

Algorithmik kontinuierlicher Systeme Aufgabenblatt 9 — Zelluläre Automaten und die Lattice-Boltzmann Methode

Allgemeines:

- Die Abgabe der Programmieraufgaben erfolgt über das Exercise Submission Tool:
<https://est.cs.fau.de>
- Sie können während der Bearbeitungszeit Ihre Abgaben im EST beliebig oft aktualisieren. Nur die aktuellste Abgabe, die in der Bearbeitungszeit hochgeladen wurde, wird gewertet.
- Auf der Übungswebsite finden Sie zu jedem Übungsblatt eine Vorlage, sowie zu jeder Aufgabe eine Datei `*_test.py` mit der Sie ihre Lösungen jederzeit selbst überprüfen können.
- Damit Sie auf eine Teilaufgabe Punkte bekommen, muss sie mit Python 3.5 im CIP Pool funktionieren. Das bestehen der mitgelieferten Tests ist notwendig, aber nicht hinreichend dafür, dass Sie Punkte bekommen.

Aufgabe 1 — Game of Life (4 Punkte)

gameoflife.py

Ziel dieser Aufgabe ist es, Conway's Game of Life 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.

a) Initialisierung 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 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.

Aufgabe 2 — Die Lattice-Boltzmann Methode (8 Punkte)

lbm.py

In dieser Aufgabe implementieren Sie einen 2D Windkanal mit der Lattice-Boltzmann-Methode. Abgesehen von einigen Grundlagen der Strömungsmechanik lernen Sie in dieser Aufgabe auch, wie man effiziente Programme mit Numpy schreibt. Am wichtigsten ist dazu, dass Sie niemals mit Python Code auf einzelnen Gleitkommazahlen arbeiten, sondern stets auf ganzen Numpy-Arrays. Zum Beispiel ist der folgende Code-schnipsel

```
for i in range(100):  
    a[i] = 3 * b[i]
```

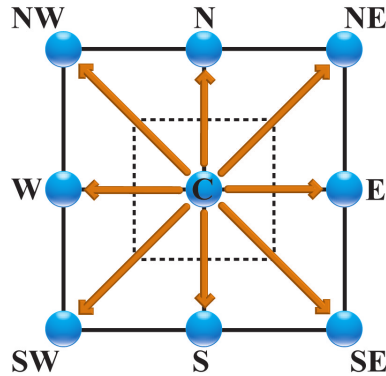
in Python deutlich langsamer als

```
a[:] = 3 * b[:]
```

da im letzteren Fall nur drei Python Befehle ausgeführt werden (Array-Referenz, Multiplikation, Zuweisung), im ersten Fall aber mehrere hundert. Die einfache Faustregel für schnellen NumPy Code lautet daher:

Wenn möglich, nie mit `for` über lange Arrays iterieren, sondern stets mit Referenzen und Slices arbeiten.

Hinweis: Dies ist auch ein guter Moment, sich noch einmal die zugehörige Numpy Dokumentation auf der Website <https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html> anzuschauen.



a) Berechnung der Dichte Als erstes implementieren Sie die Funktion `density`, welche aus einem gegebenen $9 \times W \times H$ Array aus Verteilungsfunktionen das $W \times H$ Array der Dichte jeder Zelle berechnet. Die Dichte ist definiert als

$$\rho = \sum_{q \in \{NW, N, \dots, SE\}} f_q \cdot$$

b) Berechnung der Geschwindigkeit Nun implementieren Sie die Funktion `velocity`, welche aus einem gegebenen $9 \times W \times H$ Array aus Verteilungsfunktionen die Geschwindigkeit in x -Richtung und in y -Richtung berechnet (jeweils als $W \times H$ Array). Die Geschwindigkeiten sind definiert als

$$u_x = \frac{1}{\rho} \sum_{q \in \{NW, N, \dots, SE\}} f_q e_{q,x}$$
$$u_y = \frac{1}{\rho} \sum_{q \in \{NW, N, \dots, SE\}} f_q e_{q,y}.$$

Hinweis: Die Komponenten des Richtungsvektoren ($e_{q,x}, e_{q,y}$) können Sie der globalen Variable `directions` entnehmen.

c) Berechnung der Gleichgewichtsverteilung In der Lattice-Boltzmann-Methode interessieren uns nicht nur die momentanen Verteilungsfunktionen f_q in einer Zelle, sondern auch die zugehörige Gleichgewichtsfunktionen f_q^{eq} . Die neun Einträge der Gleichgewichtsverteilung ergeben sich nach folgender Formel:

$$f_q^{eq} = w_q \rho \left(1 + 3(e_{qx}u_x + e_{qy}u_y) + \frac{9}{2}(e_{qx}u_x + e_{qy}u_y)^2 - \frac{3}{2}(u_xu_x + u_yu_y) \right)$$

Die verwendeten Variablen ergeben sich folgendermaßen:

w_q	Der q -te Eintrag der globalen Variable weights .
ρ	Die Dichte der jeweiligen Zelle (\rightarrow density(f)).
$e_{q,x}$	Der Eintrag directions[q,0] .
$e_{q,y}$	Der Eintrag directions[q,1] .
u_x, u_y	Die Geschwindigkeiten der jeweiligen Zelle (\rightarrow velocity(f))

d) Collide Sobald Sie die Gleichgewichtsverteilung berechnet haben, ist die Auswertung des Kollisionsoperators nicht mehr schwer. Hierbei wird einfach anhand folgender Formel

$$f_q = f_q - \omega (f_q - f_q^{eq})$$

jeder Eintrag f_q linear mit einem gegebenen Faktor ω gegen das Gleichgewicht verschoben. Implementieren Sie die zugehörige Funktion **collide**.

e) Stream Nun fehlt nur noch ein einziger Schritt (abgesehen von den Randbedingungen) zum fertigen Windkanal — der Stream-Operator. Implementieren Sie nun also die Funktion **stream**, die für alle *inneren* Punkte des Simulationsgebietes die Einträge mit denen der zugehörigen Nachbarzellen überschreibt.

Hinweis: Sie benötigen dazu eine Kopie der Verteilungsfunktionen, um nicht aus Versehen alte und neue Einträge zu vermischen.

f) Hindernisse Ein Vorteil der Lattice-Boltzmann-Methode ist es, dass man relativ einfach komplexe Geometrien simulieren kann. Dazu werden die Hindernisse zunächst wie normale Fluidzellen behandelt. Vor dem Streaming-Schritt werden dann aber alle Hindernisse mit den gespiegelten Einträgen ihrer jeweiligen Nachbarzellen initialisiert (No-slip Randbedingungen), sodass effektiv die Einträge von randnahen Fluidzellen gespiegelt werden. Implementieren sie die Funktion **noslip**, die diese Initialisierung für alle Zellen vornimmt, deren Koordinaten als $N \times 2$ Array **masklist** übergeben werden.

Dies ist die letzte Aufgabe dieses Sommersemesters AlgoKS. Sie bekommen für diese Aufgabe keine Punkte. Aber sobald Sie diese Aufgabe fertiggestellt haben, sollten Sie beim Ausführen der Datei **lbm.py** eine bewegte Strömungssimulation angezeigt bekommen. Sie sind eingeladen, mit dieser Simulation weiter zu experimentieren.

Experimentiervorschläge:

- Variieren Sie den Parameter **omega** zwischen 0 und 2, um mehr oder weniger turbulentes Verhalten zu simulieren. Bei Werten nahe 2 wird die Simulation spektakulärer, die Stabilität der Simulation kann aber darunter leiden.
- Passen Sie die Einträge in der Variable **flow** an, um am linken Rand mehr oder weniger Fluid einströmen zu lassen.
- Passen Sie die Form des Hindernisses in der Funktion **lbm** an.
- Variieren Sie die Größe des Gitters.