Näherung für die Länge einer Bézierkurve

Name: Benjamin Wand

Matrikelnummer: 0496238

Studiengang: Architektur B.Sc.

Fachsemester: 3

Seminar: Wissenschaftliches Arbeiten

Modul: Soziologie und Forschung der Architektur

Dozentin: Katja Stoetzer

Abgabedatum der Arbeit: 1. März 2025

Inhaltsverzeichnis

1 Einleitung

Seit 2018 erstelle ich 3D-Modelle in OpenSCAD[1], und habe 2020 mein eigenes, auf dem De Casteljau-Algorithmus[2] aufbauendes, Bézierkurven-Modul[3][4] geschrieben. Um die Effizienz der Berechnung zu optimieren, fehlt diesem aber noch eine initiale Abschätzung[5] der Länge der Kurve.

Meine Recherche zum Thema Länge von Bézierkurven ergab, zum Beispiel in Wikipedia oder bei Pomax[6], dass man die Kurve berechnet und dann ausmisst. Im Sommer 2024 habe ich festgestellt, dass mir in den meisten Fällen eine Approximation durch Summe einer aus vier Punkten bestehenden Kurve vom Ergebnis her reichen würde[7]. Ich möchte aber keine rekursive Berechnung sondern eine einfache Formel, zum Beispiel eine lineare Gleichung. Geht das denn? Diese Frage habe ich mir gestellt.

Folgendes habe ich in dieser Arbeit vor: in einem Jupyter Notebook Punkte aus Zufallszahlen zu erstellen, davon Bézierkurven zu berechnen und deren Länge zu messen, die Abstände zwischen den Punkten und die Länge der Kurven zu betrachten, und einen Zusammenhang zu ermitteln. Anschließend werde ich das Modell an Kurven aus richtigen Projekten testen.

Um in der Praxis nützlich zu sein, sind Formeln für drei, vier und fünf Kontrollpunkte von Nöten. Ausführlich werde ich in dieser Arbeit die Herleitung der Formel für vier Punkte im zweidimensionalen Raum beschreiben und auf die anderen Fälle nur am Ende kurz eingehen.

2 Hauptteil

Meine These ist, dass ein größenordnungsmäßig linearer Zusammenhang zwischen den Längen der Abstände der Punkte und der Länge der Kurve besteht.

2.1 Erwartete Formel

$$\begin{aligned} L = a \cdot |P_0 - P_1| \\ +b \cdot |P_0 - P_2| \\ +c \cdot |P_0 - P_3| \\ +d \cdot |P_1 - P_2| \\ +e \cdot |P_1 - P_3| \\ +f \cdot |P_2 - P_3| \end{aligned}$$

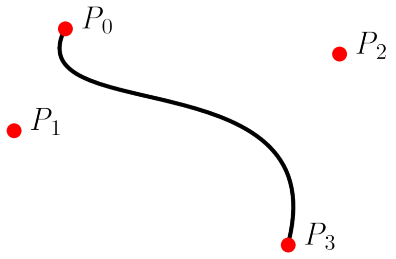

Abbildung 1: Bézierkurve mit $P_0 - P_3$

L ist die Länge der Kurve.
$|P_i - P_j|$ sind die Abstände zwischen den Punkten.
Gesucht sind die Variablen a-f.

2.2 „Menschengemachte“ Testdaten

Ein Projekt, bei dem ich mein Bézierkurvenmodul verwendet habe, sind in OpenSCAD geschriebene Ausstechförmchen[8]. Es handelt sich um das wonach es klingt: man kann die Files 3d-drucken und damit Plätzchenteig ausstechen.

Die darin vorkommenden Kurven bieten mir, im Kontrast zu den zufallsgenerierten Trainingsdaten, einen Datensatz Bézierkurven aus einem menschlichen schöpferischen Prozess, und sie sollen als Testdaten dienen.

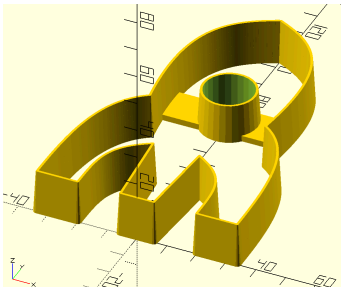

Abbildung 2: Ausstechförmchen in Raketenform

2.3 Maschinenlernen im Jupyter Notebook

Um den Berechnungsprozess im eigenen Jupyter Notebook nachzuvollziehen, ist der Code ist unter https://github.com/benjaminwand/bezier-curve-length/releases/tag/v1.0.0 [9] im File `german 4_points_2d.ipynb` , und zusätzlich im Anhang dieses Dokumentes, zu finden. Ich fahre Zelle für Zelle fort.

2.3.1 Funktionen und Bibliotheken

In dieser Zelle werden Bibliotheken importiert und Funktionen definiert. Es gibt keine Ausgabe aber das Ausführen der Zelle ist für das Funktionieren des folgenden Codes unerlässlich.

2.3.2 Bézierkurven zufallsgenerieren und vermessen

Hier werden die zufälligen Kurven generiert, und die Kontrollpunkte, Länge der Kurven, und Abstände zwischen den Kontrolpunkten zur Ansicht ausgegeben.

2.3.3 Plotten der ersten 15 zufallsgenerierten Bézierkurven

Um einen visuellen Eindruck zu bekommen womit wir es zu tun haben, werden die ersten 15 Kurven geplottet.

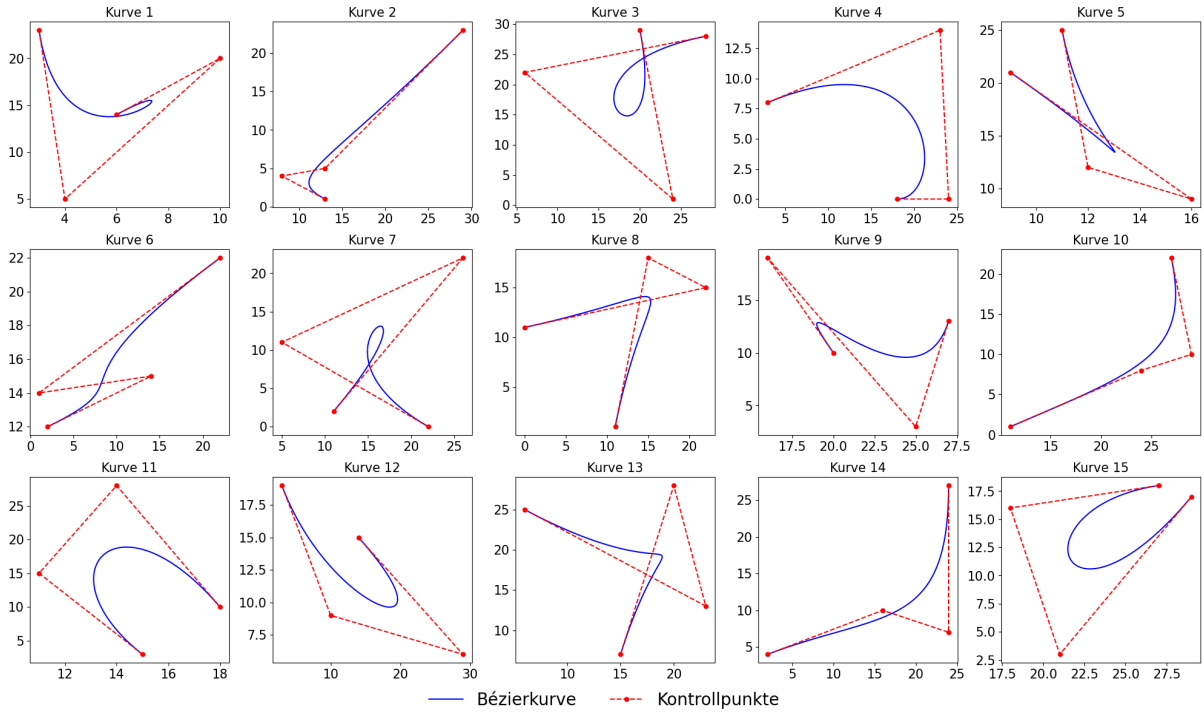

Abbildung 3: Bézierkurven aus zufallsgenerierten Punkten

Sie haben aufgrund der Zufallszahlen jetzt andere Kurven als ich. Ihre Kurven werden aber vermutlich genau so unterschiedlich geformt sein wie die hier aufgeführten.

2.3.4 Modell fitten und ausgeben

Um die Qualität des gleich entstehenden Modells bewerten zu können, werden Bereich, Durchschnitt und Median der Längen der Kurven ausgegeben.

Dann wird die große Tabelle aus Zelle 2 in zwei Teile geteilt: 80% Trainigsdaten und 20% Testdaten.

In folgenden Teil des Codes findet das Maschinenlernen im engeren Sinne statt:

```
# Initialisierung eines linearen Regressionsmodells
model = LinearRegression()
# Training des Modells mit den Trainingsdaten
model.fit(X_train, y_train)
```

Es werden Mittlere quadratische Abweichung (Mean Squared Error)[10], y-Achsenmaß (Intercept)[11] und die Gewichte der Linearen Regression ausgegeben.

```
Längen der Kurven: [7.34708, 47.28719]
Mittelwert: 25.34824
Median: 25.407114999999997

Mean Squared Error: 1.2525723257779093
Intercept: 0.028790536475248985
```

```
Gewichte der Linearen Regression:
  Variable  Gewicht  Feature
0        a  0.35524  D_P0_P1
1        b  0.39289  D_P0_P2
2        c  0.23464  D_P0_P3
3        d -0.09336  D_P1_P2
4        e  0.39145  D_P1_P3
5        f  0.35198  D_P2_P3
```

Ein y-Achsenabschnitt von nahe null ist in diesem Fall gut. Eine Mittlere quadratische Abweichung von 1.25257 bei y = [7.34708, 47.28719] ist auch gut[12], das Modell funktioniert.

Wir können die Gewichte in die Formel einsetzen und erhalten:

$$\begin{aligned} L = 0.35071 \cdot |P_0 - P_1| \\ +0.38105 \cdot |P_0 - P_2| \\ +0.23045 \cdot |P_0 - P_3| \\ -0.08912 \cdot |P_1 - P_2| \\ +0.39733 \cdot |P_1 - P_3| \\ +0.35840 \cdot |P_2 - P_3| \end{aligned}$$

2.3.5 Plotten der realen Werte versus der Vorhersage

Um zusätzlich eine visuelle Bestätigung zu erhalten, werden hier die Längen der Kurven aus der 20% Testgruppe, zusammen mit einer Linie die die Vorhersage darstellt, geplottet.

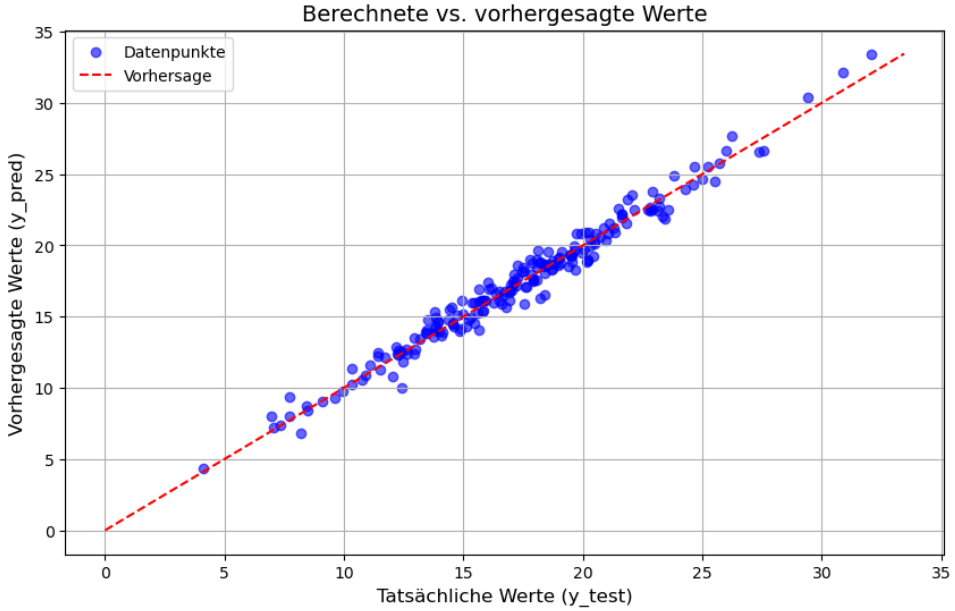

Abbildung 4: Plot der Längen aus der 20% Testgruppe vs. Vorhersage

2.3.6 Einführen und Plotten von Testdaten aus realen Projekten

Nun ist zu 99% sicher dass das Modell funktioniert und die Formel so angewendet werden kann. Da aber auf zufallsgenerierten Kurven gefittet wurde, und der Verdacht naheliegt, dass Kurven aus real existierenden Designprojekten anders sind, möchte ich es noch auf Daten aus eigenen Designprojekten ausprobieren.

Von https://github.com/benjaminwand/cookie-cutters [8] habe ich Bézierkurven mit vier Kontrollpunkten ausgewählt. Sie werden zur Ansicht geplottet.

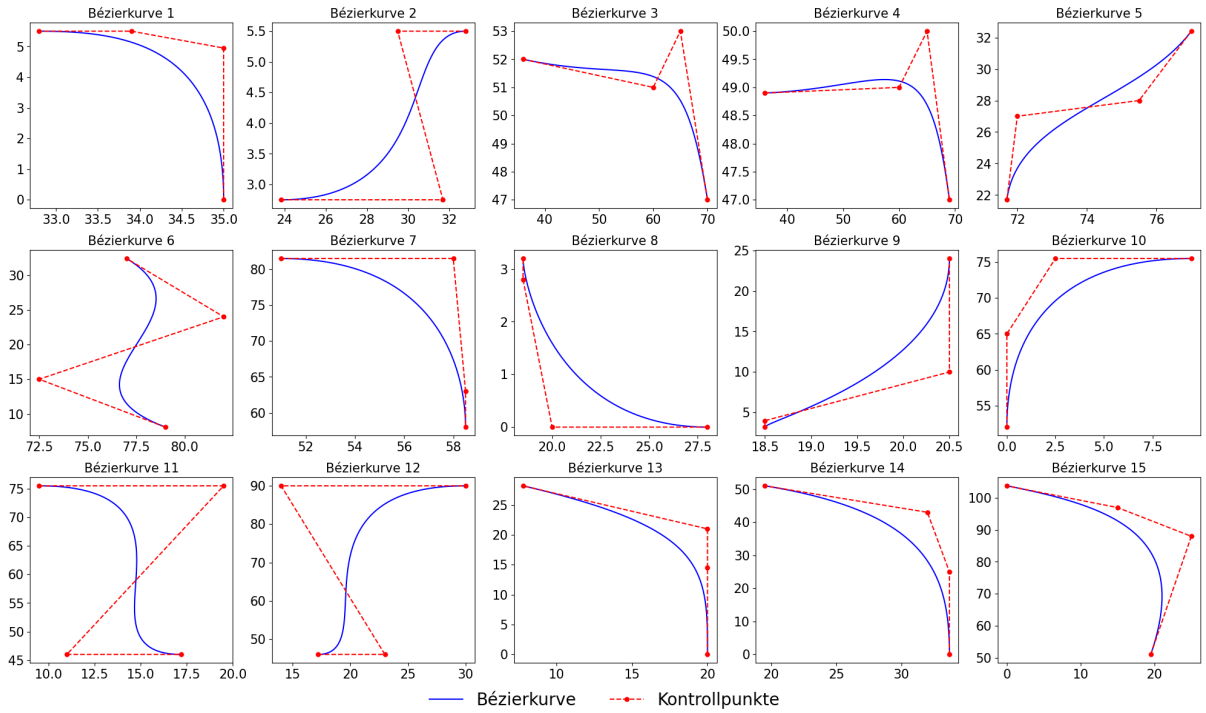

Abbildung 5: Menschengemachte Bézierkurven

Wir sehen, dass diese Kurven viel weniger bewegt aussehen als die Kurven aus den zufallsgenerierten Punkten. Bei den Kurven 1, 2, 7, 8, 9, 10, 11 und 12 habe ich ein Arbeitsprinzip angewandt, welches in der Typographie üblich ist[13]: die Linie zwischen den Endpunkten und dem Punkt daneben ist horizontal oder vertikal. Dass derartiges in zufallsgenerierten Kurven passiert, ist extrem unwahrscheinlich. Die leichte Skepsis gegenüber dem Modell und die zusätzliche Überprüfung mit echten Daten ist also gerechtfertigt.

2.3.7 Berechnung und Ausgabe der Längen und Abstände zwischen den Kontrollpunkten für den Testdatensatz

Jetzt erstellen wir eine Tabelle der Länge der Kurven und Abstände zwischen den Kurven, ähnlich wie in Zelle 2, aber von den Daten aus Zelle 6. Dies dient der Vorbereitung für den nächsten Schritt.

2.3.8 Ausgabe der Mittleren quadratischen Abweichung

Nun wird die Mittlere quadratische Abweichung berechnet und zusammen mit den Längen der Kurven, Mittelwert und Median ausgegeben.

```
Längen der Kurven: [6.5595, 61.8815]
Mittelwert: 29.3494
Median: 27.6438

Mean Squared Error (MSE): 0.583240757390104
```

Auch das ist im Verhältnis zu den Längen eine gute Mittlere quadratische Abweichung, das Modell funktioniert.

Durch geeignete Skalierung sind die „menschengemachten“ und zufallsgenerierten Kurven etwa gleich lang, die Mittlere quadratische Abweichung ist hier aber deutlich kleiner als bei den zufallsgenerierten Kurven. Ich gehe davon aus, dass das daran liegt, dass die „menschengemachten“ Kurven weniger chaotisch sind (große Radien, keine Selbstüberschneidungen, kein kreuzender Polygonzug[14]) als die zufallsgenerierten.

2.3.9 Erstellen des Plots der Vorhersage und Testdaten gegenüberstellt

Um die gute Vorhersage des Modells noch visuell darzulegen, werden hier die Längen der Kurven im Vergleich zur Vorhersage geplottet.

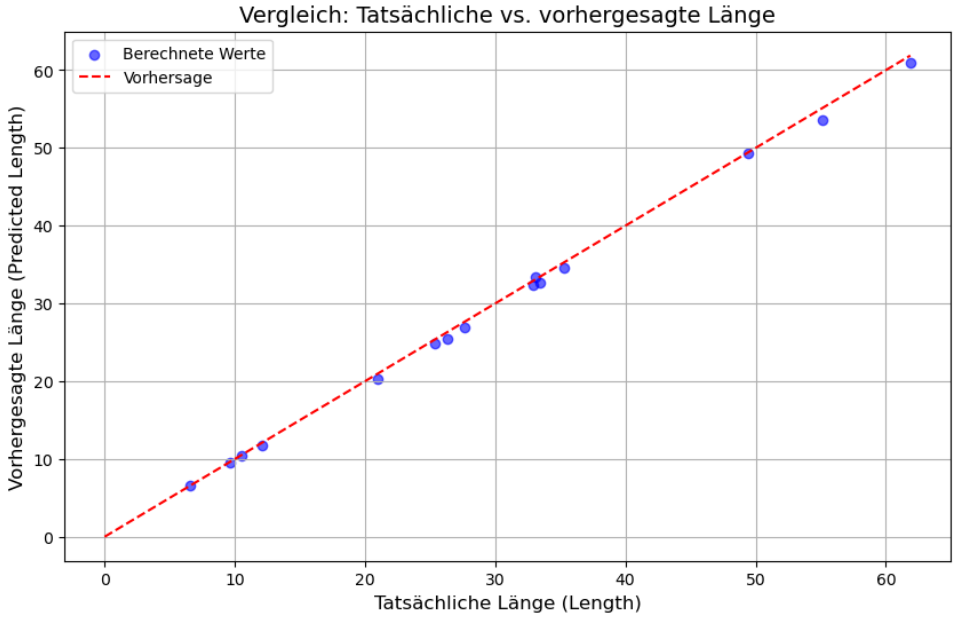

Abbildung 6: Plot der Längen der Plätzchenförmchen-Kurven vs. Vorhersage

2.4 Vereinfachung und Interpretation der Formel

Es konnte also gezeigt werden, dass das Modell eine ausreichend präzise Vorhersage macht. Wenn man es analytisch betrachtet, wäre aber ein symmetrisches Modell gesucht, der Abstand zwischen P_0 und P_1 müsste den gleichen Einfluss haben wie der Abstand zwischen P_2 und P_3, und der Abstand zwischen P_0 und P_2 müsste den gleichen Einfluss haben wie der Abstand zwischen P_1 und P_3. Durch Zusammenschau der Parameter nach mehreren Durchgängen des Fittings schienen sie sich folgenden Werte anzunähern:

$$\begin{aligned} L = {} & 0.35 \cdot |P_0 - P_1| \\ & +0.40 \cdot |P_0 - P_2| \\ & +0.23 \cdot |P_0 - P_3| \\ & -0.09 \cdot |P_1 - P_2| \\ & +0.40 \cdot |P_1 - P_3| \\ & +0.35 \cdot |P_2 - P_3| \end{aligned}$$

Der negative Wert $-0.09 \cdot |P_1 - P_2|$ fällt auf. Ich interpretiere ihn so: S-förmige Kurven sind, im Verhältnis zu den Abständen der Punkte, kürzer als C-förmige, weil bei S-förmigen P_1 und P_2 stärker gegeneinander arbeiten. Und je weiter P_1 und P_2 von einander entfernt sind, desto stärker arbeiten sie gegeneinander. Deshalb kürzt $|P_1 - P_2|$ die Kurve.

2.5 Weitere Fälle

In weiteren Files habe ich andere Fälle untersucht.

2.5.1 Drei und fünf Kontrollpunkte

Für drei und fünf Kontrollpunkte habe ich folgende Formeln ermitteln können:

$$\begin{aligned} L = {} & 0.43 \cdot |P_0 - P_1| \\ & +0.53 \cdot |P_0 - P_2| \\ & +0.43 \cdot |P_1 - P_2| \end{aligned}$$

$$\begin{aligned} L = {} & 0.32 \cdot |P_0 - P_1| \\ & +0.35 \cdot |P_0 - P_2| \\ & +0.23 \cdot |P_0 - P_3| \\ & +0.10 \cdot |P_0 - P_4| \\ & -0.13 \cdot |P_1 - P_2| \\ & +0.20 \cdot |P_1 - P_3| \\ & +0.23 \cdot |P_1 - P_4| \\ & -0.13 \cdot |P_2 - P_3| \\ & +0.35 \cdot |P_2 - P_4| \\ & +0.32 \cdot |P_3 - P_4| \end{aligned}$$

2.5.2 Drei Dimensionen

Zusätzlich habe ich in den Files mit Endung `..._3d.ipynb` Kurven in drei Dimensionen betrachtet aber nicht mit Daten aus echten Designprojekten überprüft. Die entstehenden Formeln sehen denen in zwei Dimensionen ähnlich genug dass ich die Formeln aus den zweidimensionalen Modellen in der Praxis übernehmen würde.

2.5.3 Decimal und erzwungene Symmetrie

Weil mir die persistierende Asymmetrie der Modelle missfiel, habe ich einmal Decimal[15] statt float verwendet (`german_4_points_2d_decimal.ipynb`) und einmal beim Fitten Symmetrie erzwungen(`german 4_points_2d symmetrical.ipynb`). Das hat aber zu keiner überzeugenden Verbesserung geführt.

2.5.4 Quadratische Formeln

Meine Versuche mit quadratischen Formeln (`german 4_points_2d_polynome.ipynb` und `german 4_points_2d_polynome_combinations.ipynb`) haben zu Overfitting geführt. Es bleibt also bei den linearen Gleichungen.

3 Fazit

Ziel der Arbeit war, die Länge einer Bézierkurve durch eine einfach zu berechnende Gleichung zu approximieren. Dies ist mir in Form von linearen Gleichungen gelungen.

Für drei, vier und fünf Kontrollpunkte lauten die Gleichungen:

$$
\begin{aligned}
L = 0.43 \cdot |P_0 - P_1| \\
+0.53 \cdot |P_0 - P_2| \\
+0.43 \cdot |P_1 - P_2|
\end{aligned}
$$

$$
\begin{aligned}
L = 0.35 \cdot |P_0 - P_1| \\
+0.40 \cdot |P_0 - P_2| \\
+0.23 \cdot |P_0 - P_3| \\
-0.09 \cdot |P_1 - P_2| \\
+0.40 \cdot |P_1 - P_3| \\
+0.35 \cdot |P_2 - P_3|
\end{aligned}
$$

$$
\begin{aligned}
L = 0.32 \cdot |P_0 - P_1| \\
+0.35 \cdot |P_0 - P_2| \\
+0.23 \cdot |P_0 - P_3| \\
+0.10 \cdot |P_0 - P_4| \\
-0.13 \cdot |P_1 - P_2| \\
+0.20 \cdot |P_1 - P_3| \\
+0.23 \cdot |P_1 - P_4| \\
-0.13 \cdot |P_2 - P_3| \\
+0.35 \cdot |P_2 - P_4| \\
+0.32 \cdot |P_3 - P_4|
\end{aligned}
$$

3.1 Persönliches und Ausblick

Dies war mein erstes Maschinenlernen Projekt und es hat besser funktioniert als erwartet. Das Modell hat auf den zufallsgenerierten Daten so gute Vorhersagen gemacht, dass die Daten zu filtern unnötig war. Dies hat mich erstaunt. Auch meine Intuition, dass es einen linearen Zusammenhang gibt, hat sich erfreulicherweise bestätigt.

Dass die Modelle wirklich immer asymmetrisch sind, stört mich etwas, und ich kann das zum Anlass nehmen, in Zukunft mehr über Maschinenlernen zu lernen. Des weiteren könnten die dreidimensionalen Modelle noch mit Daten aus richtigen Projekten zu überprüft werden.

3.2 Einordnung in den Wissenschaftskontext

What a time to be alive!

Ohne Codesnippets von ChatGPT hätte ich diese Arbeit wegen Ungeübtheit in Python nicht in wenigen Wochen schreiben können, sondern ich hätte Monate gebraucht. Durch die Veröffentlichung der kompletten Protokolle stelle ich der Nachwelt ein Dokument zur Verfügung, von dessen Art es hoffentlich viele geben wird: Zeugnisse der frühen Verwendung von Large Language Models[16] in wissenschaftlichen Arbeiten, in einer Form, die sich hoffentlich als legitim und zukunftsweisend herausstellen wird.

Ob meine Formeln nun auch von anderen verwendet werden werden ist unsicher, veröffentlicht sind sie jedenfalls[17]. Ich vermute, dass Hersteller von Architektur CAD Programmen bereits über vergleichbare Formeln verfügen. Da das aber proprietäre Produkte sind, behalten sie ihr Geschäftsgeheimnis für sich.

A Anhang

Diese Datei wurde am 27. Januar 2025 gerendert.

A.1 Literatur

[1] OpenSCAD. Heruntergeladen am 2024-12-11 20:29:15. [Online]. Available: https://openscad.org

[2] "De-casteljau-algorithmus," page Version ID: 224617113, Heruntergeladen am 2025-01-06 18:29:08. [Online]. Available: https://de.wikipedia.org/w/index.php?title=De-Casteljau-Algorithmus&oldid=224617113

[3] B. Wand. Bézier curves in OpenSCAD. Archive.org Snapshot vom 11. Dezember 2024. Heruntergeladen am 2025-01-01 14:20:22. [Online]. Available: https://web.archive.org/web/20241211080206/https://benjaminwand.github.io/verbose-cv/projects/bezier_curves.html

[4] "Bézierkurve," page Version ID: 248905880, Heruntergeladen am 2024-12-09 14:37:28. [Online]. Available: https://de.wikipedia.org/w/index.php?title=B%C3%A9zierkurve&oldid=248905880

[5] "Approximation," page Version ID: 224262758, Heruntergeladen am 2025-01-04 16:55:30. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Approximation&oldid=224262758

[6] M. P. Kamermans. A primer on bézier curves. Heruntergeladen am 2024-12-11 20:49:36. [Online]. Available: https://pomax.github.io/bezierinfo

[7] B. Wand. Benjamin wand - data science baby steps. Heruntergeladen am 2025-01-05 20:23:58. [Online]. Available: https://benjaminwand.github.io/verbose-cv/projects/data_science_baby_steps.html

[8] ——, "benjaminwand/cookie-cutters," original-date: 2020-09-21T15:47:34Z, Heruntergeladen am 2024-12-15 17:42:25. [Online]. Available: https://github.com/benjaminwand/cookie-cutters

[9] ——. Release january_2025 benjaminwand/bezier-curve-length. Hinzugefügt am 5.1.2025, 17:05:14. [Online]. Available: https://github.com/benjaminwand/bezier-curve-length/releases/tag/v1.0.0

[10] "Mittlere quadratische abweichung," page Version ID: 248431810, Heruntergeladen am 2024-12-16 15:03:15. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Mittlere_quadratische_Abweichung&oldid=248431810

[11] "y-achsenabschnitt," page Version ID: 231429725, Heruntergeladen am 2024-12-18 16:37:54. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Y-Achsenabschnitt&oldid=231429725

[12] N. Lang. Understanding mean squared error (MSE) - a key metric in data analysis! | data basecamp. Section: Statistics, Heruntergeladen am 2024-12-17 12:26:07. [Online]. Available: https://databasecamp.de/en/statistics/mean-squared-error

[13] Schritt für schritt zum eigenen schriftsatz... Heruntergeladen am 2024-12-18 19:06:08. [Online]. Available: https://fontforge.org/docs/old/de/editexample.html

[14] “Polygonzug (mathematik),” page Version ID: 251382540, Hinzugefügt am 2025-01-01 19:56:30. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Polygonzug_(Mathematik)&oldid=251382540

[15] decimal - decimal fixed-point and floating-point arithmetic. Heruntergeladen am 2024-12-18 20:11:56. [Online]. Available: https://docs.python.org/3/library/decimal.html

[16] “Large language model,” page Version ID: 251289716, Heruntergeladen am. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Large_Language_Model&oldid=251289716

[17] B. Wand. Approximation of bézier curve length. Heruntergeladen am 1.1.2025, 14:16:51. [Online]. Available: https://benjaminwand.github.io/verbose-cv/projects/length_bezier.html

A.2 Abbildungsverzeichnis

A.3 ChatGPT Protokolle

Für Recherche und Codesnippets verwendete ich ChatGPT. Die Chats sind hier vollständig ausgewiesen.

https://chatgpt.com/share/6755d1df-a774-8009-bd6d-da52f88cbd0e
https://chatgpt.com/share/6755d1c8-2290-8009-90ca-a64c23b56391
https://chatgpt.com/share/6755d19c-cbc8-8009-868c-59a9a27e46db
https://chatgpt.com/share/6755d258-a7fc-8009-940d-443c0fe541a9
https://chatgpt.com/share/675ae825-fb50-8009-b447-b468b94ed9e6
https://chatgpt.com/share/675d9f26-4e8c-8009-81f1-a5eb8717ddc8
https://chatgpt.com/share/6762a821-13a0-8009-a282-da9c5c39f991
https://chatgpt.com/share/676b1034-e9b4-8009-a92e-deeebbe527c6
https://chatgpt.com/share/6779642a-5ec4-8009-a320-d162de30e7d4

A.4 Jupyter Notebook

Dies ist das Jupyter Notebook für die im Paper besprochene Version der Kubischen Bézierkurve in zwei Dimensionen, `"german 4_points_2d.ipynb"`. Die anderen Versionen befinden sich auf GitHub unter
https://github.com/benjaminwand/bezier-curve-length/releases/tag/v1.0.0

```
# Version mit vier Punkten und zwei Dimensionen

# 1. Funktionen und Bibliotheken

# Löscht alle Variablen aus der aktuellen Sitzung um sicherzustellen
# dass keine alten Werte oder Definitionen den Code beeinflussen.
%reset -f

# Importieren der benötigten Bibliotheken
import numpy as np  # Für numerische Berechnungen
import pandas as pd  # Für die Arbeit mit Tabellen Datensätzen
import matplotlib.pyplot as plt # Für graphische Darstellung
from sklearn.model_selection import train_test_split  # Für das Teilen der
    Daten in Trainings- und Testsets
from sklearn.linear_model import LinearRegression  # Für die lineare
    Regressionsanalyse
from sklearn.metrics import mean_squared_error  # Für die Berechnung der
    mittleren quadratischen Abweichung (MSE)

# Definition einer Funktion für die Berechnung von Punkten auf einer kubischen
    Bézierkurve
def cubic_bezier(points, t):
    # Berechnet einen Punkt auf der Bézierkurve für den gegebenen Parameter t
    return ((1 - t)**3 * points[0] +  # Einfluss des ersten Kontrollpunkts
            3 * (1 - t)**2 * t * points[1] +  # Einfluss des zweiten
                Kontrollpunkts
            3 * (1 - t) * t**2 * points[2] +  # Einfluss des dritten
                Kontrollpunkts
            t**3 * points[3])  # Einfluss des vierten Kontrollpunkts

# Funktion zur Generierung von 100 Punkten auf einer Bézierkurve
def generate_bezier_points(points, num_points=100):
    # Generiert num_points Punkte auf der Bézierkurve
    return np.array([cubic_bezier(points, t) for t in np.linspace(0, 1,
        num_points)])

# Funktion zur Berechnung der Länge einer Kurve
def curve_length(curve):
    # Berechnet die Gesamtlänge einer Kurve basierend auf den Distanzen
        zwischen aufeinanderfolgenden Punkten
    distances = np.sqrt(np.sum(np.diff(curve, axis=0)**2, axis=1))
    return np.sum(distances)

# Funktion zur Berechnung der Distanz zwischen zwei Punkten
def point_distance(p1, p2):
    p1 = np.array(p1)  # Umwandeln der Punkte in numpy Arrays
    p2 = np.array(p2)
    # Berechnung der euklidischen Distanz
    return np.sqrt(np.sum((p1 - p2)**2))
```

```
# 2. Bézierkurven zufallsgenerieren und vermessen
# Dies erzeugt die Daten für späteres Maschinenlernen
# n kann angepasst werden aber bei großem n dauert es lange zu rendern

# Anzahl der Bézierkurven die erzeugt werden sollen
n = 1000

# Generierung von zufälligen Kontrollpunkten für n Bézierkurven
# Jede Bézierkurve hat 4 Kontrollpunkte im 2D-Raum
punkte_cubic = np.random.randint(low=0, high=30, size=(n, 4, 2))

# Initialisierung einer Liste zum Speichern der Ergebnisse
data = []

# Schleife über jede Gruppe von Kontrollpunkten
for i in range(n):
    control_points = punkte_cubic[i]  # Die Kontrollpunkte der aktuellen
        Bézierkurve
    bezier_points = generate_bezier_points(control_points)  # Generierung der
        Bézierkurve
    length = curve_length(bezier_points)  # Berechnung der Länge der
        Bézierkurve

    # Berechnung der Distanzen zwischen den Kontrollpunkten
    d01 = point_distance(control_points[0], control_points[1])  # Distanz
        zwischen P0 und P1
    d02 = point_distance(control_points[0], control_points[2])  # Distanz
        zwischen P0 und P2
    d03 = point_distance(control_points[0], control_points[3])  # Distanz
        zwischen P0 und P3
    d12 = point_distance(control_points[1], control_points[2])  # Distanz
        zwischen P1 und P2
    d13 = point_distance(control_points[1], control_points[3])  # Distanz
        zwischen P1 und P3
    d23 = point_distance(control_points[2], control_points[3])  # Distanz
        zwischen P2 und P3

    # Speichern der Ergebnisse als Eintrag in der Tabelle
    entry = {
        'P0': control_points[0].tolist(),  # Koordinaten von P0
        'P1': control_points[1].tolist(),  # Koordinaten von P1
        'P2': control_points[2].tolist(),  # Koordinaten von P2
        'P3': control_points[3].tolist(),  # Koordinaten von P3
        'Length': length,  # Länge der Bézierkurve
        'D_P0_P1': d01,  # Distanz zwischen P0 und P1
        'D_P0_P2': d02,  # Distanz zwischen P0 und P2
        'D_P0_P3': d03,  # Distanz zwischen P0 und P3
        'D_P1_P2': d12,  # Distanz zwischen P1 und P2
        'D_P1_P3': d13,  # Distanz zwischen P1 und P3
        'D_P2_P3': d23,  # Distanz zwischen P2 und P3
    }
    data.append(entry)  # Hinzufügen des Eintrags

# Erstellen eines Pandas DataFrame aus der Ergebnissammlung
df_trainingsdaten = pd.DataFrame(data)

print("Die Kontrollpunkte der Kurven, Länge der Kurven, und Abstände zwischen
    den Kontrollpunkten:\n")
print(df_trainingsdaten)
```

```
# 3. Plotten der ersten 15 zufallsgenerierten Bézierkurven zur Ansicht

fig, axes = plt.subplots(3, 5, figsize=(20, 12))  # 3x5 Raster für 15 Kurven
axes = axes.flatten()  # Umwandeln der Achsen in Eindimensionalität für
    einfachere Iteration

# Für die ersten 15 Bézierkurven
for idx, control_points in enumerate(punkte_cubic[:15]):  # Nur die ersten 15
    Kontrollpunkt-Sets
    bezier_curve = generate_bezier_points(control_points)  # Punkte auf der
        Bézierkurve generieren

    ax = axes[idx]
    control_points = np.array(control_points)  # Sicherstellen, dass
        Kontrollpunkte ein NumPy-Array sind

    # Plot der Bézierkurve
    ax.plot(bezier_curve[:, 0], bezier_curve[:, 1], color='blue')  # Blaue
        Bézierkurve
    # Kontrollpunkte plotten
    ax.plot(control_points[:, 0], control_points[:, 1], 'ro--', markersize=5)
        # Rote Kontrollpunkte
    ax.set_title(f"Kurve {idx + 1}", fontsize=15)
    ax.tick_params(axis='both', labelsize=15)  # Schriftgröe für die
        Achsenticks

# Entfernen überschüssiger Subplots, falls weniger als 15 Kurven vorhanden
for ax in axes[len(punkte_cubic[:15]):]:
    ax.axis('off')

# Globale Legende hinzufügen
fig.legend(
    ['Bézierkurve', 'Kontrollpunkte'],
    loc='lower center',
    ncol=2,
    fontsize=20,
    frameon=False
)

# Layout anpassen und anzeigen
plt.tight_layout(rect=[0, 0.05, 1, 1])  # Platz für die globale Legende unten
    schaffen
plt.show()
```

```
# 4. Modell fitten und ausgeben

# Ausgabe von Zusatzinformationen zur besseren Beurteilung
min_length = df_trainingsdaten['Length'].round(5).min()         # Kürzeste Länge
max_length = df_trainingsdaten['Length'].round(5).max()         # Längste Länge
mean_length = df_trainingsdaten['Length'].mean().round(5)       # Durchschnitt
median_length = df_trainingsdaten['Length'].round(5).median()  # Median
print(
    f"Längen der Kurven: [{min_length}, {max_length}]\n"
    f"Mittelwert: {mean_length}\n"
    f"Median: {median_length}\n"
)

# Definition der Merkmale (Features) und des Zielwerts (Target)
# X: Eingabevariablen (Abstände zwischen den Kontrollpunkten)
X = df_trainingsdaten[['D_P0_P1', 'D_P0_P2', 'D_P0_P3', 'D_P1_P2', 'D_P1_P3', '
    D_P2_P3']]
# y: Zielvariable (Länge der Bézierkurve)
y = df_trainingsdaten['Length']

# Aufteilen der Daten in Trainings- und Testdatensätze
# 80% der Daten werden für das Training verwendet, 20% für das Testen
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Erstellen und Trainieren des Modells
# Initialisierung eines linearen Regressionsmodells
model = LinearRegression()
# Training des Modells mit den Trainingsdaten
model.fit(X_train, y_train)

# Vorhersagen mit dem Modell
# Vorhersage der Zielwerte basierend auf den Testdaten
y_pred = model.predict(X_test)

# Evaluierung des Modells
# Berechnung der mittleren quadratischen Abweichung (MSE)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")  # Ausgabe der Abweichung

# Ausgabe der Modellparameter
# Die Koeffizienten der linearen Regression zeigen die Gewichtungen der
    einzelnen Merkmale
coefficients = model.coef_
intercept = model.intercept_  # Modell-Intercept
print(f"Intercept: {intercept}\n")  # Ausgabe des Intercept
features = X.columns  # Speichert die Spaltennamen von X in der Variablen
    features

# Labels für den Index erstellen, gleichen Variablennamen im Paper
labels = ['a', 'b', 'c', 'd', 'e', 'f']

# Erstellen eines DataFrame und setzen der Labels als Index
coef_df = pd.DataFrame({
    'Variable': labels,
    'Gewicht': coefficients.round(5),
    'Feature': features
})

# Ausgabe der Modellkoeffizienten
print("Gewichte der Linearen Regression:")
print(coef_df)
```

```
# 5. Plotten der realen Werte vs. der Vorhersage

# Anlage des Diagramms
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.6, color='blue', label='Datenpunkte')  #
    Streudiagramm

# Hinzufügen der Diagonalen für die Vorhersage
min_value = min(y_test.min(), y_pred.min(), 0)  # Kleinster Wert in den Daten
max_value = max(y_test.max(), y_pred.max())  # Gröter Wert in den Daten
plt.plot([min_value, max_value], [min_value, max_value], color='red', linestyle
    ='--', label='Vorhersage')

# Achsenbeschriftungen und Titel
plt.xlabel('Tatsächliche Werte (y_test)', fontsize=12)
plt.ylabel('Vorhergesagte Werte (y_pred)', fontsize=12)
plt.title('Berechnete vs. vorhergesagte Werte', fontsize=14)

# Hinzufügen einer Legende und eines Gitters
plt.legend()
plt.grid(True)

# Plot anzeigen
plt.show()
```

```
# 6. Einführen und Plotten von Testdaten aus realen Projekten

punkte_testdaten = [     # siehe https://github.com/benjaminwand/cookie-cutters
# corona virus
[[35, 0], [35, 4.95], [33.9, 5.5], [32.8, 5.5]],
[[23.835, 2.75], [31.7, 2.75], [29.5, 5.5], [32.8, 5.5]],
# chaosknoten
[[36.0, 52.0], [60.0, 51.0], [65.0, 53.0], [70.0, 47.0]],
 [[36.0, 48.9], [60.0, 49.0], [65.0, 50.0], [69.0, 47.0]],
 [[77.0, 32.4], [75.5, 28.0], [72.0, 27.0], [71.7, 21.7]],
 [[79.0, 8.1], [72.5, 15.0], [82.0, 24.0], [77.0, 32.4]],
# cactus
[[51, 81.5], [58, 81.5], [58.5, 63], [58.5, 58]],
[[28, 0], [20, 0], [18.5, 2.8], [18.5, 3.2]],
[[18.5, 3.2], [18.5, 4], [20.5, 10], [20.5, 24]],
[[0, 52], [0, 65], [2.5, 75.5], [9.5, 75.5]],
[[9.5, 75.5], [19.5, 75.5], [11, 46], [17.2, 46]],
[[17.2, 46], [23, 46], [14, 90], [30, 90]],
# fairydust rocket
[[20, 0], [20, 14.5], [20, 21], [7.8, 28.2]],
[[33.7, 0], [33.7, 25], [32, 43], [19.5, 51]],
[[19.5, 51], [25, 88], [15, 97], [0, 103.8]]
]

# Plotten der Bézierkurven aus den punkte_testdaten
fig, axes = plt.subplots(3, 5, figsize=(20, 12))  # 3x5 Raster für 15 Kurven
axes = axes.flatten()  # Achsen in flache Liste umwandeln

# Plot jeder Bézierkurve
for idx, control_points in enumerate(punkte_testdaten):
    bezier_curve = generate_bezier_points(np.array(control_points))  #
        Bézierkurve berechnen
    control_points = np.array(control_points)  # Kontrollpunkte in NumPy-Array
        umwandeln

    ax = axes[idx]
    # Bézierkurve plotten
    ax.plot(bezier_curve[:, 0], bezier_curve[:, 1], color="blue")
    # Kontrollpunkte plotten
    ax.plot(control_points[:, 0], control_points[:, 1], 'ro--', markersize=5)
    # Titel für jede Kurve
    ax.set_title(f"Bézierkurve {idx + 1}", fontsize=15)
    ax.tick_params(axis='both', labelsize=15)  # Schriftgröe für die
        Achsenticks

# Überschüssige Subplots deaktivieren (falls weniger als 15 Kurven)
for ax in axes[len(punkte_testdaten):]:
    ax.axis('off')

# Globale Legende hinzufügen
fig.legend(
    ['Bézierkurve', 'Kontrollpunkte'],
    loc='lower center',
    ncol=2,
    fontsize=20,
    frameon=False
)

# Layout anpassen und anzeigen
plt.tight_layout(rect=[0, 0.05, 1, 1])  # Platz für die globale Legende unten
    schaffen
plt.show()
```

```
# 7. Berechnung und Ausgabe der Abstände zwischen den Kontrollpunkten für den
    Testdatensatz

test_daten = []
for control_points in punkte_testdaten:

    # Kontrollpunkte in NumPy-Array konvertieren
    control_points = np.array(control_points)

    bezier_points = generate_bezier_points(control_points)  # Generierung der
        Bézierkurve
    length = curve_length(bezier_points)  # Berechnung der Länge der
        Bézierkurve

    d01 = point_distance(control_points[0], control_points[1])  # Distanz
        zwischen P0 und P1
    d02 = point_distance(control_points[0], control_points[2])  # Distanz
        zwischen P0 und P2
    d03 = point_distance(control_points[0], control_points[3])  # Distanz
        zwischen P0 und P3
    d12 = point_distance(control_points[1], control_points[2])  # Distanz
        zwischen P1 und P2
    d13 = point_distance(control_points[1], control_points[3])  # Distanz
        zwischen P1 und P3
    d23 = point_distance(control_points[2], control_points[3])  # Distanz
        zwischen P2 und P3

    # Speichern der Ergebnisse als Eintrag
    entry = {
        'D_P0_P1': d01,   # Distanz zwischen P0 und P1
        'D_P0_P2': d02,   # Distanz zwischen P0 und P2
        'D_P0_P3': d03,   # Distanz zwischen P0 und P3
        'D_P1_P2': d12,   # Distanz zwischen P1 und P2
        'D_P1_P3': d13,   # Distanz zwischen P1 und P3
        'D_P2_P3': d23,   # Distanz zwischen P2 und P3
        'Length': length, # Länge der Bézierkurve
    }
    test_daten.append(entry)

# Umwandeln der Testdaten in ein DataFrame
df_test = pd.DataFrame(test_daten)

# Berechnung und Hinzufügen der Vorhersagen für jede Zeile im DataFrame df_test
df_test['Predicted_Length'] = (
    df_test[features].dot(coefficients) + intercept
)

# Ausgabe des DataFrame
print("Die Kontrollpunkte der Kurven, Länge der Kurven, und Abstände zwischen
    den Kontrollpunkten:\n")
print(df_test)
```

```
# 8. Ausgabe der mittleren quadratischen Abweichung
# mit einigen Zusatzinformationen zur besseren Beurteilung

# Bestimmen und Ausgabe der kürzesten und längsten Länge
min_length = df_test['Length'].round(4).min()     # Kürzeste Länge
max_length = df_test['Length'].round(4).max()     # Längste Länge
mean_length = df_test['Length'].mean().round(4)   # Durchschnitt
median_length = df_test['Length'].round(4).median()  # Median

print(
    f"Längen der Kurven: [{min_length}, {max_length}]\n"
    f"Mittelwert: {mean_length}\n"
    f"Median: {median_length}\n"
)

# Vergleich der vorhergesagten und tatsächlichen Längen
mse = mean_squared_error(df_test['Length'], df_test['Predicted_Length'])
print(f"Mean Squared Error (MSE): {mse}")
```

```
# 9. Erstellen des Plots der Vorhersage und Testdaten gegenüberstellt

plt.figure(figsize=(10, 6))
plt.scatter(df_test['Length'], df_test['Predicted_Length'], color='blue', alpha
    =0.6, label='Berechnete Werte')

# Hinzufügen der Diagonalen für perfekte Vorhersage
max_value = max(df_test['Length'].max(), df_test['Predicted_Length'].max())
min_value = min(df_test['Length'].min(), df_test['Predicted_Length'].min(), 0)
plt.plot([min_value, max_value], [min_value, max_value], color='red', linestyle
    ='--', label='Vorhersage')

# Achsenbeschriftungen und Titel
plt.xlabel('Tatsächliche Länge (Length)', fontsize=12)
plt.ylabel('Vorhergesagte Länge (Predicted Length)', fontsize=12)
plt.title('Vergleich: Tatsächliche vs. vorhergesagte Länge', fontsize=14)
plt.legend()
plt.grid(True)

# Plot anzeigen
plt.show()
```