Abschlussprüfung des Moduls
D3.3 Data Science und AI Infrastrukturen


Im Studiengang
Data Science, AI und Intelligente Systeme

Michael Adzase
Ronald Amstrong Batchewoua Batchewoua

Düsseldorf
18. Januar 2025

# Inhaltsverzeichnis

# Thema und Grundlagen

Bei der vorliegenden Projektdokumentation handelt es sich um ein Projekt mit dem Namen Particle Life Simulator im Rahmen des Moduls Data Science und AI Infrastrukturen.

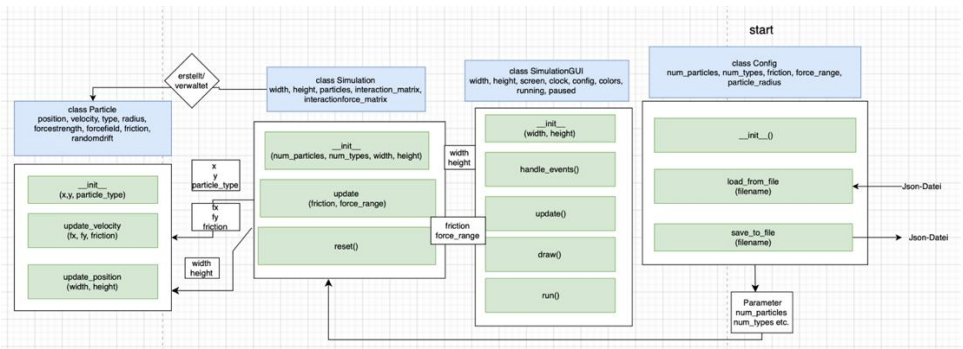
## Was ist Particle Life?

Es ist eine Simulation, bei der Partikel als Punkte in einem Darstellungsrahmen miteinander interagieren. Jedes Partikel gehört zu einem Typ mit einer Farbe (Rot, Grün, Blau oder Gelb) und diese haben Regeln und können sich gegenseitig anziehen oder abstoßen. Daraus entsteht emergentes Verhalten: Interaktionsmuster und Systeme, die nicht aus den isolierten Eigenschaften der einzelnen Teile vorausgesagt werden können. Quelle: https://www.beansandbytes.de/ki-lexikon/emergentes-verhalten#:~:text=Emergentes%20Verhalten%20bezeichnet%20ein%20Phänomen,des%20Systems%20vorhergesagt%20werden%20können.

## Erste Überlegungen zu Wirkungs-Kräften

Kräfte, die bei einer Simulation beachtet werden können, sind zum Beispiel die Gravitationskraft, die Coulomb-Kraft, Van-der-Wals-Kräfte oder die Lorentz-Kraft. Für eine vereinfachte Simulation haben wir uns erstmal gedacht, dass man sowohl die Anziehung und die Abstoßung als auch Neutralität integriert zwischen Partikeln. Ein Partikel hat einen eigenen Radius und einen Radius um sich, in dem die Kräfte wirken. Außerhalb des bestimmten Radius wirken die Kräfte nicht mehr. Innerhalb der Simulation können nicht alle Partikel konstant mit allen Partikeln interagieren, das wäre zu aufwendig, deshalb ist es sinnvoller einen Wirkungs-Radius zu bestimmen. Außerdem kann eine Reibungskraft bestehen zwischen den Partikeln, die entweder zu Richtungsänderung oder Verlangsamung führen können. Die Verlangsamung kann durch negative Beschleunigungskraft dargestellt werden, so kann diese auch durch höhere Abstoßungskräfte wiederum steigen. Die Beschleunigung eines Partikels hängt auch von seiner Masse ab. Um einen Verlauf darzustellen, brauchen wir einen Startpunkt und Zeit-Schritte, die immer wieder erneuert werden.

Unser erstes Diagramm (erstellt mit draw.io):




Wir brauchen eine Formel, die Masse und Beschleunigung berücksichtigt und der Code darf nicht unendlich viele Partikel (Objekte) erzeugen. Man braucht nicht unbedingt eine eigene Particle-Klasse, außerdem skalieren doppelte For-Schleifen schlecht, das müssen wir im weiteren Verlauf berücksichtigen.

## Coulomb-Kraft statt Lorentz-Kraft

Ursprünglich war geplant, die Lorentzkraft zu verwenden. Nach einer Teambesprechung haben wir uns für die Coulomb-Kraft entschieden.

Lorentzkraft:
$\vec{F} = q * (\vec{v} \times \vec{B})$
Probleme mit der Lorentzkraft:

- Beschreibt Kraft auf bewegte Ladungen in Magnetfeldern
- Kraft hängt von der Geschwindigkeit ab (komplizierter)
- Wir simulieren kein Magnetfeld
- Bahnen sind schwer vorhersagbar

Coulomb-Kraft:
$\vec{F} = k * (q1 * q2) / r^2 * \hat{r}$
Vorteile der Coulomb-Kraft:

- Nur vom Abstand abhängig
- Erzeugt natürliche Anziehung (unterschiedliche Ladung) und Abstoßung (gleiche Ladung)
- Bekanntes Verhalten (wie bei Planeten, Atomen)
- Einfacher zu implementieren und debuggen

## Kräfte für einfache Partikel-Simulationen

Wir haben außerdem noch überlegt, welche Möglichkeiten es gibt, die Partikel einfach miteinander interagieren zu lassen.

### 1. Anziehung / Abstoßung – Mögliche Varianten

Variante A: Konstante Kraft (am einfachsten)

```
if abstand < radius:
    kraft = stärke * richtung  # +stärke = Abstoßung, -stärke = Anziehung
```

Variante B: Lineare Kraft (je näher, desto stärker)

```
kraft = stärke * (1 - abstand / radius) * richtung
```

Variante C: Inverse Kraft (1/r, wie Gravitation vereinfacht)

```
kraft = stärke / abstand * richtung
```

Variante D: Inverse Quadrat ($1/r^2$, echte Physik) – das haben wir gewählt

```
kraft = stärke / (abstand ** 2) * richtung
```

Wir haben uns für Variante D (Coulomb mit $1/r^2$) entschieden, weil sie physikalisch fundiert ist und interessantes Verhalten erzeugt.

## 2. Reibung / Dämpfung – Mögliche Varianten

Variante A: Multiplikativ (unsere Wahl)
→ Geschwindigkeit wird jeden Frame reduziert
velocity *= 0.95  # 5% Verlust pro Frame

Variante B: Subtraktiv (wie echte Reibung)
→ Feste Menge abziehen, Richtung beibehalten
velocity -= friction * np.sign(velocity)

Variante C: Geschwindigkeitsabhängig (Luftwiderstand)
→ Je schneller, desto mehr Widerstand
velocity -= friction * velocity * abs(velocity)

Wir haben uns für Variante A entschieden, weil sie am einfachsten ist und ne stabile Simulationen erzeugt.

## 3. Rahmen – Mögliche Varianten

Variante A: Wrapping (unsere Wahl)
pythonx = x % WIDTH

Variante B: Abprallen (Reflexion)
pythonif x < 0 or x > WIDTH:
  velocity_x = -velocity_x

Variante C: Weiche Abstoßung vom Rand
pythonif x < rand_zone:
  velocity_x += kraft_stärke

Wir haben uns für Variante A (Wrapping) entschieden, damit Partikel nicht am Rand "hängen bleiben".

# Verwendete Kräfte

- Coulomb-artige Anziehung/Abstoßung → $F = q_1 q_2 / r^2$

- Stärke und Vorzeichen aus der Interaktionsmatrix
  Vereinfachung: k = 1, "Ladungen" sind Interaktionswerte

- Viskose Reibung: v_neu = v_alt × (1 - friction)
  Dämpft die Bewegung, verhindert unbegrenztes Beschleunigen.

- Periodische Randbedingungen: x = x mod WIDTH →
  Partikel, die den Rand verlassen, erscheinen auf der anderen Seite.

## Erklärung zu Coulomb

- Gegeben: Zwei Partikel mit Positionen $\vec{r}1$ und $\vec{r}2$
- Verbindungsvektor:

$\vec{r} = \vec{r}1 - \vec{r}2$

- Abstand:

$r = |\vec{r}| = \sqrt{(dx^2 + dy^2)}$

- Einheitsvektor (Richtung):

$\hat{r} = \vec{r} / r$

- Coulomb-Kraft:

$\vec{F} = k * (q1 * q2) / r^2 * \hat{r}$

k = 1 (Konstante, vereinfacht)
q1 * q2 aus Interaktionsmatrix
Positiv = Abstoßung, Negativ = Anziehung

- Beschleunigung (bei Masse = 1):

$\vec{a} = \vec{F}$

- Geschwindigkeit ändern:

$\vec{v}\_neu = \vec{v}\_alt + \vec{a} * \Delta t$

- Position ändern:

$\vec{r}\_neu = \vec{r}\_alt + \vec{v}\_neu * \Delta t$

## Beispielrechnung (aus der Teambesprechung):

- Zwei Teilchen:

Teilchen 1: Position (-1, 0), Geschwindigkeit (0, +0.5), Ladung +1
Teilchen 2: Position (+1, 0), Geschwindigkeit (0, -0.5), Ladung -1

- Schritt 1-3: Verbindungsvektor und Abstand

$\vec{r} = (-1, 0) - (1, 0) = (-2, 0)$
r = 2

$\hat{r}$ = (-1, 0)

- Schritt 4: Coulomb-Kraft

q1 * q2 = +1 * -1 = -1
$\vec{F}$1 = 1 * (-1) / 4 * (-1, 0) = (0.25, 0) → Kraft nach rechts (Anziehung!)

- Schritt 6: Geschwindigkeit ändern (Δt = 0.01)

$\vec{v}$1_neu = (0, 0.5) + (0.25, 0) * 0.01 = (0.0025, 0.5)
Schritt 7: Position ändern
x1_neu = -1 + 0.0025 * 0.01 = -0.999975
y1_neu = 0 + 0.5 * 0.01 = 0.005

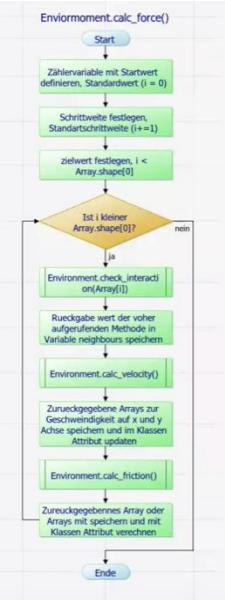
→ Nach vielen Schritten entstehen elliptische/orbitartige Bahnen

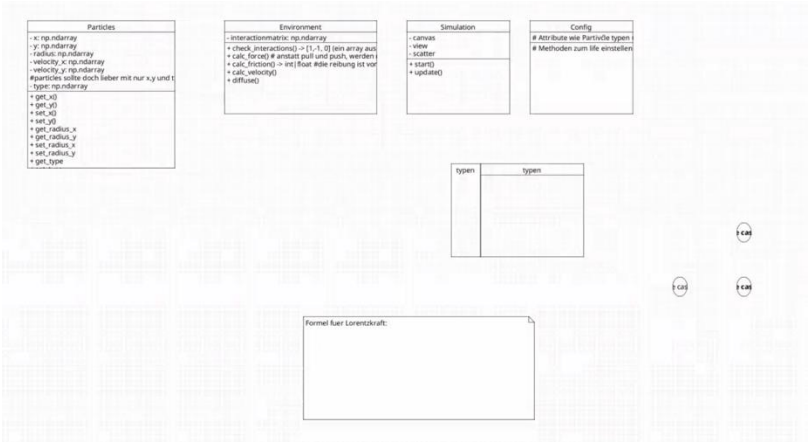
# Projektmanagement

Wir haben für das Projektmanagement GitHub, Teams und Whatsapp verwendet.

## Ablauf des Projekts

Während der Projekttreffen haben wir zuerst Ideen diskutiert, wie man die Kräfte am besten in einen Zusammenhang bringen kann. Ronald hat auf GitHub zuerst ein Repository erstellt und die ersten Ordner angelegt. Die ersten Issues, die Dokumentation und das Readme wurden von Katharina erstellt. Nach vielen Überlegungen haben wir die Struktur des Projekts geändert und Michael hat mithilfe von PapDesigner und umlet.com Diagramme erstellt:




Nachdem wir uns für eine Struktur entschieden haben, hat Ronald ein Kanban-Board und Issues hinzugefügt, die der Struktur entsprechen. Außerdem hat er CI ergänzt. Da wir noch keinen routinierten Umgang mit GitHub haben, sind bei den Issues und bei der Bearbeitung Schwierigkeiten aufgetreten. Im weiteren Verlauf werden wir beschriebene und allgemeinere Issues erstellen und diese mit den Push- bzw. Pull-Requests verbinden, um einen Überblick zu bekommen. Die Issues, die erstellt werden müssen, sind: 1. ParticleSystem Klasse, 2. Simulation Klasse, 3. Config mit Interaktionsmatrix, 4. Ordnerstruktur überarbeiten, 5. Readme updaten. Getter- und Setter-Issues #21 bis #28 können erstmal geschlossen werden, wenn man in der Partikel-Klasse @property als Decorator nimmt.

## Ablaufsplan

Während der Projekttreffen haben wir zuerst Ideen diskutiert, wie man die Kräfte am besten in einen Zusammenhang bringen kann.

**Phase 1: Setup (R.)**

- Ronald hat auf GitHub zuerst ein Repository erstellt und die ersten Ordner angelegt
- Die ersten Issues, die Dokumentation und das README wurden von Katharina erstellt

**Phase 2: Architektur (M.)**

- Nach vielen Überlegungen haben wir die Struktur des Projekts geändert
- Michael hat mithilfe von PAP-Designer und UMLet Diagramme erstellt:

- Klassendiagramm mit 4 Klassen (Particles, Environment, Simulation, Config)
- Programmablaufpläne für alle Methoden

**Phase 3: Implementierung (R., M.)**
- Ronald hat ein Kanban-Board und Issues hinzugefügt, die der Struktur entsprechen
- CI wurde ergänzt (GitHub Actions mit Conda, ruff, pytest)
- M. hat das Masken-Konzept für die Nachbarschaftssuche entwickelt
- R. hat die Config-Datei und erste Environment-Methoden implementiert

**Phase 4: Konsolidierung (Team)**
- Da wir noch keinen routinierten Umgang mit GitHub haben, sind bei den Issues und bei der Bearbeitung Schwierigkeiten aufgetreten
- Im weiteren Verlauf werden wir beschriebene und allgemeinere Issues erstellen und diese mit den Push- bzw. Pull-Requests verbinden

**Geplante Umbenennungen:**
- Environment.py → simulation.py (enthält die Physik-Simulation)
- simulation.py → visualize.py (enthält die Visualisierung)

### Aufgabenverteilung
Wir geben hier an, an welchen Aufgaben die Personen beteiligt waren.

- Michael: Architektur, 2. Klassendiagramm, Ablaufplan Pap, Masken-Konzept, Environment-Klasse

- Ronald: Issues, Config, Environment-Ergänzung, 1. Entwurf Partikel-Klasse

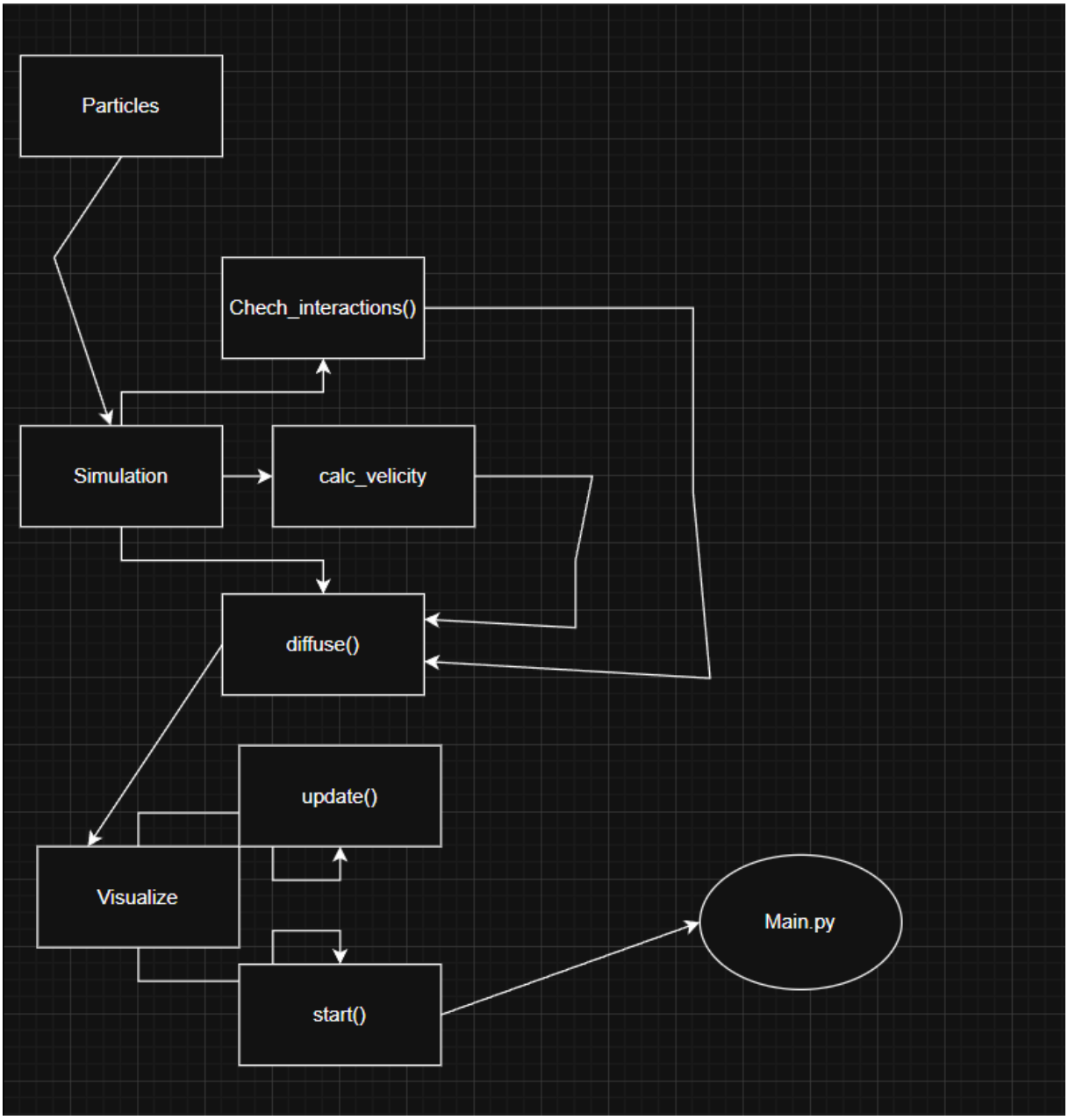
## Programmaufbau

Das Programm besteht auf Basis des objektorientierten Programmierens aus verschiedenen Klassen und einer Simulation.

## Ordnerstruktur

```
Particle_Life_Simulation
├── backend
│   ├── __init__.py
│   ├── particles.py
│   └── simulation.py
├── frontend
│   ├── __init__.py
│   └── visualize.py
├── config
│   ├── __init__.py
│   ├── config.py
│   └── constants.py
├── tests
│   ├── __init__.py
│   ├── test_particle_system.py
│   └── test_simulation.py
├── diagrams
│   └── class_diagram.png
├── main.py
├── pyproject.toml
├── README.md
├── LICENSE
├── requirements.txt
└── .gitignore
```

## Code Diagramm




# Backend

## Particles

→Sie erstellt Listen (NumPy-Arrays) für Positionen (x, y), Geschwindigkeiten (v_x, v_y) und Typen (Farben) der Teilchen.
→Wenn die Simulation startet, würfelt die Klasse zufällige Startpositionen und Typen aus, damit die Teilchen verteilt sind.
→**Getter (@property):** Erlauben anderen Programmteilen (wie der Grafik oder Physik), die Daten **anzusehen**.

→**Setter (@x.setter):** Erlauben der Physik-Engine, die Daten zu **aktualisieren** (z. B. eine neue Position nach einer Bewegung zu speichern).

## Simulation

Die Simulation-Klasse berechnet die Physik: Kräfte, Geschwindigkeiten und Positionen.

| Methode | Beschreibung |
|---|---|
| check_interactions(index) | ■ **Filterung:** Prüft, welche Teilchen innerhalb des Interaktions-Radius liegen.<br>■ **Optimierung:** Nutzt Slicing (index + 1), um Doppelberechnungen zu vermeiden (Newton III).<br>■ **Daten-Abfrage:** Identifiziert die Typen der Nachbarn und holt die passenden Interaktionswerte aus der Matrix.<br>■ **Übergabe:** Liefert die Koordinaten und Typen der relevanten Nachbarn an calc_velocity weiter. |
| Calc_velocity() | Sie berechnet die **Beschleunigung** basierend auf der Physik:<br>■ **Vektoren:** Misst Richtung und Distanz zu Nachbarn.<br>■ **Interaktion:** Holt den Kraftwert (k_ij) aus der Matrix (Anziehung/Abstoßung).<br>■ **Newton III:** Wendet die Kraft auf das aktuelle Partikel an und die Gegenkraft auf die Nachbarn.<br>■ **Update:** Verändert die **Geschwindigkeit** (velocity_x/y) der Partikel. |
| Diffuse() | ■ **Suche:** Findet via check_interactions alle Nachbarn innerhalb des Radius.<br>■ **Aufruf:** Übergibt diese Daten an calc_velocity.<br>■ **Bewegung:** Addiert die finale Geschwindigkeit auf die **Position**.<br>■ **Output:** Gibt die neuen Koordinaten an die Grafik-Engine zurück. |

## Interaktionsmatrix

Die Interaktionsmatrix definiert, wie sich verschiedene Partikeltypen zueinander verhalten:

| Wert | Bedeutung |
|---|---|
| • +1.0 | → Starke Abstoßung (gleiche "Ladung") |
| • -1.0 | → Starke Anziehung (unterschiedliche "Ladung") |
| • ±0.5 | → Schwächere Wechselwirkung |
| • 0.0 | → Neutral (keine Wechselwirkung) |

### Masken-Konzept

Statt alle Partikel-Paare zu prüfen (O($n^2$)), nutzen wir Masken für effiziente Filterung.
Wir können mit **particles.x[mask]** direkt alle Nachbarn holen, also ohne Schleife.

# Frontend

## Visualize

- Die Leinwand (Canvas): Sie erstellt das Fenster und das Koordinatensystem, in dem die Partikel leben. Mit der Maus kannst du darin zoomen oder die Ansicht verschieben.

- Die Darstellung (Scatter): Alle Partikel werden als "Marker" (Punkte) gezeichnet. Hier wird festgelegt, wie groß die Punkte sind (z. B. size=6).

- Die Farbwahl (Palette): Sie übersetzt die Nummern der Partikel-Typen (0, 1, 2, ...) in tatsächliche Farben (Rot, Grün, Blau...), damit du die Gruppen unterscheiden kannst.

- Der Timer: Das ist der wichtigste Teil. Alle 0.0016 Sekunden löst der Timer die update-Funktion aus.

-

Die update-Funktion
Jedes Mal, wenn der Timer tickt, passiert folgendes:

1. Rechnen: Sie ruft simulation.diffuse() auf, um die neuen Positionen der Partikel berechnen zu lassen.
2. Umwandeln: Sie bringt die neuen x- und y-Werte in ein Format, das die Grafikkarte versteht (np.c_[x, y]).
3. Zeichnen: Sie schickt die neuen Positionen und Farben an den Monitor, sodass die Partikel für dich "wandern".

# Config

Es sind 2 Dateien in Config/ enthalten. Die erste Datei ist config.py und die zweite ist constants.py

Config.py enthält die Einstellungen, die für das Ausführen des Programms nötig ist. Von dort wird das ganze eingestellt und erst dann an main.py übergeben.

Constants.py enthält die Konstante die für das Programmieren des Simulators und auch fürs Ausführen behilflich sind.

# Fazit & Ausblick

**1. Das Architektur-Konzept (MVC-Struktur)**

Das Projekt folgt einer klaren Trennung von Verantwortlichkeiten (Model-View-Controller Prinzip):

- **Model (**Particles.py**)**: Der reine Datenspeicher. Er weiß nicht, wie man rechnet oder zeichnet, sondern hält nur die Arrays für und die Typen bereit.

- **Controller (**Simulation.py**)**: Das Gehirn. Hier wird die Physik berechnet. Es liest Daten aus den Particles, berechnet Kräfte und Interaktion zwischen denen und schreibt die neuen Werte zurück.

- **View (**Visualize.py**)**: Das Fenster zur Außenwelt. Es nimmt die Daten und bringt sie über die GPU (VisPy) auf den Schirm.

**2. Der Kern-Algorithmus (Der Simulationszyklus)**

Pro Zeitschritt läuft folgender Prozess ab:

1. **Nachbarschaftssuche**: check_interactions filtert für jedes Teilchen aus, wer im Einflussradius liegt.

2. **Kraftberechnung**: calc_velocity schaut in die **Interaktionsmatrix**. Ist der Wert positiv, ziehen sich die Teilchen an, ist er negativ, stoßen sie sich ab.

3. **Integration**: diffuse berechnet die neue Position ().

**3. Die Interaktionsmatrix**

Die Matrix ist das wichtigste Werkzeug. Sie definiert die Regeln:
- Sie bestimmt, wie sich die Farben miteinander verhalten. Sie beantwortet die Frage, ob sich 2 Farben anziehen oder abstoßen werden.
- Durch das **Live-Fenster** können diese Regeln nun in Echtzeit überwacht werden.

**4. Technischer Stack**

- **NumPy**: Ermöglicht Vektorisierung. Statt 5000 Teilchen einzeln zu berechnen, werden ganze Arrays gleichzeitig verarbeitet.
- **VisPy**: Nutzt Canvas, um die Darstellung flüssig zu halten, indem die Rechenlast der Grafik auf die Grafikkarte ausgelagert wird.
- **Pytest**: Stellt sicher, dass die Logik (z. B. "bewegen sich Teilchen bei Anziehung wirklich aufeinander zu?") auch nach Code-Änderungen noch stimmt.