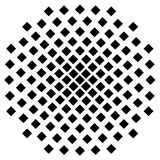
Universität Stuttgart
Fakultät 8: Mathematik und Physik
Fachbereich Mathematik, IMNG

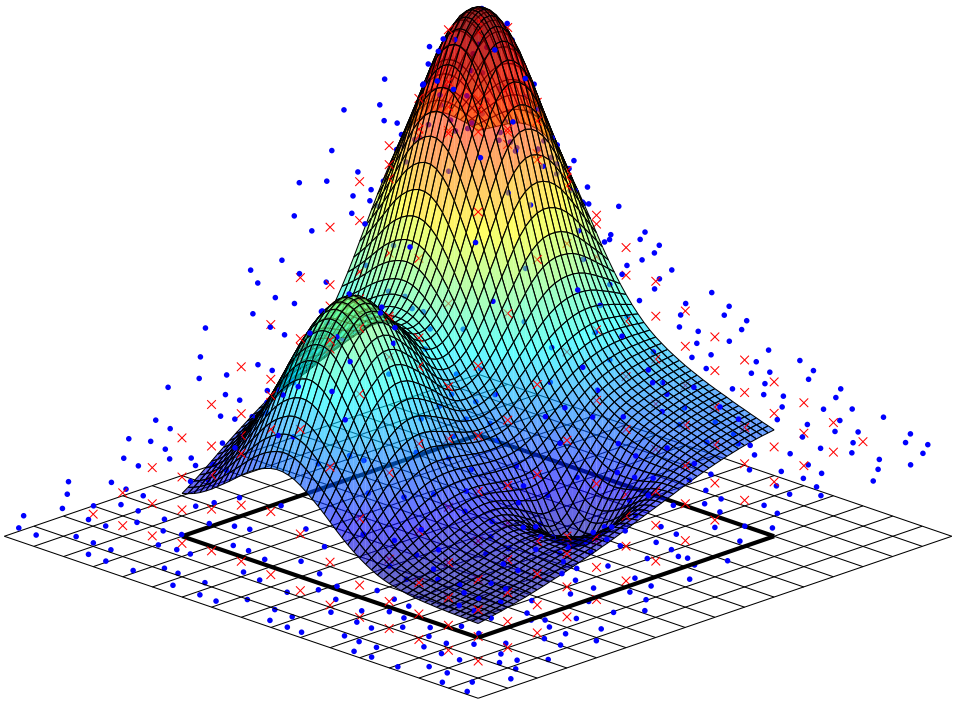
Sommersemester 2012



Bachelorarbeit

Spline-Approximation unregelmäßig verteilter Daten

Julian Valentin



Betreuer: Prof. Dr. Klaus Höllig
Datum der Abgabe: 14. August 2012

Inhaltsverzeichnis

Vorwort

Als Erstes bedanke ich mich bei meinem Betreuer, Professor Dr. Klaus Höllig, für die Unterstützung in der Schaffensphase dieser Arbeit. Zusätzlich danke ich ihm für die Vorlesungen, die mir stets gezeigt haben, was für eine schöne Wissenschaft Mathematik sein kann.

Außerdem geht ein Dankeschön an Herrn Jörg Hörner, der mir seine Funktion für die Spline-Auswertung `spl_eval` und eine Version des sehr intuitiv bedienbaren `splineplotter`s hat zukommen lassen.

Ein Dank geht auch an die vielen Menschen, die zum Beispiel mit der Entwicklung von LaTeX-Paketen (und natürlich LaTeX selbst) die Erstellung einer ästhetischen Bachelorarbeit von der technischen Seite her fast schon zum Kinderspiel gemacht haben.

Last but not least bedanke ich mich bei meiner Familie für den Beistand und die Rücksicht, nicht nur während der letzten paar Monate.

Bietigheim-Bissingen, im August 2012
Julian Valentin

1 Einleitung

Nehmen wir an, wir wollen ein dreidimensionales Geländemodell der uns umgebenden Gegend erstellen, so wie es zum Beispiel im bekannten Computerprogramm *Google Earth* zu sehen ist. Wenn wir nicht gerade einen Satelliten o. Ä. zur Verfügung haben, können wir bspw. an verschiedenen, einzelnen Punkten der Erdoberfläche die Höhe über dem Meeresspiegel messen, etwa mittels eines GPS-Empfängers. Da wir aus Gründen des Aufwands nicht an beliebig vielen Punkten (die beliebig dicht beieinanderliegen) messen können, müssen wir die sich ergebenden, diskreten Punkte irgendwie zu einer kontinuierlichen Fläche „verbinden".

Genau das ist das Problem bei der Approximation unregelmäßig verteilter Daten (engl. *scattered data*): Es seien m paarweise verschiedene Punkte $x_i \in \mathbb{R}^d$ mit zugehörigen Daten f_i $(i = 1, \ldots, m)$ gegeben (in unserem Beispiel wäre $d = 2$). Gesucht ist eine bivariate, reellwertige Funktion f aus einem bestimmten Funktionenraum, so dass f die gegebenen Daten in einem gewissen Sinne „bestmöglich" approximiert. Ein ähnliches Problem ist die Interpolation, bei der sogar $f(x_i) = f_i$ für $i = 1, \ldots, m$ gelten soll.

Die Daten heißen „unregelmäßig verteilt", weil wir im Allgemeinen nicht annehmen können, dass diese auf einem regulären Gitter liegen. Im obigen Beispiel können wir evtl. nicht immer genau an Punkten eines Gitters messen, weil einige Stellen unerreichbar sind (etwa in Gebäuden). Liegen die Daten auf einem regelmäßigen Gitter, so ist die Approximation erheblich einfacher, was man schon daran erkennt, dass der in dieser Arbeit vorgestellte Algorithmus hauptsächlich damit beschäftigt ist, durch lokale Polynom-Approximation Gitterdaten aus den unregelmäßig verteilten Daten zu generieren.

Die Erstellung von sog. DEM (digitalen Geländemodellen, engl. *digital elevation models*) ist nicht die einzige Anwendung der Approximation unregelmäßig verteilter Daten. In [1] werden weitere Anwendungsmöglichkeiten aufgezählt:

- Das Gravitationsfeld der Erde ist zwar schon an vielen verschiedenen Punkten der Erde exakt vermessen worden, aber es gibt kein allgemeines physikalisches Modell, das für die meisten Anwendungen hinreichend genau wäre.

- Eine notwendige Farbkorrektur bei der Verarbeitung von Farbfilmen kann für bestimmte Testfarben angemessen bestimmt werden. Farbkorrekturen für die anderen Farben können dann durch Interpolation errechnet werden.

- Bei der Öl-Prospektion (Lagerstätten-Erkundung) sind viele Daten nahe von Test-Bohrlöchern vorhanden. Sonst sind die Daten sehr dünn und damit ziemlich unregelmäßig verteilt.

Unregelmäßig verteilte Daten treten im Prinzip überall dort auf, wo eine Messgröße nur an unregelmäßigen, diskreten Punkten genau bestimmt werden kann, etwa weil sie wegen ihrer Kompliziertheit allgemein nicht zu beschreiben ist (oder weil man eine exakte Beschreibung gar nicht kennt). Wertet man eine Approximation von unregelmäßig verteilten Daten an

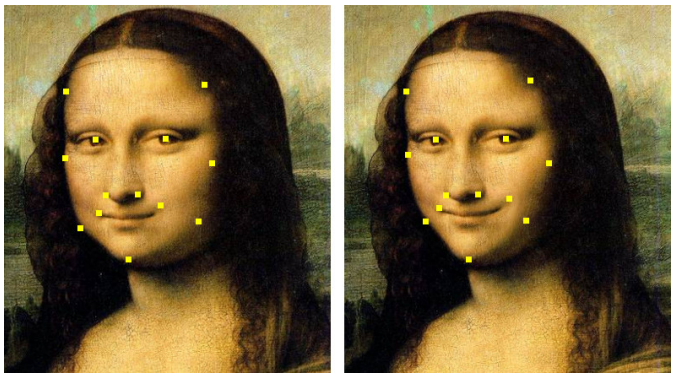


Abbildung 1.1: Bilddeformation mit MLS (Quelle: [22])

denselben Datenpunkten aus, erhält man ein einfaches Glättungsverfahren, was eine weitere (sehr allgemeine) Anwendung darstellt.

Des Weiteren ist laut [18] die weiter unten vorgestellte Methode von LS (bzw. von WLS/MLS) für die Approximation unregelmäßig verteilter Daten in der Computergrafik weit verbreitet: Ein Anwendungsbereich wäre die Deformation von digitalen Bildern mittels der Interpolation unregelmäßig verteilter Daten, wobei jede der vorhandenen (allgemeinen) Methoden angewendet werden kann. Grafische Beispiele für Methoden, die auf Triangulierungen, inversen Distanzwichtungen oder radialen Basisfunktionen basieren, finden sich in [20] – man kann aber auch die Methode der MLS anwenden: In [22] wird damit beispielhaft das berühmte Gesicht der Mona Lisa zum Lächeln gebracht und ein wenig schlanker gemacht (siehe Abbildung 1.1).

Viele Verfahren zur Approximation bzw. Interpolation unregelmäßig verteilter Daten existieren bereits, von denen wir natürlich nicht alle hier nennen können:

Bei der Methode der **kleinsten Quadrate** (*least squares*, LS) minimiert man das Fehlerfunktional $E_{\mathrm{LS}}(f) := \sum_i |f(x_i) - f_i|^2$ über einen gewissen Funktionenraum, z. B. d-variate Polynome oder Splines mit einem bestimmten Grad. Durch Wahl eines festen Punkts $x^* \in \mathbb{R}^d$ und Multiplikation mit einer Gewichtsfunktion φ, einer sog. radialen Basisfunktion (zum Beispiel $\varphi(r) = \exp(-r^2/h^2)$ mit $h > 0$), erhält man die gewichtete Methode der kleinsten Quadrate (WLS) mit dem Funktional $E_{\mathrm{WLS}}(f) := \sum_i \varphi(\|x^* - x_i\|)|f(x_i) - f_i|^2$. Lässt man den Punkt x^* variieren, so erhält man die Methode der *moving least squares* (MLS). Eine kurze Einführung in LS/WLS/MLS ist in [18] zu finden.

Die Methode der **radialen Basisfunktionen** (RBF) ist etwas allgemeiner als die WLS-Methode und verwendet nur eine Linearkombination $f(x) = \sum_i \lambda_i \varphi(\|x - x_i\|)$ von Gewichtsfunktionen (siehe [5]).

Eine weitere Methode erfordert die Erstellung einer **geeigneten Triangulierung** (z. B. mit der Delaunay-Triangulation), also die Vernetzung der Datenpunkte x_i zu Dreiecken, und verwendet dann Finite Elemente zur Approximation ([1]). Im einfachsten Fall wird zwischen den Eckpunkten der Dreiecke linear interpoliert.

Die Methode der **inversen Distanzwichtung** (IDW), die nach einer Arbeit [26] von Donald Shepard im englischen Sprachraum auch *Shepard's method* heißt, geht ähnlich vor wie die RBF-Methode und schreibt die Approximation als Linearkombination $f(x) = \sum_i f_i w_i(x)$ von Gewichtsfunktionen $w_i(x) = \|x - x_i\|^{-p} / \sum_j \|x_j - x_i\|^{-p}$ (siehe [1, 9]).

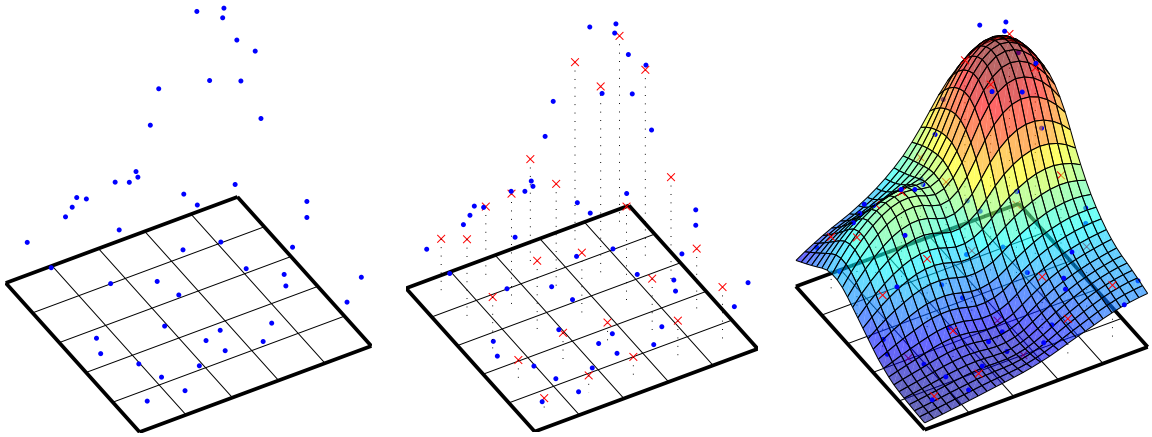


Abbildung 1.2: Algorithmus zur Spline-Approximation unregelmäßig verteilter Daten

Schließlich ist noch die etwas kompliziertere Methode des **Kriging** zu erwähnen. Sie wurde durch Georges Matheron (z. B. [17]) entwickelt, der durch die Masterarbeit [16] des Geostatistikers Danie G. Krige inspiriert wurde, der seinerseits die Verfügbarkeit von Bodenschätzen anhand von empirischen Daten untersuchte (siehe [7] zur Geschichte des Kriging), und berücksichtigt die unterschiedliche räumliche Varianz bei der unregelmäßigen Verteilung der Datenpunkte.

Ein Vergleich von einigen etablierten Algorithmen für den Fall der Interpolation findet sich in [9] und in [1].

Die in dieser Arbeit vorgestellte Methode besteht aus zwei Teilen (siehe Abbildung 1.2): Zunächst werden aus den unregelmäßig verteilten Daten per lokaler Polynom-Approximation Daten auf Punkten erzeugt, die auf einem regelmäßigen Gitter liegen. Anschließend werden per Quasi-Interpolation die Koeffizienten eines Quasi-Interpolanten ermittelt, der hier eine Linearkombination

$$Qf = \sum_{k \sim D} (Q_k f) b^n_{k,h}$$

von bivariaten B-Splines $b^n_{k,h}$ mit äquidistanten Knoten ist. Die Darstellung als Linearkombination von B-Splines auf regulären Gittern hat dabei mehrere Vorteile: Die Beschreibung ist einfach (im Prinzip gibt es für jeden Grad nur eine Basisfunktion $b^n = b^n_{0,1}$, die skaliert und verschoben wird) und es existieren einfache und sehr effektive Algorithmen u. a. zur Auswertung und Ableitung des Splines. Der so entstehende Algorithmus erlaubt eine dynamische Einstellung des Approximationsgrades und benötigt keine Triangulierung der Datenpunkte. Außerdem kann dieser leicht auf verschiedene Grade in den Koordinaten (was bei partiellen Ableitungen von Splines vorkommt) und höhere Dimensionen verallgemeinert werden.

Aufbau der Arbeit

In dieser Arbeit beschäftigen wir uns zunächst in Kapitel 2 kurz mit den für das weitere Verständnis wichtigen Definitionen und Sätzen bezüglich bivariaten Polynomen, B-Splines und Spline-Funktionen.

Anschließend werden wir uns in Kapitel 3 mit der Methode der Quasi-Interpolation in zwei Variablen vertraut machen und die Schwierigkeiten von bivariater Polynom-Approximation erklären. Am Ende dieses Kapitels werden wir diese zwei „Bausteine“ zu dem in dieser Arbeit vorgestellten Algorithmus zusammensetzen.

Dessen MATLAB-Implementierung werden wir in Kapitel 4 besprechen. Dabei gehen wir ausführlich auf die Funktionsweise des Algorithmus ein und werden diese schrittweise anhand eines Beispiels erklären. Zusätzlich werden wir ein paar numerische Eigenschaften des Algorithmus analysieren.

In Kapitel 5 werden wir schließlich den Algorithmus als beispielhafte Anwendung verwenden, um Gewichtsfunktionen für Finite Elemente zu konstruieren.

Anhang A enthält eine Liste der auf der dieser Arbeit als Anhang beiliegenden CD-ROM vorhandenen Dateien.

Notation

Definitionen, Beispiele, Algorithmen und Sätze sind innerhalb jedes Kapitels durchgehend nummeriert, wobei Beweise und nicht-bewiesene Sätze mit $\square$ beendet werden. Definitionen, Beispiele und Algorithmen enden mit $\lozenge$.

Wir schreiben $\mathbb{N} := \{1, 2, 3, \dots\}$ für die natürlichen Zahlen. Wenn wir die Null explizit mit einschließen wollen, schreiben wir $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$.

Ein Gebiet $D \subset \mathbb{R}^m$ ist eine nicht-leere, zusammenhängende, offene Teilmenge von $\mathbb{R}^m$. Die Menge $\mathcal{C}(D)$ ist die Menge aller stetigen Funktionen auf D. Landau-Notation erfolgt mit der $\mathcal{O}$-Schreibweise.

Wenn es die Umstände zulassen und der entstehende Ausdruck immer noch eindeutig ist, lassen wir manchmal unwichtige Indizes weg, zum Beispiel $b_k := b^n_{k,h}$. Falls wir einen Skalar an die Stelle schreiben, an der nach Definition eigentlich ein Vektor stehen müsste, meinen wir einen Vektor der passenden Größe, der nur den Skalar enthält, also z. B.

$$b^n_{(k_1,k_2),h} := b^{(n,n)}_{(k_1,k_2),(h,h)}, \quad h > 0,\ n \in \mathbb{N}.$$

Um die explizite Angabe von meist komplizierten Indexmengen (beispielsweise bei Summen) zu vermeiden, schreiben wir zum Beispiel $k \sim D$ anstelle der Indexmenge. In diesem Fall wird über alle relevanten Elemente summiert, d. h. die Elemente, die auf dem Gebiet D nicht verschwinden. Aufgrund der globalen Definition von B-Splines könnten wir meist sogar über $\mathbb{Z}$ bzw. $\mathbb{Z}^2$ summieren (und die Indexmenge weglassen).

$x = (x_i)_{i=1,\dots,d}$ bezeichnet einen Vektor mit den d Einträgen $x_1, \dots, x_d$. Wenn nichts anderes angegeben ist, dann ist $\|x\| := \|x\|_2$ die euklidische Norm von $x \in \mathbb{R}^d$, die durch das euklidische Skalarprodukt $\langle x, y \rangle$ für $x, y \in \mathbb{R}^d$ induziert wird.

Der Abstand eines Punktes $x \in \mathbb{R}^d$ zu einer Menge $M \subset \mathbb{R}^d$ wird mit $\operatorname{dist}(x, M) := \inf_{y \in M} \|x - y\|_2$ bezeichnet. Der Träger $\operatorname{supp} f := \overline{\{x \in \mathbb{R}^m \mid f(x) \neq 0\}}$ einer Funktion f ist der Abschluss der Nichtnullstellenmenge.

2 Bivariate Splines

In diesem Kapitel werden wir die grundlegenden Definitionen und Eigenschaften von bivariaten Polynomen (Abschnitt 2.1), B-Splines (Abschnitt 2.2) und Splines (Abschnitt 2.3) kurz benennen und wiederholen. Dies geschieht zu dem Zweck, dass die vorgestellte Approximationsmethode im weiteren Verlauf der Arbeit auf eindeutigen und gemeinsamen Definitionen fußt. Es ist bekannterweise keineswegs so, dass es immer nur eine Möglichkeit der Definition bei mathematischen Begriffen gibt. Es gibt allein schon mehrere Möglichkeiten, (bivariate) Polynome oder B-Splines zu definieren (beispielsweise die Definition von B-Splines in [13] vs. [14]). Diese Definitionen sind selbstredend äquivalent, allerdings muss für das beabsichtigte Ziel eine passende Definition ausgesucht werden.

Bei den vorgestellten Definitionen und Sätzen folgen wir weitgehend den Darstellungen in [14]. Wir werden aus Gründen des Umfangs und der Redundanz Sätze nur angeben und nicht beweisen.

2.1 Bivariate Polynome

Bivariate Polynome sind „natürliche“ Verallgemeinerungen von Polynomen einer Variablen auf zwei Variablen. Man kann ein bivariates Polynom p als eine Funktion von zwei Variablen x_1 und x_2 auffassen, so dass $p(\cdot, x_2)$ bzw. $p(x_1, \cdot)$ für jedes x_1 und x_2 stets univariate Polynome in der ersten bzw. zweiten Variablen sind.

Definition 2.1 (bivariates Polynom)**.** Ein bivariates Polynom p vom Koordinatengrad $n = (n_1, n_2)$ ist eine Linearkombination von Monomen:

$$p(x) = \sum_{k \leq n} c_k x^k, \quad x^k = x_1^{k_1} x_2^{k_2}, \tag{2.1}$$

mit Koeffizienten $c_k \in \mathbb{R}$ und $c_n \neq 0$. Die Summation erfolgt über alle Multiindizes $k \in \mathbb{N}_0^2$, die in jeder Komponente nicht größer als n sind: $k_\nu \leq n_\nu$ für $\nu = 1, 2$.

Die bivariaten Polynome vom Koordinatengrad $\leq n$ bilden einen Vektorraum, der mit $\mathbb{P}^n$ bezeichnet wird. Er hat die $\mathbb{R}$-Dimension $(n_1+1)\cdot(n_2+1)$, denn die Monome x^k mit $k \leq n$ bilden eine Basis von $\mathbb{P}^n$. Wir scheiben $\mathbb{P}^n(D)$, falls x auf ein bestimmtes Gebiet $D \subset \mathbb{R}^2$ beschränkt ist. ◊

Alternativ kann man Polynome auch über den sogenannten totalen Grad definieren, der gleich der maximalen Summe der Grade der Exponenten für jedes Monom ist. Allerdings stellt sich bei der später eingeführten Tensorprodukt-Struktur für B-Splines (siehe Abschnitt 2.2) heraus, dass dies kein guter Ansatz ist: Die partielle Ableitung eines B-Splines mit gleichem Koordinatengrad in beiden Variablen ist eine Differenz von B-Splines, deren Koordinatengrade in den Variablen unterschiedlich sind (Satz 2.5).

2.2 B-Splines

Polynome eignen sich nicht besonders gut für die Interpolation oder Approximation einer großen Anzahl von Daten. Für die Interpolation von $n+1$ Datenpunkten benötigt man im univariaten Fall i. A. ein Polynom vom Grad n. Sind nun sehr viele Daten gegeben, dann werden die Polynomgrade ziemlich hoch sein. Dabei ergeben sich mitunter starke Oszillationen zwischen den interpolierten Daten, ein Effekt, der auch als Runge-Effekt bekannt ist. Daher ist die Interpolation mit Polynomen vom Grad größer als vier nicht üblich.

Einen möglichen Ausweg bieten Splines, also stückweise Polynome. Weil sie auf jedem Abschnitt ein Polynom sind, sind sie sich ähnlich einfach zu handhaben wie Polynome. Gleichzeitig treten aber keine Oszillationen bei Interpolation oder Approximation auf, da der maximale Polynomgrad auf jedem Stück gleich ist.

Für eine Basis des Spline-Raums gibt es natürlich viele Möglichkeiten. B-Splines haben sich nicht zuletzt durch ihre vielen schönen Eigenschaften und durch die einfachen Rekursionsformeln etabliert. Sie wurden durch Isaac J. Schoenberg 1946 zuerst eingeführt, der allerdings immer behauptet hat, dass sie schon Laplace bekannt gewesen seien (siehe [4]).

Im Folgenden werden nur uniforme B-Splines betrachtet, weil nur diese von der hier vorgestellten bivariaten Approximationsmethode benötigt werden. Für eine allgemeinere Definition und die zugrunde liegende Theorie sei auf [14] bzw. [23] verwiesen. Folgende Rekursion (für allgemeine B-Splines) wurde 1972 durch de Boor ([3]) und Cox ([6]) bewiesen, weswegen die Formel (2.2) auch Cox-de-Boor-Rekursionsformel heißt.

Definition 2.2 (univariater B-Spline)**.** Der (uniforme) univariate B-Spline b^n vom Grad n ist definiert durch die Rekursion

$$b^n(x) := \frac{1}{n}\left(xb^{n-1}(x) + (n+1-x)b^{n-1}(x-1)\right), \tag{2.2}$$

startend mit der charakteristischen Funktion

$$b^0(x) := \chi_{[0,1)}(x) = \begin{cases} 1 & \text{falls } x \in [0,1), \\ 0 & \text{sonst.} \end{cases} \tag{2.3}$$

$0, \ldots, n+1$ sind die Knoten von b^n und die Intervalle $[\ell, \ell+1)$, $\ell = 0, \ldots, n$, heißen Knotenintervalle von b^n. Allgemeine uniforme B-Splines $b^n_{k,h}$ für die Gitterweite $h > 0$ ergeben sich aus b^n durch Skalierung und Verschiebung: $b^n_{k,h}(x) := b^n(x/h - k)$, $k \in \mathbb{Z}$. ◊

Die ersten vier uniformen B-Splines sind in Abbildung 2.1 dargestellt. Es handelt sich um die Grade $0, 1, 2, 3$, die aus den oben erwähnten Gründen am häufigsten verwendet werden.

B-Splines erfreuen sich an folgenden grundlegenden Eigenschaften:

Satz 2.3 (Eigenschaften von univariaten B-Splines)**.** *Der B-Spline b^n vom Grad n verschwindet außerhalb von $[0, n+1)$. Auf jedem Knotenintervall ist b^n ein nicht-negatives Polynom vom Grad $\leq n$.*

b^n ist an jedem Knoten n-fach stetig differenzierbar. Die Ableitung erfolgt durch Differenzenbildung zwischen B-Splines vom Grad $n-1$:

$$\frac{d}{dx} b^n(x) = b^{n-1}(x) - b^{n-1}(x-1). \qquad \square \tag{2.4}$$

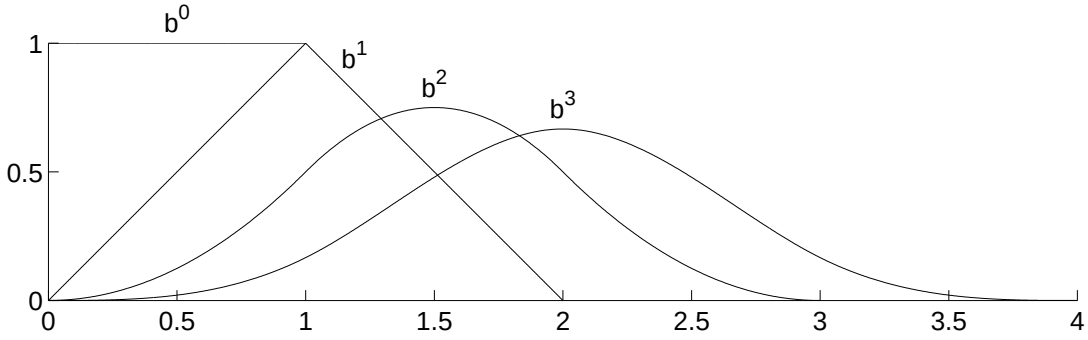
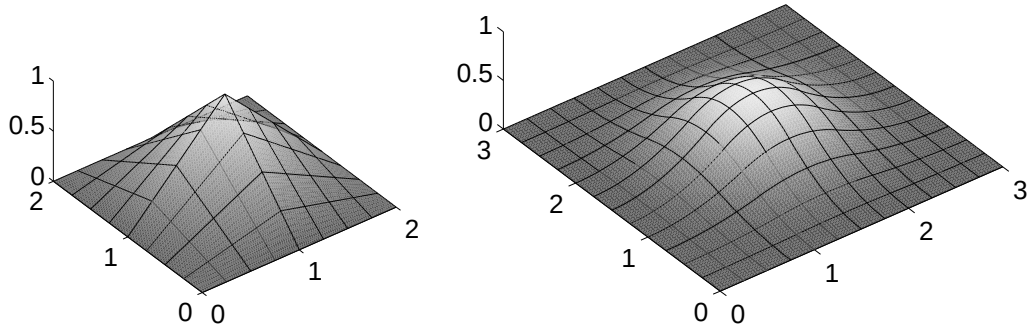


Abbildung 2.1: uniforme, univariate B-Splines vom Grad $0, 1, 2, 3$



(a) bilinear: $b^{(1,1)}$ **(b)** biquadratisch: $b^{(2,2)}$

Abbildung 2.2: uniforme, bivariate B-Splines auf ihrem Träger

Wir können die Differentiationsformel auch umschreiben (siehe [14]) zu

$$b^n(x) = \int_0^1 b^{n-1}(x-y)dy, \tag{2.5}$$

so dass b^n als Ergebnis des Prozesses einer Mittelwertbildung über b^{n-1} verstanden werden kann (Faltung von $\chi_{[0,1)}$ mit b^{n-1}). Alternativ zur Rekursionsformel (2.2) kann man B-Splines auch über die Formel (2.5) definieren, so geschehen in [13].

Ausgehend von den univariaten B-Splines definieren wir bivariate B-Splines als Tensorprodukte von B-Splines einer Variablen:

Definition 2.4 (bivariater B-Spline)**.** Ein (uniformer) bivariater B-Spline $b^n_{k,h}$ vom Grad $n = (n_1, n_2)$ für die Gitterweite $h = (h_1, h_2)$ ist ein Produkt von univariaten B-Splines:

$$b^n_{k,h}(x) := b^{n_1}_{k_1,h_1}(x_1) \cdot b^{n_2}_{k_2,h_2}(x_2), \quad k = (k_1, k_2) \in \mathbb{Z}^2. \tag{2.6}$$

Zusätzlich definieren wir $b^n := b^n_{0,1}$. ◊

In Abbildung 2.2 sind die B-Splines b^n für den Fall $n = 1$ und $n = 2$ dargestellt.

[14] zählt ein paar Vor- und Nachteile der Tensorprodukt-Konstruktion auf. Zu den wichtigsten Vorteilen gehört, dass sie einfach zu beschreiben ist und dass sich die Eigenschaften von univariaten B-Splines leicht übertragen:

Satz 2.5 (Eigenschaften von bivariaten B-Splines). *Der bivariate B-Spline $b^n_{k,h}$ vom Grad n verschwindet außerhalb des Rechtecks $[k_1, k_1+n_1+1)h_1 \times [k_2, k_2+n_2+1)h_2$. Auf jedem Gitterrechteck $[\ell_1, \ell_1+1)h_1 \times [\ell_2, \ell_2+1)h_2$, $\ell = (\ell_1, \ell_2) \in \mathbb{Z}^2$, ist $b^n_{k,h}$ ein nicht-negatives, bivariates Polynom vom Koordinatengrad $\leq n$. Auf jeder Parallelen zur ν-ten Koordinatenachse ist $b^n_{k,h}$ ein Vielfaches von $b^{n_\nu}_{k_\nu,h_\nu}$, $\nu = 1, 2$.*

$b^n_{k,h}$ ist in x_ν-Richtung n_ν-fach stetig differenzierbar. Die Ableitung erfolgt durch Differenzenbildung zwischen B-Splines vom Grad $n - e_\nu$:

$$\frac{\partial}{\partial x_\nu} b^n_{k,h} = \frac{1}{h_\nu} \left(b^{n-e_\nu}_{k,h} - b^{n-e_\nu}_{k+e_\nu,h} \right), \quad \nu = 1, 2, \tag{2.7}$$

wobei $e_\nu \in \mathbb{R}^2$ der ν-te Einheitsvektor ist. □

Einer der Hauptnachteile ist, dass die B-Spline-Basis nicht lokal verfeinert werden kann, denn Änderungen wirken sich wegen der Tensorprodukt-Struktur global aus. Dafür kann man z. B. hierarchische Basen verwenden (siehe [13, 14]).

2.3 Spline-Funktionen

Aufbauend auf den vorherigen Definitionen können wir jetzt bivariate Splines definieren. Es ist naheliegend, wie diese Definition im univariaten Fall aussieht, daher gehen wir aus Redundanzgründen sofort zu zwei Variablen über. Die Auswertung von Splines erfolgt mit dem einfachen Algorithmus von de Boor ([2]), basierend auf dem Algorithmus von de Casteljau für Bézier-Kurven.

Definition 2.6 (bivariater Spline). Ein (uniformer) bivariater Spline p vom Koordinatengrad $\leq n = (n_1, n_2)$ mit der Gitterweite $h = (h_1, h_2)$ auf dem Gebiet $D \subset \mathbb{R}^2$ ist eine Linearkombination der B-Splines, die auf D nicht verschwinden:

$$p(x) = \sum_{k \sim D} c_k b^n_{k,h}(x), \quad x \in D. \tag{2.8}$$

Der Raum aller solcher bivariaten Splines wird mit $S^n_h(D)$ bezeichnet.

Die B-Splines $b^n_{k,h}$ sind für festes n und h linear unabhängig, das heißt, die Koeffizienten c_k von p sind eindeutig bestimmt. ◊

Algorithmus 2.7 (de Boor). Es seien $p = \sum_{k \in D} c_k b^n_{k,h} \in S^n_h(D)$ ein bivariater Spline und $x = (x_1, x_2) \in D$. Dann kann $p(x)$ wie folgt berechnet werden:

(a) Bestimme $\ell = (\ell_1, \ell_2)$ und $t = (t_1, t_2)$ mit $x_\nu = (\ell_\nu + t_\nu)h_\nu$ und $t \in [0, 1)^2$.

(b) Definiere zunächst $a_{k_1} := (c_{(k_1,k_2)})_{k_2 = \ell_2 - n_2, \ldots, \ell_2}$. Für $i = n_1, \ldots, 1$ berechne sukzessive $a_{\ell_1 - j} \leftarrow \gamma a_{\ell_1 - j} + (i - \gamma) a_{\ell_1 - j - 1}$ für $j = 0, \ldots, i-1$, wobei $\gamma := j + t_1$. Bezeichne den resultierenden Vektor als $(p_{k_2})_{k_2 = \ell_2 - n_2, \ldots, \ell_2} := a_{\ell_1}$.

(c) Für $i = n_2, \ldots, 1$ berechne sukzessive $p_{\ell_2 - j} \leftarrow \gamma p_{\ell_2 - j} + (i - \gamma) p_{\ell_2 - j - 1}$ für $j = 0, \ldots, i-1$, wobei $\gamma := j + t_2$. Dann gilt am Ende $p(x) = p_{\ell_2} / (n_1! n_2!)$. ◊

Ziel der in dieser Arbeit vorgestellten Approximationsmethode wird es sein, passende Koeffizienten $c_k \in \mathbb{R}$ eines bivariaten Splines so zu finden, dass die vorgegebenen Daten „möglichst gut“ angenähert werden.

3 Quasi-Interpolation und lokale Polynom-Approximation

Nachdem wir im letzten Kapitel die wesentlichen Definitionen als Grundlage für die weitere Arbeit vorgestellt haben, werden wir in diesem Kapitel die Approximationsmethode erarbeiten, um die es im Rahmen dieser Arbeit geht. Zunächst werden wir in Abschnitt 3.1 die sog. Quasi-Interpolation in einer und in zwei Variablen erklären und ihre Eigenschaften nennen. Anschließend kümmern wir uns in Abschnitt 3.2 um die Generierung der benötigten Zwischendaten auf einem regulären Gitter. Schließlich werden wir in Abschnitt 3.3 diese beiden Schritte zusammenführen und das in diesem Abschnitt ausformulierte Problem mit einer Kombination von lokaler Polynom-Approximation und Quasi-Interpolation numerisch lösen.

3.1 Univariate und bivariate Quasi-Interpolation

3.1.1 Quasi-Interpolation in einer Variablen

Bei der in dieser Arbeit vorgestellten Approximationsmethode handelt es sich um eine Spline-Approximation. Das bedeutet, dass die gegebenen Daten durch eine Spline-Funktion $p(x) = \sum_{k\sim D} c_k b^n_{k,h}(x)$ mit gewissen Koeffizienten $c_k \in \mathbb{R}$ angenähert werden sollen. Um Fehlerabschätzungen beweisen zu können, benötigen wir gewisse Vorbedingungen an die Bildungsmethode dieser Koeffizienten. Eine Möglichkeit ist dabei die sogenannte Quasi-Interpolation.

Bei Quasi-Interpolation handelt es sich um ein lineares Spline-Approximations-Schema ([14]). Unter recht milden und „natürlichen“ Voraussetzungen kann gezeigt werden, dass Quasi-Interpolation eine effiziente Methode zur Approximation mit Splines darstellt.

Wir verfahren wieder wie in [14], jedoch werden wir aus Gründen der Kürze und der Einfachheit wie im letzten Kapitel nur uniforme Knotenfolgen betrachten. Für den allgemeinen Fall sei auf [14] verwiesen.

Definition 3.1 (univariate Quasi-Interpolation)**.** Eine Abbildung

$$Q\colon \mathcal{C}(D) \to S^n_h(D), \quad f \mapsto Qf = \sum_{k\sim D} (Q_k f) b^n_{k,h}, \tag{3.1}$$

mit dem Gebiet $D \subset \mathbb{R}$, der Gitterweite $h > 0$ und dem Grad $n \in \mathbb{N}_0$ heißt Quasi-Interpolationsoperator, falls

(a) Q_k für alle $k \in \mathbb{Z}$ ein lokal beschränktes, lineares Funktional ist, das heißt, es gibt ein $\|Q\|$, so dass für alle $k \in \mathbb{Z}$

$$Q_k\colon \mathcal{C}(D) \to \mathbb{R} \text{ linear}, \quad |Q_k f| \le \|Q\| \, \|f\|_{\infty,[k,k+n+1)h}\,, \tag{3.2}$$

für alle $f \in \mathcal{C}(D)$ mit dem Supremum $\|f\|_{\infty,U} := \sup_{x\in U} |f(x)|$ von $f \in \mathcal{C}(D)$ auf $U \subset \mathbb{R}$, und

(b) Q jedes Polynom vom Grad $\leq n$ auf sich selbst abbildet, d. h. $Qp = p$ auf D für alle $p \in \mathbb{P}^n(D)$. Äquivalent dazu ist, dass für alle $y \in \mathbb{R}$ und $k \sim D$ gilt, dass

$$\begin{aligned} Q_k p = \psi_k(y) \quad \text{mit} \quad & p(x) := (x-y)^n, \\ & \psi_k(y) := ((k+1)h - y) \cdots ((k+n)h - y). \end{aligned} \tag{3.3}$$

Das Bild Qf von f unter Q nennen wir Quasi-Interpolant und die Berechnung von Qf heißt Quasi-Interpolation. ◊

Die erste Bedingung (3.2) für einen univariaten Quasi-Interpolanten stellt sicher, dass der Quasi-Interpolationsoperator linear ist und dass die Bilder $Q_k f$ der lokalen Funktionale Q_k nur von Werten von f im Träger von $b^n_{k,h}$ abhängen. Die letztgenannte Tatsache ist sinnvoll, weil eine Veränderung $f \to \widetilde{f}$ von f in einem bestimmten Bereich natürlich nur die Werte $Q_k f$ beeinflussen sollte, bei denen die entsprechenden B-Splines im veränderten Bereich nicht verschwinden. Gleichzeitig sollten die Beträge $|Q_k f|$ der Koeffizienten des entstehenden Splines gleichmäßig durch das Maximum von f auf dem Träger von $b^n_{k,h}$ beschränkt sein.

Die zweite Bedingung (3.3) ist die wichtigere, denn durch sie wird sichergestellt, dass Polynome vom Grad $\leq n$ durch den Operator unverändert bleiben. Zu beachten ist, dass das für sich noch nicht impliziert, dass jeder Spline auf sich selbst abgebildet wird (das heißt, in diesem Fall wäre $Q|_{S^n_h(D)} = \mathrm{id}_{S^n_h(D)}$ und Q eine Projektion). In [14] wird gezeigt, dass dies aber für Quasi-Interpolanten gilt, bei denen jedes Funktional Q_k nur von Werten von f in einem einzigen Knotenintervall in D abhängt. Die wesentliche Idee dabei ist, dass ein Spline auf einem einzigen Knotenintervall einfach nur ein Polynom ist, das vom Operator auf sich selbst abgebildet wird. Wenn die Konstante $\|Q\|$ zusätzlich nur vom Grad n abhängt – nicht von h oder im allgemeinen Fall von der Knotenfolge –, dann sprechen wir von einem Standard-Projektor.

Mit den relativ schwachen Voraussetzungen eines Quasi-Interpolanten kann man zeigen (siehe z. B. [14]), dass der Fehler bei der Quasi-Interpolation die Ordnung $\mathcal{O}(h^{n+1})$ hat:

Satz 3.2 (Fehler der Quasi-Interpolation). *Es sei $Q, f \mapsto Qf = \sum_{k \sim D} (Q_k f) b^n_{k,h}$, ein Quasi-Interpolationsoperator. Dann gilt für den Fehler in einem Punkt $x \in D$:*

$$|f(x) - (Qf)(x)| \leq \frac{\|Q\|}{(n+1)!} \left\| f^{(n+1)} \right\|_{\infty, D_x} h(x)^{n+1}. \tag{3.4}$$

Dabei ist D_x die Vereinigung der Träger der für x relevanten B-Splines und $h(x) := \max_{y \in D_x} |y - x|$ der maximale Abstand von y zum Rand von D_x. □

Natürlich können wir die angegebene lokale Fehlerabschätzung auch zu einer globalen Formel abschwächen, indem wir in (3.4) die Norm $\left\| f^{(n+1)} \right\|_{\infty, D_x}$ durch $\left\| f^{(n+1)} \right\|_\infty$ und $h(x)$ durch $(n+1)h$ ersetzen (die im uniformen Fall konstante Länge der Träger der B-Splines). Man kann sogar allgemeiner den Fehler der Ableitungen der Ordnung $j = 0, \ldots, n$ bis auf eine Konstante, die nur von n abhängt, durch $\|Q\|$ multipliziert mit $\left\| f^{(n+1)} \right\|_{\infty, D_x}$ und $h(x)^{n+1-j}$ abschätzen. Da die Approximation der Ableitungen zwar auch nützlich, aber in dieser Arbeit nicht behandelt wird, wird hier nicht weiter darauf eingegangen.

3.1.2 Wahl der linearen Funktionale

Für die Wahl der Funktionale Q_k existieren mehrere Möglichkeiten. Dabei stellt man $Q_k f$ meist als Linearkombination von Werten von f im Träger $[k, k+n+1]h$ von $b^n_{k,h}$ dar:

$$Q_k f = \sum_{\alpha=0}^{m_k} w_{k,\alpha} f(x_{k,\alpha}) \tag{3.5}$$

mit Gewichten $w_{k,\alpha}$ und Stützstellen $x_{k,\alpha}$. (Wir betrachten nur stetige B-Splines, so dass der Unterschied zwischen dem hier gewählten Bereich $[k, k+n+1]h$ der Stützstellen und der Bereich $[k, k+n+1)h$ der Maximumsnorm in (3.2) unerheblich ist.) Dies hat bei Speicherung der Gewichte und der Stützpunkte in einer Tabelle den Vorteil, dass die Koeffizienten $Q_k f$ des Splines Qf schnell berechnet werden können – es muss z. B. kein lineares Gleichungssystem gelöst werden. Außerdem ist die erste Bedingung (3.2) für einen Quasi-Interpolanten automatisch erfüllt, wenn $\sum_{\alpha=0}^{m_k} |w_{k,\alpha}| \le \|Q\|$ für alle $k \sim D$, da

$$|Q_k f| \le \sum_{\alpha=0}^{m_k} |w_{k,\alpha}||f(x_{k,\alpha})| \le \|f\|_{\infty,[k,k+n+1]h} \cdot \sum_{\alpha=0}^{m_k} |w_{k,\alpha}|. \tag{3.6}$$

Die zweite Bedingung (3.3) für einen Quasi-Interpolanten kann für den Fall m_k konstant gleich n auch einfach geprüft werden, siehe unten.

Wir betrachten nun zwei Beispiele für Quasi-Interpolationsfunktionale für $D = \mathbb{R}$.

Beispiel 3.3 (Standard-Projektor)**.** Zunächst wollen wir einen Standard-Projektor konstruieren, das heißt, für $k \in \mathbb{Z}$ müssen die Stützstellen $x_{k,\alpha}$ alle in einem abgeschlossenen Knotenintervall $[\ell, \ell+1]h$ liegen. Wir fixieren $k \in \mathbb{Z}$ und wählen als Intervall das mittlere des Trägers von $b^n_{k,h}$ für gerade n und das linke mittlere Intervall für ungerade n, also $\ell = k + \lfloor n/2 \rfloor$. Nun wählen wir $n+1$ äquidistante Punkte in $[\ell, \ell+1]h$:

$$x_\alpha = \left(\ell + \frac{\alpha}{n}\right) h, \quad \alpha = 0, \dots, n, \tag{3.7}$$

dargestellt in Abbildung 3.1a. Die zweite Bedingung (3.3) wird dann zu

$$Q_k(\cdot - y)^n = \sum_{\alpha=0}^{n} w_\alpha \left(\left(\ell + \frac{\alpha}{n}\right) h - y\right)^n = \prod_{\beta=1}^{n} ((k+\beta)h - y) = \psi_k(y). \tag{3.8}$$

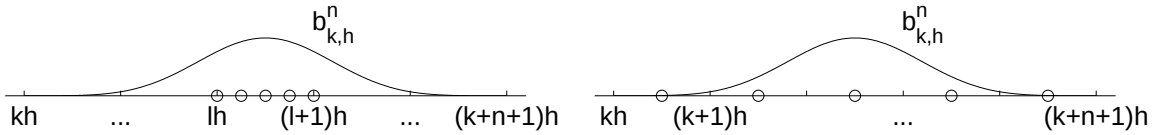
Beide Seiten sind Polynome in y vom Grad $\le n$, das heißt, anstatt die Koeffizienten explizit auszurechnen und zu vergleichen, setzen wir $n+1$ verschiedene Werte in y ein und benutzen die Tatsache, dass das interpolierende Polynom eindeutig ist. Kanonischerweise verwendet man $y = x_\nu$, $\nu = 0, \dots, n$, denn so erhalten wir das relativ einfache lineare Gleichungssystem

$$\sum_{\alpha=0}^{n} w_\alpha (\alpha - \nu)^n = \prod_{\beta=1}^{n} \left(n\beta - \nu - n\left\lfloor \frac{n}{2} \right\rfloor\right), \quad \nu = 0, \dots, n, \tag{3.9}$$

für $w_0, \dots, w_n$, das nicht mehr von k und h abhängt. Insbesondere erfüllt die Lösung die erste Bedingung (3.2), wenn wir $\|Q\| := \sum_{\alpha=0}^{n} |w_\alpha|$ wählen, und Q ist ein Standard-Projektor. ◊

Tabelle 3.1: Koeffizienten w_α des Quasi-Interpolationsoperators in Beispiel 3.4

n	Koeffizienten $w_0, \ldots, w_n$				
1	$1/2$	$1/2$			
2	$-1/8$	$5/4$	$-1/8$		
3	$-7/48$	$31/48$	$31/48$	$-7/48$	
4	$47/1152$	$-107/288$	$319/192$	$-107/288$	$47/1152$



(a) Stützstellen in einem Intervall **(b)** Stützstellen in den Intervallmittelpunkten

Abbildung 3.1: Wahl der Stützstellen für Quasi-Interpolation vom Grad $n = 4$

Beispiel 3.4 (weiterer Quasi-Interpolant)**.** Eine andere Möglichkeit für die Konstruktion eines Quasi-Interpolanten besteht darin, als Stützstellen die Mittelpunkte der Knotenintervalle zu wählen (Abbildung 3.1b):

$$x_\alpha = \left(k + \frac{1}{2} + \alpha\right) h, \quad \alpha = 0, \ldots, n, \tag{3.10}$$

für festes $k \in \mathbb{Z}$. In diesem Fall ist die zweite Bedingung (3.3) äquivalent zu

$$Q_k(\cdot - y)^n = \sum_{\alpha=0}^{n} w_\alpha \left(\left(k + \frac{1}{2} + \alpha\right) h - y\right)^n = \prod_{\beta=1}^{n} ((k+\beta)h - y) = \psi_k(y). \tag{3.11}$$

Einsetzen von $y = x_\nu$, $\nu = 0, \ldots, n$, liefert nach Kürzen von h^n mit

$$\sum_{\alpha=0}^{n} w_\alpha (\alpha - \nu)^n = \prod_{\beta=1}^{n} \left(\beta - \nu - \frac{1}{2}\right), \quad \nu = 0, \ldots, n, \tag{3.12}$$

ein lineares Gleichungssystem mit derselben Koeffizientenmatrix wie beim Standard-Projektor. Es handelt sich dabei um eine Toeplitz-Matrix (die Diagonalen beinhalten jeweils nur einen verschiedenen Eintrag), die für n gerade bzw. ungerade symmetrisch bzw. schiefsymmetrisch ist. Wie beim anderen Quasi-Interpolanten folgt, dass die Lösung weder von k noch von h abhängt, sondern nur noch vom Grad n. Somit ist die erste Bedingung (3.2) für einen Quasi-Interpolanten erfüllt und man kann bei der Implementierung die Koeffizienten für die am meisten genutzten Interpolationsgrade abspeichern. Wir geben in Tabelle 3.1 die entstehenden Koeffizienten w_α, $\alpha = 0, \ldots, n$, für die Grade $n = 1, \ldots, 4$ an. ◊

In [14] wird erwähnt, dass das letztgenannte Quasi-Interpolationsschema besonders vorteilhaft ist, weil benachbarte Funktionale alle bis auf zwei Funktionsauswertungen gemeinsam haben. Der Quasi-Interpolationsoperator aus Beispiel 3.3 hat als Standard-Projektor dagegen den Vorteil, jeden Spline auf sich selbst abzubilden. Jedoch benötigt dieser für jedes Funktio-

nal $n+1$ Funktionsauswertungen in jedem Knotenintervall, die somit kaum (nur am Rand) für andere Funktionale benutzt werden können. Außerdem ist der Operator für ungeraden Grad asymmetrisch, da das mittlere Knotenintervall eines B-Splines nicht eindeutig ist. Aus diesen Gründen werden wir uns auf den Operator aus Beispiel 3.4 beschränken.

In obigen Beispielen wurde $D = \mathbb{R}$ angenommen und der Fall, dass D zum Beispiel ein endliches Intervall darstellt, außer Acht gelassen. In diesem Fall kann es nämlich sein, dass ein paar der gewählten Stützstellen für die Quasi-Interpolation in den Bereich außerhalb von D fallen. Im Abschnitt 3.3.1 werden wir nochmals auf diese Problematik zurückkommen.

3.1.3 Bivariate Quasi-Interpolation

Die Tensorprodukt-Konstruktion von bivariaten B-Splines lässt sich im allgemeinen Fall der Definition 3.1 leider nicht auf die Konstruktion von Quasi-Interpolationsoperatoren in zwei Variablen übertragen. Zunächst einmal ist überhaupt nicht klar, wie eine solche Definition aussehen könnte, denn die univariaten Quasi-Interpolationsfunktionale wirken eben nur auf Funktionen einer Variablen, das heißt, die zu interpolierende, bivariate Funktion müsste irgendwie auf eine Dimension eingeschränkt werden. Außerdem sollte nach [14] die optimale Genauigkeit für beliebige Gebiete erhalten werden, was eine große Schwierigkeit darstellt.

Daher beschränken wir uns im Folgenden auf Quasi-Interpolationsoperatoren mit Funktionalen der Form (3.5) und Gebieten $D \subset \mathbb{R}^2$, die ein Rechteck sind. Die Anzahl m_k der für das Funktional Q_k relevanten Funktionswerte von f nehmen wir dabei der Einfachheit halber als konstant für alle k an. Wenn das zugrunde liegende Intervall endlich ist, stellt das kein Problem dar, weil dann m_k konstant auf die maximal benötigte Anzahl an Funktionswerten gesetzt werden kann und „überschüssige“ Gewichte auf null gesetzt werden können.

Definition 3.5 (bivariate Quasi-Interpolation)**.** Es seien D_1 und D_2 abgeschlossene Intervalle in $\mathbb{R}$ und

$$Q_\nu \colon \mathcal{C}(D_\nu) \to S_h^n(D_\nu), \quad f \mapsto Q_\nu f = \sum_{k_\nu \sim D_\nu} (Q_{\nu,k_\nu} f) b_{k_\nu,h_\nu}^{n_\nu}, \quad \nu = 1,2, \tag{3.13}$$

zwei Quasi-Interpolationsoperatoren mit Gitterweiten $h_\nu > 0$, Graden $n_\nu \in \mathbb{N}_0$, Gebieten D_ν und Funktionalen der Form

$$Q_{\nu,k_\nu} f = \sum_{\alpha_\nu=0}^{m_\nu} w_{\nu,k_\nu,\alpha_\nu} f(x_{\nu,k_\nu,\alpha_\nu}) \tag{3.14}$$

mit Gewichten w_{ν,k_ν,α_ν} und Stützstellen x_{ν,k_ν,α_ν} in $[k_\nu, k_\nu + n_\nu + 1]h_\nu$, so dass $\sum_{\alpha_\nu=0}^{m_\nu} |w_{\nu,k_\nu,\alpha_\nu}| \leq \|Q_\nu\|$ für alle $k_\nu \sim D_\nu$. Dann heißt die Abbildung

$$Q \colon \mathcal{C}(D) \to S_h^n(D), \quad f \mapsto Qf = \sum_{k \sim D} (Q_k f) b_{k,h}^n, \tag{3.15}$$

mit den Funktionalen Q_k, $k = (k_1, k_2)$, definiert durch

$$Q_k f := \sum_{\alpha_1=0}^{m_1} \sum_{\alpha_2=0}^{m_2} w_{1,k_1,\alpha_1} w_{2,k_2,\alpha_2} f(x_{1,k_1,\alpha_1}, x_{2,k_2,\alpha_2}) \tag{3.16}$$

bivariater Quasi-Interpolationsoperator mit Gebiet $D = D_1 \times D_2$, Gitterweite $h = (h_1, h_2)$ und Grad $n = (n_1, n_2)$.

Wieder nennen wir das Bild Qf von f unter Q Quasi-Interpolant und die Berechnung von Qf heißt Quasi-Interpolation. ◊

Zunächst sehen wir, dass diese Definition tatsächlich Sinn ergibt, indem wir bemerken, dass der Quasi-Interpolant Qf ein bivariater Spline im Sinne von Definition 2.6 ist. Die weiteren Eigenschaften von univariaten Quasi-Interpolationsoperatoren übertragen sich auf den bivariaten Fall:

Satz 3.6 (Eigenschaften bivariater Quasi-Interpolation)**.** *Es sei Q ein bivariater Quasi-Interpolationsoperator wie in Definition 3.5. Dann gilt:*

(a) Q_k ist für alle $k \in \mathbb{Z}^2$ ein lokal beschränktes, lineares Funktional. Es gibt also ein $\|Q\|$, so dass für alle $k \in \mathbb{Z}^2$

$$\begin{aligned} &Q_k \colon \mathcal{C}(D) \to \mathbb{R} \text{ linear}, \\ &|Q_k f| \le \|Q\| \, \|f\|_{\infty,[k_1,k_1+n_1+1)h_1 \times [k_2,k_2+n_2+1)h_2} . \end{aligned} \tag{3.17}$$

(b) Q bildet jedes bivariate Polynom vom Grad $\le n$ auf sich selbst ab, d. h. $Qp = p$ auf D für alle $p \in \mathbb{P}^n(D)$.

Beweis. Die Linearität von Q_k für Punkt (a) ($Q_k(f+g) = Q_k f + Q_k g$ und $Q_k(\lambda f) = \lambda(Q_k f)$ für $f, g \in \mathcal{C}(D)$ und $\lambda \in \mathbb{R}$) ist aus der Definition von $Q_k f$ ersichtlich und folgt aus der Linearität der Summation in (3.16). Die lokale Beschränktheit gilt wegen

$$|Q_k f| \le \sum_{\alpha_1=0}^{m_1} \sum_{\alpha_2=0}^{m_2} |w_{1,k_1,\alpha_1}||w_{2,k_2,\alpha_2}||f(x_{1,k_1,\alpha_1}, x_{2,k_2,\alpha_2})|. \tag{3.18}$$

Indem wir die rechte Seite durch das Maximum der Funktionswerte von f multipliziert mit der Doppelsumme der Gewichtsprodukte abschätzen, erhalten wir die angegebene Behauptung aufgrund von

$$\sum_{\alpha_1=0}^{m_1} \sum_{\alpha_2=0}^{m_2} |w_{1,k_1,\alpha_1}||w_{2,k_2,\alpha_2}| \le \|Q_1\| \cdot \|Q_2\| =: \|Q\| . \tag{3.19}$$

Für Punkt (b) reicht es durch die Marsden-Identität (siehe z. B. [14]), für alle $y \in \mathbb{R}^2$

$$Q_k p = \psi_k(y), \quad p(x) := (x-y)^n = p_1(x_1) \cdot p_2(x_2), \quad p_\nu(x_\nu) := (x_\nu - y_\nu)^{n_\nu}, \tag{3.20}$$

mit

$$\psi_k(y) := \psi_{k_1}(y_1) \cdot \psi_{k_2}(y_2), \quad \psi_{k_\nu}(y_\nu) := ((k_\nu+1)h_\nu - y_\nu) \cdots ((k_\nu+n_\nu)h_\nu - y_\nu), \tag{3.21}$$

zu prüfen, denn die p spannen für $y \in \mathbb{R}^2$ den Raum $\mathbb{P}^n$ der bivariaten Polynome vom Grad

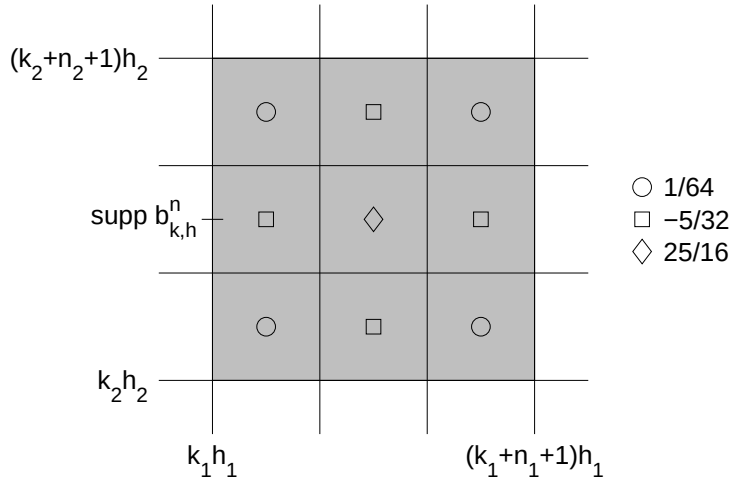


Abbildung 3.2: Gewichte eines bivariaten Quasi-Interpolationsoperators basierend auf Beispiel 3.4, hier speziell mit $h_1 = h_2$ und $n_1 = n_2 = 2$

$\leq n$ auf. Einsetzen in die Definition ergibt

$$Q_k p = \sum_{\alpha_1=0}^{m_1} \sum_{\alpha_2=0}^{m_2} w_{1,k_1,\alpha_1} w_{2,k_2,\alpha_2} p(x_{1,k_1,\alpha_1}, x_{2,k_2,\alpha_2}). \tag{3.22}$$

Diese Doppelsumme kann in zwei Summen aufgespalten werden:

$$\left(\sum_{\alpha_1=0}^{m_1} w_{1,k_1,\alpha_1} p_1(x_{1,k_1,\alpha_1}) \right) \cdot \left(\sum_{\alpha_2=0}^{m_2} w_{2,k_2,\alpha_2} p_2(x_{2,k_2,\alpha_2}) \right). \tag{3.23}$$

Die linke große Klammer ist nach (3.14) gleich $Q_{1,k_1} p_1$ und die rechte große Klammer ist gleich $Q_{2,k_2} p_2$. Weil die univariaten Quasi-Interpolationsoperatoren Q_1 und Q_2 ebenfalls Polynome reproduzieren (Gleichung (3.3)), erhalten wir somit $\psi_{k_1}(y_1) \cdot \psi_{k_2}(y_2) = \psi_k(y)$. □

Die Fehlerabschätzung aus dem univariaten Fall lässt sich durch einen ähnlichen Beweis (siehe [14]) auf den Fall zweier Variablen übertragen:

Satz 3.7 (Fehler der bivariaten Quasi-Interpolation)**.** *Es sei $Q, f \mapsto Qf = \sum_{k \sim D} (Q_k f) b^n_{k,h}$, ein bivariater Quasi-Interpolant mit Gebiet D, Gitterweite $h = (h_1, h_2)$ und Grad $n = (n_1, n_2)$. Dann gilt für den Fehler in einem Punkt $x \in D$:*

$$|f(x) - (Qf)(x)| \leq c(n) \cdot \left(h_1^{n_1+1} \left\| \partial_1^{n_1+1} f \right\|_{\infty,D} + h_2^{n_2+1} \left\| \partial_2^{n_2+1} f \right\|_{\infty,D} \right). \tag{3.24}$$

□

Abbildung 3.2 zeigt als Beispiel die Gewichte der Funktionale des bivariaten Quasi-Interpolationsoperators, der durch Verwendung von Beispiel 3.4 in beiden Variablen entsteht, hier jeweils mit Grad 2. Unabhängig vom Grad ergeben sich dabei insgesamt drei verschiedene Gewichte, entsprechend den Kombinationsmöglichkeiten.

3.2 Bivariate Polynom-Approximation

Die vorgestellte Methode der Quasi-Interpolation lässt sich allein noch nicht für die Approximation unregelmäßig verteilter Daten verwenden. Bivariate Quasi-Interpolationsoperatoren, so wie sie in Abschnitt 3.1.3 definiert wurden, benötigen nämlich die Daten $f(x_1, x_2)$ auf einem regelmäßigen Gitter, das heißt, (x_1, x_2) muss sich (im endlichen Fall) im Kreuzprodukt zweier endlicher Teilmengen von $\mathbb{R}$ befinden. Um einen Algorithmus der Approximation unregelmäßig verteilter Daten zu erhalten, müssen wir die von der Quasi-Interpolation benötigten Daten auf dem regelmäßigen Gitter aus den unregelmäßig verteilten Daten erzeugen.

Wir verwenden dazu die sogenannte lokale Polynom-Approximation. „Lokal“ deshalb, weil zur Erzeugung eines Gitterpunkts nur Daten aus einer kleinen Umgebung des Punkts herangezogen werden, um die Polynom-Grade klein zu halten. Der globale Zusammenhang zwischen den Gitterpunkten wird dann durch die Quasi-Interpolation hergestellt.

Leider ist bivariate Polynom-Approximation, oder im Spezialfall Polynom-Interpolation, erheblich schwieriger zu handhaben als der Fall einer Variablen und Gegenstand aktueller Forschung. Einen guten Überblick über bisher erarbeitete Methoden geben [11, 18, 21].

Die Schwierigkeit multivariater Interpolation erklärt sich folgendermaßen: Im univariaten Fall gibt es bekannterweise für $n+1$ paarweise verschiedene Punkte $x_1, \ldots, x_{n+1} \in \mathbb{R}$ mit Daten $f_1, \ldots, f_{n+1} \in \mathbb{R}$ genau ein interpolierendes Polynom p vom Grad $\leq n$, d. h. $p(x_i) = f_i$ für $i = 1, \ldots, n+1$. Das interpolierende Polynom lässt sich sogar mit der Lagrange-Formel explizit angeben (siehe [11, 14]), mit Dividierten Differenzen lässt sich auch Hermite-Interpolation (Interpolation der Ableitungen) einfach bewerkstelligen.

Anders ist das im multivariaten Fall, zum Beispiel für zwei Variablen: Je nach Lage der Punkte $(x_{1,i}, x_{2,i}) \in \mathbb{R}^2$ und der Daten f_i, $i = 1, \ldots, (n_1+1)(n_2+1)$, ist das Interpolationsproblem mit bivariaten Polynomen vom Koordinatengrad $\leq n = (n_1, n_2)$ nicht eindeutig oder sogar unlösbar:

Beispiel 3.8 (nicht eindeutig lösbares, bivariates Interpolationsproblem)**.** Für $n_1 = n_2 = 1$ betrachten wir die Daten zeilenweise gegeben durch

$$X = \begin{pmatrix} 0 & 1/2 & 1 & 2 \\ 0 & -1 & 1 & 1/2 \end{pmatrix}^T, \quad f = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}^T. \tag{3.25}$$

Wie man leicht durch Rechnung nachprüft, interpoliert das bivariate Polynom

$$p(x_1, x_2) = 0 + 1 \cdot x_1 + 2 \cdot x_2 - 3 \cdot x_1 x_2 \tag{3.26}$$

vom Koordinatengrad $(1, 1)$ diese Daten. Andererseits interpoliert natürlich auch das Nullpolynom die Daten, also ist die Lösung für diese Datenpunkte nicht eindeutig. ◊

Beispiel 3.9 (unlösbares, bivariates Interpolationsproblem)**.** Nun seien die Daten

$$X = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}^T, \quad f = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}^T \tag{3.27}$$

gegeben. Es kann kein Interpolationspolynom vom Grad $\leq (1, 1)$ geben, denn auf der Geraden $x_2 = 0$ wäre dies ein univariates Polynom vom Grad ≤ 1 mit drei Nullstellen. Damit müsste das Polynom trivial sein und würde den letzten Datenpunkt nicht interpolieren. ◊

Man sagt, das Interpolationsproblem für gegebene Datenpunkte $(x_{1,i}, x_{2,i}) \in \mathbb{R}^2$, $i = 1, \ldots, (n_1+1)(n_2+1)$, sei wohlgestellt (*well posed* oder nach [11] *poised*), falls für beliebige Daten f_i ein Interpolationspolynom $p \in \mathbb{P}^n$ existiert. In [11] wird dies für beliebige Vektorräume V von stetigen Funktionen verallgemeinert, indem man $\mathbb{R}^2$ durch $\mathbb{R}^d$, $\mathbb{P}^n$ durch V und $(n_1+1)(n_2+1)$ durch $\dim V$ ersetzt. Ein Raum V, so dass das Interpolationsproblem für beliebige $(\dim V)$-viele Punkte aus $\mathbb{R}^d$ wohlgestellt ist, heißt Haar-Raum. Für den Fall $d \geq 2$ erwähnt [11], dass keine nicht-trivialen Haar-Räume (der Dimension > 1) existieren. Für uns bedeutet das, dass es für jeden Grad $n = (n_1, n_2)$ immer paarweise verschiedene $(n_1+1)(n_2+1)$-viele Punkte in $\mathbb{R}^2$ gibt, so dass das Interpolationsproblem nicht *poised* ist.

Damit hängt auch die Eindeutigkeit der Lösung eines polynomiellen Approximationsproblems von der Lage der Datenpunkte ab. Bei der Polynom-Approximation sind mindestens $(n_1+1)(n_2+1)$-viele, paarweise verschiedene Datenpunkte gegeben, also $(x_{1,i}, x_{2,i}) \in \mathbb{R}^2$ mit $f_i \in \mathbb{R}$ für $i = 1, \ldots, m$ mit $m \geq (n_1+1)(n_2+1)$. Gesucht ist ein bivariates Polynom p vom Grad $\leq n = (n_1, n_2)$, das diese Punkte bestmöglich approximiert. „Bestmöglich“ heißt hier, dass die Summe der Fehlerquadrate minimiert wird, das heißt, für alle $q \in \mathbb{P}^n$ gilt

$$E(p) \leq E(q), \quad E(q) := \sum_{i=1}^{m} |q(x_{1,i}, x_{2,i}) - f_i|^2. \tag{3.28}$$

Daher nennt man das Verfahren auch die Methode der kleinsten Fehlerquadrate.

Man kann ein solches Minimierungsproblem mit Hilfe linearer Gleichungssysteme (LGS) umformulieren: Das gesuchte Polynom p mit Koeffizienten c sei

$$p(x) = \sum_{k \leq n} c_k x^k, \quad c = (c_{(0,0)}, c_{(1,0)}, \ldots, c_{(n_1,0)}, c_{(0,1)}, c_{(1,1)}, \ldots, c_{(n_1,1)}, \ldots, c_{(n_1,n_2)})^T. \tag{3.29}$$

Mit der $(m \times (n_1+1)(n_2+1))$-Matrix A mit den Zeilen a_i, $i = 1, \ldots, m$, und der rechten Seite f, beides definiert durch

$$a_i = (1, x_{1,i}, \ldots, x_{1,i}^{n_1}, x_{2,i}, x_{1,i}x_{2,i}, \ldots, x_{1,i}^{n_1}x_{2,i}, \ldots, x_{1,i}^{n_1}x_{2,i}^{n_2}), \quad f = (f_1, \ldots, f_m)^T, \tag{3.30}$$

erhalten wir ein im Allgemeinen überbestimmtes LGS $Ac = f$, so dass das Quadrat der euklidischen Norm vom Residuum des LGS der Fehlerquadratsumme entspricht:

$$E(p) = \|Ac - f\|_2^2 . \tag{3.31}$$

Eine Lösung c des Minimierungsproblems ist nun äquivalent (siehe [18, 24]) zu den Lösungen der Normalengleichungen

$$A^T A c = A^T f. \tag{3.32}$$

Dieses LGS ist eindeutig lösbar, wenn A vollen Rang hat (was i. A. nicht der Fall ist, siehe oben). Zusätzlich ist $A^T A$ symmetrisch und positiv definit ([24]), so dass dieses LGS mit Hilfe geeigneter Methoden, z. B. mit dem CG-Verfahren, numerisch gelöst werden kann.

Für unsere im Folgenden vorgestellte Approximationsmethode unregelmäßig verteilter Daten werden wir Polynom-Approximation statt -Interpolation verwenden, weil die genaue Zahl an Daten lokal um einen zu berechnenden Gitterpunkt herum nicht bekannt sein wird.

3.3 Spline-Approximation durch Quasi-Interpolation

3.3.1 Vorbemerkungen

Nun haben wir uns alle Voraussetzungen erarbeitet, die wir für die Formulierung des Approximationsproblems und dessen Lösung benötigen. Um zwei kleinere Details haben wir uns dabei noch nicht gekümmert:

Erstens wurde in Abschnitt 3.2 nicht spezifiziert, welche Punkte für die Generierung eines Gitterpunkts für die Quasi-Interpolation bei der „lokalen“ Polynom-Approximation verwendet werden. Damit das in den Gleichungen (3.29) und (3.30) definierte LGS überbestimmt ist, müssen in jedem Fall mindestens $(n_1 + 1)(n_2 + 1)$-viele Punkte verwendet werden, wobei $n = (n_1, n_2)$ den Grad der verwendeten Approximation bezeichnet. Wir werden deshalb von den gegebenen Daten voraussetzen, dass zusätzlich eine Gitterweite $h = 1/H > 0$ gegeben ist und sich in jedem Gitterquadrat mindestens ein Datenpunkt befindet. Wenn wir einen Gitterpunkt für die Quasi-Interpolation erzeugen wollen, werden wir alle Datenpunkte aus den umliegenden Gitterquadraten nehmen, und zwar aus mindestens so vielen, so dass das LGS überbestimmt ist. Dies ist natürlich nur die grobe Idee, die exakte Ausformulierung erfolgt später im Abschnitt 3.3.2.

Zweitens haben wir bei den Beispielen der univariaten Quasi-Interpolationsoperatoren im Abschnitt 3.1.2 angenommen, dass der entstehende Quasi-Interpolant Qf auf der kompletten reellen Achse $D = \mathbb{R}$ definiert sein soll. Allerdings treten in der Praxis Daten, die sich nicht in einer kompakten Teilmenge von $\mathbb{R}$ befinden, so gut wie nie auf. Normalerweise hat man es mit Daten zu tun, die sich nach Transformation im Einheitsquadrat $[0, 1]^2$ befinden – für die univariaten Beispiele bedeutet das $D = [0, 1]$. Das stellt in der Tat ein Problem dar, denn die zu interpolierende Funktion f ist nach Definition 3.1 i. A. nur auf D definiert. Die für D relevanten Funktionale des Quasi-Interpolationsoperators aus Beispiel 3.4 benötigen jedoch bis zu n Werte auf jeder Seite außerhalb von D.

Für die Lösung des zweiten Problems existieren wiederum zwei Lösungsmöglichkeiten: Die erste, aufwendigere Lösung ist die Modifikation der Funktionale, die Funktionswerte außerhalb von D benötigen. Dabei verändert man bei diesen Funktionalen die Stützstellen $x_{k,\alpha}$ in Gleichung (3.5), so dass sich diese innerhalb von D befinden, aber immer noch in $\operatorname{supp} b^n_{k,h}$ liegen. In Abbildung 3.3 ist eine mögliche Veränderung der ersten beiden Funktionale des univariaten Quasi-Interpolationsoperators aus Beispiel 3.4 zusammen mit dem ersten unveränderten Funktional für den Grad $n = 2$ und das Gebiet $D = [0, 1]$ gezeigt. Die veränderten Gewichte kann man nach der gleichen Vorgehensweise wie in Beispiel 3.4 berechnen. Natürlich könnte man die dargestellte Modifikation auch auf andere Grade verallgemeinern. Aus Platzgründen und weil diese Art der Lösung des „Problems mit dem Rand“ für unseren Algorithmus nicht verwendet wird, verzichten wir auf eine tiefer gehende Analyse dieser Lösungsmöglichkeit.

Die zweite, sozusagen „billigere“ Lösung, die wir aus Gründen der Einfachheit im Folgenden verwenden werden, besteht in der Forderung an die gegebenen Daten, dass auch außerhalb des Gebiets, auf dem der resultierende Quasi-Interpolant definiert ist, Daten gegeben sind. Diese Erweiterung des Definitionsbereichs der Daten erfolgt genau so weit, dass die Funktionale des Quasi-Interpolationsoperators, die Funktionswerte außerhalb von D benötigen, diese auch bekommen. Genauer wird auch dies im nächsten Abschnitt formuliert.

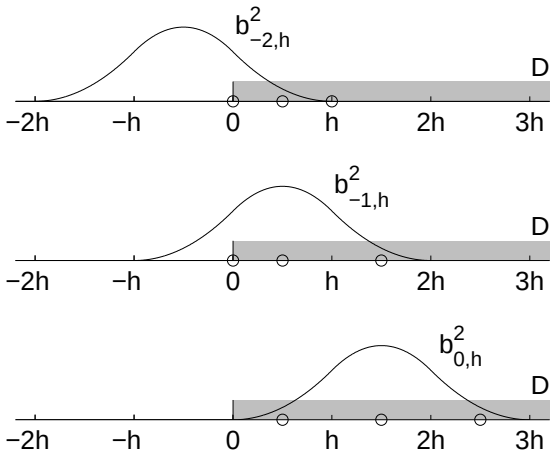


Abbildung 3.3: erste drei modifizierte Funktionale mit Stützstellen und entsprechenden B-Splines des Quasi-Interpolationsoperators aus Beispiel 3.4 für $n = 2$

3.3.2 Formulierung des Approximationsproblems und der Lösung

Die beiden eben erwähnten kleineren Schwierigkeiten gehen direkt schon in die Formulierung des zu lösenden Approximationsproblems ein. Aus Gründen der Verständlichkeit gehen wir dabei von derselben Gitterweite und demselben Grad in beiden Dimensionen aus. Es ist jedoch nicht schwierig, den vorgestellten Algorithmus auf beliebige Gitterweiten $h = (h_1, h_2)$ mit $h_\nu = 1/H_\nu$ und $H_\nu \in \mathbb{N}$ und Grade $n = (n_1, n_2)$ mit $n_\nu \in \mathbb{N}$, $\nu = 1, 2$, zu verallgemeinern.

Definition 3.10 (Approximationsproblem)**.** Gegeben seien die Gitterweite $h := 1/H > 0$ mit $H \in \mathbb{N}$, der Grad $n \in \mathbb{N}$ und die Daten

$$(x_{1,i}, x_{2,i}) \in [-rh, 1 + rh]^2, \quad f_i \in \mathbb{R}, \quad i = 1, \ldots, m, \tag{3.33}$$

mit

$$r_1 := \lceil n/2 \rceil, \quad r_2 := n, \quad r := r_1 + r_2, \tag{3.34}$$

so dass sich in jedem Gitterquadrat mindestens ein Datenpunkt befindet, das heißt, für alle $(i_1, i_2) \in \{-r, \ldots, H + r - 1\}^2$ gibt es ein $i \in \{1, \ldots, m\}$ mit

$$(x_{1,i}, x_{2,i}) \in [i_1, i_1 + 1)h \times [i_2, i_2 + 1)h. \tag{3.35}$$

Dabei zählen wir Punkte auf dem rechten oder oberen Rand als noch zum letzten Gitterquadrat gehörend, das heißt, für $i_1 = H + r - 1$ bzw. $i_2 = H + r - 1$ ersetzen wir das linke bzw. rechte Intervall durch die abgeschlossene Version.

Gesucht ist eine bivariate Spline-Approximation

$$p = \sum_{k \sim D} c_k b^n_{k,h} \in S^n_h(D) \tag{3.36}$$

auf dem Einheitsquadrat $D = [0, 1]^2$, die aus den Daten mit lokaler Polynom-Approximation und Quasi-Interpolation ermittelt wird. ◊

Zunächst einmal ist klar, dass die Einschränkung der Daten auf das erweiterte Einheitsquadrat $[-rh, 1+rh]^2$ keine Beschränkung der Allgemeinheit darstellt, denn andere Daten lassen sich einfach auf dieses Quadrat skalieren – am Ende muss man natürlich die Spline-Approximation p aus (3.36) entsprechend zurückskalieren.

Die beiden Zahlen r_1 und r_2 stellen die Anzahl der vom Algorithmus außerhalb des Gebiets $D = [0,1]^2$ zusätzlich benötigten Gitterquadrate dar. Zunächst einmal verwendet die lokale Polynom-Approximation zur Berechnung eines Gitterpunkts für n gerade genau $(n+1)^2$ Gitterquadrate mit dem Gitterquadrat in der Mitte, das den zu berechnenden Gitterpunkt enthält. Das ergibt dann in jede Richtung $n/2$ zusätzliche Gitterquadrate. Im ungeraden Fall benötigen wir immer noch mindestens $(n+1)^2$ Gitterquadrate, allerdings ist die Situation hier asymmetrisch. Damit das Gitterquadrat mit dem zu berechnenden Gitterpunkt immer noch in der Mitte liegt, verwenden wir stattdessen $(n+2)^2$ Gitterquadrate, wir tun also so, als wäre der Grad um eins größer. Für beide Fälle (n gerade und n ungerade) ergeben sich also $r_1 = \lceil n/2 \rceil$ zusätzliche Gitterquadrate in jede Richtung.

Wie oben in Abschnitt 3.3.1 bemerkt, benötigt aber auch die Quasi-Interpolation bis zu $r_2 = n$ zusätzliche Funktionswerte außerhalb des Gebiets, auf dem der Quasi-Interpolant definiert ist. Insgesamt brauchen wir also $r = r_1 + r_2$ zusätzliche Gitterquadrate in jede Richtung.

Der Algorithmus zur Bestimmung der Spline-Approximation ist aus diesen Bemerkungen vielleicht schon ersichtlich, wir formulieren ihn als Ergebnis dieses Kapitels wie folgt:

Algorithmus 3.11 (Algorithmus zur Bestimmung der Spline-Approximation)**.** Gegeben sei das Approximationsproblem aus Definition 3.10. Daraus bestimmen wir die bivariate Spline-Approximation p aus (3.36) wie folgt:

(a) *lokale Polynom-Approximation*: Für alle $(i_1, i_2) \in \{-r_2, \ldots, H + r_2 - 1\}^2$ wähle jeweils alle Punkte aus den Gitterquadraten aus (im Sinne von Definition 3.10), die die Indizes $(j_1, j_2) \in \{i_1 - r_1, \ldots, i_1 + r_1\} \times \{i_2 - r_1, \ldots, i_2 + r_1\}$ besitzen. Es sei $p_{(i_1,i_2)}$ das bivariate Polynom, das diese Punkte gemäß (3.28) approximiert. Definiere

$$f((i_1 + 1/2)h, (i_2 + 1/2)h) := p_{(i_1,i_2)}((i_1 + 1/2)h, (i_2 + 1/2)h) \tag{3.37}$$

als den Wert des Gitterpunkts an der Position $((i_1 + 1/2)h, (i_2 + 1/2)h)$.

(b) *Quasi-Interpolation*: Es sei

$$Q\colon \mathcal{C}(D) \to S_h^n(D), \quad f \mapsto Qf = \sum_{k \sim D} (Q_k f) b_{k,h}^n, \tag{3.38}$$

der bivariate Quasi-Interpolationsoperator, der durch Verwendung von Beispiel 3.4 mit Grad n in beiden Koordinaten hervorgeht. Dann ist

$$p := Qf, \tag{3.39}$$

wobei obige bei (a) definierten Daten bei der Berechnung der Koeffizienten von Qf verwendet werden. ◊

Illustrationen und weitere Erklärungen zur Funktionsweise des Algorithmus finden sich im nächsten Kapitel 4.

4 Implementierung

Ziel dieses Kapitels wird es sein, die Verwendung und die Eigenschaften der auf der beigelegten CD-ROM in MATLAB vorliegenden Implementierung des Algorithmus zur Spline-Approximation unregelmäßig verteilter Daten zu erklären. Dabei werden wir nur auf die Implementierung des grundlegenden Algorithmus 3.11 eingehen. Den Algorithmus, der als Anwendung FE-Gewichtsfunktionen approximiert, werden wir separat im nächsten Kapitel 5 erläutern.

Zunächst erklären wir in Abschnitt 4.1 den grundlegenden Aufbau der Programme sowie deren Aufruf anhand von Beispielen. In Abschnitt 4.2 werden wir den Algorithmus auf reale Daten anwenden und somit seine Praxistauglichkeit testen. Schließlich werden wir in Abschnitt 4.3 numerische Eigenschaften des Algorithmus untersuchen, zum Beispiel Konvergenz oder Zeitbedarf.

Eine Übersicht der auf der CD-ROM enthaltenen Dateien befindet sich in Anhang A.

4.1 Aufbau und Aufruf der Programme

4.1.1 Allgemeines

Die auf der CD-ROM befindliche Implementierung wurde mit der zum Zeitpunkt der Fertigstellung dieser Arbeit neuesten verfügbaren MATLAB-Version R2012a (7.14.0.739) programmiert. Beim zugrunde liegenden Betriebssystem handelt es sich um ein 64-Bit-Linux-System mit Ubuntu 12.04. Es waren zwar alle MATLAB-Toolboxes verfügbar, jedoch wurde darauf geachtet, dass das Programm auch auf einem System läuft, auf dem nur das MATLAB-Basisprogramm installiert ist. Des Weiteren sollte das Programm ohne Schwierigkeiten zudem mit älteren MATLAB-Versionen und auf anderen Betriebssystemen (wie Windows) laufen.

Konkrete Laufzeiten für einzelne Testläufe wurden auf einem zum Zeitpunkt der Fertigstellung dieser Arbeit durchschnittlichen Computer mit Vierkernprozessor (Intel Core 2 Quad Q8200, $4 \times 2{,}33\,\mathrm{GHz}$) und 4 GB Arbeitsspeicher gemessen.

Die MATLAB-Programme sind in sog. Paketen (*packages*) gegliedert, was Namenskonflikte vermeidet und die Wiederverwendbarkeit in anderen Programmen verbessert. Das Hauptpaket trägt den Namen `ScatteredData`. Darunter befinden sich verschiedene Unterpakete, davon sind am wichtigsten `ScatteredData.SplineApproximation` und `ScatteredData.WeightFunctions`. Zur Verwendung wechselt man in das Elternverzeichnis von `+ScatteredData` oder man bindet das Elternverzeichnis in den MATLAB-Pfad ein.

Eine Übersicht der vorhandenen Pakete erhält man wie üblich mit `help ScatteredData`. Der Quellcode ist mit ausführlichen Hilfetexten und Kommentaren versehen.

Das Programmpaket besitzt globale Optionen, die sich mit den Routinen `ScatteredData.change_settings` und `ScatteredData.print_settings` ändern bzw. ausgeben lassen. Die verfügbaren Optionen `use_mex` und `print_status_messages` beeinflussen die Verwendung von MEX bzw. die Ausgabe von Statusmeldungen bei der Berechnung.

4.1.2 Beispieldaten

In diesem Abschnitt werden wir die Funktionsweise der Programme anhand eines Beispiels mit fiktiven Daten erklären. Dazu wurden als Grundlage die sog. Halton-Punkte ausgewählt, die zunächst in [12] definiert wurden. Im eindimensionalen Fall heißt die entsprechende Folge van-der-Corput-Folge $(\varphi_b(n))_{n\in\mathbb{N}_0}$. Sie lässt sich folgendermaßen herleiten: Man wählt eine feste Basis $b \in \mathbb{N}$. Um $\varphi_b(n)$ für $n \in \mathbb{N}_0$ zu berechnen, schreiben wir zunächst n zur Basis b, d. h. $n = \sum_{i=0}^{m} n_i b^i = (n_m \cdots n_1 n_0)_b$. Dann ist $\varphi_b(n) := \sum_{i=0}^{m} n_i b^{-i-1}$, wir drehen also $(n_m \cdots n_1 n_0)_b$ um und lesen die entstehende Zahl als Nachkommaanteil. Eine mehrdimensionale Verallgemeinerung erhalten wir, indem wir einen Vektor $(\varphi_{b_1}(n), \ldots, \varphi_{b_d}(n)) \in [0,1)^d$ mit paarweise verschiedenen Basen b_ν bilden. In der Praxis werden meist die ersten d Primzahlen verwendet. Eine solche Folge wird Halton-Folge genannt. Sie hat den Vorteil, dass sie im Gegensatz zu pseudozufälligen Punkten das Einheitsquadrat gleichmäßig gut abdeckt, einfach zu berechnen und deterministisch ist, das heißt, hintereinander ausgeführte Aufrufe ergeben dieselben Punkte.

Ein Befehl zur Generierung von Halton-Punkten befindet sich zwar in MATLAB, allerdings nur in der Statistics Toolbox. Weil im Rahmen der vorliegenden Implementierung auf externe Toolboxes bewusst verzichtet wird, wurde eine eigene Version implementiert, die vom Programmpaket „Stochastic Simulation in Java“[1] stammt, das unter der GPLv3-Lizenz[2] verfügbar ist. Der dort verwendete Algorithmus basiert auf einer anderen Berechnungsweise, die in [27] beschrieben wird, und ist sowohl hinsichtlich Genauigkeit als auch Schnelligkeit mit der MATLAB-Version vergleichbar. Die Erzeugung von Halton-Punkten erfolgt mit

```
X = ScatteredData.Halton.gen_points(n, d, b),
```

wobei n die Anzahl der Halton-Punkte, d die Anzahl der Dimensionen und b einen Vektor bezeichnet, der die d-vielen Basen enthält (für $d \leq 3$ kann b weggelassen werden).

Die Halton-Punkte geben nur die Position der Datenpunkte im Einheitsquadrat wieder, allerdings nicht deren Funktionswerte. Um diese zur Verfügung zu stellen, existieren ebenfalls viele Möglichkeiten. Aufgrund ihrer Omnipräsenz in wissenschaftlichen Artikeln über das übergeordnete Themengebiet (z. B. [19, 28]) wird hier die bekannte Franke-Testfunktion `ScatteredData.Halton.franke` verwendet, die zuerst in [10] auf $[0,1]^2$ definiert wurde:

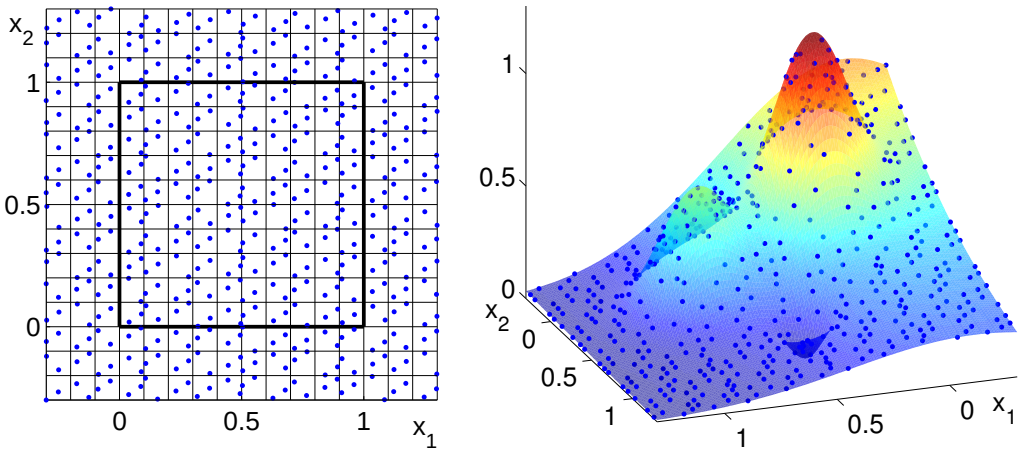
$$\begin{aligned} F(x,y) := \frac{3}{4}\exp\left(-\frac{(9x-2)^2+(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{9y+1}{10}\right) \\ - \frac{1}{5}\exp\left(-(9x-4)^2-(9y-7)^2\right) + \frac{1}{2}\exp\left(-\frac{(9x-7)^2+(9y-3)^2}{4}\right). \end{aligned} \tag{4.1}$$

Das im Folgenden verwendete Beispiel **example_halton**, nämlich die Kombination von Halton-Punkten mit der Franke-Testfunktion, kann mit der Demonstration

```
ScatteredData.Halton.demo()
```

nachvollzogen werden, wobei Einstellungen von diesem Programm interaktiv durch den Benutzer getätigt werden. Das Programm **example_halton** verwendet dabei die Standardwerte, d. h. $H = 10$ Gitterzellen pro Koordinatenrichtung in $[0,1]^2$, $q = 2$ Halton-Punkte pro Git-

[1] URL: http://www.iro.umontreal.ca/~simardr/ssj/indexe.html
[2] URL: http://www.gnu.org/licenses/gpl.txt



(a) Halton-Punkte auf $[-rh, 1+rh]^2$ **(b)** Werte der Franke-Testfunktion an den Punkten

Abbildung 4.1: $(q \cdot (H+2r)^2)$-viele Halton-Punkte und Werte der Franke-Testfunktion

terzelle, Grad $n = 2$, die Funktion `fun` = `@ScatteredData.Halton.franke` und `sd` $= 0$ (keine Addition von zufälligen Fehlern).

Abbildung 4.1 zeigt die entstehenden Datenpunkte. Gemäß Definition 3.10 werden auch außerhalb des Einheitsquadrats $[0,1]^2$, auf dem die Spline-Approximation am Ende definiert sein wird, Datenpunkte benötigt (auf r zusätzlichen Gitterquadraten in jede Richtung).

4.1.3 Ermittlung der Gitterweite

Bei realen Daten (siehe Abschnitt 4.2) wäre es laut Definition 3.10 zunächst notwendig, eine Gitterweite $h = 1/H$ mit $H \in \mathbb{N}$ zu bestimmen, so dass sich in jedem Gitterquadrat mindestens ein Datenpunkt befindet. Dies kann mit der Funktion

```
H = ScatteredData.SplineApproximation.get_opt_grid(X, n)
```

erledigt werden, wobei sich die Daten X im Einheitsquadrat $[0,1]^2$ befinden müssen. Das Programm ermittelt durch heuristische binäre Suche einen im Allgemeinen akzeptablen Wert für H, der allerdings nicht optimal sein muss (in dem Sinne, dass es kein größeres $H^* > H$ gibt, so dass obige Bedingung ebenfalls erfüllt ist), siehe auch Abbildung 4.2. Der Aufruf

```
H = ScatteredData.SplineApproximation.get_opt_grid(X, n, 'strict')
```

bewirkt dagegen die Optimalität des zurückgegebenen H.

In unserem „künstlichen“ Beispiel sind die Punkte schon so von vorneherein konstruiert, dass sich für beliebiges, gegebenes H in jedem Gitterquadrat mindestens ein Datenpunkt befindet – somit entfällt dieser Schritt bei `ScatteredData.Halton.demo()`.

4.1.4 Berechnung der Spline-Approximation

Die eigentliche Erstellung der Spline-Approximation erfolgt durch

```
[p, Y] = ScatteredData.SplineApproximation.make(X, f, H, n).
```

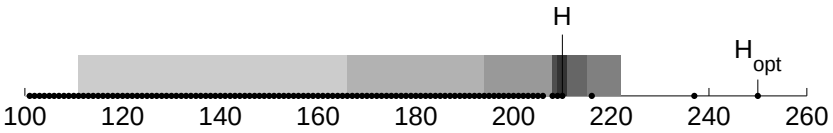


Abbildung 4.2: Funktionsweise von **get_opt_grid** am Beispiel von 200 000 Halton-Punkten. Die schwarzen Punkte auf der Zahlengerade repräsentieren die möglichen Werte für H und die verschieden hellen, ineinander verschachtelten Rechtecke stellen die durch die binäre Suche durchsuchten Intervalle dar. Der am Ende zurückgegebene Wert H unterscheidet sich etwas vom optimalen Wert H_{opt}, der beim Aufruf mit `'strict'` zurückgegeben wird.

$X = (x_{1,i}, x_{2,i})_{m\times 2}$ und $f = (f_1, \ldots, f_m)^T$ enthalten die Daten und H und n bestimmen die Gitterweite $h = 1/H$ bzw. den Approximationsgrad n. `ScatteredData.SplineApproximation.make` führt eigentlich nur zwei Zeilen aus:

Bei der ersten Zeile

```
Y = ScatteredData.SplineApproximation.gen_grid_data(X, f, H, n)
```

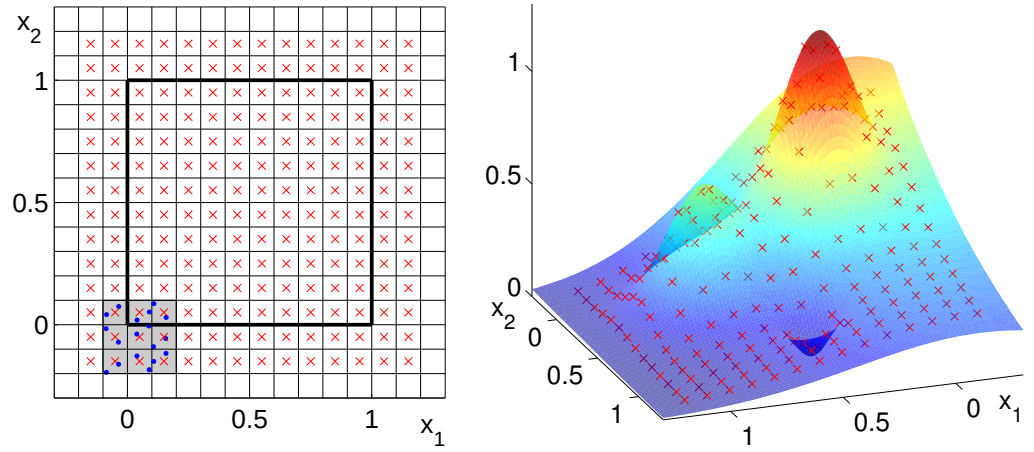
handelt es sich um den ersten Schritt in Algorithmus 3.11, die Generierung der Gitterdaten durch lokale Polynom-Approximation. Zuerst werden dabei die Daten in die Gitterzellen sortiert, das heißt, für jeden Datenpunkt $(x_{1,i}, x_{2,i})$ werden die Indizes des Gitterquadrats berechnet, in dem sich der Punkt befindet. Bei diesem Vorgang werden auch diverse Validierungen vorgenommen, zum Beispiel, ob sich alle Daten im Quadrat $[-rh, 1+rh]^2$ befinden und ob jedes Gitterquadrat mindestens einen Datenpunkt enthält. Wie es schon in Algorithmus 3.11 beschrieben wird, wird anschließend für jedes Gitterquadrat Polynom-Approximation mit den relevanten Punkten durchgeführt. Dazu werden die möglichen Monome in diesen Punkten ausgewertet und die Koeffizienten des bivariaten Polynoms mittels des Backslash-Operators ermittelt.

In Abbildung 4.3b ist das Ergebnis für unser Beispiel **example_halton** abgebildet, zur Veranschaulichung zusammen mit der Franke-Testfunktion. In Abbildung 4.3a sieht man, für welche Gitterquadrate gemäß Algorithmus 3.11 lokale Polynom-Approximation durchgeführt wurde: Auf jeder Seite bleibt ein Rand von r_1 Gitterquadraten frei. Zugleich sind in dieser Abbildung als Beispiel die Datenpunkte aus den $(2r_2+1)^2$-vielen Gitterquadraten hervorgehoben, die bei der lokalen Polynom-Approximation für das Gitterquadrat mit Index $(0, -1)$ relevant sind (zu den Indizes siehe Algorithmus 3.11).

Der zweite Schritt, der durch `ScatteredData.SplineApproximation.make` umgesetzt wird, ist die Quasi-Interpolation, die mittels

```
p = ScatteredData.SplineApproximation.quasi_interp(Y, H, n)
```

durchgeführt wird. Die Funktion benötigt die Punkte Y aus dem vorherigen Schritt und gibt die Spline-Approximation p als Struktur in einem Format zurück, das schon in der von Jörg Hörner entwickelten Funktion `ScatteredData.SplineUtil.spl_eval_matlab` (ursprünglich `spl_eval`) verwendet wird. Hauptsächlich besteht der Aufwand für `ScatteredData.SplineApproximation.quasi_interp` natürlich darin, die Kontrollpunkte des Splines gemäß Definition 3.5 zu berechnen. Als univariate Gewichte werden die Gewichte von Beispiel 3.4 in Tabelle 3.1 verwendet, die für $n = 1, \ldots, 4$ eingespeichert sind. Für $n \geq 5$ werden sie dynamisch



(a) Draufsicht mit relevanten Punkten für das Gitterquadrat mit Index $(0, -1)$

(b) durch lokale Polynom-Approximation erzeugte Punkte zusammen mit Franke-Testfunktion

Abbildung 4.3: Ergebnis nach der lokalen Polynom-Approximation

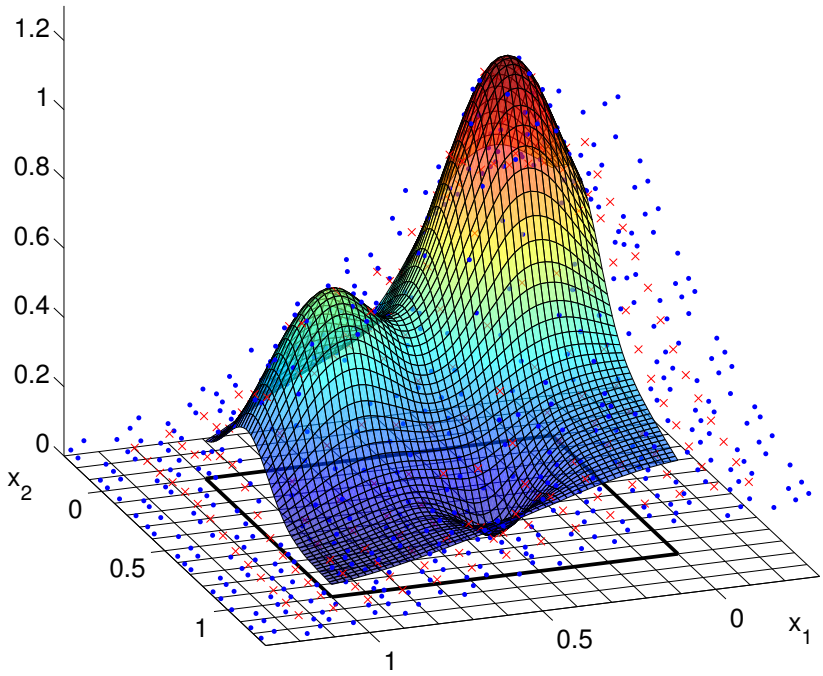


Abbildung 4.4: Ergebnis der Spline-Approximation für `example_halton`

als Lösung des zugehörigen LGS (3.12) berechnet, wobei solch hohe Approximationsgrade in der Praxis selbstverständlich nur selten eingesetzt werden.

Abbildung 4.4 (oder auch das Titelbild, bloß ohne Beschriftungen und aus einer anderen Perspektive) zeigt die resultierende Spline-Approximation für das Beispiel mit den Halton-Punkten und der Franke-Testfunktion. Man sieht, dass sich der Definitionsbereich des Splines auf $[0, 1]^2$ erstreckt, genau so, wie es von uns beabsichtigt war.

4.1.5 Evaluation und Visualisierung

Zur Evaluation des Splines steht die Funktion

```
e = ScatteredData.SplineUtil.spl_eval(p, t)
```

zur Verfügung. Bei aktivierter Option `use_mex` (siehe Abschnitt 4.1.1), die standardmäßig eingeschaltet ist, wird eine in C++ implementierte Version des Algorithmus von de Boor (siehe Algorithmus 2.7) via MEX aufgerufen, die aufgrund Multithreading auf Mehrkernprozessoren einen deutlichen Geschwindigkeitsvorteil erbringt. Binaries (fertig kompilierte Binärdateien) stehen für 64-Bit-Linux und 64-Bit-Windows zur Verfügung. Falls Multithreading nicht erwünscht oder die Multithreading-Bibliothek `pthread.h` nicht verfügbar ist, kann in der Quellcode-Datei ganz oben die Zeile `#define SPL_EVAL_MEX_MULTITHREADING` auskommentiert werden. Dort kann auch die Anzahl der Threads eingestellt werden.

Wenn die Option `use_mex` deaktiviert ist, dann wird die weiter oben erwähnte Routine `ScatteredData.SplineUtil.spl_eval_matlab` von Jörg Hörner verwendet.

Die Visualisierung des entstehenden Splines geschieht durch

```
ScatteredData.SplineApproximation.visualize(X, f, H, n, p, Y).
```

Das Ergebnis sieht dann so ähnlich aus wie in Abbildung 4.4.

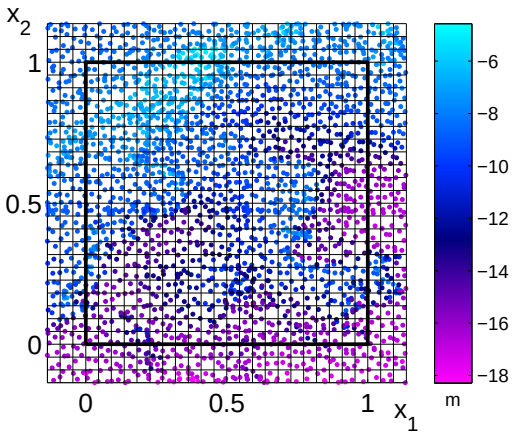
4.2 Beispiele aus der Wirklichkeit

Nach der Erklärung der Funktionalität des Programms folgen nun ein paar Illustrationen mit realen Daten. Alle Darstellungen sind zur besseren Veranschaulichung stark überhöht.

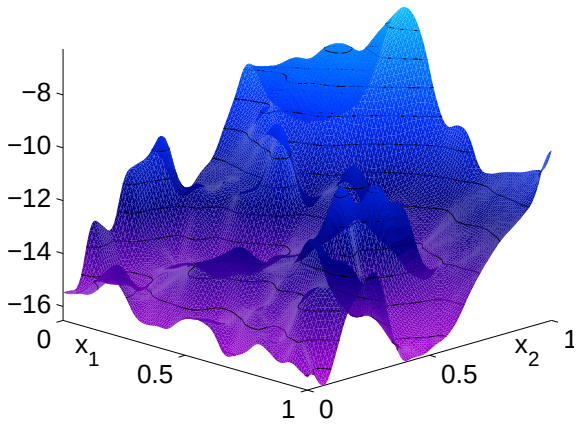
Beispiel 4.1. Abbildung 4.5a zeigt 2590 Datenpunkte aus der Bathymetrie (Topografie von Meeresböden) eines ca. $58\,\mathrm{km} \times 36\,\mathrm{km} = 2088\,\mathrm{km}^2$ großen Gebiets vor der amerikanischen Meeresküste von Texas, ungefähr 300 km südöstlich von Houston. Die Daten sind vom Stand von 1994 und stammen vom *Texas General Land Office (GLO)*, wobei die Koordinaten auf Daten der *National Oceanic and Atmospheric Administration (NOAA)* basieren. Sie wurden vom Internetprojekt *Koordinates*[3] aufbereitet und stehen unter der Creative-Commons-Attribution-3.0-Lizenz[4]. Die Daten wurden auf das Quadrat $[-rh, 1+rh]^2$ mit $h = 1/22$ und $n = 2$ transformiert. Die Farbe in den Abbildungen 4.5a und 4.5b und die Werte der vertikalen Achse in Abbildung 4.5b, die die resultierende biquadratische Spline-Approximation zeigt, entsprechen der Tiefe unter dem Meeresspiegel in Metern. ◊

Beispiel 4.2. In Abbildung 4.6a sieht man 5802 Datenpunkte aus dem in der Literatur (siehe zum Beispiel [8, 19, 28]) bekannten, im Internet verfügbaren[5] „Gletscher"-Datensatz `vol87` von Richard Franke. Die Farbe bzw. vertikale Koordinate in Abbildung 4.6b gibt die Höhe an, wahrscheinlich in Metern. Wie aus Abbildung 4.6a ersichtlich ist, liegen die Daten als Isohypsen (Linien gleicher Höhe) im Abstand von 25 m vor. Leider ist nicht mehr bekannt, z. B. wo der Gletscher liegt, wann und wie die Auswertung erfolgte oder wie groß das betrachtete Gebiet ist. Abbildung 4.6b zeigt die biquadratische Spline-Approximation mit $h = 1/10$ und Höhenlinien im Abstand von 50 m. ◊

[3] URL: `http://koordinates.com/layer/786-texas-bathymetry-1994/`
[4] URL: `http://creativecommons.org/licenses/by/3.0/`
[5] URL: `http://personal.strath.ac.uk/oleg.davydov/Franke_test_data/README`

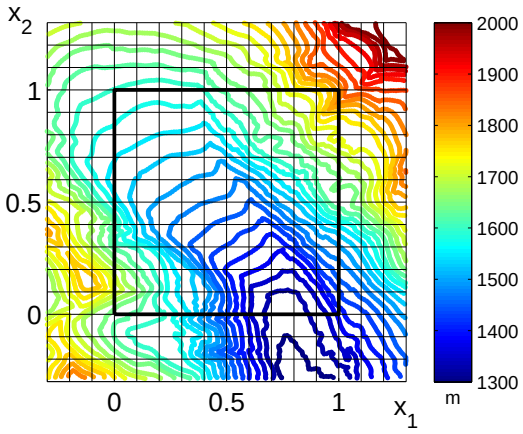


(a) Position der Datenpunkte in der Draufsicht

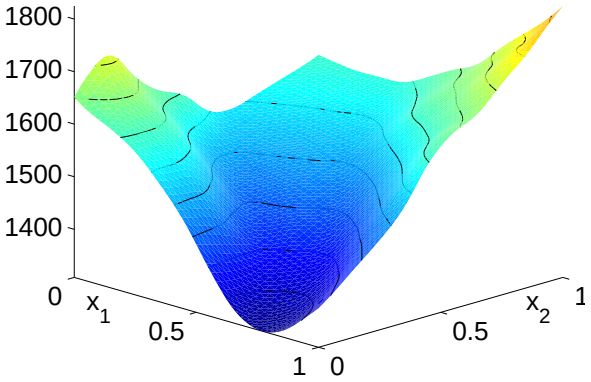


(b) Ergebnis der Spline-Approximation

Abbildung 4.5: Bathymetrie eines Ausschnitts des Meeresbodens vor der Küste von Texas

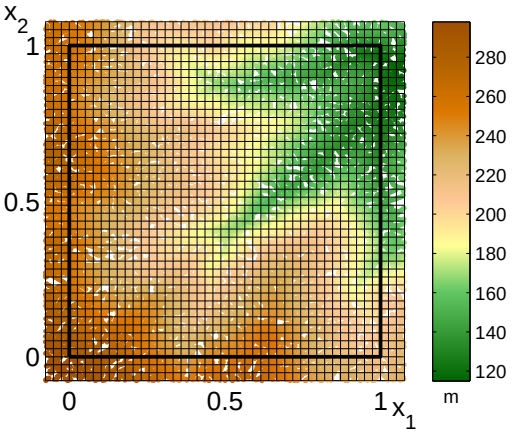


(a) Position der Datenpunkte in der Draufsicht

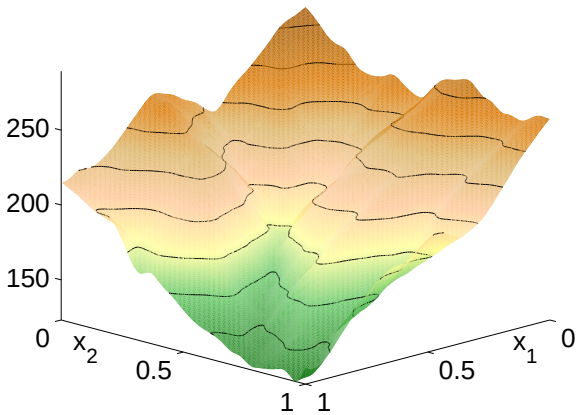


(b) Ergebnis der Spline-Approximation

Abbildung 4.6: Höhenpunkte eines Gletschers



(a) Position der Datenpunkte in der Draufsicht



(b) Ergebnis der Spline-Approximation

Abbildung 4.7: Höhenpunkte aus dem *George Denton Park* nahe Wellington, Neuseeland

Tabelle 4.1: maximaler bzw. mittlerer absoluter Fehler e_h bzw. $\overline{e_h}$ und maximaler bzw. mittlerer relativer Fehler ε_h bzw. $\overline{\varepsilon_h}$ für obige Beispiele (gerundet)

Fehlerart	Texas	Gletscher	Wellington
e_h	2,84 m	37,98 m	3,58 m
$\overline{e_h}$	0,42 m	5,10 m	0,36 m
ε_h	3,67 %	2,08 %	2,58 %
$\overline{\varepsilon_h}$	2,09 %	0,33 %	0,19 %

Beispiel 4.3. Als abschließendes Beispiel sind in Abbildung 4.7a 19 293 Höhenpunkte aus dem südwestlich an die neuseeländische Hauptstadt Wellington direkt angrenzenden *George Denton Park* zu sehen. Die Daten sind vom Stand von 2009 und wurden mittels Lidar (*light detection and ranging*), einer dem Radar verwandten Methode zur Entfernungsbestimmung, und Fotogrammetrie ermittelt und durch den Stadtrat von Wellington (*Wellington City Council (WCC)*) veröffentlicht. Der vorliegende Ausschnitt der Höhenlinien, die mit nur 1 m Abstand sehr genau sind, ist in der Realität ca. 335 m × 445 m ≈ 14,9 ha groß. Diese Daten wurden ebenfalls vom Internetprojekt *Koordinates*[6] aufbereitet und stehen unter der Creative-Commons-Attribution-3.0-Neuseeland-Lizenz[7]. Die Anzahl der Gitterquadrate pro Dimension in $[0,1]^2$ wurde optimal gewählt (siehe Abschnitt 4.1.3) und ist mit $H = 39$ der guten Auflösung der Daten entsprechend hoch. In Abbildung 4.7b ist das Ergebnis der biquadratischen Spline-Approximation mit Höhenlinien im Abstand von 20 m sichtbar. ◊

4.3 Numerische Aspekte

4.3.1 Genauigkeit

In Tabelle 4.1 sieht man den maximalen und den durchschnittlichen absoluten und relativen Fehler der Beispiele aus dem vorherigen Abschnitt, jeweils ausgewertet an allen zu approximierenden Punkten in $[0,1]^2$.

Man kann erkennen, dass eine kleinere Gitterweite bei verschiedenen Daten nicht unbedingt zu einem kleineren Fehler führt. Beispielsweise ist der Fehler beim Beispiel „Texas“ größer wie beim Beispiel „Gletscher“, obwohl beim einen Beispiel $H = 22$ und beim anderen $H = 10$ gewählt wurde. Dies hat einfach damit zu tun, wie fein die Datenpunkte im Verhältnis zu den feinen Strukturen im Gelände aufgelöst sind. Das Beispiel „Texas“ enthält starke Geländeschwankungen, die aber nur durch wenige Punkte sichtbar sind. Der Verlauf des Gletschers ist dagegen glatter und die Daten besitzen eine höhere Auflösung. Die Genauigkeit hängt also im Wesentlichen von der Komplexität der betrachteten Daten sowie von deren Auflösung ab.

Insgesamt lässt sich jedoch sagen, dass die Genauigkeit des Verfahrens recht gut ist, besonders, wenn wir die mittleren relativen Fehler für den Gletscher und für Wellington betrachten.

[6] URL: `http://koordinates.com/layer/1479-wellington-city-1m-contours-2009/`
[7] URL: `http://creativecommons.org/licenses/by/3.0/nz/`

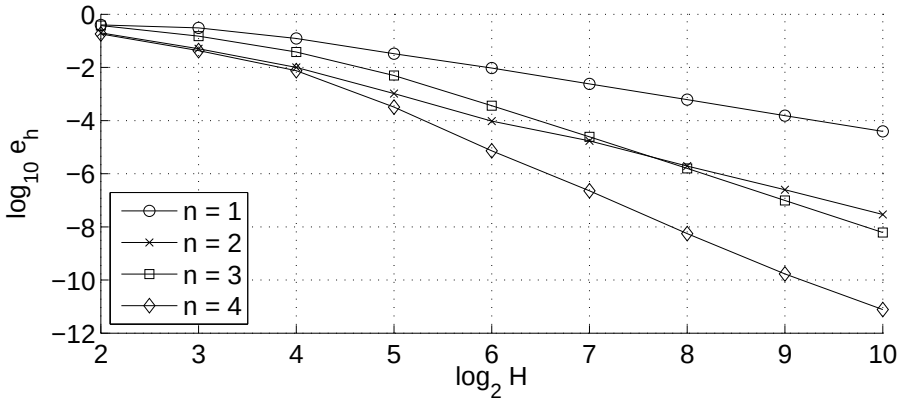


Abbildung 4.8: doppeltlogarithmischer Plot des maximalen absoluten Fehlers

Tabelle 4.2: numerische Konvergenzraten (gerundet)

n	Konvergenzraten							
1	0,3684	1,3205	1,9166	1,7762	1,9832	1,9699	2,0031	1,9676
2	1,9937	2,3187	3,2701	3,4545	2,4765	3,1499	2,9578	3,0927
3	1,3131	1,9934	2,9270	3,7701	3,8943	3,9607	3,9940	4,0056
4	2,0779	2,5084	4,5660	5,4593	5,0032	5,3473	5,0443	4,4441

4.3.2 Konvergenz

Abbildung 4.8 zeigt den maximalen absoluten Fehler e_h, der sich durch Ausführung von `ScatteredData.Halton.demo` mit unterschiedlichen Gitterweiten $h = 1/H$ und Approximationsgraden n ergibt (mit $q = 2$, `fun` = `@ScatteredData.Halton.franke` und `sd` = 0). Der Fehler ist jeweils an den Halton-Punkten gemessen. In Tabelle 4.2 sieht man die entsprechenden numerischen Konvergenzraten, also die Zweierlogarithmen $\log_2 \frac{e_h}{e_{h/2}}$ der Quotienten eines Fehlers mit dem Fehler bei halber Gitterweite. Sie liefern einen Anhaltspunkt für die tatsächliche Konvergenzrate q, mit der $e_h \leq ch^q$ gilt, und entsprechen den unterschiedlichen Steigungen in der Abbildung.

Aus der Abbildung ist ersichtlich, dass das Verfahren für diese Daten tatsächlich konvergiert. Tabelle 4.2 zeigt, dass sich die Konvergenzrate auf $n + 1$ einpendelt. Es handelt sich dabei um die Ordnung, die von Satz 3.7 vorausgesagt wird. Natürlich berücksichtigt der Satz den Fehler der lokalen Polynom-Approximation nicht, aber wenn die Punkte nicht allzu ungünstig liegen und die Datenpunkte wie hier einer Funktion entsprechen, die lokal wie ein Polynom „aussieht“, dann ist der Einfluss des Fehlers der lokalen Polynom-Approximation nicht sonderlich groß.

Dass bei sehr kleiner Gitterweite h der Fehler insbesondere für größere Grade (hier bei $n = 4$ zu sehen) wieder langsamer konvergiert, liegt daran, dass in diesem Bereich Rundungsfehler eine immer größere Rolle spielen. In unserem Beispiel bei $n = 4$ und $H = 2^{10}$ muss für jeden zu generierenden Gitterpunkt das Residuum eines LGS mit einer (50×25)-Matrix minimiert werden, deren Einträge aufgrund der hohen Monom-Potenzen sehr klein sind (bis 10^{-30}). Dementsprechend gibt MATLAB viele Warnungen der Form „*rank deficient*“ aus.

Tabelle 4.3: Ausführungszeiten (gerundet)

$\log_2 H$	2	3	4	5	6	7	8	9	10
gen_grid_data	0,01 s	0,02 s	0,04 s	0,11 s	0,34 s	1,29 s	4,94 s	19,1 s	76,0 s
quasi_interp	0,00 s	0,00 s	0,01 s	0,02 s	0,06 s	0,15 s	0,55 s	1,85 s	6,64 s
spl_eval	0,01 s	0,01 s	0,03 s	0,07 s	0,22 s	0,97 s	4,59 s	24,2 s	144 s
insgesamt	0,02 s	0,03 s	0,08 s	0,19 s	0,62 s	2,41 s	10,1 s	45,2 s	227 s

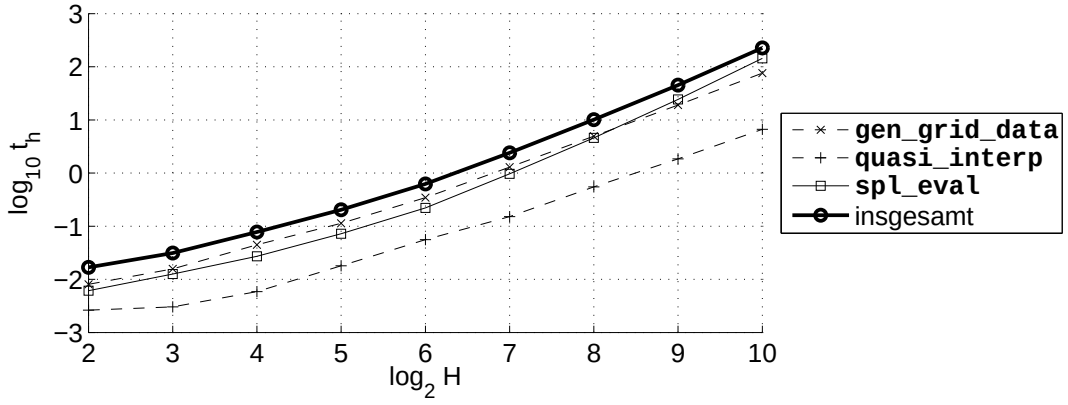


Abbildung 4.9: doppeltlogarithmischer Plot der Ausführungszeiten

4.3.3 Geschwindigkeit

Zum Abschluss des Kapitels wollen wir die Geschwindigkeit der vorliegenden Implementierung betrachten. In Tabelle 4.3 und Abbildung 4.9 sieht man die Ausführungszeiten von `ScatteredData.Halton.demo` mit Parametern wie eben, aufgeschlüsselt in die einzelnen Abschnitte **`gen_grid_data`**, **`quasi_interp`** und **`spl_eval`**. Die letzte Funktion ist die Auswertung der Spline-Approximation auf einem Gitter mit einem Fünftel der ursprünglichen Gitterweite (z. B. für ein eventuelles Plotten).

Am meisten Zeit benötigen die Generierung der Gitterdaten und die Auswertung des Splines. Wir können anhand der Zeiten erkennen, dass **`gen_grid_data`** in der Zeit $\mathcal{O}(H^2)$ arbeitet (Anzahl der Gitterpunkte steigt quadratisch). **`quasi_interp`** arbeitet ungefähr in $\mathcal{O}(H^{1,85})$ (dem Algorithmus nach eigentlich in $\mathcal{O}(H^2)$, aber MATLAB scheint den Code zu beschleunigen), bei **`spl_eval`** ist die Zeitkomplexität aus den Daten nicht ersichtlich (eigentlich $\mathcal{O}(H^2)$, evtl. verlangsamen Speicherallokationen die Ausführung).

An dieser Stelle sei angemerkt, dass die Implementierung mit Hilfe des MATLAB-Profilers auf Geschwindigkeit optimiert wurde. Zum Beispiel konnte **`gen_grid_data`** durch schnellere Bestimmung der für einen Gitterpunkt relevanten Datenpunkte um gleich mehrere Größenordnung beschleunigt werden. Einen weiteren Vorteil bietet die weiter oben erklärte Spline-Auswertungs-Funktion `spl_eval`, die (die richtige Einstellung von `use_mex` vorausgesetzt) mittels MEX eine parallelisierte Version des Algorithmus von de Boor in C++ aufruft.

Wir halten abschließend fest, dass der Algorithmus für „normale“ Gitterweiten h eine durchaus ordentliche Geschwindigkeit besitzt.

5 Anwendung bei Gewichtsfunktionen für Finite Elemente

In diesem letzten Kapitel werden wir als Anwendung des Algorithmus die Approximation von Gewichtsfunktionen für Finite Elemente betrachten. Die Natur und den Zweck dieser Gewichtsfunktionen werden wir zunächst in Abschnitt 5.1 erklären. Anschließend folgt, ähnlich wie im vorherigen Kapitel, in Abschnitt 5.2 eine Beschreibung des Aufbaus und der Art des Aufrufs der Programme auf beiliegender CD-ROM. Zum Schluss werden wir in Abschnitt 5.3 kurz die Genauigkeit und die Geschwindigkeit des Algorithmus untersuchen.

5.1 Motivation

5.1.1 Finite Elemente

Die Methode der Finiten Elemente (FE) ist eine Methode zur Lösung von sog. partiellen Differentialgleichungen (DGL). Diese tauchen besonders häufig in der Physik auf, kommen aber auch in anderen Naturwissenschaften wie Chemie und Biologie vor. Ein Beispiel ist die Poisson-Gleichung $-\Delta u = f$, mit der die Auslenkung einer am Rand fest eingespannten Membran beschrieben werden kann, auf die eine bestimmte Kraft wirkt. Ein anderes Beispiel ist die eindimensionale Wärmeleitungsgleichung $u_t = u_{xx}$, die als Verallgemeinerung der Poisson-Gleichung die Temperaturverteilung in einem Stab beschreibt.

Für eine Einführung in die FE-Methode sei man auf [13] verwiesen. Einen Überblick über die verschiedenen Methoden zur Lösung von partiellen Differentialgleichungen findet man in [24].

Zur Motivation der Gewichtsfunktionen verwenden wir das Modellproblem aus [13], die Poisson-Gleichung mit homogenen Dirichlet-Randbedingungen

$$-\Delta u = f \text{ in } D, \quad u = 0 \text{ auf } \partial D, \tag{5.1}$$

für ein Gebiet $D \subset \mathbb{R}^m$. Durch Multiplikation mit einer sog. Testfunktion v mit $v|_{\partial D} = 0$ und partielle Integration erhält man die schwache Formulierung des Poisson-Problems:

$$\forall_v \int_D \langle \nabla u, \nabla v \rangle = \int_D fv, \tag{5.2}$$

wobei die Testfunktionen v aus einem geeigneten Raum stammen, zum Beispiel aus einem sog. Sobolev-Raum. Indem man einen geeigneten (endlich-dimensionalen) Teilraum $\mathbb{B}_h$ zur Approximation verwendet, v durch $v_h \in \mathbb{B}_h$ ersetzt und u durch eine Linearkombination

$$u_h = \sum_k u_k b_k \tag{5.3}$$

einer Basis $\{b_k\}_k$ von $\mathbb{B}_h$, bestehend aus den sog. Finiten Elementen, approximiert, erhält man ein LGS, das nach den Koeffizienten u_k gelöst werden kann (Ritz-Galerkin-Approximation).

Der Standard-FE-Ansatz besteht darin, das Gebiet D zu triangulieren und anschließend auf diesen Elementen lineare, quadratische oder kubische Ansatzfunktionen zu definieren. Allerdings kann die Erzeugung der Triangulierung je nach Komplexität des Gebiets mehr oder weniger diffizil sein. Aus diesem Grund wäre es wünschenswert, eine Basis für $\mathbb{B}_h$ zu verwenden, die auf einem regulären Gitter definiert ist. Es erscheint zunächst naheliegend, aufgrund der vielen guten Eigenschaften, von denen ein paar bereits in Kapitel 2 aufgezählt wurden, eine B-Spline-Basis zu verwenden.

5.1.2 Gewichtsfunktionen

Leider bringen die rechteckigen Träger der Tensorprodukt-B-Splines mehrere Nachteile mit sich (siehe [13]). Da es B-Splines gibt, deren Träger nur einen kleinen Schnitt mit dem Gebiet besitzt, treten Stabilitätsprobleme auf, die mit der Einführung von gewichteten erweiteren B-Splines (WEB-Splines) gelöst werden können (siehe [15]). Ein anderes Problem ist, dass die B-Splines per se nicht die homogenen Randbedingungen erfüllen.

Eine Lösung des zweiten Problems besteht in der Multiplikation

$$B_k := wb_k \tag{5.4}$$

der Basis mit einer gemeinsamen Gewichtsfunktion w, die auf dem Rand verschwindet. In [13] werden einige Voraussetzungen an Gewichtsfunktionen w der Ordnung $\gamma \in \mathbb{N}_0$ gestellt: Danach sollte $w(x)$ u. a. stetig auf $\overline{D}$ sein und für bestimmte Konstanten $c_1, c_2 > 0$

$$c_1 d(x) \leq w(x) \leq c_2 d(x), \quad d(x) := \operatorname{dist}(x, \Gamma)^\gamma, \tag{5.5}$$

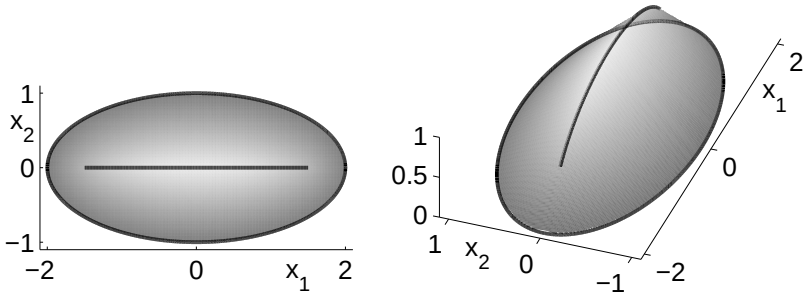
für alle $x \in D$ und eine feste Teilmenge $\Gamma \subset \partial D$ erfüllen. Insbesondere ist w im Inneren des Gebiets D positiv. Für die meisten Anwendungen ist $\gamma = 1$ (Standard-Gewichtsfunktion), in diesem Fall verschwindet $w(x)$ linear auf dem Rand.

Rvachev entwickelte ab den 1960er-Jahren die Methode der sog. R-Funktionen (Überblick siehe [25]). Mehrere vorzeichenbehaftete Gewichtsfunktionen, die im Inneren von D positiv und außerhalb von $\overline{D}$ negativ sind, für verschiedene Gebiete können zu einer Gewichtsfunktion verschmolzen werden, die einem Gebiet entspricht, das durch boolesche Operationen (wie Komplement und Schnitt) aus den einzelnen Gebieten hervorgeht.

Für Gebiete, die durch eine beliebige Kurve begrenzt sind, muss die Gewichtsfunktion anderweitig konstruiert werden. Die Abstandsfunktion selbst können wir in der Praxis nicht verwenden, weil sie schon für einfachste Gebiete unschöne „Knicke“ aufweist (siehe Abbildung 5.1). Außerdem ist sie für komplexe Gebiete schwierig zu berechnen, theoretisch müsste man für jeden Auswertungspunkt ein nichtlineares Minimierungsproblem lösen.

Eine Lösungsmöglichkeit ist folgende Methode, die in [13] beschrieben wird: In einem kleinen Streifen nahe dem Rand des Gebiets, wo die Abstandsfunktion noch glatt ist, verwenden wir die wahre Abstandsfunktion $d(x)$. Auf dem übrigen Gebiet wird die konstante Einsfunktion benutzt, wobei auf dem Randstreifen glatt überblendet wird. Das kann durch folgende Formel bewerkstelligt werden:

$$w(x) := 1 - \max(0, 1 - d(x)/\delta)^\gamma, \quad d(x) := \operatorname{dist}(x, \Gamma). \tag{5.6}$$



(a) Draufsicht **(b)** Schrägsicht

Abbildung 5.1: Abstandsfunktion für eine Ellipse mit dem nicht-differenzierbaren „Knick“ in der Mitte. Man erkennt ihn am leichtesten auf der x_2-Achse $x_1 = 0$: Dort gilt trivialerweise $d(x_1, x_2) = 1 - |x_2|$.

Dabei beeinflussen die Parameter δ und γ die Breite des Streifens bzw. die Glattheit des Übergangs auf dem inneren Rand des Streifens. δ sollte nach [13] kleiner als der minimale Krümmungsradius und kleiner als die Hälfte der Breite von kleinen Kanälen gewählt werden – zu klein sollte δ aber auch nicht sein, damit die Gradienten nicht zu groß werden.

Die so definierte Gewichtsfunktion hängt von der Abstandsfunktion $d(x)$ ab. Um $w(x)$ schnell berechnen zu können, bietet es sich an, $d(x)$ durch eine Spline-Approximation $d_h(x)$ im Sinne von Definition 3.10 und Algorithmus 3.11 zu ersetzen. Genau das verwirklicht das in diesem Kapitel vorgestellte Programm für durch Spline-Kurven berandete Gebiete. Programmiertechnisch können wir bei der Berechnung auf das im letzten Kapitel erklärte Programm zurückgreifen, die wesentliche Schwierigkeit ist natürlich die Erzeugung der Daten.

5.2 Aufbau und Aufruf der Programme

5.2.1 Allgemeines

Für die Programme für die Approximation von Gewichtsfunktionen gilt sinngemäß, was in Abschnitt 4.1.1 genannt wurde. Die für dieses Kapitel relevanten Programme befinden sich (größtenteils) im Unterpaket `ScatteredData.WeightFunctions`. Genaue Hinweise zur Benutzung erhält man, indem man zum Beispiel `help ScatteredData.WeightFunctions.gen_data` eintippt. Außerdem könnten die umfangreichen Kommentare im Quelltext der Programme bei der Lösung von eventuellen Problemen hilfreich sein.

5.2.2 Beispiel und GUI

Wie in Abschnitt 5.1 erklärt, erstellt das Programm zu einem Gebiet, das durch eine geschlossene Spline-Kurve p berandet ist, eine durch (5.6) definierte Approximation der Gewichtsfunktion. „Geschlossen“ heißt dabei, dass p stetig und T-periodisch für ein $T > 0$ ist, also $p(t + T) = p(t)$ für alle $t \in \mathbb{R}$. Auch rationale Spline-Kurven (eine Erweiterung von Spline-Kurven) werden unterstützt. Bei diesen definiert man zusätzlich zu den zweidimensionalen Kontrollpunkten skalare Gewichte und bildet dann den Quotienten aus dem Spline mit gewichteten Kontrollpunkten und dem Spline, der nur die Gewichte als Koeffizienten besitzt.

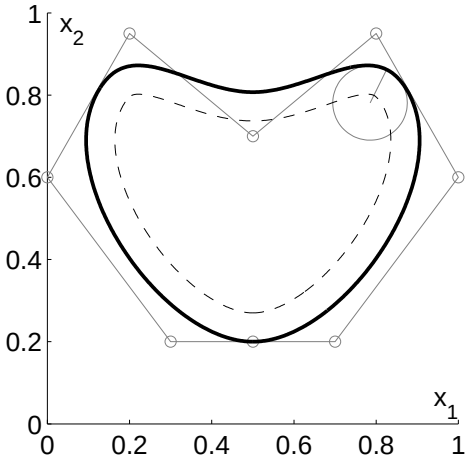


Abbildung 5.2: Rand des Testgebiets in **example_weight_function**

Sie ermöglichen eine größere Gestaltungsfreiheit als nur mit Spline-Kurven: Zum Beispiel können alle Kegelschnitte (bspw. Kreise) durch quadratische, rationale Spline-Kurven exakt dargestellt werden, was mit herkömmlichen Spline-Kurven nicht möglich ist (siehe [14]).

Das Beispiel, anhand dessen die Funktionsweise der Programme im Folgenden erläutert wird, wird durch das Programm **example_weight_function** berechnet. Die verwendete rationale Spline-Kurve vom Grad 3 besitzt sieben unterschiedliche Kontrollpunkte und ist zusammen mit dem Kontrollpolygon (grau) in Abbildung 5.2 dargestellt.

Gleichwohl wäre es ziemlich mühsam, sich die Kontrollpunkte nur im Texteditor zu überlegen – besser wäre es, über eine grafische Oberfläche die Spline-Kurve mit der Maus zu zeichnen und durch einen Knopf direkt das Ergebnis zu sehen. Ebendas bewerkstelligt

```
ScatteredData.WeightFunctions.demo().
```

Beim Aufruf dieser grafischen Benutzeroberfläche (GUI) erscheinen drei Grafikfenster. Im ersten Fenster kann man, wie in Abschnitt 5.2.3 erklärt, den Spline-Rand mit der Maus zeichnen. Das zweite Fenster besitzt auf der rechten Seite einige Schalter, mit denen die Gitterweite H, der Approximationsgrad n sowie die Parameter δ und γ aus Gleichung (5.6) eingestellt werden können. Auf der linken Seite ist der gezeichnete Rand in fett sichtbar, zusammen mit dem kleinsten Krümmungskreis, zugehörigem Krümmungsradius sowie mit dem Rand des Streifens mit Breite δ als gestrichelte Linie, damit δ korrekt eingestellt werden kann – ähnlich wie in Abbildung 5.2 (nur ohne Kontrollpolygon). Bei einem Klick auf „*Calculate*“ erscheint die Spline-Approximation von w aus (5.6) im dritten Fenster. Mit dem Knopf unten in diesem Fenster kann zwischen der Approximation und den Rohdaten gewechselt werden.

Beendet wird das Programm mit einem Klick auf die entsprechende Taste im zweiten Fenster oder mit dem Schließen eines der Fenster (dann werden alle Fenster geschlossen).

5.2.3 Erstellung des Spline-Rands

Der erste Schritt, der zur Erzeugung der Approximation einer Gewichtsfunktion nötig ist, ist natürlich die Erstellung des Spline-Rands. Entweder man macht dies per Hand direkt im Code (wie in **example_weight_function**) oder man benutzt die von Jörg Hörner zur Verfügung gestellte grafische Funktion

`ScatteredData.SplinePlotter.splineplotter()`,

die auch von `ScatteredData.WeightFunctions.demo` verwendet wird. Sie speichert den Spline in einer globalen Variable namens `SP_Data`, wobei sich das Format von der im übrigen Programmpaket von `ScatteredData` verwendeten Struktur aus `ScatteredData.SplineUtil.spl_eval_matlab` (siehe Abschnitt 4.1.4) etwas unterscheidet. Die Umrechnung in dieses Format erfolgt durch

`s = ScatteredData.SplinePlotter.convert_sp_data(SP_Data)`.

Die Bedienung von `splineplotter` gestaltet sich beinahe von selbst: Mit einem Linksklick in den leeren Bereich setzt man Kontrollpunkte (in Schwarz). Per Rechtsklick kann ein bestehender Kontrollpunkt verschoben werden. Kontrollpunkte lassen sich mit der entsprechenden Schaltfläche am unteren Rand wieder löschen. Der Grad lässt sich mit dem Schieberegler daneben einstellen. Um die anfangs uniformen Knoten zu verändern, kann man mit der rechten Maustaste die Striche unter der Spline-Kurve verschieben. Mit der mittleren Maustaste können die magentafarbenen Quadrate verschoben werden, die die Gewichte der rationalen Spline-Kurve repräsentieren. Die Verschiebung zweier Quadrate zu einem Kontrollpunkt hin erhöht sein Gewicht und „zieht“ somit die Kurve in Richtung des Punktes.

Für unsere Anwendung ist es notwendig, dass die Spline-Kurve durch Betätigung des entsprechenden Schalters geschlossen wird. Außerdem sind die Kontrollpunkte unbedingt im mathematisch positiven Sinne (gegen den Uhrzeigersinn) zu setzen, weil sonst die berechneten Einheitsnormalen in die falsche Richtung zeigen.

5.2.4 Erzeugung der unregelmäßig verteilten Daten

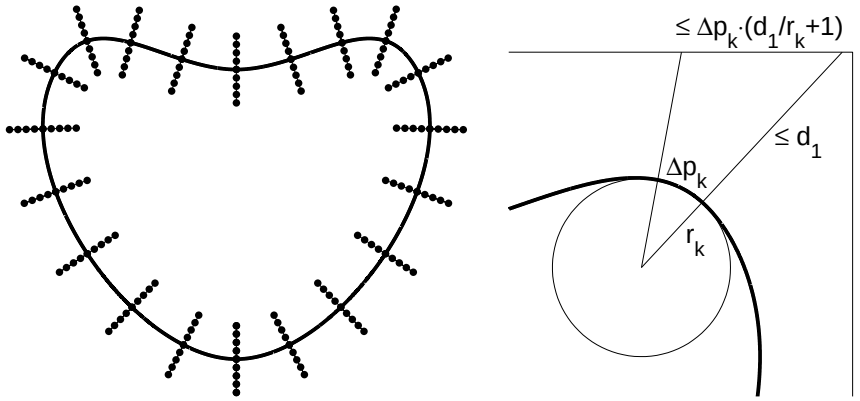
Die Generierung der unregelmäßig verteilten Daten, die auf der Abstandsfunktion basieren und als Grundlage für die Spline-Approximation dienen, erfolgt mit

`[X, f] = ScatteredData.WeightFunctions.gen_data(sc, H, n)`,

wobei `sc` die Spline-Kurve als Struktur (wie von `convert_sp_data` zurückgegeben) bezeichnet, die sich innerhalb von $[0,1]^2$ befindet.

Um die Idee der Datenerzeugung zu erklären, nehmen wir an, dass der geschlossene Spline-Rand des Gebiets D eine Jordan-Kurve parametrisiert durch $p\colon [0,1] \to [0,1]^2$ sei, also p stetig mit p injektiv bis auf $p(0) = p(1)$. Wir unterteilen nun das Parameterintervall $[0,1]$ in $m+1$ (der Einfachheit halber) äquidistante Punkte und „schießen“ von $p_k := p(t_k)$ mit $t_k := k/m$ orthogonal zu $p'_k := p'(t_k)$ auf Geraden in das Innere von D und in das Komplement von $\overline{D}$ ($k = 0, \ldots, m$). Der Abstand zum Rand $d(x)$ ist nämlich auf einem hinreichend kleinen Teilstück einer Geraden (ausgehend vom Randpunkt p_k) bekannt und muss nicht berechnet werden: Wenn wir nicht zu weit schießen (d. h. $|t|$ genügend klein) und n_k die Einheitsnormale im Punkt p_k nach innen bezeichnet, dann ist $d(p_k + tn_k) = t$ (Notation wie in (5.6)). Indem wir auf den Geraden äquidistante Punkte wählen, erhalten wir die benötigten unregelmäßigen Daten der Abstandsfunktion.

Abbildung 5.3a illustriert die Idee beispielhaft für ein unrealistisch kleines m. Wie zu sehen ist, hängt die optimale Schießweite (die in der Abbildung zu Demonstrationszwecken überall gleich kurz ist) stark vom konkreten Punkt ab. In der Kurve oben links und oben rechts, wo die Krümmung groß ist, kann nicht so weit nach innen geschossen werden wie im unteren Bereich mit kleiner Krümmung.



(a) zum Rand orthogonale Geraden mit äquidistanten Punkten **(b)** Berechnung der nötigen Anzahl m an Punkten

Abbildung 5.3: Generierung der Daten für das Testgebiet

Außerdem gilt es noch die Schwierigkeit zu lösen, die sich aus Definition 3.10 ergibt: Für jedes Gitterquadrat muss es mindestens einen Datenpunkt geben, der in diesem Gitterquadrat liegt. Dazu wählen wir die äquidistanten Punkte auf den orthogonalen Geraden im Abstand $h/2$, wobei $h = 1/H$ die gegebene Gitterweite mit $H \in \mathbb{N}$ bezeichnet. Das reicht jedoch noch nicht aus, denn wenn m zu klein gewählt ist, bleiben ebenfalls Gitterquadrate frei. Wie in Abbildung 5.3b zu sehen ist, kann die Kurve in einer Umgebung eines Punktes, in dem sie differenzierbar ist, näherungsweise mit konstanter Krümmung κ_k bzw. mit konstantem Krümmungsradius $r_k := 1/\kappa_k$ betrachtet werden. Wenn wir bei großem m zwei Punkte p_k und p_{k+1} mit Abstand Δp_k auf der Kurve wählen, können wir außerdem die Ableitung $p'(t)$ als konstant ansehen, d. h. $\Delta p_k \approx \|p'_k\| / m$. m muss nun so groß gewählt werden, dass die beiden vom Mittelpunkt des (gemeinsamen) Krümmungskreises ausgehenden Strahlen höchstens um H voneinander entfernt sind. Dabei berechnet sich der maximale Abstand zum Rand des Datengebiets als $d_1 := \sqrt{2}(1 + 2rh)$ (mit r wie in Definition 3.10).

Nach dem Strahlensatz sind die Strahlen am Rand also höchstens um

$$\Delta p_k \cdot \left(\frac{d_1}{r_k} + 1 \right) \approx \frac{\|p'_k\| \, (d_1 \kappa_k + 1)}{m} \tag{5.7}$$

voneinander entfernt. Wenn dies kleiner als h sein soll (damit jedes Gitterquadrat „getroffen“ wird), erhalten wir

$$m > \left\|p'_k\right\| (d_1 \kappa_k + 1) H. \tag{5.8}$$

Daher wählen wir mit $\kappa(t)$ der Krümmung der Kurve für den Parameter t

$$m := \left\lceil \max_{t \in [0,1]} \left\|p'(t)\right\| (d_1 \kappa(t) + 1) \cdot H \right\rceil \tag{5.9}$$

als minimales m. Eine weiter gehende Version des Programms würde die Punkte nicht äquidistant wählen, sondern die „Punktdichte“ auf dem Rand dynamisch durch eine ähnliche Formel in Abhängigkeit von Ableitung und Krümmung berechnen.

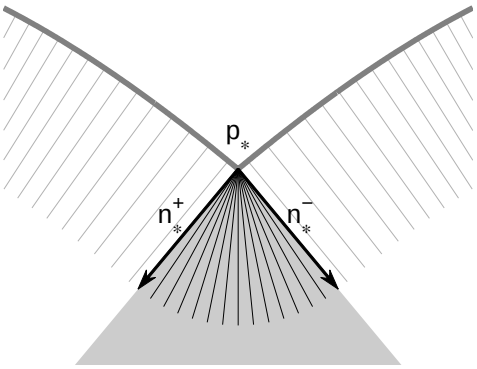


Abbildung 5.4: Ecke im Rand (fett, grau) eines Gebiets mit in das Gebiet zeigenden orthogonalen Geraden in grau, Normalenvektoren und zusätzlichen Radien

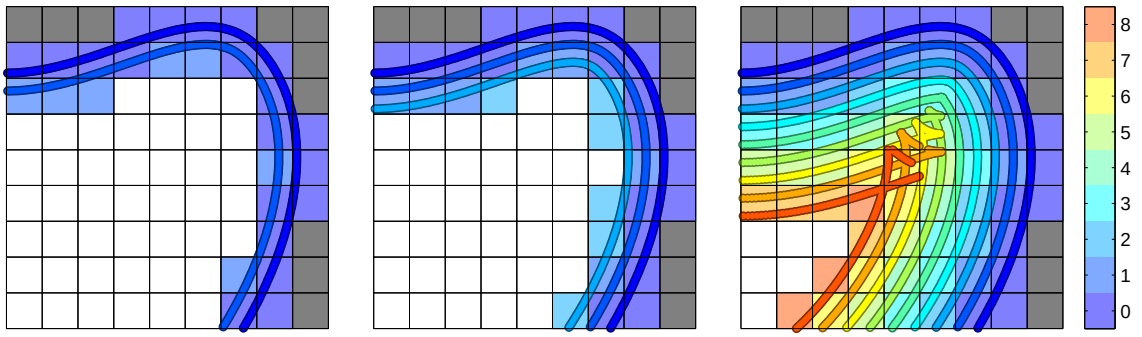
Eine weitere Erschwernis ist die Tatsache, dass der Spline-Rand „Ecken“ enthalten kann, an denen die Kurve nicht stetig differenzierbar ist. Das ist insofern ein Problem, weil bestimmte Gitterquadrate aufgrund der Unstetigkeit der Ableitung dann nicht getroffen werden. Abbildung 5.4 zeigt den Ausschnitt eines Gebiets, dessen Rand eine Ecke besitzt (der Rand wird von rechts nach links durchlaufen). Wir erkennen, dass die zum Rand orthogonalen Geraden den grau unterlegten Bereich zwischen dem links- und dem rechtsseitigen Normalenvektor n_*^- bzw. n_*^+ nicht erreichen. Um diese Schwierigkeit zu beheben, überlegen wir uns, dass in einer Umgebung der Ecke $p_* = p(t_*)$ im grau unterlegten Bereich genau die Punkte gleichen Abstand zu p_* haben, die sich auf einem Kreisbogen mit Mittelpunkt p_* befinden. Daher können wir eine gewisse Anzahl an Radien ausgehend von p_* in das Innere des Gebiets schießen und dann genauso verfahren wie bei den orthogonalen Geraden. Diese Funktionalität ist im Hilfsprogramm `ScatteredData.SplineUtil.get_corner_circles` untergebracht, das von `ScatteredData.WeightFunctions.gen_data` automatisch bei allen nicht stetig differenzierbaren Ecken aufgerufen wird. Es generiert eine gewisse Anzahl von Strahlen, so dass wie oben am Ende jede Gitterzelle mindestens ein Datenpunkt enthält.

Die Funktion unterstützt auch nach außen zeigende Ecken, denn dieselben Überlegungen können natürlich auch für diese durchgeführt werden. In diesem Fall ist die Abstandsfunktion nicht glatt, denn dann gibt es zwangsläufig einen „Knick“ im Inneren des Gebiets bei der Hälfte des Innenwinkels, egal wie klein δ in (5.6) gewählt wird. Im Fall von nach innen zeigenden Ecken (wie bei obigem Beispiel) ist die Situation umgekehrt, dort befindet sich der Knick bei den negativen Daten außerhalb des Gebiets.

Nachdem diese Schwierigkeiten beiseite geräumt wurden, wird die konkrete Datenerstellung von `ScatteredData.WeightFunctions.gen_data` wie folgt durchgeführt:

(a) Zunächst werden die Gitterzellen für die gegebene Gitterweite $h = 1/H$ mit $H \in \mathbb{N}$ in äußere und innere Zellen sowie Randzellen unterteilt, je nachdem, ob die Gitterzellen vollständig im Komplement von $\overline{D}$ oder in D selbst liegen (oder für Randzellen, ob die Spline-Kurve ∂D durch das Innere der Gitterzelle läuft). Dazu wird eine deutlich schnellere Version der MATLAB-Funktion `inpolygon` namens `inpoly` vom *MATLAB File Exchange* benutzt[1].

[1] URL: `http://www.mathworks.com/matlabcentral/fileexchange/10391-fast-points-in-polygon-test`



(a) nach dem 1. Schritt **(b)** nach dem 2. Schritt **(c)** nach dem 8. Schritt

Abbildung 5.5: Erzeugung der Daten für einen beispielhaften Ausschnitt des Testgebiets

(b) Jedem Gitterquadrat wird eine Zahl zugeordnet, wobei anfangs die Randzellen den Wert 0, die inneren den Wert $+\infty$ und die äußeren Zellen den Wert $-\infty$ erhalten.

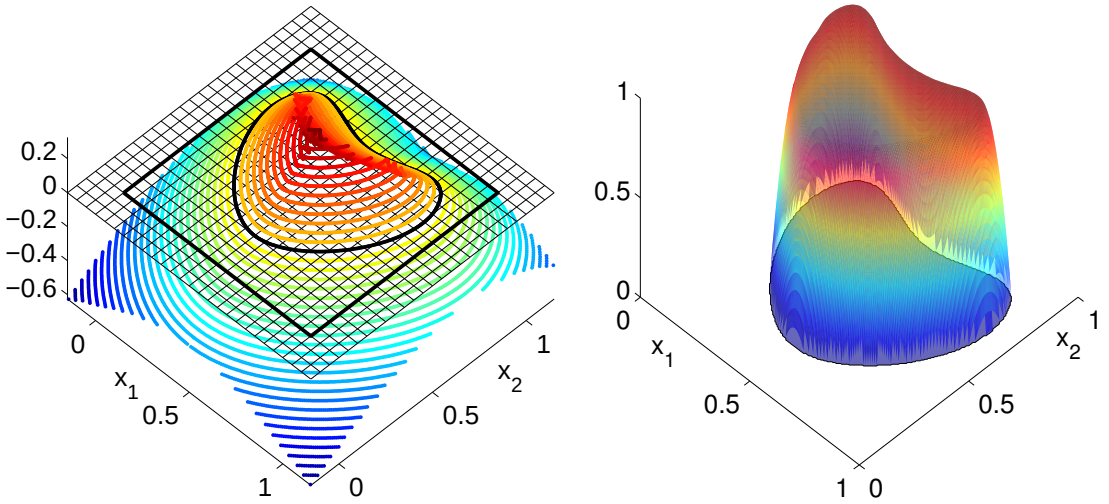
(c) Anschließend schießt der Algorithmus auf allen Geraden (orthogonale Geraden und Strahlen ausgehend von Ecken) um einen Schritt nach innen. Die Schrittweite beträgt hierbei $h/2$. Auf allen neu erreichten Gitterzellen (die den Wert $+\infty$ beinhalten) wird die zugeordnete Zahl auf 1 gesetzt.

(d) Der vorherige Schritt wird wiederholt: Die Zahlen der neu erreichten Zellen werden dann natürlich auf 2 gesetzt usw. Wenn eine äußere Zelle (mit dem Wert $-\infty$) oder eine Zelle, deren Wert im i-ten Schritt um mindestens drei kleiner ist als i, erreicht wird, wird die entsprechende Gerade „deaktiviert“, das heißt, auf ihr wird nicht mehr geschossen. Die Zahl der Geraden, auf denen geschossen wird, nimmt also monoton ab – wenn sie null erreicht, sind die Daten für das Innere des Gebiets erzeugt.

(e) Die Schritte (c) und (d) werden nun sinngemäß nochmals durchgeführt, wobei jetzt nach außen geschossen wird. Dabei erhalten die Zellen die Werte -1, -2 usw. und in (d) muss u. a. „kleiner“ durch „größer“ ersetzt werden.

Damit erhält man verteilte Daten der Abstandsfunktion, wobei die Höhe der inneren Datenpunkte im i-ten Schritt auf $i \cdot h/2$ gesetzt wird (bei den äußeren Datenpunkten ist $i \in \mathbb{Z} \setminus \mathbb{N}_0$).

In Abbildung 5.5 ist die Durchführung des Algorithmus für einen Ausschnitt unseres Testgebiets schrittweise dargestellt. Die Farbe der Gitterzellen gibt deren zugeordnete Zahl an (siehe Legende rechts). Dunkelgraue Zellen besitzen den Wert $-\infty$ und weiße Zellen den Wert $+\infty$. Die Kurven zeigen die in den unterschiedlichen Schritten erzeugten Datenpunkte. Die Zahlen am unteren Rand sagen aus, in welchem Schritt die jeweiligen Daten generiert wurden (der „0. Schritt“ in Dunkelblau ist der Rand des Testgebiets). Im gezeigten Ausschnitt werden erst im 7. Schritt Geraden „deaktiviert“, was man im mittleren Gitterquadrat an der roten Kurve erkennen kann. Insgesamt werden für das Innere des Testgebiets 14 Schritte durchgeführt, für das Komplement sind sogar weitere 26 Schritte nötig. Das komplette Ergebnis ist für $H = 20$ und $n = 2$ in Abbildung 5.6a zu sehen.



(a) Daten, die durch `ScatteredData.WeightFunctions.gen_data` generiert wurden

(b) Approximation der Gewichtsfunktion mit den Parametern $\delta = 0{,}07$, $\gamma = 2$

Abbildung 5.6: Ergebnis der Datenerzeugung und Approximation der Gewichtsfunktion für das Testgebiet

5.2.5 Evaluation und Visualisierung

Sobald die Daten erzeugt worden sind, wird zunächst Formel (5.6) auf die Daten durch

```
w = ScatteredData.WeightFunctions.apply(dist, delta, gamma)
```

angewendet, wobei diese Funktion nicht nur im Inneren des Gebiets ein Plateau der Höhe 1 erzeugt, sondern auch im Komplement des Abschlusses (mit der Höhe -1). Die so erhaltenen Daten können dann wie in Abschnitt 4.1.4 beschrieben zur Spline-Approximation verwendet werden. Der resultierende Spline kann anschließend mit den in Abschnitt 4.1.5 erwähnten Routinen ausgewertet werden.

Für die Visualisierung stehen zwei Funktionen bereit: Zum einen können die durch die Funktion `ScatteredData.WeightFunctions.gen_data` erzeugten Daten durch

```
ScatteredData.WeightFunctions.visualize_data(X, f, H, n)
```

geplottet werden, zum anderen kann man die Gewichtsfunktion nach (5.6) mit

```
ScatteredData.WeightFunctions.visualize(H, p)
```

darstellen, wobei p die Spline-Approximation bezeichnet. Die sich ergebenden Plots für das Testgebiet sind jeweils in den Abbildungen 5.6a und 5.6b sichtbar.

5.3 Numerische Aspekte

In Tabelle 5.1 sind in den ersten beiden Zeilen die absoluten Fehler (Maximum und Mittelwert) der Spline-Approximation zu w aus Formel (5.6) für das Testgebiet mit verschiedenem H zu sehen. Die Approximation ist für die dargestellten Gitterweiten $h = 1/H$ nicht sonderlich gut, für $H = 128$ liegt der maximale absolute Fehler immer noch nicht unter einem

Tabelle 5.1: maximaler bzw. mittlerer absoluter Fehler e_h bzw. $\overline{e_h}$ und Ausführungszeiten t_h für das Testgebiet mit $n = 2$, $\delta = 0{,}07$ und $\gamma = 2$ (gerundet)

$\log_2 H$	2	3	4	5	6	7
e_h	$7{,}46 \cdot 10^{-1}$	$4{,}91 \cdot 10^{-1}$	$1{,}99 \cdot 10^{-1}$	$4{,}13 \cdot 10^{-2}$	$1{,}03 \cdot 10^{-2}$	$3{,}44 \cdot 10^{-3}$
$\overline{e_h}$	$2{,}88 \cdot 10^{-1}$	$1{,}61 \cdot 10^{-1}$	$4{,}82 \cdot 10^{-2}$	$6{,}76 \cdot 10^{-3}$	$8{,}58 \cdot 10^{-4}$	$1{,}11 \cdot 10^{-4}$
t_h	0,086 s	0,085 s	0,11 s	0,13 s	0,20 s	0,48 s

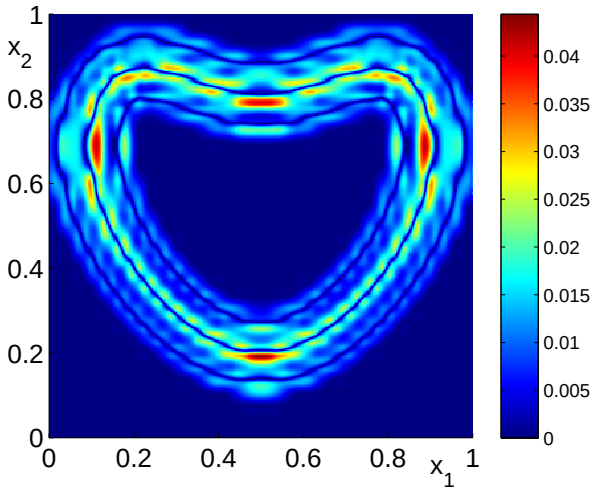


Abbildung 5.7: Zehnerlogarithmus des absoluten Fehlers für $H = 32$

Tausendstel. Das könnte auf die Gradienten von w zurückzuführen sein, die für kleines δ (hier $\delta = 0{,}07$) ziemlich hoch sind. Abbildung 5.7 zeigt den absoluten Fehler für $H = 32$. Es ist zu erkennen, dass der Fehler in der Nähe des Rands und der „Kanten“ (im Abstand von δ im Gebiet bzw. außerhalb des Gebiets) besonders klein wird. Vor bzw. hinter diesen Kanten wird der Fehler noch einmal etwas größer (aufgrund von Überschwingungen der stückweisen Polynome). Kurioserweise ist der Fehler dort am größten, wo der Rand einen Tangentenvektor parallel zu einer der Koordinatenachsen besitzt, was womöglich etwas mit der Tensorprodukt-Struktur des Spline-Raums zu tun hat.

Eine Betrachtung des relativen Fehlers ist für die Anwendung der Approximation von Gewichtsfunktionen kaum sinnvoll, da die Spline-Approximation p nicht dieselbe Nullstellenmenge wie die wahre Gewichtsfunktion w besitzt. Somit ist der relative Fehler, bei dem der absolute Fehler $|w(x_1, x_2) - p(x_1, x_2)|$ durch den Betrag $|w(x_1, x_2)|$ des wahren Werts geteilt wird, je nach Dichte der Auswertungspunkte beliebig hoch (in den Nullstellen (x_1, x_2) von w mit $p(x_1, x_2) \neq 0$ sogar unendlich bzw. nicht definiert).

In der untersten Zeile in Tabelle 5.1 sind der Vollständigkeit halber noch die Ausführungszeiten von `ScatteredData.WeightFunctions.gen_data` aufgeführt, die ja für die Approximation auf die in Abschnitt 4.3.3 erwähnten Zeiten addiert werden müssen, aber für sich einigermaßen moderat sind. Eine signifikante Beschleunigung könnte man wahrscheinlich durch Vektorisierung, Parallelisierung oder Implementierung in C++ erhalten.

A Inhalt der CD-ROM

Die als Anhang zu dieser Arbeit beigelegte CD-ROM enthält neben der MATLAB-Implementierung der Algorithmen aus Kapitel 4 und 5 im Ordner **matlab** auch eine elektronische Version der Arbeit als PDF-Datei im Ordner **pdf**. Die in C++ geschriebene MEX-Implementierung `spl_eval_mex` des Algorithmus von de Boor (siehe Abschnitt 4.1.5) befindet sich in Form von zwei Eclipse-CDT[1]-Projekten im Ordner **c++**. Die Eclipse-Projekte unterscheiden sich nur in der Architektur (x64-Linux und x64-Windows), beinhalten aber dieselbe Quellcode-Datei `spl_eval_mex.cpp`. Soll der Code kompiliert werden, müssen wahrscheinlich einige Pfade in den Einstellungen von Eclipse verändert werden.

Die DLLs im Ordner **matlab/+ScatteredData/+SplineUtil** werden nur benötigt, falls die Windows-MEX-Version des Algorithmus aufgerufen wird (ansonsten kann man sie löschen). Dabei handelt es sich um MATLAB-DLLs und DLLs aus MinGW-w64[2].

Es folgt eine vollständige Liste der auf der CD-ROM vorhandenen Dateien, strukturiert nach Verzeichnissen (diese sind in fett gedruckt):

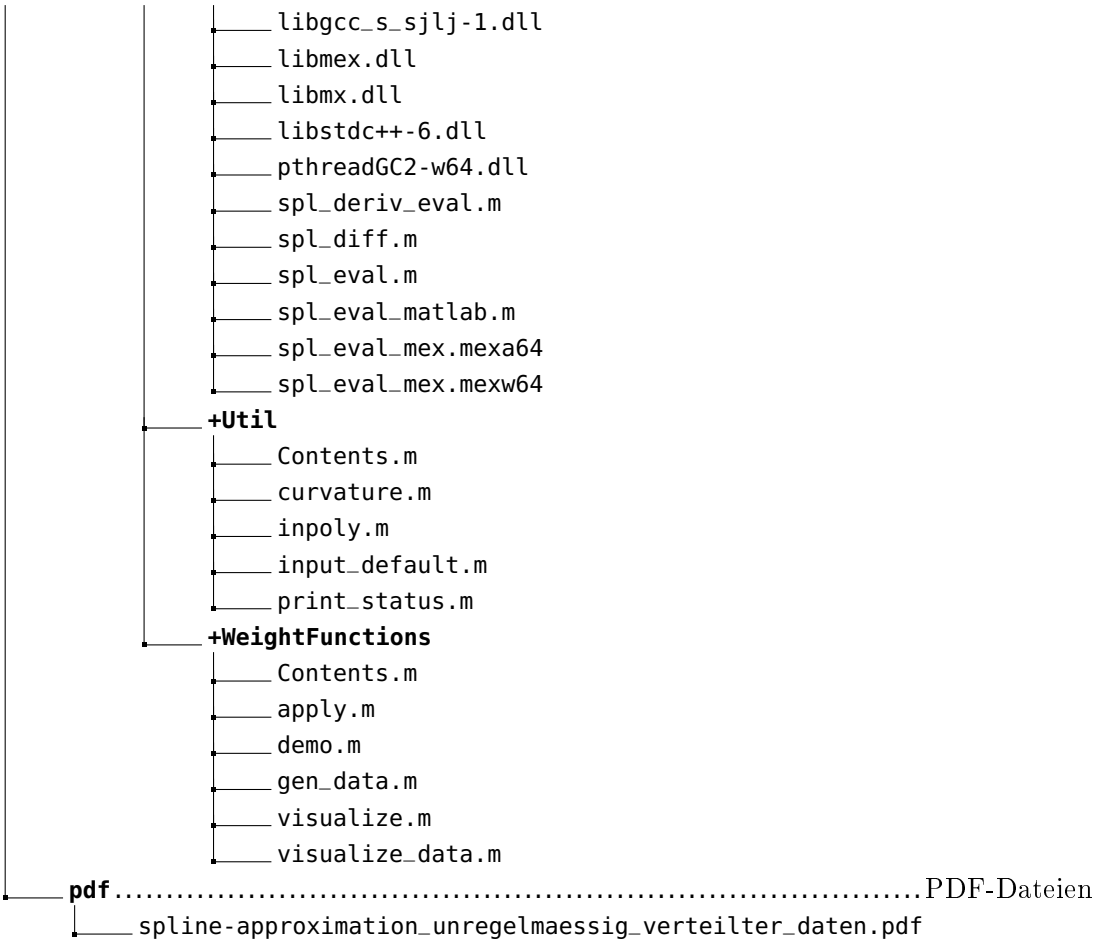
Wurzelverzeichnis
- **c++** ..... C++-Implementierungen
  - **linux**
    - **spl_eval_mex**
      - .cproject
      - .project
      - **Release**
        - makefile
        - objects.mk
        - sources.mk
        - spl_eval_mex.mexa64
        - **src**
          - spl_eval_mex.d
          - spl_eval_mex.o
          - subdir.mk
      - **src**
        - spl_eval_mex.cpp
  - **windows**
    - **spl_eval_mex**
      - .cproject
      - .project
      - **Release**
        - makefile

[1] URL: `http://www.eclipse.org/cdt/`
[2] URL: `http://sourceforge.net/projects/mingw-w64/`

- objects.mk
- sources.mk
- spl_eval_mex.mexw64
- **src**
  - spl_eval_mex.d
  - spl_eval_mex.o
  - subdir.mk
- **include**
  - pthread.h
  - pthreads_win32_config.h
  - sched.h
  - semaphore.h
- **lib**
  - pthreadGC2-w64.dll
- **src**
  - spl_eval_mex.cpp

matlab ..... MATLAB-Programme

- example_halton.m
- example_weight_function.m
- **+ScatteredData**
  - Contents.m
  - change_settings.m
  - print_settings.m
  - **+Halton**
    - Contents.m
    - demo.m
    - franke.m
    - gen_points.m
  - **+SplineApproximation**
    - Contents.m
    - gen_grid_data.m
    - get_opt_grid.m
    - make.m
    - quasi_interp.m
    - sort_data.m
    - visualize.m
  - **+SplinePlotter**
    - Contents.m
    - convert_sp_data.m
    - rspl_val.m
    - spl_val.m
    - splineplotter.m
  - **+SplineUtil**
    - Contents.m
    - get_corner_circles.m
    - get_corners.m


libgcc_s_sjlj-1.dll
libmex.dll
libmx.dll
libstdc++-6.dll
pthreadGC2-w64.dll
spl_deriv_eval.m
spl_diff.m
spl_eval.m
spl_eval_matlab.m
spl_eval_mex.mexa64
spl_eval_mex.mexw64
+Util
Contents.m
curvature.m
inpoly.m
input_default.m
print_status.m
+WeightFunctions
Contents.m
apply.m
demo.m
gen_data.m
visualize.m
visualize_data.m
pdf
PDF-Dateien
spline-approximation_unregelmaessig_verteilter_daten.pdf

Literaturverzeichnis

[1] P. Alfeld: *Scattered Data Interpolation in Three or More Variables.* In: Mathematical Methods in Computer Aided Geometric Design. Hrsg. von T. Lyche und L. L. Schumaker. New York: Academic Press, 1989, S. 1–33.

[2] C. de Boor: *A Practical Guide to Splines.* New York: Springer, 1978.

[3] C. de Boor: *On Calculating with B-Splines.* In: Journal of Approximation Theory 6.1 (1972), S. 50–62.

[4] C. de Boor: *Splines as Linear Combinations of B-Splines. A Survey.* In: *Approximation Theory II.* Hrsg. von G. G. Lorentz, C. K. Chui und L. L. Schumaker. New York: Academic Press, 1976, S. 1–47.

[5] M. D. Buhmann: *Radial Basis Functions: Theory and Implementations.* Cambridge: Cambridge University Press, 2003.

[6] M. G. Cox: *The Numerical Evaluation of B-Splines.* In: IMA Journal of Applied Mathematics 10.2 (1972), S. 134–149.

[7] N. Cressie: *The Origins of Kriging.* In: Mathematical Geology 22.3 (1990), S. 239–252.

[8] M. Fenn und G. Steidl: *Robust Local Approximation of Scattered Data.* In: Geometric Properties for Incomplete Data. Hrsg. von R. Klette u. a. Dordrecht: Springer, 2006, S. 317–334.

[9] R. Franke: *Scattered Data Interpolation: Tests of Some Methods.* In: Mathematics of Computation 38.157 (1982), S. 181–200.

[10] R. Franke und G. Nielson: *Smooth Interpolation of Large Sets of Scattered Data.* In: International Journal for Numerical Methods in Engineering 15.11 (1980), S. 1691–1704.

[11] M. Gasca und T. Sauer: *Polynomial Interpolation in Several Variables.* In: Advances in Computational Mathematics 12.4 (2000), S. 377–410.

[12] J. H. Halton: *On the Efficiency of Certain Quasi-Random Sequences of Points in Evaluating Multi-Dimensional Integrals.* In: Numerische Mathematik 2.1 (1960), S. 84–90.

[13] K. Höllig: *Finite Element Methods with B-Splines.* Philadelphia: Society for Industrial and Applied Mathematics, 2003.

[14] K. Höllig und J. Hörner: *Approximation and Modeling with B-Splines.* Preprint. 2011.

[15] K. Höllig, U. Reif und J. Wipper: *Weighted Extended B-Spline Approximation of Dirichlet Problems.* In: SIAM Journal on Numerical Analysis 39.2 (2001), S. 442–462.

[16] D. G. Krige: *A Statistical Approach to Some Basic Mine Valuation Problems on the Witwatersrand.* In: Journal of the Chemical, Metallurgical and Mining Society of South Africa 52.6 (1951), S. 119–139.

[17] G. Matheron: *The Intrinsic Random Functions and Their Applications*. In: Advances in Applied Probability 5.3 (1973), S. 439–468.

[18] A. Nealen: *An As-Short-As-Possible Introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation*. Techn. Ber. TU Darmstadt, 2004. URL: `http://www.nealen.com/projects/mls/asapmls.pdf`.

[19] J. Prasiswa: *Lokale und globale Algorithmen zur Approximation mit erweiterten B-Splines*. Diss. TU Darmstadt, 2009.

[20] D. Ruprecht und H. Müller: *Image Warping with Scattered Data Interpolation*. In: IEEE Computer Graphics and Applications 15.2 (1995), S. 37–43.

[21] T. Sauer und Y. Xu: *On multivariate Lagrange interpolation*. In: Mathematics of Computation 64.211 (1995), S. 1147–1170.

[22] S. Schaefer, T. McPhail und J. Warren: *Image Deformation Using Moving Least Squares*. In: ACM Transactions on Graphics 25.3 (2006), S. 533–540.

[23] L. L. Schumaker: *Spline Functions: Basic Theory*. 3. Aufl. Cambridge: Cambridge University Press, 2007.

[24] H.-R. Schwarz und N. Köckler: *Numerische Mathematik*. 7. Aufl. Wiesbaden: Vieweg+ Teubner, 2009.

[25] V. Shapiro: *Theory of R-Functions and Applications: A Primer*. Techn. Ber. CPA88-3. Cornell University, 1991.

[26] D. Shepard: *A Two-Dimensional Interpolation Function for Irregularly-Spaced Data*. In: *Proceedings of the 1968 23rd ACM national conference*. Hrsg. von R. B. Blue, Sr. und A. M. Rosenberg. New York: ACM, 1968, S. 517–524.

[27] X. Wang und F. J. Hickernell: *Randomized Halton Sequences*. In: Mathematical and Computer Modelling 32 (7-8 2000), S. 887–899.

[28] H. Wendland: *Local Polynomial Reproduction and Moving Least Squares Approximation*. In: IMA Journal of Numerical Analysis 21.1 (2001), S. 285–300.

Die Internetquelle [18] wird auf den folgenden Seiten vollständig wiedergegeben. Hinzugefügt wurden nur die Kopfzeile mit Internetadresse und Datum des Herunterladens sowie gelbe Markierungen im Text zur Hervorhebung der zitierten Stellen.

An As-Short-As-Possible Introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation

Andrew Nealen
Discrete Geometric Modeling Group
TU Darmstadt

Abstract

In this introduction to the Least Squares (LS), Weighted Least Squares (WLS) and Moving Least Squares (MLS) methods, we briefly describe and derive the linear systems of equations for the global least squares, and the weighted, local least squares approximation of function values from scattered data. By *scattered data* we mean an arbitrary set of points in $\mathbb{R}^d$ which carry scalar quantities (i.e. a scalar field in d dimensional parameter space). In contrast to the global nature of the least-squares fit, the weighted, local approximation is computed either at discrete points, or continuously over the parameter domain, resulting in the global WLS or MLS approximation respectively.

Keywords: Data Approximation, Least Squares (LS), Weighted Least Squares (WLS), Moving Least Squares (MLS), Linear System of Equations, Polynomial Basis

1 LS Approximation

Problem Formulation. Given N points located at positions $\mathbf{x}_i$ in $\mathbb{R}^d$ where $i \in [1 \dots N]$. We wish to obtain a globally defined function $f(\mathbf{x})$ that approximates the given scalar values f_i at points $\mathbf{x}_i$ in the least-squares sense with the error functional $J_{LS} = \sum_i \|f(\mathbf{x}_i) - f_i\|^2$. Thus, we pose the following minimization problem

$$\min_{f \in \prod_m^d} \sum_i \|f(\mathbf{x}_i) - f_i\|^2, \tag{1}$$

where f is taken from $\prod_m^d$, the space of polynomials of total degree m in d spatial dimensions, and can be written as

$$f(\mathbf{x}) = \mathbf{b}(\mathbf{x})^T \mathbf{c} = \mathbf{b}(\mathbf{x}) \cdot \mathbf{c}, \tag{2}$$

where $\mathbf{b}(\mathbf{x}) = [b_1(\mathbf{x}), \dots, b_k(\mathbf{x})]^T$ is the polynomial basis vector and $\mathbf{c} = [c_1, \dots, c_k]^T$ is the vector of unknown coefficients, which we wish to minimize in (1). Here some examples for polynomial bases: (a) for $m = 2$ and $d = 2$, $\mathbf{b}(\mathbf{x}) = [1, x, y, x^2, xy, y^2]^T$, (b) for a linear fit in $\mathbb{R}^3$ ($m = 1$, $d = 3$), $\mathbf{b}(\mathbf{x}) = [1, x, y, z]^T$, and (c) for fitting a constant in arbitrary dimensions, $\mathbf{b}(\mathbf{x}) = [1]$. In general, the number k of elements in $\mathbf{b}(\mathbf{x})$ (and therefore in $\mathbf{c}$) is given by $k = \frac{(d+m)!}{m!d!}$, see [Levin 1998; Fries and Matthies 2003].

Solution. We can minimize (1) by setting the partial derivatives of the error functional J_{LS} to zero, i.e. $\nabla J_{LS} = 0$ where $\nabla = [\partial/\partial c_1, \dots, \partial/\partial c_k]^T$, which is a necessary condition for a minimum. By taking partial derivatives with respect to the unknown coefficients $c_1, \dots, c_k$, we obtain a linear system of equations (LSE) from which we can compute $\mathbf{c}$

$$\begin{aligned}
\partial J_{LS}/\partial c_1 = 0: &\quad \sum_i 2b_1(\mathbf{x}_i)[\mathbf{b}(\mathbf{x}_i)^T \mathbf{c} - f_i] = 0 \\
\partial J_{LS}/\partial c_2 = 0: &\quad \sum_i 2b_2(\mathbf{x}_i)[\mathbf{b}(\mathbf{x}_i)^T \mathbf{c} - f_i] = 0 \\
&\quad \vdots \\
\partial J_{LS}/\partial c_k = 0: &\quad \sum_i 2b_k(\mathbf{x}_i)[\mathbf{b}(\mathbf{x}_i)^T \mathbf{c} - f_i] = 0.
\end{aligned}$$

In matrix-vector notation, this can be written as

$$\begin{aligned}
\sum_i 2\mathbf{b}(\mathbf{x}_i)[\mathbf{b}(\mathbf{x}_i)^T \mathbf{c} - f_i] &= \\
2\sum_i [\mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T \mathbf{c} - \mathbf{b}(\mathbf{x}_i) f_i] &= \mathbf{0}.
\end{aligned}$$

Dividing by the constant and rearranging yields the following LSE

$$\sum_i \mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T \mathbf{c} = \sum_i \mathbf{b}(\mathbf{x}_i) f_i, \tag{3}$$

which is solved as

$$\mathbf{c} = [\sum_i \mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T]^{-1} \sum_i \mathbf{b}(\mathbf{x}_i) f_i. \tag{4}$$

If the square matrix $\mathbf{A}_{LS} = \sum_i \mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T$ is nonsingular (i.e. $det(\mathbf{A}_{LS}) \neq 0$), substituting Eqn. (4) into Eqn. (2) provides the fit function $f(\mathbf{x})$. For small k ($k < 5$), the matrix inversion in Eqn. (4) can be carried out explicitly, otherwise numerical methods are the preferred tool, see [Press et al. 1992] [1]. In our applications, we often use the Template Numerical Toolkit (TNT) [2].

Example. Say our data points live in $\mathbb{R}^2$ and we wish to fit a quadratic, bivariate polynomial, i.e. $d = 2$, $m = 2$ and therefore $\mathbf{b}(\mathbf{x}) = [1, x, y, x^2, xy, y^2]^T$ (see above), then the resulting LSE looks like this

$$\sum_i \begin{bmatrix} 1 & x_i & y_i & x_i^2 & x_i y_i & y_i^2 \\ x_i & x_i^2 & x_i y_i & x_i^3 & x_i^2 y_i & x_i y_i^2 \\ y_i & x_i y_i & y_i^2 & x_i^2 y_i & x_i y_i^2 & y_i^3 \\ x_i^2 & x_i^3 & x_i^2 y_i & x_i^4 & x_i^3 y_i & x_i^2 y_i^2 \\ x_i y_i & x_i^2 y_i & x_i y_i^2 & x_i^3 y_i & x_i^2 y_i^2 & x_i y_i^3 \\ y_i^2 & x_i y_i^2 & y_i^3 & x_i^2 y_i^2 & x_i y_i^3 & y_i^4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \sum_i \begin{bmatrix} 1 \\ x_i \\ y_i \\ x_i^2 \\ x_i y_i \\ y_i^2 \end{bmatrix} f_i.$$

Consider the set of nine 2D points $P_i = \{(1,1), (1,-1), (-1,1), (-1,-1), (0,0), (1,0), (-1,0), (0,1), (0,-1)\}$ with two sets of associated function values $f_i^1 = \{1.0, -0.5, 1.0, 1.0, -1.0, 0.0, 0.0, 0.0, 0.0\}$ and $f_i^2 = \{1.0, -1.0, 0.0, 0.0, 1.0, 0.0, -1.0, -1.0, 1.0\}$. Figure 1 shows the fit functions for the scalar fields f_i^1 and f_i^2.

[1] at the time of writing this report, [Press et al. 1992] was available online in pdf format through `http://www.nr.com/`

[2] `http://math.nist.gov/tnt/`

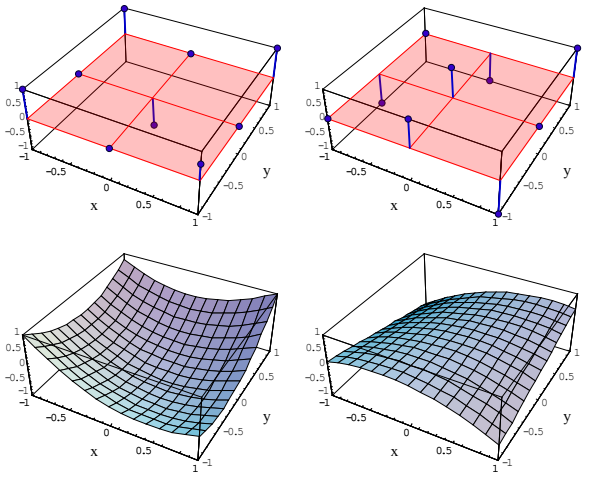


Figure 1: Fitting bivariate, quadratic polynomials to 2D scalar fields: the top row shows the two sets of nine data points (see text), the bottom row shows the least squares fit function. The coefficient vectors $[c_1,\ldots,c_6]^T$ are $[-0.834,-0.25,0.75,0.25,0.375,0.75]^T$ (left column) and $[0.334,0.167,0.0,-0.5,0.5,0.0]^T$.

Method of Normal Equations. For a different but also very common notation, note that the solution for $\mathbf{c}$ in Eqn. (3) solves the following (generally over-constrained) LSE ($\mathbf{Bc}=\mathbf{f}$) in the least-squares sense

$$\begin{bmatrix}\mathbf{b}^T(\mathbf{x}_1)\\ \vdots \\ \mathbf{b}^T(\mathbf{x}_N)\end{bmatrix}\mathbf{c}=\begin{bmatrix}f_1\\ \vdots \\ f_N\end{bmatrix}, \tag{5}$$

using the method of normal equations

$$\begin{aligned}\mathbf{B}^T\mathbf{Bc} &= \mathbf{B}^T\mathbf{f}\\ \mathbf{c} &= (\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T\mathbf{f}.\end{aligned} \tag{6}$$

Please verify that Eqns. (4) and (6) are identical.

2 WLS Approximation

Problem Formulation. In the weighted least squares formulation, we use the error functional $J_{WLS}=\sum_i \theta(\|\bar{\mathbf{x}}-\mathbf{x}_i\|)\ \|f(\mathbf{x}_i)-f_i\|^2$ for a fixed point $\bar{\mathbf{x}}\in\mathbb{R}^d$, which we minimize

$$\min_{f\in\Pi_m^d}\sum_i \theta(\|\bar{\mathbf{x}}-\mathbf{x}_i\|)\ \|f(\mathbf{x}_i)-f_i\|^2, \tag{7}$$

similar to (1), only that now the error is weighted by $\theta(d)$ where d_i are the Euclidian distances between $\bar{\mathbf{x}}$ and the positions of data points $\mathbf{x}_i$.

The unknown coefficients we wish to obtain from the solution to (7) are weighted by distance to $\bar{\mathbf{x}}$ and therefore a function of $\bar{\mathbf{x}}$. Thus, the local, weighted least squares approximation in $\bar{\mathbf{x}}$ is written as

$$f_{\bar{\mathbf{x}}}(\mathbf{x})=\mathbf{b}(\mathbf{x})^T\mathbf{c}(\bar{\mathbf{x}})=\mathbf{b}(\mathbf{x})\cdot\mathbf{c}(\bar{\mathbf{x}}), \tag{8}$$

and only defined locally within a distance R around $\bar{\mathbf{x}}$, i.e. $\|\mathbf{x}-\bar{\mathbf{x}}\|<R$.

Weighting Function. Many choices for the weighting function θ have been proposed in the literature, such as a Gaussian

$$\theta(d)=e^{-\frac{d^2}{h^2}}, \tag{9}$$

where h is a spacing parameter which can be used to smooth out small features in the data, see [Levin 2003; Alexa et al. 2003]. Another popular weighting function with compact support is the Wendland function [Wendland 1995]

$$\theta(d)=(1-d/h)^4(4d/h+1). \tag{10}$$

This function is well defined on the interval $d\in[0,h]$ and furthermore, $\theta(0)=1$, $\theta(h)=0$, $\theta'(h)=0$ and $\theta''(h)=0$ (C^2 continuity). Several authors suggest using weighting functions of the form

$$\theta(d)=\frac{1}{d^2+\varepsilon^2}. \tag{11}$$

Note that setting the parameter ε to zero results in a singularity at $d=0$, which forces the MLS fit function to interpolate the data, as we will see later.

Solution. Analogous to Section 1, we take partial derivatives of the error functional J_{WLS} with respect to the unknown coefficients $\mathbf{c}(\bar{\mathbf{x}})$

$$\begin{aligned}\sum_i \theta(d_i)\ 2\mathbf{b}(\mathbf{x}_i)[\mathbf{b}(\mathbf{x}_i)^T\mathbf{c}(\bar{\mathbf{x}})-f_i] &= \\ 2\sum_i[\theta(d_i)\mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T\mathbf{c}(\bar{\mathbf{x}})-\theta(d_i)\mathbf{b}(\mathbf{x}_i)f_i] &= \mathbf{0},\end{aligned}$$

where $d_i=\|\bar{\mathbf{x}}-\mathbf{x}_i\|$. We divide by the constant and rearrange to obtain

$$\sum_i \theta(d_i)\mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T\mathbf{c}(\bar{\mathbf{x}})=\sum_i \theta(d_i)\mathbf{b}(\mathbf{x}_i)f_i, \tag{12}$$

and solve for the coefficients

$$c(\bar{\mathbf{x}})=[\sum_i \theta(d_i)\mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T]^{-1}\sum_i \theta(d_i)\mathbf{b}(\mathbf{x}_i)f_i. \tag{13}$$

Obviously, the only difference between Eqns. (4) and (13) are the weighting terms. Note again though, that whereas the coefficients $\mathbf{c}$ in Eqn. (4) are global, the coefficients $c(\bar{\mathbf{x}})$ are local and need to be recomputed for every $\bar{\mathbf{x}}$. If the square matrix $\mathbf{A}_{WLS}=\sum_i \theta(d_i)\mathbf{b}(\mathbf{x}_i)\mathbf{b}(\mathbf{x}_i)^T$ (often termed the *Moment Matrix*) is nonsingular (i.e. $det(\mathbf{A}_{WLS})\neq 0$), substituting Eqn. (13) into Eqn. (8) provides the fit function $f_{\bar{\mathbf{x}}}(\mathbf{x})$.

Global Approximation using a Partition of Unity (PU). By fitting polynomials at $j\in[1\ldots n]$ discrete, fixed points $\bar{\mathbf{x}}_j$ in the parameter domain Ω, we can assemble a global approximation to our data by ensuring that every point in Ω is covered by at least one approximating polynomial, i.e. the support of the weight functions θ_j centered at the points $\bar{\mathbf{x}}_j$ covers Ω

$$\Omega=\bigcup_j \mathrm{supp}(\theta_j).$$

Proper weighting of these approximations can be achieved by constructing a Partition of Unity (PU) from the θ_j [Shepard 1968]

$$\varphi_j(\mathbf{x})=\frac{\theta_j(\mathbf{x})}{\sum_{k=1}^n \theta_k(\mathbf{x})}, \tag{14}$$

where $\sum_j \varphi_j(\mathbf{x})\equiv 1$ everywhere in Ω. The global approximation then becomes

$$f(\mathbf{x})=\sum_j \varphi_j(\mathbf{x})\ \mathbf{b}(\mathbf{x})^T\mathbf{c}(\bar{\mathbf{x}}_j). \tag{15}$$

A Numerical Issue. To avoid numerical instabilities due to possibly large numbers in $\mathbf{A}_{WLS}$ it can be beneficial to perform the fitting procedure in a local coordinate system relative to $\bar{\mathbf{x}}$, i.e. to shift $\bar{\mathbf{x}}$ into the origin. We therefore rewrite the local fit function in $\bar{\mathbf{x}}$ as

$$f_{\bar{\mathbf{x}}}(\mathbf{x}) = \mathbf{b}(\mathbf{x}-\bar{\mathbf{x}})^T \mathbf{c}(\bar{\mathbf{x}}) = \mathbf{b}(\mathbf{x}-\bar{\mathbf{x}}) \cdot \mathbf{c}(\bar{\mathbf{x}}), \tag{16}$$

the associated coefficients as

$$c(\bar{\mathbf{x}}) = [\sum_i \theta(d_i)\mathbf{b}(\mathbf{x}_i-\bar{\mathbf{x}})\mathbf{b}(\mathbf{x}_i-\bar{\mathbf{x}})^T]^{-1} \sum_i \theta(d_i)\mathbf{b}(\mathbf{x}_i-\bar{\mathbf{x}})f_i, \tag{17}$$

and the global approximation as

$$f(\mathbf{x}) = \sum_j \varphi_j(\mathbf{x})\ \mathbf{b}(\mathbf{x}-\bar{\mathbf{x}}_j)^T \mathbf{c}(\bar{\mathbf{x}}_j). \tag{18}$$

3 MLS Approximation and Interpolation

Method. The MLS method was proposed by Lancaster and Salkauskas [Lancaster and Salkauskas 1981] for smoothing and interpolating data. The idea is to start with a weighted least squares formulation for an arbitrary fixed point in $\mathbb{R}^d$, see Section 2, and then *move* this point over the entire parameter domain, where a weighted least squares fit is computed and evaluated for each point individually. It can be shown that the global function $f(\mathbf{x})$, obtained from a set of local functions

$$f(\mathbf{x}) = f_{\mathbf{x}}(\mathbf{x}), \quad \min_{f_{\mathbf{x}} \in \Pi_m^d} \sum_i \theta(\|\mathbf{x}-\mathbf{x}_i\|)\ \|f_{\mathbf{x}}(\mathbf{x}_i) - f_i\|^2 \tag{19}$$

is continuously differentiable if and only if the weighting function is continuously differentiable, see Levins work [Levin 1998; Levin 2003].

So instead of constructing the global approximation using Eqn. (15), we use Eqns. (8) and (13) (or (16) and (17)) and construct and evaluate a local polynomial fit continuously over the *entire* domain Ω, resulting in the MLS fit function. As previously hinted at, using (11) as the weighting function with a very small ε assigns weights close to infinity near the input data points, forcing the MLS fit function to interpolate the prescribed function values in these points. Therefore, by varying ε we can directly influence the approximatimg/interpolating nature of the MLS fit function.

4 Applications

Least Squares, Weighted Least Squares and Moving Least Squares, have become widespread and very powerful tools in Computer Graphics. They have been successfully applied to surface reconstruction from points [Alexa et al. 2003] and other point set surface definitions [Amenta and Kil 2004], interpolating and approximating implicit surfaces [Shen et al. 2004], simulating [Belytschko et al. 1996] and animating [Müller et al. 2004] elastoplastic materials, Partition of Unity implicits [Ohtake et al. 2003], and many other research areas.

In [Alexa et al. 2003] a point-set, possibly acquired from a 3D scanning device and therefore noisy, is replaced by a representation point set derived from the MLS surface defined by the input point-set. This is achieved by down-sampling (i.e. iteratively removing points which have little contribution to the shape of the surface) or up-sampling (i.e. adding points and projecting them to the MLS surface where point-density is low). The projection procedure has recently been augmented and further analyzed in the work of Amenta and Kil [Amenta and Kil 2004]. Shen et. al [Shen et al. 2004] use an MLS formulation to derive implicit functions from polygon soup. Instead of solely using value constraints at points (as shown in this report) they also add value constraints integrated over polygons and normal constraints.

References

ALEXA, M., BEHR, J., COHEN-OR, D., FLEISHMAN, S., LEVIN, D., AND T. SILVA, C. 2003. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics 9*, 1, 3–15.

AMENTA, N., AND KIL, Y. 2004. Defining point-set surfaces. In *Proceedgins of ACM SIGGRAPH 2004*.

BELYTSCHKO, T., KRONGAUZ, Y., ORGAN, D., FLEMING, M., AND KRYSL, P. 1996. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering 139*, 3, 3–47.

FRIES, T.-P., AND MATTHIES, H. G. 2003. Classification and overview of meshfree methods. Tech. rep., TU Brunswick, Germany Nr. 2003-03.

LANCASTER, P., AND SALKAUSKAS, K. 1981. Surfaces generated by moving least squares methods. *Mathematics of Computation 87*, 141–158.

LEVIN, D. 1998. The approximation power of moving least-squares. *Math. Comp. 67*, 224, 1517–1531.

LEVIN, D., 2003. Mesh-independent surface interpolation, to appear in 'geometric modeling for scientific visualization' edited by brunnett, hamann and mueller, springer-verlag.

MÜLLER, M., KEISER, R., NEALEN, A., PAULY, M., GROSS, M., AND ALEXA, M. 2004. Point based animation of elastic, plastic and melting objects. In *Proceedings of 2004 ACM SIGGRAPH Symposium on Computer Animation*.

OHTAKE, Y., BELYAEV, A., ALEXA, M., TURK, G., AND SEIDEL, H.-P. 2003. Multi-level partition of unity implicits. *ACM Trans. Graph. 22*, 3, 463–470.

PRESS, W., TEUKOLSKY, S., VETTERLING, W., AND FLANNERY, B. 1992. *Numerical Recipes in C - The Art of Scientific Computing*, 2nd ed. Cambridge University Press.

SHEN, C., O'BRIEN, J. F., AND SHEWCHUK, J. R. 2004. Interpolating and approximating implicit surfaces from polygon soup. In *Proceedings of ACM SIGGRAPH 2004*, ACM Press.

SHEPARD, D. 1968. A two-dimensional function for irregularly spaced data. In *Proc. ACM Nat. Conf.*, 517–524.

WENDLAND, H. 1995. Piecewise polynomial, positive definite and compactly supported radial basis functions of minimal degree. *Advances in Computational Mathematics 4*, 389–396.

Figure 2: The MLS surface of a point-set with varying density (the density is reduced along the vertical axis from top to bottom). The surface is obtained by applying the projection operation described by Alexa et. al. [2003]. Image courtesy of Marc Alexa.

Schriftliche Versicherung

Hiermit versichere ich,

1. dass ich die vorliegende Arbeit selbstständig verfasst habe,
2. dass ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe,
3. dass Entlehnungen aus dem Internet durch Ausdrucke belegt sind,
4. dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist und
5. dass das elektronische Exemplar der Arbeit mit den anderen Exemplaren übereinstimmt.

Bietigheim-Bissingen, den 14. August 2012

Julian Valentin