Lehrstuhl für Mathematik mit Schwerpunkt
Digitale Bildverarbeitung

Approximation und Interpolation im dreidimensionalen Raum mithilfe von Quaternionen

Bachelorarbeit von

Eduard Fast

BETREUT VON
Prof. Dr. Tomas Sauer

20.07.2021

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Aufgabe, Approximationen und Interpolationen im dreidimensionalen Raum mit Quaternionen durchzuführen und grafisch darzustellen. Dazu werden drei Algorithmen verwendet, mit welchen dies umgesetzt wird. Im theoretischen Teil der Arbeit werden die Quaternionen grundsätzlich mit allen belangvollen Regeln erklärt. Auf dessen Grundlage und wichtig für das weitere Vorgehen wird der SLERP (Spherical Linear Interploation) gezeigt. Anschließend werden die verwendeten Algorithmen nacheinander erläutert, um sie dann im praktischen Teil miteinander anhand der entstandenen Ergebnisse und Laufzeiten vergleichen zu können. Der dafür benötigte Octave-Code wird ebenfalls in der Arbeit gezeigt und das Vorgehen wird beschrieben. Zum Schluss wird das beobachtete Ergebnis nochmal erläutert.

Contents

1 Einleitung

Die folgende Bachelorarbeit konzentriert sich auf das Anwenden von Quaternionen auf Approximations- und Interpolationsalgorithmen.
Quaternionen sind eine Erweiterung der komplexen Zahlen um zwei weitere Dimensionen, die am 16. Oktober 1843 von Sir William Rowan Hamilton entdeckt wurden. Sein Ziel war es in einem Raum verschiedene Punkte addieren und multiplizieren zu können. Bei seinem ersten Ansatz hat er angefangen dieses Problem mit dreidimensionalen Zahlen zu lösen, also hat er die komplexen Zahlen um eine Dimension erweitert, wodurch Additionen zwar funktioniert haben, Multiplikationen jedoch nicht. Nach längerer Überlegung kam er auf den Gedanken keine dreidimensionalen, sondern vierdimensionale Zahlen zu verwenden, wobei er auf folgende Formel kam.

$$i^2 = j^2 = k^2 = ijk = -1 \tag{1}$$

[16]
Dieser Idee gab er den Namen "Quaternionen", was aus dem lateinischen "quaternio" stammt und "Vierheit" bedeutet. Eine Quaternion ist aufgebaut aus einem skalaren Wert w und einem dreidimensionalen imaginären Wert, mit dem die Achsen x,y und z beschrieben werden. Die allgemeine Bezeichnung für eine Quaternion ist H, nach Hamilton, womit es durch H = w + ix +jy + kz dargestellt wird. Oftmals wird eine Quaternion auch als q = [w,v] geschrieben, wobei v der dreidimensionale Vektor der imaginären Einheiten ist. Ist der Realteil hierbei null, wird die Darstellung als "reines Quaternion" bezeichnet.
Diese Entdeckung wird heute oft in Computerspielen verwendet, da man damit Bewegungen im Raum gut darstellen kann. Außerdem entsteht bei Quaternionen kein "gimbal lock", wie es bei den normalerweise benutzten Euler-Winkeln der Fall ist, was bedeutet, dass eine Lenkrichtung ausfällt, sobald sich eine Drehachse parallel zu einer anderen Drehachse befindet. Deshalb findet Hamiltons Entdeckung auch in der Schifffahrt und sogar in der Raumfahrt ihren Nutzen.[12][4]
Ziel dieser Arbeit ist es zu zeigen, wie sich die Quaternionen bei einer Approximation oder Interpolation verhalten werden. Dabei sollen möglichst glatte Kurven entstehen und ein Vergleich erstellt werden, welche Algorithmen sich am besten eignen.
Aufeinander aufbauend wird die Arbeit erst die Quaternionen erklären, um mehr über die mathematischen Rechenregeln zu erfahren. Anschließend werden die verwendeten Rotations-, Approximations- und Interpolationsmethoden erläutert und im letzten Teil werden die Ergebnisse meiner Untersuchungen präsentiert.

2 Quaternionenregeln

Nehmen wir nun die Entdeckung von Hamilton her und versuchen die einzelnen imaginären Teile einzeln zu multiplizieren. Dabei erhalten wir folgendes Ergebnisse, welche als Hamilton Regeln bekannt sind.

$$i^2 = -1 \qquad ij = k \qquad ik = -j \tag{2}$$
$$ji = -k \qquad j^2 = -1 \qquad jk = i \tag{3}$$
$$ki = j \qquad kj = -i \qquad k^2 = -1 \tag{4}$$

Man erkennt man sofort, dass $ji \neq ij$, $ki \neq ik$ und $kj \neq jk$ ist, was bedeutet, dass Hamilton bei seiner Entdeckung auf die Kommutativität verzichten musste.[12]

2.1 Rechenregeln

Diese Eigenschaft hat einen Einfluss auf einige Rechnungen mit Multiplikationen für Quaternionen. Das Assoziativ- und Distributivgesetz ist für Summen und Produkte erfüllt.

Addition und Subtraktion

Die Addition funktioniert bei den Quaternionen analog zu den komplexen Zahlen. Die Werte werden Komponentenweise aufaddiert.

$$q_1 + q_2 = (w_1 + x_1 i + y_1 j + z_1 k) + (w_2 + x_2 i + y_2 j + z_2 k) \tag{5}$$
$$= (w_1 + w_2) + (x_1 i + x_2 i) + (y_1 j + y_2 j) + (z_1 k + z_2 k) \tag{6}$$

Bei der Subtraktion wird ebenfalls alles Komponentenweise voneinander abgezogen.

$$q_1 - q_2 = (w_1 + x_1 i + y_1 j + z_1 k) - (w_2 + x_2 i + y_2 j + z_2 k) \tag{7}$$
$$= (w_1 - w_2) + (x_1 i - x_2 i) + (y_1 j - y_2 j) + (z_1 k - z_2 k) \tag{8}$$

Multiplikation

Im Falle der Multiplikation jedoch, sind die Auswirkungen der Hamilton-Regeln zu sehen.

$$(w_1 + x_1 i + y_1 j + z_1 k)(w_2 + x_2 i + y_2 j + z_2 k) = (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) + \tag{9}$$
$$(w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2) i + \tag{10}$$
$$(w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2) j + \tag{11}$$
$$(w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2) k \tag{12}$$

Man muss also darauf achten, dass man die Reihenfolge in der man die Quaternionen multiplizieren will einhält, da sonst unterschiedliche Ergebnisse herauskommen.

Division

Die Division wird nicht wie üblich durch $\frac{q1}{q2}$ dargestellt, sondern mit einer inversen Quaternion, da man aufgrund fehlender Kommutativität zwischen $q_1 q_2^{-1}$ und $q_2^{-1} q_1$ unterscheiden muss. Die Berechnung ist daraufhin analog zur Multiplikation.

Komplex Konjugierte

Die komplex Konjugierte funktioniert hier kongruent zu den komplexen Zahlen indem das Vorzeichen des Imaginärteils eines Quaternions getauscht wird.

$$q = [w, v] \qquad q^* = [w, -v] \qquad\qquad v = (ix, jy, zk) \tag{13}$$

$$qq^* = w^2 + x^2 + y^2 + z^2 \tag{14}$$

Multipliziert man das ursprüngliche Quaternion mit seiner komplex Konjugierten, fallen aufgrund den oben genannten Regeln die imaginären Werte weg.

Betrag

Den Betrag eines Quaternions berechnet man auf dieselbe Weise wie den Betrag eines einfachen Vektors, indem man alle reellen Komponenten quadriert, addiert und anschließend von allem die Wurzel zieht.

$$||q|| = \sqrt{w^2 + x^2 + y^2 + z^2} \tag{15}$$

Inverse

Jedes Quaternion besitzt auch eine inverse Form.

$$q^{-1} = q^*/||q||^2 \tag{16}$$

Normalisiertes Quaternion

Eine Quaternion gilt als normalisiert, wenn es einen Betrag von eins hat. Dies wird auch Einheitsquaternion genannt. Einheitsquaternionen geben eine Richtung vor, jedoch keine spezifische Länge. Man erhält sie, indem bei ein Quaternion q alle reellen Komponenten durch den Betrag von q teilt.

$$qe = (\frac{w}{||q||} + \frac{x}{||q||} + \frac{y}{||q||} + \frac{z}{||q||}) \tag{17}$$

Außerdem gilt:

$$qeqe^{-1} = 1 \tag{18}$$

und

$$qe^* = qe^{-1} \tag{19}$$

[11][20]

2.2 Rotationen

2.2.1 Polarform

Um Rotationen im dreidimensionalen Raum mit Quaternionen durchzuführen, wird eine Polarform benötigt. Diese ist wie folgt definiert:

$$q = ||q||cos(\frac{\theta}{2}) + isin(\frac{\theta}{2}) + jsin(\frac{\theta}{2}) + ksin(\frac{\theta}{2}) \tag{20}$$

Ist das Quaternion q nun ein Einheitsquaternion qe gilt eine andere Schreibweise

$$qe = [cos(\frac{\theta}{2}), nsin(\frac{\theta}{2})], \tag{21}$$

dabei hat der Vektor n die Länge 1.[20]

2.2.2 Rotationsformel und Rotationsmatrix

Die Rotation von Vektoren ist das Gebiet, in dem die Quaternionen am meisten verwendet werden. Dazu verwendet man folgende Formel:

$$Rot(p) = q_1 p q_1^{-1} \tag{22}$$

Dabei ist der Punkt p ein beliebiger Vektor und q_1 ein Einheitsquaternion. Damit wird ein weiterer Einheitsvektor auf einem Einheitskreis bestimmt. Der Vektor p bestimmt dabei den Ort, wo dieser Punkt Rot(v) letztendlich landet. In der Regel wird dieser Vektor als ein reines Quaternion festgelegt p = [0,v].
Die Einheintsquaternion ist definiert durch $q_1 = [cos(\frac{\theta}{2}), nsin(\frac{\theta}{2})]$, wobei n ein Vektor der Länge 1 sein muss. Das Ergebnis Rot(v) wird ein reines Quaternion sein.[6]
Das kann man zeigen, indem man Formel 22 ausformuliert. Hier muss man beachten, dass wenn es mehrere Quaternionen zu multiplizieren gibt, von links nach rechts gerechnet wird. Also wird erst das Ergebnis von q_1 und p berechnet, welches dann mit q_1^{-1} multipliziert wird.

$$Rot(p) = (w + xi + yj + zk)(0 + ui + vj + mk)(w - xi - yj - zk) \tag{23}$$

Berechnet man nun das Ergebnis der ersten beiden Quaternionen und sortiert die Imaginärteile bekommt man folgende Lösung:

$$\begin{pmatrix} -xu & -yv & -zm \\ wui & ymi & -zvi \\ wvj & -xmj & zuj \\ wmk & xvk & -yuk \end{pmatrix}$$

Multipliziert man das Ergebnis mit q_1^* erhalten wir wiederum die Lösungen für den skalaren Wert und für die Koeffizienten i,j,k. Demnach erhalten wir nach der Multiplikationsregel für Quaternionen diese Ergebnisse.

Skalarwert:

$$\begin{pmatrix} -wxu & -wyv & -wzm \\ wxu & xym & -xzv \\ wyv & -xym & yzu \\ wzm & xzv & -yzu \end{pmatrix}$$

Für den skalaren Wert erhalten wir null als Ergebnis. Das erkennt man hier sehr gut an der unten liegenden Matrix. Es gibt von jedem positiven Wert ein negatives Äquivalent.

Nach i zusammenfassen:

$$\begin{pmatrix} w^2 + x^2 - y^2 - z^2 \\ 2(xy - wz) \\ 2(wy + xz) \end{pmatrix} \begin{pmatrix} u \\ v \\ m \end{pmatrix}$$

Nach j zusammenfassen:

$$\begin{pmatrix} 2(xy + wz) \\ w^2 - x^2 + y^2 - z^2 \\ 2(yz - xw) \end{pmatrix} \begin{pmatrix} u \\ v \\ m \end{pmatrix}$$

Nach k zusammenfassen:

$$\begin{pmatrix} 2(xz - wy) \\ 2(yz + wx) \\ w^2 - x^2 + y^2 - z^2 \end{pmatrix} \begin{pmatrix} u \\ v \\ m \end{pmatrix}$$

Fügt man diese Werte nun zusammen erhält man eine orthogonale 3x3 Rotationsmatrix.

$$\begin{pmatrix} w^2 + x^2 - y^2 - z^2 & 2(xy + wz) & 2(xz - wy) \\ 2(xy - wz) & w^2 - x^2 + y^2 - z^2 & 2(yz + wx) \\ 2(wy + xz) & 2(yz - xw) & w^2 - x^2 + y^2 - z^2 \end{pmatrix} \begin{pmatrix} u \\ v \\ m \end{pmatrix}$$

[8]

Ob man jetzt die Rotationsmatrix benutzt oder die Quaternionen ist jedem selbst überlassen. Jedoch werden für die Matrix mehr Rechenschritte benötigt, weshalb es sinnvoller ist Formel 22 zu verwenden.
Um nun die Korrektheit des rotierten Quaternions zu beweisen, kann man ein paar

Kontrollschritte durchführen. Erstmals sollte das Ergebnis einen skalaren Wert von null besitzen. Das ist immer sichergestellt, solange das Anfangsquaternion ein reines Quaternion ist. Des Weiteren darf der Wert der Achse, um die gedreht wurde, nicht verändert worden sein und der Betrag des berechneten Quaternions soll dem des Ursprungsquaternions entsprechen, weil sonst keine vollständige Rotation garantiert ist.[20]

2.2.3 Konkatenation von Rotationen

Natürlich ist es auch möglich zwei oder mehrere Rotationen nacheinander durchzuführen. Dazu muss man das Ergebnis, welches aus der ersten Quaternionenrotation entsteht hernehmen und wiederum um die gewünschte Achse rotieren.

$$Rot(p) = (q_2 q_1)p(q_1^{-1}q_2^{-1}) \tag{24}$$
$$= (q_2(q_1 p q_1^{-1})q_2^{-1}) \tag{25}$$
$$= q_2 p_1 q_2^{-1} \tag{26}$$

Dies ist auch mit der Rotationsmatrix möglich, indem man zwei Matrizen miteinander multipliziert[6]. Betrachtet man dabei aber die benötigen Rechenschritte merkt man, dass diese bei der Quaternionenvariante weniger sind. Beispielsweise braucht eine Rotationsmatrix bei zwei Rotationen 3*3*3 Multiplikationen und 3*3*2 Additionen, wohingegen Quaternionen mit 4*4 Multiplikationen und 4*3 Additionen auskommen, was vor allem bei größeren Berechnungen einen signifikanten Unterschied macht.[20]

2.3 Nach- und Vorteile

Nachteile

Ein großer Nachteil von Quaternionen ist, dass man diese nur für Rotationen verwenden kann. Heißt, dass man für andere Berechnungen, wie zum Beispiel eine Translation, auf die Rotationsmatrix zurückgreifen muss, was viele aufwändige Umrechnungsschritte erfordert. Es ist zwar möglich dies auch mit Quaternionen durchzuführen, indem man eine homogene Quaternion erstellt, mit dem Vektor v als Translationsvektor, jedoch ist diese Methode so unpraktisch, dass es in der Regel nicht angewendet wird.
Ein weiterer Nachteil ist, dass Quaternionen nicht zur Standardausbildung gehören. Man müsste sich erst mit der Materie auseinandersetzen und sich das Wissen aneignen um es anwenden zu können. Da es jedoch anfänglich etwas kompliziert erscheint, greift man für Rotationen oftmals lieber auf simplere Euler Winkel zurück. Außerdem wird es dadurch auch zum Beispiel von vielen Grafikpaketen nicht unterstützt.[19]

Vorteile

Es gibt jedoch viele Vorteile. Eine Quaternionenrotation muss sich an keine Reihenfolge der Achsen halten, denn sie sind unabhängig vom Koordinatensystem. Es kann also sofort um die Drehachse drehen, die gewünscht ist. Dadurch haben Quaternionen auch kein Problem mit dem "Gimbal Lock".
Außerdem ist das Zusammenfügen von Rotationen deutlich effektiver als bei der Transformationsmatrix.
Die einzelnen Umrechnungen der Quaternionen, wie zum Beispiel die Inversenbildung oder eine Normalisierung, sind ebenfalls effizienter als mit einer Transformationsmatrix, was auch unschwer zu erkennen ist, da man bei der Matrix 3*3 = 9 Rechnungen durführen muss und bei der Quaternion nur 4.
Der wichtigste Vorteil jedoch ist die Möglichkeit leicht Quaternionen zu interpolieren und approximieren. Dazu gibt es einige Möglichkeiten, die in den folgenden Kapiteln gezeigt werden.[19]

3 Slerp

Erhält man nach einer Rotation nun zwei Quaternionen auf einem Einheitskreis, muss man einen Weg finden, um die Werte, die dazwischenliegen zu finden Eine gute Methode ist die sphärische lineare Interpolation (SLERP). Dies ist eine lineare Interpolation, die auf der Oberfläche einer Kugel durchgeführt wird. Es sind zwei Punkte gegeben. Der Startpunkt und der Endpunkt mit einem Interpolationsparameter $t \in [0, 1]$, wodurch für jeden Wert von t ein Punkt zwischen dem Start- und Endwert gefunden wird. Die Werte von t sind hierbei gleichmäßig zwischen 0 und 1 verteilt. So entsteht eine gebogene Kurve, die das Wandern der Rotation genauer beschreibt. Bei der Verwendung von ausschließlich Einheitsquaternionen erhält man die Zwischenwerte auf einem Einheitskreis durch folgende Formel. Das Ergebnis bleibt dadurch ebenfalls eine Einheitsquaternion, weshalb keine weiteren Normalisierungen notwendig sind.

$$SLERP(q_1, q_2, t) = q_1(q_1^{-1}q_2)^t \tag{27}$$

Dabei gilt die Eigenschaft, dass bei t = 0 das Ergebnis das Ausgangsquaternion q_1 ist und bei t = 1 das Endquaternion q_2. Dies lässt sich leicht durch einsetzen zeigen.[9] [18]

$$SLERP(q_1, q_2, 0) = q_1(q_1^{-1}q_2)^0 = q_1 \tag{28}$$
$$SLERP(q_1, q_2, 1) = q_1(q_1^{-1}q_2)^1 = q_1q_1^{-1}q_2 = q_2 \tag{29}$$

Des Weiteren gilt, dass alle Teilergebnisse, die von jedem t aus dem SLERP ausgerechnet werden, zusammengezählt 1 ergeben.

$$||SLERP(q_1, q_2, t)|| = ||q_1||\ ||(q_1^{-1}q_2)^t|| = 1 \tag{30}$$

Der große Unterschied hierbei zur gewöhnlichen linearen Interpolation(LERP) ist der, dass der Winkel zwischen den Quaternionen immer gleich bleibt. Bei LERP werden die Winkel in der Mitte größer, was bedeutet, dass sich die Kurve dort schneller bewegt wodurch ungewünschte Nebeneffekte bei z.B. Animationen entstehen können. Dies erkennt man sehr gut in Figure 1, denn hier wird bei der linearen Interpolation eine Gerade vom Startpunkt direkt zum Endpunkt gezogen. Diese Gerade wird dann in gleich große Stücke geteilt, wodurch vom Ursprung aus eine weitere Gerade gezogen wird, die die Punkte auf der Anfangsgeraden schneiden. So entstehen in der Mitte immer größere Winkel, was beim nebenstehendem SLERP nicht der Fall ist.
Die gleichbleibende Geschwindigkeit beim SLERP wird gezeigt, indem die zweite Ableitung der Formel (27) ausgerechnet wird und der daraus entstandene Vektor, parallel zum ursprünglichen Kurvenvektor ist.

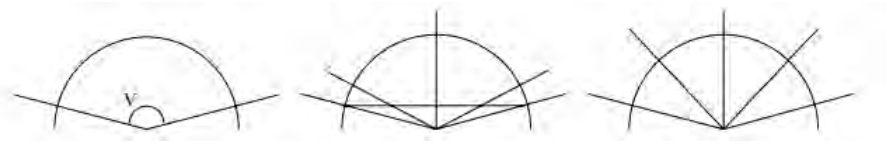

Figure 1: Bernsteinpolynome vom Grad n = 4, [10]

$$\frac{d}{dt}SLERP(q_1, q_2, t) = \frac{d}{dt}q_1(q_1^{-1}q_2)^t = SLERP(q_1, q_2, t)log(q_1q_2) \tag{31}$$

$$\frac{d^2}{dt^2}SLERP(q_1, q_2, t) = SLERP(q_1, q_2, t)log(q_1q_2)^2 \tag{32}$$

Ist der Wert von $log(q_1q_2)^2$ eine reele Zahl die kleiner oder gleich null ist, so hat die Kurve eine gleichbleibende Geschwindigkeit.
Außerhalb von Quaternionen lässt sich der SLERP auch auf einfache Vektoren anwenden. $p_1, p_2 \in \mathbf{R}^n$ werden hierbei entlang eines Kreises ineinander überführt.

$$SLERP(p_1, p_2, t) = \frac{sin(1-t)\Omega}{sin(\Omega)}p_2 + \frac{sin(t\Omega)}{sin(Omega)}p_2 \tag{33}$$

$$\Omega = \arccos(p_1p_2) \tag{34}$$

[10],[9]

Jedoch ist der SLERP-Algorithmus teilweise eingeschränkt. Die Kurve sucht sich immer den kürzesten Weg zum Ziel, weshalb Rotationen größer als π falsch dargestellt werden. Somit entspricht z. B. eine gewünschte Rotation von 270°einer Rotation von -90°. Dies kann man zeigen indem man beweist, dass der Winkel zwischen dem Anfangsquaternion und dem Quaternion, welches bei t = 0.5 berechnet wird, einen Wert von $]-\frac{\pi}{2}, \frac{\pi}{2}]$ hat. Um den Winkel Ω zu erhalten wendet man die Funktion von Formel (34) an und formt die Quaternionen auf eine Seite um:

$$\cos(\Omega) = q_1 * q_{\frac{1}{2}} \tag{35}$$

$$= \cos(\Omega) = q_1 * SLERP(q_1, q_2, \frac{1}{2}) \tag{36}$$

Setzen wir nun die Formel von SLERP von (27) hinein erhalten wir folgende Darstellung:

$$\cos(\Omega) = q_1 * q_1(q_1^{-1}q_2)^{\frac{1}{2}} \tag{37}$$

Da der Wert von $q_1^{-1}q_2$ in der Klammer ein Einheitsvektor ist, können wir dies umschreiben in $q_1^{-1}q_2 = [\cos\theta, n\sin\theta]$, wobei der Winkel θ einen Wert von $]-\pi, \pi]$

besitzt.

$$\cos(\Omega) = q_1 * (q_1[\cos(\theta), n\sin(\theta)]^{\frac{1}{2}}) \tag{38}$$
$$= \cos(\Omega) = q_1 * (q_1[\cos(\theta/2), n\sin(\theta/2)]) \tag{39}$$
$$= \cos(\Omega) = q_1[1,0] * (q_1[\cos(\theta/2), n\sin(\theta/2)]) \tag{40}$$
$$= \cos(\Omega) = ||q_1||^2 * \cos(\theta/2) \tag{41}$$
$$= \cos(\Omega) = \cos(\theta/2) \tag{42}$$

Jetzt sieht man, dass der Winkel Ω im oben beschriebenen Bereich $]-\frac{\pi}{2}, \frac{\pi}{2}]$ bleibt und somit sichergestellt ist, dass die Kurve immer den kürzesten Weg liefert.

Eine weitere Einschränkung ist die, dass eine Rotation um genau π keinen Effekt haben wird, da die Punkte sich damit antipodal zu einander befinden, was bedeutet, dass p_1 und p_2 sich auf dem Einheitskreis direkt gegenüberstehen. Grund dafür ist, dass der SLERP gar nicht genau weiß, wo er hingehen soll, da es unendlich viele kürzeste Wege gibt. Abhilfe schafft dabei ein oder mehrere Zwischenquaternionen, welche zwischen dem Start- und dem Endquaternion platziert werden.[10]

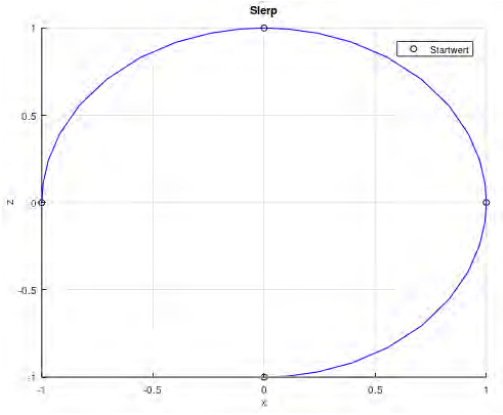

Figure 2: 270° Rotation mit Zusatzquaternionen

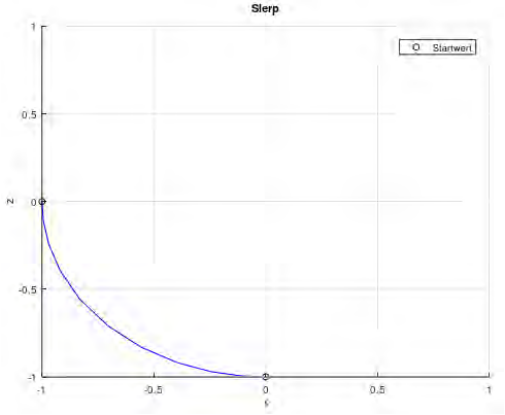

Figure 3: -90° Rotation ohne Zusatzquaternionen

4 Approximations- und Interpolationsalgorithmen

Dieses Zwischenquaternion kann jedoch in den meisten Fällen dazu führen, dass die Winkelgeschwindigkeit zwischen den einzelnen SLERP-Kontrollpunkten unterschiedlich ist, was dazu führt, dass keine glatte Kurve entsteht.[10] Um das zu verhindern, gibt es verschiedene Algorithmen, um glatte Kurven aus den SLERP-Werten zu erstellen. Hierbei unterscheidet man zwischen der approximierten und der interpolierten Kurve. Bei der Approximation wird die Laufbahn der entstandenen Kurve von den Kontrollpunkten beeinflusst, jedoch werden diese Punkte, bis auf den Anfangs- und Endpunkt, nie geschnitten. Die Interpolation hingegen sucht das Polynom, welches durch jeden einzelnen Kontrollpunkt läuft.[7]

4.1 Bezier-Kurve/de Casteljau Algorithmus

Die Bezier-Kurve wurde unabhängig voneinander von den zwei Mathematikern Paul de-Casteljau und Pierre Bezier parallel entdeckt. Dies führte bei der Veröffentlichung zu einem Streitfall, was sich auf die Namensgebung des Algorithmus ausgewirkt hat. Somit lautet die allgemeine Formel und die Kurve selbst Bezier-Kurve und das Konstruktionsprinzip de Casteljau-Algorithmus.[26]

4.1.1 Verfahren

Gegeben ist ein Kontrollpolygon mit einer Reihe an Kontrollpunkten $p_0, p_1, ..., p_n$. Daraus wird der gewünschte Punkt C(t), $t \in [0, 1]$, der immer in der konvexen Hülle der Kontrollpunkte liegt, ermittelt. Die Formel für Bezier-Kurve n-ten Grades wird wie folgt dargestellt:

$$C(t) = \sum_{i=0}^{n} B_i^n(t) p_i \tag{43}$$

Dabei steht p_i für die einzelnen Kontrollpunkte und $B_i^n(t)$ beschreibt das i-te Bernsteinpolynom n-ten Grades, welches in Formel (44) aufgeführt ist. Es gilt als Grundelement für die Erstellung von Bezierkurven

Es sei $t \in [0, 1]$, i = 0,...,n dann ergeben sich die Zwischenpunkte durch die Formel:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{44}$$

dabei ist

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} \tag{45}$$

In der folgenden Abbildung 4 ist eine Reihe an Bernsteinpolynomen bis Grad n = 4 gegeben. Daraus lassen sich einige Eigenschaften ablesen.

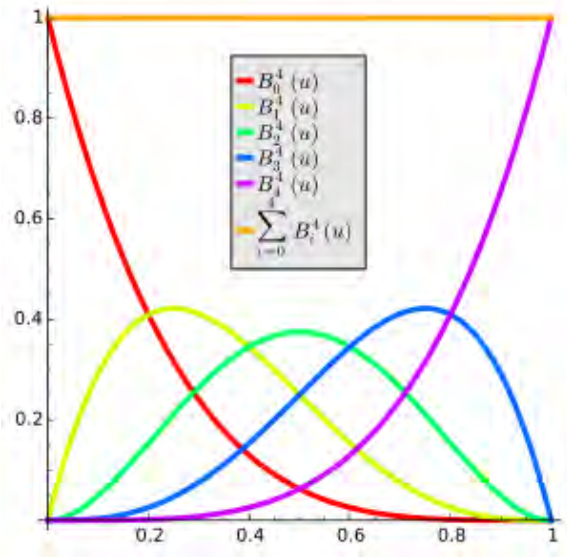

Figure 4: Bernsteinpolynome vom Grad n = 4 [17]

Durch den binomischen Grundsatz erkennt man, dass das Bernsteinpolynom eine Teilung der eins ist.

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} = (t + (1-t))^n = 1 \tag{46}$$

Dabei ist es linear unabhängig und bildet den Raum für Polynome dessen Grad kleiner oder gleich n ist und das Ergebnis bleibt für alle $t \in [0, 1]$ immer positiv. Außerdem lässt sich das Bernsteinpolynom auch rekursiv darstellen als:

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + t(B_{i-1}^{n-1}(t)) \tag{47}$$

Eine symmetrische Anordnung der Polynome ist ebenfalls vorhanden:

$$B_i^n(t) = B_{n-i}^n(1-t) \tag{48}$$

Jedes der Polynome hat dabei genau ein Maximum welches bei $t = \frac{i}{n}$ liegt.[21][17]

4.1.2 de Casteljau Algorithmus

Zur einfacheren und stabileren Berechnung entwickelte man den de Casteljau Algorithmus, welcher eines der bekanntesten Verfahren zur iterativen Kalkulation von Bezier Kurven ist. Es sei $p_i^0 = p_i$, i = 0,...,n dann ergeben sich Schrittweise die Punkte durch die Formel:

$$p_k^j(t) = (1-t)p_k^{j-1}(t) + tp_{k+1}^{j-1}(t) \tag{49}$$

für j = 1,...,n und k = 0,...,n-j.
Was man daraufhin in die Formel von (49) einfügt, bis man schließlich den gewünschten Punkt erhält.

$$C(t) = p_0^n \tag{50}$$

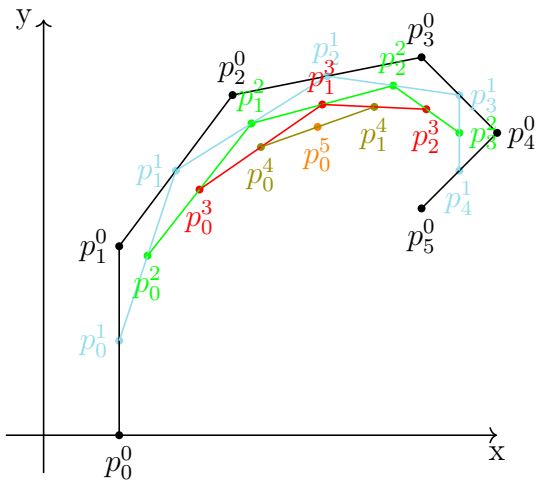

Figure 5: de-Casteljau Algorithmus für n = 5 und t = 0.5 [3]

Im Grunde genommen wird nach jedem Iterationsschritt aus den Kontrollpunkten p_i^0, durch Teilen der Verbindungslinien, neue Kontrollpunkte p_{i-1}^1 erstellt, durch die man wiederum einen neuen Iterationsschritt starten kann. Bis schließlich nur noch eine einzelne Gerade übrig bleibt, auf der der finale Punkt C(t) gesucht wird (siehe Figure 5). Der Wert t bestimmt das Teilungsverhältnis, wo der neue Punkt auf der Geraden von p_i^j bis p_{i+1}^j liegen soll. So sieht man in der Abbildung, dass nach der fünften Iteration der Wert p_0^5 auf Position t = 0.5 auf der Bezier Kurve gefunden wurde.

Wenn das nun mit mehreren Werten von $t \in [0, 1]$ durchführt wird, erhält man die vollständige Bezier-Kurve. [3][21]

Eigenschaften der Bezier-Kurve

Die Anfangs- und Endpunkte der Bezier Kurve sind die einzigen Werte, die auf den Anfangskontrollpunkten liegen. Grund dafür ist, dass für t = 0 das Ergebnis von $B_i^n(t)$ für i = 0 gleich 1 ist und für alle weiteren Werte $1 <= i <= n$ gleich 0. Parallel dazu gilt für t = 1, i = n dass der Wert von $B_i^n(t)$ gleich 1 ist und für alle weiteren

Werte von $0 <= i <= n-1$ gleich 0.

$$C(0) = \sum_{i=0}^{n} B_i^n(0) p_i = b_0 \tag{51}$$

$$C(1) = \sum_{i=0}^{n} B_i^n(1) p_i = b_n \tag{52}$$

Dazwischen folgt die Kurve dem Verlauf den Kontrollpolygons. Durch die Eigenschaft der Zerlegung der eins des Bernsteinpolynoms, ist es ersichtlich, dass die Kurve dabei die konvexe Hülle, die man durch das Verbinden der Kontrollpunkte erhält, zu keinem Zeitpunkt überschreitet. Eine weitere Besonderheit, die dadurch entsteht, ist die affine Invarianz der Bezier-Kurve. Das bedeutet, dass man eine Verschiebung, Streckung oder Spiegelung dieser nur erreicht, indem man die entsprechenden Kontrollpunkte der Bezierkurve affin transformiert.

$$C(t) = \sum_{i=0}^{n} B_i^n(t) p_i \quad wird\ zu, \tag{53}$$

$$\phi(C(t)) = \sum_{i=0}^{n} B_i^n(t) \phi(p_i) \tag{54}$$

Außerdem weist die Bezier-Kurve eine Varianzreduktion auf. Genauer gesagt schwankt die Kurve höchstens so stark wie das Kontrollpolygon.
Für eine beliebige Gerade g gilt, dass sie die Bezier-Kurve nur höchstens so oft schneidet wie das Kontrollpolygon:

$$(Schnittpkt\ von\ g\ mit\ Kurve) <= (Schnittpkt\ von\ g\ mit\ Kontrollpolygon) \tag{55}$$

[5][23]

4.2 B-Spline Kurve/ de-Boor Algorithmus

Spline bedeutet aus dem Englischen übersetzt "das Kurvenlineal" und wurde noch vor der Zeit der Computermodellierung im Schiffsbau angewendet.[15] 1946 wurden B-Splines erstmals von I.J. Schoenberg zur Glättung von statistischen Daten verwendet, was den Anfang der Spline-Approximation darstellte. Um diese einfacher berechnen zu können entdeckte C. de Boor ein paar Jahrzehnte später eine bahnbrechende Rekursionsformel für die B-Spline Kurve, die in dieser Arbeit vorgestellt wird.[24][13]

4.2.1 Normales Verfahren

Der sogenannte de-Boor Algorithmus ist im Grunde genommen eine Erweiterung oder auch Verallgemeinerung des de Casteljau Algorithmus. Dieser dient zur Berechnung von B-Spline Kurven S(t). Ähnlich wie bei der Bezier-Kurve wird der B-Spline

k-ten Grades nach folgender Formel bestimmt:

$$S(t) = \sum_{i=0}^{n} N_i^k(t) p_i, \qquad t \in (t_i, t_{i+1}) \tag{56}$$

Die Gradanzahl kann hier je nach Belieben gewählt werden. In der Regel ist dies jedoch k = 3. Dabei sind $p_0, ..., p_n$ die Kontrollpunkte, $T = (t_0, t_1, ..., t_m)$ mit m = k + n +1 die Knotensequenz und $N_i^k(t)$ die Basissplinefunktion mit Kontinuitätseigenschaften. Diese wird folgendermaßen rekursiv berechnet.

$$N_i^0(t) = \begin{cases} 1, & \text{wenn } t_i <= t < t_{i+1} \\ 0, & \text{sonst} \end{cases}$$

$$N_i^k(t) = \frac{(t - t_i)}{(t_{i+k-1} - t_i)} N_i^{k-1}(t) + \frac{(t_{i+k} - t)}{(t_{i+k} - t_{i+1})} N_{i+1}^{k-1}(t)) \tag{57}$$

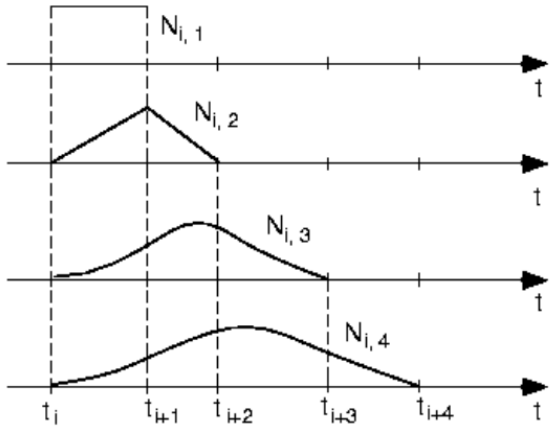

Figure 6: Basissplinepolynome vom Grad n = 1,2,3 und 4 [1]

Die normalisierten B-Splines $N_i^k(t)$ haben nun die Eigenschaft, dass sie aus Polynomen vom Grad k über die Knotensequenz T stückweise zusammengestellt sind.
Für alle Werte von $t \in [t_0, ..., t_{k+n+1}]$ gilt wie bei der Bernsteinfunktion $N_i^k(t) <= 1$ und die Positivität $N_i^k(t) >= 0$. Die Ergebnisse daraus ergeben beim Aufsummieren einen Wert von 1.
Außerdem gilt, dass $N_i^k(t)$ einen lokalen Träger besitzt, wodurch Werte von t die sich außerhalb von $[t_l, t_{l+1}]$ befinden von der Basissplinefunktion als null definiert werden. Je größer hier der Grad k ist, desto größer ist auch der lokale Träger. Das erkennt man sehr gut in Abb:6, denn jedes Polynom bekommt bei steigendem k immer mehr Einfluss auf die einzelnen Intervalle der B-Spline Kurve. [27]

4.2.2 de Boor Verfahren

Basierend auf der obigen Formel wurde von de Boor ein erweiterter Algorithmus entdeckt. Dazu hat er die Definition von $N_i^k(t)$ von (57) in die rechte Seite der Formel von (56) substituiert. Durch vereinfachen erhielt man dann eine Formel für $N_i^{k-1}(t)$, also ein Grad weniger als davor.

Erste Iteration:

$$\begin{aligned} S(t) &= \sum p_i N_i^k(t) \\ &= \sum p_i (w_{i,k}(t) N_i^{k-1}(t) + (1 - w_{i,k-1}(t)) N_{i+1}^{k-1}(t)) \\ &= \sum (w_{i,k}(t) p_i + (1 - w_{i,k-1}(t)) p_{i+1}) N_i^{k-1} \\ &= \sum p_i^1 N_i^{k-1}(t) \end{aligned} \tag{58}$$

wobei

$$w_{i,k}(t) = \frac{(t - t_i)}{(t_{i+k-1} - t_i)}, \quad 1 - w_{i,k-1}(t) = \frac{(t_{i+k} - t)}{(t_{i+k} - t_{i+k})} \tag{59}$$

und

$$p_i^1 = w_{i,k}(t) p_i + (1 - w_{i,k-1}(t)) p_{i+1} \tag{60}$$

Diese neue Definition konnte man nun wiederum in (56) substituieren, bis man schließlich bei $N_i^1(t)$ angekommen ist.
Da $N_i^1(t)$ folgendermaßen definiert ist

$$N_i^1(t) = \begin{cases} 1, & \text{wenn } t \in [t_i, t_{i+1}] \\ 0, & \text{sonst} \end{cases}$$

erhalten wir als Ergebnis für B-Spline Kurven erster Ordnung:

$$S(t) = p_i^{k-1}(t), \quad t \in [t_i, t_{i+1}] \tag{61}$$

Daraus ergibt sich folgender Algorithmus:
Sei $T = (t_0, t_1, ..., t_{k+n+1})$ die Knotenfolge, wobei n die Anzahl der Kontrollpunkte $p_0, ...p_n$ ist und m ein Wert der um ein Vielfaches kleiner ist als n. Des Weiteren bestimmen wir $r \in [k, ..., n]$ und somit auch die Stelle an der wir den gewünschten ausgegebenen Punkt haben möchten $t \in [t_r, t_{r+1}]$. Als nächstes definieren wir die ersten Punkte:

$$p_i^0(t) = p_i, \tag{62}$$

für i = r-k+1,...,r
Die restlichen Punkte erhält man folgendermaßen mit j = 1,...,m-1:

$$p_i^j(t) = (1 - w_i^{k-j+1})p_i^{j-1}(t) + w_i^{k-j+1}p_{i-1}^{j-1}(t) \tag{63}$$

für i = r-k+j+1,...,r.
Bis man den gewünschten Punkt erhält:

$$S(t) = p_r^k(t) \tag{64}$$

[25][22]

Vergleich zu de Casteljau

Man erkennt nun, dass der de Boor Algorithmus ziemlich ähnlich zum de Casteljau Algorithmus ist. Ordnet man die Knotenpunkte sogar in folgender Form an: $T = (a, ..., a, b, ..., b)$, wobei die Anzahl von a und b jeweils k+1 ist, bildet der de Boor eine einfache Bezier Kurve. Jedoch ändert sich dies sobald es dazwischen Knoten gibt, die größer als a und kleiner als b sind. Denn nun ermittelt der de Boor Algorithmus den gewünschten Punkt nicht mehr über alle Kontrollpunkte, wie es beim de Casteljau üblich ist, sondern über bestimmte Intervalle, die von k+1 Kontrollpunkten bearbeitet werden. Dadurch werden zwar mehrere Berechnungsschritte durchgeführt, jedoch fällt die Kurve am Ende genauer aus.
Im folgenden Abbildung 7 ist ein de Boor Pyramidenschema angegeben. Die benötigten Kontrollpunkte werden durch p_{r-k} bis p_r definiert. Mit der aus der Knotensequenz genommenen Werte $[t_r, t_{r-1})$.

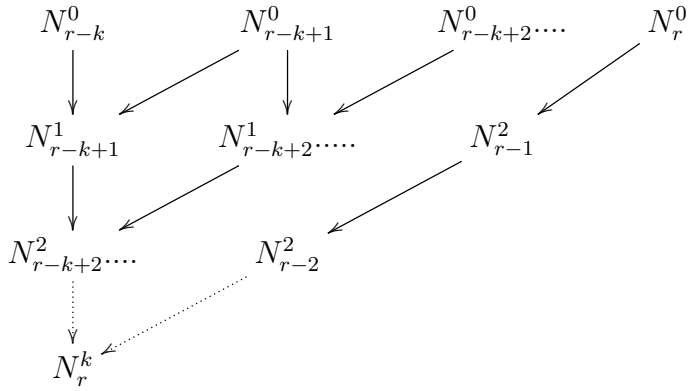

Figure 7: de-Boor Algorithmus Pyramidenschema für k = 4 [27]

Des Weiteren bleiben die beim de Casteljau verwendeten Zahlen $(1-t)$ und t aus (49) pro Berechnungsschritt unverändert, je nachdem welche Stelle man als Ergebnis haben möchte, wohingegen die Knotenpunkte den Wert von $(1-w)$ und w aus (56) bei jedem erneuten Aufrufen der Funktion verändern.[27] [1] [25]

Dies wird deutlicher, wenn es an einer Sequenz von Kontrollpunkten bei Abbildung 8 durchgeführt wird. In der unteren Darstellung sehen wir sechs Kontrollpunkte. Nehmen wir an wir wollen einen de Boor Algorithmus mit Grad k = 3 darauf anwenden. Dazu verwenden wir einen entsprechenden Knotenvektor
T = [0,0,0,0,0.25,0.75,1,1,1,1]. Als zu berechnende Stelle nehmen wir t = 0.5. Vergleichen wir das jetzt mit dem Knotenvektor merken wir, dass unsere gewählte Stelle t zwischen dem Knoten t_4 und t_5 liegt. Anhand der obigen Definition erhalten wir für r = 4, also sind die Kontrollpunkte, die Einfluss auf den Punkt nehmen p_1, p_2, p_3, p_4. Als Ergebnis erhält man folgenden Berechnungsverlauf:

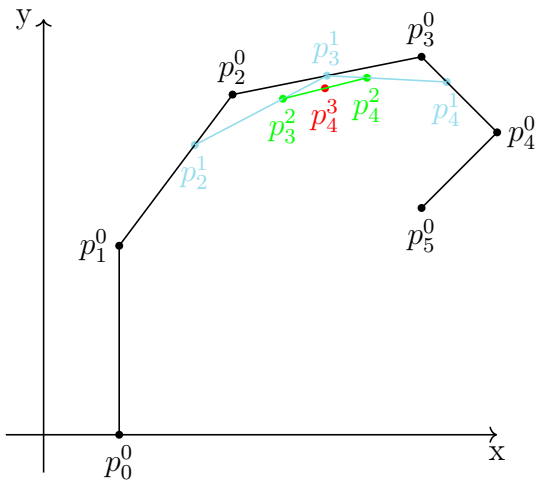

Figure 8: de-Boor Algorithmus mit Grad k = 3 und Knoten T = [0,0,0,0,0.25,0.75,1,1,1,1] an der Stelle t = 0.5 [2]

[2]
Die konvexe Hülle verkleinert sich dabei durch die Berechnungsintervalle. Denn diese liegt nun für $t_r <= u <= t_{r+1}$ im Raum der k+1 Punkte $t_{r-k}, ..., t_k$, während beim de Casteljau sie sich über alle Kontrollpunkte erstreckt. Somit wird die komplexe Hülle, die in den folgenden Darstellungen hellgrau dargestellt ist, folgendermaßen aufgebaut.
[27]

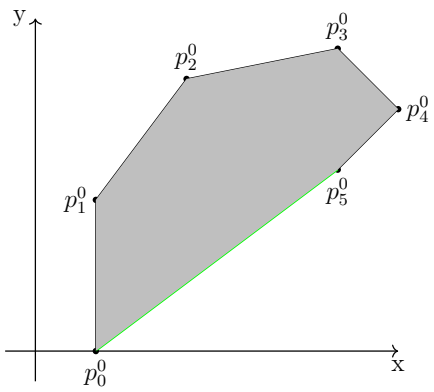

Figure 9: Konvexe Hülle Bezier-Kurve [27]

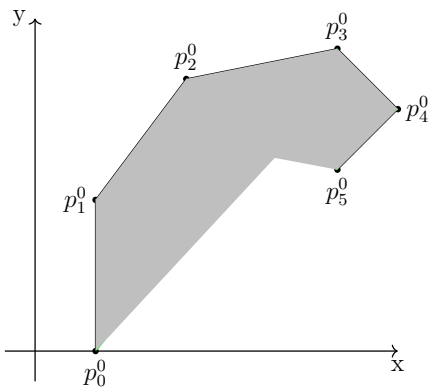

Figure 10: Konvexe Hülle B-Spline Kurve k = 3 [27]

4.2.3 Weitere Eigenschaften der B-Spline Kurve

Noch mehr Eigenschaften entstehen durch besondere Gegebenheiten der Kontrollpunkte oder der Knoten. Beispielsweise kann es sein, dass sich bei einer B-Spline Kurve vom Grad k, k Kontrollpunkte an derselben Stelle befinden $p_{r-k+1}, ..., p_r = p$. Ist dies der Fall wird die Kurve durch diesen Punkt interpolieren.
Eine weitere interessante Eigenschaft ist, wenn k+1 Kontrollpunkte auf der gleichen Geraden liegen. Nehmen wir also an wir haben $p_{r-k}, ..., p_r$ Punkte, die die gerade genannte Eigenschaft besitzen, dann liegen die Punkte t die durch $t_r <= t <= t_{r+1}$ gehen auf der Geraden und stimmen somit teilweise mit dem Kontrollpolygon überein.
Auch die Wahl der Knoten ruft Eigenschaften hervor. Stimmt zum Beispiel ein gesuchter Wert an Stelle t mit dem Wert eines Kontrollpunktes überein, dann läuft dieser dort tangential am Kontrollpolygon entlang. Dasselbe gilt vor allem für die Anfangs- und Endkontrollpunkte, weshalb sichergestellt werden soll, dass die Kontrollpunkte so gewählt werden, dass die am Anfang und am Ende mit dem Wert t übereinstimmen, um die gewünschte Kurve zu erhalten.[27]

4.3 Aitken-Neville Algorithmus

Nachdem die letzten zwei Algorithmen Approximationen waren, ist es nun interessant den Kurvenverlauf einer Interpolation zu sehen. Hierzu betrachten wir den Aitken-Neville Algorithmus.
Die Formel für diesen Algorithmus ähnelt wieder sehr stark der von de Casteljau. Einzig die Intervalle, mit denen die gewünschten Punkte berechnet werden sind anders. Es sei j = 0 und k = 0,...,n dann sind die Startpunkte.

$$p_k^0 = p_k \tag{65}$$

Alle weiteren Polynome erhällt man durch folgende rekursive Funktion

$$p_k^j(t) = qp_k^{j-1}(t) + (1-q)p_{k+1}^{j-1}(t) \tag{66}$$

für j = 1,...,n und k = 0,...,n-j-1, wobei

$$q = \frac{x(j+k)-t}{x(j+k)-x(k)}, \qquad 1-q = \frac{t-x(k)}{x(j+k)-x(k)} \tag{67}$$

Dabei ist t der Punkt wo das Ergebnis ausgerechnet werden soll. Die Werte x sind eine Liste von Werten $x \in [x_0, x_1, ..., x_n]$. Dabei ist $x_j < x_{j+1}$. Das j bestimmt die Länge von k. Die wird pro Iteration immer kleiner und kleiner.
Daraus erhält man folgendes Polynom m

$$m(t) = p_0^n \tag{68}$$

mit der Interpolationseigenschaft

$$m(t_k) = p_k \tag{69}$$

für k = 0,...,n

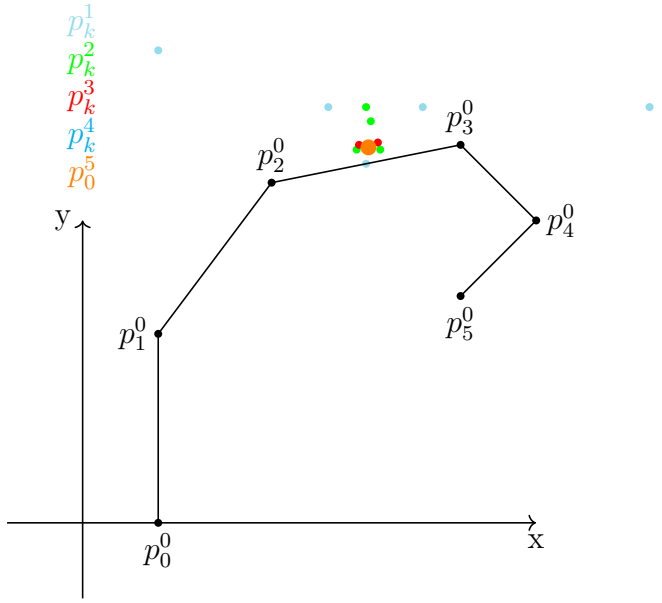

Figure 11: Aitken-Neville Algorithmus für Punkt t = 0.5(orange)

In Abbildung 11 sieht man, wie sich die einzelnen Punkte für t = 0.5 Schritt für Schritt zusammenfügen, was ein Unterschied ist zu den vorherigen Algorithmen. Denn bei jeder Iteration wird beim Aitken Neville ein Zwischenpolynom erstellt. Zum Beispiel wird aus den ersten beiden Punkten p_0^0 und p_1^0 ein Polynom $p_{0,1}^1$

berechnet, womit die Kurve die am Endergebnis herauskommen soll schon leicht angedeutet wird. Dieses Polynom wird dann für die weitere Iteration verwendet woraus man dann $p_{0,1,2}$ bekommt, was dem gewünschten Ergebnis noch näher kommt. Dies geschieht dann so lange, bis am Ende ein Polynom entsteht, welches alle Kontrollpunkte mit einspannt. Setzt man dort nun die Punkte t ein erhalten wir die Aitken-Neville Interpolation. Dies lässt sich auch hier mit einem Pyramidenschema darstellen.[25][14]

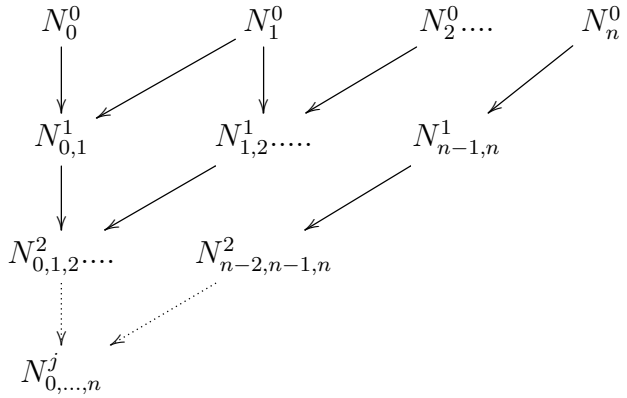

Figure 12: Aitken-Neville Algorithmus Pyramidenschema mit n Kontrollpunkten [25]

Jedoch hat dieser Algorithmus auch ein paar Nachteile. Zum einen gibt er zwar das Ergebnis aus, zeigt aber die Zwischenpolynome nicht an. Das heißt dadurch ist es nicht möglich diese zu manipulieren oder abzuleiten. Des Weiteren ist die Laufzeit sehr hoch, weshalb diese Methode in der Praxis meist nur dazu angewendet wird um einen einzigen gewünschten Punkt t zu berechnen.[25]

5 Umsetzung der Algorithmen

Um nun die oben genannten Methoden praktisch umzusetzen, muss es mithilfe einer Software angewendet werden. Dazu habe ich mich entschieden mit Octave Version 5.2.0 zu arbeiten. Dies erwies sich als sehr praktisch, da Octave das Paket "Quaternionen" enthält, womit man leicht die vierdimensionalen Zahlen verwenden kann.

SLERP

Das Quaternionenpaket konnte man mit dem Befehl "pkg load quaternion" aufrufen. Mithilfe der oben genannten Rotationsformel war es jetzt möglich die ersten Drehungen um die gewünschten Achsen zu berechnen. Aus dem Ergebnis ließ sich nun der erste SLERP programmieren. Dabei ist q1 das Startquaternion und q2 das berechnete rotierte Quaternion.

```
1000 function [q3] = simpleslerp (q1,q2,t)

1002   q3 = (q1*(conj(q1)*q2)^t)

1004 endfunction
```

Hier wird für jedes $t \in [0,1]$ der Wert für das gesuchte Quaternion gefunden. Der Befehl conj(q1) berechnet dabei die komplex Konjugierte von dem Startquaternion. Diese Funktion habe ich nun verwendet, um ein SLERP-Programm zu schreiben, in welches man beliebig viele Quaternion hineinschreiben kann. Aus allen vorhandenen Zahlen wird dann nacheinander ein SLERP erstellt. Dabei ergibt sich die Reihenfolge der SLERPs anhand des Listenplatzes der Quaternionen.

```
1000 function slerpvalue = createSlerpOfAllQuaternions (numberQuaternion, tn
     )
      for p = 1:length(numberQuaternion)-1
1002    slerp = simpleslerp(numberQuaternion(p),numberQuaternion(p+1),0:tn
     :1);
     slerpx(:,p) = slerp.x
1004    slerpy(:,p) = slerp.y
     slerpz(:,p) = slerp.z
1006    slerpvalue = quaternion(slerpx,slerpy,slerpz)
  end
1008 endfunction
```

Jetzt wird anhand einer Schleife jeder Wert von t ausgerechnet. Dazu kann man mit tn die Schrittweite angeben, um die Genauigkeit der Kurve anzupassen. Nachdem der SLERP bei 1002 im Code seine Daten erhalten hat, extrahierte ich davon die x,y und z Komponenten und speicherte diese wieder bei 1006 separat in eine Liste

aus Quaternionen.
Mit dieser Funktion werde ich in den nachfolgenden Algorithmen einen Vergleich zu den erstellen Approximationen und Interpolationen erstellen.

de Casteljau

Zuerst habe die Bezier-Kurve mithilfe des Algorithmus von de Casteljau umgesetzt.

```
%Eingabe: tn = Schrittweite
%         numberQuaternions = Anzahl Quaternionen
function y = deCasteljau(tn, numberQuaternions)
    Qvgl = numberQuternions;
    slerpvalues = createSlerpOfAllQuaternions(numberQuaternions);

     n = length(numberQuaternions)-1;
     t = 0:tn:1;

     for j = 1:1:length(t);
      for p = 1:n;
      for i = 1:n-p+1;
        numberQuaternions(:,i) = unit((1-t(j))*numberQuaternions(:,i) +
    t(j)*numberQuaternions(:,i+1));
        end
        ergebnis = numberQuaternions(:,1);
      end
     numberQuaternions = Qvgl
     gesx(:,j) = ergebnis.x;
     gesy(:,j) = ergebnis.y;
     gesz(:,j) = ergebnis.z;
   end
   PlotSlerpAndFunction(slerpvalues.x,slerpvalues.y,slerpvalues.z, gesx
   ,gesy,gesz);
   title("de Casteljau");
endfunction
```

Als Eingabe erhalten wir wie schon beim Slerp die Schrittweite tn und eine Liste von Quaternionen "numberQuaternions". Danach bestimmen wir wie im Algorithmus beschrieben die Werte für die Schleifen und lassen diese durch den Algorithmus laufen. Das Ergebnis wird wiederum in einzelne Komponenten gesx, gesy und gesz abgespeichert. Die Funktion PlorSlerpAndFunction nimmt die Werte vom SLERP und de Casteljau Algorithmus und gibt einen 3D-Plot von beiden wieder. Mit den selben Hilfsfunktionen werden nun auch der de Boor- und der Aitken-Neville Algorithmus umgesetzt.

de Boor

```
%Eingabe: m = Gradzahl
%         tn = Schrittweite
%         numberQuaternions = Liste von QUaternionen
%         V = Kontrollvektor
function ergebnis = deBoorSpherical(m,tn,numberQuaternion,V)
    Qvgl = numberQuaternion;
    slerpvalues = createSlerpOfAllQuaternions(numberQuaternion);

    n = length(numberQuaternion);
    x = 0:tn:V(m+n+1);
    i = 1:1:length(x);
 for i = 1:length(x);
  r = m+1;
  while (r < n ) && ( x(i) >= V( r+1 ) );
      r = r + 1;
  end
  for j = 1:m;
    for k = r:-1:r-m+j;
      u = ( V( k+m-j+1 ) -x(i))  / ( V( k+m-j+1) - V( k ) );
      numberQuaternion( :,k ) = unit(u * numberQuaternion(k-1) + (1-u)
    * numberQuaternion( k ));
    end
    ergebnis = numberQuaternion( :,r );
  end
  numberQuaternion = Qvgl;
  gesx(:,i) = ergebnis.x;
  gesy(:,i) = ergebnis.y;
  gesz(:,i) = ergebnis.z;

end
    PlotSlerpAndFunction(slerpvalues.x,slerpvalues.y,slerpvalues.z, gesx
    ,gesy,gesz)
    title("deBoor")
endfunction
```

Der große Unterschied hierbei ist wie schon oben erwähnt der Kontrollvektor. Da diese meist recht groß ausfallen können, vor allem bei mehreren Quaternionen, habe ich eine zusätzliche Hilfsfunktion dazu erstellt.

```
% Eingabe:  m = Gradanzahl
%           lengthquaternion = Anzahl Quaternionen
function deboorvector = deBoorVector (m, lengthquaternion)
    lengthvector = m + lengthquaternion +1;
    restofvector = lengthvector - ((m+1)*2)

    %Bestimmung der Randwerte, falls es keine Zwischenwerte gibt,
    besteht der Vektor nur aus Randwerten
    for j = 1:m+1
        deboorvector(j) = 0;
        deboorvector(lengthvector-j+1) = 1;
```

```
    end

    %Spezialfall, falls es nur einen Wert zwischen den Randwerten gibt.
    if restofvector == 1
        deboorvector(m+2) = 0.5;

    %Berchnung der Zwischenwerte wenn es mehr als einen Wert zwischen
    den Randwerte gibt. Diese werden gleichverteilt zwischen 0 und 1
    errechnet.
    elseif restofvector > 1
      sum = 0;
      for l = 1:restofvector
        sum = sum + (1/ (restofvector+1))
        deboorvector(m+1+l) = sum
    end
    endif

endfunction
```

Aitken-Neville

```
% Eingabe: tn = Schrittweite
%          numberquaternions = Liste von Quaternionen
function retval = aitkennevillespherical(xn,numberquaternion)

  Cvgl = numberquaternion;
  f = Cvgl;
  slerpvalues = createSlerpOfAllQuaternions(Cvgl);

  x = 0:xn:1
  n = length(f);
  qlength = AitkenNevilleVector(n)

  for i = 1:1:length(x);
  for j=1:n;
    for k = 1:n-j;
      t = (qlength(k+j)-x(i))/(qlength(k+j)-qlength(k));
      f(:,k) = unit(t * f(:,k) + (1-t) * f(:,k+1));
    end
    y = f(:,1);
  end
  f = Cvgl;
  gesx(i,:) = y.x;
  gesy(i,:) = y.y;
  gesz(i,:) = y.z;
end
```

```
1026    PlotSlerpAndFunction(slerpvalues.x,slerpvalues.y,slerpvalues.z,gesx,
          gesy,gesz)
        title("Aitken-Neville");
1028  endfunction
```

Ebenso wie beim de Boor habe ich hier eine Hilfsfunktion zur Bestimmung der Auswertungspunkte qlength erstellt (Zeile 1010 - Aitken Neville). Dieser erstellt einen gleichverteilten Vektor anhand der Anzahl der Quaternionen mit Startwert 0 und Endwert 1

```
1000  function anVector = AitkenNevilleVector (n)
          sum = 0;
1002      %Bestimmung der Zwischenwerte
          for i = 1:n-2
1004          sum = sum + (1/(n-1))
              zwiVector(i) = sum
1006      end
          %Zusammensetzung des Vektors
1008      anVector = [0 zwiVector 1]
      endfunction
```

Laufzeittests

Anschließend habe ich die Laufzeiten der einzelnen Algorithmen getestet. Dazu verwendete ich den in Octave vorhandenen tic-toc Befehl, um die Zeit in Sekunden zu messen. "Tic" wird dabei an den Anfang des Algorithmus geschrieben, der dann so lange einen Timer laufen lässt, bis "toc" erreicht worden ist, welches am Ende vom Algorithmus steht.
Zum Vergleich testete ich die Laufzeiten von 3-7 Kontrollquaternionen einmal mit der Schrittweite tn = 0.1 und einmal mit tn = 0.01. Für den de Boor Algorithmus wurde m = 2 für Quaternionenanzahl 3 und m = 3 für Quaternionenanzahl 4,5,6,7 verwendet. Der de Boor Vektor war immer gleichmäßig verteilt. Interessant ist

Quaternionenanzahl	de Casteljau	de Boor	Aitken-Neville
3	0.18129	0.17468	0.17961
4	0.31633	0.31306	0.31990
5	0.48847	0.36228	0.47972
6	0.68350	0.42192	0.72487
7	0.95691	0.46105	0.93281

Table 1: Laufzeit mit tn = 0.1 in Sekunden

Quaternionenanzahl	de Casteljau	de Boor	Aitken-Neville
3	1.0324	1.0207	1.0617
4	1.9642	1.9218	2.1375
5	3.2400	1.9943	3.2359
6	4.6101	2.0584	4.7362
7	6.4278	2.1085	6.7428

Table 2: Laufzeit mit tn = 0.01 in Sekunden

hier zu sehen, dass der eigentlich komplexere de Boor Algorithmus besser abschneidet als der de Casteljau Algorithmus. Dies wird vor allem bemerkbar, wenn mehr Quaternionen im Spiel sind. Auch wenn man den de Boor Vektor so anpasst, dass eine Bezier-Kurve herauskommt, erhalten wir dasselbe Ergebnis. Der Aitken-Neville Algorithmus schneidet mit den nahezu denselben Laufzeiten ab wie der de Casteljau. Bei wenigen Quaternionen gibt es jedoch keine großen Unterschiede.

6 Ergebnisse

90°Kurven

Nun ist es interessant zu sehen, was diese Algorithmen ausgeben und wie sie sich verhalten. Zunächst versuchen wir einfache gleichmäßige Rotationen. Wir beginnen mit einem Quaternion 0w+0i+0j+k und rotieren ihn einmal um 90°um die x-Achse, anschließend um 90°um die z-Achse, daraufhin um -90°um die y-Achse und zuletzt um -90°wieder um die x-Achse.

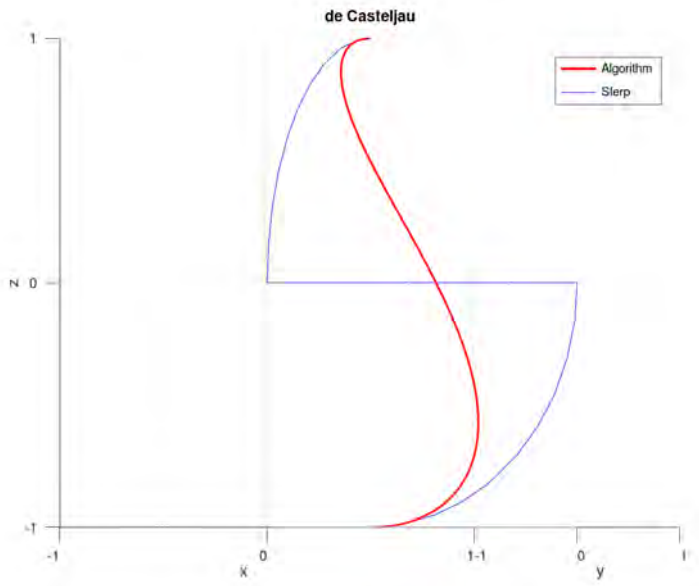

Figure 13: de Casteljau - 90°Winkel

Man erkennt hier sehr schon den Effekt vom de Boor Algorithmus. Während die Bezier Kurve sich seinen Weg direkt von Start- zum Endpunkt sucht, wird die B-Spline Kurve auf der Hälfte des Weges von einem Kontrollpunkt leicht herangezogen. Je mehr Kontrollpunkte man hat, desto mehr verstärkt sich dieser Effekt. Am genauesten trifft die Kurve jedoch der Aitken-Neville Algorithmus.

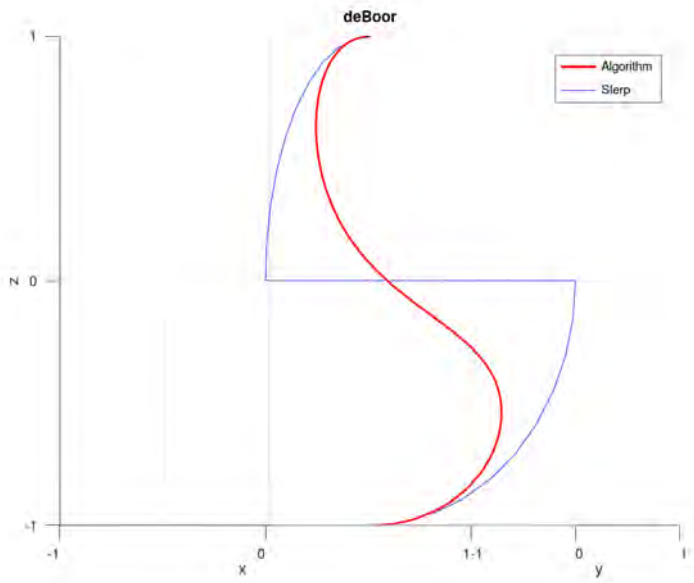

Figure 14: de Boor - 90°Winkel

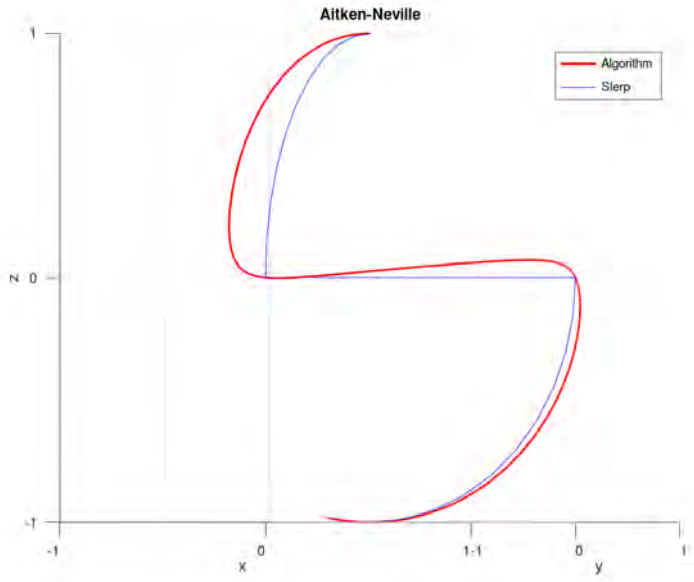

Figure 15: Aitken Neville - 90°Winkel

Spitze Winkel

Bei spitzen Winkeln steht der Aitken-Neville Algorithmus ungünstig da. Denn er muss unbedingt diesen Punkt durchlaufen, wodurch das Endergebnis eine Kurve mit einem ebenfalls sehr spitzen Winkel ergibt. Der de Boor ist dabei einigen Fällen ähnlich wie der Aitken-Neville, vor allem wenn mehrere spitze Winkel nacheinan-

der kommen, aber in den meisten Fällen erhält man eine schöne Kurve. Beim de Casteljau kommt dieses Verhalten eher selten vor.

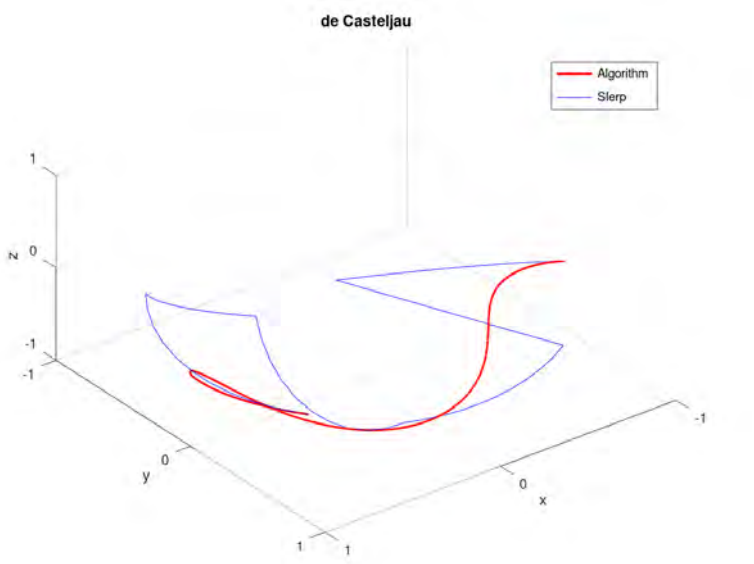

Figure 16: de Casteljau - Spitze Winkel

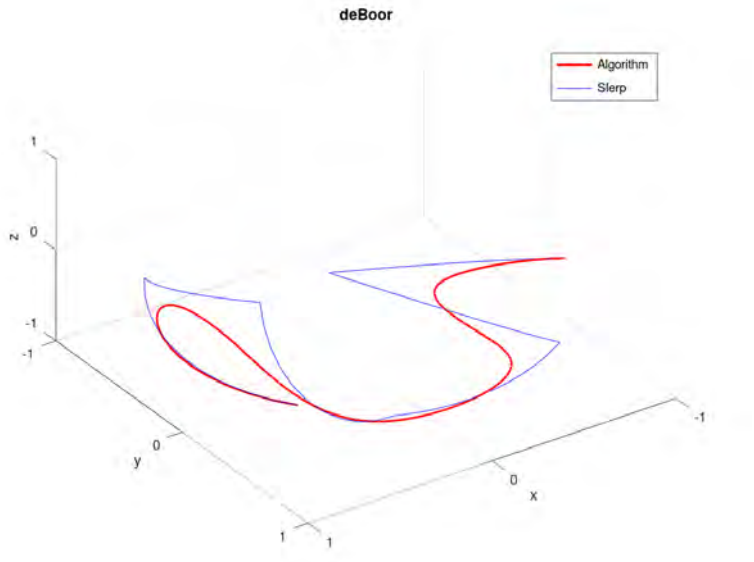

Figure 17: de Boor - Spitze Winkel

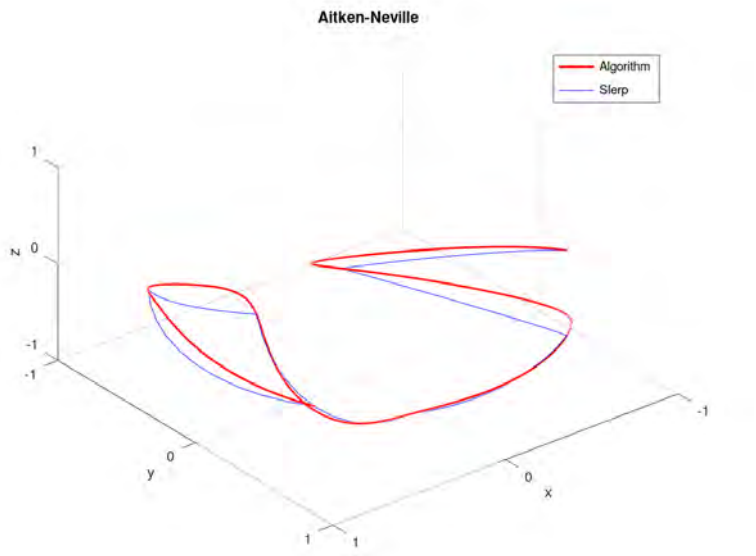

Figure 18: Aitken Neville - Spitze Winkel

Geschlossener Raum

Bei geschlossenen Räumen gibt es das Problem, dass die Kurve beim Anfangs- und Endpunkt. Da dieser ja derselbe sein muss läuft die Kurve dort zweimal durch und da die Kurve zwischen dem Anfangs und Endpunkt nicht interpoliert entsteht dort ein unschöner "Knick" in der Kurve. Dies kann man lösen, indem man die Punkte nach und vor dem Start- bzw. Endpunkt auf derselben Achse platziert. Bei der Interpolation funktioniert dies jedoch nicht, weshalb bei diesen Berechnungen die Benutzung von Approximationen besser ist.

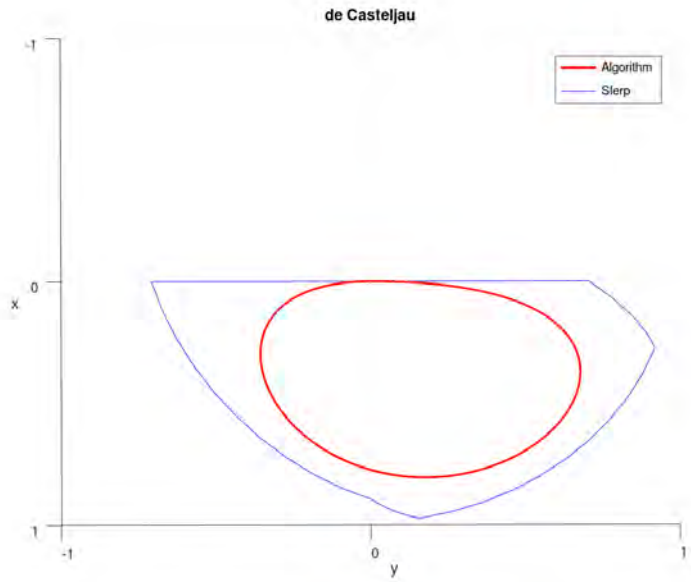

Figure 19: De Casteljau - Geschlossener Raum

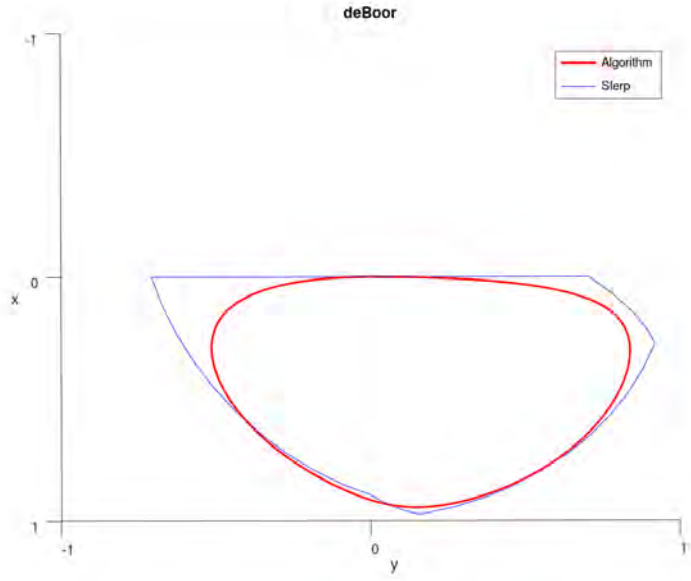

Figure 20: De Boor - Geschlossener Raum

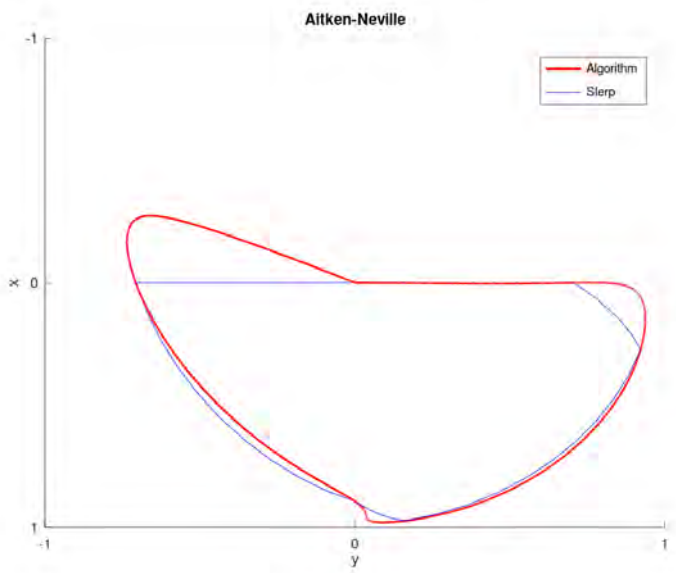

Figure 21: Aitken Neville - Geschlossener Raum

7 Schlussfolgerung

Die Ergebnisse zeigen, dass es möglich ist schöne Approximationen und Interpolationen mit Quaternionen durchzuführen. Dabei kommt der Aitken-Neville Algorithmus am nähesten an die vom SLERP erstellten Ausgangskurven heran. Was einerseits zwar gut ist, jedoch bei mehreren Quaternionen ein wenig chaotisch aussehen könnte. Durch die hohe Laufzeit ist der Algorithmus aber für eine höhere Anzahl an Quaternionen sowieso nicht empfehlenswert. Der de Casteljau ist so ziemlich das genaue Gegenteil vom Aitken Neville. Er ist die ungenaueste Methode, erstellt jedoch auch bei einer hohen Anzahl an Quaternionen eine gut nachvollziebare Kurve. Der große Nachteil ist wie man schon in der Laufzeitanalyse gesehen hat, dass er genauso langsam ist wie Aitken-Neville, weshalb man ihn nur in wirklich dringenden Fällen einsetzen sollte. Ein guter Kompromiss ist hier der de Boor Algorithmus. Ich würde ihn als die Zwischenlösung bezeichnen. Denn er ist nicht so genau wie der Aitken-Neville, jedoch viel genauer als der de Casteljau und kann auch bei mehreren Quaternionen sehr oft akzeptable Kurve abliefern. Nur bei vielen spitzen Winkeln hintereinander stößt der an seine Grenzen. Des Weiteren sticht hervor wegen der besten Laufzeit der drei Algorithmen. Auch wenn dessen Implementierung im Vergleich zu den anderen Methoden bei Weitem die Komplizierteste war.

References

[1] B-Spline-Kurve und - Basisfunktionen. https://tams.informatik.uni-hamburg.de/lehre/2006ss/vorlesung/maschinelles_lernen/folien/06-16-bw.pdf. (aufgerufen: 19.05.2021).

[2] De Boor's Algorithm. https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/de-Boor.html. (aufgerufen: 19.05.2021).

[3] Finding a Point on a Bézier Curve: De Casteljau's Algorithm. https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/de-casteljau.html. (aufgerufen: 19.05.2021).

[4] Quaternionen-Schiefkörper. https://de-academic.com/dic.nsf/dewiki/1146476. (aufgerufen: 19.05.2021).

[5] Khan K./Lobiyal D.C./Kilicman Adem. A de Casteljau Algorithm for Bernstein type Polynomials based on (p, q)-integers. pages 4–9, 2015.

[6] Martin John Baker. Maths - Transformations using Quaternions. https://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/transforms/index.htm. (aufgerufen: 19.05.2021).

[7] Liam Oliver Bast. Rendering von Freiformflächen. page 3, 2018.

[8] Moti Ben-Ari. A Tutorial on Euler Angles and Quaternions. 2014-17.

[9] Toni Barrera/Anders Hast/Ewert Bengtsson. Incremental Spherical Linear Interpolation. https://ep.liu.se/ecp/013/004/ecp01304.pdf.(aufgerufen: 19.05.2021).

[10] Eric B. Dam. Quaternions, Interpolation and Animation. 1998.

[11] David Eberli. Quaternion Algebra and Calculus pages.2-3. https://www.geometrictools.com/Documentation/Quaternions.pdf. (aufgerufen: 19.05.2021), 1999.

[12] Katrin Leschke et al. Quaternionen: von Hamilton, Basketbällen und anderen Katastrophen, 2011.

[13] Gerald Farin. Kurven und Flächen im Computer Aided Geometric Design. page 122, 1994.

[14] Thomas Gerstner. Numerische Mathematik. https://www.uni-frankfurt.de/63288571/vorlesung.pdf. (aufgerufen: 19.05.2021).

[15] Jürgen Hagler. B-splines. http://www.dma.ufg.ac.at/dma/app/link/Grundlagen%3A3D-Grafik/module/13205?step=all. (aufgerufen: 19.05.2021), 2005.

[16] Dr. Ralph Hubner. Rotationen - Der Unterschied zwischen keiner Drehung und einer Drehung um 2π. 2011.

[17] Bianca Högel. Bernsteinpolynom. https://www.biancahoegel.de/mathe/algebra/bernsteinpolynom.html. (aufgerufen: 19.05.2021).

[18] Verena Elisabeth Kremer. Quaternions and SLERP. pages 6–7, 2008.

[19] Robert Lasch. Motion Capturing mit Xsens MVN und InstantReality. 2011.

[20] Markus Lust. Quaternionen - mathematischer Hintergrund und ihre Interpretation als Rotationen. 2001.

[21] Hans Joachim Oberle. Approximationen. pages 125–130, 2014.

[22] Gerlind Plonka-Hoch. Numerik II – Numerische Analysis. pages 78–81.

[23] Ulrich Rüde. Bézier-Kurven, Modellierung von Freiformkurven und –flächen.

[24] Tomas Sauer. Splinekurven und -flächen in CAGD und Anwendung. 2013.

[25] Tomas Sauer. Einführung in die Numerische Mathematik. 2016.

[26] Hamid Fetouaki/Emma Skopin. Bezier-Kurven. http://pool-serv1.mathematik.uni-kassel.de/~fetouaki/seminar/presentation_handout.pdf. (aufgerufen: 19.05.2021), 2009.

[27] Wolfgang Straßer. Einführung in die Graphische Datenverarbeitung. pages 184–192.

List of Figures

List of Tables

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 20.07.2021

Eduard Fast