ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises: Modelling and Simulating Social Systems

Project Report

Two Predators and Two Prey Interactions

Alexander Bruun & Anton Paris

Zürich
9th of December 2018

IMPORTANT

You MUST include the ETH declaration of originality here; it is available for download on the course website or at

http://www.ethz.ch/faculty/exams/plagiarism/index_EN;

It can be printed as pdf and should be filled out in handwriting.

Agreement for free-download

We hereby agree to make our source code of this project freely available for download from the web pages of COSS. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Alexander Bruun

Anton Paris

Table of content

Abstract

Cellular automaton is an important simulation method to describe highly complex systems. One of them would be the interaction of two different preys and two different predators. With the simulation method of cellular automata and the theoretical background of the Lotka-Volterra equations some interesting conclusions can be drawn. One is the interesting case of gliders where stripes of the two predator states are moving in one direction. The other is that if the parameters match, we get similar results as expected from the Lotka-Volterra equations.

Individual contributions

Anton Paris is responsible for the code. He extended the code of Leonel Aguilar that was provided during a lecture to perform basic cellular automata simulations. Alexander Bruun is responsible for the theoretical background and the tuning of parameters to get reproducible and representable results.

Introduction and Motivations

In natural sciences cellular automaton plays an important role. It is a very powerful tool to describe highly complex and nonlinear system with a simple set of rules. It is used in a broad spectrum of topics which vary from computer sciences, mathematics, physics, biology to chemistry. The basics of cellular automaton is that a state is changed based on inputs (mostly neighbors) and its previous state, [1], [2].
In effect we have a grid containing cells. These cells can have one (or more) of a finite set of states. The grid can be of any dimension and size. Each cell has a neighborhood, which contains the cells in a certain radius, defined by the user. To begin with the simulation an initial state has to be defined in which each cell gets a state assigned. When the initialization has been made, the cells will change its state in discrete time steps depending on its own state and its neighbors, [1], [2].
A broad spectrum of simulations can be made with the cellular automaton. A very popular, yet very simple application is Conway's Game of Life, [3]. It is a two-dimensional simulation in which a cell is either alive or dead. The rules are cited in the following, this is to give a brief overview of a typical set of rules that create the base of our simulation.

"Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

I. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
II. Any live cell with two or three live neighbors lives on to the next generation.
III. Any live cell with more than three live neighbors dies, as if by overpopulation.
IV. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed; births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick. Each generation is a pure function of the preceding one. The rules continue to be applied repeatedly to create further generations.", [3].

With cellular automaton and these simple set of rules patterns occur. Some patterns are visualized in Figures 1-3 and a detailed explanation on how and why is found in [3]. Regarding our own simulation we do not expect stable patterns that last over a long period of time to arise, since the environment and pattern involved are extremely sensitive to initial condition, anyhow some reoccurring patterns could arise during simulation.
With the basics of Conway's Game of Life in mind we are expanding the two states: alive or dead into a predator and prey simulation with multiple states, in effect: neutral forest ground, rabbit, deer, wolf and bear.
At this point one is justified to ask why? To our knowledge there exist only few simulations that are researching the behavior of a two-predator and two-prey environment. The ones we have found are [5] and [6]. [5] lacks of simulation and [6] the integration of a second prey state.

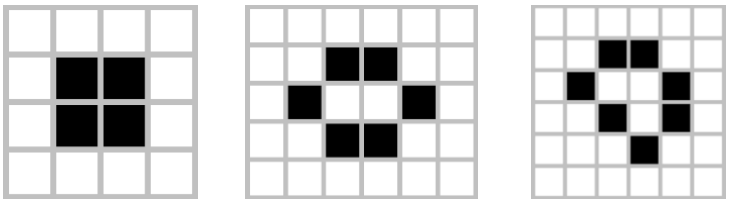


Figure 1 – Still life patterns as described by Conway's Game of Life, [3].

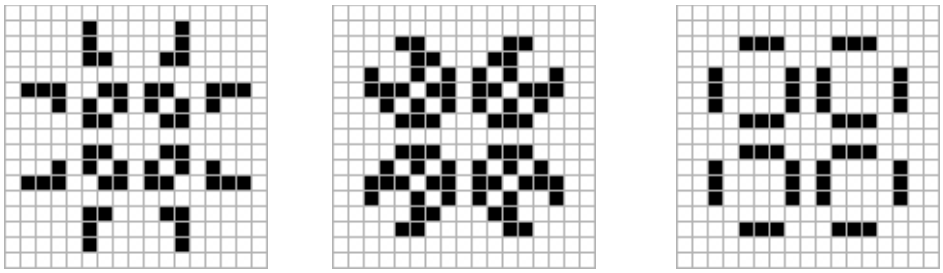


Figure 2 – Oscillating patterns as described by Conways's Game of Life, [3].

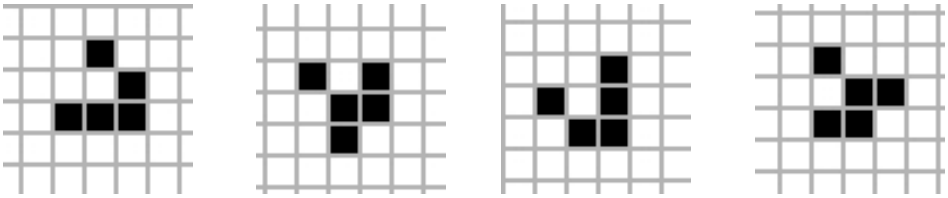


Figure 3 – Gliders as described by Conways's Game of Life, [3].

It is still necessary to mention that predator and prey simulations already exists. The two we'd like to mention are the Wa-Tor simulation by Alexander Keewatin Dewdney [7] and the Circle of Life simulation from Ruben Kälin and Patrick Misteli, [8].
We'll give a brief explanation of the both to familiarize the reader with our subject as well as to give some insight into our inspirations: The Wa-Tor simulation uses the following states: water (neutral ground), shark and fish which rely on different parameters such as reproduction cycles and starvation. The results acquired were highly dynamic with three possible outcomes: balance between sharks and fish, disappearance of sharks or the extinction of both species. A more detailed description can be found in [7]. The Circle of Life is a similar setup with the states: nothing (neutral ground), grass, antelopes and lions. Again, similar results are acquired and can be reviewed in more detail in [8].

Other questions we want answered are: Will we either get:

- pattern stabilizations into homogeneity?
- patterns stabilizing or oscillating?
- chaotic pattern evolution?
- highly complex structures with local stabilities? (all 4 bullet points from [1].)

How will parameters like food preferences, reproduction or death probability influence population dynamics? Are our results in any way representable by governing population equations or the real world?

Theory

A. J. Lotka and V. Volterra have derived equations that describe the interaction between different species. A two species interaction is described by the following set of first order differential equations, [9]:

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{1}$$

$$\frac{dy}{dt} = \delta xy - \gamma y \tag{2}$$

Where *x* is the number of prey and *y* the number of predators at time *t*. Looking at equations (1) and (2) conclusions about the dynamics of a predator prey model can be made: growth/decrease of population *x* highly depends on the parameters α, β and the number of *y*. The more generalized formula for the interaction of *i* species, called the generalized Lotka-Volterra equations, [10], has the following form:

$$\frac{dx_i}{dt} = x_i\left(r_i - \sum_{j=1}^{n} a_{ij}x_j\right) \; for \; i = 1,2,\dots,n. \tag{3}$$

If we set $n = 4$ we get:

$$\frac{dx_1}{dt} = x_1(r_1 - a_{11}x_1 - a_{12}x_2 - b_{11}y_1 - b_{12}y_2) \tag{4}$$

$$\frac{dx_2}{dt} = x_2(r_2 - a_{21}x_1 - a_{22}x_2 - b_{21}y_1 - b_{22}y_2) \tag{5}$$

$$\frac{dy_1}{dt} = y_1(-s_1 + d_{11}x_1 + d_{12}x_2) \tag{6}$$

$$\frac{dy_2}{dt} = y_2(-s_2 + d_{21}x_1 + d_{22}x_2) \tag{7}$$

Where $y_1 = x_3$ and $y_2 = x_4$ and the initial conditions x_{10}, x_{20}, y_{10} and y_{20} determine the initial species count. The distinction between *x* and *y* is to easily differ between predator and prey (x_i refers to prey and y_i to predator). In Table 1 the parameters are further explained.

Parameter	Description
r_i	Reproduction rate
a_{ij}	Disadvantage of x_i being overpopulated by x_j
b_{ij}	Disadvantage of x_i being killed by y_j
s_i	Disadvantage of y_i overpopulating
d_{ij}	Advantage of y_i hunting x_j

Table 1 – List of parameters from the Lotka-Volterra equations and their description.

As one can imagine this set of differential equations is extremely sensitive to the individual coefficients listed in Table 1 and their corresponding initial conditions. In combination with Table 2 and Figures 4-6 some example values have been plotted using MATLAB (code to reproduce results is found in the appendix), all with the initial conditions: $x_{10} = 10, x_{20} = 20, y_{10} = 3, y_{20} = 6$. An in-depth analysis of resulting equilibria from Equations (4)-(7) is found in [5].

Figure	[4]	[5]	[6]
r_1	0.3	0.3	1.6
r_2	0.8	0.8	2
a_{11}	0.1	0	0.01
a_{12}	0.3	0.3	0.2
a_{21}	0.3	0.3	0.01
a_{22}	0.1	0	0.2
b_{11}	0.8	0.8	2.1
b_{12}	0.8	0.8	0.6
b_{21}	0.6	0.6	0.6
b_{22}	0.6	0.6	2.1
s_1	0.25	0.25	1.5
s_1	0.25	0.25	0.9
d_{11}	0.6	0.6	1.5
d_{12}	0.3	0.3	0.8
d_{21}	0.6	0.6	0.8
d_{22}	0.3	0.3	1.5

Table 2 – List of example values for equation (4)-(7).

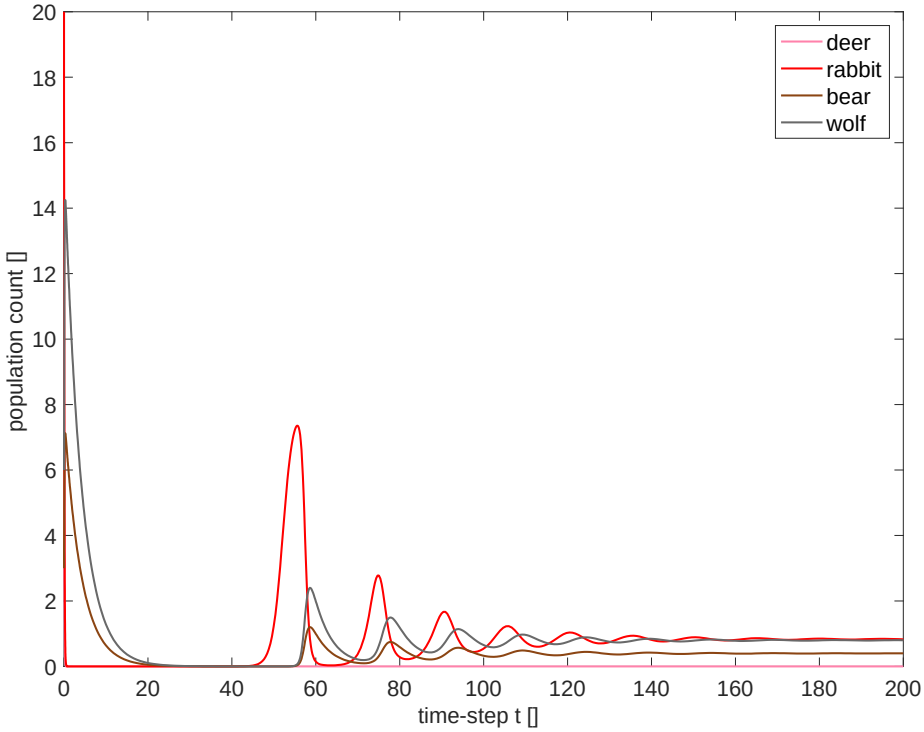


Figure 4 – Example plot for the Lotka-Volterra equations with values seen in Table [2].

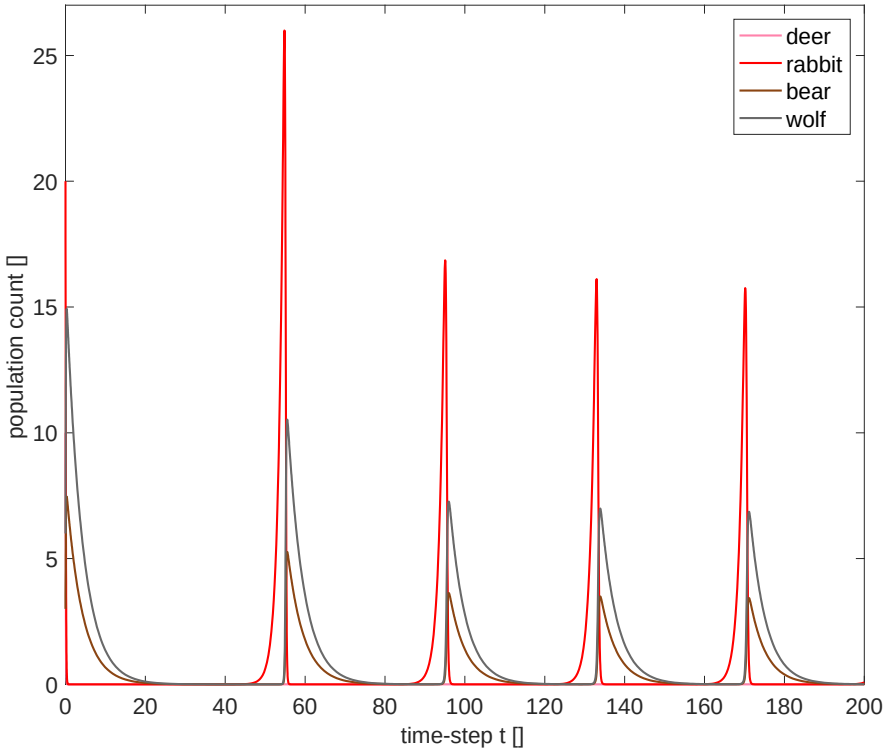


Figure 5 – Example plot for the Lotka-Volterra equations with values seen in Table [2].

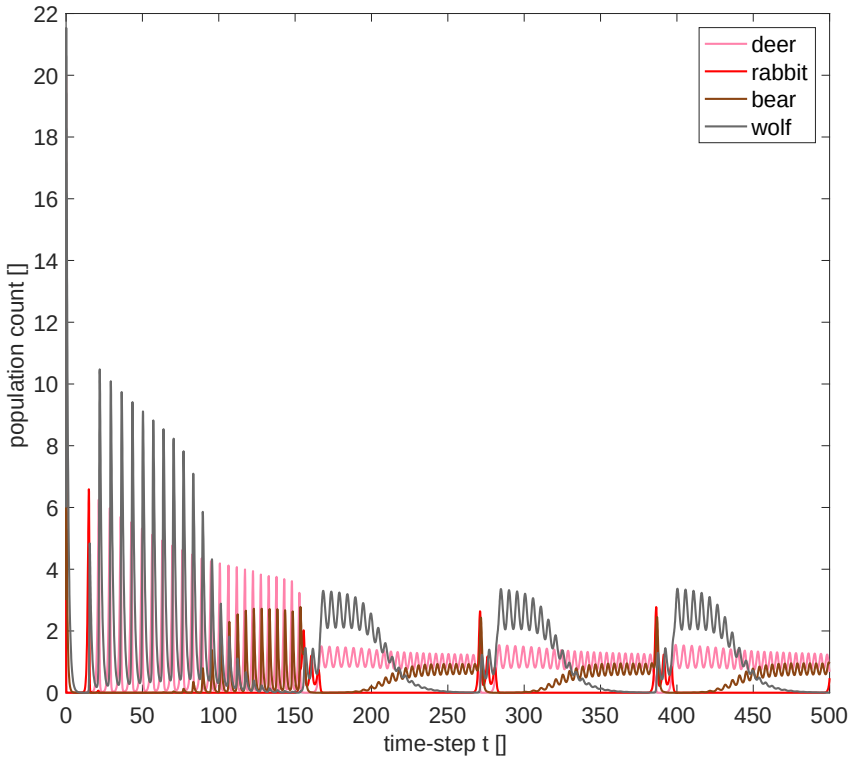


Figure 6 – Example plot for the Lotka-Volterra equations with values seen in Table [2].

Description of the Model

Cells
As mentioned before, a cell can have one of the following states:

- Forest – green – Acts as a neutral ground and continuous food source for the prey, meaning they can never die of starvation.
- Rabbit – light grey – One of the two prey states, acts as a food source for predators
- Deer – light brown – The second prey state and also a food source for predators
- Wolf – dark grey – One of the two predator states, can feed from preys but can be eaten by the bear
- Bear – dark brown – The second predator state, can feed from all other cell states (except forest)

Properties
Each cell requires a list of properties that characterize the cells behavior and interaction with the environment. The list of parameters and their description can be seen in Table 3.

Parameter Name {Value Range}	Description
type_cell {0,1,2,3,4}	Determines the state of the cell
lifespan {0,∞}	Contains the age of the cell
nutrition_level {0,∞}	Defines the nutrition level of a cell
reproduction_count {0,∞}	Contains number of offspring in a turn
death_probability {0,100}	Probability that an animal dies in a round
turn {0,1}	Determines if a cell has already had its turn in the timestep
offspring_chance {0,100}	Chances to produce offspring

Table 3 – List of cell properties with description.

Environment
In addition to the cell properties a list of parameters can be set by the user that will directly affect the behavior of the cells. The list of parameters and their description can be taken from Table 4, effective values will be discussed in the Result section.

Protocol per timestep
In the beginning all cells are initialized and shuffled randomly to determine the order in which cells are initialized. Once the order of iteration through all cells has been determined each cells' individual rules are applied. The overall first rule is to check if a cell has already been eaten (*turn 0* or *1*). This is needed in the case where a predator eats a prey (since the predator's turn might be before the prey by random order) that the prey cannot move anymore when it's the preys turn. After the initial rule is applied the cell has to check for its neighborhood.

Neighborhood
As used in many simulations (for us relevant [3] and [8], where in [8] an extension was also made) we decided to use a Moore-neighborhood, meaning the eight cells that are

directly surrounding the cell in question. The grid borders are connected, meaning there are no corners where cells can be surrounded but can "move to the other side" resulting in a torus shaped environment. After initializing the neighborhood, the rest of the rules can be applied.

Parameter name {Value Range}	Description
offspring_chance_table {0,100}	Probability for offspring if two cells of the same type qualify for reproduction
values offspring {0, ∞}	Needed nutrition level to qualify for offspring
eat_values {0, ∞}	Gain in nutrition level if one cell type eats another
nutrition_value_each_turn {0, ∞}	Increase in nutrition value for prey (eating neutral ground each round)
cost_reproduction {0, ∞}	How much it costs to reproduce one offspring
cost_movement {0, ∞}	How much it costs to move one step
nutrition_level_start {0, ∞}	How big the nutrition level is when the cells are initialized
death_values {0,100}	Death probability for a cell type
rules {0,1,2,3,4}	Defines the priority list which cell type is favorable to be eaten
population {0,1,2,3,4}	Represents animals
Probability {0,1}	Probability for an animal to be initialized in the grid

Table 4 – List of initial parameters, set by the user, with description

Rules

Movement/Eating: After scanning the neighborhood the cell will move on to a cell after its priorities set by the parameter *rules*. Eating is always possible (does not require a certain nutrition level) so if a cell can eat it will eat. Moving one step costs the cell a certain value. It is also important to note the following that a prey will not move on to a cell that is in the neighborhood of a predator if possible.

Reproduction: If a cell has a neighbor of the same state it qualifies for reproduction. Once it qualifies for reproduction both have to have a high enough nutrition level to reproduce. If the cell qualifies for the second requirement as well it will reproduce and move on to a random cell in its neighborhood (remember that it can also eat during reproduction due to the movement rule) and lose a certain value in *nutrition_level*.

Natural death: Since *death_values* are defined in the beginning the probability for a cell to die of natural death increases linearly with the by the user defined value.

After defining the rules, a general turn for a cell state looks like the following:
If a cell is a…

- Forest it remains a forest.
- Prey and has not been eaten it has three possibilities:
    - It can either die of natural death.
    - It can reproduce.
    - It can move.
- Predator and has not been eaten (bear eats wolf) it has four possibilities:

- It can die of natural death
- It can reproduce.
- It can eat a cell.
- It can move.

Implementation

The simulation was made in python. Cells are created with a class which are the basis of getting the information needed. The grid might be interpreted as a n x m matrix but is in fact a 2D list, meaning one row with n elements contains a column with m elements.

Simulation Results and Discussion

Performing the simulations many interesting cases appeared. We decided to perform simulations in a 20 x 20, 50 x 50 and 100 x 100 grid to see how grid size will affect simulation results. Some characteristics were shared by all grid sizes. First, we are going to mention phenomena shared by all grid sizes and then we are going to mention individual characteristics. Parameters for our results will be listed in each paragraph.

Gliders

Gliders as we refer to them are stripes of wolves and bears that are moving in a certain direction as seen in Figure 8. They will not appear if preys are present since they cannot eat predators but will be eaten from both sides. These gliders appear if there is a front of predators eating another front of predators. Necessary for stable sliders are large enough populations and mostly and a stripe shape and not closed islands of predators, both cases are seen in Figure 8. Usually one population is larger than the other. The populations will feed from each other and reproduce while doing so resulting in a gliding motion of the populations. Parameters for each grid size are found in Table 5, plot is visualized in Figure 7.

Parameter	20 x 20 Grid	50 x 50 Grid	100 x 100 Grid
offspring_chance_table	0, 100, 100, 100, 100		
values_offspring	0, 1, 1, 1, 1		
eat_values	0, 1, 1, 1, 1		
nutrition_value_each_turn	2		
cost_reproduction	0.5		
cost_movement	0.5		
nutrition_level_start	5		
death_values	0, 0.091, 0.167, 0.05, 0.167	[0,0.091,0.167,0.05,0.033]	
rules	0, 0, [1, 3, 0, 4], [0], [1, 3, 0, 2]	0, 0, [1, 3, 0, 4], [0], [3, 1, 0, 2]	[0], [0], [1, 3, 0, 4], [0], [1, 3, 0, 2]
population	0, 1, 2, 3, 4		
probability	0.7, 0.25, 0.03, 0.2, 0.03	0.7, 0.25, 0.01, 0.2, 0.01	

Table 5 – List of initial parameters, set by the user to get the wanted results

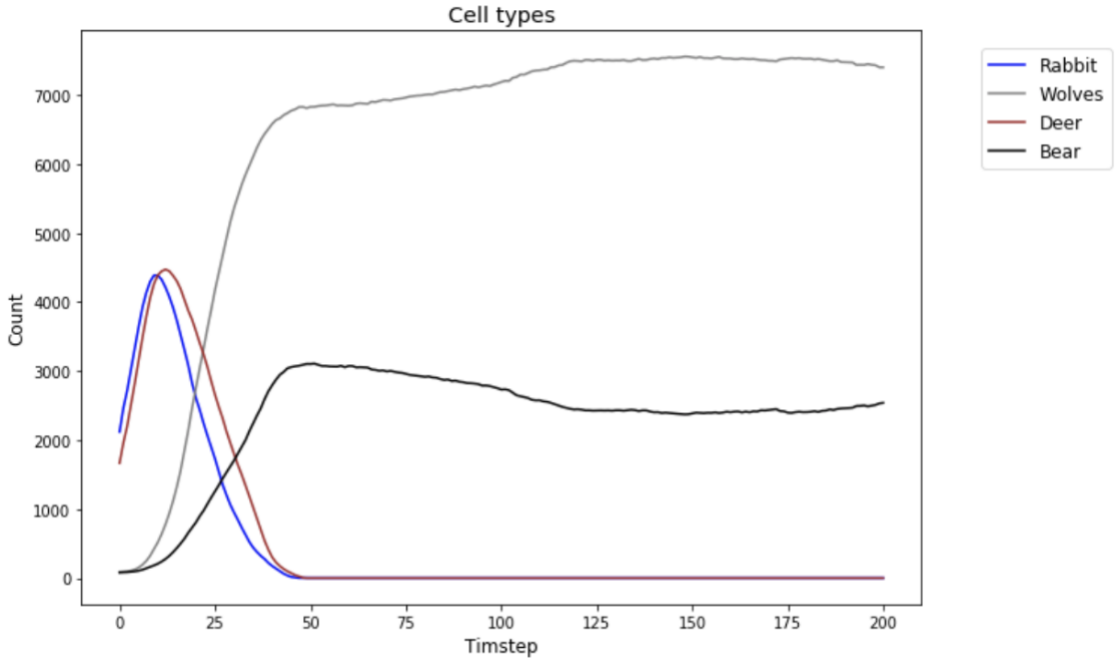


Figure 7 – Plot of population count for gliders in a 100 x 100 grid.

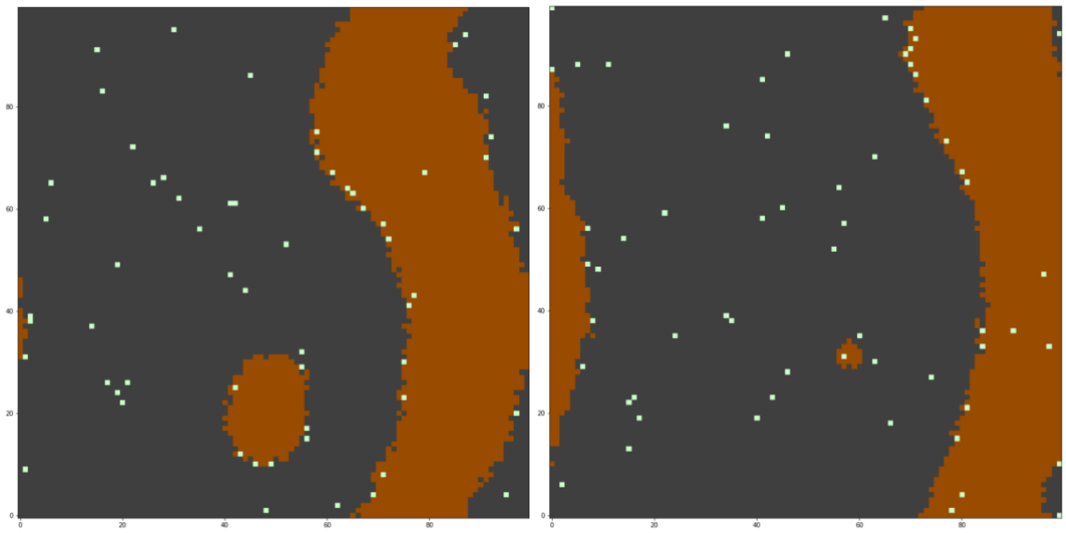


Figure 8 – Simulation of two predator populations acting as gliders in a 100 x 100 grid.

Extinction

A common result was extinction of all populations and happened mainly because of two reasons: fronts of predators eating islands of prey from the outside in as seen in Figure 9 or predators eating prey from the inside out. If predators are eating prey from the outside in prey cannot reproduce after the front and after all prey is eaten the predators will die eventually since the grid is not filled enough (if it was filled enough case overpopulation). In the other case of getting extinguished from the inside out the prey is too distributed across the grid to reproduce. Parameters for each grid size are found in Table 6, plot is visualized in Figure 10.

Parameter	20 x 20 Grid	50 x 50 Grid	100 x 100 Grid
offspring_chance_table	0, 100, 100, 100, 100		
values_offspring	0, 1, 5, 1, 5		
eat_values	0, 1, 1, 1, 1		
nutrition_value_each_turn	10		3
cost_reproduction	0.5		1
cost_movement	0.5		1
nutrition_level_start	5		
death_values	0, 0.091, 0.167, 0.05, 0.033		
rules	0, 0, [1, 3, 0, 4], 0, [3, 1, 0, 2]		0, 0, [1, 3, 0, 4], 0, [1, 3, 0, 2]
population	[0,1,2,3,4]		
probability	0.7, 0.25, 0.01, 0.2, 0.01		

Table 6 – List of initial parameters, set by the user to get the wanted results

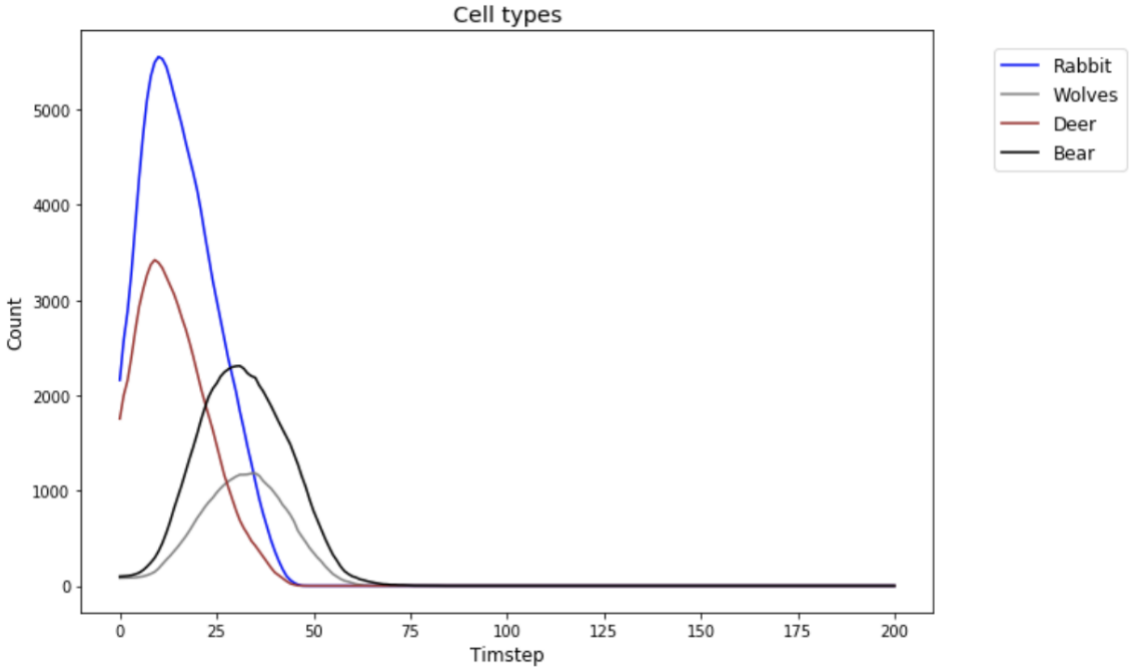


Figure 9 – Plot of population count for extinction in a 100 x 100 grid.

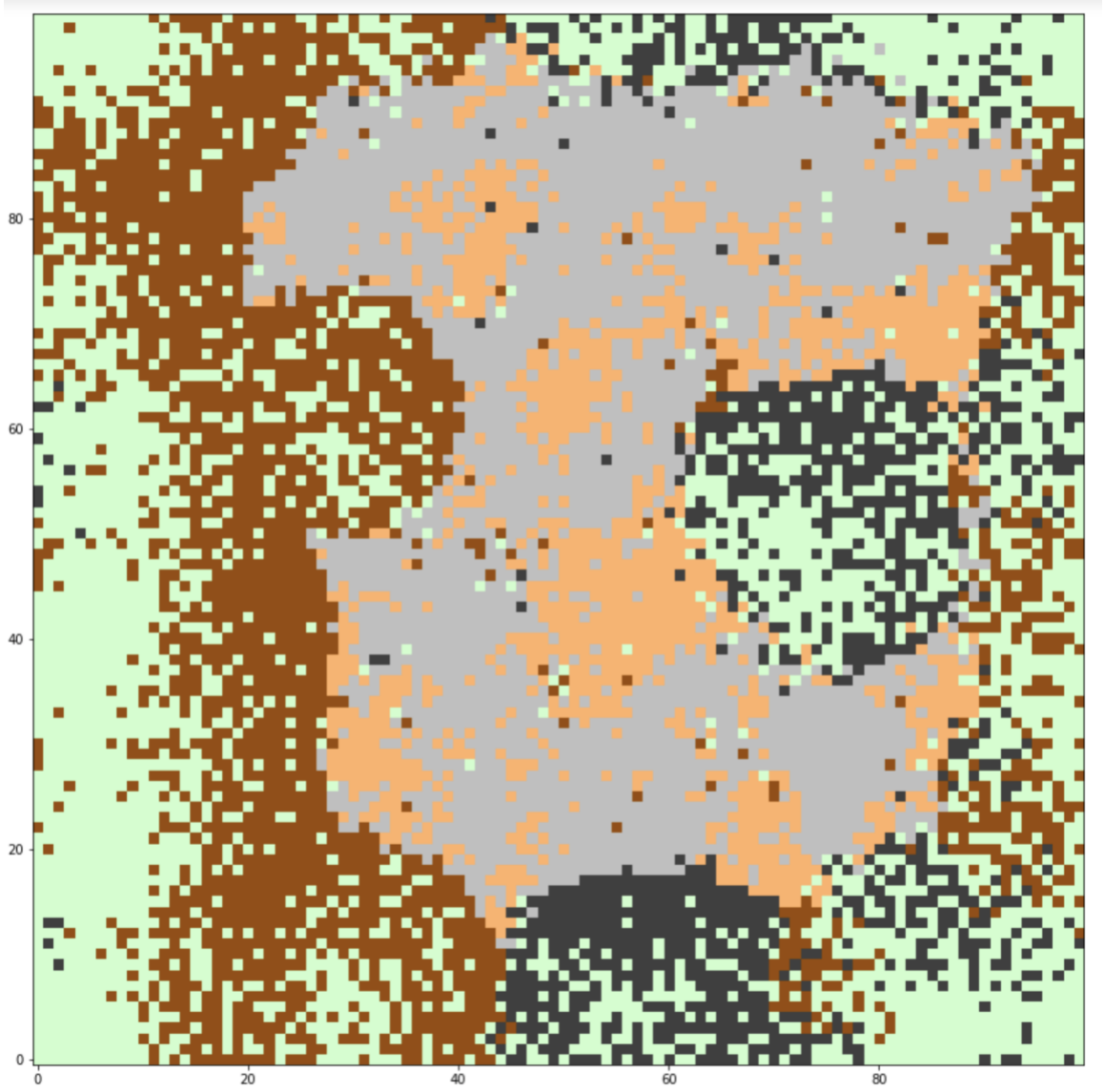


Figure 10 – Simulation of island of prey being eaten by a front of predators in a 100 x 100 grid.

Overpopulation

Depending if there are enough predators and no preys or vice versa it can happen that one population (or both preys since they cannot feed from each other) fills the whole grid. After filling the whole grid there is no coming back since the cells will reproduce if another cell dies and fill its spot. These overpopulations of prey can also contain few predators which will move through predators but cannot reproduce since either he has no friend to reproduce or if the prey already fills the spot through reproduction. An example of a grid filled with prey and some predators can be seen in Figure 12. Parameters for each grid size are found in Table 7, plot is visualized in Figure 11.

Parameter	20 x 20 Grid	50 x 50 Grid	100 x 100 Grid
offspring_chance_table	0, 100, 100, 100, 100		
values_offspring	0, 1, 5, 1, 5		
eat_values	0, 1, 1, 1, 1		
nutrition_value_each_turn	3		
cost_reproduction	1		
cost_movement	1		
nutrition_level_start	5		
death_values	0, 0.091, 0.167, 0.05, 0.033		
rules	0, 0, [1, 3, 0, 4], 0, [3, 1, 0, 2]		
population	0, 1, 2, 3, 4		
probability	0.7, 0.25, 0.01, 0.2, 0.01		

Table 7 – List of initial parameters, set by the user to get the wanted results

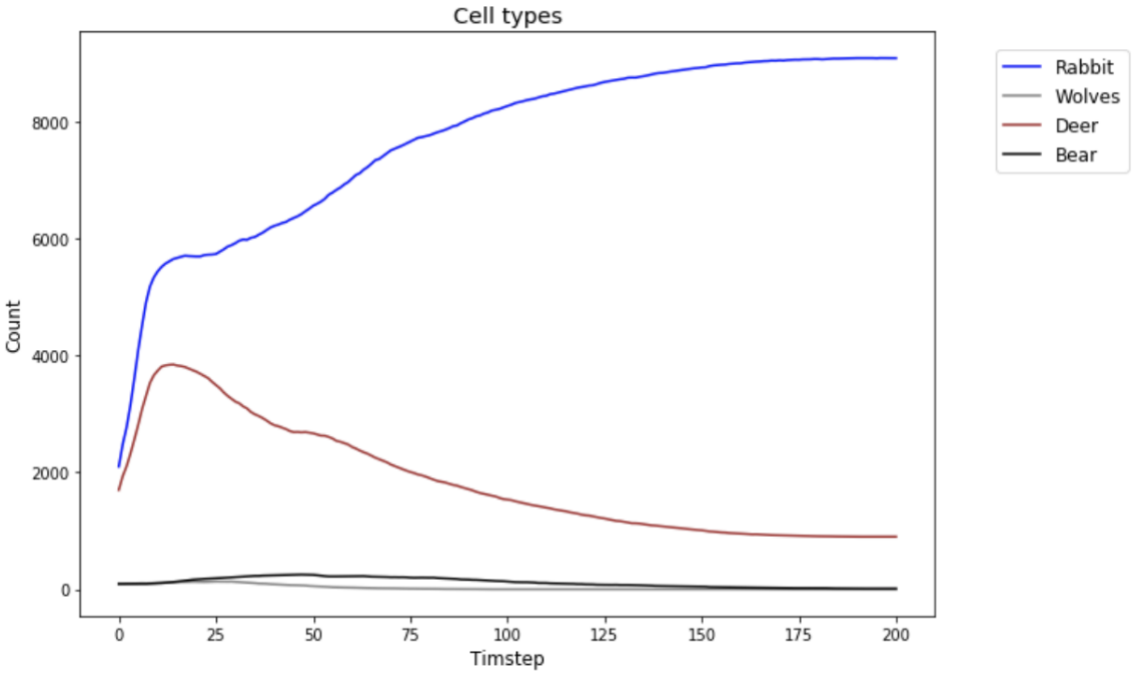


Figure 11 – Plot of population count for overpopulation in a 100 x 100 grid.

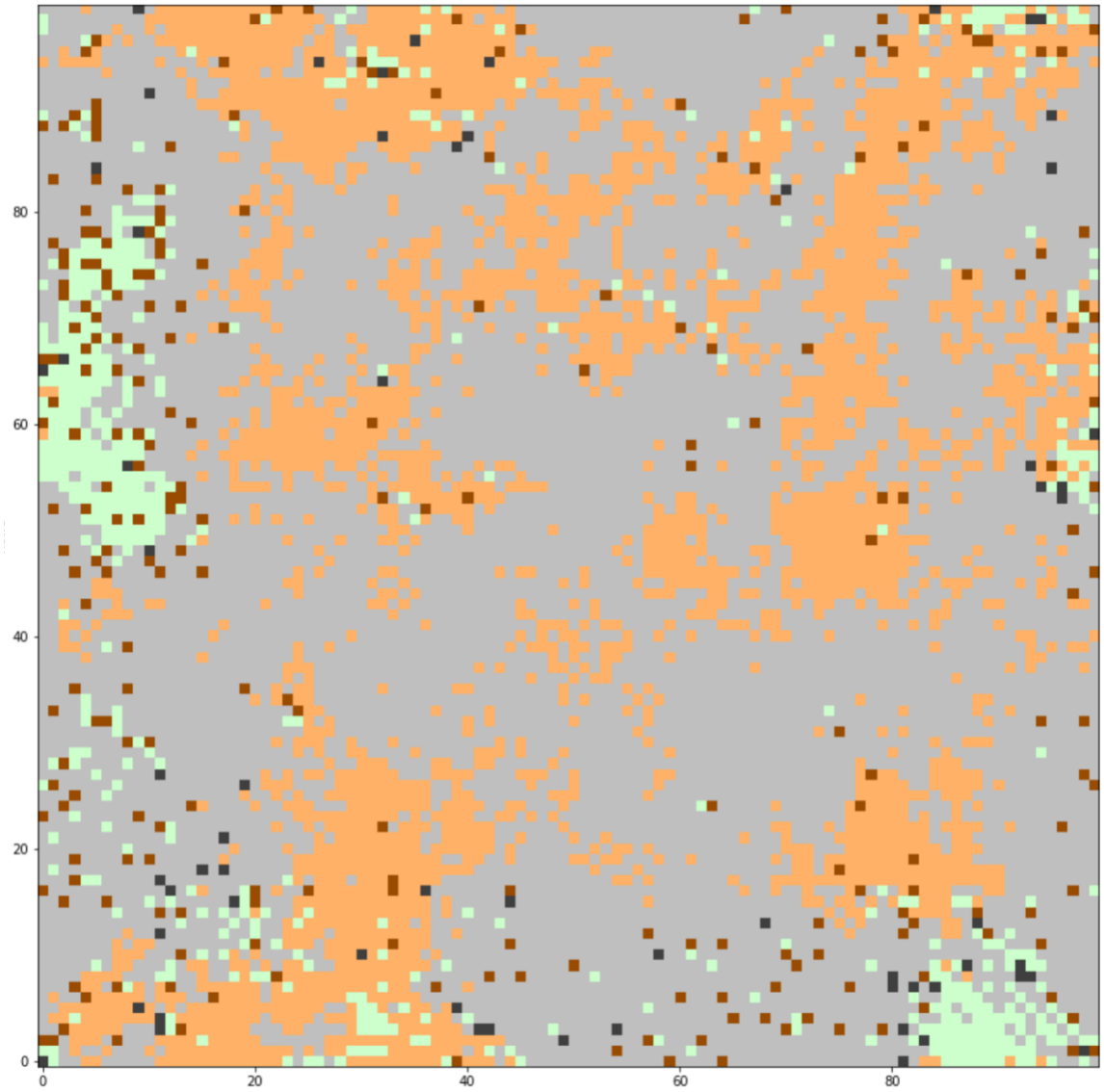


Figure 12 – Simulation of grid filled with prey containing few predators in a 100 x 100 grid.

Oscillations

The most interesting case for us are the oscillations since they represent the real world more or less (neglecting the extremely high amplitude changes). Oscillations usually occurred if there was a big island of prey and small islands of predators feeding themselves occasionally from the island as in Figure 14. Oscillations do not always happen between all four cell states but can happen in all combinations as long as at least one of them is a predator. It is also very difficult to find long lasting oscillations in a 50 x 50 grid and even more difficult in a 20 x 20 grid which is why we left out the parameters for the 20 x 20 since chances are very low to get stable oscillations. Parameters for the two grid sizes are found in Table 8, plot is visualized in Figure 13.

Parameter	50 x 50	100 x 100
offspring_chnce_table	0, 100, 100, 100, 100	
values_offspring	0, 1, 1, 1, 1	
eat_values	0, 1, 1, 1, 1	
nutrition_value_each_turn	6	4
cost_reproduction	2	1.5
cost_movement	2	1.5
nutrition_level_start	5	5
death_values	0, 0.091, 0.167, 0.05, 0.033	
rules	0, 0, [1, 3, 0, 4], 0, [3, 1, 0, 2]	
population	0, 1, 2, 3, 4	
probability	0.7, 0.25, 0.01, 0.2, 0.01	

Table 8 – List of initial parameters, set by the user to get the wanted results

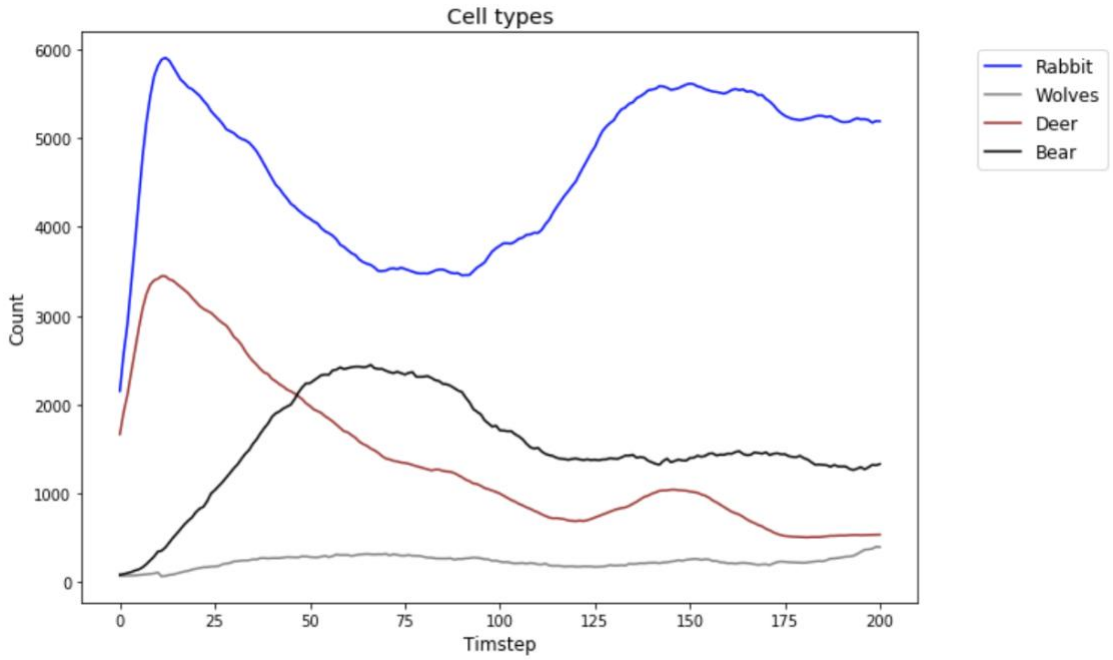


Figure 13 – Plot of population count for oscillations in a 100 x 100 grid.

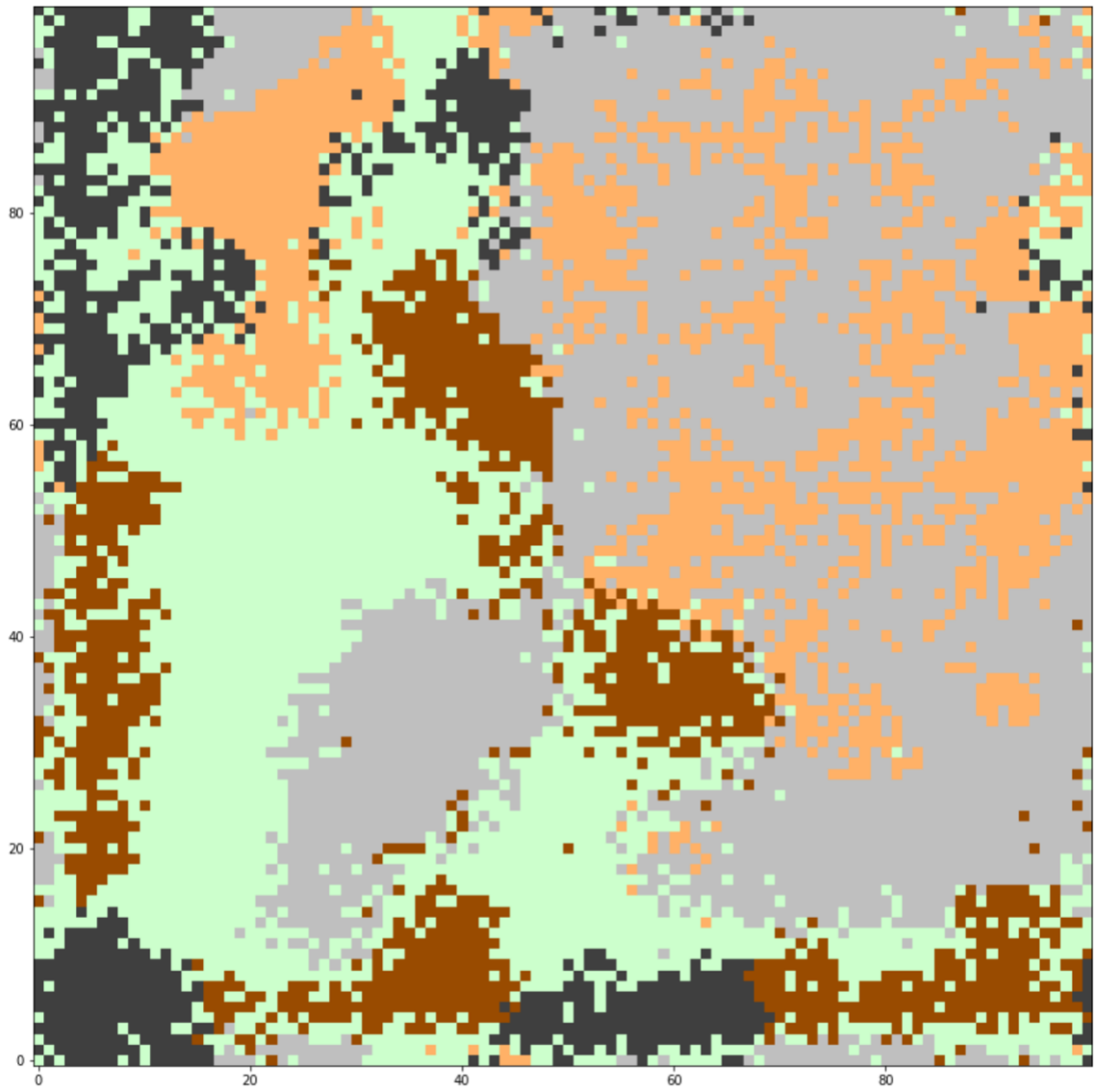


Figure 12 – Simulation of island of prey being eaten by front of predators with small islands of predators in a 100 x 100 grid.

Small grid size

Mostly ends in two cases: extinction of all species or dominance of one cell state (two for prey). The small grid size leads to an extreme sensitivity growing populations. This depends strongly how the predators behave in the beginning. Once the predator population has reached some crucial value the prey stands no chance. If the grid is filled with enough predators, they can maintain their size by reproduction and if there are too few they will die.

Large grid size

The large grid size leaves big rooms for error. The prey has a good chance of avoiding predators even if they are eaten by a front of predators and can reproduce after that. It is

easier to get stable oscillation patterns since they benefit from this room of error. On the other hand, it is difficult to maintain gliders since those are more sensitive to errors.

Medium grid size

This grid size behaves in a in-between sort of way between the small and large grid size. It contains the characteristics of prey being somewhat able to avoid predators, but they are still sensitive to errors in a way the small grid is.

Limitations

There are many factors that are not included in this simulation. One of them is the sense of packs. This can have a strong influence in decrease of a population since it will be more resistant in most cases to attacks or extinction. The other is a sense for attack and defense so the predators wills not take a decreasing population into account.

Decision of parameters

We made a set of reference parameters that will give us some time scale for death probabilities. For this a reference timescale has to be introduced. We decided to set 100 time-steps equivalent to a year, meaning one time step accounts for roughly 3.65 days. The actual numbers don't really matter since the only purpose for the scaling is to somehow get a reference of what the other parameters should be in order for the bears to die later of natural death than the other species.

Summary and Outlook

All in all, we are happy with our results. On the one hand we achieved to get stable and long-lasting oscillations especially in the large grid size. On the other hand we were surprised to achieve gliders in these simulations which we did not expect in this form. The parameters we chose are in no way the only ones to receive similar results. There are always trivial solutions to some results which we did not consider since on the one hand they result in results already achieved by other people.

Appendix

MATLAB Code for Plots

```
clear all
clc
close all
%%


r1 = 0.3;
r2 = 0.8;

a11 = 0;
a12 = 0.3;
a21 = 0.3;
a22 = 0;

b11 = 0.8;
b12 = 0.8;
b21 = 0.6;
b22 = 0.6;

s1 = 0.25;
s2 = 0.25;

d11 = 0.6;
d12 = 0.3;
d21 = 0.6;
d22 = 0.3;

%%

M = @(t, x)[x(1)*(r1-a11*x(1)-a12*x(2)-b11*x(3)-b12*x(4));x(2)*(r2-
a21*x(1)-a22*x(2)-b21*x(3)-b22*x(4));x(3)*(-
s1+d11*x(1)+d12*x(2));x(4)*(-s2+d21*x(1)+d22*x(2))]

%%

[t, x] = ode45(M, [0 100], [10 20 3 6]);

figure
plot(t,x(:,1),'linewidth',1.5,'color',[255/255, 130/255, 171/255])
hold on
plot(t,x(:,2),'linewidth',1.5,'color','r')
hold on
plot(t,x(:,3),'linewidth',1.5,'color',[139/255, 69/255, 19/255])
hold on
plot(t,x(:,4),'linewidth',1.5,'color',[105/255, 105/255, 105/255])
grid;
hold off

legend('deer','rabbit','bear','wolf')
```

Python code of simulation

Survival Game¶

Alexander Bruun, Anton Paris, [Project Survival on GitHub](https://github.com/parisj/project_survival)

A Predator Prey Simulation ¶

Based on Leonel Aguilar's 2D Cellular Automata Code example Link to GitHub example code

Overview - Table of content for faster Navigation

Import Libraries¶

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from random import shuffle
import random
import matplotlib.animation
import pandas as pd
import timeit as timeit
from copy import deepcopy
```

In [2]:

```
%matplotlib inline
#plot style
plt.rcParams["animation.html"] = "jshtml"
```

User Imput¶

Use `Ctrl + F` to find wanted variables¶

Variables that have an great impact on the simulation are noted here¶

GLOBAL VARIABLES :

`r_dim` Amount of rows

`c_dim` Amount of columns

`offspring_chance_table` Chance for offspring

`rules` List of priority for each animal (Where does the animal preferes to go)
`values_offspring` Cell must have at least this value to be able to create offspring
`eat_values` Value gained by eating an cell
`death values` List, increases `death_probability`each turn
`cost_reproduction` Cost for creating offspring
`cost_movement` Cost for moving
`nutrition_level_start` Start Value for new cells and offspring
`nutrition_value_each_turn` Nutrition value that the prey gains each turn

Meaning of the index in Lists¶
Sets parameters for Cells with the state equal to the index!
list_name[i]= [0]Forest, [1]=Rabbits, [2]=Wolves, [3]=Deers ,[4]=Bear

In [3]:

```
global r_dim, c_dim, cells
global offspring_chance_table, values_offspring,
eat_values
global death_values
global cost_reproduction, cost_movement
global rules
global nutrition_level_start,
nutrition_value_each_turn
```

Light test - (Oscillations 100x100 grid)

In [4]:

```
#Run with heavy test cell commented out

#Oscillations 100x100
r_dim= 100
c_dim= 100
cells=[]
max_steps=200

offspring_chance_table=[0,100,100,100,100]
values_offspring=[0,1,1,1,1]
eat_values=[0,1,1,1,1]
nutrition_value_each_turn= 4
cost_reproduction=1.5
cost_movement=1.5
nutrition_level_start=5
death_values=[0,0.091,0.167,0.05,0.033]
rules= [[0],[0],[1,3,0,4],[0],[3,1,0,2]]

#Population represented animals
population=[0,1,2,3,4]

#Represents the probasbility for an animal to be
initialized in the Grid
probability=[0.7,0.25,0.1,0.2,0.01]
```

Heavy test¶

The values for the heavy test can be extractet out of the report for each specific result.¶

In [5]:

```
#Set wanted parameters

#r_dim= 30
#c_dim= 30
#cells=[]
#max_steps= 30

    #--0 means 0% chance, 100 means 100%
probability for offspring.
    #--Predators do have 100% because other rules
will be applied to them
#offspring_chance_table=[0,90,100,95,100]

    #--Contains the information how high the
nutrition_level of an animal has to be to reproduce
#values_offspring=[0,4,5.5,3.95,5.5]

    #--values represent nutrition level boost they
get for EATING certain animal
    #--meaning if a bear [4] eats a wolf [2] the
bear gets the value in [2]
#eat_values=[0,8,6,12,6]

    #--the nutrition_level of the Rabbit[1] and
Deer[3] is increased each turn by:
#nutrition_value_each_turn= 4.3

    #--cost for creating offspring
#cost_reproduction=3

    #--cost for movement
#cost_movement=2.5

    #--Initial value of nutrition, for initializing
a cell and also start value for offspring
#nutrition_level_start=3.9
```

```
    #--Values represent the ageing of an animal,
value is added to death_probability each turn
#death_values=[0,0.091,0.167,0.05,0.033]

    #--Defines the behavior while looking for
possible movement
    #--Bear [4] prefers [rabbit [1], deer[3],
Wolf[2], Forest [0]]
#rules= [[0],[0],[1,3,0,4],[0],[3,1,0,2]]

    #--Population represents animals
#population=[0,1,2,3,4]

    #--Represents the probasbility for an animal to
be initialized in the Grid
#probability=[0.7,0.25,0.1,0.2,0.01]
```

Create Class Cell¶

Properties of the class Cell.¶

`type_cell` : [0] Forest, [1]=Rabbit, [2]=Wolf, [3]=Deer ,[4]=Bear
`lifespan` : how many turns the animial is alife
`nutrition_level`: Energy level, increases when animal eats something, decreases when it moves
`reproduction_count`: Counts Offspring
`death_probability`: Probability that the animal dies in a round, increases steady each turn
`turn` True if the cell already had it's turn in the timestep
`offspring_chance` Chance for mating

In [6]:

```
class Cell():
    #initialize cell function
    def __init__(self, type_cell,
offspring_chance_table):
        self.type_cell = type_cell[0]
        self.lifespan = 0
        if type_cell[0]!=0:
            self.nutrition_level=
nutrition_level_start
        else:
            self.nutrition_level=0
        self.reproduction_count=0
        self.death_probability=1
        self.turn=False

self.offspring_chance=offspring_chance_table[type_c
ell[0]]

    #getter function
    def get_type (self):
        return self.type_cell

    #PRE: Value of the wanted increase of lifespan
    #POST: Increases the lifespan of the animal by
a value
    #USED FOR: Eat Mechanism
    def increase_l (self, value):
        self.lifespan+= value
       # print(self.lifespan, "self.lifespan")

#TURN FUNCTIONS

    #PRE: -
    #POST: turn= true
    #USED FOR: Limit the number of turns in a
timestep to 1
    def set_turn(self):
```

```
            self.turn=True
            # print(self.turn,"after set turn ")

        #PRE: -
        #POST: turn= false
        #USED FOR: Start a the round to reset the turn boolean to false: Cell is allowed to apply rules
        def round_start(self):
            # print(self.turn,"before round start")
            self.turn=False


    #CHANGE CELLS FUNCTIONS



        #PRE: value of the nutrition_level that will be gained (can also be 0 if forest)
        #POST: increased nutrition_level
        #USED FOR: perform nutrition gain by eating an animal/forest
        def eat (self, value):
            self.nutrition_level+= value


        #PRE: Coordinates of a second Cell where the prime Cell(currently evaluated )
        #POST: dublicates attributes to the Cell with the Coordinates passed (coords2)
        #USED FOR: Part of the movement function, simulate a jump to a new cell (with coords 2)
        def copy_attributes(self, coords_2):
            # print("copying attributes,", cells[coords_2[0]][coords_2[1]].type_cell,self.type_cell)

cells[coords_2[0]][coords_2[1]].type_cell=self.type_cell
```

```
        cells[coords_2[0]][coords_2[1]].lifespan =
self.lifespan

cells[coords_2[0]][coords_2[1]].nutrition_level =
self.nutrition_level
        cells[coords_2[0]][coords_2[1]].turn=
self.turn

cells[coords_2[0]][coords_2[1]].reproduction_count=
self.reproduction_count

cells[coords_2[0]][coords_2[1]].death_probability=s
elf.death_probability

    #PRE:-
    #POST: increase reproduction counter
    #USED FOR: offspirng mechanism
    def offspring(self):
        self.reproduction_count+=1

    #PRE:-
    #POST: Sets a cell to type Forest and reset
attributes of the cell
    #USED FOR: simulating death
    def death (self):
        self.lifespan=0

#print(cells[coords[0]][coords[1]].type_cell,"
before")
        self.type_cell=0 #forest
        self.nutrition_level=0
        self.reproduction_count=0
        self.death_probability=1

#print(cells[coords[0]][coords[1]].type_cell,"
after")
```

```
    #PRE:-
    #POST: returns bool of random death (age based)
    #USED FOR: random death because of age
    def death_age(self):
        return (random.randint (1,100) <
self.death_probability)


    #PRE: death_values is a list with 5 values,
index of values must represent the different cell
types
    #POST: returns the ageing value for the called
cell
    #USED FOR: Information of the ageing process of
each individual animal
    def get_value_death_age(self,death_values):
        return death_values[self.type_cell]


    #PRE: friends is an integer representing how
many cells with the same cell type are in the
neighborhood
    #POST: Returns true or false, true= has the
right to reproduce, false has not the right to
reproduce
    #USED FOR: Figuring out, if the Cell has the
right to reproduce
    def offspring(self,friends,values_offspring):
        #Rabbit,
        if self.type_cell==1:
            return (friends and
self.nutrition_level>=values_offspring[self.type_ce
ll] and
random.randint(1,100)<self.offspring_chance)
        #Deer
        elif self.type_cell==3:
            return (friends and
self.nutrition_level>=values_offspring[self.type_ce
```

```
ll] and
random.randint(1,100)<self.offspring_chance)
        #Wolf, bear and forest: forest is filtered
later out!
        else:
            return (friends and
self.nutrition_level>=values_offspring[self.type_ce
ll])

    #PRE: 2D List filled with instance cell(cells),
Coordinations of motercell (after movement)
    #POST: Cell becomes a dublicate of the
mothercell with reset attributes, substracts cost
of birth of the mothercell
    #USED FOR: define the child with the attributes
and adjust the attributes ot the mothercell (adapt
nutrition level and reproduction count)
    def birth(self,coords_2,cells):
        self.lifespan=1
        self.nutrition_level=nutrition_level_start
        self.reproduction_count=0
        self.death_probability=1

cells[coords_2[0]][coords_2[1]].reproduction_count+
=1
        #print(
cells[coords_2[0]][coords_2[1]].nutrition_level,"be
fore repro")

cells[coords_2[0]][coords_2[1]].nutrition_level-
=cost_reproduction
        #print(
cells[coords_2[0]][coords_2[1]].nutrition_level,"af
ter repro")
        #print(self.type_cell, "type_cell birth",
self.nutrition_level, "nutritionlevel birth")
```

```
    #PRE:
    # Variables Overview:
        #Cells= 2D List of Cells, coords_1=
Coordinates of the Cell that is evaluated,
coords_2= Position of valid move
        #increase_value_lifespan= value increses
lifespan, increase_value_eat= value based on the
type of the cell at the endposition
        #friends= number of cells in the
neighborhood with the same cell type (meaning same
animal)
    #POST:
    #Options
        #Has already move in this timestep? nothing
happens
        #Prey gains nutrition
        #Random death by age
        #Ageing of the Cells
        #Cost of movement
        #Move from coord 1 to coord 2
        #Give birth
        #
    #USED FOR: Heart of the animation
    def set_movement(self, cells, coords_1,
coords_2, increase_value_lifespan,
increase_value_eat, friends,
nutrition_value_each_turn):
        #print(coords_1,coords_2)

#print("coords_2",cells[coords_2[0]][coords_2[1]].t
ype_cell," coords_1",self.type_cell )

#cells[coords_2[0]][coords_2[1]]=cells[coords_1[0]]
[coords_1[1]]

#print("coords_2",cells[coords_2[0]][coords_2[1]].t
ype_cell,"
```

```
coords_1",cells[coords_1[0]][coords_1[1]].type_cell
)
        #print(self.turn, "turn attribute of cell")
        if self.turn==False:

            self.set_turn()
            #Block Forest cells
            if self.type_cell==0:
                return

            if self.type_cell==1 or
self.type_cell==3:

self.nutrition_level+=nutrition_value_each_turn
            elif self.type_cell!=0 and
self.death_age():
                self.death()
              #  print("death by age")
                return


self.increase_l(increase_value_lifespan)
            self.eat(increase_value_eat)

self.death_probability+=self.get_value_death_age(de
ath_values)

            if coords_1!=coords_2:
                if self.type_cell!=0 and
self.type_cell!=3:

#print(self.nutrition_level,"before movement")
                    self.nutrition_level-
=cost_movement

#print(self.nutrition_level,"after movement")
                    if self.nutrition_level<0:
                        self.death()
```

```
                        return
                    self.copy_attributes(coords_2)
                    if
self.offspring(friends,values_offspring)and
self.nutrition_level>0 :
                        self.birth(coords_2,cells)

                    else:
                        self.death()
                    # print("death by movement")

#print("coords_2",cells[coords_2[0]][coords_2[1]].t
ype_cell,"
coords_1",cells[coords_1[0]][coords_1[1]].type_cell
)
            #print("from: ", coords_1, "to: ",
coords_2)
        return
```

In [7]:

```
#Function does not belong to the class cell but is
related!

#PRE: 2D List of Cell, Coordinates of cell, list of
eat_values
#POST: returns Value for certain cell if eaten
#USED FOR: identifiy eat value
def  get_value_eat(cells, coords, eat_values):

    return
eat_values[cells[coords[0]][coords[1]].type_cell]
```

Generate 2D List with Cells¶

In [8]:

```
#empty grid
#PRE:-
```

```
#POST: array created with [rows][colums]
def initGrid(cols,rows,array):
    for i in range (rows):
        array.append([1])
        for j in range (cols):
            array[i].append(0)
```

Fill list random¶

In [9]:

```
#POST: fills Grid with random numbers and the right
attributes
def fillGrid(rows,cols,array):
    for i in range (rows):
        for j in range (cols):

array[i][j]=Cell(random.choices(population,probabil
ity),offspring_chance_table)
```

In [10]:

```
initGrid(r_dim, c_dim, cells)
%timeit fillGrid(r_dim, c_dim, cells)

30.5 ms ± 1.97 ms per loop (mean ± std. dev. of 7
runs, 10 loops each)
```

Get informations out of Cells¶

Filters all the informations out of the Grid and safes them in an convenient way: **[attribute][row][column]**

[0] [i] [j]= **type_cell** **[1]** [i] [j]= **lifespan** **[2]** [i] [j]= **reproduction counter** **[3]** [i] [j]= **nutrition_level**

In [11]:

```
def update_grid(r_dim,c_dim,cells):
    grid_list=np.zeros((4,r_dim,c_dim))

    for i in range(0,r_dim,1):
        for j in range(0,c_dim,1):
            grid_list[0][i][j]=
cells[i][j].type_cell
            grid_list[1][i][j]=
cells[i][j].lifespan
            grid_list[2][i][j]=
cells[i][j].reproduction_count
            grid_list[3][i][j]=
cells[i][j].nutrition_level

    return grid_list
```

Testing if informations are extractet the correct way¶

In [12]:

```
grid_list=update_grid(r_dim,c_dim,cells)
```

In [13]:

```
#grid_list[0][:][:] #type_cell
#grid_list[1][:][:] #lifespan
#grid_list[2][:][:] #reproduction_count
#grid_list[3][:][:] #nutrition_level
```

Define Neighborhood¶

Return of the `getNeighbourhoodValues` function:¶

`mycoords: [row,column]` `mystate=`type.cell at the location of mycoords, scalar `neighstate: [[state,row,column],[state,row,column]...]`

- *tests are below the next cell*

How Periodic boundaries are handled in `getNeighborhood` function¶

Line 15: `[(coords[0] + i)%(r_dim), (coords[1] + j)%(c_dim)]`

In [14]:

```
#PRE: idx has to be in the interval of r_dim*c_dim
#POST: returns list with unique Coordinates which
are within bounds of the grid
def idxToCoordinates(idx,r_dim):
    return [idx%r_dim,idx//r_dim]

#PRE: Dimensions of the Grid, and idx has to be in
the interval of r_dim*c_dim
#POST: Returns unique coordinates based on the idx,
also returns a list with the 8 neighbors
coordinates
def getNeighbourhood(idx, r_dim, c_dim):
    neighb = []
    coords = idxToCoordinates(idx, r_dim)
    for j in range(-1, 2, 1):
        for i in range(-1, 2, 1):
            # Handle boundary conditions

            neighb.append([(coords[0] + i)%(r_dim),
(coords[1] + j)%(c_dim)])
    #print (coords, neighb)
    return (coords, neighb)


#PRE: idx in the interval of r_dim*c_dim, 2D list
filled with instance Cell
#POST: Returns list with coordinates linked to the
idx, the state of the coordinates linked to the idx
```

```
        # list of the neighbor's coordinates
indexing is as described in "Return of the
function"
def getNeighbourhoodValues(idx, grid, r_dim,
c_dim):
    mycoords,neighCoords=getNeighbourhood(idx,
r_dim, c_dim)


mystate=grid[mycoords[0]][mycoords[1]].type_cell
    neighstate=[]

     #neighstate= [[row,column],[row,column]...]
        #neigh iterates through neighstate[i]
        #neigh=[row,column]

    for neighc in neighCoords:
        if (neighc[0] != mycoords[0] or neighc[1]
!= mycoords[1]):


neighstate.append([grid[neighc[0]][neighc[1]].type_
cell, neighc[0], neighc[1]])

    #print("- Coords ",mycoords," = ",mystate,"
neigs ",neighstate," = ",neighCoords)
    #print(mycoords,neighstate)
    return mycoords,mystate,neighstate
```

Testing if neighborhood is extracted the right way¶

In [15]:

```
#t_idx=idxToCoordinates(10,10)
```

```
#t_mycoord,t_mystate,t_neighstate=getNeighbourhoodV
alues(10,cells,10,10)
```

In [16]:

```
#t_idx
#t_mycoord
#t_mystate
#t_neighstate
```

Filter possible neighbor cells out¶

Return of the `filteringstates` function:¶

`filtered_neighstate`:
`[[state,row,column],[state,row,column]...]`
`offspring`: `True` or `False`

In [17]:

```
#PRE:
    # mystate= scalar
    # neighstate=
[[state,row,column],[state,row,column]...]
    # rules=
[(Forest),(Rabbit),(Wolf),(Deer),(Bear)]
             #Bear= [Prefered Cells ordered]...
             #EXAMPLE Bear=[1,3,2,0]

#POST:
    #Returns filtered_neighstate
        #List of all the neighbor cells that are
considered to jump
    #Returns bool Offspring
        #True if at least one cell with same
type_cell in neighborhood
        # else False
```

```
def filteringstate(mystate, neighstate, rules):
#rule [mystate, states allowed to interact]

    #REMINDER: rules=
[[0],[0],[3,1,0],[0],[1,3,2,0]]
        #rules [i] correlates to the type_cell and
it's prefered cells.

    filter_states=rules[mystate]
    filtered_neighstate=[]


    #Loop count the number of cells with the same
type_cell in the neighborhood
    my_friends=0
    if mystate!=0:
        for friends in neighstate[0]:
            if mystate==friends:
                my_friends+=1
    #if more more than 0 friends are in the
neighborhood, one criteria for
    #creating offspring is given, so bool offspring
is true.
    offspring=(my_friends>0)
    #print(filter_states)


    #REMINDER: rules=
[[0],[0],[3,1,0],[0],[1,3,2,0]]
            #rules [i] correlates to the type_cell
and it's prefered cells.
    #EXAMPLE: rules [4]= Bear
            #Bear preferes: [1,3,2,0] in this
order!

    for possible in filter_states:
```

```
        #neighstate=
[[state,row,column],[state,row,column]...]
        #neigh iterates through neighstate[i]
        #neigh=[state,row,column]

        for neigh in neighstate:
            if possible == neigh[0]:

filtered_neighstate.append([neigh[1],neigh[2]])
        #Has to quit the loop if the the iteration
with a certain priority is finished
        #and the list of the possible cells is !=0:
so the priority of the rules is kept!
        if len(filtered_neighstate)!=0:
            return filtered_neighstate, offspring
    return filtered_neighstate, offspring
```

Apply Rule¶

Idea:¶

The `apply_rule` function gives the **coordinates back**, where the **cell, that is evaluated, will jump to**. The evaluated cell eats the cell at the other position and the value_eat will be added to the cell's nutrition_level. The

function also returns an boolean stating if the conditions are met for creating offspring.
The Variable `number`is used to randomise the direction the cell moves. It chooses a random index in the list of `neighcoord`. This is done because the `getNeighborhood` function (where the neighcoord list originates) adds the neighbor coordinates in a certain order and this order is never shuffled until in `apply_rule`

Return of the `apply_rule` function:¶

`mycoords` / `neighcoord[number]`: `[row,column]`
`value_eat`: Scalar, depending on the eaten cell. - nutrition_level will be increase by this value later on
`offspring`: `True` or `False`

The rules of the `apply_rule` function¶

`mystate==0: [Forest]`¶

The cell can not move, so it gives it's own coordinates back. `value_eat` will be 0 by definition of the `get_value_eat`function. Also the conditions for offspring are never met for a `mystate`==0 cell
Return: `mycoords`, `value_eat`, `False``

`mystate==1: [Rabbit]`¶

The cell is allowed to move if `len(neighcoord)!=0:`. `value_eat` will be 0 by definition of the `get_value_eat`function. Conditions for offspring can be met, bool will be passed through.

Return: `len(neighcoord)!=0:neighcoord[number], value_eat,offspring len(neighcoord)==0:mycoords,0, False`

mystate==2: [Wolf]¶

The cell is allowed to move if `len(neighcoord)!=0:.value_eat` will be determined of the eaten cell by the `get_value_eat`function. Conditions for offspring can be met, boll will be passed through.

Return: `len(neighcoord)!=0:neighcoord[number], value_eat,offspring len(neighcoord)==0:mycoords,0, False`

mystate==3: [Deer]¶

The cell is allowed to move `if len(neighcoord)!=0:.value_eat` will be 0 by definition of the `get_value_eat`function. Conditions for offspring can be met, bool will be passed through.

Return: `len(neighcoord)!=0:neighcoord[number], value_eat,offspring len(neighcoord)==0:mycoords,0, False`

mystate==4: [Bear]¶

The cell is allowed to move `if len(neighcoord)!=0:.value_eat` will be determined of the eaten cell by the `get_value_eat`function.. Conditions for offspring can be met, bool will be passed through.

Return: `len(neighcoord)!=0:neighcoord[number], value_eat,offspring len(neighcoord)==0:mycoords,0, False`

In [18]:

```
#PRE:
    #mystate=scalar, mycoords=[rown,column], grid=
2D list of all type_cell,
    #neighstate=
[[state,row,column],[state,row,column]...]
    #rules=
[(Forest),(Rabbit),(Wolf),(Deer),(Bear)]
             #Bear= [Prefered Cells ordered]...
             #EXAMPLE Bear=[1,3,2,0]
    #cells= 2d list of instance Cell

#POST:
    #Returns coordinates where the evaluated cell
will move to
    #value_eat that will increase nutriotion_level
of the evaluated cell
    # and a bool if the cell hast the right to
reproduce
def apply_rule(mystate, mycoords, neighstate, grid,
rules, cells):

    neighcoord, offspring=filteringstate(mystate,
neighstate, rules)
    #print(neighcoord," neighcoord")
    #print(len(neighcoord),"len(neighcoord)")
    if len(neighcoord)!=0:

        #to randomise the direction of the
movement.

number=(np.random.randint(0,len(neighcoord)))
        #print(number,"number,", mystate, "
mystate")

        if mystate==0: #forest

value_eat=get_value_eat(cells,neighcoord[number],
eat_values)
```

```
            #print(mystate,"mystate 0 apply rule",
neighcoord[number], value_eat)
            return mycoords, value_eat, False

        elif mystate==1: #rabbit

value_eat=get_value_eat(cells,neighcoord[number],
eat_values)
            #print(mystate,"mystate 1 apply rule",
neighcoord[number], value_eat)
            #print (offspring, "offspring")
            return neighcoord[number], value_eat,
offspring

        elif mystate==2: #Wolf
           # print(mystate,"mystate 2 apply rule")

value_eat=get_value_eat(cells,neighcoord[number],
eat_values)
            #print (offspring, "offspring")
            return neighcoord[number], value_eat,
offspring

        elif mystate==3: #Deer
           # print(mystate,"mystate 3 apply rule")

value_eat=get_value_eat(cells,neighcoord[number],
eat_values)
            return neighcoord[number], value_eat,
offspring

        elif mystate==4: #Bear
           # print(mystate,"mystate 4 apply rule")

value_eat=get_value_eat(cells,neighcoord[number],
eat_values)
            return neighcoord[number], value_eat,
offspring
```

```
    #Can't move:
    else:
        return mycoords, 0, False
```

Define the step¶

The `step` function¶

The `step` function is used to apply all the rules with the `apply_rule` function. At the end of `step` `update_grid` is used to make a **snapshot** of the current constelation of the cells (mainly because the `cells` are constantly changed and without creating a history)

returns `grid_list`¶

updated snapshot of `cells` with all the attributes connected to it.

`grid_list` is build up as following: **[attribute][row][column]**

[0] [i] [j]= **type_cell** **[1]** [i] [j]= **lifespan** **[2]** [i] [j]= **reproduction counter** **[3]** [i] [j]= **nutrition_level**

In [19]:

```
#PRE:
    #old grid_list ,
    #rules=
[(Forest),(Rabbit),(Wolf),(Deer),(Bear)]
             #Bear= [Prefered Cells ordered]...
             #EXAMPLE Bear=[1,3,2,0]
    # 2D list of instance Cell

#POST: returns updated grid_list of the form
described above
```

```
def step(grid_list,rules, cells):

    for i in range (r_dim):
        for j in range (c_dim):
            cells[i][j].round_start()


    all_idx = list(range(0, r_dim*c_dim,1))
    #to randomise which cell is evaluated
    shuffle(all_idx)

    #Iterate through all cells (C) in the automata
and extract its neighbourhood
    for idx in all_idx:
        #print("cell ",mycoords)
        mycoords,mystate,neighstate =
getNeighbourhoodValues(idx,cells,r_dim,c_dim)
        #print("cell ",idx," - ",mycoords)

        #neighcoord is the location where the
evaluated cell will jump to
        neighcoord, value_eat,
offspring=apply_rule(mystate,mycoords, neighstate,
grid_list[0][:][:], rules, cells)

        #def set_movement(self, cells, coords_1,
coords_2, increase_value_lifespan,
increase_value_eat):

cells[mycoords[0]][mycoords[1]].set_movement(cells,
mycoords,neighcoord,1,value_eat,
offspring,nutrition_value_each_turn)

        #print(all_idx)
        #print(mycoords,neighcoord)

    grid_list=update_grid(r_dim,c_dim,cells)
```

```
    #print(state,"state", cells[5][4].type_cell,
"cells")
    return grid_list
```

Define the simulation¶

The `simulation` function¶

The `simulation` function executes the `step` function for `max_step`times. In each step the 2D list `cells` will be changed and the returned `grid_list` of the step simulation (snapshot of the `cells` after a turn) will be stacked up in `cells_stack`

`cells_stack` is made up as following:
[step][attribute][row][column] [t] **[0]** [i] [j]= **type_cell** [t] **[1]** [i] [j]= **lifespan** [t] **[2]** [i] [j]= **reproduction counter** [t] **[3]** [i] [j]= **nutrition_level**

In [20]:

```
#PRE:
    #X= 2D list filled with instance Cell
    #max_steps= amount of timestep
    #rules=
[(Forest),(Rabbit),(Wolf),(Deer),(Bear)]
            #Bear= [Prefered Cells ordered]...
            #EXAMPLE Bear=[1,3,2,0]

#POST: returns cells_stack, cell_stack contains
max_steps snapshots of X
```

```
        #including all the information of the
attributes

def simulation(X,max_steps,rules):

    #Because X is a list in a list with instances
of cell, the initial snapshot
    #Also has to be done over the update_grid
function. Otherwise ERROR
    grid_list=update_grid(r_dim,c_dim,X)
    cells_stack=[grid_list]

    for i in range(max_steps):
        #Moves one step forward
        grid_list=step(grid_list,rules,X)
        #Stores the result

        cells_stack.append(grid_list)

    #print("cells stack:",cells_stack)
    return cells_stack
```

Persomfing the simulation¶

In [21]:

```
%time results=simulation(cells,max_steps,rules)

CPU times: user 33.1 s, sys: 332 ms, total: 33.5 s
Wall time: 35.2 s
```

Define Plots¶

Define plot Type_Cell¶

In [22]:

```
#PRE: cellHistory= 2D list of scalars,
#POST: Ploted Result timestep
def PlotAutomataHistory(cellHistory,ax=None):


    if ax is None:
        fig, ax, = plt.subplots(1, 1, figsize=(9,
9)) #Determine the number of subplots in the figure
and its size (scaling)

    minVal=0
    maxVal=4

    cmap = plt.cm.gray
    norm = plt.Normalize(minVal,maxVal)
    rgba = cmap(norm(cellHistory))
    #Filtering types out
    F = np.argwhere(cellHistory==0)
    R = np.argwhere(cellHistory==1)
    W = np.argwhere(cellHistory==2)
    D = np.argwhere(cellHistory==3)
    B = np.argwhere(cellHistory==4)

    #choosing colors
    green=[0.8,1,0.8]
    white=[0.7529,0.7529,0.7529]
    grey=[0.25,0.25,0.25]
```

```
    brown=[1,0.698,0.4]
    black=[0.6,0.298,0]

    for position,color in
zip([F,R,W,D,B],[green,white,grey,brown,black]):
        for pos in position:
            rgba[pos[0]][pos[1],:3]=color

    ax.imshow(rgba,origin='lower',
interpolation='none') #Type of plot
    #Axis labels

    ax.set_xlabel('Columns',fontsize='x-large')
    ax.set_ylabel('Rows',fontsize='x-large')
    ax.set_title('Type_cell',fontsize='xx-large')
```

Define plot lifespan¶

In [23]:

```
#Plot lifespan of all cells in the grid.

def PlotLifespan(cell_lifespan, ax=None):

    if ax is None:
        fig, ax = plt.subplots(1,1, figsize=(9,9))
    #the oldest cell will be black and the youngest
white
    #creating an returning im for adding the
colormap to it
    #possible way better solution availabel but
couldn't figure it out

im=ax.imshow(cell_lifespan,'Greys',origin='lower',a
nimated=True)
```

```
    ax.set_xlabel('Columns',fontsize='x-large')
    ax.set_ylabel('Rows',fontsize='x-large')
    ax.set_title('Lifespan/animal',fontsize='xx-
large')
    return im
```

Test plot¶

In [24]:

```
#%time PlotAutomataHistory(results[max_steps][0])
```

In [25]:

```
#%time PlotLifespan(results[max_steps][1])
```

Animate plot¶

In [26]:

```
%%capture
# Build plot and animate it

fig, ax = plt.subplots(figsize=(15,15))
fig2, ax1 = plt.subplots(figsize=(15,15))
resultsCopy= deepcopy(results)
im=PlotLifespan(results[max_steps][1],ax=ax1)

#type animation
def animate_type(j):
    ax.clear()
    PlotAutomataHistory(results[j][0],ax=ax)
```

```
#lifespan animation
def animate_life(j):
    ax1.clear()
    PlotLifespan(results[j][1],ax=ax1)

cbar=fig2.colorbar(im,ax=ax1,ticks=[0,max_steps])
cbar.ax.set_yticklabels(['youngest',
'oldest'],fontsize='large')
```

Uncomment if animation should be created¶

In [27]:

```
#TYPE_CELL ANIMATION
ani_type = matplotlib.animation.FuncAnimation(fig,
animate_type, frames=len(results))
ani_type #Animation of Type_cell
```

Out[27]:

Once Loop Reflect

In [28]:

```
#LIFESPAN ANIMATION
#ani_life =
matplotlib.animation.FuncAnimation(fig2,
animate_life, frames=len(results))
#ani_life #Animation of Lifespan
```

Plot results¶

Function for filtering out wanted attribute¶

`filter_attribute` function¶

The 4D Grid Results is filtered by `type_cell` and the wanted `attribute`.

returns 3d List with the structure of `filtered[time_step][row][column]`

`filter_type` function¶

Uses `filter_attribute` to extract all the attributes related to `type_cell`

`panda_structure`: `True / False`, is used to determine the format of the output.

`True`: attributes are first flatten and then converted into an table. Tabel has the structure: rows represent the time steps and columns each cell in the grid.

`False`: attributes are 3D lists with the structure: `name[timestep][row][column]`

In [29]:

```
#PRE:
    #Results= Results of the simulation with
indexes:
        #[step][attribute][row][column]
    #type_cell of the wanted filter: [0,1,2,3,4]=
representing the animals
    #attribute for filtering the attribute in
results [attribute]
```

```
#POST: returns "filtered" with
"filtered[steps][row][column]""
        #depending on input different attribute is
filtered out

def filter_attribute(results,type_cell,attribute):
    filtered=np.zeros([max_steps+1,r_dim,c_dim])

    #print(attribute,"--------------ATTRIBUTE------
------------")
    for t in range(max_steps+1):
        #print(t,"---------------STEP-------------
")
        for i in range(r_dim):
            #print(i, "----------------ROW---------
-----")
            for j in range(c_dim):
                #print(j,"---------------COLUMN----
---------")
                if type_cell==results[t][0][i][j]:

filtered[t][i][j]=results[t][attribute][i][j]
    #print(filtered)
    return filtered


#PRE:
    #Results= Results of the simulation with
indexes:
        #[step][attribute][row][column]
    #type_cell, scalar (0,1,2,3,4), of which
type_cell information should be extracted
    #Panda_structure, decides which format the data
should have
        #TRUE: Converts the extractet Lists into an
Table with
                #row= timesteps,
                #columns= cells
```

```
        #FALSE: Data stays in list form

#POST: depending on Panda_structure bool it return
either all the attributes
        # in 3D list format or Flatten into a table

def filter_type(results,type_cell,Panda_structure):
    type_c_m=filter_attribute(results,type_cell,0)

    #print("done type_cell",type_cell ,type_c_m)
    lifespan=filter_attribute(results,type_cell,1)
    #print("done lifespan",type_cell , lifespan)

reproduction_count=filter_attribute(results,type_ce
ll,2)
    #print("done repro",type_cell
,reproduction_count)

nutrition_level=filter_attribute(results,type_cell,
3)
    #print("done nut",type_cell, nutrition_level)

    if Panda_structure:
        t_c_f=[time.flatten() for time in type_c_m]
        #print("type_cell", type_c_m)
        #print("type_cell", t_c_f)
        l_s_f=[time.flatten() for time in lifespan]
        #print("lifespan N", lifespan)
        #print("lifespan F", l_s_f)
        r_c_f=[time.flatten() for time in
reproduction_count]
        #print("Re Pro N", reproduction_count)
        #print("Re Pro F", r_c_f)
        n_l_f=[time.flatten() for time in
nutrition_level]
        #print("Nu N", nutrition_level)
        #print("Nu F", n_l_f)
```

```
        df_type= pd.DataFrame(t_c_f)
        #print("type",df_type)
        df_life= pd.DataFrame(l_s_f)
        #print("life",df_life)
        df_repro= pd.DataFrame(r_c_f)
        #print("repro",df_repro)
        df_nutri= pd.DataFrame(n_l_f)
        #print("nutri",df_nutri)

        return df_type, df_life, df_repro, df_nutri

    return type_c_m, lifespan, reproduction_count,
nutrition_level
```

Filtering all attributes of `type_cell` [1,2,3,4]¶

In [30]:

```
#For checking if everything worked:
#T0,L0,R0,N0=filter_type(results,0,True)

%time T1,L1,R1,N1=filter_type(results,1,True)
#Rabbit
%time T2,L2,R2,N2=filter_type(results,2,True) #Wolf
%time T3,L3,R3,N3=filter_type(results,3,True) #Deer
%time T4,L4,R4,N4=filter_type(results,4,True) #Bear

CPU times: user 7.32 s, sys: 293 ms, total: 7.62 s
Wall time: 8.22 s
CPU times: user 8.46 s, sys: 243 ms, total: 8.7 s
Wall time: 8.96 s
CPU times: user 10.1 s, sys: 219 ms, total: 10.4 s
Wall time: 10.6 s
CPU times: user 7.45 s, sys: 203 ms, total: 7.65 s
```

```
Wall time: 7.9 s
```

Format Data for plots¶

In [31]:

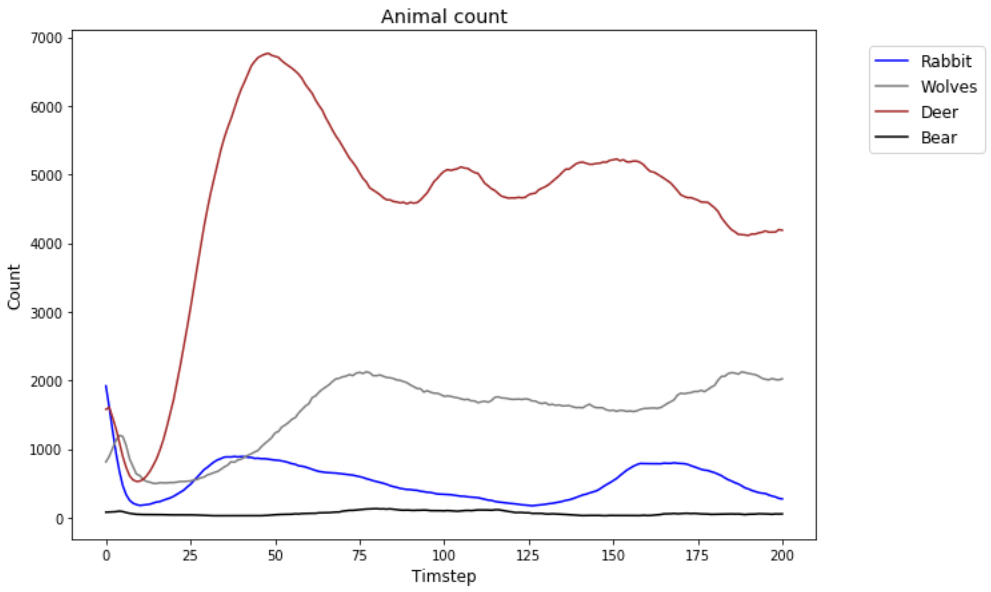
```
x=np.arange(0.0,max_steps+1,1)

#Variables for plotting

size=(10,7) #size plot
width=1.5   #linewidth
font='large'#fontsize
fonttitle='x-large'#fonttitle size


#Total of type_cell in grid of each timestep
R_t=T1[T1==1].count(axis='columns') #Rabbit
W_t=T2[T2==2].count(axis='columns') #Wolves
D_t=T3[T3==3].count(axis='columns') #Deer
B_t=T4[T4==4].count(axis='columns') #Bear

#Total livespan in grid of each timestep
R_l=L1.sum(axis='columns')
W_l=L2.sum(axis='columns')
D_l=L3.sum(axis='columns')
B_l=L4.sum(axis='columns')


#Total of lifespan per type_cell in grid of each
timestep
R_L_P=R_l.div(R_t)
W_L_P=W_l.div(W_t)
D_L_P=D_l.div(D_t)
B_L_P=B_l.div(B_t)
```

```
#fill Series Nan with 0
R_L_P=R_L_P.fillna(0)
W_L_P=W_L_P.fillna(0)
D_L_P=D_L_P.fillna(0)
B_L_P=B_L_P.fillna(0)

#Counter of Offspring
R_r=R1.sum(axis='columns')
W_r=R2.sum(axis='columns')
D_r=R3.sum(axis='columns')
B_r=R4.sum(axis='columns')

#Total Nutrition level of all type_cell in grid of
each timestep
R_n=N1.sum(axis='columns')
W_n=N2.sum(axis='columns')
D_n=N3.sum(axis='columns')
B_n=N4.sum(axis='columns')

#Total of lifespan per type_cell in grid of each
timestp
R_N_P=R_n.div(R_t)
W_N_P=W_n.div(W_t)
D_N_P=D_n.div(D_t)
B_N_P=B_n.div(B_t)

#fill Series Nan with 0
R_N_P=R_n.fillna(0)
W_N_P=W_n.fillna(0)
D_N_P=D_n.fillna(0)
B_N_P=B_n.fillna(0)
```

Results¶

In [32]:

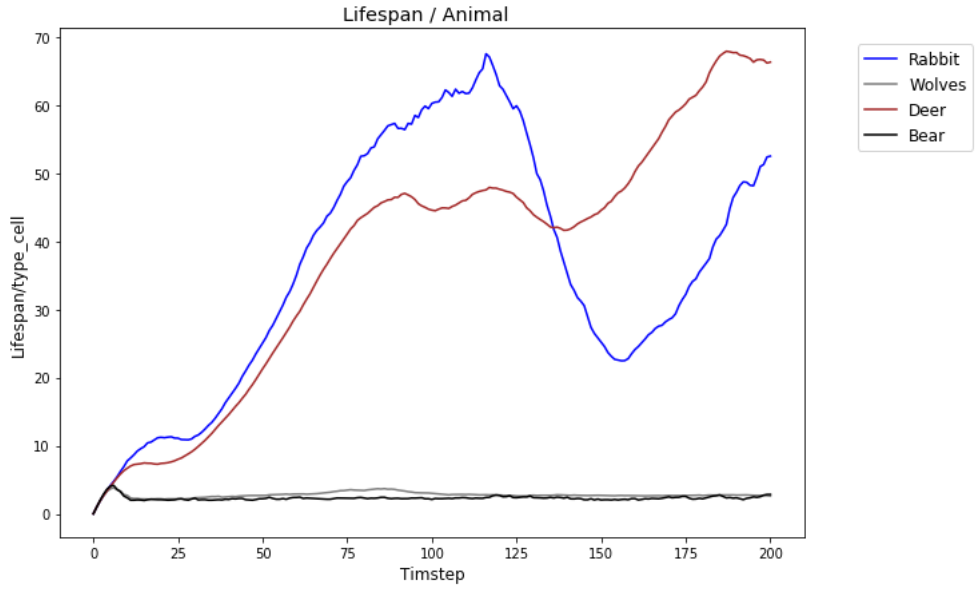
```
#TYPE_CELL PLOT (count of each animal in the grid)
plt.figure(figsize=size)
plt.plot(x,R_t,'blue',linewidth=width,
label='Rabbit')
plt.plot(x,W_t,'grey',linewidth=width,
label='Wolves')
plt.plot(x,D_t,'brown',linewidth=width,label='Deer'
)
plt.plot(x,B_t,'black',linewidth=width,label='Bear'
)
plt.xlabel('Timstep',fontsize=font)
plt.ylabel('Count',fontsize=font)
plt.title('Animal count',fontsize=fonttitle)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2,
borderaxespad=1.,fontsize=font)
plt.show()
```



In [33]:

```
#LIFSPAN/ANIMAL PLOT (If results are not too clear,
view prey and predator separately)
plt.figure(figsize=size)
```

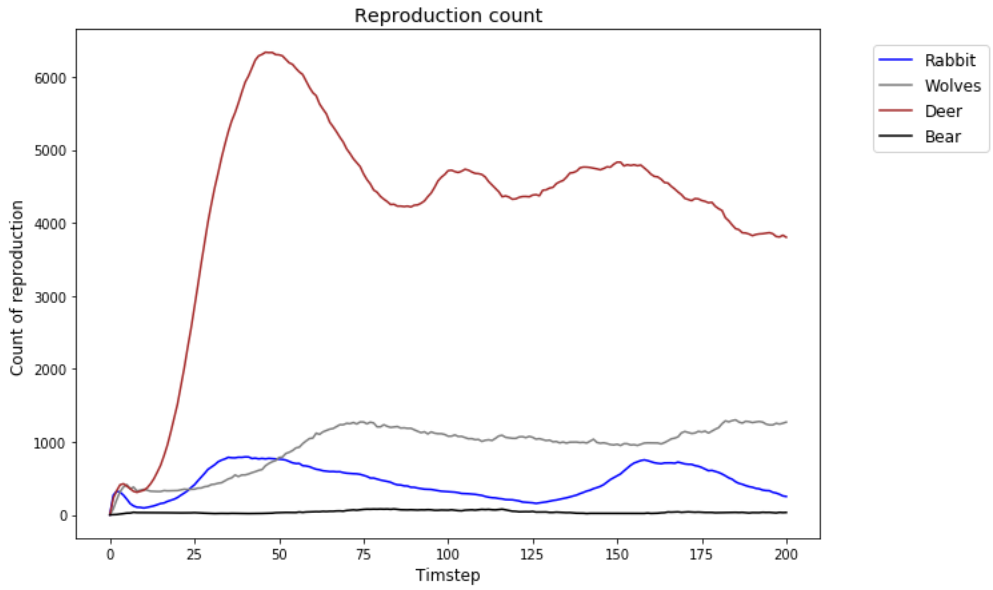
```
plt.plot(x,R_L_P,'blue',linewidth=width,
label='Rabbit')
plt.plot(x,W_L_P,'grey',linewidth=width,
label='Wolves')
plt.plot(x,D_L_P,'brown',linewidth=width,label='Dee
r')
plt.plot(x,B_L_P,'black',linewidth=width,label='Bea
r')
plt.xlabel('Timstep',fontsize=font)
plt.ylabel('Lifespan/type_cell',fontsize=font)
plt.title('Lifespan / Animal  ',fontsize=fonttitle)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2,
borderaxespad=1.,fontsize=font)
plt.show()
```



In [34]:

```
#REPRODUCTION COUNT PLOT
plt.figure(figsize=size)
plt.plot(x,R_r,'blue',linewidth=width,
label='Rabbit')
plt.plot(x,W_r,'grey',linewidth=width,
label='Wolves')
plt.plot(x,D_r,'brown',linewidth=width,label='Deer'
)
```

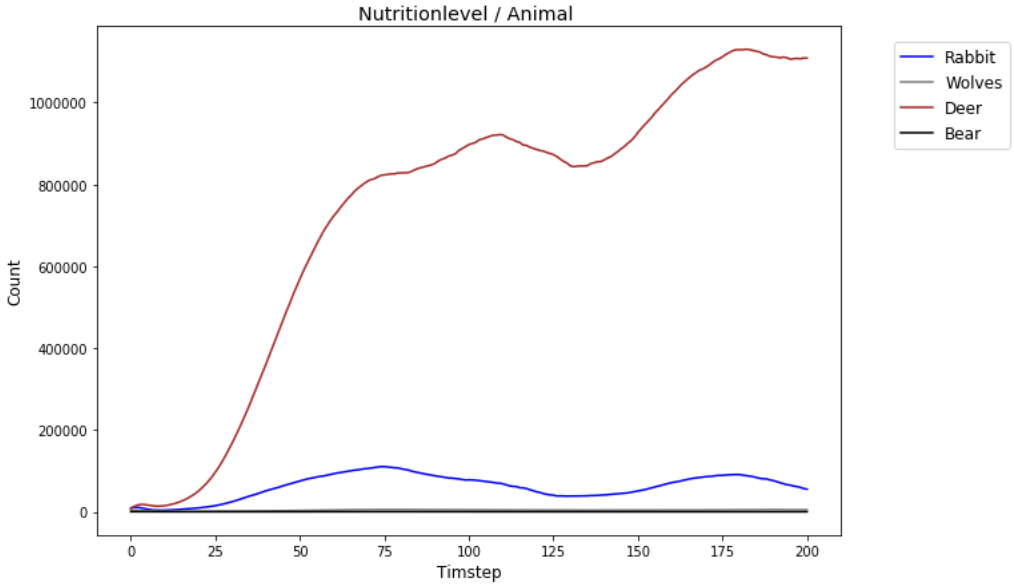
```
plt.plot(x,B_r,'black',linewidth=width,label='Bear'
)
plt.xlabel('Timstep',fontsize=font)
plt.ylabel('Count of reproduction ',fontsize=font)
plt.title('Reproduction count',fontsize=fonttitle)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2,
borderaxespad=1.,fontsize=font)
plt.show()
```



In [35]:

```
#NUTRITION LEVEL / ANIMAL PLOT (If results are not
too clear, view prey and predator separately)
plt.figure(figsize=size)
plt.plot(x,R_N_P,'blue',linewidth=width,
label='Rabbit')
plt.plot(x,W_N_P,'grey',linewidth=width,
label='Wolves')
plt.plot(x,D_N_P,'brown',linewidth=width,label='Dee
r')
plt.plot(x,B_N_P,'black',linewidth=width,label='Bea
r')
plt.xlabel('Timstep',fontsize=font)
plt.ylabel('Count',fontsize=font)
plt.title('Nutritionlevel / Animal
',fontsize=fonttitle)
```

```
plt.legend(bbox_to_anchor=(1.05, 1), loc=2,
borderaxespad=1.,fontsize=font)
plt.show()
```



Create Data set and export it¶

In [36]:

```
#Creat list with all important imformation of a
type_cell
Rabbit=pd.concat([R_t,R_L_P,R_N_P,R_r], axis=1)
Rabbit.columns=list(["Count","Lifespan","Nutrition"
,"Reproduction"])

Wolves=pd.concat([W_t,W_L_P,W_N_P,W_r], axis=1)
Wolves.columns=list(["Count","Lifespan","Nutrition"
,"Reproduction"])

Deer=pd.concat([D_t,D_L_P,D_N_P,D_r], axis=1)
```

```
Deer.columns=list(["Count","Lifespan","Nutrition","
Reproduction"])

Bear=pd.concat([B_t,B_L_P,B_N_P,B_r], axis=1)
Bear.columns=list(["Count","Lifespan","Nutrition","
Reproduction"])
```

In [37]:

```
#UNCOMMENT IF RESULTS ARE WANTED, ADJUST PATH

#Rabbit.to_csv(r'/User.../Report_Rabbits.csv')
#Wolves.to_csv(r'/User.../Report_Wolves.csv')
#Deer.to_csv(r'/User.../Report_Deer.csv')
#Bear.to_csv(r'/User.../Report_Bear.csv')
```

In [ ]:

References

[1] https://en.wikipedia.org/wiki/Cellular_automaton, 09.12.2018.
[2] Hiroki Sayama, Introduction to the Modeling and Analysis of Complex Systems. Pages 185-225, August 2015.
[3] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life, 09.12.2018.
[4] https://en.wikipedia.org/wiki/Wa-Tor, 09.12.2018.
[5] Gabriela Kirlinger, Two predators feeding on two prey species: A result on permanence, September 1989.
[6] Vahidin Hadžiabdić, Midhat Mehuljić and Jasmin Bektešević, Lotka-Volterra Model with Two Predators and Their Prey, February 2017.
[7] https://en.wikipedia.org/wiki/Wa-Tor, 09.12.2018.
[8] Patrick Misteli & Ruben Kälin, Circle Of Life, May 2014.
[9] https://en.wikipedia.org/wiki/Lotka–Volterra_equations, 09.12.2018.
[10] https://en.wikipedia.org/wiki/Generalized_Lotka–Volterra_equation, 09.12.2018.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Two Predators and Two Prey Interactions

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Bruun	Alexander
Paris	Anton

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Meggen, 09.12.2018	
	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.