Lehrstuhl für Systemsimulation

Sommersemester 2025

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

Algorithmik kontinuierlicher Systeme
Aufgabenblatt 1 — Grundlagen: Python und NumPy

- 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. Bitte ändern Sie nicht die Namen der hochzuladenden 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.

Aufgabe 1 — Arithmetik Rationaler Zahlen (7 Punkte) `arithmetic.py`

In dieser ersten Aufgabe befassen wir uns mit der Implementierung einfacher Algorithmen im Kontext der Arithmetik ganzer und rationaler Zahlen. Lösen Sie diese Aufgabe ohne weitere Module zu importieren.

a) Sieb des Eratosthenes (1.5 Punkte) Der Sieb des Eratosthenes filtert die Primzahlen aus den natürlichen Zahlen heraus. Implementieren Sie eine Funktion `primes_sieve`, welche eine natürliche Zahl n erhält und eine Liste aller Primzahlen kleiner oder gleich n zurückgibt. Die Primzahlen sollen dabei in aufsteigender Reihenfolge zurückgegeben werden.

b) Primfaktorzerlegung (1.5 Punkte) Jede natürliche Zahl besitzt eine eindeutige Zerlegung als Produkt von Primzahlen; Implementieren Sie eine Funktion `factorize`, welche diese Primfaktorzerlegung berechnet. Ihre Funktion erhält eine natürliche Zahl n und soll eine Liste von Primzahlen zurückgeben, welche im Produkt n ergeben. Die Primfaktoren sollen dabei *aufsteigend* sortiert sein. Nutzen Sie Ihre Funktion `primes_sieve` aus Teilaufgabe a).

c) Euklidischer Algorithmus (1.5 Punkte) Der euklidische Algorithmus[1] ist ein effizientes Verfahren, um den *größten gemeinsamen Teiler* $\mathrm{ggT}(a, b)$ zweier natürlicher Zahlen a und b zu berechnen. Nutzen Sie diesen und schreiben Sie eine Funktion `gcd`, welche die beiden Zahlen a und b als Argumente erhält und ihren größten gemeinsamen Teiler zurückgibt.

d) Brüche Kürzen (1 Punkt) In dieser Teilaufgabe repräsentieren wir Brüche $\frac{a}{b}$ als Tupel `(a, b)` zweier ganzer Zahlen a und b ($b > 0$). Implementieren Sie eine Funktion `qreduce`, welche einen solchen Bruch als Argument erhält und seine vollständig gekürzte Version als Tupel zurückgibt. Nutzen Sie dazu Ihre Funktion `gcd` aus der vorherigen Teilaufgabe,

[1] `https://de.wikipedia.org/wiki/Euklidischer_Algorithmus`

e) Rationale Zahlen Addieren (1.5 Punkte) Wir machen weiter mit der Addition rationaler Zahlen, wobei wir rationale Zahlen (i.e. Brüche) $\frac{a}{b}$ nach wie vor als Tupel `(a, b)` darstellen. Implementieren Sie eine Funktion `qadd`, welche zwei rationale Zahlen p und q als Tupel erhält und ihre Summe, wieder als Tupel, zurückgibt. Das Ergebnis soll dabei vollständig gekürzt sein.

Aufgabe 2 — Matrizen konstruieren mit NumPy (7 Punkte) `matrices.py`

In dieser Aufgabe wollen wir die Bibliothek NumPy nutzen, um eine Reihe spezieller Matrizen zu konstruieren. Werfen Sie während der Bearbeitung der Aufgaben regelmäßig einen Blick in die NumPy-Dokumentationsseiten `https://numpy.org/doc/stable/user/basics.creation.html` und `https://numpy.org/doc/stable/reference/routines.html`. Für viele Probleme bietet NumPy bereits eine Lösung, und mit einer kurzen Suche in der Dokumentation können Sie sich häufig viel Programmieraufwand ersparen.

a) Spiegelungsmatrix (1 Punkt) Schreiben Sie eine Funktion `reflection2d`, die zu einem gegebenen Winkel ω eine 2×2 Spiegelungsmatrix S zurückgibt, so dass $S \cdot x$ einen zweidimensionalen Vektor x an der Gerade durch den Ursprung mit Winkel ω zur Ordinate spiegelt.

b) Einheitsmatrix — Rechteckig (1 Punkt) Schreiben Sie eine Funktion `eye`, welche zwei Parameter n und m erwartet und eine $n \times m$ Matrix mit 1en auf der Diagonale zurückgibt.

c) Komposition (1 Punkt) Schreiben Sie eine Funktion `compose`, die beliebig viele Matrizen $A_1, \ldots, A_n$ mit den Dimensionen $d_1 \times d_2,\ d_2 \times d_3,\ \ldots,\ d_n \times d_{n+1}$ übergeben bekommt und diese zu einer einzelnen Matrix B mit der Dimension $d_1 \times d_{n+1}$ zusammenfügt, so dass für alle d_{n+1}-dimensionalen Vektoren x gilt $A_1 \cdot A_2 \cdot \ \ldots\ \cdot A_n\ x = Bx$.

d) Antidiagonalmatrix (1 Punkt) Eine Matrix $A \in \mathbb{R}^{n \times n}$ heißt *antidiagonal*, wenn für ihre Einträge gilt:

$$A_{ij} = 0 \quad \text{für } i + j \neq n + 1$$

Damit wird eine antidiagonale $n \times n$-Matrix durch genau n Einträge bestimmt. Schreiben Sie eine Funktion `antidiag`, die aus einer Liste an d Werten eine antidiagonale $d \times d$-Matrix mit diesen Werten erstellt.

e) Kronecker-Matrix-Produkt (1.5 Punkte) Sei $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{p \times q}$, so bezeichnet das Kronecker-Matrix-Produkt $A \otimes B$ die folgende $pm \times qn$-Blockmatrix:

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

Schreiben Sie eine Funktion `kronecker_product`, welche zu zwei Matrizen A und B ihr Kronecker-Matrix-Produkt $A \otimes B$ zurückgibt.

Hinweis: Die NumPy-Funktion `np.kron` dürfen Sie in dieser Aufgabe nicht verwenden.

f) Walsh Matrix (1.5 Punkte) Schreiben Sie eine Funktion `walsh`, die zu einem gegebenen $n \in \mathbb{N}$ eine $2^n \times 2^n$-Walsh-Matrix W_n zurückgibt, welche nach folgendem Schema aufgebaut sind:

$$W_0 = \begin{pmatrix} 1 \end{pmatrix} \qquad \text{und} \qquad W_n = \begin{pmatrix} W_{n-1} & W_{n-1} \\ W_{n-1} & -W_{n-1} \end{pmatrix}$$

Aufgabe 3 — Game of Life (6 Punkte) gameoflife.py

Ziel dieser Aufgabe ist es, Conway's Game of Life mithilfe von NumPy-Arrays zu implementieren. Dabei handelt es sich um einen einfachen Zellulären Automaten mit zwei Zuständen pro Zelle. Trotz seiner Einfachheit erzeugt der Automat erstaunliche und kaum vorhersehbare Effekte. Eine detaillierte Beschreibung des Game of Life finden Sie hier: `https://en.wikipedia.org/wiki/Conway's_Game_of_Life`. Um Ihre Lösung interaktiv zu testen ist im Notebook `Blatt01-Scratchpad` eine interaktive Animation vorbereitet.

a) Initialisierung (2 Punkte) Das Gitter auf dem das Game of Life ausgeführt wird, stellen wir als NumPy Arrays aus Nullen und Einsen dar (Verwenden sie als Elementtyp `dtype=bool`).
Implementieren Sie nun die Funktion `add_entity`, die ein Objekt (Numpy Array aus Nullen und Einsen) an die spezifizierte Stelle des Gitters schreibt, sodass der Wert des Objekts an der Stelle $[0, 0]$ auf der Gitterkoordinate $[x, y]$ liegt. Sie dürfen dabei annehmen, dass an der spezifizierten Stelle im Gitter genug Platz für das Objekt ist.

b) Zeitschritt (4 Punkte) Nun geht es an die eigentlichen Spielregeln. Implementieren Sie dazu die Funktion `next_step`, die für ein gegebenes Gitter das zugehörige Gitter des nächsten Zuges berechnet. Dafür gelten folgende Regeln:

1. Eine tote Zelle mit genau drei lebendigen Nachbarn erwacht zum Leben (d.h., wird auf den Wert 1 gesetzt).
2. Eine lebendige Zelle mit weniger als zwei lebendigen Nachbarn stirbt (d.h., wird auf den Wert 0 gesetzt).
3. Eine lebendige Zelle mit zwei oder drei lebendigen Nachbarn lebt weiter.
4. Eine lebendige Zelle mit mehr als drei lebenden Nachbarn stirbt.

Die Randbedingungen seien dabei periodisch, d.h., eine Zelle am oberen Rand hat als oberen Nachbarn die zugehörige Zelle am unteren Rand. Die anderen drei Randfälle seien analog. Wenn Sie dies richtig implementieren, sollte ein Gleiter in der Lage sein über Ränder zu fliegen.

Hinweis: Passen Sie in Python mit Bool'schen Werten auf: während im klassischen Python `True + True == 2` gilt, verhalten sich NumPy-Booleans etwas strikter: Im Kontext von NumPy gilt `True + True == True`. Eventuell müssen Sie daher Booleans zu `int` konvertieren, um sie zu addieren.