Predator-Prey Simulation

31 Dec 2024

To see version control, please refer to the

GitHub repository URL:

https://github.com/PikaChu030/Preydator-Prey-Simulation

Commit: 08bbb5d

Contents

1 Introduction

Modelling predator-prey dynamics is a foundational computational approach extensively utilized in ecological research to analyse interactions between agents over time. While such simulations have broad applications, they often encounter significant performance challenges, particularly as system scale and complexity grow. Addressing these challenges is crucial to enable real-time processing, ensure scalability, and accurately model intricate ecological interactions.

Performance optimization has long been a central focus in simulation-based research. Techniques such as loop unrolling, Single Instruction Multiple Data vectorization, and algorithmic restructuring have been shown to enhance the efficiency of computational models (Hassaballah, 2008) [1]. Additionally, profiling tools like `cProfile`, combined with optimization libraries, have proven effective in identifying and mitigating performance inefficiencies (Liang Li, 2016) [2].

This project examines a scientific algorithm designed to simulate predator-prey interactions within a two-dimensional landscape, as documented in the Predator-Prey Background Information [3]. Through detailed profiling and systematic code refactoring, the study identifies and resolves performance bottlenecks within the model. By highlighting critical hotspots, this work lays a foundation for further optimization and provides valuable insights into enhancing the efficiency of simulation.

2 Experimental Method

To maintain consistent control over variables, a systematically experimental setup was established that all input parameters were standardized. This approach ensured that performance bottlenecks were accurately identified. Given the inherent variability in simulation outcomes, numerical data collected during the experiments were averaged over ten iterations to enhance robustness and reliability.

The primary objective of hotspot identification was to optimize execution time, particularly for scenarios involving large data inputs or edge cases. Default input parameters were used, with the map configuration exhibiting the longest execution time selected as the baseline model (see Appendix A).

Profiling for execution time was conducted using Python's built-in `cProfile` module, which provided comprehensive metrics on function call frequencies, execution times, and related data. To facilitate result visualization, `SnakeViz` was employed, enabling clear identification of functions with the execution times and performance bottlenecks.

Following the initial profiling, a detailed analysis was performed using `line_profiler` to identify specific lines of code within flagged functions that contributed to increased execution times. Additionally, `memory_profiler` was utilized to evaluate memory consumption patterns, while cyclomatic complexity measurements were conducted to assess the structural intricacies of individual functions.

Insights gained from profiling guided the strategic refactoring of the source code. Refactoring efforts prioritized the optimization of data structures and the application of compilation techniques, such as vectorization and Just-In-Time (JIT) compilation, to mitigate identified bottlenecks. The correctness and integrity of the refactored code were verified through automated testing using `Pytest`.

After the refactoring, all profiling analyses were repeated, and the program's performance was rigorously compared against the baseline model. Particular attention was given to whether alterations in data structures or logic introduced trade-offs between execution time and memory consumption. Tools such as `Memray` was instrumental in examining peak memory usage and memory distribution hotspots.

3 Results

3.1 Execution time

To identify the primary bottleneck in the program's execution, `cprofile` was used to analyse the execution time. A visual representation is provided (see Appendix B), while Table 1 summarizes the cumulative execution time of the key functions.

ncalls(times)	Cumtime(sec)	Percall(sec)	filename:lineno(function)
1	5.949	5.949	.py:306(simulate)
1000	5.349	0.005349	.py:175(update_densities)
100	0.573	0.00573	.py:287(generate_write_maps)

Table 1: Execution time for key functions before code refactoring

Subsequently, `line_profiler` was employed to pinpoint hotspots within specific function. Figure 1 highlights instances of peak execution times, which were traced back to array arithmetic operations in the source code.

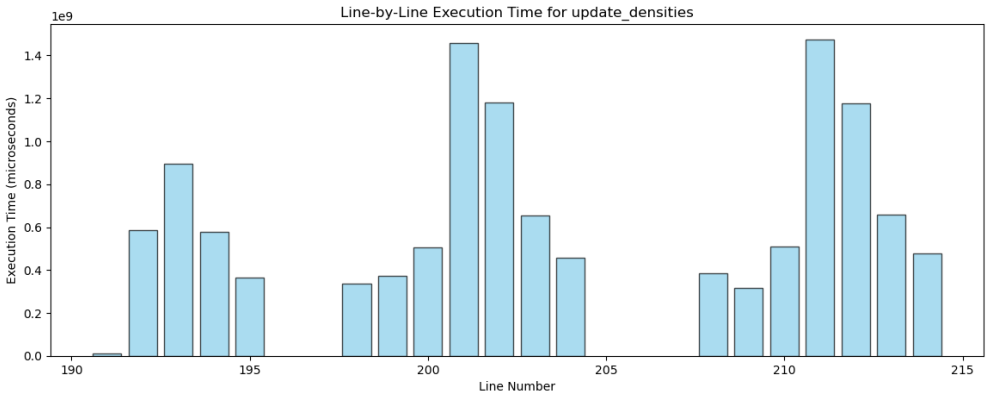

Figure 1: Line-by-line execution time in specific function

After code refactoring, the profiling tools were reapplied, with updated results presented in Table 2. The revised analysis indicates a substantial reduction in the execution time of functions, leading to a marked improvement in overall performance.

Ncalls(times)	Cumtime(sec)	Percall(sec)	filename:lineno(function)
1	1.646	1.646	.py:320(simulate)
1000	0.0089	8.89E-06	.py:176(update_densities)
100	0.3849	0.003849	.py:301(generate_write_maps)

Table 2: Execution time for key functions after code refactoring

3.2 Memory

Although code refactoring can significantly improve computational efficiency, it may also impact memory usage and modify the complexity of the source code. Detailed insights into memory usage over time for specific functions, as well as their allocation status (see Appendix C). The overall memory consumption throughout the simulation is illustrated in Figure 2. The peak memory usage is 113.748 MiB.

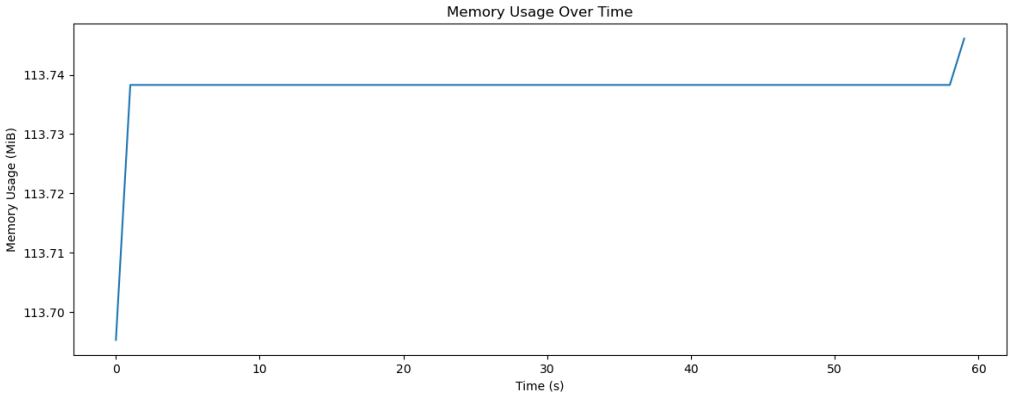

Figure 2: Memory usage over time before code refactoring

After optimizing the identified performance hotspots, the peak memory usage rises to 202.73 MiB, as shown in Figure 3. For a comprehensive analysis of memory allocation and usage patterns during the simulation (see Appendix C).

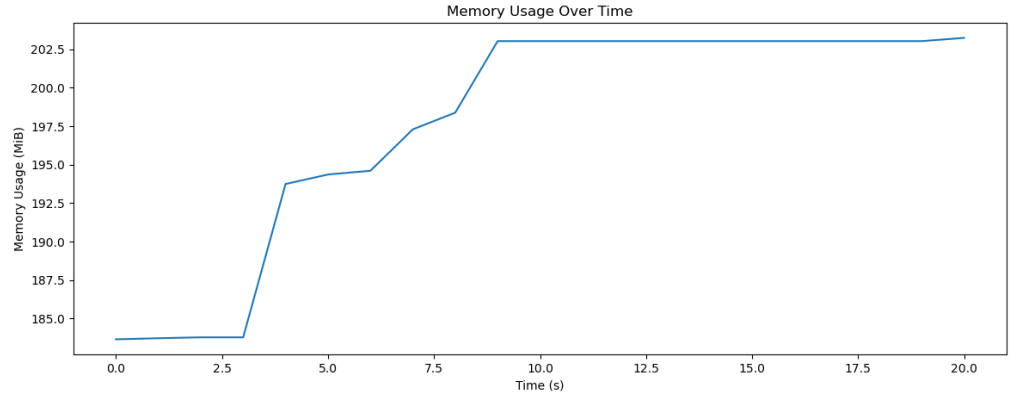

Figure 3: Memory usage over time after code refactoring

While a comprehensive analysis of the cyclomatic complexity of the source code is beyond the main scope of this project, relevant details are provided (see Appendix D).

4 Analysis and Discussion

4.1 Execution time

As shown in Table 1 (Section 3.1), the `simulate` function exhibited the highest execution time, averaging 5.949 seconds. This substantial runtime is attributed to the frequent invocation of the `update_densities` function, which constitutes a significant portion of the total execution time, establishing it as a major performance bottleneck.

A detailed line-by-line timing analysis from `line_profiler` identified the inefficiency as stemming from the use of `NumPy` arrays and their arithmetic operations. After refactoring the code, the average execution time of the `simulate` function decreased to 1.646 seconds, as presented in Table 2 (Section 3.1). All functions utilizing `NumPy` arrays also demonstrated significant reductions in runtime.

To discuss the performance improvements, a comparison of code snippets is provided (see Appendix E). The initial implementation relied on nested loops to compute over each cell, a computationally expensive approach for large arrays due to the high number of iterations. As a result, simulations involving larger datasets, such as `test.dat`, exhibited prolonged runtimes.

In contrast, the refactored implementation utilizes the `convolve2d` function to perform convolution with a predefined kernel, leading to significant efficiency gains. Two primary factors contribute to this improvement. Firstly, the `convolve2d` function is written in C, a lower-level language than Python, which optimizes CPU resource utilization and decreases execution time. Secondly, the convolution algorithm employs efficient data structures to minimize computational overhead.

Additionally, vectorization techniques were combined with JIT compiler, specifically tailored for `NumPy` arrays (see Appendix F). `Numba` translates Python code into machine code at runtime, enabling faster execution specifically suited to the data types and operations used. However, `Numba` compiles code into machine instructions, which limits the feasibility of line-by-line profiling using tools such as `line_profiler`. Consequently, it cannot offer line-by-line analysis for the optimized code.

The integration of optimized data structures and JIT compilation has significantly enhanced program performance. Execution time was reduced while maintaining a computational complexity of O(height*width).

4.2 Memory

The initial implementation, starting at 113.70 MiB and peaking at 113.748 MiB, gives a stable performance throughout the execution. In contrast, the refactored one begins at a higher memory usage of 184.37 MiB. Figure 3 (Section 3.2) shows breakpoints at

different intervals and a stepwise increase, culminating in a peak memory usage of 202.73 MiB—significantly higher than the original code's peak.

To evaluate memory efficiency, the analysis considers execution timelines, including function calls and memory allocation patterns (see Appendices B and C). Both implementations allocate an additional array equal in size to the landscape. Consequently, neither version exhibits a clear advantage in raw memory allocation.

In terms of memory handling efficiency, the original implementation employs nested loops to iterate through each element manually. Conversely, the refactored one utilizes vectorization, enabling the processing of data blocks with improved efficiency. This approach enhances memory performance, demonstrating the refactored code's superior memory handling capabilities.

However, the refactored implementation's higher memory consumption is primarily attributed to the use of `Numba`. Several factors contribute to this increase:

1. **Compilation Overheads**: `Numba` compiles Python functions into machine code at runtime, necessitating the retention of both the original Python code and the compiled machine code in memory. Additionally, intermediate representations generated during compilation further inflate memory usage.
2. **Optimized Data Structures**: `Numba` employs data structures that create temporary buffers for calculations and align data in memory for efficient access. This approach introduces padding bytes and additional memory consumption. By contrast, the default compiler utilizes lists and dictionaries, which lack memory alignment and do not require extra buffers.

The higher starting memory usage in the refactored code can be attributed to these `Numba`-specific factors. Breakpoints observed in the graph likely correspond to buffer allocations and intermediate steps during compilation activities.

Finally, cyclomatic complexity analysis (see Appendix D) reveals a slight improvement in the refactored code, reflecting enhanced maintainability. This supports the conclusion that the refactored implementation not only optimizes performance but also improves code structure, providing a valuable upgrade for computational modeling.

Although techniques such as parallelization, threading, and function caching may also influence memory usage, these aspects are beyond the scope of this project.

4.3 Trade-off

The increased memory usage associated with `Numba`, compared to the default compiler, represents a trade-off for the significant performance gains. Evaluating this trade-off requires consideration of several factors:

1. **Application Requirements**: The choice should align with the specific needs of the application. For use cases demanding rapid execution, such as real-time systems, the refactored version is preferable due to its substantial performance improvements, despite higher memory consumption.
2. **System Resources**: The suitability of the refactored code depends on the available system resources. In environments with ample memory but limited processing power, the refactored implementation offers clear advantages. Conversely, for memory-constrained systems, the original one may be viable.

Given these considerations, the trade-off is justified within the context of this project, which aims to identify performance bottlenecks and optimize execution efficiency. The refactored implementation is especially advantageous for compute-intensive tasks, where minimizing execution time outweighs the additional memory requirements.

5 Conclusions

This project successfully identified performance hotspots within the predator-prey simulation program and implemented optimizations that yielded measurable improvements. By leveraging precision profiling tools and conducting systematic code refactoring, significant advancements in execution efficiency were achieved.

The `simulate` function was identified as a primary performance bottleneck, primarily due to computational inefficiencies arising from repeated loop operations on `NumPy` arrays. Addressing these issues through advanced techniques such as vectorization and Just-In-Time (JIT) compilation led to substantial reductions in execution time while maintaining computational complexity within acceptable parameters.

Moreover, the study examined the trade-offs between execution efficiency and memory usage. While the refactored implementation demonstrated superior memory handling efficiency—attributable to vectorization—it also incurred increased peak memory consumption. This was due to the integration of `Numba`, which enhances execution speed by translating into lower-level code but introduces additional memory overhead through compilation processes and optimized data structures.

In summary, this project contributes to the growing body of research focused on optimizing computational models of predator-prey dynamics. The insights gained here provide a solid foundation for handling larger datasets and more complex arithmetic operations in future studies, advancing the scalability and efficiency of simulations.

5.1 Future work

While this project has achieved success in optimizing simulations, further research is warranted to explore additional avenues for enhancing computational models.

During the analysis, it became evident that parallelization and threading strategies significantly influence program performance. These aspects were recorded during memory profiling (see Appendix C), highlighting their potential impact. Future work should integrate these techniques to leverage multi-core processor capabilities, thereby further reducing execution times without compromising memory efficiency.

Optimization strategies, such as adaptive mesh refinement or the utilization of graphics processing units (GPUs) for parallel computation, present promising opportunities for reducing runtime and memory consumption. Investigating these methods within diverse computational environments would enable the development of models tailored to specific application needs, thereby broadening their practical utility. This will be reserved for future works and researchs.

Finally, this research establishes a foundation for interdisciplinary collaborations, integrating modelling with data science and systems biology. These collaborations have the potential to address the multifaceted challenges inherent in ecological dynamics. By pursuing these lines of inquiry, future work can continue to advance the scalability and applicability of simulation-based ecological research.

6 References

[1] M. Hassaballah (2008). *A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications.* The Computer Journal, Volume 51, Issue 6.

[2] Liang Li (2016). *The Optimization of Memory Access Congestion for MapReduce Applications on Manycore Systems*. The Computer Journal, Volume 59, Issue 3.

[3] *Predator-Prey Background Information*. (referenced 09/2023).

7 Appendices

A. Execution time for each landscape file, shown in Figure A.

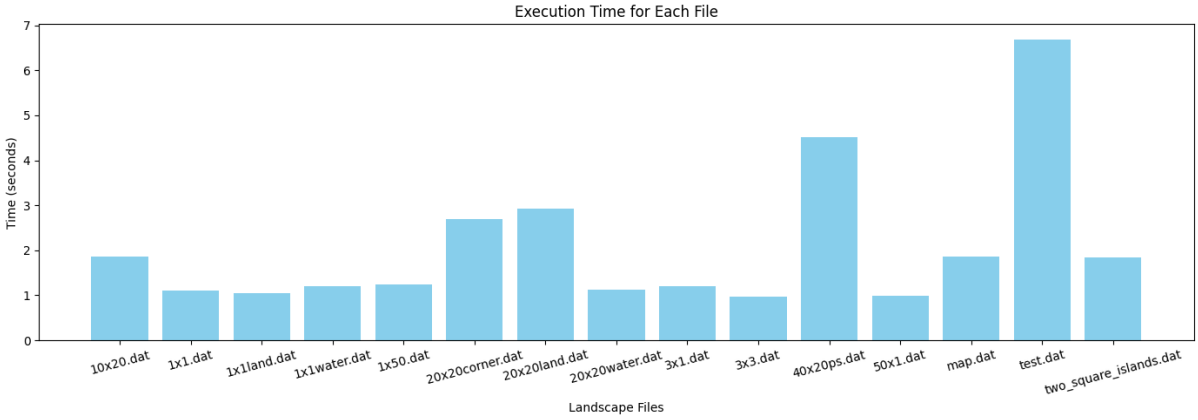

Figure A: Execution time for valid landscape files

B. Visualize execution time for `cprofile` using `SnakeViz`, shown in Figure B.1 and B.2.

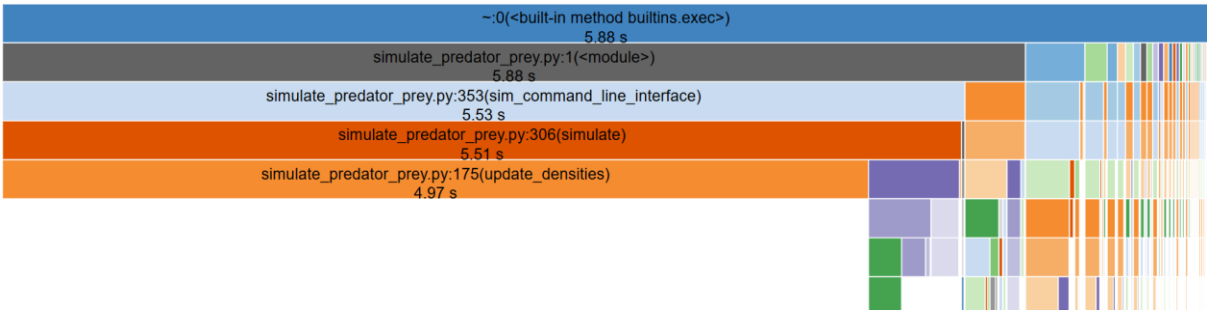

Figure B.1: Execution time for key functions before code refactoring

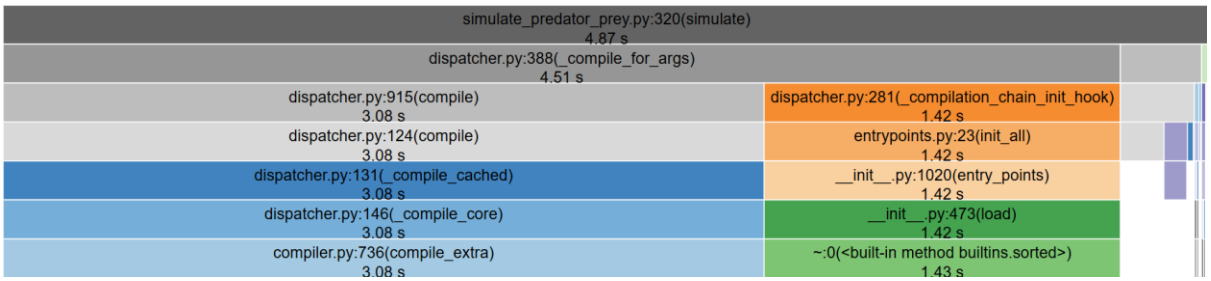

Figure B.2: Execution time for key functions after code refactoring

C. Memory allocation and size for simulation, shown in Table C.1 and C.2.

Thread ID	Size	Allocator	Allocations	Location
0x1	21.1 KiB	malloc	1	initialize_density at ./simulate_predator_prey.py:113
0x1	21.1 KiB	malloc	1	simulate at ./simulate_predator_prey.py:330
0x1	8.0 KiB	malloc	1	write_ppm_file at ./simulate_predator_prey.py:264
0x1	19.5 KiB	calloc	1	generate_color_maps at ./simulate_predator_prey.py:233
0x1	21.1 KiB	calloc	1	read_landscape at ./simulate_predator_prey.py:89

Table C.1: Memory allocation and size for simulation before code refactoring

Thread ID	Size	Allocator	Allocations	Location
0x1	19.5 KiB	malloc	1	generate_color_maps at ./simulate_predator_prey.py:253
0x1	21.1 KiB	malloc	1	initialize_density at ./simulate_predator_prey.py:115
0x1	4.0 MiB	malloc	2	write_ppm_file at ./simulate_predator_prey.py:273
0x1	698	malloc	1	run_simulation

	B			at ./test/profile_simulation.py:94
0x1	21.1 KiB	malloc	1	simulate at ./simulate_predator_prey.py:334
0x1	21.1 KiB	malloc	1	convolve2d at .l/_signaltools.py:1750
0x1	21.1 KiB	calloc	1	read_landscape at ./simulate_predator_prey.py:91

Table C.2: Memory allocation and size for simulation after code refactoring

D. Cyclomatic complexity for each function in source code, shown in Figure D.1 and D.2.

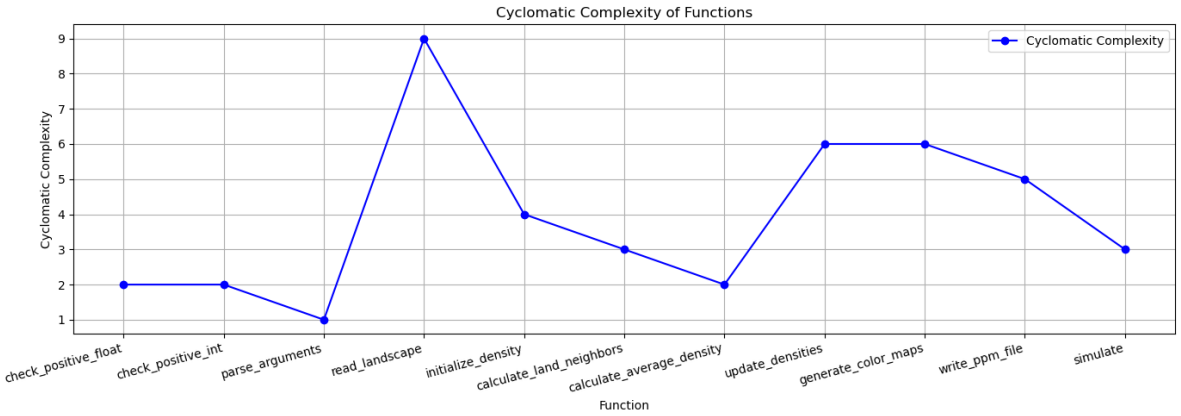

Figure D.1: Cyclomatic complexity before code refactoring

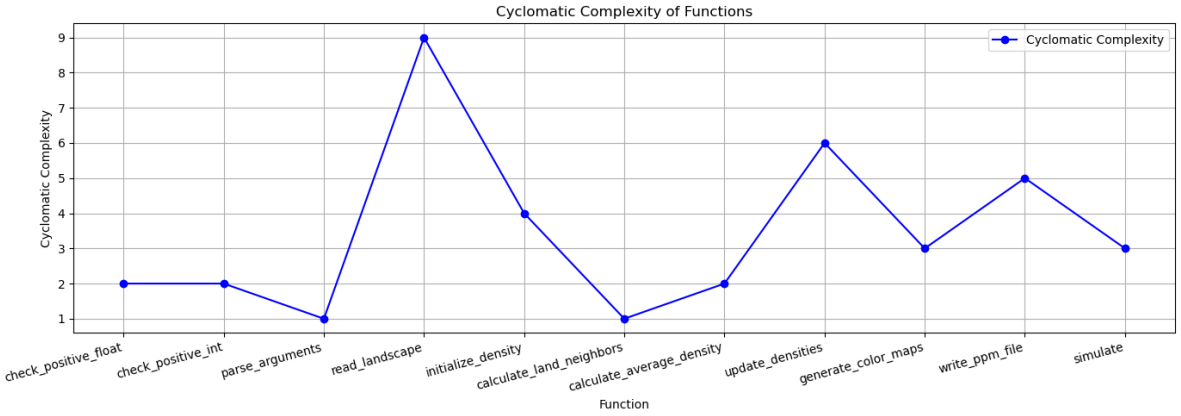

Figure D.2: Cyclomatic complexity after code refactoring

E. Implementation of array for calculating neighbours, shown in Figure E.1 and E.2.

```
num_neighbors = np.zeros_like(landscape)
for x in range(1, height + 1):
    for y in range(1, width + 1):
        num_neighbors[x, y] = (landscape[x-1, y] + landscape[x+1, y] +
                               landscape[x, y-1] + landscape[x, y+1])
return num_neighbors
```

Figure E.1: Implementation before code refactoring

```
kernel = np.array([[0, 1, 0],
                   [1, 0, 1],
                   [0, 1, 0]])
neighbors = convolve2d(landscape, kernel, mode='same', boundary='wrap')
neighbors[0, :] = neighbors[-1, :] = neighbors[:, 0] = neighbors[:, -1] = 0
return neighbors
```

Figure E.2: Implementation after code refactoring

F. Implementation of JIT compilation, shown in Figure F.

```
# Define the BiomeParameters in a way compatible with Numba
@njit
def update_densities(landscape, mouse_density, fox_density, new_mouse_density,
                     new_fox_density, neighbors, mouse_birth, mouse_death, mouse_diffusion,
                     fox_birth, fox_death, fox_diffusion, delta_t, height, width):
```

Figure F: Import JIT flag to compile the specific function