Lehrstuhl für Systemsimulation

Sommersemester 2025

Prof. Dr. Harald Köstler, Frederik Hennig, Michael Zikeli

Algorithmik kontinuierlicher Systeme
Aufgabenblatt 7 — Bézierkurven und Polynominterpolation

- Zu diesem Übungsblatt steht im StudOn-Kurs ein Zip-Verzeichnis mit Material bereit. Laden Sie dieses herunter und entpacken Sie es, bevor Sie mit den Aufgaben beginnen.
- Zu jeder Aufgabe gehört ein Python-Modul, dessen Name im Aufgabentitel gelistet ist. Füllen Sie die darin enthaltenen Funktions-Stubs mit Ihren Lösungen und geben Sie die Dateien über StudOn ab. Es handelt sich hierbei um **Einzelabgaben**. Sie können Ihre abgegebenen Dateien beliebig oft aktualisieren – nur die letzte abgegebene Version wird gewertet. **Laden Sie ihre Lösungen einzeln, oder als Zip-Archiv hoch, und ändern Sie nicht die Namen der Python-Dateien.**
- Das Material zum Übungsblatt enthält außerdem ein Jupyter Notebook zur interaktiven Entwicklung Ihrer Lösungen, sowie eine automatisierte Test-Suite. Jede Teilaufgabe wird anhand von einer Reihe an Tests automatisch bewertet. Die *öffentlichen* Testfälle können Sie entweder in dem mitgelieferten Notebook, oder durch das Skript `blatt**_tests.py` auf der Konsole ausführen. Denken Sie daran, dass das Bestehen der öffentlichen Tests keine Garantie für Korrektheit ist.

In diesem Blatt werden Sie Methoden kennenlernen, um aus einzelnen Punkten kontinuierliche Daten zu erzeugen. Bei Bézierkurven ist das Ziel, optisch ansprechende, glatte Kurven zu erzeugen. Bei der Polynominterpolation hingegen werden oszillationen in Kauf genommen, um dafür jede Stützstelle exakt zu treffen.

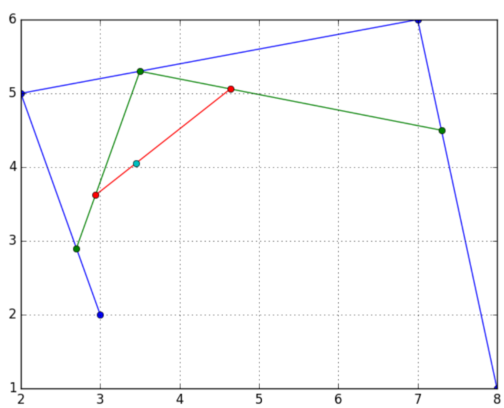

Abbildung 1: Berechnung eines einzelnen Punktes einer Bézierkurve

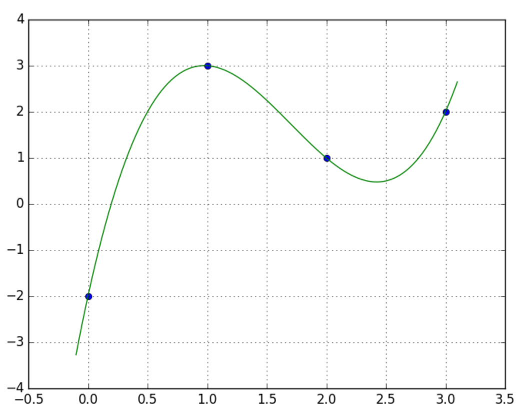

Abbildung 2: Polynominterpolation

Aufgabe 1 — Bézierkurven (12 Punkte) bezier.py

Bézierkurven erfreuen sich in Computergrafik und CAD sehr großer Beliebtheit. Sie sind optisch ansprechend und können einfach über Kontrollpolygone manipuliert werden. Außerdem existieren effiziente Algorithmen, um Bézierkurven am Computer zu approximieren. In dieser Aufgabe werden Sie einige dieser Algorithmen implementieren.

Die zentrale Datenstruktur dieser Aufgabe sind Kontrollpolygone aus Punkten $P_0, \ldots, P_n$. In allen folgenden Aufgaben werden Kontrollpolygone durch $n \times 2$ NumPy Arrays aus Gleitkommazahlen dargestellt, so dass der Index $[i, 0]$ die x-Koordinate des i-ten Punktes liefert und $[i, 1]$ die dazugehörige y-Koordinate.

a) De Casteljau Schritt (2 Punkte) Der Algorithmus von de Casteljau erlaubt es, Bézierkurven beliebig genau durch geeignete Polygonzüge zu approximieren. Ein zentraler Schritt des Algorithmus ist es, ein Kontrollpolygon mit n Punkten auf ein Polygon mit $n-1$ Punkten abzubilden, so dass für jeden Punkt des neuen Polygons P^{n-1} gilt

$$P_i^{n-1} = (1-t) \cdot P_i^n + t \cdot P_{i+1}^n. \tag{1}$$

Implementieren Sie nun die Funktion `de_casteljau_step`, die für ein gegebenes Kontrollpolygon P der Länge n und eine Gleitkommazahl t das zugehörige Polygon der Länge $n-1$ nach Formel 1 berechnet.
Hinweis: Das Jupyter Notebook zum Übungsblatt enthält ein Widget, mit dem Sie den de Casteljau-Schritt visualisieren können. Nutzen Sie dieses, um den Algorithmus interaktiv zu testen und mit ihm zu experimentieren.

b) De Casteljau (2 Punkte) Implementieren Sie nun die Funktion `de_casteljau`, die ein Kontrollpolygon P und eine Gleitkommazahl $t \in [0, 1]$ übergeben bekommt, und die über den Algorithmus von de Casteljau eine Bézierkurve an dem zu t gehörigen Punkt auswertet. Nutzen Sie Ihre Funktion aus Teilaufgabe a).

c) Bézierkurve approximieren (2 Punkte) Schreiben Sie nun die Funktion `bezier1`, die für ein gegebenes Kontrollpolygon P und eine Ganzzahl m die zu P gehörige Bézierkurve mit dem Algorithmus aus Teilaufgabe b) approximiert. Dazu soll das Interval $[0, 1]$ mit m Punkten uniform abgetastet werden.
Hinweis: Auch zu dieser Aufgabe steht im Begleit-Notebook ein Visualisierungs-Widget bereit.

d) Kontrollpunkte einfügen (2 Punkte) Bézierkurven erlauben es, den Grad der Kurve um eins zu erhöhen, ohne den Verlauf der Kurve zu verändern. Implementieren Sie dazu die Funktion `add_control_point`, die ein gegebenes Kontrollpolygon P der Länge n auf ein Kontrollpolygon Q der Länge $n+1$ abbildet. Die Punkte von Q ergeben sich dabei durch

$$Q_0 = P_0, \quad Q_n = P_{n-1}, \quad Q_i = \alpha P_{i-1} + (1-\alpha) P_i, \quad \text{mit } \alpha = \frac{i}{n}, \quad i \in 1, \ldots, n-1 \tag{2}$$

e) Bézierkurve aufteilen (2 Punkte) Der Algorithmus in Teilaufgabe c) ist ineffizient, weil die Hierarchie aus Polygonen für jeden einzelnen Punkt neu berechnet werden muss. Besser ist es, mit der Subdivisionsmethode Bézierkurven jeweils in der Mitte aufzuteilen und so rekursiv zu verfeinern.

Implementieren Sie dazu als ersten Schritt die Funktion `split_curve`, die das Kontrollpolygon P der Länge n einer Bézierkurve übergeben bekommt, und ein Tupel (L, R) mit zwei neuen Kontrollpolygonen L und R der Länge n zurück gibt, so dass L die linke Hälfte der ursprünglichen Kurve bis zum Punkt $t = \frac{1}{2}$ beschreibt und R die rechte Seite der ursprünglichen Kurve.
Hinweis: Die Punkte in L und R sind die Randpunkte der de Casteljau Pyramide.

f) Bézierkurven rekursiv approximieren (2 Punkte) Implementieren Sie nun die Funktion `bezier2`, die zu einem gegebenen Kontrollpolygon P und einer Rekursionstiefe *depth* einen Polygonzug erzeugt, der die durch das Kontrollpolygon beschriebene Bézierkurve approximiert. Bei einer Rekursionstiefe von 0 soll P unverändert zurückgegeben werden, ansonsten soll die Kurve mithilfe der Funktion aus Teilaufgabe e) aufgeteilt, die entstehenden Hälften rekursiv weiterbearbeitet und das Ergebnis wieder zusammengefügt werden.
Hinweis: Beim Zusammenfügen muss darauf geachtet werden, dass keine Punkte doppelt in das Polygon eingefügt werden.
Hinweis 2: Nutzen Sie auch hier das vorbereitete Notebook-Widget zur Visualisierung.

Aufgabe 2 — Polynominterpolation (8 Punkte) interpolation.py

In Python, bzw. der NumPy Bibliothek gibt es bereits fertige Klassen für das Handhaben von Polynomen. In den folgenden Aufgaben sollen alle Polynome als Objekte vom Typ `numpy.poly1d` repräsentiert werden. Die Dokumentation finden Sie unter `https://docs.scipy.org/doc/numpy/reference/generated/numpy.poly1d.html`.

a) Polynome in Python (2 Punkte) Als Aufwärmübung wollen wir eine simple Interpolationsgerade durch zwei Punkte legen. Implementieren Sie eine Funktion `interpolate_linearly`, die zwei Punkte, jeweils repräsentiert als Liste von zwei Gleitkommazahlen x und y, übergeben bekommt und das Polynom erster Ordnung (vom Typ `numpy.poly1d`) zurückliefert, das beide Punkte interpoliert.

b) Newton-Matrix (2 Punkte) Implementieren Sie nun die Funktion `newton_matrix`, welche ein NumPy Array mit den x-Koordinaten der Interpolationspunkte übergeben bekommt, und die Matrix auf der linken Seite des linearen Gleichungssystem aufstellt, das gelöst werden muss, um die Koeffizienten für die Newton-Basis zu bestimmen.

c) Newton-Polynom (2 Punkte) Implementieren Sie die Funktion `newton_polynomial`, die das NumPy Array aus den Koeffizienten der Newton-Basis und ein NumPy Array der x-Koordinaten übergeben bekommt, und die das zugehörige Polynom (vom Typ `numpy.poly1d`) generiert.

d) Newton-Interpolation (2 Punkte) Implementieren Sie die Funktion `interpolating_polynomial`, welche für zwei gegebene NumPy Arrays mit x- und y-Koordinaten der Interpolationspunkte und mit Hilfe der Funktionen aus den Teilaufgabe b) und c)

1. ein lineares Gleichungssystem aufstellt, um die Koeffizienten der Newton-Basis zu bestimmen,
2. das Gleichungssystem löst,
3. und aus der Lösung des Gleichungssystems (Koeffizienten) ein Polynom generiert.

Hinweis: In dieser Aufgabe ist es explizit erlaubt, Funktionen aus dem Modul `numpy.linalg` zu verwenden.
Hinweis 2: Das Notebook zum Aufgabenblatt enthält auch hier wieder Code, um das Ergebnispolynom zu plotten.