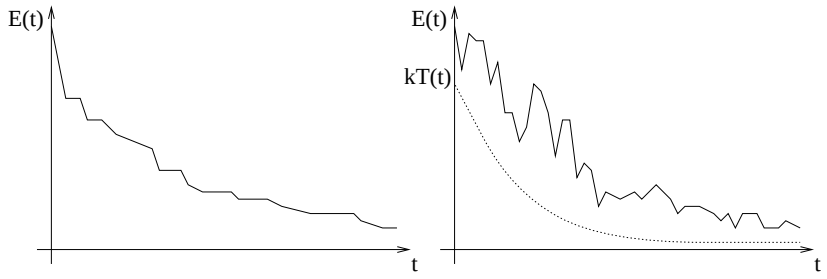

Abbildung 2.1: Skizze des Wertes der Zielfunktion E über der Iterationszeit t für Raffke-Optimierungsverfahren (links) und für Simulated Annealing (rechts). Mit dem Sinken der Temperatur kT werden positive Änderungen des Wertes der Zielfunktion immer unwahrscheinlicher.

dieser Prozeß in der Sprache der Optimierungstheorie ein Problem extrem hoher Dimension mit extrem vielen lokalen Minima ist, erstarrt die Schmelze doch zu einem Einkristall, der das globale Minimum der inneren Energie darstellt, solange die Umgebungstemperatur nur langsam genug gesenkt wird. Senkt man die Umgebungstemperatur zu schnell, kann dieses globale Minimum in der Regel nicht gefunden werden, und die Schmelze verbleibt in einem suboptimalen Minimum einer polykristallinen Struktur.

Eine Erklärung für den Erfolg der Energieminimierung während des Erstarrens einer Schmelze liegt in der Tatsache, daß die zufällige Wärmebewegung der Metallatome in der heißen Schmelze nicht nur Verringerungen des bereits erreichten Energiegehalts, sondern auch Erhöhungen desselben zulässt. In der Tat ist die Wahrscheinlichkeit, daß die Schmelze von einem Zustand innerer Energie E in einem Zustand $E+\Delta E > E$ überzugeht, durch Boltzmanns Gesetz der Thermodynamik[1] $p(\Delta E) = e^{-\Delta E/kT}$ gegeben. Hierbei ist $p(\Delta E)$ die Wahrscheinlichkeit des Übergangs zu einem um ΔE Joule (J) höheren Energieniveau, T ist die Temperatur der Schmelze in Kelvin (K), und $k = 1,38 \times 10^{-23}\mathrm{J/K}$ ist Boltzmanns Konstante der Thermodynamik[2], die den Zusammenhang zwischen der Temperatur eines Körpers und seiner inneren Energie herstellt.

Wenn also die Temperatur einer Schmelze während des Erstarrens langsam gesenkt wird, „probiert“ die Schmelze sehr viele verschiedene Konfigurationen frei aus; wird die Temperatur geringer, werden Zustände niedriger

[1] Benannt nach dem deutschen Physiker Ludwig Boltzmann (1844–1906; 62).

[2] s. o.

Energie immer wahrscheinlicher, bis beim Erreichen des Schmelzpunkts der minimale Energiezustand des Einkristalls erreicht wird. Sinkt die Temperatur dagegen zu schnell, kann der Energiezustand der Schmelze zu früh im Bereich eines suboptimalen lokalen Minimums gefangen werden, und der optimale Zustand wird nicht erreicht. Die genaue Kenntnis der optimalen Rate der Temperatursenkung ist damit von großer Bedeutung für sowohl die Geschwindigkeit der Erstarrung als auch die Güte des Kristalls: Bei geringer Geschwindigkeit dauert es sehr lange, bis die Schmelztemperatur erreicht wird, bei großer Geschwindigkeit wird eventuell nur ein schlechtes lokales Minimum erreicht.

2.1.2 Übertragung auf beliebige Optimierungsprobleme

Der Simulated-Annealing-Algorithmus ist eine weitestgehend analoge Übertragung des natürlichen Erstarrungsprozesses auf beliebige Minimierungsprobleme[3]. Ein solches allgemeines Minimierungsproblem wird anhand dieser Analogie wie folgt beschrieben:

- Eine Konfiguration ist nicht länger das Tupel der Positionen aller beteiligten Metallatome, sondern eine Variable C eines beliebigen Typs K, des *Konfigurationsraums*. Im häufigen Fall der Minimierung einer Funktion von k reellen Veränderlichen ist K eine Untermenge des Standardvektorraums $\mathbf{R}^k$.

- Die zu minimierende Funktion ist nicht mehr die innere Energie der Metallschmelze, sondern eine beliebige, auf dem Typ K definierte, reellwertige *Zielfunktion* $E\colon K \to \mathbf{R}$.

- Die Änderung der aktuellen Konfiguration C wird nun anstatt durch die zufällige Wärmebewegung der Metallatome durch eine beliebige nichtdeterministische *Übergangsfunktion* $next\colon K \to K$ beschrieben. Der Nichtdeterminismus von *next* ist für den Ablauf des Optimierungsprozesses von entscheidender Bedeutung. Die Übergangsfunktion muß probabilistische Elemente enthalten, da eine streng deterministische

[3]Da jedes Optimierungsproblem durch Negation der Zielfunktion in ein Minimierungsproblem umgewandelt werden kann, ist Simulated Annealing auch für Maximierungsprobleme geeignet.

Änderung der aktuellen Konfiguration dem Grundprinzip des Simulated Annealing zuwiderliefe und das Optimierungsergebnis beeinträchtigen könnte.

- Der Term kT aus Boltzmanns Gesetz der Thermodynamik wird durch eine beliebige monoton fallende, positive reellwertige Funktion, den *Annealing Schedule* oder die *Temperaturfunktion* $kT\colon \mathbf{N}_0 \to \mathbf{R}^+$ ersetzt. Diese Funktion bestimmt für jeden Iterationsschritt n die Wahrscheinlichkeit $p(\Delta E) = e^{-\Delta E/kT(n)}$, mit der eine Erhöhung der Zielfunktion um ΔE hingenommen wird. In diesem Zusammenhang muß noch einmal auf ein verbreitetes Mißverständnis hingewiesen werden: Die „Temperatur“ des Simulated-Annealing-Prozesses ist *nicht* der momentane Wert der Zielfunktion E, sondern ein von E völlig unabhängiger Faktor, der lediglich die Wahrscheinlichkeit bestimmt, mit der ein E erhöhender Iterationsschritt dennoch akzeptiert wird (siehe Abbildung 2.1).

Der Simulated-Annealing-Algorithmus in seiner generischen Form, auch als *Metropolis-Algorithmus* bekannt, ist in Algorithmus 2.1 angegeben:

Eingaben:

- Eine Menge K von Konfigurationen.
- Die zu minimierende Funktion $E\colon K \to \mathbf{R}$.
- Eine initiale Konfiguration $C \in K$.
- Eine nichtdeterministische Funktion $next\colon K \to K$, die eine Konfiguration in die nächste überführt.
- Eine Temperaturfunktion $kT\colon \mathbf{N}_0 \to \mathbf{R}^+$.
- Ein Prädikat *isFinished*, das das Ende der Iteration bestimmt.

Lokale:

- Ein Iterationszähler $n \in N_o$.
- Eine Funktion $prob\colon [0,1] \to \{0,1\}$ mit der Eigenschaft, daß die Wahrscheinlichkeit $p\big(prob(x) = 1\big) = x$ ist.
- Eine Konfiguration $C' \in K$.
- Eine Zahl $\Delta E \in \mathbf{R}$.

Ausgaben:

- Eine Konfiguration $C \in K$, die E minimiert.

```
n := 0;  /* Iterationszähler */
while not isFinished do
    begin
    C' := next(C);  /* Teste eine andere Konfiguration */
    ΔE := E(C') − E(C);  /* Erhöhung der Zielfunktion */
    if ΔE < 0 then  /* Guter Schritt */
        C := C';  /* Akzeptiere Schritt */
    else  /* Schlechter Schritt */
        if prob (e^(−ΔE/kT(n))) then
            C := C';  /* Akzeptiere Schritt trotzdem */
    n := n + 1;  /* Nächster Iterationsschritt */
    end  /* while */
return C;  /* Gebe endgültige Konfiguration zurück */
```

Algorithmus 2.1: Generischer Simulated-Annealing-Algorithmus.

2.1.3 Die Optimierungsgüte beeinflussende Faktoren

Die Qualität eines Optimierungsverfahrens kann grob durch zwei Parameter beschrieben werden: Die Güte der gelieferten endgültigen Konfiguration im Bezug auf das theoretisch erreichbare globale Optimum, und die Zeit, bis der Algorithmus terminiert. Im Spezialfall des Simulated Annealing wird die Qualität hauptsächlich von wiederum zwei Faktoren bestimmt: von der gewählten Übergangsfunktion *next* und der gewählten Temperaturfunktion kT.

Die geeignete Wahl der Übergangsfunktion

Theoretisch kann für den Simulated-Annealing-Algorithmus *jede beliebige* Übergangsfunktion *next* benutzt werden, solange sie nur *nichtdeterministisch* ist. Es macht gerade die Attraktivität dieses Verfahrens aus, daß man beliebige Optimierungsprobleme unter minimalem Einsatz von a-priori-Wissen über die Gestalt des Problems damit angreifen kann. Senkt man die Temperatur kT langsam genug, geht die Wahrscheinlichkeit, daß die aktuelle Konfiguration nicht das globale Minimum darstellt, gegen Null, wenn man den Algorithmus unendlich lange laufen lässt. In der Praxis hat man natürlich nicht soviel Zeit, also muß man entweder Abstriche bei der Güte des Ergebnisses machen oder etwas mehr Arbeit in die Übergangsfunktion investieren. Experimente

haben gezeigt, daß die Ausnutzung von Wissen über das spezielle Problem den Prozeß bei gleichbleibender Güte des Ergebnisses extrem beschleunigen kann. Dies ist ein weiterer Vorteil von Simulated Annealing gegenüber anderen Verfahren: Da der Iterationsschritt nicht vom Optimierungsverfahren beschränkt wird (so er nur nichtdeterministisch ist), kann Wissen über das Problem einfach ausgenutzt werden. Im speziellen Fall der Linearen-Spline-Approximation geht das soweit, daß das von vielen als „unterlegen“ oder „unzuverlässig“ abgetane Verfahren anderen, eingeschränkteren Verfahren das Wasser abgräbt. Der Iterationsschritt für Lineare-Spline-Approximation und die verwendeten Heuristiken werden im Abschnitt 3.4 ausführlich behandelt.

Die geeignete Wahl der Temperaturfunktion

Die Temperaturfunktion kT kann eine beliebige monoton fallende positiv reellwertige Funktion sein. Da die Wahl von kT analog zum Prozeß des Erstarrens einer Metallschmelze sowohl die Geschwindigkeit der Minimierung als auch die „Minimalität“ des Ergebnisses beeinflußt, ist ihre Wahl selbst wiederum ein Optimierungsproblem: Wenn kT zu schnell fällt, kann der Algorithmus im Bereich eines schlechten lokalen Minimums gefangen werden; fällt kT hingegen zu langsam, braucht der Algorithmus gewöhnlich eine sehr lange Zeit bis die Folge der Konfigurationen gegen ein dann sehr gutes Minimum konvergiert. Zum Glück wurden diverse Heuristiken für die Wahl der Temperaturfunktion entwickelt, die sich in einem weiten Bereich von Optimierungsproblemen bewährt haben. Nützliche Heuristiken für das Problem der Linearen-Spline-Approximation werden im Kapitel 4 aufgeführt.

2.2 Eine Metrik für Streudatenapproximationen

Der Kern eines jeden Optimierungsproblems ist die zu optimierende Funktion selbst. In der Regel ist diese Funktion von vornherein festgelegt, in unserem Fall der optimalen Linearen-Spline-Approximation jedoch ist das Ziel nicht die Minimierung der gegebenen Funktion, sondern ihre optimale Approximation durch einen linearen Spline mit einer festgelegten Vertexzahl. Um dieses Problem als Optimierung beschreiben zu können, muß also zuerst eine geeignete Funktion gefunden werden, die die Qualität einer Approximation beschreibt und die dann durch das Optimierungsverfahren minimiert wird.

Da sowohl die zu approximierende Funktion selbst als auch der approximierende lineare Spline Funktionen im mathematischen Sinne sind, stehen grundsätzlich sämtliche Funktionsraummetriken als Kandidaten für Gütekriterien zur Verfügung, die die Analysis so reichlich zur Verfügung stellt (siehe [6]). Im speziellen Fall, daß die zu approximierende Funktion nur durch Streudaten, also eine Menge von zufällig über den Definitionsbereich verteilten Abtastwerten gegeben ist, können jedoch die meisten dieser Metriken nicht verwendet werden, da sie Wissen über analytische Eigenschaften der beteiligten Funktionen voraussetzen, die für Streudaten nicht bekannt sind.

Ist eine Metrik aber durch ein Integral über einen Operator definiert, der die beteiligten Funktionen punktweise verknüpft, wie z.B. die L^2-Metrik

$$L^2(f,g) := \sqrt{\int_D \big(f(x) - g(x)\big)^2 \, dx} \quad , \tag{2.1}$$

dann kann dieses Integral und somit die Metrik durch eine Variation der *Monte-Carlo-Integration* angenähert werden. Der originale Monte-Carlo-Algorithmus bestimmt eine (sehr große) Menge $X := \{x_1, \ldots, x_N\} \subset D$ von zufälligen Punkten aus dem Grundbereich des gewünschten Integrals und nähert das Integral $\int_D f(x)\,dx$ durch die Formel

$$\int_D f(x)\,dx \approx D \cdot \langle f \rangle \pm D\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{2.2}$$

an, wobei D das Volumen des Integrationsbereichs ist und $\langle f \rangle$ und $\langle f^2 \rangle$ als die arithmetischen Mittel über die Funktionswerte der N Punkte definiert sind:

$$\langle f \rangle := \frac{1}{N}\sum_{i=1}^{N} f(x_i) \quad , \qquad \langle f^2 \rangle := \frac{1}{N}\sum_{i=1}^{N} \big(f(x_i)\big)^2 \quad . \tag{2.3}$$

Der „$\pm$“-Term aus Formel 2.2 ist eine Ein-Standardabweichung-Fehlerschätzung für das Integral und keine absolute Fehlerschranke; außerdem ist der Fehler nicht normalverteilt, so daß die Fehlerangabe mehr als grobe Abschätzung verstanden werden sollte.

Wie auch immer, in unserem Fall der Approximation von Streudaten ist eine der Funktionen der L^2-Metrik nur durch zufällig über den Definitionsbereich verteilte Abtastwerte definiert, also kann man diese Tatsache ausnutzen um den L^2-Abstand zwischen dem approximierenden linearen Spline und der durch die Streudaten definierten Funktion mit Hilfe des Monte-

Carlo-Algorithmus zu berechnen. Man benutzt einfach die gesamte gegebene Streudatenmenge als Abtastmenge X und bestimmt in jedem Punkt $x_i \in X$ den Abstand des Streudatenpunkts vom linearen Spline. Da unser Ziel die optimale Approximation der gegebenen Funktion, nicht die Approximation des Graphen der Funktion ist, muß in jedem Punkt der Abstand zwischen Funktion und Spline in Ordinatenrichtung bestimmt werden, nicht der orthogonale Abstand zwischen Punkt und Spline (siehe Abbildung 2.2). Zu diesem Zweck bestimmt man den Funktionswert des Spline an der Stelle des Streudatenpunkts durch lineare Interpolation.

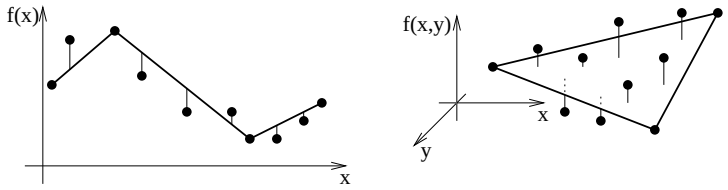

Abbildung 2.2: Einfluß von Streudatenpunkten auf den Abstand $L^2(f, s)$ zwischen Funktion f und linearem Spline s.

Ein anderer Vorteil dieser Art der Fehlerbestimmung ist, daß sie ohne Änderungen auch für Splines höherer Ordnung und auch für vektorwertige Funktionen eingesetzt werden kann, indem man als Abstand zwischen interpoliertem und gegebenem Punkt irgendeine Metrik des jeweiligen Vektorraumes verwendet. Algorithmus 2.2 beschreibt die Vorgehensweise im Detail:

Eingaben:

- Eine Menge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$ von Streudatenpunkten.
- Ein linearer Spline $a \in LS_{n,m}$, der S approximiert.

Lokale:

- Eine Zahl $E \in \mathbf{R}^+$.
- Ein Vertex $v \in (\mathbf{R}^n \times \mathbf{R}^m)$.
- Ein Simplex $c \in CS_n$.
- Ein Funktionswert $\overline{f} \in \mathbf{R}^m$.
- Eine Funktion *area*: $LS_{n,m} \rightarrow \mathbf{R}^+$, die die Fläche des Definitionsbereichs eines Splines berechnet.
- Eine Funktion *findSimplex*: $LS_{n,m} \times \mathbf{R}^n \rightarrow CS_n$, $(a, s) \mapsto c$, die den eine Stelle s enthaltenen Simplex c der Verbindungsmenge eines Splines a liefert.

- Eine Funktion *interpolate*: $CS_n \times \mathbf{R}^n \to \mathbf{R}^m$, $(c, s) \mapsto f$, die den Funktionswert f eines Simplex c an einer Stelle s bestimmt.

Ausgaben:

- Der L^2-Abstand des Splines a von der Streudatenmenge S.

$E := 0$; /* Abstandswert */
for each $v \in S$ **do**
begin
$c := \mathit{findSimplex}(a, v.s)$; /* Bestimme Simplex c, der $v.s$ enthält */
$\overline{f} := \mathit{interpolate}(c, v.s)$; /* Bestimme Wert von c an der Stelle $v.s$ */
$E := E + (\overline{f} - v.f)^2$; /* Addiere Quadrat des Ordinatenabstands */
end
return $\sqrt{(E/|S|) \cdot \mathit{area}(a)}$;

Algorithmus 2.2: Monte-Carlo-Approximation der L^2-Metrik

2.3 Triangulierungen

Für jede Dimension n wird ein n-variater linearer Spline eindeutig definiert durch das Tupel seiner Kontrollpunkte und sein Simplexnetz, das angibt, wie die Kontrollpunkte durch n-dimensionale Simplizes verbunden sind. Im univariaten Fall ist die Verbindungsmenge bereits durch die Vertexplazierung eindeutig definiert, und die Konstruktion der Verbindungsmenge ist trivial: Es genügt, die Kontrollpunkte nach ihren Stellen (Abszissenwerten) zu sortieren, und die Stellen in der Sortierung benachbarter Kontrollpunkte durch Intervalle zu verbinden.

Im Falle zweier oder mehrerer Variablen ist das Simplexnetz nicht mehr durch die Kontrollpunkte festgelegt, und die Konstruktion des Simplexnetzes ist alles andere als trivial. Die folgenden Abschnitte behandeln den bivariaten Fall, in dem das Simplexnetz eine Triangulierung ist. Sie gehen von einem festgelegten Kontrollpunkttupel aus und beschreiben verschiedene Klassen von Triangulierungen und Wege zu ihrer Konstruktion. Im folgenden bezeichnen wir die Menge aller Triangulierungen als $\mathcal{T}$ und die Menge der Triangulierungen einer festen Punktmenge P als $\mathcal{T}(P)$.

2.3.1 Delaunay-Triangulierungen

Delaunay-Triangulierungen[4] sind eine spezielle Klasse von Triangulierungen, die eine bestimmte globale Bedingung erfüllen (siehe [7], [8]). Eine Formulierung dieser Bedingung ist das *Globale Umkreiskriterium*:

Definition 2.1 Eine Triangulierung $T \in \mathcal{T}(P)$ einer Punktmenge P erfüllt das Globale Umkreiskriterium, wenn der Umkreis jedes Dreiecks $t \in T$ keine Punkte aus P außer den Eckpunkten von t enthält (siehe Abbildung 2.3). Eine Triangulierung $D \in \mathcal{T}(P)$, die das Globale Umkreiskriterium erfüllt, heißt Delaunay-Triangulierung.

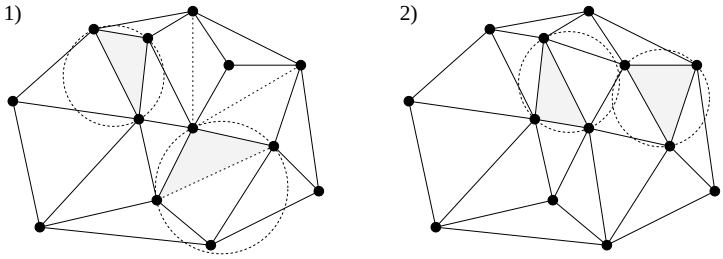

Abbildung 2.3: Allgemeine und Delaunay-Triangulierung. 1) Allgemeine Triangulierung einer Punktmenge. Die Umkreise der schattierten Dreiecke sind ausgezeichnet. Die gestrichelten Kanten verletzen die Umkreis-Bedingung. 2) Delaunay-Triangulierung der gleichen Punktmenge. Die Umkreise der schattierten Dreiecke sind ausgezeichnet.

Warum nun sind Delaunay-Triangulierungen für optimale Lineare-Spline-Approximationen von Interesse? Der erste Grund ist, daß die Delaunay-Triangulierung einer Punktmenge bis auf unbedeutende Zweideutigkeiten durch die Punktmenge festgelegt ist. Dies gibt uns die Möglichkeit, die Verbindungsmenge eines festgelegten Kontrollpunkttupels ohne Einführung weiterer Freiheitsgrade zu bestimmen. Der zweite Grund für die Bedeutung der Delaunay-Triangulierung wird ersichtlich, wenn man eine Folgerung des Globalen Umkreiskriteriums betrachtet:

Definition und Satz 2.2 Sei $T \in \mathcal{T}(P)$ eine Triangulierung einer Punktmenge P. Dann bezeichnet man das n-Tupel $A(T) := (\alpha_1, \ldots, \alpha_n) \in \mathbf{R}^n$ als

[4]Benannt nach dem russischen Mathematiker Boris Nikolaevitch Delaunay.

Winkelvektor von T, wobei $(\alpha_1, \ldots, \alpha_n)$ die nach Größe sortierte Folge aller inneren Winkel aller Dreiecke $t \in T$ ist. Seien $A(T_1) = (\alpha_1^{(1)}, \ldots, \alpha_n^{(1)})$ und $A(T_2) = (\alpha_1^{(2)}, \ldots, \alpha_n^{(2)})$ die Winkelvektoren zweier Triangulierungen $T_1, T_2 \in \mathcal{T}(P)$ der gleichen Punktmenge P. Definiert man nun eine Ordnung $\leq_{\mathcal{T}}$ auf der Menge $\mathcal{T}(P)$ durch die (lexikalische) Ordnung der Winkelvektoren,

$$\begin{aligned} T_1 \leq_{\mathcal{T}} T_2) \ :\Longleftrightarrow \\ \nexists\, 1 \leq i \leq n : \left(\alpha_i^{(1)} > \alpha_i^{(2)} \wedge \forall\, 1 \leq j < i : \alpha_j^{(1)} = \alpha_j^{(2)}\right) \quad , \end{aligned} \tag{2.4}$$

dann ist die Delaunay-Triangulierung D der Punktmenge P diejenige Triangulierung, die den größten Winkelvektor hat, mit anderen Worten,

$$\forall\, T \in \mathcal{T}(P) : T \leq D \quad . \tag{2.5}$$

Diese *Winkeloptimalität* der Delaunay-Triangulierungen hat zur Folge, daß die Länge aller Dreieckskanten minimiert wird, und daß nur „wohlproportionierte“ Dreiecke in der Triangulierung vorkommen. Benutzt man nun eine Delaunay-Triangulierung als Verbindungsmenge eines bivariaten linearen Splines, so wird der Wert des Splines an einem beliebigen Punkt innerhalb der konvexen Hülle der Kontrollpunkte nur von Kontrollpunkten beeinflußt, die so nah wie möglich an dem Punkt liegen (siehe Abbildung 2.4). Diese Eigenschaft macht die Klasse der Delaunay-Triangulierungen zu einer guten Wahl für die Approximation unbekannter Funktionen.

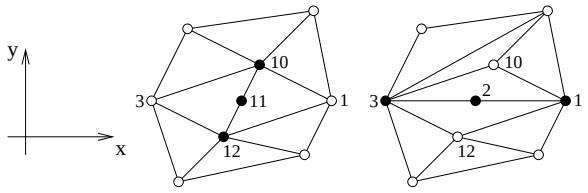

Abbildung 2.4: Beeinflussung eines Punktes durch Kontrollpunkte eines linearen Splines bei Verwendung einer Delaunay-Triangulierung (links) und einer anderen Triangulierung (rechts). Die Funktionswerte des Splines sind an den Punkten angeschrieben. Im linken Bild wird der Punkt durch „nähere“ Kontrollpunkte bestimmt.

Erzeugung von Delaunay-Triangulierungen

Zur Erzeugung einer Delaunay-Triangulierung einer gegebenen Punktmenge P benutzen wir den von Guibas, Knuth und Sharir [17] vorgeschlagenen iterativen Algorithmus. Die generelle Idee dieses Verfahrens ist das Einfügen eines Punktes pro Schritt in die momentane Triangulierung, und das anschließende „Rotieren“ illegaler Kanten (siehe Abbildung 2.5). Wir bezeichnen die Menge aller Kanten mit $\mathcal{K}$ und die Menge aller möglichen Kanten in einer Punktmenge P als $\mathcal{K}(P)$ und weiterhin eine Kante als *illegal*, wenn eines der die Kante teilenden Dreiecke das Globale Umkreiskriterium verletzt. Ist dies der Fall, dann verletzt auch das andere Dreieck das Umkreiskriterium, und weiterhin ist die Kante die Diagonale eines konvexen Vierecks. Es ist dann also möglich, die beiden Dreiecke so umzudefinieren, daß sie die andere Diagonale des durch sie definierten Vierecks als gemeinsame Kante haben; diesen Vorgang bezeichnen wir als *Kantenrotation*. Nach einer solchen Rotation erfüllen beide beteiligten Dreiecke das Globale Umkreiskriterium.

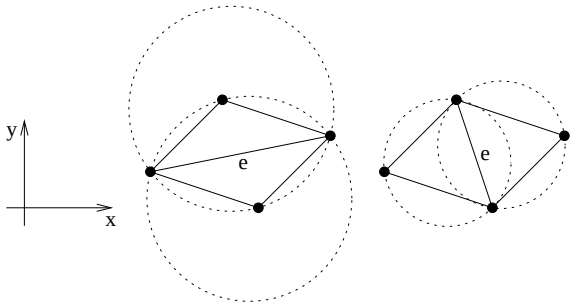

Abbildung 2.5: Rotation einer Kante e, die das globale Umkreiskriterium verletzt.

Ein wichtiger Unteralgorithmus der iterativen Delaunay-Triangulierung ist die Prozedur *rotateSuspectEdges*, die eine Menge $K \subset \mathcal{K}(P)$ von „verdächtigen“ Kanten einer Triangulierung $T \in \mathcal{T}(P)$ übergeben bekommt und das Globale Umkreiskriterium basierend auf diesen eventuell illegalen Kanten wiederherstellt. Diese Prozedur ist in Algorithmus 2.3 angegeben:

Eingaben:

- Eine Triangulierung $T \in \mathcal{T}(P)$ einer Punktmenge P.
- Eine Menge $K \subset \mathcal{K}(P)$ von verdächtigen Kanten von T.

Lokale:

- Eine Kante $k \in \mathcal{K}(P)$.
- Eine Funktion $checkEdge: \mathcal{T}(P) \times \mathcal{K}(P) \to \{0,1\}$, $(T,k) \mapsto illegal$, die testet, ob eine Kante k in einer Triangulierung T illegal ist.
- Eine Funktion $rotateEdge: \mathcal{T}(P) \times \mathcal{K}(P) \to \mathcal{T}(P) \times \mathcal{K}(P)$, $(T,k) \mapsto (T',k')$, die eine Kante k einer Triangulierung T rotiert.
- Eine Funktion $getQuadEdges: \mathcal{T}(P) \times \mathcal{K}(P) \to P(\mathcal{K}(P))$, $(T,k) \mapsto \{k_1,\ldots,k_4\}$, die die Menge aller Kanten des Vierecks in der Triangulierung T liefert, dessen Diagonale k ist.

Ausgaben:

- Die korrigierte Triangulierung T.

while $K \neq \emptyset$ **do**
  begin
  $k :\in K$; /* Wähle ein beliebiges Element aus der Kantenmenge */
  $K := K \setminus \{k\}$; /* Entferne k aus der Kantenmenge */
  if $checkEdge(T,k)$ **then** /* Teste, ob die Kante illegal ist */
    begin
    $(T,k) := rotateEdge(T,k)$; /* Rotiere Kante k */
    $K := K \cup getQuadEdges(T,k)$; /* Füge Kanten des Vierecks um k in die Kantenmenge ein */
    end
  end /* while */
return T;

Algorithmus 2.3: Überprüfung verdächtiger Kanten.

Dieser Algorithmus terminiert stets, da das Globale Umkreiskriterium durch eine endliche Zahl von Kantenrotationen stets wiederhergestellt werden kann. Übergibt man dem Algorithmus alle potentiell illegalen Kanten, so ist die Wiederherstellung des Umkreiskriteriums garantiert und das Ergebnis ist eine Delaunay-Triangulierung der Punktmenge der ursprünglichen Triangulierung.

Um nun eine Delaunay-Triangulierung einer Punktmenge P zu erzeugen, bestimmt man zuerst eine Delaunay-Triangulierung der konvexen Hülle

von P, und fügt dann alle Punkte von P in diese Triangulierung ein. Um einen Punkt $p \in P$ einzufügen, bestimmt man zuerst jenes Dreieck $t \in T$, das p überdeckt, und spaltet dieses Dreieck durch Einfügen von p in drei neue Dreiecke. Da die Kanten k_1, k_2 und k_3 von t durch das Einfügen von p potentiell illegal geworden sind, übergibt man die Menge $\{k_1, k_2, k_3\}$ an die Prozedur *rotateSuspectEdges*, die das Globale Umkreiskriterium wiederherstellt (siehe Algorithmus 2.3 und Abbildung 2.6).

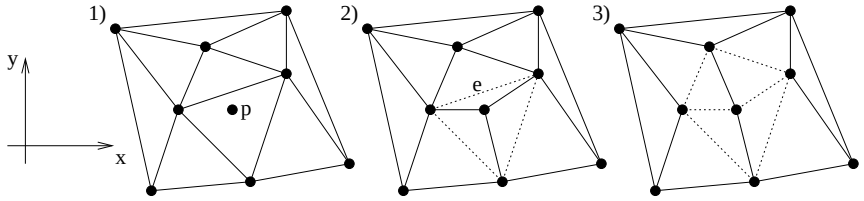

Abbildung 2.6: Einfügen eines Punktes p in eine Delaunay-Triangulierung. 1) initialer Zustand, 2) Spaltung des Dreiecks t und initiale Liste verdächtiger Kanten (e ist illegal), 3) Rotation von e und neue Kantenliste. Da keine weiteren Kanten illegal sind, ist 3) gleichzeitig der Endzustand.

Die Gültigkeit des Globalen Umkreiskriteriums ist also eine Invariante des Punkteeinfügens, und damit ist T nach Einfügen aller Punkte eine Delaunay-Triangulierung von P. Der einzige mögliche Spezialfall des Einfügens tritt auf, wenn ein einzufügender Punkt p direkt auf einer in T vorkommenden Kante liegt. Diesen Fall darf man aber getrost ignorieren und eins der p enthaltenden Dreiecke spalten, da das entstehende „degenerierte“ Dreieck von der folgenden Kantenrotation rektifiziert wird. Der vollständige Algorithmus zur Delaunay-Triangulierung einer Punktmenge P ist in Algorithmus 2.4 angegeben:

Eingaben:

- Eine Punktmenge $P \subset \mathbf{R}^2$.

Lokale:

- Eine Funktion *findConvexHull*: $P(\mathbf{R}^2) \to P(\mathbf{R}^2)$, $P \mapsto CH$, die die Menge CH derjenigen Punkte aus P liefert, die die konvexe Hülle von P aufspannen.
- Eine Funktion *triangulate*: $P(\mathbf{R}^2) \to \mathcal{T}$, $P \mapsto T \in \mathcal{T}(P)$, die eine beliebige Triangulierung T der Punktmenge P erzeugt.
- Eine Funktion *getEdges*: $\mathcal{T} \to \mathcal{K}$, $T \in \mathcal{T}(P) \mapsto K \subset \mathcal{K}(P)$, die die Menge K aller Kanten einer Triangulierung T liefert.
- Eine Funktion *insertVertex*: $\mathcal{T} \times \mathbf{R}^2 \to \mathcal{T} \times P(\mathcal{K})$, $(T \in \mathcal{T}(P), p \notin P) \mapsto \Big(T' \in \mathcal{T}\big(P \cup \{p\}\big), K \subset \mathcal{K}\big(P \cup \{p\}\big)\Big)$, die einen Punkt p in eine Triangulierung einfügt und die drei Kanten des gespaltenen Dreiecks zurückliefert.

Ausgaben:

- Eine Delaunay-Triangulierung $D \in \mathcal{T}(P)$ der Punktmenge P.

$CH :=$ *findConvexHull*(P); /* Bestimme äußere Punkte von P */
$T :=$ *triangulate*(CH); /* Trianguliere die konvexe Hülle */
$T :=$ *rotateSuspectEdges*$(T,$ *getEdges*$(T))$;
/* T ist nun Delaunay-Triangulierung der Punktmenge CH */
while $CH \neq P$ **do** /* Solange noch Punkte einzufügen sind */
begin
$p :\in P \setminus CH$; /* Wähle einen einzufügenden Punkt */
$CH := CH \cup \{p\}$; /* Füge p in die Punktmenge CH ein */
$(T, K) :=$ *insertVertex*(T,p); /* Füge p in die Triangulierung ein */
$T :=$ *rotateSuspectEdges*(T,K);
/* T ist wieder Delaunay-Triangulierung von CH */
end
return T;

Algorithmus 2.4: Delaunay-Triangulierung einer Punktmenge P.

Zum Aufwand von Algorithmus 2.4: Für $|P| = n$ kann die konvexe Hülle von P in $O(n \log n)$ Schritten bestimmt werden. Ist die Anzahl der Punkte aus P auf dem Rand der konvexen Hülle gleich k, so hat die Triangulierung der konvexen Hülle $k-3$ innere Kanten, und der initiale Rotationsschritt ist nach $O(k-3) = O(n)$ Schritten beendet. Enthält die aktuelle Triangulie-

rung T n Punkte, davon k auf dem Rand der konvexen Hülle, dann enthält T $2n-2-k = O(n)$ Dreiecke. Das einen Punkt p enthaltende Dreieck $t \in T$ kann durch geeignete Verfahren in $O\big(\log(2n-2-k)\big) = O(\log n)$ Schritten gefunden werden. Der Erwartungswert des Zeitaufwandes für die Wiederherstellung des Globalen Umkreiskriteriums ist nach einem Satz von Knuth [17] konstant, daher ist die erwartete Zeitkomplexität des gesamten Algorithmus $O(n \log n)$. Auch wenn die erwartete Zeitkomplexität die gleiche ist wie bei Erzeugung der Delaunay-Triangulierung durch klassische Verfahren, erwarten wir trotzdem nicht, daß dieser Algorithmus genauso schnell ist. Wir verwenden ihn dennoch, da der Algorithmus ebenso zur iterativen Änderung einer Triangulierung durch Einfügen oder Entfernen von Punkten benutzt werden kann; ein Problem, das in dem von uns verwendeten Iterationsverfahren in jedem Schritt auftaucht. Unsere momentane Implementierung benutzt darüberhinaus einen einfachen, linearen Algorithmus zur Auffindung des einen Punkt enthaltenden Dreiecks, daher ist die erwartete Zeitkomplexität in unserem Fall nur $O(n^2)$, was sich aber für sämtliche Testfälle ($n \leq 4.000$) als schnell genug erwiesen hat.

2.3.2 Datenabhängige Triangulierungen

Obwohl die Benutzung von Delaunay-Triangulierungen als Verbindungsmengen von linearen Splines eine gute erste Wahl für die Approximation unbekannter Funktionen ist, läßt sich doch zeigen, daß die Delaunay-Triangulierungen nur für *eine einzige* Klasse bivariater Funktion wirklich optimal sind: für die rotationssymetrischen Parabeln $f(x,y) := \big((x-x_0)^2 + (y-y_0)^2\big) \cdot a$ für beliebige a, x_0 und y_0.

Die *datenabhängige Triangulierung*, d.h. die Erzeugung einer optimalen Triangulierung für einen approximierenden linearen Spline mit festgelegtem Kontrollpunkttupel, ist selbst wiederum ein Optimierungsproblem, welches mit den gleichen Methoden behandelt werden kann wie unser (allgemeineres) Problem der optimalen Linearen-Spline-Approximation. Larry L. Schumaker beschreibt in seinem Artikel [11] den Einsatz von Simulated Annealing für die datenabhängige Triangulierung. Unser Algorithmus behandelt die Optimierung der Verbindungsmengen parallel zur Optimierung der Vertexplazierung; der Fall der datenabhängigen Triangulierung ist als Spezialfall enthalten.

Kapitel 3

Algorithmen und Datenstrukturen

In diesem Kapitel beschreiben wir die im Rahmen unseres Verfahrens angewendeten Algorithmen und die zugrundeliegenden Datenstrukturen.

3.1 Der Approximierungsalgorithmus

Wir beginnen unsere Darstellung mit einer Gesamtübersicht des Approximierungsalgorithmus (siehe Algorithmus 3.1); die folgenden Abschnitte behandeln die einzelnen Schritte und Unteralgorithmen im Detail.

Eingaben:

- Eine zu approximierende Streudatenmenge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$.
- Die Anzahl $N \in \mathbf{N}$ der Vertizes in der Approximation.
- (Optional) Ein linearer Spline $a_{\mathbf{sub}} \in LS_{n,m}(N')$ mit $N' < N$ Vertizes, der die nächstniedrigere Stufe in einer Approximationshierarchie darstellt.
- Faktoren $f_g, f_l \in \mathbf{R}^+$ und eine Wahrscheinlichkeit $p_r \in [0,1]$, die Parameter für den Algorithmus zur Änderung einer aktuellen Konfiguration sind (siehe Abschnitt 3.4).
- Die Wahrscheinlichkeit $p_0 \in (0,1)$, mit der ein schlechter Iterationsschritt zu Beginn der Iteration akzeptiert werden soll.
- Ein Faktor $f_t \in (0,1)$, der den Abfall der Temperatur kT im Verlauf der Iteration bestimmt.

Lokale:

- Ein linearer Spline $a \in LS_{n,m}(N)$ mit N Vertizes, der die aktuelle Konfigurations des Optimierungsalgorithmus darstellt.
- Eine Menge $B \subset S$ von im Verlauf der Iteration fix bleibenden Kontrollpunkten.
- Eine Funktion *initialConfiguration*: $P(\mathbf{R}^n \times \mathbf{R}^m) \times \mathbf{N} \times LS_{n,m}(N') \times [0,1] \to LS_{n,m}(N) \times P(\mathbf{R}^n \times \mathbf{R}^m)$, die die initiale Approximation einer Streudatenmenge und die Menge fixer Kontrollpunkte bestimmt (siehe Abschnitt 3.2).
- Eine Zahl $kT \in \mathbf{R}^+$, die die momentane Temperatur des Simulated-Annealing-Algorithmus beschreibt.
- Eine Funktion *estimateTemperature*: $\mathbf{R}^+ \times \mathbf{R}^+ \times [0,1] \times P(\mathbf{R}^n \times \mathbf{R}^n) \times LS_{n,m}(N) \times P(\mathbf{R}^n \times \mathbf{R}^m) \times (0,1) \to \mathbf{R}^+$, die die initiale Temperatur der Iteration bestimmt (siehe Abschnitt 3.3).
- Die Zahl *temperatureSteps* $\in \mathbf{N}$ der Iterationsschritte, nach denen die aktuelle Temperatur kT um den Faktor f_t verringert wird.
- Die Zahl *steps* $\in \mathbf{N}_0$ der bisherigen Iterationsschritte.
- Ein Prädikat *isIterationFinished*, das das Ende der Iteration bestimmt.
- Ein Abstandswert $E \in \mathbf{R}^+$ und die Änderung $\Delta E \in \mathbf{R}$ dieses Wertes im aktuellen Schritt.
- Eine Funktion *calcDistance*: $P(\mathbf{R}^n \times \mathbf{R}^m) \times LS_{n,m} \to \mathbf{R}^+$, $(S, a) \mapsto E$, die den Abstand E eines linearen Splines a von der durch a approximierten Streudatenmenge S bestimmt. Gewöhnlich benutzen wir hier die in Abschnitt 2.2 definierte Variation der L^2-Metrik; prinzipiell kann aber jede beliebige Metrik benutzt werden.
- Eine probabilistische Funktion *changeConfiguration*: $\mathbf{R}^+ \times \mathbf{R}^+ \times [0,1] \times P(\mathbf{R}^n \times \mathbf{R}^m) \times LS_{n,m}(N) \times P(\mathbf{R}^n \times \mathbf{R}^m) \to LS_{n,m}(N)$, die einen Iterationsschritt auf einen linearen Spline anwendet und den geänderten Spline zurückgibt (siehe Abschnitt 3.4). Informationen, wie der Iterationsschritt rückgängig gemacht werden kann, werden im Undo-Puffer des Ergebnissplines abgelegt (siehe Abschnitt 3.5.4).
- Eine Funktion *clearUndoBuffer*: $LS_{n,m}(N) \to LS_{n,m}(N)$, die den Undo-Puffer eines linearen Splines löscht und somit alle zurückliegenden Änderungen des Splines endgültig akzeptiert (siehe Abschnitt 3.5.4).
- Eine Funktion *undoLastChange*: $LS_{n,m}(N) \to LS_{n,m}(N)$, die die letzte Änderung eines linearen Splines zurücknimmt, sofern der Undo-Puffer des Splines nicht zuvor gelöscht wurde (siehe Abschnitt 3.5.4).

Ausgaben:

- Ein linearer Spline $a \in LS_{n,m}(N)$ mit N Vertizes, der die Streudatenmenge S (hoffentlich) optimal approximiert.

$(a, B) := initialConfiguration(S, N, a_{\text{sub}}, p_r)$;
$kT := estimateTemperature(f_g, f_l, p_r, S, a, B, p_0)$;
$temperatureSteps := 5\big(N - |B|\big)$; /* Schritte pro Temperatur */
$steps := 0$;
while not *isIterationFinished* **do**
begin
$E := calcDistance(S, a)$; /* Bisheriger Abstand */
$a := changeConfiguration(f_g, f_l, p_r, S, a, B)$;
$\Delta E := calcDistance(S, a) - E$; /* Änderung dieses Schrittes */
if $\Delta E < 0$ **then** /* Guter Schritt */
$a := clearUndoBuffer(a)$; /* Akzeptiere Schritt */
else /* Schlechter Schritt */
if $prob\left(e^{-\Delta E/kT}\right)$ **then**
$a := clearUndoBuffer(a)$; /* Akzeptiere Schritt trotzdem */
else
$a := undoLastChange(a)$; /* Mache Schritt rückgängig */
$steps := steps + 1$;
if $steps$ mod $temperatureSteps = 0$ **then**
$kT := kT \cdot t_f$; /* Verringere Temperatur um Faktor t_f */
end
return a;

Algorithmus 3.1: Der Approximierungsalgorithmus.

Die Definition des Prädikats *isIterationFinished*, das das Ende der Iteration bestimmt, kann im wesentlichen auf zwei Arten erfolgen:

- Man kann die Iteration für eine festgelegte Zahl *numberOfSteps* von Schritten laufen lassen. In diesem Fall hätte das Prädikat die Form $steps < numberOfSteps$. Wir benutzten diese Methode des Abbruchs für alle Experimente in Kapitel 4, um die Abstandsfunktionen der Experimente besser vergleichen zu können.

- Man kann die Iteration solange laufen lassen, bis der Simulated-Annealing-Algorithmus gegen ein (hoffentlich globales) Minimum konvergiert ist. Um die Konvergenz zu entscheiden, benutzen wir folgen-

des Kriterium: Die Iteration wird beendet, wenn während der letzten *numZeroSteps* Schritte kein einziger Schritt mehr vom Algorithmus akzeptiert wurde. In unseren Experimenten setzen wir *numZeroSteps* gleich 2.000.

3.2 Erzeugung einer Startkonfiguration

Um eine initiale Konfiguration für einen eine Streudatenmenge S approximierenden linearen Spline $a \in LS_{n,m}(N)$ mit N Kontrollpunkten zu bestimmen, definieren wir zuerst eine Basismenge $B \subset S$ von Vertizes aus S, die bei jeder Änderung der Konfiguration von a im Verlauf der Iteration erhalten bleiben muß. Wir verlangen, daß alle von unserem Algorithmus erzeugten Approximationen stets über der konvexen Hülle der gegebenen Streudatenmenge definiert sind. Um dies zu gewährleisten, müssen alle äußeren Vertizes von S (definiert wie in Abschnitt 1.1) in der Basismenge B enthalten sein. Wird dem Approximationsalgorithmus eine Unterapproximation $a_{\text{sub}} \in LS_{n,m}(N')$ mit $N' < N$ Kontrollpunkten übergeben, dann enthält die Vertexplazierung von a_{sub} bereits alle äußeren Vertizes von S und wir definieren diese Vertexplazierung als die Basismenge B. Anderenfalls definieren wir B als genau die Menge der äußeren Punkte von S.

Im zweiten Schritt erzeugen wir eine Konfiguration für a, die die Menge B als Vertexplazierung hat. Im univariaten Fall ist diese Konfiguration durch B festgelegt; im bivariaten Fall erzeugen wir eine Delaunay-Triangulierung der Stellen aus B als Simplexnetz. Ist B durch eine Unterapproximation a_{sub} bestimmt, kopieren wir in beiden Fällen das Simplexnetz von a_{sub}.

Im dritten Schritt wird die Konfiguration von a durch Einfügen der übrigen $N - |B|$ Kontrollpunkte vervollständigt. Hierzu wählen wir jeweils einen neuen Vertex zufällig aus der Menge S aus und fügen ihn durch den in Abschnitt 3.4.2 beschriebenen Algorithmus in die aktuelle Konfiguration von a ein. Ist im bivariaten Fall keine Unterapproximation gegeben, oder ist die Wahrscheinlichkeit p_r gleich null[1], dann stellen wir die Delaunay-Bedingung der initialen Konfiguration nach dem Einfügen jedes neuen Kontrollpunkts wieder her (siehe Abbildung 3.1). Im anderen Fall, wenn also eine Unterapproximation gegeben ist und p_r größer als null ist, gehen wir davon aus, daß die Konfiguration von a_{sub} eine optimale datenabhängige Triangulierung

[1] Dann wird im Verlauf der Iteration die Delaunay-Bedingung des Simplexnetzes erzwungen (siehe Abschnitt 1.3).

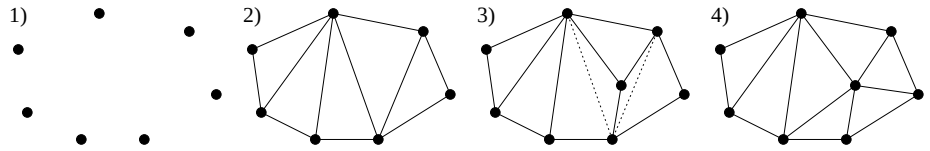

Abbildung 3.1: Erzeugung einer initialen Konfiguration. 1) Bestimmung der äußeren Vertizes von S; 2) Delaunay-Triangulierung von B; 3) Einfügen eines weiteren Vertex (die gestrichelten Kanten verletzen die Delaunay-Bedingung); 4) Wiederherstellung der Delaunay-Bedingung durch Rotation zweier Kanten.

enthält. Eine Erzwingung der Delaunay-Triangulierung würde dann diese optimale Triangulierung zerstören (siehe Abbildung 3.2). Experimente haben gezeigt, daß diese Annahme in der Regel wahr ist. Algorithmus 3.2 gibt dieses Verfahren im Detail an:

Eingaben:

- Eine Streudatenmenge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$.
- Eine Zahl $N \in \mathbf{N}$ von Kontrollpunkten.
- (Optional) Ein linearer Spline $a_{\text{sub}} = (a_{\text{sub}}.V, a_{\text{sub}}.SM) \in LS_{n,m}(N')$ mit $N' < N$ Kontrollpunkten, der eine niedrigere Stufe in einer Approximationshierarchie darstellt.
- Die Wahrscheinlichkeit $p_r \in [0,1]$, mit der im Algorithmus zur Änderung der aktuellen Konfiguration Kanten rotiert werden (siehe Abschnitt 3.4).

Lokale:

- Ein linearer Spline $a = (a.V, a.SM) \in LS_{n,m}$.
- Die Menge $B \subset a.V$ der fixen Kontrollpunkte von a.
- Ein Flag *enforceDelaunay* $\in \{0,1\}$, das angibt, ob die Delaunay-Bedingung beim Einfügen von Vertizes aufrechterhalten werden soll.
- Eine Funktion *createSimplexMesh*: $P(\mathbf{R}^n \times \mathbf{R}^m) \to CS_n$, $V \mapsto SM$, die das Simplexnetz SM einer Vertexplazierung V bestimmt. Im bivariaten Fall berechnet *createSimplexMesh* eine Delaunay-Triangulierung der Vertexplazierung V.
- Eine Funktion *insertVertex*: $LS_{n,m}(k) \times (\mathbf{R}^n \times \mathbf{R}^m) \to LS_{n,m}(k+1)$, $(a, v) \mapsto a'$, die einen Vertex v in einen linearen Spline a einfügt.

- Eine Funktion *restoreDelaunayProperty*: $LS_{n,m}(N) \to LS_{n,m}(N)$, $a \mapsto a'$, die die Delaunay-Bedingung des Simplexnetzes von a wiederherstellt.

Ausgaben:

- Die initiale Konfiguration $a \in LS_{n,m}(N)$.
- Die Menge $B \subset a.V$ der fixen Kontrollpunkte von a.

if $N' > 0$ **then** /* Ist a_{sub} gegeben? */
  begin
  /* Kopiere die Konfiguration von a_{sub}: */
  $a.V := B := a_{\text{sub}}.V$;
  $a.SM := a_{\text{sub}}.SM$;
  $enforceDelaunay := (p_r = 0)$;
  end
else
  begin
  /* Erzeuge eine Basiskonfiguration aus der konvexen Hülle von S: */
  $a.V := B := getExteriorVertices(S)$;
  $a.SM := createSimplexMesh(B)$;
  $enforceDelaunay := 1$;
  end
while $|a.V| < N$ **do**
  begin
  $v :\in S \setminus a.V$; /* Wähle einen neuen Vertex v */
  $a := insertVertex(a, v)$; /* Füge v in a ein */
  if $enforceDelaunay = 1$ **then**
    $a := restoreDelaunayProperty(a)$;
  end
return (a, B);

Algorithmus 3.2: Erzeugung einer Startkonfiguration.

Im univariaten Fall ist eine vermeintlich bessere Methode zur Erzeugung einer initialen Konfiguration denkbar: Wir könnten die neuen Vertizes aus S so auswählen, daß die Kontrollpunkte der vollständigen initialen Konfiguration von a annähernd uniform über die konvexe Hülle von S verteilt sind. Experimente haben jedoch ergeben, daß diese nahezu uniformen Kontrollpunktverteilungen, obwohl sie in der Regel besser als die rein zufälligen sind, nachteilige Auswirkungen auf den Verlauf der Gesamtiteration haben. Es ist

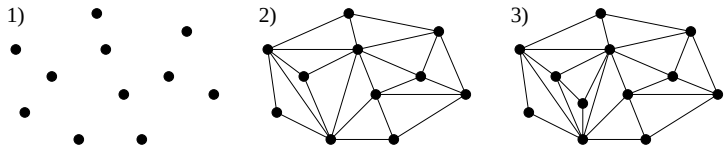

Abbildung 3.2: Erzeugung einer initialen Konfiguration. 1) Bestimmung der Vertizes einer Unterapproximation a_{sub}; 2) Kopieren des Simplexnetzes von a_{sub}; 3) Einfügen eines weiteren Vertex ohne Erzwingung der Delaunay-Bedingung.

somit günstiger, mit einer auch im univariaten Fall rein zufälligen initialen Konfiguration zu starten.

Eine weitere Variation des Verfahrens versucht, eine „optimale“ initiale Konfiguration zu erzeugen. Hierbei wird als nächster einzufügender Vertex stets derjenige Vertex der Streudatenmenge ausgewählt, der den größten Abstand von der momentanen Konfiguration von a hat. Der Abstand eines Vertex wird hierbei noch mit der Wurzel des Volumens des Simplizes gewichtet, der die Stelle des Vertex enthält. Experimente haben gezeigt, daß auf diese Art sehr gute initiale Konfigurationen erzeugt werden können. Leider sind auf diese Art erzeugte Konfigurationen aber *zu gut*, da der Iterationsprozeß nun nahe eines lokalen Minimums startet; in der Regel wird eine auf diese Art gestartete Iteration bereits nach kurzer Zeit von einer rein zufällig gestarteten Iteration „überholt“. In der Praxis ist eine solche optimale Bestimmung der initialen Konfiguration also ihren Aufwand nicht wert.

3.3 Bestimmung der Temperaturfunktion

Zwei Schritte sind zur Bestimmung der Temperaturfunktion notwendig:

- Man muß die initiale Temperatur bestimmen.
- Man muß entscheiden, wie (und wie schnell) die Temperatur im Verlauf der Iteration gesenkt werden soll.

Eine vernünftige Strategie zur Bestimmung der initialen Temperatur ist die Schätzung des Erwartungswertes der Fehlererhöhung bei Ausführung eines Iterationsschrittes aus dem Startzustand der Approximation. Ist dieser Erwartungswert gleich ΔE_0, und ist $p_0 \in (0,1)$ die vom Benutzer angegebene

Wahrscheinlichkeit, mit der ein „erwartet schlechter“ Schritt zu Beginn der Iteration akzeptiert werden soll, dann berechnet sich die initiale Temperatur nach der Boltzmann-Formel $p = e^{-\Delta E/kT}$ zu $kT := -\frac{\Delta E_0}{\log p_0}$.

Zur Bestimmung des Erwartungswertes ΔE_0 wenden wir eine gewisse Anzahl von Schritten des Iterationsalgorithmus jeweils auf die initiale Konfiguration an (d.h., wir machen die Ergebnisse dieser Schritte jeweils wieder rückgängig) und bestimmen das arithmetische Mittel aller Änderungen ΔE dieser Schritte. Algorithmus 3.3 gibt dieses Verfahren an:

Eingaben:

- Faktoren $f_g, f_l \in \mathbf{R}^+$ und eine Wahrscheinlichkeit $p_r \in [0,1]$, die Parameter für den Algorithmus zur Änderung einer aktuellen Konfiguration sind (siehe Abschnitt 3.4).
- Eine Streudatenmenge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$.
- Ein initialer linearer Spline $a \in LS_{n,m}(N)$, der S approximiert.
- Eine Menge $B \subset S$ von während der Iteration fix bleibenden Kontrollpunkten.
- Die Wahrscheinlichkeit $p_0 \in (0,1)$, mit der ein erwartet schlechter Schritt zu Beginn der Iteration akzeptiert werden soll.

Lokale:

- Der Erwartungswert $\Delta E_0 \in \mathbf{R}$ der Änderung des Abstandmaßes.
- Eine Zahl $i \in \mathbf{N}$.
- Der ursprüngliche Abstand $E \in \mathbf{R}^+$ des linearen Splines a von der Streudatenmenge S.
- Die Änderung $\Delta E \in \mathbf{R}$ des Abstands von a und S im letzten Schritt.
- Funktionen *calcDistance*, *changeConfiguration* und *undoLastChange* wie definiert in Abschnitt 3.1.

Ausgaben:

- Die initiale Temperatur kT.

$\Delta E_0 := 0$; /* Erwartete Änderung des Abstandmaßes */
for $i := 1$ **to** 100 **do**
  begin
  $E := calcDistance(S, a)$; /* Ursprünglicher Abstandswert */
  $a := changeConfiguration(f_g, f_l, p_r, S, a, B)$; /* Iterationsschritt */
  $\Delta E := calcDistance(S, a) - E$; /* Änderung dieses Schrittes */
  $a := undoLastChange(a)$; /* Mache Schritt rückgängig */
  $\Delta E_0 := \Delta E_0 + \Delta E/100$;
  end
return $-\Delta E_0 / \log p_0$;

Algorithmus 3.3: Bestimmung der initialen Temperatur.

Die Anwendung von 100 Schritten des Iterationsalgorithmus zur Schätzung des Erwartungswertes ΔE_0 hat sich als ausreichend erwiesen.

Nachdem die initiale Temperatur wie oben gezeigt bestimmt ist, muß nun entschieden werden, wie die Temperatur im Verlauf der Iteration gesenkt werden soll. Wir haben entschieden, die Temperatur für eine feste Zahl von Schritten konstant zu lassen und danach mit einem festen Faktor $f_t \in (0, 1)$ zu multiplizieren. Die Zahl der Schritte definieren wir als $temperatureSteps := 5\big(N - |B|\big)$, fünf mal die Anzahl der beweglichen Kontrollpunkte des linearen Splines a; der Faktor f_t wird als Eingabe an den Optimierungsalgorithmus übergeben. Experimente haben gezeigt, daß bei Verwendung dieser Heuristiken vergleichbare Iterationsverläufe bei konstanten p_0 und f_t und variierenden N entstehen.

3.4 Veränderung der aktuellen Konfiguration

Der Kern des Simulated-Annealing-Verfahrens ist der in jedem Iterationsschritt angewandte Algorithmus zur Änderung der aktuellen Konfiguration. Prinzipiell kann hierzu jeder Algorithmus verwendet werden, sofern er nur nichtdeterministische Elemente enthält; wir verwenden jedoch einen dreischrittigen Algorithmus, der sich in allen Experimenten als äußerst erfolgreich erwiesen hat. Die drei Schritte dieses Algorithmus zur Änderung der Konfiguration eines linearen Splines a sind: Rotation einer Kante des Simplexnetzes von a; „globales“ Bewegen eines Kontrollpunktes von a; „lokales“ Bewegen eines Kontrollpunktes von a.

Zuerst wird anhand einer konstanten Wahrscheinlichkeit $p_r \in [0,1]$ bestimmt, ob im aktuellen Schritt eine Kante rotiert werden soll. Durch die Wahl von p_r kann das Verhalten des Algorithmus im bivariaten Fall wie in Abschnitt 1.3 diskutiert beeinflußt werden: Für $p_r = 0$ werden nie Kanten rotiert, und der Algorithmus erzwingt eine Delaunay-Triangulierung während der gesamten Iteration; im anderen Extremfall $p_r = 1$ werden nie Vertizes bewegt, und der Algorithmus „degeneriert“ zu einem Algorithmus zur datenabhängigen Triangulierung (siehe Abschnitt 2.3.2). Für alle anderen Werte von p_r werden sowohl Kanten rotiert als auch Vertizes bewegt, und der Algorithmus bestimmt simultan eine optimale Vertexplazierung und ein optimales Simplexnetz. Im univariaten Fall ist p_r stets gleich null, da hier das Simplexnetz bereits durch die Vertexplazierung eindeutig festgelegt ist.

Soll im aktuellen Schritt eine Kante rotiert werden, dann bestimmt der Algorithmus zufällig eine Kante des aktuellen Simplexnetzes, die die Diagonale eines konvexen Vierecks ist. Dies stellt sicher, daß die Kante rotiert werden kann, ohne daß das Simplexnetz degeneriert. Anschließend wird diese Kante rotiert und die neue Konfiguration des linearen Splines zurückgegeben.

Soll im aktuellen Schritt ein Vertex bewegt werden, dann wird zunächst ein beweglicher Vertex v des Splines a zufällig bestimmt und das Volumen des Platelets von v berechnet. Ist dieses Volumen kleiner als $f_g \cdot E$, der momentane Abstand von a und S, dann entscheiden wir, daß v in einer „flachen“ Region der Streudatenmenge S liegt und damit an der aktuellen Stelle überflüssig ist. Wir bewegen v dann „global“ an eine zufällig bestimmte andere Stelle innerhalb der konvexen Hülle von S (siehe Abschnitt 3.4.2). Ist das Volumen des Platelets von v größer als oder gleich $f_g \cdot E$, dann entscheiden wir, daß v in einer „wichtigen“ Region, also einer Region hoher Krümmung, liegt; wir versuchen dann die Position von v weiter zu verbessern, indem wir v um eine kleine Distanz, also „lokal“, bewegen (siehe Abschnitt 3.4.3).

Eine Gesamtübersicht des Algorithmus zur Änderung der aktuellen Konfiguration ist in Algorithmus 3.4 angegeben; die folgenden Abschnitte behandeln die einzelnen Schritte im Detail:

Eingaben:

- Ein Faktor $f_g \in \mathbf{R}^+$, der die Entscheidung zwischen globalen und lokalen Vertexbewegungen beeinflußt.
- Ein Faktor $f_l \in \mathbf{R}^+$, der die maximale Schrittweite lokaler Vertexbewegungen beeinflußt (siehe Abschnitt 3.4.3).

- Die Wahrscheinlichkeit $p_r \in [0,1]$, mit der im aktuellen Iterationsschritt eine Kante rotiert wird.
- Eine Streudatenmenge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$.
- Ein linearer Spline $a = (a.V, a.SM) \in LS_{n,m}(N)$, der S approximiert.
- Die Menge $B \subset a.V$ der fixen Kontrollpunkte von a.

Lokale:

- Eine Kante $k \in CE_n$.
- Eine Funktion *getQuadDiagonal*: $LS_{n,m} \to CE_n$, $a \mapsto k$, die eine Diagonale eines konvexen Vierecks aus dem Simplexnetz von a liefert.
- Eine Funktion *rotateEdge*: $LS_{n,m}(N) \times CE_n \to LS_{n,m}(N) \times CE_n, (a,k) \mapsto (a',k')$, die eine Kante k aus dem Simplexnetz eines linearen Splines a rotiert, falls diese Kante die Diagonale eines konvexen Vierecks ist. Die Funktion gibt den geänderten Spline a' und die rotierte Kante k' zurück.
- Ein Vertex $v \in (\mathbf{R}^n \times \mathbf{R}^m)$.
- Der momentane Abstand $E \in \mathbf{R}^+$ des linearen Splines a von der Streudatenmenge S.
- Das Volumen $vP \in \mathbf{R}^+$ des Platelets von v in a (siehe Abschnitt 3.4.1).
- Eine Funktion *calcPlateletVolume*: $LS_{n,m} \times (\mathbf{R}^n \times \mathbf{R}^m) \to \mathbf{R}^+, (a,v) \mapsto vP$, die das Volumen vP des Platelets eines Vertizes v in einem linearen Spline a berechnet (siehe Abschnitt 3.4.1).
- Eine Funktion *moveVertexGlobally*: $P(\mathbf{R}^n \times \mathbf{R}^m) \times LS_{n,m}(N) \times (\mathbf{R}^n \times \mathbf{R}^m) \to LS_{n,m}(N)$, $(a,v) \mapsto a'$, die einen Vertex v eines linearen Splines a an eine zufällig bestimmte neue Stelle aus S bewegt (siehe Abschnitt 3.4.2).
- Eine Funktion *moveVertexLocally*: $P(\mathbf{R}^n \times \mathbf{R}^m) \times LS_{n,m}(N) \times (\mathbf{R}^n \times \mathbf{R}^m) \times \mathbf{R}^+ \to LS_{n,m}(N)$, $(S,a,v,f_l) \mapsto a'$, die einen Vertex v eines linearen Splines a an eine zufällig bestimmte neue Stelle aus S nahe seiner aktuellen Stelle bewegt (siehe Abschnitt 3.4.3).
- Eine Funktion *restoreDelaunayProperty*: $LS_{n,m}(N) \to LS_{n,m}(N)$, $a \mapsto a'$, die die Delaunay-Bedingung des Simplexnetzes von a wiederherstellt.

Ausgaben:

- Der lineare Spline $a \in LS_{n,m}(N)$ nach der Änderung seiner Konfiguration. Informationen zur Zurücknahme der Änderung sind im Undo-Puffer von a abgelegt.

```
if prob(p_r) then
    begin
    /* Rotiere eine Kante: */
    k := getQuadDiagonal(a);   /* Wähle zufällige Kante */
    (a, k) := rotateEdge(a, k);   /* Rotiere die Kante */
    end
else
    begin
    /* Bewege einen Vertex: */
    v :∈ a.V \ B;   /* Wähle einen beweglichen Vertex v */
    E := calcDistance(S, a);
    vP := calcPlateletVolume(a, v);
    if vP < f_g · E then   /* Ist das Platelet flach? */
        a := moveVertexGlobally(S, a, v);
    else
        a := moveVertexLocally(S, a, v, f_l);
    if p_r = 0 then   /* Delaunay-Bedingung stets aufrechterhalten? */
        a := restoreDelaunayProperty(a);
    end
```

Algorithmus 3.4: Veränderung der aktuellen Konfiguration.

3.4.1 Abschätzung des „Volumens“ eines Platelets

Um zu entscheiden, ob ein Vertex v momentan in einer unwichtigen, d.h. flachen, oder in einer wichtigen, d.h. stark gekrümmten, Region der Streudatenmenge S liegt, schätzen wir das „Volumen“ des Platelets von v. Hierzu bestimmen wir eine Minimale-Quadrate-Hyperebene H, die alle v umgebenden Vertizes approximiert, und den Punkt p, der an der gleichen Stelle wie v in der Hyperebene H liegt. Mit anderen Worten, p ist die Projektion von v auf H in Ordinatenrichtung. Wir definieren dann weiterhin h als den Abstand zwischen v und p, und A als die Fläche des Platelets von v. Abbildung 3.3 illustriert diese Definitionen für den bivariaten Fall.

Eine mögliche Definition des „Volumens“ des Platelets von v wäre dann das Volumen der „Hyperpyramide“ mit Grundfläche A und Höhe h. Dieses Volumen wäre gegeben durch $A \cdot h/2$ im univariaten Fall und durch $A \cdot h/3$ im bivariaten Fall. Um das Volumen jedoch mit der von uns verwendeten L^2-

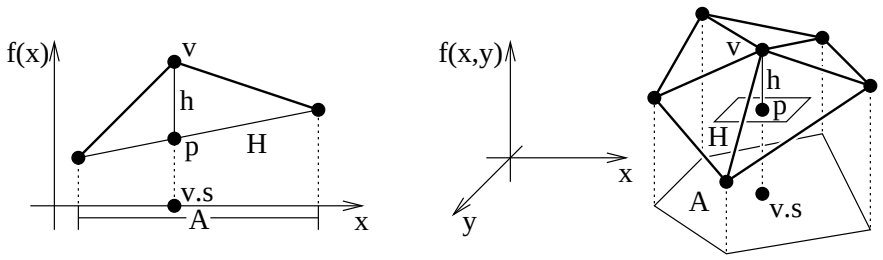

Abbildung 3.3: Abschätzung des Volumens des Platelets eines Vertizes v.

Metrik vergleichbar zu machen, definieren wir das Volumen vP im univariaten Fall als $vP := \sqrt{A \cdot h^2/2}$ und im bivariaten Fall als $vP := \sqrt{A \cdot h^2/3}$.

3.4.2 Globale Bewegung eines Vertizes

Ist das Volumen vP des Platelets eines Vertizes v kleiner als $f_g \cdot E$, der momentane Abstand von a und S, dann entscheiden wir, daß v in einer „flachen" Region der Streudatenmenge S liegt und damit an der aktuellen Stelle überflüssig ist. Wir bewegen v dann „global" an eine zufällig bestimmte andere Stelle innerhalb der konvexen Hülle von S. Diese Methode der Änderung stellt sicher, daß im Verlauf der Iteration nur so wenig Vertizes wie möglich an flachen Stellen von S verbleiben.

Um die Konfiguration eines linearen Splines a durch eine globale Bewegung des Vertizes v zu ändern, gehen wir wie folgt vor:

1. Wir entfernen den Vertex v an seiner momentanen Stelle aus der Konfiguration von a. Genauer gesagt entfernen wir v aus der Vertexplazierung von a, und wir entfernen alle Simplizes aus der Verbindungsmenge von a, die v als Eckpunkt haben (wir entfernen also genau das Platelet von v).

2. Wir füllen das entstandene „Loch" im Simplexnetz durch Einfügen neuer Simplizes. Im univariaten Fall fügen wir das Intervall ein, das die Stellen der ehemaligen Nachbarn von v als Grenzen hat; im bivariaten Fall retriangulieren wir das ehemalige Platelet von v mit einer beliebigen Triangulierung.

3. Wir fügen den Vertex v an seiner neuen Stelle in die Konfiguration von a ein, indem wir denjenigen Simplex c der Verbindungsmenge spal-

ten, der die neue Stelle enthält. Wir fügen v in die Vertexplazierung ein und ersetzen c durch die neuen Simplizes, die v mit seinen neuen direkten Nachbarn verbinden. Im univariaten Fall ersetzen wir das Intervall $[v_1.s, v_2.s]$ durch die Intervalle $[v_1.s, v.s]$ und $[v.s, v_2.s]$, wobei v_1 und v_2 die neuen linken bzw. rechten Nachbarn von v sind. Im bivariaten Fall ersetzen wir das Dreieck $(v_1.s, v_2.s, v_3.s)$ durch die Dreiecke $(v_1.s, v_2.s, v_s)$, $(v_1.s, v_3.s, v_s)$ und $(v_2.s, v_3.s, v_s)$, wobei v_1, v_2 und v_3 die neuen direkten Nachbarn von v sind.

4. Ist die Wahrscheinlichkeit p_r, mit der im Änderungsalgorithmus Kanten rotiert werden, gleich null, dann stellen wir nach der globalen Bewegung die Delaunay-Bedingung des Simplexnetzes von a durch Kantenrotationen wieder her (siehe [17]).

Abbildung 3.4 illustriert diese Schritte für den bivariaten Fall.

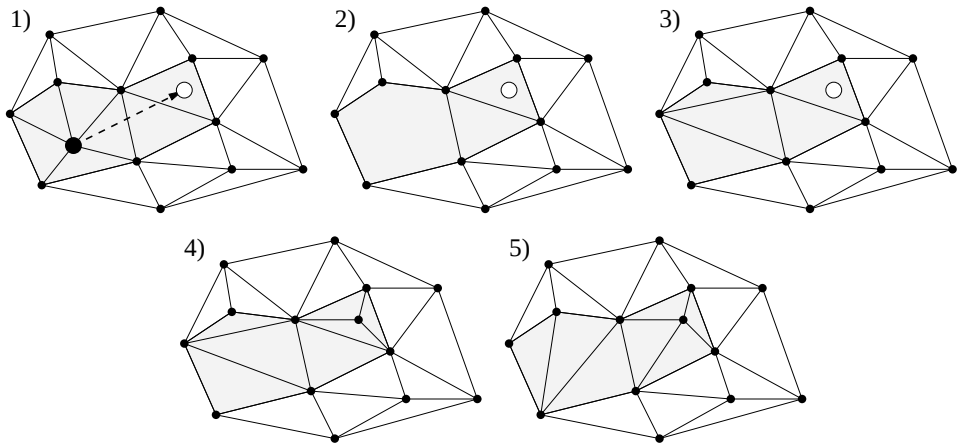

Abbildung 3.4: Globale Vertexbewegung im bivariaten Fall. 1) Initialzustand; 2) Entfernen des Vertex; 3) Füllen des Loches; 4) Einfügen des neuen Vertizes; 5) Wiederherstellen der Delaunay-Bedingung (optional).

Es gibt einen Sonderfall bei der globalen Vertexbewegung im bivariaten Fall zu beachten: Eine neue Stelle, die direkt auf einer im Simplexnetz vorkommenden Kante k liegt, muß gesondert behandelt werden, da das diese Stelle enthaltene Dreieck nicht auf die übliche Weise gespalten werden kann. Ist die Kante k Teil des Randes der konvexen Hülle der Streudatenmenge S, dann muß das die neue Stelle enthaltene Dreieck in zwei Teile gespalten

werden; ist k andererseits eine *innere* Kante, d.h. eine Kante, die von zwei Dreiecken geteilt wird, dann müssen beide k teilenden Dreiecke in jeweils zwei Teile gespalten werden. Wenn andererseits p_r gleich null ist, wenn also während der gesamten Iteration Delaunay-Triangulierungen benutzt werden, kann man letzteren Fall getrost ignorieren: Man kann eines der die neue Stelle enthaltenen Dreiecke wie üblich spalten (und somit ein degeneriertes Dreieck erzeugen), da der anschließende Kantenrotationsschritt die Triangulierung wieder rektifiziert.

Unser Algorithmus benutzt momentan einen sehr einfachen Mechanismus, um mit diesem Problem fertigzuwerden: Ist p_r gleich null, verbieten wir globale Bewegungen zu einer auf dem Rand der konvexen Hülle liegenden Stelle; ist p_r größer als null, verbieten wir zusätzlich globale Bewegungen zu einer auf einer existierenden Kante liegenden Stelle.

Algorithmus 3.5 beschreibt das angegebene Verfahren zur globalen Bewegung eines Vertizes im Detail:

Eingaben:

- Eine Streudatenmenge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$.
- Ein linearer Spline $a = (a.V, a.SM) \in LS_{n,m}(N)$, der S approximiert.
- Ein Vertex $v \in a.V$, der global bewegt werden soll.

Lokale:

- Eine Funktion $removeVertex\colon LS_{n,m}(k) \times (\mathbf{R}^n \times \mathbf{R}^m) \to LS_{n,m}(k-1)$, $(a, v) \mapsto a'$, die einen Vertex v aus einem linearen Spline a entfernt und das entstehende Loch im Simplexnetz wieder füllt.
- Ein neuer Vertex $vNew \in S$.
- Eine Funktion $isInsertionSafe\colon LS_{n,m} \times \mathbf{R}^n \to \{0,1\}$, $(a, s) \mapsto isSafe$, die feststellt, ob ein Vertex an der Stelle s in den linearen Spline a eingefügt werden kann.
- Eine Funktion $insertVertex\colon LS_{n,m}(k) \times (\mathbf{R}^n \times \mathbf{R}^m) \to LS_{n,m}(k+1)$, $(a, v) \mapsto a'$, die einen Vertex v in einen linearen Spline a einfügt.

Ausgaben:

- Der lineare Spline $a \in LS_{n,m}(N)$ nach der Änderung seiner Konfiguration. Informationen zur Zurücknahme der Änderung sind im Undo-Puffer von a abgelegt.

$a := removeVertex(a, v)$; /* Entferne v aus a */
do
  $vNew :\in S$; /* Wähle einen neuen Vertex aus S */
while not $isInsertionSafe(a, vNew.s)$;
$a := insertVertex(a, vNew)$; /* Füge $vNew$ in a ein */
return a;

Algorithmus 3.5: Globale Bewegung eines Vertizes.

3.4.3 Lokale Bewegung eines Vertizes

Ist das Volumen vP des Platelets eines Vertizes v größer als oder gleich $f_g \cdot E$, der momentane Abstand von a und S, dann entscheiden wir, daß v in einer „wichtigen" Region, d.h. einer Region hoher Krümmung, der Streudatenmenge S liegt; wir versuchen dann die Position von v weiter zu verbessern, indem wir v um eine kleine Distanz, also „lokal", bewegen.

Die maximale Schrittweite lokaler Bewegung ist durch zwei Faktoren begrenzt: Zum einen bewegen wir einen Vertex v stets nur innerhalb eines *Grenzpolygons*. Dieses Polygon wird durch die Vereinigung des Platelets von v und aller Dreiecke gebildet, die eine Kante mit dem Platelet gemeinsam haben[2]. Hierdurch stellen wir sicher, daß ein Vertex v im Zuge seiner lokalen Bewegung höchstens eine Kante im aktuellen Simplexnetz des linearen Splines a überschreitet. Zum anderen begrenzen wir die maximale Schrittweite durch einen konstanten Faktor f_l.

Da die neue Position eines Vertex bei lokaler Bewegung durch zufällige Auswahl einer Stelle der Streudatenmenge S geschieht, die innerhalb des Grenzpolygons liegt, wäre es schwierig, von vornherein alle Stellen auszuschließen, die weiter als f_l entfernt liegen. Da wir darüberhinaus ein „weiches" Abschneiden der Bewegung wünschen, gehen wir wie folgt vor: Wir bestimmen eine zufällige Stelle $s \in S$ innerhalb des Grenzpolygons von v. Sei $d := \|v.s - s\|$ der Abstand der Stellen $v.s$ und s, dann akzeptieren wir die neue Stelle s mit der Wahrscheinlichkeit $p := e^{-(d/f_l)^2}$. Hiermit erreichen wir eine Verteilung der akzeptierten Schrittweiten mit einer glockenförmigen Kurve, wobei lokale Bewegungen über sehr kurze Distanzen mit einer Wahr-

[2]Wir erlauben einem Vertex das Verlassen seines Platelets im Zuge lokaler Bewegung, da Experimente gezeigt haben, daß anderenfalls anfänglich schlechte Konfigurationen nur schwierig überwunden werden können.

scheinlichkeit von fast eins akzeptiert werden. Wir benutzen keine Normalverteilung der Distanzen, da in diesem Falle auch kurze Distanzen mit großer Wahrscheinlichkeit abgelehnt würden. Wird eine Stelle s abgelehnt, dann wählen wir solange neue Stellen innerhalb des Grenzpolygons, bis eine akzeptiert wird oder eine vorher festgelegte Zahl von Versuchen fehlgeschlagen ist. Im letzteren Fall geben wir als Ergebnis der Änderung den unveränderten linearen Spline a zurück.

Um einen Vertex v lokal zu bewegen, könnte man denselben Algorithmus wie in Abschnitt 3.4.2 verwenden. Dies wäre jedoch nicht geschickt, da die Anwendung jenes Verfahrens das Simplexnetz um v auch bei sehr kleinen Bewegungen drastisch verändern könnte, da das Simplexnetz im Platelet des bewegten Vertex bei jeder Bewegung neu erzeugt wird (siehe Abbildung 3.5).

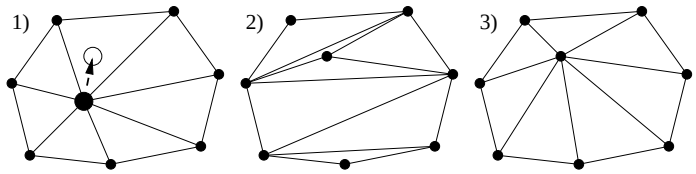

Abbildung 3.5: Nachteil bei der globalen Bewegung eines Vertizes über eine kurze Distanz im bivariaten Fall: 1) Initialzustand; 2) Ergebnis nach globaler Bewegung ohne Wiederherstellung der Delaunay-Bedingung; 3) gewünschtes Ergebnis.

Wir gehen davon aus, daß die lokale Konfiguration um einen lokal zu bewegenden Vertex bereits gut ist; wir verlangen daher, daß diese lokale Konfiguration durch eine Bewegung über eine kurze Distanz so wenig wie möglich geändert wird (siehe Abbildung 3.5, Teile 1 und 3). Um dies zu erreichen, verwenden wir eine andere Methode, um einen Vertex lokal zu bewegen. Wir lassen einen Vertex auf der Strecke zwischen seiner alten und seiner neuen Stelle „rutschen“, wobei der Vertex sozusagen die Kanten, die ihn mit allen benachbarten Vertizes verbinden, hinter sich schleppt. Wann immer einer der den Vertex umgebenden Simplizes zu degenerieren droht, rotieren wir eine Kante des betroffenen Simplizes, bevor wir die Bewegung des Vertizes fortsetzen. Hierzu bestimmen wir zunächst die Menge K aller Kanten, die das Platelet des Vertizes v an seiner ursprünglichen Position begrenzen. Während der Bewegung bestimmen wir dann diejenige Kante $k \in K$, die als nächstes von v überschritten würde. Im bivariaten Fall sind hierbei drei Fälle zu unterscheiden:

1. Der Vertex v überschreitet die Kante k selbst (siehe Abbildung 3.6). In diesem Fall müssen wir k rotieren, bevor wir die Bewegung von v fortsetzen. Durch die Rotation von k wird das Platelet von v um den zuvor jenseits von k gelegenen Simplex c ergänzt; daher müssen wir k aus der Menge K entfernen und die anderen Kanten von c in K einfügen.

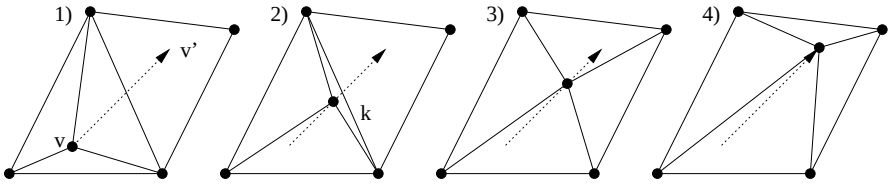

Abbildung 3.6: Überschreiten einer Kante, Fall 1. 1) Initialzustand; 2) vor weiterer Bewegung von v muß die Kante k rotiert werden; 3) Zustand direkt nach Rotation von k; 4) Endzustand.

2. Der Vertex v überschreitet die von v aus gesehen linke Verlängerung der Kante k (siehe Abbildung 3.7). In diesem Fall müssen wir diejenige Kante k_2 rotieren, die v mit dem rechten Endpunkt von k verbindet. Durch die Rotation von k_2 wird der nun jenseits von k_2 gelegene Simplex c vom Platelet von v abgespalten; daher müssen wir k_2 in die Menge K einfügen und die anderen Kanten von c aus K entfernen.

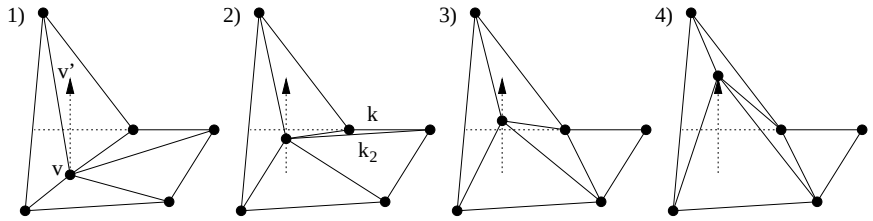

Abbildung 3.7: Überschreiten einer Kante, Fall 2. 1) Initialzustand; 2) vor weiterer Bewegung von v muß die Kante k_2 rotiert werden; 3) Zustand direkt nach Rotation von k_2; 4) Endzustand.

3. Der Vertex v überschreitet die von v aus gesehen rechte Verlängerung der Kante k (siehe Abbildung 3.8). In diesem Fall müssen wir diejenige Kante k_2 rotieren, die v mit dem linken Endpunkt von k verbindet.

Durch die Rotation von k_2 wird der nun jenseits von k_2 gelegene Simplex c vom Platelet von v abgespalten; daher müssen wir k_2 in die Menge K einfügen und die anderen Kanten von c aus K entfernen.

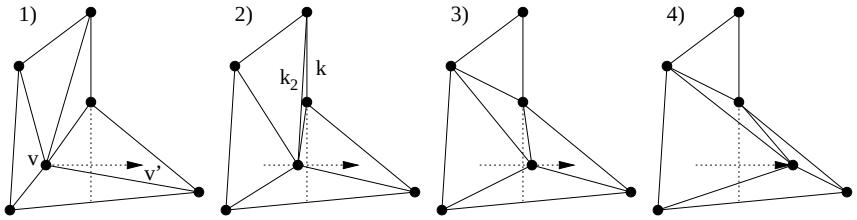

Abbildung 3.8: Überschreiten einer Kante, Fall 3. 1) Initialzustand; 2) vor weiterer Bewegung von v muß die Kante k_2 rotiert werden; 3) Zustand direkt nach Rotation von k_2; 4) Endzustand.

Diese Schritte werden solange wiederholt, wie Kanten aus K von v überschritten würden.

Dieses Verfahren der Vertexbewegung funktioniert für Bewegungen über beliebige Distanzen, sogar weit aus dem ursprünglichen Platelet des Vertizes heraus. Da der Algorithmus aber das gesamte Simplexnetz zwischen der Anfangsstelle und der Endstelle der Bewegung ändert, und da das endgültige Platelet des Vertizes beide Stellen enthält und damit bei langen Bewegungen sehr langgestreckt wird, verwenden wir für lange Bewegungen weiterhin den Algorithmus aus Abschnitt 3.4.2. Algorithmus 3.6 beschreibt das Verfahren zur lokalen Bewegung eines Vertex im Detail:

Eingaben:

- Eine Streudatenmenge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$.
- Ein linearer Spline $a = (a.V, a.SM) \in LS_{n,m}(N)$, der S approximiert.
- Ein Vertex $v \in a.V$, der lokal bewegt werden soll.
- Ein Faktor $f_l \in \mathbf{R}^+$, der die maximale Länge der Bewegung beschränkt.

Lokale:

- Ein Flag $siteAccepted \in \{0, 1\}$, das angibt, ob eine neue Stelle für den Vertex v gefunden wurde.
- Eine neue Position $vNew \in (\mathbf{R}^n \times \mathbf{R}^n)$ für den Vertex v.
- Eine Funktion $findLocalVertex$: $P(\mathbf{R}^n \times \mathbf{R}^m) \times LS_{n,m} \times (\mathbf{R}^n \times \mathbf{R}^m) \to (\mathbf{R}^n \times \mathbf{R}^m)$, $(S, a, v) \mapsto vNew$, die einen zufälligen Vertex $vNew$ aus S im Grenzpolygon von v in a liefert.

- Der Abstand $d \in \mathbf{R}^+$ der bisherigen Stelle von v von einer potentiellen neuen Stelle.
- Eine Funktion $isInsertionSafe: LS_{n,m} \times \mathbf{R}^n \to \{0,1\}$, $(a,s) \mapsto isSafe$, die feststellt, ob ein Vertex an der Stelle s in den linearen Spline a eingefügt werden kann.
- Die Anfangsstelle $s_0 \in \mathbf{R}^n$ der Vertexbewegung.
- Der Abstandsvektor $\Delta s \in \mathbf{R}^n$ der neuen und alten Stelle von v.
- Die aktuelle Position $\lambda \in [0,1]$ des Vertex v entlang der Bewegungsrichtung.
- Eine Menge $K \subset CE_n$ von Kanten aus dem Simplexnetz von a.
- Eine Funktion $getSurroundingEdges: LS_{n,m} \times (\mathbf{R}^n \times \mathbf{R}^m) \to P(CE_n)$, $(a,v) \mapsto K$, die die Menge K der das Platelet von v in a begrenzenden Kanten liefert.
- Der Koeffizient $\lambda_{\min} \in [0,1]$ des Punktes der Bewegung, an dem die Gerade der nächsten Kante aus K überschritten wird.
- Die als nächstes überschrittene Kante $k \in CE_n$.
- Der Koeffizient $\mu \in \mathbf{R}$ des Punktes, an dem die Gerade der Kante k überschritten wird. Hierbei ist $\mu < 0$, wenn der Punkt rechts vom rechten Endpunkt der Kante liegt (von v aus gesehen), $\mu > 1$, wenn der Punkt links vom linken Endpunkt der Kante liegt, und $\mu \in [0,1]$, wenn der Punkt auf der Kante liegt.
- Eine Funktion $getNextIntersection: P(CE_n) \times \mathbf{R}^n \times \mathbf{R}^n \to \big([0,1] \cup \{\infty\}\big) \times CE_n \times \mathbf{R}$, $(K, s_0, \Delta s) \mapsto (\lambda_{\min}, k, \mu)$, die die nächste Kante k aus K bestimmt, deren Gerade im Verlauf der Bewegung von s_0 nach $s_0 + \Delta s$ überschritten wird. Wird eine Kante überschritten, werden der Koeffizient $\lambda_{\min}$ des Schnittpunkts bezüglich der Bewegung, die Kante k und der Koeffizient μ des Schnittpunktes bezüglich der Kante zurückgegeben, anderenfalls ist $\lambda_{\min}$ gleich ∞ und k und μ sind undefiniert.
- Eine Funktion $moveVertex: LS_{n,m}(N) \times (\mathbf{R}^n \times \mathbf{R}^m) \times (\mathbf{R}^n \times \mathbf{R}^m) \to LS_{n,m}(N)$, $(a, v, vNew) \mapsto a'$, die einen Vertex v in einem linearen Spline a auf die neue Position $vNew$ setzt. Hierbei wird nur die Position des Vertizes gesetzt, die Nachbarschaftsbeziehungen zu anderen Vertizes und das Simplexnetz werden nicht geändert.
- Funktionen $getLeftEdge, getRightEdge: LS_{n,m} \times (\mathbf{R}^n \times \mathbf{R}^m) \times CE_n \to CE_n$, $(a, v, k) \mapsto k_2$, die diejenige Kante k_2 aus dem Simplexnetz eines linearen Splines a zurückgeben, die den Vertex v und den linken bzw. rechten Endpunkt der Kante k verbinden.

- Ein Simplex $c \in CS_n$.
- Eine Funktion *getTransSimplex*: $LS_{n,m} \times (\mathbf{R}^n \times \mathbf{R}^m) \times CE_n \rightarrow CS_n$, $(a, v, k) \mapsto c$, die den von v aus gesehen jenseits einer Kante k liegenden Simplex c in einem linearen Spline a liefert.
- Eine Funktion *getSimplexEdges*: $LS_{n,m} \times CS_n \rightarrow P(CE_n)$, $(a, c) \mapsto K$, die die Menge K der einen Simplex c begrenzenden Kanten in einem linearen Spline a liefert.
- Eine Funktion *rotateEdge*: $LS_{n,m}(N) \times CE_n \rightarrow LS_{n,m}(N) \times CE_n, (a, k) \mapsto (a', k')$, die eine Kante k aus dem Simplexnetz eines linearen Splines a rotiert, falls diese Kante die Diagonale eines konvexen Vierecks ist. Die Funktion gibt den geänderten Spline a' und die rotierte Kante k' zurück.

Ausgaben:

- Der lineare Spline $a \in LS_{n,m}(N)$ nach der Änderung seiner Konfiguration. Informationen zur Zurücknahme der Änderung sind im Undo-Puffer von a abgelegt.

```
/* Bestimme einen nahegelegenen neuen Vertex im Grenzpolygon von v: */
vNewAccepted := 0;
do
    begin
    vNew := findLocalVertex(S, a, v);
    d := ‖v.s − vNew.s‖;
    if prob(e^{−(d/f_l)^2}) then
        vNewAccepted := isInsertionSafe(a, New.s);
    end
while vNewAccepted = 0;
/* Bewege v zur neuen Position vNew: */
s_0 := v.s;
Δs := vNew.s − s_0;
λ := 0;
K := getSurroundingEdges(a, v);
while λ < 1 do
    begin
    (λ_min, k, μ) := getNextIntersection(K, s_0, Δs);
    if λ_min < ∞ then   /* Es wird eine Kante überschritten */
        begin
        if μ > 1 then   /* v läuft links vorbei */
            begin
```

```
            /* Rotiere die rechts liegende Kante: */
            k_2 := getRightEdge(a, v, k);
            (a, k_2) := rotateEdge(a, k_2);
            /* Aktualisiere die Kantenliste: */
            c := getTransSimplex(a, v, k_2);
            K := K \ getSimplexEdges(a, c);
            K := K ∪ {k_2};
            end
        else if μ < 0 then   /* v läuft rechts vorbei */
            begin
            /* Rotiere die links liegende Kante: */
            k_2 := getLeftEdge(a, v, k);
            (a, k_2) := rotateEdge(a, k_2);
            /* Aktualisiere die Kantenliste: */
            c := getTransSimplex(a, v, k_2);
            K := K \ getSimplexEdges(a, c);
            K := K ∪ {k_2};
            end
        else   /* v überschreitet k */
            begin
            /* Aktualisiere die Kantenliste: */
            c := getTransSimplex(a, v, k);
            K := K ∪ getSimplexEdges(a, c);
            K := K \ {k};
            /* Rotiere die überschrittene Kante: */
            (a, k) := rotateEdge(a, k);
            end
        end
    else   /* Keine weiteren Überschreitungen */
        λ := 1;
    end   /* while λ < 1 */
a := moveVertex(a, v, vNew);   /* Setze v auf die Endposition */
return a;
```

Algorithmus 3.6: Lokale Bewegung eines Vertizes.

3.5 Darstellung linearer Splines

Die Wahl der Datenstrukturen zur Darstellung von Streudatenmengen und diese approximierenden linearen Splines bestimmt wesentlich die Laufzeit eines einzelnen Iterationsschrittes und damit die Laufzeit des gesamten Optimierungsalgorithmus. Zur Beschleunigung des Verfahrens ist es daher nötig, die wichtigsten Grundoperationen des Iterationsschrittes herauszuarbeiten und die Datenstrukturen für diese Operationen zu optimieren. Die folgenden Abschnitte beschreiben die von uns verwendete „reiche“ Datenstruktur, mit der die meisten Grundoperationen auf linearen Splines in konstanter Zeit oder zumindest zum Verhältnis von Streudatenvertizes pro Simplizes proportionaler Zeit ausgeführt werden können.

3.5.1 Der Aufwand eines Iterationsschrittes

Die Laufzeit des Optimierungsalgorithmus wird vor allem durch zwei Faktoren bestimmt:

1. Nach jeder Änderung der Konfiguration muß der neue Abstand E der Approximation von der Streudatenmenge bestimmt werden. Wie in Abschnitt 2.2 gezeigt, muß hierzu für jeden Streudatenvertex der die Stelle dieses Vertizes enthaltende Simplex gefunden werden; danach muß der Abstand des Vertex vom Simplex bestimmt werden.

2. Die Mehrheit aller Änderungsschritte (in unseren Experimenten zwischen 60 und 80 Prozent der Schritte) im Iterationsverfahren sind lokale Vertexbewegungen. Wie in Abschnitt 3.4.3 beschrieben, muß hierzu für jede Bewegung mindestens ein neuer Streudatenvertex aus der nahen Umgebung (genauer: aus dem Grenzpolygon) des zu bewegenden Vertizes gefunden werden.

Da unser Verfahren in der Regel auf sehr umfangreiche Streudatenmengen angewendet wird, muß die Darstellung der linearen Splines für diese beiden Hauptoperationen optimiert werden. Genauere Betrachtung ergibt, daß die Kerne dieser Operationen daß Finden des eine Stelle enthaltenen Simplizes und das Aufzählen aller in einem Simplex enthaltenen Streudatenvertizes sind.

Wir entschieden deshalb, die Streudatenmenge und den sie approximierenden linearen Spline in einer gemeinsamen Datenstruktur abzulegen. Die Datenstruktur der Splines zerfällt hierbei natürlich in drei Teile:

1. Eine Menge von Vertizes, die die Vertexplazierung des Splines bilden.
2. Eine Menge von Simplizes, die das Simplexnetz des Splines bilden.
3. Eine Menge von Kanten, die die Vertizes verbinden und die Simplizes begrenzen. Im univariaten Fall fallen die Kanten mit den Vertizes zusammen und werden nicht getrennt gespeichert.

Ist diese Grundstruktur festgelegt, dann fällt es leicht, die Untermenge der in einem Simplex enthaltenen Streudatenvertizes mit diesem Simplex zu assoziieren und mit diesem zusammen zu speichern. Bei dieser Methode der Darstellung zerfällt die Streudatenmenge also in eine Anzahl einzelner, den Simplizes zugeordneter Mengen. Da das Simplexnetz eines approximierenden Splines die konvexe Hülle der Streudatenmenge abdeckt, kann jeder Streudatenvertex einem Simplex zugeordnet werden. Da jede Kante im Simplexnetz in beiden sie teilenden Simplizes enthalten ist, und da jeder Vertex von allen diesen teilenden Simplizes überdeckt wird, ist diese Zuordnung nicht eindeutig; da lineare Splines aber stets stetig sind und zur Abstandsberechnung nur der Ordinatenabstand eines Vertizes zum Spline bestimmt wird, kann ein Streudatenvertex, der direkt auf einer Kante oder auf einem der Vertizes der Vertexplazierung liegt, einem beliebigen der überdeckenden Simplizes zugeordnet werden, ohne das Abstandsmaß zu beeinflussen.

Die Berechnung des Abstands zwischen Streudatenmenge und linearem Spline ist nun einfach: Man iteriert über alle Simplizes c des Simplexnetzes und bestimmt den Abstand aller c zugeordneten Streudatenvertizes von c. Diese Fehlerberechnung kann durch ein einfaches Caching-Schema drastisch weiter beschleunigt werden: In jedem Änderungsschritt wird nur eine kleine Teilmenge der Vertexplazierung und des Simplexnetzes geändert (im Löwenanteil aller Fälle jeweils nur ein einzelnes Platelet). Wir speichern daher mit einem Simplex auch jeweils seinen aktuellen Anteil am Gesamtabstand ab, dieser braucht dann nur durch eine Neusummierung über alle Streudatenvertizes aktualisiert zu werden, wenn der betreffende Simplex durch die Operation geändert wird. Durch die Verwendung dieser Darstellung ist die erwartete Laufzeit einer einzelnen Abstandsberechnung nun proportional zu der Anzahl der Streudatenvertizes pro Simplex, da in einer Änderungsoperation im Mittel nur eine konstante Zahl von Simplizes betroffen ist. Da zur Abstandsberechnung zusätzlich die Summe der in allen Simplizes gecachten Abstandsteilsummen gebildet werden muß, ist der Gesamtaufwand proportional der Summe der mittleren Anzahl von Vertizes pro Simplex und der

Anzahl der Simplizes: $T(calcDistance) \in O\big(|S|/|a.V| + |a.V|\big)$. Der zweite Summand fällt in praktischen Anwendungen jedoch nicht ins Gewicht.

Auch die Bestimmung benachbarter Streudatenvertizes ist nun einfach: Da jeder Vertex v der Vertexplazierung Verweise auf die ihn teilenden Simplizes enthält, kann die Menge aller Streudatenvertizes im Platelet bzw. Grenzpolygon durch Vereinigung der Vertexmengen der im Platelet bzw. Grenzpolygon enthaltenen Simplizes gebildet werden. Algorithmus 3.7 gibt an, wie ein zufälliger nahegelegener Streudatenvertex gefunden wird:

Eingaben:

- Eine Streudatenmenge $S \subset (\mathbf{R}^n \times \mathbf{R}^m)$.
- Ein linearer Spline $a \in LS_{n,m}$.
- Ein Vertex $v \in (\mathbf{R}^n \times \mathbf{R}^m)$ aus der Vertexplazierung von a.

Lokale:

- Eine Menge $C \subset CS_n$ von Simplizes aus dem Simplexnetz von a.
- Eine Funktion $getPlatelet\colon LS_{n,m} \times (\mathbf{R}^n \times \mathbf{R}^m) \to P(CS_n)$, $(a, v) \mapsto C$, die die Menge C der Simplizes bestimmt, die das Platelet des Vertizes v in a bilden.
- Ein Simplex $c \in CSn$ aus dem Simplexnetz von a.
- Eine Kante $k \in CE_n$ aus dem Simplexnetz von a.
- Eine Funktion $getOppositeEdge\colon LS_{n,m} \times CS_n \times (\mathbf{R}^n \times \mathbf{R}^m) \to CE_n$, $(a, c, v) \mapsto k$, die die dem Vertex v im Simplex c gegenüberliegende Kante k in a bestimmt.
- Eine Funktion $getTransSimplex\colon LS_{n,m} \times (\mathbf{R}^n \times \mathbf{R}^m) \times CE_n \to CS_n$, $(a, v, k) \mapsto c$, die den von v aus gesehen jenseits einer Kante k liegenden Simplex c in einem linearen Spline a liefert.
- Ein Vertex $vNew \in S$.
- Eine Funktion $getSimplexVertices\colon P(\mathbf{R}^n \times \mathbf{R}^m) \times LS_{n,m} \times CS_n \to P(\mathbf{R}^n \times \mathbf{R}^m)$, $(S, a, c) \mapsto S_c$, die die Untermenge S_c der in einem Simplex c eines linearen Splines a gelegenen Streudatenvertizes aus S bestimmt.

Ausgaben:

- Ein Streudatenvertex $vNew \in S$ aus dem Grenzpolygon von v.

```
/* Bestimme einen der im Grenzpolygon liegenden Simplizes: */
C := getPlatelet(a, v);
c :∈ C;   /* Wähle einen Simplex aus dem Platelet */
if prob(0.5) then   /* Verlasse das Platelet mit Wahrscheinlichkeit 1/2 */
    begin
    /* Bestimme den c gegenüberliegenden Simplex: */
    k := getOppositeEdge(a, c, v);
    c := getTransSimplex(a, v, k);
    end
/* Bestimme die Menge der mit c assoziierten Streudatenvertizes: */
S_c := getSimplexVertices(S, a, c);
vNew :∈ S_c;   /* Wähle einen Vertex aus S_c */
return vNew;
```

Algorithmus 3.7: Bestimmung eines Vertizes im Grenzpolygon von v.

Bei Verwendung dieses Algorithmus ist die erwartete Laufzeit zur Bestimmung eines naheliegenden Streudatenvertizes proportional zu der Anzahl von Streudatenvertizes pro Simplex: $T(\mathit{findLocalVertex}) \in O\big(|S|/|a.V|\big)$. Da die beiden hier angeführten Operationen die Laufzeit des gesamten Iterationsschrittes maßgeblich bestimmen, ist die Laufzeit desselben ebenfalls proportional der Summe der mittleren Anzahl von Vertizes pro Simplex und der Anzahl der Simplizes: $T(\mathit{changeConfiguration}) \in O\big(|S|/|a.V| + |a.V|\big)$. Der zweite Summand fällt wiederum nicht ins Gewicht.

Momentan wird innerhalb unserer Datenstrukturen die Menge der Simplizes nur als ungeordnete doppelt verkettete Liste gespeichert. Diese Darstellung ist zwar einfach, hat aber den Nachteil, daß der Aufwand des Auffindens eines Simplizes, der eine gegebene Stelle enthält, linear in der Anzahl der Simplizes ist. Durch Einsatz besserer Strukturen könnte dieser Aufwand von $O(n)$ auf $O(\log n)$ gedrückt werden[3]. Die Operation, einen Simplex anhand einer enthaltenen Stelle zu finden, tritt jedoch nur in der Funktion *moveVertexGlobally* (siehe Abschnitt 3.4.2) auf. Experimente haben gezeigt, daß diese Funktion nur in etwa 5 Prozent der Iterationsschritte benutzt wird; da weiterhin das Aufrechterhalten solch komplexer Datenstrukturen sehr aufwendig ist, vermuten wir, daß eine solche Optimierung das Laufzeitverhalten des Optimierungsalgorithmus höchstens unwesentlich verbessern würde.

[3]Im univariaten Fall z.B. durch balancierte Bäume, im bivariaten Fall etwa durch Quadtrees oder Kirkpatrick-Triangulierungen (siehe [8], [9] und [10]).

3.5.2 Darstellung univariater linearer Splines

Wie bereits erwähnt, besteht die Datenstruktur univariater linearer Splines aus zwei Teilen:

1. Einer Menge von Vertizes (der Vertexplazierung), gespeichert als doppelt verkettete Liste. Jeder Vertex v ist durch seine Stelle $v.s \in \mathbf{R}$ und seinen Funktionswert $v.f \in \mathbf{R}^m$ beschrieben. Weiterhin enthält jeder Vertex Verweise $v.left$ bzw. $v.right$ auf die links bzw. rechts vom Vertex liegenden Intervalle des Simplexnetzes.

2. Einer Menge von Intervallen (dem Simplexnetz), gespeichert als doppelt verkettete Liste. Ein Intervall c ist durch Verweise $c.left$ bzw. $c.right$ auf die das Intervall nach links bzw. nach rechts begrenzenden Vertizes vollständig beschrieben. Jedes Intervall enthält weiterhin die Menge $v.data$ der diesem Intervall zugeordneten Streudatenvertizes, gespeichert als einfach verkettete Liste.

Die Zusammenhänge dieser Struktur und die Speicherung der Nachbarschaftsbeziehungen sind in Abbildung 3.9 illustriert.

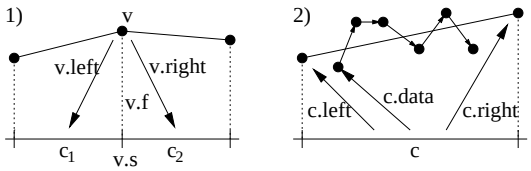

Abbildung 3.9: Darstellung univariater linearer Splines. 1) Struktur eines Vertex v; 2) Struktur eines Intervalls c.

Zu beachten ist hierbei, daß die Vertizes und Intervalle innerhalb ihrer jeweiligen Listen nicht geordnet sind, dasselbe gilt für die Streudatenvertizes in den Listen innerhalb der Intervalle. Um z.B. von einem Vertex v zu seinem rechten Nachbarn zu gelangen, muß man zuerst das rechts von v liegende Intervall $v.right$ und dann den rechten Vertex $v.right.right$ dieses Intervalls aufsuchen. Mit der gleichen Methode kann man die Nachbarn eines Intervalls bestimmen. Die beiden äußeren Vertizes eines Splines sind dadurch gekennzeichnet, daß jeweils nur eine Intervallreferenz definiert ist. Um die äußeren Vertizes schnell auffinden zu können, werden sie innerhalb der Datenstruktur zusätzlich direkt referenziert.

3.5.3 Darstellung bivariater linearer Splines

Die Datenstruktur für bivariate lineare Splines zerfällt in drei Teile:

1. Eine Menge von Vertizes (die Vertexplazierung), gespeichert als doppelt verkettete Liste. Jeder Vertex v ist durch seine Stelle $v.s \in \mathbf{R}^2$ und seinen Funktionswert $v.f \in \mathbf{R}^m$ beschrieben. Weiterhin enthält jeder Vertex eine zyklische doppelt verkettete Liste der den Vertex umgebenden Kanten und Dreiecke. Jedes Element dieser Liste enthält eine Referenz auf eine von v ausgehende Kante k_i und eine Referenz auf das im Gegenuhrzeigersinn „hinter" der Kante liegende Dreieck c_i. Die Listeneinträge sind ebenfalls im Gegenuhrzeigersinn angeordnet. Ist v ein äußerer Vertex, dann ist die Umlaufliste offen, und das letzte Listenelement enthält nur eine Referenz auf eine Kante. Sind l_1 und l_2 zwei aufeinanderfolgende Listeneinträge, dann stellt diese Anordnung sicher, daß die Kante $l_1.k$ zwischen den Dreiecken $l_1.c$ und $l_2.c$ liegt; und daß das Dreieck $l_1.c$ zwischen den Kanten $l_1.k$ und $l_2.k$ liegt. Diese Datenstruktur ermöglicht ebenfalls die Modellierung von Löchern innerhalb des Simplexnetzes, diese treten in unserem Algorithmus aber nur als Zwischenstadium in der Funktion *moveVertexGlobally* (siehe Abschnitt 3.4.2) auf und werden ansonsten verboten.

2. Eine Menge von Kanten (die Kanten des Simplexnetzes), gespeichert als doppelt verkettete Liste. Jede Kante k ist durch ihre Geradengleichung $(x, y) \cdot (n_x, n_y)^T = d$ definiert. Weiterhin enthält jede Kante zwei Referenzen auf die Endpunkte v_1 und v_2 der Kante und auf die die Kante teilenden Dreiecke c_1 und c_2. Sei $(n_x, n_y) \in \mathbf{R}^2$ der Normalvektor der Kante, dann ist v_1 der in Normalenrichtung gesehen rechte Endpunkt; v_2 liegt in Normalenrichtung links. Das Dreieck c_2 liegt auf der Seite der Kante, auf die der Normalvektor zeigt, c_1 liegt demzufolge „hinter" der Kante. Ist eine Kante Teil des Randes des Simplexnetzes, dann zeigt der Normalvektor stets nach außen, und der Eintrag $k.c_2$ ist undefiniert.

3. Eine Menge von Dreiecken (das Simplexnetz), gespeichert als doppelt verkettete Liste. Jedes Dreieck ist durch drei Referenzen auf die Eckpunkte v_1, v_2 und v_3 sowie durch drei Referenzen auf die Kanten k_1, k_2 und k_3 beschrieben. Weiterhin enthält jedes Dreieck eine Menge von assoziierten Streudatenvertizes, gespeichert als einfach verkettete

Liste. Die Eckpunkte und Kanten sind hierbei im Gegenuhrzeigersinn aufgezählt und so angeordnet, daß eine Kante k_i stets zwischen den Eckpunkten v_i und v_{i+1} liegt; und daß ein Eckpunkt v_{i+1} von den Kanten k_i und k_{i+1} geteilt wird[4].

Die Zusammenhänge dieser Struktur und die Speicherung der Nachbarschaftsbeziehungen sind in Abbildung 3.10 illustriert.

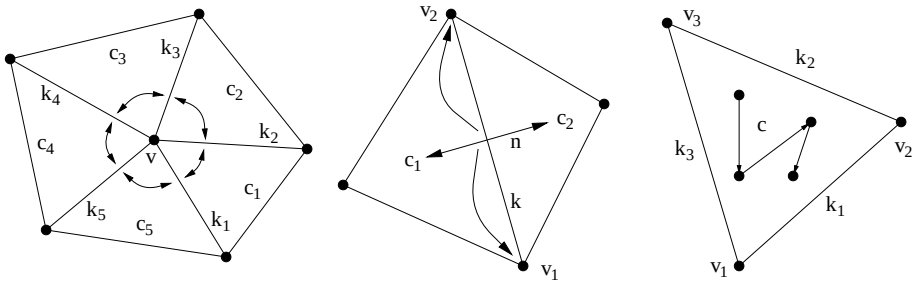

Abbildung 3.10: Darstellung bivariater linearer Splines. 1) Struktur eines Vertex v; 2) Struktur einer Kante k; 3) Struktur eines Dreiecks c.

Wie im univariaten Fall sind die drei Mengen durch ungeordnete Listen repräsentiert; um innerhalb der Datenstruktur den Nachbarschaftsbeziehungen zu folgen, müssen die Referenzen der einzelnen Elemente der Mengen benutzt werden. Um die den Rand des Simplexnetzes bildenden Kanten schnell auffinden zu können, enthält die Datenstruktur bivariater linearer Splines zusätzlich zu diesen drei Teilen noch eine zyklische doppelt verkettete Liste jener Kanten, die den Rand des Simplexnetzes definieren. Diese äußeren Kanten sind wie üblich im Gegenuhrzeigersinn angeordnet.

3.5.4 Zurücknahme von Änderungen

Im Rahmen des Optimierungsalgorithmus wird in jedem Iterationsschritt die Konfiguration der aktuellen Approximation geändert; weist der Simulated-Annealing-Algorithmus eine Änderung zurück, muß der vorherige Zustand der Konfiguration wiederhergestellt werden. Da das Verfahren speziell auf große Streudatenmengen und Approximationen mit vielen Vertizes bzw. Simplizes angewendet werden soll, wäre die Erzeugung temporärer Kopien der aktuellen Konfiguration ineffizient. Um die Rücknahme von Änderungen zu

[4]Der Bequemlichkeit halber identifizieren wir hier v_4 bzw. k_4 mit v_1 bzw. k_1.

ermöglichen, muß also innerhalb der Datenstrukturen über vergangene Änderungen Buch geführt werden.

Zu diesem Zweck ergänzen wir die Datenstrukturen um einen Undo-Puffer, in dem sämtliche zur Wiederherstellung eines alten Zustands benötigten Informationen abgelegt werden. In speziellen Anwendungen möchte man eventuell die Möglichkeit haben, mehrere in der Vergangenheit vorgenommene Änderungen zurückzunehmen; deshalb implementieren wir den Undo-Puffer als einen Stapelspeicher.

Die vier Basisoperationen jeder Änderung sind:

1. Rotation einer Kante.

2. Einfügen eines Vertizes. Im Zuge einer Einfügeoperation werden eventuell auch eine oder mehrere Kanten rotiert.

3. Entfernen eines Vertizes. Im Zuge einer Entfernoperation werden eventuell auch eine oder mehrere Kanten rotiert.

4. Lokale Bewegung eines Vertizes. Wie in Abschnitt 3.4.3 beschrieben, werden bei lokalen Vertexbewegungen in der Regel mehrere Kanten rotiert.

Wir definieren dann einen Eintrag des Undo-Puffers wie folgt:

- Eine Liste von Kanten, die vor der Rücknahme des Schrittes rotiert werden müssen. War die Operation eine Kantenrotation, enthält diese Liste nur ein Element; weitere Informationen sind dann nicht erforderlich.

- War die Operation die Einfügung eines Vertizes, dann speichern wir eine Referenz auf den eingefügten Vertex.

- War die Operation die Entfernung eines Vertizes, dann speichern wir die Stelle und den Funktionswert des entfernten Vertizes.

- War die Operation die lokale Bewegung eines Vertizes, dann speichern wir eine Referenz auf den bewegten Vertex und seine alte Stelle und seinen alten Funktionswert.

Kapitel 4

Experimente und Resultate

In diesem Kapitel beschreiben wir die Streudatenmengen, mit denen wir den Approximierungsalgorithmus getestet haben, und die jeweiligen Approximationsresultate. Weiterhin beschreiben wir einige Heuristiken zur Bestimmung der Parameter des Algorithmus, die wir im Verlauf der Experimente entwickelt haben.

Der Approximierungsalgorithmus existiert momentan in drei verschiedenen „Geschmacksrichtungen", je eine für univariate skalarwertige Funktionen, bivariate skalarwertige Funktionen und bivariate vektorwertige Funktionen; die folgenden Abschnitte behandeln die Anwendung dieser Variationen auf verschiedene Testdatensätze.

Wir haben versucht, die Testfälle in den ersten beiden Variationen analog anzuordnen: Zuerst behandeln wir Fälle, in denen die „Optimalität" des Algorithmus mathematisch bewertet werden kann; als nächstes führen wir Fälle an, die die Stärken des Verfahrens aufzeigen; danach behandeln wir Fälle von „Unterabtastungen", in denen eine zu geringe Zahl von Vertizes verlangt wird, als daß eine gegebene Funktion gut approximiert werden könnte.

Die Testfunktionen der dritten Variation unterscheiden sich konzeptionell zu sehr von den ersten beiden für eine analoge Behandlung; wir beginnen aber trotzdem mit „einfachen", d.h. detailarmen, Testfällen und gehen dann zu detailreichen über, um die Güte des Verfahrens zur Bildkompression bewerten zu können.

4.1 Univariate skalarwertige Funktionen

In diesem Abschnitt wenden wir die erste Variation des Verfahrens auf analytisch definierte Funktionen an; die Eingabemengen entstehen durch Abtastung dieser Funktionen in zufällig verteilten Stellen.

1. Der erste Testfall ist die Funktion $f(x) := x^2 - 2$, $x \in [-2, 2]$, approximiert durch einen linearen Spline mit zehn Kontrollpunkten (siehe Abbildung 4.1). In diesem Fall ist die theoretisch optimale Approximation ein linearer Spline mit gleichmäßig über den Definitionsbereich verteilten Vertizes. Da die Eingabe des Verfahrens aus zufällig verteilten Abtastwerten besteht, kann dieses Optimum nicht exakt erreicht werden; das Ergebnis unseres Algorithmus kommt diesem Optimum aber bereits nach etwa 3.000 Schritten so nah wie möglich.

2. Der zweite Testfall ist die Funktion $f(x) := 3\sin(x^2)$, $x \in [0, 4\sqrt{\pi}]$, approximiert durch einen linearen Spline mit 18 Kontrollpunkten (siehe Abbildung 4.2). In diesem Fall fiele ein mathematischer Optimalitätsbeweis schwer, aber die Anschauung legt nahe, daß das Resultat des Algorithmus global optimal ist. Weiterhin zeigt dieses Experiment die Überlegenheit des Simulated-Annealing-Verfahrens für Approximationsprobleme; wir haben es nicht geschafft, das Ergebnis von Abbildung 4.2 mit einem Raffke-Optimierungsverfahren zu reproduzieren – zumindest nicht, ohne A--priori-Wissen über die Funktion in Form einer bereits guten initialen Konfiguration zu investieren.

3. Der dritte Testfall ist die Funktion

$$f(x) := \begin{cases} 2(1-x) & \text{if } x < 1 \\ 4(1-x)(x-2) & \text{if } 1 \le x < 2 \\ 2(x-2) & \text{if } 2 \le x < 3 \\ 2(x-3)^2 & \text{if } 3 \le x \end{cases}, \quad x \in [0, 4],$$

approximiert durch einen linearen Spline mit 14 Kontrollpunkten (siehe Abbildung 4.3). Diese Funktion ist ein „schwieriger“ Fall, da sie Unstetigkeit sowohl in der nullten als auch in der ersten Ableitung aufweist. Ungeachtet dessen findet unser Verfahren eine sehr gute Approximation. Man beachte, daß das Resultat Kontrollpunkte genau an den Stellen der Unstetigkeiten aufweist (zwei Kontrollpunkte pro Unstetigkeit der nullten Ableitung), und daß die affinen Stücke $[0, 1]$ und $[2, 3]$ keine

Kontrollpunkte im inneren haben. Da die quadratischen Stücke $[1,2]$ und $[3,4]$ (nahezu) uniforme Kontrollpunktverteilungen aufweisen, nehmen wir wiederum guten Gewissens an, daß diese Approximation global optimal ist.

4. Der vierte Testfall ist die Funktion

$$f(x) := 4 \sum_{n=0}^{3} \frac{\sin\big((2n+1)x\big)}{2n+1} \quad , \quad x \in [0, 4\pi],$$

die Fourierapproximation vierter Ordnung einer Rechteckschwingung, approximiert durch einen linearen Spline mit zehn Kontrollpunkten (siehe Abbildung 4.4). Diese Funktion weist viele kleine Details auf – zuviele, um sie mit nur zehn Vertizes wiederzugeben. Dennoch findet unser Algorithmus eine sehr gute Approximation, die das Gesamtaussehen der Funktion wiedergibt und die ausgelassenen Details mittelt. Ein Beweis fällt wiederum schwer, aber wir gehen davon aus, daß das Resultat in Abbildung 4.4 für die gegebene Zahl von Vertizes global optimal ist.

5. Der fünfte Testfall ist die gleiche Funktion wie in Experiment 4, diesmal approximiert durch einen linearen Spline mit 20 Kontrollpunkten (siehe Abbildung 4.5). Diese Zahl ist immer noch zuwenig, um alle Details einzufangen. Dieser Testfall zeigt einen Nachteil jedes probabilistischen Optimierungsverfahrens auf: Da alle Details der Funktion in etwa die gleiche Bedeutung haben, ist es unmöglich vorherzusagen, welche durch das Resultat wiedergegeben werden. Jeder Lauf des Algorithmus erzeugt eine andere Approximation; alle diese Approximationen haben in etwa den gleichen Abstand von der Streudatenmenge. Dieser generelle Nachteil kann jedoch durch einen Vorteil des Simulated-Annealing-Verfahrens überwunden werden: Wie in Abschnitt 2.2 erwähnt, kann jedes Fehlermaß für die Approximation verwendet werden. Um bestimmte Details einer Funktion bevorzugt wiederzugeben, kann man also das Standard-L^2-Abstandmaß durch eine „Straffunktion“ ergänzen, die die Auslassung der gewünschten Details durch eine künstliche Erhöhung des Abstandmasses erhöht. Die Experimente 7 und 8 zeigen die Verwendung unterschiedlicher Abstandmasse bei der Approximation unter sonst gleichen Randbedingungen.

6. Der sechste Testfall ist wiederum die gleiche Funktion wie in den Experimenten 4 und 5, zu guter letzt approximiert durch einen linearen Spline mit 30 Kontrollpunkten (siehe Abbildung 4.6). Nun kann die Approximation alle Details der Funktion wiedergeben; und wiederum gehen wir davon aus, daß das in Abbildung 4.6 gezeigte Endergebnis global optimal ist.

7. Der siebte Testfall ist die Funktion $f(x) := 3\sin(x^2) \cdot e^{-1/3x}$, $x \in [0, 4\sqrt{\pi}]$, approximiert durch einen linearen Spline mit 18 Kontrollpunkten (siehe Abbildung 4.7). Diese Funktion ist vergleichbar mit der in Experiment 2 verwendeten; in diesem Fall sind aber die Schwingungen auf der rechten Seite der Funktion sehr klein im Vergleich zu denen auf der linken Seite. Das Endergebnis des Verfahrens „unterschlägt“ diese kleinen Schwingungen, da eine bessere Approximation der großen Schwingungen auf der linken Seite für den Gesamtabstand von der Streudatenmenge bedeutender ist.

8. Der achte Testfall ist die gleiche Funktion wie in Experiment 7, ebenfalls approximiert durch einen linearen Spline mit 18 Kontrollpunkten (siehe Abbildung 4.8). In diesem Experiment verwenden wir jedoch ein modifiziertes Abstandmaß, das die Approximation lokaler Extrema der Eingabemenge erzwingt. Sei S die Eingabestreudatenmenge, $E := \{x_1, \ldots, x_e\} \subset \mathbf{R}$ die Menge der Abszissenwerte der lokalen Extrema der Eingabemenge und $a := (V, CS) \in LS_{1,1}$ ein S approximierender linearer Spline mit Vertexplazierung V. Dann definieren wir als neues Abstandmaß

$$E(S, a) := L^2(S, a) + 2 \sum_{i=1}^{e} \min\{ (v.s - x_i)^2 \mid v \in V \},$$

d.h. wir addieren zum L^2-Abstand von S und a die Summe der quadrierten Abstände aller lokalen Extrema von der Stelle des jeweils nächstliegenden Vertex. Der Gewichtungsfaktor 2 wurde zur Angleichung der Straffunktion an den L^2-Abstand eingesetzt. Durch diese künstlich eingeführte Straffunktion wird die Approximierung auch der „unwichtigen“ lokalen Extrema erzwungen.

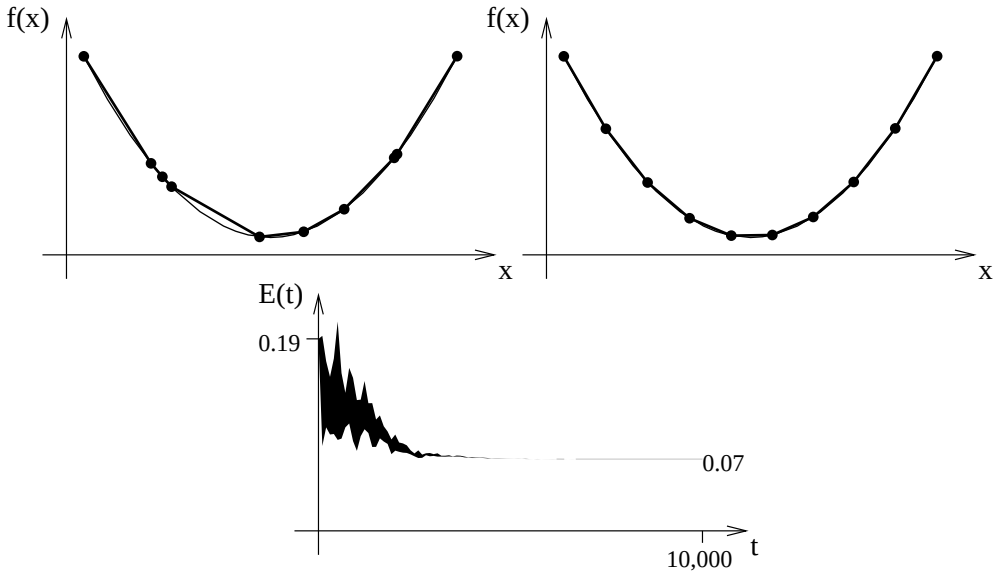

Abbildung 4.1: Experiment 1. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

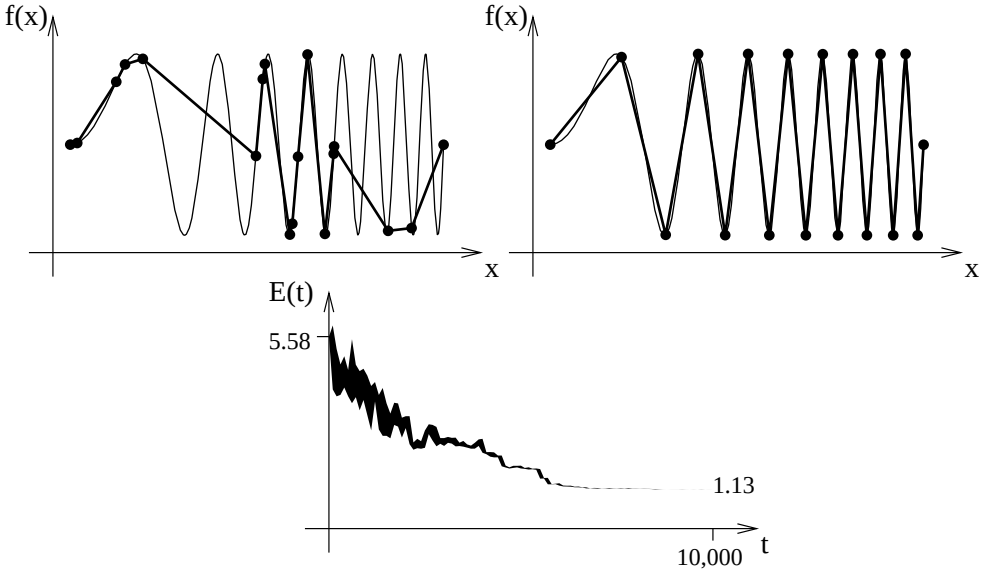

Abbildung 4.2: Experiment 2. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

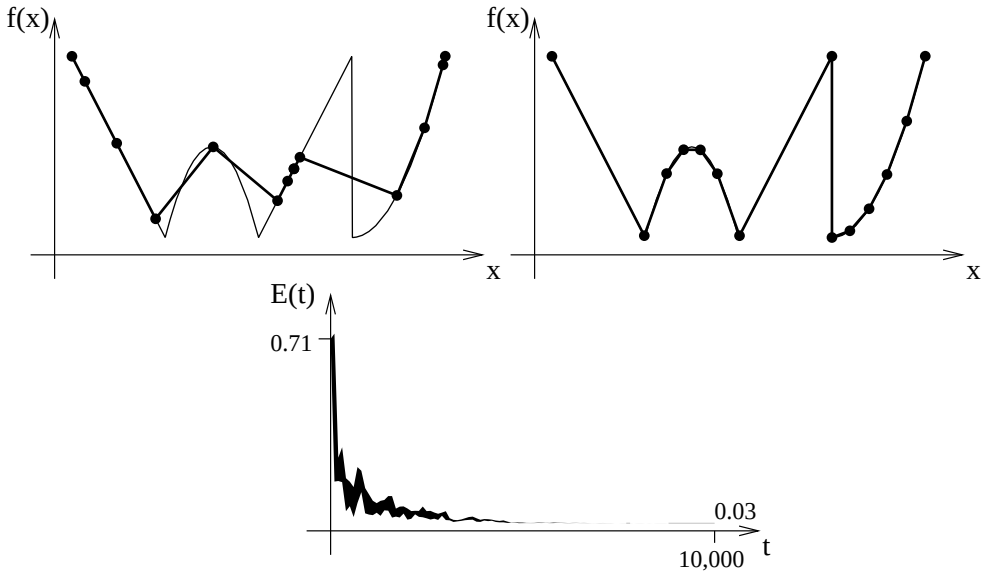

Abbildung 4.3: Experiment 3. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

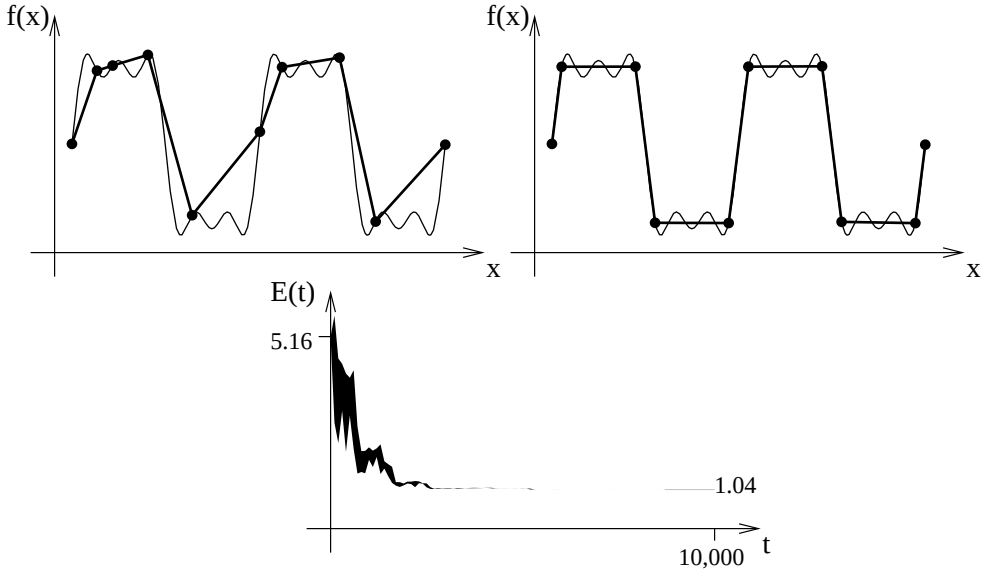

Abbildung 4.4: Experiment 4. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

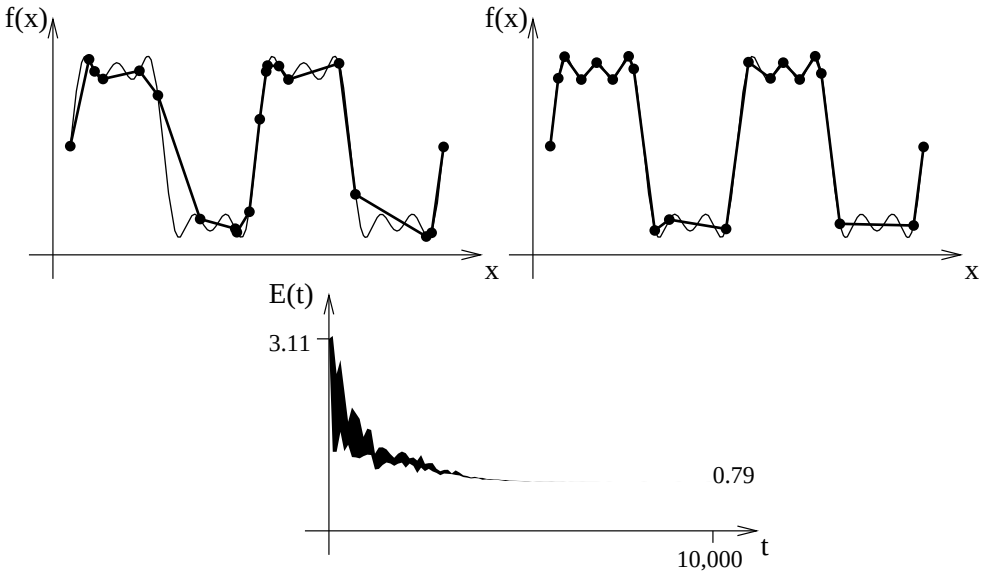

Abbildung 4.5: Experiment 5. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

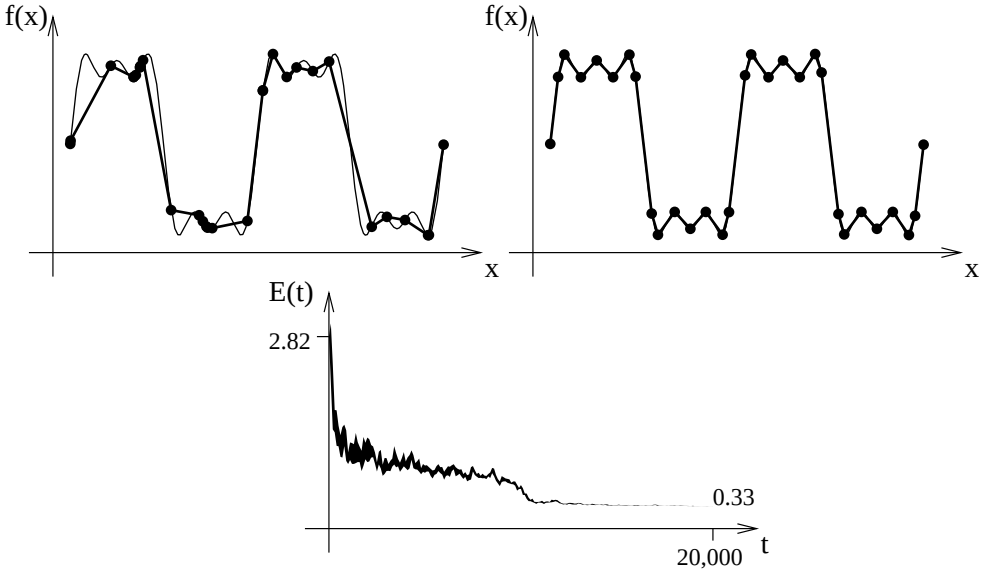

Abbildung 4.6: Experiment 6. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

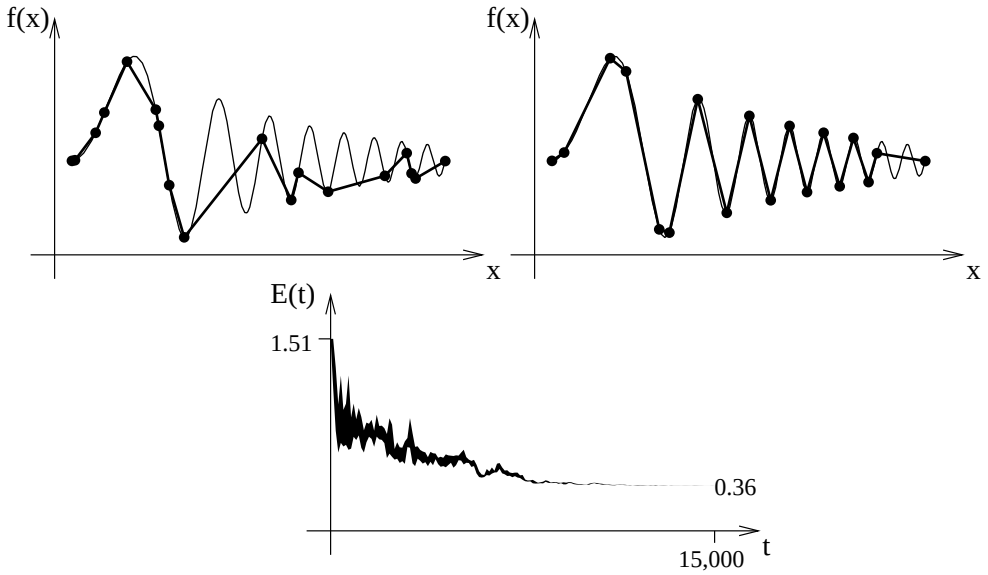

Abbildung 4.7: Experiment 7. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

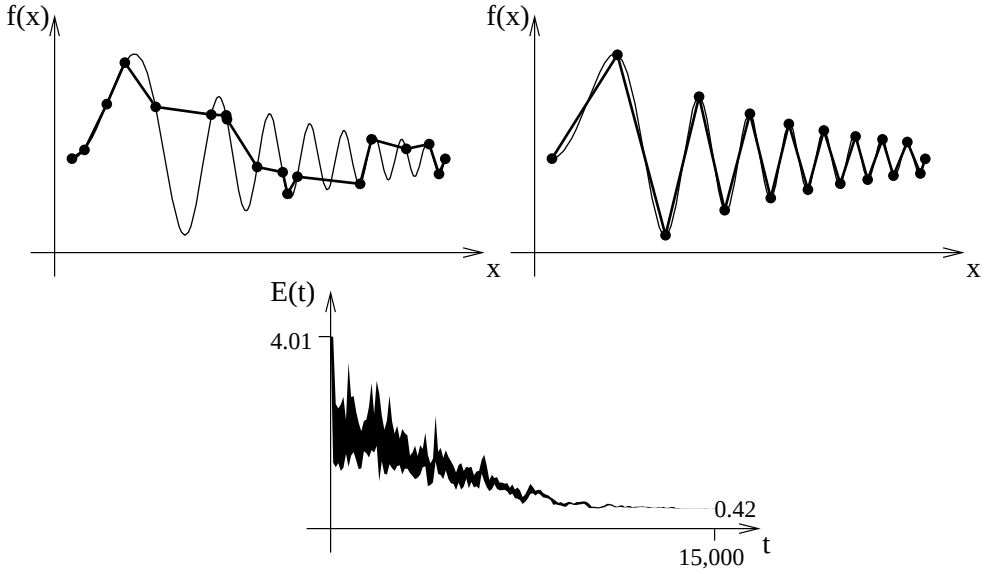

Abbildung 4.8: Experiment 8. Oben links: Anfangskonfiguration, oben rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

4.2 Bivariate skalarwertige Funktionen

In diesem Abschnitt wenden wir die zweite Variation des Verfahrens auf analytisch definierte Funktionen an; die Eingabemengen entstehen durch Abtastung dieser Funktionen in zufällig verteilten Stellen. Anschließend wenden wir das Verfahren auf Streudatenmengen an, die aus der Abtastung der Oberflächen physischer Objekte, z.B. durch Laserscans, entstanden sind; wir zeigen damit die Verwendbarkeit des Verfahrens zur Lösung von Surface-Reconstruction-Problemen.

9. Der neunte Testfall ist die Funktion $f(x,y) := \frac{1}{3}(x^2+y^2-5)$, $x,y \in [-3,3]$, approximiert durch einen linearen Spline mit 100 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildung 4.9). In diesem Fall ist die theoretisch optimale Approximierung durch eine Poisson-Verteilung der Kontrollpunkte über den Definitionsbereich und eine Delaunay-Triangulierung als Simplexnetz beschrieben. Aufgrund der Repräsentation der Funktion durch Streudaten und die Notwendigkeit, alle auf der konvexen Hülle der Stellen der Streudatenmenge liegenden Vertizes in die Vertexplazierung zu übernehmen, kann dieses Ideal nicht exakt erreicht werden; das Resultat des Verfahrens liegt dem theoretischen Optimum aber so nahe wie möglich. Man beachte weiterhin, daß die Aufrechterhaltung der Delaunay-Bedingung während der Iteration nicht erzwungen wurde; der Algorithmus hat die (für diesen Fall) optimale Triangulierung selbständig gefunden.

10. Der zehnte Testfall ist die Funktion

$$f(x,y) := \begin{cases} 0.3 & \text{falls } x^2+y^2 \leq 0.3 \\ -0.5x & \text{falls } x^2+y^2 > 0.3 \wedge x < 0 \\ x^2 & \text{falls } x^2+y^2 > 0.3 \wedge 0 \leq x \end{cases}, \quad x,y \in [-1,1],$$

approximiert durch einen linearen Spline mit 100 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildung 4.10). Diese Funktion ist „schwierig“, da sie Unstetigkeiten sowohl in der nullten als auch in der ersten Ableitung aufweist. Trotzdem findet unser Algorithmus eine sehr gute Approximation. Folgendes Punkte fallen bei der Betrachtung des Endergebnisses auf:

- Die Unstetigkeiten der nullten Ableitung sind durch Streifen sehr dünner Dreiecke markiert.

- Die Unstetigkeiten der ersten Ableitung werden durch Kantenzüge nachgezeichnet.
- Die affinen Stücke der Funktion enthalten keine Vertizes im Inneren; die lokalen Triangulierungen dieser Stücke sind völlig zufällig.
- Der quadratische Teil der Funktion weist sehr lange, dünne Dreiecke senkrecht zur Gradientenrichtung auf.
- Die Kontrollpunkte im quadratischen Teil sind entlang der Gradientenrichtung annähernd gleichmäßig verteilt.

11. Der elfte Testfall ist die gleiche Funktion wie in Experiment 10, wiederum approximiert durch einen linearen Spline mit 100 Kontrollpunkten, aber diesmal einer Delaunay-Triangulierung (siehe Abbildung 4.11). Die Aufrechterhaltung der Delaunay-Bedingung während der gesamten Iteration vergrößert den Abstand der endgültigen Approximation von der Streudatenmenge in etwa um den Faktor zwei. Dieser Testfall zeigt außerdem, daß die Nachbehandlung des Ergebnisses mit einem Algorithmus zur datenabhängigen Triangulierung (siehe Abschnitt 2.3.2) höchstens ein suboptimales Ergebnis erzeugen kann. Die Vertexplazierung aus Abbildung 4.11 unterscheidet sich stark von der aus Abbildung 4.10. Da ein Triangulierungsalgorithmus die Vertexplazierung nicht ändern kann, kann das Ergebnis von Experiment 10 durch einen zweiteiligen Algorithmus nicht erreicht werden. In diesem speziellen Fall war die allgemeine Iteration sogar schneller als die restringierte; aufgrund der höheren Anzahl von Freiheitsgraden ist dies im allgemeinen jedoch nicht der Fall.

12. Der zwölfte Testfall ist die Funktion

$$f(x,y) := 2\sum_{i=0}^{2}\sum_{j=0}^{2} \frac{\sin\big((2i+1)x\big)}{2i+1} \cdot \frac{\sin\big((2j+1)y\big)}{2j+1} \quad , \quad x,y \in [0,2\pi],$$

die Fourierapproximation dritter Ordnung einer bivariaten Rechteckschwingung, approximiert durch einen linearen Spline mit 50 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildung 4.12). Diese Funktion weist zu viele kleine Details auf, um sie mit nur 50 Vertizes wiederzugeben. Dennoch findet unser Algorithmus eine sehr gute Approximation, die das Gesamtaussehen der Funktion wiedergibt und die ausgelassenen Details mittelt.

13. Der 13. Testfall ist die gleiche Funktion wie in Experiment 12, diesmal approximiert durch einen linearen Spline mit 250 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildung 4.13). Durch die höhere Anzahl von Vertizes benötigt der Algorithmus wesentlich mehr Schritte bis zur Konvergenz, aber das Ergebnis gibt alle Details der Streudatenmenge wieder.

14. Der 14. Testfall ist ein Surface-Reconstruction-Problem. Die Eingabemenge resultierte aus der Abtastung der Haube eines Ski-Doos mittels eines Laserscanners und besteht aus 37.594 zufällig verteilten Abtastwerten. Wir interpretieren diese Streudatenmenge als die Abtastung einer bivariaten skalarwertigen Funktion $f(x, y)$und approximieren diese durch eine Hierarchie linearer Splines (siehe Abschnitt 1.2) mit 400, 700 und 1.000 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildungen 4.14, 4.15 und 4.16).

    Unser Verfahren findet eine sehr gute Rekonstruktion der ursprünglichen Oberfläche. Die Löcher der Haube werden in der Rekonstruktion durch große Dreiecke wiedergegeben; diese Dreiecke können jedoch in einem Nachbearbeitungsschritt leicht entfernt werden, da sie keine assoziierten Streudatenvertizes haben und somit vom Optimierungsalgorithmus automatisch markiert werden.

15. Der 15. Testfall ist die Rekonstruktion einer digitalen Höhenkarte des Mt. St. Helens im US-Bundesstaat Washington, bestehend aus 151.728 Vertizes. In diesem speziellen Fall sind die Abtastwerte über einem rectilinearen Gitter angeordnet; für unseren Algorithmus macht dies aber keinen Unterschied. Wie in Experiment 14 interpretieren wir die Vertexmenge als die Abtastung einer bivariaten Funktion $f(x, y)$ und approximieren diese durch eine Hierarchie linearer Splines (siehe Abschnitt 1.2) mit 400, 800 und 1.600 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildungen 4.17, 4.18 und 4.19).

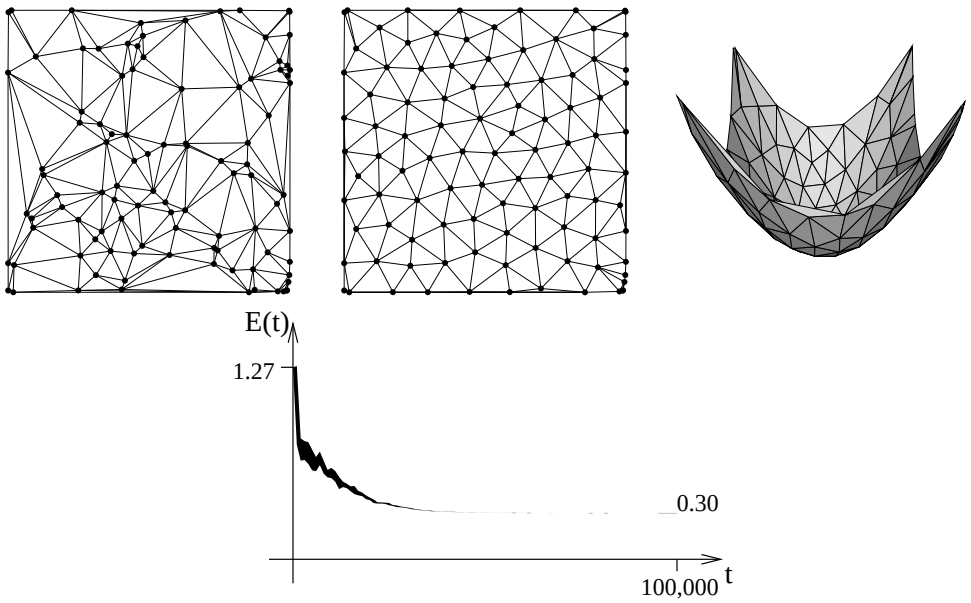

Abbildung 4.9: Experiment 9. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

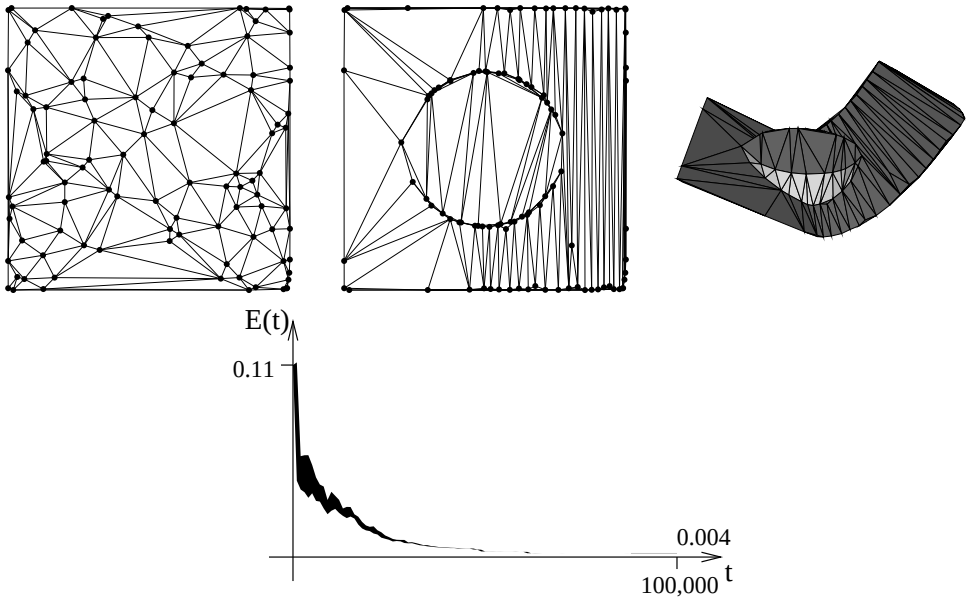

Abbildung 4.10: Experiment 10. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

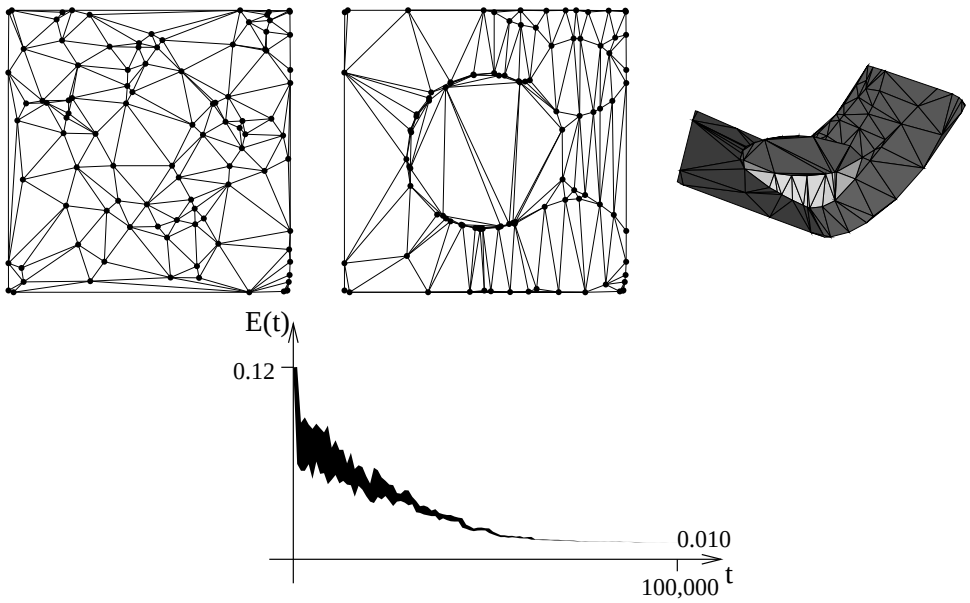

Abbildung 4.11: Experiment 11. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

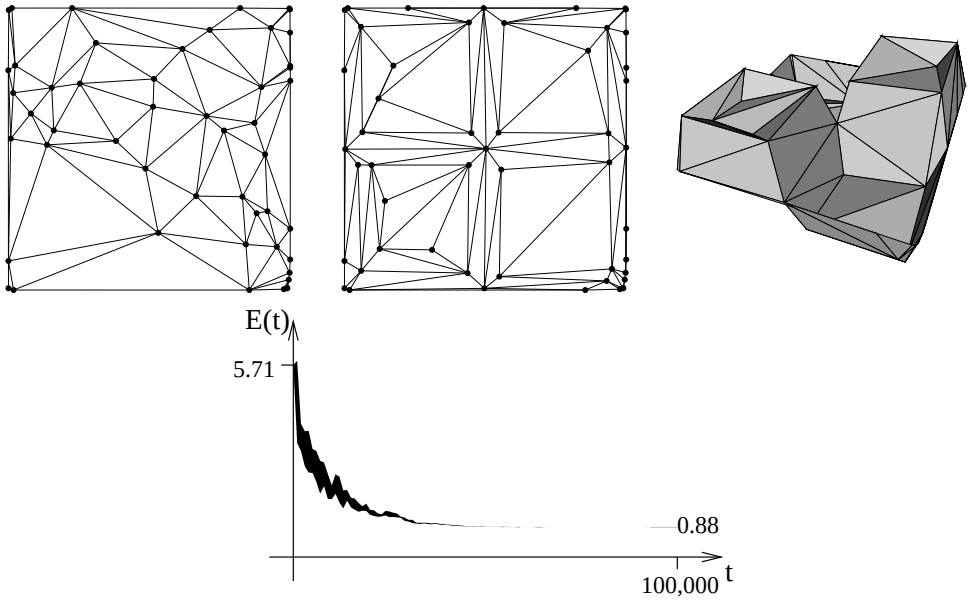

Abbildung 4.12: Experiment 12. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

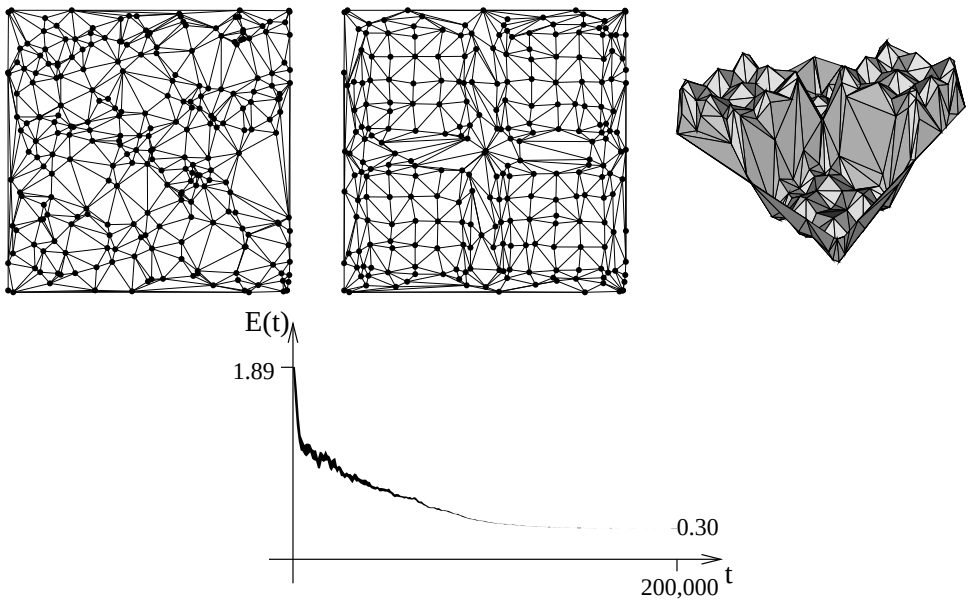

Abbildung 4.13: Experiment 13. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

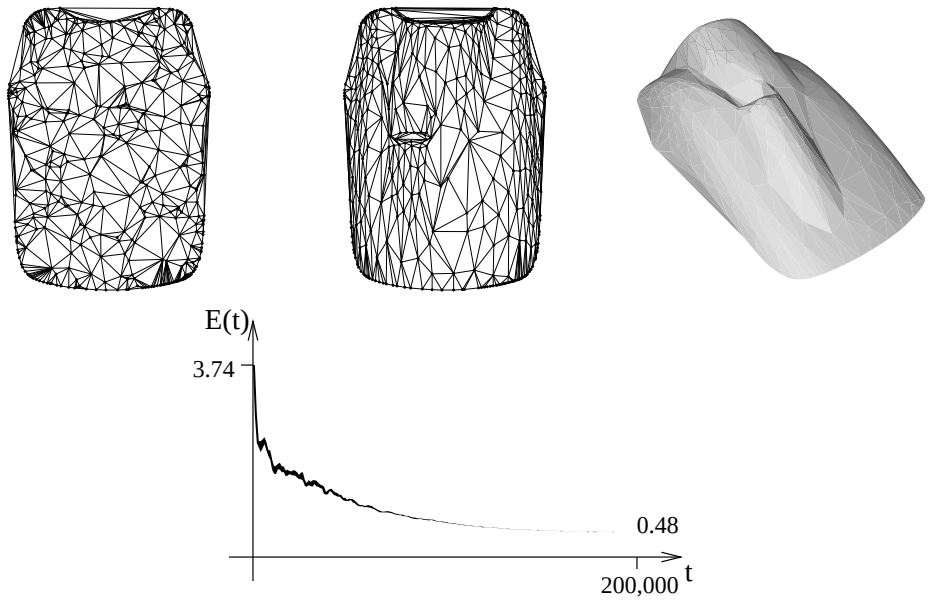

Abbildung 4.14: Experiment 14, Teil 1. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

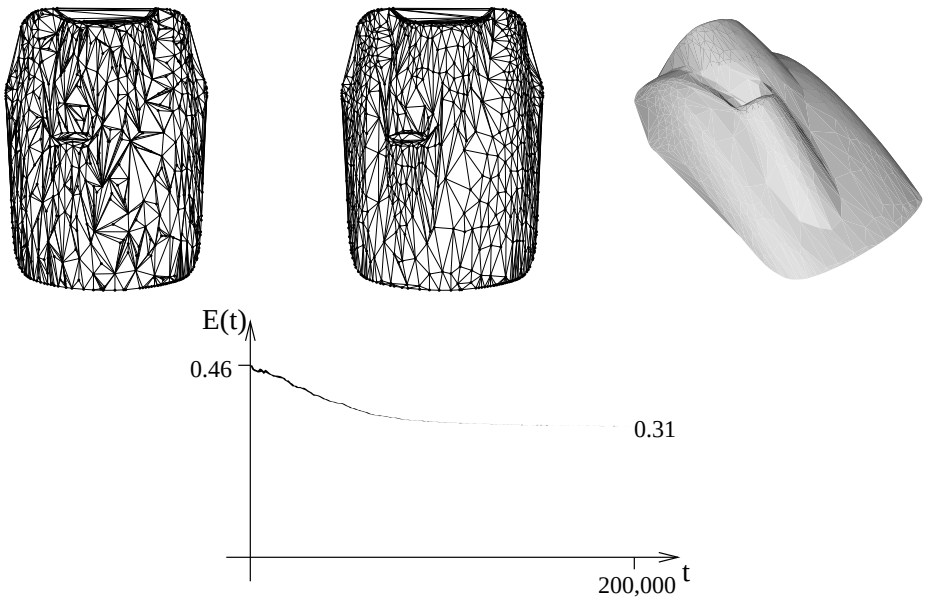

Abbildung 4.15: Experiment 14, Teil 2. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

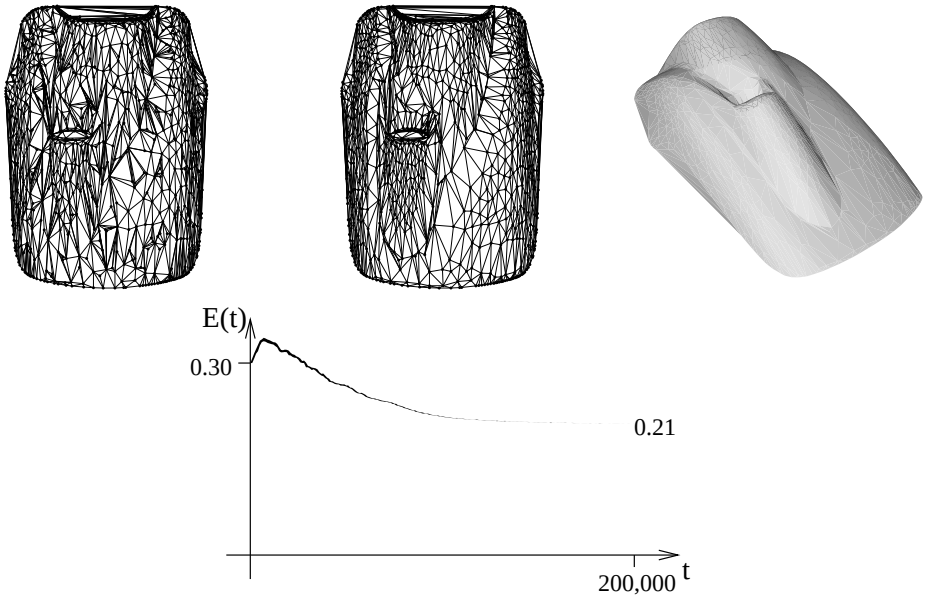

Abbildung 4.16: Experiment 14, Teil 3. Oben links: Anfangskonfiguration, oben mitte und rechts: Endkonfiguration, unten: Abstandmaß über Zeit.

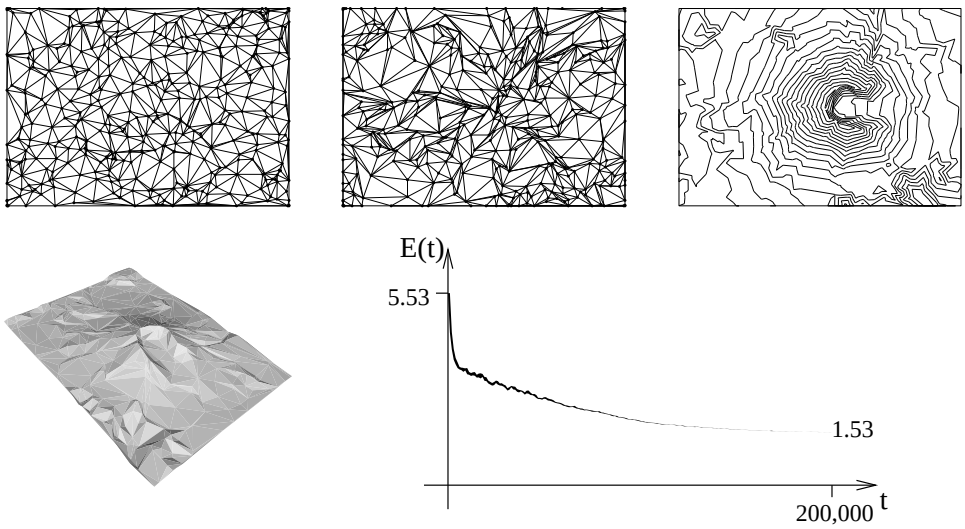

Abbildung 4.17: Experiment 15, Teil 1. Oben links: Anfangskonfiguration, oben mitte, rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

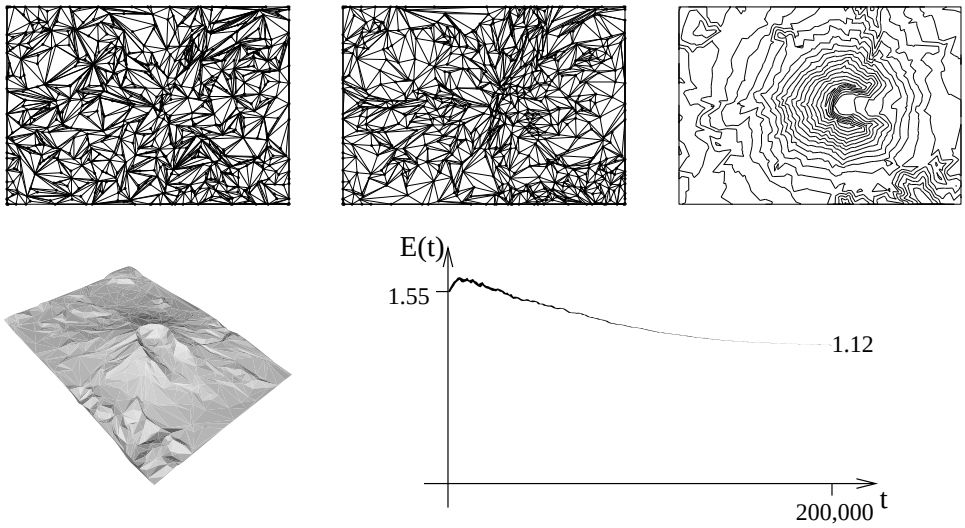

Abbildung 4.18: Experiment 15, Teil 2. Oben links: Anfangskonfiguration, oben mitte, rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

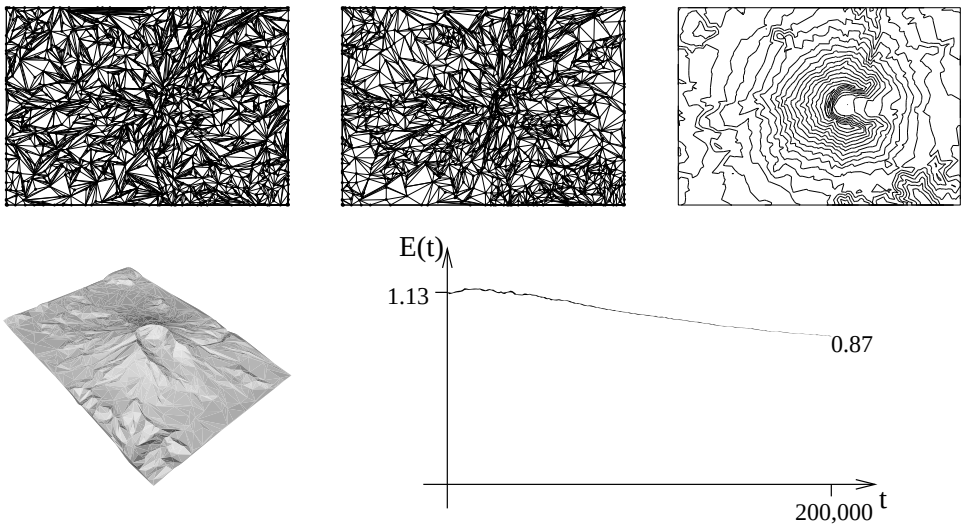

Abbildung 4.19: Experiment 15, Teil 3. Oben links: Anfangskonfiguration, oben mitte, rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

4.3 Bivariate vektorwertige Funktionen

In diesem Abschnitt wenden wir die dritte Variation des Verfahrens auf Farbbilder an. Wir interpretieren dazu Echtfarb-RGB-Bitmaps als bivariate vektorwertige Funktionen der Form

$$f: \mathbf{R}^2 \to [0,1]^3 \quad , \quad (x,y) \mapsto \big(r(x,y), g(x,y), b(x,y)\big) \quad .$$

Wir definieren die Eingabemengen des Verfahrens durch die Menge der Pixel eines Bildes: Wir verwenden die Mittelpunkte der einzelnen Pixel als Stellen und definieren die Funktionswerte der Vertizes durch die Farben der jeweiligen Pixel $(r_i, g_i, b_i) \in [0,1]^3$. Zur Bestimmung des Abstands einer Approximation von der gegebenen Streudatenmenge verwenden wir wie in den anderen Fällen die L^2-Metrik, wobei wir nun den Ordinatenabstand zweier Funktionswerte f_1, f_2 als den euklidischen Abstand $\|f_1 - f_2\|_2$ der RGB-Farbwerte, interpretiert als Vektoren des $\mathbf{R}^3$, definieren.

16. Der 16. Testfall ist ein (vergleichsweise detailarmes) Portraitfoto, abgetastet mit einer Auflösung von 212×272 Pixeln (siehe Abbildung 4.20). Wir approximieren dieses Bild durch eine Hierarchie linearer Splines mit 400, 800, 1.600 und 3.200 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildungen 4.21, 4.22, 4.23 und 4.24). In jeder Stufe der Approximation fuehren wir 250.000 Schritte des Optimierungsalgorithmus aus. Zum Vergleich zeigen wir auch eine Approximierung des Bildes durch einen direkt berechneten linearen Spline mit 3.200 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildung 4.25). Für diese direkte Approximierung führen wir die gleich Anzahl an Schritten, 1.000.000, des Optimierungsalgorithmus aus. Der Vergleich der Ergebnisse bestätigt, daß bei Verwendung einer Approximierungshierarchie das Ergebnis der einzelnen Stufen suboptimal ist; für die Vorteile der Erzeugung einer Hierarchie nehmen wir diesen Nachteil aber in Kauf.

17. Der 17. Testfall ist der bekannte und immer wieder gern verwendete Ausschnitt des Bildes von „Lena“, dem Playmate des Monats November 1972, abgetastet mit einer Auflösung von 256×256 Pixeln (siehe Abbildung 4.26). Wir approximieren dieses Bild durch eine Hierarchie linearer Splines mit 400, 800, 1.600 und 3.200 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildungen 4.27, 4.28, 4.29 und 4.30).

Zum Vergleich erzeugen wir wiederum eine direkte Approximation mit 3.200 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildung 4.31).

18. Der 18. Testfall ist ein detailreiches Foto der Golden Gate Bridge in San Francisco, abgetastet mit einer Auflösung von 329×222 Pixeln (siehe Abbildung 4.32). Wir approximieren dieses Bild durch eine Hierarchie linearer Splines mit 400, 800, 1.600 und 3.200 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildungen 4.33, 4.34, 4.35 und 4.36). Zum Vergleich erzeugen wir wiederum eine direkte Approximation mit 3.200 Kontrollpunkten und allgemeiner Triangulierung (siehe Abbildung 4.37).

Abbildung 4.20: „Oliver“. Eine farbige Version meines aktuellen Paßfotos, abgetastet mit einer Auflösung von 212×272 Pixeln.

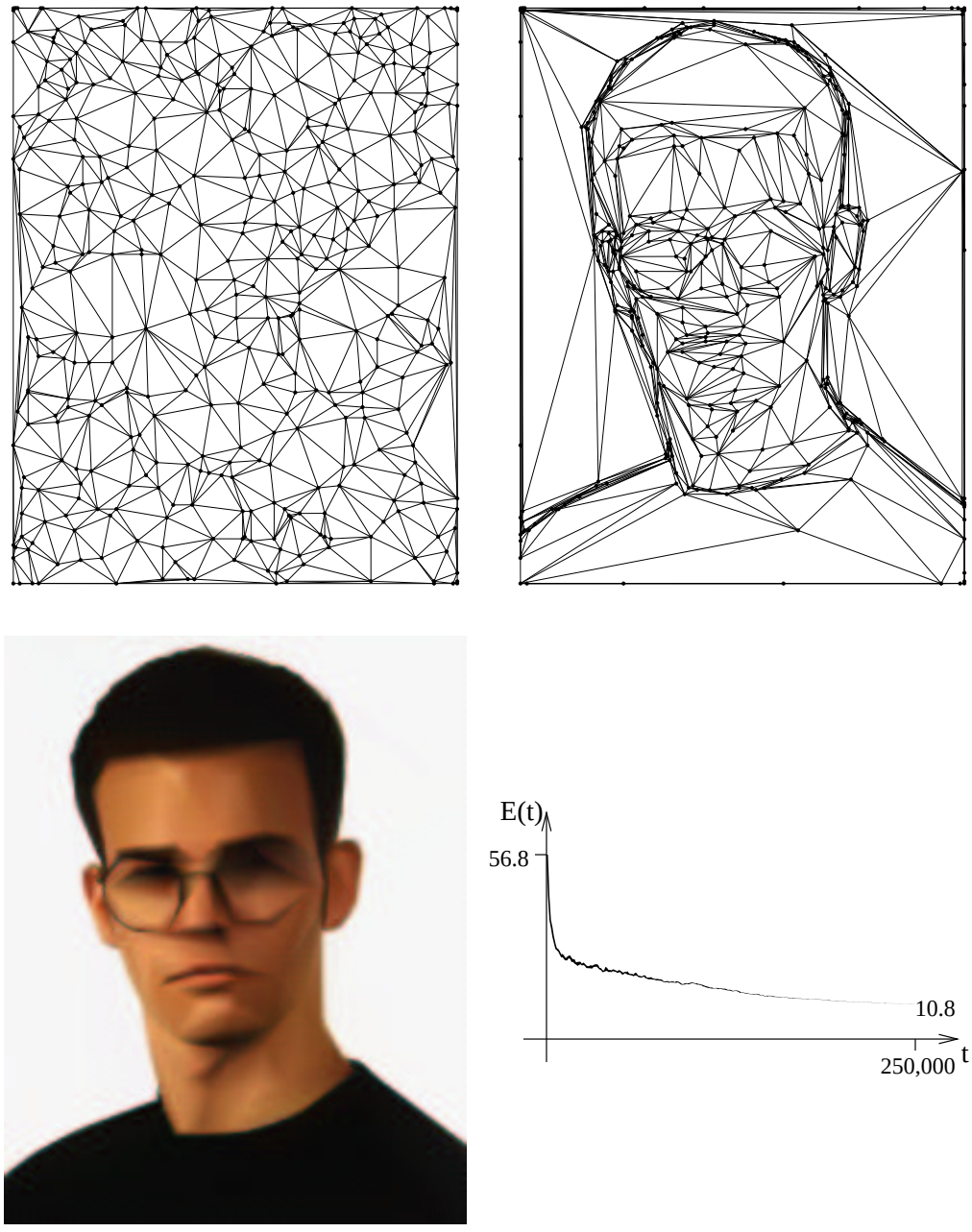

Abbildung 4.21: Experiment 16, Teil 1. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

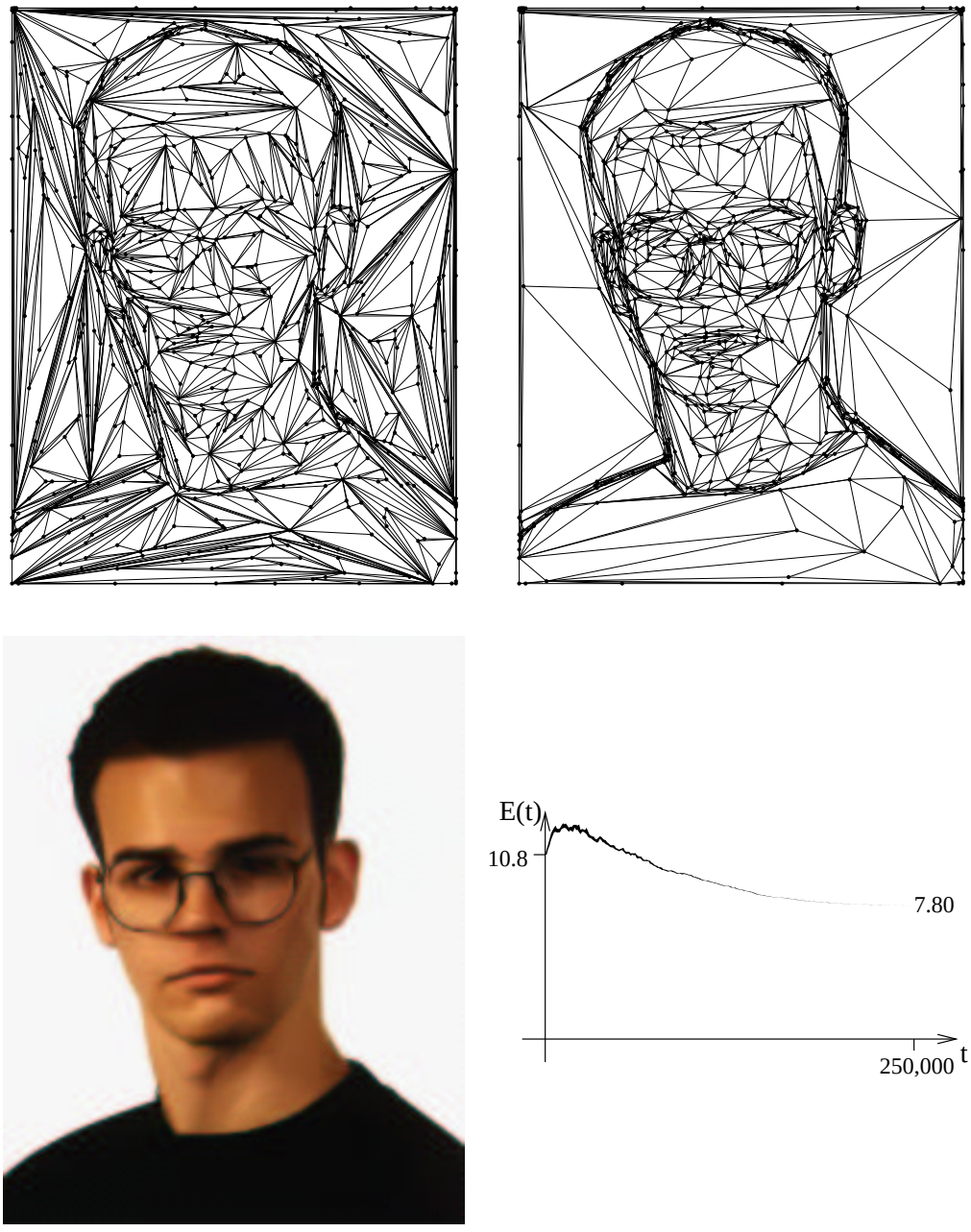

Abbildung 4.22: Experiment 16, Teil 2. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

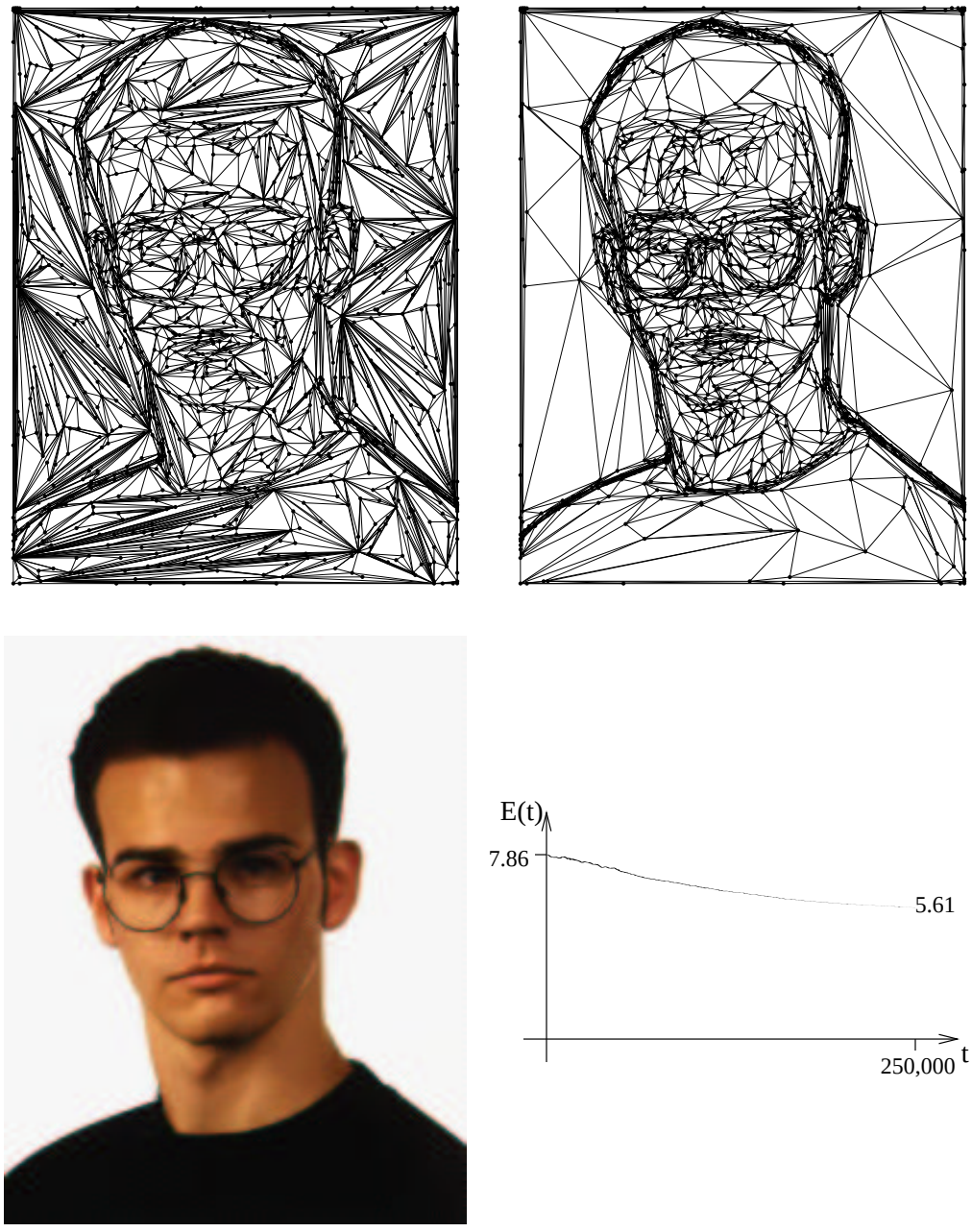

Abbildung 4.23: Experiment 16, Teil 3. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

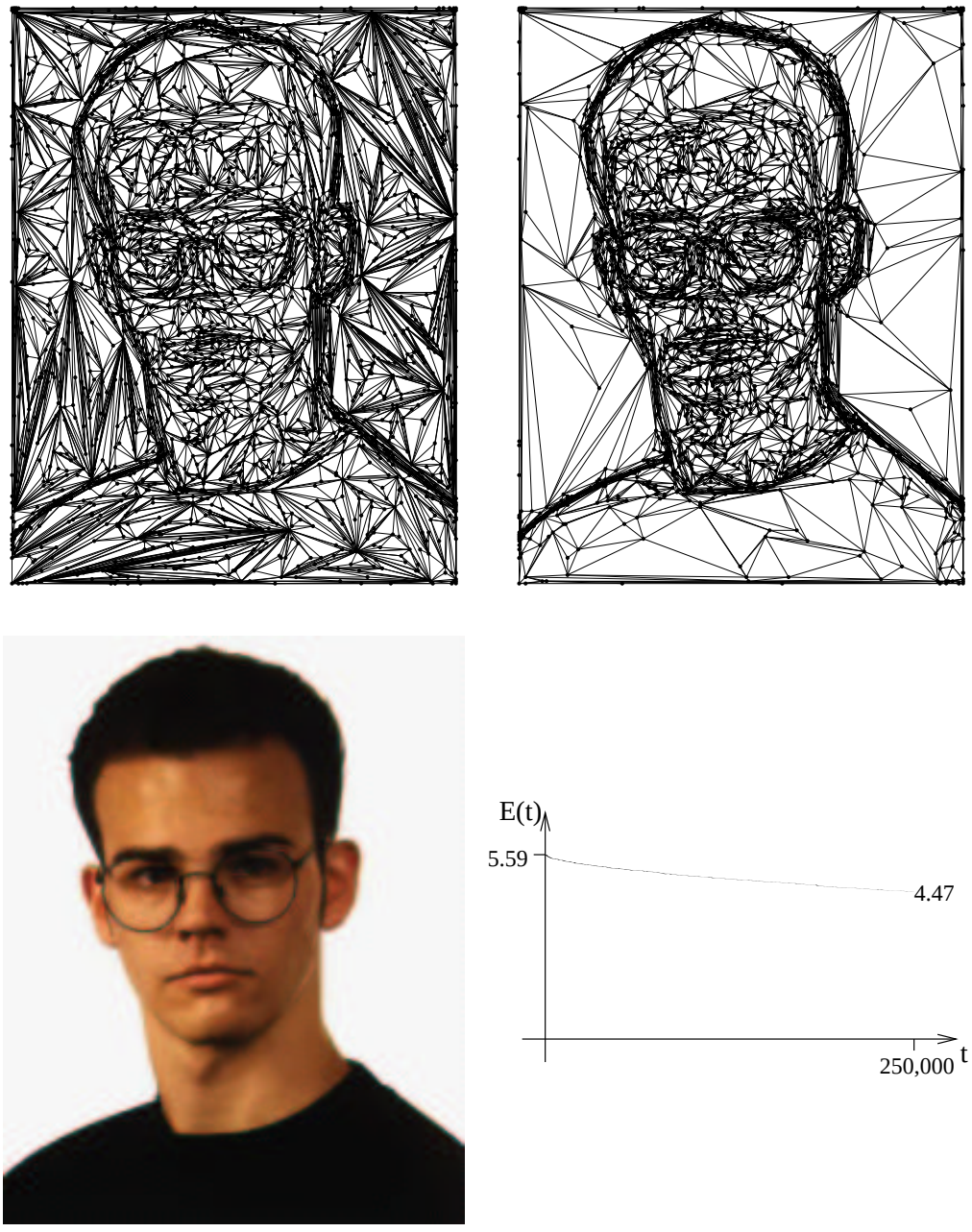

Abbildung 4.24: Experiment 16, Teil 4. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

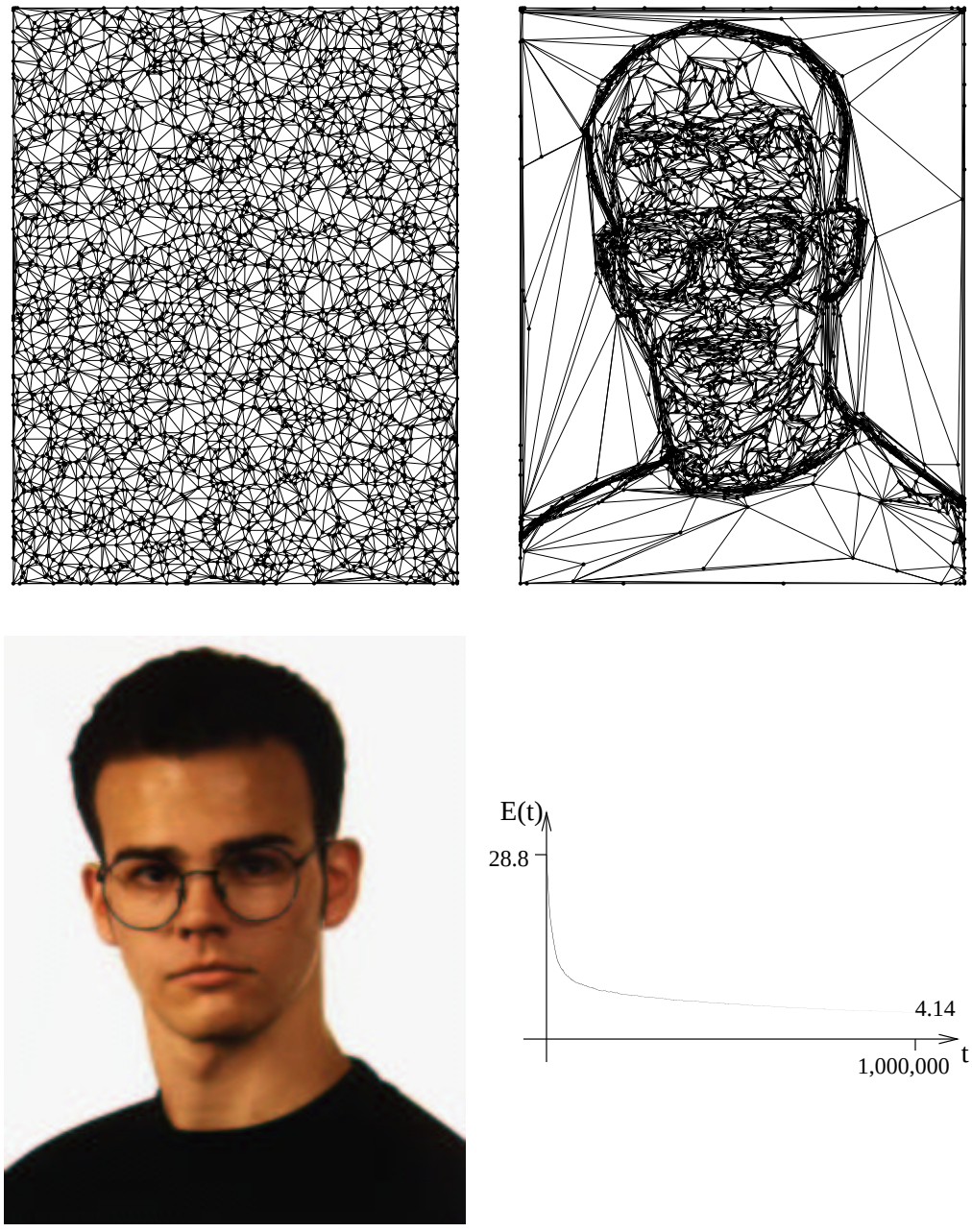

Abbildung 4.25: Experiment 16, Teil 5. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

Abbildung 4.26: „Lena“. Ausschnitt des Centerfolds der Playboy-Ausgabe von November 1972, abgetastet mit einer Auflösung von 256×256 Pixeln.

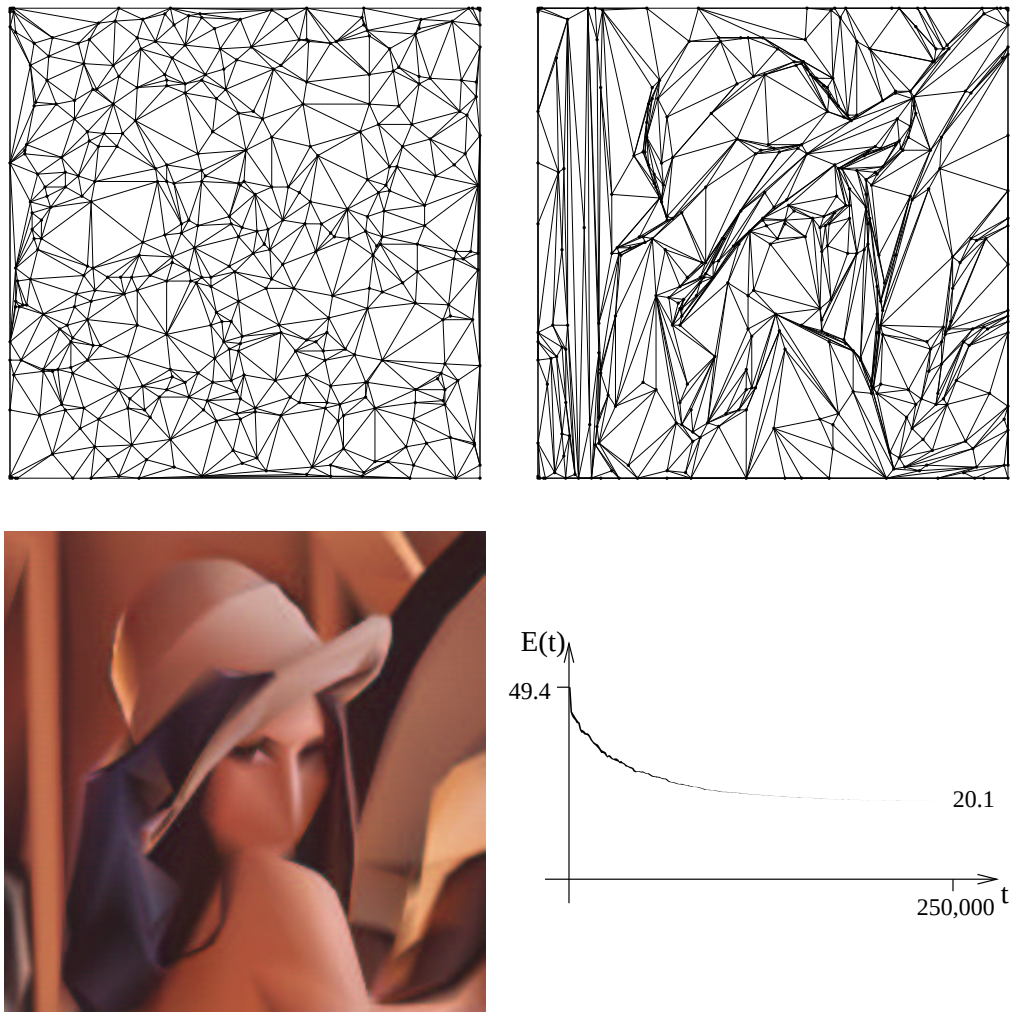

Abbildung 4.27: Experiment 17, Teil 1. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

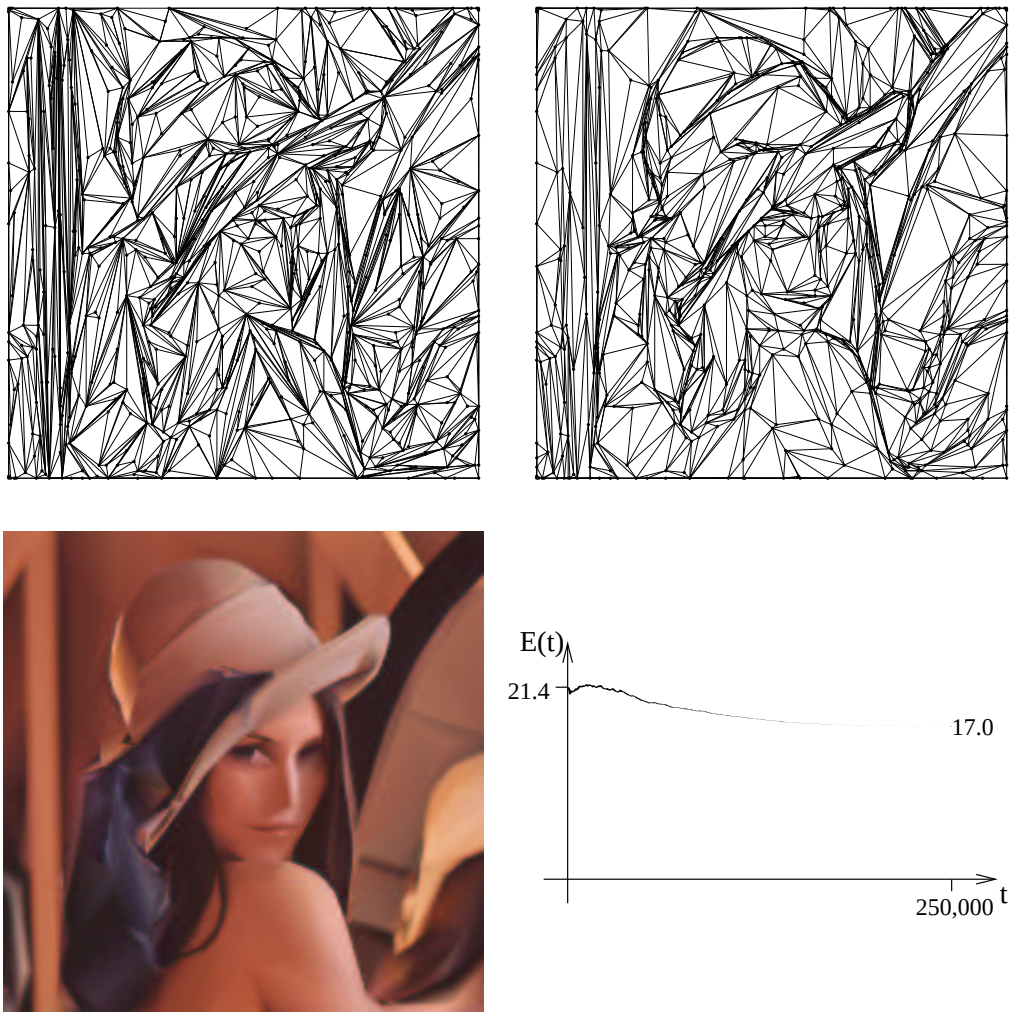

Abbildung 4.28: Experiment 17, Teil 2. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

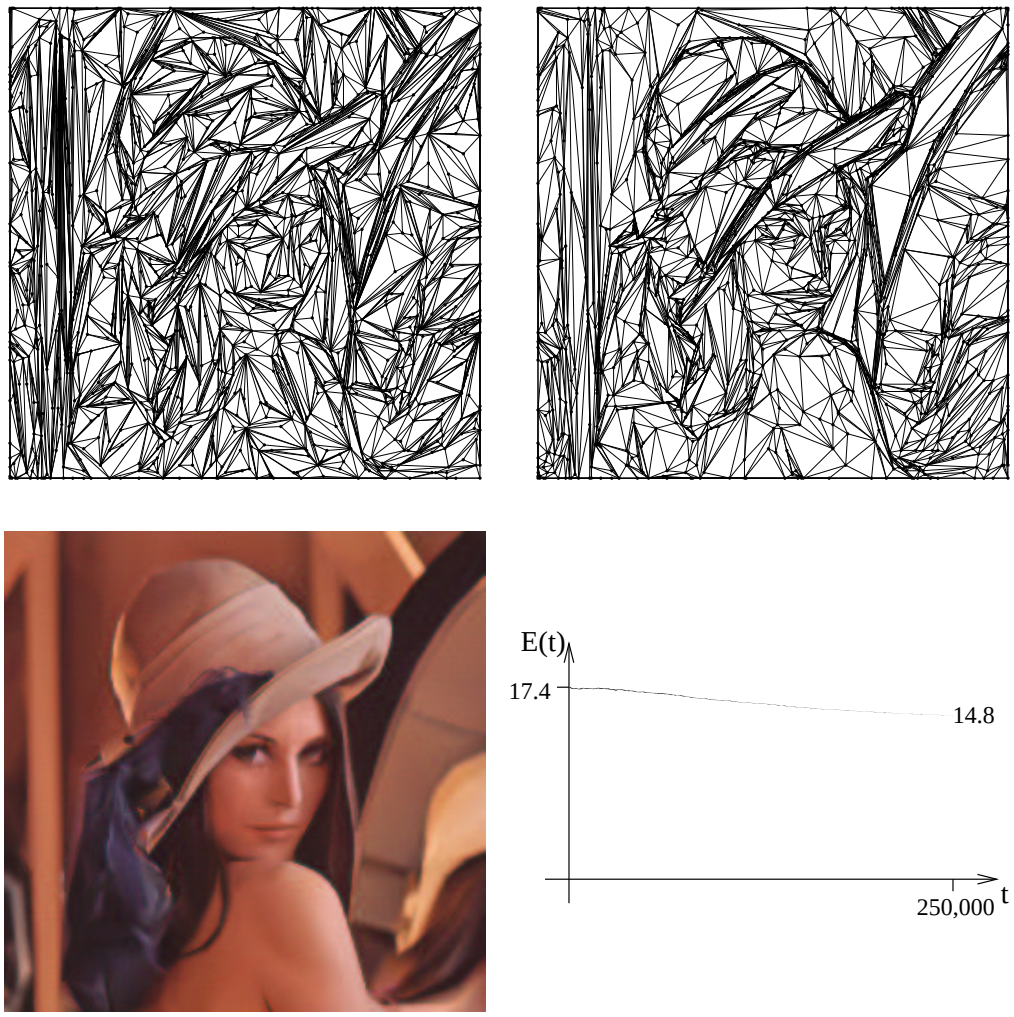

Abbildung 4.29: Experiment 17, Teil 3. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

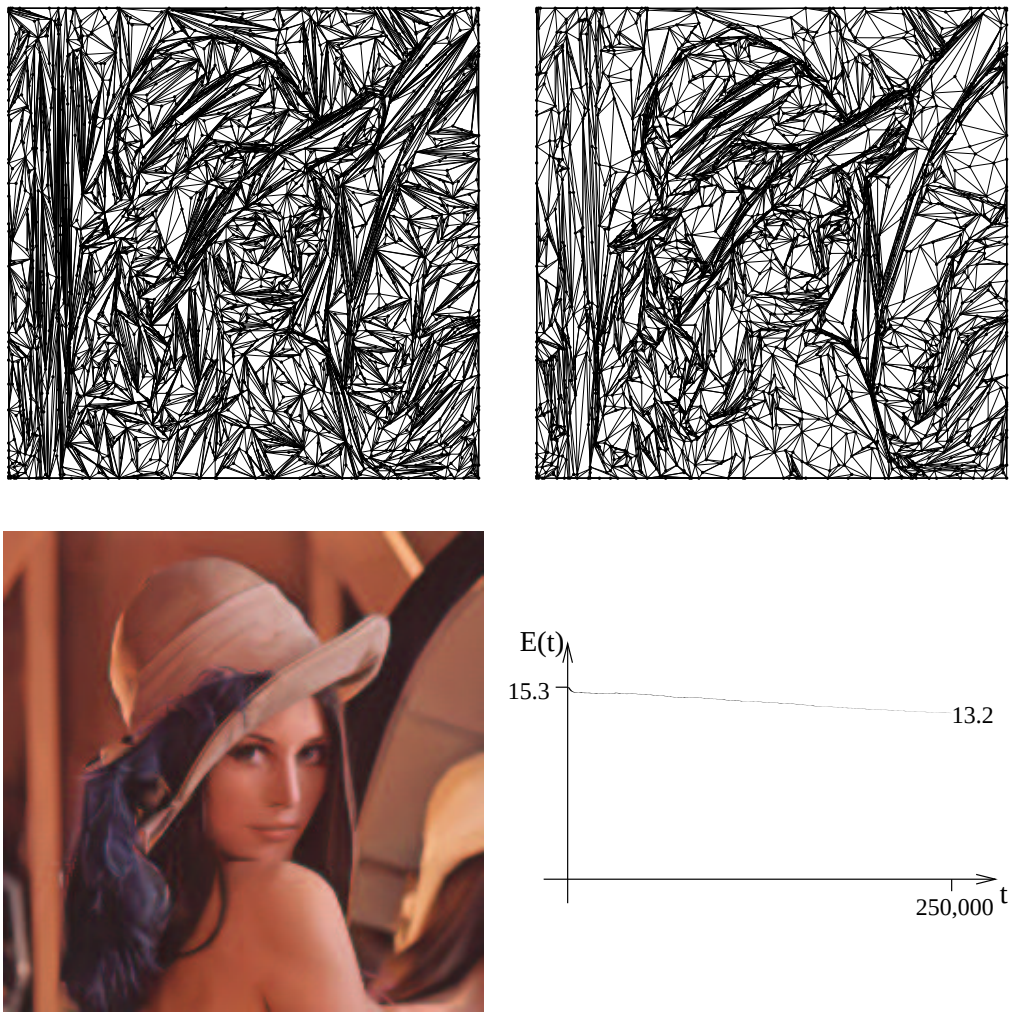

Abbildung 4.30: Experiment 17, Teil 4. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

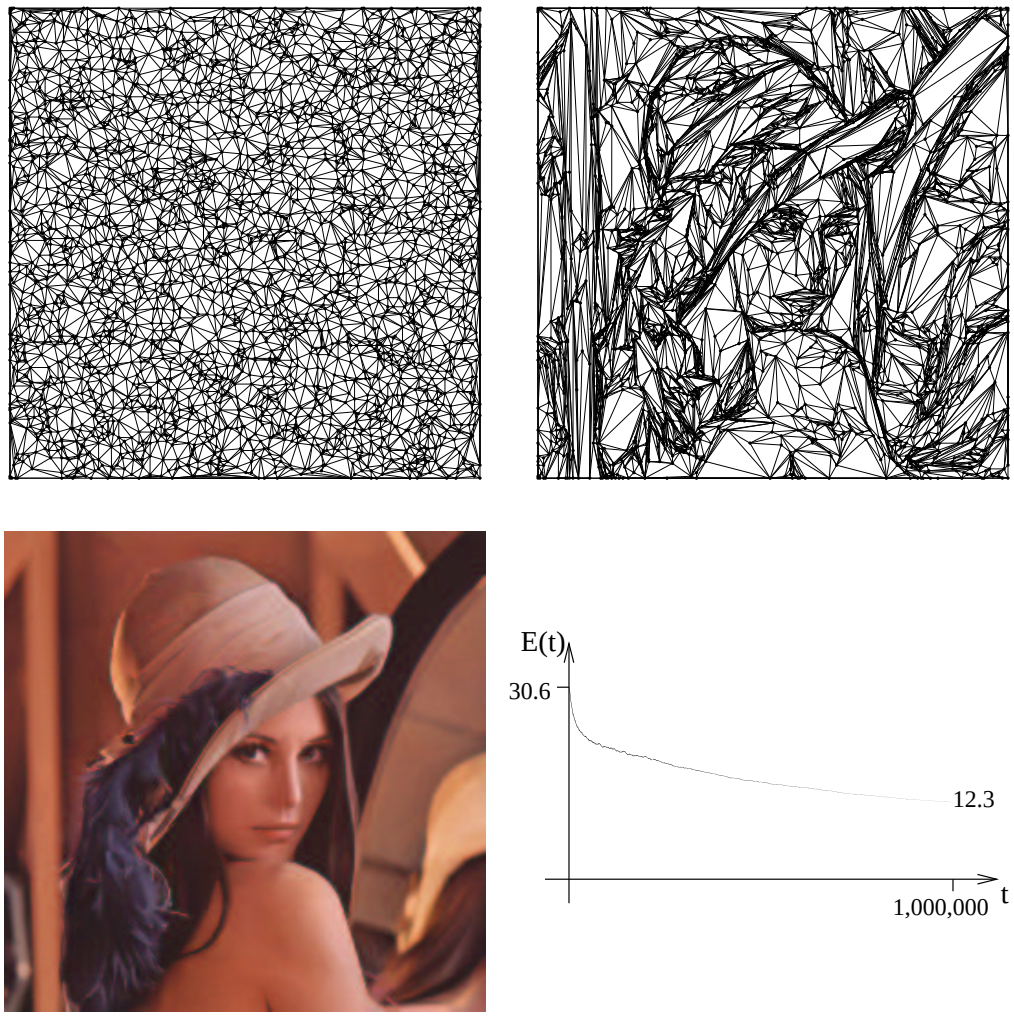

Abbildung 4.31: Experiment 17, Teil 5. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

Abbildung 4.32: „Golden Gate Bridge“. Fotografie des Wahrzeichens von San Francisco, abgetastet mit einer Auflösung von 329×222 Pixeln.

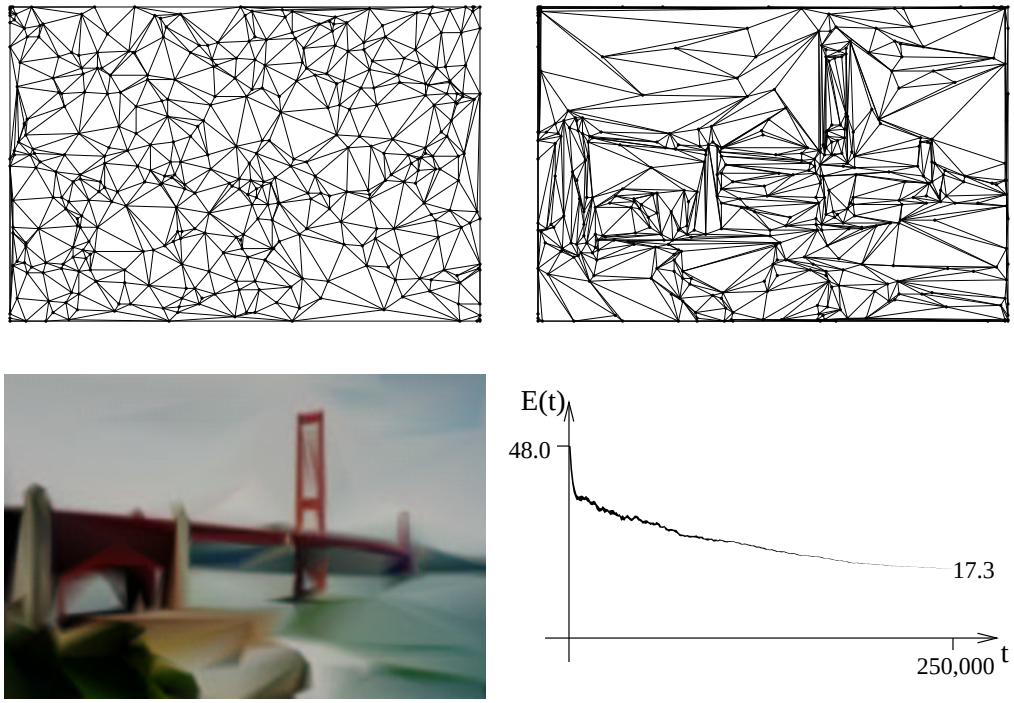

Abbildung 4.33: Experiment 18, Teil 1. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

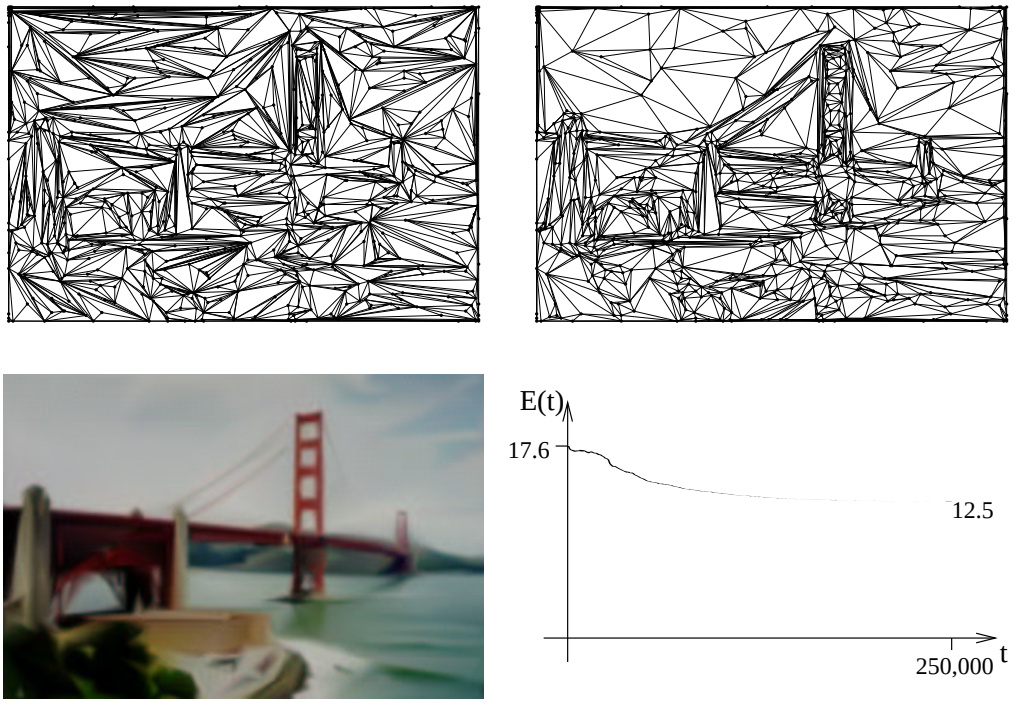

Abbildung 4.34: Experiment 18, Teil 2. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

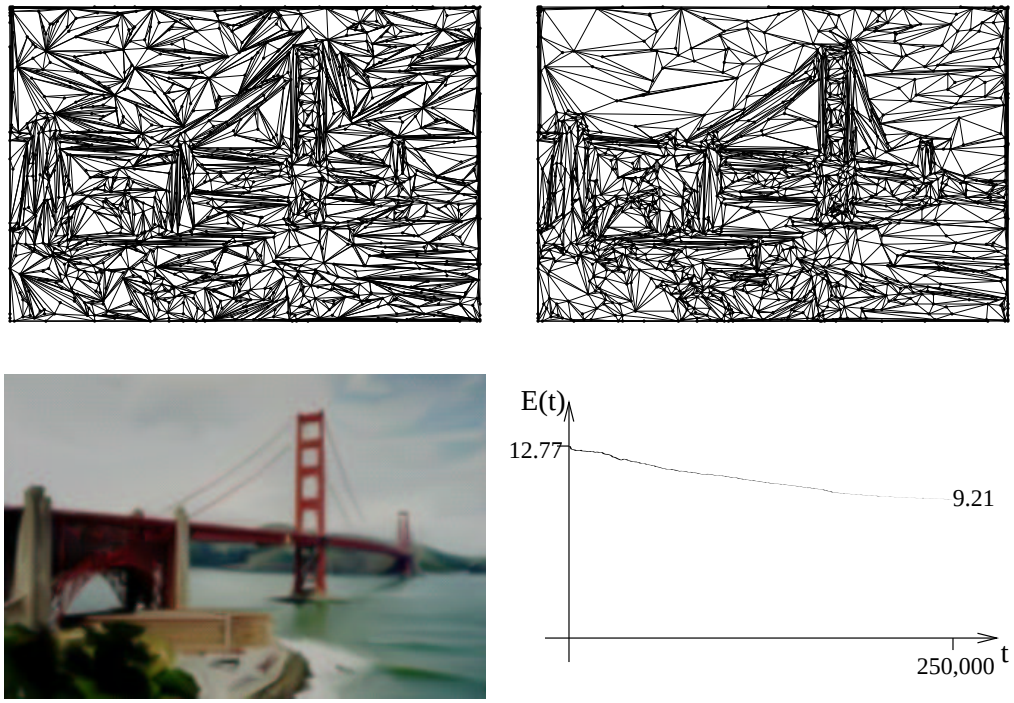

Abbildung 4.35: Experiment 18, Teil 3. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

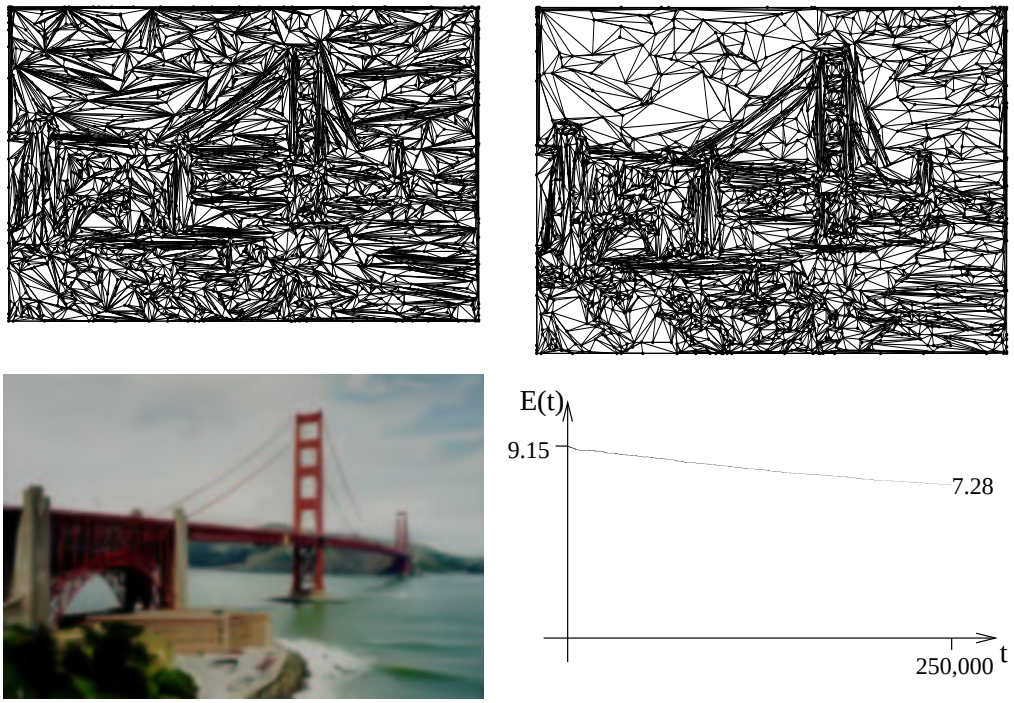

Abbildung 4.36: Experiment 18, Teil 4. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

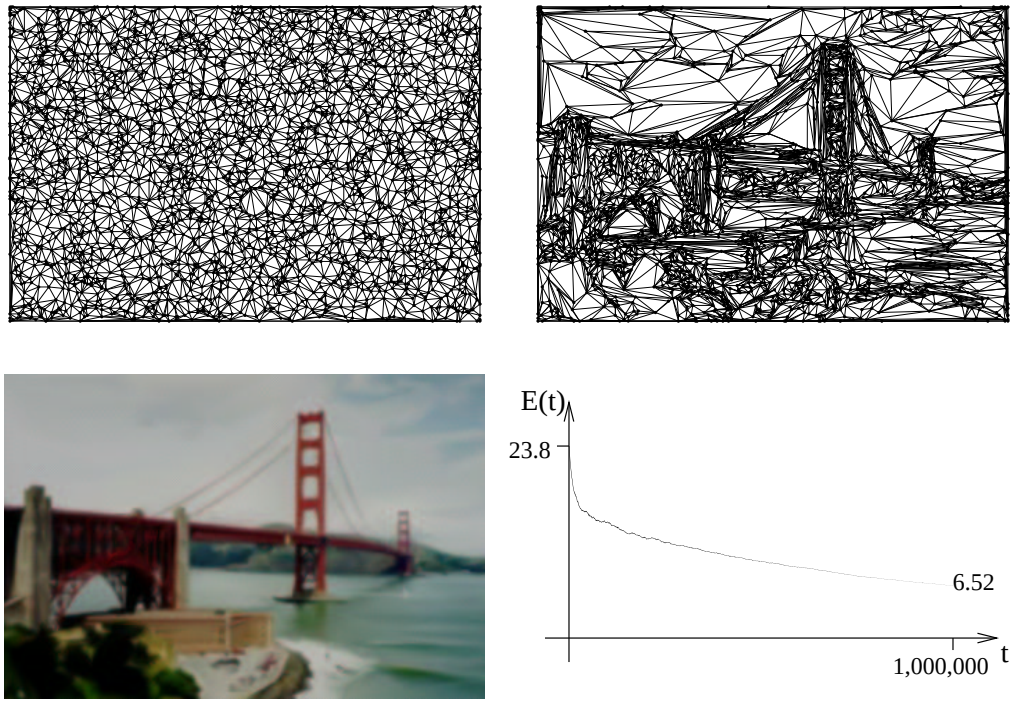

Abbildung 4.37: Experiment 18, Teil 5. Oben links: Anfangskonfiguration, oben rechts und unten links: Endkonfiguration, unten rechts: Abstandmaß über Zeit.

4.4 Heuristiken zur Parameterbestimmung

Die Beschreibung des Optimierungsalgorithmus in Kapitel 3 hat eine Reihe frei wählbarer Parameter aufgeführt, deren Werte den Verlauf des Algorithmus an einzelne Optimierungsprobleme anzupassen helfen. In diesem Abschnitt geben wir Heuristiken für die Bestimmung dieser Parameter an, die sich aus den in diesem Kapitel aufgeführten Experimenten ergeben haben.

- Der Parameter $p_0 \in (0,1)$ bestimmt die Wahrscheinlichkeit, mit dem ein erwartet schlechter Schritt zu Beginn der Iteration akzeptiert werden soll (siehe Abschnitt 3.2). In unseren Experimenten verwenden wir stets zuerst den Wert $p_0 = 1/2$.

- Der Parameter $f_t \in (0,1)$ bestimmt den Faktor, mit dem die aktuelle Temperatur des Simulated-Annealing-Verfahrens am Ende eines Temperaturschrittes multipliziert wird (siehe Abschnitt 3.3). Ein vernünftiger Wert ist $f_t = 0.95$. Ist die Zahl der Streudatenpunkte sehr groß und die Zahl der Kontrollpunkte des approximierenden linearen Splines eher gering, sollte f_t auf einen höheren Wert wie $f_t = 0.975$ gesetzt werden. Natürlich ist in diesem Fall eine höhere Zahl von Iterationen bis zur Konvergenz des Verfahrens einzukalkulieren.

- Der Parameter $f_g \in \mathbf{R}^+$ bestimmt den Schwellwert der Entscheidung, ob ein Vertex global oder lokal bewegt werden soll (siehe Abschnitt 3.4). In unseren Experimenten berechnen wir f_g als $f_g = 0.5/(N - |B|)$, wobei N die Anzahl der Kontrollpunkte des approximierenden linearen Splines und B die Menge der festen Kontrollpunkte ist. Ist eine Approximation Teil einer Hierarchie, sollte f_g im Verlauf der Berechnung der Hierarchieebenen langsam erhöht werden, etwa bis auf das 1.5-fache des ursprünglichen Wertes.

- Der Parameter $f_l \in \mathbf{R}^+$ begrenzt die Schrittweite eines Vertex bei lokaler Bewegung (siehe Abschnitt 3.4.3). In unseren Experimenten bestimmen wir zunächst die durchschnittliche Fläche A_P der Platelets in der initialen Konfiguration, und wählen dann f_l so, daß eine Bewegung eines Vertizes um den Radius des Kreises, der die Fläche A_P hat, mit der Wahrscheinlichkeit $1/2$ akzeptiert wird: $f_l = \sqrt{\frac{A_P}{\pi \ln 2}}$. Genau wie f_g sollte der Wert des Parameters f_l bei Erzeugung einer Approxima-

tionshierarchie langsam bis auf etwa das 1.5-fache des ursprünglichen Wertes erhöht werden.

- Der Parameter $p_r \in [0,1]$ bestimmt die Wahrscheinlichkeit, mit der im Iterationsschritt eine Kante rotiert wird (siehe Abschnitt 3.4). Da im univariaten Fall keine Kanten rotiert werden, ist dort stets $p_r = 0$. Im bivariaten Fall hat sich der konstante Wert $p_r = 0.15$ als vielversprechend erwiesen.

Kapitel 5

Bewertung und Ausblick

5.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit entwickelten wir ein Verfahren zur Berechnung optimaler Linearer-Spline-Approximationen an durch Streudatenmengen definierte skalar- oder vektorwertige Funktionen einer oder zweier Veränderlicher. Dieses Verfahren verwendet einen iterativen Optimierungsalgorithmus, der auf zufälliger Veränderung einer aktuellen Konfiguration und der globalen Strategie des Simulated-Annealing-Algorithmus basiert, und ist eine Verallgemeinerung des von Schumaker in [11] vorgestellten Algorithmus zur datenabhängigen Triangulierung. Wir beschrieben unseren Optimierungsalgorithmus, speziell den besonders wichtigen auf zufälligen Entscheidungen beruhenden Änderungsschritt, im Detail.

Anschließend wendeten wir den Algorithmus auf ausgewählte Testfälle univariater skalarwertiger, bivariater skalarwertiger und bivariater vektorwertiger Funktionen an und dokumentierten die Approximationsergebnisse. Im Rahmen dieser Experimente entwickelten wir auch Heuristiken zur Bestimmung der freien Parameter des Algorithmus.

5.2 Bewertung

Wir haben somit gezeigt, daß die von uns entwickelte Methode für univariate und bivariate skalarwertige Funktionen und für bivariate vektorwertige Funktionen gut funktioniert. Das Verfahren liefert gute Approximationen in vernünftiger Zeit, und in einfachen Fällen liefert es oft ein (zumindest der

Anschauung nach) global optimales Ergebnis. Wie zu erwarten war, ist die Leistung des Algorithmus in den bivariaten Fällen schlechter als im univariaten Fall. Diese Tatsache ist auf die wesentlich höhere Anzahl an Freiheitsgraden und die generell wesentlich höhere Komplexität der verwendeten approximierenden linearen Splines im bivariaten Fall zurückzuführen.

Trotz der angeführten Schwierigkeiten der bivariaten Fälle liefert unser Algorithmus erstaunlich gute Approximierungen an RGB-Farbbilder, sogar wenn nur eine kleine Anzahl von Kontrollpunkten verwendet wird. Da die Repräsentation eines Farbbildes durch einen linearen Spline auflösungsunabhängig ist, stellt unser Verfahren eine interessante Möglichkeit dar, Farbbilder in eine speicherplatz-effiziente, auflösungsunabhängige Darstellung zu transformieren.

5.3 Ausblick

Die Hauptgebiete zukünftiger Forschung sind die Generalisierung unseres Algorithmus für Funktionen dreier oder mehrerer Veränderlicher und für zeitveränderliche Funktionen, und die Anwendung des Algorithmus zur Bild- und Videokompression.

5.3.1 Höherdimensionale Funktionen

Wir haben keinen Grund zu vermuten, daß unser Algorithmus für höherdimensionale Funktionen prinzipiell nicht funktionieren würde. Die Probleme liegen in der Darstellung höherdimensionaler linearer Splines und in der Bestimmung eines angemessenen Iterationsschrittes. Im trivariaten Fall, wenn ein linearer Spline durch ein Tetraedernetz definiert ist, kommen als Basisoperationen zumindest globale oder lokale Bewegung eines Kontrollpunktes sowie „Rotation“ einer Fläche[1] oder „Rotation“ einer Kante[2] in Frage. Weiterhin gehen wir davon aus, daß die beim Übergang zu Funktionen mehrerer Veränderlicher unausweichliche Erhöhung der Anzahl der Freiheitsgrade die Effizienz des Verfahrens weiter verringert.

[1] Zwei Tetraeder, die eine Fläche gemeinsam haben, gehen über in drei Tetraeder, die eine Kante und jeweils zwei Flächen gemeinsam haben.

[2] Die inverse Operation einer Flächenrotation.

5.3.2 Zeitveränderliche Funktionen

Sogar die Anwendung des Algorithmus auf zeitveränderliche Funktionen ist denkbar: Man könnte eine zeitveränderliche Funktion von n Veränderlichen einerseits wie eine $(n+1)$-dimensionale Funktion auffassen; andererseits könnte man unabhängige n-dimensionale Approximierungen an die Funktion zu diskreten Zeitschritten $\{ n \cdot \Delta t \mid n = 0, 1, \dots \}$ bestimmen. Dabei kann man zeitliche Kohärenz zur Erhöhung der Geschwindigkeit und Qualität der Approximation ausnutzen: Nimmt man an, daß die zu approximierende Funktion sich zwischen zwei Zeitpunkten t und $t + \Delta t$ nur geringfügig ändert, dann stellt die endgültige Approximierung zum Zeitpunkt t eine gute initiale Approximierung für den Zeitpunkt $t + \Delta t$ dar.

5.3.3 Bildkompression

Zur Anwendung des Algorithmus als Bildkompressionsverfahren müssen Wege zur platzeffizienten Speicherung von linearen Splines gefunden werden. Ebenfalls zu untersuchen ist die Verwendung anderer Abstandsmaße bzw. anderer Wege, den Abstand eines Streudatenpunktes vom linearen Spline zu bestimmen. Momentan interpretieren wir Farben als RGB-Tripel und somit Vektoren des $\mathbf{R}^3$ und bestimmen den Abstand zweier Farbwerte c_1 und c_2 als den euklidischen Abstand $\|c_1 - c_2\|_2$ der zugehörigen Vektoren. Die Verwendung eines Farbmodells wie HSV oder YUV, das mehr der Physiologie des menschlichen Sehens entgegen kommt, könnte höhere Kompressionsraten bei gleichbleibender subjektiv empfundener Qualität erlauben. Die Verwendung spezieller Abstandsmaße könnte ebenfalls visuell ansprechendere Ergebnisse erzeugen oder besondere Effekte wie Kantenverstärkungen erlauben.

Um die Kompressionsergebnisse dieses Verfahrens mit anderen, pixelbasierten Verfahren wie JPEG vergleichen zu können, muß außerdem ein geeignetes Maß für den „Informationsgehalt“ eines Bildes gefunden werden. Da unser Algorithmus auflösungsunabhängig ist, hängt die Größe einer Approximation nicht von der Anzahl der Pixel, sondern nur von der Anzahl der Bilddetails ab. Wenn man ein Bild vor der Kompression mit diesem und einem pixelbasierten Verfahren vergrössert, kann man das pixelbasierte Verfahren um beliebige, aber nichts aussagende, Faktoren schlagen.

Ebenfalls interessant ist der Vergleich unseres Verfahrens mit der Bildkompression durch affine Transformationen, auch als „Fraktale Bildkompression“ bekannt (siehe [18]). Die Gemeinsamkeit zu diesem Verfahren ist der

Übergang von einer pixelorientierten Bilddarstellung zu einer Repräsentation als Kombination von Basisfunktionen. In unserem Fall sind die Basisfunktionen stets Dreiecke im Bildraum, also vergleichsweise einfach; bei der Fraktalen Bildkompression handelt es sich um iterierte Funktionensysteme. Aufgrund der größeren Menge von Basisfunktionen ist das letztgenannte Verfahren prinzipiell leistungsfähiger. Die Bestimmung guter Basisfunktionen für ein gegebenes Bild ist jedoch ein schwieriges Problem, so daß die Ergebnisse des Verfahrens in der Regel nicht so gut sind wie die Theorie erwarten liesse.

5.3.4 Videokompression

Die oben getroffenen Bemerkungen über die Approximierung zeitvariierender Funktionen gelten ebenso für Videoströme. Besonders die für Telekonferenz-Anwendungen typischen Videoströme („Talking Heads“) zeigen starke temporale Kohärenz; unser Algorithmus könnte somit zu einer Echtzeit-Kompressionsmethode für diese Art von Videoströmen führen. Die Auflösungsunabhängigkeit der Darstellung ist ein Vorteil bei der großflächigen Projektion von Videoströmen, und die Asymmetrie des Verfahrens ist für Multicast-Anwendungen von Vorteil. Obwohl die Approximierung von Bildern in Echtzeit auch für schnelle Rechner eine Herausforderung ist, kann die Reproduktion der Linearen-Spline-Darstellung bereits von auf dem Massenmarkt verfügbarer Videohardware problemlos in Echtzeit erledigt werden.

Uns schwebt ein Verfahren vor, bei dem pro Videobild nur wenige Schritte des Optimierungsverfahrens ausgeführt werden und bei dem die resultierende Approximation für ein Bild als Startwert für die Approximation des nächsten Bildes verwendet wird. Zeigt der Videostrom starke temporale Kohärenz, wird das Verfahren nach einigen Bildern eine gute Approximation liefern und diese beibehalten; Schnitte oder plötzliche Änderungen des Bildinhalts werden ebenfalls nach einigen Bildern wieder aufgefangen. Ein solcher Algorithmus skaliert gut, da die Anzahl der Iterationsschritte pro Bild durch die Geschwindigkeit der Kompressionshardware bestimmt wird; eine höhere Anzahl von Schritten bedingt eine höhere Qualität der Approximierung und ein schnelleres Reagieren auf plötzliche Änderungen. Experimente haben ergeben, daß sogar eine völlig zufällige Anfangskonfiguration bereits den Inhalt des approximierten Bildes erkennen läßt; wir vermuten daher, daß der beschriebene Algorithmus funktionieren wird und eventuell eine interessante Alternative zu herkömmlichen, pixel-orientierten Verfahren darstellen könnte.

Literaturverzeichnis

[1] Bonneau, G.-P., Hahmann, S. and Nielson, G.M., *BLaC-Wavelets: A Multiresolution Analysis with Non-nested Spaces*, in Yagel, R. and Nielson, G.M., eds., Visualization '96 (1996), IEEE Computer Society Press, Los Alamitos, Ca, pp. 43–48

[2] Eck, M., DeRose, A.D., Duchamp, T., Hoppe, H., Lounsbery, M. and Stuetzle, W., *Multiresolution Analysis of Arbitrary Meshes*, in Cook, R., ed., Proc. SIGGRAPH 1995, ACM Press, New York, NY, pp. 173–182

[3] Gieng, T.S., Hamann, B., Joy, K.I., Schussman, G.L. and Trotts, I.J., *Constructing Hierarchies for Triangle Meshes*, in IEEE Transactions on Visualization and Computer Graphics 4(2) (1998), pp. 145–161

[4] Hamann, B., *A Data Reduction Scheme for Triangulated Surfaces*, in Computer Aided Geometric Design 11(2) (1994), pp. 197–214

[5] Hamann, B., Jordan, B.J. and Wiley, D.A., *On a Construction of a Hierarchy of Best Linear Spline Approximations Using Repeated Bisection*, in IEEE Transactions on Visualization and Computer Graphics 4(4) (1998)

[6] Davis, P.J., *Interpolation and Approximation* (1975), Dover Publications, Inc., New York, NY

[7] Delaunay, B. *Sur la sphere vide*, in Otdelenie Matematicheskii i Estestvennyka Nauk 7 (1934), Izv. Akad. Nauk SSSR, pp. 793–800

[8] de Berg, M., van Kreveld, M., Overmars, M. and Schwarzkopf, O., *Computational Geometry* (1990), Springer-Verlag, New York, NY

[9] Edelsbrunner, H., *Algorithms in Combinatorial Geometry* (1987), Springer-Verlag, New York, NY

[10] Preparata, F.P., Shamos, M.I., *Computational Geometry*, third printing (1990), Springer-Verlag, New York, NY

[11] Schumaker, L.L. *Computing Optimal Triangulations Using Simulated Annealing*, in Computer Aided Geometric Design 10 (1993), pp. 329–345

[12] Nielson, G.M., *Scattered Data Modeling*, in IEEE Computer Graphics and Applications 13(1) (1993), pp. 60–70

[13] Hämmerlin, G. and Hoffmann, K.-H., *Numerical Mathematics* (1991), Springer-Verlag, New York, NY

[14] Boehm, W. and Prautzsch, H., *Numerical Methods* (1993), AK Peters Ltd., Wellesley, MA

[15] Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. *Numerical Recipes in C*, 2nd ed. (1992), Cambridge University Press, Cambridge, MA

[16] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E., in Journal of Chemical Physics 21 (1953), pp. 1087–1092

[17] Guibas, L.J., Knuth, D.E., and Sharir, M. *Randomized Incremental Construction of Delaunay and Voronoï Diagrams*, in Proc. 17th Int. Colloq.—Automata, Languages and Programming, vol. 443 of Springer Verlag LNCS (1990), Springer Verlag, Berlin, pp. 414–431

[18] Barnsley, M.F. and Hurd, L.P., *Fractal Image Compression* (1993), AK Peters Ltd., Wellesley, MA