Dokumentation Projektarbeit
Simulation Boids
Gruppe 04

Levin Axmann
Stanislav Wolf
Lars Hülsken

Semester	Sommersemester 2025
Modul	Fachpraktikum Scientifc Programming
Erstellungsdatum	19. September 2025

19. September 2025

Inhaltsverzeichnis

1 Abstract

Das Projekt des Fachpraktikums hat zum Ziel, eine eigene Implementierung einer Simulation von Schwarmverhalten zu analysieren und optimieren. Dazu werden verschiedene Ansätze ausprobiert und hinsichtlich Laufzeit verglichen. Es hat sich gezeigt, dass der quadratische Zusammenhang durch den Einsatz einer geometrischen Datenstruktur unterboten werden kann, was für große Eingabemengen sinnvoll ist. Für kleinere Eingabemengen hat sich das reine Einhalten von Grundregeln des wissenschaftlichen Programmierens ohne den Aufbau einer separaten Datenstruktur als performanter herausgestellt. Für eine sinnvolle Entscheidung wird vor Ablauf der Simulation daher ein Probelauf durchgeführt.

2 Einleitung

Im Jahr 1987 hat Craig Reynolds eine Simulation entwickelt, die das Bewegungsverhalten von Vögeln in einem Vogelschwarm am Himmel nachbilden sollte. In seinem Programm nannte Reynolds die Vögel "Boids"[2] Die Simulation basiert auf drei Verhaltensregeln für jeden einzelnen Boid. Damit baut die Simulation auf dem Prinzip der Emergenz auf, sodass sich aus diesen Regeln je Boid für die Gesamtgruppe ein komplexes Verhalten (eine Gruppendynamik) ergibt. Die Visualisierung einer solchen Simulation zeigt entsprechende Ähnlichkeiten zu ärmen.

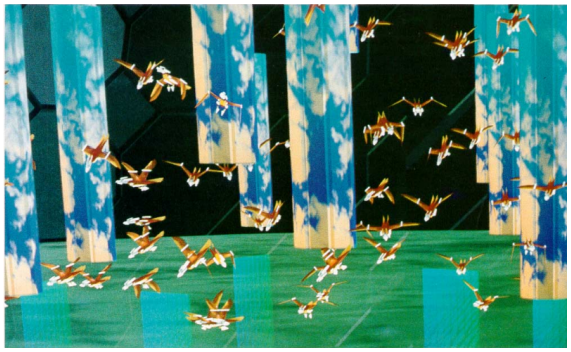


Abbildung 2.1: Darstellung eines Vogelschwarms aus dem Originalpaper [2] von Reynolds

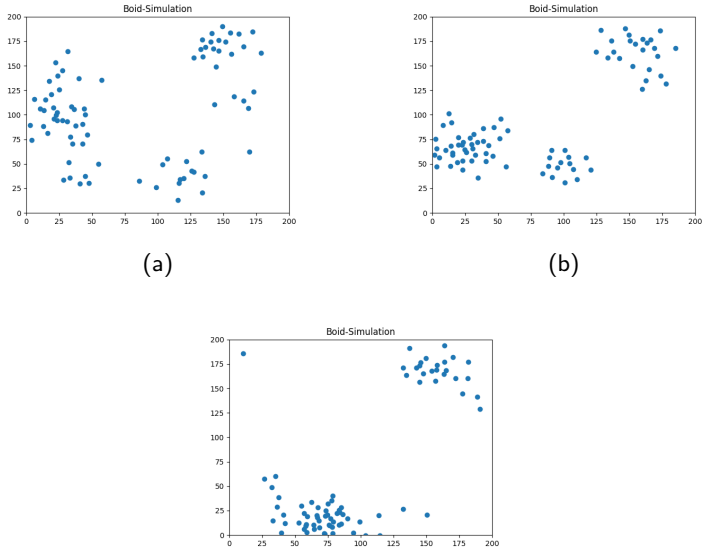
Dieses Dokument soll unser Vorgehen während des Fachpraktikums erläutern und begründen. Es erhebt keinen Anspruch darauf, den Code vollständig zu dokumentieren. Zu diesem Zwecke haben wir den Sourcecode mit dem Dokumentationstool `pdoc` dokumentiert. Die Dokumente sind der Abgabe beigefügt (in dem Ordner `docs`).

3 Grundmodell

3.1 Allgemeine Funktionsweise

Das Modell nach Reynolds funktioniert iterativ: Jeder Zeitschritt in der Simulation basiert lediglich auf seinem vorherigen Zeitschritt (außer dem ersten Schritt). Die Positionen der Boids im aktuellen Zeitschritt ergeben sich aus der Kombination der Position und Geschwindigkeit der Boids im letzten Zeitschritt.

Es existiert eine Anzahl *Flock_Size* von Boids. Diese bewegen sich in einem n-dimensionalen Raum. In unserem Fall bei $n = 2$ hat der Raum die Größe $x_{max} \cdot y_{max}$ und jeder Boid wird durch ein Paar von kartesischen Koordinaten (x, y) für die Position, sowie durch ein Paar (v_x, v_y) für die Geschwindigkeit zu jedem Zeitpunkt t, mit $t \in \mathbb{N}$, sowie $t <= STEPS$ beschrieben. Aus der Addition der Geschwindigkeitsanteile eines Boids kann außerdem eine "Blickrichtung"bestimmt werden.



(a) (b)

(c)

Abbildung 3.1: Abhängig von der Wahl der Parameter, wie in 3.3 beschrieben, bilden sich im Zeitverlauf Schwärme von Boids

Pro Zeitschritt und Boid werden, basierend auf Position und Geschwindigkeit *benachbarter* Boids, die folgenden Regeln angewandt. Die Definition und Ermittlung benachbarter Boids wird später für die Optimierung eine wichtige Rolle spielen.

3.2 Regeln

3.2.1 Separation

Idee

Jeder Boid möchte einen vernünftigen Abstand zu Nachbarn einhalten, um Kollisionen und Enge zu vermeiden.

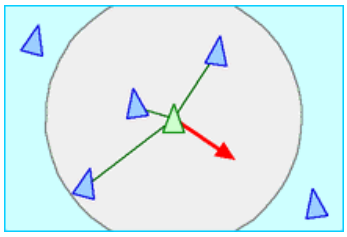


Abbildung 3.2: Grafische Darstellung der Separation-Regeln nach [1]

Umsetzung im Modell

Pro Boid i und Zeitschritt t werden pro Dimension die Positionsdifferenzen zu allen benachbarten Boids j aufaddiert.

$$acc_{x,i,t} = \sum_{j} (x_{i,t} - x_{j,t})$$

Anschließend die Geschwindkeit um diesen Summanden acc_x (multipliziert mit einem Gewichtsfaktor avoidance) verändert.

$$v_{x,i,t} = v_{x,i,t-1} + (acc_{x,i,t} \cdot \mathit{avoidance})$$

3.2.2 Alignment

Idee

Jeder Boid möchte sich an die Geschwindigkeit seiner Nachbarn anpassen.

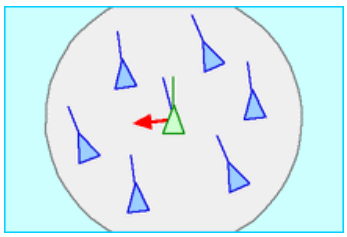


Abbildung 3.3: Grafische Darstellung der Alignment-Regeln nach [1]

Umsetzung im Modell

Pro Boid i und Zeitschritt t wird pro Dimension die durchschnittliche Geschwindkeit aller benachbarten Boids $j$ermittelt.

$$v_{avg,x,i,t} = \frac{\sum_j v_{x,j,t-1}}{n_{neigbours}}$$

Die Geschwindigkeit wird um die Differenz zur durchschittlichen Geschwindigkeit (multipliziert mit einem Gewichtsfaktor matching) angepasst.

$$v_{x,i,t} = v_{x,i,t-1} + (v_{x,i,t-1} - v_{avg,x,i,t}) \cdot \textit{matching}$$

3.2.3 Cohesion

Idee

Jeder Boid möchte seine Position an die der Nachbarn anpassen und sich zentral ausrichten.

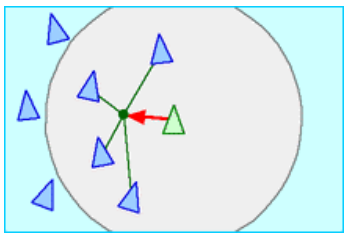


Abbildung 3.4: Grafische Darstellung der Cohesion-Regeln nach [1]

Umsetzung im Modell

Pro Boid i und Zeitschritt t wird pro Dimension die durchschnittliche Position aller benachbarten Boids j ermittelt.

$$x_{avg,i,t} = \frac{\sum_j x_{j,t}}{n_{neigbours}}$$

Die Geschwindigkeit wird um die Differenz der Position zur Durschnittsposition (multipliziert mit einem Gewichtsfaktor centering) angepasst.

$$v_{x,i,t} = x_{i,t-1} + (x_{i,t-1} - x_{avg,i,t}) \cdot \textit{centering}$$

3.3 Konfigurationsmöglichkeien

Neben den grundlegenden Paramtern $\{Flock\_Size, Steps, x_{max}, y_{max}\}$ kann jede der Regeln mit einem Faktor gewichtet werden. Somit erweitern sich die Konfigurationsmöglichkeiten um $\{avoidance, matching, centering\}$.
Da das Bewegungsverhalten der Boids im wesentlichen durch die Nachbarn bestimmt wird, muss definiert werden, wann zwei Boids benachbart sind. Dafür wird für jeden Boids eine Suchdistanz festgelegt, innerhalb derer dieser andere Boids auffinden . Zwei Boids sind benachbart, wenn die Distanz zwischen ihnen kleiner als ihre Suchdistanz ist. Diese kann pro Regel festgelegt werden. Somit erweitern sich die Konfigurationsmöglichkeiten um $\{separation\_distance, alignment\_distance, cohesion\_distance\}$.
Bei der Implementierung des Modells muss der Schwarm in dem begrenzten Bereich $0 < x < x_{max} 0 < y < y_{max}$ gehalten werden. Um diese Begrenzung umzusetzen gibt es unterschiedliche Möglichkeiten, beispielsweise:

- Periodische Randbedingungen, bei denen Boids am einen Ende des Simulationsgebietes verschwinden und am anderen Ende wieder auftauchen
- Abprallende Randbedingungen, bei denen Boids von den Rändern abprallen, wenn sie ihnen zu nahe kommen
- Ergänzung der Verhaltensregeln um eine Anziehungskraft zum Zentrum des Simulationsgebiets

In unserer Simulation haben wir abprallende Randbedingungen festgelegt.
Außerdem existieren globale Limits für die maximal wirkenden Kräfte und Geschwindigkeiten. Zusammengefasst bestehen die Konfigurationsmöglichkeiten somit aus den folgenden Parametern

$\{Flock\_Size, Steps, x_{max}, y_{max}, avoidance, matching, centering,$
$separation\_distance, alignment\_distance, cohesion\_distance\, max\_speed, max\_force\}$

4 Simulation

Bevor einzelne Verbesserungen im Kapitel Optimierung diskutiert werden, soll hier zunächst ein Gesamtüberblick über die verschiedenen Versionen gegeben werden, die im Laufe des Fachpraktikums erstellt wurden.
Diese wurden jeweils auf die Performance analysiert und anschließend zu einer neuen Version verbessert. Die Abbildung stellt den Zusammenhang der verschiedenen Versionen dar.

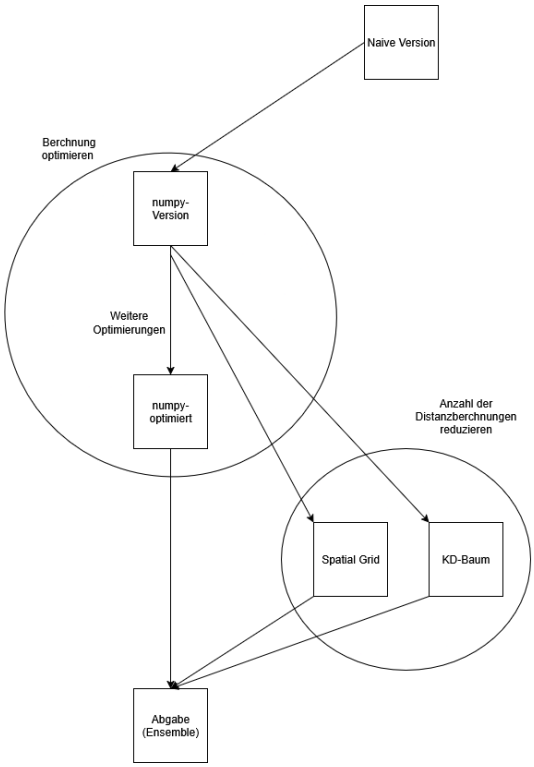


Abbildung 4.1: Zusammenhang der Versionen

Die Versionen im linken Kreis verfolgen grundsätzlich den Ansatz, die Ausführungszeit der Distanzberechnungen zu verringern (z.B. durch Reduzieren der Python-Time) Die Versionen im rechten Kreis verfolgen den Ansatz, die Anzahl der Distanzberechnungen zwischen Boids selbst zu reduzieren.
Da sich im Laufe des Praktikums herausgestellt hat, dass für unterschiedliche Flock_Sizes verschiedenen Ansätze vorteilhaft sind, haben wir uns in der finalen Version für ein Ensemble-Verfahren entschieden, das vor Ausführung der Simulation einen Probelauf zur Auswahl der Version durchführt. Dies ist in Kapitel 5.6
Detaillietere Erläuterungen dazu sind in der Tabelle zu finden.

Variante	Eigenschaft (wesentliche Verbesserungsideen)
Naiv	Objektorientiert in Python Fokus auf Verständlichkeit und Korrektheit statt auf Performance Basis für Vergleich (Korrektheit und Performance)
Numpy optimiert	Array-Operationen statt Schleifen, Offsets und Distanzen nur einmal berechnen weniger Funktionsaufrufe
Spatial Grid	Anzahl der Distanzberechnungen reduzieren durch fixe Aufteilung des Raumes
KD-Baum	Anzahl der Distanzberechnungen reduzieren durch geometrische Datenstruktur
Abgabe	Ensemble-Verfahren

Tabelle 4.1: Während des Fachpraktikums erstellte Varianten mit den wesentlichen Eigenschaften

4.1 Weitere Erläuterungen zum Aufbau

4.2 Erklärungen zur Visualisierung

Die Visualisierung wurde mit `matplotlib` erstellt. Um die Anzahl der Abhängigkeiten gering zu halten und die Ausführung im Container zu ermöglichen, haben wir uns für eine Ausgabe als GIF-Datei entschieden.

4.3 Einrichtung des Projektes

Variante 1: Mithilfe des beigelegten Shell-Skripts

Bei dieser Variante müssen lediglich die Projektdateien geladen werden, um daraufhin das Shell-Skript zu starten. Das Shell-Skript ist darauf ausgerichtet, alte Images oder Container dieses Projekts zu löschen, um diese dann neu zu erstellen. Das dient dazu, Änderungen im Dockerfile schnell wirksam zu machen.
Falls dieses Verhalten nicht gewünscht ist, können die folgenden Zeilen aus dem Shell-Skript entfernt werden:

```
1    docker rm -f boids_simulation
2    docker rmi boids_simulation
```

Variante 2: Manuelle Einrichtung des Docker-Containers

Zuerst bauen wir das Image anhand des Dockerfiles auf:

```
1    docker build -t boids_simulation .
```

Daraufhin erstellen wir einen zugehörigen Container und lassen diesen laufen:

```
1    docker run --rm --name boids_simulation -p 9999:9999 --mount type=bind
     ,source="$(pwd)",target=/home/jovyan -it boids_simulation jupyter lab
     --port=9999 --notebook-dir=/home/jovyan/
```

Der Link zur Jupyter-Lab-Umgebung sollte daraufhin in der Konsole erscheinen.

Starten der Simulation

In der Jupyter-Lab-Umgebung muss nun ein neues Terminal geöffnet werden. In diesem Terminal kann dann der folgende Befehl ausgeführt werden:

```
1    python3 main.py
```

Daraufhin wird ein Schwarm von Boids simuliert. Das Ergebnis der Simulation ist eine .gif-Datei, die im Ordner output abgelegt wird. Anpassung der Simulationsparameter Falls Verhaltens-Parameter für den Schwarm angepasst werden sollen (bspw. die Gewichtung der Verhaltensregeln), können dafür Konstanten in der config.py-Datei angepasst werden. Die Simulation kann nach Änderungen der Parameter in der config.py-Datei wieder ganz normal über den Befehl python3 main.py gestartet werden.

5 Analyse und Optimierung

5.1 Grundsätzliches zur Analyse

Alle Messungen wurden zwecks Vergleichbarkeit grundsätzlich auf identischer Hardware durchgeführt. Außerdem wurden - wenn zeitlich möglich - mehrfache Läufe durchgeführt, die Messwerte sind daher im Diagramm mit der Standardabweichung dargestellt und die Anzahl n an Wiederholungen angegeben.
Bei Wiederholungen mit gleichen Anfangsbedingungen ist die Standardabweichung in der Regel so gering, dass sie im Diagramm kaum sichtbar ist. Durch die mehrfache Ausführung mit zufälliger Wahl von Parametern soll der Einfluss von Anfangsbedingungen sowie Simulationsparametern begrenzt werden. Gerade bei den Verfahren zur Reduzierung von Abstandsberechnungen muss sonst der Einfluss von schneller Schwarmbildung diskutiert werden.

5.2 Allgemeine Verbesserungen mit numpy

Wie erwartet haben erste Analysen an der naiven Variante in objektorientiertem Python mittels scalene bestätigt, dass der Großteil der Ausführungszeit auf die Distanzberechnung zwischen den Boids entfällt. Somit ist hier das größte Optimierungspotential.

FLOCK_SIZE	Anteil Distanzberechnung in %	Anteil Python-Zeit in %
100	50	95
200	55	97
300	57	98

Tabelle 5.1: 200 Zeitschritte in der naiven Variante, $n = 5$

Außerdem ist der Anteil an Python-Zeit extrem hoch und optimierungsfähig. Das resultiert in folgenden Gesamtlaufzeiten, bei denen man klar die quadratische Laufzeit erkennt.
Das erste Ziel war, die beiden Schleifen zu ersetzen.

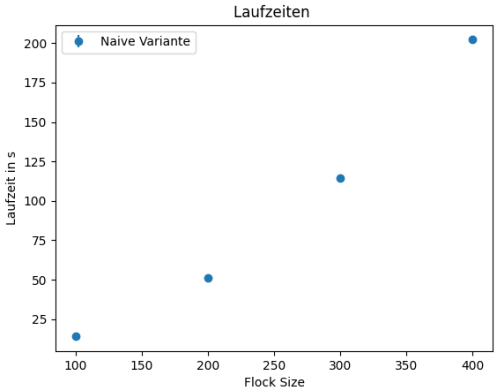


Abbildung 5.1: Laufzeiten der naiven Variante bei 200 Zeitschritten

5.2.1 Schleifen ersetzten

5.2.1.1 Grundsätzliche Idee

Grundsätzlich bestand die Programmausführung pro Zeitschritt aus 2 geschachtelten Schleifen. Da innerhalb verschiedene Operationen zu den Boids ausgeführt werden, haben wir - vor Versuchen, diese Variante mit `numba` zu optimieren - den Code insgesamt (z.B. auch von den Datenstrukturen her) so organisiert, dass wir die beiden inneren Schleifen vermeiden können.

5.2.1.2 Implementierung

```
1    for i in range(FLOCK_SIZE):
2        ...
3        for j in range(FLOCK_SIZE):
4            ...
```

Als erstes wurde die innere Schleife abgeschafft (Variante v2), so dass pro betrachtetem Boid aus der äußeren Schleife die benötigten Operationen zu den anderen Boids in vektorisierter Form auf einem Array durchgeführt werden können.
Pro Boid gibt es einen 1d-array mit den Distanzen zu allen anderen Boids. Dieses Array wird auf die Distanz gefiltert, aus den Werten das Offset berechnet und zu einem Wert akkumuliert. Dabei werden auch zuvor doppelte Operationen wie das Berechnen von Offsets in x und y Positionen vermieden, in dem Werte weiterverwendet werden könnnen, statt diese neu zu berechnen.
Dadurch wird in der Variante die separation-Regel als letztes angewandt, da dort die Adjanzentmatrix schreibend verändert wird. Das Tauschen ist laut Modell möglich, da immer nur die Positions- und Geschwindgkeitswerte aus dem vorherigen Schritt für die Berechnung notwendung sind, so dass wir in der Implementierung die für die Optimierung nützlichste Reihenfolge wähle können.
Anschließend ist es gelungen, alle Berechnungen in Operationen auf einem 2d-Array zu überführen, wodurch auch die äußere Schleife abgelöst werden konnte (Variante v3).

5.2.1.3 Analyse

Durch die Anpassungen konnte die Python-Zeit bereits bei kurzen Simulationen (100 Boids, 200 Zeitschritte, $n = 10$) von ursprünglich $(75.6 \pm 0.4)\%$ auf (10.3 ± 0.2) % verringert werden. Diese Wachsen bei größeren Flock_Sizes enorm.

FLOCK_SIZE	Anteil Python-Zeit
100	(10.3 ± 0.2) %
500	(4.9 ± 0.1) %
1000	(3.2 ± 0.1) %

Tabelle 5.2: 200 Zeitschritte, $n = 10$

Die Laufzeitunterschiede sind abhängig von der Anzahl Boids in der Abbildung dargestellt.

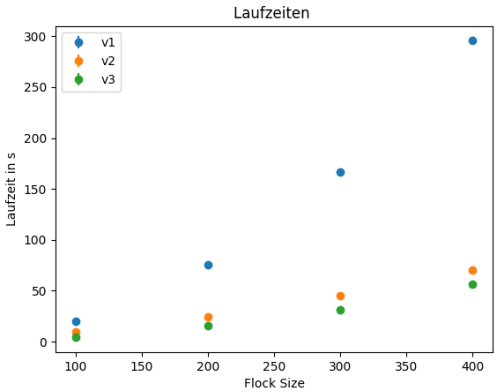


Abbildung 5.2: Laufzeiten in Abhängigkeit von der Flock_Size, Messung bei 200 Zeitschritten, $n = 2$

Es zeigt sich, dass die Laufzeit deutlich verringert werden konnte, während grundsätzlich der quadratische Zusammenhang bestehen bleibt. Deutlicher wird dies bei der Variante v3 für größere $FLOCK\_SIZEs$.

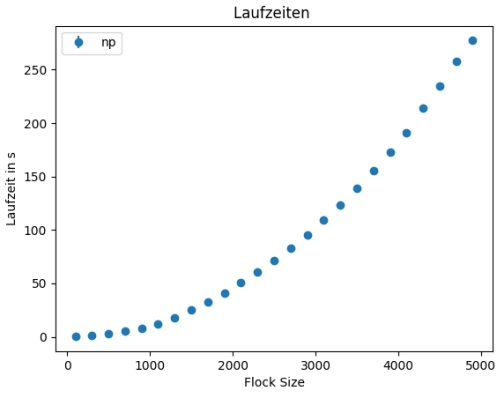


Abbildung 5.3: Laufzeiten in Abhängigkeit von der Flock_Size, Messung bei 50 Zeitschritten, $n = 5$

5.3 Spatial Grid

5.3.1 Grundsätzliche Idee

Bei der Simulation besteht das Problem, dass für die Geschwindigkeitsberechnungen pro Boid die Distanzen zu allen anderen Boids berechnet werden müssen. Für n Boids müssen wir also die Distanz zu n anderen Boids berechnen. Dafür ergibt sich eine Laufzeit von $O(n^2)$. Das Spatial Grid setzt hier an, indem es versucht die Anzahl der nötigen Distanzberechnungen zu verringern.
Beim Spatial Grid verfolgen wir die Idee, das Simulationsgebiet in viele quadratische Zellen aufzuteilen. Dadurch ergibt sich ein Gitter, das wir über das Feld legen (daher auch der Name „Spatial Grid“). Die Kantenlänge der Quadrate leiten wir aus der Sichtweite der Boids ab. In jedem Zeitschritt werden dann alle Boids jeweils einer Zelle zugeordnet. Wenn wir jetzt pro Boid die tatsächlichen Nachbarn finden wollen, müssen wir lediglich die Zelle bestimmen, in welcher sich der Boid befindet und die Distanzen zu den Boids in den acht benachbarten Zellen und der eigenen Zelle berechnen. Der entsprechende Vorgang ist in Abbildung 5.4 zu erkennen. Dadurch ist eine wesentliche Vorfilterung der Nachbarn möglich, welche die Bestimmung der Nachbarn deutlich effizienter machen kann.
In unterschiedlichsten Simulationen kann über diese Methode häufig die Nachbarschaftsbestimmung auf eine Komplexität von $O(n)$ runtergebrochen werden, da oft der Platz innerhalb einer Zelle begrenzt ist und so eingegrenzt werden kann, wie viele Nachbarn sich innerhalb der eigenen und den angrenzenden Zellen befinden können.

5.3.2 Implementierung

In der Implementierung werden vier zentrale Variablen benutzt:
`position_array` speichert die Positionen der Boids.
`speed_array` enthält die Geschwindigkeiten der Boids.
`cell_position_array` ordnet den Boids anhand ihrer Positionen Zellen in Form einer Zellkoordinate zu (also Zeile und Spalte im Gitter).

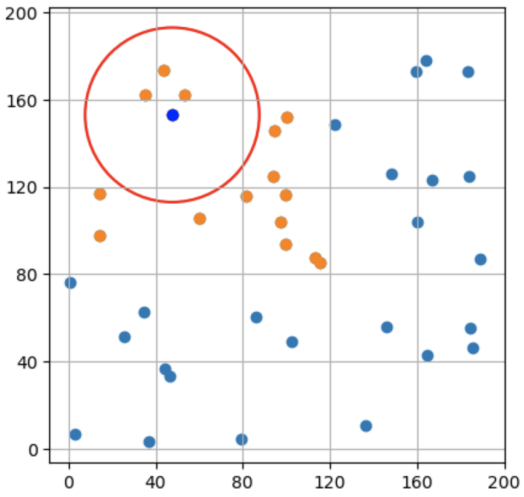


Abbildung 5.4: Visualisierung der Spatial Grid Datenstruktur für Sichtdistanz von 40.

`cell_range_array` beinhaltet für jede Zelle einen Startindex und einen Stopindex. Diese Indexwerte dienen als Verweis auf eine bestimmte Teilmenge der Indizes der Boids in den anderen Arrays.
Damit wir über zwei Indizes eine Teilmenge der Boids referenzieren können, müssen wir die Boids zuerst nach ihren Zellen sortieren. In Abbildung 5.5 wird eine beispielhafte Menge von Boids mit normaler Position und Zellposition dargestellt, welche nach den Zellpositionen sortiert wurde (erst nach Spalte, dann nach Zeile im Gitter). In farbig sind dort Indexbereiche markiert, welche Zellen repräsentieren. In Abbildung 5.6 sehen wir eine vereinfachte Darstellung des `cell_range_array`, nach dessen Befüllung mit Indexbereichen anhand der Boids in Abbildung 5.5.
Die zentralen Funktionen sind die folgenden:
`update_cell_position_array()` dient der Aktualisierung des `cell_position_arrays`.
`sort_flock_by_cell_position()` sortiert `last_position_array`, `last_speed_array`, `cell_position_array` und `cell_position_array` anhand der Zellposition.
`fill_cell_range_array()` liest die sortierten Boid-Positionen ein und erstellt daraus Indexbereiche. Zellen, in denen sich keine Boids befinden, behalten den initialisierten Indexbereich (-1, -1) bei. Die Funktionsweise der Funktion ist in Abbildung 5.5 und 5.6 visualisiert.
`get_index_list_by_cell()` liefert eine Liste mit Indizes für Boids die sich innerhalb einer spezifizierten Zelle befinden.
`get_possible_neighbour_index_list()` gibt eine Liste mit Indizes von möglichen Nachbarn für einen übergebenen Boid (bzw. dessen Index).

5.3.3 Analyse

Die zentralen Messergebnisse zum Spatial Grid sind in Abbildung 5.7 erkennbar. Im Vergleich zur naiven Variante bietet auch die Variante mit Spatial Grid einen wesentlichen Geschwindigkeitsvorteil.
Im Vergleich zur numpy-optimierten Variante und der Variante mit KD-Baum schneidet das Spatial Grid jedoch vergleichsweise schlechter ab (siehe Abbildung 5.10).
In diesem Kontext muss beachtet werden, dass das Spatial Grid unter unterschiedlichen Simulationsbedingungen besser bzw. schlechter funktioniert.

Index	x	y	cell_x	cell_y
0	36.87	2.98	0	0
1	2.89	6.51	0	0
2	34.73	62.55	0	1
3	25.54	51.51	0	1
4	0.64	76.21	0	1
5	14.3	116.65	0	2
6	14.07	97.68	0	2
7	35.18	162.42	0	4
8	44.07	36.87	1	0
9	46.31	33.18	1	0
10	79.23	4.45	1	0

Abbildung 5.5: Nach Spalte und Zeile im Grid sortierte Boids. Die Indexbereiche im `cell_range_array` sind in farbig markiert und finden sich auch in Abbildung 5.6 wieder (dort wird der ausgefüllte `cell_range_array` dargestellt).

	0	1	2	3
0	(0,1)	(2, 4)	(5, 6)	(-1, -1)
1	(8, 10)	...	...	...

Abbildung 5.6: Einträge für den (eigentlich dreidimensionalen) `cell_range_array` für die Boids aus Abbildung 5.5. Der Indexbereich $(-1, -1)$ stellt den Fall dar, dass sich keine Boids innerhalb dieser Zelle befinden. Die Einträge $(5, 6)$ in der Zelle $(0, 2)$ stehen dafür, dass innerhalb der sortierten Boidmenge die Boids mit Index 5 bis 6 innerhalb der genannten Zelle liegen.

Ballungen der Boids im Spatial Grid führen bspw. dazu, dass die Vorfilterung der Boids für die Nachbarschaftsbestimmung nur wenig nützlich ist. Bei gleichmäßig auf dem Simulationsfeld verteilten Boids kommen die Vorteile des Spatial Grids eher zum Vorschein.
Im speziellen Szenario, dass alle Boids sich in der gleichen Zelle befinden bringt das Spatial Grid rechentechnisch keine Vorteile, da die Vorfilterung der Boids dann nur zusätzliche Rechenkraft verbraucht.
Der zusätzliche Sortierschritt, in dem die Boids nach Zeile und Spalte im Grid sortiert werden trägt auch, im Vergleich zu einer klassischen Implementierung mit Listen, welche die Boids in einer Zelle beinhalten, zu einer höheren Laufzeitkomplexität bei. Wäre dieser Sortierschritt vermieden worden, hätten wir mit dieser Optimierungs-Variante möglicherweise bessere Ergebnisse erzielen können.

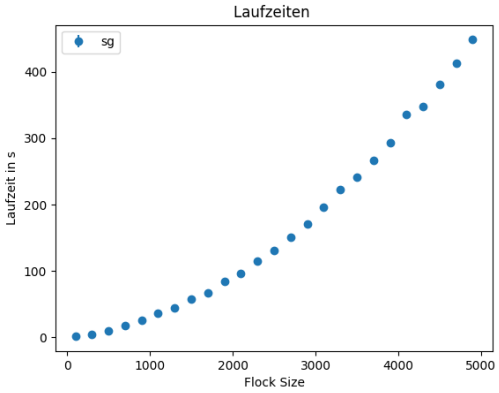


Abbildung 5.7: Spatial Grid, 50 Zeitschritte

5.4 Vereinfachte Distanzberechnung durch KD-Baum

5.4.1 Grundsätzliche Idee

KD-Bäume sind als räumliche bzw. geometrische Sonderform von binären Suchbäumen in k Dimensionen zu betrachten. Die grundlegenden Operationen wie Einfügen und Suchen bei einem KD-Baum sind aus diesem Grund ähnlich zu einem binären Suchbaum.

Damit ein KD-Baum aufgebaut werden kann ist wie bei einem binären Suchbaum sukzessive die Operation Insert aufzurufen. Angenommen bei k=2 hat man die Koordinaten X und Y. Dabei wird der aus n 2-dimensionalen Punkten bestehende Datensatz in mehrere rechteckige Regionen aufgeteilt, die durch Splitgeraden sowohl auf der X-Achse als auch auf der Y-Achse repräsentiert sind. Dabei ist die Anzahl der Splitgeraden eng mit der Anzahl der Elemente in den einzelnen Blättern verbunden, was widerum die Baumhöhe und damit auch die Suchzeit beeinflusst.

Ähnlich wie bei den binären Suchbäumen gibt es auch bei den KD-Bäumen die Möglichkeit, einen ausbalancierten Baum aufzubauen, wobei die Operation Insert die Komplexität O(logn) hat. Somit hat die Operation Aufbau eines ausgeglichenen KD-Baums bei n Datenpunkten die Komplexität O(nlogn). Bei der Grundvariante des KD-Baums handelt es sich um eine statische Datenstruktur. D.h. ein bestehender KD-Baum kann nur verändert werden indem die Update Operation neu ausgeführt wird. Grund dafür ist die Tatsache, dass kleine Anpassungen wie

einfaches Insert sehr kostspielig sind und bei großen Datenmengen wenig Vorteile bringen.

Um letztendlich die Distanz effizient zu berechnen, sind zunächst die Distanzen der k-nächsten Nachbarn inkl. deren Distanzen zum jeweiligen Punkt zu finden. Die Nächste-Nachbarn-Suche mithilfe eines KD-Baums basiert auf der Operation Search, die der Suche bei einem binären Suchbaum ähnelt. Der Unterschied besteht darin, dass man bei dem gesuchten Punkt ausgehend von der im aktuellen Knoten angegebenen Achse den jeweiligen X oder Y Wert mit dem Wert des aktuellen Knotens rekursiv vergleicht, da die Aufsplittung mehrdimensional ist. Daraufhin wird die Grundoperation Search mit der räumlichen euklidischen Abstandsberechnung kombiniert was zur Operation SearchNN führt. Im Falle der Boids Simulation ist zusätzlich noch die Distanz als Obergrenze anzugeben. Damit werden nur diejenigen Nachbarpunkte in die Lösungsmenge aufgenommen, die innerhalb dieser Schranke liegen. Somit liegt die Komplexität der Nächsten-Nachbar-Suche mithilfe des KD-Baums im schlechtesten Fall in O(logn), was pro Simulationsschritt bei n Datenpunkten zu O(nlogn) führt, was unter der quadratischen worst case Laufzeit der naiven Variante liegt.

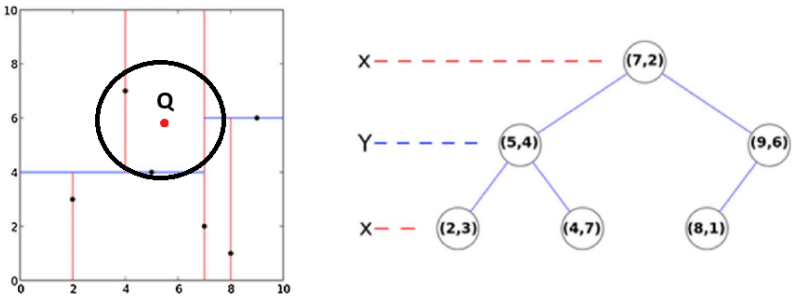


Abbildung 5.8: KD-Baum

5.4.2 Implementierung

Bei der Integration der Nächsten-Nachbar-Suche in die Simulation haben wir uns für die in SciPy implementierte Variante des KD-Baums entschieden. Zu diesem Schluss sind wir gekommen, weil zum Einen die Kosten einer effizienten Implementierung des KD-Baums von Grund auf (from scratch) den Nutzen übersteigen und zum Anderen die Implementierung des KD-Baums kein eigentliches Ziel dieses Projektes ist. In dem Python Modul tree.py wurde die Klasse Tree implementiert, die folgendermaßen aufgebaut ist:

```
1 import numpy as np
2 from scipy.spatial import KDTree
3 from config import FLOCK_SIZE
4
5 class Tree:
6
7     tree: KDTree = None
8     k = int(FLOCK_SIZE)
```

```
9
10      @classmethod
11      def update_kdtree(cls,points):
12          ...
13
14      @classmethod
15      def get_nearest_neighbor(cls, x, y, distance):
16          ...
```

Dabei enthält die Klasse Tree als Klassenvariablen den eigentlichen KD-Baum und die Anzahl der Nächsten-Nachbarn, die bei der Suche zu berücksichtigen sind. Die Klassenmethoden enthalten die Implementierungen der im letzten Abschnitt gennanten Operationen Aufbau (`update_kdtree`) und SearchNN (`get_nearest_neighbor`), wobei die Methode `update_kdtree` nichts zurückgibt. Dabei enthält die Methode `get_nearest_neighbor` ein Paar zusätzliche Zeilen Code. Diese dienen der Aufbereitung und Bereinigung des von der SciPy Methode nearest neighbor zurückgegebenen Ergebnisses, damit das Endergebnis an die Bedürfnisse der Boids Simulation effizient angepasst werden kann. Als Eingabeparameter enthält die Methode den jeweiligen Boid als X,Y Koordinate und die Distanz, die wie o.g. als Oberschranke dient. Diese Methode gibt ein mehrdimensionales NumPy Array, bei dem in jedem Index jeweils ein eindimensionales Array mit den Indices der Nächsten Nachbarn und ein eindimensionales Array mit den Distanzen zu diesen Nachbarn gespeichertist.

Die eigentliche Verwendung der Klasse Tree findet in dem Hauptmodul main.py in der Methode `main_kdt` statt. Dabei wird der KD Baum in jedem Simulationsschritt zunnächst neu aufgebaut, woraufhin dann für jeden Boid bzw. Punkt die Ergebnisse der Methode `Tree.get_nearest_neighbors` iterativ in einer Liste abgespeichert werden.

5.4.3 Analyse

Die unten stehende Abbildung gibt den Verlauf der Laufzeit abhängig von der Anzahl der Boids wieder. Die in den letzten Abschnitten erwähnte obere Laufzeitschranke von O(nlogn) bei n Boids kann tatsächlich die quadratische Laufzeit unterbieten. Dabei ist aber zusätzlich anzumerken, dass der graphische Verlauf der Simulation in der Abbildung 5.7 nicht überall einhetlich. So erkennt man, dass bis zu einer Boids Anzahl von 2000 ein linearer Verlauf der Laufzeit zu beobachten ist. D.h. im Durchschnitt und bei einer kleineren Anzahl von Boids (< 2000) könnte die tatsächliche Laufzeit sogar O(nlogn) etwas unterbieten.

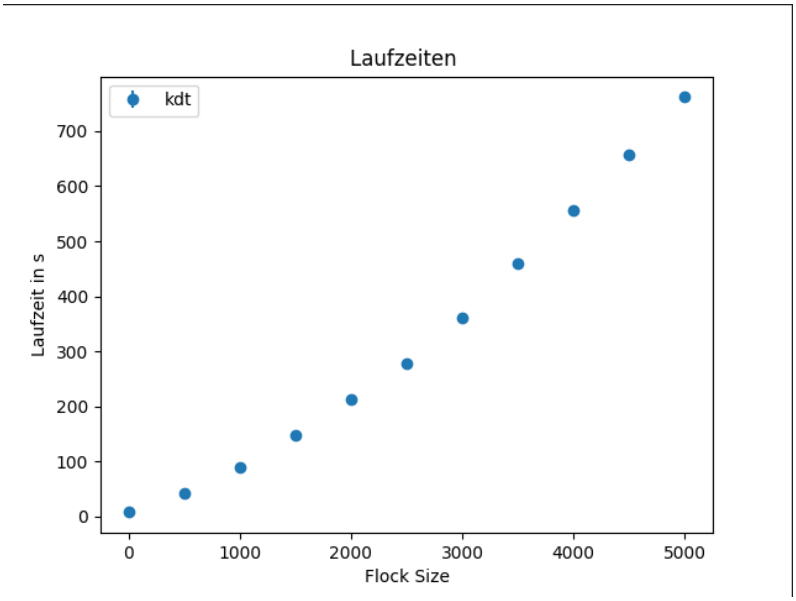


Abbildung 5.9: KD-Baum Laufzeiten in Abhängigkeit von der Flock_Size, bei 50 Zeitschritten, n=5

5.5 Weitere Verbesserungen

5.5.1 Alternative Norm für Distanzberechnung bei Nachbarschaftsbestimmung

Um zu berechnen, ob ein Boid mit einem anderen Boid benachbart ist, müssen wir überprüfen, ob die Distanz zwischen ihnen unter einem gewissen Schwellwert liegt. Dafür bilden wir die Differenz der Positionen der Boids und berechnen für diese eine Norm. Für Nachbarschaftsbestimmung wird klassischerweise die euklidische Norm genutzt:
Für einen Vektor $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ lautet diese:

$$|\vec{x}| = \sqrt{{x_1}^2 + {x_2}^2}$$

Für eine vorgegebene Sichtweite d (bspw. die `separation_distance`) überprüfen wir dann, ob die folgende Ungleichung gilt:

$$|\vec{x}| < d$$

bwz.

$$\sqrt{{x_1}^2 + {x_2}^2} < d$$

Durch Quadrieren der Ungleichung erhalten wir das folgende:

$${x_1}^2 + {x_2}^2 < d^2$$

Die Wurzel können wir also einfach weglassen und stattdessen bei der Überprüfung die quadrierte Sichtweite d^2 nutzen.
Das vermeidet das rechenaufwändige Ziehen der Wurzel bei jeder Distanzberechnung.
Unsere Konfigurationsdatei enhält deswegen entsprechend die Variablen mit den Endungen `_SQUARED`, welche eben diese unterschiedlichen quadrierten Distanzen darstellen.

Distanzberechnungen ohne die Wurzel waren bei unseren Tests etwa $1,25$ mal schneller als die Berechnungen mit Wurzel (die Messungen dazu können über das Jupyter-Notebook `distance_calculation_benchmark.ipynb` im Ordner `extra` nachvollzogen werden).

5.5.2 Function Inlining

Aus verschiedenen Gründen (Organisation, Wiederverwendbarkeit und Lesbarkeit) bietet es sich an, den Programmcode in einzelnen Funktionen zu organisieren. Aus Perspektive der Optimierung ist dies zunächst kritisch zu betrachten, da Funktionsaufrufe potenziell Kosten verursachen wie durch die Parameterübergabe. Daher ist es in kompilierten Sprachen ein typischer Ansatz, in der Optimierungsphase des Compilers Funktionen zu inlinen.
Wir haben in unserem Programm alle selbstgeschriebenen Funktionen ge-inlined und die Performance verglichen.

Variante	Laufzeit
normal	3.95 ± 0.07
inline	3.97 ± 0.06

Tabelle 5.3: 200 Boids, 200 Schritte, $n = 10$

Da die sich keine Unterschiede abseits der Standardabweichung der Messungen feststellen lassen, haben wir entschieden, die Struktur des Programms nicht zu verändern und die Anwendung der Regeln weiterhin in einzelnen Funktion zu lassen.

5.6 Finale Version als Ensemble-Verfahren

5.6.1 Grundsätzliche Idee

Es hat sich gezeigt, dass es mit dem Verfahren des KD-Baums gelungen ist, die quadratische Laufzeit zu unterbieten. Gleichzeitig hat sich gezeigt, dass bei geringeren Flock_Sizes die einfache Variante noch schneller ist, was auf den Aufwand für den Aufbau des KD-Baums zurückzuführen ist.

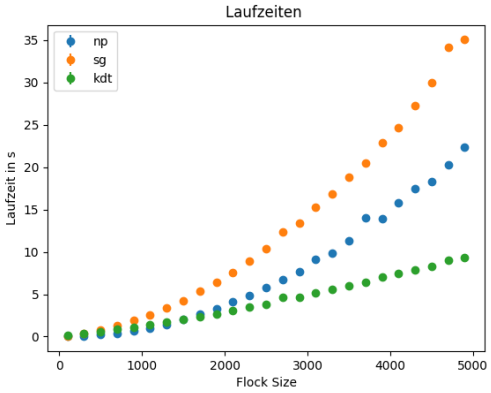


Abbildung 5.10: Laufzeitvergleich der Varianten bei 5 Zeitschritte in Abhängigkeit von der Flock_Size, $n = 1$

Da bei unseren Messungen der genaue Punkt, an dem die Variante mit dem KD-Baum die np-Variante überholt, hardwareabhängig ist, haben wir uns dafür entschieden, die finale Ver-

sion als Ensemble-Verfahren realisieren, bei dem vor der Ausführung aller Simulationsschritte ein Problelauf für einen Zeitschritt mit allen Varianten ausgeführt wird. Dies lohnt sich bei kleineren Flock_Sizes insbesondere bei einer großen Anzahl an Zeitschritten n, denn es gilt $0 < n <$ für die Programmlaufzeit $d(n)$

$$\exists n' : \forall n > n',$$

$$d(n)_{probelauf} << (d(n)_{kdtree} - d(n)_{np})$$

Bei Messungen mit einer Flock_Sizes von 200 hat sich gezeigt, dass dieser Wert bereits bei einer einstelligen Anzahl an Zeitschritten erreicht wird.

6 Git-Repository

Zur Sourcecodeverwaltung wurde git benutzt. Die verschiedenen Varianten wurden alle im Repository unter `https://github.com/Lezin901/boids_simulation` erbeitet. Die finale Version mit Docker ist unter `https://github.com/Lezin901/boids_simulation_final` abrufbar.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Reynolds, C. [1995], 'Boids Background and Update', `https://www.red3d.com/cwr/boids/`.

[2] Reynolds, C. W. [1987], 'Flocks, herds and schools: A distributed behavioral model'. **URL:** *http://dx.doi.org/10.1145/37401.37406*