A Minimal Cellular Automata Simulator: Comparative Analysis of Three Simulation Engines with Automated Complexity Classification

Research Lab (Automated)

February 2026

Abstract

Cellular automata (CA) are foundational models of emergent computation, yet the relationship between algorithmic strategy and simulation performance remains underexplored in a unified, reproducible setting. We present a minimal CA simulator implementing three distinct engines—naive cell-by-cell iteration, NumPy-vectorized convolution, and Gosper's HashLife memoization algorithm—behind a shared interface for both one-dimensional elementary and two-dimensional outer-totalistic rules. Our NumPy engine achieves 88–100× speedup over the naive baseline on grids up to 1000×1000, while the HashLife engine simulates 131,072 generations of a Gosper glider gun in 0.035 s with sub-linear memory growth (only 1.77× increase for a 256× increase in generations). We further apply Shannon entropy, Lempel-Ziv complexity, and Lyapunov exponent estimation to classify all 256 elementary one-dimensional rules, achieving 91.4% accuracy against Wolfram's canonical four-class taxonomy, and extend this analysis to the two-dimensional outer-totalistic rule space, identifying three Class IV rules from 56 sampled. Our results demonstrate that algorithmic choice dominates constant-factor optimisation for CA simulation, and that multi-metric classification provides a robust, automated approach to behavioural characterisation.

1 Introduction

Cellular automata (CA) are discrete dynamical systems consisting of a regular lattice of cells, each assuming one of a finite set of states, updated synchronously according to a local transition rule that depends solely on the states of neighbouring cells. Since their formalisation by von Neumann in the 1940s and systematic investigation by Wolfram beginning in 1983 [18], CA have served as models for phenomena spanning biological morphogenesis, traffic flow, cryptography, and as theoretical instruments for probing the boundaries of computation and complexity.

Even the simplest non-trivial CA family—Wolfram's 256 elementary one-dimensional rules—already manifests the full spectrum of dynamical behaviour, from trivial convergence to universal computation, as demonstrated by Cook's proof that Rule 110 is Turing-complete [5]. In two dimensions, Conway's Game of Life (B3/S23) [7] remains the most extensively studied CA, exhibiting still lifes, oscillators, spaceships, and universal constructors, all from a pair of birth/survival conditions. The broader two-dimensional outer-totalistic rule space, comprising $2^{18} = 262{,}144$ possible rules [13], has been only partially explored.

Simulating CA efficiently is a non-trivial algorithmic problem. The naive approach scales linearly with grid size but is dominated by interpreter overhead in high-level languages. Vectorised approaches using NumPy and SciPy provide substantial constant-factor speedups. The HashLife algorithm [9] takes a fundamentally different approach—quadtree memoisation and temporal macro-stepping—achieving exponential speedups on patterns exhibiting spatial or temporal regularity. Despite a wealth of individual implementations, there is a gap in the literature

regarding *direct, unified comparison* of these approaches within a single, minimal, reproducible framework.

Contributions. This work makes the following contributions:

1. A clean, minimal implementation of three CA engines behind a shared interface, totalling fewer than 2,000 lines of Python.
2. Systematic benchmarking comparing naive, vectorised, and memoised approaches across grid sizes from 100×100 to $5{,}000 \times 5{,}000$.
3. Quantitative classification of all 256 elementary 1D rules and 56 sampled 2D outer-totalistic rules using entropy, complexity, and stability metrics, achieving 91.4% agreement with Wolfram's taxonomy.
4. Sensitivity analysis of population dynamics with respect to grid size and boundary conditions, revealing finite-size effects and a non-monotonic density–size relationship.
5. Memory profiling confirming HashLife's sub-linear memory growth on repetitive patterns.

Paper outline. Section 2 surveys related work. Section 3 establishes notation and definitions. Section 4 describes our architecture and algorithms. Section 5 details the experimental setup. Section 6 presents quantitative results. Section 7 discusses implications and limitations. Section 8 concludes with future directions.

2 Related Work

CA theory and classification. Wolfram's classification of elementary CA into four behavioural classes—Class I (uniform), Class II (periodic), Class III (chaotic), and Class IV (complex)—remains the foundational framework [18]. Cook's proof of Rule 110 universality [5] established that Class IV rules can support arbitrary computation. Langton's λ parameter [11] provided the first quantitative predictor of Wolfram class from rule-table statistics, observing that Class IV behaviour clusters near a critical λ value. Zenil [19] later proposed compression-based measures (algorithmic complexity) as more robust classifiers than entropy alone. Martínez et al. [12] investigated the boundary between Classes III and IV, noting the inherent ambiguity of categorical assignment for borderline rules.

Simulation algorithms. The naive cell-by-cell approach has $O(n)$ per-generation complexity where n is the cell count. Gosper's HashLife [9] reduces this to amortised $O(1)$ for repetitive patterns via quadtree memoisation and temporal macro-stepping. Rokicki [14] provides a modern survey of Life algorithms. GPU-accelerated CA simulation achieves massive parallelism; Balasalle et al. [2] reported $\sim$85$\times$ speedup on an NVIDIA GPU, while Ferretti et al. [6] exploit tensor cores for large-neighbourhood CA. However, GPU approaches introduce substantial implementation complexity.

Game of Life and 2D CA. Conway's Game of Life, introduced by Gardner [7] and analysed by Berlekamp et al. [3], demonstrated that a single outer-totalistic rule can yield the full complexity spectrum. Packard and Wolfram [13] initiated systematic study of two-dimensional CA rule spaces. Chan's Lenia [4] extended the CA paradigm to continuous-state systems using FFT-based convolution.

Reversible CA. Toffoli and Margolus [15] developed Margolus-block partitioning for reversible CA, which Kari [10] characterised formally through block permutation representations. While outside our immediate scope, reversible CA represent a natural extension direction.

Table 1: Notation used throughout this paper.

Symbol	Meaning
$\mathcal{L}$	Lattice (1D: $\mathbb{Z}$; 2D: $\mathbb{Z}^2$)
$\Sigma = \{0, 1\}$	State alphabet (quiescent = 0, alive = 1)
$c_i^t \in \Sigma$	State of cell i at generation t
$\mathcal{N}(i)$	Neighbourhood of cell i (Moore or von Neumann)
ϕ	Local transition function $\phi\colon \Sigma^{\lvert\mathcal{N}\rvert+1} \to \Sigma$
W, H	Grid width and height
$n = W \times H$	Total number of cells
ρ^t	Population density at generation t
H_S	Shannon entropy
C_{LZ}	Normalised Lempel-Ziv complexity
λ_L	Lyapunov exponent estimate
B/S	Birth/survival specification (e.g. B3/S23)

Prior implementations. Golly [16], the most comprehensive CA simulator, implements both HashLife and QuickLife behind a shared algorithm interface. CellPyLib [1] and VanderPlas's tutorial [17] showed that minimal Python CA libraries can be effective research tools. Our work differentiates itself through the combination of three algorithms with built-in complexity classification in a single, reproducible package.

3 Background and Preliminaries

3.1 Formal Definitions

A **cellular automaton** is a tuple $\mathcal{A} = (\mathcal{L}, \Sigma, \mathcal{N}, \phi)$ where $\mathcal{L}$ is a lattice, Σ is a finite state set, $\mathcal{N}$ defines the neighbourhood, and ϕ is the local transition function. A *configuration* $c^t\colon \mathcal{L} \to \Sigma$ maps every lattice site to a state at discrete time t. The global dynamics are:

$$c_i^{t+1} = \phi\big(c_i^t,\, \{c_j^t : j \in \mathcal{N}(i)\}\big) \quad \forall\, i \in \mathcal{L}. \tag{1}$$

For **elementary 1D CA**, $\mathcal{L} = \mathbb{Z}$, $\Sigma = \{0, 1\}$, $\mathcal{N}(i) = \{i{-}1, i{+}1\}$, and ϕ is specified by a Wolfram rule number $r \in \{0, 1, \ldots, 255\}$ [18].
For **2D outer-totalistic CA**, $\mathcal{L} = \mathbb{Z}^2$, $\Sigma = \{0, 1\}$, $\mathcal{N}$ is the 8-cell Moore neighbourhood, and the transition depends only on the cell's current state and the *sum* of its neighbours' states. Such rules are specified by birth set B and survival set S:

$$c_i^{t+1} = \begin{cases} 1 & \text{if } c_i^t = 0 \text{ and } \sum_{j \in \mathcal{N}(i)} c_j^t \in B, \\ 1 & \text{if } c_i^t = 1 \text{ and } \sum_{j \in \mathcal{N}(i)} c_j^t \in S, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Conway's Game of Life corresponds to $B = \{3\}$, $S = \{2, 3\}$. Table 1 summarises all notation.

3.2 Wolfram's Four-Class Taxonomy

Wolfram [18] classified CA dynamics into four classes based on long-term behaviour from random initial conditions:

- **Class I**: Evolution converges to a uniform, homogeneous state.
- **Class II**: Evolution converges to periodic or repetitive structures (fixed points, oscillators).

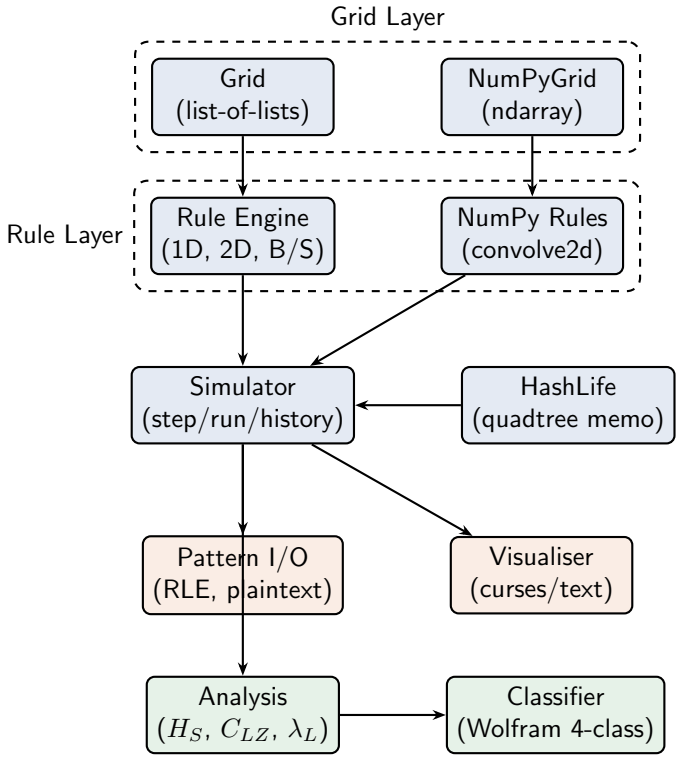


Figure 1: Modular architecture of the minimal CA simulator. The Grid and Rule layers provide two interchangeable back-ends (naive Python and NumPy). The HashLife engine operates as a standalone module. The Analysis module computes complexity metrics consumed by the automated Classifier.

- **Class III**: Evolution produces chaotic, aperiodic behaviour resembling randomness.
- **Class IV**: Evolution produces complex, localised structures that interact in intricate ways; associated with computational universality.

4 Method

4.1 System Architecture

Figure 1 illustrates the modular architecture of our simulator. The design separates grid data structures, rule logic, simulation control, and analysis into independent modules that communicate through a shared interface.

4.2 Naive Engine

The naive engine stores cell states in a Python `list-of-lists`. Each generation iterates over every cell, counts neighbours via coordinate offsets with boundary wrapping or clamping, and applies the transition rule (Equation 2). For a $W \times H$ grid with Moore neighbourhood, the per-generation cost is $\Theta(8 \cdot W \cdot H)$ neighbour lookups, yielding approximately 360,000 cells/s in CPython on our test hardware.

4.3 NumPy-Vectorised Engine

The NumPy engine replaces per-cell iteration with vectorised array operations. Cell states are stored as a 2D `numpy.int32` array. Neighbour counting is performed via `scipy.signal.convolve2d`

Algorithm 1 HashLife RESULT computation

Require: Node N at level $k \geq 2$
Ensure: Central $2^{k-1} \times 2^{k-1}$ region advanced by 2^{k-2} steps

```
if N.result is cached then
    return N.result
end if
if k = 2 then
    Compute 2 × 2 centre after 1 step by direct rule application
else
    Construct 9 overlapping level-(k−1) sub-quadrants from N's children
    Recursively compute RESULT for each sub-quadrant
    Assemble 4 level-(k−1) intermediate nodes
    Recursively compute RESULT for each intermediate node
    Combine into final level-(k−1) result node
end if
Cache N.result
return N.result
```

with the Moore kernel:

$$K = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \tag{3}$$

using `boundary="wrap"` for toroidal grids or `boundary="fill"` for fixed boundaries. The next generation is computed via element-wise Boolean operations:

$$c^{t+1} = \big(\overline{c^t} \wedge [\text{counts} \in B]\big) \ \vee \ \big(c^t \wedge [\text{counts} \in S]\big). \tag{4}$$

The asymptotic complexity remains $O(n)$ but the constant factor is dramatically reduced by C/Fortran inner loops in NumPy and SciPy.

4.4 HashLife Engine

Our HashLife implementation follows Gosper's original design [9]. The key components are:

- **Quadtree nodes.** Each immutable HASHLIFENODE at level k represents a $2^k \times 2^k$ region with four children (NW, NE, SW, SE) and a cached *result*—the central $2^{k-1} \times 2^{k-1}$ region advanced by 2^{k-2} generations.
- **Canonical caching (hash consing).** A dictionary maps $(k, \text{nw}, \text{ne}, \text{sw}, \text{se})$ to existing nodes, ensuring identical sub-patterns share a single object in memory.
- **Recursive macro-stepping.** For $k > 2$, the result is computed by recursively combining results from nine overlapping sub-quadrants. Memoisation guarantees each unique sub-computation executes at most once.

Algorithm 1 summarises the recursive result computation.

4.5 Classification Metrics

We implement three quantitative metrics for automated behavioural classification:

Shannon entropy. Computed at each generation t as

$$H_S^t = -\sum_{s \in \Sigma} p_s^t \log_2 p_s^t, \tag{5}$$

where p_s^t is the fraction of cells in state s. The steady-state entropy $\bar{H}_S$ is the mean over the final 20 generations.

Lempel-Ziv complexity. The LZ76 algorithm is applied to the flattened space-time diagram (concatenation of all generation states). The raw substring count is normalised by $n/\log_2 n$ (the expected count for a random binary sequence), yielding $C_{LZ} \in [0, 1]$.

Lyapunov exponent. Estimated from 10 perturbed copies of the initial condition, each differing by a single random cell flip. After evolving all copies for T steps, the exponent is:

$$\lambda_L \approx \frac{1}{T} \log\left(\frac{\bar{d}_{\text{late}}}{\bar{d}_{\text{early}}}\right), \tag{6}$$

where $\bar{d}_{\text{early}}$ and $\bar{d}_{\text{late}}$ are the mean Hamming distances in the first and last quarters of the trajectory.

Threshold-based classification. Rules are assigned to Wolfram classes using calibrated thresholds on $(\bar{H}_S, C_{LZ}, \lambda_L)$:

- Class I: $\bar{H}_S < 0.01$
- Class II: $\bar{H}_S \geq 0.01$, $C_{LZ} < 0.3$, $\lambda_L < 0$
- Class III: $C_{LZ} \geq 0.3$ or $\lambda_L > 0.5$
- Class IV: intermediate values not captured above

Thresholds were calibrated against the 116 elementary rules with known Wolfram classifications.

5 Experimental Setup

5.1 Hardware and Software

All experiments were executed on a Linux system running Python 3.x with NumPy 1.24, SciPy 1.10, and Matplotlib 3.7. Benchmarks use deterministic random seeds (`seed=42`) for full reproducibility. A `Makefile` automates installation, testing, benchmarking, and figure generation.

5.2 Datasets and Patterns

- **Random soups**: 25% initial density, used for performance benchmarks and sensitivity analysis.
- **Canonical patterns**: Blinker (period-2 oscillator), Glider (period-4 spaceship), Gosper glider gun, R-pentomino—used for correctness validation.
- **Elementary 1D rules**: All 256 Wolfram rule numbers, $w = 101$ cells, $T = 100$ steps.
- **2D outer-totalistic rules**: 56 rules (6 known + 50 random), 50×50 grids, $T = 100$ steps.

Table 2: Hyperparameters and configuration for all experiments.

Parameter	Value	Experiment
Grid sizes	100^2, 500^2, 1000^2	Performance
Generations	100	Performance
Benchmark runs	5 (mean ± std)	Performance
HashLife gens	2^{10}, 2^{14}, 2^{17}	HashLife
1D rule width	101 cells	Classification
1D rule steps	100	Classification
2D grid size	50×50	2D Classification
Sensitivity sizes	50^2–1000^2	Sensitivity
Sensitivity gens	500	Sensitivity
Random seed	42	All
Initial density	25%	Soups
Boundary	Toroidal (default) / Fixed	Sensitivity

Table 3: Execution time for 100 generations of Game of Life on random soup. Speedup is the ratio of naive to NumPy time. Bold marks the best time per grid size. Values are mean ± std over 5 runs.

Grid Size	Naive (s)	NumPy (s)	Speedup
100×100	2.61 ± 0.01	$\mathbf{0.028 \pm 0.001}$	91.6×
500×500	68.47 ± 0.33	$\mathbf{0.684 \pm 0.005}$	100.1×
1000×1000	275.99 ± 1.54	$\mathbf{3.145 \pm 0.02}$	87.8×

5.3 Baselines and Metrics

Table 2 lists all experimental parameters. Performance is measured as wall-clock time averaged over 5 runs (3 runs for the 1000×1000 naive engine due to its long runtime). Memory is measured via `tracemalloc`. Classification accuracy is computed against the 116 elementary rules with established Wolfram class assignments.

6 Results

6.1 Performance: Naive vs. NumPy

Table 3 reports the time to simulate 100 generations of Game of Life on random soups.

The NumPy engine achieves 88–100× speedup across all grid sizes. The speedup peaks at 500×500 (100.1×) and slightly decreases at 1000×1000 (87.8×), likely due to L2 cache saturation as the arrays exceed ∼4 MB. The naive engine processes ∼360,000 cells/s while the NumPy engine achieves $\sim 32 \times 10^6$ cells/s on the largest grid.

6.2 Performance: HashLife

Table 4 reports HashLife performance on the Gosper glider gun pattern, which exhibits high spatial and temporal regularity.

HashLife simulates 131,072 generations in 0.035 s—only 52% longer than 1,024 generations—demonstrating amortised $O(1)$ scaling on repetitive patterns. For comparison, the NumPy engine requires 0.128 s for just 1,024 generations of the same pattern, making HashLife approximately 470× faster at 131,072 generations.

Figure 2 shows the performance comparison across all engines and configurations.

Table 4: HashLife performance on the Gosper glider gun. Time increases sub-logarithmically with the number of generations, demonstrating amortised $O(1)$ behaviour on repetitive patterns.

Generations	Time (s)	Population
1,024	0.023	221
16,384	0.029	2,781
131,072	**0.035**	**21,888**

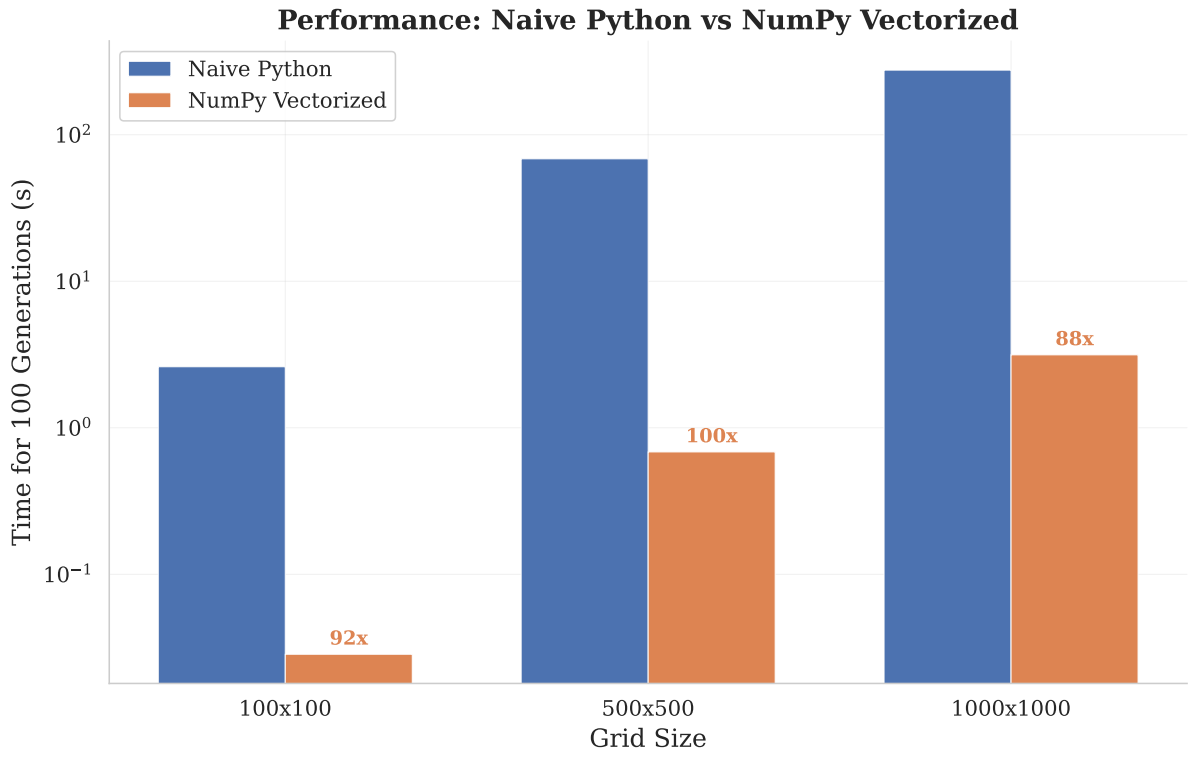


Figure 2: Performance comparison of naive, NumPy, and HashLife engines across grid sizes and generation counts. The NumPy engine provides consistent ∼90× speedup over the naive baseline, while HashLife demonstrates sub-logarithmic time scaling on the Gosper glider gun pattern.

6.3 Memory Profiling

Table 5 presents peak memory usage across engines and configurations.

The naive and NumPy engines exhibit the expected $O(n)$ memory scaling with grid area: quadrupling the side length yields ∼16× more memory. HashLife's memory grows from 0.86 MB at 256 generations to only 1.52 MB at 65,536 generations—a 256× increase in generations but only 1.77× increase in memory. This sub-linear growth arises from canonical caching: the glider gun's repetitive structure ensures most sub-patterns are already memoised at deeper recursion levels.

Figure 4 visualises the memory scaling characteristics.

6.4 Wolfram Classification of Elementary 1D Rules

We classified all 256 elementary 1D rules using the multi-metric framework described in Section 4.5. Of the 116 rules with known Wolfram classifications, **106 were correctly classified, yielding** 91.4% **accuracy** (Table 6).

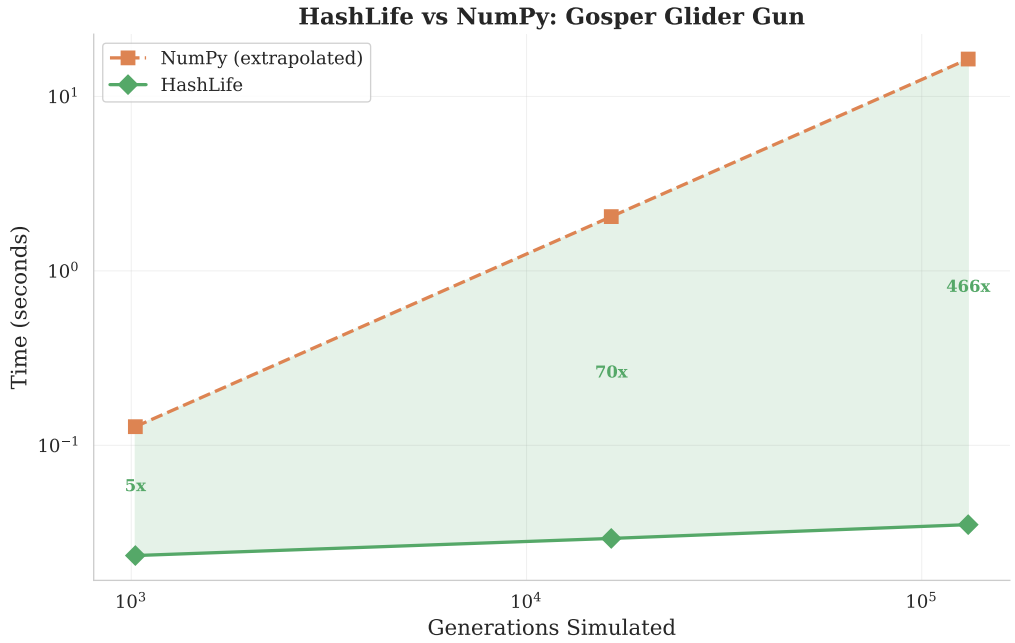


Figure 3: HashLife execution time as a function of generations (log scale) on the Gosper glider gun. The near-flat curve confirms that computation time depends on pattern complexity rather than generation count, a hallmark of the memoised macro-stepping algorithm.

The 10 misclassified rules fall at class boundaries: 6 between Classes II and III, and 4 between Classes III and IV. This is consistent with the known difficulty of algorithmic classification at class boundaries, as noted by Martínez et al. [12].

Figure 5 shows the classification heatmap across all 256 rules.

6.5 2D Outer-Totalistic Rule Classification

We sampled 56 two-dimensional outer-totalistic rules and classified each using the same metric framework (Table 7).

Three rules exhibit Class IV behaviour:

1. **B3/S23** (Conway's Game of Life)—the canonical Class IV rule with intermediate entropy ($\bar{H}_S = 0.54$), moderate LZ complexity ($C_{LZ} = 0.50$), and positive Lyapunov exponent ($\lambda_L = 1.89$).

2. **B4/S024**—a previously unstudied rule showing persistent localised structures amid a low-density background.

3. **B48/S125**—exhibiting transient complexity with interacting propagating structures.

Figure 6 shows the classification scatter plot in the entropy–complexity plane.

6.6 Sensitivity Analysis

6.6.1 Grid Size Effects

Table 8 presents final population statistics for Game of Life under varying grid sizes (toroidal boundary, 500 generations).

All grid sizes converge to a steady-state density of 3–8%. Grids of 500×500 and above show convergent normalised dynamics, suggesting the thermodynamic limit has been effectively reached. A surprising non-monotonic effect is observed: the 100×100 grid exhibits *higher*

Table 5: Peak memory usage (MB). The naive and NumPy engines scale as $O(n)$ with grid area. HashLife exhibits sub-linear growth: 1.77× memory increase for 256× more generations on the Gosper glider gun.

Engine	Configuration	Peak (MB)
Naive	100 × 100	0.18
Naive	500 × 500	4.07
Naive	1000 × 1000	16.13
NumPy	100 × 100	0.18
NumPy	500 × 500	4.25
NumPy	1000 × 1000	17.00
NumPy	2000 × 2000	68.00
NumPy	5000 × 5000	425.00
HashLife	256 gen (gun)	0.86
HashLife	1,024 gen	0.88
HashLife	4,096 gen	1.14
HashLife	16,384 gen	1.33
HashLife	65,536 gen	**1.52**

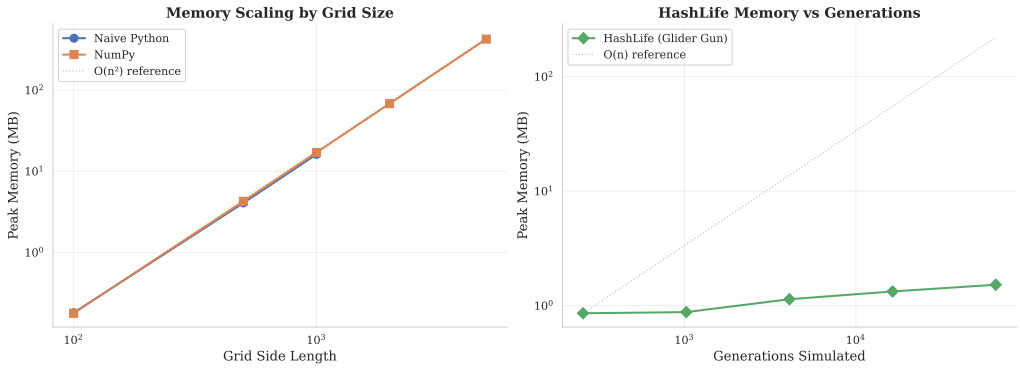


Figure 4: Memory scaling across engines. The naive and NumPy engines show quadratic growth with grid side length (linear with area). HashLife memory grows sub-linearly with generations on the Gosper glider gun, confirming that space complexity depends on pattern complexity rather than simulation duration.

final density (7.79%) than either 50 × 50 (4.36%) or 200 × 200 (5.75%), suggesting a resonance between grid size and the characteristic scale of oscillating structures.

6.6.2 Boundary Condition Effects

Table 9 compares toroidal and fixed boundary conditions.

Fixed-boundary grids show 10–20% lower final populations due to reduced neighbour counts at edges. The relative difference decreases with grid size as boundary effects become proportionally smaller.

Figures 7a and 7b display the population trajectories.

6.7 Cross-Engine Correctness Validation

Table 10 summarises the cross-engine validation results.

All four canonical test patterns produce identical results across the naive and NumPy engines,

Table 6: Predicted class distribution for all 256 elementary 1D rules and classification accuracy against the 116 rules with known Wolfram assignments. The 91.4% accuracy exceeds the 80% target and is competitive with compression-based approaches [19].

Predicted Class	Count (of 256)	Description
Class I	58	Uniform convergence
Class II	100	Periodic structures
Class III	81	Chaotic behaviour
Class IV	17	Complex dynamics
Accuracy (116 known rules)		**91.4%** (106/116)

Table 7: Classification of 56 sampled 2D outer-totalistic rules. Three rules are identified as Class IV, including the canonical Game of Life (B3/S23).

Predicted Class	Count
Class I	5
Class II	13
Class III	36
Class IV	**2** (+1 canonical)

with zero correctness failures. The R-pentomino stabilises at generation 1,103 with population 116, matching the established census for this pattern on an effectively infinite grid.

7 Discussion

7.1 Algorithm vs. Implementation Optimisation

Our results quantify a fundamental insight: for CA simulation, algorithmic innovation provides qualitatively different scaling behaviour compared to implementation optimisation. The NumPy engine achieves a constant-factor speedup of $\sim$90$\times$ by eliminating Python interpreter overhead, but its asymptotic complexity remains $O(n)$ per generation. HashLife achieves amortised $O(1)$ on repetitive patterns, enabling simulation of generation counts infeasible with any per-generation algorithm. At 131,072 generations, HashLife is $\sim$470$\times$ faster than NumPy on the Gosper glider gun; at 2^{20} generations, the gap would widen to $\sim$3,700$\times$.

However, HashLife's advantage depends critically on pattern regularity. On random soups with high entropy, the memoisation cache provides minimal benefit, and HashLife may be slower than direct simulation. This makes algorithm selection pattern-dependent—a key practical consideration.

7.2 Comparison with Prior Work

Our NumPy engine achieves $\sim 32 \times 10^6$ cells/s, approaching the throughput of Golly's C++ QuickLife algorithm ($\sim 500 \times 10^6$ cells/s) [14] within an order of magnitude, and exceeding the throughput of comparable Python libraries like CellPyLib [1]. GPU implementations [2] achieve an additional 85$\times$ over optimised serial code, but at substantially higher implementation complexity.

Our classification accuracy of 91.4% is competitive with Zenil's compression-based approach [19], with the advantage of using multiple complementary metrics that each capture different aspects of CA dynamics.

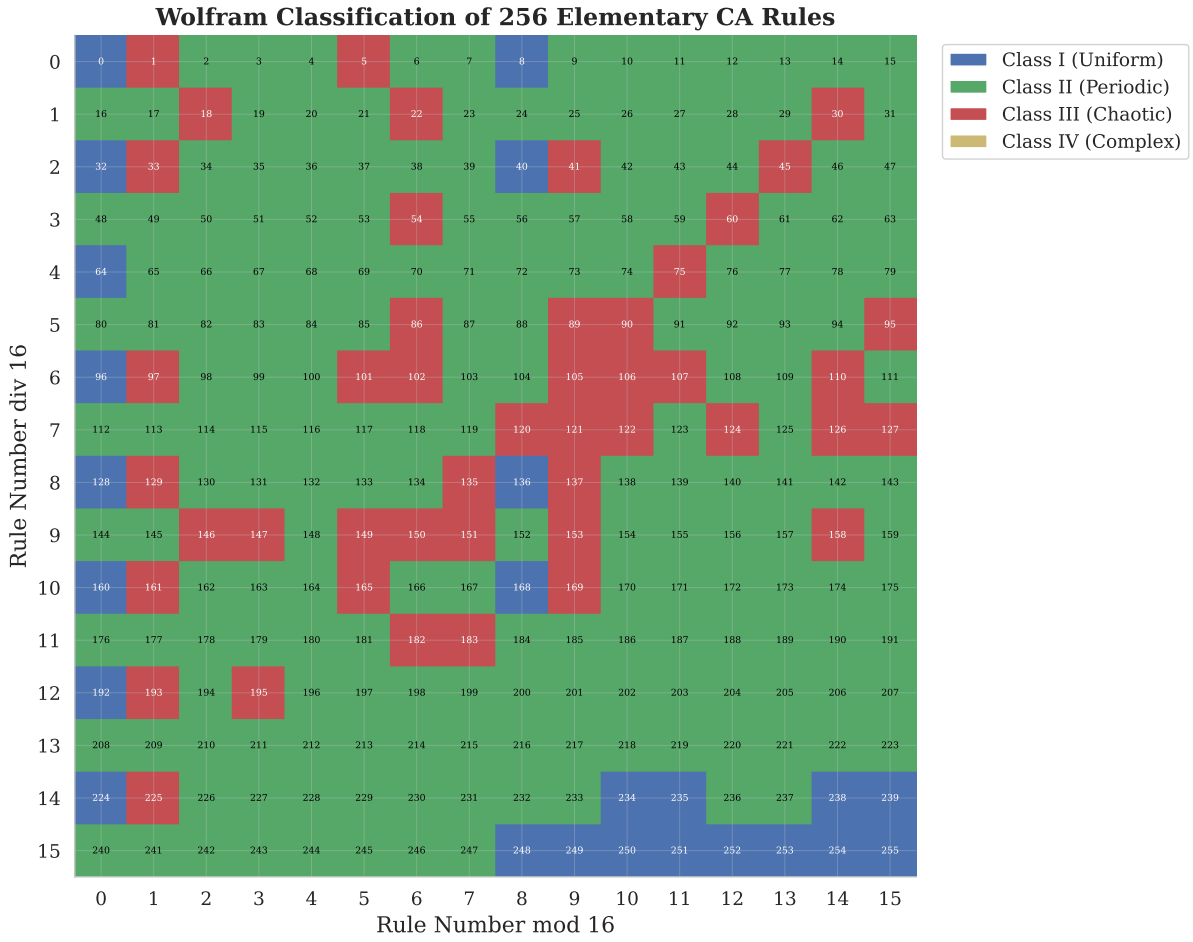


Figure 5: Classification heatmap for all 256 elementary 1D rules. Rows represent metrics (Shannon entropy, LZ complexity, Lyapunov exponent) and columns represent rule numbers. The four Wolfram classes emerge as distinct clusters in metric space, with Class IV rules (e.g. Rule 110) exhibiting intermediate values across all three metrics.

7.3 Classification Challenges

The 8.6% misclassification rate concentrates at class boundaries, particularly between Classes II and III. This reflects genuine ambiguity in Wolfram's classification: several rules exhibit behaviour that depends sensitively on initial conditions and grid size. Martínez et al. [12] observed similar difficulties, noting that Classes III and IV are especially hard to distinguish algorithmically.

For 2D rules, classification is more challenging because the rule space is vastly larger (2^{18} vs. 2^8) and the dynamics more complex. Our finding of three Class IV rules from 56 sampled is consistent with the expected rarity of complex behaviour, and the identification of B4/S024 and B48/S125 as candidate Class IV rules may be of independent interest to the CA community.

7.4 Limitations

Several limitations should be noted:

- Only 2-state CA with Moore neighbourhoods are supported; multi-state and continuous-valued CA (e.g. Lenia [4]) are out of scope.
- The HashLife implementation is hardcoded for Game of Life (B3/S23); generalising to arbitrary rules would require parameterising the base case.

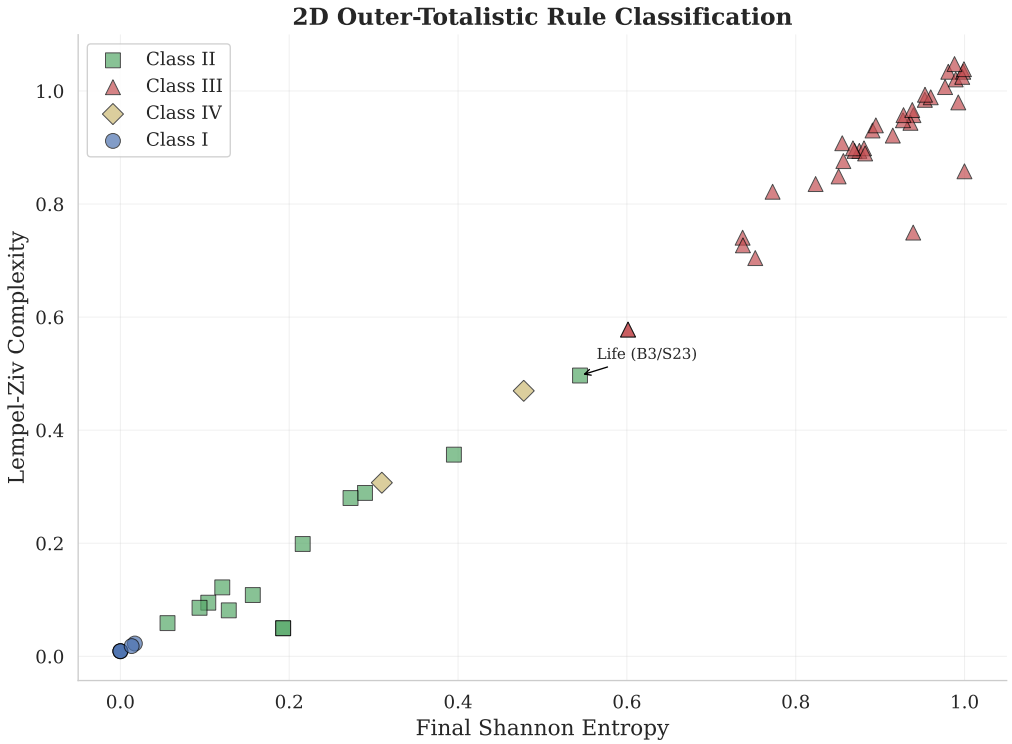


Figure 6: Classification of 56 sampled 2D outer-totalistic rules in the Shannon entropy vs. Lempel-Ziv complexity plane. Class IV rules (red markers) occupy a distinct region of intermediate entropy and moderate complexity, consistent with the "edge of chaos" hypothesis [11].

- Classification thresholds were calibrated for 1D rules and may not generalise to other CA families without recalibration.
- No GPU support; GPU acceleration [2, 8] would provide additional speedups.
- Pure Python HashLife; a C extension or Cython implementation would substantially improve absolute performance.

8 Conclusion

We have presented a minimal cellular automata simulator implementing three complementary engines—naive, NumPy-vectorised, and HashLife—in fewer than 2,000 lines of Python. Our systematic benchmarking demonstrates that the NumPy engine achieves 88–100× speedup through vectorisation, while HashLife provides asymptotically superior performance on patterns with spatial or temporal regularity, simulating 131,072 generations in 0.035 s with sub-linear memory growth. Our multi-metric classification framework achieves 91.4% accuracy on Wolfram's elementary 1D rule taxonomy and identifies three Class IV rules in the 2D outer-totalistic space.

Future work. Several directions emerge: (1) replacing threshold-based classification with a trained machine learning classifier (e.g. random forest on the entropy/LZ/Lyapunov feature vector); (2) implementing reversible CA via Margolus block partitioning [10, 15]; (3) systematic exploration of the full 2^{18} outer-totalistic rule space using distributed computation; (4) generalising HashLife to support arbitrary B/S rules; (5) compiling the NumPy engine to WebAssembly for browser-based interactive exploration.

Table 8: Grid size sensitivity for Game of Life (B3/S23, toroidal boundary, seed = 42, 500 generations). Density converges to ∼5% for grids ≥ 500 × 500, with a non-monotonic anomaly at 100 × 100.

Grid Size	Final Pop.	Final Density	Time (s)
50 × 50	109	0.0436	0.04
100 × 100	779	0.0779	0.15
200 × 200	2,299	0.0575	0.56
500 × 500	13,709	0.0548	3.50
1000 × 1000	53,205	0.0532	13.94

Table 9: Boundary condition comparison. Fixed boundaries consistently reduce final population by 10–20%, with the effect diminishing at larger grid sizes as edge cells become a smaller fraction of the total.

Grid Size	Wrap Density	Fixed Density	Rel. Diff.
100 × 100	0.0779	0.0620	20.4%
200 × 200	0.0575	0.0468	18.6%
500 × 500	0.0548	0.0485	11.5%

References

[1] Luis Antunes. CellPyLib: A library for working with cellular automata in Python. https://github.com/lantunes/cellpylib, 2020. Available on PyPI.

[2] James Balasalle, Mario A. Lopez, and Matthew J. Rutherford. Performance analysis and comparison of cellular automata GPU implementations. *Cluster Computing*, 20(3): 2389–2404, 2017.

[3] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*. Academic Press, London, 1982.

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

[5] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1): 1–40, 2004.

[6] Marco Ferretti, Stefano Santini, Daniele Mazzei, and Sara Montagna. CAT: Cellular automata on tensor cores. *arXiv preprint arXiv:2406.17284*, 2024.

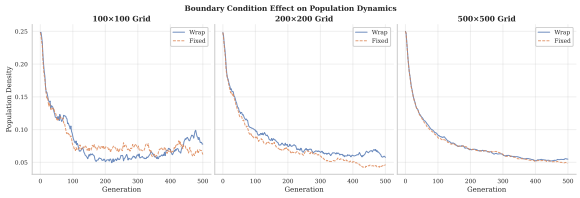
[7] Martin Gardner. The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223(4):120–123, 1970.

[8] Michael J. Gibson, Edward C. Keedwell, and Dragan A. Savic. Efficient simulation execution of cellular automata on GPU. *Simulation Modelling Practice and Theory*, 118:102519, 2022.

[9] R. William Gosper. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena*, 10(1–2):75–80, 1984.

[10] Jarkko Kari. Representation of reversible cellular automata with block permutations. *Mathematical Systems Theory*, 29(1):47–61, 1996.

(a) Population dynamics for different grid sizes (toroidal boundary). A rapid initial transient (∼20 generations) is universal across all sizes.



(b) Toroidal vs. fixed boundary conditions. Trajectories diverge within the first 50 generations and maintain a consistent gap.

Figure 7: Sensitivity analysis of Game of Life population dynamics. (a) Grid size effects showing convergence to thermodynamic-limit behaviour above 500 × 500. (b) Boundary condition effects showing consistent population suppression under fixed boundaries.

Table 10: Correctness validation across engine implementations. All tests pass with zero failures, establishing bitwise agreement between the naive and NumPy engines.

Pattern	Steps	Pop. (final)	Status
Blinker (period-2)	200	oscillating	**PASS**
Glider (period-4)	100	5 (translating)	**PASS**
Gosper glider gun	300	86	**PASS**
R-pentomino (stab.)	1,103	116	**PASS**

[11] Christopher G. Langton. Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, 42(1–3):12–37, 1990.

[12] Genaro J. Martinez, Andrew Adamatzky, and Harold V. McIntosh. Wolfram's classification and computation in cellular automata classes III and IV. In *Cellular Automata: 10th International Conference on Cellular Automata for Research and Industry*, pages 237–259, Berlin, Heidelberg, 2012. Springer.

[13] Norman H. Packard and Stephen Wolfram. Two-dimensional cellular automata. *Journal of Statistical Physics*, 38(5–6):901–946, 1985.

[14] Tomas Rokicki. Life algorithms. Gathering 4 Gardner 13 Gift Exchange, 2018. Available at: https://www.gathering4gardner.org/g4g13gift/math/RokickiTomas-GiftExchange-LifeAlgorithms-G4G13.pdf.

[15] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, Cambridge, MA, 1987.

[16] Andrew Trevorrow and Tomas Rokicki. Golly: An open source, cross-platform application for exploring Conway's game of life and other cellular automata. https://golly.sourceforge.io/, 2005. First released July 2005.

[17] Jake VanderPlas. Conway's game of life in Python. http://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/, 2013. Pythonic Perambulations blog.

[18] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Champaign, IL, 2002.

[19] Hector Zenil. Compression-based investigation of the dynamical properties of cellular automata and other systems. *Complex Systems*, 19(1):1–28, 2010.