**A Cellular-Automata Model of Skin Patterning**

**Rainer Hind**

**Supervisor: Dawn Walker**

**COM3600**

**05/12/2011**

This report is submitted in partial fulfilment of the requirement for the degree of Master of Science with Honours in Computer Science by Rainer Hind.

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Rainer Hind

Date: 02/05/11

Signature:

Abstract
Skin patterning in animals has long been a source of interest for biologists. Difficulties in studying the epidermis without interrupting its natural processes means empirical evidence can be hard to gather, especially when considering many of the processes occur at the embryo stage of development. A number of hypothetical models have been proposed, all of which emulate natural skin patterns to some degree.

In this project I will be implementing and exploring a published model of the skin patterning process and developing software to test and explore it. The software will be a cellular automaton, with the cells representing the pigment cells in the skin of animals. I will vary the governing rules and parameters of the model to investigate the effects on the patterns vary, and how the automaton can be used as a tool for study of the pattern development system.

**Table of Figures**

**Table of Listings**

# 1. Introduction

Because of the difficulty associated with non-destructively investigating the skin and its associated processes of development, there are a number of proposed hypotheses and models of how intricate patterns develop. Although the specific mechanics of pattern formation vary from model to model, the outcome is often remarkably similar, making it hard to deduce the underlying mechanism based on modelling alone.

Cellular automata are often used as simulations of a variety of natural systems, especially those in biology. Cellular automata are grid structures composed of individual 'cells'. Each cell can either represent a single biological cell, or a group of cells. Regardless, each grid-cell can communicate with a neighbourhood of cells around it. Depending on the state of its neighbours, the cell may change state. Changes occur at discrete time steps, and continue for as long as required. Despite the simple operation of each cell, emergent behaviour can be observed depending on the rules used[i].

The restricted local interactions and state-based behaviour make cellular automata ideal tools for modelling a multitude of biological processes. In the past, research has examined cellular automata as models of DNA development[ii], the tumour growth[iii] and even in HIV drug therapy[iv].

A software system will be used to simulate the process of skin patterning as an automaton. It will display the output on a grid, and enable the editing of rules and parameters. Initially, a local-interaction model from the literature will be implemented and investigated, potentially followed by a more complicated model using wider communication range.

The software will be developed to provide a platform upon which to experiment with skin patterning models. The program will allow the user to vary the initial conditions and rule parameters to produce a wide range of patterns, and facilitate further exploration of the model using the software as a learning tool.

Due to the time constraints imposed on the project, the primary aim will be to successfully implement a single model. If adequate time remains following a successful implementation, extensions to the project will be explored. These may include a graphical user interface (GUI), additional model(s), or additional software features.

Attempts at formulating models of skin patterning from the past will be explored in the next chapter, in order to arrive at a suitable candidate for implementation. After deciding upon the model, requirements, aims and objectives of the project will be explored. Available software and programming languages will be analysed to find a

suitable development platform for the simulation of the model, and potential pitfalls discussed.

# 1 Literature Survey

## 1.1 Turing's Chemical Basis of Morphogenesis and Reaction-Diffusion Derivatives

In 1952, Alan Turing's paper 'The Chemical Basis of Morphogenesis'[v] was published, and has been since recognised as influential research in the field of mathematical and developmental biology.

Turing's research revolves around the idea of two chemical messengers, known as morphogens, both with **different diffusion rates**, interacting with each other to produce the patterns visible in animal skin patterning.

He proposed a hypothetical ring of cells, with morphogens diffusing between them. For each cell in the ring, the rate of concentration change was modelled by two equations:

$$\frac{dXr}{dt} = f(X_r, Y_r) + \mu(X_{r+1} - 2X_r + X_{r-1})$$

$$\frac{dYr}{dt} = g(X_r, Y_r) + v(Y_{r+1} - 2Y_r + Y_{r-1})$$

Where:

r=1,..,n

X = morphogen X concentration

Y = morphogen Y concentration

μ = cell-to-cell diffusion constant for X

v = cell-to-cell diffusion constant for Y

f(X,Y) = rate of concentration increase for X

g(X,Y) = rate of concentration increase for y

Note: since cells are arranged into a ring, n+1 is equivalent to 1.

In his analysis, Turing describes six potential states of equilibrium arising from such a system. In the sixth case, stationary patterns form, and are described as Turing patterns; 'a kind of non-linear wave that is maintained by the dynamic equilibrium of the system'[vi].

The breakthrough aspect of Turing's research is the fact that instability can emerge from an otherwise stable system. This provided a mechanism to explain how patterns in nature could emerge from a stable, homogenous state.

Despite this groundbreaking discovery, it would be a further twenty years before the mechanism behind the patterns was properly recognised. In 1972, A. Gierer and H. Meinhardt postulated the requirement of short-range activation combined with long-range inhibition[vii]. Although Turing's equations satisfied this requirement, he would never realise it to be the key requirement of pattern emergence in reaction-diffusion systems.

The Gierer-Meinhardt model (Appendices 6.2) recognises one of the morphogens as an activator, which stimulates production of itself, as well as stimulation of the other morphogen, the inhibitor. The activator acts locally, with the inhibitor diffusing further across the cell space. The activator is autocatalytic, in that it stimulates further production of itself. It also stimulates secretion of the inhibitor, which acts to slow, and eventually stop production of the activator.

The active range of each of the morphogens is key to the model's behaviour. In order to produce Turing patterns, the model requires "a short-range positive feedback with a long-range negative feedback"[viii]. In practice, this means the inhibitor can diffuse faster, and over longer distances. The activator, on the other hand, is restricted to slow, local diffusion.

Since its original formulation in 1972, the Gierer-Meinhardt model, and other reaction-diffusion models have been used to model a wide variety of animal patterns[ix]. Research conducted in 2008 used a laser to kill pigment cells on the skin of zebrafish, which were shown to react as predicted by a Turing type reaction-diffusion model simulation[x].

Despite this, both Turing's model and the Gierer-Meinhardt model are yet to be proven biologically in relation to skin patterning. In order to prove such theories empirically, cellular communication mechanisms must be identified. Since patterns are most likely formed before hair is formed, empirical evidence would require investigating the epidermis of the embryo, an option not currently possible[xi].

## 1.2 A local activator-inhibitor model of vertebrate skin patterns

David Young, an American entomologist suggested an alternative theory for pattern generation in 1984. He cites three articles[xii],[xiii],[xiv] as evidence contradictory to Turing's proposed system of vertebrate pattern formation; instead, he suggests intercellular interaction is local only, with all communication restricted to a small neighbourhood around cells.

David Young's model is taken from Nicholas Swindale's research into the process of patterns in brain cortex[xv], and is a perfect example of how simulations of a biological

process can often be used to model another, possibly suggesting similar underlying development.

Young's simulation begins with a grid of randomly distributed coloured and uncoloured pigment cells. Young refers to these as differentiated cells (DCs, coloured), and undifferentiated cells (UCs, colourless). Each DC releases two types of morphogens: an inhibitor and an activator. The former stimulates nearby DCs to dedifferentiate, and the latter stimulates nearby UCs to differentiate. Both of these substances are able to diffuse across the grid by some limited distance, with the inhibitor having a greater range. The uncoloured cells have no effect on any surrounding cells. Eventually, equilibrium is reached between the two morphogens and the pattern no longer changes.

The process of the morphogens identifying the states of the surrounding cells is explained using a 'morphogenetic field', with activation modelled by a positive field, and inhibition negative. As distance from any given DC increases, the morphogenetic field strength will decrease. This field affects each cell, and controls their differentiation.

Differentiated cells on the 2D grid are initially randomly distributed. For each cell in position R, the sum field value is found by adding the values from all DCs in the specified neighbourhood. The net field value for each cell is given by:

$$\sum_i w(|R - R_i|)$$

where:

R = current grid point

$R_i$ = surrounding DC points

w(x) = morphogenetic field value at distance x from DC

If the value of w(x) is greater than zero, then position x is a DC. For negative net field values, the point is set to a UC. For neutral field values, the cell maintains its previous state. This is calculated for each cell on the grid, and then the new states applied at each discrete time step. The process of field value summation and state change is repeated until equilibrium is reached and the states no longer change. Young suggests five iterations are usually sufficient for a stable pattern to emerge.

By using discrete cell positions and using thresholds for pigmentation state Young has converted his model to a cellular automaton. Despite the discretised model, Young obtains very similar to results to those of Swindale, 1980[xvi], which used a continuous model. This suggests the cellular automaton retains "essential features required for exhibiting self-organization phenomena"[xvii]. Furthermore, the model is not sensitive to the initial distribution of the DCs.

Depending on the initial parameters and the boundary conditions provided, Young's model is capable of producing various vertebrate patterns. In the following image, only the inhibition field value is changed, yet each pattern varies fairly significantly.

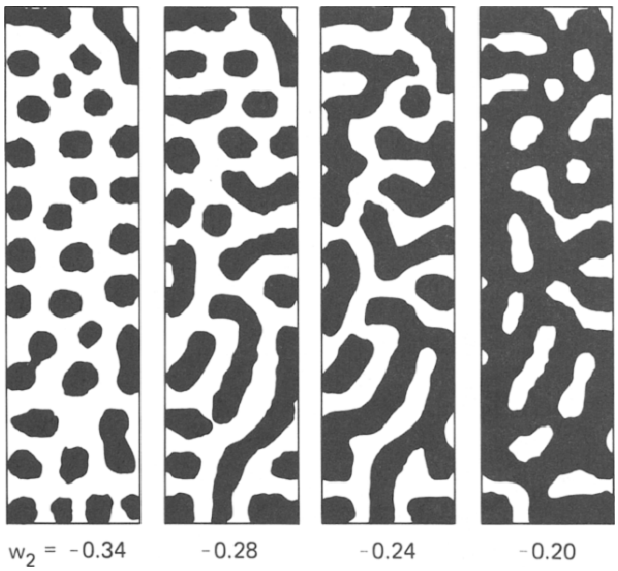


**Figure 1-1 Patterns produced using Young's model demonstrating both spots and stripes**[xviii]

Another key feature of Young's model is also demonstrated in Figure 1-1. Notice how in the first frame, where inhibition is strong, the coloured cells are unable to connect, and so produce a spotted pattern. In the final frame, inhibition is weaker and so the differentiated cells join and form stripes. Young cites this effect as further evidence for the validity of his model, suggesting a "useful theory must be able to generate both patterns" as both are probably generated by similar biological processes. In the extremes, altering the inhibition value will result in lone, pigmented cells in a plain background (very strong inhibition), or an entirely pigmented area when inhibition is weak.

Young also provides a method for generating directional stripes, like those seen on zebra skin. This is achieved by using anisotropic morphogenesis. In practice, this means the activator acts in parallel to the stripes, and the inhibitor acting perpendicularly, resulting in an elliptical field area.

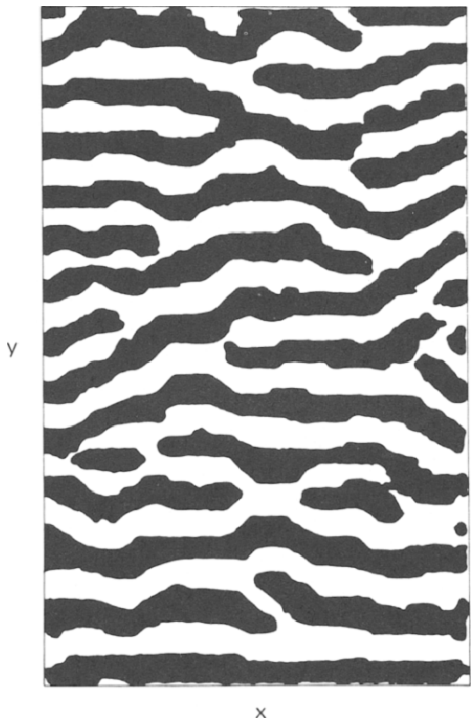


**Figure 1-2 demonstration of stripe pattern in Young's model**

Compared to Turing's model, Young's has a number of benefits. As mentioned, there is some evidence suggesting that interaction between cells is local only, making the cellular automaton the ideal modelling tool. Furthermore, Young's model is able to replicate patterns similar to those of a zebra, with branching and termination of coloured areas. The model is also not necessarily limited to diffusion based interaction of morphogens; Young also suggests "direct cell contacts or with local elastic strains in the underlying tissues" as possible underlying mechanisms, however this does not affect the behaviour of the model.

## 1.3 Morphogenetic Evolution in the Young Model

An extension to the Young's model is explored in the 2004 paper, 'Evolving Morphogenetic Fields In The Zebra Skin Pattern Based On Turing's Morphogen Hypothesis'[xix] by C.P. Graván and R. Lahoz-Beltra.

The paper uses Young's research as the basis for their model, but also implements a genetic algorithm that simulates the evolution of the two morphogens. The hypothetical genes considered represent the diffusion distance and the field value of both morphogens.

Initially, both morphogens are specified equally, in that neither is designated an activator or inhibitor, with these being attributed as a natural result of the genetic algorithm.

In order to analyse the effectiveness of the gene configuration at each generation, the algorithm computes the fitness function by considering the number of contacts

between the different coloured cells, to decide if it is a spotted or striped pattern. Depending on the degree to which spotting or striping occurs, the fitness value will range for the different genes. At each generation, recombination and mutation have a chance of occurring to further improve results.

Unfortunately, the paper suggests that the results "indicate that the evolution of morphogens does not depend on the genetic algorithm and similar results are obtained in the presence or absence of mutation"[xx] making it a poor candidate for implementation in the simulation.

## 1.4 Stephen Wolfram's A New Kind of Science

In 2002, Stephen Wolfram's bestseller 'A New Kind of Science'[xxi] was published. Following on from his work in the mid-1980s, the book offers a comprehensive analysis of cellular automata, and their applications in biology and other scientific disciplines.

In Chapter 8, Wolfram discusses biological pigmentation patterns as products of "processes whose basic rules are extremely simple"[xxii], noting mollusc patterns' similarity to simple 1D cellular automata explored earlier in the book[xxiii]. Wolfram explains how a mollusc's shell "in effect grows one line at a time, with new shell material being produced […] at the edge of the animal inside the shell"[xxiv]. A range of example 1D automata are shown, all of which bare "striking similarities to those seen on mollusc shells" [xxv]. The method for rule selection in the organism is attributed as a random choice, picked from a collection "of the simplest possibilities"[xxvi]. The fact that a layer of skin hides most mollusc patterns throughout their life is presented as evidence against the traditional natural-selection argument for pattern choice[xxvii].

Realising that the mollusc is an unusual example for pattern development in the sense that it grows one line at a time, Wolfram turns his focus to more common patterns that form across the entire skin simultaneously. In order to simulate this in cellular automata, a two-dimensional model is required. Wolfram's model suggests that each new cell will tend towards the average colour of nearby cells, and the opposite of far cells, mimicking the short-range activation, long-range inhibition interaction of reaction-diffusion models[xxviii].

Using these simple rules, and varying the weighting of cells at distances 1, 2 and 3 allows for a wide range of results to be produced (figure 2.3).

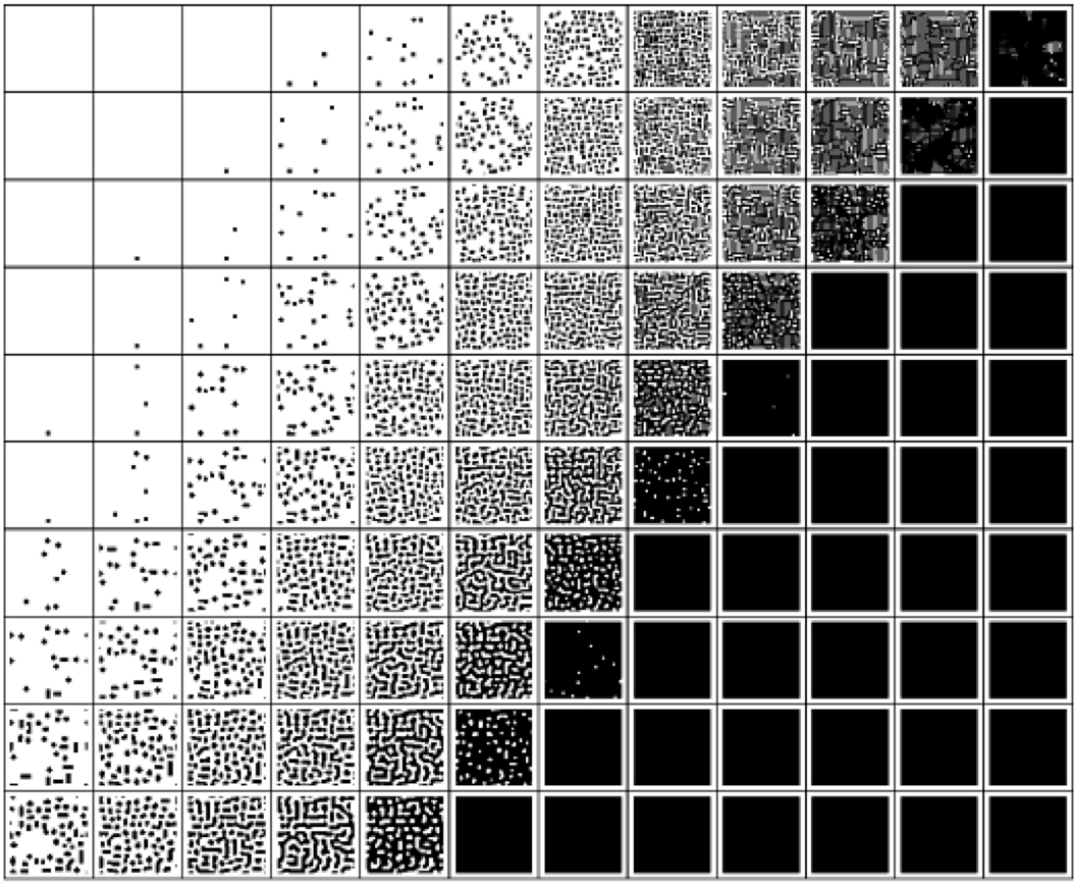

**Figure 1-3 Patterns generated using Wolfram's model. In each example, the cell itself and the nearest neighbours all have weighting 1. "Weights vary from -0.9 to 0 down the page for distance 2, and from -0.7 to 0.4 across the page for distance 3."** [xxix]

The patterns generated clearly have a resemblance to patterns seen on animals. Like Young, Wolfram acknowledged the need for directionality (representing stripes). To implement this behaviour, he introduced different weighting for the horizontal and vertical directions. This leads to the production of stripes reminiscent of those seen on animal skin (fig. 2.4).

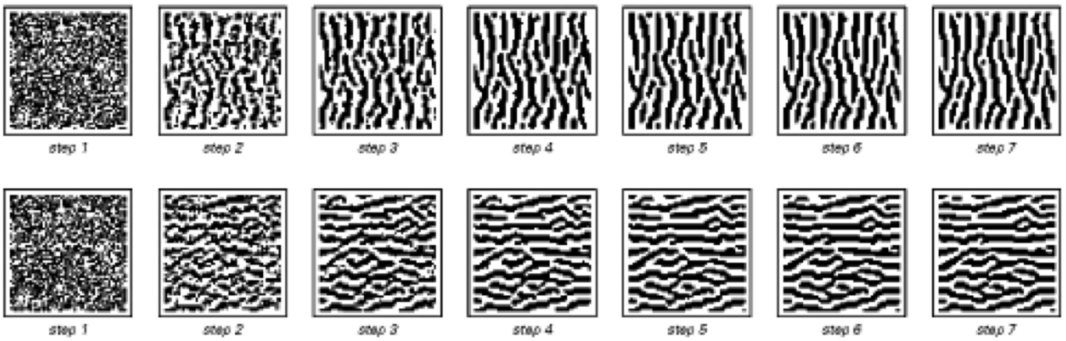


**Figure 1-4 "Examples of rules in which cells in the horizontal and vertical directions are weighted differently. In the first case, cells at distances 2 and 3 only have an effect in the vertical direction; in the second case, they only have an effect in the horizontal direction. The result is the formation of either vertical or horizontal stripes"** [xxx]

# 2 Requirements and analysis

Throughout the project there are a number of targets for implementation. Some of these are essential to the outcome of the investigation, and as such are deemed primary. Others would be desirable, but may not form part of the final system and are designated secondary aims.

## 2.1 Primary Aims and Objectives

### 2.1.1 Successfully implement a model from the literature in software

The primary aim of the project is to successfully implement a model from the literature. Based on the review, I have decided to implement David Young's model. Its local-range, discrete interactions map excellently to cellular automata (CA), and it produces a wide range of patterns. It may be possible to implement a further model if development of the Young simulation progresses well.

### 2.1.2 Develop the software as a tool to investigate model behaviour

Once the model is implemented, the software will be developed to allow it to be used as a tool for investigating the properties of the model. This will include features such as modifying the rules and parameters to allow the user to investigate the emergent properties.

### 2.1.3 Replication and analysis of patterns exhibited in the research

Identifying emergent structures and behaviour in the CA is another primary aim in the project. Properties such as striping or spotting are examples of such behaviour, and may provide an interesting basis for further investigation. Such structures can be found in animal patterns, as well as in the literature for existing models. It will be a primary aim to ensure that such patterns are properly replicated by the software.

## 2.2 Secondary Aims

### 2.2.1 Seek to develop aspects of the model identified in the investigation

After fully investigating the model, certain behaviour may be identified which warrants further analysis. It may be possible to improve upon the model, or incorporate further behaviour from another model or piece of research. The implementation of the genetic algorithm in 'Evolving Morphogenetic Fields in the Zebra Skin Pattern…' in section 1.3 of the literature survey demonstrates one such improvement to a core model. Other useful features which would aid the user in investigating such models may be identified, and could be implemented after the core model. For example, a means of implementing custom defined patterning models or rules.

### 2.2.2 Implement a further model

Once the initial model is successfully implemented, it may be possible to implement a further model and investigate its behaviour. This may involve implementing a short-range model, or a model based upon longer range of cellular interaction.

## 2.3 Testing strategy

Due to the visual nature of the model, testing quantitatively can be difficult. Instead, much of the testing will rely on visually comparing results to those presented in the literature. Although limited, there are several images of patterns generated by the Young model in the research. The model is also capable of generating multiple types of patterns, including spots, stripes, and directional stripes. If the software can successfully replicate these patterns, it is likely the implementation is correct. If problems arise and the published results are insufficient to verify the implementation, it may be necessary to compare the patterns produced to real-world skin patterns.

To improve the likelihood of a successful final implementation, software components will be tested individually in unit tests, to ensure their correctness. This will involve verifying their output, and ensuring it conforms to what's expected. By building on trusted functions from the start, the scope for error is significantly reduced, and the source of any error is much easier to locate.

At each stage of integrating these components, they will be tested to confirm the interaction between them is working correctly.

If a graphical user interface (GUI) is implemented, the various components and inputs will be tested for validity. For example, the values accepted by the system will be checked, as well as the veracity of on screen information and results. It will also be important to ensure that the interaction between the interface and the program code is implemented correctly, and all functions are called correctly.

By testing both the underlying components and the final output, it should be possible to ensure a valid model implementation.

## 2.4 Choice of Software

To implement a model of skin patterning and investigate its properties, the simulation software must be decided upon. This can be split into two categories: 'off-the-shelf' software, in which an existing program is used, and bespoke software, where the system is developed especially for this purpose.

### 2.4.1 Existing Software

### 2.4.2 Cellab (Rudy Rucker and John Walker)

http://www.fourmilab.ch/cellab/

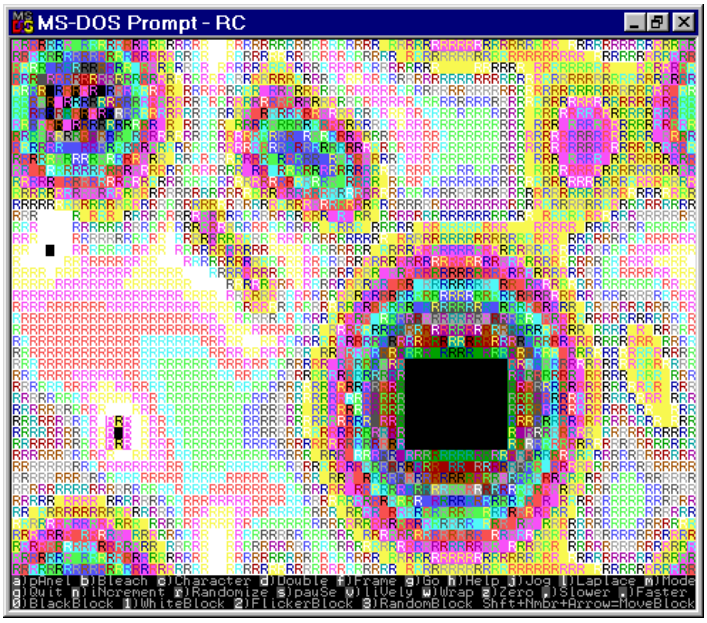


**Figure 2-1 Example output from Cellab**[xxxi]

In 1988, the co-founder and then chairman of Autodesk, John Walker, hired Rudy Rucker, a mathematics professor and computer scientist from San Jose State University with an interest in cellular automata[xxxii],[xxxiii]. Over the next two years, Rucker and Walker would develop 'Rudy Rucker's Cellular Automata Laboratory', a popular CA application that formed the first part of the Autodesk Science Series[xxxiv]. Released in June 1979, the software remained available until 1994. After failing to gain renewed publisher interest, the software was repackaged into a Windows application and distributed as freeware.

Arguably the most extensive and customisable CA software commercially available, Cellab is probably the best existing software candidate for use in this investigation. Described by the authors as the "fastest and best 2D cellular automata program ever written", it does provide a number of flexible options essential for modelling the literature, as well as an extensive user guide (converted from a 256 page book released with the original software[xxxv]).

New rules can be specified in C, BASIC or Pascal, and the program has also been updated to allow rule definition in Java. The user can specify colour palettes, and the program comes with a large number of existing rules from a wide range of fields, including chemical reactions, heat wave manifestation[xxxvi] and turbulent flow[xxxvii].

Despite this customisability, there are a small number of key features that detract from the appeal of Cellab. Firstly, and most detrimentally, is the limited neighbourhood of each cell. Interaction is restricted to the eight nearest cells, preventing implementation of short-range activation/long-range inhibition. More advanced features, such as modifying the evaluation rules, are done in assembly language, or as DLL files written in the C language. When contrasted with the corresponding process if writing bespoke software, this is obviously a major obstacle,

and would likely impede development time and flexibility. Furthermore, Rucker describes the software as "semi-obsolete" as a result of its incompatibility with Windows 7, requiring the user to run the program from within a virtual machine[xxxviii]; a further inconvenience.

Although Cellab provides a rich toolkit for investigating CAs, the aforementioned disadvantages eliminate it as a possible software platform for the research. More customisability is essential to implement more advanced functionality from the models.

### 2.4.3 Five Cellular Automata

*http://www.hermetic.ch/pca/pca.htm*

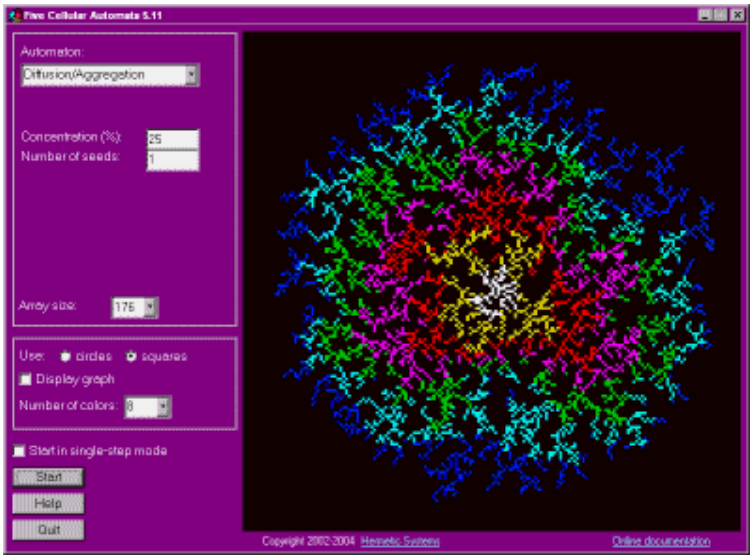


**Figure 2-2 Example screenshot from FCA**

'Five Cellular Automata' is a freeware application capable of simulating five separate automata. It includes a GUI allowing the user to configure a small number of basic parameters, including concentration and seed count. Images of the graphical display can be saved at any step in the cycle, by pausing the simulation. Aside from the limited parameters visible on the main screen, there is no option to modify the rules dictating the behaviour of the automaton. As such, this application is not appropriate for modelling the varied conditions put forward in the literature.

### 2.4.4 CellLab (Paras Chopra)

http://paraschopra.com/software/celllab/

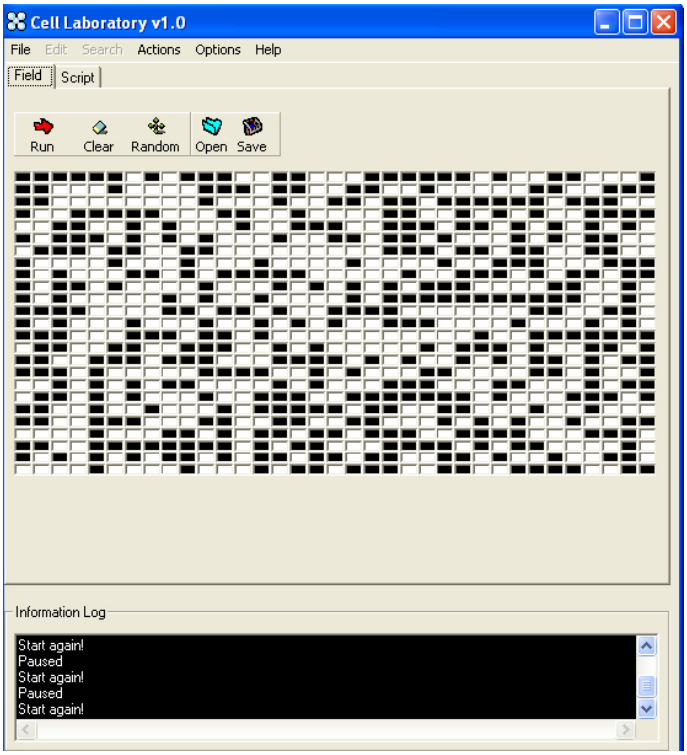


**Figure 2-3 Screenshot from CellLab, from readme**

CellLab (note the additional 'L') is another tool for investigating 2D cellular automata. Although less extensive than Rudy Rucker's software, CellLab does permit custom rules and includes a number of other options and controls, including zoom, boundary conditions and pattern identification.

Rules are defined in VBScript only, and interaction is again restricted to nearest neighbour. The source code is accessible, but files have no comments and documentation is limited to a short readme[xxxix], making modification unnecessarily complicated. As a result, CellLab is not a viable candidate for implementation.

These applications are not the only CA software, but represent the extent of their functionality and limitations. Software ranges from simple applets with fixed rules, to full pieces of software with modifiable parameters. Although many of these capture some desired functionality for simulating the models in the literature, most lack the extensive customisability desired. Graphical outputs are often restricted, and specifying complex rules can be difficult or not possible. Furthermore, many of the programs include unnecessary features and configuration options that can complicate an otherwise simple implementation. This general non-specificity is what eliminates off-the-shelf software from being a viable candidate for modelling the literature research.

### 2.4.5 Bespoke Software

In order to implement a bespoke system, the development language must be decided upon. The languages decided on as possible candidates are Java, MATLAB and Python. All three provide the same basic functionality required to implement the system in some fashion, but have a number of differentiating features that are potentially useful in the development of the software.

All three options are high-level languages, which means they provide a level of abstraction from the machine's binary instructions, therefore 'requiring little knowledge of the computer on which it will be run'[xl]. They are also object-oriented, allowing the system to be modelled "as a set of objects which can be controlled and manipulated in a modular manner"[xli]. Both of these factors allow a faster development time, as less knowledge of underlying mechanisms is required. Tested modules can be combined to achieve higher functionality, reducing scope for error and segmenting code into easily understood, logical components.

All three languages also provide the same basic features and control structures required to implement the system, and all come with comprehensive documentation and user base, with a large support base of existing code. Essentially, the program can be created in either, and so only main features that differentiate the three will be considered.

### 2.4.6 Sun Java[xlii]

Java is the first possible development language for the modelling software. The most notable feature of Java is the fact that it is compiled into Java bytecode, which is then interpreted by the Java Virtual Machine (JVM) for the system it runs on. This allows the code to be run on any platform for which there exists a JVM.

Thousands of existing packages and libraries are available for incorporating additional functionality into the language, including mathematics, statistics, graphics and database tools.

Included in Java is the Swing toolkit, a useful API for creating graphical user interfaces to interact with the program. Just like the code itself, the Swing interfaces are platform-independent, and will run on any platform that supports the JVM.

### 2.4.7 Mathworks MATLAB[xliii]

MATLAB is another high-level language with object-oriented capabilities, more focussed on technical and scientific computing than most languages, including Python and Java. Bundled with MATLAB is a wide array of existing functions for various domains of mathematics, and an enormous range of 'toolboxes' available to extend the functionality to more specific classes of problems.

Many of MATLAB's standard functions support matrices and vectors natively, enabling them to be operated upon without the use of loops. This is ideal for working with a CA, as it allows each cell to be updated with a single call of a function.

Arguably the most valuable feature in MATLAB is its comprehensive graphics functions for visualising data. This includes a huge amount of graphing options and display parameters, and provides simple methods for generating visually impressive graphics from data. Complementary to the handling of matrices and vectors is the ability to map matrices directly to images, vastly reducing the complication of converting state-array to image, and updating the image at each time step.

If a GUI is added to the system, basic controls can be easily added without any external libraries. Also bundled with the MATLAB environment is the GUIDE, a layout manager for designing and organising graphical interface components, which may be an advantage when implementing the literature models.

Unfortunately, MATLAB is the only language of the three discussed which is not free to use, and is particularly expensive for non-students.

### 2.4.8 Python[xliv]

Python offers a relatively similar environment to Java. Dozens of libraries exist to aid development, including graphics, GUI and vector operations[xlv]. Numpy and Scipy are two such libraries, providing functionality for a large number of scientific computing applications, as well as adding the ability to treat matrices as fundamental types. This is a feature shared with MATLAB, without the price tag. Matplotlib is a "2D plotting library which produces publication quality figures", and aims to emulate MATLABs interface[xlvixlvii].

A more specific-purpose package is CAGE, a cellular automata simulation engine providing much of the functionality necessary to implement CA models[xlviii]. Published documentation is extremely limited, but it may be an aid to developing the final CA software if Python is used.

All three languages include features advantageous to development. Java and Python provide expandable, general-purpose environments, more than capable of implementing the model automata. In particular, Python, combined with SciPy, implements much of the core scientific computing functionality in MATLAB, making it a strong candidate for implementation.

Despite this, MATLAB includes most of the key general-purpose features of Java and Python, but also provides a richer mechanism for graphic display in a more focussed scientific computing environment. Almost all functionality required is provided out-of-the-box, resulting in a simpler setup process and end product. The university provides a student licence, so cost of the software will not be an issue, eliminating the largest disadvantage explored in the analysis. As a result, MATLAB will be the language in which the software will be developed.

## 2.5 Requirements

In developing the software to simulate the model, certain requirements must be satisfied in order to reach a successful implementation. As with the aims of the project, these can be separated into essential requirements that are core to the running of the simulation, and desirable requirements: features that would benefit the system but are not essential.

### 2.5.1 Model-Based Requirements

**Customise model (Essential)**

In order to increase the flexibility of testing and possibly incorporate numerous models, it must be possible to alter the rules governing the behaviour of the cellular automata. This can be achieved using modular programming practices, making it possible to alter certain behaviour without affecting other unrelated aspects of the system.

**Edit parameters (Essential)**

It should be possible to modify model parameters easily, to investigate their effects. Unlike modifying the model, these changes will be minor, and not alter its general behaviour. For example, changing the activation or inhibition field values in the Young model; although this affects the pattern generation, it is still the same underlying model.

At a basic level, this can be partially achieved by using variables in the program to represent common parameters, allowing them to be easily modified without affecting other code or having to repeat changes throughout.

**Vary initial states of the automata (Essential)**

The initial states of cells in the automata should be variable. A means of generating a random initial distribution is essential, but ideally this will also include options to manually set input states, from a file for example.

**Represent short-long range interaction (Essential)**

Unlike much of the existing software explored in the previous pages, the program developed must be capable of representing short-range activation, and long-range inhibition. This means cells must be capable of interacting beyond their nearest neighbours.

### 2.5.2 Software-Based Requirements

**Command Line interface (Essential)**

Initially, the command line will be used to run and test the application. Even without the use of a GUI, parameters should be modifiable at the command line.

**Display output (Essential)**

At any stage in the running of the automata, the visual representation should be viewable. This requires a grid of cells that updates at each time step, visible on screen.

**Speed (Essential)**

The program should run at a reasonable speed, capable of representing the formation of the pattern through multiple generations in a reasonable space of time. In MATLAB, vectorising functions can increase speed and since most of the mathematics involves matrices this shouldn't be a problem. Furthermore, MATLAB's core maths functions are processor-optimised for the system it runs on, resulting in faster execution speed than equivalent code in other languages[xlix].

**Adaptability (Essential)**

Modifying behavioural aspects such as the rule set of the model should be relatively easy. By implementing good programming practices, such as compartmentalisation of code into logical subsections, modifications can be made without disturbing other functionality. This flexibility is what makes a bespoke solution ideal, and will improve ease and efficiency of testing.

**Store images of simulations (Desirable)**

To analyse the behaviour of the CA under varying conditions, a visual record of the output must be stored. This may just be of the final output, after the cells have reached equilibrium, but ideally the system should be able to store images in time increments to show the development of the pattern. This would provide a record of the simulation which could demonstrate aspects of the pattern formation otherwise unnoticed. A system of automated image generation would also improve testing efficiency, as it would not need to be done manually.

**Modify running speed (Desirable)**

Being able to modify the running speed of the program could be helpful in identifying behavioural characteristics which may be hard to notice at regular speed, allowing more detailed analysis of emergent properties. If this is not implemented, the simulation can be re-run using stored intermediary states and therefore is not essential.

**Control basic operations with GUI controls (Desirable)**

Simple actions, such as pausing the running of the application, modifying the speed and capturing screenshots of the current display should ideally be controlled by simple on-screen buttons. This will improve usability, and therefore efficiency.

## 2.6 Potential Problems

When implementing an existing model, there are a number of potential pitfalls that must be considered. It may be impossible to implement solely using the literature if the author has not provided an adequate specification. Even if the model is

implemented it may not be possible to replicate published results, for example if the parameters are incorrect or incomplete in the definition of the model.

For larger automata, there may also be issues with speed of execution. As mentioned previously, this may be improved by further vectorising code, a benefit of using Matlab.

In order to minimise this problem, the results of the CA will be tested to ensure model behaviour is correct, by comparing results to those documented in the literature. If any inconsistencies occur, they can be identified immediately and further time is not wasted by exploring an incorrect implementation of the model.

# 3 Design Approach

Due to the limited time constraints imposed on the project, as well as the novel nature of the software, predicting the degree of implementation which will be achieved is difficult. To avoid running out of time near the end of the project with an incomplete implementation, a traditional waterfall development lifecycle will be avoided. In such a scheme, each stage is completed in one pass. The design for the entire software is created, and then development continues to the implementation and finally testing. Once completed, each section is usually fixed, preventing reconsideration of design decisions.

Instead, a more iterative approach will be adopted. An initial prototype will be developed in which only the core Young model requirements will be implemented. The prototype can be used to identify any issues early on, and design reconsiderations can be made. Rectifying such problems at an early stage prevents disruption of subsequent deadlines, and the incremental design allows for a far more flexible development process.

Following a successful prototype implementation, focus will shift towards improving the final software, and implementing secondary requirements. Any problems with the prototype can be resolved and if an area for improvement is identified during the initial development it can be developed afterwards. Furthermore, aims and deadlines can be set dynamically, with more up to date expectations depending on what is already achieved.

# 4 Primary Requirements and Initial Prototype

The first challenge in the project is to implement the basic Young model. If this is successful, subsequent improvements and desirable extensions can be explored. This chapter will focus on the design, implementation and testing of the initial phase of development.

## 4.1 Design

In order to implement the core functionality of the system, a means of representing an automaton is required. Figure 4-1 demonstrates the required behaviour necessary to fulfil the primary requirements.

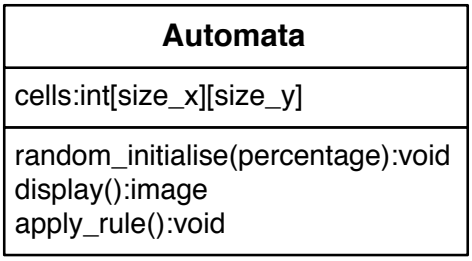


**Figure 4-1 Class diagram of required automata behaviour**

The grid size is another attribute of the automata; however this is an inherent attribute of the cell states array.

Due to the lack of unique attributes of the automata class, it can be represented as a matrix without the use of a class. The nature of the Matlab programming language allows us to define functions that act on a matrix without having to define a new class unnecessarily. As a result, the following functions will be required:

### 4.1.1 random_distribution(size_x,size_y,percent)

This function will take three parameters representing the size of the grid to be generated, and the percentage of cells in that grid to be set to 'alive'. The function will return a matrix containing a random distribution of dead cells (UCs) and live cells (DCs) (of the percentage specified).

### 4.1.2 display_grid(grid)

Taking a binary matrix of cell states, this function displays the cells on screen. Each undifferentiated cell should be displayed in one colour, with the differentiated cells in another to allow visualisation of the cell data. Because of Matlab's data visualisation capabilities this should be very simple.

### 4.1.3 apply_young(grid,(model parameters))=new_grid

The implementation of Young's model for pattern development will be coded in this function. Taking a binary matrix of size MxN representing cell states at a given time step, t, as its parameter, the function will return a MxN matrix representing the cell states at time step t+1. The function also accepts the model parameters (optional), satisfying the requirement to edit model parameters easily. The algorithm will be based on the following pseudo-code, adapted from Young's research paper[1]:

```
For each grid point at position R:
      Field value of all nearby DCs summed (∑ᵢ w(|R − Rᵢ|))
            If ∑ᵢ w(|R − Rᵢ|) > 0 then point at R set to 1 (DC)
            If ∑ᵢ w(|R − Rᵢ|) < 0 then point at R set to 0 (UC)
            If ∑ᵢ w(|R − Rᵢ|) == 0 then point at R remains unchanged
```

**Listing 1 Pseudo-code for Young algorithm**

---

### 4.1.4 run_simulation(time_steps, (model parameters))

This top-level function will allow the user to specify a number of time steps to be simulated, and optionally the specific parameter values to use. For each generation it will display the automata grid, thereby reducing the number of function calls required to run a simulation, improving the ease of use.

The interaction of these functions is documented in Figure 4-2.

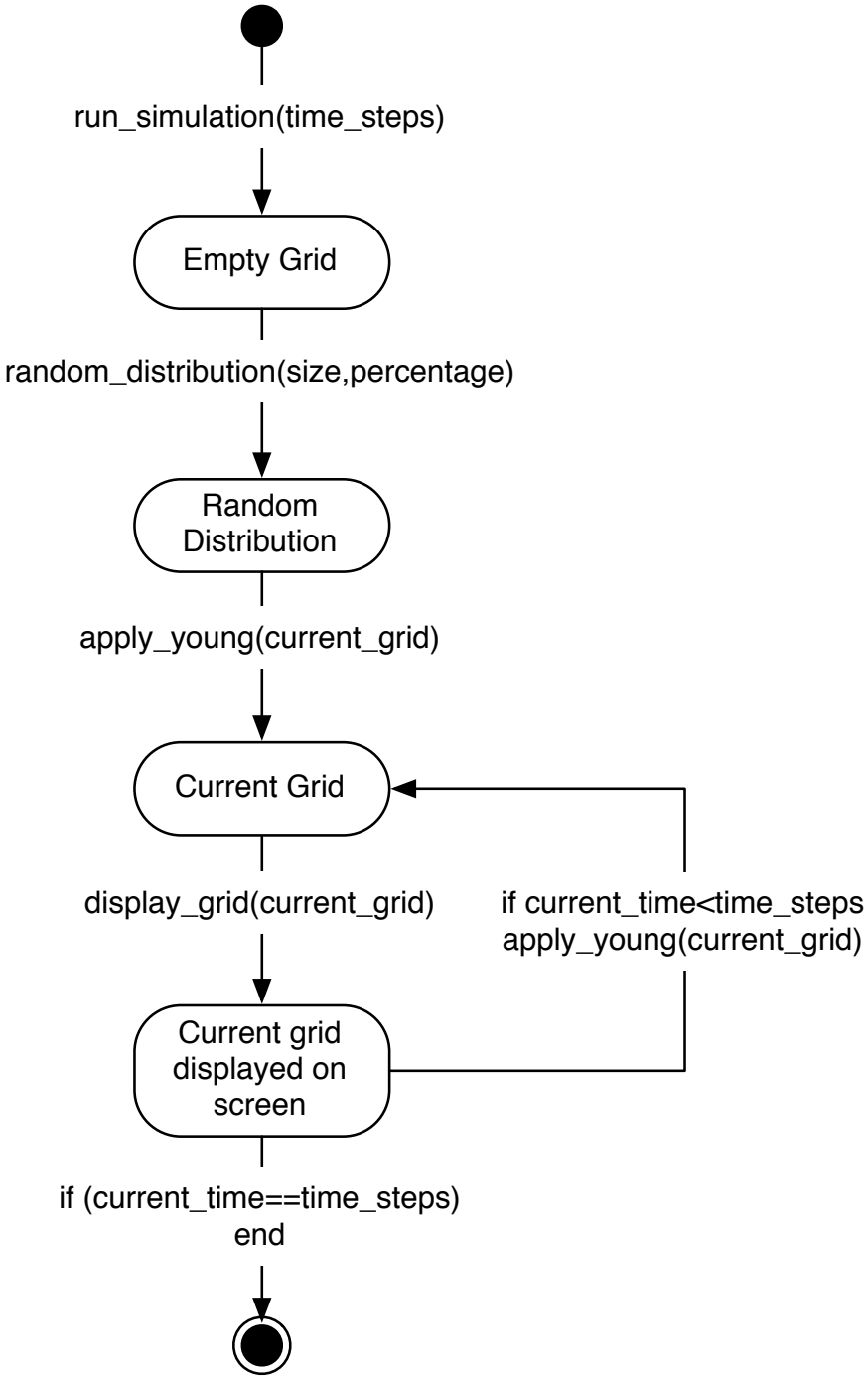


**Figure 4-2 Activity diagram showing control process in prototype**

## 4.2 Implementation

### 4.2.1 random_distribution(size_x,size_y,percent)

The first step in the initial implementation was to create the method to distribute live cells randomly in the initial grid. This would provide a starting point to view the

progression of the cells through the time steps. It was possible to implement this in one step due to Matlab's flexible indexing and matrix operations (Listing 2).

```
grid=(rand(size_y,size_x)<=prob);
```

**Listing 2 Concise method of generating a random initial distribution (from Prototype/random_distribution.m)**

### 4.2.2 display_grid(grid)

Because of Matlab's advanced imaging and data visualisation capabilities, implementing the display method is simple. Firstly, the colour scheme is set, and then the image displayed directly.

### 4.2.3 apply_young(grid, act_range, inh_range, act_field, inh_field)=new_grid

**The initial implementation of the Young model uses a nested for-loop to consider each cell from the grid in turn, and calculate the summed field value for its neighbourhood.**

Listing 3 describes the process used to sum the activator/inhibitor fields in Prototype/apply_young.m.

- *For each cell i at co-ordinate (x,y) in grid*
  - *Generate a matrix, position, the same size as the grid*
  - *Set the cell at (x,y) in position to 1 (to represent the current cell location)*
  - *Generate same size matrix, distances, containing the euclidean distances from current cell to every other cell using bwdist(position).*
  - *For each morphogen area (activator and inhibitor)*
    - *(Sum of grid) where (distance < radius) and (grid value == 1)*
    - *Multiply by field value and add result to field value*
  - *If total field value >0 set cell i to dead*
  - *If total field value > 0 set cell i to alive*
  - *Next cell*
- *Return updated grid*

**Listing 3 Pseudo code for prototype Young model implementation**

### 4.2.4 run_simulation(generations, (model parameters))

Finally, the method to run a series of generations of the cell states was created. This creates a split plot of the images at each generation and displays them on screen. The final state is displayed separately for convenience. By default, the function uses the values first used in the research paper, but these values can also be manually specified if preferred. This allows convenience of not having to type the parameters each time, with the flexibility to if necessary.

## 4.3 Testing

### 4.3.1 Random distribution testing

The implementation of the random distributed was first tested visually, ensuring cells appeared to be properly distributed.

A test file was also created, in which 1000 grids of size 20x20 were created (400 cells each) with a 10% random distribution. The number of live cells were calculated and stored, and then the mean number of live cells per grid calculated over the 100 grids. If the distribution is indeed random, one would expect a mean of 40 live cells per grid over the 1000 tests. Listing 4 shows the results of the test, repeated a total of 3 times.

**Predicted average:** 40 (10% of 400 cell grid (20x20))
**Actual average:** 40.021
**Standard deviation:** 5.9296

**Predicted average:** 40 (10% of 400 cell grid (20x20))
**Actual average:** 39.852
**Standard deviation:** 6.1968

**Predicted average:** 40 (10% of 400 cell grid (20x20))
**Actual average:** 40.078
**Standard deviation:** 6.0728

**Listing 4 Results of random distribution test**

Evidently, the distribution is correctly random. However, due to this random nature the standard deviation shows that there is a relatively significant variation in the number of live cells. As a result, the function was modified to the version shown in Listing 5.

```
prob=percent/100;
grid=zeros(size_y,size_x);

% While number of live cells is less than amount specified
while sum(sum(grid)) < ceil(prob*size_x*size_y)
    % Generate random x & y co-ordinates
    x=round(rand(1)*size_x);
    y=round(rand(1)*size_y);

    % Ignore any 0 co-ordinates
    if x~=0 & y~=0
         grid(y,x)=1;
    end
end
```

**Listing 5 Updated method of generating random conditions (from random_distribution.m)**

The test function was run again and the results compared (Listing 6). This time, a perfect score was achieved on each of three tests, with an average of 40 cells, and a standard deviation of 0.

**Predicted average:** 40 (10% of 400 cell grid (20x20))
**Actual average:** 40
**Standard deviation:** 0

**Predicted average:** 40 (10% of 400 cell grid (20x20))
**Actual average:** 40
**Standard deviation:** 0

**Predicted average:** 40 (10% of 400 cell grid (20x20))
**Actual average:** 40
**Standard deviation:** 0

**Listing 6 Random distribution test with updated method**

Although the new method has improved the reliability of the random conditions, it is considerably slower. Using the old method, 1000 random grids of size 100x100 (10%) could be generated in 0.186 seconds, an average time of 0.000186 seconds. For the new method however, the time taken to generate the same 1000 grids is 18.13 seconds, an average of 0.018 seconds. Since one initial grid is only ever calculated at one time, this change in execution time will be unnoticeable. As such, the new method will be adopted in the final program.

### 4.3.2 Model implementation testing

To test the implementation of Young's algorithm, the patterns generated by the software were compared against the published results under the same parameters.

Starting from the random initial distribution, the model was simulated over 5 generations using values specified in the research: an activation field value of 1 with a radius of 2.3, an inhibition field value ranging from -0.34 to -0.2, with a range of 6.01, and a grid size of 100*25. The final states could then be compared to the results shown in the research. Figure 9-8 in Appendices 9.4.1 shows the two sets of patterns, both generated and published.

Each image was the result of five generational increments. The strong similarity of the two sets of images suggests the software is functioning properly and the model is correctly implemented.

In the first images (inhibition field value = -0.34), differentiated cells mainly form spots, with some joining occurring to form striped shapes.

The large areas of white space are a result of the strong inhibition value, and as such become less prominent in the next image where the inhibition is reduced. Here, most of the spots have joined, forming short stripes.

The third pair of images shows how the stripes and spots are able to join more completely when the inhibition value drops further. DCs are joined, almost continuously through

Finally, with an inhibition value of -0.2, the DCs form a single continuous mass, with a few undifferentiated spots and small stripes. Results from both the research paper and the software are very similar again, suggesting a proper implementation of the rules.

### 4.4 Prototype evaluation

Now that a basic prototype implementation has been achieved, a number of pitfalls have been identified. First and foremost is the running speed of the algorithm. Using Matlab's tic-toc functions[li], the time to execute was recorded for the apply_young() function. While sufficient for smaller automata (50x50 grid, 0.6 seconds to update), larger grids can take a significant amount of time to update (100x100 grid, 7 seconds). This is a result of the use of the nested for-loop in the apply_rule method. When considering improvements to the software, it may be possible to alter the function to use a faster, vectorised form.

Unfortunately, the use of bwdist() to find the Euclidean distance makes this particularly difficult, since a position matrix needs to be generated for each step. Vectorising this process is difficult since functions such as bwdist that operate on a 2D matrix do not accept multidimensional arrays, limiting the ability to completely vectorise the code.

Furthermore, using the Euclidean distance to identify both circular neighbourhoods makes it difficult to generate an ellipse, since only the distance (and not direction) to each cell is known. As a result of these two issues, a different algorithm or modification will be required in order to implement striping as described in the research paper, and to improve the running speed.

## 5 Improving the execution speed of the algorithm

Following a successful prototype implementation, it is now our aim to add improvements and additional features, as well as improve the performance of existing code. In order to continue testing the model, the next set of reproducible results are the striped patterns in Young's research. Since the existing method is both slow and incapable of creating these patterns, the first step is to find a new, more efficient method.

Through researching potential alternatives, it was discovered that the method for summing neighbourhood values is very similar to a convolution matrix, a method often used in image processing for processes such as anti-aliasing and image filters.

In order to create a convolution matrix, two sub-matrices are required: the original matrix, and a kernel of weights. The kernel is usually a smaller matrix, for example

3x3, which represents the neighbourhood of each cell. Each weight in the kernel represents the contribution of the cell in that position in the neighbourhood.

For each cell in the original image matrix, the surrounding cells in its neighbourhood are multiplied by the corresponding weight in the kernel, with the centre kernel value representing the current cell in the image. The sum of all the neighbourhood values multiplied by the corresponding weight is stored in a new matrix, the convolution matrix. The final convolution matrix contains the neighbourhood cell for each cell from the original image.

In terms of the patterning model, the kernel can be seen as each cells neighbourhood, and the values in the kernel representative of the field values at that cell. The image matrix can be seen to represent the grid of cells. By producing the convolution matrix of these two matrices, the sums of the field values have been calculated. Previously, this step would have been done by the lengthy bwdist() process in the nested for-loop. From here, the values are simply set to 1 or 0 depending on their sum value.

Since Matlab has a wide range of image processing and matrix functions built-in, it is no surprise to learn that there exists an existing function for computing the convolution matrix of an image given a kernel matrix. As a result, all that needs to be created in order to implement the model in this way is a means of creating a kernel.

Since the in-built function will be fully optimised, the code is likely to work significantly faster. Another benefit is that this method is extremely flexible with regards to the rule computed. As long as an appropriate kernel can be implemented, any neighbourhood arrangement can be calculated. As such, an elliptical neighbourhood should be no problem.

## 5.1 Design

In order to model Young's algorithm with a convolution matrix, a kernel needs to be defined which represents the neighbourhood. This will consist of a matrix, with two circular/elliptical areas of values: an inner area in which cells contain the activation field value, and an outer area in which cells contain the inhibition field value. Any cells in the neighbourhood which are to be ignored should have a kernel value of zero, since then the corresponding cell value will not be added to the sum. An example of such a kernel is shown in the appendices, in Figure 9-5.

To create such a kernel, a means of filling circular/elliptical areas with specified values is required. Not only will the size of the matrix need to be variable (to allow for differing kernel sizes), but it must also allow for a variable circle size within the kernel to allow the inner activation area to be created without occupying the entire matrix. As such, the function will be defined as in Section 5.1.1.

### 5.1.1 circle_matrix(matrix_size, fraction, x_ellipse, y_ellipse)=matrix

The circular region of values will be created using Matlab's flexible indexing to specify all the cells within the circle. To generate these indexes, the following equation of a circle will be used:

$$(x-i)^2 + (y-j)^2 = r^2$$

Where *(i,j)* is the centre of the circle, and *r*=radius of the circle. The radius will be calculated from the *fraction* parameter, which defines how much of the matrix the circular area occupies (between 0 (no circle) and 1 (full size circle)).

This equation can also be extended to provide a means of generating an ellipse:

$$\frac{(x-i)^2}{a^2} + \frac{(y-j)^2}{b^2} = r^2$$

Where *a* is the scaling factor associated with the x axis, and b is the scaling factor associated with the y axis. The parameter *x_ellipse* will be used as the value of *a*, and *y_ellipse* as the value of *b.*

An algorithm will be implemented to represent these equations and produce valid indices for the resulting circles in a matrix. The cells indexed by the equations will be set to 1, producing a matrix containing the circular area of 1s. This will be returned by circle_matrix(), and can subsequently be used to index any other matrix.

It is worth noting that circle_matrix(1, 0.5, 1, 1) will produce the same matrix as circle_matrix(1,1,0.5,0.5). The *fraction* parameter is included for simplicity when generating circular values.

The updated random_distribution function detailed in Section 4.4 will be used, as well as the original method of displaying the grid on-screen. Other functions required are detailed below.

### 5.1.2 young_kernel(act_range, act_field, act_xellipse, act_yellipse, inh_range, inh_field, inh_xellipse ,inh_yellipse)

This function will be a higher-level function that will make several calls to circle_matrix in order to generate the final kernel, consisting of the inner activation area and outer inhibition area. It will also define the size of the kernel matrix which is dependent on the radius of the inhibitor. As well as the normal range and field value parameters for both inhibitor and activator, the function also accepts values that define the shape of an ellipse (*inh_xellipse, inh_yellipse*, *act_xellipse and act_yellipse* to represent the x and y scaling of the circles, used for striping). If the function is called with no parameters, the default values will be used.

### 5.1.3 apply_rule(grid, kernel)=new_grid

Taking the current grid of states and the kernel representing the neighbourhood, this function will apply the rule using Matlab's inbuilt convolution matrix function, conv2. This will return a matrix the same size as the original grid, with each value

representing the neighbourhood sum of the corresponding cell. Using these sum values, apply_rule will set the new states, and return the updated grid of states.

### 5.1.4 run_simulation(time_steps, (model parameters)

This will function as before, providing a means of running a simulation over multiple generations and minimising input required at the command line. This time, however, it will use the new convolution method (apply_rule) to apply the model.

As before, the function will also accept optional parameters for the Young model, enabling the user to alter model parameters easily.

The updated interaction between these functions is displayed in Figure 5-1.

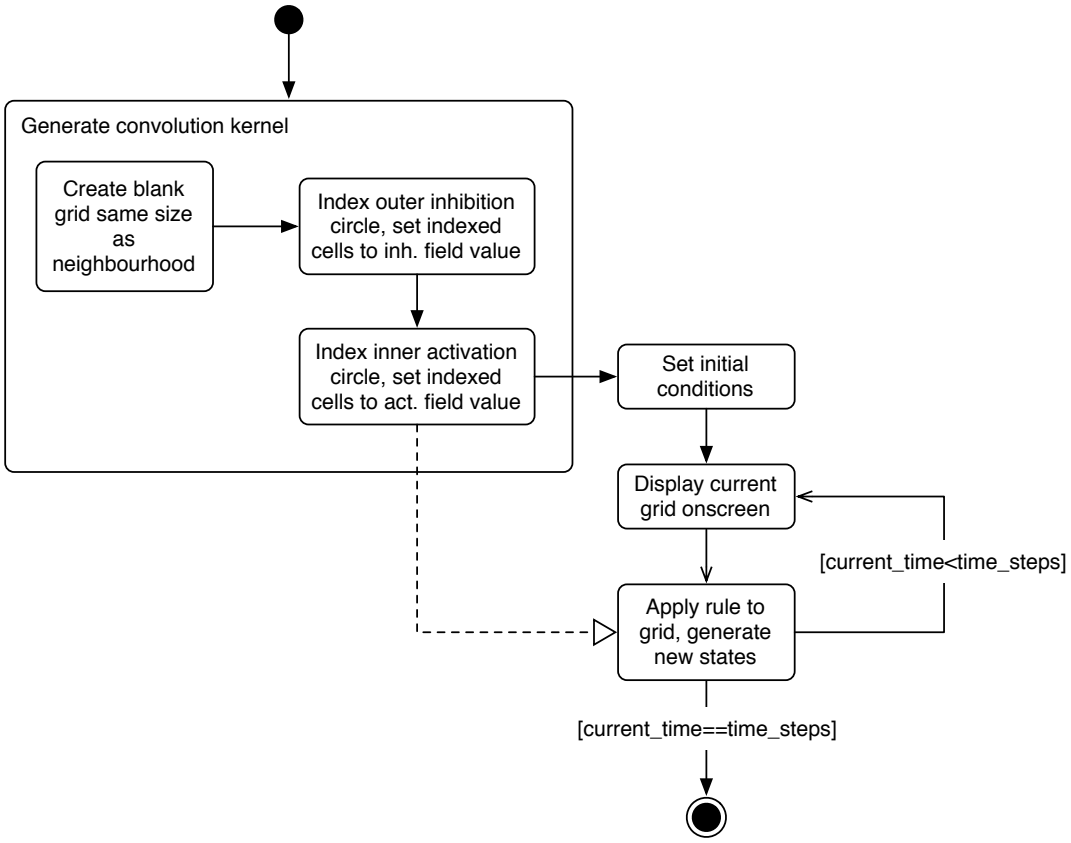


**Figure 5-1 Activity diagram of simulation processes (updated method)**

## 5.2 Implementation

### 5.2.1 circle_matrix(matrix_size, fraction, x_ellipse, y_ellipse)=matrix

The function for generating a circular area of values first calculates the midpoint of the matrix. It also uses the *fraction* value as a parameter specifying how much of the matrix the circle will occupy, in order to calculate the radius of the circle. A grid of cell indices is generated with meshgrid() and the cells lying within the circle are indexed with a vectorised form of the circle equation discussed in Section 5.1.1. The

values lying in the area defined by the equation are set to 1, producing the final matrix containing a circle of 1s. This resulting circle, which is returned by the function, can be used multiplied by a specific value or used to index other matrices. The process of generating the matrix can be seen in Listing 7.

```
% Calculate mid point of the matrix
mid=((matrix_size-1)/2)+1;

% Calculate radius of circle
radius=fraction*(matrix_size/2);

% Initialise matrix
matrix=zeros(matrix_size);

% Create meshgrid containing all indexes for the matrix
[y x]=deal(1:matrix_size);
[X Y] = meshgrid(x,y);

% Using equation for an ellipse/circle, set all values within circle to 1.
% Circle centred at midpoint.
matrix(((X-mid).^2)/x_ellipse^2+((Y-mid).^2)/y_ellipse^2<radius^2)=1;
```

**Listing 7 Generating a circular area of values (from circle_matrix.m)**

### 5.2.2 young_kernel(act_range, act_field, act_xellipse, act_yellipse, inh_range, inh_field, inh_xellipse, inh_yellipse)

To generate a kernel to represent the Young rule, a blank grid of zeros is first created. The size of this grid is calculated by doubling the radius and adding one (for the centre point). The value is also rounded up to the nearest integer, so that decimal values for the inhibitor range can be used.

Two circle matrices are then generated, one for each morphogen area. The two calls to circle_matrix use kernel size as the parameter for *matrix size*, and the circumference of the field area divided by total kernel size as the *fraction* parameter. The ellipse parameters are also passed, allowing striped pattern kernels to be generated. Finally, the circle matrices are used as the indices in kernel, setting the values to the corresponding field values for both morphogen areas (activator and inhibitor). This process can be seen in Listing 8.

```
% Kernel size is double the radius, plus the centre value. The radius is
% rounded upwards to the nearest integer so that decimal values can be used
% in the inhibitor range.
kernel_size=ceil(inh_range)*2+1;

% Initialise kernel
kernel=zeros(kernel_size);

% Get two index matrixes
activator_circle=circle_matrix(kernel_size,(inh_range*2+1)/kernel_size,inh_a,
inh_b);
inhibitor_circle=circle_matrix(kernel_size,(act_range*2+1)/kernel_size,act_a,
act_b);

% Using the index matrixes, the final kernel matrix is indexed and
% appropriate values set.
kernel(activator_circle==1)=inh_field;
kernel(inhibitor_circle==1)=act_field;
```

**Listing 8 Generating the Young kernel using circle_matrix (from young_kernel.m)**

### 5.2.3 apply_rule(grid, kernel)=new_grid

Listing 9 shows the new method for apply_rule, now separate from the generation of the kernel. Since the shape of the neighbourhood is generated separately (with young_kernel) the code required to update the grid is minimal, relying primarily on the single call to conv2().

```
% Initialise grid to store updated state values. All states initially start
% the same as the current ones.
new_grid=grid;

% Sum for each value calculated and stored
sum=conv2(grid*1,conv_matrix,'same');

new_grid(sum>0)=1; % Values in the new states matrix set to alive if sum>0
new_grid(sum<0)=0; % Values in the new states matrix set to dead if sum<0
% Otherwise the cell state remains the same as the previous generation
```

**Listing 9 Updated body of apply_rule.m**

### 5.2.4 run_simulation(time_steps, (model parameters))

The code for run_simulation has changed very little, now first calling young_kernel() and passing the resultant matrix to apply_rule as the convolution kernel.

## 5.3 Testing

### 5.3.1 Circle matrix and kernel testing

The new method for updating an automaton according to the Young model relies on two steps: generating the kernel and applying it with conv2(). For generating the kernel, the main functionality lies in the calls to circle_matrix, which was tested first. A range of circular matrices were generated, and the dimensions confirmed visually. For example, a circle matrix of size 10, and *fraction* parameter 0.6 was generated. An image of the resulting matrix was generated, and it was visually checked that a circle six units wide was generated, centred in the matrix. This process was repeated for elliptical matrices, with scaling across both x and y dimensions to ensure the correct implementation of the circle equation.

Examples of such tests are shown in Appendices 9.4.2.

Once circle_matrix was shown to be working correctly, the only remaining step in generating the matrix was to check it was being correctly called by young_kernel(). This involved visually comparing the generated kernels to the dimensions specified in the call to young_kernel. Again, multiple such tests were performed with a range of parameters but are excluded for brevity. Both striped and regular kernels were successfully generated, which is to be expected due to the successful testing of circle_matrix(). See Appendices 9.4.3 for examples of such tests.

### 5.3.2 Execution Speed

Since the new algorithm was implemented, the most notable change in the system is the decrease in time to run a simulation. Using the old algorithm for a relatively large grid (200x200), the time to execute was recorded. The result for a seven-generation simulation was 659.84 seconds; almost 11 minutes.

For smaller grids (50x50) with otherwise identical parameters, the time to execute is only 4.83 seconds. This shows how the time can balloon exponentially with a relatively small increase in grid size.

Using the new method, execution time has been significantly reduced to a point where it is no longer a problem. For the smaller grid size, the entire seven generations are calculated in 0.1-0.2 seconds, a 4.5 second difference and 22 times as fast. For larger grids, the difference is even more noticeable: still only taking between 0.1 and 0.2 seconds to complete; over 6,000 times faster. Even with a grid size of 3000x3000, the entire simulation only takes 5.7 seconds. This improved speed is more than satisfactory, especially compared to the prototype implementation.

### 5.3.3 Reproducing results

The new method was finally tested for correctness by comparing the patterns generated by it to the patterns presented in the research for a range of inhibition field values. As with the prototype, the simulation produces similar results with small spots gradually joining up to form a continuous striped pattern. See Figure 5-2 for a comparison.

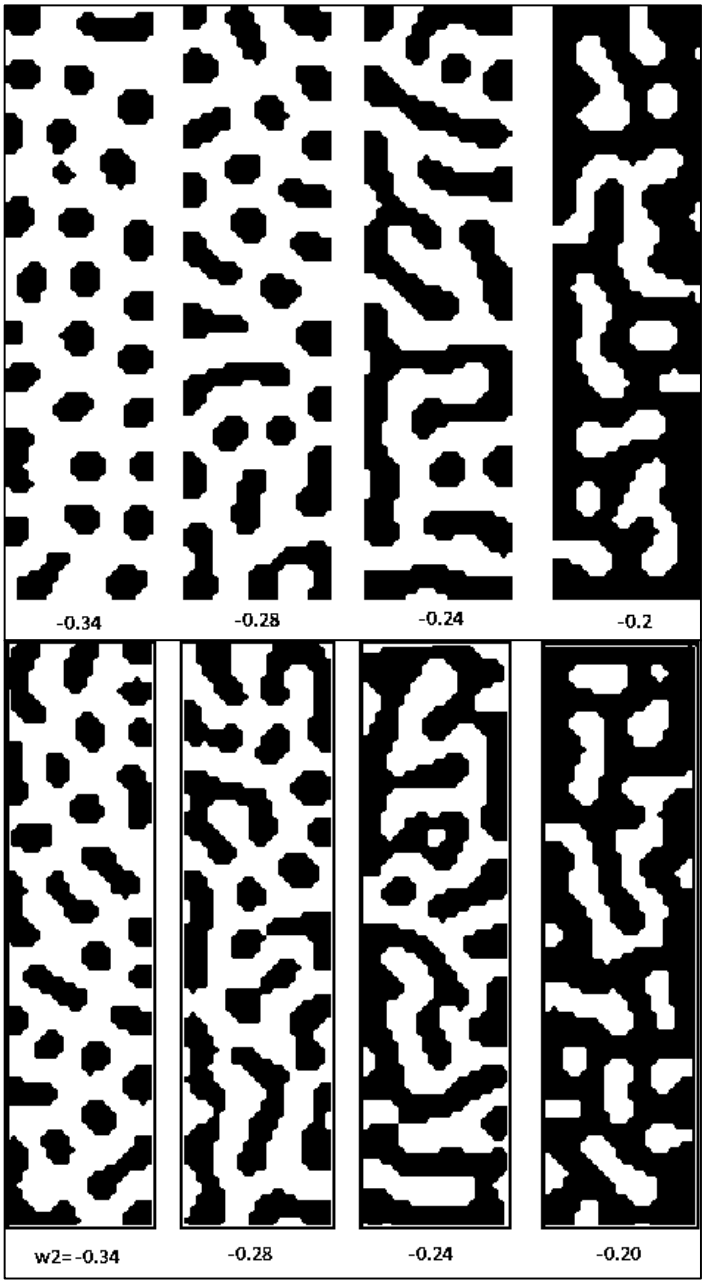


**Figure 5-2 Reproduction of published results using new update method**

Since the new algorithm also supports elliptical neighbourhoods through the parameters in circle_matrix.m, the subsequent striped results from the research can also be reproduced. These can then be compared to Young's results, and similarities or anomalies identified.

Young's description of the striped parameters makes no mention of the field values used for either activator, or inhibitor. As a result, the previously used default activator field value of 1 is used, and the inhibitor field value set to -0.24. Where Young's ellipse parameters are absolute (inhibitor: $a^1$=2.3, $b^1$=1.38, activator: $a^2$=3.61, $b^2$=6.01)[lii], the values accepted by the software are relative. As a result, these values

are changed to $a^1=1$, $b^1=0.6$, $a^2=0.6$, $b^2=1$. The pattern in Figure 5-3 was produced (contrast adjusted to be equal):

Generated results Young's results

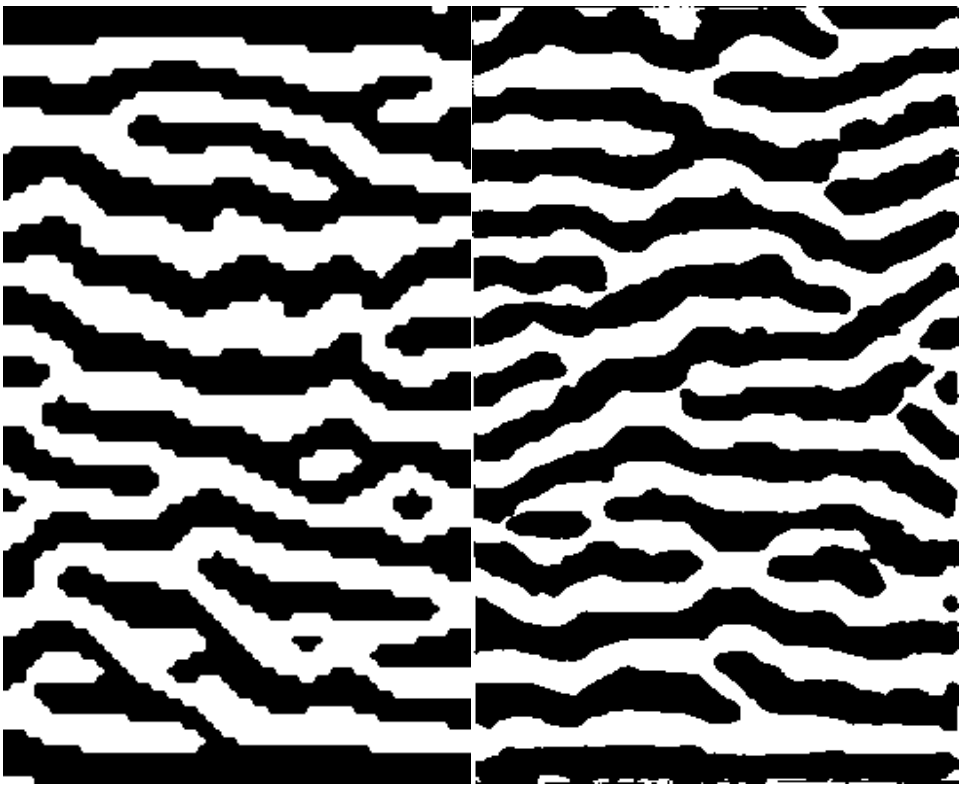

**Figure 5-3 Comparison of software results (left) with published results (right)**

Evidently, the two patterns are very similar. The fact that the model can produce striped patterns so similar to the published results as well as being able to replicate the regular patterns is strong evidence that the implementation is correct. Furthermore, by swapping the orientation of the two ellipses, vertical striping can be produced, as in Figure 5-4.

**Figure 5-4 Vertical striping produced with new update method**

# 6 Additional Improvements and Features

## 6.1 Design

Following the satisfactory command-line implementation of the model using the new, faster method, the remaining time could be devoted to implementing new features and improvements from the secondary aims and requirements. These will be designed and implemented with a graphical user interface in mind as many of them will be reliant on interaction with said interface.

### 6.1.1 Wolfram model implementation

The patterning model developed by Stephen Wolfram, discussed previously in the literature will also be implemented in software. In order to implement it, all that will be necessary is a new kernel. The current cell and all those immediately surrounding it contribute weight 1 to the kernel, and cells at distances 2 and 3 have varying negative kernel values. As a result, a new function, wolfram_kernel(dist1, dist2,striped), will be created in order to generate the kernel depending on the parameters specified by the user.

Taking the two field values as parameters, it will return a grid matching the description given in the literature. The optional third parameter will be a string value specifying the direction of the striping. If set, this will cause the weightings to only be

set in either the horizontal or vertical direction, as described in the literature. The kernel will be a constant size of 7x7: a radius of 3 cells with a central point.

Once the kernel is generated, the existing apply_rule method can be used to generate the next generation of the automaton.

### 6.1.2 Torus boundary conditions (wrap-around)

Currently, the automata treats boundary cells in the same way as cells in the centre of the grid. Because of the lack of surrounding cells at the edges, fewer cells are actually considered in the neighbourhood of boundary cells. As a result, the cells nearer the edges do not represent the natural progression of the rule as correctly as they otherwise would, which also has a knock on effect for all other cells in the grid. In order to improve this, a toroidal grid shape will be emulated, in which each grid edge is wrapped around to join the opposite side, effectively forming a torus (a donut shape).

In order to implement this into the method of applying a rule to an automaton, the cells at the borders of the grid will be padded with the values from the opposite side of the grid. This will occur before calculating the next generation of states.

The size of the padded area is dependent on the size of the convolution kernel. Since the kernel always has a central point (representing the current cell), the extent of the neighbourhood on either side of the current cell is equal to *(kernel_size-1)/2*. If a border of that size is appended to each side of the grid, each original cell will have a full neighbourhood.

After applying the rule to the padded grid, it will need to be trimmed back to the original size, maintaining the original central area. This should be easily achievable with Matlab's indexing commands.

### 6.1.3 Easily modify individual rule parameters

For both models, it should be possible for the user to easily modify the parameters governing the rule behaviour. For the Young model, this includes the range and field value of both morphogens. It should also include a checkbox of whether or not striping is to be used, and if so, the parameters to control the shape of the ellipses. Since the methods for generating both kernels accept the various model parameters directly, all that needs to be done is to provide a means of calling the functions using the values in the GUI.

In order to improve usability, a number of design decisions will be included. Firstly if striping is not enabled, the relevant controls will be disabled to eliminate any confusion. To make altering the shape of the elliptical regions simpler, sliders will be included to alter the values. This will provide visual indication of the minimum and maximum values and allow the ellipse ratio to be set 'by eye'. Linking the value of the slider with the text box will provide the user with the choice, so neither method will be forced.

### 6.1.4 Simulation playback and controls, including play-speed

The main requirement of the design is to be able to easily view the progression of the cells at each generation and modify the parameters defining the model. The user should also be able to cycle through the generations, and view intermediary states. For this reason, a set of playback buttons will be included, allowing the user to go forward and back through the generations, as well as displaying an animation of them.

In order to implement these controls, the states of the cells at each generation must be stored in advance. Otherwise there is no way to progress backwards through the generations to view previous states. As such, when saving any changes to the rule parameters, or when the software is opened, the cell states will be generated and stored in an array.

When the user presses a button to step forward or back, a counter representing the generation number will be accordingly adjusted and the image set to the matching matrix in the array of states. On the initial loading of the system, a default number of generations will be generated in one pass. This will mean the grids for subsequent generations are already stored in memory, and so there will be no delay in updating the display at each step. If the user wishes to view further generations, these will be generated in real-time and added to the array of states. For the most part, it is expected that the user will not need any more states since it usually only takes a small number of generations to stabilise. This method of storing some initial states provides a good balance between performance in the animation, and time to generate the array of states at each rule change/start of program.

The play button will allow the user to watch an animation of the progression of the patterns. A slider on-screen will allow the user to control how fast the image changes. This can be implemented with a pause between each call to update the display. The slider will set the length of this pause, effectively modifying the playback speed.

### 6.1.5 Saving images of simulations

Matlab's comprehensive matrix and image manipulation features should allow for a relatively easy implementation of saving images of the automata. This feature will allow for saving the current single image, or alternatively all images from the simulation.

### 6.1.6 Custom rule creation / rule modification

This feature will form a major part of the software, allowing the user to define their own kernel. By doing so, the user isn't limited to the two default models. Furthermore, variations on these models can be generated and the effect explored.

The user will be able to define a kernel of any size, which will initially be populated with zeros. Modifying the kernel will then be possible in two ways. Firstly, a grid representing the kernel will be visible on screen in a table. In each cell will be the corresponding field value, which can be modified directly on a cell-by-cell basis, simply by typing the new value into the table.

Alternatively, a method for generating elliptical/circular regions of field values will be available. The user will specify the dimensions of the circle, and the values with which to fill it. This will then call circle_matrix() to generate the values, and insert them into the centre of the table representing the kernel.

Finally, an option to specify which values to overwrite will be available. This will enable the user to generate a small, activator circle for example, and then generate a larger circle for inhibitor values over the top of it without replacing the inner, smaller circle of values. This will be reliant on the user marking a checkbox, and is optional.

It will also be possible to load an existing rule, and modify it in the same manner. This is particularly useful for making minor changes that cannot be done using the typical model parameters or with the circular value generator, in order to study the effect on pattern generation.

Once designed, the kernel can be saved to file and used to update the automata in the same way as either of the default kernels.

### 6.1.7 Manually setting initial conditions

As well as a random initial distribution, it will also be beneficial to allow the user to manually distribute live cells on the grid in order to investigate the effects of changing starting conditions.

In order to maximise usability, the user should simply have to click the cell that they want to toggle on the display, and it will update instantly. In order to prevent accidental modifications, the relevant checkbox must be ticked in order to modify the grid states. Furthermore, cells will only be editable on the initial screen.

In order to implement this, a means of establishing the selected cell's co-ordinates will be necessary. These co-ordinates can then be used as the index to the cell on the grid, and the value toggled. Finally, the on-screen display of the automaton will be updated to represent the updated cell states.

### 6.1.8 Saving & loading of initial conditions and simulations

By saving an initial grid, the exact simulation can be reproduced at a later date if the same rule is used. If interesting structures emerge in a specific combination of rule and starting conditions, they can be stored and explored at a later date.

In order to allow this, an option will exist to allow the initial grid to be saved to file. The user can then load it from file, and as long as the same rule is selected, reproduce each state from the simulation.

Matlab's save() function will allow the initial grid to be stored to a .mat file, which can be easily loaded later.

### 6.1.9 Alter the colour scheme

A menu will be implemented enabling the user to alter the colour scheme used to display the grid. Since Matlab uses a colour map to convert values to their corresponding colours, setting the current colour map will instantly update the colours displayed on screen, and as such it will be a simple case of calling the inbuilt function colormap()[liii] and specifying the scheme to use depending on the selected colour scheme.

### 6.1.10 Visualise rule kernels

In order to demonstrate the shape and range of the activator and inhibitor regions in the rule neighbourhood, a method of visualising the kernel will be developed. This will help the user understand how the various settings affect the rule properties. Since the kernel itself is just another matrix, it should be possible to reuse the display_grid() function.

### 6.1.11 Graphic user interface and validation

In order to incorporate the above functions into the software and improve usability, a graphical user interface will be developed. As well as the improvements detailed, basic options such as grid size, rule selection and image display will all be present on the GUI. A separate screen that can be called from the main display will allow the user to modify and create rules.

Figure 6-1-Figure 6-2 document the interface design for the two main screens. Error messages and additional GUI element design are contained in Appendices 9.1.

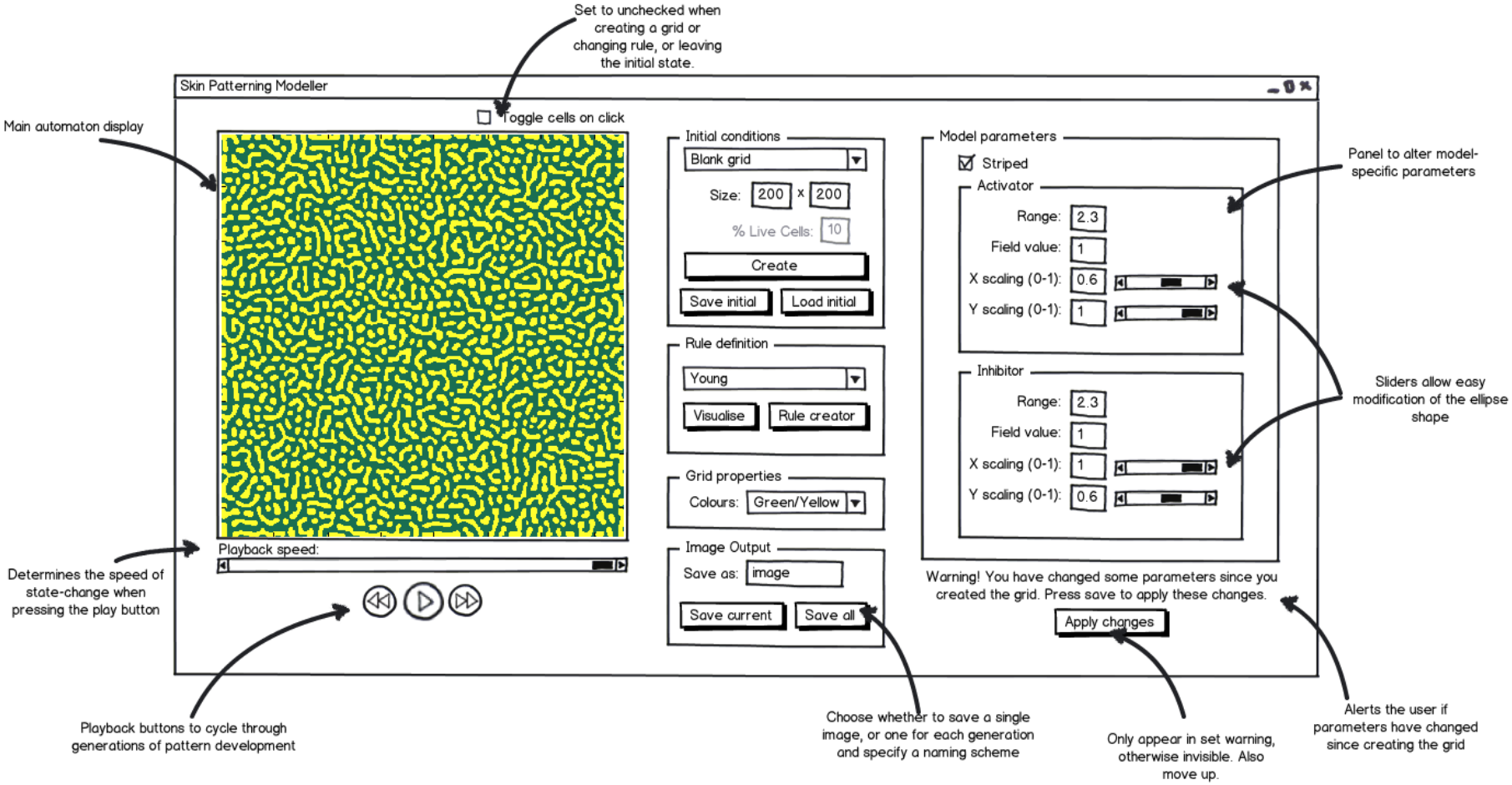


**Figure 6-1 Main GUI screen design**

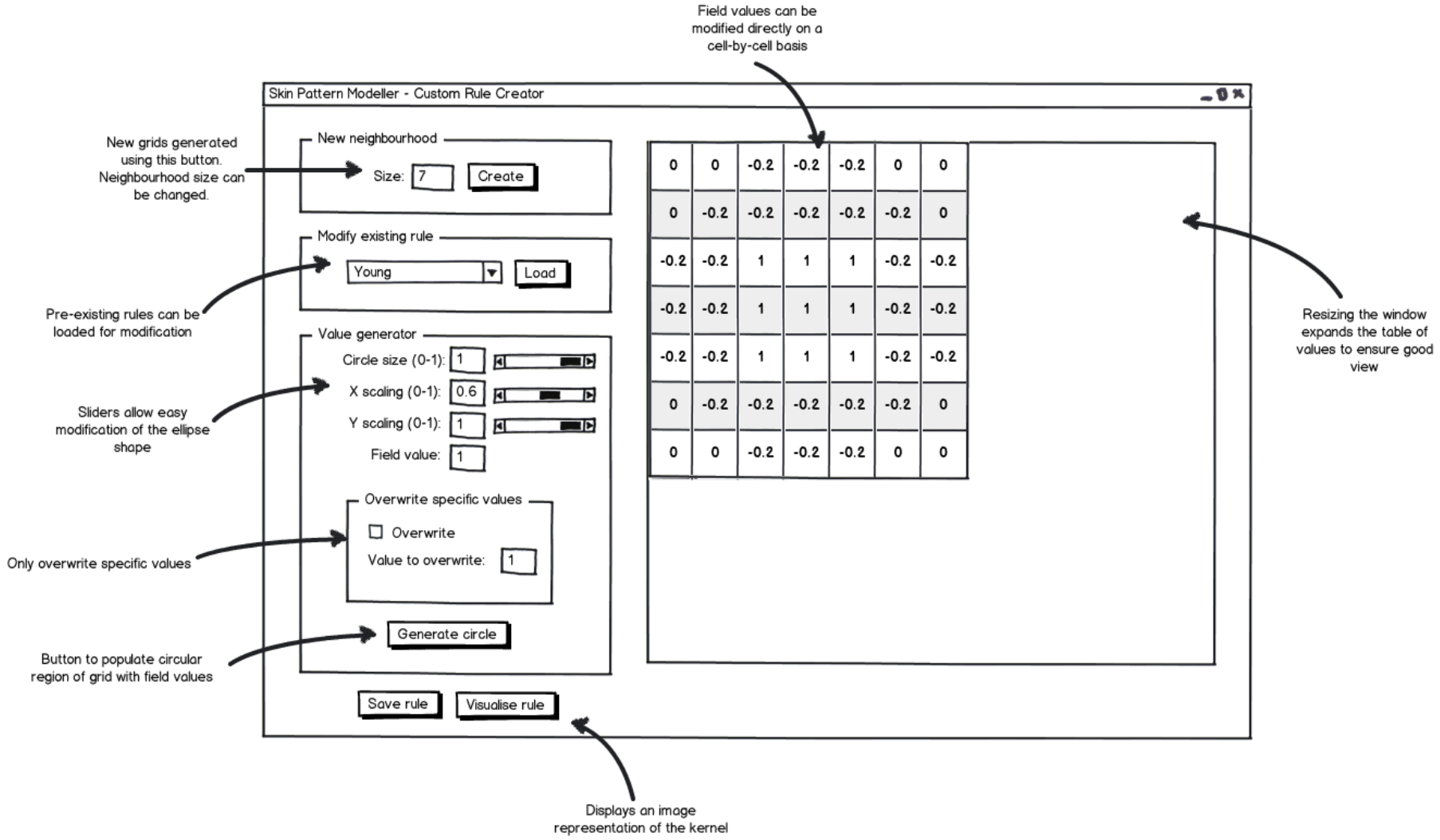


**Figure 6-2 Rule editor screen design**

## 6.2 Implementation

The GUI screens were split into two screens: the main display (young_gui_2.m) and the custom rule creator (rule_creator.m). As well as the .m code files, the design and layout is contained in a .fig file for each screen.

Many of the larger functions, and those shared between the two GUI screens, were split into separate function files to improve clarity and simplify the code. Each GUI screen also has an associated .m file with setup code and screen-specific functions.

The software initialises to the main GUI screen which shows the current automata and the control panel for the various settings, as designed in Figure 6-1. Pressing the custom rules button shows the relevant screen, which enables the modification of existing or new rules.

The GUI screens use handles to each object on the display to differentiate the various components. The handles structure of each screen also stores the initial grid configuration and the subsequent automata states, allowing all functions to access the information and update it as required.

When the rule modification screen is called by the main screen, the handles structure is passed along as a parameter. This allows the secondary screen to update the handles of the main screen, so that components such as the dropdown list of rule names can be refreshed instantly.

### 6.2.1 Wolfram rule implementation

In a similar way to the Young model, the Wolfram model is simulated by first generating a kernel, using wolfram_kernel.m. This is then used in the execution of apply_rule.m to generate the subsequent grid states.

In order to pass the parameters specified in the GUI to the function for generating the kernel, get_wolfram_matrix.m is used, which polls the text boxes, passes the values to wolfram_kernel, and returns the generated kernel.

The get_rule.m function decides which function to call depending on the current rule selected in the dropdown menu. For the two default rules, the relevant kernel creation is called, whereas if a custom rule is selected, it simply loads the kernel from file.

### 6.2.2 Torus boundary conditions (wrap-around)

In order to pad the matrix with the adjacent values in a wrap-around grid, a function pad_matrix(grid, kernel) was developed. First of all, the desired border size is calculated as the radius of the convolution kernel excluding the centre cell ($\frac{kernel\ size-1}{2}$).

The built in function padarray() is then called with the 'circular' option. According to Matlab's documentation, this 'pads with circular repetition of elements'[liv], which is ideal. The calculated border size is also included, as well as the grid to be padded.

The function is called at the start of apply_rule.m, and the subsequent generation calculated using the padded matrix. Before the updated grid of states is returned, the original, central area of the grid is extracted using Matlab's versatile indexing of matrices.

### 6.2.3 Simulation playback and controls, including play-speed

Two functions manage the storage of the various cell states for playback on screen. The first of these functions is create_states(), which is called when a change to the rule or grid is applied. It retrieves the initial grid from the GUI handles and recursively applies the current rule to the grid for each of the generations, and stores them. It resets the current generation number to 1, and calls the second function: set_state(). This function then checks the current generation number, and displays the corresponding grid of cells created by create_states.

The forward and back buttons modify the generation number and call the set_state to display the new current grid. The playback button cycles through each of the generations with a pause in-between; the time between each image is dependent on the size of the slider.

### 6.2.4 Saving images of simulations

As designed, the software is capable of storing either a single image, or one for each generation. Both functions save the file by the same method, but for the 'save all' method the generation number is appended to the file number. For either method, the saving is done using the imwrite()[lv] function which is an inbuilt Matlab command.

### 6.2.5 Modify individual rule parameters

The two default rules (Young and Wolfram) both have a separate panel used to configure the model-specific parameters. These allow the models to be finely tuned and experimented with. If a specific configuration of parameters is of particular interest, it can be saved as a custom rule, allowing it to be easily recalled at a later date and modified in the rule screen if desired.

Since the function for calculating the kernels for each of the rules takes the parameters as an input, altering the parameters is as simple as passing the values from the text boxes in the panel, to the method for generating the rule kernel. This functionality is implemented in the functions get_wolfram_matrix and get_young_matrix.

### 6.2.6 Custom rule creation / rule modification

The functionality for custom rules is contained primarily in rule_creator.m, the configuration file for the secondary GUI screen. Since most of the functionality (creating circular area of values, loading existing rules) was already implemented for other aspects of the project, much of the code simply involved calling existing functions or configuring components on screen.

The kernel is represented by a table on screen. For blank grids, a call to the inbuilt zeros(grid_size) generates the values and then this is set as the data source for the table.

Similarly, for loading existing rules, the code calls the existing get_rule() function, and sets the table data to the returned kernel.

To generate the circular region of values, the existing circle_matrix function is used. In order to use it in this instance, the relevant values simply need to be retrieved from the text boxes and passed to the function. If the checkbox for overwriting values is checked, an if-statement handles the indexing of values in the circle in order to only overwrite those values specified (Listing 10).

Once the circle has been generated, it is added to the table on screen.

```
% If overwrite checkbox is set then only replace the specified values
if get(handles.checkbox_replace,'value')
    data(data==replace_value & circle~=0)...
            =circle(data==replace_value & circle~=0);
else
    % Otherwise only insert values from circle which aren't 0
    data(circle~=0)=circle(circle~=0);
end
```

**Listing 10 Code managing the overwrite values functionality**

Rules created in the editor can also be visualised. Since the visualise function was already created for the main screen, this section of code simply involves calling the existing function.

Once the rule has been modified and the save button pressed and the user is prompted for a name to save the rule as. In order to ensure a valid name is used, the built-in function genvarname()[lvi] is called on the rule name, generating the final rule name. This constructs a valid variable name from the text, which is essential since the rule is actually stored as a variable prior to saving it. The rule is then added to rules.mat, the file storing each convolution kernel.

The modify rule screen also uses a custom resize function, which maintains the size of the control panel whilst allowing the rule table to resize. This enables the user to extend the table to allow a fuller view of the current kernel.

### 6.2.7 Manually setting initial conditions

Manually setting the initial conditions begins with choosing between a blank starting grid or a random distribution. Once generated, the user can tick the checkbox above the grid image to enable toggling of values in the initial conditions.

When clicking the image area, the callback code for the axes retrieves the last point that was clicked. The co-ordinates are rounded to an integer, and passed to the toggle_cell(x,y) function.

The toggle_cell function checks that the grid displayed is the initial grid. If so, the current cell state is inverted using a logical NOT command (~). The initial grid handle is set to the updated grid, and the subsequent states regenerated (create_states). If the current grid is not

the initial grid and the user attempts to modify the states, an error message is displayed and the states remain unchanged (Listing 11).

```
function toggle_cell(x,y)
% Get handles structure
handles=guidata(gcf);

% Only allow modifications to the initial grid
if (handles.generation==1)
    grid=handles.states(:,:,handles.generation);

    %Invert value
    grid(y,x)=~grid(y,x);

    % Set new initial grid and regenerate states
    handles.initial_grid=grid;
    handles=create_states(handles);

    % Update handles
    guidata(gcf,handles);
else
    % If not initial grid show error msg.
    errordlg('You can only modify cells on the initial grid');
end
```

**Listing 11 Function which toggles the state of a clicked cell**

### 6.2.8 Saving & loading of initial conditions and simulations

Since the states at each generation are regenerated upon changing the grid, all that needs to be stored to recreate a simulation is the initial grid. As long as the same rule is selected, each generation should be recreated exactly.

When saving the initial conditions, a dialog presents the user with an input for the name of the initial grid. Again, genvarname() produces a valid variable name from the user input which is used as a the filename of the rule in the 'Initial Conditions' folder. The initial grid is saved as the contents of the file.

To load an initial grid/simulation pair, a file browser dialogue showing the initial conditions folder is displayed. The user selects the file to be loaded, the initial grid is set and the subsequent states generated for playback.

### 6.2.9 Alter the colour scheme

Altering the colour scheme is implemented by a simple case statement in the callback code for changing the value in the colour scheme dropdown. For each colour pair, the relevant colour-map is loaded, which takes effect instantly.

### 6.2.10 Visualise rule kernels

Visualising the shape of the rule kernel is used by both GUI screens and as such is separated into a separate file. It creates a new window with the title 'Rule shape', normalises the field

values to be positive (squares and square roots), multiplies the value by 30 (to increase contrast between separate regions) and draws to the window.

#### 6.2.11 Graphic user interface and validation

In order to validate the values entered by the user into the numerous text boxes, two functions were implemented to check and sanitize the input. All numeric text boxes in both GUI screens call validate_edit_numeric when modified to check they are numeric.

For standalone numeric text boxes, validate_edit_numeric accepts the handle to the text box as its first parameter, and the default value as the optional second parameter. If no default value is specified, it will be set to 1.

The function attempts to get the numeric value of the text box. If this is non-numeric, the contents are set to the default value, and an error message displayed informing the user of this change. If the value is valid, it remains unchanged.

For values linked to a slider, such as the ellipse size text boxes, there is also a range of acceptable values (between 0 and 1). As such, after validating the input to be numeric, the function checks that the value lies in the correct range using validate_slider_edit. If not, it is rounded to 0 or 1. This ensures the value cannot be set outside the valid range of the slider.

The two main screens are shown in Appendices 9.3.

### 6.3 Testing

#### 6.3.1 Torus boundary conditions (wrap-around)

A function was created to test various conditions of the padded matrices (test_pad_matrix.m). Firstly, the size of the padded matrix after having the boundary areas removed was compared to the original grid size. Theoretically these should be identical, and such was the case.

The contents of the area inside the padded matrix were also compared to the contents of the original grid. Since padding the matrix should leave the central area unchanged, the two should be equal, which was the case.

Finally, the test function generates an example matrix, pads it, and displays the result as an image. This enabled the visual clarification of the repeating pattern, which was confirmed in Figure 9-9 in the appendices. Notice the repeating areas of yellow which match the contents of the red box.

#### 6.3.2 Easily modify individual rule parameters

Since both models call young_kernel and wolfram_kernel respectively, these were tested first. For the Young kernel, the main functionality lies in the calls to circle_matrix, which has already been tested. As a result, it was simply necessary to visually confirm the calls to circle matrix were being performed appropriately, and were producing kernels of the correct shape and values. To do this, various Young kernels were generated and the image of the kernel compared to the parameters.

An example of such a kernel is shown in Appendices 9.4.3. Multiple such tests were performed, but are excluded for brevity. Both striped and regular kernels were successfully generated, which is to be expected due to the successful testing of circle_matrix().

Since the Wolfram rule is of a constant size, and values at the various distances are the only variables, testing was simpler. It was checked that the specified values were actually inputted into the various regions in the kernel (distance 2 and 3), and also that the various regions were of the correct size and location. Visual confirmation can be seen in Appendices 9.4.4.

### 6.3.3 Alter the colour scheme

Visual confirmation of the change in colour was performed manually, and it was verified that the correct colour pair labels were used in the dropdown list.

### 6.3.4 Simulation playback and controls, including play-speed

Testing of the playback controls involved a number of manual tests. Firstly, it was checked that the buttons correctly advanced and reduced the generation number, and caused the correct generation number and image to be displayed as expected.

If the default number of states is exceeded by pressing forward, new states should be generated and appended to the state array until total stability was reached. This was confirmed to be working correctly, and an error message notifies the user that stability had been reached and no further states existed (Appendices 9.4.5).

Finally the play button was tested, and the progression through the states observed. The playback speed slider was also adjusted to confirm it affected the speed correctly. The minimum value specified in the code (0.1s) ensures it doesn't run too fast, even at the maximum setting.

### 6.3.5 Saving images of simulations

An automaton was set up with a distinctive initial pattern by setting only a single pixel to 'live'. This allowed the pattern to be clearly distinguishable, allowing visual confirmation of the images stored.

Images were generated with the 'save all' button, and another separate image for the 'save current' button, both with the name 'test'. The files produced were compared successfully to the on-screen display in Appendices 9.4.5. Each image was also 110 pixels in both directions, the same size as the grid specified; suggesting files images are being saved correctly. Furthermore, the images had the correct filenames: test#.png with the generation number appended for the 'save all' images, and test.png for the single image.

### 6.3.6 Wolfram rule implementation

As the method for generating the Wolfram kernel has been successfully tested, all that remains to be tested is the patterns generated by applying the rule with the kernel.

Although the final patterns presented in Wolfram's research are very small, they appear to match the patterns generated by the software (see Appendices 9.4.7) for both regular and

striped patterns. Since the Wolfram rule uses the same method to apply the rule, and relies only on a modified kernel, performance is the same, and equally satisfactory.

### 6.3.7 Custom rule creation / rule modification

To test the rule modification system, it was first confirmed that loading and creating blank rules functioned correctly. This involved saving the Young rule as a custom rule, and then proceeding to load it on the rule screen. The resulting grid of values was verified to be the correct grid in Figure 9-25.

It was attempted to create a blank grid of even size, which should be disallowed since there would be no central cell. As expected an error message was displayed (Figure 9-26), the grid size was rounded up to the next odd integer and the grid generated properly. When a valid, odd neighbourhood size is specified, the blank grid is created instantly (Figure 9-27).

If a non-numeric value is entered into the neighbourhood size text box, or any of the other text boxes, an error message is displayed by the validation function. If a value larger than 1 is entered into the text boxes linked to sliders, both are automatically set to 1 as required.

Generating a circle relies on circle matrix which has been previously tested, so all that was necessary was to check it was being called correctly. Figure 9-29 shows an example of a circle correctly generated and displayed in the table.

As well as generating elliptical/circular region of values, the rule editor also allows values in individual cells to be edited directly. The grid from Figure 9-29 was successfully modified and saved to check correct operation (Figure 9-30).

The functionality of the checkbox enabling the user to overwrite values was also tested. Starting with a kernel of zeros, it was attempted to generate a circle of 1s with overwrite checked, and the overwrite value set to 4. This resulted in no change to the grid, since there were no 4s to overwrite, as expected.

Next, a small central circle of 2s was generated (Figure 9-31), and then a larger circle of 3s generated over the top with overwrite disabled. Since no value to overwrite was specified, the circle entirely replaced the inner circle (as expected) (Figure 9-32).

Finally, ‘overwrite values’ was checked with an overwrite value of 0, and the larger circle generated over the smaller one again. This resulted in the inner circle remaining in place, and the larger circle taking a ring shape around it (Figure 9-33). All implemented behaviour with regards to the circular value generator is therefore working as desired.

The final step in the custom rule system is saving the rules. The system should sanitise the name inputted by the user, and so rule names containing illegal characters were inputted and shown to be correctly adjusted (Appendices 0, Figure 9-34 & Figure 9-35). Furthermore, it was checked that new rules were automatically added to the existing dropdown list of rule names on the main screen. As a result of this test, it was noticed that the dropdown for loading existing rules was not updated on the custom rule screen, and so this was amended in the code.

### 6.3.8 Manually setting initial conditions

In order to test modification to the initial conditions, the toggle checkbox was left unchecked and the grid clicked. As intentioned, no change occurred. The grid was then advanced to the 2nd generation, and again the grid was clicked to see if the error message was displayed. As expected, no error message was displayed since the checkbox was clear.

To check the correct operation of the state toggling under normal conditions, the grid was again set to the initial generation. The toggle checkbox was ticked, and cells were correctly toggled when clicking them.

To ensure changes in the initial grid remained after changing the grid, and also permeated through to subsequent states, the generation was advanced. As expected, the pattern had changed around the area modified in the initial state. Also, if trying to modify the second generation of states, an error message is displayed informing the user only the initial grid can be changed. All initial grid modification behaviour is therefore working as expected.

### 6.3.9 Saving & loading of initial conditions and simulations

To test the saving and loading of initial conditions and the subsequent generations, a 60x60 grid was created with a distinctive initial pattern of only a few live cells, and saved to file (Figure 9-36). Images of each generation were saved (with the save all images button), and the initial conditions loaded from file (Figure 9-37). The images were again saved to file, and it was confirmed that both outputs were identical, showing the correct functionality of the system (Figure 9-38).

The input also uses the same method for sanitising inputted names (genvarname), which has been tested previously.

### 6.3.10 Visualise rule kernels

In order to confirm the correct operation of visualising rules, a number of test kernels were generated and the shape visible in the rule editor compared to the image shown when visualising the rule. Several examples of such comparisons are shown in Appendices 0.

It was also checked that when modifying rule parameters in the main screen, the visualisation would represent the most recent version.

# 7 Evaluation

Through the course of the project, a piece of software has been implemented which allows the user to investigate predefined or custom rules of skin patterning. Unlike the majority of CA applications, it does not limit the user to a predetermined neighbourhood size. By implementing long-range interaction between cells, models based around morphogenetic diffusion across the skin can be represented. Furthermore, a precise control over the definition of the rule allows the user to investigate the effect of various configurations, and discover new patterns and behaviours in the modelling process. This flexibility in rule design is complimented by supplementary features, such as the circular value generator, which allows complicated rules to be defined without a laborious setup process.

Due to the abstract nature of the software and lack of similar software, evaluating the results can be difficult. Aims, objectives and requirements described in the analysis provide a checklist to compare the final implementation to.

Aims and objectives

Firstly, all three primary aims and objectives have been successfully achieved in their entirety. A model (Young's model) from the literature has been successfully implemented (1), the software has been extensively developed as a tool to investigate model behaviour (2), and finally, analysis and replication of patterns exhibited in the research (3) has been successfully achieved for various patterns (and both models).

In terms of the secondary aims, a further model (Wolfram[lvii]) has been implemented. Although the other objective, 'seek to develop aspects of the model identified' was not accomplished, this was mainly due to a lack of behaviour being discovered which was not already documented in the original research. With further time, it may be possible to discover such behaviour, although exploring the model in-depth is time-consuming, and so resources were focussed on improving the software.

Requirements:

**Command Line interface (Essential)**

Although a user-friendly GUI has been implemented, there also exists a means to run simulations at the command line. The user is able to call a single command, run_simulation() which will simulate the Young model over multiple generations, and produce an image on screen of the automaton at each step using default values. The user can also specify parameters affecting the model, in order to study the effect on the final pattern produced. Since the model was first implemented for the command-line, and later improved to include a GUI, many of the features of the final software are not present in the command line interface. This was a conscious decision to avoid excessive work to update the command-line implementation, since most users will not require it. Despite this, the aim can be considered implemented successfully.

**Display output (Essential)**

Displaying the pattern of cells is essential to the project, in order to understand the development of the model. Initially implemented in the command line, the method of displaying the output was extended in the final GUI, allowing the user to play an animation of the pattern development, as well as view individual slides and navigate through each generation. As such, this objective has been exceeded.

**Speed (Essential)**

Through the development of the initial prototype, it was realised that the execution of the speed was inadequate and an alternative method had to be developed. The improved algorithm gave an exponential increase in speed, and has resulted in a very satisfactory user

experience. By generating the states at each generation before playback, the animation of the automaton is very smooth, with very little slowdown noticeable to the user. The ability to maintain such good performance at very large grid sizes is also testament to the speed, and as such this objective can be considered to be achieved successfully, if not exceeded.

**Adaptability (Essential)**

The separation of rule definition and rule application in the use of the convolution method, with a separate kernel, has led to a very adaptable program. This has also facilitated the ability to define custom rules, which would not have been as simple if using a predetermined neighbourhood method. Furthermore, code developed for the command line interface could be adapted and reused without corrupting existing functionality. This enhanced the iterative development process, since existing modules could be modified to enhance functionality at each step in along the way.

**Represent short-long range interaction (Essential)**

Since the custom rule system allows kernels of any size to be defined, this goal is certainly achieved. The circle matrix function also satisfies this goal by generating kernels of circular inhibition and activation areas. Since the user can also define the individual weights of cells in the grid, extremely complex neighbourhoods can be generated, exceeding the original objective.

**Vary initial states of the automata (Essential)**

As well as the original goal of being able to generate random initial conditions and save/load from file, an interface for toggling the state of individual cells far exceeds the objective set in the analysis. Not only can interesting configurations be saved, but initial conditions can also be hand-picked by the user in real-time, and the resulting change in pattern observed immediately. This allows a far more interactive exploration process for the user, and directly shows the impact of changes to the model.

**Edit parameters (Essential)**

The original aim for editing parameters was to allow the user to set variables such as field values and morphogen range, which has been successfully implemented with the additional panels for the default rule. Furthermore, the development of the rule modification system has also provided a fine-grained method of changing very specific rule details, greatly exceeding the original expectations.

**Customise model (Essential)**

Aside from the default Young model, the rule from Wolfram's 'A New Kind of Science' has also been implemented, along with customisation options for modifying the parameters, satisfying the requirement to customise the model used. Beyond this, the custom rule system also allows for new rules to be implemented, without any programming knowledge, exceeding this requirement.

**Modify running speed (Desirable)**

The slider under the grid image on the main screen of the GUI controls the playback speed of the grid. By pre-generating most of the generations, the image can be updated with no wait time in between, and the requirement is satisfied.

**Control basic operations with GUI controls (Desirable)**

The interface developed for the software provides a user-friendly method of exploring the pattern development models. Playback of the simulation as a whole, or on a generation-by-generation basis is possible, and is smooth and easy to control and understand. Sliders, dropdown menus and labelled buttons all ensure margin for error is minimalized and navigation is as simple as possible. The rule creator screen allows the user to scroll and resize the rule display area, to ensure as wide a view of the kernel is provided. The layout of the screen components are uncluttered and laid out logically, furthering the ease of use for the user. Since it was only intended to control basic operations, the ability to modify the rule and associated controls so comprehensively represents an exceeding of this requirement.

As far as the author is aware, there is no real alternative software specifically for modelling skin patterning with cellular automata. As a result, it is not possible to directly compare the performance and results of the software with existing programs. However, as discussed in the analysis, there does exist a small number of programs capable of running general classes of automata. Presumably due to their age, two of the programs discussed are incompatible with modern incarnations of the Windows operating system and so cannot be compared to the software created. The remaining program, ‘Five Cellular Automata’ (FCA), provides a limited benchmark to compare our software with.

Capable of modelling the game of life, as well as viral replication and diffusion, Five Cellular Automata only allows a limited number of parameters to modify the rule with, and as mentioned on the creator’s website ‘The greatest weakness is the inability to draw your own starting positions; this makes it much less useful as a way to experiment with the concepts and discover new behaviours on your own’. This is in contrast to the software developed for this project, which appears to be far more flexible, both in terms of rule definition and modification as well as the variation of initial conditions.

Five Cellular Automata does appear to have one minor advantage in that the options available to alter the appearance of the display seem to be more comprehensive. The user can specify if the cells should be represented by circles or squares, and each cell can take more than two states (colours), producing a richer visual output. Although the models implemented in the project make no mention of such a colour scheme, it may be possible to implement more complicated behaviour, for example blending cells at the edges of pattern elements to give more lifelike patterns. Unfortunately, time for such an improvement was not available and would require more research into the biological basis for any such patterning. On the other

hand, FCA has no options to alter the colour scheme, whereas our software allows this to be selected.

The software developed also provides greater flexibility in grid sizing, which in FCA is limited to eight predefined sizes. Furthermore, FCA provides no mechanism for saving images of simulations, and no method of saving simulations for later. This aspect of the software we developed is incredibly useful as an educational tool, allowing the user to store interesting rule combinations to revisit later and also observe how changes made can affect the end result.

A limitation of our software is the restriction to 2D rules. Many of the famous 1D Wolfram rules[lviii] produce interesting patterns, often compared to striping on animal skin and shells. It would be an excellent improvement to allow the user to explore these, and compare them to other rules in the software. Unfortunately, time restrictions have prevented their implementation, although it would be relatively easy to implement in the future, for example with special cases in the apply_rule method for 1D automata.

As well as being limited to 1-dimension, the rules modelled can only be those with a binary neighbourhood threshold (sum greater than or less than zero). This prevents automata such as the Game of Life, with more complicated state change conditions, from being implemented into the existing framework. As with the other limitations, this is relatively minor and could easily be implemented with more time.

The ability to visualise a cells neighbourhood aids the understanding of the underlying process for the user. Although the current visualise method does help the user to understand the shape of the neighbourhood, it would be beneficial to also display the field values in each region. Due to the small range of field values, the contrast of the image was too low, and so values had to be multiplied by 100. Also, negative values appeared the same as zero on the image, and so inhibitor regions weren't visible on the visualise screen. As a result, the values had to be normalised by squaring and subsequently square-rooting them, removing the ability to differentiate between inhibitor and activator. If further time were allocated it would surely be possible to modify this function to provide a more detailed account of the rule kernel, including the values in each region. Unfortunately, the default Matlab imaging capabilities did not allow for this by default, and so it had to be avoided in favour of implementing more critical features.

Despite these limitations the software is very flexible. The ability to define your own rules is a unique 'selling point', and allows the software to be used as a tool for investigation of such rules. By enabling the modification of initial conditions and the cell-by-cell behaviour of each rule, the user can observe countless effects and emergent behaviour, providing an invaluable tool for defining and exploring 2D automata.

The idea of morphogens as chemical messengers transported by diffusion is central to the concept of patterning modelling. The implementation of the function to generate elliptical regions of values provides a means of representing such a process, and grants the user the ability to formulate their own rules using the same basic concept.

The circular value generator could be improved by implementing a live preview of the shape of the ellipse. Although visualising the rule performs a similar task, it would be helpful to see the shape and size of the circle before generating the values. This could be implemented in a separate, live display containing an image of a circle which stretched and squashed as the parameters were changed. This would help the user select the best parameters for the circle shape before committing the values to the matrix.

Much of the code developed is reused throughout the software. For example, the two validation functions enable every cell to be checked, reducing the scope for user error with only two small functions. Code from the initial implementation has been adapted and re-used, and such time-saving steps have facilitated a rapid development process which has enabled the implementation of further improvements.

The choice to use Matlab as the development language turned out to be a valuable decision. The various imaging and matrix manipulation features have provided a large number of 'shortcuts', taking care of many of the basic functions which would have otherwise occupied precious development time. The optimised matrix and vector functions of Matlab have also proved to be invaluable; the implementation of the rule system as a convolution matrix has improved the execution speed by a factor of thousands.

The development strategy also contributed to the success of the project. After the initial prototype was created, it was realised that the execution speed would be insufficient. This allowed the superior method to be developed, before proceeding to develop further improvements. Had a waterfall model been used it may not have been until the final stages of implementation that the problem was discovered, resulting in a slow or limited size automata for the final implementation. Instead, we were able to shift to the most relevant stage after each aspect of implementation was achieved, allowing for plenty of time to improve the software.

The command-line implementation of the software is sufficient and properly models the Young model as aimed, but the GUI based implementation is even better. By providing an easy-to-use front end, the software is far more feasible as a learning tool. With even a basic understanding of the pattern development process, the model can be altered with great ease. Components like the sliders for generating circles and the direct editing of cells in the rule kernel makes it far easier to understand and make changes, and from there see the effects on the final pattern. The playback system also allows the user to see the full development of the pattern at a variable speed, to get a good understanding of the underlying process.

The core model was able to generate a wide range of patterns for both Young and Wolfram's rules, including spotting, striping, directionality and emergent behaviour from a variety of initial conditions. Each component of the model representation has been tested for veracity, and the similarity to published results emphasises the likelihood of a valid implementation.

With sufficient time the software could be used to investigate the models implemented by experimenting with parameters and initial conditions to explore the range of patterns created.

The flexibility of the program would make this process far simpler than it otherwise would be, demonstrating the value of the program as a learning/investigational tool.

# 8 Conclusion

Overall, the project to develop the modelling software has been successful. A review of competing models provided a strong basis for development, and identified ideal candidates for a cellular automata based implementation.

The resulting aims, objectives and requirements clearly outlined the work required, and helped guide the progress of development. A flexible design methodology enabled milestone targets to be set dynamically which improved the quality of the final product by identifying the most valuable extensions at each stage and allowing major problems to be identified early in the project before they caused any significant delays.

A successful prototype early on provided adequate time to implement a further model and develop the software to allow its use as a learning tool to explore model and pattern behaviour. Sophisticated features for varying the simulation parameters allow a comprehensive investigation of the development process hypothesised in the literature, and give an interactive demonstration of the models.

The two models from the literature were implemented successfully, and enabled the production of a range of emergent patterns. The software also allows the user to create variations of these rules and alter the defining parameters, which is something yet to be achieved by an existing program. Furthermore, the user can define entirely custom rules and automata and explore their behaviour interactively.

To complement the flexibility of the rules defining pattern development, initial conditions can also be adjusted, stored and loaded, providing another avenue of exploration. As noted by the author of the competing Five Cellular Automata, a lack of this is the ‘greatest weakness’ of the software, and so to include it is a great achievement, which extends the possibilities substantially.

Despite the visual nature of the software and the inherent testing difficulty, a range of unit tests have ensured the correct functionality of each component in the software and built up a trusted set of modules. Integration tests have been carried out which ensure the correct interaction between existing modules and patterns generated have also been successfully compared to those published by the model authors across both regular and striped patterns. Altogether, the rigorous tests increase the probability that a reliable implementation of the models has been achieved.

Several minor extensions have been proposed, most of which would require only minimal adjustments. This is in part due to the programming practices and separation of functionality into specialised components which can be altered without affecting other aspects. The majority of the aims and requirements have been successfully implemented, and those

missing are trivial. Non-functional requirements, like running speed and usability, have also been satisfied, resulting in an easy-to-use piece of software.

Considering the lack of applications focussed specifically on skin pattern modelling, and the limited range of general purpose CA software, the program developed has a number of unique features which set it apart. Firstly, the representation of long range interaction is not usually possible, and therefore preventative of most skin development models. Defining custom rules is also a differentiating feature, and when combined with the ability to specify initial conditions, provides a level of customisation not seen in the existing software analysed. It also allows a far more comprehensive investigation of the model behaviour, which is ideal for using the program as a learning tool.

Research published in the journal Nature Genetics since the completion of the project presents experimental evidence for an activator-inhibitor system responsible for the generation of ridges on the palate in mammals, of the reaction-diffusion type proposed by Turing[lix]. The morphogenetic components of this system has been identified, which suggests that in the future it may be possible to identify the quantities of these chemicals and build a model in the software which emulates the process. The discovery of such a system highlights the relevance of the work, and provides a basis for further study. With new techniques for investigating the biological processes underlying development, new, empirically derived models may emerge which can be simulated with the software or provide avenues of extending the software.

# 9 Appendices

## 9.1 GUI designs

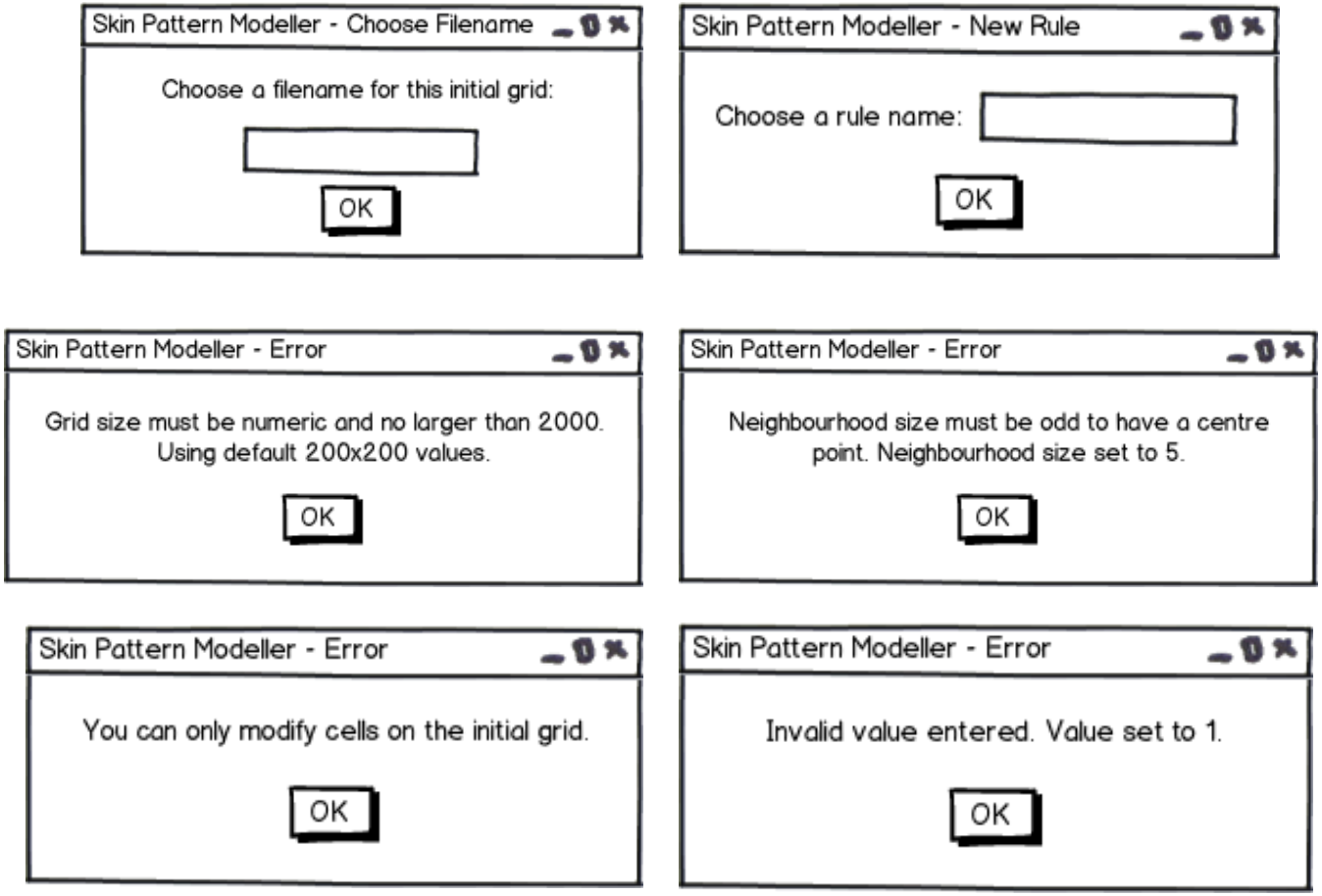


**Figure 9-1 Error messages which will be implemented**

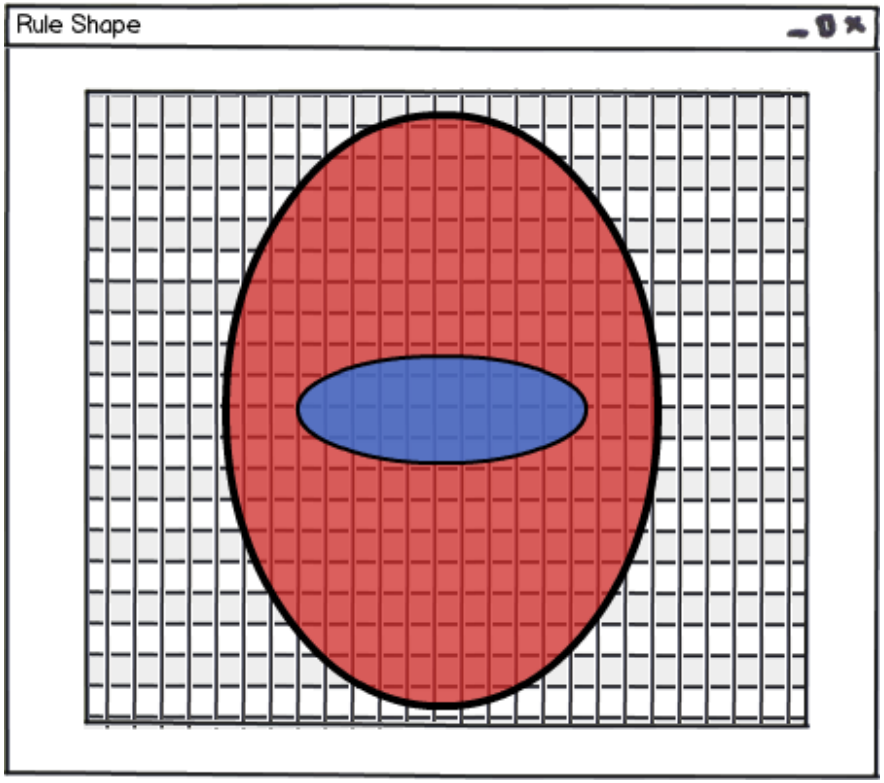


**Figure 9-2 Visualise rule screen design**

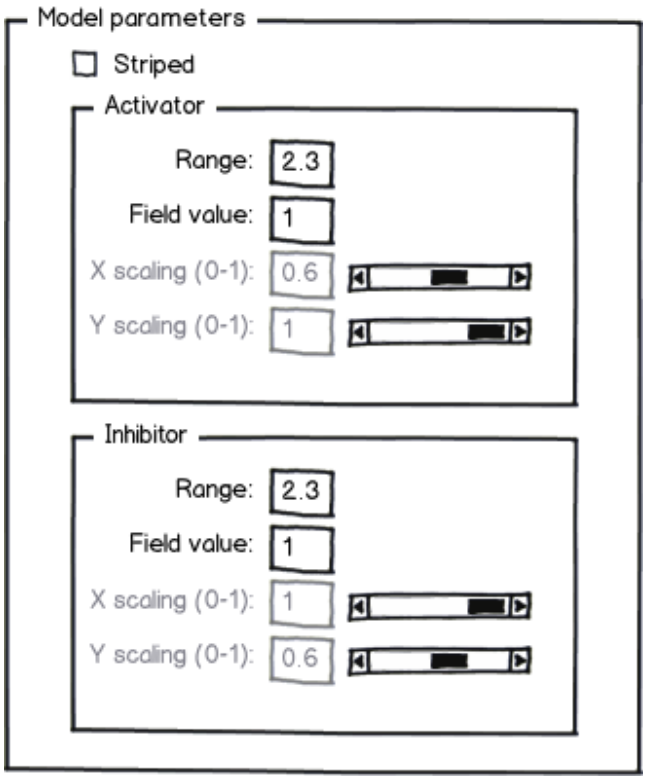


**Figure 9-4 Young parameter panel with striping unchecked**

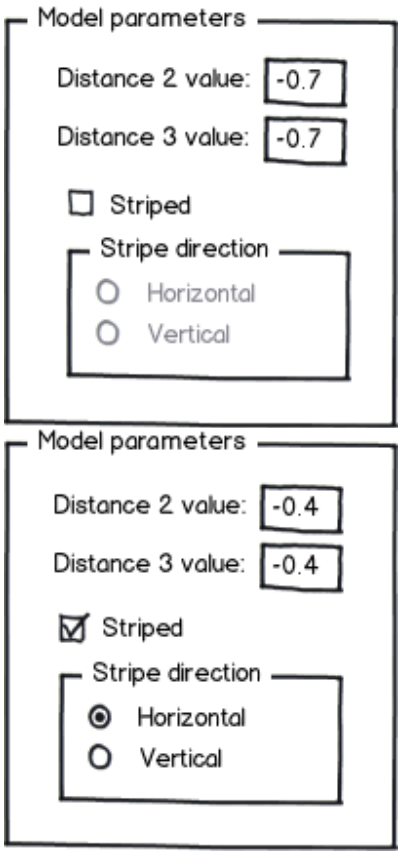


**Figure 9-3 Wolfram parameter panel with and without striping checked**

## 9.2 Example convolution kernel for the Young model

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 | 0 | 0 | 0 |
| 0 | 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 | 0 |
| 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 |
| 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 |
| -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 1 | 1 | 1 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 |
| -0.24 | -0.24 | -0.24 | -0.24 | 1 | 1 | 1 | 1 | 1 | -0.24 | -0.24 | -0.24 | -0.24 |
| -0.24 | -0.24 | -0.24 | -0.24 | 1 | 1 | 1 | 1 | 1 | -0.24 | -0.24 | -0.24 | -0.24 |
| -0.24 | -0.24 | -0.24 | -0.24 | 1 | 1 | 1 | 1 | 1 | -0.24 | -0.24 | -0.24 | -0.24 |
| -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 1 | 1 | 1 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 |
| 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 |
| 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 |
| 0 | 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 | 0 |
| 0 | 0 | 0 | 0 | -0.24 | -0.24 | -0.24 | -0.24 | -0.24 | 0 | 0 | 0 | 0 |

**Figure 9-5 Example of a kernel representing the Young method neighbourhood**

| | Inner green circle | Outer orange circle | Outer blue cells |
|---|---|---|---|
| Field value: | 1 | -0.24 | Since value is zero, cell does not contribute to sum and in-effect is ignored. |
| Range: | 2.3 | 6 | - |

**Table 1 Description of kernel values**

## 9.3 GUI Designs

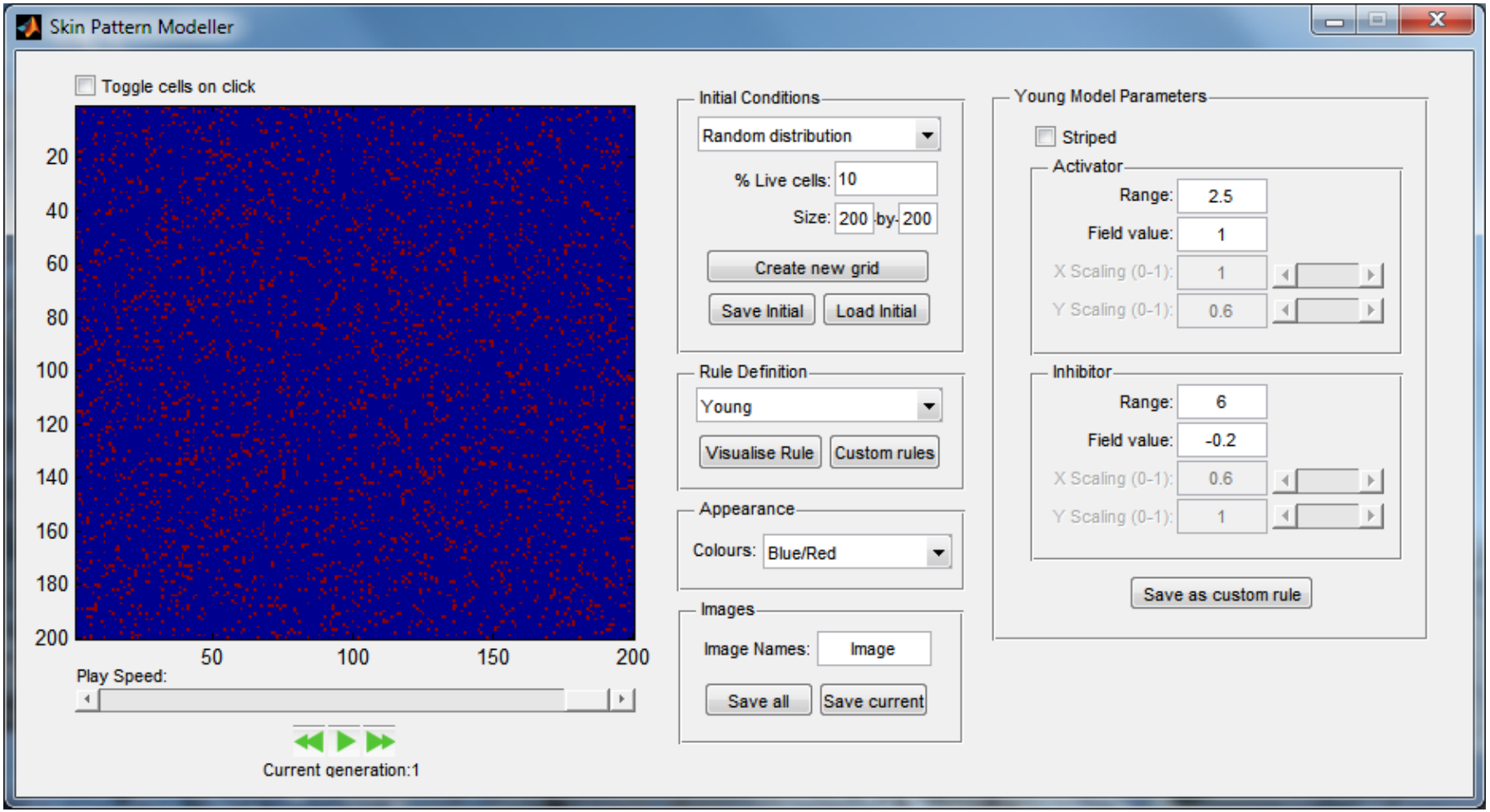


**Figure 9-6 GUI main window screenshot**

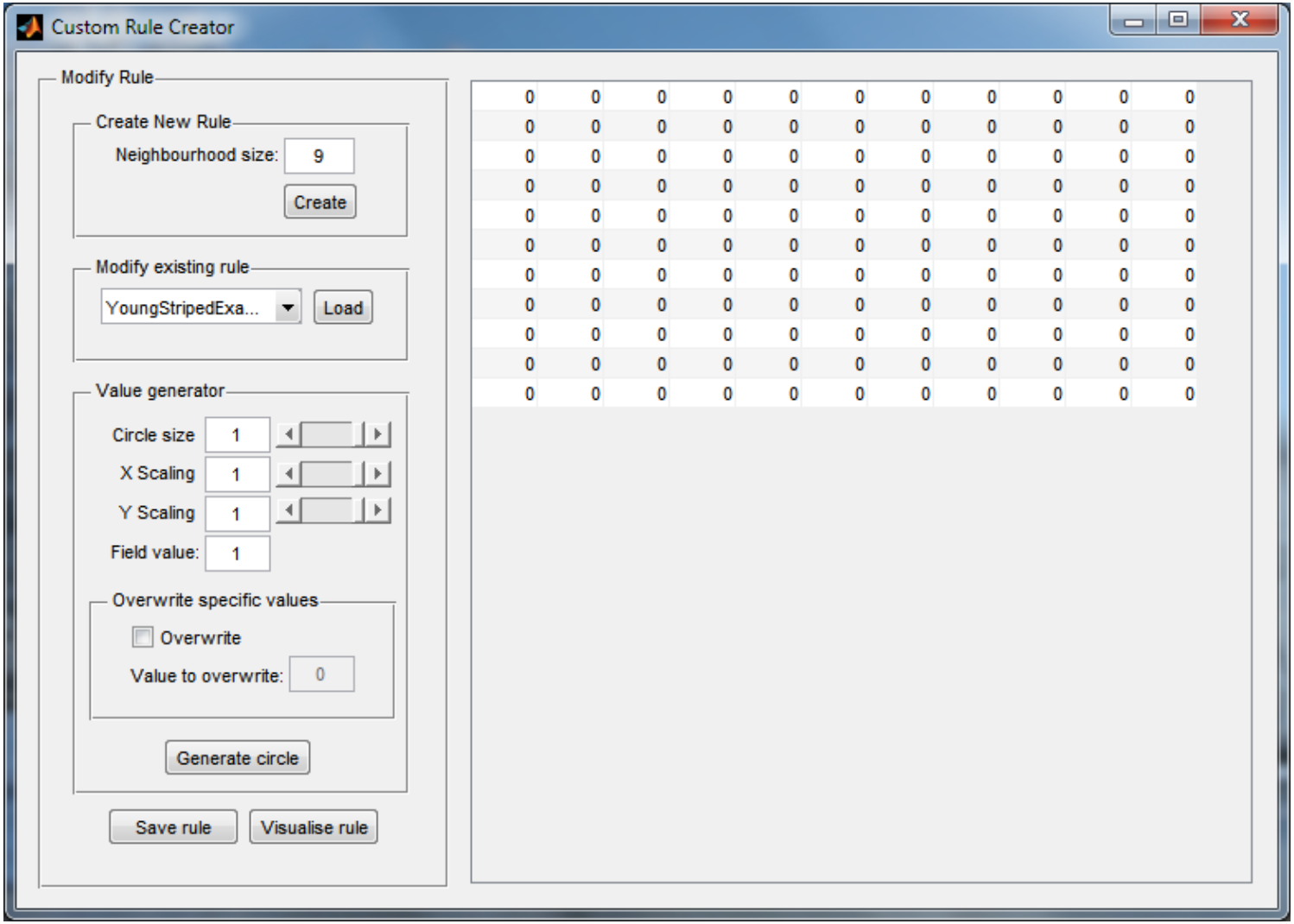


**Figure 9-7 Rule editor window screenshot**

## 9.4 Testing results

### 9.4.1 Comparison of generated patterns to published results

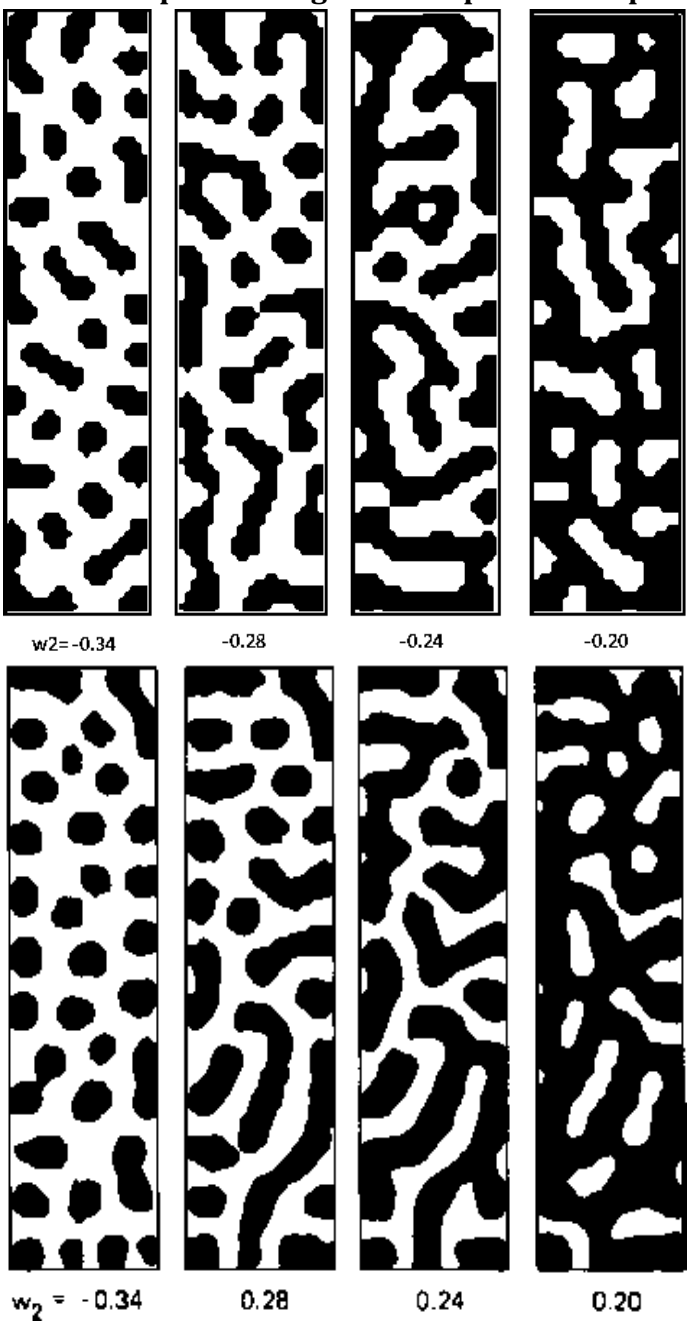


**Figure 9-8 Comparing generated results (top row) with published results (bottom row) using prototype method**

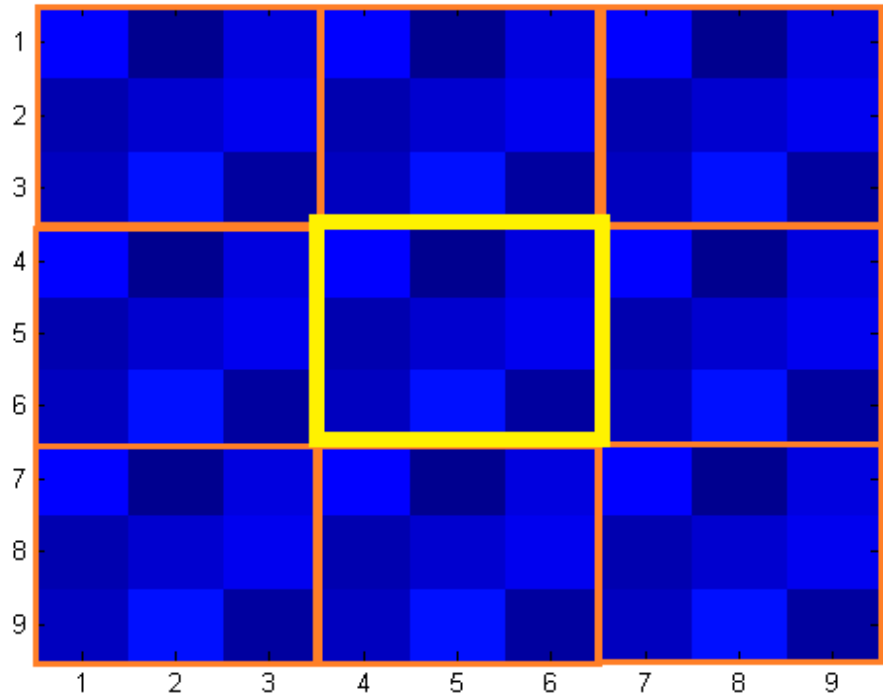


**Figure 9-9 Visual check of border padding**

### 9.4.2 circle_matrix()

circle_matrix(10,0.6,1,1) should produce a circle with circumference 6. Figure 9-10 shows the correctly generated matrix.

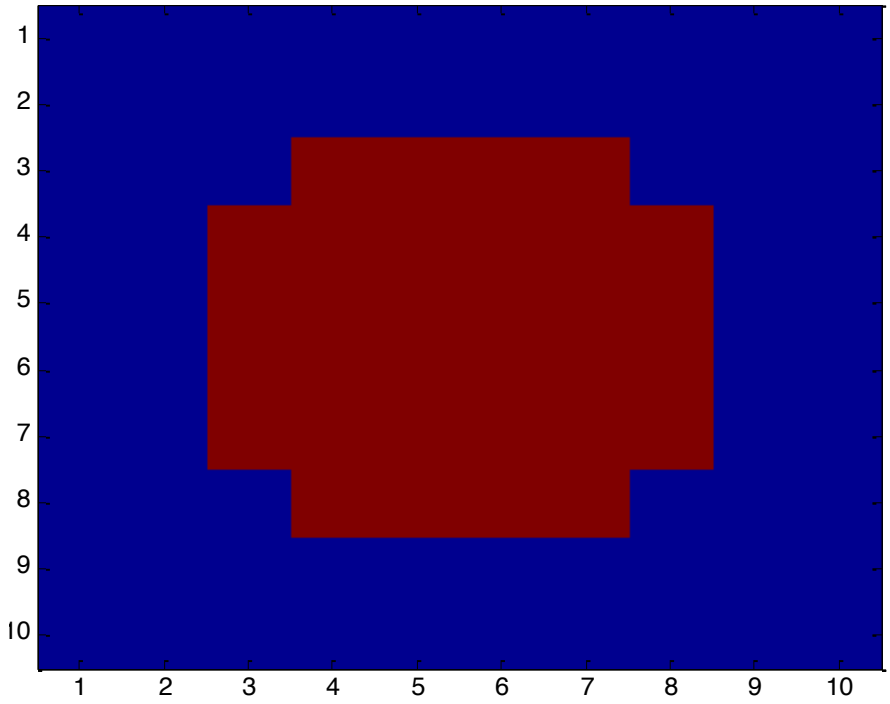


**Figure 9-10 Correctly generated circular matrix**

circle_matrix(10,1,1,0.6) should produce an ellipse, with width 10 and height 6. Figure 9-11 shows the correctly generated matrix.

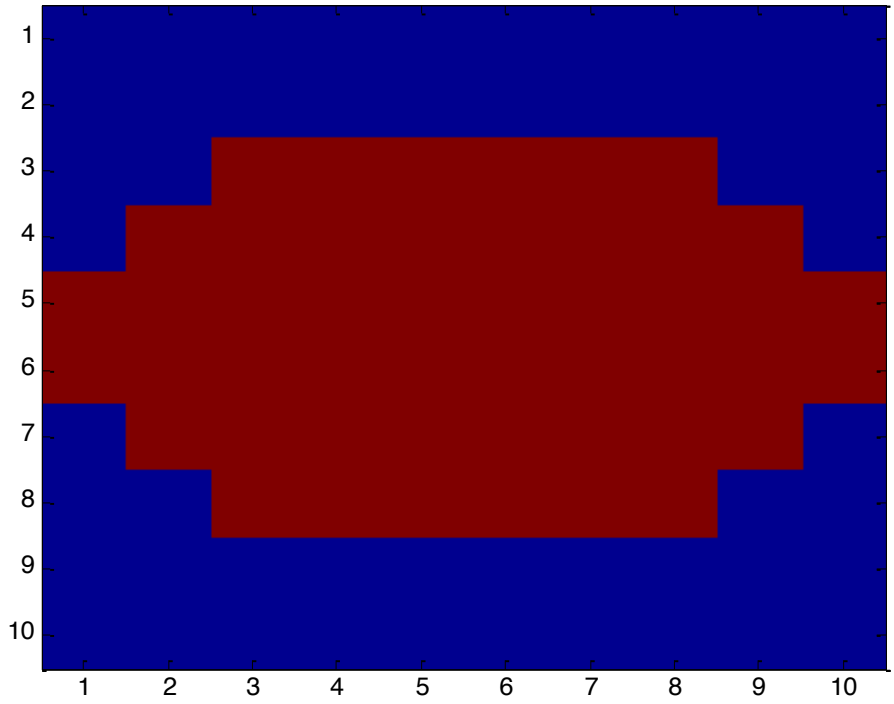


**Figure 9-11 Correctly generated elliptical matrix**

### 9.4.3 Young kernel generation

young_kernel(3,2.3,1,1,10,-0.2,1,1) should produce an activation area of radius 3, and an inhibition area of radius 20. As shown in Figure 9-12, the kernel is of the correct shape. The field values were also checked using manual clarification by selecting the relevant points on the grid, which displays the actual value.

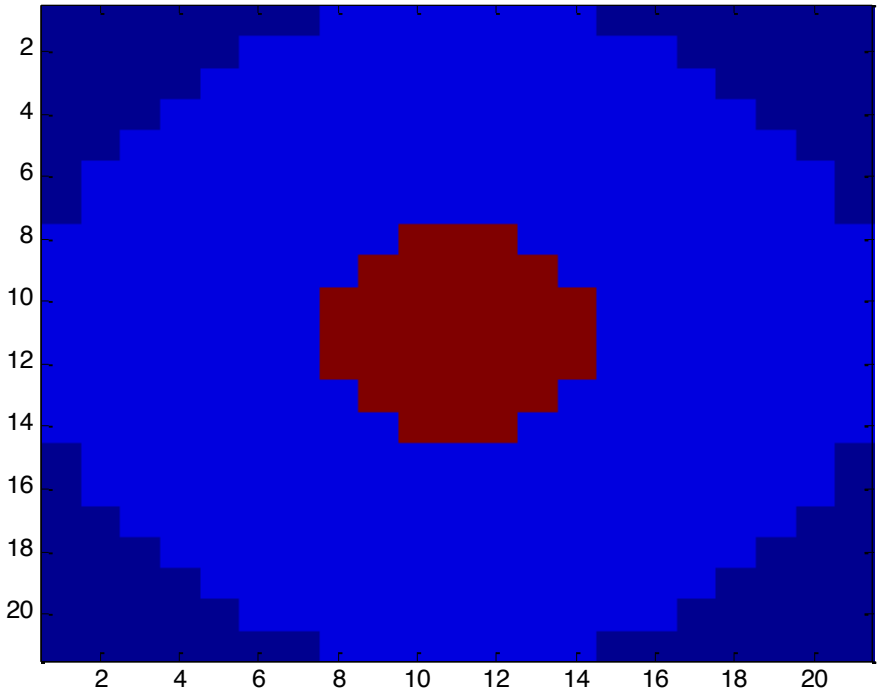


**Figure 9-12 Image produced to confirm correct kernel shape for Young rule**

An ellipse was also generated using young_kernel(2,1,1,0.6,6,0.24,0.6,1). The inhibitor is set to positive 0.24, so that the region can be seen in the image. The values are also multiplied by 100 to improve contrast in the image. The values are generated in the correct locations and show the correct values (Figure 9-13).

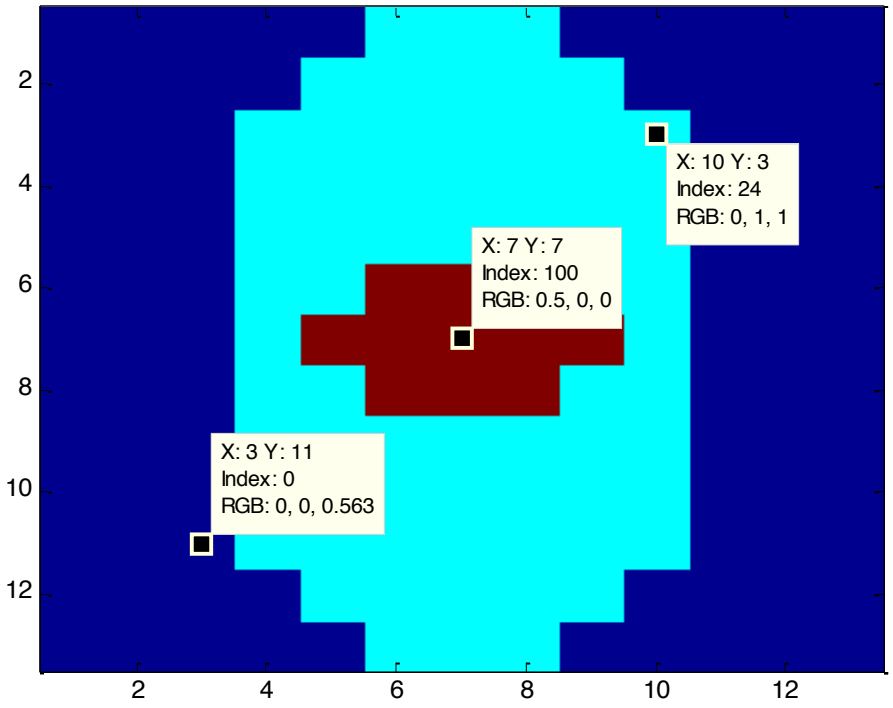

**Figure 9-13 Correctly generated Young kernel (elliptical)**

### 9.4.4 Wolfram kernel generation

wolfram_kernel(10,100) should produce a central area of value 1, size 3x3, and then two surrounding rings of value 10 and 100. Figure 9-14 shows the correctly generated kernel.

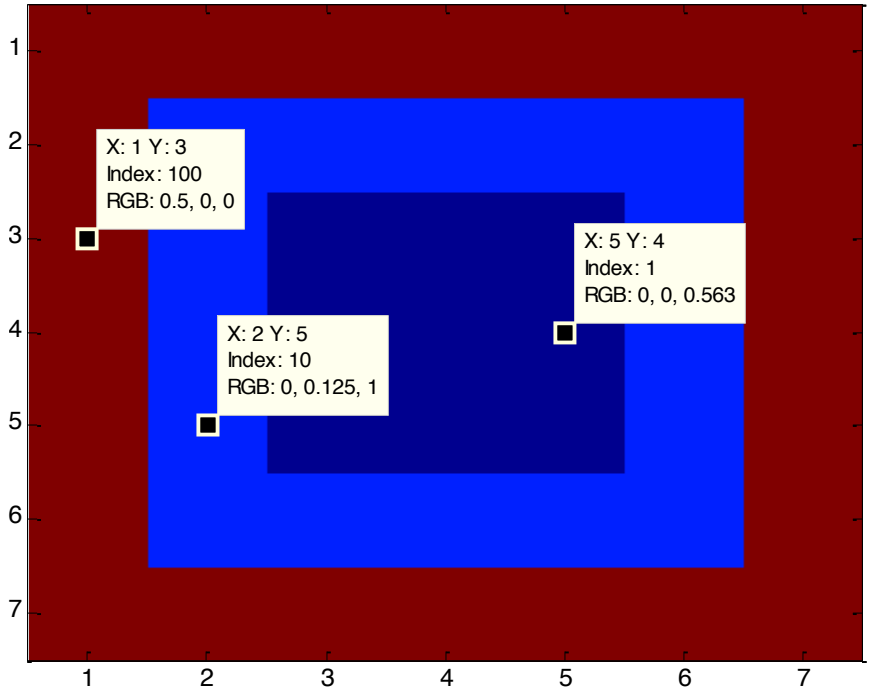


**Figure 9-14 Image produced to confirm correct kernel shape for Wolfram rule**

For striped wolfram kernels, only the cells horizontally or vertically, depending on the striped, should contain the values at distance 2 and 3. wolfram_kernel(10,100,'h') was called to ensure correct implementation. Figure 9-15 shows the correctly generated kernel, which was also working correctly for the vertical striping kernel.

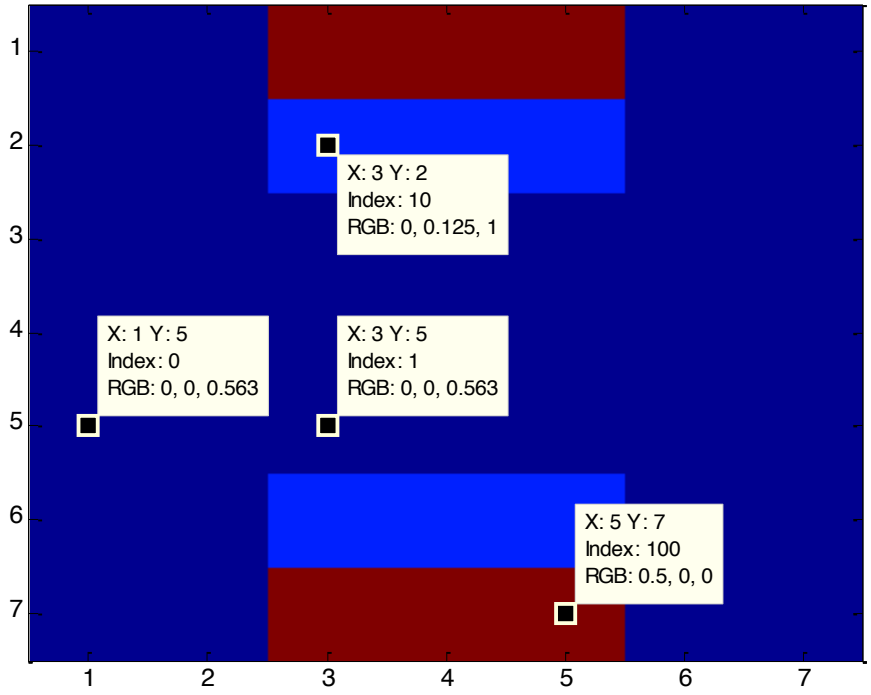


**Figure 9-15 Correctly generated Wolfram rule kernel (striped)**

### 9.4.5 Playback control

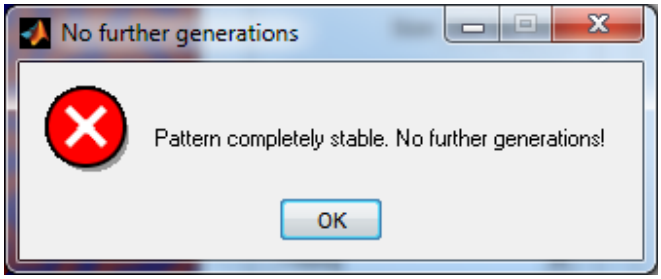


**Figure 9-16 Error message shown when trying to advance beyond stable state**

### 9.4.6 Saving images of simulations

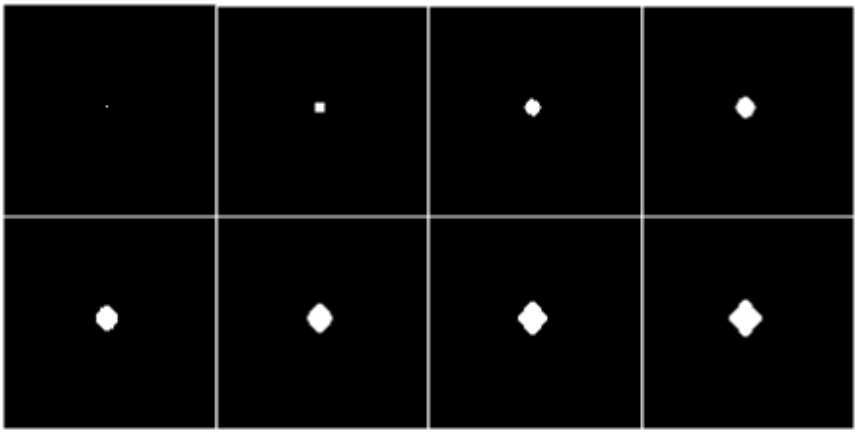

**Figure 9-17 Eight values stored when pressing 'save all'**

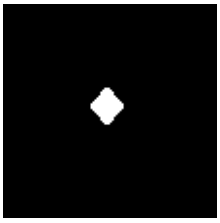

**Figure 9-18 Single image stored when pressing 'save current' (8th gen, same as previous image)**

### 9.4.7 Wolfram rule implementation

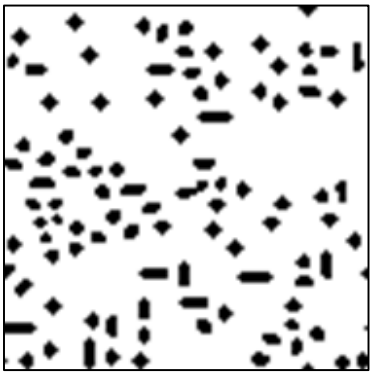

**Figure 9-19 Pattern generated with Wolfram rule, using -0.4 as both values**

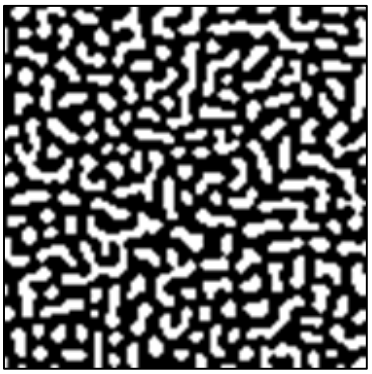

**Figure 9-20 Pattern generated with Wolfram rule, using -0.2 as both values**

When comparing [Figure 9-19] and [Figure 9-20] to Wolfram's own results [Figure 9-21] (limited resolution), they appear to match closely, further suggesting a correct implementation.

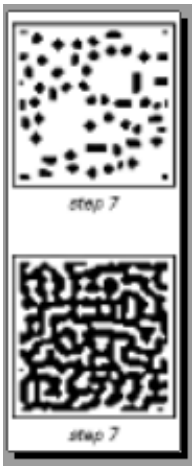


**Figure 9-21 Published results from Wolfram's 'A New Kind of Science'**

Furthermore, [Figure 9-22] and [Figure 9-23] show striped patterns generated using Wolfram's rule, which also match closely the (limited) results presented in the research [Figure 9-24].

**Figure 9-22 Horizontally striped pattern generated with Wolfram rule by software**

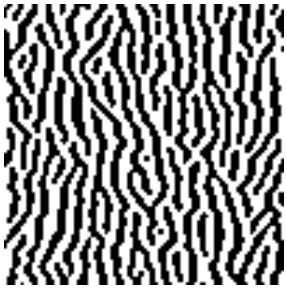

**Figure 9-23 Vertically striped pattern generated with Wolfram rule by software**


**Figure 9-24 Published results of striped Wolfram patterns**

### 9.4.8 Rule modification and custom rules

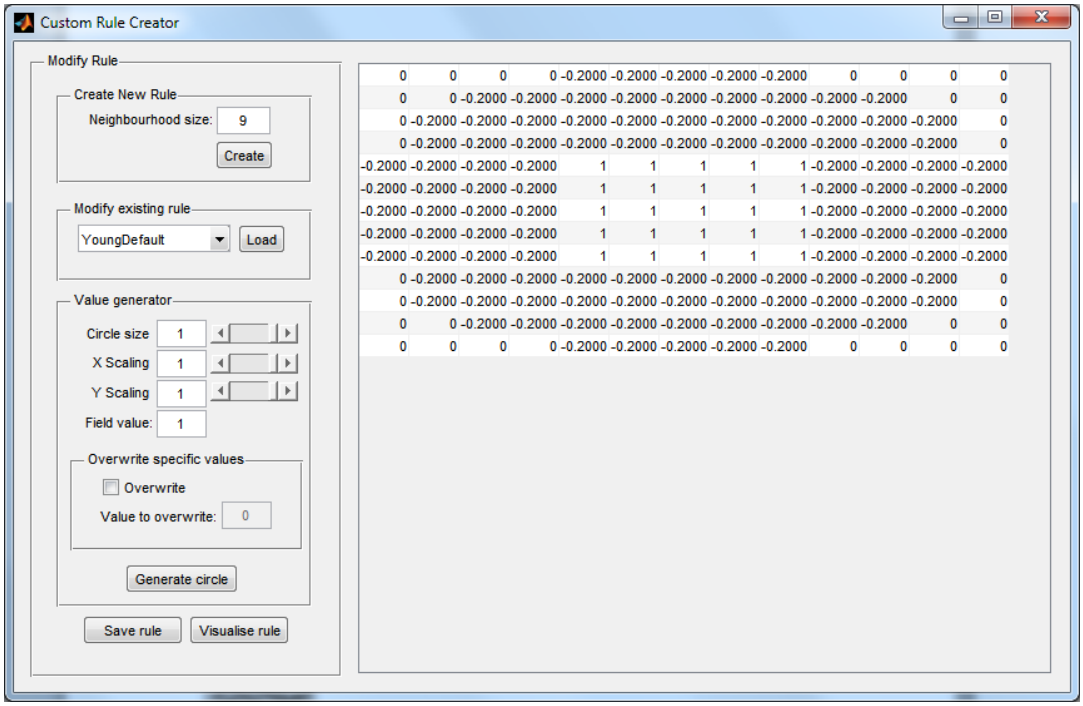


**Figure 9-25 Screenshot of correctly loaded rule in custom rule screen**

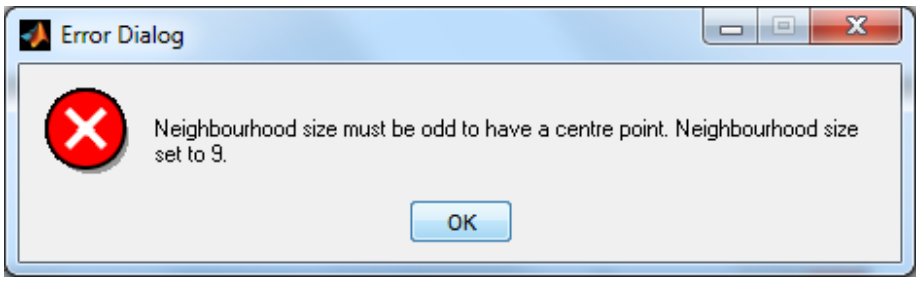


**Figure 9-26 Attempting to create an even-sized neighbourhood**

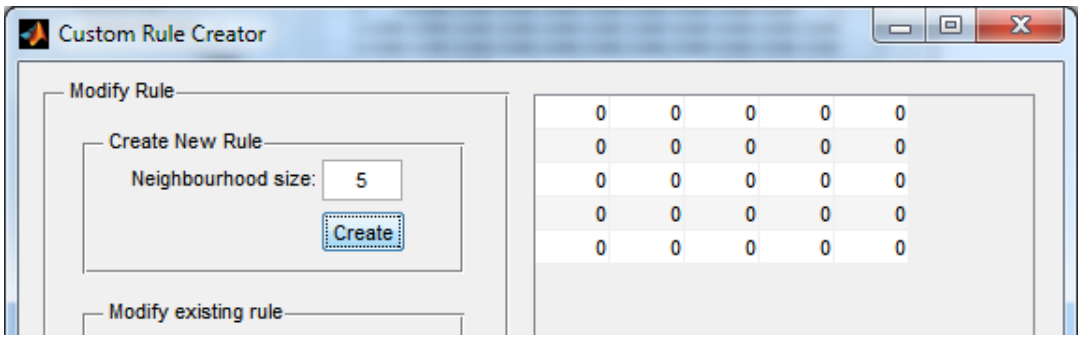


**Figure 9-27 Successful generation of a valid empty kernel**

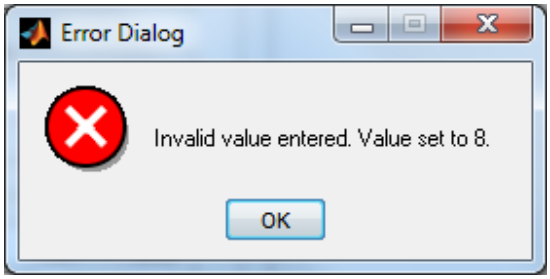


**Figure 9-28 Attempting to enter a non-numeric neighbourhood size**

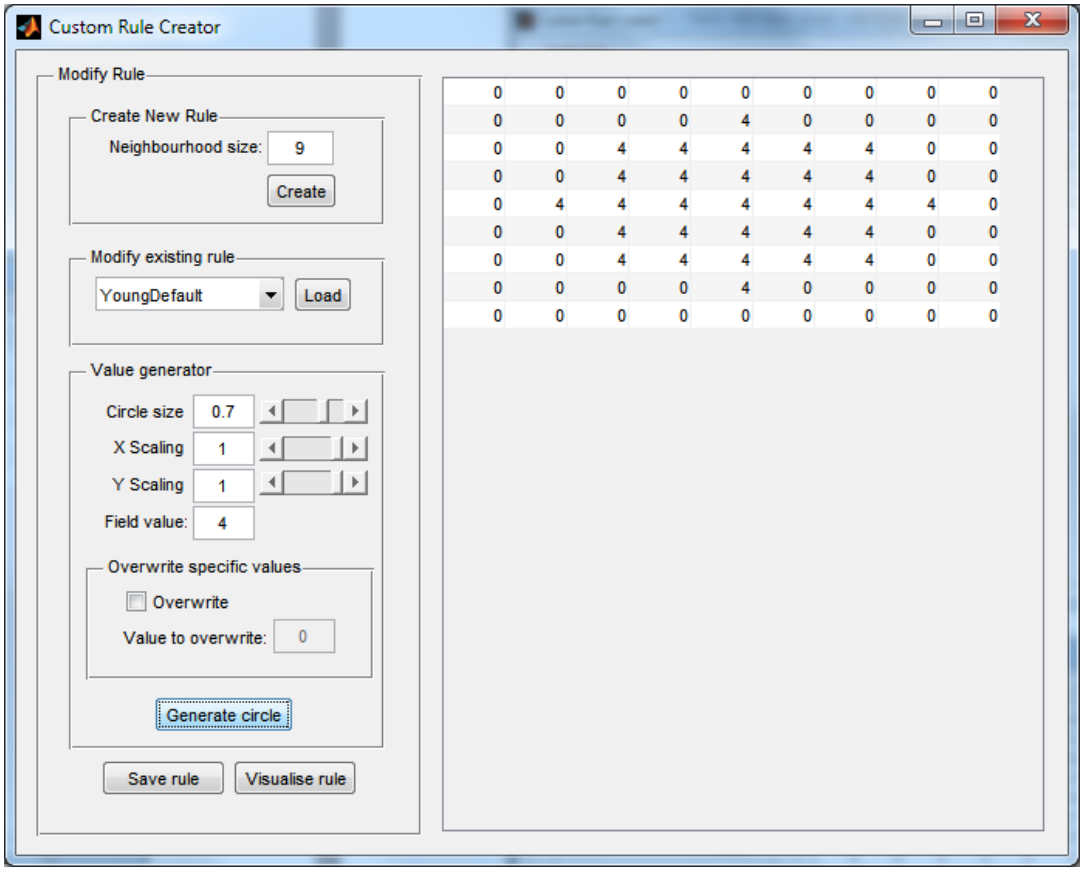


**Figure 9-29 Correctly generated circle (0.7 times size of full matrix)**

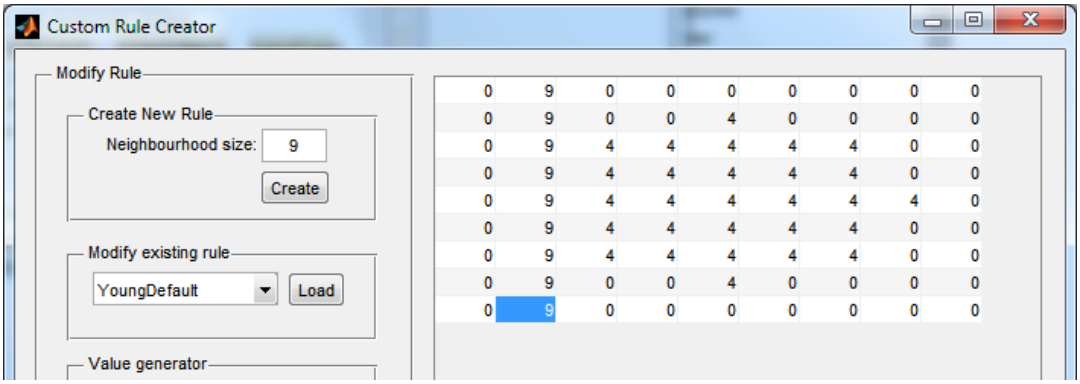


**Figure 9-30 Grid demonstrating manual, cell-by-cell editing (column set to 9s)**

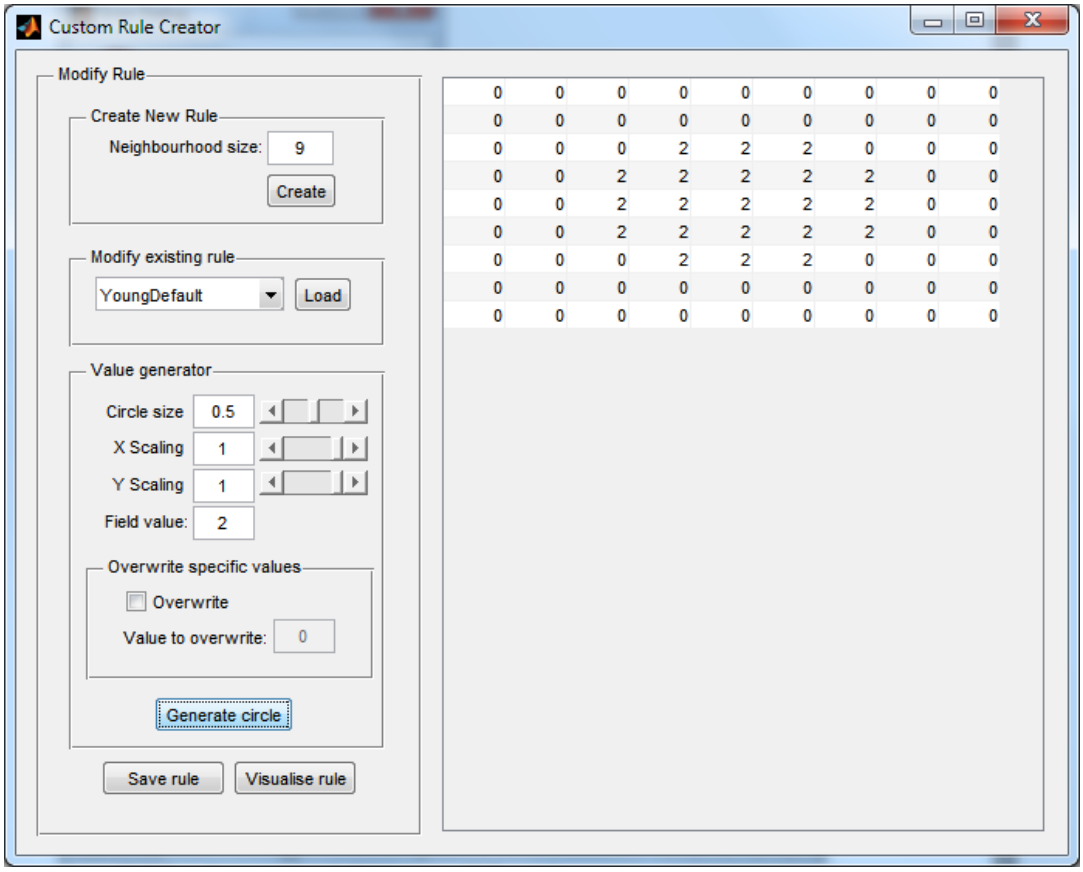


**Figure 9-31 Small, inner circle of 2s generated**

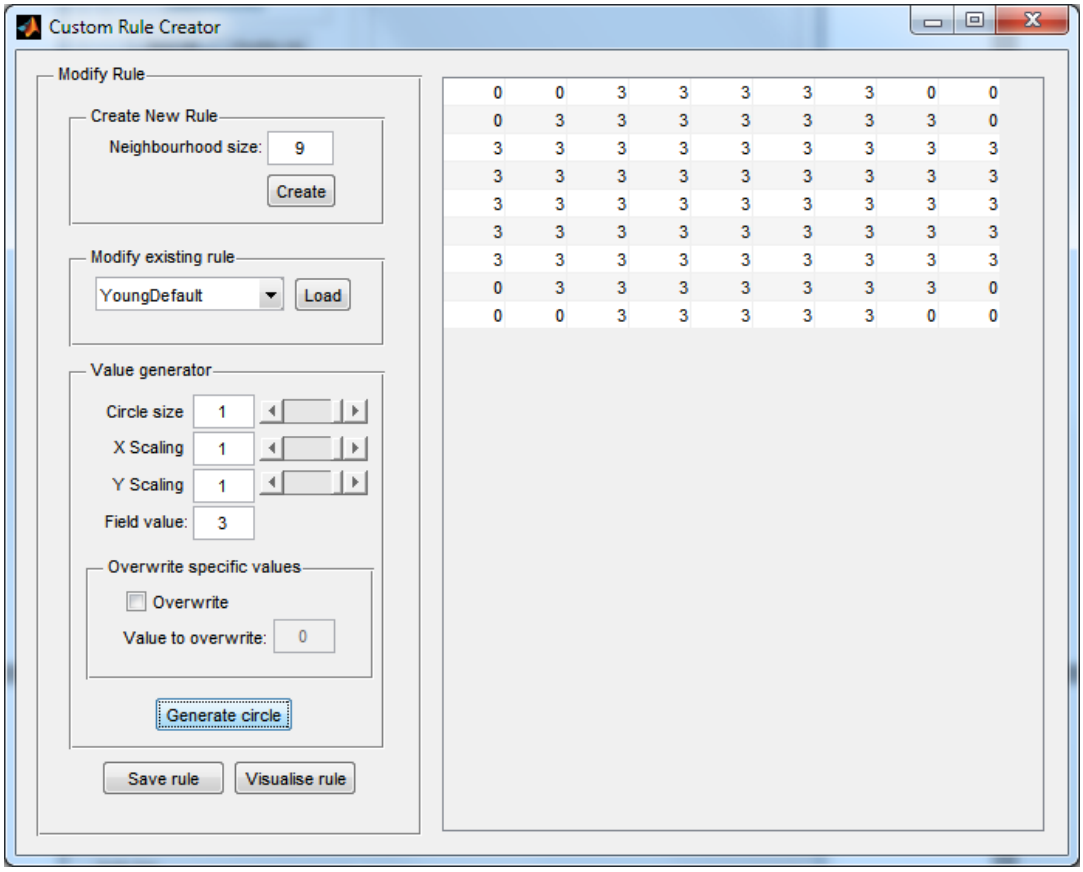


**Figure 9-32 Larger circle generated with no overwrite value specified, so overwrites everything**

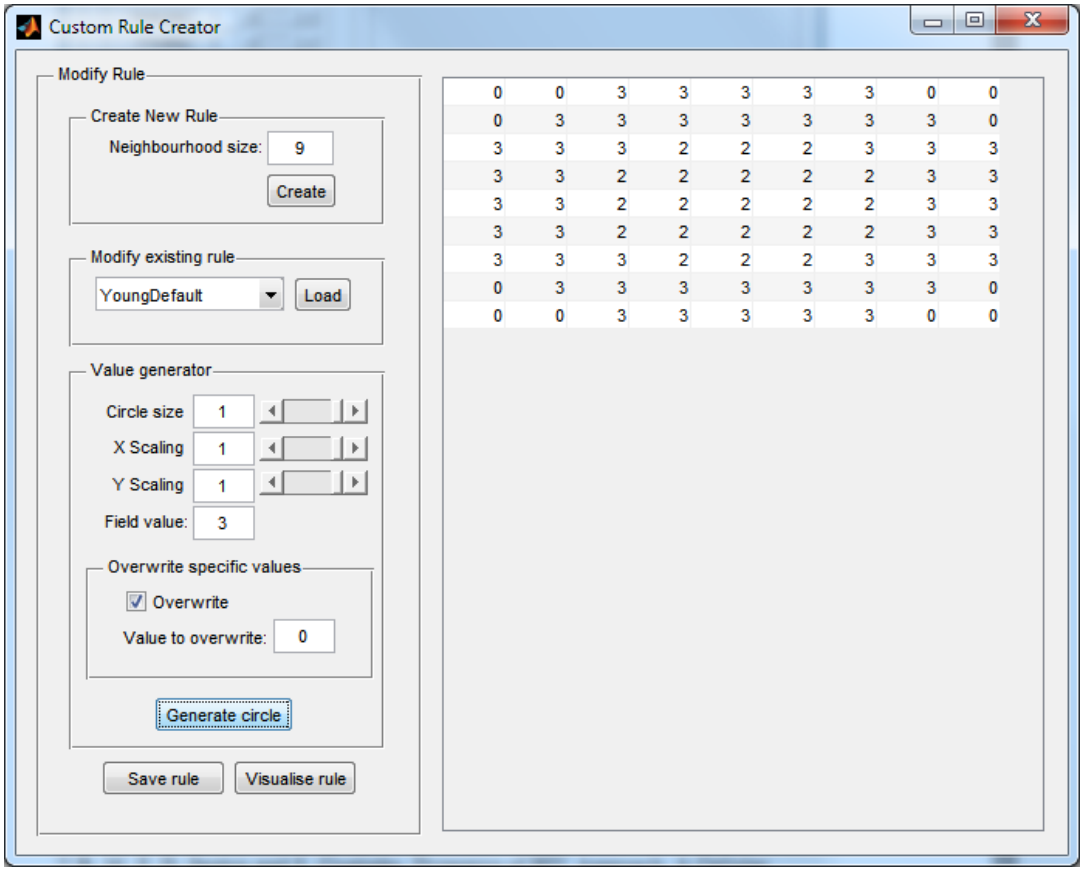


**Figure 9-33 Larger circle generated with overwrite enabled and set to 0, leaving original smaller circle in place and only overwriting 0s**

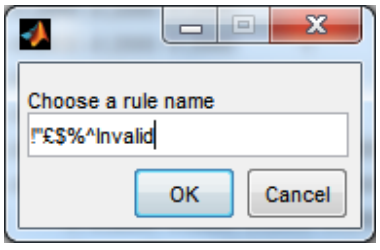


**Figure 9-34 An invalid rule name entered**

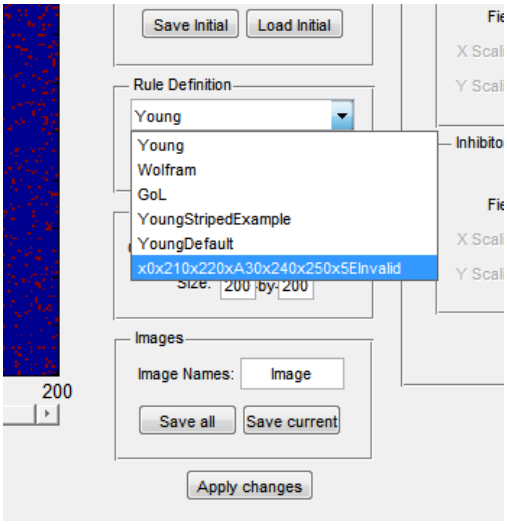


**Figure 9-35 Rule name sanitised and instantly added to main screen dropdown**

### 9.4.9 Saving & loading of initial conditions and simulations

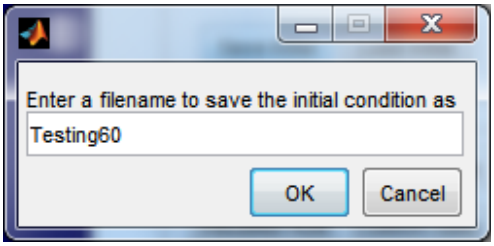


**Figure 9-36 Grid saved as 'Testing60'**

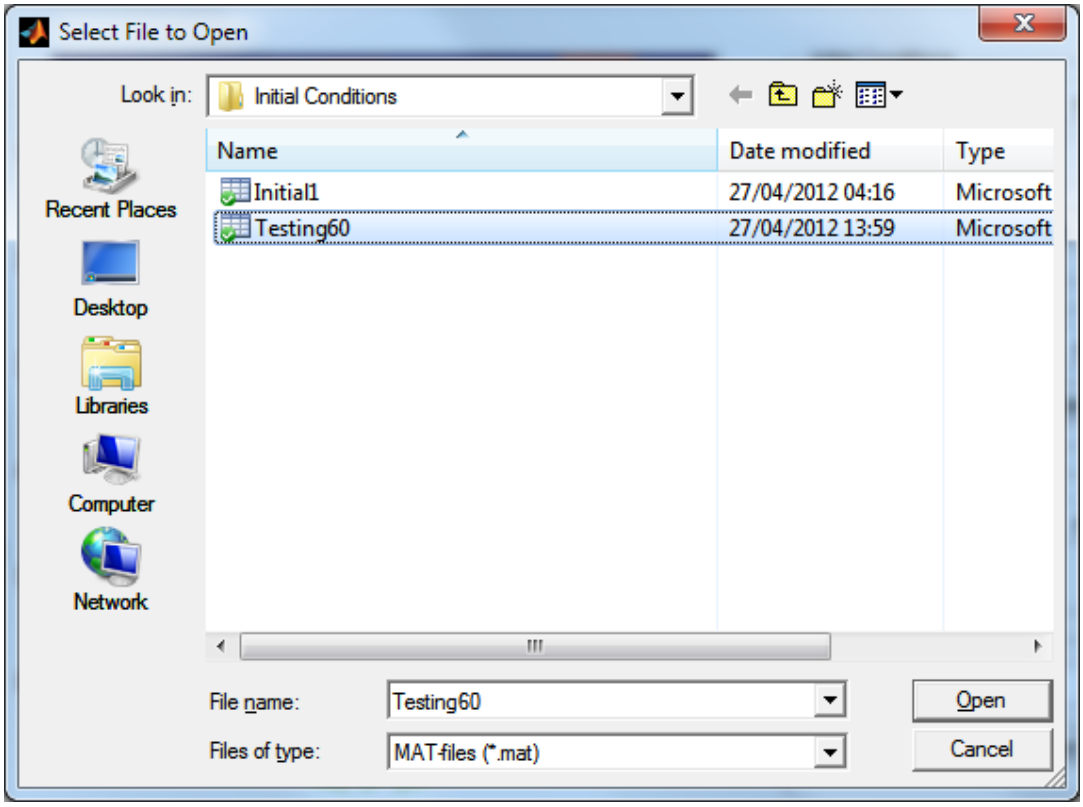


**Figure 9-37 Testing60 loaded from file**

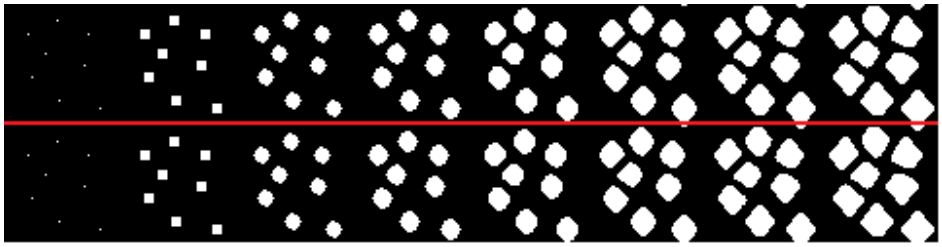

**Figure 9-38 Images generated before (top row) and after (bottom row) saving to file**

### 9.4.10 Visualise rule kernels

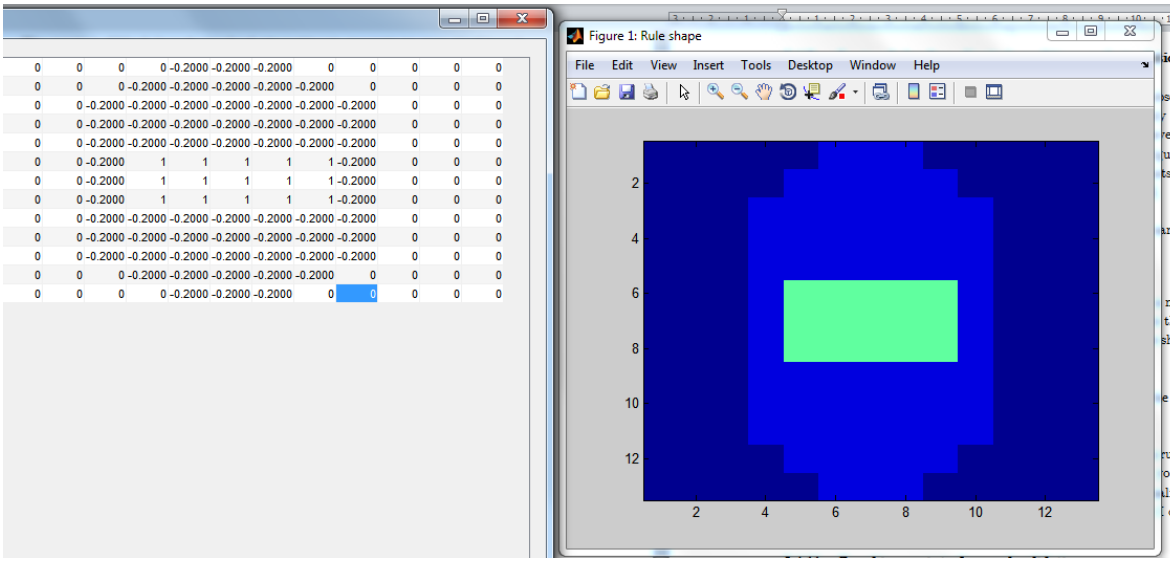


**Figure 9-39 Image matches the shape of the kernel seen on the rule editor**

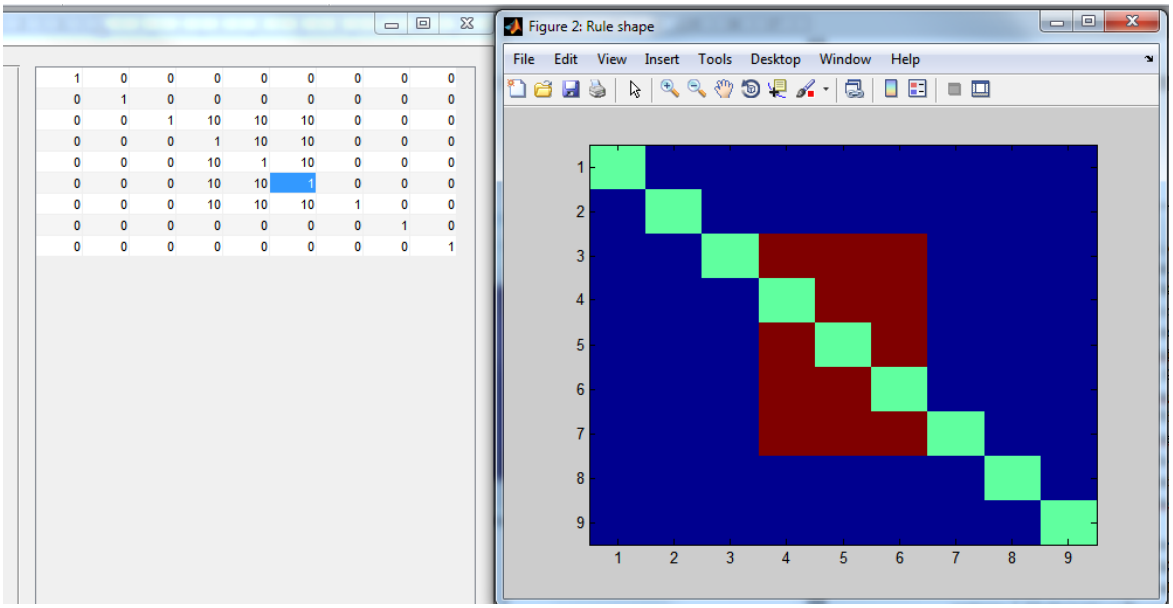


**Figure 9-40 Image matches the shape of the kernel seen on the rule editor**

# 10 References

[i] http://mathworld.wolfram.com/CellularAutomaton.html

[ii] Christian Burks, Doyne Farmer, Towards modelling DNA sequences as automata, Physica D: Nonlinear Phenomena, Volume 10, Issues 1-2, January 1984, Pages 157-167, ISSN 0167-2789, 10.1016/0167-2789(84)90258-6.

[iii] On Cellular Automaton Approaches to Modeling Biological Cells, Mark S. Alber, Maria A. Kiskowski, James A. Glazier and Yi Jiang, in Mathematical Systems Theory in Biology, Communication, and Finance, D. N. Arnold and F. Santosa, editors (IMA 134, Springer-Verlag, New York, 2002), 1-40.

[iv] R. M. Z. D. Santos and S. Coutinho. Dynamics of HIV Approach: A Cellular Automata Approach. Phys. Rev. Lett., 87(16):102–104, 2001.

[v] Turing, A. M. (1952). "The chemical basis of morphogenesis." Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences 237(641): 37.

vi Kondo, S. and T. Miura (2010). "Reaction-diffusion model as a framework for understanding biological pattern formation." science 329(5999): 1616.

[vii] Gierer, A. and H. Meinhardt (1972). "A theory of biological pattern formation." Biological Cybernetics 12(1): 30-39.

[viii] Meinhardt, H. and Gierer, A. (2000), Pattern formation by local self-activation and lateral inhibition. BioEssays, 22: 753–760.

[ix] Koch, A.J., Meinhardt, H., 1994. Biological pattern formation: from basic mechanism to complex structures. Rev. Mod. Phys. 66, 1481±1507.

[x] Nakamasu, A., G. Takahashi, et al. (2009). "Interactions between zebrafish pigment cells responsible for the generation of Turing patterns." Proceedings of the National Academy of Sciences 106(21): 8429.

[xi] Jonathan B.L, B. (1981). "A model for generating aspects of zebra and other mammalian coat patterns." Journal of Theoretical Biology 93(2): 363-385, page 379

[xii] S. K. Frost and G. M. Malacinski, The developmental genetics of pigment mutants in the Mexican axolotl, Devel. Genet. 1:271-294 (1980).

[xiii] F. Kirschbaum, Untersuchung iiber das Farbmuster des Zebrabarbe Brach.vdunio rerio (Cyprinidae, Teleostei), Wilhelm Roux's Arch. 177:129-152 (1975).

[xiv] . R. Schmidt, Chromatophore development and cell interactions in the skin of

Xiphophorine fish, Wi/he/m Roux's Arch. 184:115-134 (1978).

[xv] Visual cortex maps are optimized for uniform coverage, N. V. Swindale, D. Shoham, A. Grinvald, T. Bonhoeffer, M. Hübener, Nature neuroscience, Vol. 3, No. 8. (August 2000), pp. 822-826.

[xvi] N. V . Swindale, A model for the formation of ocular dominance stripes, Proc. Roy. Soc. London Ser. B 208:243-264 (1980)

[xvii] David A, Y. (1984). "A local activator-inhibitor model of vertebrate skin patterns." Mathematical Biosciences 72(1): 51-58, page 54

[xviii] David A, Y. (1984), page 54

[xix] C.P. Gravan, R. Lahoz-Beltra, Evolving morphogenetic fields in the zebra skin patterns based on Turing's morphogen hypothesis, Int. J. Appl. Math. Comput. Sci. 14, 351 (2004)

[xx] Graván and Lahoz-Beltra, page 356

[xxi] Wolfram, S. and M. Gad-el-Hak (2003). "A New Kind of Science." Applied Mechanics Reviews 56: B18

[xxii] Wolfram, S. and M. Gad-el-Hak (2003), page 423

[xxiii] Wolfram, S. and M. Gad-el-Hak (2003), page 423

[xxiv] Wolfram, S. and M. Gad-el-Hak (2003), page 424

[xxv] Wolfram, S. and M. Gad-el-Hak (2003), page 424

[xxvi] Wolfram, S. and M. Gad-el-Hak (2003), page 425

[xxvii] Wolfram, S. and M. Gad-el-Hak (2003), page 425

[xxviii] Wolfram, S. and M. Gad-el-Hak (2003), page 427

[xxix] Wolfram, S. and M. Gad-el-Hak (2003), page 428

[xxx] Wolfram, S. and M. Gad-el-Hak (2003), page 429

[xxxi] http://www.fourmilab.ch/cellab/manual/chap2.html

[xxxii] http://www.cs.sjsu.edu/~rucker/biography.htm

[xxxiii] http://www.fourmilab.ch/cellab/manual/chap5.html

[xxxiv] http://www.fourmilab.ch/cellab/

[xxxv] http://www.fourmilab.ch/cellab/

[xxxvi] http://www.fourmilab.ch/cellab/manual/chap3.html#t2

[xxxvii] http://www.fourmilab.ch/cellab/manual/chap2.html (ASCII section)

[xxxviii] http://www.cs.sjsu.edu/~rucker/cellab.htm

[xxxix] http://paraschopra.com/sourcecode/celllab/CellLab_src.zip (source code, contains readme)

[xl] http://wordnetweb.princeton.edu/perl/webwn?s=high-level%20language

[xli] http://oxforddictionaries.com/definition/object-oriented

[xlii] http://www.oracle.com/technetwork/java/javase/overview/index.html

[xliii] http://www.mathworks.co.uk/products/matlab/

[xliv] http://python.org/

[xlv] http://wiki.python.org/moin/GuiProgramming

[xlvi] http://matplotlib.sourceforge.net/

[xlvii] http://www.scipy.org/NumPy_for_Matlab_Users

[xlviii] http://pypi.python.org/pypi/CAGE/1.1.2

[xlix] http://www.mathworks.co.uk/products/matlab/description5.html

[l] David A, Y. (1984), page 54

[li] http://www.mathworks.co.uk/help/techdoc/ref/tic.html

[lii] David A, Y. (1984), page 55

[liii] http://www.mathworks.co.uk/help/techdoc/ref/colormap.html

[liv] http://www.mathworks.co.uk/help/toolbox/images/ref/padarray.html

[lv] http://www.mathworks.co.uk/help/techdoc/ref/imwrite.html

[lvi] http://www.mathworks.co.uk/help/techdoc/ref/genvarname.html

[lvii] Wolfram, S. and M. Gad-el-Hak (2003), page 427

[lviii] Wolfram, S. and M. Gad-el-Hak (2003), page 423

[lix] http://www.nature.com/ng/journal/v44/n3/full/ng.1090.html