Ökosystemsimulation

Jonas Berggren, Jacob Maxton

December 20, 2019

Abstract

In diesem Dokument erklären wir, wie wir versucht haben eine numerische Simulation eines Ökosystems zu entwickeln, und was für Erkenntnisse sich daraus ziehen lassen. Außerdem gehen wir auf die Schwierigkeiten ein, auf die wir gestoßen sind, und, wieso wir es nicht geschafft haben unsere ursprünglich beabsichtigten Ereignisse zu erhalten. Unsere Simulation basiert auf Objektorientierter Programmierung und simuliert die Beute-Räuber-Beziehung zwischen Kaninchen und Füchsen.

Contents

1 Die Simulation

1.1 Objektorientierte Programmierung und numerischen Simulationen(Jonas)

Objektorientierte Programmierung bezeichnet das Programmieren mit Hilfe so genannter Klassen und Objekte. Ein Objekt, oder eine Instanz einer Klasse ist beispielsweise ein Individuum des Typs Kaninchen. In dem Beispiel würde die Klasse den Kaninchen im allgemeinen entsprechen. Klassen kann man sich vorstellen wie Methodenkarten, in denen eine Reihe von Informationen gespeichert sind. Diese können Instanzvariablen oder Methoden sein. Instanzvariablen sind Variablen die jede Instanz trägt. Methoden sind Funktionen oder Anweisungen die jede Instanz der Klasse ausführen kann. Klassen haben außerdem Hierarchien. Eine Instanz einer sogenannten Unterklasse hat automatisch alle Methoden der übergeordneten Klasse. In unserer Simulation sind die Klassen `Rabbit` und `Fox` Unterklassen zu `Animal`. Somit erben sie alle Methoden und Variablen von `Animal`.

Bei numerischen Simulationen wird ein Szenario aus der reellen Welt durch eine Computersimulation nachgestellt. Dabei wird das System zum Zeitpunkt t betrachtet. Auf Grundlage dessen wird der Zustand des Systems zum Zeitpunkt $t + h$ berechnet. Dabei gilt, je kleiner h ist, desto genauer ist die Simulation. Alle Änderungen werden angewandt und der Prozess wird wiederholt bis der gesamte Zeitraum Simuliert wurde, den es zu betrachten gilt. Dies ist nützlich um die Gültigkeit von Modellen zu prüfen oder z.b. die Stabilität des betrachteten Systems zu testen.

1.2 Konzept der Simulation(Jonas)

In der Simulation werden Pflanzen, Kaninchen und Füchse simuliert. Jedes Tier ist entweder eine Instanz der Klasse `Fox` oder `Rabbit`, die jeweils Unterklassen der Klasse `Animal` sind. In Animal sind alle Methoden gespeichert, die für Kaninchen und Füchse identisch ausgeführt werden, wie der Konstruktor `__init__` oder `movetargeted` zum gezielten bewegen. Methoden die für Füchse und Kaninchen unterschiedlich sind, sind in den jeweiligen Unterklassen gespeichert. Diese sind z.b. `findtarget`, zum finden aller potentiellen Ziele. Auf die unterschiedlichen Instanzvariablen und Methoden wird aber in Kapitel 1.3 weiter eingegangen. Alle Tiere haben ein kreisförmiges Sichtfeld, was nicht die komplette Karte abdeckt. Sie haben alle ein, mit der Zeit zunehmendes, Bedürfnis sich fortzupflanzen und zu essen. Alle Bedürfnis werden bei deren Erfüllung verringert. Wird der Hunger zu stark kann das Tier verhungern und sterben. Außerdem kann ein Tier zu jedem Zeitpunkt, mit einer Wahrscheinlichkeit sterben, die vom alter abhängig ist. Kaninchen essen pflanzen und Füchse essen Kaninchen, wobei gegessen Kaninchen sterben. Kaninchen können vor Füchsen flüchten, die sich innerhalb des Sichtfelds befinden.

1.3 Methoden(Jonas)

1.3.1 Aussuchen eines Ziels

Dies wird durch die Methode `findtarget` geregelt, die aus der Hauptschleife aufgerufen wird. `findtarget` ist sowohl eine Methode der Klasse `Fox` als auch der Klasse `Rabbit`. Dort sind sie aber unterschiedlich definiert, dar Füchse sich anders verhalten sollen als Kaninchen. Füchse betrachten nur andere Tiere und speichern alle geeigneten Partner in die Liste der Partner, und alle Kaninchen in die Liste der Potentiellen Mahlzeiten. Kaninchen betrachten aber Tiere und

Pflanzen. Pflanzen werden zum Essen gespeichert, potentielle Partner zu Partnern und Füchse zu Fluchtpunkten. Objekte werden jedoch nur gespeichert wenn sie sich innerhalb des Sichtradius `self.sens` befinden. Anschließend wird geprüft welches Ziel angesteuert werden soll. Dazu wird aus jeder Liste das nächste Element gesucht. Wenn ein Kaninchen ein Fuchs sieht hat die Flucht immer oberste Priorität. Der Fuchs wir anvisiert und der Bewegungsvektor wird mit -1 multipliziert. Danach wird geprüft ob Hunger oder Libido stärker wirkt und demnach wird entschieden ob das nächste Essen oder der nächste Partner anvisiert wird.

1.3.2 Bewegung

Die Bewegung der Tiere wird durch die Methoden `movetargeted` und `moverandom` definiert. Jedes Tier hat zwei Arten wie es sich fortbewegen kann. Wenn es ein bestimmtes Ziel hat, wird `movetargeted` aufgerufen, und das Tier bewegt sich entlang des Vektors von der eigenen Position zum Ziel. Dabei ist der Bewegungsvektor auf die Bewegungsgeschwindigkeit normiert. Wenn kein Ziel in Sicht ist, wird `moverandom` aufgerufen, und sie bewegen sich zufällig. In beiden Fällen wird die Methode `collision` jedes mal aufgerufen. Diese verhindert das Tiere aus der Karte raus laufen.

1.3.3 Fortpflanzung und Mutation

Es können nur gleichrassige, heterosexuelle und altersgerechte Paare ein Kind zeugen. Außerdem müssen beide Eltern für eine bestimmte Zeit keiner Kinder gezeugt haben, und sie dürfen für eine Längere Zeit nicht mit einander Kinder gezeugt haben. Haben sich zwei gefunden wird der Liste der Tiere eine neue Instanz der Klasse hinzugefügt. Für die Definition der mutierbaren Eigenschaften wird der Durchschnitt aus den jeweiligen Werten der Eltern als Mittelwert angenommen. Der Wert des Kindes weicht um x von diesen Durchschnitt ab, wobei x eine zufällige Zahl zwischen -0.1 und $+0.1$ ist.

Die Eltern speichern sich gegenseitig als ehemalige Partner ab und betrachten sich anschließen für eine festgelegte Frist nicht mehr als mögliche Partner.

1.3.4 Tod

Tiere können auf drei unterschiedliche Arten sterben. Sie können verhungern, sie können durch die Altersabhängige Funktion sterben oder Kaninchen können gefressen werden. Dies wird durch die Methoden `starve`, `die` und `eat` geregelt. Die Methoden `starve` und `die` werden bei jeder Iteration aufgerufen, und `eat` wenn ein Kaninchen gefressen wird.

Die Todeswahrscheinlichkeitsdichte wird durch eine Funktion beschrieben nach dem Muster:

$$P = a(e^{-t+b} \cdot c \cdot t + d) \tag{1}$$

Die Grafen zeigen die Todeswarscheinlichkeitsdichte um eine Größenordnung vergrößert.

Figure 1: Todeswahrscheinlichkeitsdichte

Durch so eine Funktion ist P für ein kleines t relativ groß. P erreicht im positiven Wertebereich ein Minimum und wächst dann approximativ linear an. Dies fasst Kindersterblichkeit und Altersschwäche in ein Funktion zusammen. Die Funktion P kann Werte Annehmen die größer sind als 1. Das ist aber aufgrund der Funktionsweise des Programms nicht problematisch. Beim sterben durch gegessen werden löscht der Fuchs die gegessene Instanz aus der Liste der Tiere.

1.3.5 Andere Methoden

Hinzu kommen noch weiter Hifsmethoden wie z.b. `distance`, die den Abstand zwischen $self$ und einem beliebigen Punkt berechnet. Diese sind jedoch für das allgemeine Verständnis unsere Arbeit nicht essentiell.

1.4 Frontend (Jacob)

Was bringt die beste Simulation, ohne dass man sich die Werte angucken kann? Gar nichts. Deswegen haben wir 4 Dateien, die automatisch von dem Script in einem Ordner zusammengefasst werden:

Log.txt

Die Log.txt ist eine Textdatei die, oh wie wunder, den Log der Simulation enthält. Hier wir immer wenn ein Tier stirbt, ein neuer Tag beginnt oder irgendetwas anderes Passiert, dies in eine neue Zeile eingetragen. Dies sorgt für eine unglaublich große Datei, die vor allem mit kurzen Zeilen gefüllt ist. Deswegen will man sich als Mensch diese Datei nur angucken, wenn man unbedingt muss. Muss man aber nicht, da die wichtigen Informationen auch in den anderen Dateien Stehen:

Info.txt

Info.txt ist schon wieder eine Textdatei, aber diesmal eine deutlich kürzere: Die Info.txt enthält:Die Laufzeit (die Anzahl der Tage der Simulation), sowie die Anzahl und Gründe der Todesfälle nach Tierart gegliedert. So kann man aus dieser Datei genau auslesen, wie viele Füchse im Laufe der Simulation an Hunger gestorben sind, oder wie viele Hasen von Füchsen gefressen wurden.

Viel einfacher für den Menschen anzuschauen sind jedoch die beiden Videos, kreativ benannt: video.avi und video2.avi. Video.avi ist hier eine Videodatei, die eine Zeitliche Auflösung der Mutationen aufzeigt: Hier werden Dreidimensionale Graphen dargestellt, auf jeder der Achsen jeweils einer der Werte, die wir mutieren: Geschwindigkeit, Hungerresistenz und ein Skalar für den Sexualtrieb. Diese sehen ungefähr so aus: Hier können wir eine sehr frische Zivilisation sehen; die Werte entsprechen noch sehr den Ursprünglichen Werten. Wir können an diesen Graphen auch auslesen, dass die geringste Hungerresistenz, die jemals bei einem Tier in dieser Simulation vorhanden war 0.5 ist. Dies erkennt man, da Ober- und Untergrenzen der Achsen so gesetzt sind, dass die geringsten und größten innerhalb einer Simulation verzeichneten Werte die Ober- und Untergrenzen festlegen. Das liegt daran, dass in dem Video 15 Graphen pro Sekunde zu einem Video zusammengeschnitten sind, und diese andernfalls unlesbar wären. Ähnlich ist auch video2.avi, welches eine Visuelle Repräsentation der Simulation darstellt. Die Punkte, die sich nicht bewegen, sind hier Pflanzen, die dunklen bewegenden Punkte Hasen und die Hellen bewegenden Punkte Füchse. Auch dieses Video ist mit 15 fps gerendert, und weil ein Bild jeweils einem Tag entspricht werden in den Videos also immer 15 Tage pro Sekunde dargestellt.

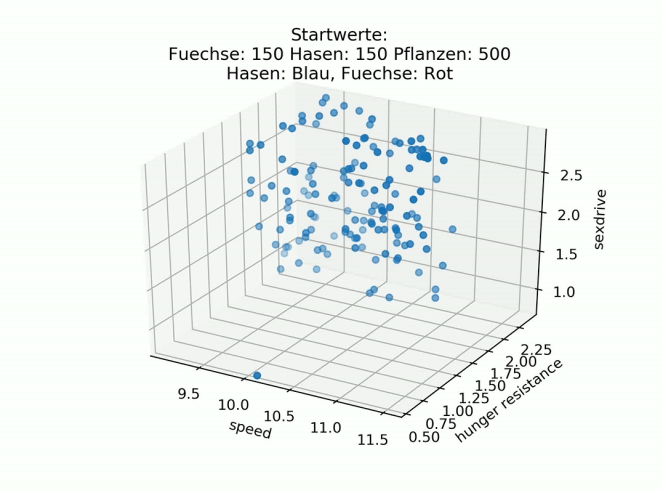

Figure 2: Lotka-Volterra-Regeln

2 Die Analyse

2.1 Theorethisch (Jonas)

Uns ist es mit den Mitteln, die wir haben nicht gelungen Die Simulation so zu gestalten, dass es lange genug dauert bis eine der Tierarten ausstirbt, um daraus Erkenntnisse ziehen zu können. Dies ist dadurch zu erklären, dass uns nicht gelungen ist ein Verhältnis von alle Parametern zu finden, was die Simulation haltbar macht.

In der Ökologie gelten so genannte Lotka-Volterra-Regeln. Diese sagen eine Periodische Oszillation der Populationen von Raub- und Beutetieren voraus. Diese wird durch das Narhungsangebot für die Räuber sowie, das Risiko gegessen zu werden für die Beutetiere, verursacht. Uns ist es nicht gelungen unsere Werte so anzupassen, dass das Minimum dieser Oszilation über 0 bleibt. Ökosysteme haben eine sehr heikle Balance. Ins besondere Räuber dürfen weder zu gute noch zu schlechte Jäger sein, damit

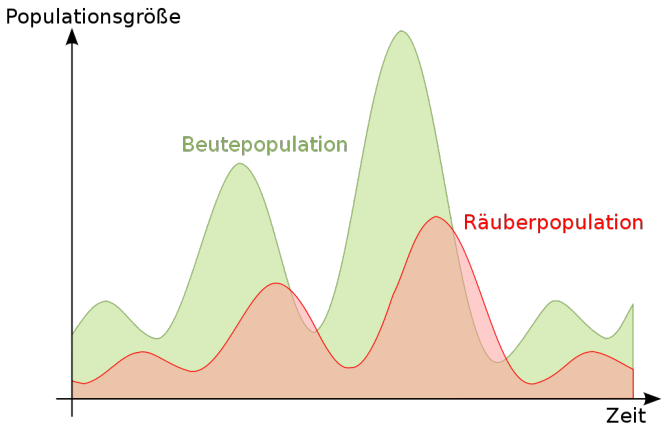

Figure 3: Lotka-Volterra-Regeln

das System so bestehen bleiben kann. Sind sie zu schnell, zu zahlreich, zu fortpflanzungsfähig, zu hungerresistent, etc. besteht schnell das Risiko, dass sie ihrer Beutetiere ausrotten und somit sich selbst. Sind sie in diesen Eigenschaften zu schwach können sie auch mit umfangreichem Essens Angebot nicht überleben. Deswegen ist die Vielfalt von Ökosystemen so wichtig für deren Fortbestand.

2.2 Praktisch (Jacob)

Warum haben wir unser Ziel nicht erreicht? Wie Jonas schon ausgeführt hat ist es sehr kompliziert, die richtigen Werte zu finden. Was man also machen müsste ist, sehr viele Werte auszuprobieren. Hier kommen ein großes Problem auf informatischer Seite: Zeit. Es dreht sich alles um Zeit. Jeder Tag einer Simulation darrt im schnitt 0.1 bis 0.4 Sekunden,. Da jede Simulation auf grundlegendem Zufall basiert (bei der Bewegung, Mutation etc.), müssen wir also mehrere Simulationen mit den gleichen Werten machen, und die Mittelwerte nehmen, damit wir sicher sagen können, dass eine Wertekombination wirklich gut ist, und nicht einfach nur durch Zufall lange angehalten hat. Mein Computer kann gleichzeitig 3 Simulationen laufen lassen kann (er hat 4 Kerne, also eine Simulation pro Kern und ein Kern für die Kontrolle und Verarbeitung der Informationen), deswegen haben wir pro Werte jeweils 5 Simulationen gewählt, dadurch konnten wir den Fehler durch den Zufall zufriedenstellend abschwächen, ohne die Zeit pro Wertekombination in die Länge zu ziehen. Das führt dazu, dass jeder Tag in diesen Simulationen letztendlich 0.2 bis 0.8 Sekunden braucht, da ja nicht nur eine Simulation, sondern 5 Simulationen berechnet werden müssen, davon jedoch nur 3 gleichzeitig. Zwei Kerne können nicht an der selben Simulation arbeiten. Machen wir also eine einfache Rechnung: Gehen wir von einer durchschnittlichen Laufzeit von 100 Tagen aus. Dann brauchen wir, um jeweils eine Wertekombination zu testen: 100 * 0,2s = 20s oder, mit dem Größeren Wert, 100 * 0,8s = 80s. Dieser Wert ist für eine Wertekombination. Und jetzt kommt das Problem: Wir wissen nicht, welche Werte gut sind, und welche schlecht. Wir müssen den Computer also raten lassen. Aber wenn jetzt jeder Wert 50 Sekunden (Mittelwert zwischen 20 und 80) darrt, dann sind das, mit den Schritten dass der Teste-Script auch noch erkennen muss, ob das gerade gut oder schlecht war, eine Minute pro Wertekombination. Und damit schaffen wir 60 Wertekombinationen pro Stunde – einfach Raten geht also nicht. Nehmen wir also einen Ansatz aus der AI-Technologie: Machine Learning, genarrt: einen Genetic Algorytm. Dieser nimmt letztendlich die beiden besten Wertekombinationen, die er hat, und bastelt aus den beiden eine neue. Dieses wird nun mit einer geringen Wahrscheinlichkeit um einen geringen Wert abgeändert, und getestet. Danach beginnt das ganze von Vorne. Theoretisch ist dieser Anlauf ein sehr guter, und führte bei meinen anderen Anwendungen auch schon mehrfach zu erfolgreichen AIs. Was ich hier nicht bedacht hatte: Genetic Algorythms brauchen tausende Generationen, um wirklich ein gutes Ergebnis zu liefern. Und auch nur eintausend Generationen laufen zu lassen hat bei unseren Geschwindigkeiten 8 Stunden gebraucht. Noch dazu ist eine Simulation auszutarieren eine sehr komplizierte Aufgabe. Als Vergleich: Eine AI zu schreiben, und zu trainieren, die Minesweeper Spielt (eine sehr leichte Aufgabe) hatte bei mir mit einem Genetic Algorythm 14.000 Generationen gebraucht, bis es das hinbekommen hat – dank einer Effizienteren Programmsprache und eines einfacheren Problems habe ich damals aber 500 Generationen pro Minute hinbekommen. Bei der Simulation erwarte ich also (im Nachhinein) sogar sechsstellige Generationenzahlen, was (mit dem Wert der Minute Pro Simulaiton) 69,4 Tage dauern würde. Diese Zahl ist wahrscheinlich sogar sehr optimistisch, da mit besseren Werten die Simulationen länger dauern würden, und die maximale Länge der Simulationen auf 10.000 Tage festgelegt ist. Es war also von Anfang an unmöglich, die Simulation auf die Art und Weise, wie wir sie gemacht haben, so auszutarieren, dass wir Werte

haben, mit denen wir eine Population haben, die überhaupt überlebensfähig ist. Und erst, wenn nicht alle 500 Hasen in den ersten 20 Tagen sterben, können wir untersuchen, wie sich die Hasen verändern, wenn wir z.B. mehr Füchse haben, oder weniger Essen.

3 Verbesserungsmöglichkeiten (beide)

Eine Möglichkeit das Programm zu verbessern wäre ein sogenanntes Gridsystem zu implementieren. Dabei wird das Feld in mehrere Subfelder unterteilt. Die Objekte werden in einer Matrix gespeichert wobei jeder Index der Matrix ein Subfeld zugeordnet ist. Das bietet den Vorteil, dass die Tiere bei Jeder Iteration nicht mehr die die Position aller Objekte abfragen müssen, sondern nur die der Objekte die sich im selben bzw. in einem der Felder befinden, die mit dem Sichtkreis überlappen. Somit kann die Laufzeit pro Iteration stark reduziert werden.

Wasserndem kann eine andere, schnellere Programmiersprache verwendet werden. Für Umfangreiche wissenschaftliche Anwendungen kann eine erweiterte Version eines solchen Programms auf stärkeren Computern ausgeführt werden.

4 Anwendungen

Eine Umfangreichere Simulation dieser Art könnte nützlich sein um die Auswirkungen von Menschlichem Eingriff in Ökosysteme wie Klimaerwärmung, Lebensraumsverändungen oder aussterben einer Art abzuschätzen. Dies könnte nützlich sein um Notwendigkeit und Dringlichkeit von Umweltmaßnahmen abzuschätzen.

5 Fazit (beide)

Wir haben mit Hilfe Objektientierter Programmierung eine Vereinfachte Simulation eines Ökosystems entwickelt indem wir uns auf die Wechselwirkung zwischen zwei Tierarten konzentriert haben. Dabei sind wir auf unterschiedliche Schwierigkeiten gestoßen. Zuerst mussten wir entscheiden wie genau unsere Simulation sein soll. Anschließend war es sehr schwierig all Willkürlich gewählten Parameter so zu beziffern, dass keine der Arten zu schnell ausstirbt. Dabei haben wir festgestellt wie fragil auch so ein einfaches Ökosystem ist. Daran sind wir letztendlich bei unserem Vorhaben, die Auswirkung der Umgebung auf einen Hasen zu untersuchen, gescheitert.