To Lenia and Beyond

Part II NST Physics
Computing Project

April, 2024

Abstract

This computing project undertakes a study of cellular automata, exploring from the foundational Conway's Game of Life to more sophisticated variants such as Primordia, Lenia and the novel Particle Lenia. Simulations of Conway's Game of Life, Primordia and Lenia are conducted on a 2-dimensional lattice, where we investigate various equilibrium patterns influenced by distinct local rules. In Lenia, we also analysed the behaviour of the most well-known creature, Orbium. Orbium displays characteristics of being stable and adaptive to environmental fluctuations. Among hundreds of discovered Lenia creatures, two multi-kernel animals from Astrium family were also implemented and simulated. Additionally, Particle Lenia was implemented on a 2-dimensional canvas, with the lattice setting in previous cellular automata replaced by a group of particles, together with a newly-introduced motion law. We also carry out an energy analysis of Particle Lenia, which shows the tendency for particles to minimise its local energy field. Implementation is carried out in Python, adopting an object-oriented programming approach to ensure robustness and modularity. Various programming techniques are employed to provide high performance simulation, such as complex array manipulations using JAX, as well as simulation visualisation supported by Matplotlib and IPython. (The word count for this report is 2791.)

1 Introduction

Cellular automata (CA) represent a fascinating class of models that simulate the time evolution of systems through discrete interactions within a grid-based structure. The concept was originally discovered in the 1940s by Stanislaw Ulam and John von Neumann [1] and popularised by John Conway's Game of Life [2]. Cellular automata have been extensively used to study phenomena from natural pattern formation to theories of computation and artificial life. In this project, we have explored and implemented a range of CA from simple to complex systems, including explanations of the underlying physics and visualisation of computing simulations.

In Section 2, we begin with the classical cellular automaton: Conway's Game of Life [2], characterised by its discrete space, state, and time setting on a two-dimensional lattice. In Section 3 and Section 4, we explore Primordia and Lenia, as extensions and generalisations to Conway's Game of life [3]. Primordia incorporates continuous states and time but remains spatially discrete, while Lenia extends to continuous dimensions in space, state, and time. Section 5 explores a more advanced variant, Particle Lenia, which substitutes the traditional lattice with a population of particles and adds a motion law to regulate their dynamics [4]. Implementation details, including the coding environment, algorithms, and complexity analysis, are presented in Section 6. The report concludes with a comprehensive summary in the final section.

2 Conway's Game of Life

Conway's Game of Life (GoL) is a prime example of a cellular automaton devised in 1970 by John Horton Conway. The game is played on a two-dimensional lattice with toroidal boundary conditions, and each cell is either live or dead at a certain time. For every time step, the interaction between any cell and its surrounding eight neighbours determines the subsequent state of the central cell. The lattice evolves with an initial state and all the following patterns follow according to four rules:

1. *Underpopulation*: Any live cell with fewer than two live neighbours dies.
2. *Stabilisation*: Any live cell with two or three live neighbours lives on to the next generation.
3. *Reproduction*: Any dead cell with exactly three live neighbours becomes a live cell.
4. *Overpopulation*: Any live cell with more than three live neighbours dies.

We can summarise the above rules mathematically by introducing the *growth function* and the *kernel*. As there are only two cell states for GoL, the live cell state and the dead cell state can be represented by 1 and 0 respectively. In this case, the *growth function* G: $[0,1] \rightarrow [-1,1]$ is a mapping from the current cell state to the increment of the cell state in the next time step. It has three possible outcomes: a value of 1 signifies that the cell will be alive, -1 indicates that the cell will be dead, and 0 means that the cell's state will remain unchanged. As for the *kernel* K, the convolution between kernel K and lattice A is an alternative to summing all eight neighbours as this range will be modified in the following sections. To simplify the description, let us also denote the convolution product be the *potential distribution* U. Therefore, the evolution process for Game of Life and later cellular automata can be expressed as [3]:

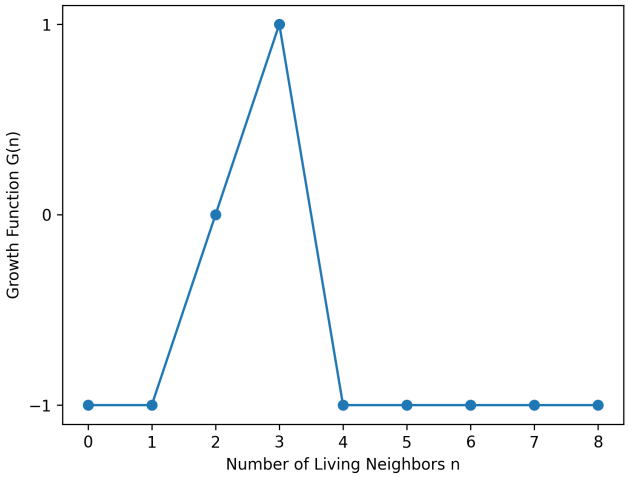
$$U^t(x) = K * A^t(x) \tag{1}$$

$$G^t(x) = G\left(U^t(x)\right) \tag{2}$$

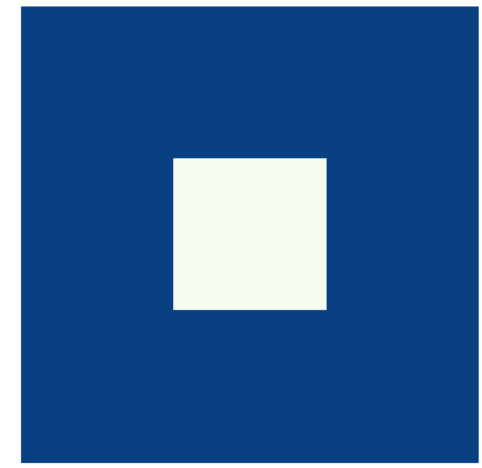
$$A^{t+\Delta t}(x) = \left[A^t(x) + \Delta t G^t(x)\right]_0^1 \tag{3}$$

where x is any site in the lattice, index t denotes the function state at time t and $[f]_0^1$ represents a clip function which limits f to be in the range $[0,1]$. For Conway's Game of Life, the growth function is a step-wise function as shown in Figure 1a and the normalised kernel is just a 3×3 matrix shown in Figure 1b.

The simple rules of the Game of Life enable us to discover intriguing properties of cellular evolution. Starting from a random initial state in Figure 2a, the lattice evolves to an equilibrium-like state 2b in 10 time steps. Throughout this evolutionary process, various patterns such as 'Star' emerge as shown in 2c2d2e2f in a timely manner.

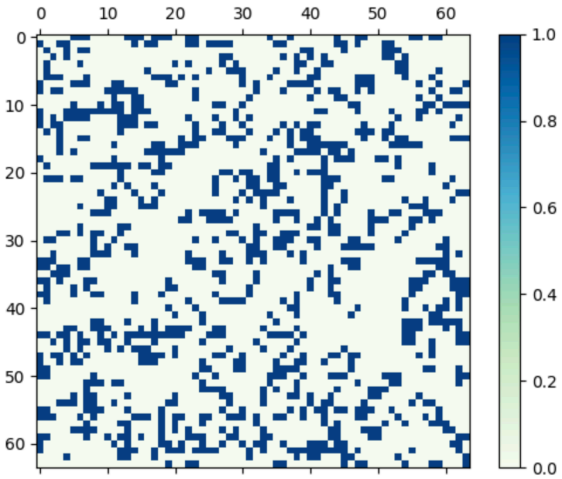


(a) Growth function G

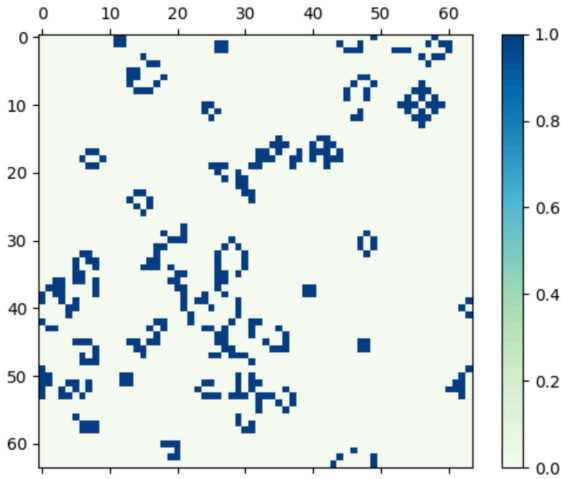


(b) Kernel K

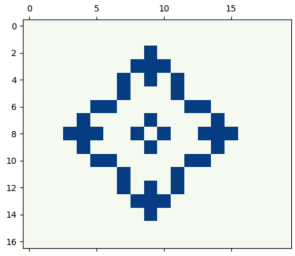
Figure 1: Game of Life functions



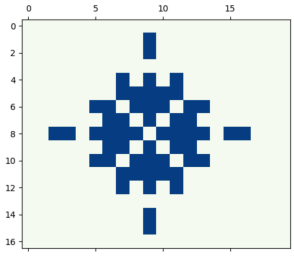
(a) Initial Random State



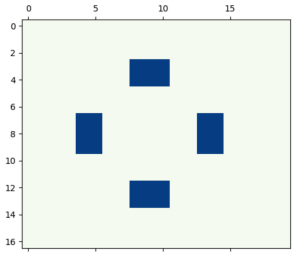
(b) Equilibrium-like State



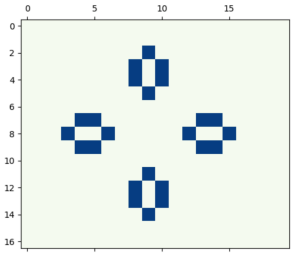
(c) Star state 1



(d) Star state 2



(e) Star state 3

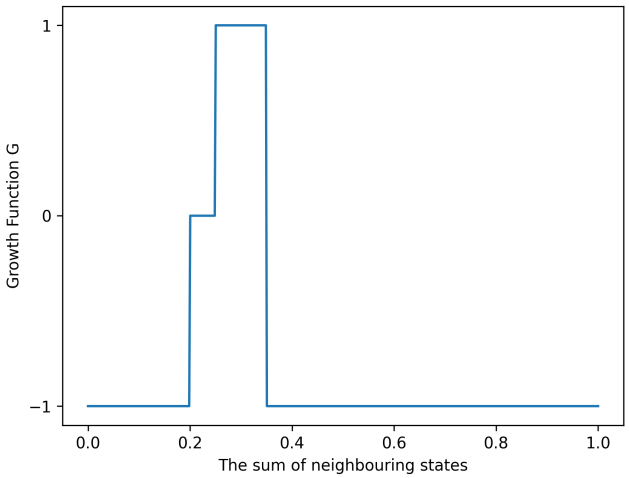


(f) Star state 4

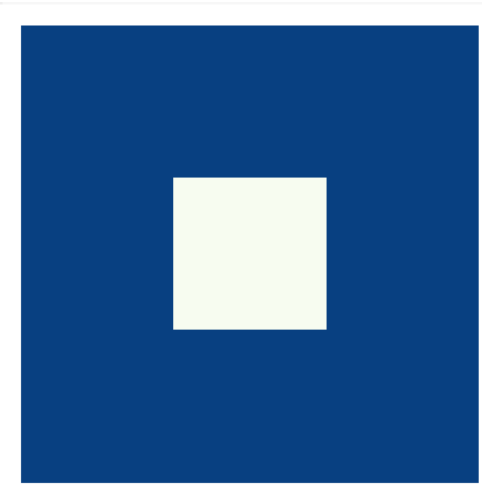
Figure 2: Game of Life simulation

3 Primordia

Evolving from Conway's Game of Life, Primordia is a more advanced CA. Primordia remains discrete in space, yet continuous in state and time. Unlike the binary state of cells in the Game of Life being strictly alive or dead, Primordia allows for a fluid continuum in the state, ranging from fully alive to completely dead. This change requires a modification in the growth function as it is continuous now. To expand the transition from discrete to continuous, we can readily set the time to be continuous by making Δt in equation 3 infinitesimally small. The new continuous growth function and the kernel are shown in Figure 3. The process of evolution from a random initial state is shown in figure 4



(a) Growth function G



(b) Kernel K

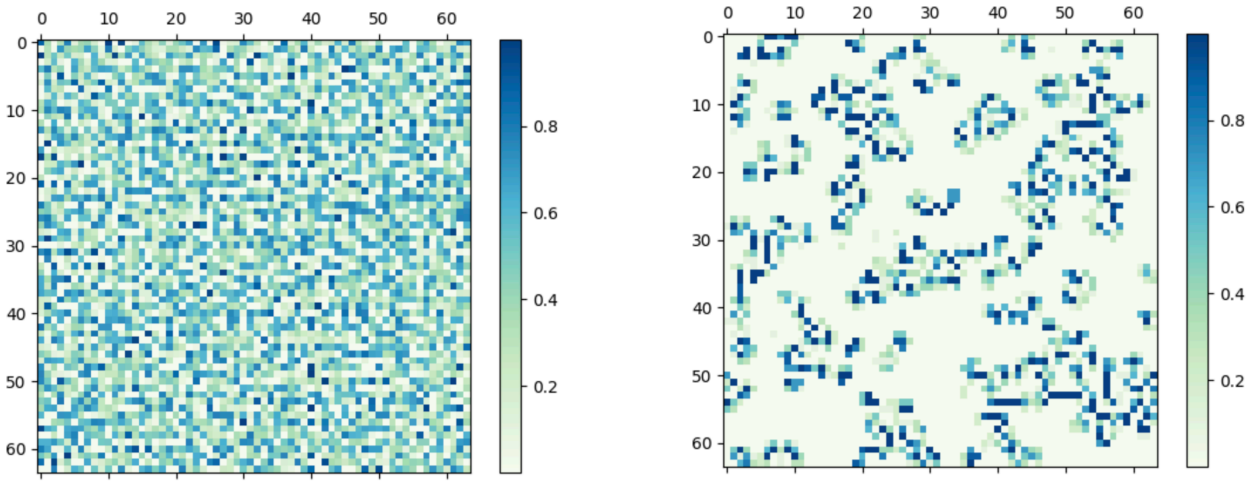
Figure 3: Primordia functions

4 Lenia

Through Conway's Game of Life and Primordia, we can finally introduce Lenia, a two-dimensional cellular automaton that is continuous in state, time and space. Lenia shows a great resemblance to biological organisms in real life. With the state being continuous, we choose from the following smooth growth functions:

$$G(U) = \begin{cases} 2\exp\left[-\frac{(\mathrm{U}-\mu)^2}{2\sigma^2}\right] - 1 & \text{exponential} \\ 2\left[1 - \frac{(U-\mu)^2}{9\sigma^2}\right]^4 - 1 & \text{polynomial} \end{cases} \tag{4}$$

Alternative to the discrete kernel shown previously, various smooth kernels can be adapted to Lenia. Each kernel K is determined by a kernel core function K_c and a kernel shell function K_s. K_c represents the general shape of each kernel layer and K_s indicates the positions



(a) Initial Random State

(b) Equilibrium-like State

Figure 4: Primordia simulation

and number of shells in the kernel function. By taking r to be the normalised distance to the central cell, two well-investigated core functions are:

$$K_c(r) = \begin{cases} \exp\left[4 - \frac{1}{\mathrm{r}(1-\mathrm{r})}\right] & \text{exponential} \\ [4r(1-r)]^4 & \text{polynomial} \end{cases} \tag{5}$$

Kernel shell functions create multiple copies of the core function by taking parameter β_i from a 1-dimensional vector β. The number of elements n in β is equivalent to the number of peaks and each peak corresponds to an equidistant concentric ring in the overall kernel function.

$$K_s(r, \beta) = \beta_i K_c(nr \bmod 1) \text{ for } \frac{i-1}{n} \leq r \leq \frac{i}{n} \tag{6}$$

Finally, the overall kernel is normalised:

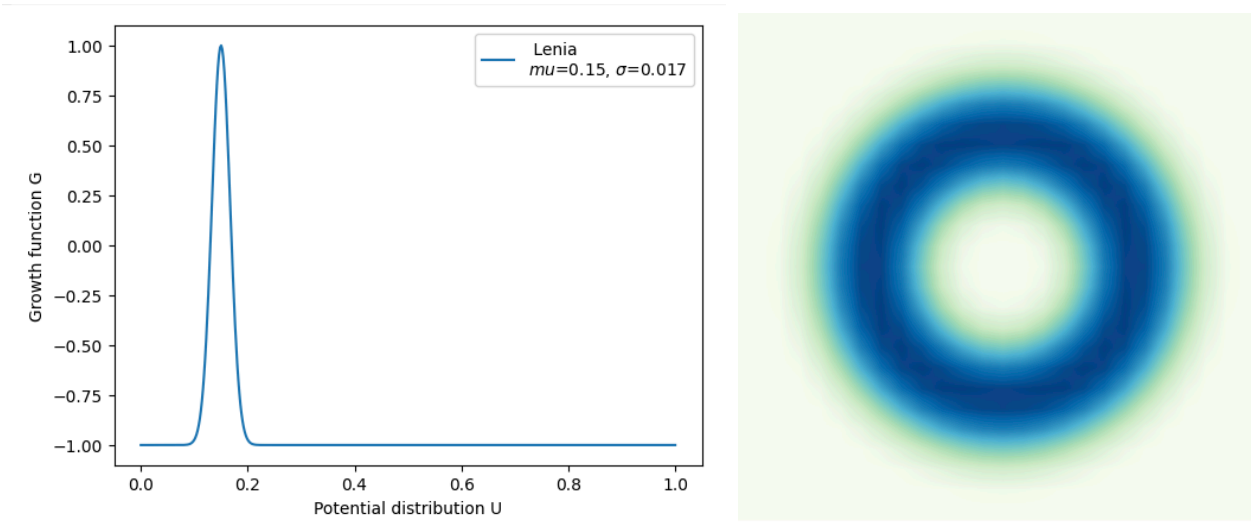
$$K = \frac{K_s}{|K_s|} \tag{7}$$

If β only contains one number, then the overall kernel is simply represented by the core kernel function. [3]

A pipeline of Lenia in Section 6.2.1 is available for reference.

4.1 Single Ring Kernel

Some iconic Lenia creatures employ a single-ring kernel. The kernel and growth function is shown in Figure 5. Again, we start from the evolution of a random initial state in 6a and 6b. The artificial cells emerge in the equilibrium state where we can see some independent



(a) Growth function G

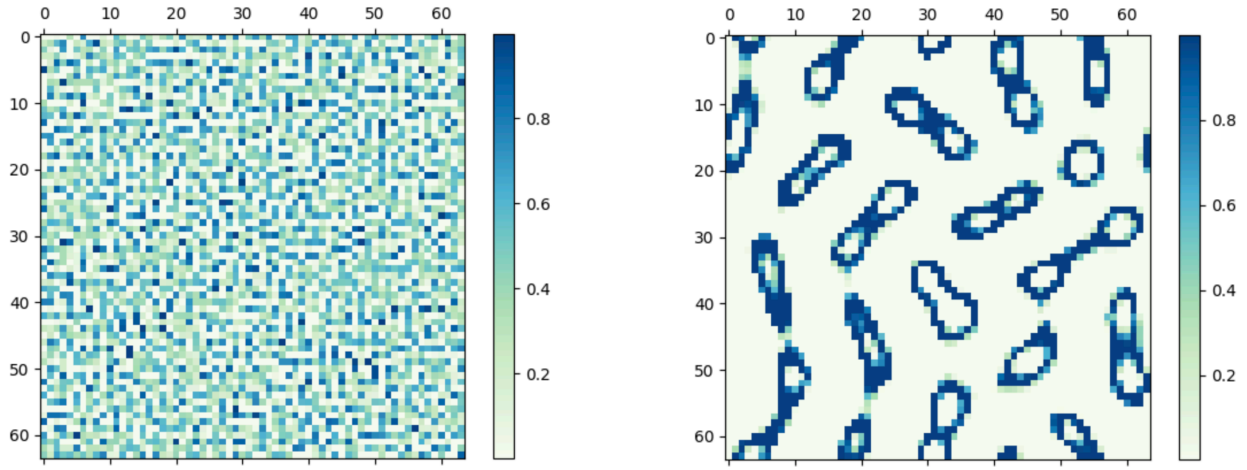
(b) Kernel K

Figure 5: Lenia functions

round cells and linked cells. By adjusting the parameters μ and σ for the growth function, those cells can also form worm-like chains at equilibrium and adapt perfect spatial symmetry. Then, the most well-known Lenia creature Orbium was simulated and shown in Figure 6c, 6d and 6e. The single Orbium moves linearly and steadily with constant mass and momentum. Orbium also exhibit unique interaction properties whereby they can collide and rebound from each other while maintaining their structural integrity as shown in Figure 6f-6h.

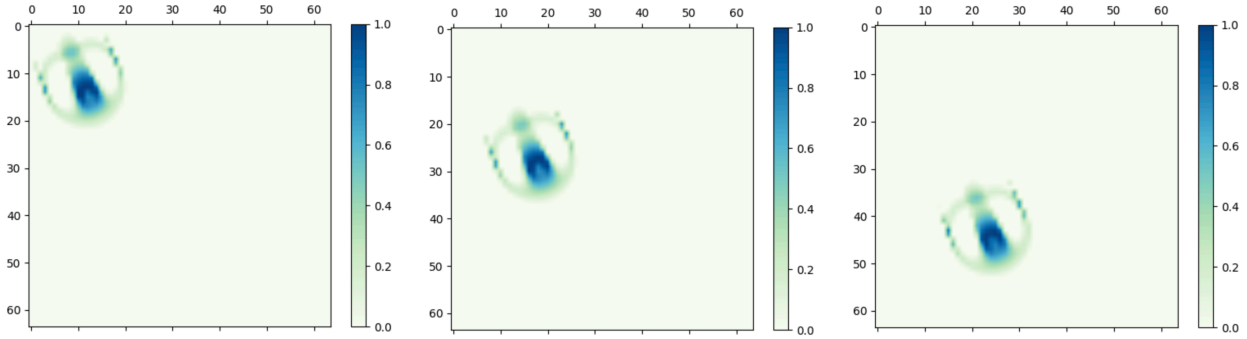
4.2 Multiple-Ring Kernel

More interesting Lenia creatures and behaviours have been discovered by employing a multiple-ring kernel. The research conducted to date has primarily focused on hundreds of creatures with bimodal(eg. Hexadentium, Parakronium), trimodal (eg. Geminium, Circogeminium) and tetramodal modes(eg. Quadrium, Heptafolium) [5]. In this project, I implemented two distinct tetramodal creatures from the Astridae family: Astrium Currens and Astrium Nausia. Despite the similar shape, Astrium Currens is characterised by its linear motion, in contrast to Astrium Nausia, which demonstrates a rotational motion. Figure 7 includes functions used in Currens case and the simulated motion. Figure 8 displays the case for Nausia.

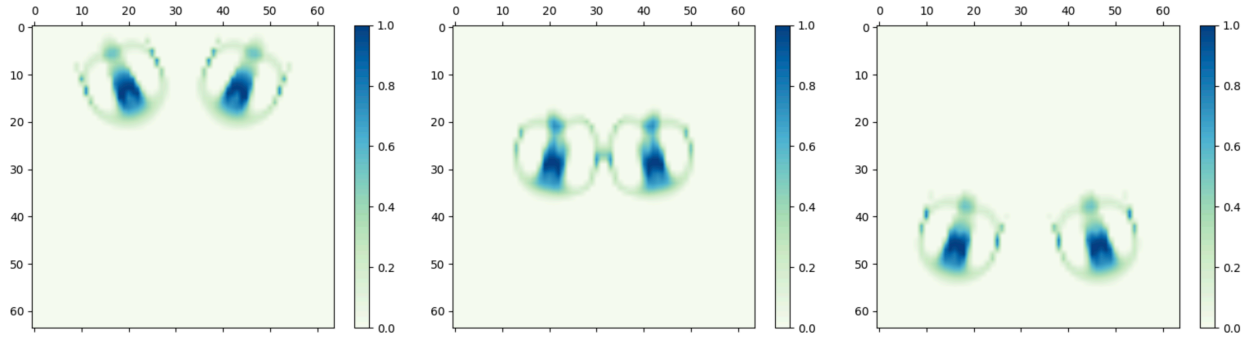


(a) Initial Random State

(b) Equilibrium State

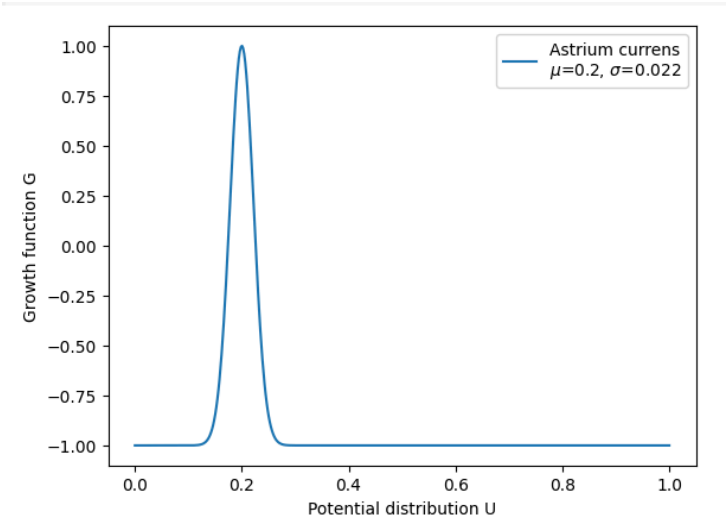


(c) Orbium state 1 (d) Orbium state 2 (e) Orbium state 3

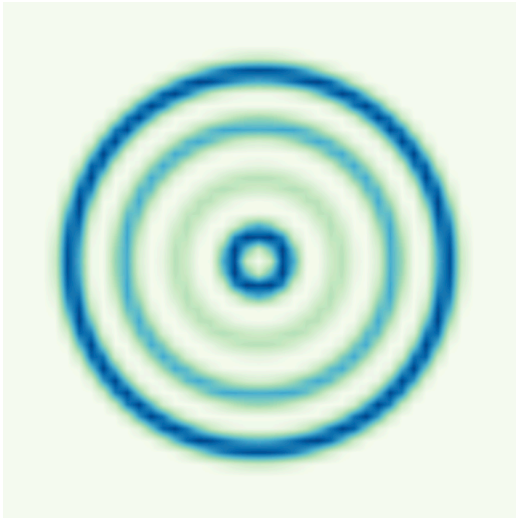


(f) Two Orbium state 1 (g) Two Orbium state 2 (h) Two Orbium state 3

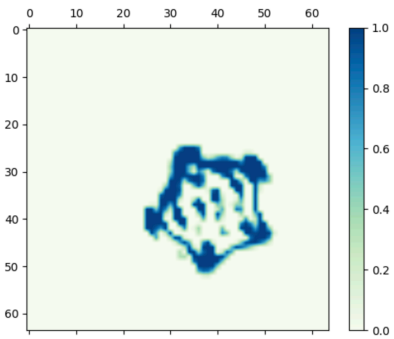
Figure 6: Lenia with single ring kernel simulation



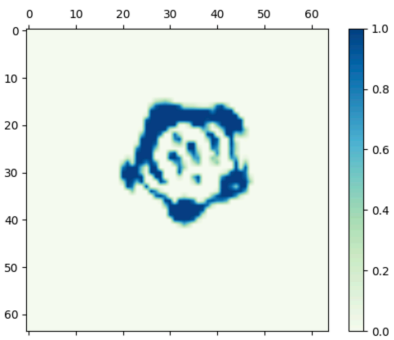
(a) Growth function G



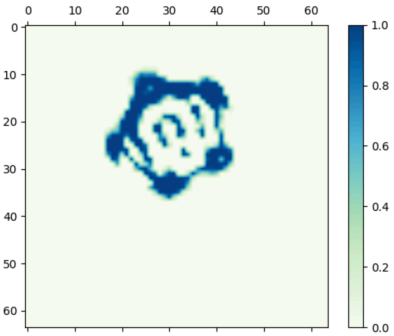
(b) Kernel K



(c) Currens state 1

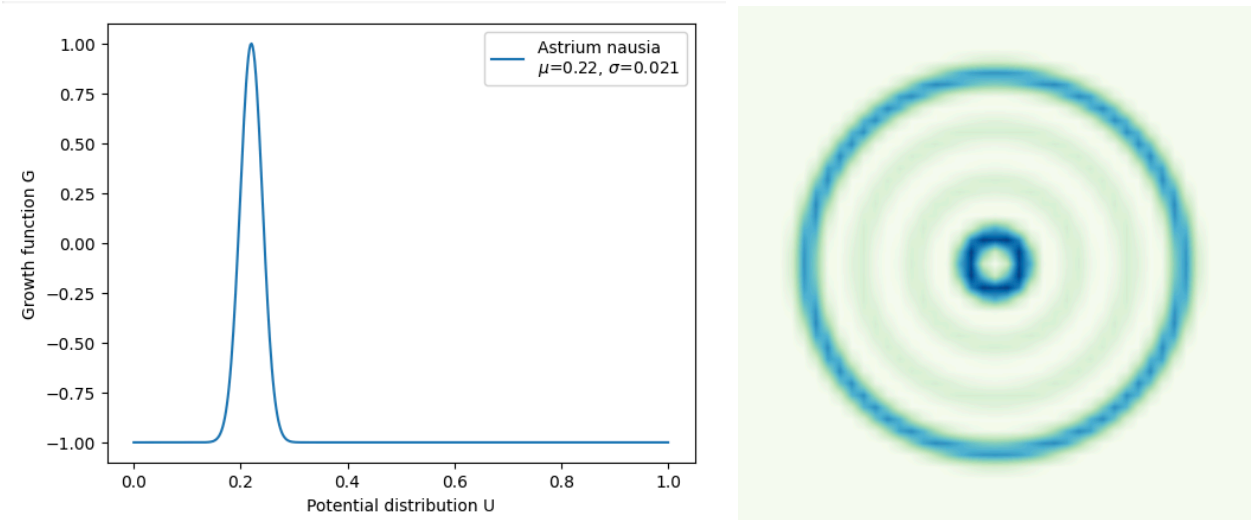


(d) Currens state 2



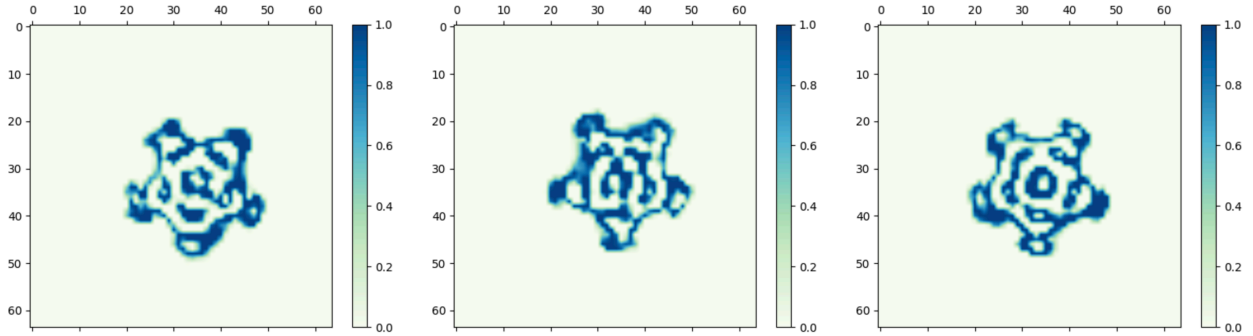
(e) Currens state 3

Figure 7: Astrium Currens



(a) Growth function G

(b) Kernel K



(c) Nausia state 1

(d) Nausia state 2

(e) Nausia state 3

Figure 8: Astrium Nausia

5 Particle Lenia

Although the original Lenia shows impressive performance in simulating the biological behaviours of artificial lives, there is no conservation law to confine the system, which leads to explosive growth or extinction behaviours readily. A recent approach towards this problem is called Flow Lenia [6], formulating the PDE system by introducing a flow field. By introducing such a flow field, the Lenia system is mass-conserved and fluid-like, preventing abrupt changes in creature behaviours. In this section, we focus on a variant called Particle Lenia, which can be seen as an alternative to Flow Lenia by representing Lenia system as a fixed number of particles, rather than a lattice [4].

The idea for Particle Lenia is inspired by Lennard-Jones potential — particles tend to attract when they are relatively far away yet begin to repel each other when they are close. In Particle Lenia, a motion law is added to constrain the motion of the system. We introduce a *energy field* $\mathbf{E}$, with its gradient equal to the rate of change of positions $\mathbf{x}$ of particles.

$$\frac{d\mathbf{x}}{dt} = -\nabla \mathbf{E}(\mathbf{x}) = -\left[\frac{\partial E_i}{\partial \mathbf{x}_i}\right]^T \tag{8}$$

$\mathbf{E}$ is defined as

$$\mathbf{E} = \mathbf{R} - \mathbf{A}(\mathbf{U}) \tag{9}$$

where $\mathbf{R}$ is a*repulsion potential field* and $\mathbf{A}$ is an *attraction potential field* (Note: $\mathbf{A}$ is not the lattice defined in previous sections any more!). For i^{th} particle at position $\mathbf{x}_i$, scalar values of R_i and A_i can be found by:

$$R_i = \frac{c_{rep}}{2} \sum_j \max\left[(1 - \mathrm{r_{ij}}), 0\right]^2 \tag{10}$$

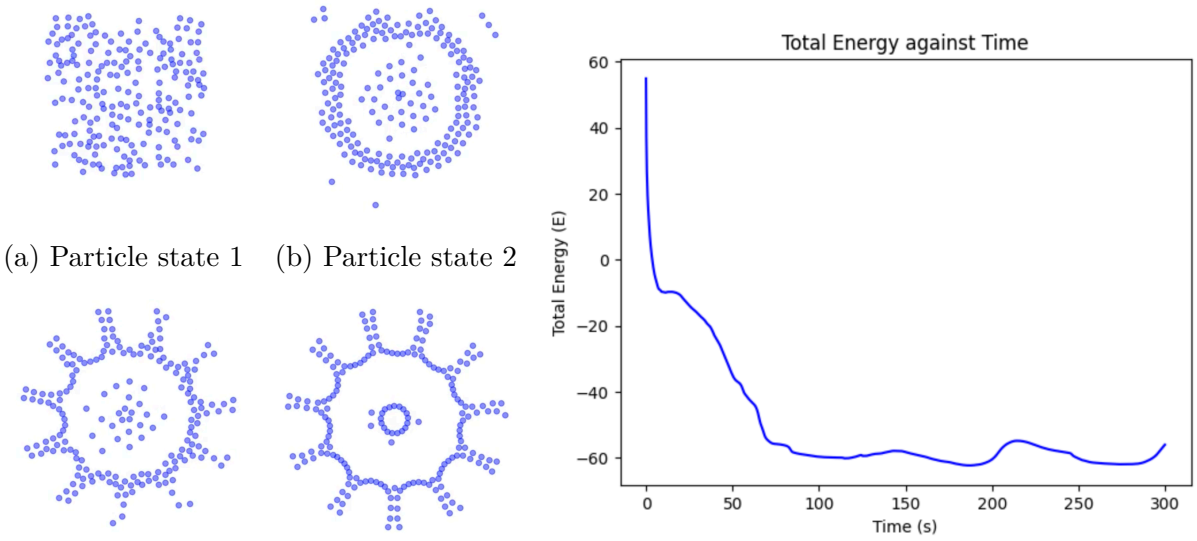
where r_{ij} is the Euclidean distance between i^{th} and j^{th} particles and c_{rep} is the repulsion strength coefficient, set to 1 by default. And:

$$A_i = \mathbf{A}(U_i) = \exp\left[-\frac{(\mathrm{U_i} - \mu_\mathrm{g})^2}{\sigma_\mathrm{g}^2}\right] \tag{11}$$

$$U_i = \sum_j K(r_{ij}) = \sum_j w_k \exp\left[-\frac{(\mathrm{r_{ij}} - \mu_\mathrm{k})^2}{\sigma_\mathrm{k}^2}\right] \tag{12}$$

Here w_k, μ_k, σ_k, μ_g, σ_g are constant parameters that characterise the interaction range.

There is a lot going on mathematically, readers can refer to the pipeline in Section 6.2.1. By analysing the above equations, we speculate that particles will minimise repulsion potential whereas flow to areas with higher attraction potential. Thus, starting from a random initial state of 200 particles, we expect particles to form some kind of pattern that is more stable. The result together with the energy analysis is shown in Figure 9. Particles initially form a multiple-ring structure and then gradually extend radially outward to seek a lower repulsion field. As expected, total energy mostly decreases throughout the



(a) Particle state 1 (b) Particle state 2

(c) Particle state 3 (d) Particle state 4 (e) Total energy against time plot

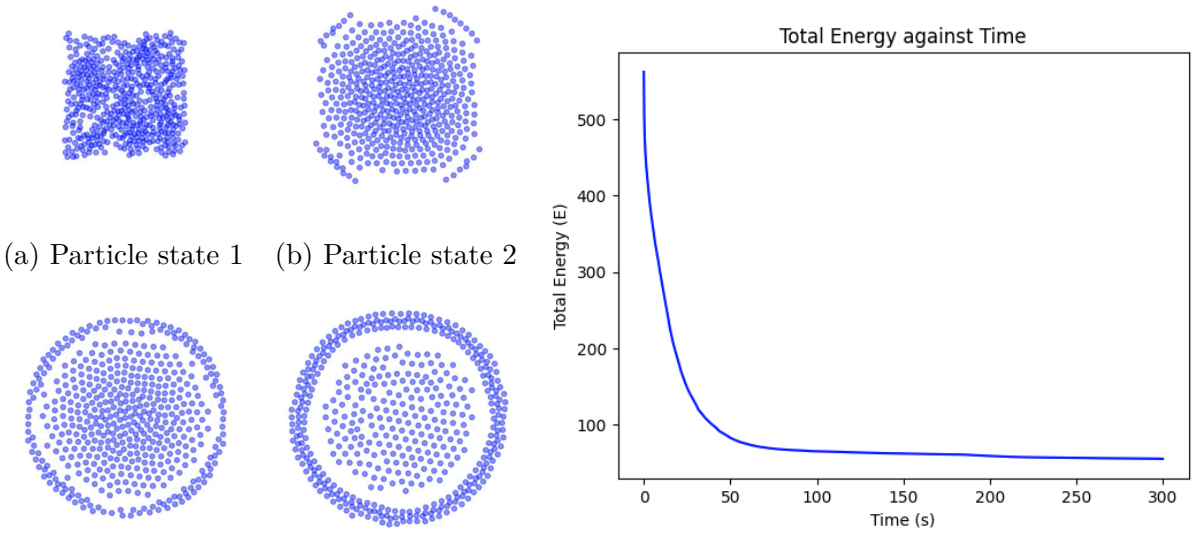
Figure 9: Particle Lenia simulation, 200 particles

motion upon phase transition, indicating the effectiveness of the idea of adding a motion law. Nevertheless, it is notable that total energy experiences some fluctuations during the motion, where transient increases in energy are observed. This phenomenon seems to violate the motion law as particles should only seek ways to minimise the total energy. According to [4], this is because each particle can only access information linked to itself and minimise its energy without considering the system as a whole.

This problem can be addressed when more particles are added to the system. Figure 10 shows the simulation and energy plot of 500 particles. When a higher population density is present, the initial repulsion potential is higher so that particles are pushed outwards with a greater strength. This higher initial speed allows particles to form a larger ring that accommodates more particles at equilibrium. Since each particle's motion is held up by its peers, the system is more stable to fluctuations as there are more particles included in the system. Consequently, the result of the total energy is a smooth curve that aligns with our expectations. Alternatively, we can also adopt a global energy rule instead of a local energy rule to get rid of local random fluctuations [4]:

$$\frac{d\mathbf{x}}{dt} = -\left[\frac{\partial E_{total}}{\partial \mathbf{p}_i}\right]^T \tag{13}$$

More in-depth investigation about the stability and adaptability of Particle Lenia can be found in [7].



(a) Particle state 1 (b) Particle state 2

(c) Particle state 3 (d) Particle state 4 (e) Total energy against time plot

Figure 10: Particle Lenia simulation, 500 particles

6 Implementation

6.1 Environment and Package

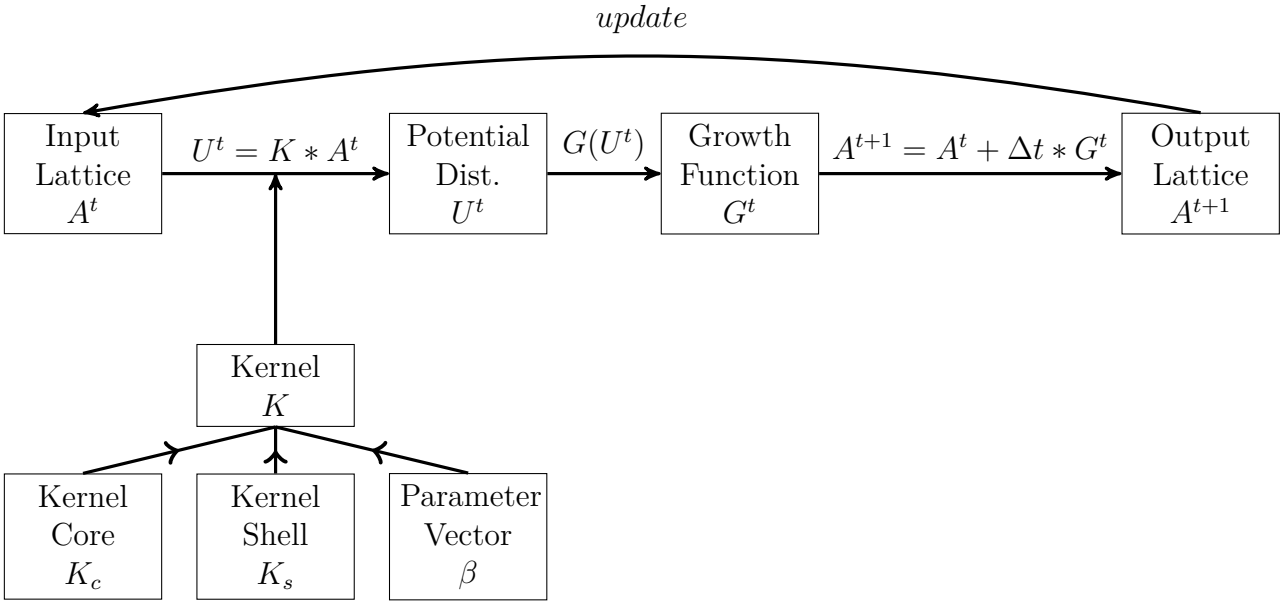
Language and Editor: Python 3, Google Colab Notebook
Package:

- Numpy — for array manipulation and fundamental numerical operations.
- Jax Numpy — for higher performance numerical computing
- Scipy — for 2-dimensional signal convolution
- Matplotlib.pylab — for growth function and kernel visualisation
- Matplotlib.animation — for lattice visualisation
- IPython.display — for play animation in line

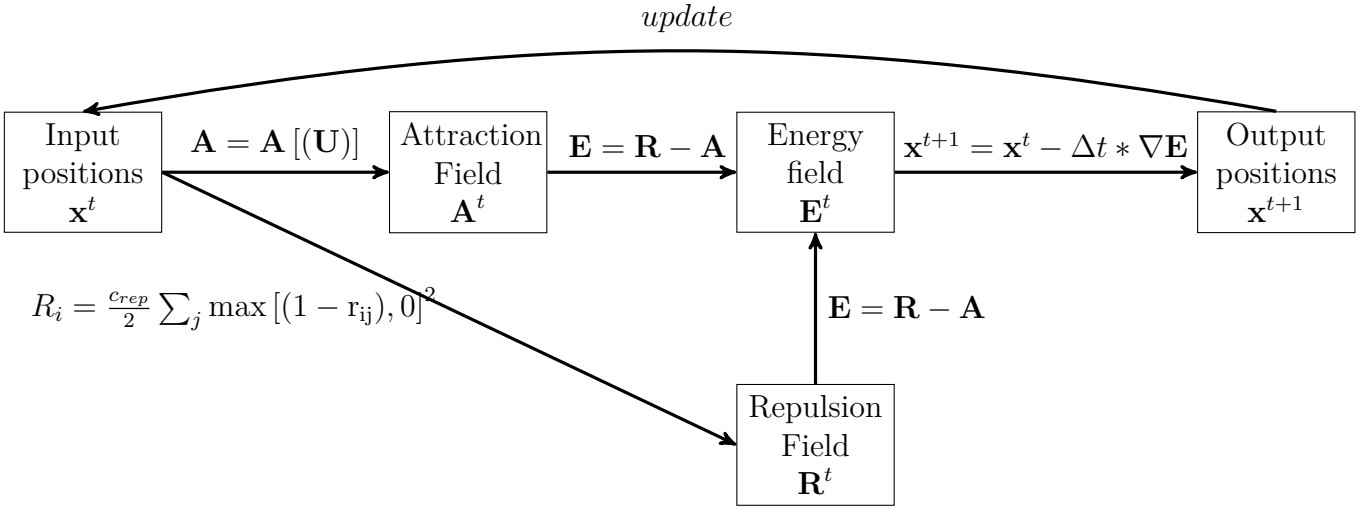
6.2 Algorithms

6.2.1 Execution Steps

Lenia:



Particle Lenia:



6.2.2 Pseudocode

The code structure can be found in Appendix A and a complete version of the Python code can be found in the submitted ZIP file. The development of this code was inspired by tutorials written by [8] and [4].

Algorithm 1 Lenia Algorithm

```
class Lenia:
    def __init__(self, R, peaks=None, mu, sigma, dt, kernel_type, delta_type):
        Initialize parameters
    def kernel_core(self, r):
        return exponential or polynomial function
    def kernel_shell(self, distance):
        Compute indices and peak values for kernel layers
        return Product of kernel_core and peak values
    def kernel(self, R):
        Create and normalise a distance grid matrix
        return Normalised kernel matrix
    def growth_function(self, U):
        return exponential or polynomial function
    def iteration(self, frameNum, img, lattice):
        Update lattice using FFT and inverse FFT
        Update image and return updated elements
```

Algorithm 2 Particle Lenia Algorithm

```
class Particle_Lenia:
    def __init__(self, μ_k, σ_k, w_k, μ_g, σ_g, c_rep, dt):
        Initialize parameters
    def gaussian_bump(self, r, μ, σ):
        return exponential function
    def e_field(self, points, x):
        Calculate r using Euclidean distance
        U ← Sum of Gaussian bumps ×w_k
        G ← Gaussian bump of U
        R ← Repulsion field
        E ← R − G
        return E
    def grad_e(self, points):
        Calculate gradient of E field
        return Vectorised gradient
    def total_energy(self, points):
        Vectorise e_field calculation
        Sum energies for total energy
        return total energy
    def update_particle(self, frameNum, img, points):
        Compute change in points
        Update points
        Set new image data
        Update stored energies
        return img
```

6.3 Complexity Analysis

In the original Lenia system, the complexity is dominated by the step calculating potential distribution U through the convolution of kernel and lattice. If a general convolution method is adopted, we expect a complexity $O(N^2M^2)$ for a $N \times N$ lattice and a $M \times M$ kernel. Alternatively, Fast Fourier Transform (FFT) can be implemented here to effectively speed up the program. The complexity for a 2-dimensional FFT of size $N \times N$ is $O(N^2\log(\mathrm{N}^2))$, which is exponentially lower than the general method. Because we can only use Numpy FFT for two equal-sized matrices, in the following comparison, we set $M = N$. The complexity comparison for the two methods is shown in Figure 11, from which we can deduce the FFT significantly improves the time complexity especially for large particle numbers.

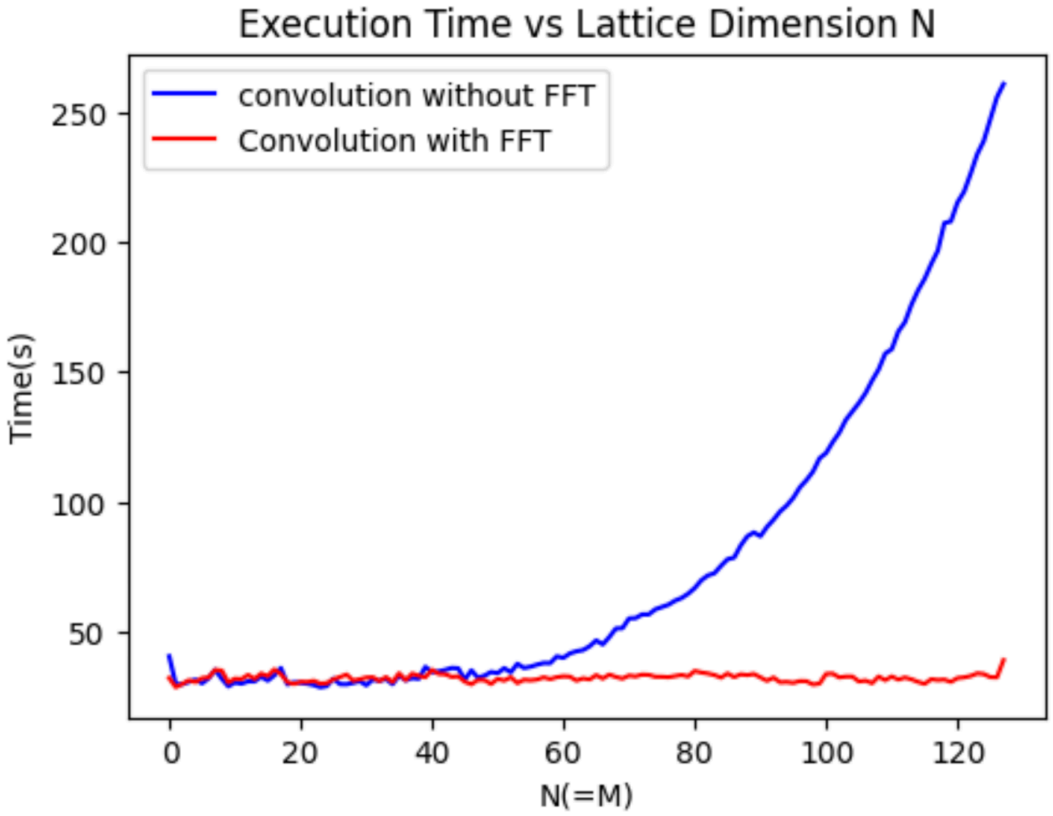


Figure 11: Lenia time complexity comparison of two methods

In Particle Lenia, the overall complexity is $O(n^2)$, where n is the number of points simulated. In fact, three key steps in Particle Lenia all require the complexity $O(n^2)$: calculating energy field E, finding the gradient of E and calculating the total energy. In practice, the program simulated 200 points for 3000 frames, which took approximately 200 seconds.

7 Conclusion

In summary, this computing project navigated through the evolution of cellular automaton, starting from the basic yet profound Conway's Game of Life to the more complex Lenia and Particle Lenia models. Interesting equilibrium patterns and behaviours of various Lenia patterns have been revealed along the implementation which was done in an object-oriented way using Python 3. In Lenia, the Fast Fourier Transform method was adopted to calculate the convolution product, with a leading complexity of $O(N^2M^2)$ for a lattice with size $N \times N$ and kernel size $M \times M$. In Particle Lenia, the computational load was primarily driven by the calculations of the energy field, gradient, and total energy, each scaling as $O(n^2)$, where n is the total number of particles in the system. To manage the increased demands of more extensive simulations and detailed energy analysis, the JAX package was employed, significantly enhancing performance and enabling the handling of larger systems and more frames.

Further work can be done to investigate the fluid aspect of Lenia as proposed in Flow Lenia. Flow Lenia allows multiple updating rules to coexist in an environment and enables us to find stable creatures more easily than the original Lenia. In addition, Particle Lenia can also extend to 3-dimensional space through energy-based models, which can closely mimic the interactions and complexities observed in biological organisms.

References

[1] John von Neumann. The general and logical theory of automata. *Cerebral Mechanisms in Behavior – The Hixon Symposium*, pages 1–41, 1951.

[2] Martin Gardner. The fantastic combinations of john conway's new solitaire game 'life'. *Mathematical Games*, 223(4):120–123, 1970.

[3] Bert Wang-Chak Chan. Lenia: Biology of artificial life. *Complex Systems*, 28(3):251–286, October 2019.

[4] Alexander Mordvintsev, Eyvind Niklasson, and Ettore Randazzo. *Particle Lenia and the Energy-based Formulation*. 2022.

[5] Bert Wang-Chak Chan. Lenia web demo.

[6] Erwan Plantec, Gautier Hamon, Mayalen Etcheverry, Pierre-Yves Oudeyer, Clément Moulin-Frier, and Bert Wang-Chak Chan. Flow-lenia: Towards open-ended evolution in cellular automata through mass conservation and parameter localization, 2023.

[7] Kazuya Horibe, Keisuke Suzuki, Takato Horii, and Hiroshi Ishiguro. Exploring the adaptive behaviors of particle lenia: A perturbation-response analysis for computational agency. In *The 2023 Conference on Artificial Life*, ALIFE 2023. MIT Press, 2023.

[8] Bert Chan, Will Cavendish, and Slackermanz. *Tutorial: From Conway's Game of Life to Lenia*.

A Code Structure

```
/
├── creatures/ ..... Imported creature data
├── tools/ ..... Tool functions (mainly visualisation functions)
├── game_of_life/ ..... Conway's Game of Life implementation and visualisation
├── primordia/ ..... Primordia implementation and visualisation
├── lenia_main/ ..... Lenia implementation only
│   ├── lenia_orbium/ ..... Orbium visualisation
│   ├── lenia_currens/ ..... Astrium Currens visualisation
│   └── lenia_nausia/ ..... Astrium Nausia visualisation
└── particle_lenia_main/ ..... Particle Lenia implementation only
    └── particle_simu/ ..... Particle Lenia visualisation
```