Dept. Mathematik
Naturwissenschaftlich - Technische Fakultät

Interpolation räumlicher Kurven mit Anwendungen auf die Vliesstoffproduktion

Bachelorarbeit

vorgelegt von
Julius Busse
am 21. Dezember 2021

Gutachter
Prof. Dr. Volker Michel
Prof. Dr. Robert Plato

Inhaltsverzeichnis

1. Einleitung

1.1. Zusammenfassung

In dieser Arbeit stellen wir zunächst ein Modellierungsproblem für einen diskretisierten dreidimensionalen Faden vor. Zu dessen Lösung leiten wir Gleichungssysteme für die kubische Splineinterpolation mit wahlweise natürlichen oder Hermite-Randbedingungen her. Mithilfe von Software lösen wir dann ein Interpolationsproblem und danach das Modellierungsproblem am Würfelgitter. Schlussendlich werden noch Lösungsansätze am Parallelepipedgitter vorgestellt.

1.2. Einführung in Vliesstoffe und das Thema der Arbeit

Der Begriff „Vliesstoff“ wird vom europäischen Dachverband für Vliesstoffproduktion wie folgt definiert:

Als Vliesstoffe bezeichnet man bestimmte Anordnungen von Fasern nicht näher spezifizierter Art, Länge und von beliebigem Durchmesser, welche durch physikalische oder chemische Methoden zusammengehalten werden, wobei Stoffe, die durch Weben, Zusammenknüpfen oder Methoden der Papierherstellung entstanden sind, nicht als Vliesstoffe gelten [4, Übersetzung durch den Autor].

Abbildung 1: Der Autor mit einer FFP-2-Atemschutzmaske.

Vliesstoffe werden in diversen Anwendungen verwendet, etwa bei Wärmeisolation, Flüssigkeitsfiltration oder beim Bau von Verkehrswegen als Geovliesstoffe (siehe [2]). Die Verwendung von medizinischen Masken, wie etwa von FFP-2-Masken (Abbildung 1 und 2), ist momentan

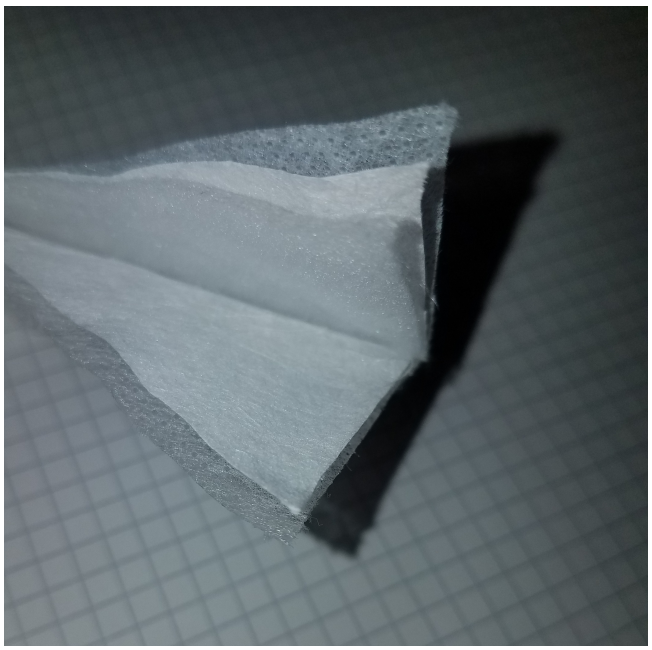

Abbildung 2: Eine Nahaufnahme einer aufgeschnitten FFP-2-Atemschutzmaske.

üblich. Die inneren Schichten dieser Masken bestehen ebenfalls aus Vliesstoffen, insbesondere aus solchen, die mit *Meltblown*-Verfahren hergestellt werden [11]. Bei *Meltblown*-Verfahren wird Kunststoff geschmolzen und durch Düsen in einen Luftschacht geleitet. Es wird dabei etwa Polyamid, Polypropylen, PET oder PMMA verwendet. Im Luftschacht werden die einzelnen Fäden verwirbelt und legen sich auf die Vliesablage (Abbildung 3). Solche Düsen sind in Abbildung 4 gezeigt. Den resultierenden Stoff unter dem Mikroskop kann man in Abbildung 5 sehen.

Nun sind die Bewegungen der Luft im Schacht zu beschreiben. Die simulierte Luftgeschwindigkeit ist ohne Betrachtung der Temperaturabgabe der Fäden unter Umständen keine gute Approximation an die Realität, siehe etwa [3, Abbildung 7]. Zum Zwecke der Beschreibung der Temperaturabgabe dieser heißen Fäden an die Umgebung betrachten wir einen solchen Faden genauer.

1.3. Kernproblem der Modellierung

Es ist also ein Faden in einem Luftschacht zu modellieren. Dieser Faden soll durch eine Funktion $f : [0, 1] \to \mathbb{R}^3$ gegeben sein. Es soll für ein gegebenes Gitter G (etwa aus Parallelepipeden) – dessen Form später noch genauer beschrieben wird – die Länge, die mittlere Temperatur und die Dicke des Fadens in jedem Gitterelement approximiert werden. Dazu sind die Werte $f(x) \in \mathbb{R}^3$ auf dem äquidistanten Gitter $\Delta_n := \{0 = x_0 < \cdots < x_n = 1\}$ gegeben. Die Funktion f sei an den Intervallrändern differenzierbar mit bekannter Ableitung. Weiterhin seien die Temperaturen des Fadens am selben Gitter durch t und die Dicke durch d gegeben. Wir wollen zu diesem Zwecke zunächst eine f interpolierende Funktion s_f mit guten Differenzierbarkeitseigenschaften erhalten, welche wir auch zwischen den Stützstellen auswerten können [10, S.131].

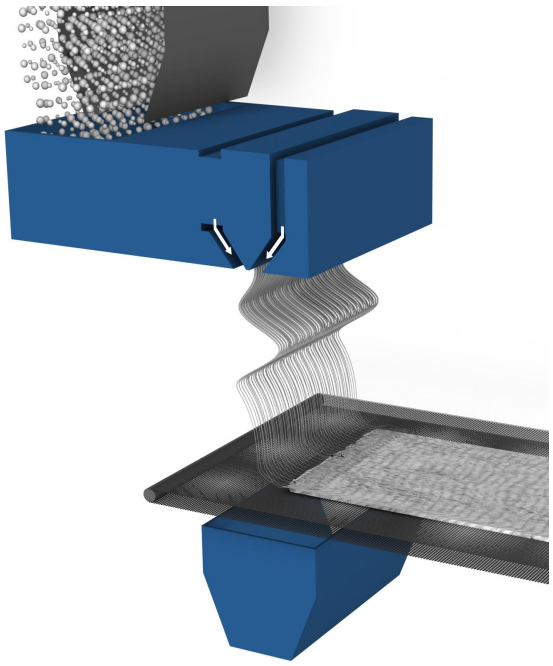

Abbildung 3: Produktion eines *Meltblown*-Vliesstoffs [5].

Dann konstruieren wir auch noch Funktionen, welche die Temperatur und die Dicke interpolieren. Dort genügt es, das Interpolationsproblem in einer Dimension zu lösen.

Es ist nun noch eine Methodik zu entwickeln, mit der man die Fadenlänge und die durchschnittliche Temperatur bzw. Fadendicke in jedem Gitterelement bestimmen kann. Dazu gehen wir wie folgt vor:

1. Interpoliere den Faden f sowie die Temperatur t und die Dicke d durch geeignete Splinekurven im $\mathbb{R}^3$, etwa den Faden f durch kubische Splines mit wahlweise natürlichen oder Hermite-Randbedingungen und t und d durch kubische Splines mit natürlichen Randbedingungen.

2. Berechne die Schnittpunkte des interpolierenden Splines mit den Grenzen zwischen den Gitterelementen des Gitters G. Die Schnittstellen nennen wir $\{y_j\}_{j=1}^{m-1}$ und ordnen sie bezüglich des zugehörigen Kurvenparameters s aufsteigend an (siehe Abbildung 6). Außerdem nennen wir den Wert der Kurve am linken Intervallrandpunkt y_0 und den am rechten Intervallrandpunkt y_m.

3. Berechne das Integral

$$L_j := \mu(\mathcal{C}_{y_j,y_{j+1}}) = \int_{\mathcal{C}_{y_j,y_{j+1}}} 1 \,\mathrm{d}s = \int_{y_j}^{y_{j+1}} |s'(x)| \,\mathrm{d}x,$$

   für $j = 0, \ldots, m-1$ numerisch, wobei $\mathcal{C}_{a,b} = \{s(t) : t \in [a,b]\}$ mit $0 \leq a < b \leq 1$.

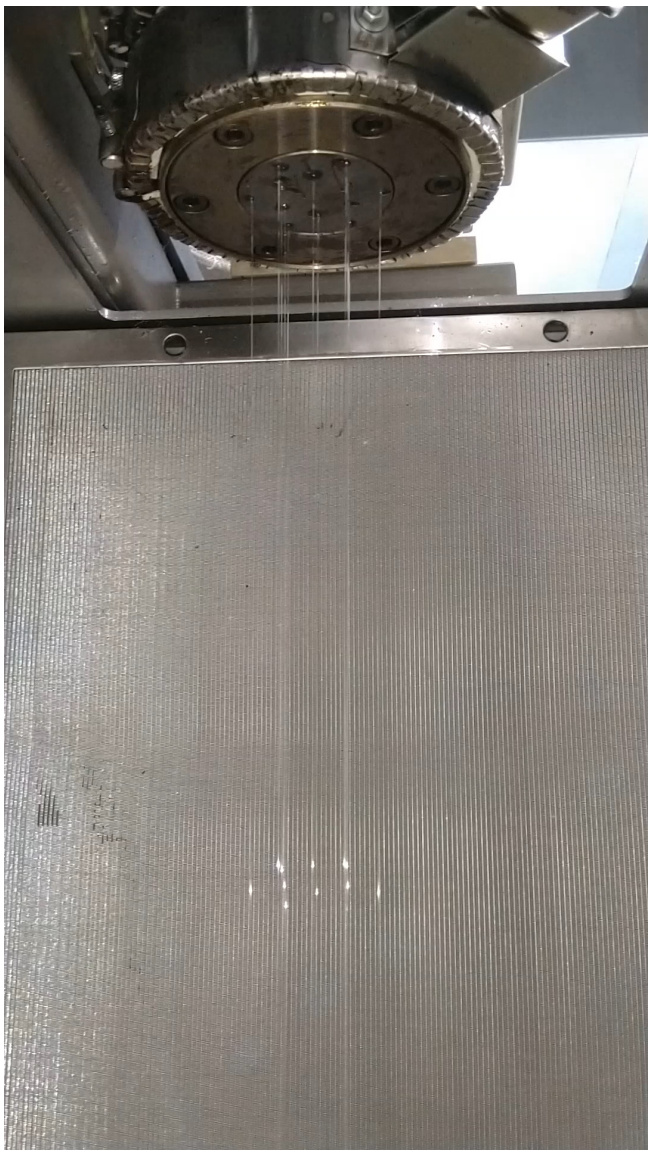

Abbildung 4: Bild einer Schmelzspinnanlage der Universität Maastricht [1].

4. Berechne die Integrale

$$T_j := \int_{\mathcal{C}_{y_j,y_{j+1}}} t(s)\,\mathrm{d}s, \quad D_j := \int_{\mathcal{C}_{y_j,y_{j+1}}} d(s)\,\mathrm{d}s,$$

   für $j = 0, \ldots, m-1$.

5. Schlussendlich ergibt sich der Mittelwert der Temperatur durch $\frac{T_j}{y_{j+1}-y_j}$ und der Mittelwert der Dicke durch $\frac{D_j}{y_{j+1}-y_j}$. Sollte der Fall $L_j = 0$ auftreten, so sind die Mittelwerte der Temperatur und der Dicke nicht definiert und somit auch nicht interessant.

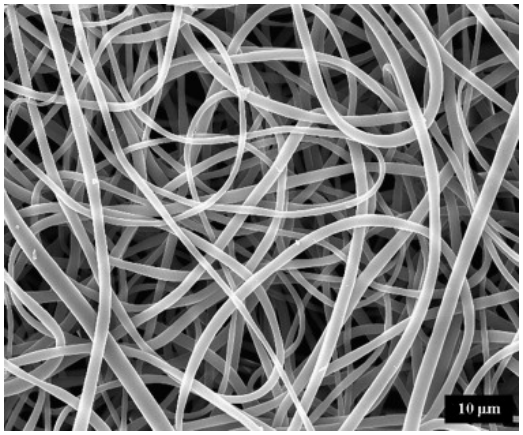

Abbildung 5: Mikroskopische Aufnahme eines Vliesstoffs (aus [8, Abbildung 52]).

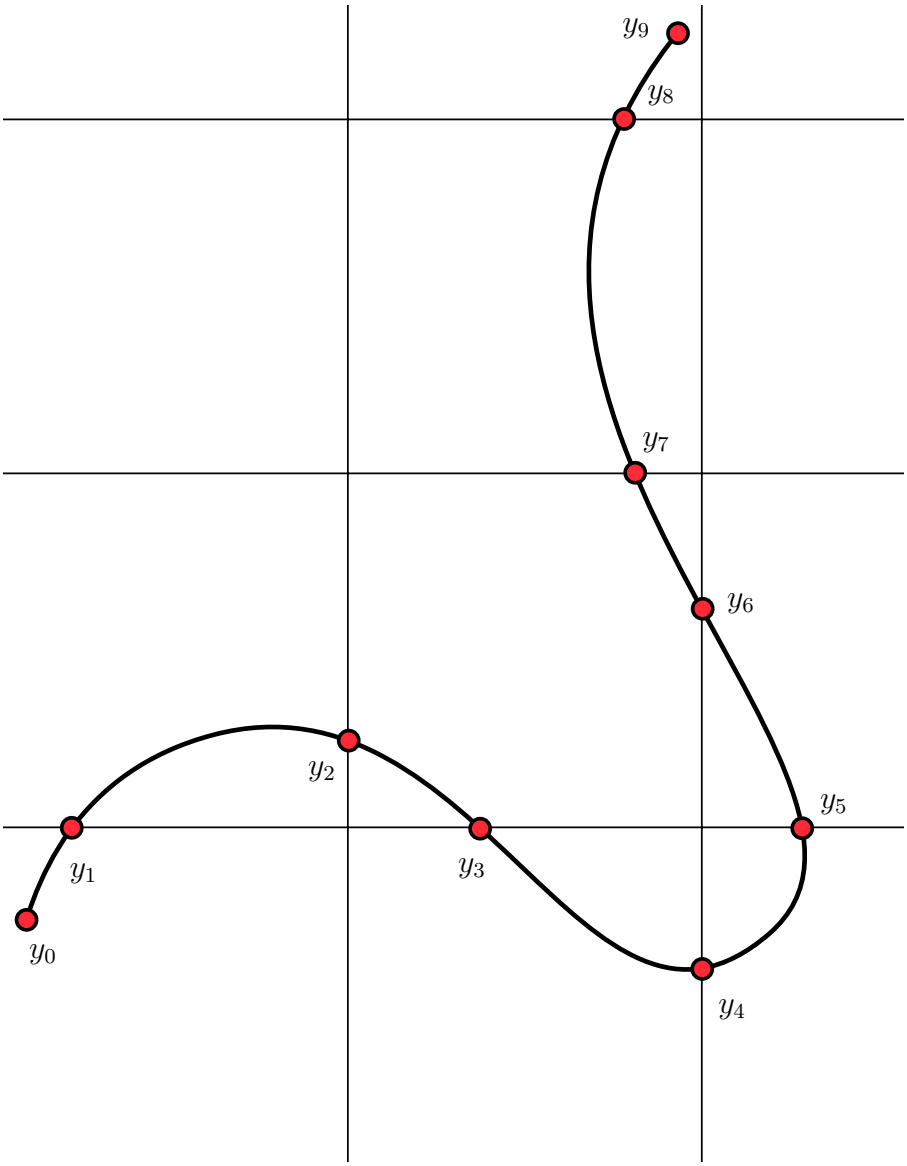

Abbildung 6: Illustration der Bedeutung der y_j in zwei Dimensionen.

2. Grundlagen

Für spätere Fehlerabschätzungen und Minimaleigenschaften benötigen wir einige Grundlagen, die im folgenden Teil der Arbeit vorgestellt und teilweise auch bewiesen werden.

Definition 2.1. Im Folgenden definieren wir zu einer Zerlegung $\Delta_n := \{a = x_0 < x_1 < \cdots < x_n = b\}$ die Menge

$$\mathrm{S}_m(\Delta_n) := \{s \in \mathrm{C}^{m-1}[a,b] \ : \ s|_{[x_j,x_{j+1}]} \in \Pi_m, \ j = 0, \ldots, n-1\}$$

und nennen sie die Menge der *Splines vom Grad m zur Zerlegung* Δ_n.

Definition 2.2. Des Weiteren nutzen wir zur Kennzeichnung von Funktionswerten einer Funktion f auf einem Gitter $\Delta_n := \{x_0 < \cdots < x_n\}$ die Bezeichnung $f_j := f(x_j)$ für $j = 0, \ldots, n$.

Aus [7, S. 29] wurde die folgende Abschätzung mit Beweis übernommen:

Lemma 2.3. *Jede strikt diagonaldominante Matrix* $A = (a_{j,k})_{j,k=1,\ldots,N}$ *ist regulär. Weiterhin gilt die Abschätzung*

$$\|x\|_\infty \leq \max_{j=1}^{N} \left[\left(|a_{jj}| - \sum_{\substack{k=1 \\ k \neq j}}^{N} |a_{jk}| \right)^{-1} \right] \|Ax\|_\infty,$$

für $x \in \mathbb{R}^N$.

Beweis. Sei x_j der betragsmäßig größte Eintrag des Vektors $x \in \mathbb{R}^N$, also $|x_j| = \|x\|_\infty$. Wir schätzen dann ab:

$$\begin{aligned}
\|Ax\|_\infty \geq |(Ax)_j| &= \left| \sum_{k=1}^{N} a_{jk} x_k \right| \\
&\geq |a_{jj}| \cdot |x_j| - \left| \sum_{\substack{k=1 \\ k \neq j}}^{N} a_{jk} x_k \right| \\
&\geq |a_{jj}| \cdot |x_j| - \sum_{\substack{k=1 \\ k \neq j}}^{N} |a_{jk}| \cdot |x_k| \\
&\geq |a_{jj}| \cdot |x_j| - \sum_{\substack{k=1 \\ k \neq j}}^{N} |a_{jk}| \cdot \|x\|_\infty \\
&= \left(|a_{jj}| - \sum_{\substack{k=1 \\ k \neq j}}^{N} |a_{jk}| \right) \|x\|_\infty.
\end{aligned}$$

Aus der strikten Diagonaldominanz erhalten wir, dass der resultierende Faktor vor dem $\|x\|_\infty$ größer Null ist. Eine deshalb zulässige Umstellung ergibt nun das Resultat

$$\|x\|_\infty \leq \left(|a_{jj}| - \sum_{\substack{k=1 \\ k \neq j}}^{N} |a_{jk}| \right)^{-1} \|Ax\|_\infty$$

Da diese Abschätzung für ein bestimmtes, also fest gewähltes j gilt, gilt sie gewiss auch, wenn wir auf der rechten Seite ein Maximum über alle j nehmen. Die Behauptung ist somit gezeigt. □

Dieses Lemma verwenden wir später, um Fehlerabschätzungen interpolierender kubischer Splines mit natürlichen Randbedingungen nachzuweisen, da dort strikt diagonaldominante Matrizen auftreten.

Aus [7, S. 26] wurde das folgende Lemma übernommen.

Lemma 2.4. *Wir nutzen im Folgenden die Kurzschreibweise $M_j := s_j''$ für $j = 0, \ldots, n$. Falls die Zahlen $M_0, \ldots, M_n \in \mathbb{R}$ (auch Momente genannt) die Gleichungen*

$$h_{j-1} M_{j-1} + 2(h_{j-1} + h_j) M_j + h_j M_{j+1} = \underbrace{6 \frac{f_{j+1} - f_j}{h_j} - 6 \frac{f_j - f_{j-1}}{h_{j-1}}}_{=: g_j}$$

für $j = 0, \ldots, n-1$ erfüllen, so liefert der Ansatz

$$s(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3, \quad \text{für } x \in [x_j, x_{j+1}]$$

für $j = 0, \ldots, n-1$ mit den Koeffizienten

$$\begin{aligned} a_j &:= f_j, \\ b_j &:= \frac{f_{j+1} - f_j}{h_j} - \frac{h_j}{6} \left(M_{j+1} + 2M_j \right), \\ c_j &:= \frac{M_j}{2}, \\ d_j &:= \frac{M_{j+1} - M_j}{6h_j}, \end{aligned}$$

einen kubischen Spline $s \in S_3(\Delta_n)$, welcher bezüglich f und Δ_n interpolatorisch ist, also den Bedingungen $s(x_j) = f(x_j)$ für $j = 0, \ldots, n$ genügt.

Aus [7, S. 28] erhalten wir das folgende Gleichungssystem für die eben erklärten Momente M_j:

Lemma 2.5. *Die natürlichen Randbedingungen $s_0'' = s_N'' = 0$ liefern gemeinsam mit Lemma*

2.4 das Gleichungssystem

$$\begin{pmatrix} 2(h_0+h_1) & h_1 & 0 & \dots & 0 \\ h_1 & 2(h_1+h_2) & h_2 & \ddots & \vdots \\ 0 & h_2 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & h_{N-2} \\ 0 & \dots & 0 & h_{N-2} & 2(h_{N-2}+h_{N-1}) \end{pmatrix} \begin{pmatrix} M_1 \\ \vdots \\ M_{N-1} \end{pmatrix} = \begin{pmatrix} g_1 \\ \vdots \\ g_{N-1} \end{pmatrix}.$$

Die Hermite-Randbedingungen $f'_0 = s'_0 = b_0$, $f'_N = s'_N = b_{N-1} + 2c_{N-1}h_{N-1} + 3d_{N-1}h_{N-1}^2$ liefern unter Verwendung von Lemma 2.4 die zusätzlichen Bedingungen

$$2h_0M_0 + h_0M_1 = -6f'_0 + 6\frac{f_1 - f_0}{h_0} =: g_0,$$
$$h_{N-2}M_{N-2} + 2h_{N-1}M_N = 6f'_N - 6\frac{f_N - f_{N-1}}{h_{N-1}} =: g_N.$$

In Matrix-Vektor-Schreibweise ergibt sich dieses Gleichungssystem als

$$\begin{pmatrix} 2h_0 & h_0 & 0 & \dots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \ddots & \vdots \\ 0 & h_1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2(h_{N-2}+h_{N-1}) & h_{N-1} \\ 0 & \dots & 0 & h_{N-1} & 2h_{N-1} \end{pmatrix} \begin{pmatrix} M_0 \\ \vdots \\ M_N \end{pmatrix} = \begin{pmatrix} g_0 \\ \vdots \\ g_N \end{pmatrix}.$$

Beweis. Die zusätzlichen Bedingungen ergeben sich durch Einsetzen der Definition für a_j, b_j, c_j und d_j und anschließendes Umstellen. □

Diese Gleichungssysteme werden später benötigt, um Darstellungen für interpolierende kubische Splines zu erhalten. Diese nutzen wir später zur Berechnung von Fehlerschranken.

Aus [6, S. 144] übernehmen wir leicht modifiziert das Lemma von Holladay ohne Beweis:

Lemma 2.6. *(Holladay) Wenn eine Funktion $f \in \mathrm{C}^2[a,b]$ und ein kubischer Spline $s \in \mathrm{S}_3(\Delta_n)$ in den Knoten übereinstimmen, so gilt*

$$\|f'' - s''\|_2^2 = \|f''\|_2^2 - \|s''\|_2^2 - 2\left([f' - s']s''\right)(x)\Big|_{x=a}^{b}.$$

Die folgende Aussage ist übernommen aus [7, S. 23].

Satz 2.7. *Gegeben seien eine Funktion $f \in \mathrm{C}^2[a,b]$ und ein kubischer Spline $s_f \in S_3(\Delta_n)$, die in den Knoten übereinstimmen. Dann gilt die Identität*

$$\|f''\|_2^2 - \|s''\|_2^2 = \|f'' - s''\|_2^2,$$

sofern natürliche oder Hermite-Randbedingungen vorliegen.

Beweis. Wir erhalten diese Aussage leicht als Korollar von Lemma 2.6. Bei Hermite-Randbedingungen verschwindet der Ausdruck $f' - s'$ am Rand, bei natürlichen Randbedingungen der Ausdruck s''. Offenbar verschwindet der Ausdruck $([f' - s']s'')(x)\big|_{x=a}^{b}$ also sowohl bei natürlichen als auch bei Hermite-Randbedingungen, womit die Behauptung gezeigt ist. □

Auf Basis dieses Satzes erhalten wir interessante Minimaleigenschaften interpolierender kubischer Splines, sofern gewisse Randbedingungen erfüllt sind.

Der nachfolgende Satz ist etwa in [7, S. 11] zu finden.

Satz 2.8. *Ist $f \in \mathrm{C}^{n+1}[a,b]$, und $p_n(f) \in \Pi_n$ sei das Polynom, was f an den paarweise verschiedenen Stützstellen $x_0 < \cdots < x_n \in [a,b]$ interpoliert. Dann gibt es zu jedem $x \in [a,b]$ eine Zwischenstelle $\xi \in (\min(x_0,x), \max(x_n,x))$ mit*

$$f(x) - p_n(f)(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^{n} (x - x_k).$$

Für spätere Fehlerabschätzungen interpolierender linearer Splines werden wir diesen Satz für den Fall $n = 1$ anwenden.

Die folgende Aussage ist mit Beweis aus [7, S. 85 f] übernommen.

Lemma 2.9. *Für eine durch eine Vektornorm induzierte Matrixnorm $\|\cdot\| : \mathbb{R}^{N\times N} \to \mathbb{R}_+$ und jede Matrix $B \in \mathbb{R}^{N\times N}$ mit $\|B\| < 1$ ist die Matrix $I + B$ regulär, und es gilt*

$$\|(I+B)^{-1}\| \leq \frac{1}{1 - \|B\|}.$$

Beweis. Dies beweisen wir mit der umgekehrten Dreiecksungleichung für $x \in \mathbb{R}^N$:

$$\begin{aligned} \|(I+B)x\| = \|x + Bx\| &\geq \|x\| - \|Bx\| \\ &\geq \|x\| - \|B\|\|x\| = (1 - \|B\|)\|x\|, \end{aligned}$$

da $1 - \|B\| > 0$ gilt, folgern wir hieraus, dass die rechte Seite nur für $x = 0$ identisch Null werden kann. Demnach besteht der Kern von $I + B$ nur aus der Null, womit wir Regularität von $I + B$ folgern.

Wir erhalten die Aussage nun mit Substitution von $y = (I + B)x$, es ergibt sich also

$$\|y\| \geq (1 - \|B\|)\|(I+B)^{-1}y\| \Leftrightarrow \|(I+B)^{-1}y\| \leq \frac{1}{1 - \|B\|}\|y\|.$$

Da die Matrix $I + B$ als reguläre Matrix insbesondere surjektiv ist, lässt sich jedes $y \in \mathbb{R}^N$ in solch einer Form darstellen. Damit ergibt sich die zu zeigende Ungleichung. □

Für spätere Fehlerabschätzungen interpolierender kubischer Splines mit Hermite-Randbedingungen treten Matrizen auf, deren Zeilensummennorm man mit dieser Ungleichung abschätzen kann.

3. Bézier-Kurven

3.1. Problemstellung

Wir nutzen im Folgenden Bézier-Kurven, um folgende Interpolationsprobleme zu lösen:

$$s(x_j) = f(x_j), \quad s \in \mathrm{C}^2[0,1], \quad s''(x_0) = s''(x_n) = 0 \qquad \text{(natürliche Randbed.)}$$
$$s(x_j) = f(x_j), \quad s \in \mathrm{C}^2[0,1], \quad s'(x_0) = f'(x_0),\ s'(x_n) = f'(x_n) \qquad \text{(Hermite-Randbed.)},$$

für $j = 1, \ldots, n$ bezüglich einer gewissen Unterteilung Δ_n.

3.2. Definition: Bézier-Kurven

Definition 3.1. Die Definition ist übernommen aus [12, S. 209 ff]. Zu $n \in \mathbb{N}$ definieren wir zunächst die $n+1$ *Bernstein-Polynome* vom Grad n für $i = 0, \ldots, n$ wie folgt:

$$B_i^n(t) := \binom{n}{i} t^i (1-t)^{n-i}.$$

Zu vorgegebenen Punkten $b_0, \ldots, b_n \in \mathbb{R}$, den *Bézier-Punkten* oder *Kontrollpunkten*, definieren wir

$$p(t) := \sum_{i=0}^{n} b_i B_i^n(t),$$

und nennen p das zugehörige *Bézier-Polynom* vom Grad n. Die dadurch parametrisierte Kurve $c := p([0,1])$ heißt *Bézier-Kurve*.

3.3. Lösung des Interpolationsproblems mit Bézier-Kurven

Wir wollen nun zunächst mithilfe dieser Bézier-Polynome C^2-Interpolationen von Mengen von Punkten im $\mathbb{R}^3$ beziehungsweise im $\mathbb{R}$ konstruieren. Wie man am untenstehenden Gleichungssystem erkennen kann, sind hierfür kubische Bézier-Polynome geeignet, also $n = 3$. Mit der oben definierten Form ergibt sich folgende allgemeine Form:

$$s(t) = \sum_{i=0}^{3} B_i^3(t) b_i = (1-t)^3 b_0 + 3t(1-t)^2 b_1 + 3t^2(1-t) b_2 + t^3 b_3.$$

In Abbildung 7 sehen wir ein Beispiel für ein Bézier-Polynom. Die Kontrollpunkte b_0, b_1, b_2, b_3 sind in orange markiert, das Polynom selbst in schwarz. Man erkennt, dass Start- und Endpunkt durch b_0 bzw. durch b_3 kontrolliert werden, und die Ableitungen an Start- und Endpunkt durch b_1 bzw. b_2. Das legt die Verwendung zur Konstruktion einer C^2-Interpolierenden nahe.

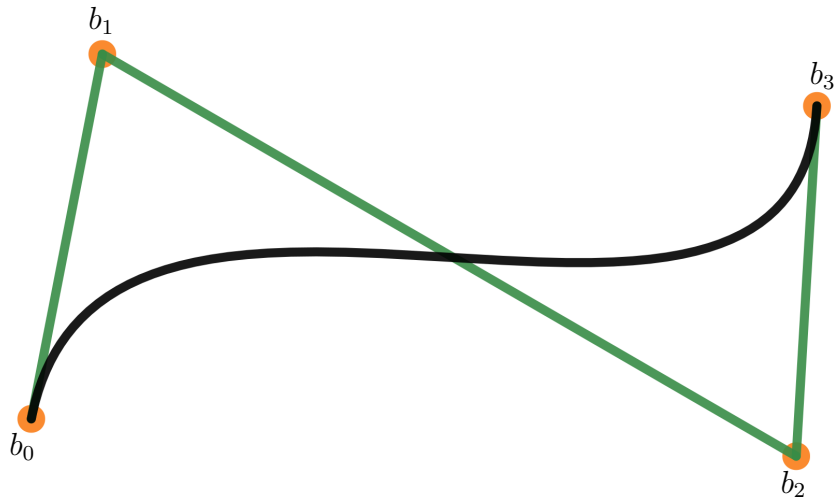

Abbildung 7: Ein Bézier-Polynom mit eingezeichneten Kontrollpunkten (in orange).

Wir wollen nun diese Bézier-Polynome wie Polynome bei Splines zweimal stetig differenzierbar verbinden, also fordern wir

$$s'_{i-1}(1) = s'_i(0), \quad s''_{i-1}(1) = s''_i(0), \qquad \text{für alle } i = 1, \dots, n \tag{1}$$

sowie zunächst die natürlichen Randbedingungen

$$s''_0(0) = s''_{n-1}(1) = 0. \tag{2}$$

Nun zur konkreten Situation: Wir wollen die Funktion $f : [0,1] \to \mathbb{R}^3$ oder $f : [0,1] \to \mathbb{R}$ interpolieren, wobei wir sie auf dem äquidistanten Punktgitter $\Delta_n = \{0 = x_0 < \cdots < x_n = 1\}$ auswerten können. Genaueres zur Konstruktion der Splinekurve im dreidimensionalen Fall findet man in [10, S. 131] Die entsprechenden Funktionswerte bezeichnen wir mit $f_i := f(x_i)$, $i = 0, \dots, n$. Dafür nutzen wir die folgenden n Kurven

$$s_i(t) := (1-t)^3 f_i + 3t(1-t)^2 \alpha_i + 3t^2(1-t)\beta_i + t^3 f_{i+1}, \quad i = 0, \dots, n-1.$$

Die Interpolationseigenschaft ist in der zusammengesetzten Kurve also bereits erfüllt, wie man in der Definition erkennen kann. Gesucht sind also noch Parameter α_i und β_i für $i = 0, \dots, n-1$, so dass die zusammengesetzte Kurve

$$s : [0,1] \to \mathbb{R}, \quad s(t) := \begin{cases} s_i(nt - \lfloor nt \rfloor), & \text{wenn } \lfloor nt \rfloor = i \text{ und } t \neq 1 \text{ für } i = 0, \dots, n-1 \\ s_{n-1}(1), & \text{wenn } t = 1 \end{cases}$$

C^2 ist. Die Ableitungen der Teilkurven errechnen sich wie folgt.

$$s_i'(t) = 3\left[-(1-t)^2 f_i + (1-3t)(1-t)\alpha_i + t(2-3t)\beta_i + t^2 f_{i+1}\right],$$
$$s_i''(t) = 6\left[(1-t)f_i + (3t-2)\alpha_i + (1-3t)\beta_i + t f_{i+1}\right].$$

Bei späterer Verwendung von Hermite-Randbedingungen ist zu beachten, dass man durch die Kettenregel einen Faktor n erhält, was durch die Skalierung im Argument von s begründet wird. Nun setzen wir diese Resultate in die Bedingungen von Gleichung (1) ein und erhalten somit für die einfache stetige Differenzierbarkeit:

$$\begin{aligned} & s_{i-1}'(1) = s_i'(0) \\ \Leftrightarrow\ & 3(-\beta_{i-1} + f_i) = 3(-f_i + \alpha_i) \\ \Leftrightarrow\ & \alpha_i + \beta_{i-1} = 2f_i, \end{aligned} \tag{3}$$

für $i = 1, \ldots, n-1$. Für die zweifache stetige Differenzierbarkeit ergibt sich analog

$$\begin{aligned} & s_{i-1}''(1) = s_i''(0) \\ \Leftrightarrow\ & 6(\alpha_{i-1} - 2\beta_{i-1} + f_i) = 6(f_i - 2\alpha_i + \beta_i) \\ \Leftrightarrow\ & \alpha_{i-1} + 2\alpha_i = 2\beta_{i-1} + \beta_i, \end{aligned} \tag{4}$$

für $i = 1, \ldots, n-1$. Nun wollen wir ein Gleichungssystem aufstellen und benötigen dazu nur noch die natürlichen Randbedingungen (2):

$$s_0''(0) = 0 \Leftrightarrow f_0 - 2\alpha_0 + \beta_0 = 0$$

und

$$s_{n-1}''(1) = 0 \Leftrightarrow \alpha_{n-1} - 2\beta_{n-1} + f_n = 0.$$

Das bisherige Gleichungssystem lautet also

$$\begin{aligned} \alpha_i + \beta_{i-1} &= 2f_i, && i = 1, \ldots, n-1, \\ \alpha_{i-1} + 2\alpha_i &= 2\beta_{i-1} + \beta_i, && i = 1, \ldots, n-1, \\ f_0 - 2\alpha_0 + \beta_0 &= 0, \\ \alpha_{n-1} - 2\beta_{n-1} + f_n &= 0. \end{aligned}$$

Nun ist unser Ziel die Elimination der β_j. Dafür nutzen wir Gleichung (3) und stellen wie folgt um:

$$\begin{aligned} \alpha_i + \beta_{i-1} &= 2f_i, && i = 1, \ldots, n-1, \\ \Leftrightarrow\ \beta_i &= 2f_{i+1} - \alpha_{i+1}, && i = 0, \ldots, n-2. \end{aligned} \tag{5}$$

Wir brauchen auch noch eine Darstellung für β_{n-1}, dafür nutzen wir (4) für $i = n-1$:

$$\begin{aligned}
\alpha_{(n-1)-1} + 2\alpha_{n-1} &= 2\beta_{(n-1)-1} + \beta_{n-1} \\
&\Leftrightarrow \beta_{n-1} = \alpha_{n-2} + 2\alpha_{n-1} - 2\beta_{n-2} \\
&\overset{(5)}{\Leftrightarrow} \beta_{n-1} = \alpha_{n-2} + 2\alpha_{n-1} - 2(2f_{n-1} - \alpha_{n-1}) \\
&\Leftrightarrow \beta_{n-1} = \alpha_{n-2} + 4\alpha_{n-1} - 4f_{n-1}. \qquad (6)
\end{aligned}$$

Zunächst betrachten wir unser Gleichungssystem ohne die Gleichungen (5) und (6). Wir wollen β_j in den übrigen Gleichungen eliminieren und ein Gleichungssystem erhalten, was wir nach den α_i auflösen. Nachdem diese Lösung berechnet wurde, rekonstruieren wir die β_i. Wir kümmern uns nun um die Elimination der β_i in den folgenden Gleichungen:

$$\begin{aligned}
f_0 - 2\alpha_0 + \beta_0 &= 0, \\
\alpha_{i-1} + 2\alpha_i &= 2\beta_{i-1} + \beta_i, \quad i = 1, \ldots, n-2, \\
\alpha_{n-1} - 2\beta_{n-1} + f_n &= 0.
\end{aligned}$$

Zu diesem Zwecke rechnen wir:

$$\begin{aligned}
& f_0 - 2\alpha_0 + \beta_0 = 0 \\
\overset{(5)}{\Leftrightarrow}\ & f_0 - 2\alpha_0 + 2f_1 - \alpha_1 = 0 \\
\Leftrightarrow\ & f_0 + 2f_1 = 2\alpha_0 + \alpha_1.
\end{aligned}$$

Für die zweite Gleichung ergibt sich:

$$\begin{aligned}
\alpha_{i-1} + 2\alpha_i &= 2\beta_{i-1} + \beta_i, && i = 1, \ldots, n-2 \\
\overset{(5)}{\Leftrightarrow} \alpha_{i-1} + 2\alpha_i &= 2(2f_i - \alpha_i) + (2f_{i+1} - \alpha_{i+1}), && i = 1, \ldots, n-2 \\
\Leftrightarrow \alpha_{i-1} + 4\alpha_i + \alpha_{i+1} &= 2(2f_i + f_{i+1}), && i = 1, \ldots, n-2.
\end{aligned}$$

Um die β_i schlussendlich zu eliminieren, stellen wir die letzte Gleichung noch wie folgt um:

$$\begin{aligned}
& \alpha_{n-1} - 2\beta_{n-1} + f_n = 0 \\
\overset{(6)}{\Leftrightarrow}\ & \alpha_{n-1} - 2(\alpha_{n-2} + 4\alpha_{n-1} - 4f_{n-1}) + f_n = 0 \\
\Leftrightarrow\ & 2\alpha_{n-2} + 7\alpha_{n-1} = 8f_{n-1} + f_n.
\end{aligned}$$

Fassen wir diese Resultate nun zusammen, so erhalten wir:

$$\begin{aligned} 2\alpha_0 + \alpha_1 &= f_0 + 2f_1 && \\ \alpha_{i-1} + 4\alpha_i + \alpha_{i+1} &= 2(2f_i + f_{i+1}), && i = 1, \ldots, n-2 \\ 2\alpha_{n-2} + 7\alpha_{n-1} &= 8f_{n-1} + f_n, && \end{aligned} \tag{7}$$

wobei sich die erste und letzte Gleichung aus den natürlichen Randbedingungen ergeben. Wir wollen dieses System nun als Matrix-Vektor-Gleichungssystem schreiben. Wir erhalten ein System der Form

$$Ax = b, \quad \text{mit } A \in \mathbb{R}^{n\times n} \text{ und } x, b \in (\mathbb{R}^3)^n,$$

beziehungsweise im eindimensionalen Fall mit $x, b \in \mathbb{R}^n$. Aus (7) wird folgendes Matrix-Vektor Gleichungssystem:

$$\begin{pmatrix} 2 & 1 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 4 & 1 \\ & & & & 2 & 7 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-2} \\ \alpha_{n-1} \end{pmatrix} = \begin{pmatrix} f_0 + 2f_1 \\ 2(2f_1 + f_2) \\ 2(2f_2 + f_3) \\ \vdots \\ 2(2f_{n-2} + f_{n-1}) \\ 8f_{n-1} + f_n \end{pmatrix}.$$

Die Tridiagonalmatrix A ist offenbar strikt diagonaldominant, also regulär. Demnach ist das zugehörige Gleichungssystem eindeutig lösbar. Die inverse Matrix wird durch Programmkode berechnet. Was bleibt, ist die Rekonstruktion der β_i aus den nun bestimmten α_i, wofür wir Gleichungen (5) und (6) nutzen.

Kommen wir nun zur Herleitung des Gleichungssystems bei Hermite-Randbedingungen. Dafür sind nur die ersten und letzten Zeilen von A und b zu verändern. Zunächst stellen wir die Gleichungen für Hermite-Randbedingungen auf. Diese sind

$$\begin{aligned} s_0'(0) &= f_0' \quad \text{und} \\ s_{n-1}'(1) &= f_n', \end{aligned}$$

wobei die Kurzschreibweisen $f_0' := s'(0) = ns_0'(0)$ und $f_n' := s'(1) = ns_{n-1}'(1)$ verwendet werden.

Wir formen die Bedingung für den linken Intervallrand wie folgt um:

$$\begin{aligned} s_0'(0) &= f_0' \\ \Leftrightarrow 3n[-f_0 + \alpha_0] &= f_0' \\ 3\alpha_0 &= 3f_0 + f_0'/n. \end{aligned}$$

Für die Bedingung am rechten Intervallrand erhalten wir

$$
\begin{aligned}
s'_{n-1}(1) &= f'_n \\
\Leftrightarrow 3n[-\beta_{n-1} + f_n] &= f'_n \\
\overset{(6)}{\Leftrightarrow} 3f_n - 3[\alpha_{n-2} + 4\alpha_{n-1} - 4f_{n-1}] &= f'_n/n \\
\Leftrightarrow 3\alpha_{n-2} + 12\alpha_{n-1} &= 3f_n + 12f_{n-1} - f'_n/n.
\end{aligned}
$$

Die Faktoren n stammen hierbei jeweils wie zuvor erwähnt aus der Kettenregel. Ersetzen wir nun also die jeweils ersten und letzten Zeilen von A und b durch die Gleichungen, die wir aus den Hermite-Randbedingungen erhalten, so ergibt sich das folgende Gleichungssystem:

$$
\begin{pmatrix}
3 & 0 & & & & & \\
1 & 4 & 1 & & & & \\
 & 1 & 4 & 1 & & & \\
 & & \ddots & \ddots & \ddots & & \\
 & & & 1 & 4 & 1 & \\
 & & & & 3 & 12
\end{pmatrix}
\begin{pmatrix}
\alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{n-2} \\ \alpha_{n-1}
\end{pmatrix}
=
\begin{pmatrix}
3f_0 + f'(0)/n \\
2(2f_1 + f_2) \\
2(2f_2 + f_3) \\
\vdots \\
2(2f_{n-2} + f_{n-1}) \\
-f'(1)/n + 3f_n + 12f_{n-1}
\end{pmatrix}.
$$

Auch diese Tridiagonalmatrix ist strikt diagonaldominant und somit regulär.

3.4. Resultate

Es werden nun Resultate vorgestellt, die mit Daten produziert wurden, die freundlicherweise vom Fraunhofer-Institut für Techno- und Wirtschaftsmathematik zur Verfügung gestellt wurden [9]. Dieselbe Kurve wurde einmal an 500 und einmal an 20 äquidistanten Punkten in $[0, 1]$ ausgewertet. Unsere optische Verifizierung der Resultate hat nun die Gestalt, aus den 20 Punkten die restlichen Punkte zu rekonstruieren. Der Begriff „Fehler“ bezieht sich in diesem Abschnitt also auf die Abweichung der berechneten interpolierenden Kurven zu einem gegebenen Datensatz, welcher feiner diskretisierte Positionsdaten des Fadens enthält. Somit stellt der kleine Fehler eine gute optische Verifizierung der Korrektheit der Interpolation dar.

Die Resultate sind in Abbildung 8, 9, 10 und 11 dargestellt. Da in Abbildung 8 kein Unterschied zwischen Interpolation mit natürlichen Randbedingungen und mit Hermite-Randbedingungen zu erkennen wäre, wurde der markierte Ausschnitt des Bildes in Abbildung 9 größer dargestellt. Durchgeführt wurde die Interpolation sowohl mit natürlichen (immer in grün), als auch mit Hermite-Randbedingungen (immer in blau). Schön zu erkennen ist ebenfalls, dass der Fehler am Rand bei Hermite-Randbedingungen deutlich geringer als der bei natürlichen Randbedingungen ist (siehe Abbildung 10).

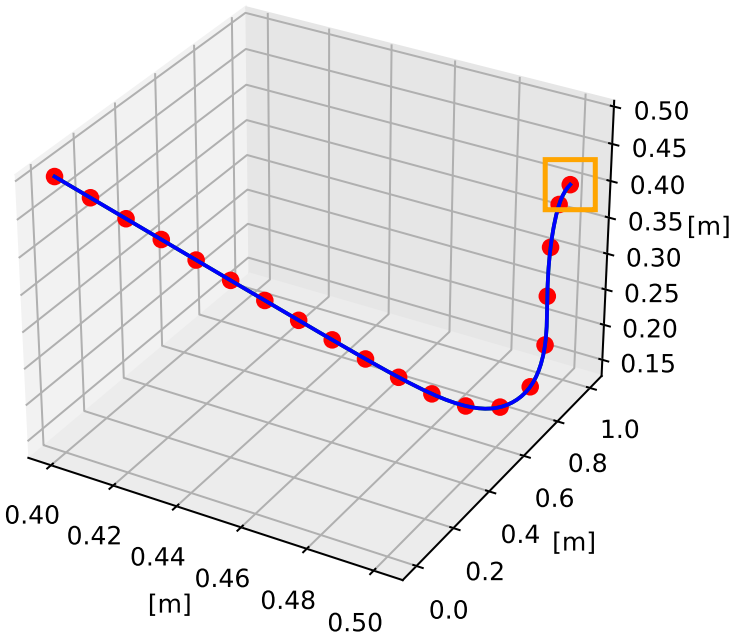

Abbildung 8: Bézier-Interpolation einer räumlichen Kurve mit natürlichen und Hermite-Randbedingungen. Der markierte Bereich ist in Abbildung 9 genauer zu erkennen.

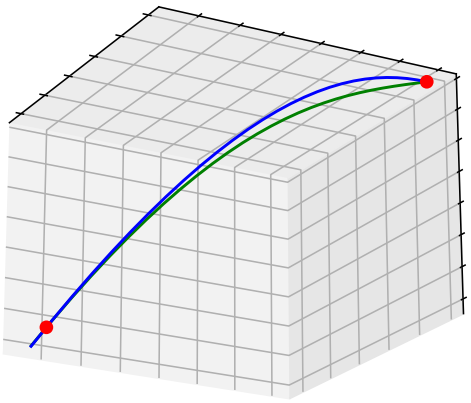

Abbildung 9: Unterschied zwischen Interpolation mit natürlichen (grün) und Hermite-Randbedingungen (blau) am linken Intervallrand. Leicht gedrehter Ausschnitt aus Abbildung 8.

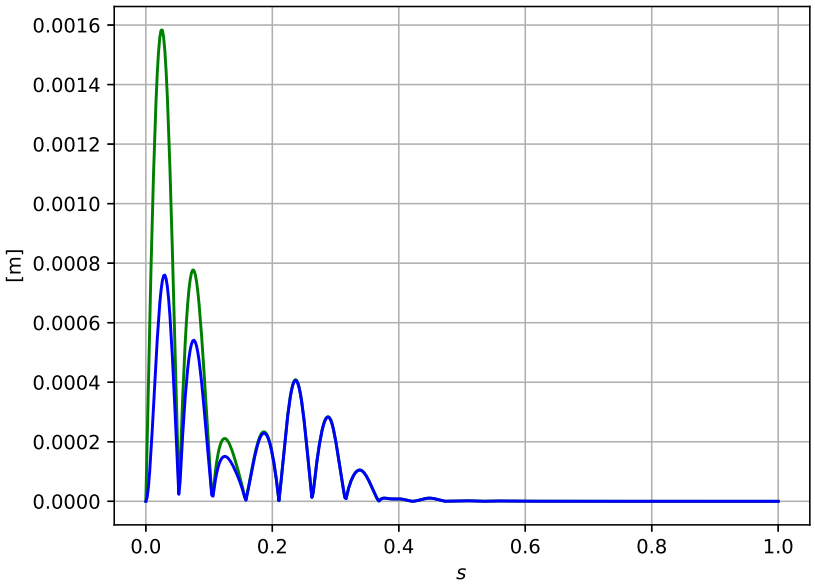

Abbildung 10: Fehler bei Bézier-Interpolation mit natürlichen und Hermite-Randbedingungen bei Kurvenparameter s.

In Abbildung 9 wurden die Bézier-Kurven mit natürlichen und mit Hermite-Randbedingungen in denselben Plot gezeichnet, wir betrachten hier den linken Intervallrand genauer. Klar zu erkennen ist die stärkere Krümmung der Splinekurve mit Hermite-Randbedingungen am Intervallrand (oben im Bild).

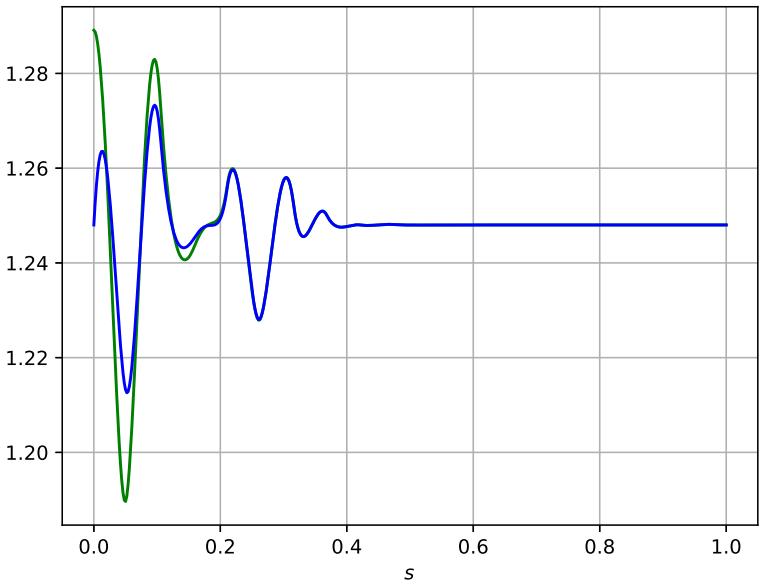

Abbildung 11: Norm der ersten Ableitung des interpolierenden Splines mit natürlichen und Hermite-Randbedingungen bei Kurvenparameter s ($|s'_f(s)|$).

In Abbildung 11 sehen wir den Betrag der Ableitung von $s_f(s)$ nach dem Kurvenparameter

s. Diese Funktion wird bei der späteren Berechnung des Wegintegrals erster Art numerisch integriert.

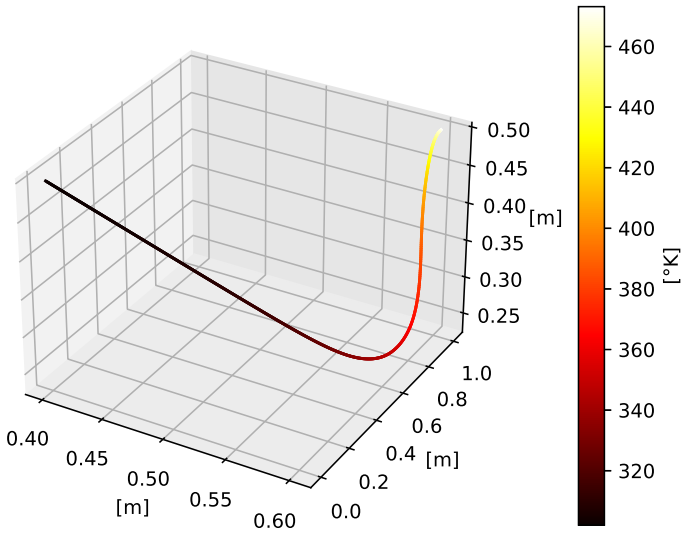

Abbildung 12: Durch Bézier-Kurven mit Hermite-Randbedingungen interpolierter Faden mit durch Bézier-Kurven mit natürlichen Randbedingungen interpolierter Temperatur.

Die interpolierende Bézier-Kurve wurde in Abbildung 12 mit der interpolierten Temperatur und in Abbildung 13 mit der interpolierten Dicke eingefärbt. Erinnern wir uns an das Anwendungsbeispiel, so lässt sich das im Bild rechts oben liegende Ende der Kurve mit der Austrittsstelle des Fadens aus der Düse identifizieren. Der Faden verlässt die Düse also mit einer höheren Temperatur, und erkaltet dann durch Temperaturabgabe an die Umgebung. Außerdem verliert der Faden nach Austritt aus der Düse an Dicke.

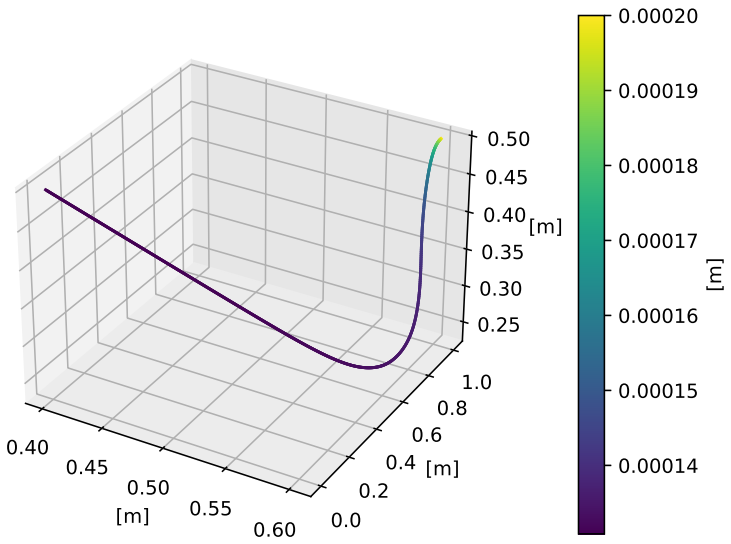

Abbildung 13: Durch Bézier-Kurven mit Hermite-Randbedingungen interpolierter Faden mit durch Bézier-Kurven mit natürlichen Randbedingungen interpolierter Fadendicke.

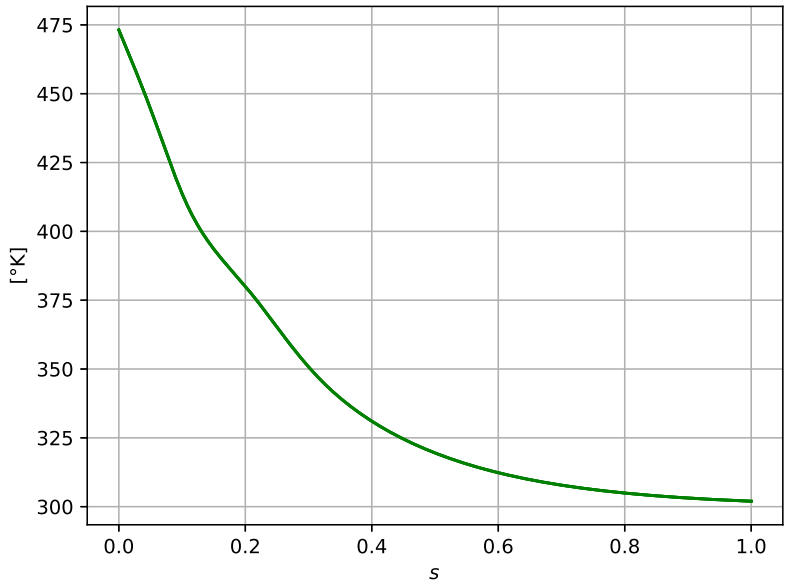

Abbildung 14: Durch Bézier-Kurven mit natürlichen Randbedingungen interpolierte Temperatur bei Kurvenparameter s.

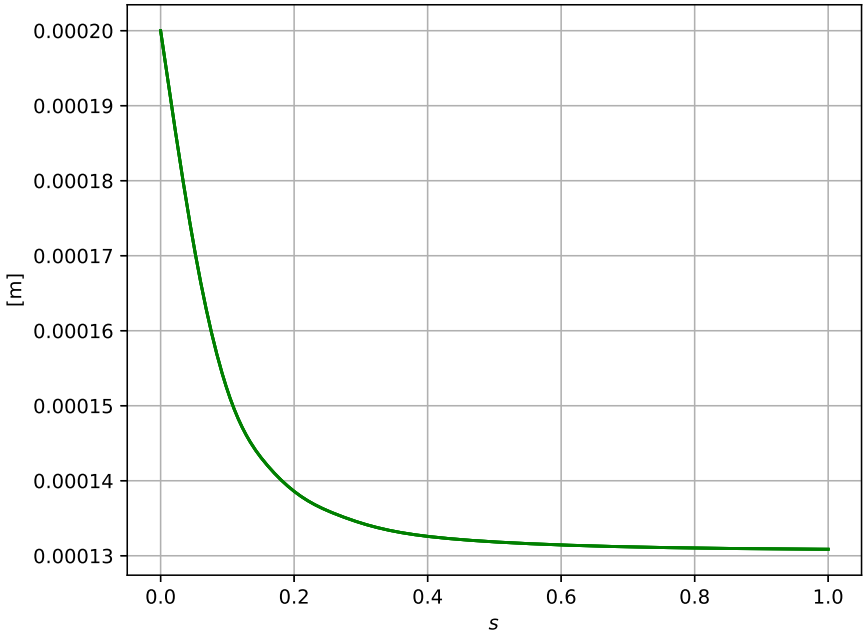

Abbildung 15: Durch Bézier-Kurven mit natürlichen Randbedingungen interpolierte Dicke bei Kurvenparameter s.

3.5. Zwischenfazit

Wir haben also Gleichungssysteme für die Berechnung interpolatorischer Bézier-Kurven mit wahlweise natürlichen oder Hermite-Randbedingungen hergeleitet, die eindeutig lösbar sind. Mithilfe dieser Gleichungssysteme können wir nun Mengen von Punkten im $\mathbb{R}^3$ oder in $\mathbb{R}$ mit Bézier-Kurven interpolieren. Nun haben wir einen Schritt in Richtung der Lösung des ursprünglich gestellten Modellierungsproblems getan, da wir diese Splines numerisch effizient auswerten können und auf analytischem Wege Schnittstellen etwa mit einem Würfelgitter bestimmen können. Bevor wir damit fortfahren, führen wir noch einige Fehlerabschätzungen durch und beweisen Minimaleigenschaften von Splinekurven mit gewissen Randbedingungen.

4. Fehlerabschätzungen und Minimaleigenschaften

Wir werden uns nun mit einigen Fehlerabschätzungen und Minimaleigenschaften interpolierender Splines mit natürlichen beziehungsweise Hermite-Randbedingungen beschäftigen.

4.1. Natürliche Randbedingungen

Als Korollar zu Satz 2.7 erhält man die folgende Minimaleigenschaft:

Satz 4.1. *Sei $f \in \mathrm{C}^2[a,b]$ und sei $s_f \in \mathrm{S}_3(\Delta_n)$ der interpolierende Spline mit natürlichen oder mit Hermite-Randbedingungen. „Interpolierend“ bedeutet hier, dass $f(x_i) = s_f(x_i)$ für $i = 0, \ldots, n$, wenn $\Delta_n = \{x_0, \ldots, x_n\}$. Dann hat s_f unter allen interpolierenden C^2-Funktionen die geringste Krümmung, also*

$$\|s''\|_2 \leq \|h''\|_2, \qquad \forall h \in \{g \in \mathrm{C}^2[a,b] : g(x_j) = f(x_j) \text{ für } j = 0, \ldots, n\}.$$

Beweis. Diese Aussage ergibt sich aus Satz 2.7 mit der Rechnung

$$\|s''\|_2^2 = \|f''\|_2^2 - \|f'' - s''\|_2^2 \leq \|f''\|_2^2. \qquad \square$$

In [7, S. 30 ff] finden sich Fehlerabschätzungen für interpolierende kubische Splines. Zunächst beweisen wir dafür das folgende Resultat:

Satz 4.2. *Zu einer gegebenen Funktion $f \in \mathrm{C}^4[a,b]$ mit $f''(a) = f''(b) = 0$ bezeichne $s_f \in \mathrm{S}_3(\Delta_n)$ den interpolierenden kubischen Spline mit natürlichen Randbedingungen. Dann gilt*

$$\max_{j=1}^{n-1} |s_f''(x_j) - f''(x_j)| \leq \frac{3}{4} \|f^{(4)}\|_\infty h_{\max}^2,$$

wobei die Bezeichnung $h_{\max} := \max_{j=0}^{n-1} h_j = \max_{j=0}^{n-1} (x_{j+1} - x_j)$ verwendet wird.

Beweis. Wir betrachten dazu zunächst das Gleichungssystem der Momente bei natürlichen Randbedingungen aus Lemma 2.5. Mit der Schreibweise

$$M_j := s_f''(x_j), \quad j = 0, \ldots, n$$

ergibt sich dieses nach leichter Modifikation in Form von Multiplikation der jeweils j-ten Zeile von B und g mit $\frac{1}{3(h_{j-1}+h_j)}$ als

$$B \begin{pmatrix} M_1 \\ \vdots \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} \hat{g}_1 \\ \vdots \\ \hat{g}_{n-1} \end{pmatrix}, \tag{8}$$

wobei $\hat{g}_j = 2\frac{f_{j+1}-f_j}{h_j(h_{j-1}+h_j)} - 2\frac{f_j - f_{j-1}}{h_{j-1}(h_{j-1}+h_j)}$ für $j = 1, \ldots, n-1$ ist und die Matrix $B \in \mathbb{R}^{(n-1)\times(n-1)}$

die Form

$$B = \frac{1}{3}\begin{pmatrix} 2 & \nu_0 & & & \\ \xi_1 & 2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & 2 & \mu_{n-3} \\ & & & \xi_{n-2} & 2 \end{pmatrix},$$

mit $h_j = x_{j+1} - x_j$, $\nu_j = \frac{h_{j+1}}{h_j + h_{j+1}}$ und $\xi_j = 1 - \nu_j$ für $j = 1, \ldots, n-1$ hat. Wir wollen nun die Matrix B und den Vektor $\hat{g} = (g_1, \ldots, g_{N-1})$ aus Gleichung (8) genauer untersuchen. Durch Taylorentwicklung der zweiten Ableitung f'' um x_j erhält man die Darstellungen

$$f''(x_{j-1}) = f''(x_j) - h_{j-1} f^{(3)}(x_j) + \frac{h_{j-1}^2}{2} f^{(4)}(\xi_j), \tag{9}$$

$$f''(x_{j+1}) = f''(x_j) + h_j f^{(3)}(x_j) + \frac{h_j^2}{2} f^{(4)}(\hat{\xi}_j), \tag{10}$$

mit Zwischenstellen $\xi_j, \hat{\xi}_j$. Durch Addition von (9) multipliziert mit $\frac{h_{j-1}}{3(h_{j-1}+h_j)}$ und (10) multipliziert mit $\frac{h_j}{3(h_{j-1}+h_j)}$ und anschließendem Umstellen erhält man eine Approximation an die zweite Ableitung $f''(x_j)$:

$$\begin{aligned} f''(x_j) + R_j + \delta_j &= \frac{h_{j-1}}{3(h_{j-1}+h_j)} f''(x_{j-1}) + \frac{2}{3} f''(x_j) + \frac{h_j}{3(h_{j-1}+h_j)} f''(x_{j+1}), \\ R_j &:= \frac{1}{3}(h_j - h_{j-1}) f^{(3)}(x_j), \\ \delta_j &:= \frac{1}{6(h_{j-1}+h_j)} \left(h_{j-1}^3 f^{(4)}(\xi_j) + h_j^3 f^{(4)}(\hat{\xi}_j) \right), \quad j = 1, \ldots, n-1. \end{aligned}$$

In Matrixschreibweise ergibt sich dann

$$B \begin{pmatrix} f''(x_1) \\ \vdots \\ f''(x_{n-1}) \end{pmatrix} = \begin{pmatrix} f''(x_1) \\ \vdots \\ f''(x_{n-1}) \end{pmatrix} + \begin{pmatrix} R_1 \\ \vdots \\ R_{n-1} \end{pmatrix} + \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_{n-1} \end{pmatrix}. \tag{11}$$

Weiterhin ergibt eine Taylorentwicklung von f um x_j die Darstellungen

$$f(x_{j+1}) = f(x_j) + h_j f'(x_j) + \frac{h_j^2}{2} f''(x_j) + \frac{h_j^3}{6} f^{(3)}(x_j) + \frac{h_j^4}{24} f^{(4)}(\eta_j), \tag{12}$$

$$f(x_{j-1}) = f(x_j) - h_{j-1} f'(x_j) + \frac{h_{j-1}^2}{2} f''(x_j) - \frac{h_{j-1}^3}{6} f^{(3)}(x_j) + \frac{h_{j-1}^4}{24} f^{(4)}(\hat{\eta}_j), \tag{13}$$

mit Zwischenstellen $\eta_j, \hat{\eta}_j \in [a, b]$. Durch Umstellen von (12) im Produkt mit $\frac{2}{h_j}$ und von (13) im Produkt mit $\frac{2}{h_{j-1}}$ und anschließender Auflösung nach $f(x_{j-1}), f(x_j)$ und $f(x_{j+1})$ erhält man

die Gleichungen

$$2\frac{f(x_{j+1})-f(x_j)}{h_j} = 2f'(x_j) + h_j f''(x_j) + \frac{h_j^2}{3}f^{(3)}(x_j) + \frac{h_j^3}{12}f^{(4)}(\eta_j),$$
$$-2\frac{f(x_j)-f(x_{j-1})}{h_{j-1}} = -2f'(x_j) + h_{j-1} f''(x_j) - \frac{h_{j-1}^2}{3}f^{(3)}(x_j) + \frac{h_{j-1}^3}{12}f^{(4)}(\hat{\eta}_j).$$

Addieren wir diese beiden Gleichungen nun und multiplizieren anschließend mit $\frac{1}{h_{j-1}+h_j}$, so erhalten wir folgende Approximation für $f''(x_j)$:

$$f''(x_j) + R_j + \hat{\delta}_j = \underbrace{2\frac{f_{j+1}-f_j}{h_j(h_{j-1}+h_j)} - 2\frac{f_j - f_{j-1}}{h_{j-1}(h_{j-1}+h_j)}}_{=\hat{g}_j}, \quad j = 1, \ldots, n-1,$$
$$\hat{\delta}_j := \frac{1}{12(h_{j-1}+h_j)}\left(h_j^3 f^{(4)}(\eta_j) + h_{j-1}^3 f^{(4)}(\hat{\eta}_j)\right),$$

beziehungsweise in Matrix-Vektorschreibweise

$$\begin{pmatrix} f''(x_1) \\ \vdots \\ f''(x_{n-1}) \end{pmatrix} = \begin{pmatrix} \hat{g}_1 \\ \vdots \\ \hat{g}_{n-1} \end{pmatrix} - \begin{pmatrix} R_1 \\ \vdots \\ R_{n-1} \end{pmatrix} - \begin{pmatrix} \hat{\delta}_1 \\ \vdots \\ \hat{\delta}_{n-1} \end{pmatrix}. \tag{14}$$

Nun verwenden wir die eben hergeleitete Identität (14) für die Identität (11). Es ergibt sich mit der Bezeichnung

$$M_j^{(f)} := f''(x_j), \quad j = 1, \ldots, n-1$$

und den Kurzschreibweisen $M^{(f)} = (M_1^{(f)}, \ldots, M_{n-1}^{(f)})$, $\hat{g} = (\hat{g}_1, \ldots, \hat{g}_{N-1})$, $R = (R_1, \ldots, R_{n-1})$, $\delta = (\delta_1, \ldots, \delta_{n-1})$ und $\hat{\delta} = (\hat{\delta}_1, \ldots, \hat{\delta}_{n-1})$ also die Identität

$$BM^{(f)} = M^{(f)} + R + \delta = (\hat{g} - R - \hat{\delta}) + R + \delta = \hat{g} + \delta - \hat{\delta}. \tag{15}$$

Erinnern wir uns nun noch an das Gleichungssystem (8). Subtrahieren wir dieses von (15), so ergibt sich

$$B(M^{(f)} - M) = \delta - \hat{\delta}.$$

Die Matrix B ist als strikt diagonaldominante Matrix regulär. Um $M^{(f)} - M$ abzuschätzen betrachten wir

$$|b_{jj}| - \sum_{\substack{k=1 \\ k\neq j}} b_{jk} \geq \frac{2}{3} - \frac{\xi_j}{3} - \frac{\nu_j}{3} = \frac{1}{3}(2 - (1-\nu_j) - \nu_j) = \frac{1}{3}, \quad j = 1, \ldots, n-1,$$

wobei b_{jk} das Element der Matrix B in Zeile j und in Spalte k bezeichnet. Mit dem Lemma 2.3

erhalten wir nun die Abschätzung

$$\begin{aligned}\max_{j=0}^{N} |M_j^{(f)} - M_j| &\leq 3\max\{|\delta_1| + |\hat{\delta}_1|, \ldots, |\delta_{n-1}| + |\hat{\delta}_{n-1}|\} \\ &\overset{(\clubsuit)}{\leq} \frac{3}{4} h_{\max}^2 \|f^{(4)}\|_\infty,\end{aligned}$$

wobei wir in (♣) wie folgt abschätzen:

$$\begin{aligned}|\delta_j| + |\hat{\delta}_j| &\leq \frac{1}{6(h_{j-1}+h_j)} \left(h_{j-1}^3 \|f^{(4)}\|_\infty + h_j^3 \|f^{(4)}\|_\infty\right) \\ &\quad + \frac{1}{12(h_{j-1}+h_j)} \left(h_j^3 \|f^{(4)}\|_\infty + h_{j-1}^3 \|f^{(4)}\|_\infty\right) \\ &= \frac{1}{4} \frac{h_{j-1}^3 + h_j^3}{h_{j-1}+h_j} \|f^{(4)}\|_\infty \\ &\overset{(\spadesuit)}{\leq} \frac{1}{4} h_{\max}^2 \|f^{(4)}\|_\infty.\end{aligned}$$

(Bei (♠) nutzen wir die Faktorisierung $a^3 + b^3 = (a+b)(a^2 - ab + b^2)$).

Damit ist die Abschätzung nachgewiesen. □

Auch der nachfolgende Satz wurde aus [7] übernommen.

Satz 4.3. *Sei $f \in \mathrm{C}^4[a,b]$ und sei $s_f \in S_3(\Delta_n)$ ein kubischer Spline, der f interpoliert. Weiterhin nutzen wir analog zum vorherigen Satz die Kurzschreibweisen $h_j = x_{j+1} - x_j$ für $j = 0, \ldots, n-1$ und*

$$h_{\max} := \max_{j=0}^{n-1} h_j, \qquad h_{\min} := \min_{j=0}^{n-1} h_j.$$

Falls die Ungleichung

$$\max_{j=0}^{n} |s_f''(x_j) - f''(x_j)| \leq C \|f^{(4)}\|_\infty h_{\max}^2$$

mit einer Konstanten $C > 0$ erfüllt ist, so gelten mit der Setzung $c := \frac{h_{\max}}{h_{\min}} \left(C + \frac{1}{4}\right)$ die folgenden Abschätzungen für jedes $x \in [a,b]$:

$$|s_f(x) - f(x)| \leq \left(c - \frac{1}{8} \cdot \frac{h_{\max}}{h_{\min}}\right) \|f^{(4)}\|_\infty h_{\max}^4, \tag{16}$$

$$|s_f'(x) - f'(x)| \leq 2\left(c - \frac{1}{8} \cdot \frac{h_{\max}}{h_{\min}}\right) \|f^{(4)}\|_\infty h_{\max}^3, \tag{17}$$

$$|s_f''(x) - f''(x)| \leq 2\left(c - \frac{1}{8} \cdot \frac{h_{\max}}{h_{\min}}\right) \|f^{(4)}\|_\infty h_{\max}^2, \tag{18}$$

$$|s_f^{(3)}(x) - f^{(3)}(x)| \leq 2c \|f^{(4)}\|_\infty h_{\max} \quad (x \neq x_j). \tag{19}$$

Im Vergleich zum Original bei [7] ist die Konstante c bei (16), (17) und (18) jeweils durch $c - \frac{1}{8} \cdot \frac{h_{\max}}{h_{\min}}$ ersetzt worden. Somit ist eine Verbesserung der Fehlerabschätzung erzielt worden.

Beweis. Wir weisen zunächst die Abschätzungen für die dritten Ableitungen (19) nach. Of-

fenbar entsprechen die Einschränkungen der zweiten Ableitungen $s_f''|_{[x_j,x_{j+1}]}$ Polynomen ersten Grades. Wir erhalten somit für die dritte Ableitung die folgende Darstellung:

$$s_f^{(3)}(x) = \frac{s_f''(x_{j+1}) - s_f''(x_j)}{h_j}, \text{ für } x \in (x_j, x_{j+1}),\ j = 0, \ldots, n-1. \tag{20}$$

Um eine Approximation für $f^{(3)}$ zu konstruieren, betrachten wir nun die Taylorentwicklung von f'' um $x \in [x_j, x_{j+1}]$:

$$f''(x_{j+1}) = f''(x) + (x_{j+1} - x)f^{(3)}(x) + \frac{(x_{j+1} - x)^2}{2} f^{(4)}(\alpha_j), \tag{21}$$

$$f''(x_j) = f''(x) + (x_j - x)f^{(3)}(x) + \frac{(x - x_j)^2}{2} f^{(4)}(\beta_j), \tag{22}$$

mit Zwischenstellen $\alpha_j, \beta_j \in [x_j, x_{j+1}]$. Betrachten wir die Subtraktion (21)-(22) und dividieren anschließend durch h_j, so erhalten wir durch Umstellung die Darstellung

$$f^{(3)}(x) = \frac{f''(x_{j+1}) - f''(x_j)}{h_j} - \frac{(x_{j+1} - x)^2}{2h_j} f^{(4)}(\alpha_j) + \frac{(x - x_j)^2}{2h_j} f^{(4)}(\beta_j). \tag{23}$$

Durch die Bildung der Differenz (20)-(23) ergibt sich die Identität

$$\begin{aligned} &s_f^{(3)}(x) - f^{(3)}(x) \\ &= \frac{s_f''(x_{j+1}) - f''(x_{j+1})}{h_j} - \frac{s_f''(x_j) - f''(x_j)}{h_j} + \frac{(x_{j+1} - x)^2 f^{(4)}(\alpha_j) - (x - x_j)^2 f^{(4)}(\beta_j)}{2h_j}. \end{aligned}$$

Erinnern wir uns an die Voraussetzungen aus dem Satz, so ergibt sich unter der Betrachtung der folgenden Abschätzung

$$\begin{aligned} |(x_{j+1} - x)^2 - (x - x_j)^2| &\leq (x_{j+1} - x)^2 + (x - x_j)^2 \\ &= x_{j+1}^2 - 2x_{j+1}x_j + x_j^2 - 2x_{j+1}x + 2x_{j+1}x_j + 2x^2 - 2xx_j \\ &= (x_{j+1} - x_j)^2 - 2(x_{j+1} - x)(x - x_j) \\ &\leq (x_{j+1} - x_j)^2 \leq h_{\max}^2 \end{aligned}$$

schlussendlich das Resultat

$$\begin{aligned} |s_f^{(3)}(x) - f^{(3)}(x)| &\leq \|f^{(4)}\|_\infty \frac{1}{h_{\min}} \left(Ch_{\max}^2 + Ch_{\max}^2 + \frac{h_{\max}^2}{2} \right) \\ &\leq \underbrace{\frac{h_{\max}}{h_{\min}} \left(2C + \frac{1}{2} \right)}_{=2c} \|f^{(4)}\|_\infty h_{\max}. \end{aligned}$$

Die Fehlerabschätzung (19) ist hiermit nachgewiesen.

Die verbleibenden Fehlerabschätzungen erhalten wir aus (19) durch Integration. Zur Fehlerabschätzung der zweiten Ableitungen (18) bezeichne x_j den zu einem beliebigen $x \in [a, b]$

nächstgelegenen Knoten, womit offenbar $|x - x_j| \leq \frac{h_{\max}}{2}$ erfüllt ist. Mit dem Hauptsatz der Differential- und Integralrechnung erhalten wir

$$s_f''(x) - f''(x) = s_f''(x_j) - f''(x_j) + \int_{x_j}^{x} s_f^{(3)}(y) - f^{(3)}(y)\,\mathrm{d}y.$$

Abschätzend fahren wir wie folgt fort:

$$\begin{aligned}
|s_f''(x) - f''(x)| &\leq C\|f^{(4)}\|_\infty h_{\max}^2 + 2c\|f^{(4)}\||x - x_j|h_{\max} \\
&\leq (C + c)\|f^{(4)}\|_\infty h_{\max}^2 \\
&\leq 2\left(c - \frac{1}{8}\frac{h_{\max}}{h_{\min}}\right)\|f^{(4)}\|_\infty h_{\max}^2,
\end{aligned}$$

wobei noch $C \leq c - \frac{1}{4}\frac{h_{\max}}{h_{\min}}$ eingeht. Damit ist die Abschätzung (18) für den Fehler bei den zweiten Ableitungen nachgewiesen. Zur Abschätzung der ersten Ableitungen (17) erkennen wir, dass die Stützstellen x_j für $j = 0, \ldots, N$ Nullstellen von $s - f$ sind. Mit dem Satz von Rolle erhalten wir für $j = 1, \ldots, N$ die Existenz eines $y_j \in [x_{j-1}, x_j]$ mit $(s_f' - f')(y_j) = 0$. Wir wählen zu einem Punkt $x \in [a, b]$ die nächstgelegene Nullstelle von $s_f' - f'$ und nennen sie y_j. Es gilt dann offenbar $|x - y_j| \leq h_{\max}$. Mit dem Hauptsatz der Differential- und Integralrechnung erhalten wir nun

$$\begin{aligned}
|s_f'(x) - f'(x)| &= |s_f'(x) - f'(x) - (s_f'(y_j) - f'(y_j))| \\
&= \left|\int_{y_j}^{x} s_f''(y) - f''(y)\,\mathrm{d}y\right| \\
&\leq 2\left(c - \frac{1}{8}\cdot\frac{h_{\max}}{h_{\min}}\right)\|f^{(4)}\|_\infty h_{\max}^2|x - y_j| \\
&\leq 2\left(c - \frac{1}{8}\cdot\frac{h_{\max}}{h_{\min}}\right)\|f^{(4)}\|h_{\max}^3.
\end{aligned}$$

Damit haben wir auch die Abschätzung (17) für die ersten Ableitungen bewiesen. Schlussendlich bleibt noch die Abschätzung (16) für $|s - f|$ zu zeigen. Für ein $x \in [a, b]$ und den dazugehörigen nächstgelegenen Knoten x_j (mit $|x - x_j| \leq \frac{h_{\max}}{2}$) erhalten wir

$$\begin{aligned}
|s_f(x) - f(x)| &= |s_f(x) - f(x) - (s_f(x_j) - f(x_j))| \\
&= \left|\int_{x_j}^{x} s_f'(y) - f'(y)\right| \\
&\leq 2\left(c - \frac{1}{8}\cdot\frac{h_{\max}}{h_{\min}}\right)\|f^{(4)}\|_\infty h_{\max}^3|x - x_j| \\
&\leq \left(c - \frac{1}{8}\cdot\frac{h_{\max}}{h_{\min}}\right)\|f^{(4)}\|_\infty h_{\max}^4,
\end{aligned}$$

womit die erste Fehlerabschätzung (16) ebenfalls nachgewiesen ist. □

Als Korollar der beiden vorherigen Sätze erhalten wir die folgende Aussage.

Satz 4.4. *Sei $f \in \mathrm{C}^4[a,b]$ mit $f''(a) = f''(b)$ und $s_f \in \mathrm{S}_3(\Delta_n)$ sei der interpolierende kubische Spline mit natürlichen Randbedingungen. Dann gilt*

$$\begin{aligned}
\|s_f - f\|_\infty &\leq \frac{7}{8}\frac{h_{\max}}{h_{\min}}\|f^{(4)}\|_\infty h_{\max}^4 \\
\|s_f' - f'\|_\infty &\leq \frac{7}{4}\frac{h_{\max}}{h_{\min}}\|f^{(4)}\|_\infty h_{\max}^3 \\
\|s_f'' - f''\|_\infty &\leq \frac{7}{4}\frac{h_{\max}}{h_{\min}}\|f^{(4)}\|_\infty h_{\max}^2 \\
|s_f^{(3)}(x) - f^{(3)}(x)| &\leq 2\|f^{(4)}\|_\infty h_{\max} \qquad (x \neq x_j).
\end{aligned}$$

4.2. Hermite-Randbedingungen

In [12, S. 174 ff] finden wir Fehlerabschätzungen zu interpolierenden kubischen Splines mit Hermite-Randbedingungen.

Satz 4.5. *Sei $\Delta_n := \{a = x_0 < \cdots < x_n = b\}$ eine Zerlegung von $[a,b]$, h_j bezeichne die Schrittweite $h_j := x_{j+1} - x_j$ für $j = 0, \ldots, n-1$ und $h_{\max} := \max_{j=0}^{n-1} h_j$. Liegt eine Funktion $f \in \mathrm{C}^4[a,b]$ vor ($\mathrm{C}^2[a,b]$ genügt für (i)−(iii)), so sei $s_f \in S_3(\Delta_n)$ der die Funktion f interpolierende kubische Spline, der den Hermite-Randbedingungen $s_f'(a) = f'(a)$, $s_f'(b) = f'(b)$ genügt. Weiterhin sei $s^f \in S_1(\Delta_n)$ derjenige lineare Spline, der die Funktion f interpoliert. Dann gilt*

(i) $\|f - s^f\|_\infty \leq \frac{1}{8}h_{\max}^2\|f''\|_\infty$

(ii) $\|s_f''\|_\infty \leq 3\|f''\|_\infty$.

(iii) $\|f'' - s_f''\|_\infty \leq 4\|f'' - s\|_\infty$ *für alle* $s \in S_1(\Delta_n)$.

(iv) $\|f'' - s_f''\|_\infty \leq \frac{1}{2}h_{\max}^2\|f^{(4)}\|_\infty$.

(v) $\|f' - s_f'\|_\infty \leq \frac{1}{2}h_{\max}^3\|f^{(4)}\|_\infty$.

(vi) $\|f - s_f\|_\infty \leq \frac{1}{16}h_{\max}^4\|f^{(4)}\|_\infty$.

Beweis. (i) Wir benötigen hierfür Satz 2.8. Wenden wir diesen Satz auf unseren speziellen Fall ($n = 1$) an, so erhalten wir zunächst für $x \in (x_j, x_{j+1})$:

$$\begin{aligned}
|f(x) - s^f(x)| &= \left|\frac{f^{(1+1)}(\xi)}{2}\prod_{k=0}^{1}(x - x_{k+j})\right| \qquad \text{mit } \xi \in [a,b] \\
&\overset{(*)}{\leq} \left|\frac{f''(\xi)}{2}\left(x_j - \frac{x_j + x_{j+1}}{2}\right)\left(x_{j+1} - \frac{x_j + x_{j+1}}{2}\right)\right| \\
&= \left|\frac{f''(\xi)}{2}\left(\frac{-h_j}{2}\right)\left(\frac{h_j}{2}\right)\right| \\
&\leq \left|\frac{f''(\xi)}{2}\cdot\frac{h_{\max}^2}{4}\right| \leq \frac{h_{\max}^2}{8}\|f''\|_\infty,
\end{aligned}$$

wobei sich $(*)$ dadurch ergibt, dass quadratische Polynome der Form $(x-a)(x-b)$ $(a<b)$ ihr betragliches Maximum auf $[a,b]$ bei $\frac{a+b}{2}$ annehmen. Durch Anwendung dieses Resultates auf alle Intervalle (x_j, x_{j+1}) $(j=0,\dots,n-1)$ ergibt sich die Aussage.

(ii) Für die Berechnung der Momente $M_j := s_f''(x_j)$ für $j=0,\dots,n$ der kubischen Spline-Interpolierenden s_f bei Hermite-Randbedingungen nutzt man folgendes Gleichungssystem, welches wir aus Lemma 2.5 kennen:

$$\underbrace{\begin{pmatrix} 2 & \lambda_0 & & & \\ \mu_1 & 2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & 2 & \lambda_{n-1} \\ & & & \mu_n & 2 \end{pmatrix}}_{=\hat{A}} \underbrace{\begin{pmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{pmatrix}}_{=M} = \underbrace{\begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}}_{=d}$$

(siehe [12, S. 168]), wobei die in $\hat{A}$ vorkommenden Koeffizienten wie folgt definiert sind:

$$\lambda_0 := 1, \quad \lambda_j := \frac{h_j}{h_{j-1}+h_j}, \quad \mu_j := 1-\lambda_j \quad \text{für } j=1,\dots,n-1, \quad \mu_n := 1.$$

Die Einträge des Vektors d erklären wir als

$$\begin{aligned} d_0 &:= \frac{6}{h_0}\left(\frac{f(a+h_0)-f(a)}{h_0} - f'(a)\right), \\ d_j &:= \frac{6}{h_{j-1}+h_j}\left(\frac{f(x_j+h_j)-f(x_j)}{h_j} - \frac{f(x_j)-f(x_j-h_{j-1})}{h_{j-1}}\right), \quad j=1,\dots,n-1, \\ d_n &:= \frac{6}{h_{n-1}}\left(f'(b) - \frac{f(b)-f(b-h_{n-1})}{h_{n-1}}\right). \end{aligned}$$

Mithilfe einer Taylor-Entwicklung zeigen wir nun die Existenz eines $\xi_j \in [a,b]$, sodass $d_j = 3f''(\xi_j)$ für $j=0,\dots,n$.

Wir rechnen zunächst für $j = 1, \dots, n-1$:

$$\begin{aligned}
d_j &= \frac{6}{h_{j-1}+h_j}\left[\frac{f(x_j+h_j)-f(x_j)}{h_j} - \frac{f(x_j)-f(x_j-h_{j-1})}{h_{j-1}}\right] \\
&\overset{(*)}{=} \frac{6}{h_{j-1}+h_j}\left[\frac{1}{h_j}\left(f(x_j)+f'(x_j)h_j+\frac{f''(\hat{\xi}_j)}{2}h_j^2 - f(x_j)\right)\right. \\
&\qquad \left. -\frac{1}{h_{j-1}}\left(f(x_j)-f(x_j)+f'(x_j)h_{j-1}-\frac{f''(\tilde{\xi}_j)}{2}h_{j-1}^2\right)\right] \\
&= \frac{6}{h_{j-1}+h_j}\left[f'(x_j)+\frac{f''(\hat{\xi}_j)}{2}h_j - f'(x_j)+\frac{f''(\tilde{\xi}_j)}{2}h_{j-1}\right] \\
&= \frac{3}{h_{j-1}+h_j}\left(f''(\hat{\xi}_j)h_j + f''(\tilde{\xi}_j)h_{j-1}\right) \\
&\overset{(**)}{=} 3f''(\xi_j),
\end{aligned}$$

wobei bei $(*)$ die Differenzierbarkeitseigenschaften ausgenutzt wurden, indem eine Taylor-Entwicklung durchgeführt wird und bei $(**)$ gewichtet gemittelt wird. Die Aussage bleibt nun noch für $j = 0$ und $j = n$ zu zeigen. Dafür rechnen wir

$$\begin{aligned}
d_0 &= \frac{6}{h_0}\left[\frac{f(a+h_0)-f(a)}{h_0} - f'(a)\right] \\
&= \frac{6}{h_0}\left[\frac{1}{h_0}\left(f(a)+f'(a)h_0+\frac{f''(\xi_0)}{2}h_0^2 - f(a)\right) - f'(a)\right] \\
&= \frac{6}{h_0}\left[\frac{f''(\xi_0)}{2}h_0\right] \\
&= 3f''(\xi_0),
\end{aligned}$$

bei $j = n$ gehen wir analog vor.

Für den nächsten Schritt benötigen wir $\|\hat{A}^{-1}\|_\infty \leq 1$. Dies zeigen wir mithilfe von Lemma 2.9. Wir können die Matrix $\hat{A}$ umschreiben, um die Voraussetzungen nachzuweisen:

$$\hat{A} = 2I + \hat{B},$$

wobei

$$\hat{B} = \begin{pmatrix} 0 & \lambda_0 & & & \\ \mu_1 & 0 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & 0 & \lambda_{n-1} \\ & & & \mu_n & 0 \end{pmatrix}.$$

Mit den Setzungen $\mu_0 = 1 - \lambda_0 = 0$ und $\lambda_n = 1 - \mu_n = 0$ gilt offenbar

$$\|\hat{B}\|_\infty = \max_{j=0}^{n} |\lambda_j| + |\mu_j| = \max_{j=0}^{n} \lambda_j + 1 - \lambda_j = 1.$$

Damit folgern wir $\|(1/2)\hat{B}\|_\infty = 1/2$. Nun können wir Lemma 2.9 auf $(1/2)\hat{A} = I + (1/2)\hat{B}$ anwenden und erhalten

$$\left\| \left(\frac{1}{2} A \right)^{-1} \right\|_\infty = \left\| \left(I + \frac{1}{2} \hat{B} \right)^{-1} \right\|_\infty \leq \frac{1}{1 - \|(1/2)\hat{B}\|_\infty} = 2,$$

also $\|A^{-1}\|_\infty \leq 1$.

Es folgt hiermit

$$\begin{aligned} \|s_f''\|_\infty &= \max_{j=0}^{n} |s_f''(x_j)| = \max_{j=0}^{n} |M_j| = \|M\|_\infty \\ &= \|\hat{A}^{-1} d\|_\infty \leq \|\hat{A}^{-1}\|_\infty \|d\|_\infty \\ &\leq \|d\|_\infty \leq 3\|f''\|_\infty. \end{aligned}$$

(iii) Zu dem linearen Spline $s \in S_1(\Delta_n)$ definiere man $u : [a, b] \to \mathbb{R}$ durch

$$u(x) := \int_a^x (x - t) s(t) \, \mathrm{d}t.$$

Dann ist aufgrund von

$$\begin{aligned} \frac{\mathrm{d}^2}{\mathrm{d}x^2} u(x) &= \frac{\mathrm{d}^2}{\mathrm{d}x^2} \int_a^x (x - t) s(t) \, \mathrm{d}t \\ &= \frac{\mathrm{d}^2}{\mathrm{d}x^2} \left(x \int_a^x s(t) \, \mathrm{d}t - \int_a^x t s(t) \, \mathrm{d}t \right) \\ &= \frac{\mathrm{d}}{\mathrm{d}x} \left(x s(x) + \int_a^x s(t) \, \mathrm{d}t - x s(x) \right) \\ &= \frac{\mathrm{d}}{\mathrm{d}x} \int_a^x s(t) \, \mathrm{d}t \\ &= s(x) \end{aligned}$$

$u'' = s$ und wegen $u \in \mathrm{C}^2[a, b]$ ist auch $u \in S_3(\Delta_n)$. Natürlich ist $s_u = u$ und $s_{f-u} = s_f - s_u$. Hiermit erhalten wir unter Anwendung von (ii):

$$\|f'' - s_f''\|_\infty = \|(f-u)'' - s_{f-u}''\|_\infty \leq \|(f-u)''\|_\infty + \|s_{f-u}''\|_\infty \leq 4\|f'' - u''\|_\infty = 4\|f'' - s\|_\infty,$$

woraus sich (iii) ergibt.

(iv) Setzt man in $\|f'' - s_f''\|_\infty \leq 4\|f'' - s\|_\infty$ für s den interpolierenden linearen Spline $s^{f''}$ ein, so erhalten wir mit (i) die Behauptung (iv).

(v) Wegen $(f - s_f)(x_j) = 0$, $j = 0, \ldots, n$ und des Satzes von Rolle existieren $\xi_j \in (x_j, x_{j+1})$ mit $(f - s_f)'(\xi_j) = 0$, $j = 0, \ldots, n-1$. Nach Definition der maximalen Schrittweite $h_{\max}$ gibt es ferner zu jedem $x \in [a, b]$ einen Zwischenwert $\xi_j = \xi_j(x)$ mit $|\xi_j(x) - x| \leq h_{\max}$. Für $x \in [a, b]$ erhält man daher mit dem vierten Teil des Satzes

$$|f'(x) - s_f'(x)| = \left| \int_{\xi_j(x)}^{x} [f''(t) - s_f''(t)] \, \mathrm{d}t \right| \leq h_{\max} \|f'' - s_f''\|_\infty \leq \frac{h_{\max}^3}{2} \|f^{(4)}\|_\infty,$$

und hieraus (v).

(vi) Wegen $(f - s_f)(x_j) = 0$ für $j = 0, \ldots, n-1$ verschwindet der $f - s_f$ interpolierende lineare Spline an diesen Stützstellen. Aus (i) und (iv) erhält man daher mit

$$\|f - s_f\|_\infty \leq \frac{h_{\max}^2}{8} \|(f - s_f)''\|_\infty \leq \frac{h_{\max}^4}{16} \|f^{(4)}\|_\infty$$

auch (vi). □

4.3. Zwischenfazit

Wir haben also nun einige Fehlerabschätzungen für die Splineinterpolation mit wahlweise natürlichen oder Hermite-Randbedingungen nachgewiesen. Dass die Fehler jeweils von der Ordnung $\mathcal{O}(h_{\max}^4)$ sind, bestätigt uns in der Wahl von Splines als interpolierende Funktion.

5. Verbleibende numerische Ergebnisse

5.1. Ergebnisse am Würfelgitter

Am achsenparallelen Würfelgitter mit beliebiger Gitterbreite, welches durch den Ursprung geht, wurde das Problem der Findung der Gitterschnittpunkte y_j analytisch gelöst. Weiterhin wird numerisch die Länge der Splinekurve in jedem Gitterelement sowie die mittlere Temperatur und Fadendicke in jedem Gitterelement berechnet.

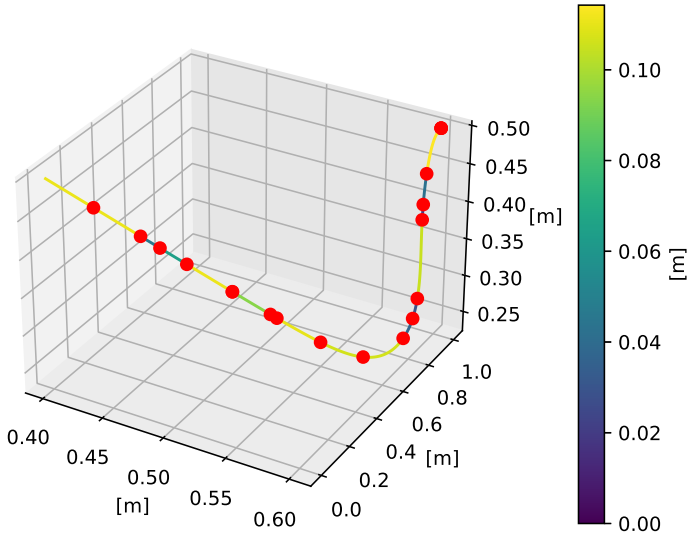

Abbildung 16: Die Länge der Kurve in den jeweiligen Gitterelementen und die Schnittpunkte mit den Gitterflächen wurden farblich markiert. Die Bedeutung der rot eingezeichneten Punkte wird aus Abbildung 6 klar.

5.2. Bestimmung des enthaltenden Gitterelements am Parallelepipedgitter

Die bisherigen Ergebnisse wurden an einem Würfelgitter erzielt. Allerdings ist dies nicht der allgemeinste Fall. Räumliche Kurven sind nun auf Parallelepipedgittern zu betrachten, die allerdings von der mathematischen Beschreibung deutlich aufwändiger sind. Daher sind zunächst einige allgemeine Betrachtungen angebracht.

Wir konstruieren uns ein Gitter aus Parallelepipeden. Dafür wählen wir zunächst einen Startpunkt $s_0 \in \mathbb{R}^3$ und drei linear unabhängige Vektoren $x, y, z \in \mathbb{R}^3$. Unser Gitter soll in dem von diesen Vektoren erzeugten und um s_0 translatierten Span

$$\{x \in \mathbb{R}^3 \mid x = \lambda_1 x + \lambda_2 y + \lambda_3 z + s_0,\ \lambda_j \in [0,1]\}$$

liegen. Dafür konstruieren wir uns die Gitterpunkte wie folgt: Seien $\lambda_0^{(x)} < \cdots < \lambda_n^{(x)}, \lambda_0^{(y)} <$

$\cdots < \lambda_n^{(y)}, \lambda_0^{(z)} < \cdots < \lambda_n^{(z)}$ alle aus $[0,1]$. Im Programmkode werden diese später zufällig generiert. Dann definieren wir dazu die Menge der Gitterpunkte G als

$$G := \{\tilde{x} \in \mathbb{R}^3 \mid \tilde{x} = \lambda_i^{(x)} x + \lambda_j^{(y)} y + \lambda_k^{(z)} z + s_0, \text{ wobei } 0 \leq i, j, k \leq n\},$$

wobei die Gitterelemente die entsprechend definierten Parallelepipede sind. Diese nummerieren wir mit einer dreifachen Schleife durch die Dimensionen durch. Auf diese Art und Weise kommen die Indizes in Tabelle 1 zustande.

Um nun für einen beliebigen Punkte $\tilde{x} \in \mathbb{R}^3$ das enthaltende Gitterelement zu bestimmen, setzen wir $A = (x|y|z)$. Wir erhalten den Punkt $\tilde{x}_0 := \tilde{x} - s_0$ nun als Linearkombination von x, y und z, indem wir $\lambda = A^{-1}\tilde{x}_0$ berechnen. Es ergibt sich dann $\tilde{x} = \lambda_1 x + \lambda_2 y + \lambda_3 z$. Das enthaltende Gitterelement ergibt sich nun, indem wir die so erhaltenden λ_1, λ_2 und λ_3 mit den vorher definierten $\lambda_j^{(x)}$, $\lambda_j^{(y)}$ und $\lambda_j^{(z)}$ vergleichen.

5.3. Ergebnisse am Parallelepipedgitter

An dem so gegebenen Gitter haben wir numerisch effiziente Methoden konstruiert, die für eine fein diskretisierte Splinekurve von $[0,1]$ nach $\mathbb{R}^3$ schnell berechnen können, in welchem Gitterelement die einzelnen Punkte liegen. Ist etwa der f_j enthaltende Parallelepiped G_j bereits bekannt, so überprüfen wir zur Findung des f_{j+1} enthaltenden Gitterelements zunächst, ob f_{j+1} in G_j liegt. Danach überprüfen wir alle 26 Nachbarn von G_j. Erst wenn diese Überprüfungen fehlschlagen, überprüfen wir alle anderen Parallelepipede. Für diese *Brute-Force*-Methode wurden auch schnelle Abbruchbedingungen definiert, um die Effizienz zu erhöhen.

Wir wählen nun den Vektor $s_0 = (-0,1; -0,1; -0,1)$ als Gitterstartpunkt. Die Vektoren x, y und z setzen wir auf $x = (1,5; -0,05; 0)$, $y = (0; 1,4; 0)$ und $z = (0; 0; 1,2)$ und erzeugen uns jeweils 20 $\lambda_j^{(k)}$ für jedes $k \in \{x, y, z\}$, wie oben beschrieben. Diese Werte sind so gewählt, dass die gegebenen Punkte komplett in dem durch x, y und z erzeugten und um den Gitterstartpunkt s_0 translatierten Span liegen. An einem unter diesen Setzungen generierten Gitter ergibt sich etwa das folgende Resultat, wobei in der ersten Spalte alle Indizes aufgelistet sind, deren zugehörige Gitterelemente mindestens einen Punkt der Splinekurve enthalten, und in der zweiten Spalte die Anzahl der Punkte der Splinekurve angegeben ist, die im jeweiligen Gitterelement liegen. Der Spline wurde an 500 äquidistanten Punkten in $[0,1]$ ausgewertet.

Tabelle 1: Resultat der Methoden am Parallelepipedgitter.

Gitterelementindex	**Anzahl Elemente darin**
3249	77
3269	32
3289	64
3309	6
Fortsetzung auf der nächsten Seite	

Tabelle 1 – Fortsetzung.

Gitterelementindex	**Anzahl Elemente darin**
3327	13
3328	16
3329	2
3347	2
3366	2
3367	8
3385	29
3386	49
3404	25
3405	13
3424	19
3425	7
3445	15
3465	3
3466	25
3467	11
3468	8
3469	59
3489	15

Allerdings stellt diese Tabelle noch keine Grundlage für eine numerische Integration dar, da der Betrag der ersten Ableitung von s_f nicht notwendigerweise konstant ist. Demnach müsste man den Wert von $|s'_f(x)|$ noch einfließen lassen.

5.4. Zwischenfazit

Die Gitterschnittstellen y_j konnten also am achsenparallelen Würfelgitter analytisch gefunden werden. Wenn wir nun Methoden der numerischen Integration auf die anfangs vorgestellten auftretenden Wegintegrale

$$L_j = \int_{\mathfrak{C}_{y_j, y_{j+1}}} 1 \, \mathrm{d}s$$

anwenden, haben wir das Problem am Würfelgitter gelöst. Für das Problem am Parallelepipedgitter haben wir einige nützliche Methoden entwickelt, etwa eine numerisch effiziente Methode mit der wir überprüfen können, in welchem Gitterelement eines Parallelepipedgitters ein gewisser Punkt liegt. Weiterhin haben wir darauf aufbauend eine Methode entwickelt, um Resultate wie aus Tabelle 1 zu generieren. Diese können die Grundlagen für weitere Arbeiten bilden.

6. Schluss

6.1. Schlussfolgerungen

Wir haben also die Existenz und Eindeutigkeit der kubischen Splineinterpolation mit natürlichen bzw. Hermite-Randbedingungen nachgewiesen, einige interessante Fehlerabschätzungen und Minimaleigenschaften gezeigt, und schlussendlich numerische Methoden konstruiert, mit denen das eingangs gestellte Problem am achsenparallelen Würfelgitter vollständig gelöst werden kann. Außerdem haben wir Lösungsansätze am Parallelepipedgitter konstruiert.

6.2. Offene Probleme / Ausblick

Einige Problemstellungen bleiben noch offen. Eine Grundlage für weitere Arbeiten ist etwa durch die Problemstellung der Entwicklung von Methoden für das Finden der Schnittstellen von kubischen Splines mit einem Parallelepipedgitter gegeben. Außerdem wäre eine natürliche Fortsetzung der bisher erreichten Resultate die Entwicklung von Methoden, mit denen man das vorgestellte Problem auch auf einem Gitter von allgemeinen Hexaedern, Tetraedern oder sogar auf aus vielen verschiedenen Gitterelementen bestehenden allgemeinen Gittern lösen kann.

6.3. Danksagung

Ich habe in dieser Arbeit mit dem Fraunhofer-Institut für Techno- und Wirtschaftsmathematik (ITWM) zusammengearbeitet. Von dort aus stammen auch die Simulationsdaten, welche für die Interpolation verwendet werden. Ich bedanke mich bei Herrn Michel für die aktuelle Themenstellung und die Unterstützung bei Fragen, sowie bei meinen Kommilitonen Matthias Braun für die Lieferung guter Ideen bezüglich des Parallelepipedgitters und bei Linus Henke für die Idee, `plt.figure` eine Zahl als Argument zu übergeben, sodass mehrere Plots gleichzeitig angezeigt werden können.

Unter `mailto:julius.busse@student.uni-siegen.de` bin ich bezüglich Fragen, Anmerkungen und Verbesserungsvorschlägen gerne zu erreichen. Die neueste Version dieses Dokuments findet der Leser unter `https://julius-gs.de/bachelorarbeit`.

A. Kode

A.1. Methoden zur Berechnung von interpolierenden Bézier-Kurven

```
#!/usr/bin/env python3
# Autor: Julius Busse
# -*- coding: utf-8 -*-
import numpy as np
import sympy as sp
from sympy import *
import scipy.integrate as integrate
import matplotlib.pyplot as plt
import matplotlib as matplotlib
import sys
import math

from ensreader import Ensight
from fiberreader import Fiber
############
#  Bezier  #
############

# Diese Funktion löst das LGS für die Koeffizienten von Bezier-Kurven
def get_bezier_coef(points, ableitung, randbed):
    # j=0,...,n, also:
    n = len(points) - 1

    # Matrix aufbauen und
    # Vektor auf der R.S. erstellen
    C = 4 * np.identity(n)
    np.fill_diagonal(C[1:], 1)
    np.fill_diagonal(C[:, 1:], 1)
    P = [2 * (2 * points[i] + points[i + 1]) for i in range(n)]
    if randbed == 'nat':
        C[0, 0] = 2
        C[n - 1, n - 1] = 7
        C[n - 1, n - 2] = 2
        P[0] = points[0] + 2 * points[1]
        P[n - 1] = 8 * points[n - 1] + points[n]
    elif randbed == 'hermite':
        C[0, 0] = 3
        C[0, 1] = 0
        C[n-1, n-2] = 3
        C[n-1, n-1] = 12
        P[0] = 3*points[0] + 1/n*ableitung[0,:]
        P[n - 1] = -1/n*ableitung[n,:] + 3*points[n] + 12*points[n-1]
    else:
        raise ValueError("unzulässige randbedbedingungen")

```

```
46      # Lösen und Punkte b rekonstruieren
47      A = np.linalg.solve(C, P)
48      if points.ndim == 2:
49          B = np.zeros((n,3))
50      else:
51          B = np.zeros(n)
52      for i in range(n - 1):
53          B[i] = 2 * points[i + 1] - A[i + 1]
54      B[n - 1] = A[n-2]+4*A[n-1]-4*points[n-1]
55
56      return A, B
57
58  def get_cubic(a, b, c, d):
59      # Gibt die allgemeine Gleichung eines kubischen Bezier-Polynoms wieder
60      return lambda t: ((1-t)**3*a+3*(1-t)**2*t*b+3*(1-t)*t**2*c+t**3*d)
61
62  def get_cubic_normableitung(a, b, c, d):
63      return lambda t : np.linalg.norm((t**2)*3*(-a+3*b-3*c+d) +
64              t*2*(3*a-6*b+3*c)+(-3*a+3*b))
65
66  def get_bezier_cubic(points, ableitung, randbed):
67      # Löst nach Koeff. und gibt Vektor mit jedem einzelnen Bezier-Polynom aus
68      A, B = get_bezier_coef(points, ableitung, randbed)
69      return [
70          get_cubic(points[i], A[i], B[i], points[i + 1])
71          for i in range(len(points) - 1)
72      ]
73
74  def get_bezier_cubic_quick(points,A,B):
75      return [
76          get_cubic(points[i], A[i], B[i], points[i + 1])
77          for i in range(len(points) - 1)
78      ]
79
80  def evaluate_bezier_quick(x,points,A,B):
81      curves = get_bezier_cubic_quick(points,A,B)
82      n = len(curves)
83      if (not isinstance(x,float)):
84          return np.array((0,0,0))
85          print("ich habe es geschafft")
86      else:
87          if x < 1:
88              return curves[int(np.floor(n*x))]((n*x)%1)
89          else:
90              return curves[len(curves)-1](1)
91
92  def get_bezier_cubic_normableitung(points, ableitung, randbed):
93      # Löst nach Koeff und gibt Vektor mit jeder einzelnen
94      # Normableitung aus
```

```
95      A, B = get_bezier_coef(points, ableitung, randbed)
96      return [
97          get_cubic_normableitung(points[i], A[i], B[i], points[i+1])
98          for i in range(len(points) - 1)
99      ]
100
101 def evaluate_bezier(x,points, ableitung, randbed):
102     # Wertet jedes Teilpolynom auf [0,1] aus und hängt alles aneinander
103     # => Bezier-KURVE
104     A, B = get_bezier_coef(points, ableitung, randbed)
105     curves = get_bezier_cubic_quick(points,A,B)
106     n = len(curves)
107     if x < 1:
108         return curves[int(n*x-((n*x)%1))]((n*x)%1)
109     else:
110         return curves[len(curves)-1](1)
111
112 def evaluate_bezier_normabl(x,points,ableitung, randbed):
113     # Wertet die Normableitung an einem bestimmten Punkt aus
114     curves = get_bezier_cubic_normableitung(points, ableitung, randbed)
115     n = len(curves)
116     if x < 1:
117         return n*curves[int((n*x)-((n*x)%1))]((n*x)%1)
118         # Der Faktor n ist NICHT völlig willkürlich, sondern kommt aus der
119         # Kettenregel.
120     else:
121         return n*curves[len(curves)-1](1)
122
123 def evaluate_bezier_vector(points, feinheit, ableitung, randbed):
124     # Wertet die Bezier-Kurve an n äquidistanten Punkten aus
125     # n: An wie vielen Punkten soll ausgewertet werden
126     # Etwa: n=500, len(curves)=20
127     A,B = get_bezier_coef(points, ableitung, randbed)
128     C=np.zeros((feinheit,3))
129     for m in range(feinheit):
130         t=m/(feinheit-1)
131         C[m,:]=evaluate_bezier_quick(t,points,A,B)
132     return C
133
134 def evaluate_bezier_normabl_vector(points, feinheit, ableitung, randbed):
135     # Wertet die Bezier-Normableitung an n äquidistanten Punkten
136     # aus
137     A=np.zeros(0)
138     for m in range(feinheit):
139         t=m/(feinheit-1)
140         A = np.append(A, evaluate_bezier_normabl(t, points, ableitung, randbed))
141     return A
142
143 def bezier_initializor(fiber, finefiber, randbed):
```

```
144     feinheit = len(finefiber.x)
145     points = np.transpose(np.array([fiber.x, fiber.y, fiber.z]))
146     ableitung=np.transpose(np.array([fiber.dxds, fiber.dyds, fiber.dzds]))
147     x, y, z = points[:,0], points[:,1], points[:,2]
148     bezierpath = evaluate_bezier_vector(points, feinheit, ableitung, randbed)
149     px, py, pz = bezierpath[:,0], bezierpath[:,1], bezierpath[:,2]
150     finepoints = np.transpose(np.array([finefiber.x, finefiber.y,
151         finefiber.z]))
152     return points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints
153 
154 ##############################################################################
155 #                              Alle Plotter ab hier                          #
156 ##############################################################################
157 
158 def finefiber_plot(fiber, finefiber, randbed):
159     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
160             bezier_initializor(fiber, finefiber, randbed)
161     plt.figure(42)
162     finefiberpoints = np.transpose(np.array([finefiber.x, finefiber.y,
163         finefiber.z]))
164     qx, qy, qz = finefiberpoints[:,0], finefiberpoints[:,1], finefiberpoints[:,2]
165     ax3 = plt.axes(projection = '3d')
166     ax3.plot(qx,qy,qz,'ro')
167     ax3.set_xlabel('[m]')
168     ax3.set_ylabel('[m]')
169     ax3.set_zlabel('[m]')
170     plt.savefig("../../TeX/figures/finefiber_plot_new.eps",dpi=300)
171 
172 def bezier_interp_plot(fiber, finefiber):
173     ######################################
174     #  Plottet die interpolierende Kurve  #
175     ######################################
176     points, ableitung, x, y, z, bezierpath, pxnat, pynat, pznat, finepoints = \
177             bezier_initializor(fiber, finefiber, 'nat')
178     points, ableitung, x, y, z, bezierpath, pxherm, pyherm, pzherm, finepoints = \
179             bezier_initializor(fiber, finefiber, 'hermite')
180     plt.figure(40)
181     ax2 = plt.axes(projection = '3d')
182     ax2.plot(x,y,z,'ro')
183     ax2.plot(pxnat,pynat,pznat,'g-')
184     ax2.plot(pxherm,pyherm,pzherm,'b-')
185     ax2.plot(pxherm[0],pyherm[0],pzherm[0],marker = 's', markerfacecolor = 'none',
186             markeredgewidth = 2, markeredgecolor = 'orange',linestyle = 'dashed', marker
187             = 20)
188     ax2.set_xlabel('[m]')
189     ax2.set_ylabel('[m]')
190     ax2.set_zlabel('[m]')
191     plt.savefig("../../TeX/figures/bezier_plot_new.eps",dpi=300)
192 
```

```
193 def bezier_mit_temp(fiber, finefiber, randbed):
194     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
195             bezier_initializor(fiber, finefiber, randbed)
196     feinheit = len(finefiber.x)
197     temp = evaluate_bezier_vector(fiber.T,feinheit, ableitung, 'nat')
198     cmap = plt.cm.hot
199     norm = plt.Normalize(vmin = np.min(temp), vmax =
200             np.max(temp))
201     color = cmap(norm(temp[:,0]))
202     print(color.shape)
203     print(px.shape)
204     plt.figure(2)
205     ax4 = plt.axes(projection = '3d')
206     ax4.plot(x,y,z,'bo')
207     for j in range(len(px)-2):
208         ax4.plot(px[j:j+2],py[j:j+2],pz[j:j+2],'-',c=color[j,:])
209     ax4.set_xlabel('[m]')
210     ax4.set_ylabel('[m]')
211     ax4.set_zlabel('[m]')
212     sm = plt.cm.ScalarMappable(cmap = cmap, norm = norm)
213     plt.colorbar(sm, label="[°K]",pad=0.12)
214     plt.savefig("../../TeX/figures/var_bezier_mit_temp.eps",dpi=300)
215
216 def bezier_mit_dicke(fiber, finefiber, randbed):
217     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
218             bezier_initializor(fiber, finefiber, randbed)
219     feinheit = len(finefiber.x)
220     thck = evaluate_bezier_vector(fiber.diam,feinheit, ableitung, 'nat')
221     cmap = plt.cm.viridis
222     norm = plt.Normalize(vmin = np.min(thck), vmax =
223             np.max(thck))
224     color = cmap(norm(thck[:,0]))
225     plt.figure(900)
226     ax5 = plt.axes(projection = '3d')
227     for j in range(len(px)-2):
228         ax5.plot(px[j:j+2],py[j:j+2],pz[j:j+2],'-',c=color[j,:])
229     ax5.set_xlabel('[m]')
230     ax5.set_ylabel('[m]')
231     ax5.set_zlabel('[m]')
232     sm = plt.cm.ScalarMappable(cmap = cmap, norm = norm)
233     plt.colorbar(sm, label="[m]", pad=0.12)
234     plt.savefig("../../TeX/figures/bezier_mit_dicke_new.eps",dpi=300)
235
236 def bezier_fehler_plot(fiber, finefiber):
237     ##########################################################################
238     #  Plottet die Differenz zwischen der Interpolation und der feinen Kurve  #
239     ##########################################################################
240     points, ableitung, x, y, z, bezierpathnat, px, py, pz, finepoints = \
241             bezier_initializor(fiber, finefiber, 'nat')
```

```
242     points, ableitung, x, y, z, bezierpathherm, px, py, pz, finepoints = \
243             bezier_initializor(fiber, finefiber, 'hermite')
244     feinheit = len(finefiber.x)
245     fig, ax = plt.subplots()
246     bezierdiffnat = np.zeros(len(finepoints))
247     bezierdiffherm = np.zeros(len(finepoints))
248     for j in range(len(finepoints)):
249         bezierdiffnat[j]=np.linalg.norm(bezierpathnat[j,:]-finepoints[j,:])
250         bezierdiffherm[j]=np.linalg.norm(bezierpathherm[j,:]-finepoints[j,:])
251     plt.plot(np.linspace(0,1,feinheit),bezierdiffnat,'g-')
252     plt.plot(np.linspace(0,1,feinheit),bezierdiffherm,'b-')
253     ax.set_xlabel('$s$')
254     ax.set_ylabel('[m]')
255     plt.grid()
256     plt.savefig("../../TeX/figures/bezier_fehler_new.eps",dpi=300)
257
258 def bezier_normabl_plot(fiber, finefiber):
259     ##########################################
260     #  Plottet die Normableitung des Splines  #
261     ##########################################
262     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
263             bezier_initializor(fiber, finefiber, 'hermite')
264     feinheit = len(finefiber.x)
265     fig, ax = plt.subplots()
266     beziernormablnat = evaluate_bezier_normabl_vector(points,feinheit, ableitung,
267             'nat')
268     beziernormablherm = evaluate_bezier_normabl_vector(points,feinheit, ableitung,
269             'hermite')
270     plt.plot(np.linspace(0,1,feinheit),beziernormablnat,'g-')
271     plt.plot(np.linspace(0,1,feinheit),beziernormablherm,'b-')
272     ax.set_xlabel('$s$')
273     ax.set_ylabel('')
274     plt.grid()
275     plt.savefig("../../TeX/figures/bezier_normabl_new.eps",dpi=300)
276
277 def bezier_temp_plot(fiber, finefiber, randbed):
278     ##############################################
279     #  plottet die Interpolation der Temperatur  #
280     ##############################################
281     if randbed == 'hermite':
282         raise ValueError('Bitte für das eindimensionale Zeug natürliche\
283         Randbedingungen nutzen')
284     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
285             bezier_initializor(fiber, finefiber, randbed)
286     feinheit = len(finefiber.x)
287     fig, ax = plt.subplots()
288     temp = evaluate_bezier_vector(fiber.T,feinheit, ableitung, randbed)
289     plt.plot(np.linspace(0,1,feinheit),temp, 'g-')
290     ax.set_xlabel('$s$')
```

```
291     ax.set_ylabel('[°K]')
292     plt.grid()
293     plt.savefig("../../TeX/figures/bezier_temp_new.eps",dpi=300)
294 
295 def bezier_thck_plot(fiber, finefiber, randbed):
296     ########################################
297     #  Plottet die Interpolation der Dicke  #
298     ########################################
299     if randbed == 'hermite':
300         raise ValueError('Bitte für das eindimensionale Zeug natürliche\
301         Randbedingungen nutzen')
302     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
303             bezier_initializor(fiber, finefiber, randbed)
304     feinheit = len(finefiber.x)
305     fig, ax = plt.subplots()
306     thck = evaluate_bezier_vector(fiber.diam,feinheit, ableitung, randbed)
307     plt.plot(np.linspace(0,1,feinheit),thck, 'g-')
308     ax.set_xlabel('$s$')
309     ax.set_ylabel('[m]')
310     plt.tight_layout()
311     plt.grid()
312     plt.savefig("../../TeX/figures/bezier_thck_new.eps",dpi=300)
313 
314 def linke_randbedingungen_visualisieren(fiber, finefiber):
315     ##########################################################################
316     #  Zeigt den Unterschied zwischen den Randbedingungen am linken Rand  #
317     ##########################################################################
318     randbed = 'nat'
319     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
320             bezier_initializor(fiber, finefiber, randbed)
321     feinheit = len(finefiber.x)
322     plt.figure(2)
323     ax7 = plt.axes(projection = '3d')
324     hermitekurve = evaluate_bezier_vector(points, feinheit, ableitung, 'hermite')
325     natkurve = evaluate_bezier_vector(points, feinheit, ableitung, 'nat')
326     ax7.plot(natkurve[0:28,0], natkurve[0:28,1], natkurve[0:28,2], 'g-')
327     ax7.plot(hermitekurve[0:28,0], hermitekurve[0:28,1], hermitekurve[0:28,2], 'b-')
328     ax7.view_init(-20,-29)
329     ax7.plot(points[0:2,0], points[0:2,1], points[0:2,2], 'ro')
330     ax7.set_xticklabels([])
331     ax7.set_yticklabels([])
332     ax7.set_zticklabels([])
333     plt.tight_layout()
334     plt.savefig("../../TeX/figures/linke_randbedingungen_new.eps",dpi=300)
```

```
1 #!/usr/bin/env python3
2 # Autor: Julius Busse
3 # -*- coding: utf-8 -*-
```

```
import numpy as np
import sympy as sp
from sympy import *
import scipy.integrate as integrate
import matplotlib.pyplot as plt
import sys
import math

from ensreader import Ensight
from fiberreader import Fiber
from fiberreader import add_diameter_to_fiber_object
from bezier_methoden import *

def main():
    if len(sys.argv) <= 1:
        cfdfilename = "../daten/cfd/cube_6x6/flow.case"
    else:
        cfdfilename = sys.argv[1]
    if len(sys.argv) <= 2:
       fiberfilename = "../daten/fiber/fiber_cube_mixed_coarse_20/fiber_2.mat"
       finefiberfilename = "../daten/fiber/fiber_cube_mixed/fiber_2.mat"
    else:
        fiberfilename = sys.argv[2]
    cfd = Ensight(cfdfilename)
    fiber = Fiber(fiberfilename)
    finefiber = Fiber(finefiberfilename)
    # finefiber_plot(fiber, finefiber, 'nat')
    # bezier_interp_plot(fiber, finefiber)
    # bezier_fehler_plot(fiber, finefiber)
    # bezier_normabl_plot(fiber, finefiber)
    # bezier_temp_plot(fiber, finefiber, 'nat')
    # bezier_thck_plot(fiber, finefiber, 'nat')
    bezier_mit_temp(fiber, finefiber, 'hermite')
    # bezier_mit_dicke(fiber, finefiber, 'hermite')
    # linke_randbedingungen_visualisieren(fiber, finefiber)
    plt.show()

if __name__ == '__main__':
    main()
```

A.2. Methoden zur Lösung des Problems am Würfelgitter

```
#!/usr/bin/env python3
# Autor: Julius Busse
# -*- coding: utf-8 -*-
from bezier_methoden import *
from matplotlib.colors import ListedColormap, BoundaryNorm, Normalize
import matplotlib as matplotlib
from matplotlib.collections import LineCollection
```

```
8  import matplotlib.cm as cm
9
10 def findcubicroots(a,b,c,d,intlower,intupper):
11     # Findet Nullstellen des kubischen Polynoms ax^3+bx^2+cx+d
12     # zwischen intlower und intupper
13     # Arbeitet analytisch
14     x = symbols('x')
15     lsgmenge = solveset(Eq(a*x**3+b*x**2+c*x+d,0),x)
16     index1=0
17     reellelsg = np.zeros(0)
18     for lsg in lsgmenge:
19         # Überprüfe, ob die Lösungen reell sind und im geforderten
20         # Intervall liegen
21         if lsg.is_real and intlower <= lsg and lsg <= intupper:
22             reellelsg = np.append(reellelsg,float(lsg))
23             index1 += 1
24     return(reellelsg)
25
26 def findgridintersections(a,b,c,d,gridwidth):
27     # Findet Schnittstellen mit dem äquidistanten Würfelgitter mit
28     # Gitterlänge gridwidth
29     reellelsg = np.zeros(0)
30     extremakandidaten = findmaxcubic(a,b,c,d)
31     numlower = np.floor(extremakandidaten[0]/gridwidth)
32     numlower = int(numlower)
33     numupper = np.ceil(extremakandidaten[1]/gridwidth)
34     numupper = int(numupper)
35     for j in range(numlower,numupper+1):
36         # Sucht nach Nullstellen von p(x)-j*gridwidth in einem
37         # geeigneten Intervall
38         lsglokal = findcubicroots(a,b,c,d-j*gridwidth,0,1)
39         reellelsg = np.append(reellelsg,lsglokal)
40     return(reellelsg)
41
42 def evalcubic(a,b,c,d,x):
43     return a*x**3+b*x**2+c*x+d
44
45 def findmaxcubic(a,b,c,d):
46     # Findet die Extremstellen von ax^2+bx^2+cx+d zwischen 0 und 1
47     lrand = d
48     rrand = a+b+c+d
49     # Berechnung der Nullstellen der ersten Ableitung
50     exstelle = np.array([lrand, rrand])
51     radikand = ((2*b)/(3*a))**2-c/(3*a)
52     if radikand >= 0:
53         # Überprüft, ob die Nullstellen reell sind
54         mitte1 = -b/(3*a)+sqrt(radikand)
55         mitte2 = -b/(3*a)-sqrt(radikand)
56         # Überprüfe die Funktionswerte an den Kandidaten für
```

```
57          # Extremstellen
58          if 0 <= mitte1 and mitte1 <= 1:
59              polwert = a*mitte1**3+b*mitte1**2+c*mitte1+d
60              exstelle = np.append(exstelle, polwert)
61          if 0 <= mitte2 and mitte2 <= 1:
62              polwert = a*mitte2**3+b*mitte2**2+c*mitte2+d
63              exstelle = np.append(exstelle,polwert)
64      maxwerte = np.zeros(0)
65      for stelle in exstelle:
66          maxwerte = np.append(maxwerte,evalcubic(a,b,c,d,stelle))
67      maxwerte = np.sort(maxwerte)
68      # maxwerte[0]: Minimum, maxwerte[n-1]: Maximum
69      return (maxwerte[0], maxwerte[len(maxwerte) - 1])
70
71
72  def findgridintersection_multiple_bezier(points,gridwidth,ableitung,randbed):
73      # Findet Gitterschnittstellen einer dreidimensionalen
74      # Bezier-Kurve mit natürlichen Randbedingungen, welche points
75      # interpoliert
76      [A,B] = get_bezier_coef(points,ableitung, randbed)
77      n = len(A)
78      reellelsg = np.zeros(0)
79      for dim in range(3):
80          for j in range(n):
81              # Iteriert die findgridintersections-Funktion
82              a = points[j,dim]
83              b = A[j,dim]
84              c = B[j,dim]
85              d = points[j+1,dim]
86              lsglokal = findgridintersections(
87                      -a+3*b-3*c+d,3*a-6*b+3*c,-3*a+3*b,a,gridwidth)
88              lsglokal = (lsglokal+j)/n
89              reellelsg = np.append(reellelsg,lsglokal)
90      return(np.sort(reellelsg))
91
92  def inwelchemwuerfelsindwir(x,gridwidth):
93      # Überprüft in welchem Würfel des Würfelgitters mit Gitterweite
94      # gridwidth x liegt
95      ecken = np.zeros(3)
96      for dim in range(3):
97          ecken[dim] = np.floor(x[dim]/gridwidth)
98      return ecken
99
100 def bezierintegralcalculator(points,ableitung,randbed,low,up):
101     # Numerische Integration
102     return integrate.quad(evaluate_bezier_normabl, low, up,
103             args=(points,ableitung,randbed,))
104
105 def compute_diameter(fiber, rho, dIn, uIn, TIn):
```

```
106     rhoIn = rho(TIn)
107     return np.sqrt(rhoIn / rho(fiber.T) * uIn / fiber.u * dIn*dIn)
108 
109 def wuerfelgitter(fiber,finefiber,gridwidth,randbed):
110     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
111             bezier_initializor(fiber, finefiber, randbed)
112     temp = fiber.T
113     thck = fiber.diam
114     # Gibt ein Array aus, wo zu jedem Wüfel die Länge und mittlere
115     # Temperatur des enthaltenen Fadens berechnet wird
116     gitterschnitt = findgridintersection_multiple_bezier(points,gridwidth,
117             ableitung, randbed)
118     n = len(gitterschnitt)
119     A = np.zeros((n,6))
120     # In A schreiben wir die Ergebnisse rein
121     for j in range(n-1):
122         # Bestimme Koordinaten des Würfels
123         wuerfelmitte = evaluate_bezier((gitterschnitt[j] +
124             gitterschnitt[j+1])/2,points, ableitung, randbed)
125         welcherwuerfel = \
126                 inwelchemwuerfelsindwir(wuerfelmitte, gridwidth)*gridwidth
127         # print("Wir sind im Würfel ", welcherwuerfel)
128         A[j,:3] = welcherwuerfel
129         # Bestimme Länge der Kurve im Würfel
130         kurvenlaenge=bezierintegralcalculator(points, ableitung, randbed, gitterschnitt[
131                 gitterschnitt[j+1])[0]
132         A[j,3] = kurvenlaenge
133         # print("da hat die kurve die länge ", kurvenlaenge)
134         # Bestimme mittlere Termperatur der Kurve im Würfel
135         temperatur = integrate.quad(evaluate_bezier, gitterschnitt[j],
136                 gitterschnitt[j+1], args =
137                 (temp,ableitung,'nat',))/(gitterschnitt[j+1]-gitterschnitt[j])
138         # print(" und das mittel über die temperatur beträgt ", temperatur)
139         A[j, 4] = temperatur[0]
140         # Bestimme mittlere Dicke der Kurve im Würfel
141         thickness = integrate.quad(evaluate_bezier, gitterschnitt[j],
142                 gitterschnitt[j+1], args =
143                 (thck,ableitung,'nat'))/(gitterschnitt[j+1]-gitterschnitt[j])
144         # print(" und das mittel über die fadendicke beträgt ", thickness)
145         A[j,5] = thickness[0]
146     return A
147 
148 def wuerfelgitter_mit_fadenlaenge(fiber, finefiber, gridwidth, randbed,
149         wuerfelgitter_resultat):
150     points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
151         bezier_initializor(fiber, finefiber, randbed)
152     temp = fiber.T
153     thck = fiber.diam
154     gitterschnitt = findgridintersection_multiple_bezier(points,gridwidth,
```

```
155             ableitung, randbed)
156     A, B = get_bezier_coef(points, ableitung, randbed)
157     ax2 = plt.axes(projection = '3d')
158     ############################
159     #  länge im gitterelement  #
160     ############################
161     plt.figure(1)
162     for j in range(len(gitterschnitt)-2):
163         kurve = [evaluate_bezier_quick(x,points,A,B) for x in
164                 np.linspace(gitterschnitt[j],gitterschnitt[j+1],20)]
165         kurve = np.array(kurve)
166         cmap = cm.viridis
167         norm = Normalize(vmin = np.min(wuerfelgitter_resultat[:,3]), vmax =
168                 np.max(wuerfelgitter_resultat[:,3]))
169         color = cmap(norm(wuerfelgitter_resultat[j,3]))
170         ax2.plot(kurve[:,0], kurve[:,1], kurve[:,2], color=color#,lw=5
171                 )
172         ax2.plot(kurve[0,0], kurve[0,1], kurve[0,2], 'ro')
173         # fig.colorbar(p)
174     ax2.set_xlabel('[m]')
175     ax2.set_ylabel('[m]')
176     ax2.set_zlabel('[m]')
177     sm = plt.cm.ScalarMappable(cmap = cmap, norm = norm)
178     plt.colorbar(sm, label='[m]', pad=0.12)
179     plt.savefig("../../TeX/figures/bezier_markierte_gitterlaenge_new.eps",dpi=300)
180     # fiber von gitterschnitt[j-1] bis gitterschnitt[j] plotten, einfärben mit
181     # wuerfelgitter_resultat[j,3,4 oder 5]
```

```
1  #!/usr/bin/env python3
2  # Autor: Julius Busse
3  # -*- coding: utf-8 -*-
4  from wuerfelgitter_methoden import *
5
6  def main():
7      # Selbe Variablen wie in fibersplines
8      if len(sys.argv) <= 1:
9          cfdfilename = "../daten/cfd/cube_6x6/flow.case"
10     else:
11         cfdfilename = sys.argv[1]
12     if len(sys.argv) <= 2:
13        fiberfilename = "../daten/fiber/fiber_cube_mixed_coarse_20/fiber_6.mat"
14        finefiberfilename = "../daten/fiber/fiber_cube_mixed/fiber_6.mat"
15     else:
16         fiberfilename = sys.argv[2]
17     cfd = Ensight(cfdfilename)
18     fiber = Fiber(fiberfilename)
19     finefiber = Fiber(finefiberfilename)
20     gridwidth = 0.1
```

```
    # A = wuerfelgitter(fiber,finefiber,gridwidth,'nat')
    # np.save("speicher/wuerfelgitter",A)
    A = np.load("speicher/wuerfelgitter.npy")
    wuerfelgitter_mit_fadenlaenge(fiber, finefiber, gridwidth, 'hermite', A)
    plt.show()


if __name__ == '__main__':
    main()
```

A.3. Methoden zum Generieren eines eigenen Paralellepipedgitters

```
#!/usr/bin/env python3
# Autor: Julius Busse
# -*- coding: utf-8 -*-

# Diese Datei soll ein eigenes Gitter von Quadern oder Parallelepipeden
# generieren.
# Drei Punkte werden benötigt, um die Orientierung im Raum anzugeben
# dann teile ich jede Achse zufällig in Teilabschnitte auf.
import numpy as np
import matplotlib.pyplot as plt
import random as random

def gittergenerator(n):
    start = np.array((-0.1,-0.1,-0.1)) # Gitter startet im Ursprung und wird am ende
    # um "start" verschoben
    xachse =np.array((1.5,-0.05,0))
    yachse = np.array((0,1.4,0))
    zachse = np.array((0,0,1.2))
    achsen = np.empty((n,3))
    for dim in range(3):
        for j in range(n-2):
            achsen[j,dim] = random.random()
            achsen[n-2,dim] = 0
            achsen[n-1,dim] = 1
        achsen[:,dim] = np.sort(achsen[:,dim])
    # gitter = np.empty(((n-1)**3,3,4))# Anzahl Quader, Dimensionen, Punkte pro
    # # Gitterelement
    # index = 0
    # for j in range(n-1):
    #     for i in range(n-1):
    #         for k in range(n-1):
    #             gitter[index,:,0] = (xachse * achsen[j,0]   + yachse *
    #                     achsen[i,1]   + zachse * achsen[k,2] + start)
    #             gitter[index,:,1] = (xachse * achsen[j+1,0] + yachse *
    #                     achsen[i,1]   + zachse * achsen[k,2] + start)
    #             gitter[index,:,2] = (xachse * achsen[j,0]   + yachse *
    #                     achsen[i+1,1] + zachse * achsen[k,2] + start)
```

```
38      #              gitter[index,:,3] = (xachse * achsen[j,0]    + yachse *
39      #                      achsen[i,1]    + zachse * achsen[k+1,2] + start)
40      #              index += 1
41      # return gitter # (n-1)^3 x 3 x 4 array
42      return start, xachse, yachse, zachse, achsen
43
44  def main():
45      gitter = gittergenerator(10)
46      ax2 = plt.axes(projection = '3d')
47      ax2.plot(gitter[0,0,:], gitter[0,1,:], gitter[0,2,:],'ro')
48      plt.show()
49
50  if __name__ == '__main__':
51      main()
```

A.4. Methoden zur Lösung des Problems am Paralellepipedgitter

```
1   #!/usr/bin/env python3
2   # Autor: Julius Busse
3   # -*- coding: utf-8 -*-
4   # from ensreader import *
5   from bezier_methoden import *
6   from eigenesgittergenerieren import *
7   from sympy.solvers import solve
8   from sympy import Symbol
9   from sympy import sin
10  from sympy.abc import x
11  from sympy import nsolve
12  from fiberreader import *
13  import numpy as np
14
15  # cfdfilename = "../daten/cfd/cube_6x6/flow.case"
16  # coords, tet, pyr, penta, hexa, scalar_vars, vector_vars = read_ensight(cfdfilename)
17  # print(coords)
18  # Das Problem ist am eigenen Gitter gelöst, da das in cfdfilename liegende
19  # Gitter keine ebenen Flächen hat.
20
21  def binichinquaderj(gitter,j,p):
22      start, xax, yax, zax, achsen = gitter
23      n = achsen.shape[0]
24      prel = p - start
25      A = np.transpose(np.array((xax, yax, zax)))
26      lambdavar = np.linalg.solve(A, prel)
27      jz = int(j % n)
28      zhilfe = (j-jz)/n
29      jy = int(zhilfe % n)
30      yhilfe = (zhilfe-jy)/n
31      jx = int(yhilfe % n)
32      if (achsen[jx,0] <= lambdavar[0] and lambdavar[0] <= achsen[jx+1,0]
```

```
            and achsen[jy,1] <= lambdavar[1] and lambdavar[1] <= achsen[jy+1,1]
            and achsen[jz,2] <= lambdavar[2] and lambdavar[2] <= achsen[jz+1,2]):
        return 1
    return 0

def inwelchemquadersindwir(gitter, p):
    start, xax, yax, zax, achsen = gitter
    n = achsen.shape[0]
    n = (n-1)**3
    for j in range(n+1):
        if binichinquaderj(gitter, j, p):
            return j

def nachbarfinder(nges,a): # n: Gesamtanzahl Quader (kubische Zahl), j: Quaderindex
    nachbarn = np.array((a))
    n = np.round(nges**(1/3))
    # Wir müssen alle 27-1 Kombinationen erfassen:
    for i in range(-1,2):
        for j in range(-1,2):
            for k in range(-1,2):
                nachbarn = np.append(nachbarn,a+i*1+j*(n-1)+k*(n-1)**2)
    nachbarn = np.sort(nachbarn)
    nachbarn = nachbarn[nachbarn >= 0]
    nachbarn = nachbarn[nachbarn < nges]
    nachbarn = [int(a) for a in nachbarn]
    return nachbarn

def parallelepipedgitter(fiber,finefiber,gitter,randbed, feinheit):
    points, ableitung, x, y, z, bezierpath, px, py, pz, finepoints = \
            bezier_initializor(fiber, finefiber, randbed)
    temp = fiber.T
    thck = fiber.diam
    kurve = evaluate_bezier_vector(points,feinheit,ableitung,randbed)
    A = np.empty((feinheit,4))
    index = 0
    letzterquader = 0
    for p in kurve:
        A[index,0:3] = p
        if binichinquaderj(gitter,letzterquader,p):
            A[index, 3] = letzterquader
            print("Ich bleibe im selben Gitterelement")
        else:
            nachbarerfolg = 0
            nachbarliste = nachbarfinder(gitter.shape[0],letzterquader)
            for nachbar in nachbarliste:
                if binichinquaderj(gitter,nachbar,p):
                    A[index, 3] = nachbar
                    letzterquader = nachbar
                    nachbarerfolg = 1
```

```
                    print("Mein neues Gitterelement wurde numerisch\
                            effizient gefunden")
            if not nachbarerfolg:
                print(inwelchemquadersindwir(gitter,p))
                A[index,3] = inwelchemquadersindwir(gitter,p)
                print(A[index,3])
                letzterquader = int(A[index,3])
                print("Mein neues Gitterelement wurde nur durch\
                        Brute-Force gefunden :(")
        index += 1
    return A

def parallelepipedgitter_punkte_in_element(A):
    relevante_gitterelemente = sorted(set(A[:,3].astype(int)))
    B = np.zeros((2,0)) for gitterindex in relevante_gitterelemente:
        res = sum(1 for i in A[:,3] if i == gitterindex)
        B = np.append(B, np.array([gitterindex, res]))
    B.shape = [-1, 2]
    print(B)
    return B
```

```
#!/usr/bin/env python3
# Autor: Julius Busse
# -*- coding: utf-8 -*-
import array_to_latex as a2l
from bezier_methoden import *
from eigenesgittergenerieren import *
from sympy.solvers import solve
from sympy import Symbol
from fiberreader import *
import numpy as np
from parallelepipedgitter_methoden import *

def main():
    # Selbe Variablen wie in fibersplines
    if len(sys.argv) <= 1:
        cfdfilename = "../daten/cfd/cube_6x6/flow.case"
    else:
        cfdfilename = sys.argv[1]
    if len(sys.argv) <= 2:
        fiberfilename = "../daten/fiber/fiber_cube_mixed_coarse_20/fiber_2.mat"
        finefiberfilename = "../daten/fiber/fiber_cube_mixed/fiber_2.mat"
    else:
        fiberfilename = sys.argv[2]
    cfd = Ensight(cfdfilename)
    fiber = Fiber(fiberfilename)
    finefiber = Fiber(finefiberfilename)
    randbed = 'hermite'
```

```
28      # n=20
29      # gitter = gittergenerator(n)
30      # np.save("speicher/gitter", gitter)
31      gitter = np.load("speicher/gitter.npy",allow_pickle=True)
32      # print(inwelchemquadersindwir(gitter,np.array((1,1,1))))
33      print(gitter)
34      # A = parallelepipedgitter(fiber,finefiber,gitter,'hermite',500)
35      # np.save("speicher/parallelepipedgitter_resultat", A)
36      A = np.load("speicher/parallelepipedgitter_resultat.npy")
37      B = parallelepipedgitter_punkte_in_element(A)
38      print(B)
39      print(a2l.to_ltx(B, frmt = '{:.0f}', arraytype = 'bmatrix'))
40
41  if __name__ == '__main__':
42      main()
```

B. Literatur

[1] Foto der FET-101 Schmelzspinnanlage im Labor vom AMIBM der Universität Maastricht, zur Verfügung gestellt von [9].

[2] W. Albrecht, H. Fuchs, and W. Kittelmann. *Vliesstoffe: Rohstoffe, Herstellung, Anwendung, Eigenschaften, Prüfung.* WILEY-VCH Verlag, Chemnitz, 2000.

[3] W. Arne, N. Marheineke, J. Schnebele, and R. Wegener. Fluid-fiber-interactions in rotational spinning process of glass wool production. *Journal of Mathematics in Industry*, 1:article number 2, 2011.

[4] EDANA. ISO and CEN definition of nonwovens. `https://www.edana.org/docs/default-source/edana-nonwovens/iso-and-cen-definition-of-nonwovens.pdf`, (letzter Zugriff: 12.12.2021).

[5] Freudenberg Performance Materials. `https://www.freudenberg-pm.com/materialien/vliesstoffe/meltblown-vliesstoffe`, (letzter Zugriff: 13.12.2021).

[6] R. W. Freund and R. H. W. Hoppe. *Stoer/Bulirsch: Numerische Mathematik 1 (10. Auflage).* Springer, Davis, Augsburg/Houston, 2007.

[7] R. Plato. *Numerische Mathematik kompakt (4. Auflage).* Vieweg + Teubner, Siegen, 2010.

[8] D. H. Reneker and A. L. Yarin. Electrospinning jets and polymer nanofibers. *Polymer*, 49:2387 – 2425, 7. Februar 2008.

[9] A. Schmeißer. Fraunhofer Institut für Techno- und Wirtschaftsmathematik (ITWM).

[10] H. R. Schwarz and N. Köckler. *Numerische Mathematik (7. Auflage).* Vieweg + Teubner, Paderborn, 2009.

[11] WDR. Sendung mit der Maus vom 23.08.2020. `https://kinder.wdr.de/tv/die-sendung-mit-der-maus/av/video-sachgeschichte-maske-funktion-100.html`, (letzter Zugriff: 12.12.2021).

[12] J. Werner. *Numerische Mathematik 1.* Vieweg, Braunschweig, 1992.

C. Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende schriftliche Bachelorarbeit selbstständig verfasst und keine anderen als die von mir angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinne nach entnommen sind, wurden in jedem Fall unter Angabe der Quellen (einschliesslich des World Wide Web und anderer elektronischer Text- und Datensammlungen) kenntlich gemacht. Dies gilt auch für die beigegebenen Zeichnungen, bildlichen Darstellungen, Skizzen und dergleichen. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung der Bachelorarbeit ausschliesst und weitere angemessene Sanktionen zur Folge haben kann.

Siegen, den 21.12.2021

_____

(Julius Busse)