Implementierung und Vergleich verschiedener Strategien zum Abfliegen einer Flugroute mit einem Multikopter

Christopher Wrobel

8. November 2016

Inhaltsverzeichnis

1 Einleitung

Seit Beginn der modernen Fortbewegung ist das Ziel in der Weiterentwicklung die Verbesserung der Sicherheit, Bequemlichkeit, Effizienz und vor allem Geschwindigkeit. Das Streben nach der Erfüllung dieser Kriterien kann man an der Entwicklung der Verkehrsmittel im Verlauf der Zivilisation sehen. Zuerst zu Fuß gefolgt von Pferden mit oder ohne Kutschen. Nach großen Fortschritten in der Mechanik wurde diese auch zügig in Form von Heißluftballons, Dampflokomotiven, Luftschiffen, Flugzeugen und letztendlich Raketen angewendet. Seit der raschen Entwicklung der Computertechnologie wurde diese auch sehr schnell in vielen Verkehrsmittel eingesetzt, sodass heute ein normales Auto mehr Rechenleistung hat, als die Raketen bei den Apollo-Missionen. Diese Rechenleistung wird meist für den Benutzerkomfort verwendet. Dieser Komfort kann verschieden sein, von Navigationssystemen bis zu Multimediainhalten. Aber am ambitioniertesten ist wohl der Traum des autonomen Fahrens. Der Internetkonzern Google hat z.B. ein vollständig selbst fahrendes Auto konstruiert und getestet[3]. Diese Technologie schreitet sogar schon auf den öffentlichen Markt mit selbst parkenden Autos bis zu Tesla's „Driver Assistance" welche mithilfe von Lernalgorithmen und Schwarmintelligenz ein sehr fortgeschrittener Autopilot ist[6]. Auch auf Schienen werden schon autonome Fahrsysteme eingesetzt, wie z.B. in Nürnberg wo zwei U-Bahn Linien voll automatisch ohne Fahrer fahren.[2] In der Luftfahrt sind Autopiloten schon seit langem unentbehrlich. Sie helfen den Piloten bei Landeanflügen oder um auf Kurs zu bleiben. Aber auch in kleineren Maßstäben gibt es autonome Systeme, wie z.B. in dem Modellbau besonders bei Multikoptern (auch UAV oder Drohnen genannt). Diese Fluggeräte haben unglaubliches Potential in einer neuen Infrastruktur. Die Firma Amazon hat beispielsweise Tests durchgeführt, ob man diese Drohnen als alternative Zustellungsmöglichkeit gebrauchen könnte[5]. Um dem Ziel, eine sichere Luft-Infrastruktur aufzubauen, näher zu kommen beschäftigt sich diese Arbeit mit der Implementierung und dem Vergleich verschiedener Strategien zum Abfliegen einer Flugroute mit einem Multikopter.

2 Interpolation - Theorie

Die Interpolation ist eine mathematische Problemstellung, welche den Zwischenraum zwischen gegeben Punkten erahnt. Diese Problemstellung hat inzwischen viele Lösungsansätze hervorgebracht. Diese Methoden werden am häufigsten bei Messwerten angewandt um aus einer Messreihe mit begrenzter Anzahl von Messpunkten z.B. eine Funktion zu erstellen. In dieser Arbeit werden jedoch die Methoden dazu verwendet eine Flugroute aus einzelnen gegebenen Wegpunkten zu berechnen. Dies

wird zunächst nur auf zwei Dimensionen bearbeitet, da bei komplexen Flugrouten die Komplexität eher bei den Flugrichtungsänderungen auftritt, als bei Höhenänderungen.

2.1 Lineare Interpolation

Die einfachste Art der Interpolation ist die lineare Interpolation. Bei dieser Methode verbindet man die gegebenen Punkte mit Geraden, also mit Funktionen des Grads 1. Berechnet wird dies mit der allgemeinen Geradenform $mx + t$. Mit einem Gleichungssystem mit 2 Unbekannten mit 2 Punkten als Bedingungen.

$$y_i = m \cdot x_i + t$$
$$y_{i+1} = m \cdot x_{i+1} + t$$

Diese Formeln lassen sich jeweils umstellen, dass man m und t errechnen kann.

$$m = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \tag{1}$$

$$t = y_{i+1} - mx_{i+1} \tag{2}$$

Die einzelnen Geraden werden dann einfach aneinander gehängt (Formel 3) um eine komplette Flugroute $g(x)$ zu bekommen.

$$g(x) = \begin{cases} f_1(x) & x_1 \leq x \leq x_2 \\ f_2(x) & x_2 \leq x \leq x_3 \\ f_3(x) & x_3 \leq x \leq x_4 \\ \vdots & \vdots \\ f_{n-1}(x) & x_{n-1} \leq x \leq x_n \end{cases} \tag{3}$$

Der Nachteil von dieser Methode liegt darin, dass die an den Punkten spitze Ecken entstehen können (Siehe Abbildung 1). Dadurch bremst der Multikopter beim Anflug auf die Wegpunkte wieder ab, um den Richtungswechsel zu vollziehen. Man könnte auch den Multikopter über das Ziel „hinausschießen“ lassen um eventuell Zeit zu sparen, aber da diese Methode, der Bezierkurve (siehe 2.5) ähnelt wird diese in dieser Arbeit nicht weiter behandelt.

2.2 Polynominterpolation

Die Polynominterpolation ist eine Interpolationsmethode, welche die Eigenschaft von Polynomen nutzt, dass es für n Punkte immer eine Polynomfunktion $n - 1$-

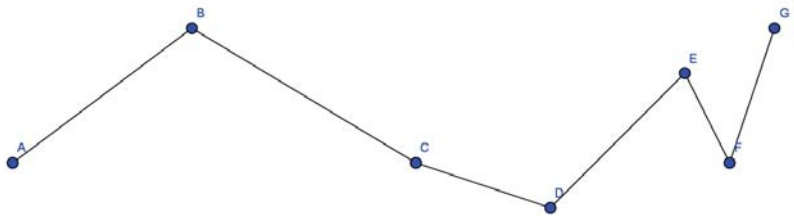

Abbildung 1: Linear interpolierte Wegpunkte

ten Grades gibt, die durch alle Punkte läuft. Diese Funktion wird berechnet, indem man eine leere Polynomfunktion $a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_nx^n$ mit den vorgegebenen Punkten als Bedingungen auflöst. Diese Methode funktioniert bei einer geringen Zahl von Punkten ziemlich gut. Aber da der Grad der Funktion mit zunehmender Punktanzahl immer größer wird, werden die Ausschläge der Kurve zwischen den Punkten immer größer und somit unbrauchbar für effiziente Flugrouten (siehe Abbildung 2).

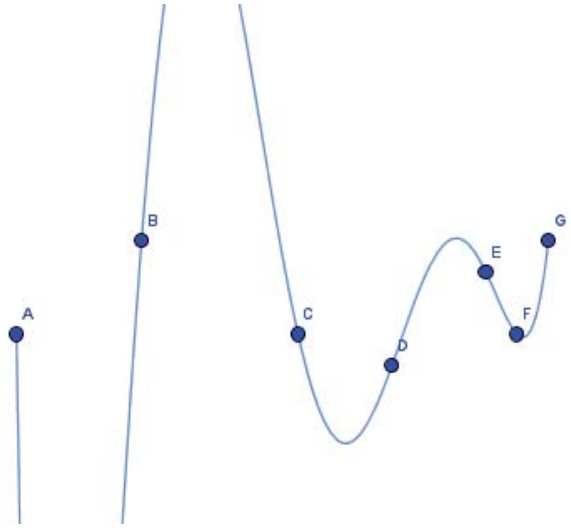

Abbildung 2: Polynominterpolisationsgraph des 6-ten Grades

2.3 Kubische Spline Interpolation

Der Begriff Spline stammt von einem Zeichenwerkzeug, welches vorwiegend zum technischen Zeichnen verwendet wurde. Mit diesem Werkzeug ist es möglich mehrere Punkte auf einer Zeichnung mit einer möglichst kurzen, ästhetischen Kurve zu

verbinden. Aus diesem Konzept wurden dann die mathematischen Splines.
Anstelle eine Funktion für alle Punkte zu finden, rechnet man bei der Kubischen Spline Interpolation (folgend auch „KSI“) kubische Funktionen aus, die ausschließlich für die Intervalle zwischen den Punkten gelten. Somit hat jedes Intervall seine eigene Funktion und hat damit keine unnötigen Schwankungen. Die einzelnen Funktionen werden zum Schluss verbunden (Formel 3) um einen gesamten Graphen zu erzeugen.
Da die Funktionen für jedes Intervall immer den Grad 3 haben, braucht es insgesamt 4 Bedingungen um die Funktion eindeutig bestimmen zu können. Die offensichtlichen Bedingungen sind die beiden Punkte die interpoliert werden sollen. Die weiteren zwei Bedingungen sind, dass die erste Ableitung vom Anfangspunkt $f_i'(i)$ ($i = 0, 1, 2, ..., n$ und n ist die Anzahl der Punkte) als auch die erste Ableitung des Endpunktes $f_i'(i+1)$ gleich den ersten Ableitungen an den selben Punkten von der vorigen und der folgenden Funktion sind (siehe Formel 5). Dies gewährleistet, dass keine Knicke in dem Graphen von $g(x)$ vorkommen da die Steigung an den Wegpunkten ineinander übergehen.

$$\begin{aligned} f_i(x_i) &= y_i \\ f_i(x_{i+1}) &= y_{i+1} \\ f_i'(x_i) &= f_{i-1}'(x_i) \\ f_i'(x_{i+1}) &= f_{i+1}'(x_{i+1}) \end{aligned} \tag{4}$$

Für die Funktion $f_i(x)$ und deren Ableitungen kann man die allgemeine kubische Polynomfunktion einsetzen.

$$\begin{aligned} ax_i^3 + bx_i^2 + cx_i + d &= y_i \\ ax_{i+1}^3 + bx_{i+1}^2 + cx_{i+1} + d &= y_{i+1} \\ 3ax_i^2 + 2bx_i + c &= f_{i-1}'(x_{i+1}) \\ 3ax_{i+1}^2 + 2bx_{i+1} + c &= f_{i+1}'(x_{i+1}) \end{aligned} \tag{5}$$

Ein Problem, dass sich dabei stellt ist, dass das erste Intervall als auch das letzte Intervall keine vorherige bzw. nachfolgende Funktion haben. Um diesen Intervallen Bedingungen zu geben gibt es 3 Möglichkeiten: eingespannte Splines, natürliche Splines und periodische Splines. Der eingespannte Spline hat als 2 Randbedingungen vorgegebene Werte für die Ableitung $g'(a)$ und $g'(b)$. Diese Methode setzt aber voraus für jede Situation immer unterschiedliche Werte einzusetzen. Die natürlichen Splines haben als Randbedingungen $g''(a) = 0$ und $g''(b) = 0$. Diese Splines beginnen und enden mit der Krümmung 0 d.h. gerade. Die letzte Version sind die periodi-

schen Splines, welche die Randbedingungen haben $g^{(k)}(a) = g^{(k)}(b)$ für $k = 0, 1, 2$. Sprich, diese Splines haben den selben y-Wert am ersten und am letzten Punkt, haben die selbe Steigung an eben genannten Punkten, und die selbe Krümmung an diesen Punkten. Dadurch lässt sich dieser Spline beliebig oft hintereinander hängen und ergibt eine unendliche Kurve.
In dieser Arbeit fallen 2 Methoden zum Gewinnen der Randbedingungen weg, da sie für das Finden einer möglichst effektiven Router unpassend sind. Das wäre zum einen der eingespannte Spline, da das Programm letztendlich fähig sein soll ohne zusätzliche manuelle Eingaben eine passende Route zu finden. Und zum anderen die periodischen Splines, weil die Route nicht beliebig oft hintereinander abgeflogen werden soll. Übrig bleiben nur noch die natürlichen Splines, welche in diesem Fall geeignet sind, da sie keine Form von manuellem Input benötigen. [1] [7] [9]
Um diese Spline eindeutig auszurechnen gibt es jedoch ebenfalls Probleme. Eins davon wäre, dass man mit einem linearen Gleichungssystem unmöglich alle Intervalle auf einmal lösen kann. Dies wäre hier in diesem Fall aber nötig, da um das Intervall i auszurechnen man sowohl die schon aufgelöste Funktion des Intervalls $i + 1$ als auch des Intervalls $i - 1$ braucht. Um dieses Problem zu umgehen wurde die 4. Bedingung in Formel 4 entfernt. Somit hat das aktuelle Intervall immer den Zwang sich mit der ersten Ableitung an die vorhergehende Funktion anzupassen, damit ein glatter Übergang entsteht. Das Problem das damit einhergeht ist der Verlust der 4. Bedingung und diese muss mit einer anderen Bedingung ersetzt werden, damit das Gleichungssystem eindeutig lösbar bleibt. Hierfür kann man die zweite Ableitung verwenden. Man fügt die Bedingung hinzu, dass nicht nur die erste, sondern auch die zweite Ableitung mit der vorhergehenden Funktion übereinstimmen muss. Mathematisch formuliert wäre es dann $f_i''(x_i) = f_{i-1}''(x_i)$. Somit kann man, solange man die Funktion für die vorhergehende Funktion $i - 1$ hat, immer die eigene Funktion für das Intervall i ausrechnen. Dies bewirkt aber ebenfalls, dass die Randbedingung am Ende des Splines an den Anfang verschoben wurde. Somit hat man $4n - 2$ Bedingungen, die automatisch errechnet werden können und 2 Randbedingungen, die manuell vorgegeben werden müssen. Insgesamt sind es somit $4n$ Bedingungen, weshalb das Gleichungssystem eindeutig lösbar ist.

2.4 lineare Interpolation mit Spline-Elementen

Diese Methode der Interpolation ist ein Hybrid aus der linearen Interpolation (2.1) und der Kubischen Spline Interpolation (2.3). Die einzelnen Intervalle zwischen den Punkten werden wie in der Kubischen Spline Interpolation mit einer kubischen Funktion interpoliert. Die Bedingungen für diese sind nach wie vor die 2 zu interpolie-

renden Punkte und die erste Ableitung des vorigen Intervalls am Anfangspunkt des aktuellen Intervalls. Die vierte Bedingung ist aber, dass die erste Ableitung am Punkt $i+1$ die selbe ist wie die Ableitung an demselben Punkt von dem nächsten Intervall von der linearen Interpolation.

$$\begin{aligned} f_i(x_i) &= y_i \\ f_i(x_{i+1}) &= y_{i+1} \\ f_i'(x_i) &= f_{i-1}'(x_i) \\ f_i'(x_{i+1}) &= f_{lin\,i+1}'(x_{i+1}) \end{aligned}$$

Für die Funktion $f_i(x)$ und deren Ableitungen kann man die allgemeine kubische Polynomfunktion einsetzen.

$$\begin{aligned} ax_i^3 + bx_i^2 + cx_i + d &= y_i \\ ax_{i+1}^3 + bx_{i+1}^2 + cx_{i+1} + d &= y_{i+1} \\ 3ax_i^2 + 2bx_i + c &= f_{i-1}'(x_i) \\ 3ax_{i+1}^2 + 2bx_{i+1} + c &= f_{lin\,i+1}'(x_{i+1}) \end{aligned} \tag{6}$$

Mit dieser Methode erhält man eine Kurve welche möglichst linear ist aber vor Knicken ausholt um einen sanfteren Übergang zu ermöglichen (siehe Abbildung 3), denn die Gleichheit der ersten Ableitung garantiert den Übergang ohne Knick. Zudem wird die Funktion schon in die Richtung des nächsten Punktes geleitet.

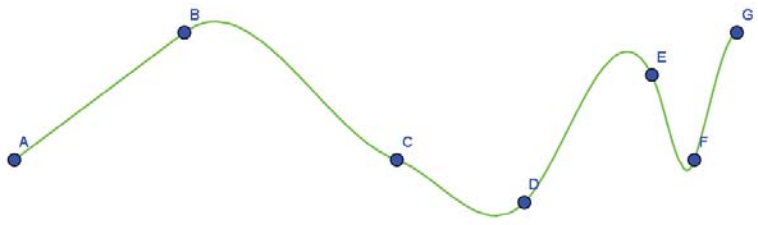

Abbildung 3: lineare Interpolation mit Spline-Elementen

2.5 Bezierkurven

Bezierkurven sind Kurven, welche zwei Punkte mithilfe von zusätzlichen Paramentern interpolieren. Dieser Parameter ist ebenfalls als Punkt in einem Koordinatensystem darzustellen, und dieser gibt zum einen an, in welche Richtung die Kurve den Startpunkt verlassen soll, oder im Endpunkt einfallen soll. Zum anderen gibt dieser Punkt aber auch an, wie weit sich die Kurve von den Punkten entfernen darf. In dieser Arbeit wird aber nur eine bestimmte Version der Bezierkurve hergenommen. Dabei wird die Kurve zwischen 2 Punkten mit nur einem Parameter berechnet. Der zusätzliche Parameter (Punkt B) wird in diesem Fall der Geschwindigkeitsvektor des Multikopters am Anfangspunkt A sein. Der Kontrollpunkt B wird zugleich für den Endpunkt C verwendet. Zur Berechnung von dem Punkt bei abgeflogenen 20% des Intervalls werden zuerst jeweils 20% des Vektors $\overrightarrow{AB}$ und des Vektors $\overrightarrow{BC}$ berechnet. Zwischen der Spitze des neuen Vektors $\overrightarrow{AB}_{20\%}$ und der Basis des neuen Vektors $\overrightarrow{BC}_{20\%}$ wird ein Vektor $\overrightarrow{\overrightarrow{AB}_{20\%}\overrightarrow{BC}_{20\%}}$ berechnet, von dem ebenfalls 20% der Länge erechnet werden. Wenn man diesen Vektor als Ortsvektor relativ zu A repräsentiert hat man den Punkt bei 20% des Intervalls $[A;C]$[4] (Vergleiche Abbildung 4).

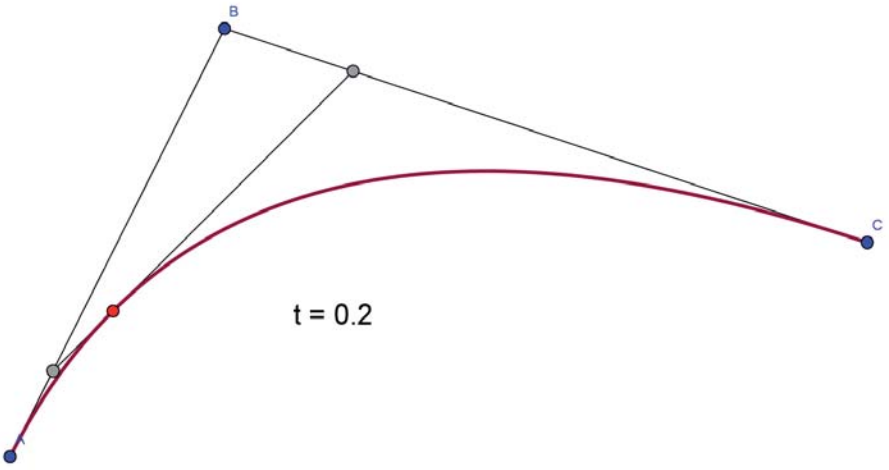

Abbildung 4: Bezierkurvenpunkt bei 20% abgeflogener Flugroute (Roter Punkt)

3 Implementierung der Interpolationsmethoden

Die Implementierung für Demonstrationszwecke wird hier in der Programmiersprache C# stattfinden. Die programmierten Methoden geben Rohwerte aus, welche dann in einen Funktionsplotter eingegeben werden können (in diesem Fall Geogebra), um die Ergebnisse zu überprüfen. Die Ergebnisse ausgeben kann man zum einen über

die Funktion, zum anderen über Werte, welche in regelmäßigen Abständen errechnet werden. In den folgenden Implementierungen kommt das `struct Punkt` zum Einsatz und wird deshalb vorab hier besprochen. Dieses `struct` beinhaltet die zwei Variablen `x` und `y` und einen Konstruktor, welcher diesen Variablen einen Wert gibt.

Code 1: Das `struct Punkt`

```
1 struct Punkt
2 {
3   public double x, y;
4   public Punkt(double m, double n)
5   {
6           x = m;
7           y = n;
8   }
9 }
```

Zudem gibt es für Beispielzwecke auch ein Array `Punkte[]`.

3.1 Lineare Interpolation

Wie in Formel 1 und 2 gezeigt, ist es ziemlich einfach die Funktionen für die Intervalle zwischen den Punkten zu finden.

Code 2: Methode für die lineare Interpolation

```
1 static public void linear(Punkt[] Punkte)
2 {
3     for(int i = 0; i < Punkte.GetLenght(0) - 1; i++)
4     {
5         double m, t;
6         m = (Punkte[i + 1].y - Punkte[i].y) / (Punkte[i +
              1].x - Punkte[i].x);
7         t = Punkte[i + 1].y - m * Punkte[i + 1].x;
8   }
9 }
```

In dieser Methode wird eine `for`-Schleife gestartet, die so oft Funktionen für die Intervalle berechnet wie es Intervalle gibt. In Zeile `6` ist die Formel 1 und in Zeile `7` die Formel 2 enthalten.
Diese Implementation wird jedoch nur von der linearen Interpolation mit Spline-Elementen aus Kapitel 2.4 verwendet, da der Quadrocopter mit seiner Firmware na-

mens „Arducopter“nativ die lineare Interpolation zwischen Wegpunkten unterstützt. Für die Versuche wird die Methode von Arducopter verwendet. [8]

3.2 Kubische Spline Interpolation

Die in 2.3 beschriebene Theorie wurde ebenso in Code umgesetzt. Um das Gleichungssystem auszurechnen wurde die Library „MathNet Numerics“ verwendet. Die Werte die hierbei herauskommen sind jedoch entweder fehlerhaft oder unpassend für die Berechnung von Flugrouten. Wie man auf der Abbildung 5 sieht, könnte man meinen, dass der Anfang der Spline gute Werte berechnet. Dies liegt jedoch daran, dass die manuellen Startwerte bei dieser Berechnung vorgegeben waren. Diese wurden zuvor nämlich aus einer Polynominterpolation der ersten 3 Punkte berechnet. Gegen Ende des Graphen sieht man schon deutliche Schwankungen, welche für Flugrouten komplett unpassend sind. Es wurde auch versucht die Werte für die zweite Ableitung künstlich anzupassen um unnötige Schwankungen zu verhindern. Wenn die zweite Ableitung einen gewissen Schwellwert überschreitet, wird diese durch Multiplikation mit einem Faktor, der zwischen 1 und 0 liegt, reduziert. Jedoch hat diese Maßnahme nur in begrenztem Maße geholfen und war somit auch hinfällig.
Um trotzdem einen Vergleichswert bei den Versuchen zu haben, wurde für die Spli-

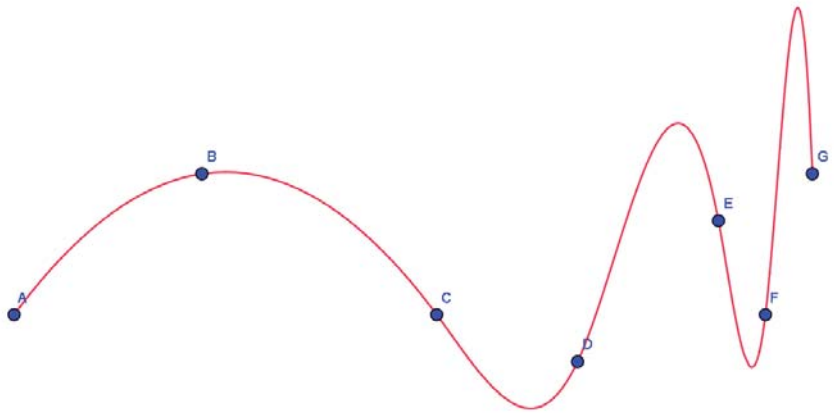

Abbildung 5: Ausgabe der Implementierten Spline Methode

neberechnung die Methode in der Arducopter Firmware verwendet.

3.3 Bezierkurven

Damit die Bezierkurve überhaupt berechnet werden kann, braucht man neben den zwei Punkten für jedes Intervall auch die Geschwindigkeit von dem Flugobjekt am

Anfangspunkt des Intervalls. An die Funktion `bezier` wird also das Array `Punkte` weitergegeben aber auch die Geschwindigkeit in Form eines Vektors. Anschließend wird der Kontrollpunkt für die Bezierkurve berechnet, indem der Ortsvektor des ersten Punktes mit dem Geschwindigkeitsvektor addiert wird. Anschließend muss man nur noch, wie in Kapitel 2.5 beschrieben, den Rechenschritten Folgen um an n% des Intervalls die erwünschte Position zu berechnen.
In den folgenden Versuchen wird diese Methode aber nicht verglichen und ausprobiert, da bei der Implementation einige Probleme aufgetreten sind. Zum einen ist die die Kommunikation mit dem Programm, welches sich mit dem Quadrocopter verbindet, wegen mangelnder API-Funktionalität schwer umsetzbar. Und zum anderen ist die Latenz zwischen dem Senden der Daten vom Quadrocopter und dem Empfangen dieser durch die Bodenstation zu groß, um in Echtzeit Berechnungen durchzuführen.

3.4 Vorhersagen zu den verschiedenen Methoden der Interpolation in der Anwendung

Diese Methoden sollen in dieser Arbeit hinsichtlich der Flugdauer, Sicherheit und Effizienz verglichen werden. Die Geschwindigkeit bezieht sich nur auf die horizontale durchschnittliche Geschwindigkeit und wird hier in $\frac{m}{s}$ angegeben. Die Sicherheit ist die subjektive Einschätzung des Flugverhaltens in den verschiedenen Methoden. Die Effizienz wird angegeben in benötigten mAh pro Streckenlänge der linearen Interpolation. Hier wird die Streckenlänge der linearen Interpolation hergenommen, da diese immer die kürzeste Strecke ist. Bei einer Strecke, welche z.B. 5 mal länger wäre, könnte trotz unnötig langer Strecke eine gute Effizienz berechnet werden. Um dies zu verhindern wird die Effizienz als abstrakte Zahl dargestellt, welche sich aus $\frac{mAh}{Strecke_{linear}\ in\ [m]}$ errechnen lässt.

Die lineare Interpolation hat trotz ihrer Einfachheit gute Chancen die schnellste Methode zu sein, da diese Interpolationsmethode immer die kürzeste Strecke berechnet. Damit kann es, verglichen mit anderen Methoden, mit einer niedrigeren Geschwindigkeit die selbe Zeit in Anspruch nehmen. Bei der Sicherheit sieht es schon anders aus, da bei den spitzen Kurven der Multikopter ein riskantes Manöver durchführen muss, welches bei anderen Umwelteinflüssen wie Wind, gefährlich werden kann. Bei der Effizienz hat diese Methode wiedermal einen Vorteil, da sie die kürzeste Strecke abfliegt. Aber ungewiss bleibt, ob die aufzubringende Energie in den scharfen Kurven nicht zu groß ist.

Die Kubische Spline Interpolation hat wegen ihrer ununterbrochenen Form einen

Methode	Zeit	Sicherheit	Effizienz
Lineare Interpolation	++	- -	+
Kubische Spline Interpolation	+	+	++
Bezierkurven	+	+	++
lin. Spline	++	++	++

Tabelle 1: Vorhersagen der Leistung der Methoden bewertet von - - bis ++

Vorteil in der Geschwindigkeit, da sie die Kurven so abfliegt, dass die Richtungsänderung auf eine größere Zeit verteilt ist. Außerdem sind bei dieser Methode die engen Kurven weniger riskant, da sie abgerundet sind. Die Effizienz ist aber im Nachteil, da die Strecke verlängert ist und der Multikopter sich länger in der Luft halten muss.

Die Interpolation mithilfe von Bezierkurven verhindert scharfe Ecken in der Flugroute, indem sie die aktuelle Geschwindigkeit des Multikopters miteinbezieht. Dadurch ist diese Methode viel flexibler und somit auch weniger anfällig auf äußere Einflüsse, da diese mit in die Berechnung mit einfließen. Dies kann sogar so weit gehen, dass diese dem Multikopter helfen schneller auf einer Strecke voran zu kommen. Trotzdem ist bei dieser Methode die Strecke ebenfalls länger als das Minimum. Aber die Flugzeit sollte durch die Vermeidung von scharfen Kurven ebenfalls verbessert werden, trotz der zusätzlichen Strecke.

Die lineare Interpolation mit Spline-Elementen (folgend auch lin. Spline) vereint das beste aus der Welt der linearen Interpolation und der Kubischen Spline Interpolation. Diese Methode hat eine möglichst kurze Strecke, um eine möglichst kurze Flugdauer zu ermöglichen, aber zugleich auch abgerundete Ecken, sodass die Sicherheit und Effizienz nicht beeinträchtigt werden.

Zusammenfassen lässt sich das am besten in einer Tabelle (siehe Tabelle 1).

4 Vergleich und Fazit

Der Versuchsaufbau besteht aus einem Quadrocopter, welcher mit einer Funkverbindung in Echtzeit Daten zur Fluglage, Position, Geschwindigkeit, etc. an einen Computer, der als Bodenstation dient, sendet. Vor jedem Flug werden von diesem Computer die errechneten Routen an den Quadrocopter in Form von Wegpunkten gesendet. Der Quadrocopter erhält somit nur Wegpunkte, welche er linear abfliegt. Ursprünglich war geplant den Code zum Berechnen der Flugrouten auf dem Quadrocopter-Boardcomputer berechnen zu lassen, jedoch gibt es zwei Probleme mit diesem Ansatz. Zum einen, dass die Firmware, die auf dem Boardcomputer ist, fast den ganzen Speicherplatz des EEPROM einnimmt. Und zum anderen, dass der Prozessor des Boardcomputers zu langsam ist, die Berechnungen auszuführen und

währenddessen noch das Fluggerät stabil in der Luft zu halten.
Für die Tests gibt es vier verschiedene Arten von Flugrouten, welche jeweils die gleiche linear Interpolierte Länge und die selbe Anzahl an Wegpunkten haben. Die Hauptunterschiede zwischen den Flugrouten sind die Winkel. In Typ 1 der Flugrouten sind die Winkel stets größer als 120°. Typ 2 hat Winkel, welche größer als 90° aber kleiner als 120° sind. Typ 3 hat Winkel zwischen 45° und 90° und Typ 4 hat Winkel zwischen 0° und 45°. In Abbildung 6 sind die Flugrouten, die für die Versuche verwendet worden sind, abgebildet. Es wurde versucht die Winkel so unterschiedlich wie möglich zu gestalten, damit eine hohe Diversität entsteht um möglichst viele Fälle abzudecken.
Diese Flugrouten haben allesamt 8 Wegpunkte und sind insgesamt 110 Meter lang.

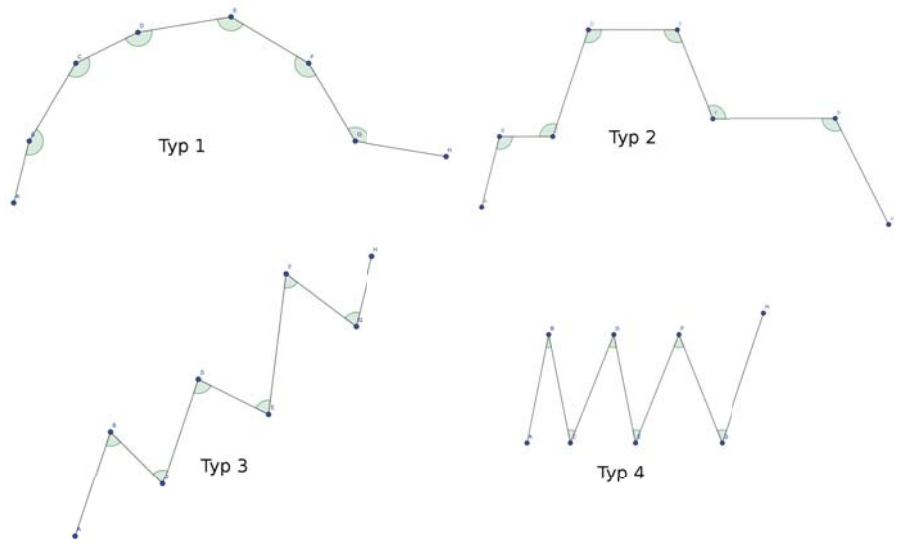

Abbildung 6: Verschiedene Typen von Flugrouten

4.1 Vergleich

Die folgenden Messungen wurden draußen gemacht und sind deshalb auch von externen Faktoren wie z.B. Wind abhängig. Diese Messwerte können dementsprechend hohe Messungenauigkeiten aufweisen. Aus den Messdaten aus den Tabellen 2 bis 5 kann man verschiedene Schlüsse ziehen. Zum einen, dass die lineare Interpolation mit Spline-Elementen aus Kapitel 2.4 von der Zeit und auch von der Effizienz am schlechtesten abschneidet. Dies ist jedoch nicht unbedingt wegen der Methode an sich, sondern eher, wie sie für diese Versuche implementiert wurde. Für jede Flugroute wurden bei dieser Strategie nämlich 70 Wegpunkte erstellt um die Kurve in den Intervallen zu definieren. Jedoch ist die Firmware auf dem Quadrocopter so implementiert, dass der Quadrocopter an jedem Wegpunkt langsamer wird. Somit wurde

der Quadrocopter für jedes Intervall 10 mal mehr abgebremst als bei den anderen Strategien. Bei einer direkten Implementation würde das Ergebnis für Flugdauer und somit auch Effizienz sicher besser abschneiden. Beim vergleichen von der linearen Interpolation und der KSI fällt auf, dass außer bei Typ 3 (Tabelle 4) die Flugdauer für die Flugrouten nahe identisch sind. Wegen möglichen Messungenauigkeiten kann man hier keine Überlegenheit feststellen und der Messwert in Tabelle 4 ist höchstwahrscheinlich ein Messfehler, welcher durch externe Faktoren hervorgerufen wurde. Die Effizienzunterschiede zwischen der linearen Interpolation und der KSI sind für Typ 1 und 2 zugunsten der linearen Interpolation, aber für Typ 3 und 4 zugunsten der KSI. Eine Erklärung dafür wäre, dass die Energie, die benötigt wird um an jedem Wegpunkt die Richtung zu ändern, für immer kleiner werdende Winkel größer wird, als die Energie, die benötigt wird um die zusätzliche Strecke der KSI zu fliegen. Jedoch muss gesagt werden, dass Angaben über den Stromverbrauch meist ziemlich ungenau sind, da viele zusätzliche Faktoren eine Rolle spielen. Die Akkus werden unterschiedlich schnell entladen, abhängig von ihrer Fülle, Temperatur, etc. Deshalb könnte eine andere Erklärung sein, dass dies zufällige Messungenauigkeiten sind. Zuletzt wird noch die subjektive Einschätzung zur Sicherheit von den verschiedenen Strategien verglichen. Dazu sei gesagt, dass durch die gute Implementierung der Firmware des Quadrocopters es keine gefährlichen Situationen gab. Deshalb hängt die Sicherheit der Strategien an der Entfernung und Abschweifungen die das Fluggerät während des Fluges macht. Am besten wird das veranschaulicht an den berechneten Flugrouten in den Abbildungen 7 bis 10. Bei Typ 1 berechneten alle Methoden eine Flugroute ohne große Umwege, da durch die großen Winkel bei den Wegpunkten diese auf ähnliche Ergebnisse gekommen sind. Bei den anderen drei Typen von Flugrouten bleibt der Quadrocopter nah genug an den ursprünglichen Wegpunkten, dass man diese noch als sicher einstufen kann.

4.2 Fazit

Als Fazit für diese Erkenntnisse kann man sagen, dass die lineare Interpolation in den meisten Fällen am besten geeignet ist um Flugrouten zu berechnen. Aber dies gilt nur für Multikopter, da diese sehr schnell ihre Richtung ändern können. Die Methoden, die in dieser Arbeit schlechter abschnitten, können bei Fluggeräten anderer Art, welche keine engen Kurven fliegen können, vielleicht besser geeignet sein.

Methode	**Flugdauer in Sekunden**	**Sicherheit**	**Effizienz in $\frac{mAh}{m}$**
lineare Interpolation	21	++	3,9
KSI	20	++	4,5
lin. Spline	28	++	4,2

Tabelle 2: Werte für die verschiedenen Strategien für die Flugroute des Typen 1

Methode	**Flugdauer in Sekunden**	**Sicherheit**	**Effizienz in $\frac{mAh}{m}$**
lineare Interpolation	21	++	3,3
KSI	24	+	5,4
lin. Spline	45	+	2,4

Tabelle 3: Werte für die verschiedenen Strategien für die Flugroute des Typen 2

Methode	**Flugdauer in Sekunden**	**Sicherheit**	**Effizienz in $\frac{mAh}{m}$**
lineare Interpolation	40	++	6,6
KSI	48	+	4,1
lin. Spline	63	+	5,7

Tabelle 4: Werte für die verschiedenen Strategien für die Flugroute des Typen 3

Methode	**Flugdauer in Sekunden**	**Sicherheit**	**Effizienz in $\frac{mAh}{m}$**
lineare Interpolation	59	++	4,2
KSI	60	+	3,8
lin. Spline	65	+	4,5

Tabelle 5: Werte für die verschiedenen Strategien für die Flugroute des Typen 4

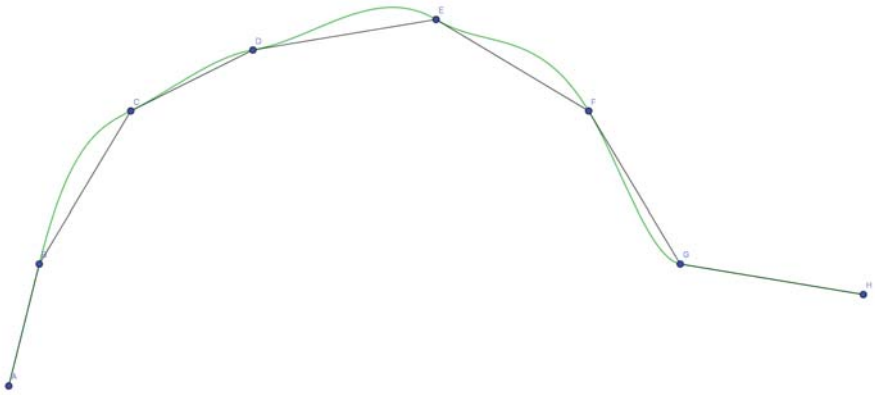

Abbildung 7: Flugkurven der lineare Interpolation (schwarz) und lineare Interpolation mit Spline-Elementen (grün) auf Typ 1

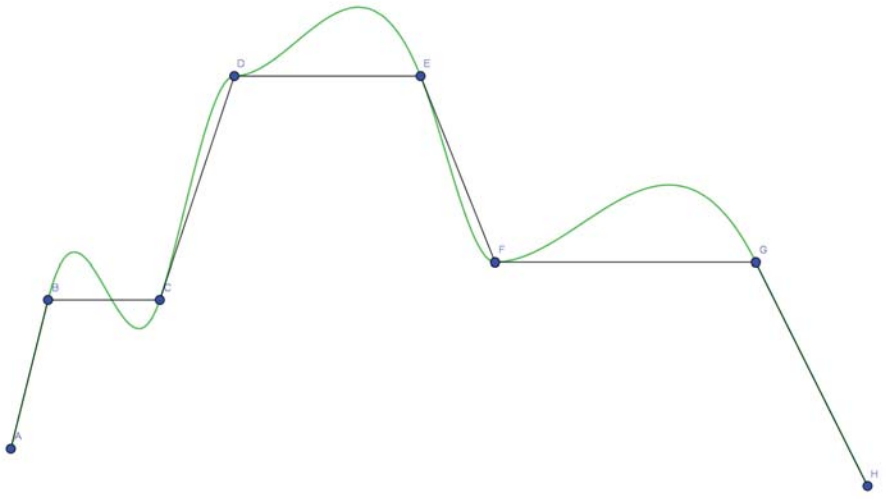

Abbildung 8: Flugkurven der lineare Interpolation (schwarz) und lineare Interpolation mit Spline-Elementen (grün) auf Typ 2

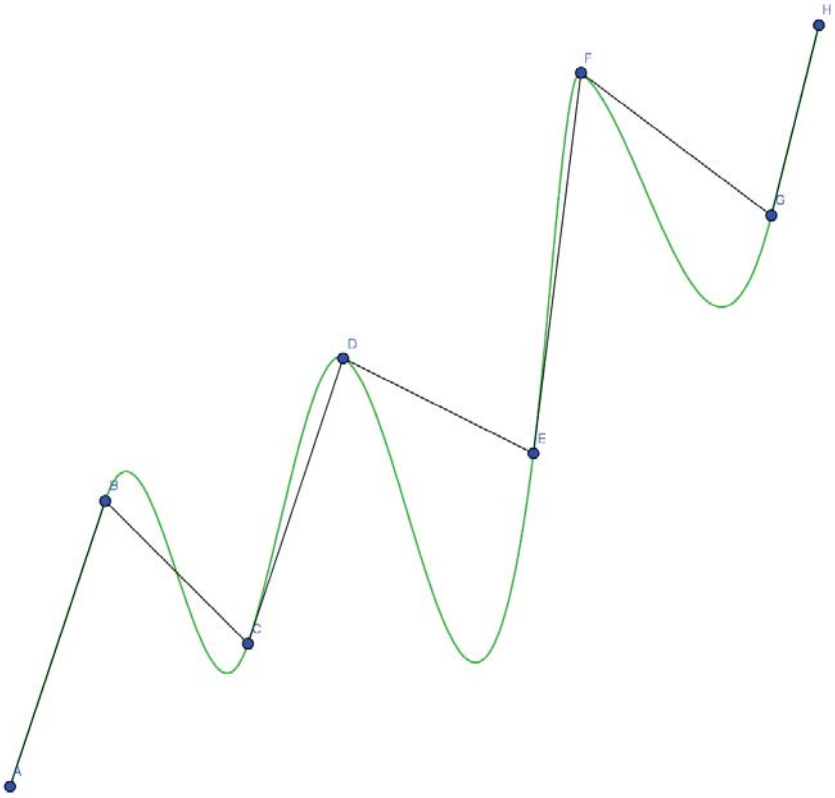

Abbildung 9: Flugkurven der lineare Interpolation (schwarz) und lineare Interpolation mit Spline-Elementen (grün) auf Typ 3

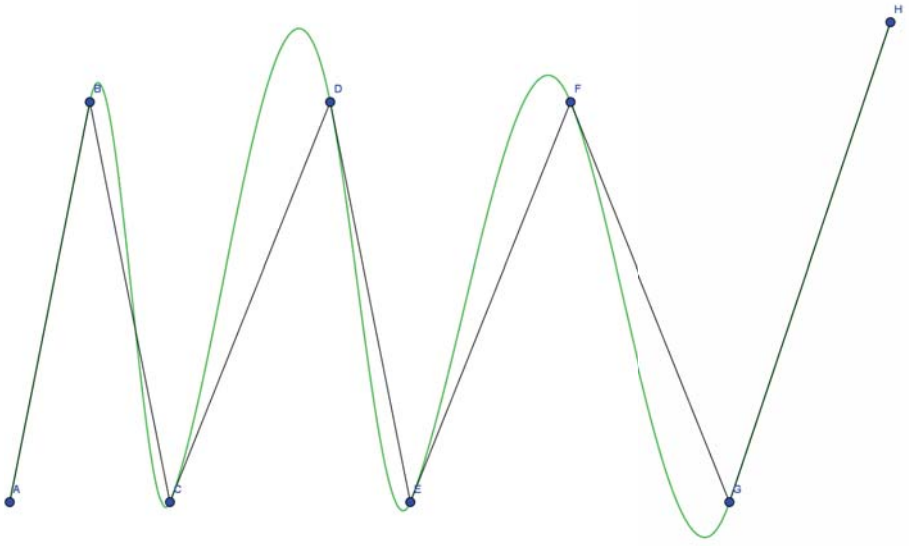

Abbildung 10: Flugkurven der lineare Interpolation (schwarz) und lineare Interpolation mit Spline-Elementen (grün) auf Typ 4

5 Anhang

Literatur

[1] Carl de Boor. *Splinefunktionen*. Lectures in mathematics. Birkhäuser, Basel [u.a.], 1990. ISBN 0-8176-2514-3.

[2] INFRA Dialog Deutschland GmbH. Nürnbergs u-bahn fährt auf zwei linien ganz ohne fahrer. *INFRA Dialog Deutschland GmbH*. URL http://www.damit-deutschland-vorne-bleibt.de/Blickpunkt/Infrastruktur-aktuell/04493/Artikel/Nuernbergs-U-Bahn-faehrt-auf-zwei-Linien-ganz-ohne-Fahrer/04106.

[3] John Markoff. Google cars drive themselves, in traffic, Stand: 07.11.2016. URL http://www.spiegel.de/wirtschaft/unternehmen/amazon-testet-paket-lieferung-per-drohne-in-grossbritannien-a-1104751.html.

[4] David Salomon. *Curves and Surfaces for Computer Graphics*. Springer Science+Business Media Inc, New York, NY, 2006. ISBN 0-387-24196-5. doi: 10.1007/0-387-28452-4. URL http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10229038.

[5] ssu/dpa. Amazon lotet paketlieferung per drohne aus, Stand: 07.11.2016. URL http://www.spiegel.de/wirtschaft/unternehmen/amazon-testet-paket-lieferung-per-drohne-in-grossbritannien-a-1104751.html.

[6] Tesla. Model s - owners guide, Stand: 07.11.2016. URL https://www.tesla.com/sites/default/files/model_s_owners_manual_touchscreen_7.1_das_ap_north_america_r20160112_en_us.pdf.

[7] Thomas Richter and Thomas Wick. *Einführung in die Numerische Mathematik*. 2012. URL http://www.numerik.uni-hd.de/~lehre/SS12/numerik0/gesamt.pdf.

[8] Andrew Tridgell, Randy Mackay, and Michael du Breuil. Arducopter, Stand: 07.11.2016. URL http://ardupilot.org/dev/index.html.

[9] Dr. Daniel Weiß. Numerische mathematik - vortrag, Stand: 07.11.2016. URL https://www.youtube.com/watch?v=c26sr-tuB6A.

5.1 Quellcode

```
1  namespace ConsoleApplication1
2  {
3      struct Punkt
4      {
5          public double x, y;
6          public Punkt(double m, double n, Vektor v)
7          {
8              x = m;
9              y = n;
10             double a = v.x + x;
11             double b = v.y + y;
12             Punkt bezierPunkt = new Punkt(a, b);
13             Vektor bezierVektor = v;
14         }
15         public Punkt(double m, double n)
16         {
17             x = m;
18             y = n;
19         }
20     }
21
22     struct Vektor
23     {
24         public double x, y;
25         public Vektor(double m, double n)
26         {
27             x = m;
28             y = n;
29         }
30         public Vektor VektorMul(double i)
31         {
32             x = x * i;
33             y = y * i;
34             return this;
35         }
36     }
37
38     class Program
39     {
```

```
40          static void Main(string[] args)
41          {
42              Punkt[] Loesung = new Punkt[101];
43              Punkt[] Punkte = new Punkt[8];
44              //Graphics
45              //Punkte[0] = new Punkt(1, 2);
46              //Punkte[1] = new Punkt(5, 5);
47              //Punkte[2] = new Punkt(10, 2);
48              //Punkte[3] = new Punkt(13, 1);
49              //Punkte[4] = new Punkt(16, 4);
50              //Punkte[5] = new Punkt(17, 2);
51              //Punkte[6] = new Punkt(18, 5);
52
53          //Typ1
54              Punkte[0] = new Punkt(0, 0);
55              Punkte[1] = new Punkt(1, 4);
56              Punkte[2] = new Punkt(4, 9);
57              Punkte[3] = new Punkt(8, 11);
58              Punkte[4] = new Punkt(14, 12);
59              Punkte[5] = new Punkt(19, 9);
60              Punkte[6] = new Punkt(22, 4);
61              Punkte[7] = new Punkt(28, 3);
62
63          //Typ2
64              //Punkte[0] = new Punkt(1, 1);
65              //Punkte[1] = new Punkt(2, 5);
66              //Punkte[2] = new Punkt(5, 5);
67              //Punkte[3] = new Punkt(7, 11);
68              //Punkte[4] = new Punkt(12, 11);
69              //Punkte[5] = new Punkt(14, 6);
70              //Punkte[6] = new Punkt(21, 6);
71              //Punkte[7] = new Punkt(24, 0);
72
73          //Typ3
74              //Punkte[0] = new Punkt(1, 1);
75              //Punkte[1] = new Punkt(3, 7);
76              //Punkte[2] = new Punkt(6, 4);
77              //Punkte[3] = new Punkt(8, 10);
78              //Punkte[4] = new Punkt(12, 8);
79              //Punkte[5] = new Punkt(13, 16);
80              //Punkte[6] = new Punkt(17, 13);
```

```
81              //Punkte[7] = new Punkt(18, 17);
82
83          //Typ4
84              //Punkte[0] = new Punkt(1, 1);
85              //Punkte[1] = new Punkt(2, 6);
86              //Punkte[2] = new Punkt(3, 1);
87              //Punkte[3] = new Punkt(5, 6);
88              //Punkte[4] = new Punkt(6, 1);
89              //Punkte[5] = new Punkt(8, 6);
90              //Punkte[6] = new Punkt(10, 1);
91              //Punkte[7] = new Punkt(12, 7);
92
93              Vektor velocity = new Vektor(.1, 0.1);
94              //double percent = 0;
95              //string desX;
96              //string desY;
97
98              //double tempx = Punkte[0].x;
99              //double tempy = Punkte[0].y;
100             //for (int i = 0; i < 10; i++)
101             //{
102             //    Punkte[i].x = Punkte[i].x - tempx;
103             //    Punkte[i].y = Punkte[i].y - tempy;
104             //}
105
106             //for (int i = 0; i < 11; i++)
107             //{
108             //    Punkt Destination = bezier(Punkte, velocity
                    , percent);
109             //    desX = Destination.x.ToString("0.00",
                    System.Globalization.CultureInfo.
                    InvariantCulture);
110             //    desY = Destination.y.ToString("0.00",
                    System.Globalization.CultureInfo.
                    InvariantCulture);
111             //    Console.Write($"({desX},{desY}), ");
112             //    percent += .1;
113             //}
114
115             Loesung = Spline(Punkte, 10.0);
116
```

```
117             //for(int i = 0; i < 101; i++)
118             //{
119             //    Console.Write(Loesung[i].x + "," + Loesung[
                    i].y + "\n");
120             //}
121
122             while (true) ;
123
124         }
125
126     static public Punkt[] Spline(Punkt[] Punkte, double
            density)
127         {
128             double f_a = 0;
129             double f_aa = 0;
130             double Faktor = 1.5;
131             double Schwellwert = .005;
132             Punkt[] Loesung = new Punkt[101];
133             double x2, y2;
134
135             double dx1 = Math.Sqrt((Punkte[1].x - Punkte[0].x
                    ) * (Punkte[1].x - Punkte[0].x) + (Punkte[1].y
                     - Punkte[0].y) + (Punkte[1].y - Punkte[0].y))
                    ;
136
137             y2 = 3*(Punkte[1].y - Punkte[0].y) / (dx1 * dx1);
138             x2 = 3 * (Punkte[1].x - Punkte[0].x) / (dx1 * dx
                    1);
139             f_a = x2 / y2;
140
141             for (int i = 0; i < Punkte.Length - 1; i++)
142             {
143
144             var A = Matrix<double>.Build.DenseOfArray(new
                    double[,] {
145                 {Math.Pow(Punkte[i].x, 3), Math.Pow(Punkte[i
                        ].x, 2), Punkte[i].x, 1},
146                 {Math.Pow(Punkte[i+1].x, 3), Math.Pow(Punkte[
                        i+1].x, 2), Punkte[i+1].x, 1},
147                 {Math.Pow(Punkte[i].x, 2) * 3, Punkte[i].x *
                        2, 1, 0},
```

```
148                 {6 * Punkte[i].x, 2, 0, 0}
149                 });
150                 var l = Vector<double>.Build.Dense(new double
                        [] { Punkte[i].y, Punkte[i + 1].y, f_a, f_
                        aa });
151                 var s = A.Solve(l);
152                 for (int j = 0; j < density; j++)
153                 {
154                     string desX = "";
155                     string desY = "";
156                     double x = 0;
157                     double y = 0;
158                     x = ((Punkte[i + 1].x - Punkte[i].x) * j
                            / density) + Punkte[i].x;
159                     y = (s[0] * Math.Pow(x, 3)) + (s[1] *
                            Math.Pow(x, 2)) + (s[2] * x + s[3]);
160                     desX = x.ToString("0.0000000", System.
                            Globalization.CultureInfo.
                            InvariantCulture);
161                     desY = y.ToString("0.0000000", System.
                            Globalization.CultureInfo.
                            InvariantCulture);
162                     //Console.Write($"({desX},{desY}), \n");
163                     //Console.Write($"{desX},{desY}\n");
164                     Punkt Koord;
165                     Koord.x = x;
166                     Koord.y = y;
167                     Loesung[j] = Koord;
168                 }
169                 f_a = 3 * Math.Pow(Punkte[i + 1].x, 2) * s[0]
                        + 2 * Punkte[i + 1].x * s[1] + s[2];
170                 f_aa = 6 * Punkte[i + 1].x * s[0] + 2 * s[1];
171
172                 //if (f_aa > Schwellwert)         //f_aa ist
                        die 2te Ableitung
173                 //    f_aa *= Faktor;         //die 2te
                        Ableitung wird verkleinert
174
175                 //Punkt pruef;
176                 //double m, t;
177                 //pruef = linear2(Punkte, i + 2);
```

```
178                 //m = pruef.x;
179                 //t = pruef.y;
180
181                 //if (m - f_a > 0.3)
182                 //    f_a = m;
183
184                 string a, b, c, d;
185                 a = s[0].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
186                 b = s[1].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
187                 c = s[2].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
188                 d = s[3].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
189                 Console.Write($"a_{i + 1}(x) = Wenn[{Punkte[i
                        ].x} <= x <= {Punkte[i + 1].x}, ({a})*x
                        ^3+({b})*x^2+({c})*x+{d}]\n");
190             }
191             return Loesung;
192         }
193
194         static public Punkt[] linSpl(Punkt[] Punkte, double
                density)
195         {
196             double f_a = 0;
197             double m, t;
198             m = 0;
199
200             Punkt[] Loesung = new Punkt[101];
201             for (int i = 0; i < Punkte.Length - 1; i++)
202             {
203                 if (i == 0)
204                 {
205                     Punkt pruef;
206                     pruef = linear2(Punkte, i);
207                     m = pruef.x;
```

```
208                 f_a = m;
209             }
210
211             var A = Matrix<double>.Build.DenseOfArray(new
                    double[,] {
212             {Math.Pow(Punkte[i].x, 3), Math.Pow(Punkte[i
                    ].x, 2), Punkte[i].x, 1},
213             {Math.Pow(Punkte[i+1].x, 3), Math.Pow(Punkte[
                    i+1].x, 2), Punkte[i+1].x, 1},
214             {Math.Pow(Punkte[i].x, 2) * 3, Punkte[i].x *
                    2, 1, 0},
215             {Math.Pow(Punkte[i+1].x, 2) * 3, Punkte[i+1].
                    x * 2, 1, 0}
216             });
217             var l = Vector<double>.Build.Dense(new double
                    [] { Punkte[i].y, Punkte[i + 1].y, f_a, m
                    });
218             var s = A.Solve(l);
219             for (int j = 0; j < 10; j++)
220             {
221                 string desX = "";
222                 string desY = "";
223                 double x = 0;
224                 double y = 0;
225                 x = ((Punkte[i + 1].x - Punkte[i].x) * j
                        / density) + Punkte[i].x;
226                 y = (s[0] * Math.Pow(x, 3)) + (s[1] *
                        Math.Pow(x, 2)) + (s[2] * x + s[3]);
227
228                 x *= 0.00003;
229                 y *= 0.00003;
230
231                 x += 48.2719362;
232                 y += 11.5787503;
233
234                 desX = x.ToString("0.000000", System.
                        Globalization.CultureInfo.
                        InvariantCulture);
235                 desY = y.ToString("0.000000", System.
                        Globalization.CultureInfo.
                        InvariantCulture);
```

```
236                     //Console.Write($"({desX},{desY}), \n");
237                     //Console.Write($"{desX},{desY}\n");
238                     Console.Write($"{(i * 10 + j)+1}  0   3
                            16  0.000000    0.000000    0.000000
                              0.000000    {desX}   {desY}
                           10.000000   1\n");
239                     Punkt Koord;
240                     Koord.x = x;
241                     Koord.y = y;
242                     Loesung[j] = Koord;
243                 }
244                 f_a = 3 * Math.Pow(Punkte[i + 1].x, 2) * s[0]
                        + 2 * Punkte[i + 1].x * s[1] + s[2];
245
246
247                 if (i < Punkte.Length - 3)
248                 {
249                     Punkt pruef;
250                     pruef = linear2(Punkte, i + 2);
251                     m = pruef.x;
252                     t = pruef.y;
253                 }
254                 string a, b, c, d;
255                 a = s[0].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
256                 b = s[1].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
257                 c = s[2].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
258                 d = s[3].ToString("0.000000", System.
                        Globalization.CultureInfo.InvariantCulture
                        );
259                 //Console.Write($"l_{i+1}(x) = Wenn[{Punkte[i
                        ].x} <= x <= {Punkte[i+1].x}, ({a})*x^3+({
                        b})*x^2+({c})*x+({d})]\n");
260             }
261             return Loesung;
262         }
```

```
263
264         static public void linear(Punkt[] Punkte)
265         {
266             for (int i = 0; i < Punkte.Length - 1; i++)
267             {
268                 double m, t;
269                 m = (Punkte[i + 1].y - Punkte[i].y) / (Punkte
                        [i + 1].x - Punkte[i].x);
270                 t = Punkte[i + 1].y - m * Punkte[i + 1].x;
271                 for (int j = 0; j < 10; j++)
272                 {
273                     string desX = "";
274                     string desY = "";
275                     double x = 0;
276                     double y = 0;
277                     x = ((Punkte[i + 1].x - Punkte[i].x) * j
                            / 10.0) + Punkte[i].x;
278                     y = m * x + t;
279                     desX = x.ToString("0.0000000", System.
                            Globalization.CultureInfo.
                            InvariantCulture);
280                     desY = y.ToString("0.0000000", System.
                            Globalization.CultureInfo.
                            InvariantCulture);
281                     //Console.Write($"({desX},{desY}), ");
282                     //Console.Write($"{desX},{desY}\n");
283                 }
284             }
285         }
286         tatic public Punkt bezier(Punkt[] Punkte, Vektor
                velocity, double percent)
287         {
288
289             Punkt Kontrollpunkt = new ConsoleApplication1.
                    Punkt();
290             Kontrollpunkt.x = Punkte[0].x + velocity.x;
291             Kontrollpunkt.y = Punkte[0].y + velocity.y;
292
293             //Vektor vekAK = new ConsoleApplication1.Vektor(
                    Kontrollpunkt.x - Punkte[0].x, Kontrollpunkt.y
                     - Punkte[0].y);
```

```
294                 Vektor vekAK = new ConsoleApplication1.Vektor(
                        velocity.x, velocity.y);
295                 Vektor vekAKperc = new ConsoleApplication1.Vektor
                        ();
296                 Vektor vekKB = new ConsoleApplication1.Vektor(
                        Punkte[1].x - Kontrollpunkt.x, Punkte[1].y -
                        Kontrollpunkt.y);
297                 Vektor vekKBperc = new ConsoleApplication1.Vektor
                        ();
298                 vekAKperc = vekAK.VektorMul(percent);
299                 vekKBperc = vekKB.VektorMul(percent);
300                 //Console.Write("X:{0} ; Y:{1}", vekAKperc.x ,
                        vekAKperc.y);
301                 Vektor vekPerc = new ConsoleApplication1.Vektor(
                        vekAK.x - vekAKperc.x + vekKBperc.x, vekAK.y -
                         vekAKperc.y + vekKBperc.y);
302                 Punkt Destination = new Punkt(vekAKperc.x +
                        Punkte[0].x + vekPerc.VektorMul(percent).x,
                        vekAKperc.y + Punkte[0].y + vekPerc.VektorMul(
                        percent).y);
303
304                 return Destination;
305             }
306         }
307 }
```

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Unterschleißheim den 25.11.16
Ort Datum

C. Wobel
Unterschrift des Verfassers