Ecosystem project

Computational Model

by

Inbar koursh

This project is submitted under the scope of 1 unit and as partial fulfillment of the requirements for obtaining a grade in computational science

This work was performed under the guidance of Shlomo Rozenfeld

October 2020

Table of contents

Acknowledgments

I would like to express my appreciation to Shlomo Rozenfeld for sharing valuable insights and ideas whenever we conversed. I would also like to thank my mother Dr. Daphna Mezad Koursh who helped me with technical pointers on how to write a paper.

Finally, I would like to thank Daniel Schiffman and Justin Helps for the inspiration for this project. Daniel Schiffman's nature of code's ecosystem project as well as Justin Helps Primer YouTube videos were invaluable resources for this project.

Abstract

Studying natural biological systems using a simulated ecosystem and reinforcement learning.

This project attempts to mimic similar simulations of ecosystems to further understand the factors that lead to successful species. This is done by studying the correlation between the recorded variables at any given time.

The model was successfully established. It demonstrated a significant correlation between all the 13 variables that were examined. This demonstrates the long thought idea that all an ecosystem is a highly complex system in which every change impacts the entire system.

Background

It is expected that the reader would hold a fair knowledge of python and understand the concept behind evolutionary-based neural networks.

Neuroevolution is a machine learning technique that applies evolutionary algorithms to construct artificial neural networks, taking inspiration from the evolution of biological nervous systems in nature. Compared to other neural network learning methods, neuroevolution is highly general; it allows learning without explicit targets, with only sparse feedback, and with arbitrary neural models and network structures. Neuroevolution is an effective approach to solving reinforcement learning problems and is most commonly applied in evolutionary robotics and artificial life. [1]

[1] Joel Lehman and Risto Miikkulainen (2013) Neuroevolution. Scholarpedia, 8(6):30977., revision #137053

For example, it is next to impossible to say if any one chess move at the start of the game is a good move. However, in neuroevolution, you can train the network by giving it a score on the whole game and not just on 1 move (unlike traditional approaches).

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It's high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components. Python is open-sourced and is available for free on all major platforms. [2]

The goal of this work was to create a model based on Neuroevolution that would be used to mimic an ecosystem. The model would then be analyzed to gain insight into the factors that contribute to the success of the ecosystem.

Methods

Algorithm

Note: the identifier - agent/creature is used interchangeably.

The idea of this algorithm is to give every creature different traits and different neural networks as a brain with each network tackling a specific type of behavior. Unlike other algorithms, I do not seek to find the optimal solution to the simulation, but rather observe a simulated version of evolution and draw conclusions based on the effects of certain factors on the behavior of the creatures. Because of this, they can do as they wish. They are not forced to breed, nor is the size of their pregnancy hardcoded. They are essentially given "free will". All factors that can be reasonably expected to be subject to evolution, are.

They can move freely around the simulation based on their speed and their movement neural network. They interact with the other creatures in the simulation. They eat, they reproduce, they age and they die. No master system breeds them to get the perfect creature, instead, they choose if and when to reproduce. Furthermore, they choose how much of their mass they are willing to give to their children. Importantly mass is not created or destroyed in reproduction, simply transferred.

[2] What is Python? Executive Summary. (n.d.). Retrieved August 30, 2020, from https://www.python.org/doc/essays/blurb/

Note that the coordinate system ranges from -1 to +1 but the world is round. This means that agents can move from -0.99 to 0.99 (assuming 0.2 movement speed). This also carries with it the challenges of distance calculations for collision detection and others. This was done as a response to the agents sometimes getting stuck in +1/-1.

Dependencies

Pip dependencies.

1. configparser
2. copy
3. gc
4. os
5. random
6. time
7. typing
8. NumPy
9. matplotlib
10. Pillow
11. openpyxl
12. savReaderWriter
13. pickle
14. json
15. argparse
16. sys

Apt dependencies.

1. FFmpeg

Custom dependencies.

1. Modular neural network library (nn.py)

For quick setup, you can use the included requierments.txt file.

Global Simulation Parameters

These are the global constants that define the simulation and their default values:

1. INT_CONST = 1 - Energy cost for intelligence (IQ, EQ)
2. MOV_CONST = 5 - Energy cost for moving
3. ENLB_CONST = 0.6 - energy to mass ratio used to determine if an agent is sick, if so, the agent starts to lose health
4. ENGB_CONST = 0.4 – energy to mass ratio used to determine if an agent is healthy, used to gain health and to gain mass if the agent is still growing
5. ENL_CONST = 1 - the amount of health lost if an agent's energy is under the threshold.
6. ENG_CONST = 4 - the amount of energy gained if an agent is above the energy threshold
7. MAX_LIFE_SPAN = 200 - the maximum number of steps an agent with mass 100 can survive after reaching maturity
8. AGE_CONST = ENG_CONST - (100 / MAX_LIFE_SPAN) - Age suffered every step, derived from the maximum life span and the amount of health an agent can gain per turn.
9. POP_DENCITY = 1 - controls how concentrated is the populous (scales the map size)
10. AGING_TIME = 0.99 - the mass percentage required before maturity (age starts)
11. G_SPEED_FACTOR = 10 - controls universal speed, provides a way to control gravity and friction
12. FOOD_CONST = 50 - how much energy a food item contains
13. START_MASS_P = 0.95 – The mass to final mass ratio for the initial population
14. G _COL_CONST = 0.1 - the global accuracy for collisions (actual is scaled)
15. MIN_IQ = 1 - the minimum number of hidden neurons (per layer).
16. MAX_IQ = 10 – the maximum number of hidden neurons (per layer)
17. MIN_EQ = 1– the minimum number of hidden neurons (per layer)
18. MAX_EQ = 10 - the maximum number of hidden neurons (per layer)
19. FOOD_FLUCT = 1 - 0.3 – the percentage the food count is allowed to dip before food is reproduced.

20. GROUP_FACTOR = 100 - A factor applied to the collision constant to define the minimum distance between 2 agents that are in the same group

Simulation Specific Variables

- size_factor = 1 / (agents / POP_DENCITY)
- col_const = G _COL_CONST * self.size_factor

The rest of the simulation specific variables are either used to keep track of statistics or are necessary for internal calculations (e.g.: agents).

Agent Traits and Properties

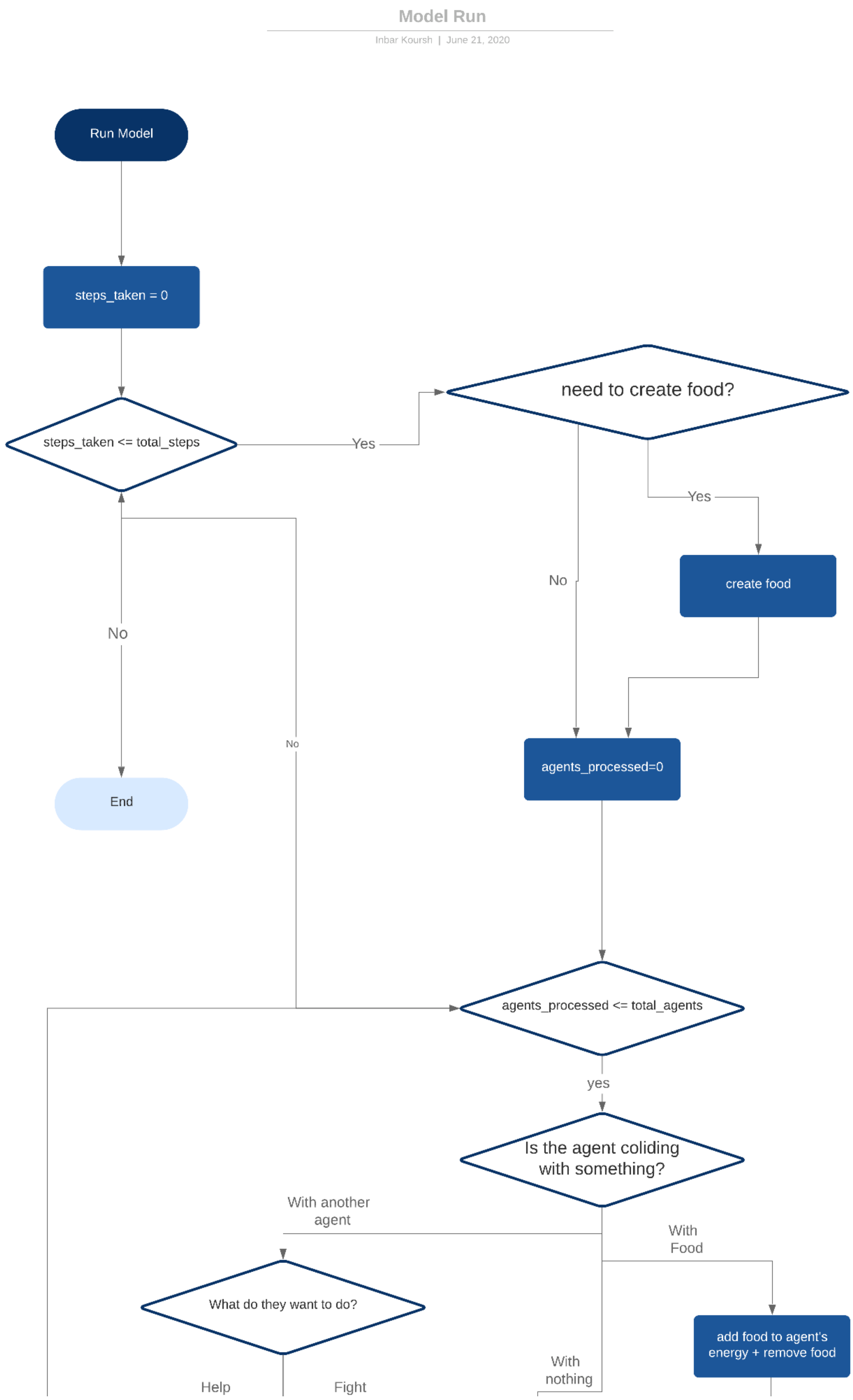
These are the agent traits tracked by the algorithm:

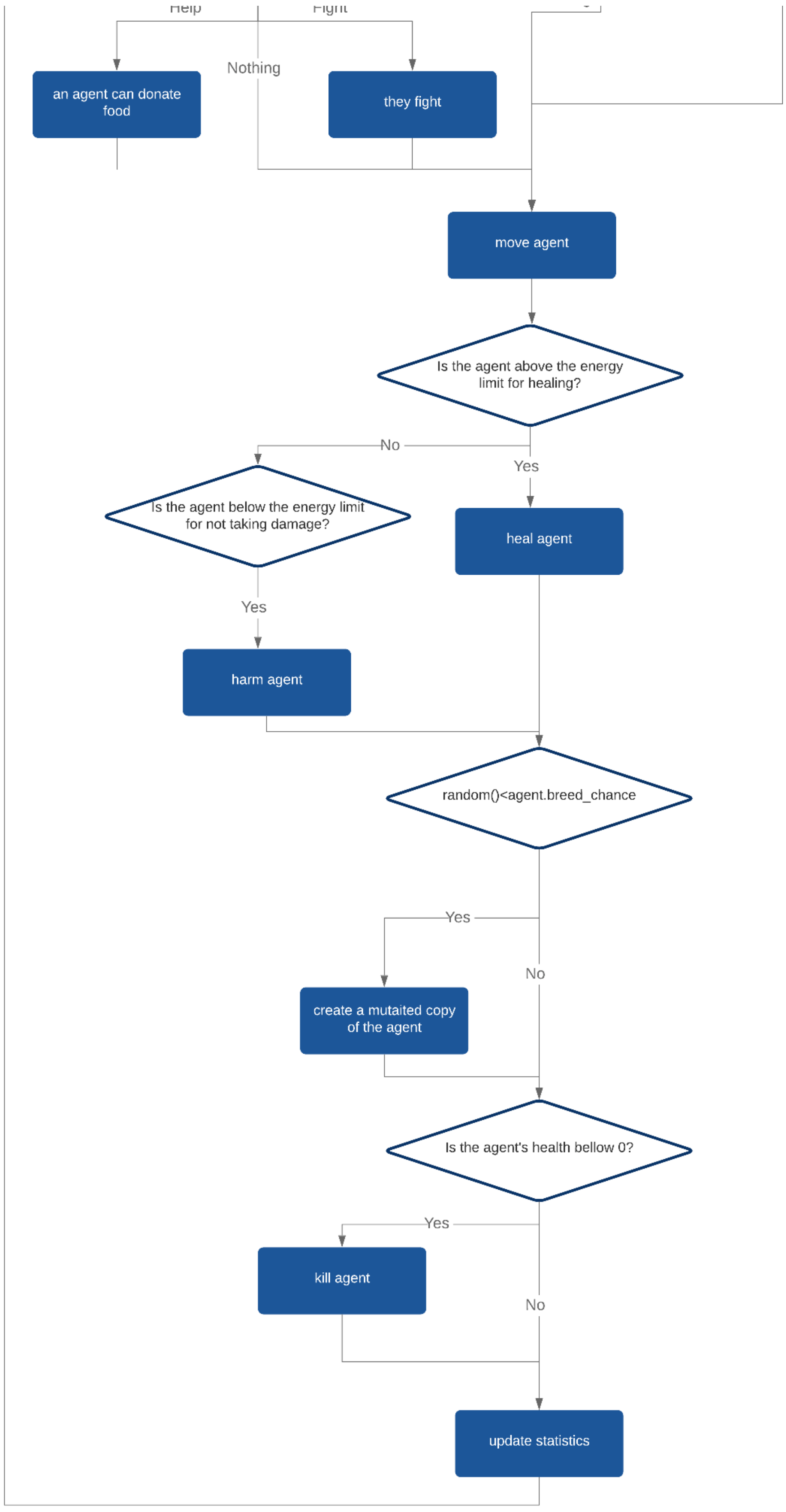
1. Movement brain - the neural network in charge of moving the agent. Structure [3, IQ, IQ, 1], inputs[distance to the nearest food item, distance to the nearest agent, is the nearest agent stronger], output - a value between 0 and one, that is mapped between ±max speed, applied to the agents x coordinate. Activation function: sigmoid
2. IQ - this determines the number of hidden neurons per hidden layer in the neural network in charge of moving the agent.
3. Social brain - the neural network in charge of agent interactions. Structure [4, EQ, EQ, 2], inputs[if they are close family, the agent's energy to mass ratio, the other agent's food to mass ratio, what agent is stronger], output [probability of the agent attacking, probability of the agent assisting (donating 1 food item)]. Activation function: sigmoid
4. EQ - this determines the number of hidden neurons per hidden layer in the neural network in charge of agent interactions
5. Mass - the mass of the agent. Note that the agent's mass will grow over time as it gets older.
6. Final mass - indicates when the creature reaches maturity. It is the maximum value a creature mass can take. However, it is also a requirement for reproduction, a creature must reach this a factor of this final mass before it can reproduce.

7. Energy - the amount of remaining energy the agent has. It starts equal to the agent's mass. Every time it moves or thinks, energy is deducted. It can gain or lose health each step depending on if his energy is above or below a certain level.
8. Speed - the maximum distance an agent can travel each turn. See Movement Brain (1). Speed has an inverse relation to mass (1/mass). Additionally, speed is scaled by the environment's size factor and global speed factor. See constants.
9. Health - the creature's "life". If it is below 0, the creature is removed from the simulation. This is effectively the creature's fitness function.
10. Breed Mass Divider - the initial mass of an agent's child relative to its actual mass.
11. Breed Chance - the chance of the agent to breed every step, note that breeding requires energy.
12. Position (X) - the creature's X coordinate on the simulation. This dictates his interaction with other agents, as well as if he is close enough to pick up a food item.
13. ID - helper variable used by the simulation to keep track of agents as well as to monitor close family.
14. Parent ID - used to assert if 2 agents are close family (parent/child or siblings)
15. Size factor - helper variable used by the sim's movement and collision engines to control the size of the simulation.

Model Architecture Outline

This is a rough overview of a model run


Model Run
Inbar Koursh | June 21, 2020
Run Model
steps_taken = 0
steps_taken <= total_steps
Yes
need to create food?
Yes
create food
No
No
End
No
agents_processed=0
agents_processed <= total_agents
yes
Is the agent coliding with something?
With another agent
With Food
What do they want to do?
add food to agent's energy + remove food
With nothing
Help
Fight


Help
Fight
Nothing
an agent can donate food
they fight
move agent
Is the agent above the energy limit for healing?
No
Yes
Is the agent below the energy limit for not taking damage?
heal agent
Yes
harm agent
random()<agent.breed_chance
Yes
No
create a mutaited copy of the agent
Is the agent's health bellow 0?
Yes
kill agent
No
update statistics

Reproduction Mechanism

Because of the unique makeup of the agents, a slightly abnormal reproduction system is used.

First, an agent must "decide" that it wants to reproduce. This is decided according to that agent's breed chance (which conveys the probability that it will reproduce).

Then both of the agent's "brains" are copied and then mutated. Mutation to the neural networks is done by adding to each neuron a gausian distribution scaled by 0.1.

The final mass of the new creature is mutated by +-10. Its initial mass is its final mass multiplied by its parents "breed mass divider". Its position is relatively close to its parent. And it's breed mass and breed mass divider are mutated by a uniform random distribution spanning 0.01.

Complexity and Optimizations:

Because a collision detection system needs to know the closest agent and food item to every agent, the original algorithm had an $O(n^2)$ complexity. It used the following structure:

```
For every agent:
        For every agent:
                …
        For every food item:
                …
```

Because of the immense amount of time this algorithm required, I optimized it to have an O(n) complexity. This is achieved via the use of sorting functions. A full explanation can be found in the code snippets explanation which provides a detailed explanation for the unintuitive parts of the algorithm.

To test the time improvement, I modeled the time needed for each step of a non-optimized sim (each step is done on a newly generated sim).

Note that these results portray a worst-case scenario where the agents are randomly ordered instead of slightly unordered.

Fig. 1 represents an overview of the newer and older models and the time needed to calculate a step for any number of agents.

Fig. 2 represents a close up of the faster model and shows the average compute time of 5 tests with the top and bottom bars representing the maximum and minimum values in those tests.

Fig 1.

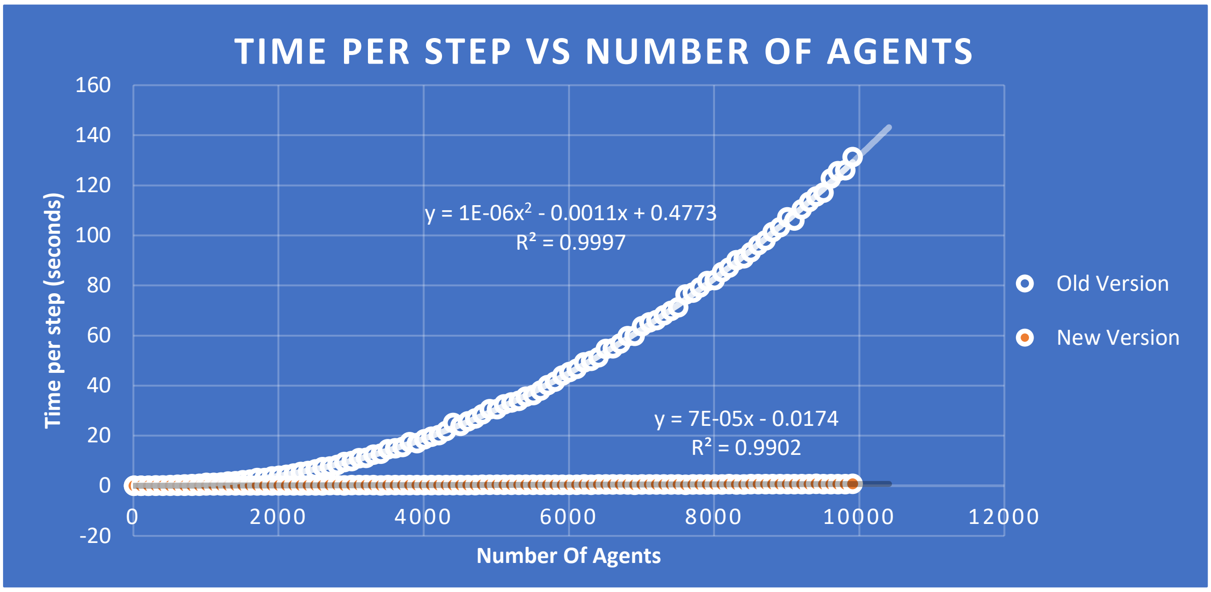
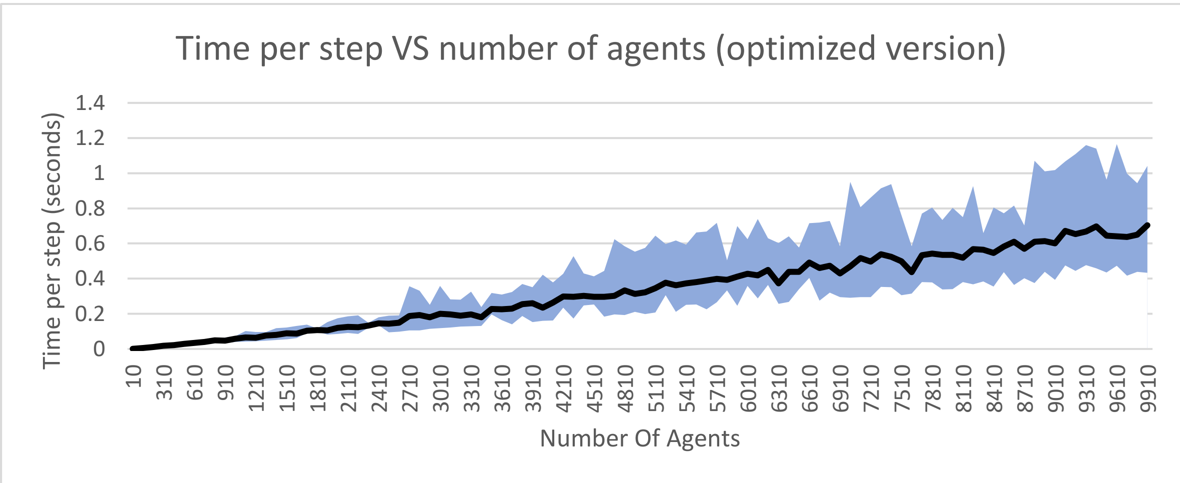


Fig. 2



Even so, it can be seen, that the newer version drastically outperforms its predecessor.

Code Snippets Explanations

These are parts of the code that might require additional explanation to understand

- *Collision detection*

  As you can see in the complexity section of the algorithm, it is O(n) instead of O(n^2). I was able to do this by sorting the agents and the food items by position. Finding the closest agent is easy, simply calculate the distance to the next and previous agent, from there you can check if either of them is smaller than the minimum collision distance. With food items, it's a bit trickier, you keep track of the current food index (starting at 0) then, for every agent, loop from the food index until you find a food item with an x coordinate larger than the agents. Then move the food index 1 back. Now, similar to the agent scenario, you compare the distance to the 2 nearest food items. This is, of course, a general overview, in practice, you need to take into consideration what happens if the food is eaten and the circular map. I contemplated whether to allow multiple interactions at once, this would be achieved by searching left and right until no match is detected as opposed to simply searching the closest 2. I decided against this course of action for now, but I leave this as a potential improvement for further versions.

- *Group detection*

  This is relatively straightforward compared to the others. It is severely limited and cannot act intelligently or keep track of groups as they evolve and clash. Instead, it simply defines a maximum distance to the previous agent where you are still considered part of the group.

- *Neural Network optimization*

  Because the neural networks of the agents sometimes have concurrent 1 neuron long layers, they are removed. This improves the overall runtime performance while having no impact on the cognitive performance of the neural networks.

Statistics

SPSS was used to run statistical correlations and general analysis on the data generated by 3 successive simulations.

The bivariate correlation was done using Person`s r.

Results

The following results are the processed outputs of 3 simulations. The initial agent count was set at 500. Each simulation ran for 1,000,000 steps.

Tables 1, 3, and 5 present the bivariate correlation between the average of the following variables (the average is calculated per step): number of steps, number of agents, average agent mass, amount of food consumed, average agent IQ, average agent EQ, average agent breeding mass divider, average agent breed chance, fight count relative to population size, help count relative to population size, ignore count relative to population size, number of groups, and close family ratio in groups for sims 1,2 and 3 respectively.

Tables 2, 4, and 6 present basic statistical analysis for each of the averages of the variables described in tables 1, 3, and 5. Specifically, they track: mean, median, Std. Deviation, range, minimum, and maximum.

Figures 3, 4, and 5 present the values of the previously mentioned variables, over the span of the simulation. While they are insightful and help instill a general idea as to the simulation's properties, they have not been studied in detail. Instead, more general statistics are used, such as the total mean and median of each variable for each simulation, and the correlation between simulation variables. If you wish to see a higher resolution version of the graphs, please see the PNG files here.

Figure 6 represents an animation of a separate simulation spanning 1,000 steps. It is only presented as a proof of concept to allow for intuitive visual understanding and is not meant for meaningful analysis.

The raw SPSS files along with the plt charts will be available in the GitHub repository accompanying this project. See appendix 2.

Sim 1

Elapsed time (spell free CPU server) - 18D 11H 9M 37S

Table. 1

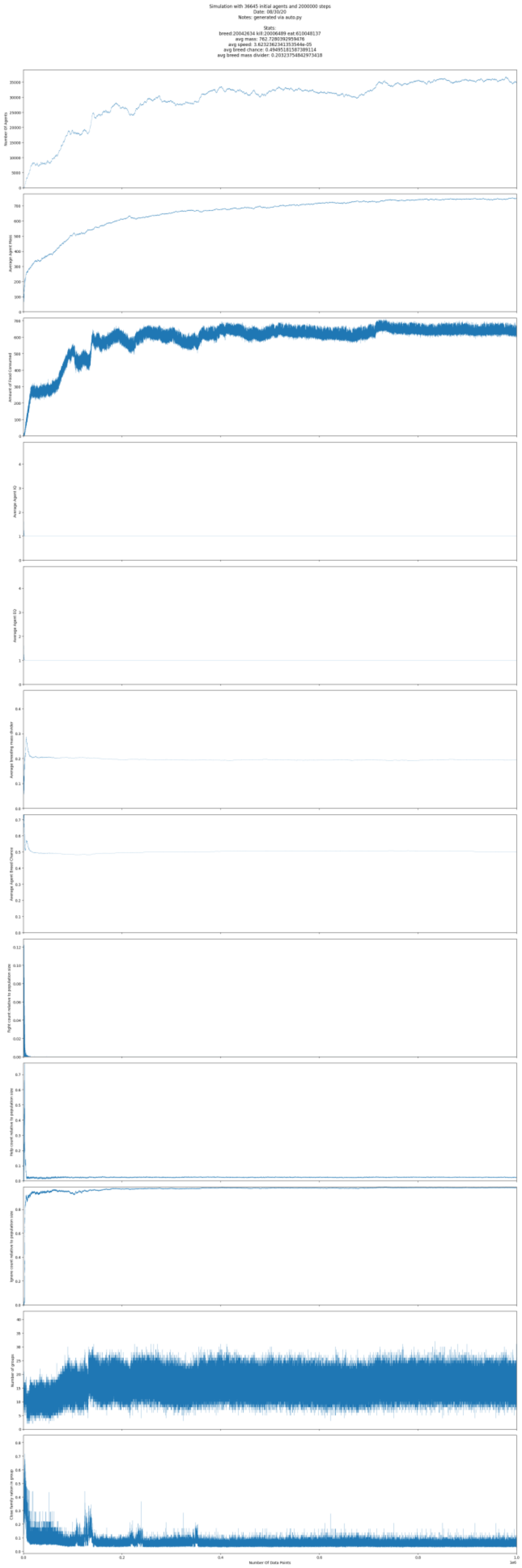
		Steps	Number Of Agents	Average Agent Mass	Amount of Food Consumed	Average Agent IQ	Average Agent EQ	Average breeding mass divider	Average Agent Breed Chance	Fight count relative to population size	Help count relative to population size	Ignore count relative to population size	Number of groups	Close family ration in group
Steps	Pearson Correlation	1	.749**	.730**	.523**	-.025**	-.025**	.241**	-.220**	-.055**	-.075**	.245**	.096**	-.269**
	Sig. (2-tailed)		0.000000E+0	0.000000E+0	0.000000E+0	2.0676E-134	2.0676E-134	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Number Of Agents	Pearson Correlation	.749**	1	.973**	.921**	-.070**	-.070**	-.168**	-.058**	-.157**	-.164**	.451**	.254**	-.506**
	Sig. (2-tailed)	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Average Agent Mass	Pearson Correlation	.730**	.973**	1	.909**	-.096**	-.096**	-.208**	-.065**	-.196**	-.195**	.490**	.227**	-.516**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Amount of Food Consumed	Pearson Correlation	.523**	.921**	.909**	1	-.104**	-.104**	-.297**	-.113**	-.231**	-.245**	.527**	.304**	-.589**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Average Agent IQ	Pearson Correlation	-.025**	-.070**	-.096**	-.104**	1	1.000**	.357**	.151**	.290**	.047**	-.382**	.057**	.146**
	Sig. (2-tailed)	2.0676E-134	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Average Agent EQ	Pearson Correlation	-.025**	-.070**	-.096**	-.104**	1.000**	1	.357**	.151**	.290**	.047**	-.382**	.057**	.146**
	Sig. (2-tailed)	2.0676E-134	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Average breeding mass divider	Pearson Correlation	.241**	-.168**	-.208**	-.297**	.357**	.357**	1	-.373**	-.254**	-.186**	.099**	-.085**	.192**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Average Agent Breed Chance	Pearson Correlation	-.220**	-.058**	-.065**	-.113**	.151**	.151**	-.373**	1	.593**	.443**	-.607**	-.020**	.306**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	4.43363E-87	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Fight count relative to population size	Pearson Correlation	-.055**	-.157**	-.196**	-.231**	.290**	.290**	-.254**	.593**	1	.554**	-.789**	-.014**	.339**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	1.13972E-45	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Help count relative to population size	Pearson Correlation	-.075**	-.164**	-.195**	-.245**	.047**	.047**	-.186**	.443**	.554**	1	-.722**	-.033**	.503**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	1.3573E-237	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Ignore count relative to population size	Pearson Correlation	.245**	.451**	.490**	.527**	-.382**	-.382**	.099**	-.607**	-.789**	-.722**	1	.085**	-.569**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Number of groups	Pearson Correlation	.096**	.254**	.227**	.304**	.057**	.057**	-.085**	-.020**	-.014**	-.033**	.085**	1	-.573**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	4.43363E-87	1.13972E-45	1.3573E-237	0.000000E+0		0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
Close family ration in group	Pearson Correlation	-.269**	-.506**	-.516**	-.589**	.146**	.146**	.192**	.306**	.339**	.503**	-.569**	-.573**	1
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000

**. Correlation is significant at the 0.01 level (2-tailed).

Table. 2

		Number Of Agents	Average Agent Mass	Amount of Food Consumed	Average Agent IQ	Average Agent EQ	Average breeding mass divider	Average Agent Breed Chance	Fight count relative to population size	Help count relative to population size	Ignore count relative to population size	Number of groups	Close family ratio in group
N	Valid	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	0
	Missing	0	0	0	0	0	0	0	0	0	0	0	1000000
Mean		32645.80	707.3483662	610.0481370	1.000523417	1.000523417	.1975626697	.4992347051	.0000476249	.0231473024	.9749230900	16.10122700	
Median		34788.00	745.9694299	631.0000000	1.000000000	1.000000000	.1962600826	.4995908947	.0000000000	.0226543177	.9794606486	16.00000000	
Std. Deviation		6595.006	95.17685043	82.65808613	.0367404014	.0367404014	.0059629664	.0084273072	.0014921066	.0138740283	.0363477334	3.432059713	
Range		39104	727.2776330	716.0000000	3.915625000	3.915625000	.4161827403	.2501035345	.1290322581	.7763157895	.9865538366	42.00000000	
Minimum		44	51.47084233	.0000000000	1.000000000	1.000000000	.0563782873	.4817307825	.0000000000	.0000000000	.0000000000	1.000000000	
Maximum		39148	778.7484754	716.0000000	4.915625000	4.915625000	.4725610277	.7318343171	.1290322581	.7763157895	.9865538366	43.00000000	

Fig. 3


Simulation with 36645 initial agents and 2000000 steps
Date: 08/30/20
Notes: generated via auto.py
Stats:
breed:20042634 kill:20006489 eat:610048137
avg mass: 762.7280392959476
avg speed: 3.6232362341353544e-05
avg breed chance: 0.49495181587389114
avg breed mass divider: 0.20323754842973418
Number Of Data Points

Sim 2

Elapsed time (spell free CPU server) - 3H 12M 9S

Table 3.

		Steps	Number of Agents	Average Agent Mass	Amount of Food Consumed	Average Agent IQ	Average Agent EQ	Average breeding mass divider	Average Agent Breed Chance	Fight count relative to population size	Help count relative to population size	Ignore count relative to population size	Number of groups	Close family ratio in group
Steps	Pearson Correlation	1	-.089**	.519**	-.047**	-.027**	-.027**	.937**	-.300**	-.087**	-.066**	.140**	-.498**	.280**
	Sig. (2-tailed)		0.000000E+0	0.000000E+0	0.000000E+0	2.8897E-165	2.8897E-165	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Number of Agents	Pearson Correlation	-.089**	1	-.630**	.934**	-.015**	-.015**	.088**	.020**	-.017**	-.023**	.482**	-.070**	.505**
	Sig. (2-tailed)	0.000000E+0		0.000000E+0	0.000000E+0	1.95804E-52	1.95804E-52	0.000000E+0	3.78940E-85	2.37500E-67	6.7438E-119	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Average Agent Mass	Pearson Correlation	.519**	-.630**	1	-.575**	-.017**	-.017**	.318**	.004**	-.052**	-.040**	-.323**	-.335**	-.157**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0		0.000000E+0	5.66859E-62	5.66859E-62	0.000000E+0	1.070452E-4	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Amount of Food Consumed	Pearson Correlation	-.047**	.934**	-.575**	1	-.013**	-.013**	.120**	.008**	-.015**	-.020**	.476**	-.111**	.491**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0		2.87516E-40	2.87516E-40	0.000000E+0	1.23063E-15	9.40253E-53	8.69632E-87	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Average Agent IQ	Pearson Correlation	-.027**	-.015**	-.017**	-.013**	1	1.000**	.061**	-.322**	.350**	.396**	-.047**	.056**	-.017**
	Sig. (2-tailed)	2.8897E-165	1.95804E-52	5.66859E-62	2.87516E-40		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	6.53072E-66
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Average Agent EQ	Pearson Correlation	-.027**	-.015**	-.017**	-.013**	1.000**	1	.061**	-.322**	.350**	.396**	-.047**	.056**	-.017**
	Sig. (2-tailed)	2.8897E-165	1.95804E-52	5.66859E-62	2.87516E-40	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	6.53072E-66
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Average breeding mass divider	Pearson Correlation	.937**	.088**	.318**	.120**	.061**	.061**	1	-.243**	-.054**	-.027**	.261**	-.469**	.358**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	4.7170E-165	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Average Agent Breed Chance	Pearson Correlation	-.300**	.020**	.004**	.008**	-.322**	-.322**	-.243**	1	-.354**	-.364**	-.009**	.022**	-.048**
	Sig. (2-tailed)	0.000000E+0	3.78940E-85	1.070452E-4	1.23063E-15	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	7.52374E-20	2.4885E-103	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Fight count relative to population size	Pearson Correlation	-.087**	-.017**	-.052**	-.015**	.350**	.350**	-.054**	-.354**	1	.962**	-.046**	.093**	-.061**
	Sig. (2-tailed)	0.000000E+0	2.37500E-67	0.000000E+0	9.40253E-53	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Help count relative to population size	Pearson Correlation	-.066**	-.023**	-.040**	-.020**	.396**	.396**	-.027**	-.364**	.962**	1	-.048**	.079**	-.049**
	Sig. (2-tailed)	0.000000E+0	6.7438E-119	0.000000E+0	8.69632E-87	0.000000E+0	0.000000E+0	4.7170E-165	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Ignore count relative to population size	Pearson Correlation	.140**	.482**	-.323**	.476**	-.047**	-.047**	.261**	-.009**	-.046**	-.048**	1	-.222**	.347**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	7.52374E-20	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Number of groups	Pearson Correlation	-.498**	-.070**	-.335**	-.111**	.056**	.056**	-.469**	.022**	.093**	.079**	-.222**	1	-.403**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	2.4885E-103	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
Close family ratio in group	Pearson Correlation	.280**	.505**	-.157**	.491**	-.017**	-.017**	.358**	-.048**	-.061**	-.049**	.347**	-.403**	1
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	6.53072E-66	6.53072E-66	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	
	N	999550	999550	999550	999550	999550	999550	999550	999550	999550	999550	999550	999550	999550

**. Correlation is significant at the 0.01 level (2-tailed).

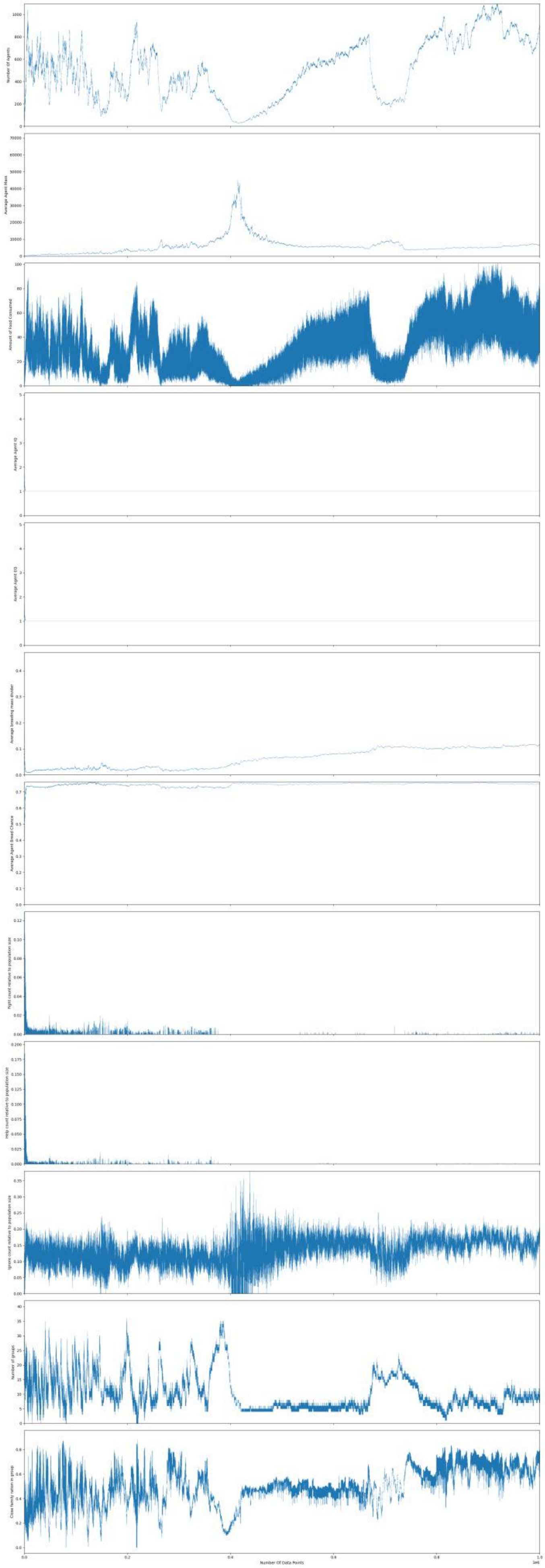
Table 4.

		Number of Agents	Average Agent Mass	Amount of Food Consumed	Average Agent IQ	Average Agent EQ	Average breeding mass divider	Average Agent Breed Chance	Fight count relative to population size	Help count relative to population size	Ignore count relative to population size	Number of groups	Close family ratio in group
N	Valid	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999550
	Missing	0	0	0	0	0	0	0	0	0	0	0	450
Mean		424.5197310	13972.28757	27.52425000	1.000549055	1.000549055	.0988433827	.7403312553	.0000739787	.0000897025	.1303282436	7.463551000	.5356712952
Median		421.0000000	9502.153108	25.00000000	1.000000000	1.000000000	.1076129657	.7440492087	.0000000000	.0000000000	.1343638526	5.000000000	.5256024096
Std. Deviation		274.3366793	13233.26920	19.60046632	.0347066764	.0347066764	.0474813391	.0129776352	.0014084915	.0023309005	.0402967390	5.703148779	.1840700737
Range		1073.000000	72632.91304	101.0000000	4.084507042	4.084507042	.4629336375	.3951758832	.1294117647	.2052401747	.3797468354	42.00000000	.9113190731
Minimum		23.00000000	52.22489083	.0000000000	1.000000000	1.000000000	.0081897683	.3665264462	.0000000000	.0000000000	.0000000000	.0000000000	.0000000000
Maximum		1096.000000	72685.13793	101.0000000	5.084507042	5.084507042	.4711234058	.7617023294	.1294117647	.2052401747	.3797468354	42.00000000	.9113190731

Fig 4.

Simulation with 78 initial agents and 2000000 steps
Date: 08/14/20
Notes: generated via auto.py

Stats:
breed:261410 kill:261832 eat:27524250
avg mass: 39058.307692307695
avg speed: 3.5161835602857863e-06
avg breed chance: 0.7476215690701774
avg breed mass divider: 0.14632261510151898



Sim 3

Elapsed time (spell free CPU server) - 4D 9H 36S

Table. 5

		Steps	Number of Agents	Average Agent Mass	Amount of Food Consumed	Average Agent IQ	Average Agent EQ	Average breeding mass divider	Average Agent Breed Chance	Fight count relative to population size	Help count relative to population size	Ignore count relative to population size	Number of groups	Close family ration in group
Steps	Pearson Correlation	1	.962**	.888**	.884**	-.022**	-.022**	.776**	.095**	-.084**	.752**	.866**	-.092**	-.271**
	Sig. (2-tailed)		0.000000E+0	0.000000E+0	0.000000E+0	1.5228E-111	1.5228E-111	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Number of Agents	Pearson Correlation	.962**	1	.872**	.952**	-.032**	-.032**	.732**	.120**	-.106**	.759**	.855**	-.153**	-.359**
	Sig. (2-tailed)	0.000000E+0		0.000000E+0	0.000000E+0	1.9828E-230	1.9828E-230	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Average Agent Mass	Pearson Correlation	.888**	.872**	1	.849**	-.058**	-.058**	.841**	-.125**	-.176**	.767**	.930**	-.175**	-.124**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Amount of Food Consumed	Pearson Correlation	.884**	.952**	.849**	1	-.043**	-.043**	.696**	.058**	-.129**	.716**	.816**	-.204**	-.375**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Average Agent IQ	Pearson Correlation	-.022**	-.032**	-.058**	-.043**	1	1.000**	.404**	-.470**	.182**	-.034**	-.111**	.062**	.009**
	Sig. (2-tailed)	1.5228E-111	1.9828E-230	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	1.3298E-257	0.000000E+0	0.000000E+0	3.67404E-18
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Average Agent EQ	Pearson Correlation	-.022**	-.032**	-.058**	-.043**	1.000**	1	.404**	-.470**	.182**	-.034**	-.111**	.062**	.009**
	Sig. (2-tailed)	1.5228E-111	1.9828E-230	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	1.3298E-257	0.000000E+0	0.000000E+0	3.67404E-18
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Average breeding mass divider	Pearson Correlation	.776**	.732**	.841**	.696**	.404**	.404**	1	-.427**	.018**	.707**	.775**	-.078**	-.047**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	8.00933E-69	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Average Agent Breed Chance	Pearson Correlation	.095**	.120**	-.125**	.058**	-.470**	-.470**	-.427**	1	-.113**	-.115**	-.090**	.083**	-.227**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Fight count relative to population size	Pearson Correlation	-.084**	-.106**	-.176**	-.129**	.182**	.182**	.018**	-.113**	1	.073**	-.277**	.055**	.004**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	8.00933E-69	0.000000E+0		0.000000E+0	0.000000E+0	0.000000E+0	8.626659E-5
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Help count relative to population size	Pearson Correlation	.752**	.759**	.767**	.716**	-.034**	-.034**	.707**	-.115**	.073**	1	.812**	-.007**	-.169**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	1.3298E-257	1.3298E-257	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	1.10530E-12	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Ignore count relative to population size	Pearson Correlation	.866**	.855**	.930**	.816**	-.111**	-.111**	.775**	-.090**	-.277**	.812**	1	-.061**	-.133**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0		0.000000E+0	0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Number of groups	Pearson Correlation	-.092**	-.153**	-.175**	-.204**	.062**	.062**	-.078**	.083**	.055**	-.007**	-.061**	1	-.135**
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	1.10530E-12	0.000000E+0		0.000000E+0
	N	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
Close family ration in group	Pearson Correlation	-.271**	-.359**	-.124**	-.375**	.009**	.009**	-.047**	-.227**	.004**	-.169**	-.133**	-.135**	1
	Sig. (2-tailed)	0.000000E+0	0.000000E+0	0.000000E+0	0.000000E+0	3.67404E-18	3.67404E-18	0.000000E+0	0.000000E+0	8.626659E-5	0.000000E+0	0.000000E+0	0.000000E+0	
	N	999998	999998	999998	999998	999998	999998	999998	999998	999998	999998	999998	999998	999998

**. Correlation is significant at the 0.01 level (2-tailed).

Table. 6

		Number of Agents	Average Agent Mass	Amount of Food Consumed	Average Agent IQ	Average Agent EQ	Average breeding mass divider	Average Agent Breed Chance	Fight count relative to population size	Help count relative to population size	Ignore count relative to population size	Number of groups	Close family ration in group
N	Valid	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	999998
	Missing	0	0	0	0	0	0	0	0	0	0	0	2
Mean		6814.29	1582.573095	261.7613720	1.000458672	1.000458672	.0556022766	.9228355159	.0001227537	.0802587796	.7728563072	12.13154100	.2611748256
Median		6833.00	1649.639629	263.0000000	1.000000000	1.000000000	.0594981904	.9210438871	.0000000000	.0827962938	.7974492364	12.00000000	.2563353998
Std. Deviation		2663.685	335.2913388	76.58496167	.0353978979	.0353978979	.0111601734	.0101707889	.0017326552	.0118894032	.0898872933	3.532779988	.1105391846
Range		12846	1982.265633	472.0000000	3.674518201	3.674518201	.4766288511	.4725134421	.1789473684	.3578947368	.8775796599	44.00000000	.9211668104
Minimum		41	50.43897216	.0000000000	1.000000000	1.000000000	.0157544666	.4812349089	.0000000000	.0000000000	.0000000000	.0000000000	.0307851415
Maximum		12887	2032.704605	472.0000000	4.674518201	4.674518201	.4923833176	.9537483510	.1789473684	.3578947368	.8775796599	44.00000000	.9519519520

Fig. 5

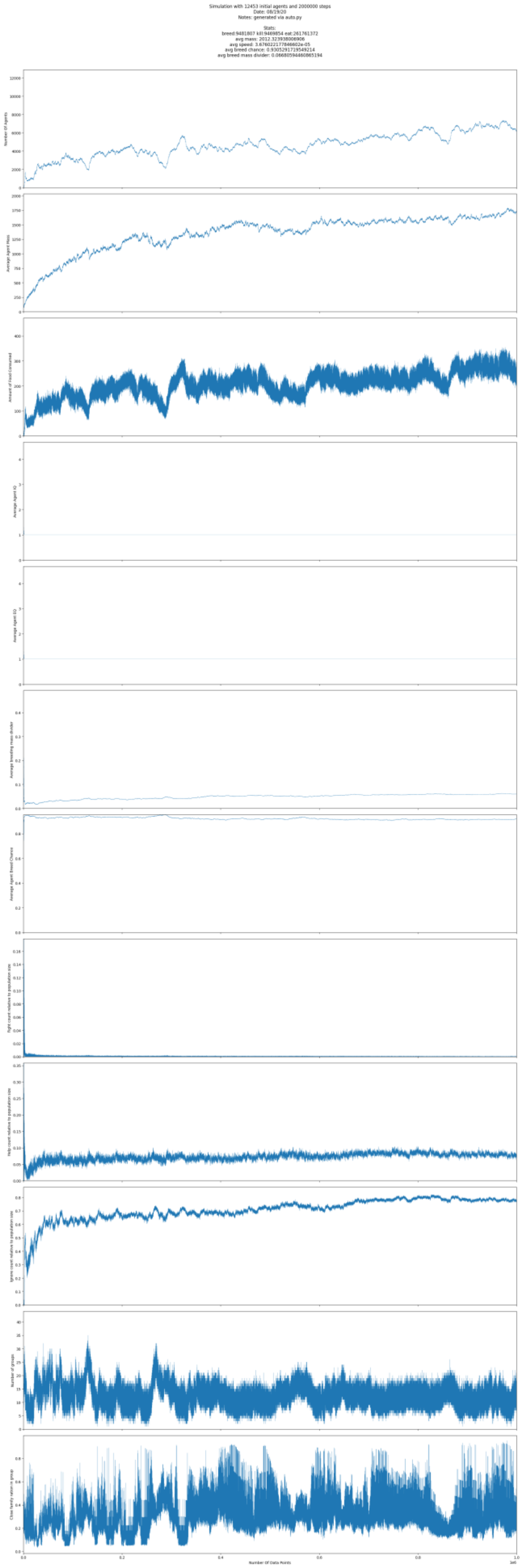
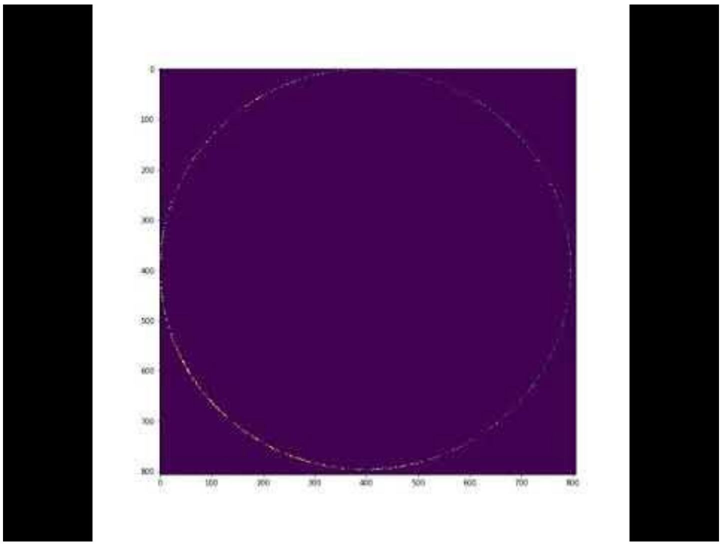

Simulation with 12453 initial agents and 2000000 steps
Date: 08/19/20
Notes: generated via auto.py
Stats:
breed:9481807 kill:9469854 eat:261761372
avg mass: 2012.323938006906
avg speed: 3.676022177846602e-05
avg breed chance: 0.9305291719549214
avg breed mass divider: 0.06680594460865194

Fig. 6



Discussion

My objective was to create a computational model that can be used to describe the evolution of an ecosystem. As such, a large part of my results is the algorithm itself, that can be used to obtain much greater insights into the behavior of populations than I was able to.

Getting a stable populous is more challenging than it seems. At first, the population skyrocketed into the tens of thousands and quickly overwhelmed my computer's capabilities. Other times, they quickly died, leaving no obvious reason for their sudden demise. As it turns out I had broken one of the most basic laws of physics, and as such created a “hack”. I had forgotten to regulate the reproduction mechanic and allowed for blatant violations of the conservation of mass and energy. Yet even with those fixed, A lot of time was spent fine-tuning the parameters to allow for stable growth. This included implementing a mechanism that allowed for scaling the simulation. Because the simulation’s spatial coordinates range from -1 to +1, I had to adjust the necessary distance for collision, as well as the maximum speed of the agents as to virtually scale the coordinate system without actually doing so. Another such alteration was the addition of many variables to the creature's traits, most prominently “breed chance” and “breed mass divider”. This had the added benefit of making the simulation much more realistic and indicative of the real world, but its primary objective was to remove the need to manually tune these parameters.

To analyze the results from the simulations I performed a bivariate correlation between all the variables using Person's r. It appears that they are all at least loosely correlated to one another (correlation is significant at the 0.01 level - 2-tailed). This is fascinating. Even though the correlation itself might be weak, this demonstrates that all factors in an ecosystem drastically affect the others.

I would argue that simulations a and c are more successful than simulation b. This is because the total number of agents, as well as food consumed, is higher. Because they use more of their available resources, I consider them to be better. A factor that I found works well for measuring the success of the simulation is the time it takes to complete. Sim b measured at ~3H, simulation c was ~4D while simulation a took a staggering ~18D to complete.

The differences between simulations a and c to simulation b will become more apparent once we start examining the correlations.

Some of these correlations are expected. For example:

- <u>The correlation between the number of agents and the food consumed</u>

  ρ_a=0.921 ρ_b=0.934 ρ_c=0.952

  This is mostly self-explanatory. The more food consumed the higher the agent's health, the more children it can have.

- <u>Average agent mass and close family ratio in group</u>

  ρ_a=-0.516 ρ_b=-0.124 ρ_c=-0.157

  This is a loose correlation between the average agent mass and the close family ratio in the group. While they may seem unconnected at first, the more massive an agent is the more offspring it has. This effect is however counteracted by the dilution of family members by the increased number of agents in groups. To get a better understanding of the strength of this correlation we can look to the STD. deviation of the mass in each simulation. Unfortunately, STD over time was not tracked and is a potential improvement for the future.

These correlations indicate that the simulation is at least partially successful as it produces results that are expected using common intuition.

I will now cover some more interesting correlations.

- Agent breed mass divider with the progress of the simulation (Steps)

  ρ_a=0.241 ρ_b=0.937 ρ_c=0.776

  This makes a lot of sense to me. I propose that at the beginning of the simulation, before agents were able to overcome the constant food search, having children was a big risk and so the new children were given limited mass. But as time went on the risk decreased and agents could now afford to gamble a lot of their health. It is important to note that although I talk about the agent taking risks, I am referring to the concept of the selfish gene. The continuation of the gene is dependent on its current carrier but also on its offspring. Although genes are not used directly in this model, the general concept still applies.

  The reason that the first simulation does not fall under the same pattern would be because of its lower breed chance compared to the other two. This allowed it to mostly start on 0.2 instead of reaching it after a prolonged period.

- Close family and number of agents

  ρ_a=-0.506 ρ_b=0.505 ρ_c=-0.359

  This is interesting because the correlations appear to be reversed. ρ_b is expected as having more close family seems to increase the chances of survival as family members are less likely to attack other family members. However, I propose that the reason for the correlation we observe in sims a and c is because of the increased number of agents diluting the close family ratio. The more agents, the less close family.

  I believe that the reason close family affects the number of agents and not the other way around in sim b is because of the fewer number of agents that increase the effect of each agent and its family. This is contrary to sims a and c where a large number of agents appear to dilute most of the advantages.

- Amount of food consumed and average agent mass

  ρ_a=0.908 ρ_b=-0.575 ρ_c=0.849

  Here the correlations between sim a and sim c to sim b are also reversed. While ρ_c and ρ_a seem expected, the more food consumed the higher the average mass, ρ_b is the exact opposite.

  The number of agents does not seem to be the cause because they both have similar ratios regarding the number of agents and food consumed: ρ_a=0.921 ρ_b=0.934 ρ_c=0.952. In fact, simulation c seems to have more agents when there is more food, yet still, its agents have a higher agent mass.

  I propose that the reason for this is that the second simulation was too erratic for such a correlation. Because food only affects an agent's mass directly if he is still growing and agent mass itself is constantly changing due to childbirth. The number of agents in sim2 hover around 424 on average, while the number in sim3 is on average 6,833.

  This is in addition to factors like the high agent mass in sim 2 ~9,502 compared to sim 3 ~1,582 which would reduce the effects of factors like food consumed on the average mass.

Another interesting observation is that agents chose to mostly ignore one another instead of fighting. I have been unable to determine the reason for this behavior although I suspect it arises from a mistake in the fight mechanic where fighting might have more drawbacks than rewards. It is however possible that the agents with a higher tendency to fight simply killed themselves. This is in line with the idea that animals refrain from cannibalism.

I would suggest that in the future, more analysis should be done on the differences between the simulations, as well as ground the global factors in reality.

Assumptions and Limitations

Throughout the project, I have made several key assumptions.

- Agents cannot learn. At first glance, this may seem like a glaring flaw, as I'm treating agents as individuals yet without the ability to learn. However, I would argue that the learning process of an individual is largely irrelevant because opposed to natural agents, these agents can pass much more information via their mutated cloned brains to their children.

- Agents do not need a partner to reproduce. They reproduce via parthenogenesis, that is to say, that they reproduce alone. This has distinct drawbacks as the resulting children are only modified versions of their parent instead of blending 2 different genes. Unfortunately, this system is set up in a way that cannot be easily transferable to two parents. This is mostly because the neural networks that each creature possesses are slightly different and are not guaranteed to have the same number of neurons per hidden layer.

- I make several assumptions concerning the way that mass governess the ability of the creatures. I make no allowances for muscle mass to body mass ratio and seem to be giving a large mass a great advantage. I would address this by saying that one can look at the variable I call mass, as a number that aggregates mass, as well as muscle mass to body mass ratio. This would become a concern however if mass data were to be used to corroborate real-world data or to test the validity of this model and run experiments. As such future simulation should make use of 2 different variables instead of relying solely on one.

$$a = \frac{F}{m}$$

$$F = m\ k^{*}$$

$$a = \frac{\cancel{m}k}{\cancel{m}} = k$$

As you can see, an agent’s acceleration is not related to its mass but rather to its muscle mass to body mass ratio. In future versions, this can be resolved by using 2 variables for mass.

* k is the muscle mass to body mass ratio

** distance is measured for every step, as such the time for each step is defined to be 1 arbitrary time unit

- Agents age in a constant fashion, regardless of their mass. This, in turn, diminishes the effect of aging on larger creatures. However, life expectancy does seem to be linearly correlated to mass.[3]
- The sim only uses 1 axis. While a 2 or even 3 axis system might represent reality better, I believe that the computational advantages outweigh the drawbacks. Even though 2 dimensions would allow for more advanced hunting strategies and better evasion. To partially compensate for this, agents can “jump” over other agents, in other words – the simulation doesn't check to see if there is an agent in the way before a creature moves. This would, however, be an avenue for further improvement should anyone express interest in improving the model
- Because the simulation attempts to work via the macro world and not to simulate physics, it necessarily is less accurate. The assumptions I made are closely connected to this fact. And while I have done my best to minimize them, as well justify them to a certain degree, they still present limitations to the accuracy of the simulation. For example, all creatures are equally efficient. While this does not accurately describe the natural world, because all creatures strive for maximum efficiency, I believe that this is acceptable. This assumption allows me to ignore the physics side of things as creating a physics simulation and allowing agents to develop efficient solutions for aerodynamics as well as cell level reactions would be a near-complete waste of computational resources for modeling macro events.

[3] Szekely, P., Korem, Y., Moran, U., Mayo, A., & Alon, U. (2015). The Mass-Longevity Triangle: Pareto Optimality and the Geometry of Life-History Trait Space. PLOS Computational Biology, 11(10), e1004524. https://doi.org/10.1371/journal.pcbi.1004524

- IQ and IQ are immutable. Because of the nature of neural networks, IQ and EQ are completely static. They do not change from parent to child, unlike other traits. I believe that this is the reason that all the agents in every simulation I ran, were reduced to 1 EQ and IQ after the first couple of steps. IQ and EQ produce unwanted strain on the agent's energy repository. In the early development stages, IQ and EQ produce no meaningful advantage, yet I believe that in later stages this may change.

  This poses an interesting challenge, one that is beyond my ability to solve. However, a possible partial solution would be to allow for a steady supply of new random agents throughout the simulation's life cycle. This would not fully resolve the issue because these agents would need to start from scratch, however, it would allow for the introduction of agents with high EQ and IQ.

- The constants of the simulation, while not arbitrary, are not necessarily indicative of the natural world and as such they misrepresent reality. This could probably be overcome by dedicating time to each constant and attempting to approximate it.

- Additionally, while the simulation does keep track of numerous variables and statistics, it does not present the full picture. To partially compensate for this, I created the animate function that creates a rendering of the simulation. However, it is difficult to ascertain any meaningful amount of information beyond the events that occur on the macro scale

- The food class in the simulation always boats the same food reward. This is an interesting avenue for further discussion. Should the food amounts be generated randomly the agent might be provided with multiple food items and it will have to pick intelligently between distance and food amount. Furthermore, the introduction of seasons would also be an interesting factor.

- Agents are not able to maintain a memory of past interactions and communicate information concerning them, nor are they able to identify other agents with whom they interact. This might be fixed by using LSTM or RNN instead of a simple neural network, in addition to ids being passed while interacting.

- Agents can only interact with 1 agent and 1 food item per step. I detail in my code snippet explanation a possible solution that should mostly maintain the efficiency improvements from the sorting solution. Even so, the step function will become an $O(n^2)$ algorithm.

- Certain creatures might have an unfair advantage because of the ordering of the agents. While this is not addressed directly, the order of the agents is not fixed and is related to their position values. To see why, please see Code Snippets Explanations: collision detection. Because of this, and because the analysis is done on the level of the population and not individual agents, I do not see this as a significant problem.

There are of course other limitations and assumptions that I did not mention here or realize.

Conclusion

In conclusion, a simulated ecosystem can grant us new insights into the inner workings of complex interactions between innumerable values. As I have demonstrated there exist loose correlations between all the variables I have studied, most of whom are verified at beyond the 5-sigma level. I was able to gain interesting insights into the mechanics of these interactions and causes and implications

Appendix 1 – Using the simulation

Initial installation (for all use cases)

1. Clone the code from https://github.com/ikoursh/EcoSystemProject by running:

```
git clone https://github.com/ikoursh/EcoSystemProject
```

2. Then enter the model directory:

```
cd ecosystemproject/proj
```

3. Finally, install the dependencies:

```
pip install -r requierments.txt
```

If you want to add support for animation you will need an installation of FFmpeg. To do so you can go here (don’t forget to add it to PATH, this is a great explanation for windows)

Note that pip install ffmpeg does not seem to work, however, Linux users may use apt.

GUI (Windows only officially supported)

This option might be a bit buggy and has not been tested on Linux or Mac. It is however the preferred option for beginners.

Simply install the program using the included installer (ecosystemproject/GUI/dist/EcoSystem-Project-GUI Setup 1.0.0.exe)

Then open the program, and in the settings tab enter the path to auto.py.

You can now create simulations using the new simulation menu. Simulations will automatically display in your “my simulations” menu. Once a simulation has finished you can click on it to see the output files.

Note that EnvSim will attempt to run in the background and can be terminated using the task manager or the slightly buggy tray bar. Once the program is quit the simulation will too and they cannot be resumed.

Note, if you chose to enable spell integration you must commit all local changes before trying to use Spell.

CLI

This option is easy and flexible. Run “python auto.py –help” for more information about the parameters. It is very easy to use and user friendly.

Python

This is the least recommended option. For casual users, the GUI or CLI would be easier, and for advanced users, a custom implementation would probably be better suit their needs.

Nevertheless, for those who want fine control yet simplicity you can import sim2 directly. Please check the API documentation for more information.

* if you attempt to process close family ratio via the generated SAV(SPSS) file you might encounter multiple errors. This can be fixed by rounding the values to fewer decimal points. To do this click transform -> compute variable, enter the name of the new variable and enter RND(Close_family_ratio_in_group *10**x) / 10**x (where x is the precision, 16 is the recommended value).

Appendix 2 - Code Documentation

All of the code in this project is available in this project's GitHub repository along with all previous versions. To see the latest codebase go here: https://github.com/ikoursh/EcosystemProject/

The core code used in the project has been documented via Google's API doc guidelines.

The documentation has been compiled using SPHINX and is available here: http://ecosystem.inbarkoursh.com/ as well as on this project's GitHub repository here: https://github.com/ikoursh/EcosystemProject/tree/master/docs/

Appendix 3 - Code

It is recommended that all code be viewed using this project's GitHub repository and not here.

The code for the GUI as well as the documentation is only available in the GitHub repository.

model2.py:

```
#!/usr/bin/env python3

# Ecosystem project - studying natural biological systems using a simulated ecosystem and reinforcement learning.
# Copyright (C) 2020 Inbar Koursh
```

```
#
# This program is free software: you can redistribute it and/or modify it under
# the terms of the GNU General Public License as published by the Free Software
# Foundation, either version 3 of the License, or (at your option) any later
# version.
#
# This program is distributed in the hope that it will be useful, but WITHOUT
# ANY WARRANTY; without even the implied warranty of  MERCHANTABILITY or FITNESS
# FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License along with
# this program.  If not, see <http://www.gnu.org/licenses/>.

import configparser
import copy
import gc
import os
import random
import time
from typing import Tuple

import matplotlib

import matplotlib.pyplot as plt

import numpy as np
from matplotlib import animation
from nn import NeuralNetwork

from PIL import Image  # work with metadata via pillow
from PIL import PngImagePlugin

import openpyxl  # work with excel
import savReaderWriter  # work with spss

import pickle  # support serialization

import json  # work with GUI

matplotlib.get_backend()

import sys

if sys.version_info[0] < 3 or sys.version_info[1] < 4:
    raise Exception(
        "Python 3.4 or higher is required. Download the latest version here
https://www.python.org/downloads/ ")

if sys.version_info[0] > 3:
    print("WARNING this program was write to support python 3. Use any future versions at your
discretion")

dir_path = os.path.dirname(os.path.realpath(__file__))

config = configparser.ConfigParser(inline_comment_prefixes="#")
config.read(os.path.join(dir_path, 'config.ini'))

INT_CONST = float(config["VARIABLES"]["INT_CONST"])
MOV_CONST = float(config["VARIABLES"]["MOV_CONST"])
```

```python
ENLB_CONST = float(config["VARIABLES"]["ENLB_CONST"])
ENGB_CONST = float(config["VARIABLES"]["ENGB_CONST"])
ENL_CONST = float(config["VARIABLES"]["ENL_CONST"])
ENG_CONST = float(config["VARIABLES"]["ENG_CONST"])
MAX_LIFE_SPAN = float(config["VARIABLES"]["MAX_LIFE_SPAN"])

AGE_CONST = config["VARIABLES"]["AGE_CONST"]
if AGE_CONST == "auto":
    AGE_CONST = ENG_CONST - (100 / MAX_LIFE_SPAN)
else:
    AGE_CONST = float(AGE_CONST)

POP_DENCITY = float(config["VARIABLES"]["POP_DENCITY"])
AGING_TIME = float(config["VARIABLES"]["AGING_TIME"])
G_SPEED_FACTOR = float(config["VARIABLES"]["G_SPEED_FACTOR"])
FOOD_CONST = float(config["VARIABLES"]["FOOD_CONST"])
START_MASS_P = float(config["VARIABLES"]["START_MASS_P"])
G_COL_CONST = float(config["VARIABLES"]["G_COL_CONST"])
MIN_IQ = float(config["VARIABLES"]["MIN_IQ"])
MAX_IQ = float(config["VARIABLES"]["MAX_IQ"])
MIN_EQ = float(config["VARIABLES"]["MIN_EQ"])
MAX_EQ = float(config["VARIABLES"]["MAX_EQ"])
FOOD_FLUCT = float(config["VARIABLES"]["FOOD_FLUCT"])
GROUP_FACTOR = float(config["VARIABLES"]["GROUP_FACTOR"])


def map_from_to(x: float, a: float, b: float, c: float, d: float) -> float:
    """
    Map a value to a new range

    Used in :meth:`model2.Agent.think` to map inputs and outputs to and from ranges that the neural network requires

    Args:
        x(float): Value
        a(float): Initial range start
        b(float): Initial range end
        c(float): Final range start
        d(float): Final range end


    Returns:
        float: Mapped value
    """
    return (x - a) / (b - a) * (d - c) + c


# create folders for model outputs
GRAPHS_FOLDER = "/graphs-0.3"
ANIMATION_FOLDER = '/animations-0.1'
if not os.path.isdir('.' + GRAPHS_FOLDER):
    os.mkdir('.' + GRAPHS_FOLDER)
if not os.path.isdir('.' + ANIMATION_FOLDER):
    os.mkdir('.' + ANIMATION_FOLDER)

if not os.path.isdir("./saved"):
    os.mkdir("./saved")
```

```
class Agent:
    """
    A class that models the behavior of wild creatures.


    Args:
        x (float): The agents initial position. See :attr:`model2.Agent.x`
        id(int): An id used by the simulation class to keep track of agents. See :attr:`model2.Agent.id`

        iq (int): The number of neurons in every hidden layer of the agent's neural network that is responsible
for movement. See :attr:`model2.Agent.iq`

        eq (int): The number of neurons in every hidden layer of the agent's neural network that is responsible
for agent to agent interactions. See :attr:`model2.Agent.eq`

        mass(int): The mass of the agent represents its size. See :attr:`model2.Agent.mass`
        final_mass(int): The final mass of the agent. See :attr:`model2.Agent.final_mass`

        breed_mass_div(float): The initial mass to final mass ratio of the agents children. See
:attr:`model2.Agent.breed_mass_div`
        breed_chance(float): The chance for the agent to breed at every step. See
:attr:`model2.Agent.breed_chance`

        size_factor(float): A variable used by the sim's movement and collision engines to control the size of the
simulation. See :attr:`model2.Sim.size_factor`

        move_brain(NeuralNetwork): The neural network the agent will use to decide where to move. This is
either provided by the creature's parent (via a mutated brain) or is generated from scratch (for the initial
population) See :attr:`model2.Agent.move_brain`.
        social_brain(NeuralNetwork): The neural network the agent will use to interact with other agents. This
is either provided by the creature's parent (via a mutated brain) or is generated from scratch (for the initial
population) See :attr:`model2.Agent.social_brain`.

        parent_id(int): This variable is used to detect close family by the simulation. It is then used as a variable
in agent interactions. See :attr:`model2.Agent.parent_id`

    Attributes:
        energy(float): Tracks the amount of energy an agent has. energy is acquired by eating and lost by
moving or thinking. Energy allows agents to grow and regenerate health. If an agent has low energy, he will
take damage. See :meth:`model2.Agent.think`, :meth:`model2.Agent.move`, :meth:`model2.interact`

        speed(float): The maximum velocity at which the creature can move. Inversely related to mass (1/mass)
        health(float): How much damage can a creature sustain. When it reaches 0, the creature dies.

        x (float): The agents position. See :meth:`model2.Agent.move`.

        id(int): An id used by the simulation class to keep track of agents. See :attr:`model2.Agent.parent_id` for
use case.

        iq (int): The number of neurons in every hidden layer of the agent's neural network that is responsible
for movement. See :attr:`model2.Agent.move_brain` for use case.

        eq (int): The number of neurons in every hidden layer of the agent's neural network that is responsible
for agent to agent interactions. See :attr:`model2.Agent.social_brain` for use case

        mass(int): The mass of the agent, represents its size.
        final_mass(int): The final mass of the agent.

        breed_mass_div(float): The initial mass to final mass ratio of the agent's children. See
```

```
:meth:`model2.Agent.breed`.
      breed_chance(float): The chance for the agent to breed at every step. See :meth:`model2.Agent.breed`

      size_factor(float): A variable used by the sim's movement and collision engines to control the size of the
simulation. See :attr:`model2.Sim.size_factor`

      move_brain(NeuralNetwork): The neural network the agent will use to decide where to move. Structure:
[3, iq, iq, 1] See :meth:`model2.Agent.think` and :attr:`model2.Agent.iq`.
      social_brain(NeuralNetwork): The neural network the agent will use to interact with other agents.
Structure [4, eq, eq, 2] See :meth:`model2.interact` and :attr:`model2.Agent.eq`.

      parent_id(int): This variable is used to detect close family by the simulation. It is then used as a variable
in agent interactions See :attr:`model2.Agent.id`.


    """

    def __init__(self,
                 iq,
                 eq,
                 mass,
                 x,
                 id,
                 final_mass,
                 breed_mass_div,
                 breed_chance,
                 size_factor,
                 move_brain=None,
                 social_brain=None,
                 parent_id=None):
        self.iq = iq
        self.eq = eq
        if move_brain is not None:
            self.move_brain = move_brain
        else:
            if iq == 1:  # if the size of the hidden layers is 1, the amount of hidden layers doesn't matter
                self.move_brain = NeuralNetwork([3, 1])
            else:
                self.move_brain = NeuralNetwork([3, iq, iq, 1])

        if social_brain is not None:
            self.social_brain = social_brain
        else:
            if eq == 1:  # if the size of the hidden layers is 1, the amount of hidden layers doesn't matter
                self.social_brain = NeuralNetwork([4, eq, 2])
            else:
                self.social_brain = NeuralNetwork([4, eq, eq, 2])

        self.parent_id = parent_id

        self.mass = mass
        self.energy = mass
        self.speed = (1 / mass) * size_factor * G_SPEED_FACTOR
        self.health = mass
        self.final_mass = final_mass

        self.breed_mass_div = breed_mass_div
        self.breed_chance = breed_chance
```

```
        self.x = x

        self.id = id

        self.size_factor = size_factor

    def age(self) -> None:
        """
        Make the agent experience aging

        See Also:
            :attr:`model2.Agent.health`
        """
        if self.mass > self.final_mass * AGING_TIME:
            self.health -= AGE_CONST

    def think(self, d_food: float, d_agent: float, s_agent: 'Agent') -> float:
        """
        Trigger the agent's :attr:`model2.Agent.move_brain` and make it decide where to go.
        Energy is subtracted for thinking using :attr:`model2.Agent.iq`.

        See Also:
            :attr:`model2.Agent.move`

        Args:
            d_food(float): The distance to the nearest food item
            d_agent(float): The distance to the nearest agent
            s_agent(Agent): The strength of the nearest agent

        Returns:
            float: DX, fed into :meth:`model2.Agent.move`

        """
        out = self.move_brain.feed_forward(
            [map_from_to(d_food, -1, 1, 0, 1), map_from_to(d_agent, -1, 1, 0, 1),
             int(s_agent.mass > self.mass)])
        self.energy -= self.iq * INT_CONST
        return map_from_to(float(out[0]), 0, 1, -self.speed, self.speed)

    def move(self, dx: float) -> None:
        """
        Apply the velocity calculated in the think function to the agent's position
        Energy is subtracted for moving. See :attr:`model2.Agent.energy`

        Args:
            dx: Distance to travel

        See Also:
            :attr:`model2.Agent.think`

        """
        self.x += dx
        if self.x > 1:  # make the world round
            self.x = 1 - self.x
        if self.x < -1:
            self.x = -self.x - 1
        self.energy -= MOV_CONST * dx

    def breed(self, nid: int) -> 'Agent':
```

```
        """
        Make the agent have a child.
        Note that the child's mass is removed from the agent's mass on childbirth


        See Also:
            :attr:`model2.Agent.breed_chance`
            :attr:`model2.Agent.breed_mass_div`

        Args:
            nid(int): The id of the new child

        Returns:
            Agent: A child that is a mutated version of the agent, None if the agent died in childbirth
        """
        nb = copy.deepcopy(self.move_brain)
        nb.mutate()

        nsb = copy.deepcopy(self.social_brain)
        nsb.mutate()

        nm = np.ceil((self.mass + random.randrange(-10, 10)) * self.breed_mass_div)
        self.health -= nm
        self.mass -= nm
        if self.health <= 0 or self.mass < 1 or nm < 1:
            self.health = -1
            return None  # died in childbirth
        return Agent(self.iq, self.eq, nm, self.x + random.uniform(0.001, -0.001), nid,
                     np.ceil(nm / self.breed_mass_div),
                     self.breed_mass_div + random.uniform(0.01, -0.01), self.breed_chance + random.uniform(0.01,
-0.01),
                     self.size_factor, nb, nsb, self.id)

    def eat(self, food: int) -> None:
        """
        Make the agent eat, increases agent energy (:attr:`model2.Agent.energy`) and mass
(:attr:`model2.Agent.mass`) if the agent has not reached its final mass and has energy to spare

        Args:
            food(int): The amount of food the agent should eat
        """
        if self.mass < self.final_mass and self.energy / self.mass > ENGB_CONST:
            self.mass += food  # update mass and speed
            self.speed = (1 / self.mass) * self.size_factor * G_SPEED_FACTOR
        else:
            self.energy += food

    def __str__(self) -> str:
        return str(self.id)


def fight(a1: Agent, a2: Agent) -> None:
    """
    Makes the given agents fight
    Health is lost according to the other agent's mass, but the same amount of energy is gained (note: this
expects one agent to die so the other can eat him)

    See Also:
        :attr:`model2.Agent.energy`
```

```
    :attr:`model2.Agent.mass`
    :attr:`model2.Agent.health`

  Args:
    a1(Agent): Agent 1
    a2(Agent): Agent 2
  """
  a1.health -= a2.mass
  a2.health -= a1.mass

  a1.energy += a2.mass
  a2.energy += a1.mass


def interact(a1: Agent, a2: Agent, s) -> None:
  """
  Makes the given agents interact
  Energy is subtracted according to :attr:`model2.Agent.eq`. See :attr:`model2.Agent.energy`

  Args:
    a1(Agent): Agent 1
    a2(Agent): Agent 2
    s(Sim): The simulation that is requesting the interaction (used to update statistics)
  """
  a1.energy -= a1.eq * INT_CONST  # subtract energy used for interaction thought
  a2.energy -= a2.eq * INT_CONST
  close_family = a1.parent_id == a2.id or a2.parent_id == a1.id
  s1 = a1.social_brain.feed_forward(
    [1 if close_family else 0, a1.energy / a1.mass, a2.energy / a2.mass, 1 if a1.mass > a2.mass else 0])
  s2 = a2.social_brain.feed_forward(
    [1 if close_family else 0, a2.energy / a2.mass, a1.energy / a1.mass, 1 if a1.mass < a2.mass else 0])

  if s1[0] > 0.5 or s2[0] > 0.5:  # if either agent wants to fight
    fight(a1, a2)
    s.fight += 1
  if s1[1] > 0.5:  # if an agent wants to help
    a1.energy -= FOOD_CONST
    a2.energy += FOOD_CONST
    s.help += 1
  if s2[1] > 0.5:
    a1.energy += FOOD_CONST
    a2.energy -= FOOD_CONST
    s.help += 1
  else:
    s.nothing += 1


class Food:
  """ Food class

  Args:
    x(float): Food position

  Attributes:
    x(float): Food position
  """

  def __init__(self, x: float) -> None:
    self.x = x
```

```
def mk_round(d: float) -> float:
    """
    Gets the smallest distance between 2 objects on a circle with flattened coordinates from -1 to 1

    Args:
        d(float): The distance to process


    Returns:
        float: The processed distance
    """
    if d > 1:  # if the distance between them is greater than 1, then the other distance must be smaller Ex.
1.2 -> -0.8
        return d - 2
    if d < -1:  # if the distance between them is smaller than -1, then the other distance must be smaller Ex. -
1.2 -> 0.8
        return -d - 1
    else:
        return d


class Sim:
    """A class used to model a simulated population


    Args:
        agents: The number of agents the simulation should start with. See :class:`model2.Agent`
        food_count: The amount of food that should be provided. See :class:`model2.Food`

    Attributes:

        agents(list[Agent]): A list of all the agents that are alive in the simulation.
        size_factor(float): A constant that scales the simulation. calculated via  1 / (agents / POP_DENCITY)
        col_const(float): The minimum distance at which 2 objects are considered to be colliding.
        food_count(int): The amount of food that should be provided.

        breed(int): The number of times agents breed in total
        kill(int): The number of times agents kill one another in total
        fight(int): The number of times agents fight one another in total
        help(int): The number of times agents help one another in total
        nothing(int): The number of times agents ignore one another in total

        id(int): A variable used to keep track of the issued id's to agents. See :attr:`model2.Agent.id`

        eat(int): The number of times agents ate in the last step

        food(list[Food]): A list of all the food items
        gcsteps(int): The number of total steps taken (if run/animate is used more than once)
        dataPoints(int): The nummber of datapoints recorded

        i_OT(list[int]): The x array for the model's graphs - composed of datapoints over time

        number_of_agents_OT(list[int]): A list of the number of agents over time
        mass_OT(list[float]): A list of the average agent mass over time. See :attr:`model2.Agent.mass`
        eat_OT(list[int]): A list of the rate of eating over time. See :meth:`model2.Agent.eat`

        iq_OT(list[float]): A list of the average iq over time. See :attr:`model2.Agent.iq`
```

```
        eq_OT(list[float]): A list of the average eq over time. See :attr:`model2.Agent.eq`

        breed_mass_div_OT(list[float]): A list of the average mass to final mass ratio of newborns over time. See
:attr:`model2.Agent.breed_mass_div`
        breed_chance_OT(list[float]): A list of the average breeding chase of agents over time. See
:attr:`model2.Agent.breed_chance`

        fight_OT(list[int]): A list of the amount of fighting over time. See :meth:`model2.fight`,
:meth:`model2.interact`
        help_OT(list[int]): A list of the number of creatures helping one another over time. See
:meth:`model2.interact`
        nothing_OT(list[int]): A list of the number of creatures ignoring one another over time. See
:meth:`model2.interact`

    Raises:
        TypeError: If agents or food_count aren't integers

    """

    def __init__(
            self,
            agents: int = 500,
            food_count: int = None,
    ) -> None:
        if food_count is None:
            food_count = 5 * agents

        if (not isinstance(food_count, int)) or (not isinstance(agents, int)):
            raise TypeError

        self.size_factor = 1 / (agents / POP_DENCITY)
        self.col_const = G_COL_CONST * self.size_factor

        self.food_count = food_count
        self.breed = 0
        self.kill = 0
        self.fight = 0
        self.help = 0
        self.nothing = 0
        self.id = 0
        self.eat = 0
        self.agents = []
        self.food = []
        self.gcsteps = 0
        self.dataPoints = 0
        self.interactions = 0
        for i in range(agents):  # create initial population
            mass = np.ceil(random.randrange(1, 100))
            self.agents.append(
                Agent(int(random.randrange(MIN_IQ,
                                           MAX_IQ)), int(random.randrange(MIN_EQ, MAX_EQ)),
                      np.ceil(mass * START_MASS_P),
                      random.uniform(-1, 1), self.id, mass, random.random(), random.random(), self.size_factor))
            self.id += 1

        # create statistic helpers
        self.group()
        self.i_OT = []
```

```
        self.number_of_agents_OT = []
        self.mass_OT = []
        self.eat_OT = []

        self.iq_OT = []
        self.eq_OT = []

        self.breed_mass_div_OT = []
        self.breed_chance_OT = []

        self.interactions_OT = []

        self.fight_OT = []
        self.help_OT = []
        self.nothing_OT = []

        self.relative_groups_OT = []
        self.close_family_in_group_OT = []
        self.cfood()

        return

    def get_fn(self) -> str:
        """
        A helper function that generates a file name for the graph/animation

        Returns:
            str: A unique filename

        See Also:
            :meth:`model2.Sim.graph`
            :meth:`model2.Sim.animate`
        """
        return '{}-{}-{}'.format(
            len(self.agents), self.gcsteps,
            time.strftime("%d%M%Y%H%M%S", time.localtime()))

    def save(self, file: str = None) -> None:
        """
        Save the simulation state to a file.

        Args:
            file: The path to save the state to.

        Throws:
            ValueError: if the filename does not end with .envs
        """
        if file is not None and file.endswith(".envs"):
            raise ValueError("File must end with .envs")
        pickle.dump(self, open(file if file is not None else "save/" + self.get_fn() + ".envs", "w"), 4)

    @classmethod
    def load(cls, filename: str) -> "Sim":
        """
        Load a simulation from a file
        Args:
            filename: the path to the file

        Returns:
```

```
            the loaded sim
        """
        with open(filename, 'rb') as f:
            return pickle.load(f)

    def run(self, steps: int = 1000, print_freq: int = None, max_attempts: int = 1, data_point_freq: int = 10,
            gui: bool = False) -> Tuple[bool, int]:
        """
        Runs the mode

        Args:
            max_attempts: The maximum amount of attempts the simulation should try before quitting, -1 is
effectively infinity
            steps(int): The number of steps to run the model
            print_freq(int): The frequency to print progress updates
            data_point_freq: The frequency to update data points. Note: the maximum number of data points for
excel is 16,383.

        See also:
            :meth:`model2.Sim.step`
            :meth:`model2.Sim.update_stats`

        Returns:
            (tuple): tuple containing:
                (bool: If the simulation was successful
                (int): Amount of times the simulation failed
        """
        if max_attempts == -1:  # if maximum attempts is -1, make it effectively infinite
            max_attempts = 2 ** 32

        sim_copy = copy.deepcopy(self)  # create a copy of the sim to use as a restore point

        if print_freq is None:  # if print frequency is not specified, set it so that the model prints every 1% of
steps
            print_freq = steps / 100
        for a in range(max_attempts):
            failed = False
            for i in range(steps):
                if not self.step():  # call step, if it failed, stop this attempt
                    self.__dict__.update(copy.deepcopy(sim_copy).__dict__)  # resetting the sim to its original state
                    failed = True
                    break
                if i % print_freq == 0:
                    self.progress(steps, i, gui)
                if i % data_point_freq == 0:
                    self.update_stats()  # update statistics
            if not failed:
                self.progress(steps, steps, gui)
                return True, a
        return False, max_attempts

    def group(self) -> int:
        """
        Group agents via position
        Returns:
            number of groups
        """
        prev_a = self.agents[0]
        prev_a.group = 0
```

```
    for a in self.agents:
        a.group = prev_a.group + (0 if abs(a.x - prev_a.x) < self.col_const * GROUP_FACTOR else 1)
        prev_a = a
    return prev_a.group

def progress(self, steps: int, csteps: int, gui: bool) -> None:
    """
    Print model progress
     Args:
        steps(int): How many steps are there in total
        csteps(int): The current step number
    """
    if gui:
        print(json.dumps({
            "steps": csteps,
            "food": len(self.food),
            "agents": len(self.agents)
        }))
        sys.stdout.flush()
    else:
        print(
            "{}% ({} of {}) current population size: {} amount of food: {}"
                .format(round((csteps / steps) * 100, 2),
                        csteps, steps,
                        len(self.agents),
                        len(self.food)))  # print status

def update_stats(self) -> None:
    """
    Update model statistics
    """
    agent_count = len(self.agents)  # save time
    self.gcsteps += 1
    self.dataPoints += 1

    # append statistics:
    self.i_OT.append(self.gcsteps)
    self.number_of_agents_OT.append(agent_count)
    self.interactions_OT.append(self.interactions)
    self.interactions = 0

    # group based statistics:
    self.relative_groups_OT.append(self.group())

    # in order to reduce compute time all for data collection will occur once

    helper_mass = np.ndarray([agent_count])
    helper_iq = np.ndarray([agent_count])
    helper_eq = np.ndarray([agent_count])
    helper_bmd = np.ndarray([agent_count])
    helper_bch = np.ndarray([agent_count])
    helper_group_i = -1
    helper_group_size = []
    helper_close_family_group = []

    for i in range(agent_count):
        a = self.agents[i]
        helper_mass[i] = a.mass
        helper_iq[i] = a.iq
```

```
        helper_eq[i] = a.eq
        helper_bmd[i] = a.breed_mass_div
        helper_bch[i] = a.breed_chance
        if a.group != helper_group_i:
            a.group = helper_group_i
            helper_close_family_group.append([])
            helper_group_size.append(0)
            helper_group_i += 1

        helper_close_family_group[helper_group_i].extend([a.id, a.parent_id])
        helper_group_size[helper_group_i] += 1

    self.close_family_in_group_OT.append(np.mean(
        [(len(helper_close_family_group[i]) - len(set(helper_close_family_group[i]))) / helper_group_size[i]
for i
         in
         range(helper_group_i)]))  # the average amount of duplicates is the amount of close family
    self.mass_OT.append(np.mean(helper_mass))
    self.eat_OT.append(self.eat)
    self.eat = 0
    self.iq_OT.append(np.mean(helper_iq))
    self.eq_OT.append(np.mean(helper_eq))
    self.breed_mass_div_OT.append(np.mean(helper_bmd))
    self.breed_chance_OT.append(np.mean(helper_bch))

    self.fight_OT.append(self.fight / agent_count)
    self.help_OT.append(self.help / agent_count)
    self.nothing_OT.append(self.nothing / agent_count)

    self.help = 0
    self.fight = 0
    self.nothing = 0

  def cfood(self) -> None:
    """
    Create food
    """
    self.food = []
    for i in range(int((1 - FOOD_FLUCT) * self.food_count)):
        self.food.append(Food(random.uniform(-1, 1)))

  def step(self) -> bool:
    """
    Update the model

    Returns:
        bool: If all the agents in the model are dead
    """
    self.gcsteps += 1
    if len(self.food) < FOOD_FLUCT * self.food_count:
        self.cfood()

    agent_count = len(self.agents)

    if agent_count <= 1:
        print("ALERT: the model has died")
        return False
    self.agents.sort(
        key=lambda ag: ag.x
```

```
    ) # sort agents by position, allows to quickly determine the closest agent with low complexity
    self.food.sort(key=lambda ag: ag.x)
    food_index = 0  # used to find closest food item with low complexity

    for a in range(agent_count):
        agent_count = len(self.agents)

        if a >= agent_count:  # agents can be removed but the range isn't updated
            return True

        ax = self.agents[a].x

        tf = None
        lf = None
        for i in range(food_index, len(self.food)):
            if self.food[i].x > ax:
                food_index = i - 1
                tf = self.food[i]
                break
            else:
                lf = self.food[i]

        if tf is None:
            tf = self.food[0]
        if lf is None:
            lf = self.food[-1]

        dtf = mk_round(tf.x - ax)
        dlf = mk_round(lf.x - ax)

        dfood = dtf if dtf < dlf else dlf

        if abs(
                dfood
        ) < self.col_const:  # if the abs distance is smaller than the required collision const
            self.food.remove(tf if dtf < dlf else lf)  # remove food
            self.agents[a].eat(FOOD_CONST)  # eat food
            self.eat += 1  # update food statistic
            food_index -= 1

        # because agents have been sorted by x values, it is easy to find the closest agent by comparing the
agent before and the one after

        ta = self.agents[a + 1 if a != agent_count - 1 else 0]
        la = self.agents[(a - 1) if a != 1 else agent_count - 1]

        dta = mk_round(ta.x - ax)
        dla = mk_round(la.x - ax)

        if dta < dla:
            dagent = dta
            a_s = ta
        else:
            dagent = dla
            a_s = la

        if abs(dagent) < self.col_const:
            self.interactions += 1
            interact(self.agents[a], a_s, self)
```

```
        self.agents[a].move(
            self.agents[a].think(
                dfood, dagent, a_s
            )  # pass the environment variables to the brain of the agent
        )  # updating agent position

        self.agents[a].age()  # applying age effect

        if self.agents[a].energy < ENLB_CONST * self.agents[a].mass:  # if the agent is sick, lose health
            self.agents[a].health -= ENL_CONST

        if self.agents[a].energy > ENGB_CONST * self.agents[a].mass:  # if the agent is healthy, gain health
            self.agents[a].health += ENG_CONST

        if (random.random() < self.agents[a].breed_chance
                and self.agents[a].health > 0
                and self.agents[a].mass >= self.agents[a].final_mass):  # determine if an agent will breed
            nk = self.agents[a].breed(self.id)
            if nk is not None:  # if the agent did not die in childbirth, append the newborn to the sim
                self.breed += 1
                self.id += 1
                self.agents.append(nk)

        if self.agents[a].health < -1e-5:  # if the agent's health is <=0, kill it
            self.kill += 1
            self.agents.remove(self.agents[a])
    return True

def stats(self) -> str:
    """
    Get basic statistics

    Returns:
        str: A string containing basic statistics

    """
    return "breed:{} kill:{} eat:{}\n".format(
        self.breed, self.kill, sum(self.eat_OT)
    ) + "avg mass: {}\n".format(np.mean([
        a.mass for a in self.agents
    ])) + "avg speed: {}\n".format(np.mean([
        a.speed for a in self.agents
    ])) + "avg breed chance: {}\n".format(
        np.mean([a.breed_chance for a in self.agents
                 ])) + "avg breed mass divider: {}\n".format(
        np.mean([a.breed_mass_div for a in self.agents]))

def graph(self, info: str = None, output=("plt", "excel")) -> str:
    """
    Graph the recorded statistics in a plt plot, in an excel spreadsheet or in an ssps compatible file.

    Args:
        output (Tuple[str]): the output formats to use.
        info(str): Additional notes for the plt plot. If None is passed the function will ask via input so if you
don't want info, pass an empty string.

    Returns:
        str: folder name for output
```

```
    """
    compatible_out = ["plt", "excel", "spss"]
    e = False
    for ro in output:
        if ro not in compatible_out:
            e = True
            print("WARNING, output format {} is not supported, it will be skipped".format(ro))
    if e:
        print("We currently support " + str(compatible_out))

    if info is None:
        info = input("Enter additional information about the sim: ")

    titles = [
        "Number Of Agents", "Average Agent Mass",
        "Amount of Food Consumed", "Average Agent IQ", "Average Agent EQ",
        "Average breeding mass divider", "Average Agent Breed Chance", "Fight count relative to
population size",
        "Help count relative to population size", "Ignore count relative to population size",
        "Number of groups", "Close family ration in group"
    ]

    values = [
        self.number_of_agents_OT, self.mass_OT, self.eat_OT, self.iq_OT, self.iq_OT,
        self.breed_mass_div_OT, self.breed_chance_OT, self.fight_OT, self.help_OT, self.nothing_OT,
        self.relative_groups_OT, self.close_family_in_group_OT
    ]
    extention = "png"
    fn = "graphs-0.3/" + self.get_fn()
    os.mkdir(fn)

    try:
        if "plt" in output:
            if len(titles) != len(values):
                raise Exception("Error len of titles must match len of vars")

            fig, axs = plt.subplots(len(values), sharex='all', figsize=(20, 60))
            metadata = dict()
            for i in range(len(values)):
                axs[i].plot(self.i_OT, values[i], linewidth=0.25)
                axs[i].axes.set_ylim([0, max(values[i])])
                axs[i].set_ylabel(titles[i])

                metadata["Final" + titles[i]] = values[i][-1]

            axs[0].axes.set_xlim([0, self.dataPoints])
            axs[0].set_title(
                "Simulation with {} initial agents and {} steps\nDate: {}\nNotes: {}\n\nStats:\n{}\n"
                    .format(len(self.agents), self.gcsteps, time.strftime("%D"), info,
                            self.stats()), )

            axs[-1].set_xlabel("Number Of Data Points")

            plt.tight_layout()
            plt.autoscale()

            pltfn = fn + "/plt." + extention
            fig.savefig(pltfn, bbox_inches='tight')  # save graph
            # add metadata:
```

```python
            im = Image.open(pltfn)
            meta = PngImagePlugin.PngInfo()
            for x in metadata:
                meta.add_text(x, str(metadata[x]))
            im.save(pltfn, extention, pnginfo=meta)
    except:
        print("error in generating plt file")
    transposed_data = []
    for i in range(self.dataPoints):
        transposed_data.append([j[i] for j in values])
    try:
        if "excel" in output:
            if len(values[0]) > 1048576:
                print("to manny data points, skipping excel")
            else:
                wb = openpyxl.Workbook(write_only=True)
                sheet = wb.create_sheet()
                sheet.append(titles)
                for i in transposed_data:
                    sheet.append(i)
                wb.save(fn + "/excel.xlsx")
    except:
        print("error in generating excel file")

    if "spss" in output:
        savFileName = fn + '/spss.sav'
        varNames = [i.replace(" ", "_") for i in titles]
        varTypes = dict()
        for t in varNames:
            varTypes[t] = 0
        with savReaderWriter.SavWriter(savFileName, varNames, varTypes) as writer:
            for i in range(self.dataPoints):
                writer.writerow(transposed_data[i])

    return os.getcwd() + "\\" + fn.replace("/", "\\");

  def animate(self, steps, res_mult=5, fps=10, bitrate=20000, print_freq=10, data_point_freq: int = 10,
            gui: bool = False) -> str:
    """
    Creates an animated model of the simulation

    Args:
        steps(int): The number of steps to animate
        res_mult: The size of each frame (plt figure size)
        fps(int): The animation's FPS (frames per second)
        bitrate(int): The animation's bitrate (higher -> less compression)
        print_freq(int): How frequently to print progress updates
        data_point_freq: The frequency to update data points. Note: the maximum number of data points for
excel is 16,383.

    Returns:
        str: The filename of the animation

    Danger:
        As of now this function has a memory leak. Until it is resolved you will need extremely high ram
capacity
    """
    # TODO: fix memory leak
    ims = []
```

```
    fig = plt.figure(figsize=(res_mult, res_mult))
    for i in range(steps):
        if not self.step():
            return
        if i % data_point_freq == 0:
            self.update_stats()
        if i % print_freq == 0:
            self.progress(steps, i, gui)
        acu = self.size_factor / 25
        row = [0 for i in np.arange(0, 1, acu)]
        for f in self.food:
            try:
                row[round(f.x / acu)] = 100
            except:
                if round(f.x / acu) > 1 / acu:
                    row[-1] = 100
                else:
                    row[0] = 100
        for a in self.agents:
            try:
                row[round(a.x / acu)] = 255
            except:
                if round(a.x / acu) > 1 / acu:
                    row[-1] = 255
                else:
                    row[0] = 255

        r = len(row) / (2 * np.pi)

        p = np.zeros((np.ceil(2 * r) + 10, np.ceil(2 * r) + 10))

        dist_proj = 2 * np.pi / len(row)
        angle = 0
        for v in row:
            x = round(r * np.cos(angle) + r)
            y = round(r * np.sin(angle) + r)
            p[y][x] = v
            p[y + 1][x] = v
            p[y - 1][x] = v
            p[y][x + 1] = v
            p[y][x - 1] = v
            angle += dist_proj

        ims.append([
            plt.imshow(p,
                       interpolation='nearest',
                       aspect='auto')
        ])
        acu = None
        r = None
        dist_proj = None
        p = None
        row = None
        angle = None
        gc.collect()
    self.progress(steps, steps, gui)

    ani = animation.ArtistAnimation(fig,
                                    ims,
```

```
                    interval=100,
                    blit=False,
                    repeat_delay=1000)

    # Set up formatting for the movie files
    Writer = animation.writers['ffmpeg']
    writer = Writer(fps=fps, metadata=dict(artist='Me'), bitrate=bitrate)

    fn = "animations-0.1/" + self.get_fn() + '.mp4'
    ani.save(fn, writer=writer)

    # optimize:
    # de-initialize variables:
    fig = None
    ims = None
    ani = None
    writer = None
    Writer = None
    # call garbage collector:
    gc.collect()
    # return animation file name
    return fn
```

nn.py:

```
#!/usr/bin/env python3

# Ecosystem project - studying natural biological systems using a simulated ecosystem and reinforcement learning.
# Copyright (C) 2020 Inbar Koursh
#
# This program is free software: you can redistribute it and/or modify it under
# the terms of the GNU General Public License as published by the Free Software
# Foundation, either version 3 of the License, or (at your option) any later
# version.
#
# This program is distributed in the hope that it will be useful, but WITHOUT
# ANY WARRANTY; without even the implied warranty of  MERCHANTABILITY or FITNESS
# FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License along with
# this program.  If not, see <http://www.gnu.org/licenses/>.


import configparser
import numpy
import os

dir_path = os.path.dirname(os.path.realpath(__file__))

config = configparser.ConfigParser(inline_comment_prefixes="#")
config.read(os.path.join(dir_path, 'config.ini'))

if config["GPU"]["USE_GPU"].lower() == "true":
  print("use GPU is on")
  try:
    import cupy as xp
  except:
    raise Exception(
      "Error importing CuPy. To ensure that your installation is setup properly go to "
```

```python
        "https://docs-cupy.chainer.org/en/stable/install.html")
else:
    import numpy as xp
var = numpy.ndarray


def sigmoid(a: numpy.ndarray) -> numpy.ndarray:
    """
    Sigmoid Function

    Args:
        a(numpy.ndarray): Input

    Returns:
        numpy.ndarray: Result
    """
    return 1 / (1 + xp.exp(-a))


def mutate(a: numpy.ndarray) -> numpy.ndarray:
    """
    Mutate Function

    Args:
        a(numpy.ndarray): Input


    Returns:
        numpy.ndarray: Result
    """
    return a + xp.random.normal(0, 0.1)


class NNLayer:
    """
    A class that represents a single neural network layer


    Args:
        nodes(int): The number of nodes this layer should have
        prev_nodes(int): The number of nodes the previous layer has

    Attributes:
        nodes(int): The number of nodes this layer should have
        prev_nodes(int): The number of nodes the previous layer has

        weights(numpy.ndarray): A matrix consisting of the weights of the layer
        bias(numpy.ndarray): A vector consisting of the biases of the layer
    """

    def __init__(self, nodes: int, prev_nodes: int) -> None:
        self.nodes = nodes
        self.prev_nodes = prev_nodes
        self.weights = xp.random.random((prev_nodes, nodes))
        self.bias = xp.random.random(nodes)

    def feed_forward(self, xs: numpy.ndarray) -> numpy.ndarray:
        """
        Feed forward inputs into the layer
```

```
        .. math:: a^{(L)} = \\sigma (\\sum_{i=1}^{m} W_{i}^{(L)}  a_{i}^{(L-1)}  + b)

        Args:
            xs(numpy.ndarray): Inputs

        Returns:
            numpy.ndarray: Layer outputs for the inputs
        """
        if len(xs) != self.prev_nodes:
            raise Exception(
                "error, wrong input size, expected {} got {}".format(
                    self.prev_nodes, len(xs)))
        return sigmoid(xp.add(xp.matmul(xp.array(xs), self.weights), self.bias))

    def mutate(self) -> None:
        """
        Mutates the layer
        """
        self.weights = mutate(self.weights)
        self.bias = mutate(self.bias)

    def __repr__(self) -> str:
        return "weights: {} bias: {}".format(self.weights, self.bias)


class NeuralNetwork:
    """
    A neural network

    Args:
        nodes(list[int]): A list of the amount of nodes for every layer

    Attributes:
        layers(np.ndarray[NNLayer]): A list of all the layers composing the neural network

    Raises:
        Exception: If the nodes list is shorter than 2

    """

    def __init__(self, nodes: list) -> None:
        if len(nodes) < 2:
            raise Exception(
                "error, the neural network needs to have an input and output layer"
            )
        self.layers = xp.ndarray([len(nodes) - 1], dtype=NeuralNetwork)
        for i in range(len(nodes) - 1):
            self.layers[i] = NNLayer(nodes[i + 1], nodes[i])

    def feed_forward(self, xs: numpy.ndarray) -> numpy.ndarray:
        """
        Feed forward the input throughout all the layers of the neural network

        Args:
            xs(numpy.ndarray): Input


        Returns:
```

```
        numpy.ndarray: The output of the neural network

    Warnings:
        it is advised that all inputs be between 0 and 1
    """
    ys = None
    for l in self.layers:
        if ys is None:
            ys = l.feed_forward(xs)
        else:
            ys = l.feed_forward(ys)
    return ys

def __repr__(self) -> str:
    return str([l.__repr__()
            for l in self.layers]).replace("\\n",
                                    "").replace(",", "\n")

def mutate(self) -> None:
    """
    Mutate all the layers of the neural network
    """
    for layer in self.layers:
        layer.mutate()
```

auto.py:

```
#!/usr/bin/env python3

# Ecosystem project - studying natural biological systems using a simulated ecosystem and reinforcement learning.
# Copyright (C) 2020 Inbar Koursh
#
# This program is free software: you can redistribute it and/or modify it under
# the terms of the GNU General Public License as published by the Free Software
# Foundation, either version 3 of the License, or (at your option) any later
# version.
#
# This program is distributed in the hope that it will be useful, but WITHOUT
# ANY WARRANTY; without even the implied warranty of  MERCHANTABILITY or FITNESS
# FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License along with
# this program.  If not, see <http://www.gnu.org/licenses/>.


from model2 import Sim
import argparse
import math

parser = argparse.ArgumentParser(description='Run an ecosystem simulation',
formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument("-s", type=int, dest="steps",
            help='Number of steps to run the sim', default=1000)

parser.add_argument("-p", type=int, dest="pop",
            help='Initial population size', default=500)

parser.add_argument("-f", type=int, dest="food",
            help='Ammount of food')
```

```
parser.add_argument("-a", help="enable animation", dest="animate", action='store_true')
parser.add_argument("-v", help="enable verbose", dest="v", action='store_true')

max_e = 1048576
parser.add_argument("-dp", type=int, dest="dp",
            help='Maximum number of data points, defaults to excel maximum (' + str(
              max_e) + ') if excel is used. Else, defaults to the number of steps', default=max_e)

parser.add_argument("--spss", help="output data in SPSS data format. Note that this will NOT force data
points to max.",
            dest="spss", action='store_true')

parser.add_argument("--no-excel",
            help="Don't output data to excel format. Will be enabled automatically if the number of data
points exceed " + str(
              max_e),
            dest="no_excel", action='store_true')

parser.add_argument("--no-plt", help="Don't generate plt preview",
            dest="no_plt", action='store_true')

parser.add_argument("--gui", help="Used to output progress in json format for GUI (in beta)",
            dest="gui", action='store_true')

args = parser.parse_args()

ms = Sim(args.pop, args.food)

# because this model will only run once, we can maximise the amount of data points without exceeding
the maximum of 18277
# total_data_points = steps/data_point_freq

args.dp = min(args.dp, args.steps)  # make sure that data points is smaller than steps
if args.dp is None:
  args.dp = args.steps if args.no_excel else min(args.steps, max_e)
data_point_freq = math.floor(args.steps / args.dp)
data_point_freq += 1 if data_point_freq == 0 else 0

if args.v:
  print("data point frequency selected {}".format(data_point_freq))
  print("expected data points: {}".format(args.steps / data_point_freq))

if args.animate:
  ms.animate(args.steps, data_point_freq=data_point_freq, gui=args.gui)
else:
  ms.run(args.steps, data_point_freq=data_point_freq, gui=args.gui)

req_formats = ()
if not args.no_plt:
  req_formats += ("plt",)
if not args.no_excel:
  req_formats += ("excel",)
if args.spss:
  req_formats += ("spss",)

print("Simulation complete. Requested files are stored at: " + ms.graph(info="generated via auto.py",
output=req_formats))
```

Appendix 4 - License

Note that this license does not cover the code included which is license under GPL V3. See the GitHub repository for more information concerning licensing. If you have any questions open a GitHub issue requesting clarification.