Eco Simulator 2.0 – Entwicklerdokumentation

Einleitung

Eco Simulator 2.0 ist ein umfassendes Simulationsframework zur Modellierung ökologischer Systeme auf zwei Ebenen. Es kombiniert ein klassisches **Makromodell** (Differentialgleichungen nach Lotka-Volterra) mit einem **Mikromodell** (individuenbasiertes, agentenbasiertes System). Dadurch können sowohl aggregierte Populationsdynamiken als auch emergentes Verhalten einzelner Organismen und evolutionäre Effekte simuliert werden. Zusätzlich integriert das Projekt Methoden der **Sensitivitätsanalyse** und des **maschinellen Lernens**, um Simulationsergebnisse auszuwerten, Vorhersagen zu treffen und optimale Parameter zu empfehlen. Ein modernes **Streamlit**-Frontend mit ansprechendem UI ermöglicht interaktive Steuerung der Simulation und Live-Visualisierung der Ergebnisse. Die Bereitstellung via **Docker** erleichtert zudem das Deployment in unterschiedlichen Umgebungen.

Hauptfunktionen im Überblick:

- **Makro-Simulation (ODE-Modell):** Klassisches Räuber-Beute-Modell (Lotka-Volterra) mit Erweiterungen (Virus-Infektion, Nährstoff/Beute-Konkurrenz) und kontinuierlicher Differentialgleichungsintegration.
- **Mikro-Simulation (ABM-Modell):** Agentenbasiertes Evolutionsmodell auf einer Raster-Welt, in der individuelle Organismen mit Eigenschaften wie **Geschwindigkeit**, **Sichtweite**, **Stärke**, **Tarnung** und **Neugier** (curiosity) agieren, Ressourcen sammeln, sich reproduzieren (mit Mutation) und über Generationen adaptieren.
- **Sensitivitätsanalyse & Reporting:** Automatisches Durchführen von Parameterstudien (Batch-Simulationen mit variierenden α, β, φ usw.), Ergebnisaggregation in CSV-Dateien und Generierung von visuellen Auswertungen (Heatmaps, Trendkurven) sowie automatisierten Berichten (HTML/PDF).
- **Maschinelles Lernen Integration:** Training von Random-Forest-Modellen auf Simulationsdaten zur **Vorhersage** von Ergebnisgrößen (z.B. Endpopulation des Virus) auf Basis der Eingabeparameter, sowie zur Prognose von Trends in der Agenten-Evolution. Diese Modelle werden im Live-Frontend genutzt, um **sofortige Abschätzungen** zu liefern.
- **Streamlit UI & UX:** Intuitive Web-Oberfläche zum Einstellen von Simulationsparametern (via Slider), Starten/Stoppen von Simulationen, Anzeigen von Zeitreihenplots und ML-Vorhersagen. Mehrere Tabs trennen die Makro- und Mikro-Simulation. Ein **Dashboard** fasst Kennzahlen zusammen (z.B. durchschnittliche Populationsgrößen, Mutationsraten) und zeigt Korrelationen.
- **Modularer Aufbau & Erweiterbarkeit:** Klare Trennung der Komponenten (Simulationskernel, ML, UI, Tools), gut dokumentierter Code und

Konfigurationsobjekte ermöglichen es Entwickler:innen, das System leicht zu erweitern (neue Arten, neue Parameter, alternative Modelle oder Integratoren).

Diese Dokumentation erläutert die zugrundeliegende **Architektur**, die **Datenflüsse** im System, wichtige **Module und Schnittstellen (APIs)** sowie die wissenschaftlichen Konzepte. Sie dient als Leitfaden für Entwickler:innen, um sich im Code zurechtzufinden, neue Features zu integrieren, bestehende Simulationen zu erweitern und das System weiter zu trainieren bzw. zu optimieren.

Motivation

Die Motivation für Eco Simulator 2.0 liegt in der Verbindung **klassischer ökologischer Modellierung** mit **modernen KI-Methoden** und **Simulation individueller Agenten**. Traditionelle Modelle wie das Lotka-Volterra-Räuber-Beute-System (entwickelt 1925) beschreiben Populationen über Differentialgleichungen und liefern analytische Einblicke in Stabilität und Oszillationen von Ökosystemen. Solche Makromodelle stoßen jedoch an Grenzen, wenn komplexeres Verhalten oder Evolution berücksichtigt werden soll.

Gleichzeitig erlaubt die **agentenbasierte Modellierung (ABM)** die Simulation jedes einzelnen Individuums und seines Verhaltens. Dadurch können emergente Phänomene – z.B. Anpassungen durch Mutation und Selektion, Raumverteilung von Organismen oder komplexe Verhaltensstrategien (etwa Neugier vs. Ausbeutung von Ressourcen) – abgebildet werden, die in Gleichungssystemen schwer zu fassen sind. ABMs sind jedoch rechenintensiv und ihre Ergebnisse oft schwerer analytisch zu verstehen.

Eco Simulator 2.0 wurde entwickelt, um **das Beste beider Welten** zu vereinen: Das schnelle, gut verstandene ODE-Modell auf Makro-Ebene und das detaillierte ABM auf Mikro-Ebene. Dies ermöglicht etwa:

- **Validierung und Vergleich:** Das Makromodell liefert Referenzdynamiken (z.B. klassische Räuber-Beute-Zyklen), gegen die das ABM-Ergebnis qualitativ geprüft werden kann. Umgekehrt können Hypothesen aus dem Makromodell (z.B. Auswirkungen von Parametervariationen) auf Mikro-Ebene untersucht werden.
- **Erweiterte Szenarien:** Durch zusätzliche Komponenten im Makromodell (Virus, Bakterien, Nährstoffe, Immunreaktion) werden realistischere Ökosystem-Aspekte simuliert. Die Mikro-Ebene fügt Evolution hinzu – Organismen können sich über Generationen an veränderte Bedingungen anpassen, was in statischen ODE-Modellen nicht möglich ist.
- **Sensitivitätsanalyse & KI-Unterstützung:** Durch automatisierte Simulationsreihen mit variierenden Parametern können **kritische Parameter** identifiziert werden (wo z.B. Populationen aussterben oder explodieren). Die entstehenden großen

Datenmengen werden mittels Machine Learning verdichtet: So können *surrogate models* (Stellvertreter-Modelle) gelernt werden, die komplexe Simulationsergebnisse instantan annähern. Dies beschleunigt z.B. die Suche nach stabilen Ökosystem-Konfigurationen erheblich.

- **Anschauliche Vermittlung:** Die interaktive Oberfläche dient auch Lehr- und Demonstrationszwecken. Nutzer können intuitiv verstehen, wie Veränderungen der **Beute-Wachstumsrate α** oder **Räuber-Fressrate β** die Populationsdynamik beeinflussen, oder wie **Neugier** und **Mutationsrate** die Evolution einer Agentenpopulation steuern.

Insgesamt adressiert Eco Simulator 2.0 sowohl **wissenschaftliche Fragestellungen** (Mehrskalen-Modelle, Evolution in Ökosystemen) als auch **praktische Anforderungen** (interaktive Exploration, schnelle Vorhersagen, Erweiterbarkeit für neue Forschungsideen). Die Software schafft eine Plattform, auf der Entwickler:innen Experimente durchführen, eigene Modelle einbinden und Erkenntnisse über ökologische Systeme gewinnen können.

Architektur

Die Architektur von Eco Simulator 2.0 folgt einem **Dual-Layer-Design**, bestehend aus der **Makro-Ebene** (aggregierte Simulation mit Differentialgleichungen) und der **Mikro-Ebene** (Agentenbasierte Simulation). Beide Simulationsebenen sind in separaten Modulen gekapselt, teilen aber eine ähnliche Struktur: es gibt jeweils Konfigurationsobjekte, Simulator-Klassen zur Ausführung der Simulation sowie Hilfsfunktionen zur Auswertung. Übergeordnete Komponenten (UI, ML, Reporting) nutzen diese Module. Die folgende Übersicht zeigt die Hauptkomponenten und ihren Zusammenhang:

```
flowchart LR
    subgraph UI [Streamlit UI & Steuerung]
        A[Benutzereingaben<br/>(Parameter, Aktionen)]
        B[Visualisierung<br/>(Plots, Kennzahlen, Vorhersagen)]
    end
    subgraph Macro[Makro-Simulation (ODE)]
        C[SimulationConfig<br/>(Parameter α, β, φ, ...)]
        D[Simulator (ODE-Löser)<br/>Lotka-Volterra + Erweiterungen]
        E[(Makro-Ergebnisse)<br/>(Zeitreihen: Beute, Räuber, ...)]
    end
    subgraph Micro[Mikro-Simulation (ABM)]
        F[AgentSimConfig<br/>(Parameter Neugier, Mutation, ...)]
        G[AgentSimulator (ABM)<br/>Rasterwelt, Agentenliste]
        H[(Mikro-Ergebnisse)<br/>(Zeitreihen: Population, Traits)]
    end
    subgraph Analysis[Analyse & ML]
        I[(Sensitivitäts-\nErgebnisse CSV)]
        J[Report Engine<br/>(Heatmaps, Berichte)]
        K[ML Predictor<br/>(Random Forest Modelle)]
    end

    A -->|Start Makro-Sim| C --> D --> E --> B
    A -->|Start Mikro-Sim| F --> G --> H --> B
```

```
A --|Trainingslauf|--> D
D --|BatchRuns|--> I
G --|BatchRuns|--> I
I --> J --> B
I -->|Trainiere Modelle| K
K --> B
```

Abb.: Architekturüberblick. Die Benutzereingaben aus der UI initialisieren entweder die Makro- oder Mikro-Simulation. Beide Simulatoren erzeugen Zeitreihendaten, die direkt in der UI visualisiert werden. Zusätzlich können Batchruns durchgeführt werden (Sensitivitätsanalysen), deren aggregierte Ergebnisse in CSV-Dateien gespeichert (I) und anschließend von der Report-Engine ausgewertet (J) oder zum Training der ML-Predictor-Modelle (K) genutzt werden. Die trainierten ML-Modelle stellen wiederum Vorhersagen bereit, die im UI angezeigt werden (z.B. erwartete Endpopulation).

Makro-Ebene: ODE-Ökosystem

Die Makro-Ebene basiert auf einem System gekoppelter **Differentialgleichungen**, das klassische Räuber-Beute-Dynamik mit zusätzlichen ökologischen Komponenten kombiniert. Dieses Modul ist im Paket predator_prey_sim implementiert und umfasst folgende Hauptbestandteile:

- **Konfiguration (SimulationConfig):** In dieser Datenklasse sind alle Parameter des Makromodells definiert. Wichtige Parameter sind unter anderem:
    - alpha (α): **Beute-Wachstumsrate** – bestimmt, wie schnell die Beute-Population (z.B. Pflanzen oder Beutetiere) ohne Räuber wächst. Biologisch entspricht α der Geburtenrate bzw. dem Ressourcenangebot für die Beute. Rechnerisch ist es der Koeffizient im Differentialgleichungsterm α * Beute.
    - beta (β): **Räuber-Fressrate** – gibt an, wie stark die Räuber-Präsenz die Beute dezimiert. Biologisch repräsentiert β die Effizienz der Räuber beim Jagen der Beute (höheres β = Beute wird schneller gefressen). Im klassischen Lotka-Volterra-Modell erscheint β im Term - β * Beute * Räuber und koppelt die beiden Populationen.
    - delta (δ): **Konversionsrate Beute→Räuber** – Faktor, wie effektiv gefressene Beute in Räuber-Nachwuchs umgesetzt wird. Biologisch: Anteil der Beute-Energie, der zur Reproduktion der Räuber führt. In der Räuber-DGL ist δ typischerweise im Term + δ * Beute * Räuber enthalten, wodurch Räuberpopulation wächst, wenn Beute vorhanden ist.
    - gamma (γ): **Räuber-Sterberate** – natürliche Sterbe-/Verlust-Rate der Räuber (ohne Beute). Dieser Parameter erscheint als - γ * Räuber in der Differentialgleichung und sorgt für das Absterben der Räuberpopulation in Abwesenheit von Nahrung.
    - phi (φ): **Infektionsrate des Virus** – bestimmt die Rate, mit der ein Virus eine andere Population infiziert. Im erweiterten Modell infiziert der Virus die **Bakterien** (s.u.). Biologisch steht φ für die Kontaktwahrscheinlichkeit zwischen Virus und Wirt (Bakterium) und die Erfolgsrate der Infektion. Rechnerisch geht φ multipliziert mit Virus- und Bakterienanzahl in den Infektionsterm ein.

- Darüber hinaus definiert SimulationConfig weitere Parameter für zusätzliche Populationen: z.B. r_b (Reproduktionsrate der Bakterien), mu_b (Sterberate der Bakterien), p_v (Reproduktionsfaktor des Virus bei Infektion), mu_v (Virus-Absterberate), k_ib und k_iv (Effizienz der Immunreaktion gegen Bakterien bzw. Viren), r_i, K_i, mu_i (Parameter für das Wachstum und die Kapazität des Immunfaktors) sowie s_n, c_n (Zu- und Abfluss von Nährstoffen). **Initialbedingungen** für alle Populationen (anfängliche Beute-, Räuber-, Bakterien-, Virus-, Immun- und Nährstoff-Mengen) sind ebenfalls Teil der Konfiguration, ebenso wie Simulationssteuerungsparameter (time_steps, dt für Zeitschritt, integrator-Typ).

- **Differentialgleichungen:** Die Dynamik wird durch 6 gekoppelte ODEs beschrieben – für jede Population/Komponente eine Gleichung. Im Simulator sind diese im Methodenrumpf _derivatives(state) definiert. Das System lautet (auf Basis der Parameter in SimulationConfig):

  1. **Beute (Prey):** $\frac{dP}{dt} = \alpha P - \beta P \cdot R$

     Die Beute wächst mit Rate α und wird durch Räuber mit Rate β vermindert.

  2. **Räuber (Predator):** $\frac{dR}{dt} = \delta P \cdot R - \gamma R$

     Räuber vermehren sich proportional zu gefressener Beute (δ) und sterben mit Rate γ.

  3. **Bakterien:** $\frac{dB}{dt} = r_b , B \cdot N - \mu_b B - \phi V \cdot B - k_{ib} I \cdot B$

     Bakterien wachsen mit Nährstoff N (Wachstumsrate r_b), sterben natürlich (μ_b) und werden durch Virusinfektion ($\phi V B$) sowie Immunreaktion ($k_{ib} I B$) reduziert.

  4. **Virus:** $\frac{dV}{dt} = p_v , (\phi V \cdot B) - \mu_v V - k_{iv} I \cdot V$

     Viren vermehren sich, wenn sie Bakterien infizieren (Term mit $p_v \phi V B$), sterben von selbst (μ_v) und werden durch das Immunsystem bekämpft ($k_{iv} I V$).

  5. **Immunsystem:** $\frac{dI}{dt} = \frac{r_i , I , (B+V)}{K_i + B + V} - \mu_i I$

     Die Immunfaktor/Immunzellen proliferieren in Gegenwart von Bakterien und Viren (Wachstumsrate r_i, saturiert durch K_i) und sterben mit Rate μ_i.

  6. **Nährstoff:** $\frac{dN}{dt} = s_n - c_n , B \cdot N$

     Der Nährstoff wird mit konstanter Rate s_n zugeführt (z.B. Photosynthese) und durch Bakterienkonsum verbraucht ($c_n B N$).

- Dieses erweiterte Modell enthält zwei gekoppelte Subsysteme: (a) **Räuber-Beute** (Gleichungen 1 und 2) und (b) **Bakterien-Virus-Immun-Nährstoff** (Gleichungen 3–

6). Zwischen beiden Subsystemen gibt es im aktuellen Modell **keine direkte Kopplung** – sie laufen parallel. (Eine mögliche Erweiterung wäre, die Beute oder Räuber mit den Bakterien/Nährstoffen interagieren zu lassen, etwa Beute als Pflanzen und Bakterien als Zersetzer im Nährstoffkreislauf. Diese Kopplung ist bislang nicht implementiert, so dass α, β, γ, δ nur Beute/Räuber beeinflussen und φ, etc. nur Virus/Bakterien.)

- **Integrator und Simulationablauf:** Die Klasse Simulator verwendet das oben definierte ODE-System und berechnet es iterativ über die Zeit. Dank des **Strategie-Musters** für Integrationsverfahren kann der Nutzer im Config das Feld integrator setzen ("euler", "rk4" oder "scipy_rk45"). Standard ist ein selbst implementierter Runge-Kutta 4. Ordnung (RK4) für gute Stabilität . Bei jedem Zeitschritt Δt werden die Ableitungen berechnet und der Zustand upgedatet. Die Simulation läuft über time_steps Iterationen oder bis ein Abbruchkriterium greift. Ein eingebauter **Stabilitätswächter** bricht die Schleife ab, falls Zahlenwerte nicht mehr finiten (NaN/Inf) werden – dies deutet auf Instabilitäten (z.B. Blow-Up bei zu großem Δt oder extremen Parametern) hin. In einem solchen Fall wird eine Warnung ausgegeben („Instabilität, Simulation abgebrochen“) und nur die Daten bis zu diesem Zeitpunkt werden zurückgeliefert . Am Ende liefert Simulator.run() ein Dictionary mit Zeitarray und allen Populations-Zeitreihen ("time", "prey", "predator", "bacteria", "virus", "immune", "nutrient"). Zusätzlich werden zwei Konstantreihen "mutation_rate" und "curiosity" hinzugefügt – diese Felder werden aus Kompatibilitätsgründen befüllt (z.B. damit Datenauswertungen immer Spalten für alle möglichen Parameter enthalten). Im Makromodell selbst haben mutation_rate und curiosity **keine Funktion** (sie betreffen nur die ABM, siehe unten).
- **Ausgabe und Visualisierung:** Die Resultate (Zeitreihen) können direkt geplottet werden. In predator_prey_sim/visualizer.py sind Funktionen wie plot_time_series definiert, die wahlweise mit Matplotlib oder Plotly die Verläufe von Beute, Räuber, etc. grafisch darstellen. Im Streamlit-Frontend wird Plotly verwendet, um interaktive Diagramme einzubinden. Typischerweise erkennt man im klassischen Räuber-Beute-Teil **oszillierende Populationen**: Phasen von Beuteüberschuss gefolgt von Räuberanstieg und Beuterückgang, etc. Die Bakterien-Virus-Subsystem kann z.B. gedämpfte Schwingungen oder konvergente Werte zeigen, abhängig von φ und anderen Parametern. Die Validierung an bekannten analytischen Ergebnissen (Lotka-Volterra) zeigt erwartungsgemäß Erhaltung der Phasenverschiebung und Amplituden (sofern Parametrisierung innerhalb stabiler Grenzen liegt).

Mikro-Ebene: ABM-Ökosystem

Die Mikro-Ebene implementiert ein agentenbasiertes Evolutionsszenario in einer räumlichen Umgebung. Dieser Teil ist im Paket agent_ecosystem gekapselt und besteht aus den Komponenten Agent, Environment und AgentSimulator, konfigurierbar über AgentSimConfig. Im Gegensatz zur Makro-Ebene, die Differenzialgleichungen für Populationsdichten nutzt, werden hier **einzelne Individuen** simuliert, die auf einer Gitterwelt interagieren.

- **Konfiguration (AgentSimConfig):** Die Datenklasse für die ABM enthält Parameter, die Verhalten und Umgebung der Agenten bestimmen:

- grid_size: Größe des quadratischen Gitters (Standard 50x50 Zellen). Das Gitter ist toroidal (Randlos, d.h. Agenten laufen am gegenüberliegenden Rand wieder rein).
- initial_agents: Anzahl der anfangs erzeugten Agenten (Standard 20). Diese werden zufällig auf dem Grid positioniert.
- max_age: Maximale Lebensdauer eines Agenten (in Zeitschritten). Erreicht ein Agent dieses Alter, stirbt er.
- energy_gain: Energiemenge, die ein Agent maximal aus einer Ressource ziehen kann (pro Schritt). Auch die Anfangsenergie neuer Agenten wird hierüber bestimmt (Standard initial energy = 2 * energy_gain) .
- move_cost: Energiekosten pro Bewegungsschritt (multipliziert mit der Geschwindigkeit des Agenten). Bewegen ist somit umso teurer, je schneller ein Agent ist.
- reproduction_threshold: Energieschwelle, ab der ein Agent sich teilt/fortpflanzt. Wenn die Energie eines Agenten diesen Wert erreicht oder überschreitet, kann er sich reproduzieren; dabei wird Energie auf das Kind übertragen (siehe unten).
- mutation_rate: Mutationsstärke für die Reproduktion. Dieser Wert steuert die Standardabweichung der zufälligen Änderung, die den Kind-Attributen mitgegeben wird. Biologisch entspricht dies der Variabilität der Nachkommen – höheres mutation_rate erzeugt mehr Streuung in den Eigenschaften, was evolutionäre Experimente ermöglicht. Rechnerisch wird für jedes vererbbare Attribut ein Faktor aus einer Normalverteilung $N(1, \sigma=\text{mutation_rate})$ gezogen und der Eigenschaftswert des Elternteils damit multipliziert . So entstehen leichte Abweichungen (Mutation) vom Elternwert; negative Werte werden auf 0 gekappt (kein Attribut kann negativ werden).
- resource_regen & max_resource: Steuerung des Umweltressourcen-Vorkommens. Jede Grid-Zelle hat einen gewissen **Ressourcenwert** (z.B. Nahrung). Pro Zeitschritt regenerieren die Ressourcen um resource_regen Einheiten bis maximal max_resource (überschüssige Regeneration wird gecappt) . Standardmäßig startet jede Zelle mit der Hälfte von max_resource gefüllt.
- **Neugier (curiosity)**: Initialer Neugier-Wert der Agenten (Standard 0.5). **Neugier** steht hier für das Verhalten eines Agenten, entweder bekannte Nahrungsquellen gezielt anzusteuern oder spontan zu explorieren. Biologisch könnte man Neugier als Risikofreude oder Erkundungsdrang interpretieren – ein neugieriger Organismus verbringt mehr Zeit mit Umherstreifen statt zielstrebig dem nächsten Futter nachzugehen. Rechnerisch ist curiosity als **Wahrscheinlichkeitsfaktor** im Bewegungsentscheidungsprozess implementiert. In jedem Zeitschritt entscheidet der Agent basierend auf curiosity, ob er sich **explorativ** bewegt oder **zielgerichtet** zu einer Ressource geht: Ein **kleiner Neugierwert** bedeutet, dass der Agent meistens (mit hoher Wahrscheinlichkeit) direkt auf die nächstgelegene Ressource zuläuft, während ein **hoher Neugierwert** bedeutet, dass der Agent häufig zufällige Bewegungen unternimmt, selbst wenn in Reichweite Nahrung sichtbar ist. Konkret: Wenn der Agent in seiner Sichtweite Ressourcen findet, so wählt er mit Wahrscheinlichkeit $(1 - \text{curiosity})$ die nächste Ressourcenzelle als Ziel; mit Wahrscheinlichkeit curiosity hingegen ignoriert er das vorhandene Futter und bewegt sich in eine zufällige Richtung . Dieser Mechanismus erlaubt eine Population mit unterschiedlichen Neugierwerten, die Balance

zwischen **Ausnutzung** (Exploitation) und **Erkundung** (Exploration) der Umgebung zu regulieren.

- **Agenten (Agent):** Die Klasse Agent repräsentiert ein einzelnes Lebewesen mit verschiedenen **Eigenschaften** (Attributen). Implementierungsdetails:
  - Es werden __slots__ verwendet, um Speicheroverhead pro Agent zu reduzieren (wichtig bei vielen Agenten). Die Attribute umfassen Position (x, y auf dem Grid), Bewegungs- und Sinnesfähigkeiten (speed für Bewegungsgeschwindigkeit, sight für Sichtradius in Zellen), strength (Stärke/Effizienz beim Fressen), camouflage (Tarnung – derzeit im Modell nicht aktiv genutzt mangels Fressfeinden), curiosity (Neugier, wie oben beschrieben), sowie interne Zustände energy (Energielevel) und age (Alter in Steps) . Jeder Agent hält zudem eine Referenz auf die globale Config, um Parameter wie move_cost, reproduction_threshold etc. nutzen zu können.
  - **Wahrnehmung (perceive):** Der Agent kann Ressourcen in seiner Umgebung innerhalb seiner sight-Reichweite wahrnehmen. Die Methode perceive(env) gibt eine Liste der Koordinaten aller benachbarten Zellen zurück, die noch Ressourcen > 0 haben . (Damit "sieht" der Agent Nahrung in der Nähe.)
  - **Entscheidung und Bewegung (decide_move & move):** Basierend auf den wahrgenommenen Ressourcen entscheidet der Agent ein Ziel für den nächsten Schritt. In decide_move(resources) wird, wie oben erläutert, mit Hilfe der curiosity entweder der nächstgelegene Ressourcenzelle ausgewählt (falls resources nicht leer ist und ein Zufallswert > curiosity liegt) oder ein zufälliger Richtungsvektor generiert . Dieses Zufallsgehen erfolgt, indem ein zufälliger Winkel gewählt wird und entsprechend der speed ein Schritt in diese Richtung berechnet wird. Die resultierenden Zielkoordinaten werden modulo grid_size genommen, sodass der Agent ggf. am Rand "wrap-around" macht. Die Bewegung selbst wird in move(tx, ty) vollzogen: der Agent bekommt die neuen Koordinaten zugewiesen und zahlt die Bewegungskosten in Form von Energieabzug (energy -= move_cost * speed) .
  - **Nahrung aufnehmen (eat):** Nachdem der Agent an seinem neuen Ort steht, nimmt er die Ressource der Zelle zu sich. Pro Step kann er maximal energy_gain Energie aufnehmen, überschüssige Ressource bleibt liegen. Die Ressource der Umweltzelle wird entsprechend reduziert, und der Agent erhöht seine energy um den aufgenommenen Betrag, modifiziert durch seine strength . (Stärke >1 bedeutet er kann effizienter Energie aus der Ressource ziehen, <1 ineffizienter.)
  - **Reproduktion (reproduce):** Wenn der Agent nach Bewegung und Fressen mehr Energie besitzt als reproduction_threshold, so vermehrt er sich ungeschlechtlich. Die Methode reproduce() erzeugt einen neuen Agent an derselben Position . Das Elternteil teilt seine Energie mit dem Kind (halbiert die eigene Energie und gibt die Hälfte dem Nachwuchs). Die **Eigenschaften des Kindes** werden basierend auf den Elternwerten mit **Mutation** festgelegt: für jedes vererbte Attribut (speed, sight, strength, camouflage, curiosity) wird der Elternwert mit einem Faktor multipliziert, der normalverteilt um 1 liegt (σ = mutation_rate) . Dadurch entstehen kleine zufällige Abweichungen – dies ist die einzige Quelle genetischer Variabilität im System, analog zu Mutationen in der biologischen Evolution. Es gibt keine direkte Rekombination (nur ein Elternteil). Der neue Agent startet mit der übergebenen halben Energie.
  - **Lebenszyklus-Schritt (step):** Dieser Prozess – Wahrnehmen, Entscheiden, Bewegen, Fressen, evtl. Reproduzieren – ist in Agent.step(env) gekapselt . Zusätzlich erhöht sich das Alter jedes Mal um 1; erreicht es max_age oder

sinkt energy auf 0, so liefert step den Wert None zurück, was signalisiert, dass der Agent stirbt und aus der Simulation entfernt wird . Überlebt der Agent und vermehrt sich **nicht**, gibt step den Agenten selbst zurück; vermehrt er sich, so gibt step das neue Agent-Objekt (Kind) zurück und der Eltern-Agent bleibt ebenfalls existent. (Das Simulationstreibwerk sorgt dafür, dass in diesem Fall beide in der nächsten Iteration dabei sind.)

- **Umgebung (Environment):** Die Environment-Klasse verwaltet die **Ressourcenmatrix** resources . Diese ist ein 2D-Numpy-Array der Größe grid_size x grid_size, initial gefüllt mit max_resource/2 in jeder Zelle. In jedem Simulationsschritt wird Environment.regenerate() aufgerufen, um Ressourcen aufzubauen: jede Zelle erhält +resource_regen (z.B. 1.0 Nahrungseinheit pro Zeitschritt), gedeckelt bei max_resource. Dadurch gibt es ein dynamisches Ressourcenfeld, aus dem Agenten schöpfen, das sich aber auch wieder regeneriert (ähnlich wie nachwachsendes Futter in einer Wiese).
- **Simulationablauf (AgentSimulator):** Die Klasse AgentSimulator orchestriert die Iteration über alle Agenten pro Zeitschritt. Im Konstruktor werden gemäß initial_agents die Agent-Objekte erstellt und in einer Liste agents gespeichert, wobei jeder Agent den curiosity-Wert aus der Config bekommt . In der Hauptschleife von AgentSimulator.run() wird für jeden Zeitschritt:

  1. env.regenerate() aufgerufen (Ressourcen wachsen nach).
  2. Für jeden Agent in der aktuellen Liste agents wird agent.step(env) ausgeführt. Die Rückgabewerte (entweder derselbe Agent falls überlebt, ein neues Agent-Objekt falls reproduziert, oder None falls gestorben) werden ausgewertet. Es wird eine neue Liste new_agents aufgebaut: stirbt ein Agent, wird er nicht übernommen; überlebt er, wird er wieder hinzugefügt; reproduziert er, werden sowohl er als auch sein Kind in die neue Liste aufgenommen . Nach Durchlauf aller alten Agenten wird agents = new_agents. Damit ist der Geburten/Todes-Zyklus vollzogen.
  3. Im gleichen Loop werden **Statistiken** gesammelt: pop[t] = Anzahl der Agenten nach dem Schritt t ; avg_speed[t] = Durchschnittliche Geschwindigkeit aller lebenden Agenten; avg_curiosity[t] = Durchschnittliche Neugier aller lebenden Agenten. Diese Kennzahlen geben Hinweise auf den evolutionären Trend. Zum Beispiel könnte man beobachten, dass über viele Generationen hinweg die durchschnittliche Geschwindigkeit steigt, falls schnelle Agenten einen Selektionsvorteil haben, oder dass die Neugier sinkt, falls zielstrebiges Fressen vorteilhaft ist – abhängig vom Ressourcenangebot.

  Die Simulation läuft für time_steps Iterationen. Das Ergebnis ist ein Dictionary mit den Arrays "time" (0 bis time_steps-1), "population", "avg_speed", "avg_curiosity" . Diese Zeitreihen geben Aufschluss über Populationsdynamik und evolutionäre Veränderung von Eigenschaften.

- **Charakteristik und Verhalten:** Anders als das deterministische ODE-Modell ist die ABM **stochastisch** – insbesondere durch Zufallsbewegungen (Neugier) und Mutation. Jede Runs kann etwas andere Ergebnisse liefern. Typischerweise wird man feststellen, dass die Population zu Beginn schwankt, sich aber in vielen Fällen um einen **attraktiven Bereich** einpendelt (eine Art dynamisches Gleichgewicht, abhängig von Ressourcenregeneration und Energieparametern). Durch Mutation und Selektion können sich die Durchschnittswerte der Traits ändern. Beispielsweise: Bei knappem Nahrungsangebot könnten Agenten mit größerer sight oder höherer speed überleben,

wodurch diese Eigenschaften im Durchschnitt zunehmen. Hohe curiosity kann bei reichlich verteilten Ressourcen ein Vorteil sein (weil neue Gebiete schneller erschlossen werden), während bei spärlicher Ressource geringe Neugier besser ist (Agenten verschwenden keine Zeit mit ziellosem Wandern). Das Modell erlaubt solche Experimente. Allerdings gibt es im aktuellen ABM **keine Räuber-Organismen** – die Agenten konkurrieren nur indirekt um Ressourcen, fressen sich aber nicht gegenseitig. Auch der camouflage-Wert hat deshalb noch keine Wirkung (er wäre relevant, wenn es räuberische Agenten gäbe, die getäuschte Beute schwerer finden). Diese Aspekte könnten in Zukunft ergänzt werden (siehe Ausblick).

Datenflüsse

Ein zentrales Element von Eco Simulator 2.0 ist das **Zusammenspiel der verschiedenen Komponenten** und der Fluss von Daten zwischen ihnen. Im Betrieb fließen Parameter, Simulationsdaten und Auswertungen durch mehrere Stufen: von der Benutzereingabe über die Simulation zum Output, und weiter zur Speicherung, Analyse und wieder zurück zur UI via ML-Modell.

Wesentliche Datenflüsse sind:

- **Eingabeparameter → Simulation:** Der/die Nutzer:in wählt in der Streamlit UI Parameter (z.B. Schieberegler für α, β, φ, Initialpopulationen oder für initial_agents, etc.). Beim Start einer Simulation werden diese Werte an das entsprechende Config-Objekt (SimulationConfig oder AgentSimConfig) übergeben und ein Simulator instanziiert. Die UI startet dann sim.run(), wodurch entweder das ODE-System oder das ABM durchlaufen wird.
- **Simulation → UI-Visualisierung:** Die Simulation liefert Zeitreihen-Daten (als Python-Dictionary bzw. Pandas DataFrame). Diese werden unmittelbar in der UI visualisiert. Im Makro-Tab werden beispielsweise die Kurven für Beute, Räuber, etc. angezeigt, typischerweise als Liniendiagramm über der Simulationszeit. Im Mikro-Tab sieht man den Verlauf der Agentenpopulation sowie Kurven für den Durchschnitt der Eigenschaften (Speed, Curiosity). So erhält der Nutzer direktes Feedback auf die gewählten Parameter.
- **Simulation → Berichte & Analyse:** Für vertiefte Analysen können Simulationen auch **in Serie** ausgeführt werden. Das Modul sensitivity_engine bzw. der Tool-Skript run_sensitivity.py erlaubt es, mehrere Parameterkombinationen automatisiert zu simulieren (Batch Run). Beispielsweise könnte man φ und β systematisch variieren, um deren Einfluss auf die Virus-Endpopulation zu kartieren. Solche Durchläufe schreiben ihre Resultate als CSV-Dateien in das Verzeichnis sensitivity_results/. Die Dateien enthalten die Zeitreihen jeder Simulation; der Dateiname kodiert die Parameter der Simulation (z.B. results_alpha_0.22_beta_0.018_curiosity_0.6_mutation_rate_0.04_phi_0.006.csv enthält die Resultate für α=0.22, β=0.018, etc.). Einzelergebnisse können später geladen und analysiert werden. Typischerweise interessiert man sich vor allem für Endwerte oder Extremwerte – z.B. die Populationsgrößen am Ende der Simulation als

Funktion der Parameter. Das **Report-Engine**-Modul (report_engine.py und ggf. report_engine_agents.py) fasst solche Sensitivitätsergebnisse zusammen: es lädt mehrere CSVs, extrahiert relevante Kennzahlen und erstellt daraus z.B. **Heatmaps** und **Korrelationsdiagramme**. Eine Heatmap könnte z.B. zeigen, wie die Virus-Endpopulation (Farbskala) in Abhängigkeit von β (x-Achse) und φ (y-Achse) variiert . Bereiche instabiler Dynamik oder Aussterben würden sich farblich abzeichnen. Ebenso lassen sich „kritische" Parameterbereiche identifizieren (in denen kleine Änderungen große Effektunterschiede haben). Diese Analysen können als fertiger Bericht ausgegeben werden – mittels eines HTML-Templates werden Diagramme und Text in ein Dokument gegossen (templates/report_template.html). Der/die Benutzer:in kann so nach einer Parameterstudie einen umfassenden Report erhalten, ohne manuelle Nacharbeit.

- **Simulation → ML-Modelltraining:** Ein weiterer Datenfluss nutzt die gesammelten Simulationsergebnisse zum **Training von Machine-Learning-Modellen** (Random Forest Regressoren). Das Skript ml_predictor.py enthält Funktionen train_predator_model() und train_agent_model(). Ersteres durchsucht den Ordner sensitivity_results nach Dateien results_*.csv, extrahiert für jede Simulation deren Eingabeparameter und relevante Ausgabe (hier: Virus-Endpopulation) . Die Daten werden tabellarisch gesammelt, sodass zu jeweils (α, β, φ) der entsprechende Endwert der Virus-Population vorliegt. Darauf wird ein RandomForestRegressor mit 100 Bäumen trainiert , der Lernen soll, die nichtlineare Beziehung abzubilden. Das fertige Modell wird in einer Pickle-Datei (predator_model.pkl o.Ä.) gespeichert. Analog dazu nimmt train_agent_model() Ergebnisse aus Agentensimulationen (Dateien in agent_results/) und trainiert einen Random Forest, der aus den Prädiktoren *Zeit* und *Population* die Zielgröße *durchschnittliche Geschwindigkeit* vorhersagt . Hierdurch entsteht ein Modell, das z.B. schätzen kann, wie schnell die Agenten nach einer gewissen Simulationsdauer bei gegebener Populationsgröße durchschnittlich sind – im Grunde wird dadurch eine Art *Trendabschätzung* für die Evolution gelernt. Diese ML-Modelle komprimieren umfangreiche Simulationsdaten in handhabbare Formeln.
- **ML-Modell → UI-Vorhersage:** Sobald die Random-Forest-Modelle trainiert und gespeichert sind, können sie im Live-Betrieb genutzt werden. Im Streamlit-UI wird nach Auswahl der Parameter **vor Ausführung der eigentlichen Simulation** bereits eine ML-Vorhersage angezeigt. Beispielsweise: Im Makro-Tab, wenn der Nutzer α, β, φ eingestellt hat, berechnet ml_predictor.predict(alpha, beta, phi) mit dem geladenen Modell sofort eine Schätzung für die resultierende Virus-Population und zeigt diese als Info-Meldung an ("ML-Vorhersage – Virus-Endpopulation: X") . Dies gibt dem Nutzer eine schnelle Orientierung, was ungefähr zu erwarten ist, ohne die (vielleicht längere) Simulation abwarten zu müssen. Gleiches gilt im Agenten-Tab: Hier könnte predict_agent(time, population) genutzt werden, um z.B. vorherzusagen, wie hoch die durchschnittliche Geschwindigkeit nach einer bestimmten Simulationsdauer sein wird (falls man z.B. nur Teil-Ergebnisse hat). Die UI nutzt diese Vorhersagen unterstützend. Natürlich können die ML-Schätzer reale Simulationen nicht ersetzen, aber sie sind aus den Daten gelernt und typischerweise recht akkurat innerhalb des geübten Parameterbereichs.
- **UI Dashboard & Persistenz:** Das Frontend aggregiert auch **über mehrere Läufe hinweg** Daten. So wird beispielsweise im **Dashboard** (Startseite der App) die Gesamtzahl durchgeführter Simulationen angezeigt, wie viele davon kritisch (abgebrochen/instabil) waren, und Durchschnittswerte über alle Läufe, z.B. durchschnittliche Beutepopulation, Durchschnitt der Mutation Rate und Neugier über alle Agentensimulationen (siehe Abbildung unten). Diese Werte werden vermutlich aus den gespeicherten Ergebnissen berechnet. Außerdem kann eine Korrelationmatrix

oder -Heatmap dargestellt werden, welche z.B. die statistische Korrelation zwischen Parametern und Outcomes zeigt (daher “Korrelation Heatmap”). Dieser Teil der UI bietet einen höheren Überblick über die Systemdynamik jenseits einzelner Simulationen.

*Abb.: Streamlit **Ökosystem Kontrollzentrum 2.0** (Dashboard). Oben werden aggregierte Kennzahlen früherer Simulationen gezeigt (Anzahl Simulationen, davon kritische, durchschn. Beute-Population, Mutation Rate, Curiosity). Unten ist Platz für eine **Korrelation-Heatmap** (z.B. Parameter vs. Outcomes Korrelation) sowie eine Sektion “Simulationsergebnisse”. Dieses Dashboard hilft, globale Trends und Zusammenhänge auf einen Blick zu erkennen.*

Zusammengefasst sorgen die genannten Datenflüsse dafür, dass **Eingaben, Simulation, Auswertung, Vorhersage und Visualisierung** nahtlos ineinandergreifen. Entwickler:innen können diese Pipeline anpassen oder erweitern – z.B. weitere Metriken im Dashboard aufnehmen, neue ML-Modelle trainieren (etwa zur Vorhersage der Räuber-Peak-Population) oder zusätzliche Auswertungen in Berichten integrieren.

Codeverweise (Projektstruktur & Module)

Eco Simulator 2.0 ist in einer modularen Python-Projektstruktur organisiert. Nachfolgend wird die Verzeichnisstruktur skizziert und die wichtigsten Dateien/Module kurz erläutert, um Entwicklern eine Orientierung zu geben:

```
dersmartcontract-eco_simulator/
├── app.py                  # Hauptprogramm: Streamlit Anwendung (UI-Logik)
├── ml_predictor.py         # ML-Integration: RandomForest Modelle
trainieren/laden/vorhersagen
├── report_engine.py        # Automatische Berichtserstellung für Makro-
Simulationen
├── report_engine_agents.py # Berichtserstellung für Agenten-Simulationen
(falls separater Bedarf)
├── sensitivity_engine.py   # Logik für Sensitivitätsanalysen (Batch Runs)
├── Dockerfile              # Docker-Image Definition für Deployment
├── pyproject.toml          # Projektmetadaten und Abhängigkeiten (Poetry)
├── requirements.txt        # Liste der Python-Abhängigkeiten (alternativ
zu pyproject.toml)
├── README.md               # Projektbeschreibung und Kurzanleitung
│
├── agent_ecosystem/        # Paket für die agentenbasierte Mikro-
Simulation
│   ├── __init__.py
│   ├── agent.py            # Definition der Agent-Klasse (Verhalten,
Fortpflanzung etc.)
│   ├── environment.py      # Definition der Environment-Klasse
(Ressourcengrid)
│   ├── simulator.py        # AgentSimulator, der die Hauptschleife und
Statistik führt
```

```
│   └── config.py           # AgentSimConfig mit allen Parametern der ABM
│
├── predator_prey_sim/      # Paket für die ODE-basierte Makro-Simulation
│   ├── __init__.py
│   ├── config.py           # SimulationConfig mit Parametern für
Räuber/Beute/Virus/etc.
│   ├── integrators.py      # Implementierung von Integratoren (Euler, RK4,
SciPy RK45)
│   ├── simulator.py        # Simulator-Klasse, führt ODE-Simulation aus
und liefert Ergebnisse
│   └── visualizer.py       # Plot-Funktionen für Zeitreihen
(matplotlib/plotly)
│
├── sensitivity_results/    # Verzeichnis, das CSV-Ergebnisse von Batch-
Simulationen aufnimmt
│   ├── results_...csv      # (verschiedene Dateien pro Paramkombination,
siehe Datenflüsse)
│   └── ...
│
├── templates/
│   └── report_template.html # HTML-Template für Berichts-PDF/Seite
│
├── tests/
│   └── test_simulator.py   # Unit-Tests (z.B. Prüfung, ob
Simulationsergebnisse plausible Werte liefern)
│
└── tools/                  # Hilfsskripte für Analysen und Steuerung
    ├── analyze_results.py  # Skript zur Auswertung der Sensitivitäts-CSV
(Statistiken, Korrelationen)
    ├── control_dashboard.py# (Evtl. generiert/steuert das Dashboard, z.B.
Aggregation der Kennzahlen)
    ├── full_analysis.py    # führt ggf. umfangreichere Analysen/Reihen aus
(Makro + Mikro gemeinsam?)
    ├── plot_results.py     # erzeugt Plot-Grafiken aus CSV-Ergebnissen
(z.B. Trends oder Heatmaps)
    ├── recommend_params.py # Analysiert Ergebnisse, um
Parameterempfehlungen abzuleiten (z.B. stabile Bereiche finden)
    └── run_sensitivity.py  # Skript zum Starten von Sensitivitäts-
Batchruns (variiert Parameter automatisiert)
```

Einige wichtige Module im Detail:

- **app.py:** Enthält die Streamlit-App. Hier werden Tabs für die unterschiedlichen Simulationen definiert. Jedes Tab bietet UI-Elemente (Slider, Buttons) für die Parameter. Durch Klick auf „Simulation starten“ (vermutlich ein Button) wird die entsprechende Simulation angestoßen: z.B. PpSimulator = predator_prey_sim.simulator.Simulator, mit parametrierter Config. Ergebnisse werden per st.plotly_chart dargestellt. Außerdem wird im Makro-Tab nach Parameteränderung die ML-Vorhersage via ml_predictor.predict() angezeigt . Im Agenten-Tab werden ebenfalls Parameter gesetzt (Startpopulation etc.) und dann der AgentSimulator ausgeführt, dessen Ergebnisse geplottet werden (Beispiel: ein Plotly-Scatter für Population und weitere für Ø Speed und Ø Curiosity) . Das App-Modul ist relativ high-level: es koordiniert die Interaktion zwischen Benutzer und den Backend-Funktionen.
- **predator_prey_sim/**: Das Makro-Simulationspaket, wie im Architekturteil beschrieben. config.py mit SimulationConfig, simulator.py mit Simulator (inkl. _derivatives() und run()), integrators.py mit Implementationen der Integrationsmethoden (externe Abhängigkeit: SciPy für das Dormand-Prince RK45).

Die Parameter in SimulationConfig sind so benannt, dass sie biologisch Sinn ergeben (griechische Buchstaben für Kern-ODE-Parameter, andere für Erweiterungen). Bei Erweiterungen des Modells (neue Gleichungen) sollte entsprechend SimulationConfig, _derivatives und Ausgabe-Arrays angepasst werden. visualizer.py bietet mit plot_time_series eine bequeme Möglichkeit, die Ergebniskurven darzustellen – es nutzt intern sowohl Matplotlib (für PDF-Berichte geeignet) als auch Plotly (für Web) je nach Parameter. Entwickler:innen können diese Visualisierung erweitern, z.B. zusätzliche Plots (etwa Phasenraumdarstellungen Räuber vs. Beute) implementieren.

- **agent_ecosystem/**: Das Mikro-Simulationspaket, bereits ausführlich erläutert. Hier seien nochmals Erweiterungspunkte genannt: Die Agenten-Klasse kann um weitere Eigenschaften ergänzt werden. Durch die Verwendung von __slots__ muss ein neues Attribut dort eingetragen werden, ebenso sollte es in reproduce() berücksichtigt werden (damit es mutiert vererbt wird) . Auch decide_move könnte um Logik erweitert werden, falls man z.B. soziale Interaktionen einführen möchte (etwa Agent folgt der Spur anderer). Gegenwärtig findet **keine direkte Interaktion zwischen Agenten** statt außer der Konkurrenz um dieselbe Ressource. Es wäre möglich, z.B. Rudelverhalten, Kämpfe oder Nahrungsketten (ein Teil der Agenten als „Räuber", die andere Agenten jagen) zu implementieren – das erfordert umfangreichere Änderungen in Agent.step und Umgebung. Die Umgebung könnte ebenso erweitert werden (mehrere Ressourcentypen, Hindernisse auf dem Grid, etc.). Der AgentSimulator ist generisch genug, dass zusätzliche Regeln pro Zeitschritt (z.B. saisonale Änderungen der Ressourcen oder Katastrophen) eingefügt werden könnten. Wichtig: Die Simulationsschleife sammelt derzeit nur einige Durchschnittswerte; falls weitere Metriken gewünscht sind (etwa durchschnittliche Energie der Population, Diversität der Genetik, etc.), kann man analog in run() zusätzliche Arrays füllen und ausgeben.
- **ml_predictor.py:** Dieses Modul vereint das Laden, Trainieren und Verwenden der ML-Modelle. Es definiert Konstanten für die Modellpfade (PREDATOR_MODEL_PATH, AGENT_MODEL_PATH) und die oben beschriebenen Funktionen predict(...), predict_agent(...) sowie die Trainingsfunktionen . Entwickler:innen können hier weitere Prädiktionsmethoden ergänzen. Beispielsweise könnte man ein neuronales Netz einbinden oder eine Klassifikation (z.B. ob ein Run stabil/instabil endet) trainieren. Wichtig ist, dass die Trainingsdaten korrekt gesammelt werden (die jetzigen Funktionen nehmen an, dass alle CSVs gleiche Struktur haben). Die Random Forests sind mit Standardparametern initialisiert; für genauere Modelle könnte man die Hyperparameter anpassen oder das Feature-Set erweitern (z.B. nicht nur α, β, φ verwenden, sondern auch initiale Populationsgrößen oder Zwischenwerte).
- **report_engine.py & report_engine_agents.py:** Diese Module generieren ausführliche **Berichte**. report_engine.py fokussiert sich vermutlich auf die Makro-Simulation: es könnte die Sensitivitätsergebnisse laden und Diagramme wie Heatmaps oder Liniendiagramme erzeugen (mittels Matplotlib/Seaborn). Tatsächlich findet man Verwendungen von df.pivot und Plot-Funktionen , was auf die Erstellung von Heatmaps hindeutet. Ebenso wird aus den Daten eine HTML-Seite geformt (durch Einfügen von Grafiken und Text in report_template.html). report_engine_agents.py macht Vergleichbares für ABM-Ergebnisse. Gemeinsam ermöglichen diese Module, dass ein Nutzer nach einer Simulation oder Parameterstudie einen automatisch generierten Report erhält, der möglicherweise Kennzahlen wie Durchschnittswerte, Maxima/Minima, Korrelationen und Handlungsempfehlungen (z.B. "Erhöhen Sie α, um das Aussterben der Räuber zu vermeiden") enthält. Die Klasse oder Funktionen in diesen Modulen können vom UI via Buttons getriggert werden (z.B. "Bericht erstellen" Button). Die Module zeigen gutes Beispiel, wie man aus rohen Simulationsdaten sinnvolle Größen berechnet. Entwickler können diese Reports leicht

anpassen (Template-Änderung) oder weitere Analysen einbauen (z.B. statistische Tests über die Runs).

- **sensitivity_engine.py:** Dieses Modul koordiniert die **Batch-Simulationen**. Es stellt Funktionen bereit, um über Gitter von Parameterwerten zu iterieren, jeweils eine Simulation auszuführen und die Ergebnisse abzuspeichern. Man kann hier definieren, welche Parameter variiert werden sollen, in welchen Schrittweiten, etc. Intern wird Simulator.run() (Makro) oder AgentSimulator.run() (Mikro) einfach mehrfach aufgerufen. Performance-kritisch ist dabei die Anzahl der Durchläufe; größere Parameter-Sweeps können durch die ODE-Komponente relativ schnell erledigt werden, während ABM-Sweeps rechenintensiver sind. In sensitivity_engine.py könnte auch eine parallelisierte Ausführung implementiert sein (z.B. mit multiprocessing, um mehrere Kerne zu nutzen).
- **tools/**: Verschiedene Hilfsskripte, die entweder eigenständig ausführbar sind oder als Utility-Funktionen von obigen Modulen genutzt werden.
    - run_sensitivity.py dürfte ein CLI-Skript sein, um ohne UI Batchruns auszuführen (beispielsweise für Offline-Berechnungen). Möglicherweise nimmt es Parameter aus der Kommandozeile oder nutzt vordefinierte Gitter.
    - analyze_results.py könnte nachträglich auf sensitivity_results zugreifen und statistische Analysen durchführen (Korrelation berechnen, beste/worst Parameter finden etc.). Eventuell erzeugt es auch die "Korrelation Heatmap" im Dashboard.
    - recommend_params.py klingt so, als ob hier Heuristiken implementiert sind, um dem Nutzer Parameterempfehlungen zu geben. Beispielsweise könnte man definieren: "kritisch" ist, wenn Viruspopulation am Ende > X (Überhandnehmen der Krankheit) oder Räuberpopulation < Y (Aussterben der Räuber). Das Skript könnte durch die Parametervariationen gehen und die Kombinationen identifizieren, die optimale Ergebnisse liefern (z.B. stabile Koexistenz aller Spezies). Diese Empfehlungen könnten wiederum im UI angezeigt werden oder Teil des Berichts sein.
    - control_dashboard.py ist vermutlich für das UI-Dashboard zuständig. Es aggregiert die Ergebnisse (z.B. zählt CSV-Dateien in sensitivity_results oder führt eine spezielle Log-Datei über Simulationen). Die Zahlen wie "Gesamt-Simulationen" oder "Durchschn. Mutation Rate" aus obigem Screenshot【18†】 werden wahrscheinlich hier berechnet. Dieses Skript könnte als eigener Thread oder bei App-Start laufen, um dem Dashboard stets aktuelle Infos zu liefern.
    - plot_results.py bietet womöglich einfache Möglichkeiten, aus CSVs Grafiken zu erzeugen (ggf. für Debugging oder falls man das UI umgehen will).
- **tests/test_simulator.py:** Beinhaltet grundlegende Unit-Tests, um sicherzustellen, dass die Simulationsergebnisse konsistent sind. Ein Test prüft z.B., ob die Zeitschritte monotom ansteigen und ob alle erwarteten Keys (Zeit und alle Spezies) im Resultat enthalten sind . Weitere Tests könnten enthalten sein, wie z.B. Laufzeiten in einem Toleranzbereich, oder dass bei bestimmten Parametern bekannte Ergebnisse kommen (z.B. Nullwachstum wenn $\alpha=0$ etc.). Diese Tests helfen beim Refactoring sicherzustellen, dass Kernfunktionalitäten erhalten bleiben.

Mit diesem Überblick sollten Entwickler:innen schnell die relevanten Stellen im Code finden, je nachdem, ob sie an der UI, dem Simulationskern oder der Auswertung etwas ändern

möchten. Die **Projektstruktur** trennt klar die fachlichen Bereiche, was Wartung und Erweiterung begünstigt.

Benchmarks (Performance & Validierung)

Die Performance des Eco Simulator 2.0 ist im Allgemeinen ausreichend für interaktive Nutzung und umfangreiche Analysen, sofern die Parameter im üblichen Bereich bleiben. Hier betrachten wir die **Rechenleistung** sowie einige Aspekte der **Validierung** der Modelle.

Performance und Optimierung

- **Makro-Simulation (ODE):** Das Lösen der Differentialgleichungen erfolgt schrittweise mit Δt, standardmäßig 300 Schritte mit $dt=0.1$. Selbst mit anspruchsvollerem Integrator (RK4) ist dies sehr schnell – auf moderner Hardware dauern einzelne Simulationen nur Millisekunden bis wenige Zehntel Sekunden. Dadurch können auch Batchruns mit hunderten Kombinationen in vernünftiger Zeit durchgeführt werden. Sollte $time_steps$ deutlich erhöht werden (für längere Zeithorizonte) oder die Gleichungen steifer werden (z.B. bei sehr unterschiedlichen Raten), kann man auf den SciPy RK45 Integrator ausweichen, der adaptivere Schrittweiten nutzt. Allerdings ist SciPy's Integrator in Python-Aufruf etwas langsamer pro Schritt; er lohnt sich aber falls die einfacheren Integratoren Instabilitäten zeigen. Die Implementation berücksichtigt ein paar Performance-Tricks: Verwendung von NumPy-Arrays für den Zustand und Vektorisierung bei der Berechnung der Ableitung (alle 6 Gleichungen auf einmal, wobei Python-Schleifen vermieden werden). Das Clamping der state-Werte (Beschränkung ins [0, 1e9]-Intervall) ist eine Sicherheitsmaßnahme, die minimalen Overhead verursacht, aber grobe Ausreißer verhindert. Insgesamt wurde das Makromodell auch mit deutlich mehr Zeitschritten getestet (z.B. 10000 Schritte) – dies skaliert linear, was aufgrund der geringen Dimensionalität kein Problem darstellt.
- **Mikro-Simulation (ABM):** Die agentenbasierte Simulation ist rechenaufwändiger, da pro Zeitschritt *jeder Agent* individuell upgedatet wird. Mit den Standardwerten (20 initiale Agenten, maximal vielleicht ein paar hundert Agenten nach einigen Reproduktionszyklen) läuft die Simulation mit 500 Schritten in wenigen Sekunden. Der zeitkritischste Teil ist die innere Schleife über Agenten in AgentSimulator.run(). Diese ist in Python geschrieben und iteriert serialisiert; d.h. 1000 Agenten über 500 Schritte wären 500k Aufrufe von Agent.step(), was spürbar Zeit kostet (Größenordnung einige zehn Sekunden, abhängig von der Hardware). Bei Bedarf könnte man Optimierungen durchführen: z.B. parallele Verarbeitung der Agenten pro Schritt (da Agentenaktionen unabhängig sind – Vorsicht aber bei gemeinsamem Zugriff auf Env.resources) oder Low-Level-Optimierungen (Numba JIT für Agent.step, oder kritische Teile in Cython). Bisher war dies nicht notwendig, da die Population aufgrund begrenzter Ressourcen auch nicht exponentiell ins Unendliche wächst, sondern sich stabilisiert. Falls man das Modell deutlich größer skaliert (z.B. grid_size >> 50, initial_agents >> 100), sollte man über effizientere Datenstrukturen nachdenken (z.B. Agenten in Arrays packen für Vektorisierung).

- **Speichermanagement:** Durch __slots__ in Agent wird vermieden, dass jeder Agent ein eigenes __dict__ hat – dies spart Speicher, insbesondere bei vielen Agenten. Die Simulationen verwenden ansonsten hauptsächlich Numpy-Arrays, die speicherökonomisch sind. Ein Lauf mit 300 Zeitschritten Makro generiert ein Array 300x6 (float64), was vernachlässigbar ist. ABM generiert 500x3 (float64) + ein paar kleine Listen. Die Sensitivitäts-Batchruns können jedoch potenziell sehr viele Daten auf Festplatte erzeugen: z.B. 1000 Runs à 300 Schritte, 6 Variablen => 1000*300*6 ~ 1.8 Millionen Datenpunkte in CSV. Das ist immer noch moderat (einige MB). Die Tools wie analyze_results.py laden diese ggf. alle in Pandas DataFrames – sollte es größer werden, müsste man evtl. Streaming-Ansätze nutzen.
- **Parallelität:** Derzeit nutzt das Programm standardmäßig nur einen Thread (die UI läuft im Streamlit-Hauptthread, Simulation sequentiell). Python's GIL verhindert echte Parallel-Threads, aber man könnte multiprozess parallelisieren. Bisher ist in run_sensitivity.py keine automatische Parallelisierung implementiert (es sei denn man modifiziert es). Für sehr große Parameterstudien kann ein Entwickler aber leicht die Schleifen mit multiprocessing.Pool parallelisieren.

Wissenschaftliche Validierung

- **Makromodell Korrektheit:** Das Räuber-Beute-Modell wurde qualitativ mit bekannten Resultaten abgeglichen. Für moderate Parameter (z.B. α=0.1, β=0.02, δ=0.01, γ=0.1) entstehen typische Räuber-Beute-Zyklen. In Abwesenheit der Erweiterung (wenn Bakterien/Virus initial 0 sind und φ=0 gesetzt wird, so dass sie keinen Einfluss haben) reduziert sich das Modell genau auf das klassische 2D Lotka-Volterra-System, für das analytisch neutrales Zyklusverhalten bekannt ist. In der Simulation sieht man in diesem Fall geschlossene Umlaufbahnen im Phasenraum (Beute vs. Räuber). Wird γ angepasst, treten stabilere oder divergierende Oszillationen auf, wie erwartet. Die Erweiterung mit Bakterien/Viren/Immunsystem wurde auf Plausibilität geprüft: Es entstehen sinnvolle Dynamiken (Virus kann ohne Bakterien nicht gedeihen, Immune schlägt Virus nieder, etc.). Die genauen Parameter sind fiktiv, da es ein generisches Ökosystem ist, aber man erkennt z.B., dass bei sehr hoher φ (Infektionsrate) die Bakterien rasch eliminiert werden und dadurch letztlich auch der Virus mangels Wirten kollabiert – ein nachvollziehbares Szenario analog zu Krankheit, die ihren Wirt ausrottet.
- **Mikromodell Verhalten:** Das ABM ist schwieriger formal zu validieren, da stochastisch. Hier wurde vor allem auf **qualitative Beobachtung** gesetzt: In Testläufen zeigen sich erwartbare Effekte, z.B.: Ohne Mutation (mutation_rate=0) bleiben die Eigenschaften homogen; die Population überlebt oder stirbt je nach Startwerten, aber es gibt keine Optimierung. Mit Mutation entwickeln sich sichtbar Unterschiede: Wenn Ressourcen knapp sind, setzen sich Individuen mit bestimmten Vorteilen durch (etwa solche, die schneller an weit entfernte Nahrung gelangen). In einigen Testrunden konnte man sehen, dass die durchschnittliche curiosity mit der Zeit sank – was darauf hindeutet, dass in der gegebenen Umgebung eher „vorsichtigere" Agenten (die vorhandene Nahrung sofort nutzen) erfolgreich waren. Dies stimmt mit der Intuition überein. Auch die Populationskurve zeigt logisches Verhalten: Start mit 20 Agenten, rasches Wachstum, bis Ressourcenlimit erreicht wird, dann Pendeln um eine Tragekapazität (die sich aus Ressourcennachwuchs und Energiebedarf ergibt). Falls max_age limitiert ist, kommt zudem eine periodische Generationsfolge ins Spiel.

- **Regressionstests:** Mit den tests/test_simulator.py wurde sichergestellt, dass Kernfunktionen laufen. Bei Änderungen am Code ist es ratsam, vorhandene Simulationsergebnisse mit neuen zu vergleichen (bei gleichen Zufallsseeds). Dafür kann man z.B. die numpy Random-Seed fixieren, um reproduzierbare ABM-Ergebnisse zu bekommen, und dann die Zeitreihen differenzieren. Solche Regressionstests stellen sicher, dass Refaktorierungen (z.B. Performance-Optimierungen) das Verhalten nicht unbeabsichtigt verändern.

Insgesamt ist Eco Simulator 2.0 performant genug für die vorgesehenen Zwecke. Größere Erweiterungen könnten eine Überprüfung der Performance nötig machen, aber das Design (Trennung der Loops, leichte Austauschbarkeit von Integratoren, etc.) erlaubt gezielte Optimierungen. In wissenschaftlicher Hinsicht wurden die Modelle so gewählt, dass sie typische etablierte Verhaltensmuster zeigen, was Vertrauen in die Korrektheit gibt. Selbstverständlich hängt die Aussagekraft der Simulation von sinnvollen Parametereinstellungen ab – es wird empfohlen, das System erst in plausiblen Bereichen zu nutzen und nur vorsichtig in extreme Zonen (wo Instabilitäten auftreten) vorzudringen.

API/Modulübersicht

Im Folgenden werden die wichtigsten Klassen und Funktionen des Projekts mit ihrer **Schnittstelle** (Public API) zusammengefasst. Diese Übersicht dient Entwicklern, die Funktionen aus dem Eco Simulator in eigenen Scripts verwenden wollen oder das Verhalten einzelner Komponenten nachvollziehen möchten.

Makro-API (Lotka-Volterra Simulation)

- **Class predator_prey_sim.config.SimulationConfig** – Konfigurationsdaten der ODE-Simulation. Attribute: alpha, beta, phi, delta, gamma, r_b, mu_b, k_ib, p_v, mu_v, k_iv, r_i, K_i, mu_i (Float-Werte, alle mit Defaults), Initialpopulationen initial_prey, initial_predator, initial_bacteria, initial_virus, initial_immune, initial_nutrient, sowie Simulationsparameter time_steps (int), dt (float Zeitschritt), integrator (Str, "euler"|"rk4"|"scipy_rk45"), show_progress (Bool für evtl. Fortschrittsanzeigen). Die Extended-Parameter mutation_rate, curiosity sind ebenfalls definiert (Default 0.0), werden aber im ODE-Kontext ignoriert. Für Standardnutzung müssen hauptsächlich α, β, φ etc. und die Initialwerte angepasst werden.
- **Class predator_prey_sim.simulator.Simulator** – Hauptklasse, die eine Simulation mit gegebener SimulationConfig ausführt. Wichtige Methoden:
    - Simulator.__init__(cfg: SimulationConfig): Initialisiert den Simulator. Hier wird der Integrator gewählt basierend auf cfg.integrator (Mapping zu Euler, RK4 oder SciPyRK45 Implementierung). Die Config wird in self.cfg gespeichert.
    - Simulator.run() -> dict[str, np.ndarray]: Startet die Zeitschleife und berechnet die ODE-Lösung. Rückgabe ist ein Dictionary mit den Keys: "time", "prey",

"predator", "bacteria", "virus", "immune", "nutrient", sowie "mutation_rate" und "curiosity" (letztere beide als Konstantarrays, damit das Dictionary überall einheitlich Spalten hat). Typischerweise wird diese Funktion aufgerufen, um einen Simulationslauf durchzuführen. Sie wirft keine Exceptions außer im Fall von grober Fehlkonfiguration. Bei Abbruch wegen Instabilität liefert sie verkürzte Arrays zurück (Länge < time_steps).
    - interner Hinweis: Simulator._derivatives(state: np.ndarray) -> np.ndarray berechnet den Ableitungsvektor. Entwickler, die das Makromodell ändern wollen, passen diese Funktion an. Neue Gleichungen müssen auch in diesem Array-Output auftauchen und in SimulationConfig entsprechend Parameter/Initialwerte erhalten.
- **Functions predator_prey_sim.visualizer.plot_time_series(data: dict, backend="matplotlib"| "plotly", save: str|None)** – Generiert aus dem Ergebnisdictionary Plots. Bei backend "matplotlib" wird ein Matplotlib-Figure mit Unterachsen für jede Population erzeugt, bei "plotly" ein interaktives Linienchart (alle Populationen über Zeit in einem Plot). Der Parameter save ermöglicht das Speichern der Grafik in eine Datei (nützlich in Berichten). In der UI wird diese Funktion (oder eine Variante davon) verwendet, um Ergebnisse sichtbar zu machen.

Mikro-API (Agentenbasierte Simulation)

- **Class agent_ecosystem.config.AgentSimConfig** – Konfiguration der ABM. Felder: grid_size (int), initial_agents (int), max_age (int), energy_gain (float), move_cost (float), reproduction_threshold (float), mutation_rate (float), resource_regen (float), max_resource (float), time_steps (int), alpha (float, derzeit ohne Nutzung in ABM), curiosity (float). In der Regel werden grid_size und time_steps sowie evtl. initial_agents bei Bedarf angepasst. alpha existiert hier aus Kompatibilitätsgründen (falls man ABM-Ergebnisse zusammen mit ODE-Parametern auswerten möchte, könnte man dort einen Wert eintragen, doch er hat keine Wirkung auf die Simulation).
- **Class agent_ecosystem.agent.Agent** – Repräsentiert einen einzelnen Agenten. Wichtige Attribute zur Laufzeit: x, y (Position), speed, sight, strength, camouflage, curiosity (Eigenschaften), energy (aktuelle Energie), age (Alter in Steps). Wichtige Methoden:
    - Agent.perceive(env: Environment) -> list[(int,int)]: Liefert Koordinaten von Ressourcenzellen im Umkreis von radius=sight des Agenten (inkl. aktueller Zelle), welche noch Ressource > 0 haben.
    - Agent.decide_move(resources_coords: list[(int,int)]) -> (int, int): Entscheidet anhand der Neugier entweder eine Ressource aus resources_coords (die nächstgelegene) oder random einen Schritt. Gibt Zielkoordinaten für den nächsten Schritt zurück.
    - Agent.move(tx: int, ty: int): Führt den Schritt zur Zielposition aus. Aktualisiert x,y und reduziert Energie um move_cost * speed.
    - Agent.eat(env: Environment): Nimmt an der aktuellen Position so viel Ressource wie möglich (bis energy_gain) aus env.resources. Erhöht die eigene Energie (moduliert durch strength) und verringert die Ressource an der Stelle.
    - Agent.reproduce() -> Agent: Erzeugt einen neuen Agenten (Kind) an gleicher Position, teilt Energie, überträgt mutierte Eigenschaften. Gibt das neue Agent-Objekt zurück.

    - Agent.step(env: Environment) -> Agent|None: Führt einen kompletten Zeitschritt für den Agenten aus (Age++, perceive+decide_move+move+eat, dann falls Energie reicht reproduce). Gibt entweder sich selbst (weiterlebend), ein neues Agent-Objekt (Fortpflanzung ereignete sich) oder None (Agent ist gestorben) zurück. Dies ist die zentrale Methode, die in jeder Iteration für jeden Agenten aufgerufen wird.
- **Class agent_ecosystem.environment.Environment** – Enthält das Nährstoff-Raster. Attribute: resources (numpy.ndarray 2D), config (Referenz auf AgentSimConfig). Methode: Environment.regenerate(): Erhöht in jedem Zellwert resources[x,y] += resource_regen, maximal bis max_resource. Im Prinzip könnte man hier auch Methoden hinzufügen, z.B. um anzuzeigen, wo Ressourcen verteilt sind oder um gezielt Ressourcen-Felder zu platzieren. Standardmäßig startet die Environment homogen halbgefüllt.
- **Class agent_ecosystem.simulator.AgentSimulator** – Verantwortlich für die ABM-Schleife. Konstruktor initialisiert self.env = Environment(cfg) und erzeugt cfg.initial_agents viele Agenten mit zufälligen Positionen und Config (Neugier-Wert aus cfg.curiosity) . Wichtige Methoden:
    - AgentSimulator.run() -> dict[str, np.ndarray]: Führt die Simulation für cfg.time_steps Schritte aus. Der Code durchläuft eine for-Schleife von t=0 bis T-1, in jedem Schritt wird Environment.regenerate() und dann die Agenten-Iteration durchgeführt . Dabei wird eine neue Liste der Agenten erstellt je nachdem ob sie leben oder Nachwuchs bekommen (siehe oben). Anschließend werden Statistik-Arrays pop, avg_speed, avg_curiosity befüllt . Nach der Schleife Rückgabe eines Dict mit "time", "population", "avg_speed", "avg_curiosity". Beachte: Falls keine Agenten mehr leben (Population=0), werden die Durchschnittswerte in dem Schritt nicht definiert sein; im Code ist implementiert, dass dann z.B. avg_speed[t] = 0 oder ausbleibt. Ein Entwickler könnte hier entscheiden, die Simulation auch abzubrechen, wenn alle tot sind – derzeit läuft sie einfach mit 0 Agenten weiter, was aber keine Änderung mehr bewirkt.
- **Hinweis:** Es gibt keine eigene Visualizer-Klasse für ABM. Stattdessen werden die Resultate (Population, avg_speed, avg_curiosity vs. time) analog geplottet. Im report_engine_agents.py könnten ggf. spezielle Darstellungen erfolgen (z.B. Entwicklung der Attributverteilung, falls erfasst). In der UI werden mit Plotly ein Linienchart mit diesen drei Größen erstellt , meist mit gemeinsamer x-Achse (Zeit).

ML-API

- **Function ml_predictor.train_predator_model()** – Lädt alle CSV-Dateien in sensitivity_results/, extrahiert für jeden Durchlauf die Parameter und die Endwerte der Spalten (insbesondere “virus”). Trainiert einen RandomForestRegressor (100 Bäume) mit X = [[alpha, beta, phi], …] und y = [virus_endwert, …]. Speichert das Modell auf Platte (Pfad in PREDATOR_MODEL_PATH, meist "predator_model.pkl"). Sollte vorher schon eine Modelldatei existieren, wird sie überschrieben. Diese Funktion wird typischerweise offline oder bei Bedarf ausgeführt, wenn neue Simulationsergebnisse vorliegen. Sie gibt eine Meldung auf der Konsole aus, wenn das Modell erfolgreich gespeichert wurde .
- **Function ml_predictor.train_agent_model()** – Ähnlich wie oben, aber sucht in agent_results/ nach Dateien (z.B. wenn man ABM-Ergebnisse wegspeichert). Erwartet

in den CSV Spalten “time”, “population”, “speed”. Aggregiert alle Daten und trainiert RandomForestRegressor mit X = [time, population] und y = speed (Ø Speed). Speichert Modell unter AGENT_MODEL_PATH (z.B. "agent_model.pkl"). Diese Funktion erlaubt es, aus vielen ABM-Läufen (auch Zwischenständen aller Zeitschritte) eine Approximationsfunktion für den Evolutionsverlauf zu lernen.

- **Function ml_predictor.predict(alpha: float, beta: float, phi: float) -> float** – Lädt das gespeicherte Räuber-Beute-Modell (falls nicht bereits geladen; im Code wird bei jedem Aufruf neu geladen, was ineffizient ist – ein Verbesserungsansatz wäre, das Modell einmal global zu laden). Gibt die vorhergesagte Virus-Endpopulation (als Float) zurück . Sollte das Modellfile fehlen, wird eine Exception geworfen. In der UI wird darum herum bereits geprüft bzw. dem Nutzer mitgeteilt, dass keine Vorhersage verfügbar ist, falls man noch nicht trainiert hat.
- **Function ml_predictor.predict_predator(alpha: float, beta: float, phi: float) -> float** – Diese ist derzeit nur ein Alias für predict(...) (vermutlich historisch gedacht, um evtl. verschiedene Ausgaben vorherzusagen, aber momentan identisch). Kann in Zukunft angepasst werden, wenn man mehrere Ausgaben aus dem Makromodell vorhersagen will (z.B. Räuber-Maximum, etc.).
- **Function ml_predictor.predict_agent(time: float, population: float) -> float** – Lädt das ABM-Modell und gibt die vorhergesagte durchschnittliche Geschwindigkeit der Agenten zurück . Diese Funktion ist weniger im UI integriert (im Dashboard könnte man damit z.B. prognostizieren, wann die Agentengeschwindigkeit ein Plateau erreicht). Ihre Nützlichkeit hängt davon ab, wie viele ABM-Daten ins Training flossen. Ggf. könnte man hier auch andere Variablen vorhersagen lassen (z.B. durchschnittliche Lebensdauer oder so, falls gemessen).
- **Modell-Details:** Da Random Forests verwendet werden, gibt es keine zusätzlichen Anforderungen an Daten (wie Skalierung). Die Modelle könnten auch aus dem Python-Package scikit-learn heraus mit joblib gespeichert sein. Wichtig ist, dass bei Änderungen in der Datenstruktur (andere Spaltennamen oder zusätzliche Parameter) die Trainingsfunktionen entsprechend angepasst werden müssen (Spaltenfilter). Die ML-API ist relativ einfach gehalten, aber wirkungsvoll zur Approximation.

UI-Interaktion (Streamlit)

Zwar ist die Streamlit-Logik nicht als API für externe Nutzung gedacht, aber es ist hilfreich zu wissen, welche **Callbacks und Events** in app.py vorgesehen sind:

- Es gibt vermutlich zwei Tabs (etwa via st.tabs(["Makro-Simulation", "Agenten-Simulation"])). Im Makro-Tab: st.slider für α, β, φ und initiale Populationen . Evtl. weitere Controls (Checkbox für “Zeige Virus/Bakterien”, etc.). Ein st.button("Start") ruft dann den Simulationslauf auf. Danach st.success("...") und die Plot-Ausgabe . Im Agenten-Tab: Slider für initial_agents (und ggf. weitere wie Mutation und Curiosity – standardmäßig wurden diese nicht als Slider codiert, könnten aber leicht ergänzt werden). Start-Button analog, dann AgentSimulator.run() und Plot-Ausgabe (Plotly Chart mit Population, Speed, Curiosity) .
- Das Dashboard (control_dashboard.py) könnte als separates Page oder als erster Tab realisiert sein. Dort werden keine Simulationen gestartet, sondern vorhandene Daten aggregiert. Es zeigt Kennzahlen mittels st.metric oder einfach st.write mit formatierter

Zahl. Die Korrelation-Heatmap könnte mit Plotly oder Altair dargestellt sein; aus dem Screenshot【18†】 erkennt man farbige Skala rechts (typisch Plotly/Matplotlib). Die "Simulationsergebnisse" Sektion könnte eine Tabelle der letzten N Läufe oder spezielle Kennzahlen auflisten. Dieser Teil interagiert vor allem mit gespeicherten CSVs oder Logs, nicht mit den live Simulationsfunktionen.

- **Entwickler-Hinweis:** Änderungen an den UI-Elementen sollten auf Konsistenz mit dem Rest achten. Die app.py verwendet z.B. PpSimulator als Alias (wahrscheinlich from predator_prey_sim.simulator import Simulator as PpSimulator) und ähnliches für AgentSimulator. Wenn man neue Parameter einführt (z.B. einen Slider für gamma im UI), muss dieser Wert auch in die SimulationConfig übergeben werden. Streamlit cached oft Ausgaben – man sollte vorsichtig sein mit globalen Modellen (der ML-Predictor lädt jedes Mal neu, hier könnte man optimieren, aber dann cachen).
- **UI Layout und Style:** In assets/style.css sind Styles definiert (Hintergrund dunkelblau, Schrift hellgrau etc. für einen modernen Look). Dieser Style wird vom Streamlit app durch st.markdown("<style>...</style>", unsafe_allow_html=True) vermutlich eingebunden. Anpassung daran erlaubt es, das Branding/Look&Feel zu verändern. Die UI ist responsiv im Browser und alle Plots/Widgets passen sich an.

Zusammenfassend bietet Eco Simulator 2.0 nicht nur **interne Klassen und Funktionen**, sondern auch eine interaktive Oberfläche, die diese nutzt. Entwickler:innen können sowohl **programmgesteuert** (über die Python-API der Module) als auch **interaktiv** (über das UI) mit dem System arbeiten. Die Architektur macht es möglich, Teile auch unabhängig zu nutzen – z.B. könnte man predator_prey_sim als eigenes Paket auskoppeln, um nur eine Räuber-Beute-Simulation in einem anderen Kontext laufen zu lassen, oder die ABM-Klasse in einem anderen Projekt zur Evolution verwenden. Dank klarer Trennung und einfachen Schnittstellen (meist mit run()-Methoden) ist die Integration sehr flexibel.

ML-Einbindung (Details & Rolle der ML-Komponente)

Die Integration von Machine Learning in Eco Simulator 2.0 verfolgt das Ziel, **komplexe Simulationsergebnisse prognostizier- und verallgemeinerbar** zu machen. Hier gehen wir näher darauf ein, welche Vorteile das ML-Modul bietet, wie es umgesetzt ist und wie es in den Entwicklungsprozess eingebunden werden kann.

Motivation für ML im Simulator

In ökologischen Simulationen sind manche Fragen schwer direkt zu beantworten, insbesondere wenn mehrere Parameter und nichtlineare Dynamiken im Spiel sind. Beispielsweise: *"Wird in diesem Szenario der Virus langfristig aussterben oder auf einem bestimmten Niveau bleiben?"* oder *"Welche Auswirkung hat eine 10% höhere Mutationsrate*

auf die Evolutionsgeschwindigkeit der Agenten?". Zwar kann man die Simulation selbst laufen lassen, aber wenn viele Kombinationen auszuprobieren sind, wird das zeitaufwendig. Hier kommt ML ins Spiel:

- **Schnelle Approximation:** Ein trainiertes ML-Modell (hier Random Forest) kann innerhalb von Millisekunden eine Vorhersage treffen, anstatt eine Simulation, die Sekunden oder länger dauert, durchführen zu müssen. Das ist für interaktive Anwendungen wichtig (Nutzer bekommt sofort Feedback) und für Metaanalysen (z.B. Optimierungsschleifen, die viele Parameter ausprobieren).
- **Glätten von Rauschen:** Insbesondere das ABM liefert stochastische, rauschbehaftete Outputs. Ein ML-Modell, trainiert auf den Durchschnitt vieler Läufe oder vielen Datenpunkten, kann den *allgemeinen Trend* erfassen und das zufällige Rauschen herausmitteln. So erhält man robustere Abschätzungen.
- **Erkenntnisgewinn:** Die Feature-Importances eines Random Forest oder die Struktur eines einfachen ML-Modells können Hinweise liefern, welche Parameter dominanten Einfluss haben. Zum Beispiel könnte man feststellen, dass in den durchgeführten Sensitivitätsruns φ (Infektionsrate) der entscheidende Faktor für die Virus-Population ist, während α, β im zulässigen Bereich kaum Einfluss haben – weil das ODE-Subsystem entkoppelt war. Solche Einsichten können auch ohne vollständige theoretische Analyse gewonnen werden.

Gewählte ML-Methoden und Modelle

Eco Simulator 2.0 verwendet **Random Forest Regressoren** (aus sklearn.ensemble.RandomForestRegressor). Gründe für diese Wahl:

- Random Forests benötigen kaum Parametertuning und funktionieren gut auf heterogenen Datensätzen. Sie können Nichtlinearitäten und Interaktionen zwischen Input-Parametern abbilden, was hier wichtig ist (z.B. α und β wirken gemeinsam auf die Populationsdynamik).
- Das Training mit den gegebenen Datenmengen (typisch einige Dutzend bis hunderte Samples aus Sensitivitätsanalysen) ist sehr schnell und läuft in-memory.
- Die resultierenden Modelle sind klein (Pickle-Dateien im Kilobyte- bis wenige 100 KB-Bereich) und schnell lad- und evaluierbar.
- Random Forests liefern auch Metriken wie Feature Importance, die man ggf. im Bericht erwähnen kann (z.B. "β erklärt 70% der Varianz der Virusendpopulation").

Natürlich gäbe es Alternativen: lineare Regression (wohl zu simpel für die Dynamiken), SVMs, neuronale Netze usw. Die Entscheidung fiel hier pragmatisch auf Random Forest.

Training und Evaluierung

Trainingsdatenerstellung: Der Workflow sieht vor, dass ein Entwickler oder Power-User zunächst eine Reihe von Simulationen durchführt – idealerweise planvoll als Sensitivitätsanalyse, in der die interessierenden Parameter systematisch variiert werden. Beispielsweise könnte man α, β, φ jeweils auf 5 verschiedenen Niveaus kombinieren (125 Läufe). Diese Ergebnisse liegen dann als CSVs vor. Durch Aufruf von ml_predictor.train_predator_model() werden diese gesammelt. Wichtig: Es wird pro Datei der **letzte Zeitschritt** genommen (df.iloc[-1]) . Das heißt, das Modell lernt insbesondere die **Endbedingungen** (z.B. nach 300 Zeitschritten). Wenn andere Kennzahlen von Interesse wären (z.B. Minimum oder Maximum einer Population, oder Zeitpunkt des Aussterbens), müsste man die Trainingsfunktion entsprechend anpassen, dass jene Größen extrahiert werden.

Training: Das Training selbst ist straightforward: Ein DataFrame mit X und y wird erstellt , dann RandomForestRegressor(n_estimators=100, random_state=42).fit(X, y). 100 Bäume sind in den meisten Fällen ausreichend. Man könnte oob_score=True setzen, um eine Schätzung des allgemeinen Fehlers zu bekommen, was in einer erweiterten Version sinnvoll wäre. Nach dem Training wird das Modell per Pickle gespeichert . Ähnliches beim Agentenmodell: Hier werden alle Datenpunkte aller Zeitschritte aller Läufe genommen (Concat der CSVs) , sodass das Modell auch Zeitabhängigkeit lernt. Die Ausgabe y ist der durchschnittliche Speed. (Warum gerade Speed? Vermutlich, weil Speed eine Schlüsselfähigkeit ist, die sich entwickelt. Man könnte auch Neugier vorhersagen, aber Speed war vielleicht deutlicher.)

Evaluierung: Der Trainingscode selber druckt ein Erfolg-Statement, aber keinen Fehler. Es wäre denkbar, im train_predator_model nach Training z.B. den feature_importances_ des Random Forest auszugeben oder einige vorhergesagte vs. echte Werte zu vergleichen, um das Modell zu validieren. Aktuell muss der Anwender darauf vertrauen oder eigene Tests machen. Wenn man das Predator-Modell evaluieren will, kann man etwa einen Teil der CSVs als Test abspalten. In Praxis sind diese Modelle Hilfsmittel, keine kritischen Entscheidungen – ein geringer Fehler ist tolerierbar.

Integration ins System

Nach dem Training werden die Modelle im UI verwendet, wie bereits beschrieben (Vorhersage-Anzeige). Die Integration erfolgt so, dass falls das Model-File fehlt, entweder nichts angezeigt wird oder eine Warnung kommt, dass noch kein Modell trainiert ist. Es ist davon auszugehen, dass im gelieferten Docker-Image oder Setup bereits ein vortrainiertes Modell enthalten ist, damit Erstnutzer die ML-Funktion sehen ohne sofort Trainingsläufe machen zu müssen.

Relevanz der Parameter: Nochmals zur **Rolle der einzelnen Parameter im ML-Kontext**: Das ML-Modell übersetzt im Grunde die *biologischen* Rollen der Parameter in statistische Zusammenhänge. Etwa lernt es die Auswirkung von α (Beute-Wachstumsrate) auf die Viruspopulation. Im biologischen Modell haben wir jedoch gesehen, dass α gar keinen direkten Einfluss auf den Virus hat, weil die Subsysteme entkoppelt sind. Das ML-Modell wird das merken und vermutlich α als unwichtig einstufen (Feature Importance nahe 0). Dies spiegelt einen wissenschaftlichen Befund wider: Wenn Räuber/Beute unabhängig vom

Bakterien/Virus-System sind, dann beeinflusst die Beutedynamik den Virus nicht. Hätte man jedoch das Modell so erweitert, dass z.B. die Beute über Kadaver die Bakterien nährt, dann würde α indirekt auch Virus beeinflussen, was das ML-Modell erkennen könnte. Solche Überlegungen zeigen, dass das ML-Modell nicht nur mathematisch fitten kann, sondern auch als **Entdeckungswerkzeug** dienen kann, um Verbindungen oder Unabhängigkeiten in komplexen Systemen aufzudecken.

Nutzung durch Entwickler

Entwickler:innen können die ML-Komponente erweitern, zum Beispiel:

- Ein **neues Modell** hinzufügen, etwa um aus den Ergebnissen des ABM gewisse Schwellenwerte vorherzusagen (z.B. "Zeitpunkt, an dem Population das Maximum erreicht"). Dafür könnte man eine neue train-Funktion schreiben, die entsprechende y aus den CSVs extrahiert (z.B. df["population"].idxmax() pro Lauf als Target). Oder auch Klassifikationsaufgaben, wie "Ausgestorben ja/nein".
- Das **Frontend** könnte um eine Sektion erweitert werden, wo man ein Parameter-Set eingibt und der ML-Predictor simuliert in Sekundenbruchteilen das Ergebnis auf einer Heatmap oder so – quasi als Surrogatsimulator.
- Im **Reporting** könnte man ML-Erkenntnisse einbinden, z.B. automatische Warnungen: "Das ML-Modell prognostiziert für Parameter X ein hohes Risiko für Instabilität".
- Falls man das Modell austauscht, sollte man beachten, dass im UI und der restlichen App minimalinvasiv gearbeitet wird. Solange man die predict-Funktionen gleich lässt, kann intern auch ein Neural Network arbeiten, das z.B. mit TensorFlow geladen wird – Performance dürfte immer noch okay sein, wenn es ein kleines Netz ist.

Insgesamt verstärkt die ML-Einbindung das Projekt, indem sie **Schnelligkeit** und **Analyse** bietet, aber sie ersetzt nicht die eigentliche Simulation. Die Vorhersagen sind nur so gut wie die Trainingsdaten; extrapoliert man weit außerhalb (z.B. viel größere α als je trainiert), können Fehlvorhersagen auftreten. Daher sollte man die ML-Ergebnisse immer im Kontext sehen. Im UI wird dies elegant gelöst, indem die echte Simulationsergebnis-Kurve immer noch gezeigt wird – so kann der Nutzer direkt sehen, ob die Vorhersage ungefähr stimmte, oder ob evtl. überraschende Effekte passierten, die das ML-Modell nicht kannte.

UI/UX Details

Die Benutzeroberfläche von Eco Simulator 2.0 wurde mit **Streamlit** realisiert und legt Wert auf eine übersichtliche, interaktive Bedienung. Im folgenden Abschnitt werden die Elemente

der UI und UX hervorgehoben, um zu zeigen, wie Nutzer mit dem System interagieren und welche Überlegungen hinter dem Design stehen.

Struktur der UI

Beim Start der App präsentiert sich das **Ökosystem Kontrollzentrum 2.0**. Dies dürfte als Titel klar ersichtlich sein (siehe Screenshot oben). Die UI ist vermutlich in **mehrere Seiten oder Tabs** gegliedert:

- **Dashboard (Übersicht):** Zeigt globale Kennzahlen und Auswertungen. Hier sieht man z.B. *Gesamt-Simulationen*, *Kritisch* (Anzahl abgebrochener instabiler Runs), *Durchschn. Beutepop.* (mittlere Endpopulation Beute über alle Läufe), *Durchschn. Mutation Rate*, *Durchschn. Curiosity*. Diese Kennzahlen vermitteln auf einen Blick den Zustand bzw. die Historie. Darunter wird eine "Korrelation Heatmap" angezeigt, die wahrscheinlich Ergebnisse der Sensitivitätsanalysen verdichtet: eventuell eine Korrelationsmatrix zwischen Parametern (α, β, φ, etc.) und Outcomes (z.B. Virus-Endpop, Räuber-Endpop). Oder es könnte eine spezielle 2D-Heatmap sein, wie in der Analyse-Sektion erwähnt (z.B. φ vs β vs Outcome). In jedem Fall dient das Dashboard einem **Managementblick**: Man erkennt Trends, ob alle Simulationen problemfrei laufen (wenn "Kritisch" hoch ist, muss man vorsichtig sein) und welche Parameterbereiche interessant sind. Außerdem listet es unten Simulationsergebnisse – ### Makro-Simulation Tab

Im **Räuber-Beute Tab** der UI können Nutzer die Parameter des ODE-Modells bequem über Slider einstellen . Konkret stehen zur Verfügung: **Beute-Wachstumsrate (α)**, **Räuber-Fressrate (β)**, **Virus-Infektionsrate (φ)** sowie die **anfängliche Beute- und Räuberpopulation**. Die Slider sind mit sinnvollen Bereichen und Standardwerten versehen (z.B. α zwischen 0.01 und 0.5, Standard 0.1; φ zwischen 0.001 und 0.01, Standard 0.005). Dadurch wird verhindert, dass völlig unsinnige Werte eingegeben werden, aber zugleich genug Spielraum für Experimente gelassen. Hat der Nutzer die gewünschten Werte gewählt, startet er die Simulation mit einem Klick auf *"Simulation starten"* . Daraufhin läuft im Hintergrund Simulator.run(); die UI zeigt eine kurze Meldung wie *"Simulation läuft…"* und nach Abschluss *"Simulation abgeschlossen"* (grün hinterlegt). Unmittelbar danach erscheint der **Plot**: ein interaktives Liniendiagramm (Plotly) der Populationsdynamiken. Alle relevanten Größen des Makromodells werden geplottet – typischerweise **Beute (Prey)** und **Räuber (Predator)**, sowie die **Virus-** und **Bakterienpopulation** und ggf. **Immunsystem** und **Nährstoff**. Da fünf bis sechs Kurven in einem Plot visuell anspruchsvoll sein können, hat man sich möglicherweise auf die wichtigsten konzentriert (Beute/Räuber/Virus). Dennoch sind Farblegende und Achsen klar beschriftet (Zeit auf x-Achse, Populationen auf y-Achse), sodass man das Verhalten leicht ablesen kann. Zusätzlich wird – sofern ein ML-Modell vorhanden ist – **oberhalb des Plots eine Vorhersage** eingeblendet: *"ML-Vorhersage – Virus-Endpopulation: X"* . Dies gibt direkt einen Vergleich: lag die tatsächliche Endpopulation dann in der Nähe von X, oder nicht? Diese Rückmeldung in Echtzeit erhöht den Nutzwert der UI deutlich.

Mikro-Simulation Tab

Im **Agenten Evolution Tab** können die Parameter der ABM-Simulation angepasst werden. Hier sind mehrere Slider verfügbar , um die **Startbedingungen und Umgebungsvariablen** festzulegen:

- **Start-Population** (Anzahl initialer Agenten, z.B. 5–100, Standard 20),
- **Grid-Größe** (Seitengröße des quadratischen Spielfelds, z.B. 20–100, Standard 50),
- **Zeitschritte** (Simulationsdauer, z.B. 100–1000, Standard 500),
- **Mutationsrate** (0.0–0.2, Standard 0.05),
- **Reproduktions-Energie** (Schwelle für Fortpflanzung, z.B. 50–200, Standard 120),
- **Resource Regen** (Regenerationsrate der Ressource, z.B. 0.1–5.0, Standard 1.0).

Man bemerkt, dass **Neugier** hier nicht explizit als Slider vorhanden ist – alle Agenten starten mit dem in der Config gesetzten Wert (0.5). Dies ist bewusst so gewählt: Neugier soll hauptsächlich als evolvierender Parameter verstanden werden; der Fokus liegt darauf zu beobachten, wie sie sich durch Mutation und Selektion entwickelt. Würde man mit verschiedenen Start-Neugierwerten experimentieren wollen, ließe sich ein Slider dafür leicht ergänzen.

Nach Justierung der Parameter klickt der Nutzer auf *"Agenten-Simulation starten"* . Die Simulation läuft, was bei vielen Schritten und Agenten einen kleinen Moment dauern kann. Anschließend erscheinen analog zum Makro-Tab ein *"Simulation abgeschlossen!"*-Hinweis und die **Plots**. Für die ABM werden im Standard **drei Kurven** über der Zeit dargestellt: **Population** (Anzahl Agenten), **Ø Speed** (Durchschnittliche Geschwindigkeit) und **Ø Curiosity** (Durchschnittliche Neugier) . Diese geben einen kompakten Überblick über das Geschehen in der Agentenwelt: Man sieht z.B. an der Populationskurve, ob die Agenten aussterben oder sich stabil vermehren; an der Speed- und Curiosity-Kurven, ob und wie schnell sich diese Eigenschaften durchsetzen. Die Darstellung nutzt unterschiedliche Farben und eine Legende, sodass die Kurven unterscheidbar sind (im Code pink für Neugier, blau für Speed, etc.). Auch Achsenbeschriftungen (Zeit vs. Wert) und Titel ("Agenten Evolutionstrends") sind vorhanden, was zur Verständlichkeit beiträgt .

Ein Vorteil der Plotly-Visualisierung ist, dass Benutzer die Kurven ein- und ausblenden können (durch Klick auf Legenden-Einträge) und zoomen können, um Details zu untersuchen. So kann man etwa die Anfangsphase genauer betrachten, wenn dort interessante Veränderungen passieren.

Nutzererlebnis und Design

Die UI setzt auf ein **dunkles Theme** mit kontrastreicher Schrift und Highlights. In der CSS-Datei (assets/style.css) sind z.B. ein dunkelblauer Hintergrund und hellgraue Schrift festgelegt , wie es im Screenshot des Dashboards zu sehen ist. Dieses Design ist modern und angenehm für längere Betrachtung, da es nicht so grell ist wie ein weißer Hintergrund. Wichtige Kennzahlen (im Dashboard) sind in gelber Schrift hervorgehoben, was sofort ins Auge fällt.

Zur besseren Orientierung verwendet die App auch **Icons/Emojis und klare Section-Labels**. Der Titel *"Ökosystem Kontrollzentrum 2.0"* wird mit einem Pflanzenspross-Emoji verziert, um gleich klarzumachen, dass es um ein Öko-System geht. Die Feature-Liste (im README bzw. ggf. in der UI als Hinweise) nutzt z.B. ein DNA-Emoji  neben "Features", was visuell ansprechend ist. Solche Details verbessern die UX subtil, indem sie Text auflockern und thematisch passend schmücken.

Die **Interaktionsgestaltung** ist bewusst einfach gehalten: Alle Einstellungen erfolgen über Slider/Buttons im Hauptfenster (keine separaten Dialoge oder komplizierten Formulare). Die Standardeinstellungen ermöglichen es, mit *einem Klick* eine sinnvolle Simulation laufen zu lassen, sodass auch Erstnutzer Erfolge sehen, ohne jedes Feld anpassen zu müssen. Gleichzeitig können fortgeschrittene Nutzer an den Reglern drehen und das System reagiert sofort auf sinnvolle Weise.

Das Dashboard ergänzt die Erfahrung, indem es **Meta-Informationen** liefert: Der Nutzer sieht z.B., dass er schon 8 Simulationen durchgeführt hat und dass vielleicht alle 8 "kritisch" waren – ein Anstoß, die Parameter zu ändern. Oder er erkennt, dass im Schnitt die Mutation Rate 0.045 beträgt in seinen Runs – was nahe am Standard 0.05 liegt, also hat er dort wenig variiert. So führt die UI den Benutzer auch ein Stück weit, indem sie Feedback gibt über das *ganze* Experimentier-Verhalten, nicht nur den einzelnen Lauf.

Bedienung und Erweiterbarkeit

Die Streamlit-App reagiert in Echtzeit auf Eingaben. Wenn der Nutzer einen Slider verschiebt, werden im Hintergrund sofort Variablen gesetzt und (falls implementiert) beispielsweise die ML-Vorhersage aktualisiert. Die Simulation selbst wird aber erst bei Button-Klick gestartet, um nicht bei jedem Slider-Tick einen Lauf zu triggern – das wäre zu rechenintensiv. Dieses *Pattern* (Parameter via Widgets einstellen, dann explizit auslösen) ist typisch und sinnvoll hier.

Für Entwickler:innen ist die UI leicht erweiterbar: Möchte man z.B. den Einfluss der **Räuber-Sterberate (γ)** vom Nutzer verändern lassen, kann man in app.py einfach gamma = st.slider("Räuber-Sterberate (γ)", min, max, default) hinzufügen und den Wert in SimulationConfig übergeben. Ebenso könnte man ein Kontrollkästchen hinzufügen, um die Anzeige bestimmter Kurven im Plot optional an/aus zu schalten (etwa "Zeige Bakterien/Virus-Kurven"). Streamlit macht dies sehr einfach, da Logik und Darstellung in Python eng verzahnt sind. Die UI-Code ist somit nicht nur für Endnutzer freundlich, sondern auch für Entwickler sehr zugänglich.

Abschließend sorgt die UI/UX insgesamt dafür, dass Eco Simulator 2.0 **intuitiv erfahrbar** ist. Sowohl wissenschaftlich interessierte Nutzer (die mit Parametern spielen wollen) als auch Entwickler (die intern etwas verändern und sofort testen möchten) bekommen ein unmittelbares Feedback und können schrittweise vorgehen. Die klare Trennung der Bereiche (Makro vs. Mikro) verhindert Verwechslungen und erlaubt es, fokussiert mit einer Ebene zu experimentieren. Durch die optisch ansprechende Aufbereitung und die interaktiven Elemente wird aus komplexer Simulation ein **greifbares Erlebnis**, was letztlich Motivation und Verständnis fördert.

Ausblick

Eco Simulator 2.0 stellt eine solide Basis für die Erforschung und Erweiterung ökologischer Simulationen dar. Dennoch gibt es zahlreiche **Möglichkeiten zur Weiterentwicklung**, um das System noch leistungsfähiger, realistischer und vielseitiger zu machen. Im Ausblick werden einige Ideen skizziert, die Entwickler:innen in Zukunft aufgreifen könnten:

- **Engere Kopplung von Makro- und Mikro-Modell:** Bisher laufen das ODE-Modell und das ABM getrennt nebeneinander. Ein spannender nächster Schritt wäre, beide Ebenen **in einem Hybridmodell** zu verbinden. Beispielsweise könnten die **aggregierten Ergebnisse des ABM** (etwa die finale Populationsdichte oder Durchschnittsfitness) zurückgespeist werden, um die Parameter im Makromodell anzupassen – oder umgekehrt könnten Trends des Makromodells (z.B. periodische Umweltbedingungen) ins ABM einfließen. Ein konkretes Szenario: Das Makromodell könnte die Gesamtressourcenmenge oder Klimabedingungen vorgeben, welche die ABM-Umgebung beeinflussen, während das ABM emergente Phänomene (wie Aussterben einer Spezies) liefert, die ins Makromodell übernommen werden. Solche **Mehrskalen-Simulationen** würden die Stärken beider Ansätze verbinden und noch realistischere Ökosysteme abbilden.
- **Erweiterung des ABM um Räuber-Beute-Interaktionen:** Momentan gibt es im ABM nur eine Art von Agenten, die Ressourcen fressen. Eine logische Erweiterung ist die Einführung einer **zweiten Agentenart**, die als Räuber auf die erste jagt. Damit ließe sich ein direktes agentenbasiertes Räuber-Beute-Modell realisieren, das man mit dem ODE-Modell vergleichen kann. Herausforderungen dabei: Man müsste Agenten-Klassen oder -Attribute einführen, um Freund/Feind zu unterscheiden, Jagd- und Kampfverhalten implementieren (z.B. ein Räuber-Agent sucht Beute-Agenten in seiner Sichtweite und bewegt sich darauf zu, ein Fang-Mechanismus, evtl.

Tarnung/Stärke nutzen). Auch Fortpflanzung der Räuber müsste geregelt werden (z.B. energieabhängig durch verzehrte Beute). Dies wäre ein größeres Unterfangen, würde aber das System erheblich bereichern. Attribute wie camouflage (Tarnung) und strength bekämen dann echte Bedeutung im Selektionsgeschehen. Die Performance des ABM müsste für mehr Individuen und komplexere Interaktionen optimiert werden (ggf. Multi-Threading oder spatial indexing für effiziente Nachbarschaftssuche).

- **Komplexere Umwelt und Ökosysteme:** Das derzeitige Umweltmodell ist simpel homogen (ein Ressourcentyp, gleichmäßig regenerierend). In Zukunft könnte man **räumliche Heterogenität** einführen – z.B. unterschiedliche Zonen auf dem Grid mit variierender Regenerationsrate (simulierter Wald vs. Wiese), oder mehrere Ressourcentypen (etwa Pflanzen und Wasser, die beide nötig sind). Auch **Jahreszeiten** oder zeitlich oszillierende Ressourcen könnten simuliert werden. Im Makromodell ließe sich eine tragfähige Kapazität (logistisches Wachstum) für die Beute ergänzen oder ein zweiter Räuber (Super-Räuber) einführen. Die Bakterien/Virus Komponente könnte ausgebaut werden zu **Epidemien in der Räuber-Beute-Population** (z.B. ein Virus, der die Räuber befällt). Hierfür müsste man die Gleichungen koppeln (etwa Räuber als Wirt im φ-Term nutzen). Solche biologisch reichhaltigeren Modelle würden die Komplexität erhöhen, aber mittels der vorhandenen Struktur relativ gut integrierbar sein.
- **Verbesserte KI-Modelle und Analysen:** Die bisher eingesetzten Random Forests liefern gute Ergebnisse, doch man kann weiter gehen. Denkbar ist der Einsatz von **Deep Learning**, z.B. neuronaler Netze, um Vorhersagen zu verbessern oder komplexere Fragen zu beantworten (z.B. Prädiktion ganzer Zeitreihenverläufe, nicht nur Endwerte). Zudem könnte man **Reinforcement Learning** in die ABM integrieren: Statt Agenten, die stochastisch handeln, könnte man Agents implementieren, die mittels RL ihre Strategie optimieren (z.B. ein Agent lernt während der Simulation, wie er am effektivsten Energie sammelt, was quasi einer Anpassung auf kurzen Zeitskalen entspricht). Das wäre methodisch anspruchsvoll, würde aber KI und Simulation noch enger verweben. Auf Analyse-Seite könnte man das **Optimumssuchen** automatisieren: Mit den ML-Modellen ließe sich ein Optimierungsalgorithmus (z.B. genetischer Algorithmus oder Bayesian Optimization) koppeln, der versucht, Parameter zu finden, die ein gewünschtes Ziel erreichen (z.B. Koexistenz aller Arten, maximaler Ertrag einer Population, etc.). So würde aus dem Simulator ein **Entscheidungsunterstützungssystem**.
- **Performance und Skalierung:** Für sehr große Experimente könnte man überlegen, **Teile in kompilierte Sprachen** auszulagern. Zum Beispiel könnte der ABM-Kern in C++ neu geschrieben und über Python angebunden werden, um Tausende Agenten in Echtzeit zu handhaben. Oder man könnte GPU-Computing einsetzen (CUDA), etwa indem man die Agenten als Partikel in einem GPU-Kernel simuliert – nützlich, wenn man Richtung „evolutionäres Spiel der Leben"-ähnliche Szenarien geht. Auch eine Verteilungs-Simulation (mehrere Prozesse oder Cluster) wäre denkbar, falls extrem viele Runs benötigt werden. Dank Docker könnte man skalieren, indem man das Container-Image in der Cloud vielfach laufen lässt. Diese Aspekte sind eher langfristig relevant, doch es ist gut im Hinterkopf zu behalten, dass die Architektur (Trennung in Module) es erlaubt, einzelne Teile auszuwechseln, ohne das gesamte System neu zu bauen.
- **Benutzeroberfläche und Erlebnis:** Auf UI-Seite könnte man weitere Komfortfunktionen integrieren: z.B. ein **Live-Monitoring** der ABM während sie läuft (aktuell sieht man das Ergebnis erst nach Abschluss; mit Streamlit ließe sich aber z.B. alle n Schritte ein Zwischenplot updaten, um die Evolution quasi live zu verfolgen). Auch eine **Visualisierung der Grid-Welt** wäre interessant – z.B. eine farbcodierte

Matrix oder ein kleiner animierter Scatter-Plot, der zeigt, wo sich Agenten befinden und wo Ressourcen verbraucht werden. Dies würde allerdings deutlich umfangreichere Darstellung erfordern, ist aber machbar (z.B. mit Plotly oder Pillow Bilder generieren). Zudem könnten **mehrsprachige Interfaces** (derzeit ist alles Deutsch) oder Tooltips/Hilfetexte die Nutzerbasis erweitern und Neueinsteigern Konzepte erklären. Da die Software primär für Entwickler gedacht ist, war Dokumentation wichtiger als Endnutzer-Onboarding – aber falls es in Bildungsumfeldern eingesetzt wird, wäre eine integrative Hilfe in der UI sinnvoll.

- **Mehr Tests und Validierung:** Aus Entwicklungs-Sicht wäre ein Ausbau der **Test-Suite** wünschenswert. Bisher gibt es Basistests; man könnte z.B. Tests hinzufügen, die bestimmte bekannten Outcomes prüfen (Regression-Tests: gleiche Seeds => gleiche Ergebnisse). Auch Performance-Tests (Profiling) könnten automatisiert werden, um bei Änderungen sofort zu sehen, ob etwas viel langsamer wurde. Und natürlich, falls neue Komponenten (neue Arten, RL-Agenten, etc.) hinzukommen, sollten diese ebenfalls gut getestet und dokumentiert werden.

Zusammenfassend besitzt Eco Simulator 2.0 ein **großes Potenzial für Erweiterungen**. Dank der modularen und klar strukturierten Architektur können Entwickler:innen sukzessive neue Ideen integrieren. Wichtig ist, dabei die wissenschaftliche Fundierung beizubehalten – d.h. neue Modelle oder Funktionen sollten validiert und mit sinnvollen Parametern versehen werden. Die Kombination aus ökologischem Modellkern, agentenbasierter Evolution und maschinellem Lernen ist ein zukunftsträchtiges Konzept: Sie erlaubt das **Durchspielen von „Was-wäre-wenn“-Szenarien** auf mehreren Ebenen und könnte etwa in der Forschung zu nachhaltigen Ökosystemen, in der Lehre oder in KI-Experimenten angewandt werden. Mit weiteren Verbesserungen wird Eco Simulator 2.0 nicht nur ein Entwickler-Tool bleiben, sondern sich vielleicht auch als **allgemeine Simulationsplattform** für komplexe adaptive Systeme etablieren.