Growth patterns of 2-dimensional elementary cellular automata

Kian Siadat

Student ID: 2155638

BSc Computer Science, University of Birmingham

Supervisor: Jens Christian Claussen

20 September 2023

Abstract

In the 1-dmiensional elementary cellular automata, there are rules that produce population sequences that repeats a base sequence that multiplies by some scalar value at doubling intervals. This includes rule 90 and rule 22 among others. These types of patterns show complex behaviour, and in this project we will be searching for rules that produce the same behaviour in 2-dimensional cellular automata.

We will use 2-dimensional 2-state cellular automata with a Von Neumann neighbourhood. Specifically, we will look at the complete rulesets of totalistic, pure outer-totalistic, and a custom outer-totalistic variant to find what proportion of rules display this behaviour, as well as a sample of the complete cellular automata.

In this report we will learn about how cellular automata work, the specific behaviour we are looking for, the practical implementation, and finally the results.

Acknowledgements

I would like to thank my supervisor Jens Christian Claussen for all the help and advice he gave me for this project, and for having the initial idea for the project direction which turned out to be very interesting.

Table of contents

Introduction

Before looking at all the fine details of the project I will first explain what cellular automata are and how they work for any readers unfamiliar with the topic. I will also explain what behaviour we are looking for in relation to the research question and provide a brief outline on what the rest of the report will contain.

What are cellular automata?

The idea of cellular automata was first developed in the 1960s by John Von Neumann [1] and Stanislaw Ulam [2]. The basic idea is to have discrete cells that can interact with each other using basic rules, and to create some emergent complex or ordered behaviour. Von Neumann focused largely on how it could be applied to self-replicating machines, whereas Ulam focused on the relation to crystal growth.

Later in 1970, John Conway would create the game of life [3], a 2-dimensional cellular automata that is able to generate complex behaviour from simple rules. It is capable of generating self-replicating structures and can even perform universal computation [5].

The idea was further developed by Stephen Wolfram [4], who performed an analysis of the entire 'elementary cellular automata'. He also categorised the behaviour in to 4 classes:

- Class 1 quickly converges to uniform behaviour.
- Class 2 quickly converges to non-uniform but stable behaviour.
- Class 3 shows random behaviour.
- Class 4 develops structures that interact with each other in complex ways indefinitely.

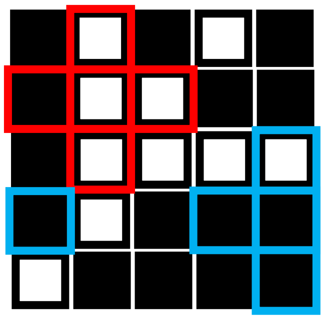
Stephen wolfram's classification of cellular automata is subjective to an extent and relies on visual analysis. As the field has grown, more recent research has focused largely on the mathematical properties of cellular automata. For example, classifying cellular automata using power spectra [6], or relating different cellular automata using differential equations [7].

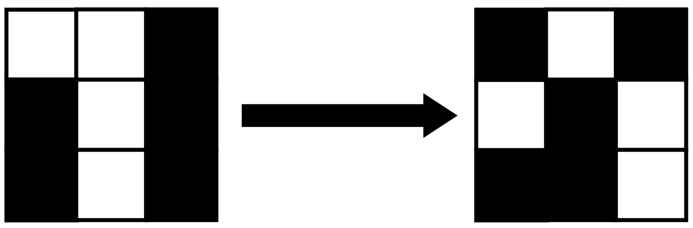
For this project we are using 2 dimensional cellular automata, so each cellular automata consists of a 2D grid where each location in the grid is a cell. There can be other types of cellular automata though, such as 1D which would be an array of cells, 3D, 4D and so on. Each cell holds a single state value, here we are looking at the simplest case in this regard so each cell can be 0 or 1. This is often referred to as dead or alive respectively.

For each simulation we start with our initialised grid, then at each timestep we update the state of each cell based on its own state and the states of its neighbours. This is the part where complex patterns can emerge from simple rules. It is also important that the cells update in parallel, this ensures that the order the cells update in does not affect the outcome of the simulation. It also means that cells are only ever influenced by values from the previous timestep, keeping each timestep completely distinct which allows us to analyse the changes.

We can choose to update the cells in any way, but we will keep it consistent across all timesteps in a simulation, and consistent across all cells in the grid. We can use multiple different rules in separate simulations and then analyse and compare the differences in the results they produce.

The neighbourhood of cell can be anything we choose, for 2D cellular automata Moore neighbourhoods are common (which includes the centre cell and all 8 cells around it), however for this project we will use a Von Neumann neighbourhood (which includes the centre cell and the 4 cells directly adjacent). Also, for dealing with the edges of the grid we will simply allow the neighbourhoods of edge cells to wrap around to the other side of the grid.





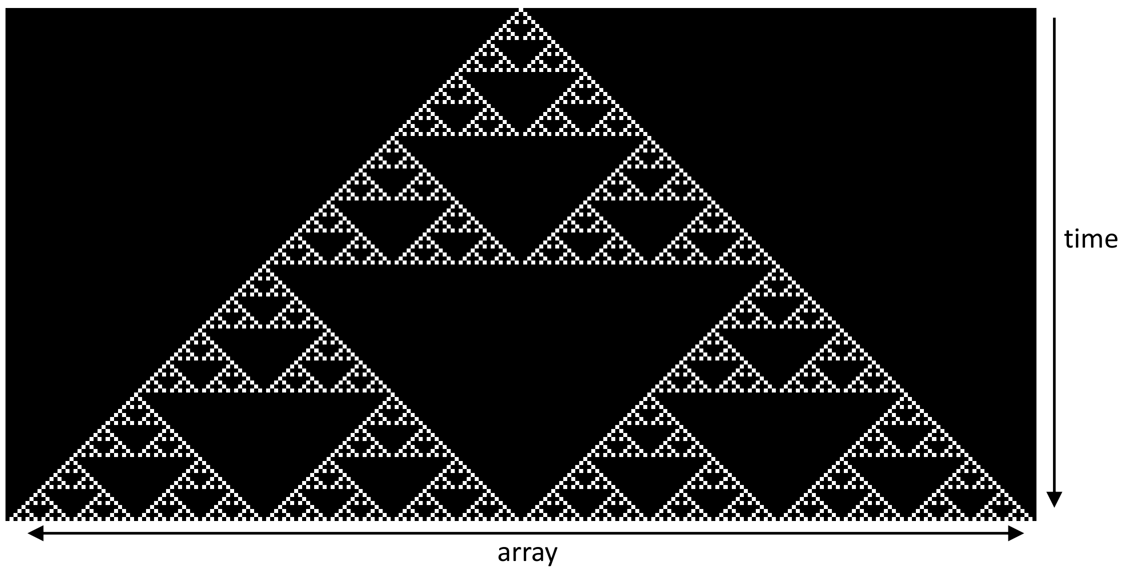
Note that these grids are very small, and we will use a larger grid in practice.

Also note that for these and all further diagrams I will use white to represent 1, and black to represent 0.

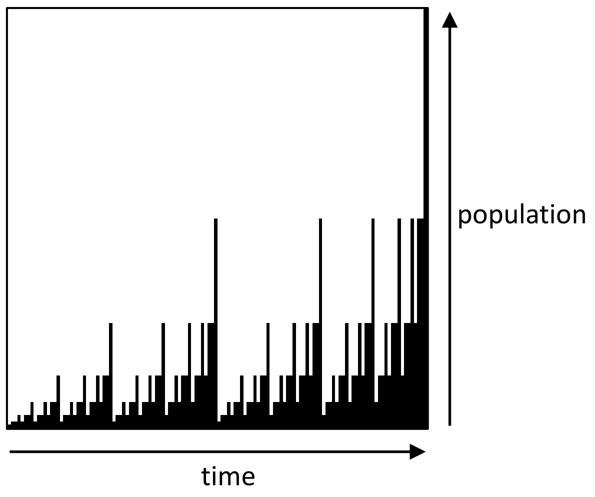
Given the grid at any particular timestep, we can say that the population at that timestep is the total number of live cells. Since live cells have a state of 1, and dead cells have a state of 0, the population is just the sum of the states of all cells. If we take the population across all timesteps of a simulation we get a population sequence, and we can use that population sequence to analyse the rule that we used to produce it.

What behaviour are we looking for?

In the 1-dimensional cellular automata, instead of using a grid, we instead use an array of cells where the neighbourhood includes the centre cell and its left and right neighbours. One of the rules referred to as 'rule 90' [8] produces a very interesting result. This rule produces a 1 in any cell if exactly 1 of its neighbours (ignoring itself) has a state of 1. If you take the array at each timestep and place it below the previous configurations to form an image you get a Sierpinski pattern.



If we look at the population sequence produced by rule 90 we get this result:
[1, 2, 2, 4, 2, 4, 4, 8, 2, 4, 4, 8, 4, 8, 8, 16, 2, 4, 4, 8, 4, 8, 8, 16, 4, 8, 8, 16, 8, 16, 16, 32, …]
If we take the first value from this sequence and multiply it by 2 we get the next value.
Then we can take the first 2 values, and multiply by 2 to get the next 2 values.
We can also do the same with the first 4, 8, 16 values, and so on for ever.



Bringing it back to the research question, what we want to know is what proportion of the 2D cellular automata update rules produce population sequences that have the same repetition property as the rule 90 population sequence. We will also compare different subsets (or variations) of rules to see what difference it makes. We will also include rules that show this behaviour for starting sequences other than 1, for example the pattern may only apply using a starting sequence of 4, which would still be valid.

Report outline

First, we will examine the different versions of cellular automata including complete, totalistic, and outer-totalistic varieties and why doing a comprehensive analysis on all possible rules is infeasible for this project. We will also see why I made certain decisions for the cellular automata parameters.

Then we will see the implementation of the project including the general structure of the code, databases, and the generated research data.

Then we will analyse the results. This will include what proportion of rules fit the type of pattern we are looking for, as well as categorising the patterns and seeing how the categories differ among different rulesets.

The final section will include a discussion about the successes and problems faced in the project, and a brief conclusion to summarise the results.

After the main section are the appendices which contains extra information. This includes research data that couldn't be fit in to the main report without making it difficult to read, and instructions on how to run the code.

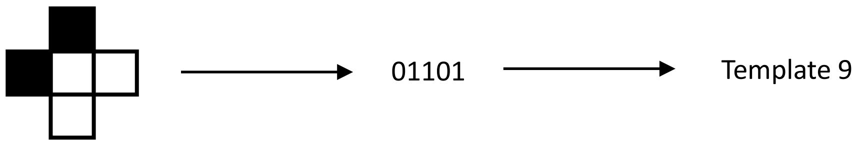
Problem analysis

Complete cellular automata

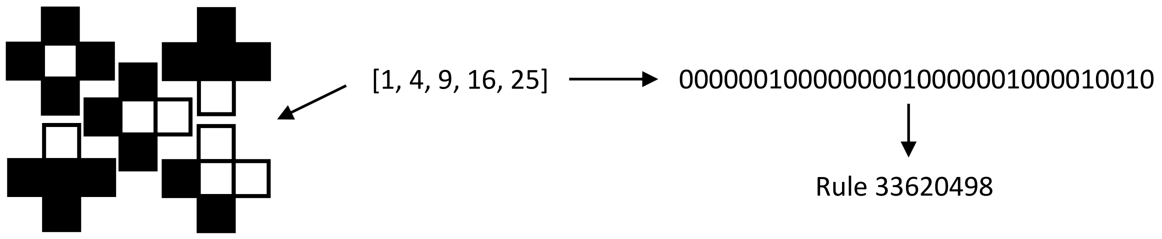
For each cell, its neighbourhood consists of 5 cells including itself, and these 5 inputs are used to determine the new state. Since each cell has 2 possible states there are $2^5 = 32$ possible input states, I will refer to these as 'templates' to keep the text concise but be aware that other literature does not do this.

Now we can define the update rule as a mapping from templates to states; essentially, we assign a 0 or 1 to each template to say what output it produces. Now at each update step we can look at each cell and see which template its neighbourhood matches and assign its new value based on our mapping (aka our rule).

We can represent the template as a bitstring where each bit is a state, which then allows us to name the template based on the numerical value that this bitstring evaluates to. For this project I format templates as [north, east, south, west centre], so for example:



We can use the same naming scheme for rules as well. We simply represent each mapping as a single bit in the bitstring and find the numerical value. For example, a rule which only maps templates [1, 4, 9, 16, 25] to a value of 1 would convert as follows:



It is of course possible to do the reverse as well, given a rule as an integer, we can calculate the bitstring and use that to find what the template mappings are.

Since there are 32 templates in our cellular automata, each rule is a 32 length bitstring, so there are $2^{32} = 4294967296$ possible rules (approximately 4.3 billion) in the complete cellular automata. This is far too many rules to simulate within the timescale of this project which is why we will focus on ways of reducing these rules to a smaller subset.

Totalistic cellular automata

Before understanding outer-totalistic cellular automata, we first need to understand totalistic cellular automata, a class of rules first formally proposed by Wolfram in 1983 [9].

Instead of having a mapping for every template like in complete cellular automata, totalistic cellular automata have a mapping for each possible neighbourhood sum. So, the updated value for any cell is dependent only on the sum of the states of cells in its neighbourhood (including itself). For a Von Neumann neighbourhood there are 6 possible sums from 0 to 5, this gives us $2^6 = 64$ rules.

These rules are a subset of the complete cellular automata rules. Having a subset is good because it allows us to do some analysis without having to simulate 4 billion rules. However, 64 is too few for the scope of this project, so although we will use them, we will focus more on outer-totalistic cellular automata.

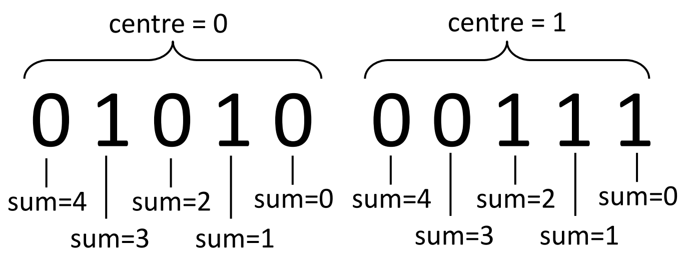
The formatting is similar to the complete cellular automata. For example, a rule that produces a 1 only when the sum is [0, 1, 3] would have a bitstring of 001011 and would be called rule 11.

Outer-totalistic cellular automata

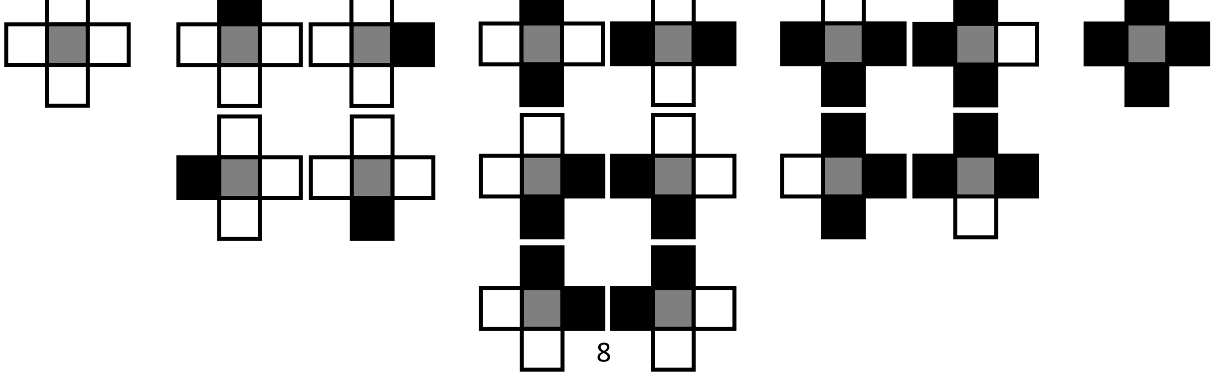
Outer-totalistic cellular automata are similar to totalistic versions, except that the update depends on some combination of the centre cell's state, and the sum of the states of its neighbours (excluding itself). This gives us 5 possible sums from 0 to 4, and for each of these the centre cell can take 2 possible values, which gives us $2^{2*5} = 2^{10} = 1024$ rules. This is a sufficiently large number of rules for this project but is still a feasible amount to simulate which is why we will focus on outer-totalistic cellular automata.

Totalistic cellular automata rules are a subset of outer-totalistic cellular automata rules, which are in turn a subset of complete cellular automata rules. This means that by focusing mostly on outer-totalistic rules we can reduce the number of rules we have to simulate to a manageable amount, and all the totalistic rules will be included in this set.

The formatting I used for outer-totalistic rules is slightly different to the previous examples.



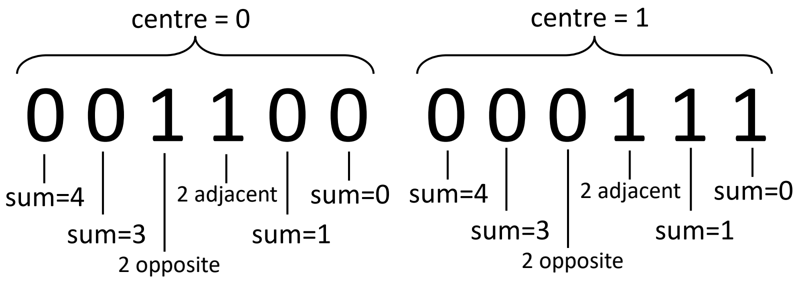
Looking at each possible sum there is more variation in the possible inputs for when the sum is 2, than for every other value.



To be precise, each variation for the other sums is the same template, just rotated. But when the sum equals 2, the 2 cells that have a value of 1 can be either opposite or adjacent to each other. This allows us to create our own custom version of the outer-totalistic cellular automata where we distinguish the cases where the sum equals 2 as either opposite or adjacent.

Why would we do this though? Since the aim is to find out what proportion of rules generate the type of population sequence we are looking for, we will be able to find out whether the custom outer-totalistic ruleset has a different proportion to the pure outer-totalistic ruleset. This will give information about which template mappings are more favourable to producing the results we are looking for.

In terms of quantity, we now have $2^{2*6} = 2^{12} = 4096$ rules. This is larger than the pure outer-totalistic rules but still manageable. I chose to format them in the following way:



Design and parameter choices

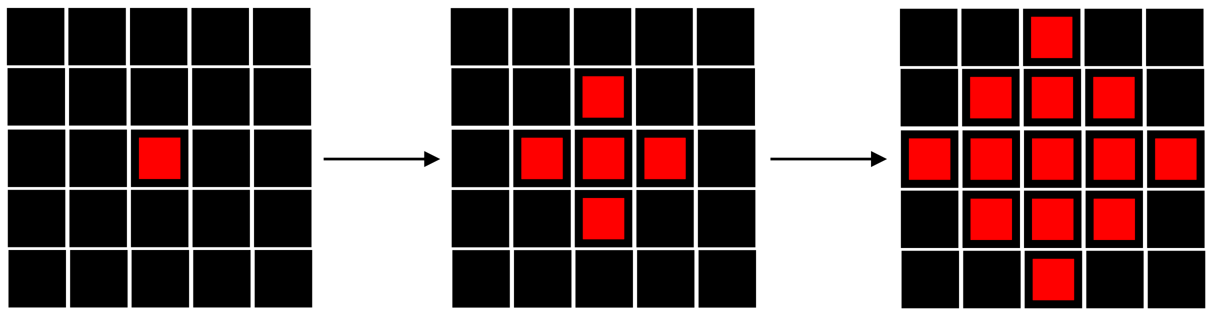
Why are we focusing on 2D cellular automata? 1-dimensional cellular automata have been thoroughly examined and there isn't anything for me to contribute. 2 and higher dimensions of cellular automata have not been comprehensively researched and so there is still work that I can do. The reason to choose 2D is that as you add dimensions, the number of rules increases exponentially and quickly becomes infeasible. 2D is the lowest dimension cellular automata that has not been fully explored, and it is also a lot easier for people to visualise than 3D, 4D etc.

The reason to use a Von Neumann neighbourhood is similar. I could have chosen to use Moore neighbourhood which includes the centre cell and its 8 neighbours however increasing the neighbourhood size like this also causes exponential blow in the number of rules. A Moore neighbourhood would have $2^{10} = 1024$ totalistic rules, $2^{2*9} = 2^{18} = 262144$ outer-totalistic rules, and 2^{512} complete rules which is approximately 10^{154}. With a Moore neighbourhood I would only be able to fully explore a totalistic ruleset, whereas with a Von Neumann neighbourhood I can fully explore a totalistic, outer-totalistic, and an outer-totalistic variant ruleset.

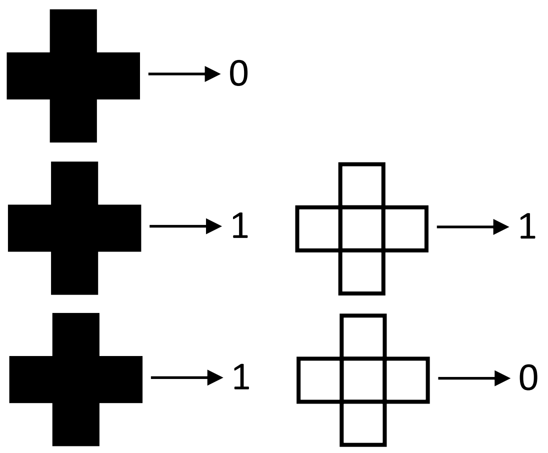
For the simulation I will initialise the grid with all cells having a state of 0, except for the centre cell which will have a value of 1. There are multiple reasons for this. Firstly, using the 1-dimensional rule 90 only produces the type of population sequence we are looking for when starting with a single cell that has a value of 1. When trying to replicate the same behaviour we should provide the same environment. Just having a single cell with a state of 1 will also provide the least chaotic data and will be easier to analyse. Furthermore, using the same initial grid configuration each time means that every run of the simulation with the same rule, will always produce the same population sequence.

Looking at the population sequence from rule 90, we can see that it peaks at timesteps [4, 8, 16, 32, …]. I decided to set each simulation to run for 33 timesteps. This should be long enough so that it's clear if the population sequence matches the pattern, but short enough so as not to waste computing time.

At the first timestep, the only cell influenced by the centre cell is the centre cell. At each successive timestep this grows to the cell's neighbours, and then their neighbours and so on. This means that the size of the complex section of the grid grows by 1 in each direction with each timestep. 32 timesteps beyond the first means (2*32) + 1 = 65, so an area 65 wide and tall. This is important because we want to avoid the pattern growing to edge of the grid, then interfering with itself via the neighbourhoods that join each edge. Knowing this, I decided to give the grid dimensions of 67 wide and tall to provide enough space for the pattern to grow while leaving 1 layer of border cells around it in the last timestep.



It is also worth noting that no complex behaviour can generate from the rest of the grid. Since all cells outside the influence of the centre cell start as 0, 1 of 3 things will happen:



Background will remain all 0s.

Background will all flip to 1s, then remain that way.

Background will all flip to 1s, then flip back to all 0s and repeat this forever.

In regard to the cellular automata variations, totalistic is a subset of pure outer-totalistic, which is a subset of custom outer-totalistic, which is a subset of the complete cellular automata. There are too many rules in the complete set, which is why I will only cover the other sets in full. However, I will take a random sample of complete rules for comparison. The reason to cover all the different rulesets is to see if there is any difference in the proportion of rules that show the behaviour we are looking for.

Implementation

I decided to use python for the implementation. This is because python is the language that I am most familiar with, and I didn't want to waste time getting stuck on the specifics of other languages.

I also decided to go with a functional programming approach. The main reason for this is that using an object oriented approach just wasn't really necessary. The total amount of code is not too large, and the high amount of modularity from a functional approach is very beneficial for this project.

In terms of external libraries, I used math and random for general coding utility, matplotlib.image and os for creating and reading files, and sqlite3 for database functionality.

Code file structure

There are only 4 code files for this project. template.py, simulation.py, outer_simulation.py, and outer_db.py.

template.py contains the most basic functions. This includes functions to convert between integer and bitstring for rules and templates, a function to plot a graph of a population sequence, and a function to select a random non-repeating sample of rules. These are all functions which don't so much on their own but exist to be imported and used in the other files.

simulation.py contains all the functions required to perform a complete simulation of a complete rule. This file has 2 main functions. One of them is an evaluator that takes a rule and neighbourhood and returns the new cell state. The other is a simulator which runs the simulation as a whole and makes calls to the evaluator function as part of that. The simulator also has a flag which if set to true causes it to generate image files of the grid at each timestep of the simulation.

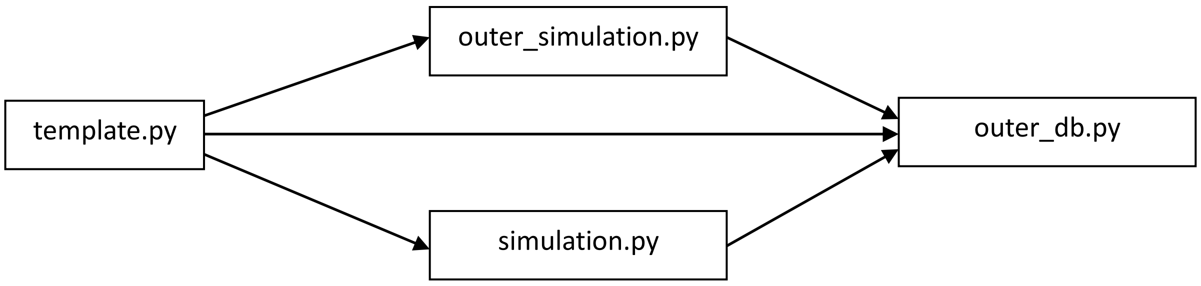
outer_simulation.py contains all the functions required to perform a complete simulation of a totalistic, or outer-totalistic rule. This works similarly to simulation.py where we split the code into simulator and evaluator. However, this time there is 1 simulator and 3 evaluators (for totalistic, pure outer-totalistic, and custom outer-totalistic). This way the simulator takes the evaluator it will use as an argument, and then makes calls to it. This modularity makes it very easy to add a new ruleset by adding a new evaluator and passing it as an argument.

The reason that simulation.py and outer_simulation.py were not merged was to do with the way the rules are represented in the code. In simulation.py the rule is converted to a list of all the templates that map to 1 (each template is an array of bits), in the evaluator each template is compared to the neighbourhood to see if there is a match. However, in outer_simulation.py the rule is stored as an array of bits, in the evaluator the sum is calculated and used with the rule in a logic statement to determine the result. Because of the nature of the complete cellular automata compared to all the other varieties it was just easier to code it this way. It would have been possible to combine them in to 1 simulator with 3 evaluators, however it just wasn't a priority and would have taken too much effort to change something that already works.

outer_db.py is the highest level file. It contains everything that involves interacting with the databases. Since all of the main functions of the code involve reading from or writing to a database at some point, most of the functions that a user might want to directly call are in this file. It does import functions from the other files though to be used. It contains several functions for database manipulation. But the main functions the user might call are the following:

- Simulate a ruleset and save the population sequence to a database.
- Perform filtering on the population sequences to find which ones fit the pattern.
- Get sum bitstrings of a group of rules.
- Get the population sequences of a group of rules.

Below is an illustration of the general structure where arrows represent imported functions.



There is also some redundant code that was mean for purposes that were eventually dropped, although this is a small proportion of the overall code and mostly exists within template.py.

Database structure

Each ruleset has its own database, and each database contains 3 tables:

pre_sequence contains the population sequence data for each rule.

growth contains a rule number column, and 3 other columns: up, flat, and fall. These record whether a sequence increases, remains the same, or drops across any timestep. The purpose of this is that if you want to manually look through the population graphs to find interesting behaviour you want to filter out graphs that only grow, or only remain the same, or only decrease. We can achieve this by only looking at population sequences that increase and decrease at some point. Adding the flat column is unnecessary but it was trivial to add to the code, and at the time I was unsure if it would provide any value or not.

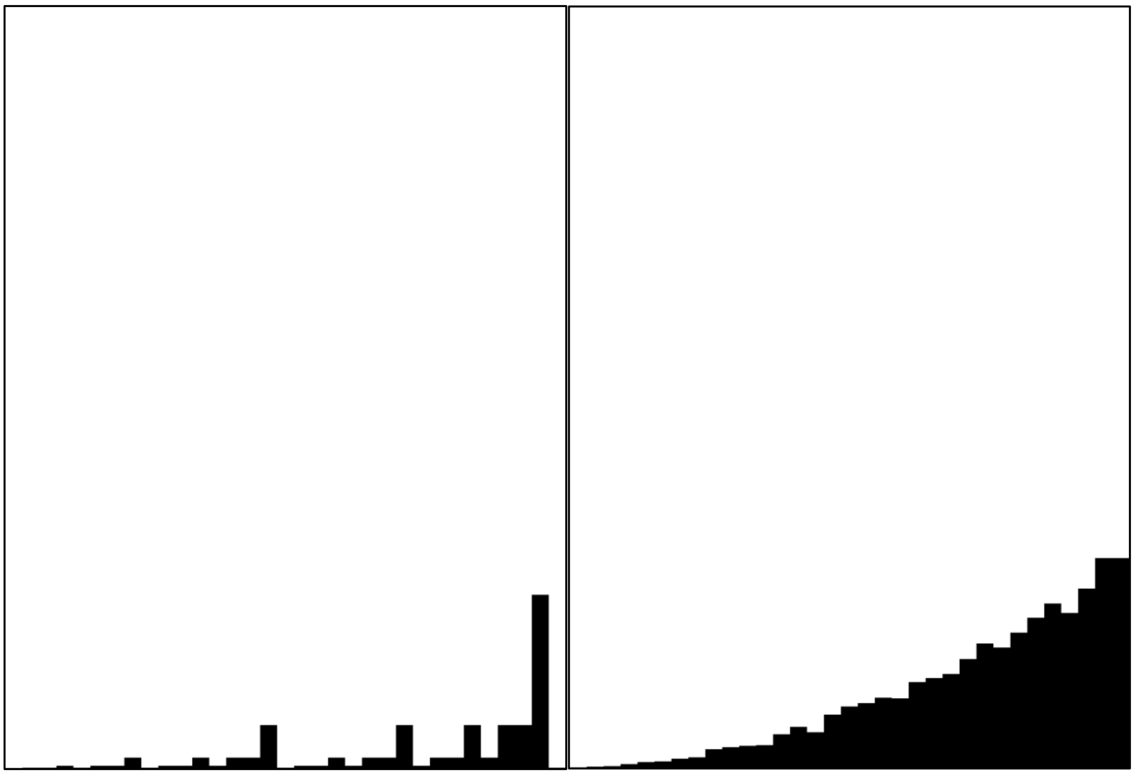
filter contains a few columns for features that were never implemented. The columns that are relevant though are rule number, auto, base, and scale. Auto represents whether the population sequence fits the pattern, base and scale contain information about the parameters of it. Base means the length of the starting sequence of the pattern, and scale means how much it multiplies by at each step. For example, a rule with auto=1, base=4 and scale=2 would in its sequence have its second 4 values be equal to 2 times the first 4, the second 8 values equal to 2 times the first 8, and so on.

Functionality

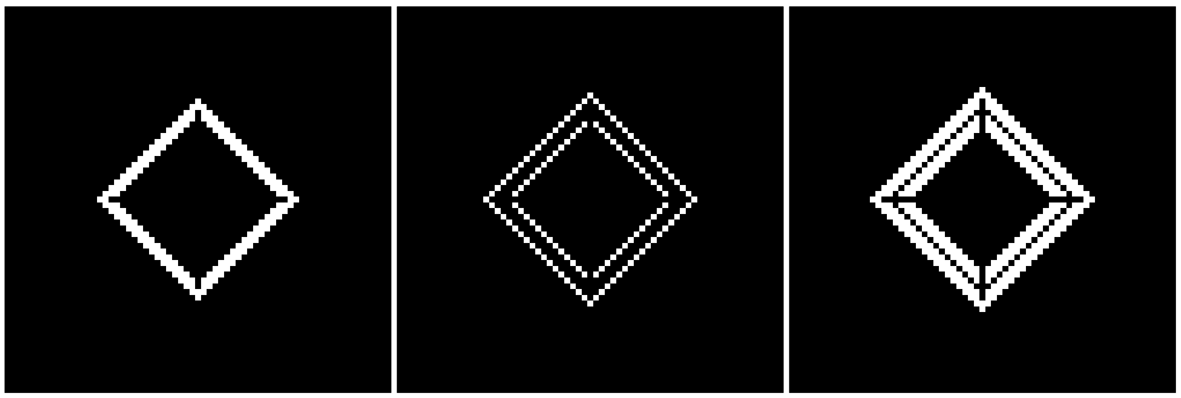
In this section I will list the major functionalities of the code.

- Simulating an individual rule and returning the population sequence. This includes the option to generate image files of the grid at each timestep in the simulation.
- Simulating a set of rules and storing the population sequence data and the growth data.
- Generating graphs for a single/set of sequences in the database.
- Filtering sequences to find which ones fit the pattern as well as their base and scale.
- Calculating the sum of bitstrings of a set of rules.
- Outputting the population sequences of a set of rules.

Examples of population graphs (totalistic rule 64, and pure outer-totalistic rule 205):



Examples of grid data (from custom outer totalistic rule 385):



Results

How to interpret the data

Every population sequence has been run through a filter that tells us if it matches the pattern. It also gives us 2 properties: base, and scale. The base refers to the length of the starting sequence, and the scale refers to how much the sequence multiplies by at each step.

For example, the sequence from the Sierpinski pattern has base=1, scale=2. Meaning that starting with the first value, double it to get the second, then double those to get the next 2 and so on. While a base of 2, 4, etc also fits, the filter returns the lowest valid base.

In the general case base=n, scale=m, means that for all i>=1, the first n^i values of the population sequence when multiplied by m gives the following n^i values.

There are 4 main cases for population sequences:

- The sequence does not match at all. We can ignore these rules.
- Base=1, scale=1. These rules produce no change whatsoever ever in the simulation. Their population sequence is [1, 1, 1, 1, 1, …].
- Base>1, scale=1. These are periodic patterns that may be stationary or mobile, where the length of the period is the base. An example sequence would be [1, 2, 3, 1, 2, 3, …] which has a base of 3. This is a type of complex behaviour although it's not the type we are looking for in this project.
- Scale>1. These are the rules that we are looking for. The base could be any positive integer, but since the simulation only runs for 33 timesteps, the largest we can find is base=16.

Totalistic ruleset

For the totalistic ruleset, there are 2 rules that produce a growing repeating pattern, 2/64 = 3.125% of the overall ruleset. Specifically rules 2 and 10, which both have a base of 2, and scale of 4. They also share the same population sequence:
[1, 5, 4, 20, 4, 20, 16, 80, 4, 20, 16, 80, 16, 80, 64, 320, 4, 20, 16, 80, 16, 80, 64, 320, 16, 80, 64, 320, 64, 320, 256, 1280, 4]

Rule 2 has bitstring [0, 0, 0, 0, 1, 0] which means that a cell's state will only become 1 if the neighbourhood sum is also 1.
Rule 10 has bitstring [0, 0, 1, 0, 1, 0] which means that a cell's state will only become 1 if the neighbourhood sum is 1 or 3. The case where the sum is 3 never actually occurs given our stating configuration, which means that both rules are functioning exactly the same.

We can also analyse the rules by summing the bitstrings. By doing so we get:
[0, 0, 1, 0, 2, 0]
Since there are only 2 rules, values of 0 means both have 0 in this bit, and that to produce this sequence requires those bits to be 0. The same applies the value of 2 but in reverse, and a value of 1 means that bit doesn't matter.
We can use this information to create a generalised rule to produce a repeating sequence:

- New cell state = 0, if neighbourhood sum is in [0, 2, 4, 5]

- New cell state = 1, if neighbourhood sum is 1

You can check the grid data in appendix C to see how the pattern develops.

Pure outer-totalistic ruleset

For this ruleset there are 40 rules that produced growing repeating patterns. 40/1024 = 3.9%. Each of these rules have a scale of 4. We can further break this down in to 2 groups, those which have a base of 1, and those which have a base of 2. The 2 groups also have 2 distinct patterns, and 2 distinct population sequences that are consistent within each group:
Base=1: [1, 4, 4, 16, 4, 16, 16, 64, 4, 16, 16, 64, 16, 64, 64, 256, 4, 16, 16, 64, 16, 64, 64, 256, 16, 64, 64, 256, 64, 256, 256, 1024, 4]
Base=2: [1, 5, 4, 20, 4, 20, 16, 80, 4, 20, 16, 80, 16, 80, 64, 320, 4, 20, 16, 80, 16, 80, 64, 320, 16, 80, 64, 320, 64, 320, 256, 1280, 4]

There are 32 rules that have base=1, and 8 rules that have base=2. Looking at the bit sums we get:
Base=1: [0, 16, 0, 32, 0, 16, 16, 16, 16, 0]
Base=2: [0, 4, 0, 8, 0, 0, 4, 4, 0, 8]

For base=1, there are 5 bits with a value of 16. Each can take 2 possible values 0 or 1, and $2^5 = 32$. Since there are 32 rules in this group that means each combination of these bits is present within the rules in this group, which means that those bits must be irrelevant to the pattern and the sequence. For base=2 we can apply the same logic, 3 irrelevant bits, and $2^3 = 8$ which is why there are 8 rules in this group.

We can again use this information to simplify the logic.

For base = 1 patterns:

- New cell state = 0 if:
    - Current state = 0 and neighbour’s sum is in [0, 2, 4]
    - Current state = 1 and neighbour’s sum is 0
- New cell state = 1 if:
    - Current state = 0 and neighbour’s sum is 1

For base = 2 patterns:

- New cell state = 0 if:
    - Current state = 0 and neighbour’s sum is in [0, 2, 4]
    - Current state = 1 and neighbour’s sum is in [1, 4]
- New cell state = 1 if:
    - Current state = 0 and neighbour’s sum is 1
    - Current state = 1 and neighbour’s sum is 0

To see the grid data for these patterns, refer to appendix B for base=1, and appendix C for base=2.

Custom outer-totalistic ruleset

For this ruleset there are 160 rules that fit our pattern. Proportionally this is 160/4096 = 3.9% which is the same proportion as the pure outer-totalistic ruleset. And in much the same way we can split them by population sequence in to the same 2 groups which each have a scale of 4 again (because the population sequences are the same):
Base=1: [1, 4, 4, 16, 4, 16, 16, 64, 4, 16, 16, 64, 16, 64, 64, 256, 4, 16, 16, 64, 16, 64, 64, 256, 16, 64, 64, 256, 64, 256, 256, 1024, 4]
Base=2: [1, 5, 4, 20, 4, 20, 16, 80, 4, 20, 16, 80, 16, 80, 64, 320, 4, 20, 16, 80, 16, 80, 64, 320, 16, 80, 64, 320, 64, 320, 256, 1280, 4]

In fact, the proportional split between the 2 groups is the same. There are now 128 rules where base=1 and 32 rules where base is 2, which are both 4 times the size of the groups in the pure outer-totalistic ruleset.

Looking at the bit sums explains why:
Base=1: [0, 64, **64**, 0, 128, 0, 64, 64, **64**, 64, 64, 0]
Base=2: [0, 16, **16**, 0, 32, 0, 0, 16, **16**, 16, 0, 32]

Base=1 has 7 irrelevant bits, 2^7 = 128. Base=2 has 5 irrelevant bits, 2^5 = 32. But why are they structured like this? Looking at the values we can see that the bit that represents when the sum is 2 and the inputs are opposite, is always irrelevant. In fact, if we removed that bit we would get exactly the same rules as the pure outer-totalistic group. This means that we have just added an extra 2 irrelevant bits to the rules, and 2^2 = 4, so there are 4 times as many rules total, and 4 times as many that fit the pattern which leads to the proportion being the same. This also tells us that given our starting configuration, in the pure outer-totalistic group the case where 2 neighbour cells are opposite didn't come up in any of the rules that fit the pattern.

Knowing this I'm not going to write out the if statement that represents the logic of these rules, since it will be essentially the same as the pure outer-totalistic rules and you can refer to that.

Complete ruleset

For the complete ruleset, I took a random sample of the rules. There are about 4 billion in total, and I sampled 4096 which is a very small proportion. The sample size is not enough to conclusively say what proportion of rules fit the pattern, but we can look at the different sub-groups of rules to see what patterns arise.

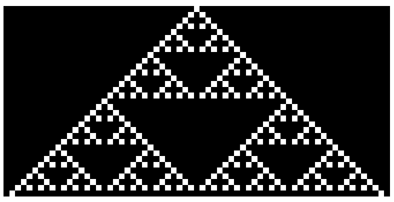
There are 176 rules that fit the pattern. 176/4096 = 4.3%. While we don't know for certain what the overall proportion would be, for this sample it is similar to the outer totalistic rules. The major difference is in the categories of patterns we get. While the totalistic and outer-totalistic rulesets all generated rules with a scale of 4, for the complete ruleset we have rules with scales of 2, 3, and 4.

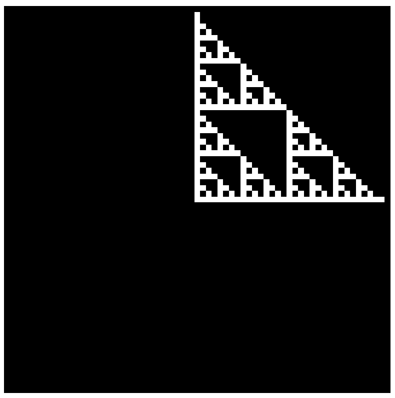
168 of the 176 rules have a scale of 2, and we can split them in to 2 groups based on population sequence:
Base=1 (153 rules): [1, 2, 2, 4, 2, 4, 4, 8, 2, 4, 4, 8, 4, 8, 8, 16, 2, 4, 4, 8, 4, 8, 8, 16, 4, 8, 8, 16, 8, 16, 16, 32, 2]

Base=2: (15 rules): [1, 3, 2, 6, 2, 6, 4, 12, 2, 6, 4, 12, 4, 12, 8, 24, 2, 6, 4, 12, 4, 12, 8, 24, 4, 12, 8, 24, 8, 24, 16, 48, 2]

We can see that for base=1, the population sequence is the same as that of 1-dimensional rule 90. Looking at the grid data (in appendix D) we can see that it's because it is just a 1-dimensional rule existing in a 2-dimensional space. There is also a variant that expands in 2 adjacent directions, this wouldn't be possible in a 1-dimensional cellular automata but it does create the same population sequence since they are both developing in the same way in each direction that they grow, it's just in 2 opposite directions, or 2 adjacent directions. You can see an example in appendix E.

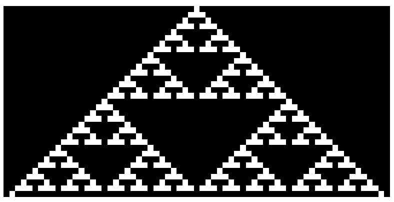
We can layer the grid data from the rules that expand in opposite directions to get a Sierpinski pattern, and we can overlay in place the grid data from rules that expand in adjacent directions to generate something very similar (which corresponds to rule 60 [8]):

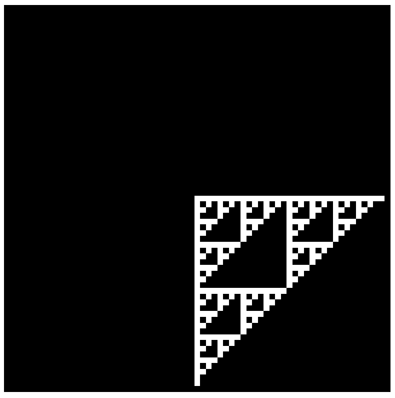




For base=2, the pattern is very similar. The difference between them and the group with base=1 is actually the same as for the outer-totalistic rules. Being that a cell with a state of 1, whose neighbours have a sum of 0 produces a 1. For the base=1 rules that case would produce a 0. You can see examples of this in appendices F and G.

The base=2 rules are also split in to rules that expand in opposite directions, and rules that expand in adjacent directions. We can do the same layering/overlay:





Here we can see that the adjacent expanding rules generate the same pattern as when the base is 1, but the opposite expanding rules generate something slightly different, which is the pattern generated by layering the 1-dimensional rule 22 [8], which tells us that these rules are functioning the same as rule 22. Rule 22 produces a 1 in a cell if the neighbourhood sum (including itself) is equal to 1 which makes it a totalistic rule.

3 of the sampled rules have a scale of 4. Each of these rules have a base of 1 and the same population sequence:
[1, 4, 4, 16, 4, 16, 16, 64, 4, 16, 16, 64, 16, 64, 64, 256, 4, 16, 16, 64, 16, 64, 64, 256, 16, 64, 64, 256, 64, 256, 256, 1024, 4]

The sequence matches the scale=4, base=1 groups from the outer-totalistic rules, and it matches because the patterns generated are the same. You can refer to appendix B to see the grid data. After looking at the scale=2 patterns, we can see that this is the same but expanding in 4 directions instead of 2, which explains the different scales. That also means that this pattern is a 2-dimensional version of the 1-dimensional rule 90. By looking at the grid data, we can see how layering it into a 3-dimensional object would give us a square based pyramid style Sierpinski-like pattern. It's not something I have the means or time to create so I will leave the visualisation to your imagination.

None of these rules matched the scale=4, base=2 outer-totalistic rules, but they must exist in the complete ruleset since the outer-totalistic rules are a subset of the complete rules. The reason we didn't find any must then be because of the relatively small sample size. But knowing that the rules we did find are a 2-dimensional version of rule 90, these base=2 rules we didn't find would be a 2-dimensional version of rule 22.

5 of the sampled rules have a scale of 3. Each of these rules have a base of 1 and the same population sequence:
[1, 3, 3, 9, 3, 9, 9, 27, 3, 9, 9, 27, 9, 27, 27, 81, 3, 9, 9, 27, 9, 27, 27, 81, 9, 27, 27, 81, 27, 81, 81, 243, 3]

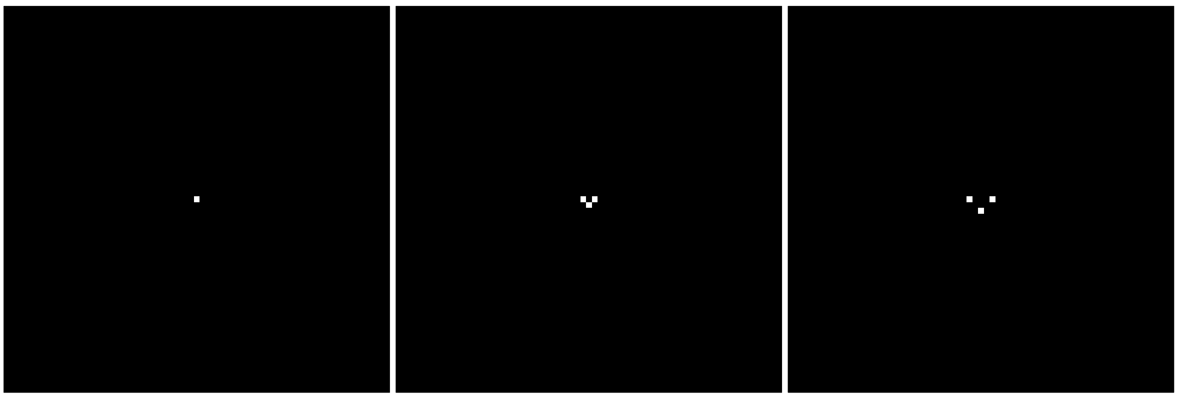
By looking at the grid data (in appendix H) we see some very interesting results. The rule is forming Sierpinski patterns, without any need for us to layer or overlay different timesteps. The grid data also shows that the pattern is expanding in 3 directions. Since the patterns that expand in 2 directions have a scale of 2, and the patterns that expand in 4 directions have a scale of 4, this explains why this pattern has a scale of 3.

This pattern could not exist in the totalistic or outer-totalistic rulesets since they are non-directional rules. They only consider the centre cell and the sum of it's neighbours (or just the sum of all of them) which doesn't take in to account what each individual neighbour's state is, so they cannot distinguish between just the left neighbour being 1, or just the right neighbour being 1 for example. The complete ruleset maps every possible input separately so can make directional distinctions.

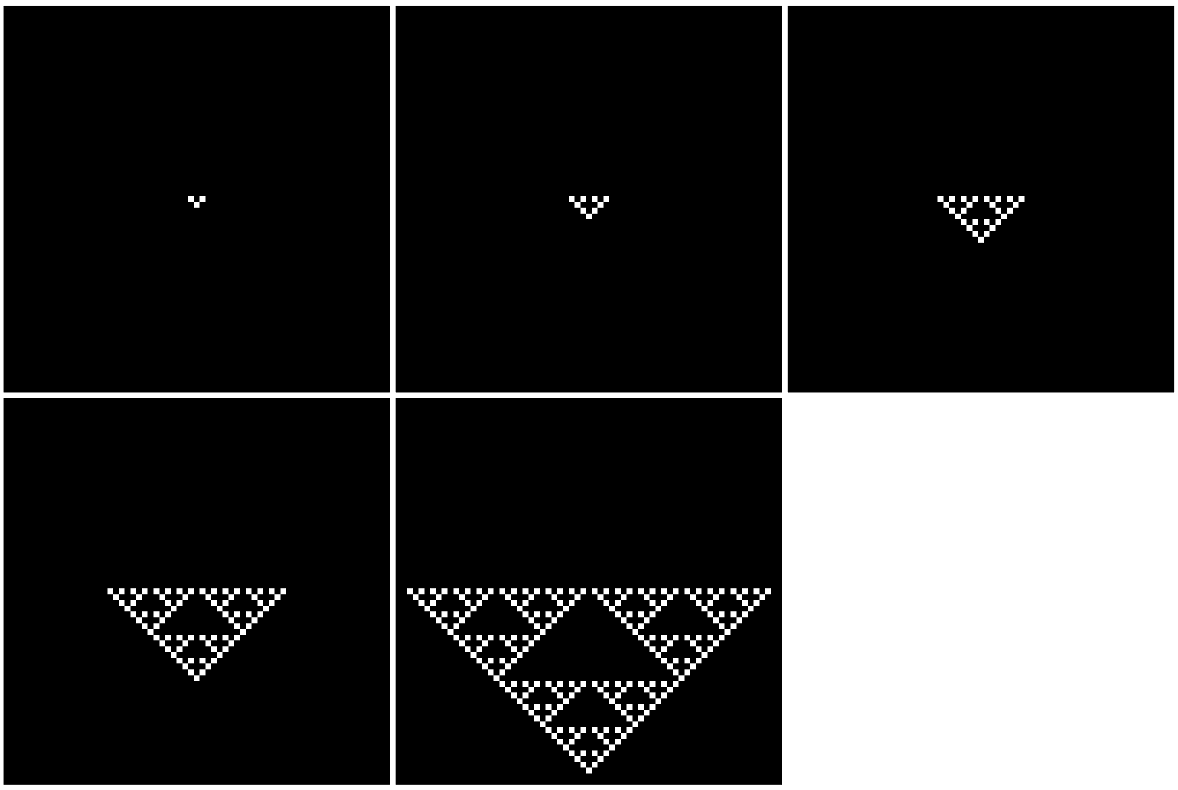
Looking at the grid data, we can see that the way it develops at each timestep is the same as rule 90, but in 3 directions instead of 2. We can consider it to be a 2.5-dimensional rule. This is also made clear by the population sequence. It may not be immediately obvious how we seem to generate much more complex behaviour than the 2-directional, or 4-directional patterns, so I will explain.

Looking at the pattern, we can categorise each timestep as either expansion or collapse. In the expansion stage each cell with state=1 'expands' outwards into its neighbours in 3 directions, then in the collapse stage, each continuous triangle block of pattern 'collapses' to 3 cells at each corner of the triangle. Given the first timestep as a base case, for each timestep after that; each even timestep is an expansion, and each odd timestep is a collapse.

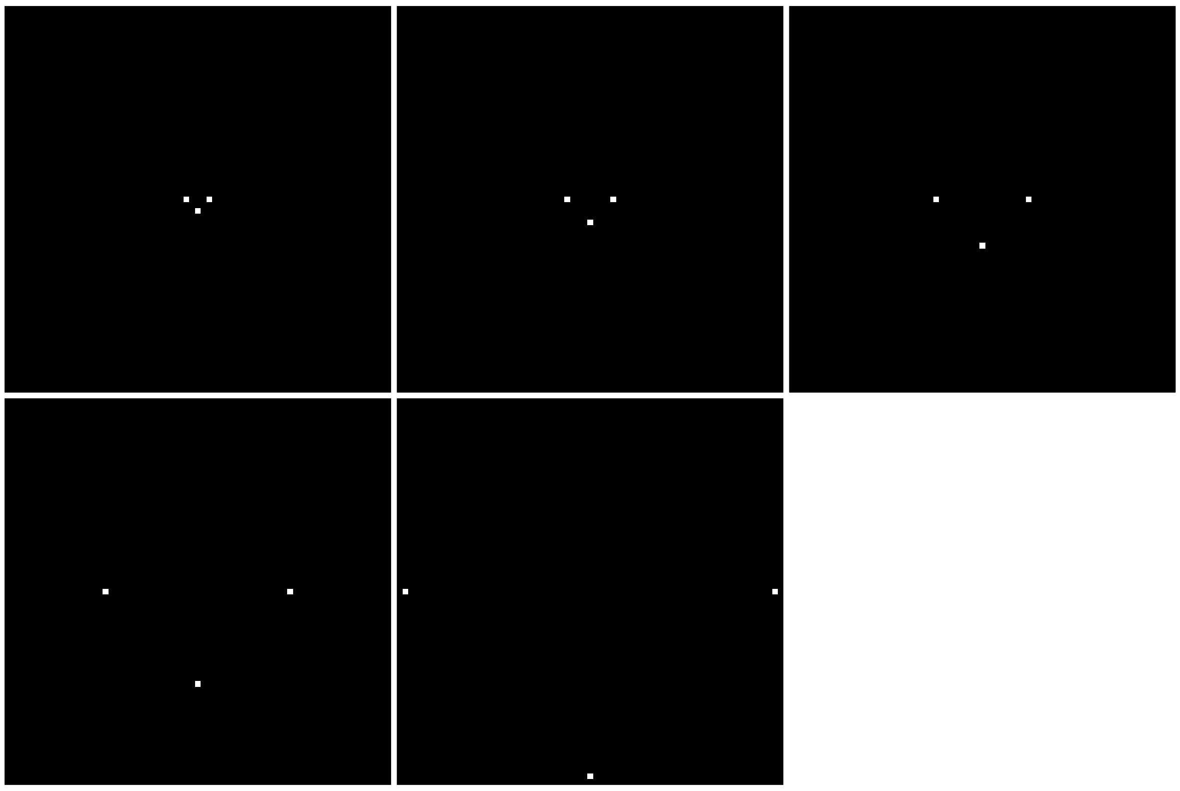
Below is the timesteps 1 to 3, which is base case, expansion, and collapse:



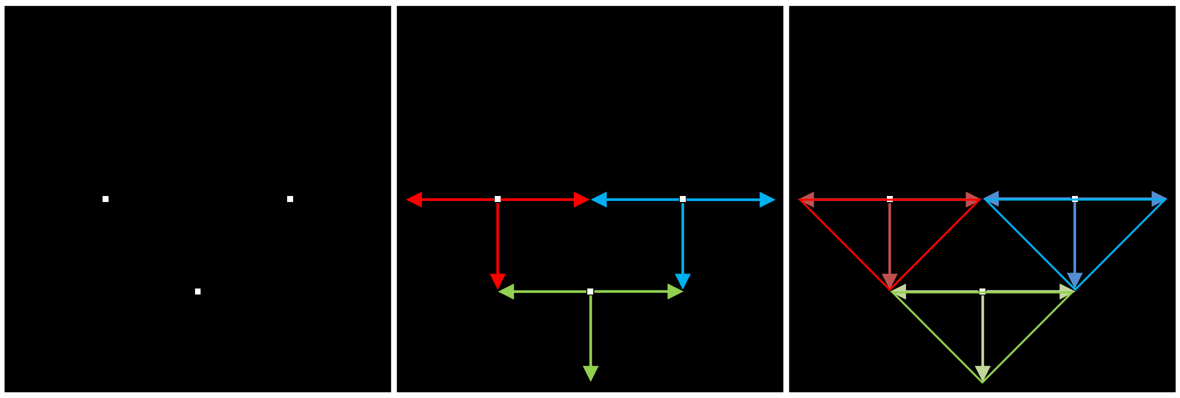
Furthermore, the pattern forms in to 1 continuous block at timesteps [2, 4, 8, 16, 32]:



And there is a subsequent total collapse, where the pattern reduces to just 3 cells with state=1, at timesteps [3, 5, 9, 27, 33]:



If we consider that we start with a single state=1 cell, and at some point we will reach a total collapse with just 3 state=1 cells. Then those 3 cells will expand outwards, however they won't expand upwards (in this example). This leaves a triangle gap in the middle of the pattern:



Each of these 3 cells in the total collapse are also 3 separate base cases, which is why each of the 3 triangles marked in the diagram replicate the same pattern within themselves recursively until we get down to single pixels at which point the pattern cannot get any smaller.

Discussion

This project successfully explored several aspects of patterns emerging from the totalistic, outer-totalistic and complete 2-dimensional cellular automata. We were able to analyse the relationships between the different patterns that emerged, and to compare them to the 1-dimensional elementary cellular automata. It was also a pleasant surprise to find the 3-scale patterns, which generate Sierpinski patterns in place without any extra processing.

There were also several other interesting findings along the way which didn't make it into the final report, such as some periodic patterns. It would have been interesting to explore why they produce the behaviour that they do. For example, exploring why they all had alternating backgrounds, and to compare that to Moore neighbourhood cellular automata such as the game of life to see how parameters such as neighbourhood affect the possibility for complex behaviour.

In terms of problems, I think the main one was a lack of direction initially. While the general aim of the project was decided early on, a lot of time was spent features that ended up not being used at all.

For example, I spent a while trying to see if it would be feasible to simulate the complete ruleset by reducing by removing all rules that are rotationally or a mirrored equivalent of another rule, but eventually found it was still unfeasible.

I also spent a lot of time performing manual filtering on the population graphs for rules, to try and see if that yielded any different results to the automatic filtering and explore why. Despite there being some interesting results, it was too subjective, and I was unable to provide any meaningful insight.

Instead of spending time on these it might have been more productive to include other features, such as analysing the power spectra of the rules, which provides another way to categorise the results [6]. Or focusing on other mathematical patterns. For example, the Ulam cellular automata [2] is known to have complex features and is contained within the outer-totalistic cellular automata as rules 87 and 95 (using this project's encoding) but doesn't specifically fit the mathematical sequence we used as criteria.

Conclusion

The overall proportion of rules that fit the growing repeating pattern is:

Totalistic	2 / 64	≈ 3.1%
Pure outer-totalistic	40 / 1024	≈ 3.9%
Custom outer-totalistic	160 / 4096	≈ 3.9%
Complete	176 / 4096	≈ 4.3%

This shows a similar proportion across each ruleset, but the sample size for the complete ruleset is about 1 millionth of the total ruleset, so it should not be considered precise.
Both outer-totalistic rulesets have the exact same proportion, which is because when the sum is 2, only the adjacent case matters, so distinguishing between that and the opposite case doesn't change the overall proportion.
Furthermore, while the totalistic ruleset has a slightly different proportion to the outer-totalistic rulesets, it's as close as it can be. 40/1024 = 2.5/64, so only 2.5 matching rules would have given the exact same proportion which is impossible, and either 2 or 3 matching rules is as close as it is possible to get.

The totalistic rules are a subset of the outer-totalistic rules, which are a subset of the complete rules. So any patterns found in the totalistic ruleset can be found in the outer-totalistic rulesets and so on. This influences the patterns we find in each set:

Totalistic:

- 2/2 [1, 5, 4, 20, 4, 20, 16, 80, 4, 20, 16, 80, 16, 80, 64, 320] base=2, scale=4
  4-directional version of the 2-directional rule 22.
  Refer to appendix C.

Pure outer-totalistic:

- 32/40 [1, 4, 4, 16, 4, 16, 16, 64, 4, 16, 16, 64, 16, 64, 64, 256] base=1, scale=4
  4-directional version of the 2-directional rule 90. Refer to appendix B.
- 8/40 [1, 5, 4, 20, 4, 20, 16, 80, 4, 20, 16, 80, 16, 80, 64, 320] base=2, scale=4
  4-directional version of the 2-directional rule 22. Refer to appendix C.

Custom outer-totalistic:

- 128/160 [1, 4, 4, 16, 4, 16, 16, 64, 4, 16, 16, 64, 16, 64, 64, 256] base=1, scale=4
  4-directional version of the 2-directional rule 90. Refer to appendix B.
- 32/160 [1, 5, 4, 20, 4, 20, 16, 80, 4, 20, 16, 80, 16, 80, 64, 320] base=2, scale=4
  4-directional version of the 2-directional rule 22. Refer to appendix C.

Complete:

- 153/176 [1, 2, 2, 4, 2, 4, 4, 8, 2, 4, 4, 8, 4, 8, 8, 16] base=1, scale=2
  Same as rule 90. Has 2 variations that develop in opposite and adjacent directions.
  Refer to appendices D and E.
- 15/176 [1, 3, 2, 6, 2, 6, 4, 12, 2, 6, 4, 12, 4, 12, 8, 24] base=2, scale=2
  Same as rule 22. Has 2 variations that develop in opposite and adjacent directions.
  Refer to appendices F and G.
- 5/176 [1, 3, 3, 9, 3, 9, 9, 27, 3, 9, 9, 27, 9, 27, 27, 81] base=1, scale=3
  3-directional version of the 2-directional rule 90. Refer to appendix H.
- 3/176 [1, 4, 4, 16, 4, 16, 16, 64, 4, 16, 16, 64, 16, 64, 64, 256] base=1, scale=4
  4-directional version of the 2-directional rule 90. Refer to appendix B.

There is the pattern where base=2, scale=4 that occurs in sets other than the complete ruleset. This pattern must also be in the complete ruleset since it contains those other sets. The reason we didn't find it then is because of the small sample size.
The outer-totalistic rulesets contain the pattern where base=1, scale=4 which doesn't occur in the totalistic ruleset. This is simply because that pattern develops from rules that aren't totalistic since they consider the centre cell and the neighbour cells separately.
The complete ruleset also contains patterns that don't occur in the other rulesets. Specifically with scale=2, and scale=3. This is because the totalistic and outer-totalistic rules treat all directions equally and so must develop in all 4 directions equally causing a scale of 4. The complete ruleset is directional and takes in to account all possible inputs which allows it to develop differently in different directions. We find a scale of 2 when a rule only develops in 2 directions, and a scale of 3 when it develops in 3 directions.

We saw how we can generate Sierpinski patterns and similar patterns, by layering or overlaying the grid data from rules where the scale=2.

We also saw how the scale=3 rules generate Sierpinski patterns without any extra effort required beyond simulating the rule. Despite the data appearing much more complex than the other rules at first glance, we demonstrated how it is a natural result of applying the same basic rules but in 3 directions instead of 2 or 4.

Finally, it is important to remember that these rules were only tested with a specific initial grid configuration, and that with a randomised start for example, rules that produced the same patterns as each other in our tests could produce different patterns to each other.

This is the end of the main section of the report. After this are the references, and the appendices which contains extra information:

- In appendix A you will find lists of the specific rules that produced each category of pattern.
- In appendices B to H, you will find grid data for different patterns.
- In appendix I you will find information on how to access and run the code.

References

[1] Von Neumann, J. (1966) ‘Theory of self-reproducing automata’
http://www.arise.mae.usp.br/wp-content/uploads/2018/03/Theory-of-self-reproducing-automata.pdf

[2] Ulam, S. (1962) ‘On some mathematical problems connected with patterns of growth of figures’
https://oeis.org/A002858/a002858.pdf

[3] Conway, J. (1970) ‘The fantastic combinations of John Conway's new solitaire game "life"’
https://web.stanford.edu/class/sts145/Library/life.pdf

[4] [8] Wolfram, S. (2002) ‘A new kind of science’
https://www.wolframscience.com/nks/ [4] (231-235). [8] (55)

[5] Berlekamp, E. Conway, J. Guy, R. (2004) ‘Winning ways for your mathematical plays, volume 4’
https://bobson.ludost.net/copycrime/Winning%20Ways%202nd%20Edition/Winning.Ways.for.Your.Mathematical.Plays.V4--1568811446.pdf (927-961)

[6] Nagler, J. Claussen, J. (2005) ‘1⁄fα spectra in elementary cellular automata and fractal signals’
https://journals.aps.org/pre/abstract/10.1103/PhysRevE.71.067103

[7] Kawaharada, A. (2014) ‘Ulam’s cellular automaton and Rule 150’
https://projecteuclid.org/journals/hokkaido-mathematical-journal/volume-43/issue-3/Ulams-cellular-automaton-and-Rule-150/10.14492/hokmj/1416837570.short

[9] Wolfram, S. (1983) ‘Statistical mechanics of cellular automata’
https://content.wolfram.com/uploads/sites/34/2020/08/statistical-mechanics-cellular-automata.pdf
(61)

Appendix A

Lists of rules that fit the pattern.

Totalistic:
[2, 10]

Pure outer totalistic:
Base = 1: [64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 320, 322, 324, 326, 328, 330, 332, 334, 336, 338, 340, 342, 344, 346, 348, 350]
Base = 2: [65, 69, 73, 77, 321, 325, 329, 333]

Custom outer totalistic:
Base = 1: [128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 640, 642, 644, 646, 648, 650, 652, 654, 656, 658, 660, 662, 664, 666, 668, 670, 672, 674, 676, 678, 680, 682, 684, 686, 688, 690, 692, 694, 696, 698, 700, 702, 1152, 1154, 1156, 1158, 1160, 1162, 1164, 1166, 1168, 1170, 1172, 1174, 1176, 1178, 1180, 1182, 1184, 1186, 1188, 1190, 1192, 1194, 1196, 1198, 1200, 1202, 1204, 1206, 1208, 1210, 1212, 1214, 1664, 1666, 1668, 1670, 1672, 1674, 1676, 1678, 1680, 1682, 1684, 1686, 1688, 1690, 1692, 1694, 1696, 1698, 1700, 1702, 1704, 1706, 1708, 1710, 1712, 1714, 1716, 1718, 1720, 1722, 1724, 1726]
Base = 2: [129, 133, 137, 141, 145, 149, 153, 157, 641, 645, 649, 653, 657, 661, 665, 669, 1153, 1157, 1161, 1165, 1169, 1173, 1177, 1181, 1665, 1669, 1673, 1677, 1681, 1685, 1689, 1693]

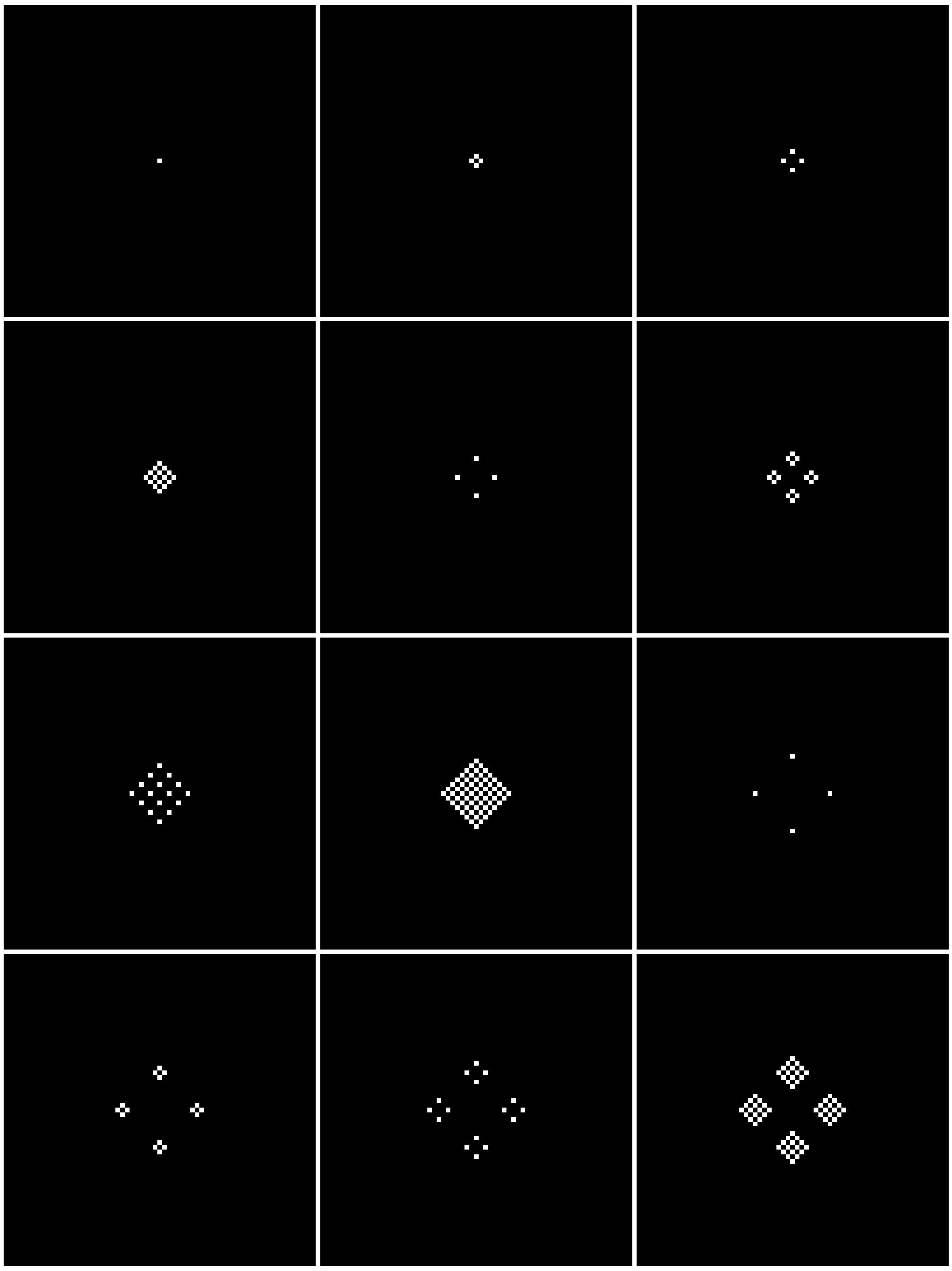
Complete:
Scale = 2, base = 1: [92696, 9709076, 75011732, 218238320, 229782380, 241654808, 242475628, 246646560, 262515320, 297881720, 327050060, 327508624, 333632428, 343099436, 358455906, 370411812, 392511536, 445769762, 448534852, 466616760, 474039468, 475122756, 475592262, 595825912, 619122568, 786946472, 919894556, 927181316, 934420792, 1060669232, 1143953616, 1165624632, 1165665926, 1183654968, 1212439068, 1240923116, 1245421764, 1281442552, 1286564880, 1287186460, 1316360972, 1436183504, 1504400056, 1511656090, 1513068960, 1521485382, 1556390452, 1559884704, 1564696304, 1580054744, 1591345936, 1621177120, 1623753560, 1701749196, 1743814936, 1759611296, 1792869768, 1927341620, 1941588868, 1944159824, 2009409136, 2020029648, 2024858172, 2028895412, 2075951372, 2081474488, 2135236632, 2153263396, 2154047206, 2185478256, 2191757552, 2194729772, 2242611320, 2260035880, 2273895212, 2278075664, 2375789196, 2424804136, 2461679796, 2473167364, 2481291824, 2494944552, 2508744198, 2529177400, 2565561736, 2593941964, 2611732620, 2616251332, 2621558800, 2760042796, 2795413202, 2819427576, 2824265946, 2853572242, 2930445836, 2963650160, 2971337860, 2977318630, 2989320720, 3006716708, 3010496772, 3012200728, 3020812680, 3028462008, 3034607916, 3088192594, 3102106508, 3196571312, 3262754020, 3265950084, 3272663404, 3277173858, 3301547808, 3319269784, 3337511748, 3338011188, 3400278204, 3402091400, 3436088340, 3470123304, 3477078512, 3498200440, 3581717476, 3649828880, 3662618496, 3668486072, 3695596052, 3696087192, 3714491308, 3723090000, 3723170488, 3732608296, 3778095116, 3791682236, 3795474580, 3991515856, 3993953340, 4002059530, 4007870848, 4031471352, 4035355804, 4064557836,

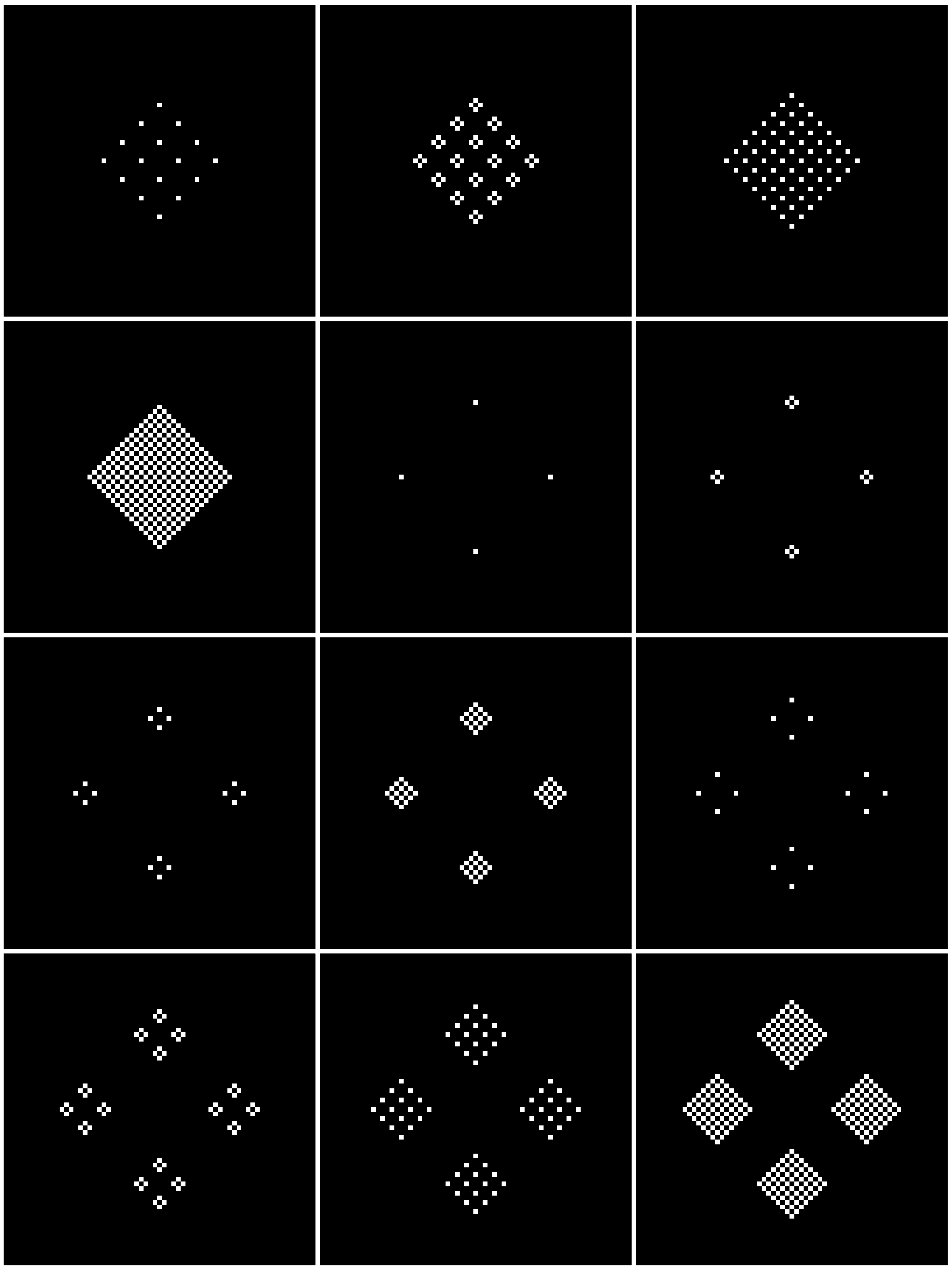
4074553676, 4079647108, 4086740114, 4097365510, 4103869380, 4110807128, 4154628428, 4227359012, 4237500940, 4242578412, 4244274716]
Scale=2, base = 2: [233374342, 2162224538, 2337546726, 2505688166, 2810376646, 2959135110, 3036720530, 3128820166, 3182495398, 3405101314, 3423197786, 3479341402, 3773585942, 4133512466, 4264416514]
Scale=3: [677623956, 1144717732, 2215871340, 3775502484, 3892817328]
Scale=4: [3927868, 207817628, 2296500028]

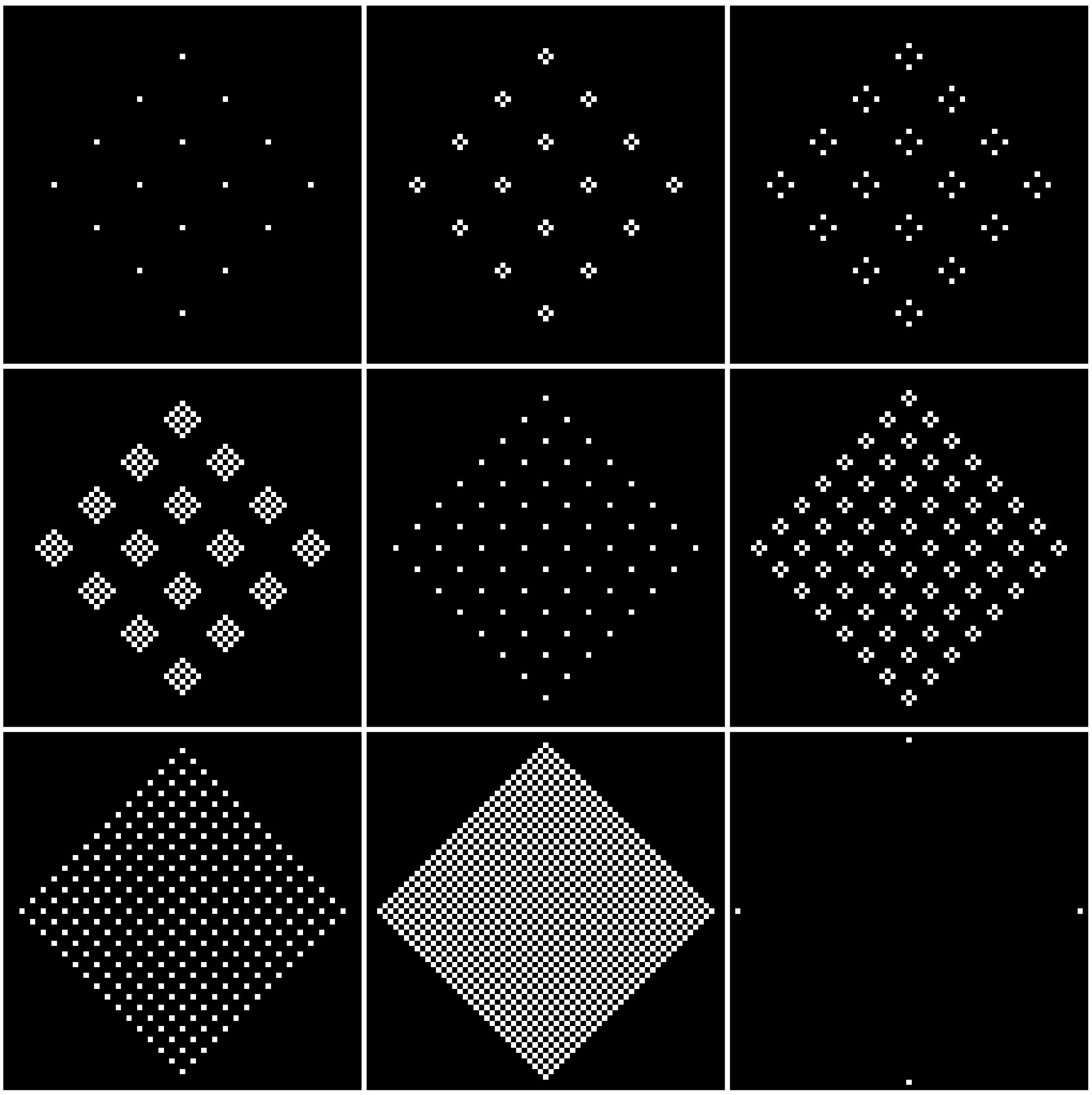
Appendix B

Grid data for outer-totalistic rules where base=1, and complete rules where scale=4.

The order should be read left to right, top to bottom, the same as text.



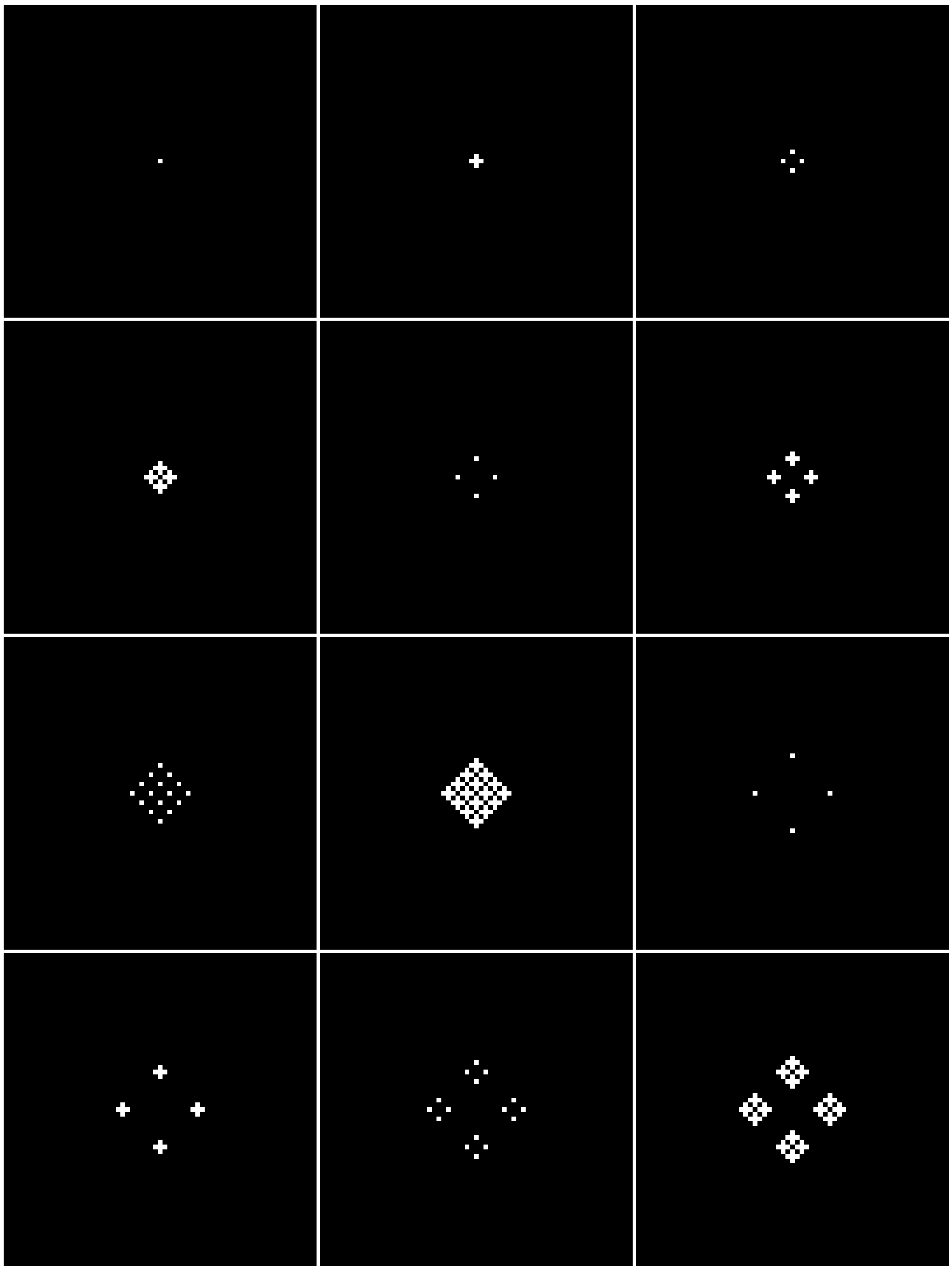


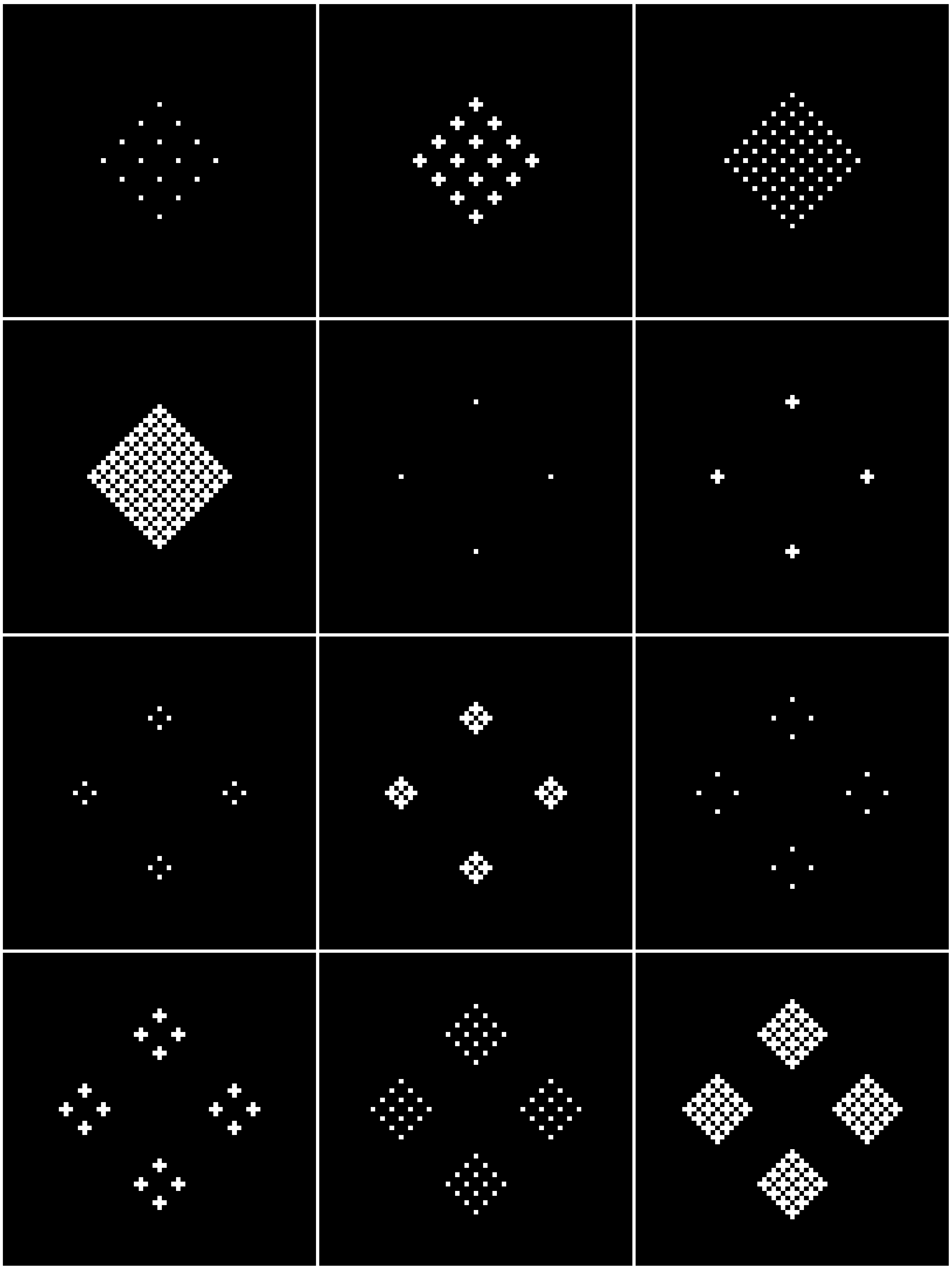


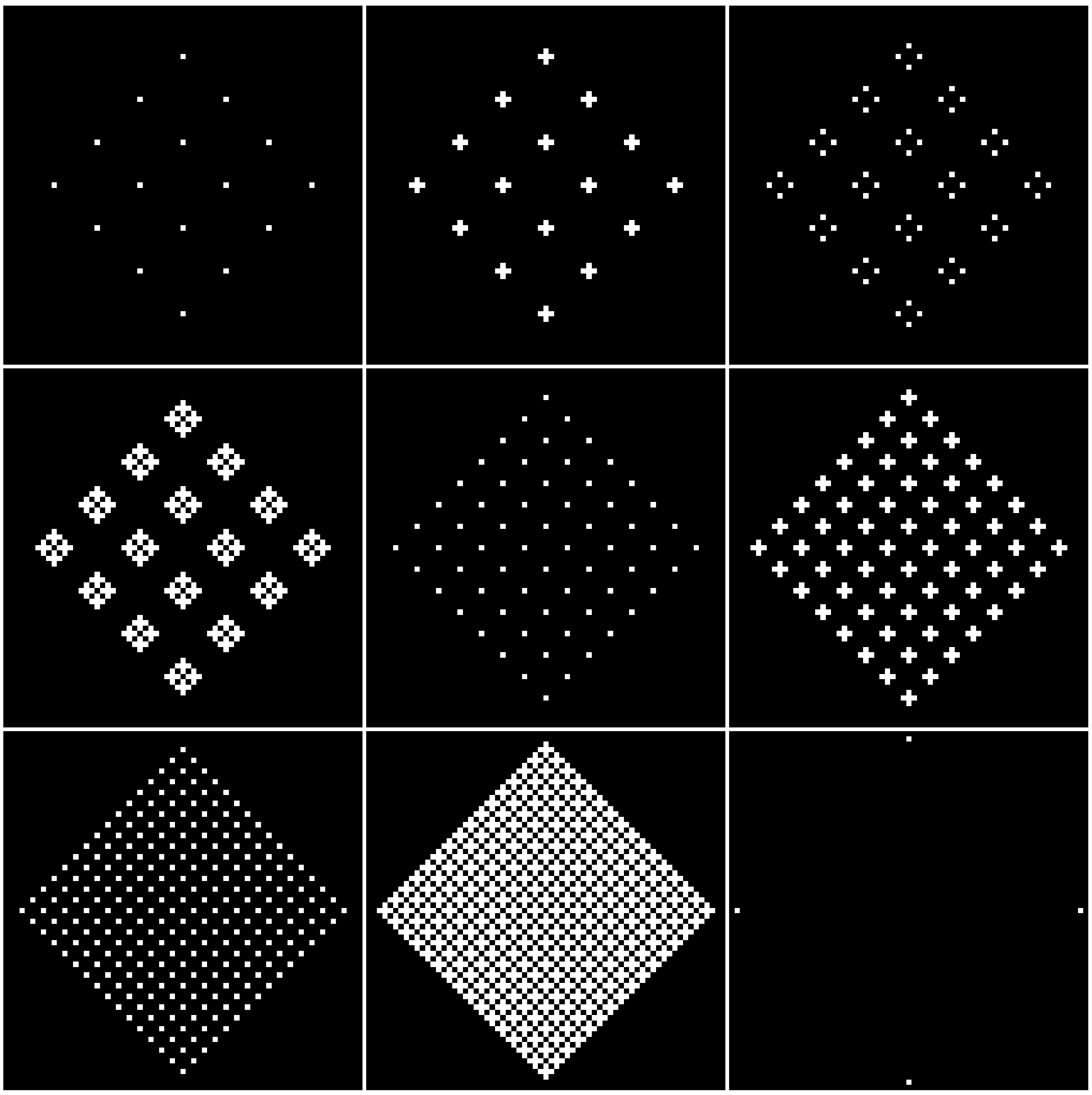
Appendix C

Grid data for totalistic rules, and outer-totalistic rules where base=2.

The order should be read left to right, top to bottom, the same as text.



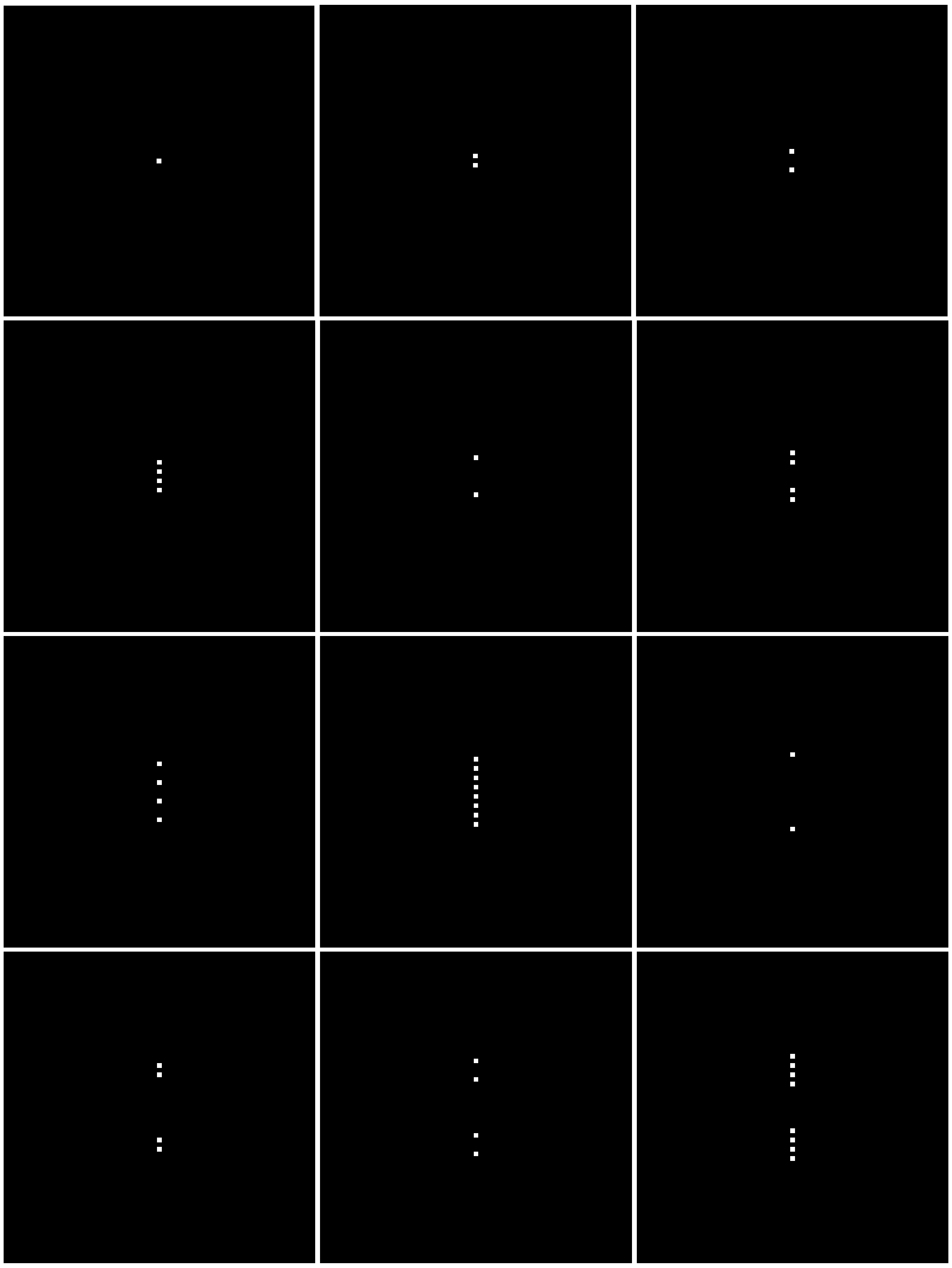


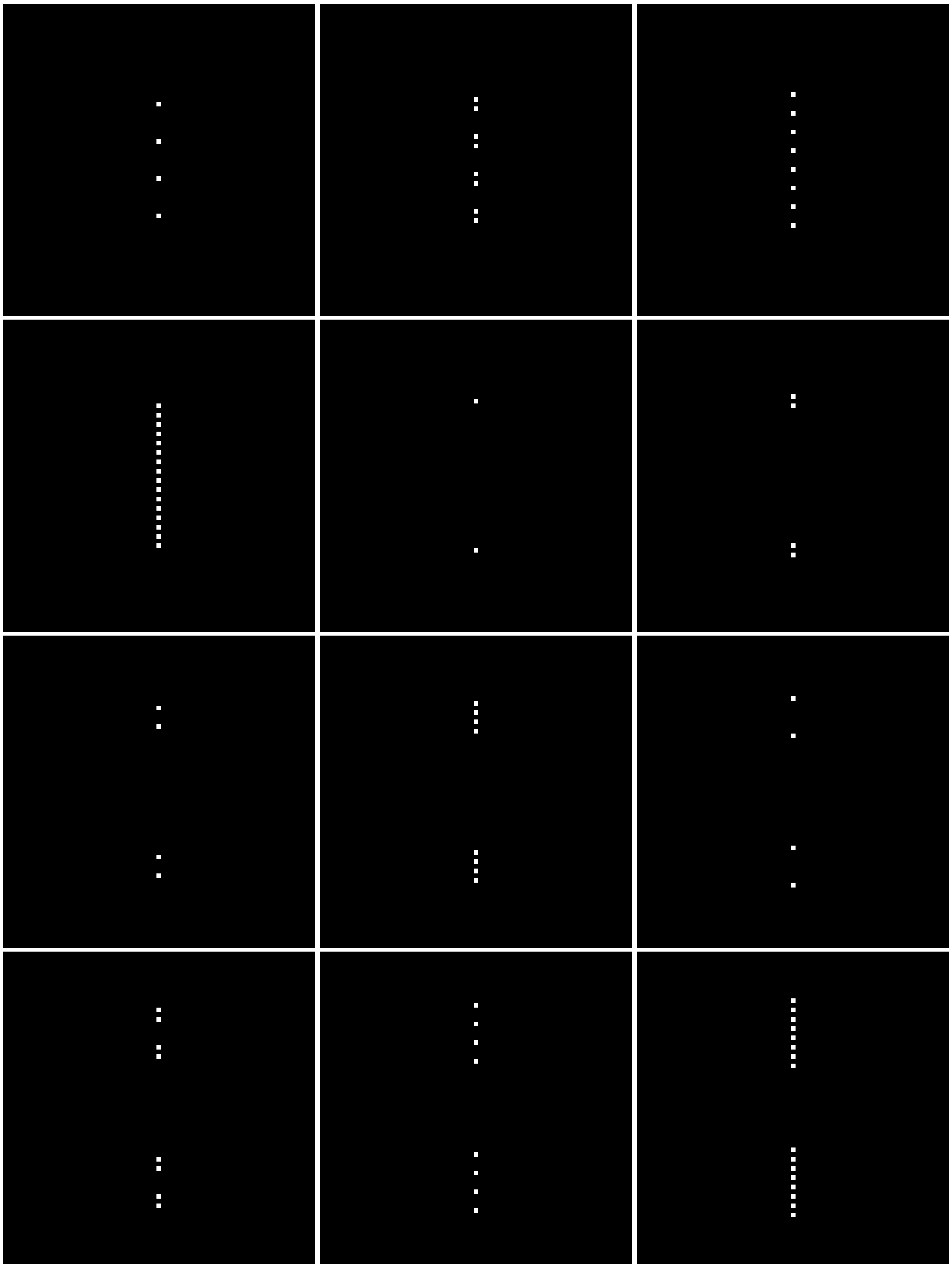


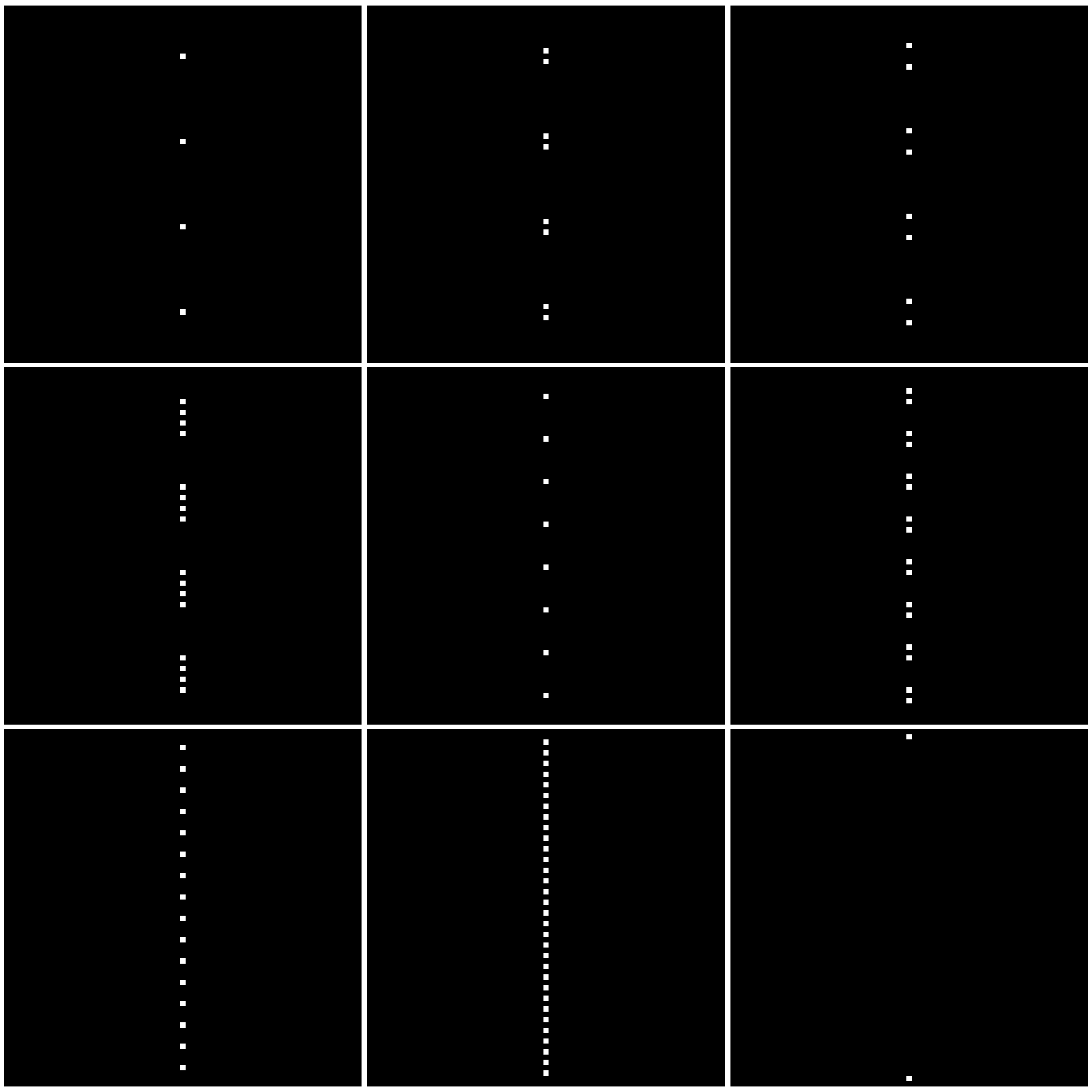
Appendix D

Grid data for some of the complete rules with scale=2, base=1.

The order should be read left to right, top to bottom, the same as text.



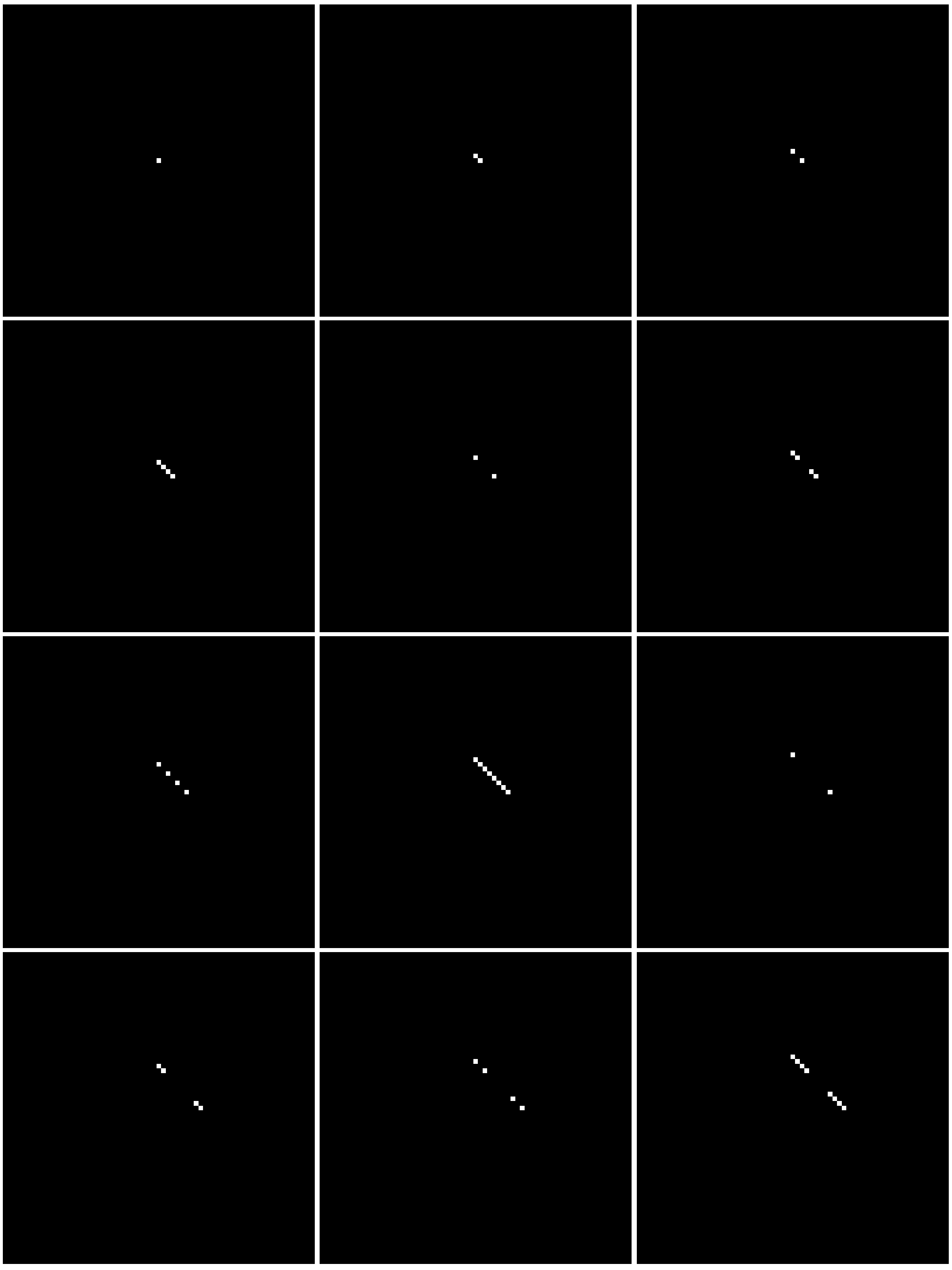


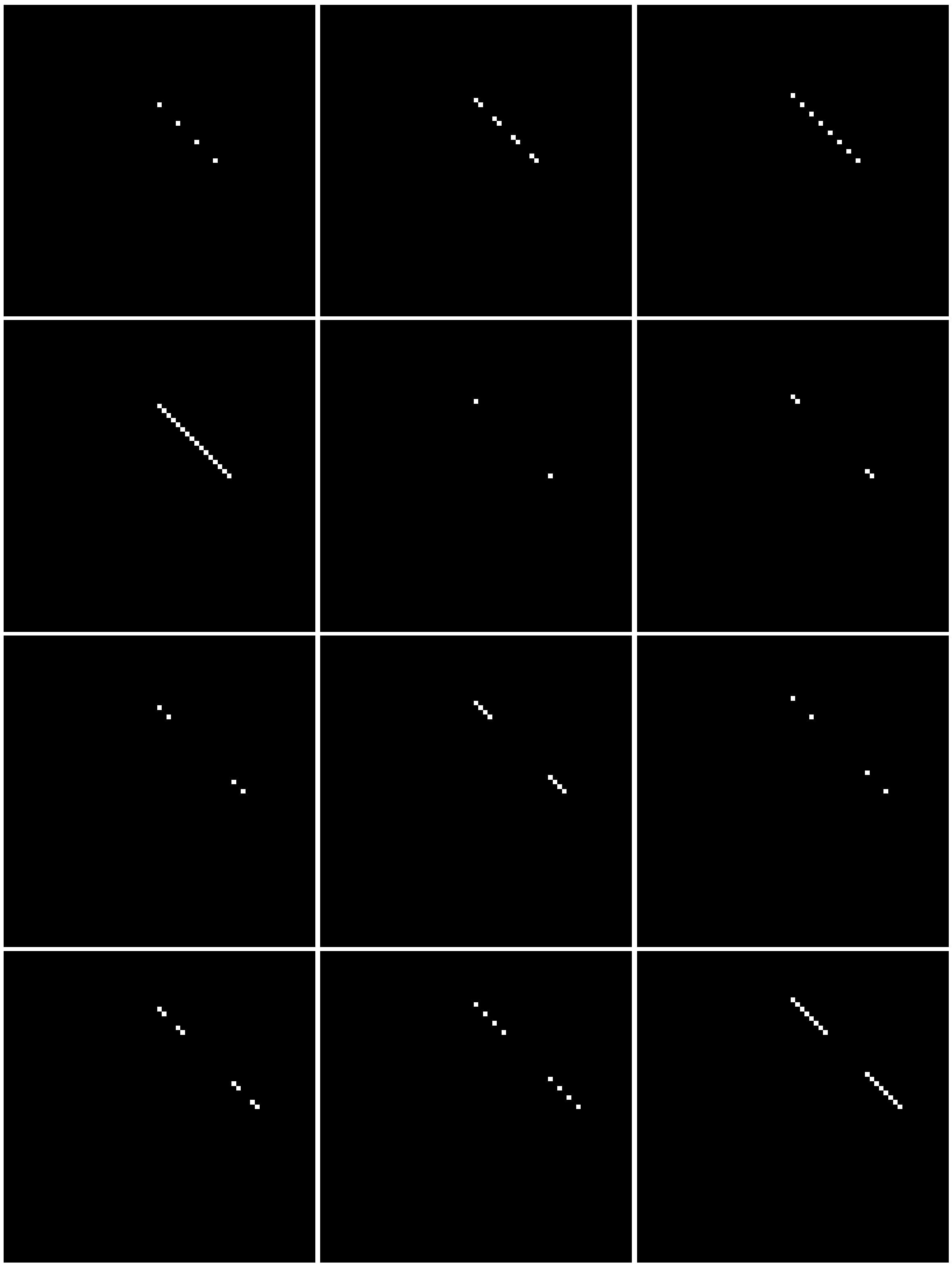


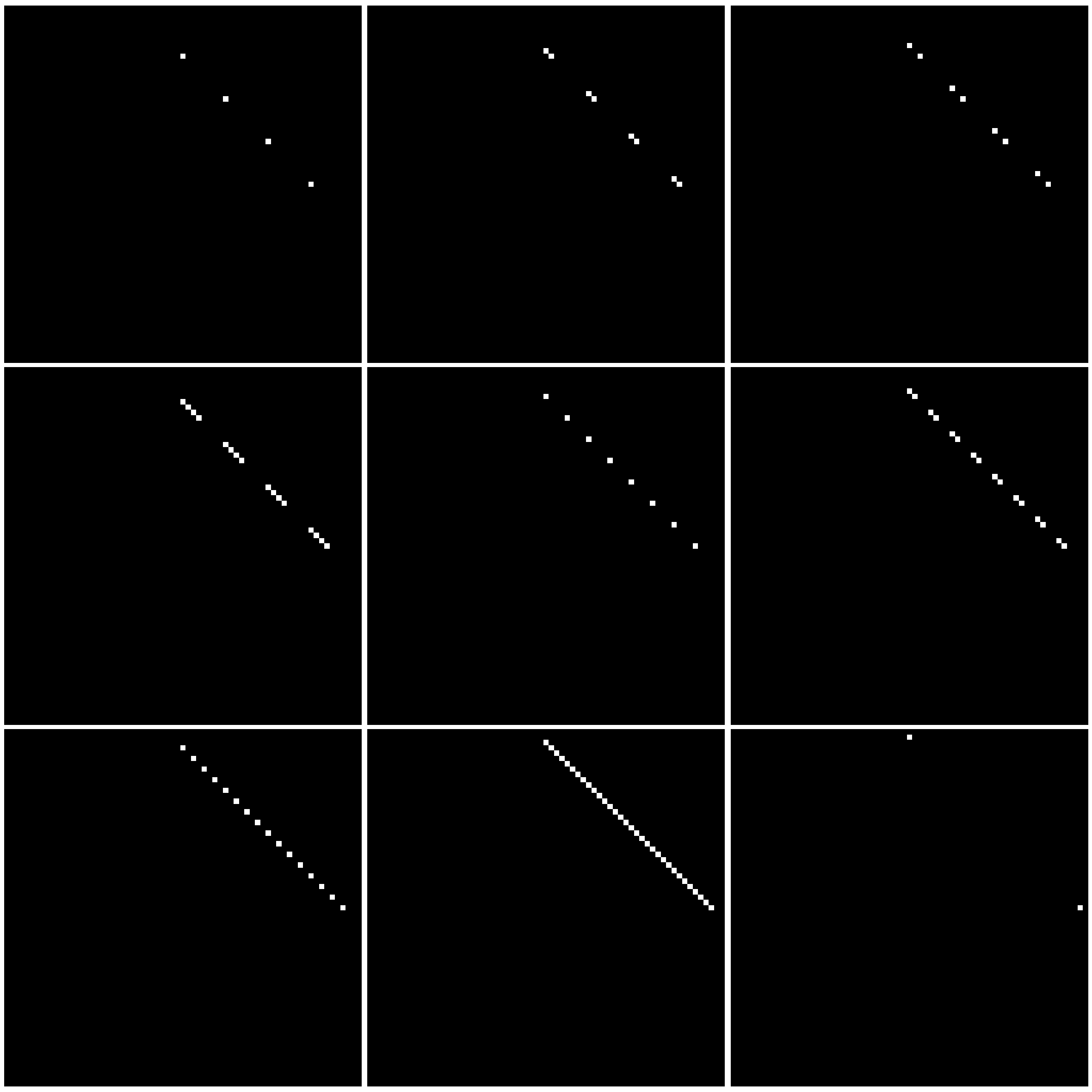
Appendix E

Grid data for some of the complete rules with scale=2, base=1.

The order should be read left to right, top to bottom, the same as text.



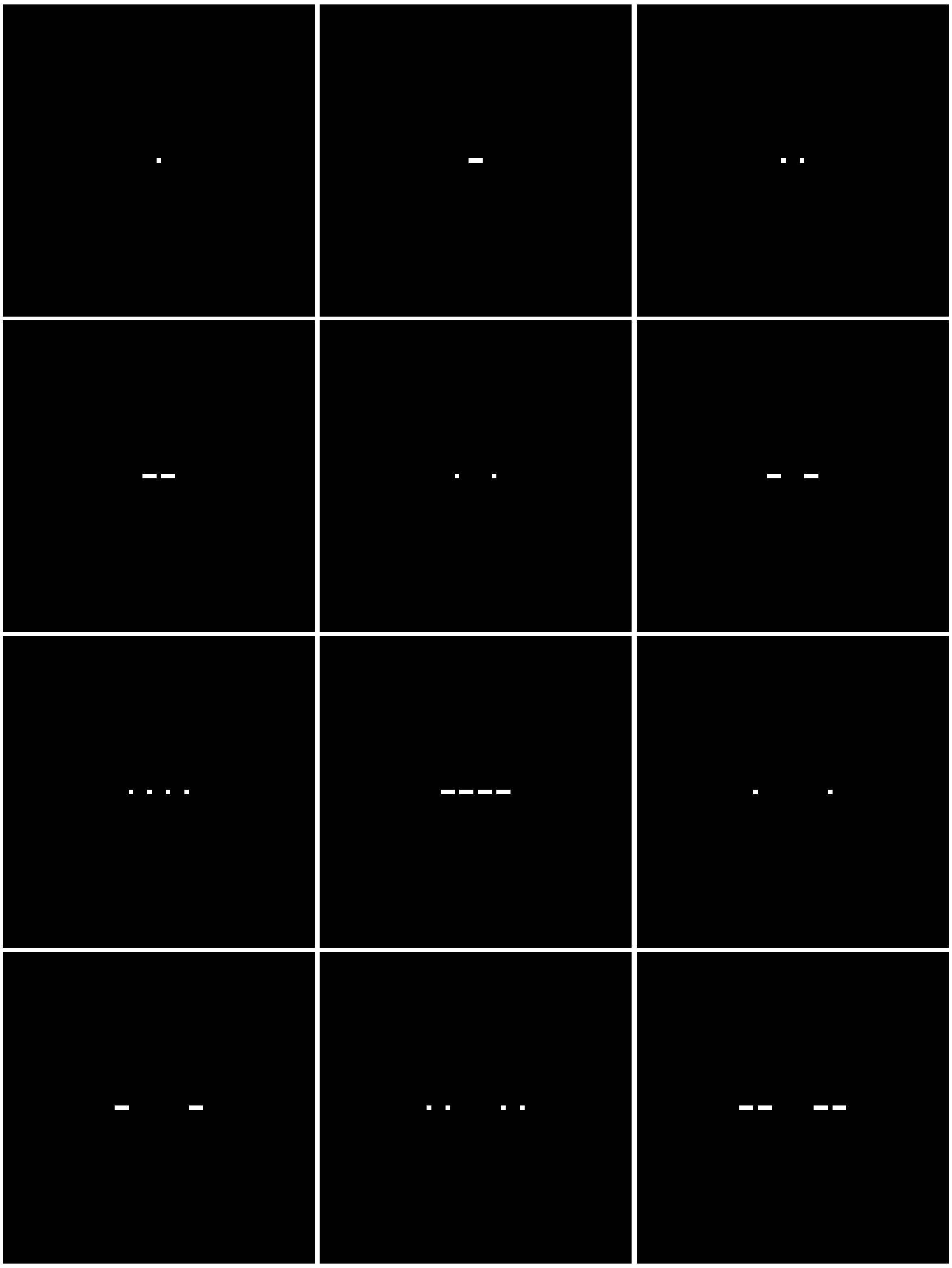


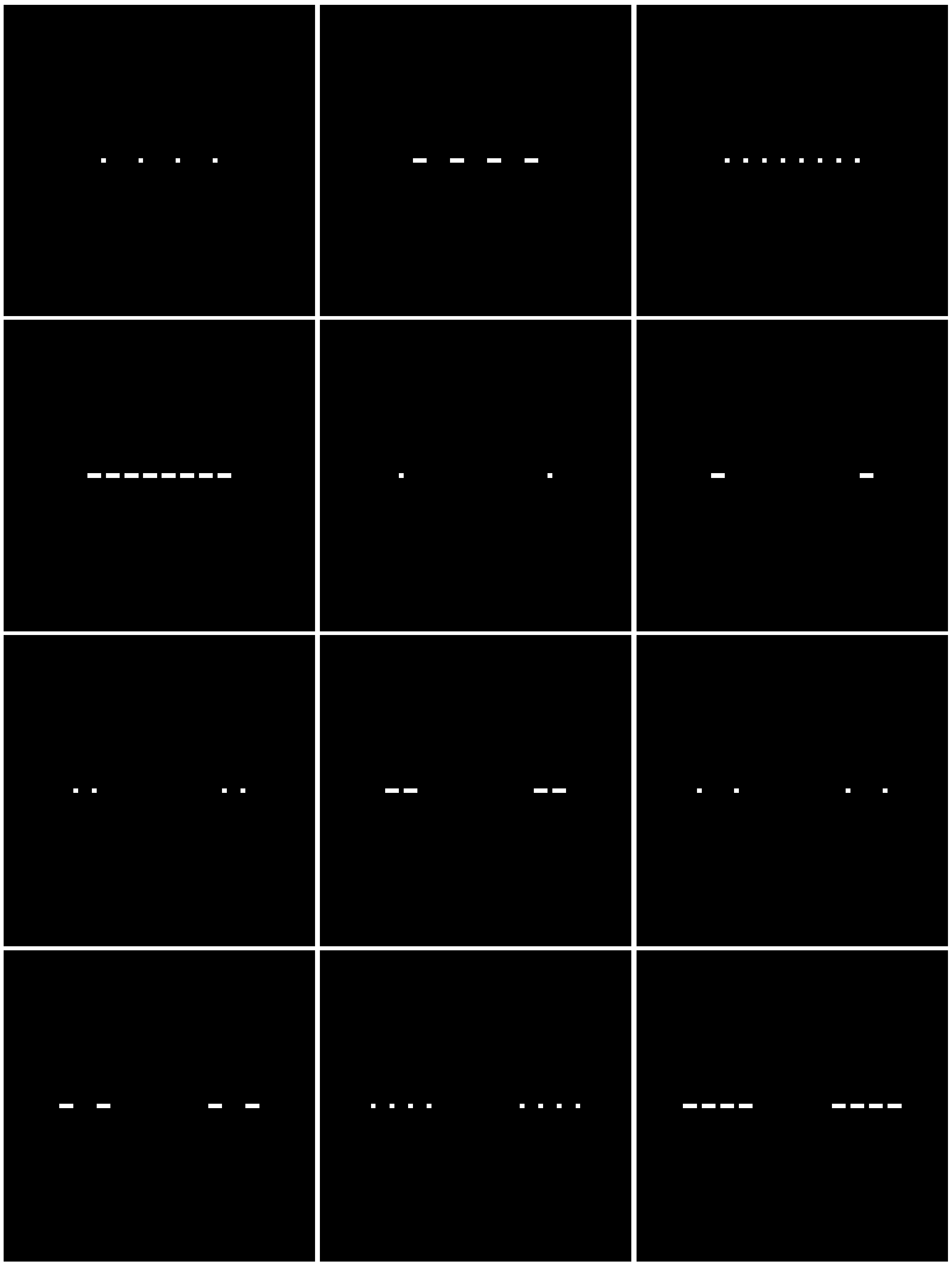


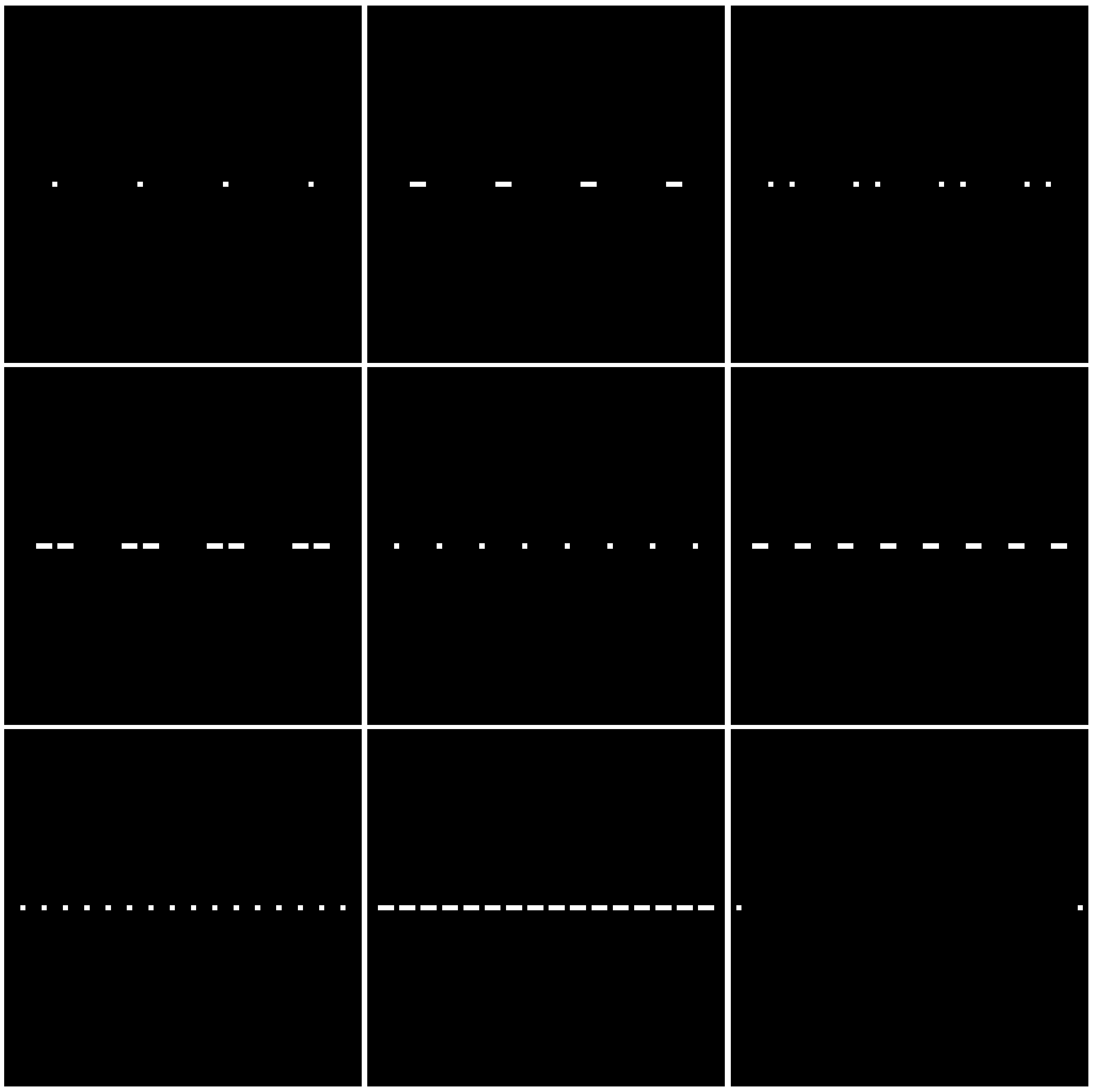
Appendix F

Grid data for some of the complete rules with scale=2, base=2.

The order should be read left to right, top to bottom, the same as text.



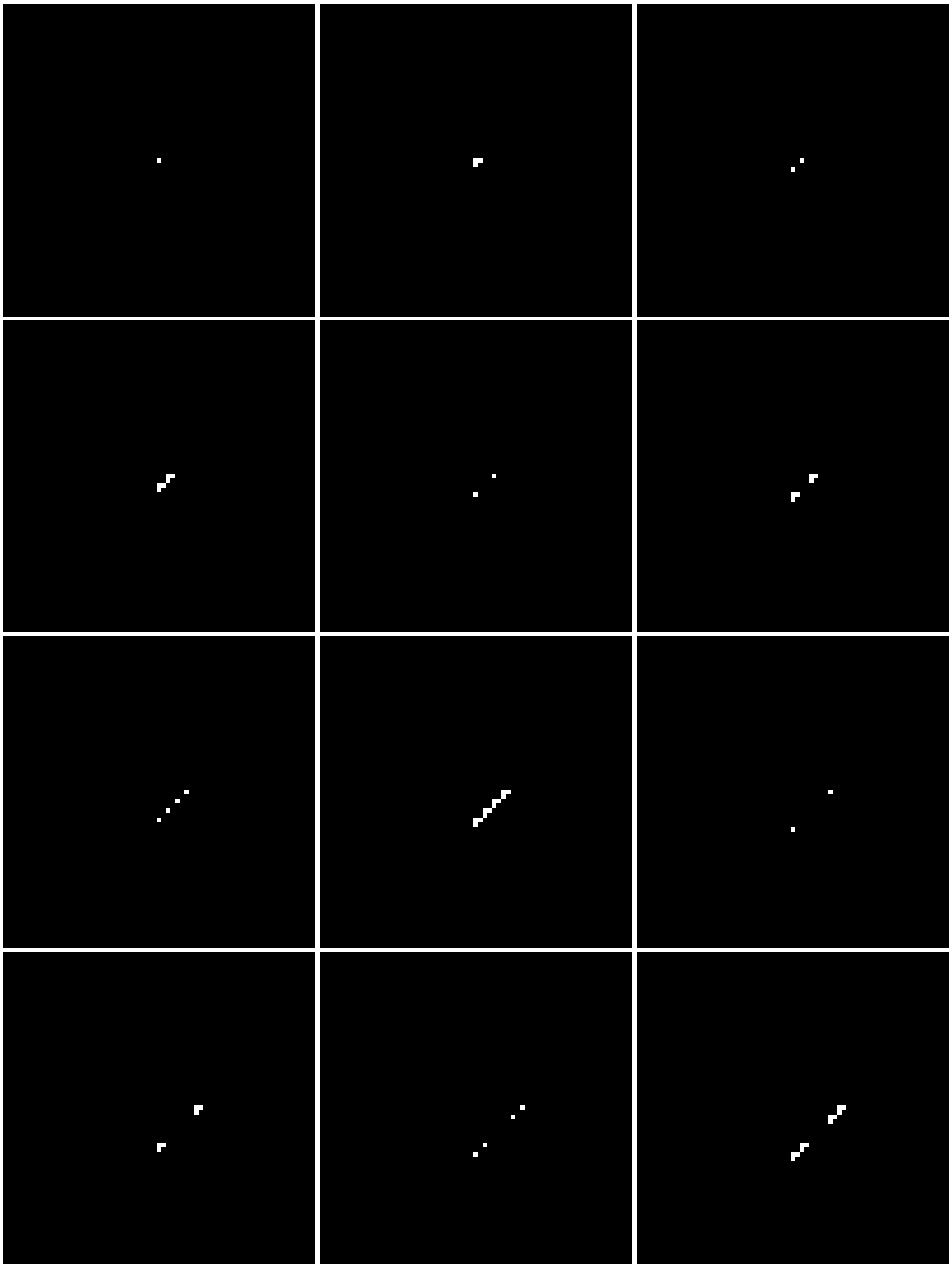


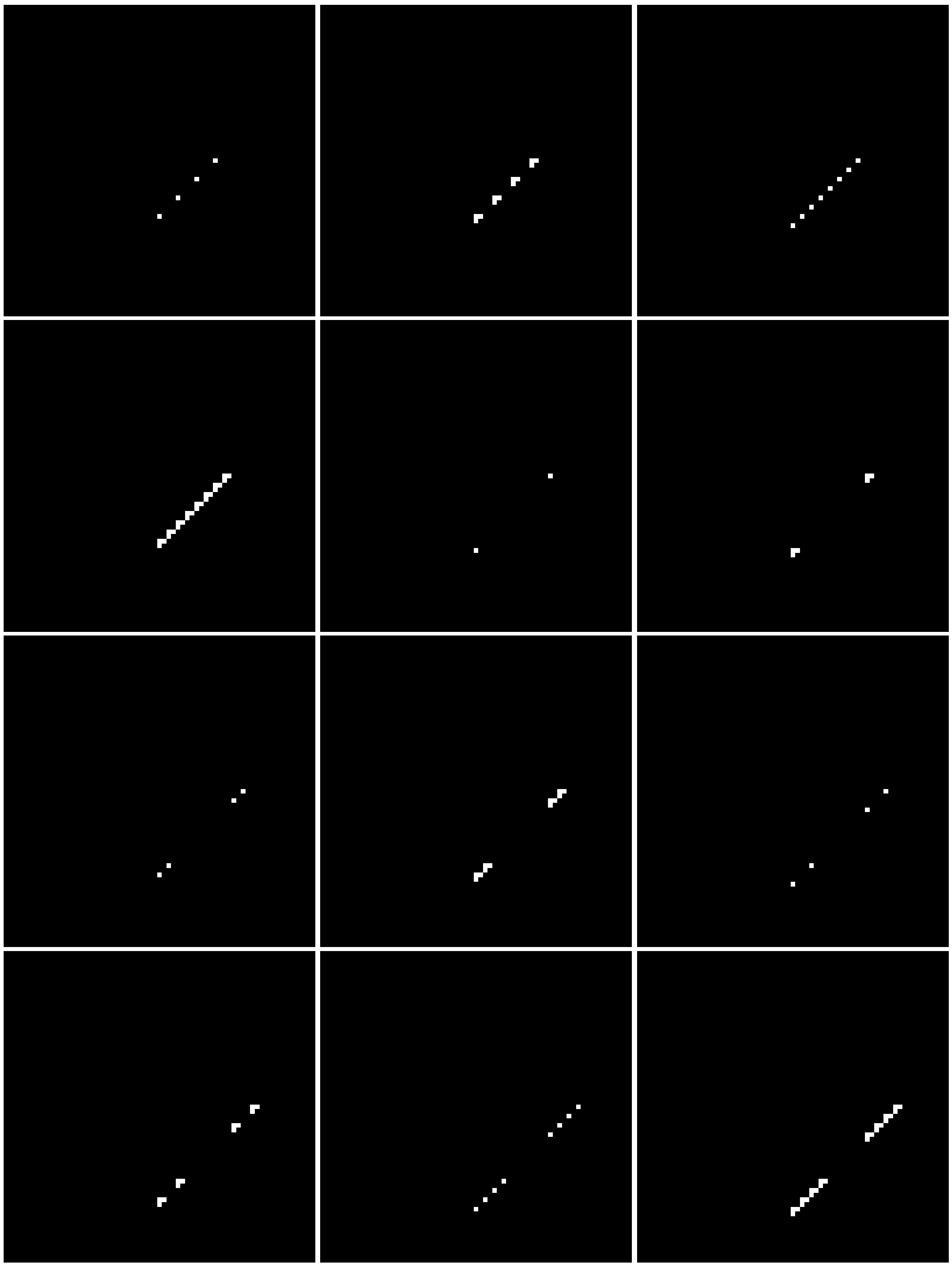


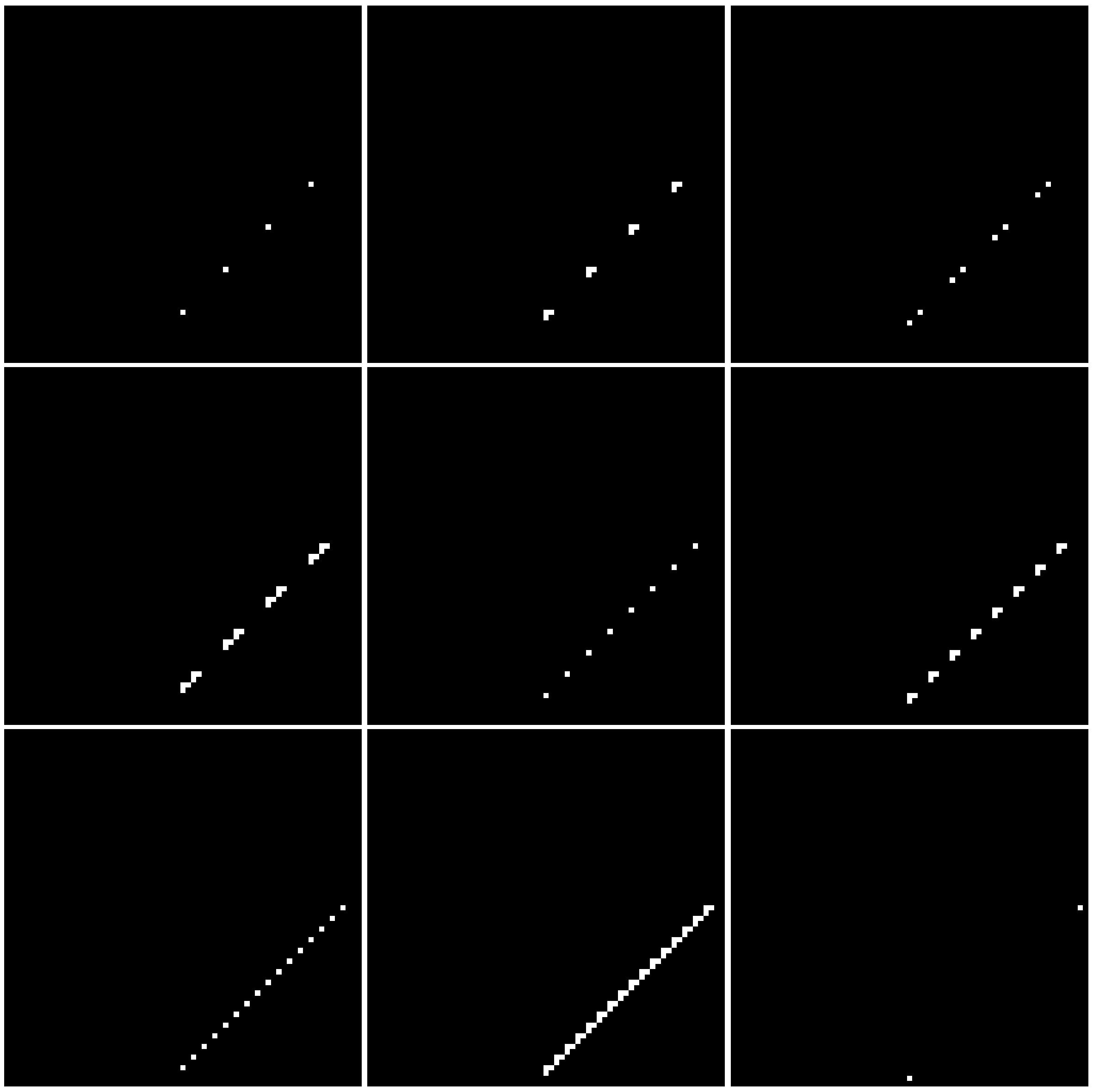
Appendix G

Grid data for some of the complete rules with scale=2, base=2.

The order should be read left to right, top to bottom, the same as text.



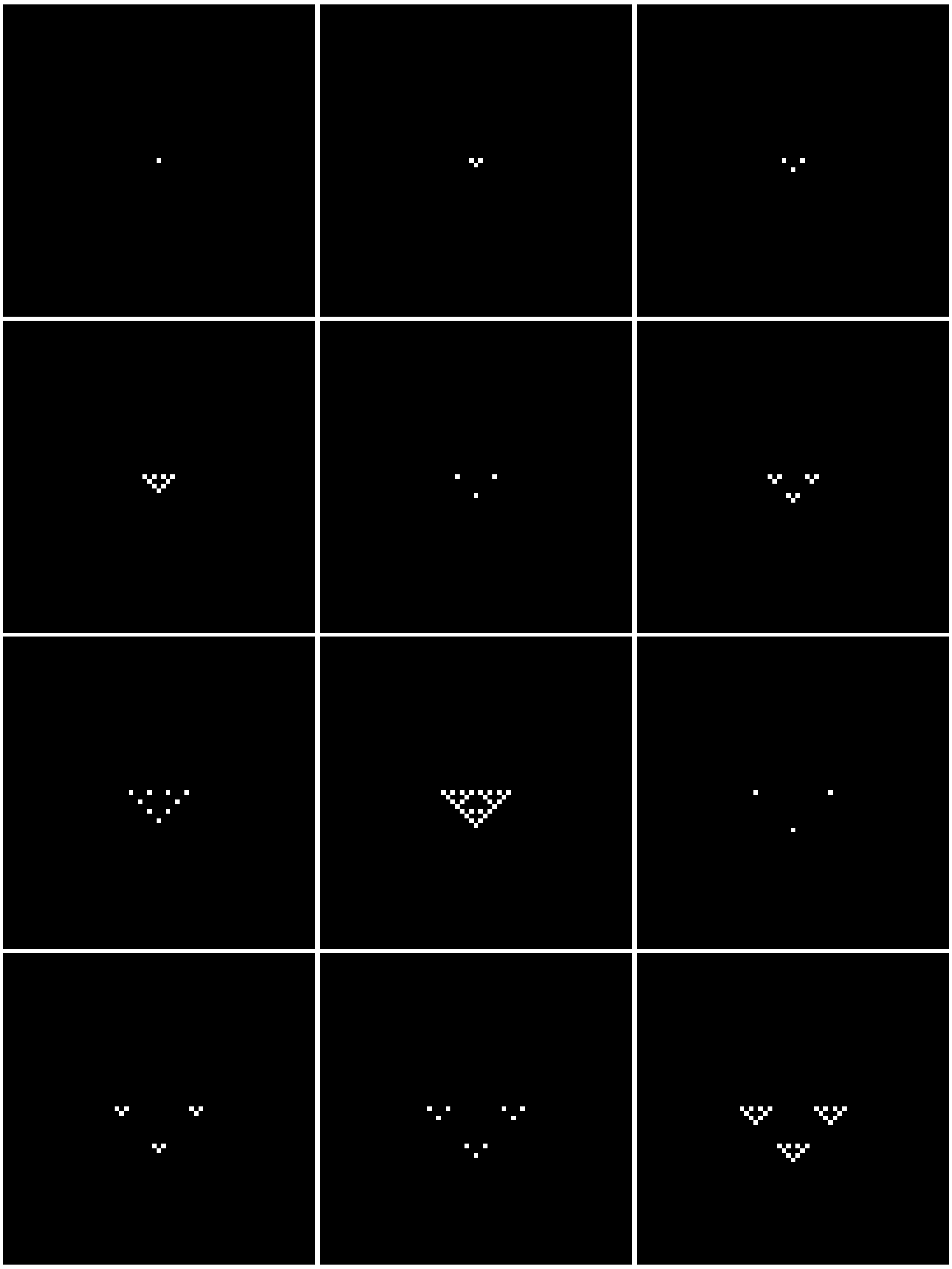


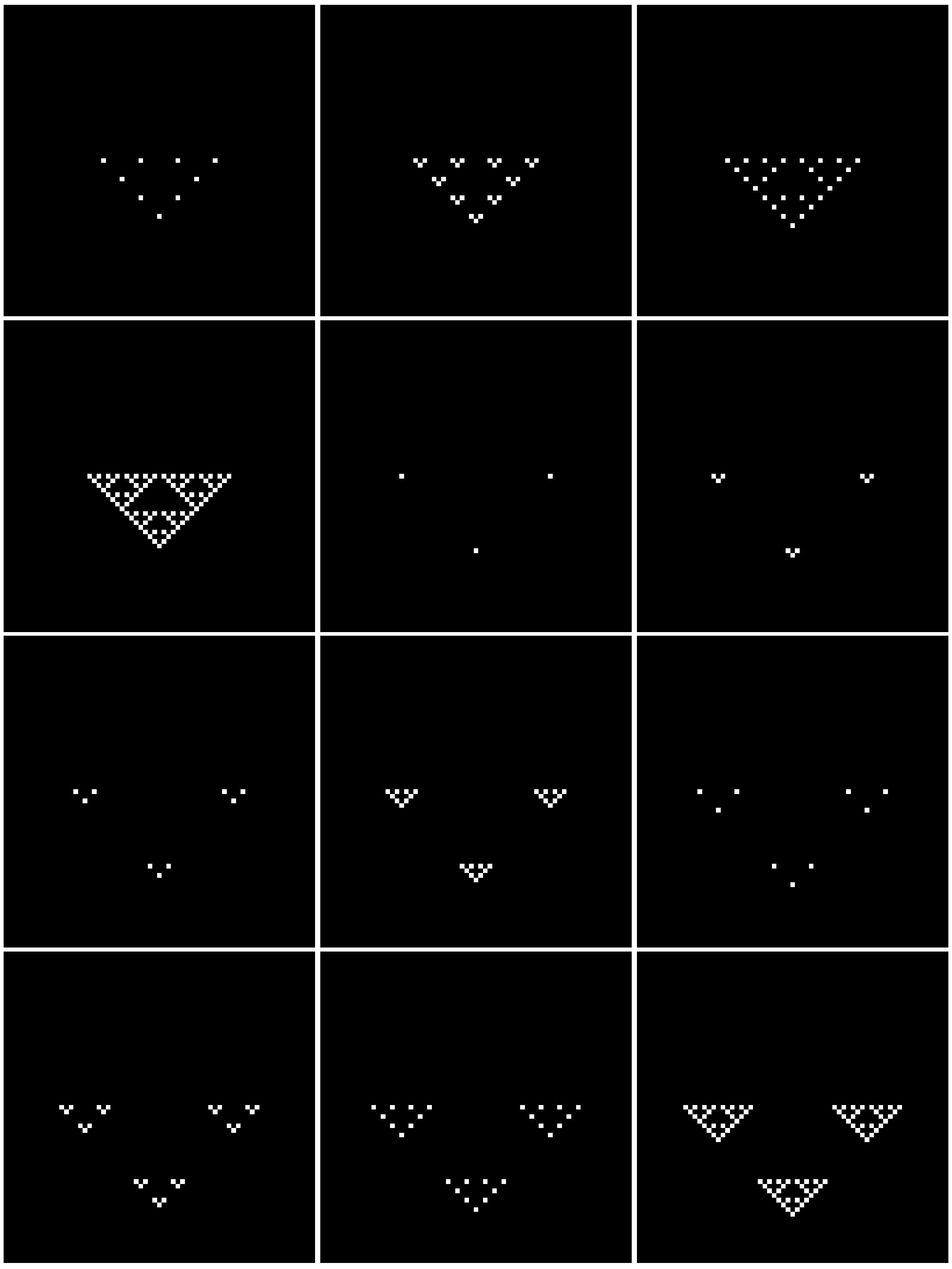


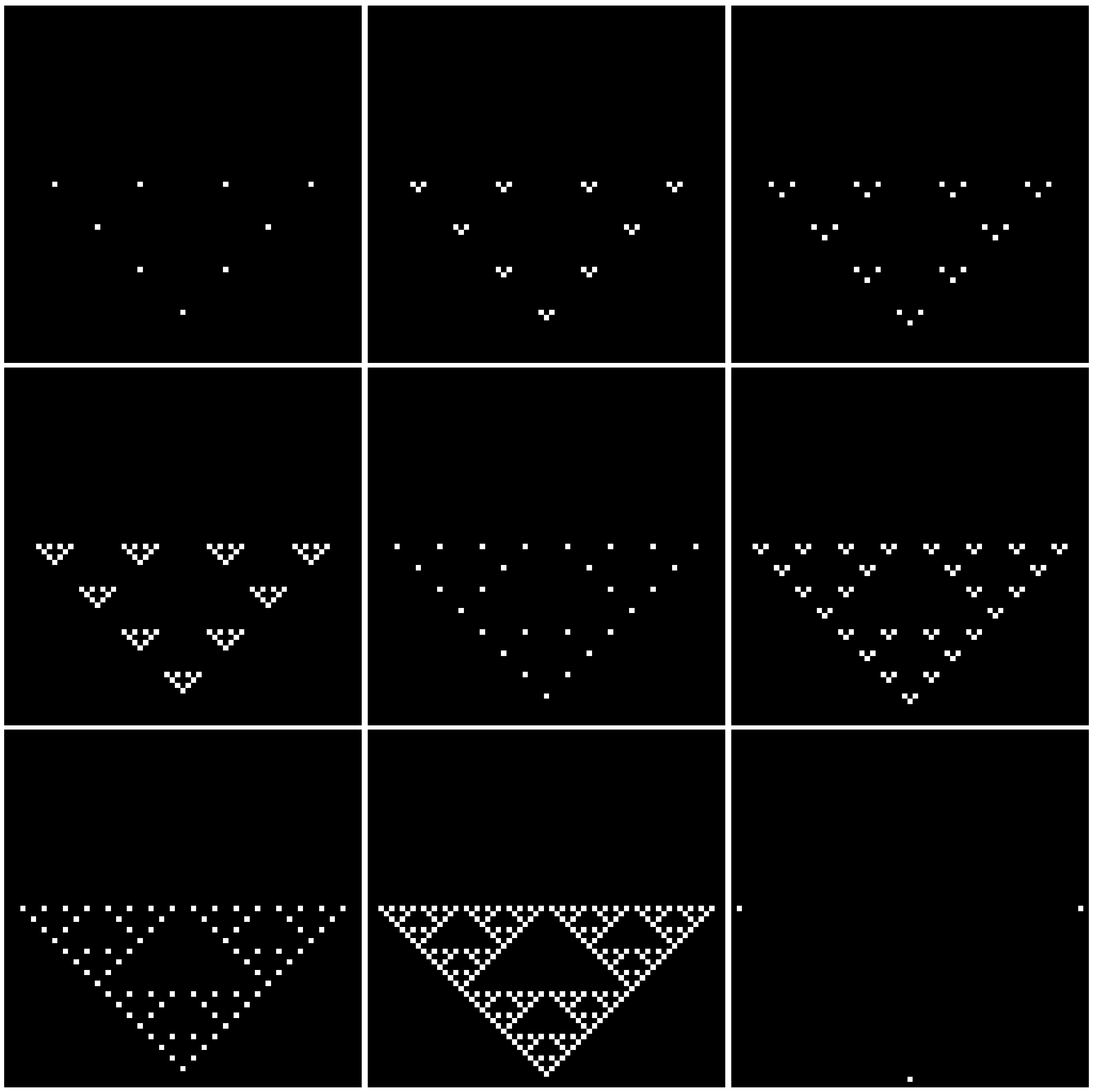
Appendix H

Grid data for the complete rules with scale=3.

The order should be read left to right, top to bottom, the same as text.







Appendix I

You can find a git repository containing all the code and data at https://git.cs.bham.ac.uk/projects-2022-23/kjs038.

I will not explain every function here, especially since each function has a comment in the code to explain it's purpose. I will explain how to run 4 of the major sets of functions. I have made an effort to make this as easy for others to run as possible, but there are some things you need to know first:

- I will use the term <ruleset> in the examples. When you see this, you should substitute it with the appropriate keyword based on what ruleset you want to operate with:
    - simple = totalistic ruleset
    - base = pure outer-totalistic ruleset
    - distinct = custom outer-totalistic ruleset
    - sample = complete ruleset

  E.g. to run 'function_<ruleset>(x)' for the totalistic set you would use 'function_simple(x)'
- When calling these functions, you first need to open outer_db.py, got to the very end of the file and replace the term 'pass' with the function call you wish to make. Then run the file.
- If you wish to access the databases separately, they use the following names:
    - outer_ca_simple.db = totalistic
    - outer_ca_base.db = pure outer-totailstic
    - outer_ca_distinct.db = custom outer_totalistic
    - complete_ca.db = complete

simulate_<ruleset>()
This will run the simulation for the whole ruleset and store the results in the database. For the complete ruleset it will instead take a random sample of size 4096 to use.
After this has run you can find the population sequence data in the pre_sequence table.
This will throw an error if you run it when the pre_sequence or growth tables already exists.

filter_<ruleset>()
This will take the population sequence data from pre_sequence to determine if each rule matches the pattern, and stores the results in the filter table.
Each rule has a Boolean value 'auto' associated with it that stores whether it matches, and 2 values 'base' and 'scale' that describe the nature of the sequence.
This will throw an error if you run it when the filter table already exists.

print(sum_<ruleset>(condition))
outputs the sum of the bit arrays of rules that match the condition (based on filter), and the total number of rules selected.
It automatically selects for auto=1, so conditions should only include base and/or scale.
E.g. 'base=1 and scale>1'

get_sequence_<ruleset>(condition)
prints the population sequence of all rules that match the condition.
Condition behaviour is the same as for the previous function.
Intended for the user to verify that a small ruleset have identical sequences, don't use for large sets.