Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

B-Spline-Approximation mit automatischem Differenzieren

Katrin Bauer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. habil. Miriam Schulte
Betreuer/in:	Stefan Zimmer
Beginn am:	1. Juni 2021
Beendet am:	1. Dezember 2021

Kurzfassung

B-Splines spielen eine wichtige Rolle bei der Approximation von Funktionen. Die Wahl der Knoten beeinflusst die Form der Splines und damit die Qualität der Approximation. Die Knoten zu optimieren ist allerdings aufwändig. Bibliotheken wie JAX und PyTorch enthalten leistungsfähige Werkzeuge zum automatischen Differenzieren. Insbesondere für die Optimierung von Parametern eröffnet dies Möglichkeiten, die über klassische Techniken zur Parametrisierung weit hinausgehen.

Diese Bachelorarbeit untersucht die Umsetzung von Splineapproximationen in PyTorch und JAX. Die hier vorgestellte Implementierung ermöglicht die Ableitung des Approximationsfehlers nach den Knotenpositionen mittels automatischem Differenzieren. Um den Approximationsfehler zu minimieren, vergleichen wir bekannte Optimierungsalgorithmen, welche die Ableitung verwenden. Neben den Knoten ist der Polynomgrad ein zentraler Parameter für Splines. Diese Arbeit betrachtet Funktionsapproximationen mit Fractional Splines, welche die Ableitung nach dem Splinegrad ermöglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Die Optimierung von Parametern ist Ziel vieler Machine Learning Bibliotheken. Hierfür bieten diese Bibliotheken leistungsfähige Werkzeuge zum automatischen Differenzieren von Funktionen, welche nahezu beliebig implementiert sein können. PyTorch und JAX sind zwei solcher, weit verbreiteter Bibliotheken für die Programmiersprache Python.

Stückweise Polynome bzw. Splines kommen in vielen verschiedenen Anwendungen wie der Approximation von Funktionen oder Testdaten, der Computergrafik oder numerischen Simulationen zum Einsatz. Bei der Approximation einer Funktion wird eine einfache Repräsentation gesucht, die diese Funktion möglichst gut annähert und gleichzeitig möglichst wenig Parameter hat. Splines haben sich als solche Repräsentation bewährt. Eine oft verwendete Basis des Splineraums bilden die B-Splines dank ihres kompakten Trägers und Glattheit in den Knoten. Die Approximation mit B-Splines ist über den Splinegrad, die Koeffizienten der Basisfunktionen und die Positionen der Knoten parametrisiert. Aufgrund der Komplexität der Optimierung werden die Knoten meist fest gewählt, bevor die Koeffizienten mit Hilfe der Methode der kleinsten Quadrate anhand der Testdaten bestimmt werden können. Alternativ kann das Approximationsproblem als nicht-lineares Optimierungsproblem in den Knoten und Koeffizienten formuliert werden. Die Knoten optimal zu wählen erweist sich allerdings als schwierig. Bekannte Methoden zur Knotenwahl, welche die Ableitung nach den Knoten benötigen, nähern diese typischerweise mittels finiter Differenzen an [BR68; DT17; SS95]. Automatisches Differenzieren erlaubt es nun diese Ableitung exakt zu berechnen.

Neben den Knoten ist der Splinegrad ein zentraler Parameter für Splines. Unser und Blu [UB00] definieren Fractional Splines, welche Splines auf reelle Grade verallgemeinern. Die Implementierung von Fractional Splines erlaubt die Ableitung nach dem Splinegrad, sodass der Splinegrad optimiert werden kann.

Diese Arbeit stellt die Umsetzung der Splineapproximation in Python mit den Frameworks PyTorch und JAX vor. Die hier beschriebene Implementierung der Berechnung des Approximationsfehlers ermöglicht die Ableitung des Fehlers nach den Knotenpositionen. Die Berechnung der Ableitung ermöglicht die Verwendung bekannter Optimierungsalgorithmen, um den Approximationsfehler mittels Verschiebung der Knoten zu minimieren. Nach der Beschreibung der Implementierung untersucht diese die Abhängigkeit des Approximationsfehlers von den Knotenpositionen anhand einiger Beispielfunktionen und vergleicht verschiedene Optimierungsalgorithmen. Abschließend stellt diese Arbeit die Funktionsapproximation mit Fractional Splines vor, deren Implementierung die Ableitung des Fehlers nach dem Splinegrad ermöglicht.

Gliederung

Diese Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen führt die grundlegenden Definitionen dieser Arbeit ein.

Kapitel 3 – Umsetzung beschreibt die Implementierung der Splineapproximation in JAX und PyTorch.

Kapitel 4 – Ergebnisse zeigt den Einfluss der Knoten auf den Approximationsfehler anhand einiger Beispiele.

Kapitel 5 – Fractional Splines stellt eine Verallgemeinerung der B-Splines auf reelle Splinegrade vor, welche das Ableiten nach dem Splinegrad erlaubt.

Kapitel 6 – Zusammenfassung fasst die Ergebnisse der Arbeit zusammen.

2 Grundlagen

Dieses Kapitel führt in Abschnitt 2.1 die grundlegenden Definitionen ein. Dabei folgt die Notation größtenteils Höllig und Hörner [HH13]. Zunächst werden B-Splines sowie der von ihnen aufgespannte Splineraum definiert. Anschließend wird die Approximation von Testdaten mit Splines beschrieben und der Approximationsfehler definiert. Abschließend beschreibt Abschnitt 2.2 das Konzept des automatischen Differenzierens.

2.1 Definitionen

Den Splinegrad bezeichnen wir mit $k \in \mathbb{N}$ und die sortierte Folge der Knoten mit $\mathbf{t} = (t_0, \ldots, t_{n+k})$, wobei $n \in \mathbb{N}$ die Dimension des aufgespannten Splineraums ist. Über den Knoten $\mathbf{t}$ sind n B-Splines $b^k_{j,\mathbf{t}}$ definiert, welche über die folgende Rekursionsgleichung definiert werden können [HH13, Abschnitt 4.1]:

Definition 2.1.1 (B-Spline)
Sei $k \in \mathbb{N}$ der Splinegrad und $(t_0, \ldots, t_{n+k})$ eine Folge von Knoten. Sei $x \in \mathbb{R}$ und $j \in \{0, \ldots, n-1\}$. Der B-Spline $b^k_{j,\mathbf{t}}$ *für ist rekursiv definiert durch*

- *für $k = 0$*

$$b^0_{j,\mathbf{t}}(x) = \begin{cases} 1 & \text{falls } t_j \leq x < t_{j+1} \\ 0 & \text{sonst} \end{cases}$$

- *und für $k \geq 1$*

$$b^k_{j,\mathbf{t}}(x) = \gamma^{k-1}_{j,\mathbf{t}}(x) \cdot b^{k-1}_{j,\mathbf{t}}(x) + (1 - \gamma^{k-1}_{j+1,\mathbf{t}}(x)) \cdot b^{k-1}_{j+1,\mathbf{t}}(x)$$

  mit Gewichten

$$\gamma^{k-1}_{j,\mathbf{t}}(x) = \begin{cases} \frac{x - t_j}{t_{j+k} - t_j} & \text{falls } t_{j+k} = t_j \\ 0 & \text{sonst.} \end{cases}$$

Ein Vorteil dieser Definition ist, dass sie sich leicht implementieren lässt. Die genaue Implementierung wird in Abschnitt 3.2 beschrieben.

Eine wichtige Eigenschaft der B-Splines ist zudem ihr kompakter Träger. Ein B-Spline $b^k_{j,\mathbf{t}}$ hat den kompakten Träger $[t_j, t_{j+k+1})$ und ist durch die Knoten $\{t_j, \ldots, t_{j+k+1}\}$ vollständig definiert. Wir verlangen im Folgenden, dass

$$t_j < t_{j+k+1}$$

für $j \in \{0, \ldots, n-1\}$ gilt, damit der Träger keines B-Splines leer ist. Auf dem Intervall $[t_k, t_n)$ bilden B-Splines eine Partition der konstanten Eins-Funktion und spannen den Splineraum auf.

Definition 2.1.2 (Splines und Splineraum)
Sei $k \in \mathbb{N}$ der Splinegrad und $(t_0, \ldots, t_{n+k})$ eine Folge von Knoten. Ein Spline *ist eine Linearkombination aus B-Splines*

$$s^k_{\mathbf{t},\mathbf{c}} : [t_k, t_n) \to \mathbb{R}, \qquad s^k_{\mathbf{t},\mathbf{c}}(x) = \sum_{j=0}^{n-1} c_j \cdot b^k_{j,\mathbf{t}}(x)$$

für $x \in [t_k, t_n)$ und Koeffizienten $\mathbf{c} = (c_0, \ldots, c_{n-1})$. Den Splineraum $S^k_{\mathbf{t}}$ *über den Knoten* $\mathbf{t}$ *definieren wir als*

$$S^k_{\mathbf{t}} = \{s^k_{\mathbf{t},\mathbf{c}} \mid \mathbf{c} \in \mathbb{R}^n\}.$$

Splines werden verwendet um Funktionen zu approximieren. Um den Approximationsfehler eines Splines zu bestimmen, tasten wir die Funktion in m Auswertestellen ab. Dabei ist die Anzahl der Testdaten typischerweise deutlich größer als die Anzahl der Basisfunktionen ($m \gg n$). Als Fehler verwenden wir die quadrierte euklidische Distanz zwischen den abgetasteten Funktionswerten und den Splinewerten. Zudem normieren wir den Fehler durch die Anzahl m der Auswertestellen.

Definition 2.1.3 (Approximationsfehler)
Sei $k \in \mathbb{N}$ der Splinegrad und $(t_0, \ldots, t_{n+k})$ eine Folge von Knoten. Gegeben seien Paare aus Auswertestellen und Funktionswerten $(x_0, f_0), \ldots, (x_{m-1}, f_{m-1})$. Der Approximationsfehler eines Splines *s ist definiert als*

$$E(s) = \frac{1}{m} \sum_{i=0}^{m-1} (s(x_i) - f_i)^2.$$

Das Ziel der Splineapproximation ist nun den Spline zu finden, der die Testdaten am besten approximiert. Für eine feste Wahl der Knoten $\mathbf{t}$ lassen sich die Koeffizienten, die den Approximationsfehler minimieren, mit der Methode der kleinsten Quadrate bestimmen [Boo78, S. 249–255]. Den resultierenden Spline nennen wir ℓ^2-Bestapproximationsspline.

Definition 2.1.4 (ℓ^2-Bestapproximationsspline)
Sei $k \in \mathbb{N}$ der Splinegrad und $\mathbf{t} = (t_0, \ldots, t_{n+k})$ eine Folge von Knoten. Gegeben seien Paare aus Auswertestellen und Funktionswerten $(x_0, f_0), \ldots, (x_{m-1}, f_{m-1})$. Der ℓ^2-Bestapproximationsspline *über der Knotenfolge $\mathbf{t}$ ist der Spline*

$$s^k_{\mathbf{t}} = \operatorname*{argmin}_{s \in S^k_{\mathbf{t}}} E(s).$$

Der ℓ^2-Bestapproximationsspline $s^k_{\mathbf{t}}$ ist genau dann eindeutig definiert, wenn die *Schoenberg-Whitney-Bedingungen* erfüllt sind, d. h.

$$\forall j \in \{0, 1, \ldots, n-k-2\}\ \exists x_{i_j} : \quad t_j < x_{i_j} < t_{j+k+1}$$

gilt [SW53]. Im Folgenden gehen wir immer davon aus, dass diese Bedingungen erfüllt sind.

Ein wichtiges Werkzeug zur Berechnung des ℓ^2-Bestapproximationsspline ist die Matrix der Funktionswerte des B-Splines in den Auswertestellen.

Definition 2.1.5 (Interpolationsmatrix)
Sei $k \in \mathbb{N}$ der Splinegrad und $(t_0, \ldots, t_{n+k})$ eine Folge von Knoten. Seien weiter $x_0, \ldots, x_n$ Auswertestellen mit $m \geq n$. Die Interpolationsmatrix *ist definiert als*

$$A = (b^k_{j,\mathbf{t}}(x_i))_{\substack{0 \leq i < m \\ 0 \leq j < n}} \in \mathbb{R}^{m \times n}.$$

Mit der Interpolationsmatrix A lässt sich das Problem die Koeffizienten $\mathbf{c} \in \mathbb{R}^n$ des ℓ^2-Bestapproximationssplines zu finden als lineares Minimierungsproblem formulieren:

$$E(s^k_{\mathbf{t}}) = \min_{\mathbf{c} \in \mathbb{R}^n} E(s^k_{\mathbf{t},\mathbf{c}}) = \frac{1}{m} \cdot \min_{\mathbf{c} \in \mathbb{R}^n} \|A\mathbf{c} - \mathbf{f}\|_2^2.$$

Neben den optimalen Koeffizienten kann die Splineapproximation auch als nicht-lineares Minimierungsproblem in den Knoten $\mathbf{t}$ formuliert werden:

$$\min_{\mathbf{t} \in \mathbb{R}^{n+k}} E(s^k_{\mathbf{t}})$$

Während sich die optimalen Koeffizienten effizient mit der Methode der kleinsten Quadrate bestimmen lassen, ist die Optimierung der Knoten deutlich aufwändiger. Die Existenz der optimalen Knoten ist nur gegeben, falls mehrfache Knoten, d. h. identische Knoten $t_j = t_{j+1}$, erlaubt werden [BR68]. Die optimalen Knoten sind aber im Allgemeinen nicht eindeutig. Zudem kann es lokale Minima geben, deren Knoten nicht das globale Optimum sind. Wir beschränken uns im Folgenden auf die Suche nach einem lokalen Minimum.

In dieser Arbeit wird die Optimierung der Knoten anhand ihrer Ableitung vorgestellt. Hierfür wird der Gradient des Approximationsfehlers bezüglich der Knoten

$$\nabla_{\mathbf{t}} E(s^k_{\mathbf{t}})$$

berechnet. Während die analytische Berechnung dieser Ableitung sehr aufwändig ist, ist die Bestimmung mit automatischem Differenzieren einfach umzusetzen.

2.2 Automatisches Differenzieren

Automatisches Differenzieren ist eine Methode zur exakten Berechnung der Ableitung von Funktionen, welche als Programme gegeben sind [CG93]. Konzeptionell wird die Ableitung berechnet, indem die Kettenregel auf einzelne arithmetische Operationen angewandt wird, deren Ableitung bereits bekannt ist. Durch Verwendung eines Frameworks in Kombination mit der Überladung von Operatoren können nahezu beliebige Funktionen abgeleitet werden, ohne dass zusätzlicher Programmieraufwand entsteht. In dieser Arbeit wird die Implementierung der Splineapproximation in der Programmiersprache Python mit den Frameworks PyTorch [PGM+19] und JAX [BFH+18] vorgestellt. Wie in Abschnitt 3.6 gezeigt wird, erfordert das Berechnen der Ableitung mittels automatischem Differenzieren nur wenige zusätzliche Codezeilen.

3 Umsetzung

Dieses Kapitel betrachtet die Implementierung der Berechnung des Approximationsfehlers der Splineapproximation.

In Abschnitt 3.1 werden zunächst die Frameworks PyTorch und JAX vorgestellt. Anschließend wird die Auswertung der B-Splines in Abschnitt 3.2 beschrieben. Diese setzt das Einsortieren der Auswertestellen in die Knotenintervalle voraus, welches Abschnitt 3.3 beschrieben wird. In Abschnitt 3.4 werden die beschriebenen Algorithmen zusammengesetzt um die Interpolationsmatrix aufzustellen. Abschnitt 3.5 zeigt die Berechnung des Approximationsfehlers. Darauf aufbauend beschreibt Abschnitt 3.6 das Differenzieren des Fehlers und Abschnitt 3.7 die Anwendung der Methoden der Optimierer.

3.1 Allgemeines zu den Frameworks

Im Rahmen dieser Arbeit wurde die Splineapproximation in den Frameworks JAX [BFH+18] und PyTorch [PGM+19] implementiert. Sowohl JAX als auch PyTorch sind Machine Learning Frameworks, die Vektoroperationen sowie deren automatisches Differenzieren anbieten. Beide Frameworks bieten eine ähnliche API wie das Python-Modul `numpy`[1]. Insbesondere unterstützen beide Frameworks Broadcasting, welches effiziente elementweise Vektoroperationen ermöglicht. Während `numpy` allerdings nur die Ausführung auf der CPU unterstützt, können Funktionen von PyTorch und JAX auch auf GPUs und TPUs ausgeführt werden.

Broadcasting Die Vektorklassen der beiden Frameworks überladen arithmetische sowie boolsche Operatoren und verallgemeinern diese auf elementweise Vektoroperationen. Dabei müssen die beteiligten Vektoren nicht notwendigerweise die gleiche Größe haben. Broadcasting[2] beschreibt die Behandlung von Vektoren mit unterschiedlicher Größe bei solchen Vektoroperationen. Die Verwendung von Broadcasting statt einer Schleife in Python ist häufig deutlich effizienter, da die Frameworks in einer kompilierten Sprache wie C über die Arrays iterieren. Ein weiterer Vorteil ist, dass ein Programm oft für verschieden große Eingabearrays wiederverwendet werden kann. Listing 3.1 zeigt einige Beispiele für häufige Vektoroperationen. Tabelle 3.1 gibt für die Operationen in den Zeilen 17, 19 und 25 die Größen der beteiligten Arrays an, welche einige Regeln des Broadcastings demonstrieren. In Zeile 17 sind die beiden Arrays `a` und `b` gleich groß, sodass sie elementweise addiert werden können. In Zeile 19 hingegen haben die Arrays `a` und `d` unterschiedlich viele Dimensionen. Beim Broadcasting von Arrays mit unterschiedlich vielen Dimensionen werden diese an der innersten (bzw. rechten) Dimension ausgerichtet. Danach werden die Dimensionen

[1] NumPy-Dokumentation, https://numpy.org/

[2] Broadcasting, https://numpy.org/doc/stable/user/basics.broadcasting.html

von rechts nach links verglichen. Zwei Dimensionen sind kompatibel, wenn sie gleich sind oder eine der beiden Dimensionen 1 ist. Falls eine Dimension 1 ist, wird das zugehörige Array entlang dieser Dimension gebroadcastet. Um die Dimension, entlang welcher gebroadcastet werden soll zu verschieben, kann wie in Zeile 23 mithilfe eines `None` im Index-Operator eine neue Dimension der Größe 1 eingefügt werden. Danach kann das Array `c[:, None]` über `a` gebroadcastet werden.

Listing 3.1 Beispiele für Vektoroperationen und Broadcasting

```
1  # Import eines der Module
2  from torch import tensor as array, where
3  from jax.numpy import array, where
4  from numpy import array, where
5
6  a = array([[1, 2, 3], [4, 5, 6]])                    # a.shape == (2, 3)
7  b = array([[0, 2, 4], [6, 8, 10]])                   # b.shape == (2, 3)
8  c = array([1, 2])                                    # c.shape == (2,)
9  d = array([1, 2, 10])                                # d.shape == (3,)
10 e = array([11, 12, 13])                              # e.shape == (3,)
11 f = array([True, False, True])                       # f.shape == (3,)
12 g = array([1, 1, 0, 2, 1])                           # g.shape == (5,)
13
14 # Skalare Multiplikation
15 3 * a == array([[3, 6, 9], [12, 15, 18]])           # shape == (2, 3)
16 # Elementweise Addition
17 a + b == array([[1, 4, 7], [10, 13, 16]])           # shape == (2, 3)
18 # Elementweise Addition mit Broadcasting von d
19 a + d == array([[2, 4, 13], [5, 7, 16]])            # shape == (2, 3)
20 # Differenz der benachbarten Vektoreinträge
21 d[1:] - d[:-1] == array([1, 8])                     # shape == (2,)
22 # Einfügen einer Dimension
23 c[:, None] == array([[1], [2]])                     # shape == (2, 1)
24 # Elementweise Addition mit Broadcasting von c[:, None]
25 a + c[:, None] == array([[2, 3, 4], [6, 7, 8]])     # shape == (2, 3)
26 # Auswählen einzelner Einträge aus d mit Bit-Maske f
27 d[f] == array([1, 10])                              # shape == (2,)
28 # Auswählen einzelner Einträge aus d mit Index-Array g
29 d[g] == array([2, 2, 1, 10, 2])                     # shape == (5,)
30 # Auswählen aus zwei Arrays mit der Funktion where
31 where(f, d, e) == array([1, 12, 10])                # shape == (3,)
32 where(f, d, e) == array([d[j] if f[j] else e[j] for j in range(3)])
```

Zeile 17	
`a`	$: 2 \times 3$
`b`	$: 2 \times 3$
`a + b`	$: 2 \times 3$

Zeile 19	
`a`	$: 2 \times 3$
`d`	$: 3$
`a + d`	$: 2 \times 3$

Zeile 25	
`a`	$: 2 \times 3$
`c[:, None]`	$: 2 \times 1$
`a + c[:, None]`	$: 2 \times 3$

Tabelle 3.1: Größen der Arrays beim Broadcasting in Listing 3.1 in den Zeilen 17, 19 und 25

Vergleich zu NumPy Sowohl PyTorch als auch JAX bieten eine ähnliche API wie NumPy. Beispielsweise lassen sich die Operationen in Listing 3.1 mit jedem der drei Module ausführen. Analog geben auch die Abschnitte 3.2 bis 3.5 zum Auswerten des Approximationsfehlers häufig nur

ein Programm an, welches sich mit beiden Frameworks ausführen lässt. Trotz der Ähnlichkeit gibt es allerdings auch einige Unterschiede, die bei der Überführung von `numpy`-Code in die jeweiligen Frameworks beachtet werden müssen. Der folgende Abschnitt fasst einige dieser Unterschiede zusammen, die mir während dieser Arbeit begegnet sind.

Der wesentliche Unterschied zwischen PyTorch und NumPy ist die Bezeichnung einiger Funktionen. Tabelle 3.2 zeigt einige Beispiele für unterschiedlich benannte Funktionen.

Semantik	torch	numpy	jax.numpy
Erstellen eines Arrays	`arr = tensor([1, 2, 3])`	`arr = array([1, 2, 3])`	`arr = array([1, 2, 3])`
Konkatenieren	`cat((arr1, arr2))`	`concatenate((arr1, arr2))`	`concatenate((arr1, arr2))`
Transponieren	`arr.t()`	`arr.transpose()`	`arr.transpose()`
Index-Update	`arr[1] = 2`	`arr[1] = 2`	`arr = arr.at[1].set(2)`

Tabelle 3.2: Vergleich einiger Funktionen von `torch`, `numpy` und `jax.numpy`

Während PyTorch dem imperativen Programmierparadigma folgt, basiert JAX auf funktionaler Programmierung. Eine wichtige Folge dieser Designentscheidung ist, dass JAX-Arrays immutable sind, d. h. keine In-Place-Updates erlauben. Weitere Auswirkungen zeigen sich erst bei Funktionstransformationen wie der Just-In-Time-Kompilierung (JIT). Um JAX-Programme effizient auszuführen, bietet JAX einen Just-In-Time-Compiler an. Dieser basiert auf dem Tracen der Operationen einer Programmausführung und anschließender optimierter Kompilierung. Dies schränkt die Kontrollfluss-Primitive ein, die ein Programm verwenden darf, da beim Tracing jede Operation auch ausgeführt werden muss. Als Folge können Verzweigungen nicht JIT kompiliert werden, sondern müssen durch JAX-Funktionen wie `jax.numpy.where` oder `jax.lax.cond` ersetzt werden. Ebenso können Schleifen, deren Schleifenbedingung vom Wert der Eingabe abhängt, nicht JIT-kompiliert werden. Des Weiteren basiert der JIT-Compiler auf dem Konzept der reinen Funktionen. Globale Nebeneffekte wie das Schreiben globaler Variablen oder das Ausgeben von Werten in der Konsole werden beim Tracing nicht beachtet und in der kompilierten Funktion nicht immer ausgeführt.

Ein weiteres Konzept, das insbesondere beim Benchmarken von JAX-Funktionen wichtig ist, ist der *Asynchronous Dispatch*[3]. Dabei versucht JAX Overhead durch den Python-Interpreter zu reduzieren, indem es Operationen nebenläufig ausführt, während es die Kontrolle an Pythons Kontrollfluss zurückgibt. Beim Benchmarken muss daher sichergestellt werden, dass nicht nur die Zeit zum Starten der Berechnung gemessen wird. Hierfür bietet JAX die Funktion `block_until_ready()`, die auf das Ergebnis wartet. Alle JAX-Benchmarks in dieser Arbeit wurden mit `jit` kompilierten Programmen ausgeführt, auf deren Ergebnis mit `block_until_ready()` gewartet wurde.

Für einige der vorgestellten Programme wurden die Laufzeiten auf einer CPU gemessen. Der verwendete Prozessor ist eine Intel® Core™ i9-10980XE CPU mit 3.00 GHz und 64 GiB. Alle Berechnungen verwenden einfache Genauigkeit. Vor der Messung wurden die Programme mindestens einmal ausgeführt. JAX-Programme werden während dieser ersten Ausführung JIT-kompiliert. Eine wichtige Einschränkung der Vergleichbarkeit der Laufzeiten ist, dass die Anzahl der Threads nicht festgelegt wurde. Während PyTorch sechs Threads verwendet, verwendet JAX standardmäßig nur einen Thread.

[3] Asynchronous Dispatch, https://jax.readthedocs.io/en/4510-2/async_dispatch.html

Beide Frameworks werden aktiv entwickelt. Diese Arbeit verwendet JAX in der Version 0.2.24[4] sowie PyTorch in der Version 1.9.1[5].

3.2 Auswertung der B-Splines

Der folgende Abschnitt beschreibt die Implementierung der Auswertung einzelner B-Splines. Hierfür leitet Abschnitt 3.2.1 zunächst die verwendeten Algorithmen her. Anschließend stellt Abschnitt 3.2.2 die Implementierung in Python vor.

3.2.1 Algorithmen

Um die Interpolationsmatrix aufzustellen, werden die Werte der B-Splines in jeder Auswertestelle x_i benötigt. Im Folgenden betrachten wir zunächst einen Algorithmus für die Auswertung eines einzelnen B-Splines. Anschließend wird die Berechnung einer Matrixzeile, d. h. die Auswertung mehrerer B-Splines in einer Auswertestelle, beschrieben. Die Berechnung der gesamten Matrix ergibt sich durch Iterieren über die Auswertestellen.

Die rekursive Definition 2.1.1 der B-Splines lässt sich direkt in einen iterativen Algorithmus zu deren Auswertung überführen. Algorithmus 3.1 beschreibt einen einzelnen Rekursionsschritt in Pseudo-Code. Zur einfacheren Notation späterer Algorithmen wird der Rekursionsschritt für mehrere benachbarte B-Splines ausgeführt. Der Algorithmus aggregiert daher $n + 1$ benachbarte Splinewerte zu n Splinewerten von nächsthöherem Grad. Die Schleife über j in Zeile 7 iteriert über die B-Splines. Die Berechnung der Gewichte und der Rekursionsvorschrift wird in den Zeilen 8 bis 10 umgesetzt. Abbildung 3.1 skizziert die Berechnung. Die Laufzeit des Algorithmus 3.1 ist offensichtlich in $O(n)$.

Algorithmus 3.1 Rekursionsschritt bei der Auswertung eines B-Splines

1: **Input**
2: x Auswertestelle $x \in \mathbb{R}$
3: l Splinegrad $l \geq 1$
4: $\mathbf{t}$ Knoten $\mathbf{t} = (t_0, \ldots, t_{n+l}) \in \mathbb{R}^{n+l+1}$
5: $\mathbf{b}$ Splinewerte vom Grad $l-1$, d. h. $\mathbf{b} = (b^{l-1}_{0,\mathbf{t}}, \ldots, b^{l-1}_{n,\mathbf{t}})$
6: **procedure** RECURSIVE_STEP($x, l, (t_0, \ldots, t_{n+l}), (b^{l-1}_{0,\mathbf{t}}, \ldots, b^{l-1}_{n,\mathbf{t}})$)
7: **for** $j = 0, \ldots, n-1$ **do**
8: **if** $t_{j+l} = t_j$ **then** $\gamma^{l-1}_{j,\mathbf{t}} \leftarrow \frac{x - t_j}{t_{j+l} - t_j}$ **else** $\gamma^{l-1}_{j,\mathbf{t}} \leftarrow 0$ // Berechne Gewichte
9: **if** $t_{j+1+l} = t_{j+1}$ **then** $\gamma^{l-1}_{j+1,\mathbf{t}} \leftarrow \frac{x - t_{j+1}}{t_{j+1+l} - t_{j+1}}$ **else** $\gamma^{l-1}_{j+1,\mathbf{t}} \leftarrow 0$
10: $b^{l}_{j,\mathbf{t}} \leftarrow \gamma^{l-1}_{j,\mathbf{t}} \cdot b^{l-1}_{j,\mathbf{t}} + (1 - \gamma^{l-1}_{j+1,\mathbf{t}}) \cdot b^{l-1}_{j+1,\mathbf{t}}$ // Berechne Splinewert vom Grad l
11: **end for**
12: **return** $(b^{l}_{0,\mathbf{t}}, \ldots, b^{l}_{n-1,\mathbf{t}})$
13: **end procedure**

[4] JAX GitHub, https://github.com/google/jax/tree/jax-v0.2.24
[5] PyTorch GitHub, https://github.com/pytorch/pytorch/tree/v1.9.1

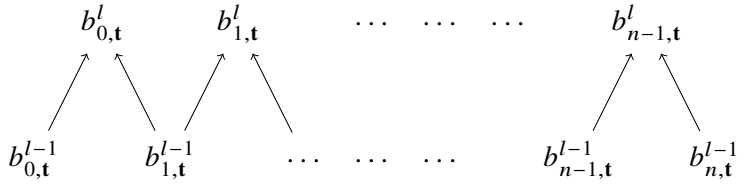

Abbildung 3.1: Skizze eines Rekursionsschritts entsprechend Algorithmus 3.1

Algorithmus 3.2 nutzt diese Rekursionsschritte nun, um einen einzelnen B-Spline auszuwerten. Da ein B-Spline vom Grad k durch die $k+2$ Knoten in seinem Träger vollständig definiert ist und um die Notation zu vereinfachen, betrachtet Algorithmus 3.2 einen Knotenvektor $\mathbf{t}$ aus $k+2$ Knoten. In der ersten Schleife in den Zeilen 6 bis 8 werden zunächst die $k+1$ Splinewerte $b^0_{0,\mathbf{t}}, \ldots, b^0_{k,\mathbf{t}}$ vom Grad 0 initialisiert. Die zweite Schleife in den Zeilen 9 bis 11 setzt die Rekursion über den Splinegrad um. Abbildung 3.2 skizziert die Struktur der Auswertung eines B-Splines entsprechend Algorithmus 3.2 am Beispiel eines quadratischen Splines ($k = 2$).

Algorithmus 3.2 Auswertung eines einzelnen B-Splines

1: **Input**
2: x Auswertestelle $x \in \mathbb{R}$
3: k Splinegrad $k \geq 1$
4: $\mathbf{t}$ Knoten $\mathbf{t} = (t_0, \ldots, t_{k+1}) \in \mathbb{R}^{k+2}$
5: **procedure** B_SPLINE($x, k, (t_0, \ldots, t_{k+1})$)
6: **for** $j = 0, \ldots, k$ **do** // Initialisiere Splinewerte vom Grad 0
7: **if** $t_j \leq x < t_{j+1}$ **then** $b^0_{j,\mathbf{t}} \leftarrow 1$ **else** $b^0_{j,\mathbf{t}} \leftarrow 0$
8: **end for**
9: **for** $l = 1, \ldots, k$ **do** // Schleife statt rekursiver Auswertung
10: $(b^l_{0,\mathbf{t}}, \ldots, b^l_{k-l,\mathbf{t}}) \leftarrow$ RECURSIVE_STEP$(x, k, (t_0, \ldots, t_{k+1}), (b^{l-1}_{0,\mathbf{t}}, \ldots, b^{l-1}_{k-l+1,\mathbf{t}}))$
11: **end for**
12: **return** $b^k_{0,\mathbf{t}} = b^k_{0,\mathbf{t}}(x)$
13: **end procedure**

Zur Laufzeitanalyse des Algorithmus 3.2 betrachten wir, wie oft Zeile 10 in Algorithmus 3.1 ausgeführt wird. Die der l-ten Iteration der Schleife in Zeile 3.2 aggregiert $k - l + 2$ Splinewerte zu $n = k - l + 1$ Werten. Der Aufwand der l-ten Iteration ist daher $k - l + 1$. Damit ist der Gesamtaufwand zur Auswertung eines einzelnen Matrixeintrags

$$\sum_{l=1}^{k}(k - l + 1) = \sum_{l=1}^{k} l = \frac{k(k+1)}{2} \in O(k^2).$$

Die Auswertung eines B-Splines in einer Auswertestelle entspricht einem einzelnen Matrixeintrag der Interpolationsmatrix. Im Folgenden betrachten wir die Berechnung einer Matrixzeile, d. h. die Auswertung mehrerer B-Splines in einer festen Auswertestelle.

Algorithmus 3.3 wertet jeden Matrixeintrag einer Zeile getrennt aus, indem er über die B-Splines iteriert. Die Anzahl der Operationen dieses Algorithmus liegt in $O(n \cdot k^2)$. Er lässt sich allerdings noch deutlich optimieren.

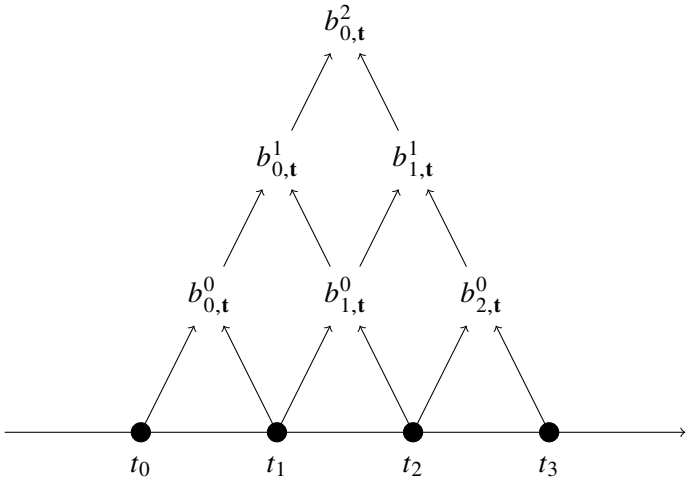

Abbildung 3.2: Skizze der rekursiven Auswertung eines quadratischen B-Splines entsprechend Algorithmus 3.2

Algorithmus 3.3 Naive Auswertung einer Zeile der Interpolationsmatrix

1: **Input**
2: x Auswertestelle $x \in \mathbb{R}$
3: k Splinegrad $k \in \mathbb{N}$
4: $\mathbf{t}$ Knoten $\mathbf{t} = (t_0, \ldots, t_{n+k}) \in \mathbb{R}^{n+k+1}$
5: **procedure** EVAL_B_SPLINE_NAIVE($x, k, (t_0, \ldots, t_{n+k})$)
6: **for** $j = 0, \ldots, n-1$ **do**
7: $b^k_{j,\mathbf{t}} \leftarrow$ B_SPLINE($x, k, (t_j, \ldots, t_{j+k+1})$)
8: **end for**
9: **return** $(b^k_{0,\mathbf{t}}, \ldots, b^k_{n-1,\mathbf{t}})$
10: **end procedure**

Die erste hier vorgestellte Optimierung nutzt die Struktur der rekursiven Auswertung. In Algorithmus 3.3 werden in jedem Aufruf von B_SPLINE($x_i, k, (t_j, \ldots, t_{j+k+1})$) die B-Splines von niedrigerem Grad $l < k$ neu ausgewertet.

Dabei führt Algorithmus 3.3 das Dreiecksschema aus Abbildung 3.2 für jeden Matrixeintrag getrennt aus. Die Struktur der Rekursionsvorschrift für mehrere benachbarte B-Splines ist in Abbildung 3.3 skizziert. In dem Beispiel in Abbildung 3.3 würde Algorithmus 3.3 den ersten Splinewert $b^2_{0,\mathbf{t}}$ mittels dem Funktionsaufruf B_SPLINE($x, k, (t_0, \ldots, t_3)$) berechnen. Bei diesem Aufruf werden zunächst die Werte $b^0_{0,\mathbf{t}}, b^0_{1,\mathbf{t}}, b^0_{2,\mathbf{t}}$, daraus die Werte $b^1_{0,\mathbf{t}}$ und $b^1_{1,\mathbf{t}}$ und damit der Splinewert $b^2_{0,\mathbf{t}}$ auswertet. Zur Berechnung des nächsten Matrixeintrags $b_{1,\mathbf{t}}$ würden danach $b^0_{1,\mathbf{t}}, b^0_{2,\mathbf{t}}, b^0_{3,\mathbf{t}}$, anschließend $b^1_{1,\mathbf{t}}, b^1_{2,\mathbf{t}}$ und daraus $b^2_{1,\mathbf{t}}$ ausgewertet. Insbesondere werden also die Zwischenergebnisse $b_{1,\mathbf{t}}, b_{2,\mathbf{t}}$ und $b^1_{1,\mathbf{t}}$ mehrfach ausgewertet.

Dies lässt sich vermeiden, indem jedes Zwischenergebnis bzw. jeder Pfeil in Abbildung 3.3 genau einmal ausgewertet wird. Algorithmus 3.4 zeigt den modifizierten Algorithmus. Dabei iteriert die äußere Schleife in Zeile 9 in Algorithmus 3.4 über die Splinegrade. Die Schleife über

die Matrixeinträge wurde mit der inneren Schleife in RECURSIVE_STEP kombiniert. Tatsächlich unterscheidet er sich von Algorithmus 3.2 für einen einzelnen B-Spline nur in den Schleifenvariablen und Länge des Spline-Arrays. Im Gegensatz zu Algorithmus 3.2 wertet Algorithmus 3.4 allerdings eine ganze Matrixzeile aus.

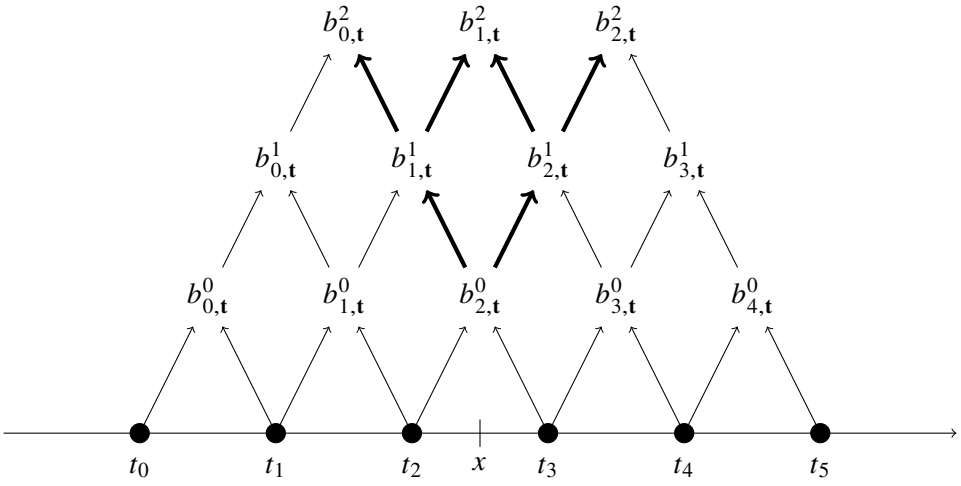

Abbildung 3.3: Skizze der rekursiven Auswertung von mehreren quadratischen B-Splines

Algorithmus 3.4 Auswertung einer Zeile der Interpolationsmatrix ohne mehrfache Auswertungen

Input
  x Auswertestelle $x \in \mathbb{R}$
  k Splinegrad $k \in \mathbb{N}$
  $\mathbf{t}$ Knoten $\mathbf{t} = (t_0, \ldots, t_{n+k}) \in \mathbb{R}^{n+k+1}$
procedure EVAL_B_SPLINE_IMPROVED($x, k, (t_0, \ldots, t_{n+k})$)
  for $j = 0, \ldots, n+k-1$ **do** // Initialisiere Splinewerte vom Grad 0
    if $t_j \le x < t_{j+1}$ **then** $b^0_{j,\mathbf{t}} \leftarrow 1$ **else** $b^0_{j,\mathbf{t}} \leftarrow 0$
  end for
  for $l = 1, \ldots, k$ **do**
    $(b^l_{0,\mathbf{t}}, \ldots, b^l_{n+k-l-1,\mathbf{t}}) \leftarrow$ RECURSIVE_STEP$(x, l, (t_0, \ldots, t_{k+1}), (b^{l-1}_{0,\mathbf{t}}, \ldots, b^{l-1}_{n+k-l,\mathbf{t}}))$
  end for
  return $(b^k_{0,\mathbf{t}}, \ldots, b^k_{n-1,\mathbf{t}})$
end procedure

Eine weitere, naheliegende Optimierung ist die Nutzung des kompakten Trägers der B-Splines. Daraus resultiert, dass die Interpolationsmatrix nur dünn besetzt ist. Genauer ist ein B-Spline $b^k_{j,\mathbf{t}}$ nur im Intervall $[t_j, t_{j+k+1})$ ungleich Null. In einer feste Auswertestelle x in einem Knotenintervall $t_j \le x < t_{j+1}$ sind also nur die B-Splines $b^k_{j-k,\mathbf{t}}, \ldots, b^k_{j,\mathbf{t}}$ ungleich Null. Somit reicht es diese $k+1$ B-Splines in der Auswertestelle x explizit auszuwerten. Analog können auch die Auswertungen der Splinewerte von niedrigerem Splinegrad reduziert werden. Beispielsweise folgt aus $t_2 \le x < t_3$ in

Abbildung 3.3, dass nur der stückweise konstante B-Spline $b^0_{2,\mathbf{t}}$ in x ungleich Null ist. Entsprechend sind von den linearen B-Splinewerten nur $b^1_{1,\mathbf{t}}$ und $b^2_{2,\mathbf{t}}$ ungleich Null. In Abbildung 3.3 beschreiben die dicken Pfeile Beiträge ungleich Null. Algorithmus 3.5 wendet diese Optimierungen an.

Algorithmus 3.5 Optimierte Auswertung der nicht-verschwindenden Matrixeinträge einer Zeile

Input
  x Auswertestelle $x \in \mathbb{R}$
  $\mathbf{t}$ Knoten $\mathbf{t} = (t_0, \ldots, t_{2k+1})$ mit $t_k \leq x < t_{k+1}$
procedure EVAL_B_SPLINE_OPTIMIZED(x, $(t_0, \ldots, t_{2k+1})$)
  $b^0_{k,\mathbf{t}} \leftarrow 1$
  for $l = 1, \ldots, k$ **do**
    $b^{l-1}_{k-l,\mathbf{t}} \leftarrow 0$
    $b^{l-1}_{k+1,\mathbf{t}} \leftarrow 0$
    $(b^l_{k-l,\mathbf{t}}, \ldots, b^l_{k,\mathbf{t}}) \leftarrow$ RECURSIVE_STEP$(x, l, (t_{k-l}, \ldots, t_{k+l+1}), (b^{l-1}_{k-l,\mathbf{t}}, \ldots, b^{l-1}_{k+1,\mathbf{t}}))$
  end for
  return $(b^k_{0,\mathbf{t}}, \ldots, b^k_{k,\mathbf{t}})$
end procedure

Genau wie in Algorithmus 3.2 ist die Anzahl der Auswertungen einzelner B-Splines in Algorithmus 3.5

$$\sum_{l=1}^{k}(k-l+1) = \frac{k(k+1)}{2} \in O(k^2).$$

Insbesondere ist damit der Aufwand der Auswertung einer Matrixzeile unabhängig von der Anzahl der Basisfunktionen. Im Gegensatz zu Algorithmus 3.4 setzt dieser Schritt allerdings voraus, dass die Auswertestellen zuvor in die Knotenintervalle einsortiert werden. Dieser Sortierschritt wird in Abschnitt 3.3 genauer beschrieben.

Die beschriebenen Algorithmen lassen sich direkt in Python-Programme überführen. Im folgenden Abschnitt wird die Implementierung in PyTorch und JAX vorgestellt.

3.2.2 Implementierung

Das Listing 3.2 zeigt die Implementierung der Auswertung eines B-Splines entsprechend Algorithmus 3.2. Das Programm kann sowohl mit JAX als auch PyTorch verwendet werden. Das verwendete Framework wird mit einem der `import`-Befehle in den ersten zwei Zeilen ausgewählt. Mittels Broadcasting und Verwendung unterschiedlicher Eingabegrößen kann das selbe Programm auch für die Auswertung einer Matrixzeile wie in Algorithmus 3.3 und Algorithmus 3.4 verwendet werden. Die Eingabegröße `t.shape = (k+2, n, m)` entspricht Algorithmus 3.3, welcher verschachtelte Schleifen für die Auswertung Matrixeinträge und der Rekursionsvorschrift verwendet. Die Arrayeinträge sind dabei $\texttt{t}[j_1, j_2, i] = t_{j_2+j_1}$ für den Knotenvektor $(t_0, \ldots, t_{n+k}) \in \mathbb{R}^{n+k+1}$. Die erste Dimension entspricht dem Knotenvektor, welcher als Eingabe für Algorithmus 3.2 verwendet wird. Die mittlere Dimension entspricht einer Schleife über die n Basisfunktionen bzw. Spalten der Interpolationsmatrix. Die letzte Dimension entspricht einer Schleife über die m Auswertestellen bzw. Zeilen der Interpolationsmatrix.

Wie in Algorithmus 3.4 beschrieben, lässt sich die Schleife über die Matrixeinträge einer Zeile mit der Schleife über die Rekursionsvorschrift kombinieren. Für die Umsetzung muss lediglich die Eingabegröße `t.shape = (n+k+1, m)` mit `t[`j, i`] =` t_j verwendet werden.

Die Verwendung von Broadcasting statt einer Schleife setzt voraus, dass die Schleifeniterationen unabhängig voneinander sind, d. h. dass sie nicht das Ergebnis vorheriger Iterationen verwenden. Da dies für die Schleife über die Splinegrade l nicht der Fall ist, kann diese nicht durch Broadcasting ersetzt werden. Zudem verwendet das Programm die Funktion `where` statt Python's `if-else`-Primitiv, um Broadcasting zu ermöglichen.

Listing 3.2 Implementierung der naiven Variante der Auswertung einer Matrixzeile, entspricht Algorithmus 3.3 und 3.4

```python
from torch import where, tensor as array
from jax.numpy import where, array

def eval_b_spline_naive(x, t, k):
    """
    Parameters:
        x: (m,) - Auswertestellen
        t: (k+2, n, m) oder (n+k+1, m) - Knoten
        k: int - Splinegrad
    Returns:
        array - B-Splinewerte
            Das Array hat die Größe (1, n, m) falls t.shape == (k+2, n, m)
            oder (n, m) falls t.shape == (n+k+1, m)
    """
    b = where((t[:-1] <= x) & (x < t[1:]), 1., 0.) # shape == (k+1, n, m)
    x_minus_t = x - t  # shape == (k+2, n, m)

    for l in range(1, k+1):
        denominator = t[l:] - t[:-l]
        is_knot_interval_not_empty = denominator != 0.

        left_weights = where(
            is_knot_interval_not_empty[:-1],
            x_minus_t[:-l-1] / denominator,
            array(0.)
        )
        right_weights = where(
            is_knot_interval_not_empty[1:],
            1 - (x_minus_t[1:-l]) / denominator,
            array(0.)
        )
        # left_weights.shape == right_weights.shape == (k-l+1, n, m)

        b = left_weights * b[:-1] + right_weights * b[1:]
    return b
```

Mithilfe des Programms `eval_b_spline_naive` in Listing 3.2 kann bereits die Interpolationsmatrix aufgestellt werden. Dies wird Listing 3.3 in demonstriert. Die Funktion `interpolation_matrix_naive` entspricht konzeptionell Algorithmus 3.4 mit dem Unterschied, dass es mehrere Zeilen der Interpolationsmatrix berechnet. Hierfür fügt `t[:, None]` eine Dimension ein, d. h. `t[:, None].shape == (n+k+1, 1)`, die in `eval_b_spline_naive` auf die Größe `(n+k+1, m)` gebroadcastet wird.

Listing 3.3 Berechnung der Interpolationsmatrix

```python
def interpolation_matrix_naive(x, t, k):
    """
    Parameters:
        x: (m,) - Auswertestellen
        t: (m+n+1,) - Knoten
        k: int - Splinegrad
    Returns:
        array: (m, n) - Interpolationsmatrix
    """
    return eval_b_spline_naive(x, t[:, None], k).transpose(1, 0)
```

Listing 3.4 Implementierung der optimierten Variante der Auswertung einer Matrixzeile, entspricht Algorithmus 3.5

```python
from torch import cat as concatenate, ones_like, zeros_like, where, tensor as array
from jax.numpy import concatenate, ones_like, zeros_like, where, array

def eval_b_spline_optimized(x, t, k):
    """
    Parameters:
        x: (m,) - Auswertestellen
        t: (2*(k+1), m) - Knoten
            Vorbedingung: t[k] <= x < t[k+1] für alle x
        k: int - Splinegrad
    Returns:
        b: (k+1, m)
    """
    b = ones_like(x)[None, :]  # shape == (1, m)
    x_minus_t = x - t  # shape == (2*(k+1), m)

    for l in range(1, k+1):
        b = concatenate((zeros_like(x)[None, :], b, zeros_like(x)[None, :]))
        # b.shape == (l+2, m)

        denominator = t[k:k+l+2] - t[k-l:k+2]  # shape == (l+1, m)
        is_knot_interval_not_empty = denominator != 0.

        left_weights = where(
            is_knot_interval_not_empty[:-1],
            x_minus_t[k-l:k+1] / denominator[:-1],
            array(0.)
        )
        right_weights = where(
            is_knot_interval_not_empty[1:],
            1 - x_minus_t[k-l+1:k+2] / denominator[1:],
            array(0.)
        )
        # left_weights.shape == right_weights.shape == (l+1, m)

        b = left_weights * b[:-1] + right_weights * b[1:]
    return b
```

Wie in Algorithmus 3.5 gezeigt, kann die Auswertung optimiert werden, wenn die Auswertestellen zuvor in die Knotenintervalle einsortiert wurden. Das Listing 3.4 zeigt die Implementierung dieser optimierten Variante. Die Verwendung dieses Programms um die gesamte Matrix aufzustellen wird erst in Abschnitt 3.4 gezeigt, nachdem in Abschnitt 3.3 das Einsortieren der Auswertestellen in die Knotenintervalle beschrieben wurde.

Laufzeit

Abbildung 3.4 zeigt die Laufzeiten der PyTorch-Implementierungen für `eval_b_spline_naive` und `eval_b_spline_optimized` für verschiedene Splinegrade und Eingabegrößen. Die dargestellte Laufzeit ist jeweils die mittlere Laufzeit bei 300 Wiederholungen. Die Eingabegröße `t.shape`$=(k+2, k+1, m)$ entspricht konzeptionell Algorithmus 3.3, in welchem über alle nicht-verschwindenden Basisfunktionen iteriert und alle Basisfunktionen getrennt voneinander ausgewertet werden. Bei der Eingabegröße `t.shape`$=(2k+2, m)$ wird hingegen die Optimierung aus Algorithmus 3.4 angewandt, in welchem jedes Zwischenergebnis nur einmal berechnet wird. In jedem Fall werden genau $k+1$ Splines ausgewertet. Tabelle 3.3 zeigt die Laufzeit pro Auswertestelle gemittelt über die letzten drei Datenpunkte aus Abbildung 3.4.

Für Eingabegrößen $m \leq 10^5$ dominiert der Overhead, der durch den Python-Interpreter entsteht, die Laufzeit. Für größere Eingaben wächst der Aufwand wie erwartet linear in der Anzahl der Auswertestellen m. Der Unterschied zwischen den Implementierungen wird für größere Splinegrade deutlich. Während die drei Varianten für $k = 1$ nahezu die gleiche Laufzeit haben, ist die optimierte Variante für $k = 4$ dreimal schneller als die naive Implementierung.

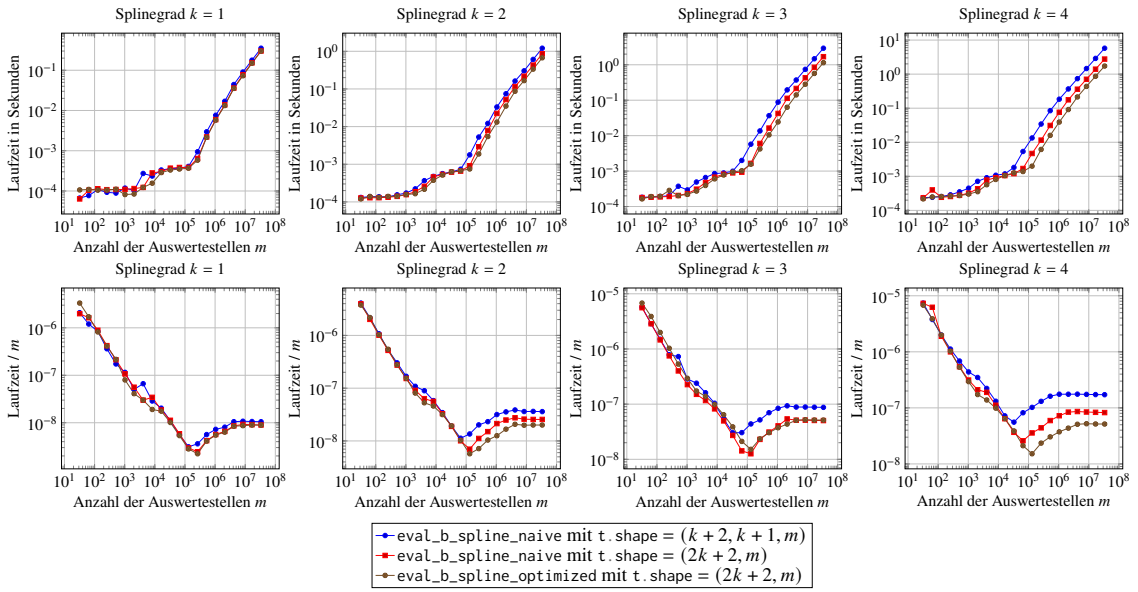

Abbildung 3.4: Laufzeiten der PyTorch-Implementierungen für `eval_b_spline_naive` und `eval_b_spline_optimized` für verschiedene Splinegrade k und Auswertestellenanzahlen m. Die Legende gibt die Eingabegröße des Knoten-Arrays `t` an. Die untere Zeile zeigt das Verhältnis der Laufzeit zu der Anzahl der Auswertestellen.

Funktion `t.shape`	`eval_b_spline_naive` $(k+2, k+1, m)$	`eval_b_spline_naive` $(2k+2, m)$	`eval_b_spline_optimized` $(2k+2, m)$
$k = 1$	$1.06 \cdot 10^{-8}$	$9.09 \cdot 10^{-9}$	$8.84 \cdot 10^{-9}$
$k = 2$	$3.61 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$2 \cdot 10^{-8}$
$k = 3$	$8.8 \cdot 10^{-8}$	$5.08 \cdot 10^{-8}$	$3.4 \cdot 10^{-8}$
$k = 4$	$1.73 \cdot 10^{-7}$	$8.34 \cdot 10^{-8}$	$5.17 \cdot 10^{-8}$

Tabelle 3.3: Mittelwert der Laufzeit pro Auswertestelle in Sekunden für $m \in \{2^{23}, 2^{24}, 2^{25}\}$

3.3 Einordnen der Auswertestellen in die Knoten

Um nur die nicht-verschwindenden B-Splines auszuwerten, müssen zunächst die Knoten im Träger dieser B-Splines bestimmt werden. Dieser Abschnitt beschreibt mögliche Algorithmen zum Einsortieren der B-Splines in die Knotenintervalle.

Gegeben ist also eine Folge von Auswertestellen $(x_0, \ldots, x_{m-1})$ und ein Knotenvektor $(t_0, \ldots, t_{n-1})$. Gesucht ist nun für jede Auswertestelle x_i der Knotenindex j_i, sodass das angrenzende Knotenintervall $[t_{j_i}, t_{j_i+1})$ die Auswertestelle enthält, d. h. $t_{j_i} \leq x_i < t_{j_i+1}$ gilt.

Für diese Aufgabe bieten JAX und PyTorch vorgefertigte Funktionen `jax.numpy.searchsorted` und `torch.bucketize`. Die Funktion `torch.bucketize` ist dabei eine Implementierung für `searchsorted`, die auf eindimensionale Tensoren spezialisiert ist. Der Aufruf dieser Funktionen wird in Listing 3.5 gezeigt. JAX implementiert `searchsorted` mittels binärer Suche für jede der Auswertestellen. Der asymptotische Aufwand liegt daher in $O(m \log n)$ für m Auswertestellen und n Knoten.

Listing 3.5 Verwendung der built-in `searchsorted`-Implementierungen

```python
# Mit PyTorch
from torch import bucketize
def searchsorted_builtin(t, x):
    """
    Parameters:
        x: (m,) - Auswertestellen
        t: (n+k+1,) - Knoten, sortiert
    Returns:
        i: (m,) - Indizes, dtype=int
            t[i-1] <= x < t[i]
    """
    return bucketize(x, t, right=True)


# Mit JAX:
from jax.numpy import searchsorted
def searchsorted_builtin(t, x):
    return searchsorted(t, x, side='right')
```

Im Rahmen dieser Arbeit wurden zudem zwei manuelle Implementierungen für die Berechnung von `searchsorted` umgesetzt.

Die erste Variante verwendet Broadcasting sowohl über die Knoten als auch über die Auswertestellen. Hierbei wird zunächst eine Bitmaske $(b_{j,i})_{\substack{0 \le j < n-1 \\ 0 \le i < m}}$ mit $b_{j,i} = \texttt{True}$ genau dann, wenn $x_i \in [t_j, t_{j+1})$ gilt, berechnet. Mithilfe dieser Bitmaske werden anschließend die gesuchten Indizes ausgewählt. Listing 3.6 zeigt die Implementierung mit PyTorch und JAX. Da die Bitmaske genauso groß wie die

Listing 3.6 Eigene Implementierung für `searchsorted` umgesetzt mit Broadcasting

```
from torch import where, arange, broadcast_to
from jax import where, arange, broadcast_to

def searchsorted_broadcasting(t, x):
    """Gleiche Signatur wie searchsorted_builtin"""
    bit_mask = where((t[:-1, None] <= x) & (x < t[1:, None]), True, False)
    # bit_mask.shape == (n+k, m)

    num_knots = t.shape[0]
    indices = arange(1, num_knots)[:, None]
    indices = broadcast_to(indices, bit_mask.shape)
    indices = indices[bit_mask] # shape == (m,)

    return indices
```

Interpolationsmatrix ist, liegt der Aufwand der Berechnung in $O(mn)$. Damit geht die Verbesserung der Komplexität, die mittels dem Einsortieren erreicht werden soll, verloren.

Unter der Annahme, dass die Auswertestellen sortiert sind, lässt sich die asymptotische Komplexität auf $O(m+n)$ verbessern. Diese Verbesserung ist Ziel der zweiten Implementierungsvariante, die in Listing 3.7 dargestellt ist. Dabei wird mit einer Python-Schleife über die Auswertestellen und Knoten iteriert und dabei die Auswertestellen einsortiert.

Listing 3.7 Eigene Implementierung für `searchsorted` umgesetzt mit einer Python-Schleife

```
# Dieses Programm kann nur mit PyTorch verwendet werden.
from torch import zeros_like, int32

def searchsorted_python_loop(t, x):
    """Gleiche Signatur wie searchsorted_builtin"""
    knot_index_per_sample = zeros_like(x, dtype=int32)

    iter_knots = enumerate(t)
    j, knot = next(iter_knots)
    for i, sample in enumerate(x):
        # Finde nächstgrößeren Knoten
        while knot <= sample:
            try:
                j, knot = next(iter_knots)
            except StopIteration:
                break

        knot_index_per_sample[i] = j

    return knot_index_per_sample
```

Die Umsetzung mittels einer Python-Schleife hat allerdings Nachteile, die die Laufzeit deutlich beeinträchtigen. Zum einen erzeugt Python's Interpreter bei der Ausführung der Schleife einen relativ großen Overhead. Zum anderen ist die innere `while`-Schleife in JAX nicht JIT-kompilierbar, da die Anzahl der Iterationen von den Werten der Knoten und Auswertestellen abhängt.

Laufzeit

Abbildung 3.5 zeigt die Laufzeiten der vorgestellten Implementierungen `searchsorted` in PyTorch sowie der Funktion von JAX. Die Ergebnisse zeigen, dass die Funktionen der Frameworks deutlich effizienter sind. Für Eingabegrößen mit wenigen Knoten erkennt man zudem, dass die Implementierung mit Broadcasting schneller ist als die Implementierung mit einer Schleife. Der Vorteil der Implementierung mit einer Schleife zeigt sich erst für eine Anzahl der Knoten $n \geq 2048$. Die Laufzeit on $O(m+n)$ der Implementierung mit einer Schleife zeigt sich besonders im mittleren Graph, der die Laufzeiten für $m = 4096$ und variable Anzahl n der Knoten darstellt. Für größere Knotenvektoren mit $m \leq n$ dominiert die Schleife über die Knoten die Laufzeit, was zu einer linearen Laufzeit in n führt. Praktisch relevanter ist allerdings der Fall, in welchem weniger Knoten als Auswertestellen verwendet werden, d. h. $n \leq m$. In diesem Fall ist die Schleife linear in der Anzahl m der Auswertestellen. Das Laufzeitverhalten der Framework-Funktionen in $O(m \log n)$ ist kaum von linearem Wachstum in der Anzahl der Auswertestellen zu unterscheiden. Der rechte Graph zeigt die Laufzeit, wenn die Anzahl der Knoten und Auswertestellen gleichschnell wächst. Er verdeutlicht erneut den Unterschied der Komplexität der Implementierungen. Während der Aufwand der Implementierung mit Broadcasting quadratisch in $m = n$ wächst, zeigen die anderen Implementierungen lineares Wachstum.

Insgesamt demonstriert dieser Vergleich, dass die Verwendung von existierenden Funktionen der Frameworks deutlich schneller ist als manuelle Implementierungen. Zudem beschleunigt Broadcasting Programme so sehr, dass der asymptotische Vorteil einer Python-Schleife erst für große Eingaben (mehr als 2000 Knoten) sichtbar wird.

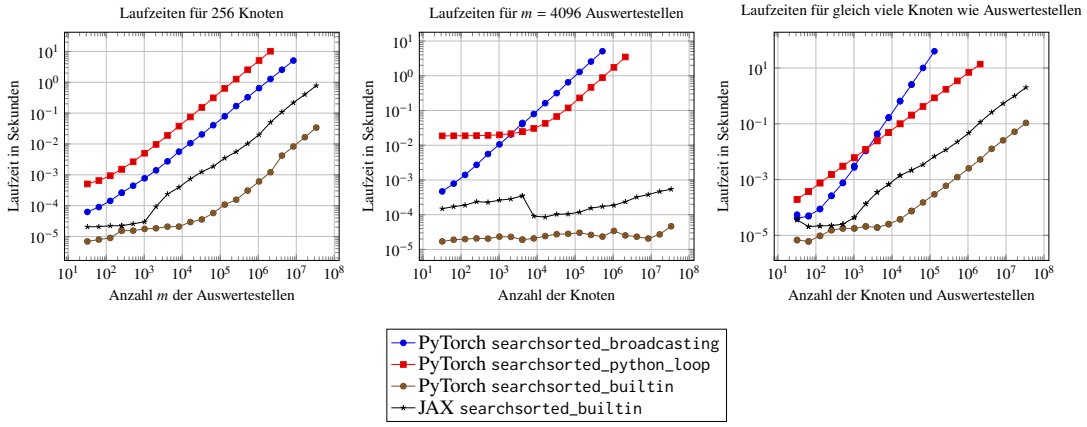

Abbildung 3.5: Laufzeiten der Implementierungen für `searchsorted` auf einem der PCSGS-Rechner. Links: Feste Anzahl der Knoten, variable Anzahl der Auswertestellen. Mitte: feste Anzahl der Auswertestellen, variable Anzahl der Knoten. Rechts: Anzahl der Knoten gleich Anzahl der Auswertestellen

Konstruktion der Knoten im Träger

Aus den mittels `searchsorted` berechneten Indizes j mit $x_i \in [t_{j-1}, t_j)$ müssen die Knoten im Träger nicht-verschwindender B-Splines konstruiert werden, welche als Parameter an `eval_b_spline_optimized` übergeben werden. Für $x_i \in [t_j, t_{j+1})$ sind die nicht verschwindenden B-Splines $b^k_{j-k-1,\mathbf{t}}, \ldots, b^k_{j-1,\mathbf{t}}$. Die zugehörigen Knoten in deren Träger sind $(t_{j-k-1}, \ldots, t_{j+k})$. Die Implementierung dieses Schritts besteht im Wesentlichen aus Index-Shifts und ist in Listing 3.8 dargestellt. Zusätzlich wird ein Array aus den Indizes der B-Splines konstruiert. Diese Indizes stimmen mit den Spaltenindizes der Interpolationsmatrix überein, in welche die berechneten Splinewerte eingetragen werden.

Listing 3.8 Konstruktion der Knoten im Träger aus dem Index des Knotenintervalls

```
from torch import arange
from jax.numpy import arange

def construct_knots_and_indices(indices, t, k):
    """
    Parameters:
        indices: (m,) - Indizes mit t[indices-1] <= x < t[indices] für alle x
        t: (n+k+1,) - Knoten
        k: int - Splinegrad
    Returns:
        support_knots: (2*(k+1), m) - Knoten im Träger nicht verschwindener B-Splines pro Auswertestelle
            Es gilt: support_knots[j, i] = t[indices[i] - k-1 + j]
        column_indices: (k+1, m) - Spaltenindizes der Interpolationsmatrix
            Es gilt: column_indices[j, i] = indices[i] - k-1 + j
    """
    shifter = arange(-k-1, k+1)
    shifted_knot_indices = indices + shifter[:, None]

    # Wähle Knoten im Träger
    support_knots = t[shifted_knot_indices] # shape == (2*(k+1), m)

    # Spaltenindizes der Einträge in der Interpolationsmatrix
    column_indices = shifted_knot_indices[:k+1] # shape == (k+1, m)

    return support_knots, column_indices
```

3.4 Aufstellen der Matrix

Mit den berechneten Splinewerten aus Listing 3.4 und Spaltenindizes aus Listing 3.8 kann nun die Interpolationsmatrix aufgestellt werden. Die Allokation des benötigten Speichers sowie des Einsetzen der Splinewerte wird in Listing 3.9 gezeigt. Hierfür werden zunächst die leere Matrix mit Nullen initialisiert und danach die Splinewerte eingetragen. Da Arrays in JAX immutable sind, wird in JAX ein neues Array erstellt, während in PyTorch das bestehende Array in-place modifiziert wird. Die Speicherallokation hat eine Komplexität in $O(mn)$ und ist damit die aufwändigste Operation bei der Berechnung der Interpolationsmatrix.

Listing 3.9 Einsetzen der Werte in die Interpolationsmatrix

```
from torch import arange, zeros
from jax.numpy import arange, zeros

def scatter_into_matrix(values, column_indices, m, n):
    row_indices = arange(0, m)[None, :]
    interpolation_matrix = zeros((m, n))

    # Mit PyTorch:
    interpolation_matrix[row_indices, column_indices] = values
    # Mit JAX:
    interpolation_matrix = interpolation_matrix.at[row_indices, column_indices].set(values)

    return interpolation_matrix
```

Listing 3.10 setzt die gezeigten Algorithmen zusammen um die Interpolationsmatrix aufzustellen.

Listing 3.10 Aufstellen der Matrix

```
def interpolation_matrix_optimized(x, t, k):
    indices = searchsorted(t, x)
    support_knots, column_indices = construct_knots_and_indices(indices, t, k)
    values = eval_b_spline_optimized(x, support_knots, k)
    interpolation_matrix = scatter_into_matrix(values,
                                               column_indices,
                                               m=x.shape[0],
                                               n=t.shape[0]-k-1)
    return interpolation_matrix
```

Laufzeit

Abbildung 3.6 zeigt die Laufzeit der Implementierungen `interpolation_matrix_naive` (Listing 3.3) und `interpolation_matrix_optimized` (Listing 3.10) jeweils mit JAX und PyTorch. Die Programme in JAX wurden vor der Messung JIT-kompiliert. Da die Auswertung der B-Splines in Listing 3.2 bzw. Listing 3.4 eine Schleife über den Splinegrad enthält, kann die Funktion nicht mit dem Parameter `k` als Variable kompiliert werden. Stattdessen kann der Splinegrad `k` während der Kompilierung als konstant betrachtet werden. Die JIT-kompilierte Funktion von `interpolation_matrix_naive` ist `jitted_fun = jax.jit(interpolation_matrix_naive, static_argnames='k')`. Der Befehl, dessen Laufzeit in Abbildung 3.6 gezeigt wird, ist damit `jitted_fun(x, t, k=k).block_until_ready()`.

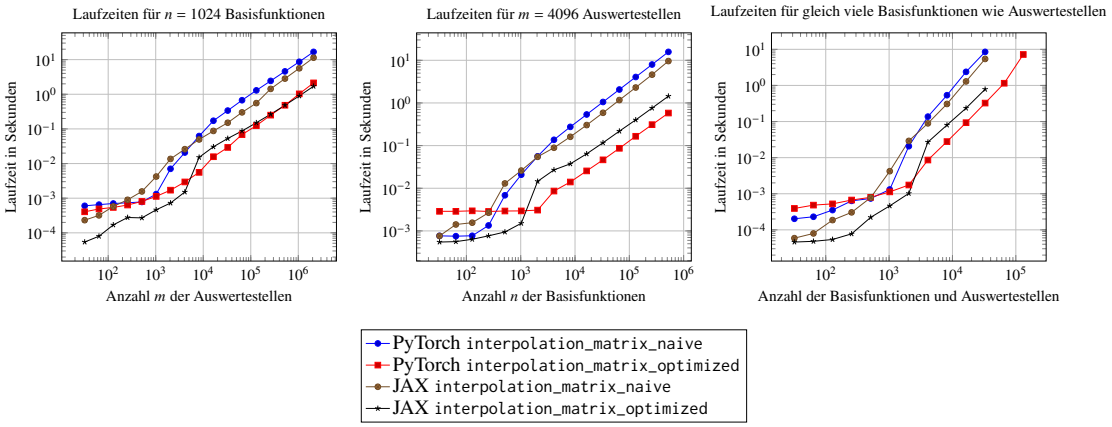

Abbildung 3.6: Laufzeiten der Implementierungen für das Aufstellen der Interpolationsmatrix auf einem der PCSGS-Rechner. Links: Feste Anzahl der Basisfunktionen, variable Anzahl der Auswertestellen. Mitte: feste Anzahl der Auswertestellen, variable Anzahl der Basisfunktionen. Rechts: Anzahl der Basisfunktionen gleich Anzahl der Auswertestellen

3.5 Berechnen des Approximationsfehlers

Aus der Interpolationsmatrix können mithilfe der Methode der kleinsten Quadrate die Koeffizienten des Bestapproximationssplines und damit der Approximationsfehler bestimmt werden. Hierfür bieten sowohl PyTorch als auch JAX die Funktion `lstsq`. Die Berechnung des Approximationsfehlers $E(s_{\mathbf{t}}^k)$ aus Definition 2.1.3 ist in Listing 3.11 dargestellt. Leider ist die Funktion `lstsq` mit PyTorch nicht differenzierbar.[6] Um automatisches Differenzieren zu ermöglichen, ist daher eine manuelle Implementierung notwendig.

Listing 3.11 Berechnung des Approximationsfehler eines Splines mit optimalen Kontrollpunkten

```
1  from torch.linalg import lstsq
2  from jax.numpy.linalg import lstsq
3
4  def approximation_error_of_lsqspline(t, x, f, k):
5      """
6      Parameters:
7          t: (n,) - Knoten, sortiert
8          x: (m,) - Auswertestellen
9          f: (m,) - Anzunähernde Funktionswerte in den Auswertestellen
10         k: int - Splinegrad
11     Returns:
12         float - Approximationsfehler
13     """
14     A = interpolation_matrix(x, t, k)
15     c, residuals, _, _ = lstsq(A, f[:, None])
16     # residuals[0] == sum((f - A @ c)**2)
17     return residuals[0] / x.shape[0]
```

[6] GitHub Issue zum automatischen Differenzeiren von `torch.linalg.lstsq`, https://github.com/pytorch/pytorch/issues/27036#issue-499938066

3.6 Differenzieren des Fehlers

Automatisches Differenzieren ermöglicht es die implementierten Funktionen abzuleiten. Während die von NumPy bekannten Vektoroperationen, in beiden Frameworks eine ähnliche API bieten, zeigen sich beim Differenzieren konzeptionelle Unterschiede.

Die Berechnung des Gradienten ist in PyTorch etwas technischer und orientiert sich am Ablauf des Reverse Modes. Das Vorgehen hierfür wird in Listing 3.12 demonstriert. Zunächst wird die Eingabe, nach welcher abgeleitet werden soll, mit dem Flag `requires_grad` markiert. Danach wird in einem Forward-Pass der Fehler berechnet, um danach im Backward-Pass den Gradienten zu berechnen. Anschließend kann der Gradient aus dem Attribut `grad` der Eingabe ausgelesen werden.

Listing 3.12 Berechnung des Gradienten mit PyTorch

```
# Mit PyTorch

t.requires_grad_(True)
error = approximation_error_of_lsqspline(t, x, f, k)
error.backward()
gradient = t.grad
```

Im Gegensatz dazu ist in JAX, passend zum funktionalen Programmierparadigma, auch die Ableitung eine Funktion. Hierfür bietet JAX die Funktion `grad`, welche auf Eingabe einer Funktion, eine Funktion zurückgibt, die den Gradienten berechnet. Der Parameter, nach welchem abgeleitet werden soll, wird mit dem Argument `argnums` an `jax.grad` festgelegt. Alternativ konstruiert `value_and_grad` eine Funktion, die sowohl den Funktionswert als auch den Gradienten berechnet. Die Verwendung von `value_and_grad` wird in Listing 3.13 gezeigt. Zusätzlich bietet JAX die Funktionen `jax.jacfwd`, `jax.jacrev` und `jax.hessian`, um die Jacobi-Matrix im Forward oder Reverse Mode und die Hesse-Matrix zu bestimmen.

Listing 3.13 Berechnung des Gradienten mit JAX

```
# Mit JAX
from jax import value_and_grad

error_and_grad_fun = value_and_grad(approximation_error_of_lsqspline,
                                    argnums=0)
error, gradient = error_and_grad_fun(t, x, f, k)
```

Laufzeit

Abbildung 3.7 zeigt die Laufzeit der Berechnung des Approximationsfehlers mit Listing 3.11 sowie die Laufzeit zur Auswertung des Gradienten mit dem Code aus Listing 3.12 für PyTorch bzw. Listing 3.13 für JAX. In JAX ist die Auswertung des Gradienten nahezu genauso schnell wie die Auswertung der Fehlerfunktion. Da die Framework-Funktion `torch.linalg.lstsq` nicht automatisch differenziert werden kann, wurde für die Laufzeitmessung in PyTorch eine eigene Implementierung für `lstsq` mittels der QR-Zerlegung verwendet. Für diese Implementierung ist

das automatische Differenzieren etwas langsamer als die Auswertung des Approximationsfehlers. Sowohl mit JAX als auch mit PyTorch liegt der Aufwand von `lstsq` in $O(mn^2)$ für m Auswertestellen und n Basisfunktionen. Dennoch sind sowohl die Berechnung des Fehlers als auch dessen Ableitung in PyTorch schneller als in JAX.

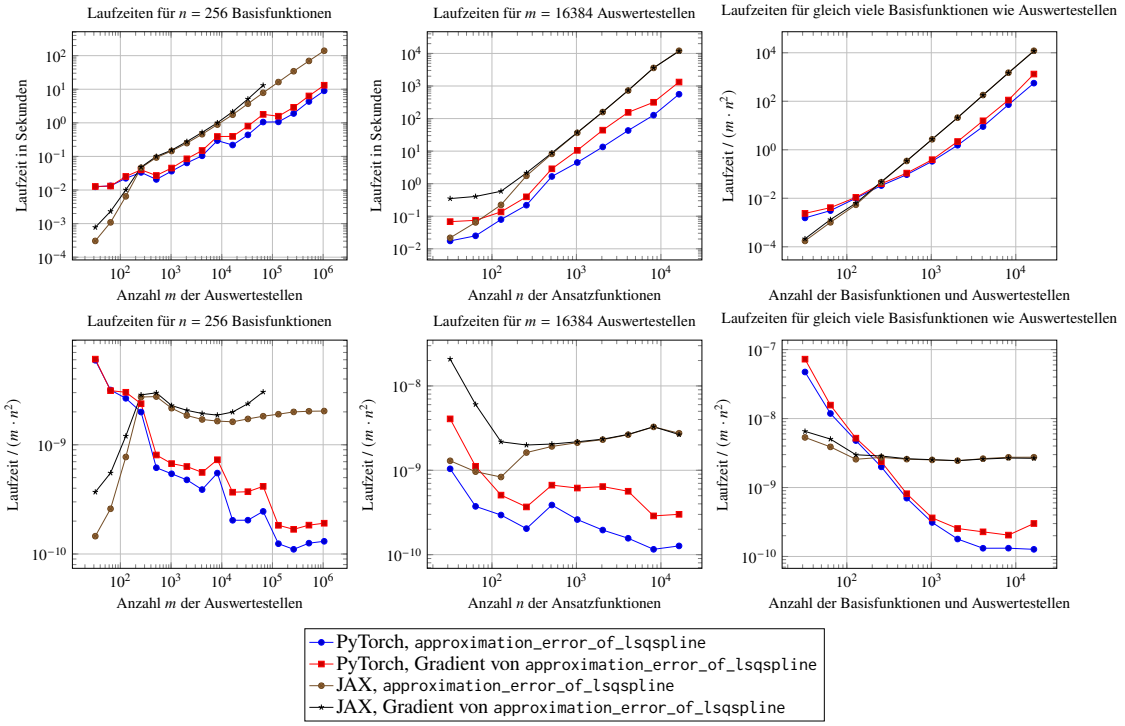

Abbildung 3.7: Laufzeiten der Berechnung des Approximationsfehlers und des Gradienten auf einem der PCSGS-Rechner. Die untere Zeile zeigt das Verhältnis der Laufzeit zu der Anzahl der Auswertestellen und dem Quadrat der Anzahl der Basisfunktionen. Links: Feste Anzahl der Basisfunktionen, variable Anzahl der Auswertestellen. Mitte: feste Anzahl der Auswertestellen, variable Anzahl der Basisfunktionen. Rechts: Anzahl der Basisfunktionen gleich Anzahl der Auswertestellen

3.7 Optimierer

Der Gradient kann zur Optimierung verwendet werden. Hierfür bietet PyTorch im Submodul `torch.optim`[7] und das Modul `optax` [HBV+20] für JAX verschiedene Optimieralgorithmen an. Alle Algorithmen haben gemein, dass sie eine Fehlerfunktion iterativ mithilfe des Gradienten minimieren. PyTorch's Minimierer sind Objekte, die die Parameter in-place modifizieren. Listing 3.14 zeigt die Optimierung des Knotenvektors `t` am Beispiel des Optimierers Stochastic Gradient Descent (SGD) von PyTorch. In den Zeilen 8–10 wird die Fehlerfunktion ausgewertet und der Gradient bezüglich der Knoten `t` bestimmt. Zeile 11 führt einen Optimierschritt aus.

[7] Dokumentation von `torch.optim`, https://pytorch.org/docs/1.9.1/optim.html

Listing 3.14 Optimierung der Knoten mit PyTorch

```
1  import torch
2
3  def optimize_knots(t, x, f, k, iterations, optim_kwargs):
4      t.requires_grad_(True)
5      optimizer = torch.optim.SGD([t], **optim_kwargs)
6
7      for i in range(iterations):
8          optimizer.zero_grad()
9          e = approximation_error_of_lsqspline(t, x, f, k)
10         e.backward()
11         optimizer.step()
12
13     return t
```

Für JAX bietet `optax` einige Optimieralgorithmen und andere Gradiententransformationen. Listing 3.15 zeigt das Schema der Verwendung am Beispiel des Optimierungsalgorithmus SGD. Ein Optimierer von `optax` besteht aus den zwei Funktionen `init` und `update`. Der Zustand `opt_state` des Optimierers wird getrennt in Zeile 5 initialisiert. In den Zeilen 10–12 wird ein Optimierschritt ausgeführt. Hierfür wird zunächst der Gradient ausgewertet. Anhand des Gradienten und des Zustands `opt_state` berechnet die Funktion `update` das Update für die Parameter und passt den Zustand an. Die Funktion `optax.update` wendet die berechneten Updates an.

Listing 3.15 Optimierung der Knoten mit JAX und `optax`

```
1  import jax, optax
2
3  def optimize_knots(t, x, f, k, iterations, optim_kwargs):
4      opt = optax.sgd(**optim_kwargs)
5      opt_state = opt.init(t)
6
7      grad_fun = jax.grad(approximation_error_of_lsqspline)
8
9      for i in range(iterations):
10         t_grad = grad_fun(t, x, f, k)
11         updates, opt_state = opt.update(t_grad, opt_state)
12         t = optax.apply_updates(t, updates)
13
14     return t
```

Die vorgestellten Programme zur Optimierung haben einige Einschränkungen, die bei der Verwendung beachtet werden müssen. Beispielsweise kann die Sortierung der Knoten durch die Optimierung verletzt werden. Um dies zu verhindern, kann die Fehlerfunktion um die Sortierung der Knoten mittels `torch.sort` bzw. `jax.numpy.sort` ergänzt werden. Zudem ist der Gradient für mehrfache Knoten nicht definiert. Um diesem Problem entgegenzuwirken, kann entweder auf die Verwendung mehrfacher Knoten verzichtet oder Updates in den `NaN`-Einträgen verhindert werden. Beispielsweise führen Schwetlick und Schütze [SS95] einen Glättungsterm ein, der einen minimalen Abstand zwischen den Knoten erzwingt. In dieser Arbeit brechen die vorgestellten Optimierungen ab, falls mehrfache Knoten auftreten.

4 Ergebnisse

Dieses Kapitel zeigt Beispiele für die Optimierung der Knoten mithilfe des Gradienten. Abschnitt 4.1 untersucht den Approximationsfehler in Abhängigkeit der Knotenpositionen. Abschnitt 4.2 vergleicht verschiedene Optimierer.

4.1 Approximationsfehler und Gradientenfelder

Der folgende Abschnitt betrachtet die Abhängigkeit des Approximationsfehlers von der Knotenposition anhand einiger Beispielfunktionen. Zu Beginn werden stückweise Polynome betrachtet, welche durch Splines exakt dargestellt werden können. Anschließend werden die Exponentialfunktion sowie die Sinusfunktion betrachtet. Für die einfachere Darstellung beschränkt sich die Untersuchung größtenteils auf zwei innere Knoten.

In den nachfolgenden Betrachtungen wurden am Rand $(k+1)$-fache Knoten für den Splinegrad k verwendet. Zur Bestimmung des Approximationsfehlers wurden die Funktionen auf dem Intervall $[0, 1)$ äquidistant abgetastet. Die Auswertestellen sind $x_i = \frac{i}{1500}$ für $0 \leq i < 1500$.

Stückweise Polynome

Als Erstes betrachten wir das stückweise Polynom

$$f_{1,a} : [0,1) \to \mathbb{R}, \quad f_{1,a}(x) = \begin{cases} 0 & \text{falls } x \leq \frac{1}{2} \\ \left(x - \frac{1}{2}\right)^a & \text{falls } x > \frac{1}{2} \end{cases}$$

mit Polynomgrad $a \in \{1, 2, 3\}$ über zwei gleich langen Teilintervallen. Diese Funktion soll nun mithilfe von Splines approximiert werden. Abbildung 4.1 zeigt den Approximationsfehler der Bestapproximationssplines in Abhängigkeit der zwei inneren Knoten für verschiedene Splinegrade k. Dabei haben die Knotenvektoren die Form $(0, \ldots, 0, t_1, t_2, 1, \ldots, 1)$ mit $(k+1)$-fachen äußeren Knoten und variablen, inneren Knoten $t_1 \leq t_2 \in [0, 1]$. Zusätzlich zeigt die Abbildung den Gradienten als Richtungsfeld.

Falls der Grad a von $f_{1,a}$ gleich dem Splinegrad k ist $(a = k)$, ist die Approximation exakt, falls einer der inneren Knoten auf der Intervallgrenze $\frac{1}{2}$ liegt. Falls der Splinegrad um eins größer ist, wird ein doppelter Knoten in $\frac{1}{2}$ für eine exakte Approximation benötigt. Für noch größere Splinegrade ist das Minimum nicht mehr eindeutig. Stattdessen entstehen aufgrund der Symmetrie in jedem der beiden Teilintervalle lokale Minima. Für kleinere Splinegrade $(k < a)$ ist das Minimum eindeutig mit $\frac{1}{2} < t_1 < t_2 < 1$. In diesem Fall werden die Knoten in den Bereich größerer Steigung von $f_{1,a}$ geschoben. In jedem Fall führt ein Gradientenabstieg beginnend in den äquidistanten Knoten zu einer optimalen Knotenposition.

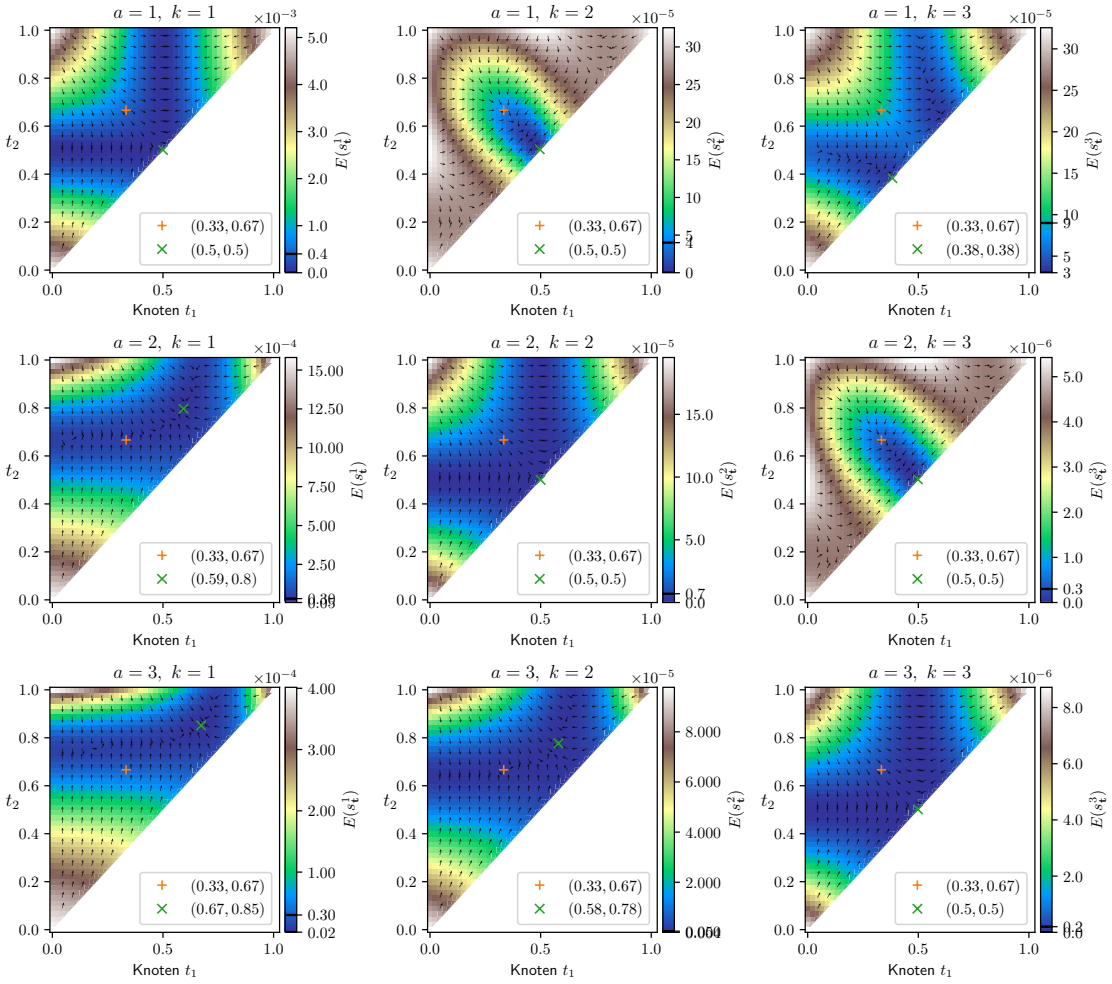

Abbildung 4.1: Approximationsfehler und dessen Gradient für die Referenzfunktion $f_{1,a}$

Abbildung 4.2 zeigt den Approximationsfehler für die Funktion

$$f_{2,a} : [0, 1) \to \mathbb{R}, \quad f_{2,a}(x) = \begin{cases} 0 & \text{falls } x \leq \frac{1}{3} \\ \left(x - \frac{1}{3}\right)^a & \text{falls } x > \frac{1}{3} \end{cases}$$

mit $a \in \{1, 2, 3\}$. Sie unterscheidet sich von der zuvor betrachteten Funktion $f_{1,a}$ nur in der Grenze der Teilintervalle. Für niedrige Splinegrade $k \leq a$ ähnelt das Verhalten des Fehlers jenem bei der Funktion $f_{1,a}$. Falls der Spline glätter als die Funktion $f_{2,a}$ ist, bilden sich allerdings lokale Minima. Während die Splineapproximation für $k = a + 1$ und einem doppelten Knoten in $\frac{1}{3}$ exakt ist, entsteht zusätzlich ein lokales Minimum, das beide Knoten im längeren Teilintervall positioniert.

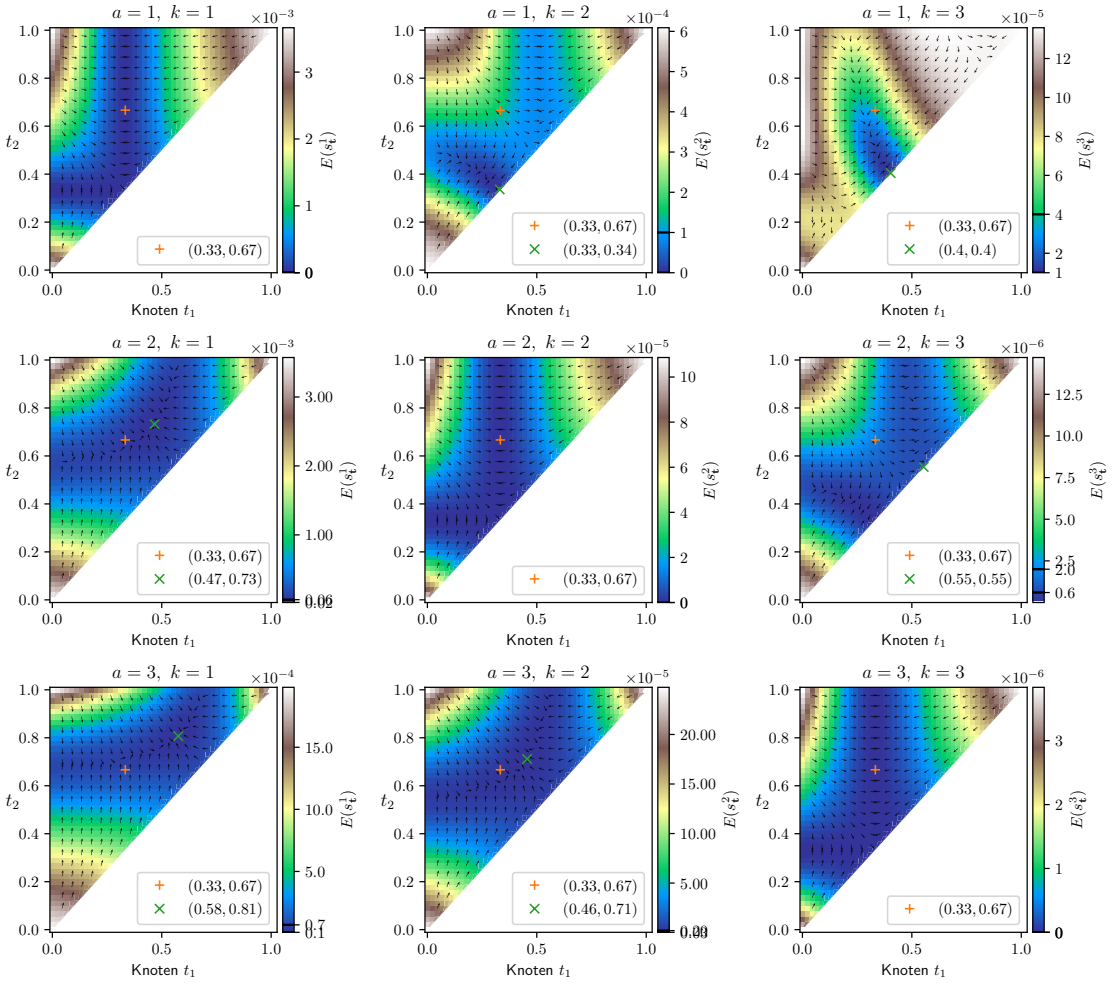

Abbildung 4.2: Approximationsfehler und dessen Gradient für die Referenzfunktion $f_{2,a}$

Die Beobachtung, dass für niedrigere Splinegrade die Knoten in den Bereich größerer Steigung geschoben werden, während für glättere Splines die Knoten in das größere Intervall geschoben werden, bestätigt sich bei Betrachtung der Funktion

$$f_{3,a} : [0,1) \to \mathbb{R}, \quad f_{3,a}(x) = \begin{cases} \left|x - \frac{1}{3}\right|^a & \text{falls } x \leq \frac{1}{3} \\ 0 & \text{falls } x > \frac{1}{3} \end{cases}$$

für $a \in \{1,2,3\}$. Abbildung 4.3 zeigt den zugehörigen Approximationsfehler. Falls der Spline glatter ist als die Funktion $a \leq k$ ist der Fehler gleich wie in für die Funktion $f_{2,a}$ in Abbildung 4.2. Für kleinere Splinegrade ($k < a$) hingegen befinden sich die optimalen Knoten in dem kleineren, linken Teilintervall.

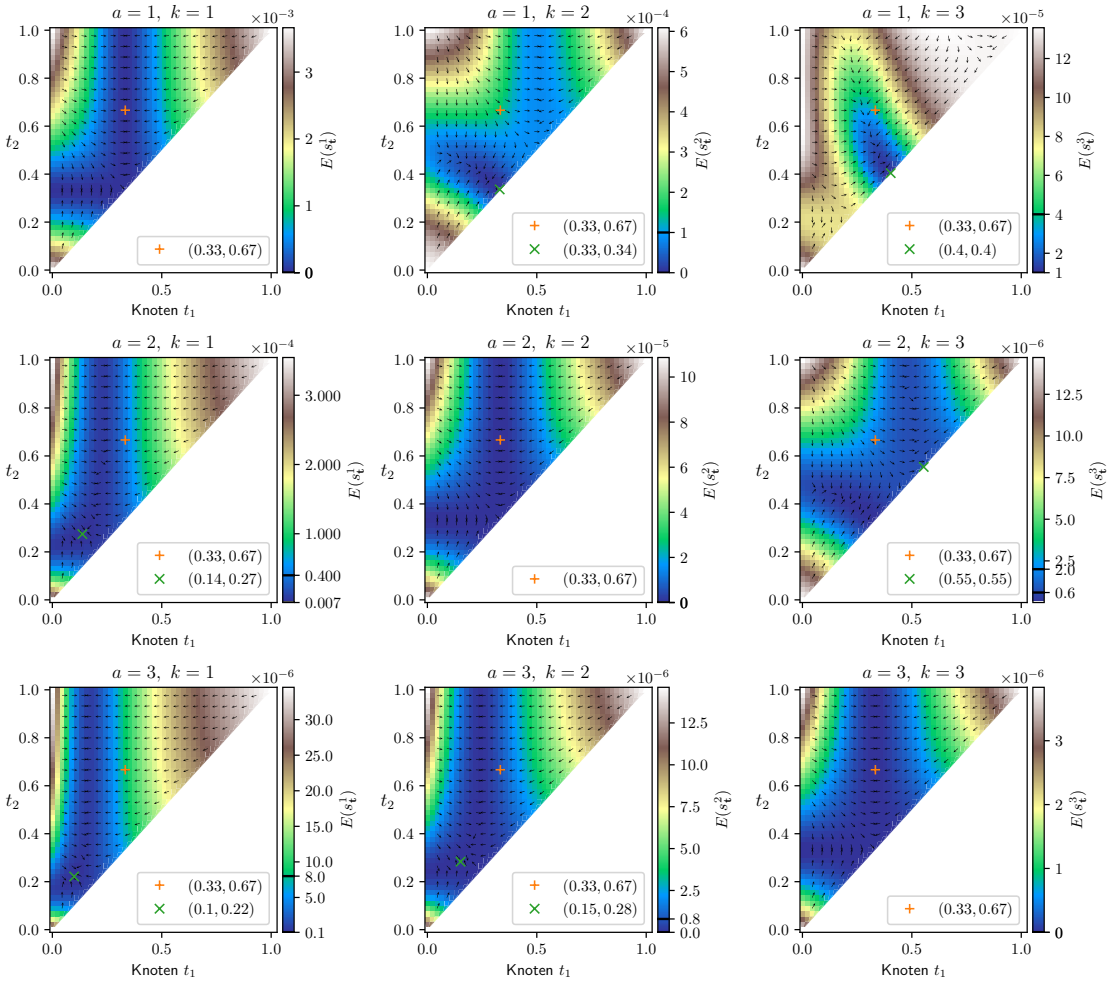

Abbildung 4.3: Approximationsfehler und dessen Gradient für die Referenzfunktion $f_{3,a}$

Die bisherigen Funktionen teilen den Definitionsbereich nur in zwei Teilintervalle. Als Beispiel für ein stückweises Polynom über drei Teilintervallen betrachten wir die Funktion

$$f_{4,a} : [0,1) \to \mathbb{R}, \quad f_{4,a}(x) = \begin{cases} 0 & \text{falls } x < \frac{1}{3} \\ \left(x - \frac{1}{3}\right)^a & \text{falls } \frac{1}{3} \le x < \frac{2}{3} \\ \left(x - \frac{1}{3}\right)^a + \left(x - \frac{2}{3}\right)^a & \text{falls } \frac{2}{3} \le x \end{cases}$$

für $a \in \{1, 2, 3\}$. Die Funktion $f_{4,a}$ ist so gewählt, dass sie a-mal stetig differenzierbar ist, und die stückweise konstante a-te Ableitung

$$f_{4,a}^{(a)}(x) = \begin{cases} 0 & \text{falls } x < \frac{1}{3} \\ a! & \text{falls } \frac{1}{3} \le x < \frac{2}{3} \\ 2a! & \text{falls } \frac{2}{3} \le x \end{cases}$$

hat. Abbildung 4.4 zeigt die zugehörige Fehlerverteilung. Falls der Polynomgrad a von $f_{4,a}$ mit dem Splinegrad k übereinstimmt, dann liegen die optimalen Knoten auf den Intervallgrenzen $\frac{1}{3}$ und $\frac{2}{3}$. Für kleinere Splinegrade zeigt sich wieder, dass der Fehler minimiert wird, indem die Knoten im Bereich der größeren Steigung positioniert werden. Für glättere Splines zeigen sich allerdings mehrere Minima. Für $a = 1$ und $k \in \{2, 3\}$ positionieren die lokalen Optima die Knoten näher an den Intervallenden. Zwei der lokalen Minima positionieren doppelte Knoten jeweils nahe der Intervallgrenzen $\frac{1}{3}$ oder $\frac{2}{3}$. Das dritte Minimum positioniert die Knoten an den unterschiedlichen Enden des Definitionsbereichs. Für $a = 2$ und $k = 3$ positioniert ein weiteres lokales Minimum doppelte Knoten in der Mitte des Intervalls.

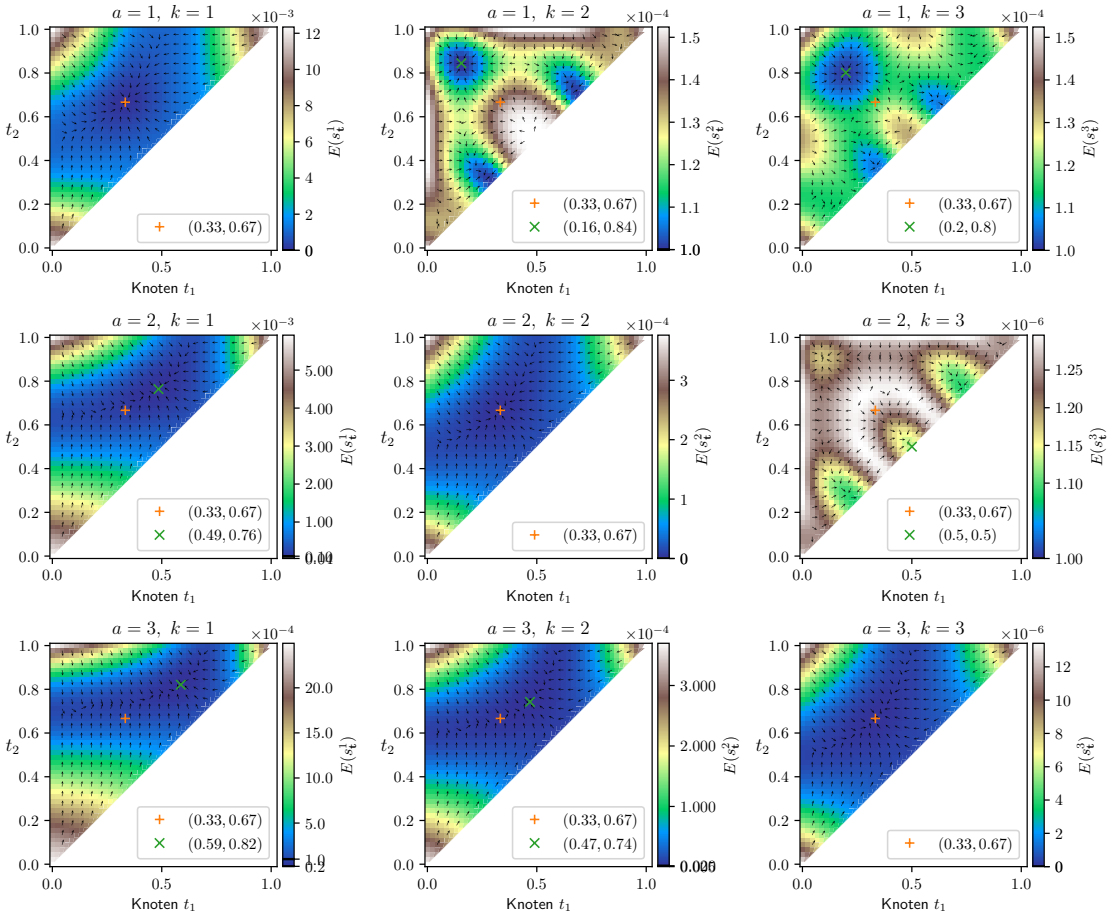

Abbildung 4.4: Approximationsfehler und dessen Gradient für die Referenzfunktion $f_{4,a}$

Abbildung 4.5 zeigt den Approximationsfehler der Funktion

$$f_{5,a} : [0, 1) \to \mathbb{R}, \quad f_{5,a}(x) = \begin{cases} 0 & \text{falls } x < \frac{1}{3} \\ \left(x - \frac{1}{3}\right)^a & \text{falls } \frac{1}{3} \leq x < \frac{2}{3} \\ \left(x - \frac{1}{3}\right)^a - \left(x - \frac{2}{3}\right)^a & \text{falls } \frac{2}{3} \leq x \end{cases}$$

für $a \in \{1, 2, 3\}$. Sie unterscheidet sich von $f_{4,a}$ nur im dritten Teilintervall, sodass die a-te Ableitung symmetrisch um die Mitte $\frac{1}{2}$ des Intervalls ist. Die a-te Ableitung der Funktion $f_{5,a}$ ist gegeben durch

$$f_{5,a}^{(a)}(x) = \begin{cases} 0 & \text{falls } x < \frac{1}{3} \\ a! & \text{falls } \frac{1}{3} \leq x < \frac{2}{3} \\ 0 & \text{falls } \frac{2}{3} \leq x. \end{cases}$$

Im Vergleich zu der Fehlerverteilung von $f_{4,a}$ für $k = a - 1$ fällt auf, dass die Knoten nicht nach rechts sondern in die Mitte des Intervalls geschoben werden. Dies lässt sich dadurch erklären, dass die Funktion $f_{5,a}$ im rechten Teilintervall ein stückweises Polynom vom Grad $a - 1$ ist, d. h. durch einen Spline vom Grad $k = a - 1$ exakt approximiert werden könnte.

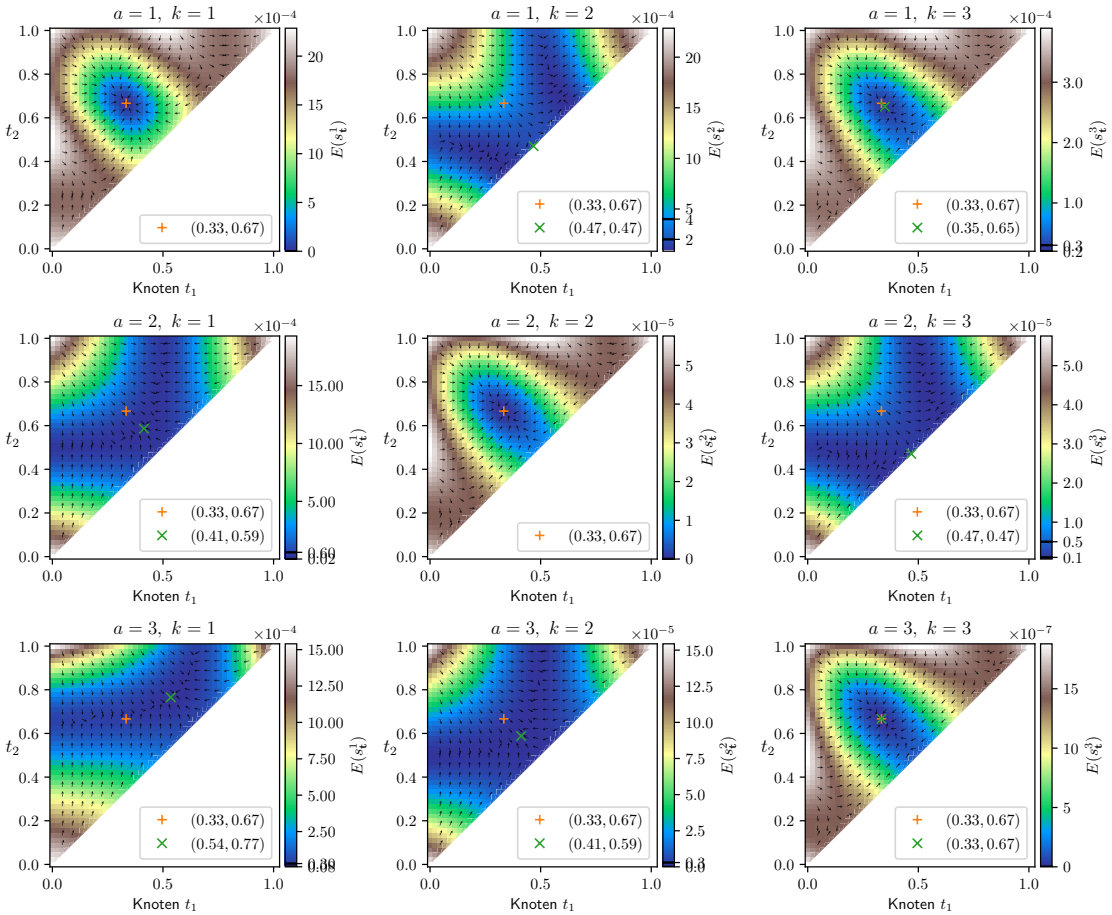

Abbildung 4.5: Approximationsfehler und dessen Gradient für die Referenzfunktion $f_{5,a}$

Exponentialfunktion

Als Beispiel für eine unendlich oft stetig differenzierbare Referenzfunktion betrachten wir die Exponentialfunktion

$$f_{6,a} : [0, 1) \to \mathbb{R}, \quad f_{6,a}(x) = \exp(ax)$$

für $a \in \{1, 10\}$. Abbildung 4.6 zeigt den Approximationsfehler der Splineapproximation für die Splinegrade $k \in \{1, 2, 3\}$. Sowohl für $a = 1$ als auch für $a = 10$ befinden sich die optimalen Knotenposition rechts von den äquidistanten Knoten. Auffällig ist, dass dieser Effekt für größere a und kleinere Splinegrade k deutlich stärker ist. Während für $a = 1$ die äquidistanten Knoten fast optimal sind, befinden sich für $a = 10$ die Knoten deutlich näher am rechten Intervallrand. Das eindeutige Minimum ermöglicht zudem die Optimierung mittels Gradientenabstieg.

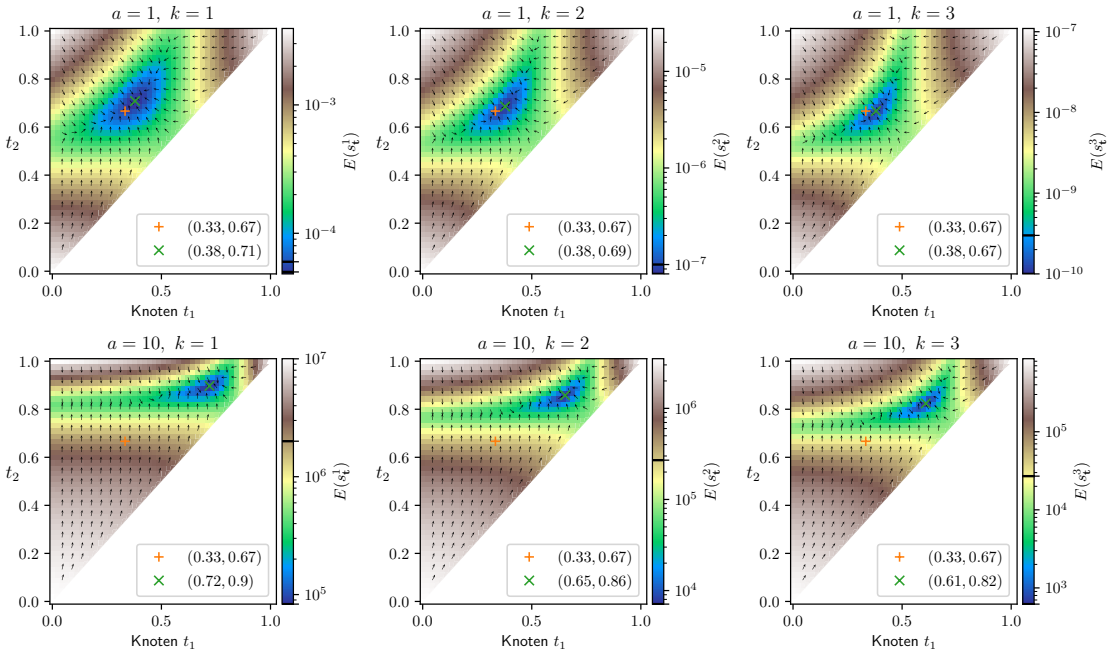

Abbildung 4.6: Approximationsfehler und dessen Gradient für die Exponentialfunktion $f_{6,a}$

Mit Hilfe des Gradienten lassen sich auch die Knotenpositionen von Splines mit mehr als zwei Knoten optimieren. Abbildung 4.7 zeigt den knotenoptimierten, quadratischen Spline mit 3 und 8 Knoten für $f_{6,a}$ mit $a = 10$. Wie zu erwarten befinden sich die optimalen Knoten weiter rechts als die äquidistanten Knoten. Zudem zeigt Abbildung 4.7 die Differenz $f_{6,a}(x) - s_{\mathbf{t}}^2(x)$ zwischen der Referenzfunktion und dem Spline. Die Differenz für den knotenoptimierten Spline ist kleiner und besser über das ganze Intervall verteilt. Die Knoten wurden mit `optax`'s RMSProp-Optimierer mit Nesterov-Momentum optimiert. Die Optimierung wird in Abschnitt 4.2 genauer untersucht.

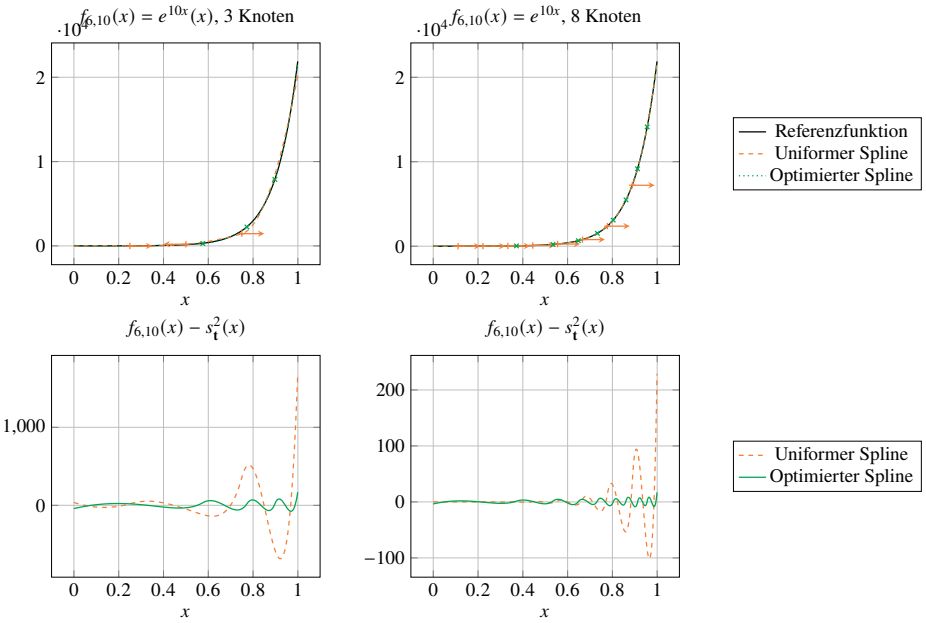

Abbildung 4.7: Bestapproximationsspline für 3 und 8 innere Knoten, jeweils mit äquidistanten (orange) und mit optimierten (grün) Knoten

Sinus-Funktion

Als letztes Beispiel betrachten wir die skalierte Sinus-Funktion

$$f_{7,a} : [0, 1) \to \mathbb{R}, \quad f_{7,a}(x) = \sin(ax2\pi)$$

für $a \in \{\frac{1}{2}, 1, \frac{3}{2}, 2\}$. Abbildung 4.8 zeigt die Funktion sowie die linearen Bestapproximationssplines jeweils mit äquidistanten und optimierten Knoten.

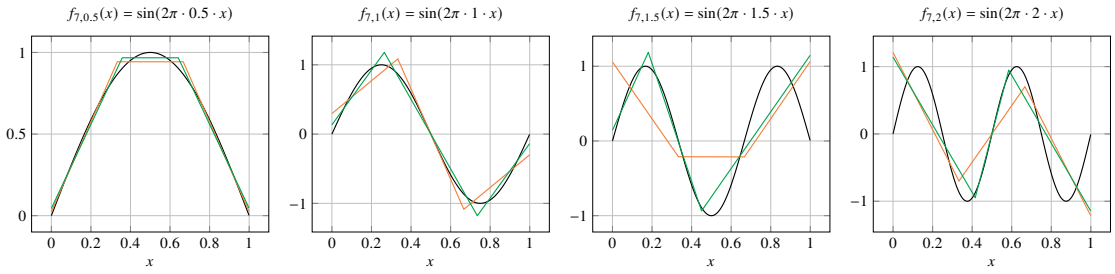

Abbildung 4.8: Sinusfunktion $f_{7,a}$ und lineare Splines mit äquidistanten und optimierten Knoten

Abbildung 4.9 zeigt den Approximationsfehler in Abhängigkeit der zwei inneren Knoten. Auffällig ist, dass sich das Verhalten von Splines mit geradem Grad von jenen mit ungeradem Grad unterscheidet. Für ungerade Splinegrade werden die optimalen Knoten näher an den Extremstellen positioniert. Für gerade Splinegrade befinden sich die optimalen Knotenpositionen näher an den Nullstellen der Sinusfunktion. Wenn die Sinusfunktion genau so viele inneren Nullstellen wie der Spline innere Knoten hat, dann befinden sich die optimale Knoten in den Nullstellen. Diese Beobachtung passt zu der bekannten Eigenschaft, dass der Approximationsfehler mit der k-ten Ableitung der

Referenzfunktion zusammenhängt. Falls die Skalierung a zu groß gewählt wird, kann ein Spline mit zwei inneren Knoten die Funktion $f_{7,a}$ nicht auf dem gesamten Intervall gut annähern und es entstehen lokale Minima.

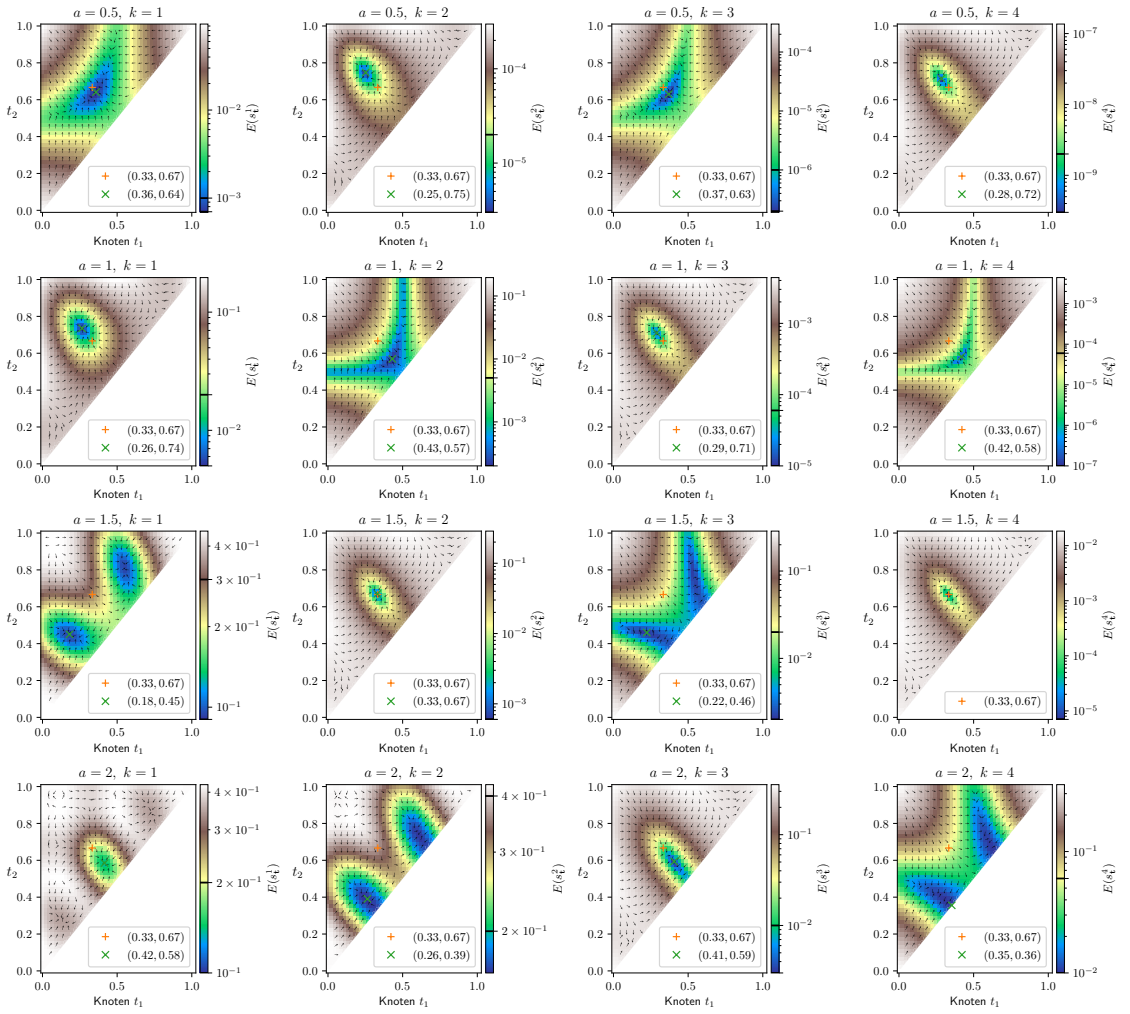

Abbildung 4.9: Approximationsfehler und dessen Gradient für die Sinusfunktion $f_{7,a}$

Abbildung 4.10 zeigt die kubischen Bestapproximationssplines für $f_{7,2}$ mit mehr als zwei Knoten. Ähnlich wie bei der Optimierung von zwei Knoten befinden sich die optimierten Knoten nahe der Extremstellen. Der Graph der Differenz zwischen der Funktion $f_{7,2}$ und der Splines illustriert die Verbesserung des Fehlers.

Abbildung 4.11 zeigt den Approximationsfehler in Abhängigkeit von der Knotenzahl. Die optimierten Knoten wurden dabei mittels Gradientenabstieg beginnend in den äquidistanten Knoten berechnet.

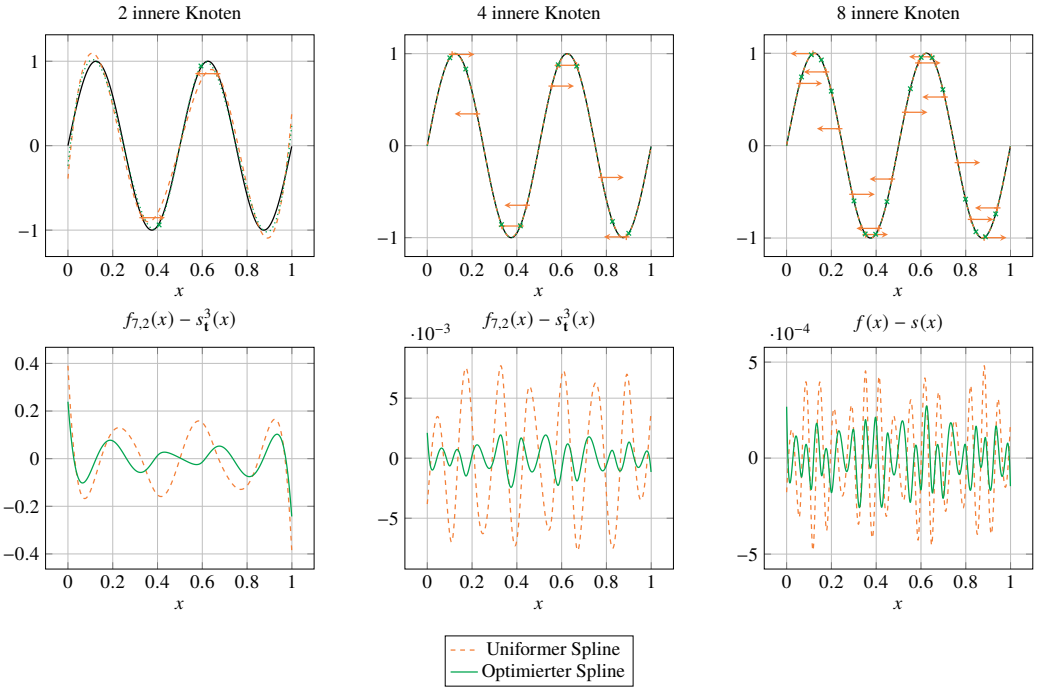

Abbildung 4.10: Sinus-Funktion $f_{7,2}(x) = \sin(4\pi x)$ und kubische Bestapproximationssplines für 2, 8 und 16 innere Knoten. Erste Zeile: Funktionsgraph von $f_{7,2}$ (schwarz), uniformer Spline (orange) und knotenoptimierter Spline (grün). Zweite Zeile: Differenz zwischen der Funktion und den Splines

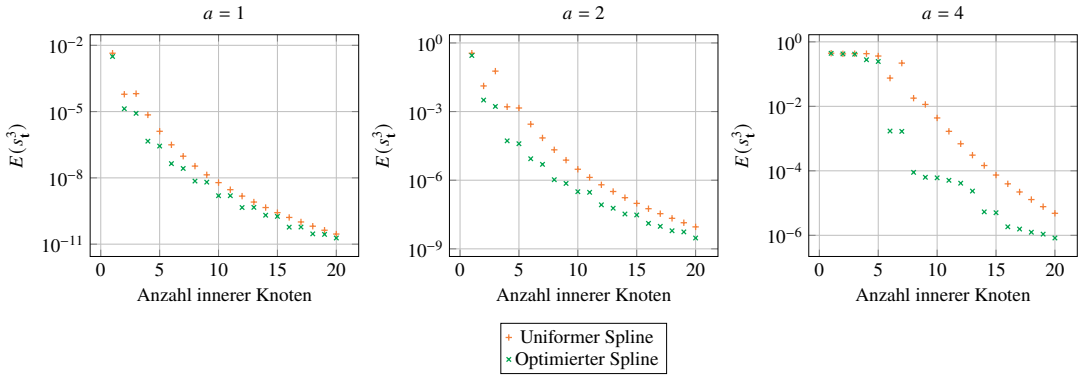

Abbildung 4.11: Approximationsfehler des uniformen und knotenoptimierten kubischen Bestapproximationssplines in Abhängigkeit der Anzahl der inneren Knoten für $f_{7,1}(x) = \sin(2\pi x)$ (links), $f_{7,4}(x) = \sin(4\pi x)$ (Mitte) und $f_{7,2}(x) = \sin(8\pi x)$ (rechts)

4.2 Vergleich der Optimierer

Der folgende Abschnitt vergleicht die Optimierer Stochastic Gradient Descent (SGD) und RMSProp mit den Minimierungsmethoden von `scipy.optimize.minimize`. SGD und RMSProp werden sowohl von `torch.optim` als auch von `optax` [HBV+20] angeboten. Die folgenden Beispiele verwenden die Funktionen von `optax`. Für den Vergleich betrachten wir erneut die Optimierung von zwei inneren Knoten bei der Approximation der Exponentialfunktion $f_{6,10} : [0, 1) \to \mathbb{R}, f_{6,a}(x) = \exp(ax)$. Abbildung 4.12 zeigt den Pfad der Optimierer Stochastic Gradient Descent (SGD) [SMDH13] und RMSProp [TH+12] für verschiedene Momenta.

Ein Update-Schritt von SGD addiert den mit der Lernrate skalierten Gradienten zum aktuellen Parameterwert. Bezeichnet h die Lernrate, $\mathbf{t}_i$ die Schätzung der Parameter im Schritt i und $\mathbf{d}_i$ den Gradienten des Approximationsfehlers, dann ist der Update-Schritt von SGD gegeben durch

$$\mathbf{t}_{i+1} := \mathbf{t}_i - h \cdot \mathbf{d}_i.$$

Da der Update-Schritt proportional zum Gradienten ist, ist die Konvergenz des einfachen SGD-Optimierer sehr stark abhängig von der Wahl der Lernrate h und der Norm des Gradienten. In Abbildung 4.13 erreichen die ersten 100 Iterationen mit SGD ohne Momentum nicht das Minimum in $(t_1, t_2) \approx (0.65, 0.86)$, da der Gradient im Verlauf der Optimierung sehr klein wird.

Diesem Problem kann mithilfe eines Momentums entgegengewirkt werden. Die Verwendung eines Momentums beschleunigt die Optimierung, indem es in einem Geschwindigkeitsvektor $\mathbf{v}_i$ die Gradienten akkumuliert. Der Momentumparameter $m :=$ `momentum` beschreibt die Trägheit. Der Update-Schritt ist dann gegeben durch

$$\begin{aligned}\mathbf{v}_{i+1} &:= m \cdot \mathbf{v}_i - \mathbf{d}_i,\\ \mathbf{t}_{i+1} &:= \mathbf{t}_i + h \cdot \mathbf{v}_{i+1}.\end{aligned}$$

Abbildung 4.12 zeigt, dass dies die Optimierung beschleunigt, indem die gleichbleibende Richtung, welche t_1 vergrößert, sich akkumuliert. Gleichzeitig führt die Akkumulation der Richtung zu stärkerer Oszillation entlang der Veränderung von t_2.

Das Nesterov-Momentum oszilliert deutlich schwächer, da es im Update-Schritt den aktuellen Gradienten stärker gewichtet. Die Update-Vorschrift mit Nesterov-Momentum ist gegeben durch

$$\begin{aligned}\mathbf{v}_{i+1} &:= m \cdot \mathbf{v}_i - \mathbf{d}_i,\\ \mathbf{t}_{i+1} &:= \mathbf{t}_i + h \cdot (m \cdot \mathbf{v}_{i+1} - \mathbf{d}_i).\end{aligned}$$

Der RMSProp-Optimierer normiert die Einträge des Gradienten durch einen laufenden Mittelwert. Zusätzlich zur Lernrate $h :=$ `learning_rate` ist er über die Parameter $\gamma :=$ `decay` und $\varepsilon :=$ `eps` parametrisiert. Sei $\mathbf{t}_i = (t_{i,1}, t_{i,2})$ die aktuelle Parameterschätzung, $\mathbf{d}_i = (d_{i,1}, d_{i,2})$ der Gradient des Fehlers bezüglich der Parameter und $\mathbf{a}_i = (a_{i,1}, a_{i,2})$ der bisherige laufende Mittelwert. Dann ist die Update-Vorschrift gegeben durch

$$\begin{aligned}a_{i+1,j} &= a_{i,j} \cdot \gamma + d_{i,j}^2 \cdot (1 - \gamma),\\ t_{i+1,j} &= t_{i,j} - h \cdot \frac{g_{i,j}}{\sqrt{d_{i+1,j} + \varepsilon}}\end{aligned}$$

für $j \in \{1, 2\}$. Analog zu SGD kann auch der RMSProp-Optimierer mit einem Momentum kombiniert werden.

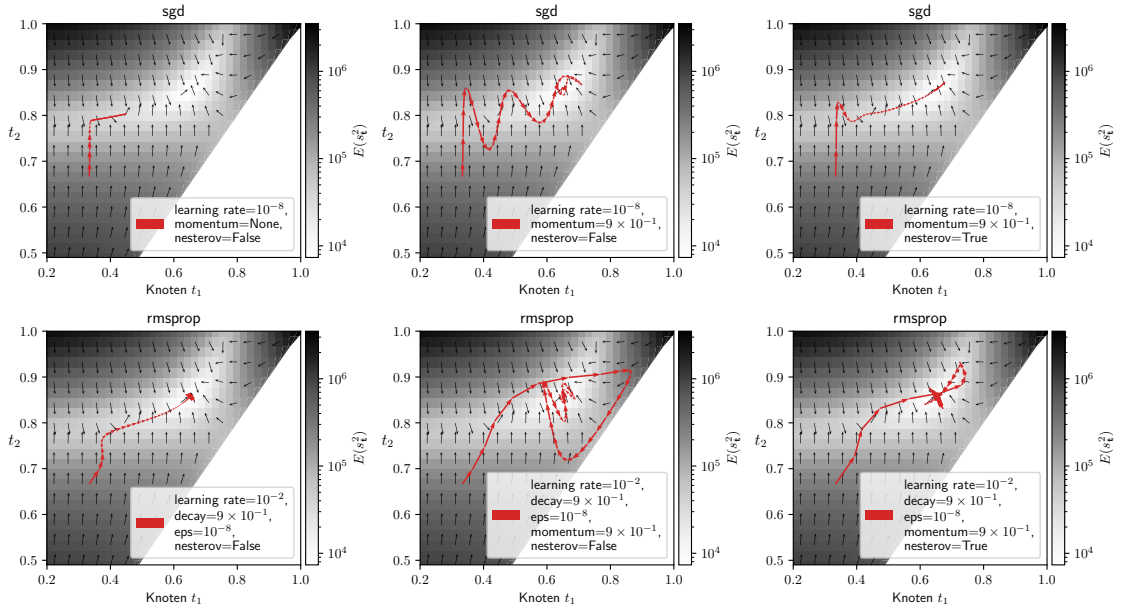

Abbildung 4.12: Pfad der Optimierer `sgd` und `rmsprop` von `optax` für 100 Iterationen. Links: kein Momentum, Mitte: mit Momentum, Rechts: mit Nesterov-Momentum

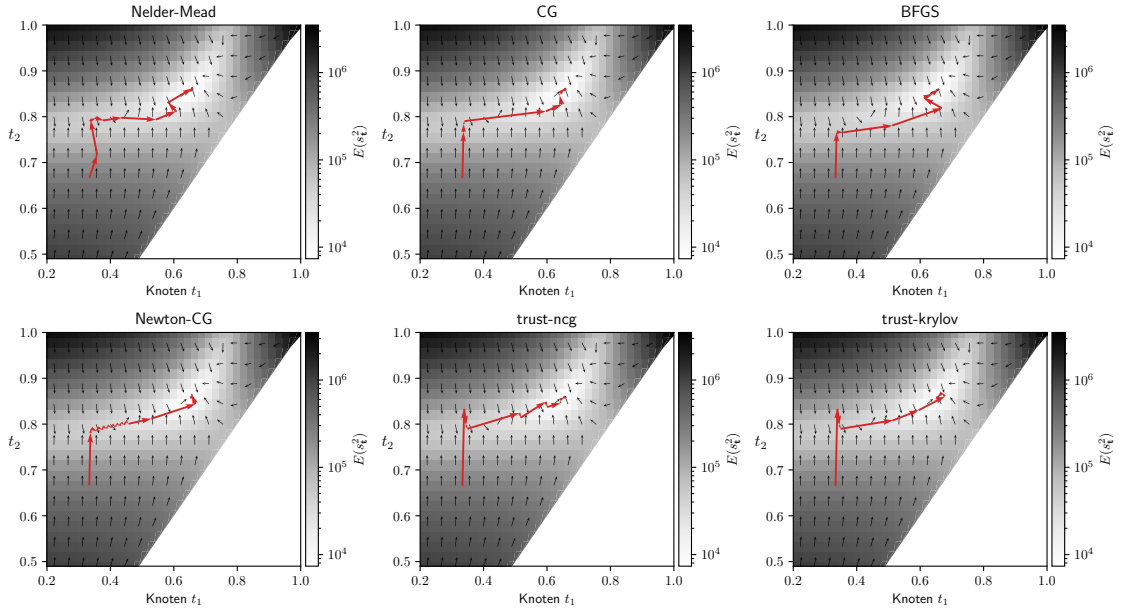

Abbildung 4.13: Pfad der Minimierungsmethoden von `scipy.optimize.minimize`

Zum Vergleich zeigt Abbildung 4.13 die Pfade für einige Minimierungsmethoden, welche von `scipy.optimize.minimize`[1] angeboten werden. Wie Tabelle 4.1 zeigt, verwendet die Methode von Nelder und Mead [NM65] als einzige keinen Gradienten. In den anderen Methoden wurde der Gradient durch automatisches Differenzieren mittels `jax.grad` ausgewertet. Die Verfahren `trust-ncg` und `trust-krylov` verwenden zusätzlich die Hesse-Matrix. Diese wurde mit der Funktion `jax.hessian` bestimmt.

method	nit	nfev	njev	nhev
Nelder-Mead	47	89	0	0
BFGS	14	33	23	0
CG	15	46	40	0
Newton-CG	25	28	91	0
trust-ncg	17	18	15	14
trust-krylov	14	16	16	13

Tabelle 4.1: Anzahl der Iterationen (`nit`) sowie Auswertungen der Funktion (`nfev`), des Gradienten (`njev`) und der Hesse-Matrix (`nhev`) der Minimierungsmethoden von `scipy.optimize.minimize`

[1] Dokumentation der Minimierer von SciPy, https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

5 Fractional Splines

Unser und Blu [UB00] definieren Fractional Splines, welche die bekannten Splines auf reelle Splinegrade verallgemeinern. Dieses Kapitel führt in Abschnitt 5.1 zunächst die benötigten Definitionen für Fractional Splines ein. Die Implementierung in JAX ermöglicht das automatische Differenzieren der Fractional Splines und damit des Approximationsfehlers. Abschnitt 5.2 zeigt einige Beispiele für die Approximation mit Fractional Splines.

5.1 Definition

Ein wichtiges Hilfsmittel für die Definition der Fractional B-Splines ist die Gamma-Funktion. Diese verallgemeinert die Fakultätsfunktion auf reelle Zahlen. Zudem kann mit ihr der Binomialkoeffizient verallgemeinert werden.

Definition 5.1.1 (Gamma-Funktion und Binomialkoeffizient)
Die Funktion $\Gamma : \mathbb{R} \to \mathbb{R}$ *ist definiert als*

$$\Gamma(u) = \int_0^{+\infty} x^{u-1} e^{-x} dx$$

für $u > 0$ *und induktiv* $\Gamma(u) = u^{-1}\Gamma(u+1)$ *für* $u < 0$*. Der* Binomialkoeffizient *von reellen Zahlen* $u, v \in \mathbb{R}$ *ist*

$$\binom{u}{v} = \frac{\Gamma(u+1)}{\Gamma(v+1)\Gamma(u-v+1)}.$$

Für eine einfachere Notation führen wir außerdem den modifizierten Binomialkoeffizienten

$$\left|\begin{matrix} u \\ v \end{matrix}\right| = \binom{u}{v + u/2}$$

ein.

Die Funktion Γ erfüllt $\Gamma(n+1) = n!$ für $n \in \mathbb{N}$ und verallgemeinert damit die Fakultät. Auf ähnliche Weise verallgemeinern die Fractional B-Splines die bekannten B-Splines auf reelle Splinegrade $\alpha \geq -1$. Die folgende Definition führt die Approximation der Centered Fractional B-Splines aus Theorem 2.5 in [UB00] ein.

Definition 5.1.2 (Centered fractional B-Spline)
Sei $\alpha \geq -1$ *ein reeller Splinegrad und* $N \in \mathbb{N}$*. Die* Approximation eines Centered Fractional B-Splines *ist*

- *für $\alpha \notin 2\mathbb{N}$ ungerade*

$$\beta^{\alpha}_{*,N}(x) = \frac{1}{2\sin(\frac{\pi}{2})\Gamma(\alpha+1)} \sum_{j=-N}^{N} (-1)^{j+1} \left| \begin{matrix} \alpha+1 \\ j \end{matrix} \right| |x-j|^{\alpha} + C^{\alpha}_{N}$$

mit dem Korrekturterm

$$C^{\alpha}_{N} = \frac{\alpha+1}{\pi \tan(\frac{\pi}{2}\alpha)} \left(\frac{1}{N} - \frac{1}{2N^2} \right), \text{ falls } \alpha \notin \mathbb{N}, \text{ sonst } C^{\alpha}_{N} = 0.$$

- *für $\alpha \in 2\mathbb{N}$ gerade*

$$\beta^{2n}_{*,N}(x) = \frac{(-1)^n}{2n!\pi} \sum_{j=-N}^{N} (-1)^{j+1} \left| \begin{matrix} 2n+1 \\ j \end{matrix} \right| |x-j|^{2n} \log|x-j| + C^{2n}_{N}$$

mit dem Korrekturterm

$$C^{2n}_{N} := \frac{4n+2}{\pi^2} \left(\frac{1+\log N}{N} - \frac{\log N}{2N^2} \right).$$

Für $N \to \infty$ erhalten wir die in [UB00, Gleichung 2.8, 2.9] definierten Centered Fractional B-Splines

$$\beta^{\alpha}_{*} := \lim_{N\to\infty} \beta^{\alpha}_{*,N}.$$

Der Korrekturterm C^{α}_{N} verbessert die Konvergenz $\beta^{\alpha}_{*,N} \to \beta^{\alpha}_{*}$ und konvergiert gegen $\lim_{N\to\infty} C^{\alpha}_{N} = 0$. Für ungerade, ganzzahlige Splinegrade $\alpha = k$ erhalten wir die bekannten B-Splines auf äquidistanten Knoten in den ganzen Zahlen. Genauer gilt

$$\beta^{k}_{*} = b^{k}_{0,\mathbf{t}}$$

mit $\mathbf{t} = (-\frac{k+1}{2}, -\frac{k+1}{2}+1, -\frac{k+1}{2}+2, \ldots, \frac{k+1}{2})$.

Der Raum der Fractional Splines vom Grad α ist [UB00, Gleichung 3.7]

$$\mathbf{S}^{\alpha}_{*} := \left\{ s \mid \exists c \in \ell^2 : s(x) = \sum_{j\in\mathbb{Z}} c(j)\beta^{\alpha}_{*}(x-j) \right\}.$$

Die Summe über j besteht aus unendlich vielen Summanden, da Fractional B-Splines keinen kompakten Träger haben. Für die Berechnung im Computer schneiden wir die Summe ab, d. h. wir betrachten nur Splines der Form

$$s^{\alpha}_{c,N,j_{\min},j_{\max}}(x) = \sum_{j=j_{\min}}^{j_{\max}} c_j \beta^{\alpha}_{*,N}(x-j)$$

mit den Koeffizienten $(c_{j_{\min}}, c_{j_{\min}+1}, \ldots, c_{j_{\max}})$. Damit hat die Approximation mit Fractional Splines neben dem Splinegrad α und den Koeffizienten c noch die weiteren Parameter $N, j_{\min}, j_{\max}$.

Approximation mit Fractional Splines Analog zur Approximation mit den bekannten Splines können Fractional Splines für die Funktionsapproximation verwendet werden. Hierfür seien Auswertestellen und abgetastete Funktionswerte $(x_0, f_0), \ldots, (x_{m-1}, f_{m-1})$ gegeben. Wir betrachten analog zu Definition 2.1.3 den Approximationsfehler

$$E(s) = \frac{1}{m} \sum_{i=0}^{m-1} (s(x_i) - f_i)^2$$

eines Fractional Splines s. In den folgenden Beispielen betrachten wir den Approximationsfehler einiger Beispielfunktionen in Abhängigkeit von dem Splinegrad α. Dabei fixieren wir jeweils $N = 2000$ und $j_{\min}, j_{\max}$ symmetrisch um die Auswertestellen $\mathbf{x}$. Damit ist die Anzahl der Ansatzfunktionen bzw. die Dimension des Splineraums ist $n = j_{\max} - j_{\min} + 1$. Die Koeffizienten c werden mit der Kleinste-Quadrate-Methode optimal gewählt. Damit erhalten wir die Fehlerfunktion in Abhängigkeit von α:

$$E_{N,j_{\min},j_{\max}}(\alpha) := \min_{c \in \mathbb{R}^n} E(s^{\alpha}_{c,N,j_{\min},j_{\max}})$$

5.2 Beispiele

Als erstes Beispiel betrachten wir die Identität auf dem Intervall $[-10, 10]$

$$f_1 : [-10, 10] \to \mathbb{R}, \quad f_1(x) = x.$$

Zur Bestimmung des Approximationsfehlers verwenden wir 1500 abgetasteten Funktionswerte $f_1(x_i)$ in äquidistanten Auswertestellen $x_i = -10 + 20 \cdot \frac{i}{1500}$ für $0 \leq m < 1500$. Abbildung 5.1 zeigt den Approximationsfehler in Abhängigkeit vom Splinegrad α sowie dessen Ableitung nach α.

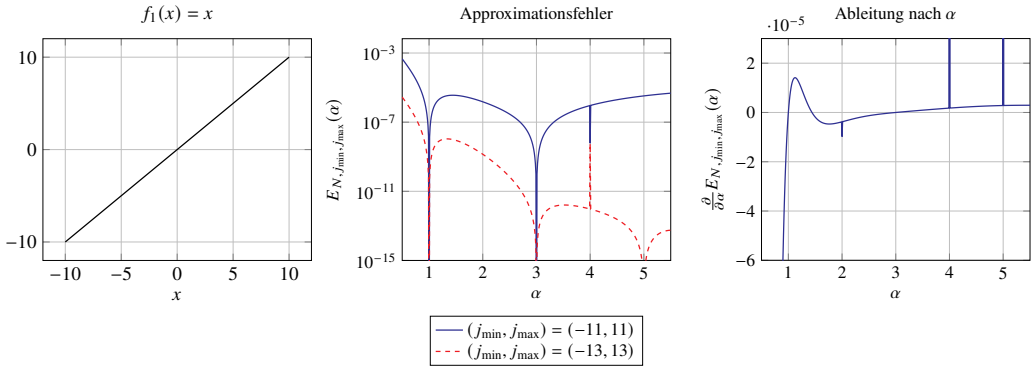

Abbildung 5.1: Funktionsgraph (links), Approximationsfehler (Mitte) und Ableitung nach dem Splinegrad α (rechts) für die Referenzfunktion f_1

Unser und Blue [UB00, Abschnitt 4.1] zeigen, dass die Approximation von Polynomen vom Grad $k \in \mathbb{N}$ mit Fractional Splines vom Grad $\alpha > k - 1$ exakt ist. Daher illustriert Abbildung 5.1 vor allem die Auswirkung der abgeschnittenen Summen. Aufgrund des kompakten Trägers der klassischen B-Splines ist die Approximation für ganzzahlige, ungerade Splinegrade α exakt, falls $j_{\min} \leq -10 - \frac{\alpha-1}{2}$ und $j_{\max} \geq 10 + \frac{\alpha-1}{2}$. Dementsprechend ist die Approximation mit Splines vom Grad $\alpha \in \{1, 3\}$ exakt, falls $j_{\min} \leq -11$ und $j_{\max} \geq 11$. Für $j_{\min} \leq -12$ und $j_{\max} \geq 12$ ist die

Approximation auch mit dem Splinegrad $\alpha \in \{5, 7\}$ exakt. Für nicht-ganzzahlige Splinegrade $\alpha \notin \mathbf{Z}$ hat die Wahl der Summengrenzen $j_{\min}$ und $j_{\max}$ einen größeren Einfluss als der Splinegrad. Für nicht-ganzzahlige Grade α ist der Approximationsfehler stetig und kann nach α abgeleitet werden. Der rechte Graph in Abbildung 5.1 zeigt die Ableitung des Fehlers $E_{2000,j_{\min},j_{\max}}(\alpha)$ nach α für $(j_{\min}, j_{\max}) = (-11, 11)$.

Als Zweites betrachten wir die Potenzfunktion

$$f_2 : [0, 10] \to \mathbb{R}, \quad f_2(x) = x^{\frac{3}{2}}.$$

von nicht-ganzzahligem Grad $\frac{3}{2}$. Abbildung 5.2 zeigt den Approximationsfehler für verschiedene Summengrenzen $j_{\min}, j_{\max}$ sowie die Ableitung des Fehlers nach α für $(j_{\min}, j_{\max}) = (-6, 16)$. Wie für f_1 hängt auch für f_2 der Approximationsfehler stärker von der Anzahl der Basisfunktionen als vom Splinegrad ab. Für $(j_{\min}, j_{\max}) = (0, 10)$ ist der Approximationsfehler auch in ganzzahligen α stetig. Sobald weitere Basisfunktionen links bzw. rechts hinzugefügt werden, entstehen allerdings Unstetigkeiten in den ganzzahligen Splinegraden. Für $j_{\min} \leq -6$ und $j_{\max} \geq 16$ ist der optimale Splinegrad $\alpha \approx \frac{3}{2}$. Allerdings wird die Ableitung für zu viele Basisfunktionen außerhalb des Definitionsbereichs $[0, 10]$ instabil.

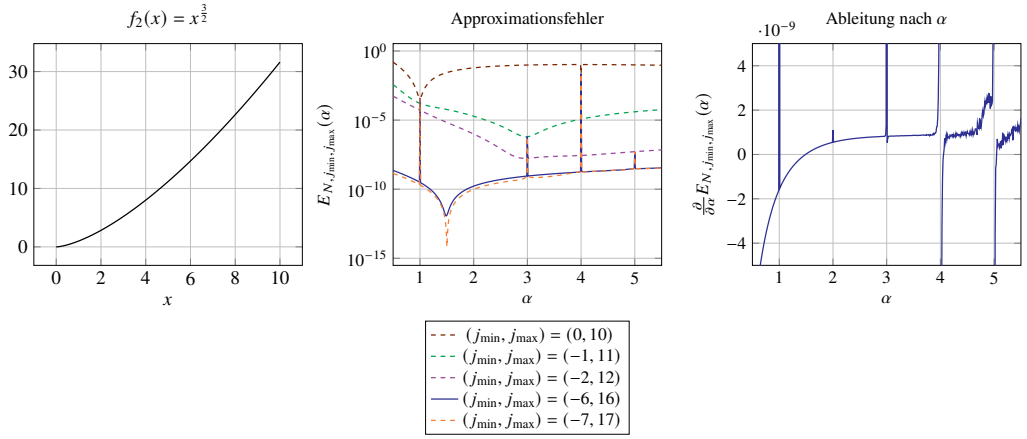

Abbildung 5.2: Funktionsgraph (links), Approximationsfehler (Mitte) und Ableitung nach dem Splinegrad α (rechts) für die Referenzfunktion f_2

Abbildung 5.3 zeigt den Approximationsfehler und dessen Ableitung für die Funktion

$$f_3 : [0, 10] \to \mathbb{R}, \quad f_3(x) = \sin\left(\frac{\pi x}{2}\right).$$

Die Sinusfunktion wurde so skaliert, dass sich die Extremstellen und Nullstellen in den ganzen Zahlen befinden. Für dieses Beispiel verbessert sich die Approximation mit zunehmendem Splinegrad deutlich stärker als durch die Verwendung zusätzlicher Basisfunktionen am Rand.

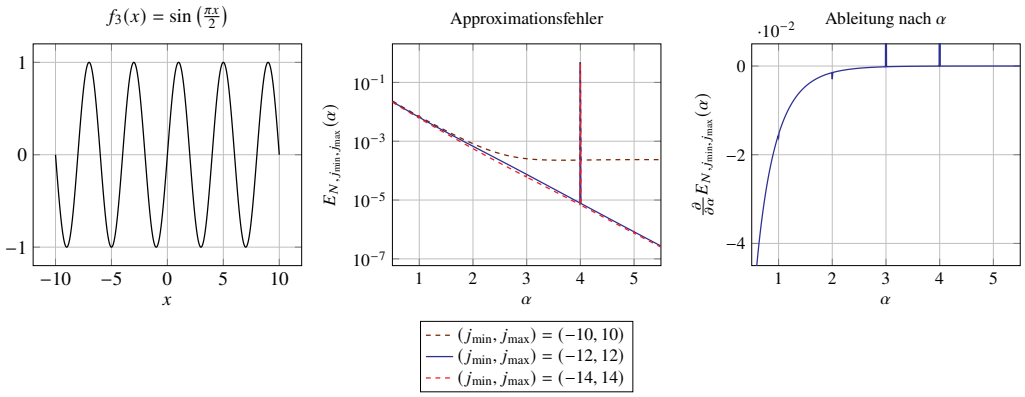

Abbildung 5.3: Funktionsgraph (links), Approximationsfehler (Mitte) und Ableitung nach dem Splinegrad α (rechts) für die Referenzfunktion f_3

6 Zusammenfassung

In dieser Arbeit wurde die Splineapproximation auf Basis von B-Splines untersucht und implementiert. Dabei wurde die Umsetzung in den Frameworks JAX mit PyTorch verglichen. Obwohl die Implementierung der Berechnung des Approximationsfehlers sogar syntaktisch sehr ähnlich ist, wurden auch einige Unterschiede der beiden Frameworks diskutiert. Beide Frameworks ermöglichen die Ableitung des Fehlers mittels automatischem Differenzieren. Das Verhalten des Approximationsfehlers sowie dessen Gradient bezüglich der Knoten wurde anhand einiger Beispielfunktionen untersucht. Diese Untersuchung zeigte, dass die Wahl der Knoten den Approximationsfehler stark beeinträchtigt. Selbst wenn nur ein lokales Minimum gefunden wird, wird der Fehler durch die Verschiebung der Knoten entlang der Gradienten oft deutlich verbessert. Mit Hilfe des Gradienten können etablierter Optimierungsalgorithmen angewandt werden, um die Knoten zu optimieren. Einige bekannte Algorithmen wurden vorgestellt und verglichen.

Abschließend wurde die Approximation mit Fractional Splines betrachtet. Während die Berechnung der Fractional Splines für praktische Anwendung zu aufwändig ist, liefern sie spannende Erkenntnisse in der theoretischen Betrachtung. Beispielsweise ermöglicht der reelle Grad der Fractional Splines die Ableitung der Splines und damit des Fehlers nach dem Splinegrad. Trotz der Komplexität der Berechnung von Fractional Splines kann automatisches Differenzieren diese Ableitung berechnen. Dies ermöglicht Optimierungen des Splinegrads, welche über den Rahmen dieser Arbeit hinausgehen.

Ausblick

Die vorgestellte Berechnung der Ableitung eröffnet neue Möglichkeiten für die Optimierung der Knoten eines Splines. Während bereits einige Optimierungsalgorithmen vorgestellt wurden, bleibt zu untersuchen, welche der Algorithmen sich für praktische Anwendungen eignen. Insbesondere die Optimierung von mehr als zwei inneren Knoten wurde in dieser Arbeit nur sehr kurz betrachtet. Zudem führt die Verwendung der Optimierungsalgorithmen zu den typischen Fragen nach der geeigneten Wahl der Parameter. Die Kombination aus verschiedenen Initialisierungen für die Knoten, Parametern der Optimierer und die Wahl der Abbruchbedingung bieten weiteren Spielraum für eine effizientere Optimierung der Knoten.

Literaturverzeichnis

[BFH+18] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.24. 2018. URL: http://github.com/google/jax (zitiert auf S. 19, 21).

[Boo78] C. de Boor. *A practical guide to splines*. Englisch. Applied mathematical sciences ; 27. Literaturverz. S. 377 - 387. New York ; Berlin ; Heidelberg [u.a.]: Springer, 1978. ISBN: 3540903569 (zitiert auf S. 18).

[BR68] C. de Boor, J. R. Rice. „Least Squares Cubic Spline Approximation, II - Variable Knots“. In: 1968 (zitiert auf S. 15, 19).

[CG93] G. F. Corliss, A. Griewank. „Operator overloading as an enabling technology for automatic differentiation“. In: 1993 (zitiert auf S. 19).

[DT17] V. T. Dung, T. Tjahjowidodo. „A direct method to solve optimal knots of B-spline curves: An application for non-uniform B-spline curves fitting“. In: *PLOS ONE* 12.3 (März 2017), S. 1–24. DOI: 10.1371/journal.pone.0173857. URL: https://doi.org/10.1371/journal.pone.0173857 (zitiert auf S. 15).

[HBV+20] M. Hessel, D. Budden, F. Viola, M. Rosca, E. Sezener, T. Hennigan. *Optax: composable gradient transformation and optimisation, in JAX!* Version 0.0.9. 2020. URL: http://github.com/deepmind/optax (zitiert auf S. 39, 51).

[HH13] K. Höllig, J. Hörner. *Approximation and Modeling with B-Splines*. Philadelphia, PA: Society for Industrial und Applied Mathematics, 2013. DOI: 10.1137/1.9781611972955. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972955. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972955 (zitiert auf S. 17).

[NM65] J. A. Nelder, R. Mead. „A Simplex Method for Function Minimization“. In: *Comput. J.* 7 (1965), S. 308–313 (zitiert auf S. 53).

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. „PyTorch: An Imperative Style, High-Performance Deep Learning Library“. In: *Advances in Neural Information Processing Systems 32*. Hrsg. von H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett. Curran Associates, Inc., 2019, S. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (zitiert auf S. 19, 21).

[SMDH13] I. Sutskever, J. Martens, G. Dahl, G. Hinton. „On the importance of initialization and momentum in deep learning“. In: *Proceedings of the 30th International Conference on Machine Learning*. Hrsg. von S. Dasgupta, D. McAllester. Bd. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 2013, S. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html (zitiert auf S. 51).

[SS95] H. Schwetlick, T. Schütze. „Least squares approximation by splines with free knots“. In: *BIT Numerical Mathematics* 35 (1995), S. 361–384 (zitiert auf S. 15, 40).

[SW53] I. J. Schoenberg, A. Whitney. „On Pólya frequence functions. III. The positivity of translation determinants with an application to the interpolation problem by spline curves“. In: *Transactions of the American Mathematical Society* 74.2 (1953), S. 246–259 (zitiert auf S. 18).

[TH+12] T. Tieleman, G. Hinton et al. „Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude“. In: *COURSERA: Neural networks for machine learning* 4.2 (2012), S. 26–31 (zitiert auf S. 51).

[UB00] M. Unser, T. Blu. „Fractional Splines and Wavelets“. In: *SIAM Review* 42.1 (2000), S. 43–67. DOI: 10.1137/S0036144598349435. eprint: https://doi.org/10.1137/S0036144598349435. URL: https://doi.org/10.1137/S0036144598349435 (zitiert auf S. 15, 55–57).

Alle URLs wurden zuletzt am 29. 11. 2021 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift