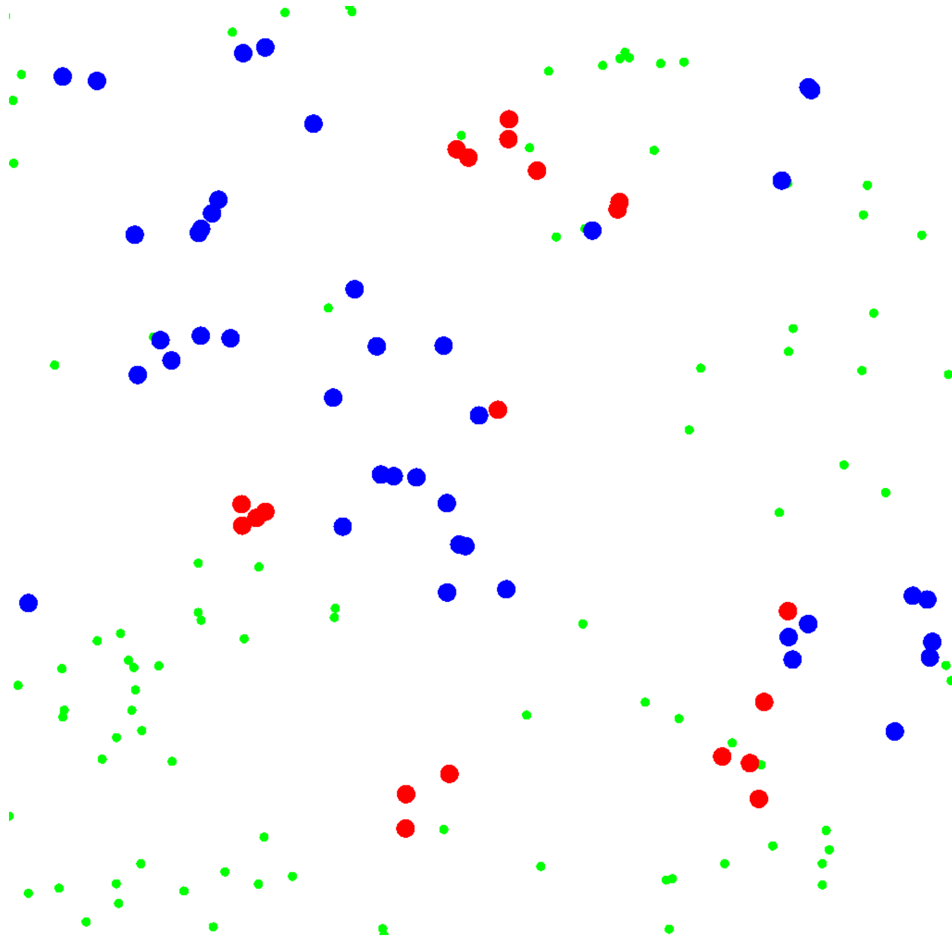
Seminar Report

Predator and Prey Simulation in Python

Leander Feldmann, Jannis Rowold



Tutor: Jack Ogaja

Georg-August-Universität Göttingen
Institute of Computer Science

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Problem Formulation . . . 3
- 1.3 Organization of the report . . . 3
- 1.4 Authors . . . 3

2 Foundation **4**
- 2.1 The predator-prey model . . . 4
- 2.2 Implementation . . . 4
- 2.3 Environment . . . 4
- 2.4 Plants . . . 4
- 2.5 Creatures . . . 4

3 Sequential Implementation **6**
- 3.1 General structure . . . 6
- 3.2 Implementation . . . 6
  - 3.2.1 init_ecosystem() . . . 6
  - 3.2.2 conditions . . . 6
  - 3.2.3 sense() . . . 7
  - 3.2.4 move() . . . 7
  - 3.2.5 eat() . . . 7
  - 3.2.6 reproduce() . . . 7
  - 3.2.7 check_alive() . . . 7
- 3.3 Problems . . . 7

4 Naive Parallel Approach **8**
- 4.1 General structure . . . 8
- 4.2 Implementation . . . 8
  - 4.2.1 init_parallel_computing() . . . 8
  - 4.2.2 send_to_workers() . . . 8
  - 4.2.3 recv_from_workers() . . . 9
  - 4.2.4 recv_from_main() . . . 9
  - 4.2.5 determine_own_creatures() . . . 9
  - 4.2.6 send_to_main() . . . 9
- 4.3 Problems . . . 9

5 Box Approach **10**
- 5.1 Concept . . . 10
- 5.2 Structure . . . 10
- 5.3 Implementation . . . 11
  - 5.3.1 get_boxes() . . . 11
  - 5.3.2 sort_data() . . . 11
  - 5.3.3 determine_own_boxes() . . . 11
  - 5.3.4 determine_neighboring_boxes() . . . 12
- 5.4 Problems . . . 12

6 Recursive Box Approach **13**
- 6.1 Concept . . . 13
- 6.2 Problems . . . 14

7 Better Box 15
7.1 Concept . . . 15
7.2 Structure . . . 15
7.3 Implementation . . . 16
7.3.1 get_sub_boxes() . . . 16
7.3.2 determine_own_sub_boxes() . . . 16
7.3.3 get_sub_box_layers() . . . 16
7.3.4 sense() . . . 16

8 Performance Analysis 17
8.1 Benchmark . . . 17
8.2 Interpretation . . . 18

9 Conclusion 18

10 Appendix 19

1 Introduction

1.1 Motivation

Humans have always been interested in understanding the underlying dynamics of the life on our planet. This knowledge enables us to use those dynamics for our own purposes. Whether we abuse them for our own enrichment or use them to preserve ecosystems that got out of balance, the understanding of those dynamics is the key part. But collecting data on the changes of ecosystems takes a huge amount of time. We also might be interested how different external influences might affect a closed system. If we wanted to actually test this we either risk destroying the system through our tests or we have to carefully rebuild the system on a smaller scope. But to simulate an ecosystem in real life is quiet difficult and complex. If we want to collect an extensive amount of data about the development of an ecosystem in a short amount of time we have to simulate it. This simulation must be able to run much faster than the actual processes it represents. The best way to achieve those goals without risking to cause any real harm is to use computer simulations. Simulating complex systems virtually can still be quiet time consuming. But there are methods to increase the performance of a simulation. A quiet powerful one is parallelization. The goal of this method is to distribute the workload of the simulation from one onto multiple processes that run in parallel.

1.2 Problem Formulation

The goal of this project is to build a parallel framework for predator-prey simulation. We are using the programming language Python 3. To implement parallel processing we are using MPI.
We are not focused on building a realistic or detailed simulation. We want to build a parallel framework around a simple predator-prey simulation that can later be used to efficiently run more complex simulations.

1.3 Organization of the report

At first we introduce the concept of a predator-prey model. Then we discuss our different approaches to implement the model. We start with a simple sequential Implementation. The next approach parallelizes the sequential Implementation. Afterwards we try to further refine the model. This leads to the Box Approach and the recursive Box Approach. In the end we discuss our final implementation which is tailored to fit our computational possibilities. This last implementation also abuses the simplicity of our predator-prey model. Afterwards we benchmark and compare the different approaches and implementations. In the end we summarize our results and offer an outlook onto further ways of improvement.

1.4 Authors

This project is carried out by Jannis Rowold and Leander Feldmann. Both study Mathematical Data Science B.Sc. at the Georg-August-Universität Göttingen. Jannis Rowold mainly worked on the structure and refinement of the Code. He developed and implemented a time efficient visualization method using PyOpenGL [2]. He also contributed the benchmarking. Leander Feldmann developed the different approaches and contributed first implementations of them. He also wrote the report.

2 Foundation

At first we give a quick introduction into the predator-prey model.

2.1 The predator-prey model

The predator-prey model simulates an ecosystem with plants and two kinds of species. One is a herbivorous species that feeds on the plants. The other one is predatory and feeds on the herbivores. Therefore we call the first species the prey and the second one the predator. Simulating this model can provide information on the dynamics of the development of the populations over time. The model often includes evolution of some traits of both species. This can provide inside into changes of the species over time.[3]

2.2 Implementation

As stated earlier we want to have a simple model.

2.3 Environment

Our environment is a square plane. The plane is defined by it's limits. Since it's a square the x and y limits have to be the same. Objects can be located at every point (x_0,y_0) with x_0 and y_0 floats between the limits.
We do not want creatures to be able to leave the plane. We also do not want the plane to have strict borders. Therefore we implemented it in a way that the opposing edges of the plane are connected with each other. If a creature leaves the plane on one side it immediately reenters it from the opposite one. This keeps the restriction of the plane to be finite without any restrictions in the movement of the creatures.

2.4 Plants

At the start of each run a selected amount of plant spawn points gets drawn randomly. Plants spawn on random locations near a random spawn point. The maximum number of plants is fixed and set before the run. Through the number of plants and spawn points we can influence if plants spawn evenly distributed across the whole plane or if they should cluster at certain points. If a plant gets eaten a new plant respawns a set amount of iterations later.

2.5 Creatures

Both predator and prey share the same general life cycle. A creatures most important feature is its energy. If a creature runs out of energy it dies. If a creatures energy surpasses a certain threshold the creature reproduces. Energy is gained through eating. Energy is lost through moving around. Even if a creature does not move it loses a small amount of energy each iteration. Reproduction also costs energy.
Every creature has distinct traits. Every creature has a distinct sensing radius. A creature is able to sense objects (creatures and plants) that are at most as far away from it as its sensing radius. Therefore a creature can sense 360° around and not just objects that are at a certain angle to it. If a creature is near an edge of the plane and its sensing radius is big enough it can sense objects on the opposing side. Every creature has a distinct speed value. This value determines how far a creature can move on average each iteration.
The traits of the first generation of creatures get initialized with random values around a given expected value. If a creature reproduces its offspring's traits are drawn randomly with the parents traits as expected value. The species evolve on those traits. Higher speed and better sensing comes with a price. The energy cost of each movement consists of a fixed amount each iteration that correlates with the creatures sensing distance and a variable amount that is proportional to the distance the creature traveled. Further sensing and higher speed lead to a higher energy consumption.
At the start of each run a set amount of prey spawn points get drawn randomly. A selected amount of prey spawns in at random locations around those spawn points. As with the food the selection of

the parameters lead to wider spreads or closer clustering of prey in the beginning. A selected amount of predator spawn on random locations on the plane. If a new creature is born it spawns at a random location around its parent.
The life cycle of prey can be summarized by:

- sense your surrounding
- move to plants and avoid getting eaten
- if able eat
- if able reproduce
- repeat until energy runs out or eaten

The one of predator by:

- sense your surrounding
- move to prey
- if able eat
- if able reproduce
- repeat until energy runs out

Now we want to implement this concept in python. At first we start with a sequential implementation.

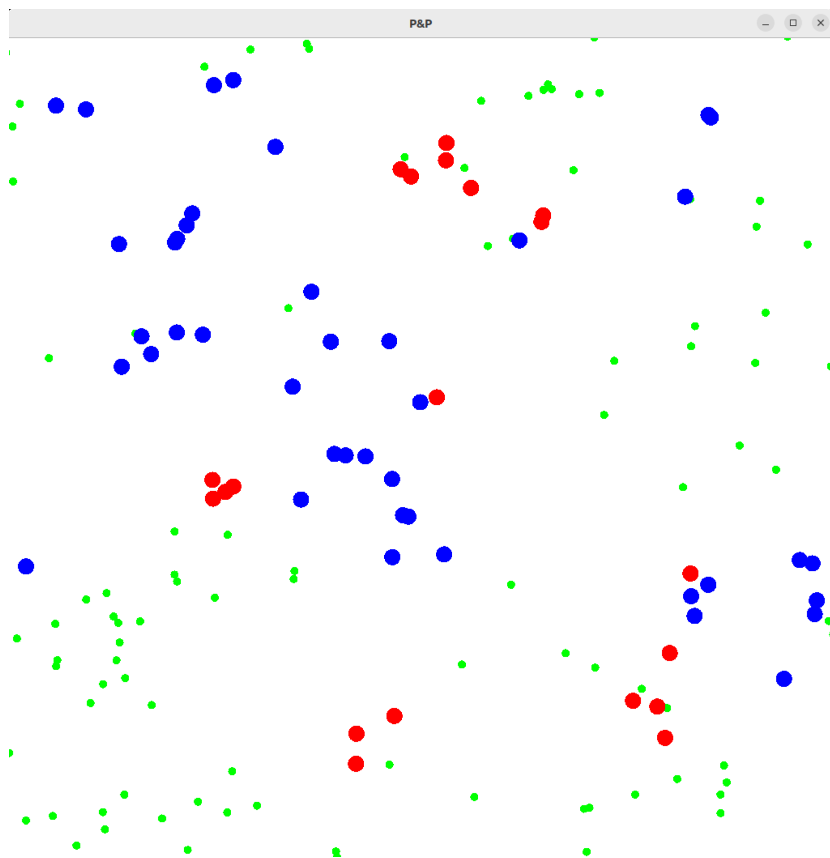


Figure 1: Visualization of our simulation (Red: Predator, Green: Plants, Blue: Prey)

3 Sequential Implementation

3.1 General structure

To simulate a dynamic ecosystem with multiple individually different and autonomous instance we need to define some underlying structures.
Since the main actions every creature has to take repeat themselves constantly its a quiet natural way to implement the ecosystem as a loop that repeatedly simulates those actions. This loop should run until either a set stop criterion is reached or the ecosystem has come to a standstill.
The main goal of every creature is to survive for as long as possible. To accomplish that, a creature has to avoid every circumstance that could lead to its death. Those circumstances are the complete loss of energy through starvation and for prey the danger of getting eaten by a predator. To avoid Starvation and getting eaten a creature has to move around. To make sure the creature moves to a beneficial position it has to sense its surroundings first. If a prey is near enough to a plant or a predator to a prey they need to be able to eat it. If a creature has enough energy it also needs to be able to reproduce If a creature was not able to avoid starvation or getting eaten it has to die and be removed
So we need a main loop that in some way includes sensing, moving, eating, reproducing and dying. Before we enter the main loop the ecosystem has to be initialized.
There are different approaches to structure these functions in the main loop. You could for example define a model in which a creature can only move or eat or reproduce every iteration. This would be possibly more realistic, depending on the kind of creatures we want to simulate.
We chose the simplest approach. This means our creatures are able to do all of those things in one iteration.
This leads to the following structure:

```
init_ecosytsem()
while conditions:
    sense()
    move()
    eat()
    check_alive()
    reproduce()
```

3.2 Implementation

3.2.1 init_ecosystem()

This section of the program creates the plane and spawns a set number of prey, predator and plants. The prey spawn randomly randomly around a set of random spawn points. The number of those is set as a hyperparameter. The initial values of the sensing and speed values are chosen randomly around a set expected value. Every creature starts with the same amount of energy. The different values are stored in separate lists. Every creature is unambiguously identified through its species and its index. Every value of a creature is stored at the same index. The predator spawn randomly anywhere on the plane. Their sense, speed and energy values are chosen in the same way as those of the prey. The plants also spawn randomly around a set of random spawn points. The number of those is also set as an hyperparameter.

3.2.2 conditions

The simulation should run until either a set maximum amount of iterations is reached or if one species becomes extinct. Those conditions ensure that the simulation always comes to an end and can not run indefinitely. Since predator-prey simulations are interested in the coexistence of those species we also want to stop as soon as this coexistence ends. If the prey dies out the predator are unable to find food and are going to become extinct either way. The prey could live without a predator population but since that is not the scope of such a model we are not interested in this circumstance.

3.2.3 sense()

The sense function should return everything that is in the sensing distance of a given creature. We use the simplest approach. For every creature we calculate the distances to all the other creatures and every plant to get the ones which are inside its sensing radius.

3.2.4 move()

With all the information the creature gathers it decides where to move. At first we determine how far the creature is able to move this iteration. This distance is chosen randomly with the creatures speed value as its expected value. Afterwards the creature determines in which direction it wants to move. We use a really basic decision making process. If a prey is sensing plants it will always move towards the nearest one. If a prey does not sense any plants but predator it will move away from the nearest predator. If a prey only senses other prey it will again move away from the nearest other prey. If a predator senses prey it will always move towards the one closest by. If it senses other predator instead it will move away from them. If a predator only senses plants it will move towards the nearest one. The decision making could be further refined and also be made a subject of evolution. But since our main focus is in the performance of our model and not in the actual behavior of our creatures we use this simple logic. If a creature wants to move towards an object and the distance that the creature can travel is greater than the distance to the object it will only move to the position of the object. Otherwise it will always move the whole chosen distance into its chosen direction. After the new position is calculated it needs to be checked if the creature crossed a border of the plane. If it did so it's new coordinates get replaced with it's coordinates modulo the length of the plane. This ensures that the creatures can not escape our plane.
If the movement is finished the creature looses energy. The energy lost is proportionate to the moved distance and therefore roughly to the speed value but also to the sense value of the creature. Afterwards the move function returns the updated positions and energy values of all the creatures.

3.2.5 eat()

The eat function consists of two parts. At first needs to be checked who is able to eat. Only than can those creatures actually eat. Since the positions of all our objects are represented through tuples of floats it would be insufficient to check, if a prey and a plant for example are on the same exact position to decide if the prey could eat or not. Therefore we have to define radii in which food has to be located to be eaten. Now we just have to check if the distance between a creature and its potential food is smaller than the corresponding eating radius. If this condition is satisfied the creature eats its food. If a prey eats a plant it gains a fixed amount of energy. The plant gets deleted. If a predator eats a prey it gains a fixed amount of energy plus a varying amount proportional to the energy of the prey. The energy of the prey gets set to 0 afterwards.

3.2.6 reproduce()

Every creature with its energy level above a certain threshold gets to reproduce. For every of those eligible creatures a new creature of the same kind spawns on a random location in near proximity to its parent. The offspring's speed and sense values are chosen randomly with their parents ones as expected values. All new creatures start out with one fixed energy level. The reproduction costs the parents a fixed amount of energy.

3.2.7 check_alive()

To round out the iteration every creature gets checked on if their energy level is still greater than zero. Every creatures whose energy has dropped to zero gets deleted.

3.3 Problems

This implementation of our model is really slow. Especially the sensing section of the program takes very long and makes up more than 99.9% of the entire runtime. If we want to further improve the performance of our simulation the sensing section should be our main focus.

4 Naive Parallel Approach

In the sequential approach all the creatures are processed one after another. As we have seen previously this takes up quiet some time. To improve the performance of our simulation we want to make use of parallel computing.

4.1 General structure

We have to determine which parts of our code should be parallelized and which should not.
The most important part to parallelize is the sensing function. Since this function has to be computed for every individual creature and only relies on given data, it can and should be parallelized. Since the move function relies solely on the results of the sensing function it can also be parallelized. The eating section on the other hand relies at least for the predator not just on the movement of the predator itself but also on the movement of prey. So we need to make sure, that the result of the movement of every prey is available to every predator. We also need to prevent two creatures eating the same plant or prey. So we also need the information about what has already been eaten and what has not. If we would split the eating function onto different cores and tried to run it in parallel those cores would constantly need to exchange those pieces of information. This alone could take up even more time than a sequential implementation. Therefore we keep our sequential implementation of the eating section. So we need to recollect the results of the movement before we can calculate the eating functions. The reproduction and check_alive functions could be parallelized. But since the data has to be recollected on the main process before the functions are getting called and they only take up about 0.0003% and 0.00007% of the runtime in the sequential approach respectively it would not boost the performance noticeably. In some cases it would even take more time to send and recollect the data to and from the other cores than it would take to compute the functions sequentially. Therefore we compute those functions on the main process as well.
This gives us the following structure.

```
init_parallel_computing()
init_ecosystem()
main process:
    while condition:
        send_to_workers()
        recv_from_workers()
        eat()
        check_alive()
        reproduce()

worker processes:
    recv_from_main()
    determine_own_creatures()
    sense(own_creatures)
    move(own_creatures)
    send_to_main()
```

4.2 Implementation

4.2.1 init_parallel_computing()

We are using MPI to manage the communication between the different processes. Since we are working in Python 3 we are using the Python Module mpi4py [1] which provides an extensive implementation of MPI Applications in Python. To enable this communication a communicator needs to be initialized.

4.2.2 send_to_workers()

Since the workers should compute sense() and move() they need all the information that those functions need. Those are the information about the locations of every creature, the sensing distances, speed and energy levels of every creature and the locations of the plants. So all of this data needs to get send

to the workers. We are using point-to-point communication between the main process and the worker processes. Since all worker processes get all the data we could also use collective communication.

4.2.3 recv_from_workers()

The updated positions and energy levels of the creatures that were processed by the individual cores get recollected in the main process. We additionally get a list from every worker that contains the food objects (plants or preys respectively) that are the nearest to the creatures to check if they can be eaten. After receiving the data from the workers we want to sort them into their original order. This is necessary because we identify our objects via their indices in their storage structures. If those indices change we have no definite way to distinguish creatures. The creatures were allocated to the worker processes in coherent subsets of the original lists of creatures. Those subsets were allocated according to the rank of the process. The worker processes did not alter the order of their creatures. So to maintain the original order we just need to string the lists returned by the worker processes together in order of their rank. This ensures the required structure of our data.

4.2.4 recv_from_main()

The worker processes receive all the information about the creatures and the plants from the main process.

4.2.5 determine_own_creatures()

The main idea of parallelization is that multiple processes process a smaller part of the task at the same time. Therefore we need to distribute our creatures onto the different processes. To minimize the runtime over all the processes we have to distribute the creatures as evenly as possible. Because of that every core gets the same amount of prey and the same amount of predator. To ensure that every process chooses a different set of creatures we allocate parts of the creatures based on the rank of the process. Assume we have n worker processes and the main process has rank 0. The process with rank 1 is allocated the first n-th of the prey and the first n-th of the predator. The process with rank 2 the second n-th and so on. It is not practical to define these allocations in the main process and send every process just its creatures since the sensing function still computes the distances to all of the other objects. Therefore every process still needs all the information.

4.2.6 send_to_main()

After the new movement is calculated the information needs get get passed back to the main process. We do not have to send all the data back just the parts that were altered. Those are the positions and energy levels of the own creatures (the ones whose movement was calculated) of the process. We also send back the information about witch eatable object is nearest to every creature to later check if it can be eaten right now. Since we want to bring the creatures back into their original order after we recollect them we also return a list of the original indices of our creatures back to the main process.

4.3 Problems

This Approach offers a huge performance improvement. But it is still far from perfect. Although the sensing function needs now less time to be computed it is still quiet slow. Even if a creature has nothing inside of its sensing radius we still compute all distances to every other object. Those unnecessary computations take up a lot of time. To further improve our simulation we have to minimize those calculations.

5 Box Approach

5.1 Concept

The idea behind this approach is to pre-filter which objects are definitely outside of a creatures sensing radius. This prevents us from calculating distances to those objects. To achieve this we divide the plane into square boxes. The sides of those boxes are equal or greater in length than the highest sensing radius any creature has. This ensures that everything that a creature can sense is definitely inside its own or one of the bordering boxes. Therefore we do not have to calculate the distances to the objects in the other boxes. Since parallel computing offers a tremendous speed up we also want to parallelize this approach. In the earlier approaches we allocated the creatures based on their indices in their storage structure which corresponds with their age. Now wont to allocate them boxwise.

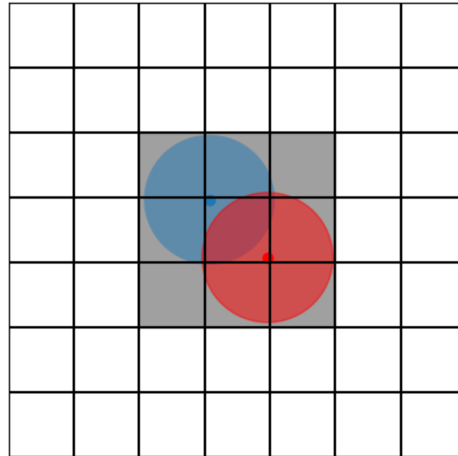


Figure 2: The field of view of two creatures (blue, red) inside one box and the approximated field of view for the box (gray)

5.2 Structure

Since this approach in general only changes the inputs of the sensing function the overall structure stays roughly the same as in the naive parallel approach. There are only a few changes that have to be made. We need to add a section that initializes and updates the boxes. This should happen on the main process before the data gets send to the workers. The structure of the workers also stays more or less the same. The main change here is the way the different creatures get allocated to the processes. Since we want to allocate the creatures based on their location and therefore their box every process gets distinct boxes. The processes should compute the sense and movement of their the creatures inside those boxes. Because the sensing radius of the creatures inside the boxes expands into the neighboring boxes we need to determine those. Then we calculate the distances to every object inside the allocated and its neighboring boxes. Afterwards the movement gets calculated the updated values get send back to the main process as before. After recollecting the data we ant to bring them into their original order. Since the allocations to the worker processes were independent from the indices of the creatures we can not just string the creatures of the different worker processes together. We have to draw up lists containing all the original indices of predator and prey for every box. Those lists allow us to sort our data after recollecting it.

This leads to the following structure.

```
init_parallel_computing()
init_ecosystem()
main process:
    while condition:
        get_boxes()
        send_to_workers()
        recv_from_workers()
        sort_data()
        eat()
        check_alive()
        reproduce()

worker processes:
    recv_from_main()
    determine_own_boxes()
    determine_neighboring_boxes(own_boxes)
    sense(own_boxes, neighboring_boxes)
    move(own_boxes)
    send_to_main()
```

5.3 Implementation

5.3.1 get_boxes()

To create the boxes we have to start by computing how many boxes we have to create. This depends on the maximum sensing radius any creature has. We want to divide the whole plane into boxes of the same size. So the size of the plane has to be a multiple of the box size. Therefore we can not just choose the maximum radius as the size of our boxes. Instead we have to find the smallest divisor of the plane size that is bigger than the maximum sensing radius. This is our box size. Therefore we end up with $(\frac{plane_size}{box_size})^2$ boxes. We create lists of the prey, predator and plants in every box.

5.3.2 sort_data()

To restore the original order of the creatures inside their lists we create lists with the original indices of the creatures in every box. We then can use those lists to sort the creatures into their original order.

5.3.3 determine_own_boxes()

We distribute the boxes evenly over the worker processes. Therefor we use the same principle as with the creatures before. So the process with rank 1 gets the first few boxes, the one with rank 2 the next ones and so on. How many boxes every process gets depends on the amount of boxes and cores.

5.3.4 determine_neighboring_boxes()

The boxes are stored in a list and every box is unambiguously identified by its index. The boxes are indexed as shown in the figure below.

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Figure 3: Environment is split

We can calculate the indices of the neighboring boxes, given the total number of boxes and therefore the number of boxes in each row and column.

5.4 Problems

This Approach has three main problems. As long as the creatures are spread out evenly across the boxes it works quiet will. But as soon as many creatures gather in individual boxes the workload of the different worker processes gets out of balance. Since we have to wait until every process is finished before we can continue the iteration this can increase the runtime dramatically. Another problem arises if we have more cores at our disposal than boxes in our plane. Since every box gets allocated to just one core we can not use more cores than we have boxes. The third thing that can be further improved is that even though this approach decreases the amount of unnecessarily calculated distances it does not extinguish them. Even in the best case scenario the actual sensing radius makes up only about a third of the area we take objects from to calculate their distance.
To improve the Box Approach in all of those areas we need to expand its concept.

6 Recursive Box Approach

6.1 Concept

The third problem of the Box Approach is very similar to the one that lead to Box Approach in the first place. We want to minimize the unnecessary calculations of distances to objects out of a creatures sensing distance. And since we have a similar problem we try to append a similar solution.
We can divide the boxes from our previous approach into smaller smaller subboxes. We have to choose how many sub boxes we want to use and the obvious solutions are to choose a square number. This leads to fitting a set amount of square subboxes into our boxes.
But what advantages do those subboxes provide?
The first advantage is that we can further decrease the area in which we try to sense objects. If we subdivide the boxes into $n \cdot n$ subboxes and a creature is inside a certain subbox we only have to check the subboxes that are at most n subboxes off into any direction.

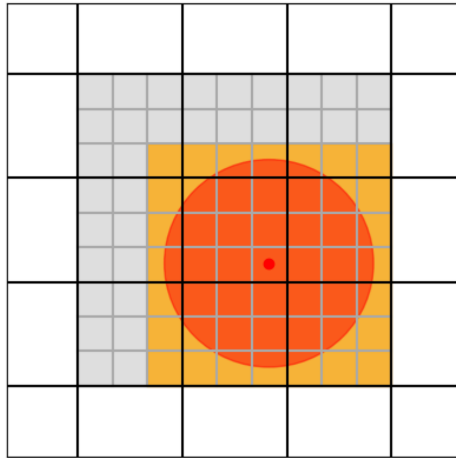


Figure 4: A creature, its field of view (FOV) (circle) and the approximated FOV in the Box Approach (gray) and in the recursive Box Approach (orange, recursion depth one)

We still have some area in our sensing box that is outside of our sensing radius. But this area is only around 65% of the surplus area in the Box Approach.
Another problem of the Box approach was the possible imbalance between the amounts of creatures in the boxes. To avoid this problem we can redistribute all the sub box with at least one creature inside it to a new worker process.

We assume for the moment that we have an infinite amount of cores. Our main process starts by dividing the plane into Boxes as in the normal Box Approach. Those Boxes are called Boxes I. Those Boxes get send to the Worker processes on level I. The workers I check if the Boxes I have more than one creature inside. If not, they calculate the sensing and movement of the creature as long as there is one. If there are more than one creature in the Box I the Box gets split in Subboxes, Boxes II. Every Box II that has at least one creature inside gets send to a worker process of level II. The workers II repeat the same process as the workers I and send every non empty Box III to workers of level III. This cycle repeats until every creature is in a Box on its own and its sensing and movement gets calculated.

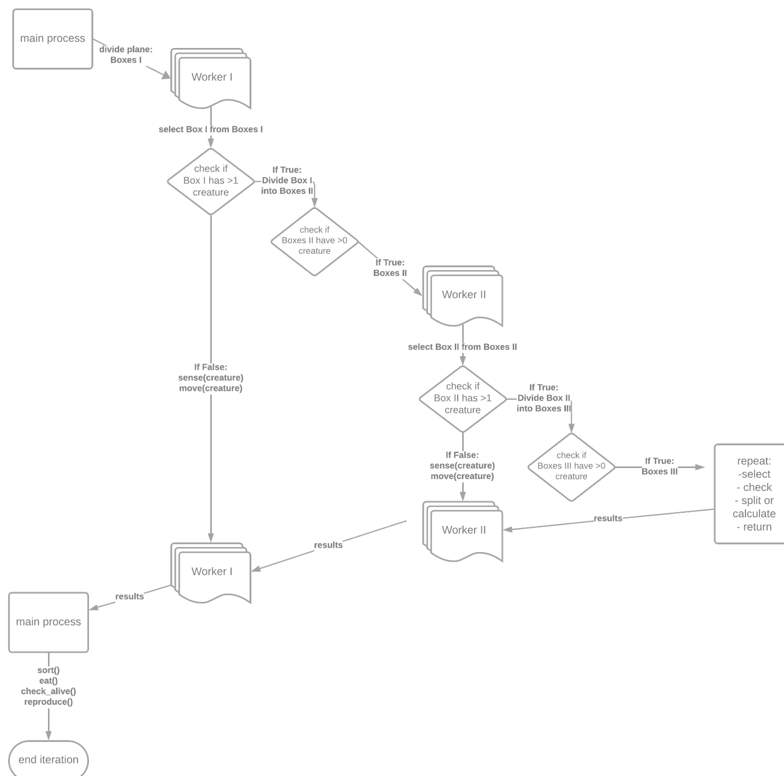


Figure 5: Workflow of process

This process ensures that no worker process has to calculate the sensing and movement of more than one creature. The creatures are perfectly distributed and the raw calculation time of those functions is minimized.

6.2 Problems

This approach is not practical. If every creature should be processed on its own core some cores are just further distributing creatures onto other cores we would need way more cores than creatures. This is technically not feasible since especially for larger models with many creatures there just wont be enough cores available. Another problem is that even though the calculation time of the sense and move functions is minimized the approach will not be as fast as possible. The many box divisions and the needed communication between the different cores would add up and get noticeable.
There are possible workarounds though. If we have a certain amount of cores and now how many creatures exist at the start of an iteration we can calculate how many creatures every core would have to process if the creatures were spread evenly. Now we could check how many creatures are inside of the Boxes I. If there are at least double the amount of creatures that every core should process we can split the Box I into Boxes II. Then we can check if there are more than double of the optimal amount in any Box II. If so we split this Box again. In this way we group Boxes together that have less than double the optimal amount of creatures inside. We can then send those Boxes to the workers.
But there are still other ways to gain even more performance.

7 Better Box

7.1 Concept

Since our movement function is quiet simple we do not actually need all the information a creature senses. Our creatures are only interested if they sense prey, predator or plants and if so where the nearest ones are located. We can use this in combination with the Subboxes of the recursive Box approach to minimize the amount of distance calculations in the sensing function. We can divide our plane and our resulting boxes as often as we want. In the end every creature is located inside one Box of the finest layer. And we now which boxes of this layer approximate the sensing radius of the creature. Until now we have just calculated the distances to every creature inside those boxes surrounding our box without caring about their order. Now we start to just calculate the distances of all creatures in the same box as our creature. As we find the first object we store the distance to it and which object it was. If we find another object we compare its distance to the previously found one. If it is nearer we replace the earlier found one with it. Else we do not store the information and go on. If we have found a predator, a prey and a plant, excluding the creature itself we do not have to look further. If we have not found the nearest predator, prey or plant yet we move on to the neighboring boxes and search through them. If we still have not found everything we move on to the next outer layer. We continue this process until we either find one object of every kind or until we have searched all layers of boxes that lie inside the approximated sensing radius.

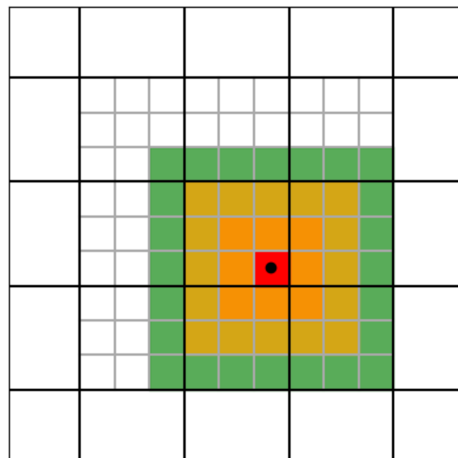


Figure 6: A creature and the different layers surrounding it

If we wanted to further complicate our sensing function but still limit us to the nearest x objects of each kind we can simply expand this approach and do not stop at one.
This approach is not viable if we actually want to use all information a creature senses. This could be the case if we use creatures with Neural-Network brains.

7.2 Structure

To implement this approach we choose to use two layers of boxes. This enables the option of pre-filtering the distances of objects inside the sensing radius of a creature. Using more layers would lead to a higher computation cost since we need to compile much more boxes. In which circumstances another layer of boxes could be justified would need to be tested. We chose the second layer to divide the Boxes I into 9 subboxes. This was the amount with the best performance on average. Those Boxes and Subboxes need to be initialized and updated at the start of every iteration. Otherwise we kept the main structure of the Box Approach.

```
init_parallel_computing()
init_ecosystem()
main process:
    while condition:
        get_boxes()
        get_sub_boxes()
        send_to_workers()
        recv_from_workers()
        sort_data()
        eat()
        check_alive()
        reproduce()

worker processes:
    recv_from_main()
    determine_own_boxes()
    determine_own_sub_boxes(own_boxes)
    determine_neighboring_boxes(own_boxes)
    get_sub_box_layers(own_subboxes)
    sense(own_sub_boxes, neighboring_sub_boxes)
    move(own_sub_boxes)
    send_to_main()
```

7.3 Implementation

7.3.1 get_sub_boxes()

This function calculates the subboxes given the number of boxes and subboxes per box. It allocates the objects based on their location onto the corresponding subboxes.

7.3.2 determine_own_sub_boxes()

The subboxes are defined through their index. Through the index and the number of boxes and subboxes and the size of the plane the exact position of every subbox can be calculated and vice versa. Therefore we can calculate the indices of the subboxes that are inside of a box just given its index and the general knowledge of the chosen box and subbox sizes.

7.3.3 get_sub_box_layers()

This function is given a subbox index and calculates the indices of the subboxes in the surrounding layers. Since we chose every box to contain nine subboxes we need to go three sub box layers far to cover the whole sensing radius of every creature in the subbox. The first layer is the box itself. the second layer consist of all directly neighboring subboxes. The third layer consists of all subboxes that are exactly 2 steps away. The forth layer consists of subboxes that are 3 steps away. The function returns a list containing the indices of all the layers

7.3.4 sense()

Since we do not want every information any more we need to change the sensing distance. For every creature inside a subbox that is inside a box that is assigned to the current worker process we want to find the nearest objects. Therefore we start to search in the first layer, the box itself. We calculate all distances to the objects inside and store the minimum. If every kind of object, predator, prey, and plant is found we break the sensing process for this creature. Otherwise we continue with the next layers. If we have not found a prey, predator or plant we mark this witch a special value. In the end we have the information if there is a nearest predator, prey and plant inside the sensing radius of every creature and if so where it is.

8 Performance Analysis

Now we want to compare our different approaches. To do so we are testing the performance of the different approaches on different amounts of cores. We are testing on 2 to 20 cores.
But first we have to take care of a huge problem. Since our simulation relies on random chance every run is different. That makes it impractical to directly compare the duration of different runs with each other. We could try to fix certain seeds for the random events. But when we use a different amount of cores and the creatures get distributed onto the worker processes in a different way the simulation still progresses differently. Therefore our best chance is to run the different approaches multiple times on the same amount of cores and use the average performance. This could still distort our results but it is our best solution.

8.1 Benchmark

We benchmarked the different approaches with the set starting conditions of 100 prey, 20 predator and 100 plants and ran the simulation for a maximum of 100 iterations. We calculated the average amount of iterations executed per second.

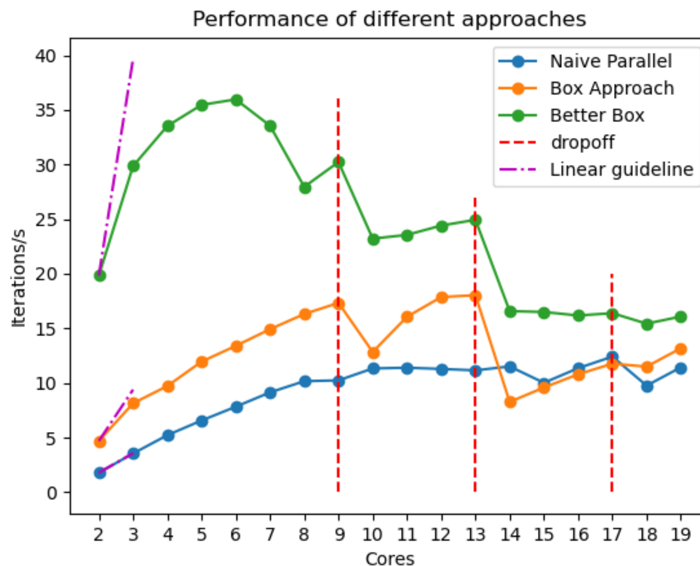


Figure 7: Benchmark

The naive parallel approach was the slowest on average as expected. But it scaled nearly linear for the most part. Compared with the other approaches it scaled the best. On 14 to 17 cores it even outperformed the Box Approach. If we were to use even more cores the trends suggest that it might even outperform the Better Box implementation at some point. The Approach performed the best on 17 cores. The Box Approach started faster than the naive approach but way slower than Better Box. On 2 to 9 cores the Box Approach also scaled nearly linear but had huge drops in performance on 10 and 14 cores. Due to those drops the approach scaled noticeably worse than the naive approach overall. It performed best on 13 cores. Better Box performed way better than the other approaches. In its peak it reached an average of over 35 iterations per second. This is around twice as fast as the Box Approach at its peak. But Better box scales really bad. Even though the performance on 2 to 6 cores increases noticeably it decreases afterwards. The performance on 7 and 8 cores drops even stronger than it had increased previously. On 8 cores it performs even worse than it did on 3. Afterwards the performance has two notable dropoffs going from 9 to 10 and from 13 to 14 cores. In between those dropoffs the performance increases a bit but not as noticeable as previously. On 14 cores and more the Approach performs worse than on 2 cores and also worse than the Box Approach in its peak performance. But Better Box is still the fastest on every tested amount of cores. It performed best on 6 cores.

8.2 Interpretation

The data shows that even though Better Box is the fastest approach in our tests it is not fully optimized for the use on larger amounts of cores. The fact that the performance does not increase for more than 6 cores might come from a bad distribution of the workload onto the cores. If the processing of one Box in Better Box takes way longer than the processing of all the other Boxes it does not matter if we further distribute the other Boxes onto more and more processes. A way to improve this would be to distribute the creatures inside of this Box onto multiple processes. The fact that the performance even goes down shows that the addition of more cores increases the computational cost. This increase is likely to come from tasks as the Communication between the processes and the recollection and sorting of the data the workers generate.
The same statement holds for the Box Approach.
We have no explanation for the huge drops in performance after adding the 10th the 13th and the 17th core. Those dropoffs seem to influence the approaches that distribute the creatures boxwise way more than the naive approach. This could be caused by a technical limitation but since it does not affect every approach in the same way this seems rather unlikely.

9 Conclusion

In general we saw that the use of parallel computing increases the performance of a predator-prey simulation massively. How much it increases depends on its concrete implementation. We developed different approaches with different strength and weaknesses. This lead to a best performing approach that reached an average performance of over 35 iterations/s in its optimal configuration. If we visualize this approach we can guarantee a framerate of over 30 frames per second[4]. This framerate corresponds to the television standard for movies. We can therefore declare our implementation as fast enough to provide the framework for an on average fluent visualization. But this approach is not optimal. This was shown by the really bad scaling if more than 6 cores are used. Aside from that there are also other ways of improving our performance. Additional to parallelization we could precompile the functions we use. This would possibly boost the performance even more than the parallelization already has. Most of the functions in our code would need to be restructured to be able to get precompiled.

10 Appendix

List of Figures

1 Visualization 5
2 Field ov view 10
3 Grid 12
4 Improve Field of view search 13
5 Workflow of process 14
6 Search of better box approach 15
7 Benchmark 17

References

- Mpi4py
- PyOpenGL
- Taleb Obaid: *The Predator-Prey Model Simulation* Basrah, Journal of Science, 2013
- Television framerate