Continuous-Valued Cellular Automata in Two Dimensions

by Rudy Rucker

Department of Computer Science, Emeritus
San Jose State University, San Jose, CA

rudy@rudyrucker.com

This paper appeared in *New Constructions in Cellular Automata* (Santa Fe Institute Studies in the Sciences of Complexity Proceedings), edited by David Griffeath and Cristopher Moore , Oxford University Press 2003.

Abstract. We explore a variety of two-dimensional continuous-valued cellular automata (CAs). We discuss how to derive CA schemes from differential equations and look at CAs based on several kinds of non-linear wave equations. In addition we cast some of Hans Meinhardt's activator-inhibitor reaction-diffusion rules into two dimensions. Some illustrative runs of *CAPOW*, a CA simulator, are presented.

1. Introduction

A *cellular automaton*, or CA, is a computation made up of (1) finite elements called cells. Each cell contains the same type of state. The cells are updated in (2) parallel, using a rule which is (3) homogeneous and (4) local.

In slightly different words, a CA is a computation based upon a (1) grid of cells, with each cell containing an object called a *state*. The states are updated in discrete steps, with all the cells being effectively updated at the same time. (3) Each cell uses the same algorithm for its update rule. (4) The update algorithm computes a cell's new state by using information about the states of the cell's nearby spacetime neighbors, that is, using the state of the cell itself, using the states of the cell's nearby neighbors, and using the recent prior states of the cell and its neighbors.

The states do not necessarily need to be single numbers, they can also be data structures built up from numbers. A CA is said to be *discrete-valued* if its states are built from integers, and a CA is *continuous-valued* if its states are built from real numbers.

As Norman Margolus and Tommaso Toffoli have pointed out, CAs are well-suited for modeling nature [1]. The parallelism of the CA update process mirrors the uniform flow of time. The homogeneity of the CA update rule across all the cells corresponds to the universality of natural law. And the locality of CAs reflect the fact that nature seems to forbid action at a distance.

The use of finite spacetime elements for CAs are a *necessary evil* so that we can compute at all. But one might argue that the use of discrete states is an *unnecessary evil*. In the old days, speed and storage considerations made it impractical to carry out large CA computations using real numbers as the cell states, but today's desktop machines no longer have these limitations. The author and his students have developed a shareware software package for Windows called *CAPOW*, which we have used for exploring continuous-valued CAs [2]. The paper [3] contains information about our investigations

of one-dimensional continuous-valued CAs, and the present paper presents some of the phenomena found in two-dimensional continuous-valued CAs.

In the CAs we've been investigating, we take "real number" to mean IEEE single-precision floating-point number, what the C language terms a *float* rather than a *double*. We have experimented with double precision floating point numbers, but their use does not seem to change the qualitative features of our simulations. Double precision floating point numbers have the drawback of requiring larger memory buffers and of cutting simulation speed. Because of considerations of speed and memory, we have been looking at relatively small 2D CAs, with a dimension 120 cells wide by 90 cells high.

In Section 2 of this paper we discuss how we derive CA schemes from sets of differential equations and Section 3 presents some material relating to our specific methods of simulation. In Section 4 we discuss some two-dimensional continuous-valued CAs that are based on reaction-diffusion systems that use an activator-inhibitor reaction. In Section 5 we look at CAs based on linear and non-linear wave equations and in Section 6 we briefly consider the possibility of developing some "reaction-wave" CAs. And Section 7 suggests some paths for further investigations.

Before proceeding, let's confront three possible objections to the study of continuous-valued cellular automata.

Objection 1. Since you are running your computation on a digital machine, your so-called continuous values are really discrete numbers, so you are doing nothing new.

Over typical lab scales of minutes and hours, there is a qualitative difference between a CA whose state is only a few bits, and a CA whose state is a floating point number. You can indeed simulate crude things like heat flow with only a few hundred discrete states, but numerical viscosity kills off subtler continuum behaviors like wave motion. With states that are single precision floating point numbers, simulation of a one or two dimensional wave will persist through millions of updates, but if we coarsen the grain down to something like ten bits of state, a wave simulation quickly dies out. Here's a table of results gotten by simulating a one-dimensional wave with the *CAPOW* software, which contains a "State Grain" control for altering the coarseness of the real numbers used.

Coarseness of "Reals"	**Number of Updates Until a Wave Dies Out.**
0.1	50 updates
0.01	200 updates
0.001	800 updates

Table 1. The longevity of a linear wave scheme using varying minimum sizes of real number.

Objection 2. When you use floating point numbers, computational round-off destroys the possibility of having time-reversible rules.

This issue was raised by Norman Margolus during the "Constructive CAs" conference. Margolus reasons that since physics is reversible, the CAs we use should be reversible as well. Margolus's concern about the *CAPOW* program was that its use of floating-point numbers for its "real numbers" would make the rules irreversible due to computational round-off.

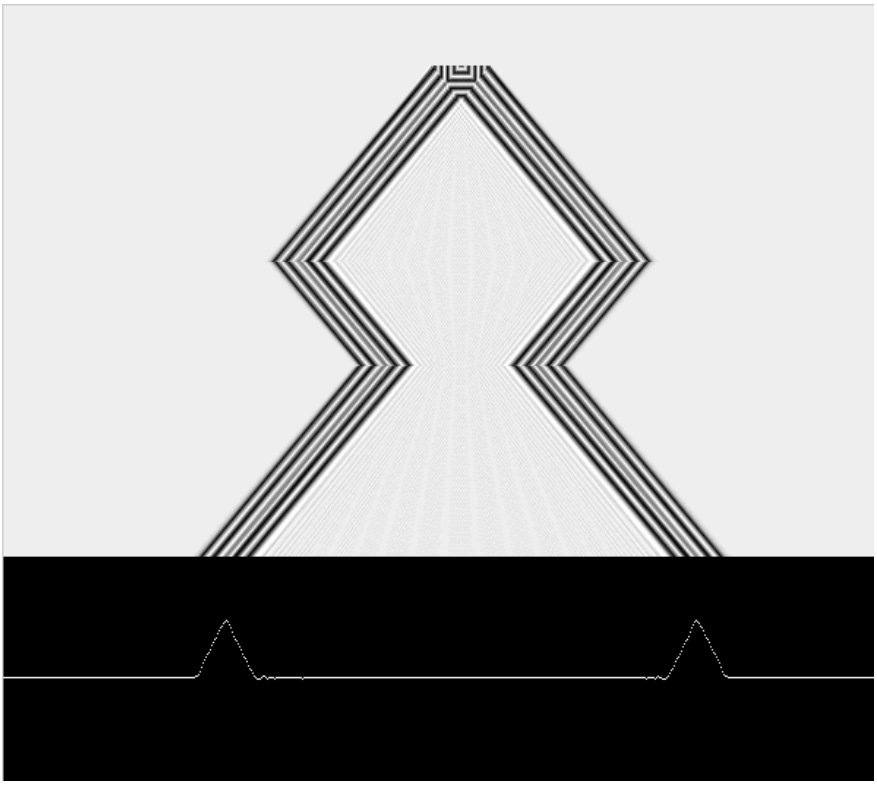

Figure 1. A one-dimensional wave schema being run with a **Wave** constant of 0.694. The rule was seeded with a single triangular spike and was time-reversed twice. The figure shows an instantaneous snapshot of the cells' intensity values at the bottom, while the upper part shows a spacetime diagram of the intensity patterns, with the earlier times at the top and the later times at the bottom. This simulation was "time-reversed" by exchanging the "past" and "future" cell buffers.[Reversible Wave B&W.tif]

Tests conducted since the conference reveal that the computational round-off is not noticeable enough to destroy the possibility of rule reversibility over the simulation runs that we use, typically on the order of a thousand to a hundred thousand updates. Figure 1 shows an example of a one-dimensional wave equation rule being reversed.

Of course our rules can only be reversible when they are based on a reversible scheme such as the Wave Scheme introduced in Section 3. The Diffusion Scheme which we use is inherently irreversible. Although it is possible to model diffusion in terms of the reversible motions of a deterministic gas of "heat particles," such a strategy would seem to limit us to simulating systems much smaller and simpler than those investigated here.

Objection 3. To study continuous-valued CAs is to repeat existing work in numerical analysis and finite element simulations.

Finite element methods are indeed an inspiration for continuous-valued CAs. But the CA approach has a different mind-set and leads to different kinds of investigations. The CA approach involves: an emphasis on *experiment* and observation rather than on theory and proof; an *artificial life* orientation in which one is actively on the lookout for unexpected and emergent properties of the simulation; and the use of *genetic algorithm*

methods for effectively searching the large phase spaces of the rules. A historical distinction between CA work and finite elements simulations is that the latter tend to be run on supercomputers, while CA programs are usually rapidly running, attractive, interactive shareware graphics programs for desktop machines. It is our hope that thinking in terms of continuous-valued CAs can lead to a productive unification of work from diverse fields.

2. Continuous-Valued CAs and Differential Equations

Most of the continuous-valued rules we've investigated so far have been inspired by systems of differential equations. These include one-dimensional and two-dimensional forms of equations based on diffusion, wave motion, oscillators, activator-inhibitor reactions, and various combinations of these equations.

Our cell states typically include a real-number value **u**, and our CA update scheme typically has the form **uNew = Update(u,…)**. Various neighbor-state values can appear as arguments to the Update rule. We write **uNew** rather than **u** on the left because in order to preserve parallelism we think in terms of first computing the **uNew** values for every cell before then starting to view these as the current **u** values.

In working with CA schemes of this nature, one needs to worry both about the numerical accuracy of a scheme and about its stability. (See [3].) The accuracy relates to how well the CA is simulating an actual differential equation. The stability relates to whether or not the CA simulation goes completely out of control. When a rule enters an unstable regime it will generally produce arbitrarily large and small values. A good heuristic in seeing if a scheme is likely to be stable is that it should set **uNew** to something the size of **u** plus something the size of a constant times the difference between two cell values.

In order to discuss our practices in converting systems of equations into schemes for CA rules, let's suppose we are looking for a scheme to be used at a given cell **C**. In one dimensional rules, we call the cells left and right neighbors L and R. In two-dimensional rules, we call the cell's neighbors **E**, **NE**, **N**, **NW**, **W**, **SW**, **S**, and **SE**. In two dimensions we distinguish between the von Neumann neighborhood of all eight neighbors, and the von Neumann neighborhood of only the four neighbors **E**, **N**, **W**, and **S**.

Table 2 lists some symbols we'll use for various cell neighborhood values. We'll use **uPast** and **uNew** to stand for the value in the cell at, respectively, the prior and the following update. It's going to be useful to use the term **uTimeAvg** to stand for the average of these two "time neighbors." And we'll use **uNabeAvg** to stand for the average state values in **C**'s immediate neighbors, excluding **C** itself. In addition, we write **uR** and **uL** to stand for the **u** values in the cell's right and left neighbors **R** and **L**, write use **uE**, **uN**, **uW**, and **uS** to stand for the intensity values of the update cell's four von Neumann neighbors, and so on.

In a one-dimensional case where we only look at nearest neighbors, **uNabeAvg** is (**uL** + **uR)/2**. In a two-dimensional case where we use the von Neumann neighborhood, **uNabeAvg** is **(uE+uN+uW+uS)/4**. And in the two-dimensional Moore neighborhood case, we choose to weight the corner cells a bit less, and use a **uNabeAvg** of **(uE+uN+uW+uS+ 0.75*(uNE + uNW** + u**SW** + u**SE) / 7**.

Symbol	**Meaning**	**Compute As:**
u	Current value of cell state	
uPast	Prior value of cell state	
uNew	Next value of cell state	
uNabeAvg	Average of Space Neighbors	**(uL + uR)/2** or **(uE+uN+uW+uS)/4** or **(uE+uN+uW+uS + 0.75*(uNE + uNW** + u**SW** + u**SE)) / 7**
uTimeAvg	Average of Time Neighbors	**(uPast + uNew)/2**

Table 2. Notation for cell neighborhood values.

To convert a differential equation into a CA scheme, we write the equation in a form that uses expressions of the form $\mathbf{u_t}$ or $\mathbf{u_{tt}}$ as opposed to expressions of the form $\partial\mathbf{u}/\partial\mathbf{t}$ or $\partial^2\mathbf{u}/\partial\mathbf{t}^2$. And we use the dimension-independent $\nabla^2\mathbf{u}$ to stand, in one dimension, for $\mathbf{u_{xx}}$ or $\partial^2\mathbf{u}/\partial\mathbf{x}^2$, and to stand, in two dimensions, for $\mathbf{u_{xx}} + \mathbf{u_{yy}}$ or $(\partial^2\mathbf{u}/\partial\mathbf{x}^2 + \partial^2\mathbf{u}/\partial\mathbf{x}^2)$. And then we use the substitutions in Table 3. Note that we do not worry about putting in any $\Delta\mathbf{x}$, $\Delta\mathbf{y}$ or $\Delta\mathbf{t}$ terms because any meaningful values for these terms can end up being incorporated into one of the CA scheme's parameters.

$\mathbf{u_t}$	**uNew - u**
$\mathbf{u_{tt}}$	**uTimeAvg - u**
$\nabla^2\mathbf{u}$	**uNabeAvg - u**

Table 3. Some CA approximations.

Consider a diffusion equation with a parameter called **Diffusion**.

$\mathbf{u_t}$ **= Diffusion** $* \nabla^2\mathbf{u}$
becomes
(Diffusion Scheme) **uNew = u + Diffusion * (uNabeAvg - u)**

Note that this scheme satisfies the stability heuristic mentioned at the start of this section; that is, **uNew** is **u** plus something the order of a difference between the **u** values of two neighbors.

The Diffusion Scheme can be thought of taking a weighted average of the cell and its neighbors. That is, we can write the Diffusion Scheme as **uNew = (1 - Diffusion) * u + Diffusion * uNabeAvg**. As the **Diffusion** parameter ranges from zero and unity, the weight shifts from the cell to its neighbors.. If the number of neighbor cells is **k**, then using a **Diffusion** value of **k/(k+1)** makes a straight average of the cell and its neighbors.

One should not use a **Diffusion** value greater than one, as this leads to instability. The reason for this becomes clear if we consider a situation where the **uNabeAvg** is zero. In this case, if **Diffusion** were greater than one, then a single update would change a positive **u** to a negative **uNew**.

Now consider a wave equation with a parameter called **Wave.**
$\mathbf{u_{tt} = Wave * \nabla^2 u}$
becomes
(Wave Scheme) **uNew = (2*u - uPast) + 2*Wave*(uNabeAvg - u).**

Our stability heuristic is satisfied because the first term on the right side is of the order of **u**, while the second term is the order of a difference in the **u** values of two neighbors.

In terms of the differential equations, the **Wave** parameter is actually $(c * \Delta \mathbf{t} / \Delta \mathbf{x})^2$, or the square of the speed of the wave in the medium times the square of the time-step divided by the square of the space step. (See [3].) For a cellular automatist it's more practical to just think of the single parameter **Wave**.

One significant thing to notice about the Wave Schema is that it's time-reversible. As already mentioned in Section 1, we can swap the positions of **uNew** and **uPast** to get a scheme that retrodicts the past instead of predicting the future. In practice, we reverse a running Wave Schema CA by exchanging the roles of the buffers that hold the cells' past values and the cells' future values.

We can also think of the right-hand side of the Wave Scheme as being two times a weighted average of the cell and its neighbors with the old value of the cell being subtracted off. That is, we can write the Wave Scheme as **uNew = 2*((1 - Wave)*u + Wave*uNabeAvg) - uPast**. As the **Wave** parameter ranges from zero to one, the weight shifts from the cell to its neighbors, and values greater than one give instability.

Instead of deriving our schemes for continuous-valued CA rules from differential equations, it's also possible to develop new CA rules simply by playing with the schemes. In [4], Fermi, Pasta and Ulam described an early computer experiment in which they changed the one-dimensional wave equation rule by adding a **Nonlinearity** parameter and a factor that involves the squares of the differences between the neighboring cell values to produce this scheme. (See [5] for a history of the Fermi-Pasta-Ulam work through the mid 1970s.) As mentioned above, we write **uR** and **uL** to stand for the **u** values in the cell's right and left neighbors **R** and **L**.

(One-dimensional Quadratic Wave Scheme)
uNew = (2*u - uPast) + 2* Wave*(uNabeAvg - u +
Nonlinearity*((uR-u)^2 - (u-uL)^2)))

An analysis by Dan Ostrov in [3] establishes that in one dimension, this scheme corresponds to a nonlinear wave equation of the following form.

$$\mathbf{u_{tt} = Wave * u_{xx} + 2 * Nonlinearity * u_x * u_{xx}}$$

We'll say more about nonlinear waves in Section 6.

3. Investigating Continuous-valued CAs

As with other CAs, we simulate parallelism by maintaining separate buffers to hold the current cell values and the new cell values being computed. Since the Wave Schema CAs compute the future on the basis of values both from the present and the past

cell values, we actually need to maintain three buffers for these rules.

In running CAs with continuous-valued state variables, one needs to prevent the state values from taking on unreasonably large values that can produce floating-point overflow. A simple way to do this is to pick a range of values that you will allow the variables to lie in, and to then clamp them to stay in this range, where to "clamp" a variable **u** to a range (**Min**, **Max**) means that if **u** is less than **Min** we set it to **Min**, and if **u** is greater than **Max** we set it to **Max**. The brute-force clamping approach protects the integrity of the simulation in times when the rules have no physical analog.

As an alternate approach to keeping variable values in range, one can "wrap" them instead of clamping them. That is, if **u** is slightly larger than **Max**, one changes it to **u - Max**, and so on. Although this approach has the virtue of preserving more information about the value of **u**, it seems in some cases to have the disadvantage of excessively churning things up, and we have not used it very much. As a matter both of elegance and of physical verisimilitude, one usually tries to design the rules and their parameters so that no special measures are needed to keep the variables in range.

In order to display our CAs, we pick one of the state variables to be the display variable **u**. We then use a map called **Band** to map **u's** range onto the integers up to some largish number **MaxColorIndex** (typically 1000). We have been using simple linear maps for **Band**, although other kinds of maps could be useful, for instance to exaggerate the color changes near some critical value. We use a **Palette** array of **MaxColorIndex** of RGB color values and we display **u** as **Palette[Band(u)].**

Our standard design for **Palette** is to randomly choose some "anchor colors" for some of the **Palette** entries, and then to use linear interpolation in RGB space to ramp the entries whose indices lie between the indices of the anchor colors. This produces a smoothly shaded effect. When generating monochrome images, we simply alternate white and black for the anchor colors, producing a **Palette** which is a shaded series of gray-tone stripes.

We've experimented with a variety of possible boundary conditions for our CAs. The most commonly used is the periodic boundary condition, in which a one-dimensional CA space is treated as a circle and a two-dimensional CA space is treated as a torus. This is the only boundary condition used in the examples discussed in this paper.

There are various possible ways to seed the continuous-valued CAs. Among them are: a constant starting value at all cells, a two-dimensional sine wave pattern, a single tent-like spike, multiple spikes that the user can place with the mouse, and a fully randomized initial state.

Finding the best set of parameter values for a given rule is difficult. The search spaces are, after all, very large and perhaps very chaotically organized. We've used an evolutionary search strategy that seems to have first been introduced by Richard Dawkins in his classic *Blind Watchmaker* program [6]. Like *Blind Watchmaker*, *CAPOW* allows the user to view nine rules at once, to select a visually appealing rule by clicking on it, and to thereby have the other eight rules become mutations of the chosen rule. The mutation rate is user-selectable, and there other kinds of randomization options as well. In other words, many of the rules discussed here have been found by directed search methods. Figure 2 shows an image of the *CAPOW* window with nine different CA rules active.

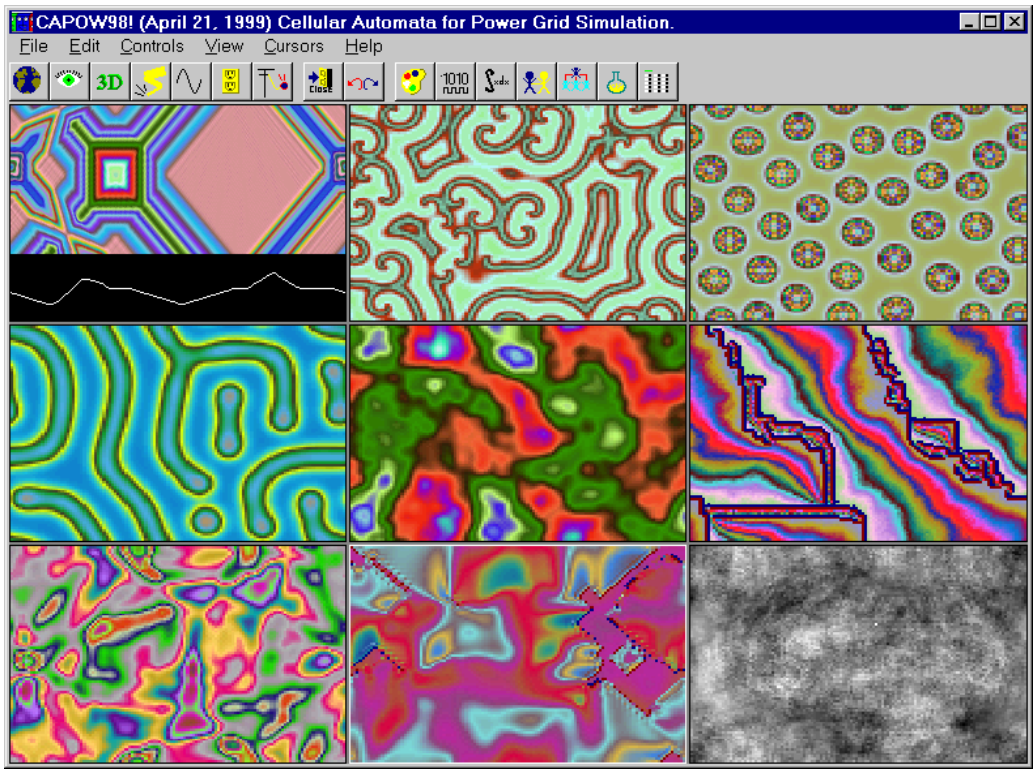

Figure 2. (Color Plate 1) A picture of the CAPOW program showing nine CAs at once. The images shown correspond to the examples used in this paper. The rule in the upper-left corner is one-dimensional and all the others are two-dimensional. Row by row from left to right the rules are a one-dimensional wave, the Zhabo Worm rule, the Turing Leopard rule, the Turing Stripe rule, a linear wave, a quadratic wave, a cubic wave, a homeostatic cubic wave, and the Cloud rule. [CAPOW.tif]

In searching for interesting rules, we use a refinement of Stephen Wolfram's familiar classification of 1-D CAs into four kinds: (I) Those that die, (II) those that repeat, (III) those with non-repeating persistent structures, (IV) pseudorandom. If we view this classification as a spectrum of increasing complexity, it seems logical to have the pseudorandom rules come last, even though Wolfram chose to list the last two classes in the opposite of the expected order. It's useful to distinguish between spatial and temporal periodicity in type II. In two dimensions we have a special kind of complex rule that, rather than exhibiting discrete gliders, shows the self-organizing scroll patterns identified with the Belousov-Zhabotinsky reactions in chemistry. With this in mind, we distinguish between two cases of type III as well.

Complexity Type	**Wolfram Type**	**Attractor**	**Behavior**
I	1	Point	Dies out
IIa	2	Cycle	Fixed space pattern
IIb	2	Cycle	Periodic cycle
IIIa	4	Strange	Self-organizing scrolls
IIIb	4	Strange	Moving gliders or globs

IV	3	Pseudorandom	Chaotic seething

Table 4. The complexity types of two-dimensional CAs.

4. Reaction-Diffusion Systems

Some of the most interesting patterns in nature come from reaction-diffusion systems in which a chemical reaction is taking place while the components of the reaction are being diffused. Much of the research in this area has focused on reactions which involve two substances: an autocatalyzing activator which also produces an inhibitor substance. (See the classic Alan Turing paper [7] and the recent non-technical survey [8].)

In [9], Hans Meinhardt formulates several differential equation schemes for reaction-diffusion systems based on activator-inhibitor reactions. This book also includes a disk with the executable and the BASIC source code for Meinhardt's SP program, which displays continuous-valued one-dimensional CAs based on a wide range of activator-inhibitor-diffusion systems. Meinhardt's work was the major inspiration for our development of the *CAPOW* software.

The work in this section is based on Meinhardt's differential equation scheme for an activator-inhibitor diffusion rule with activator saturation. Depending on how the parameters are set, we can get every possible CA complexity type with the exception of IIIb.

We can easily find rules of type I, which rush to take on the maximum or minimum value for all cells, and remain frozen there.

The rules of type IIa are of particular interest. These rules converge to static-appearing patterns resembling the coats of animals such as leopards and zebras. These kinds of reaction-diffusion patterns are often called *Turing patterns*, as Turing's motivation for considering these rules was indeed to find ways to generate stable patterns which emerge in morphogenesis. The rules of type IIb show a uniform oscillation up and down. If these rules oscillate wildly enough to hit the maximum values and experience "clamping" then recurrent dot patterns are introduced by the clamping process.

The rules of type IIIa are those in which certain wave-like structures form and move about. Among these *travelling wave* patterns, of particular significance are those in which scrolls self-organize. The scroll-forming patterns are instances of the ubiquitous two-dimensional CA rules sometimes called *Zhabotinsky rules*. None of the rules of this kind investigated so far seem to show stable moving patterns that characterize complexity type IIIb.

And finally it is always easy to find setting that produce type IV patterns which seethe wildly.

Meinhardt formulates these rules in terms of two real number variables **a** and **b** which represent the intensity of, respectively, the activator and the inhibitor. In our simulations we've typically let **a** and **b** range from 0 to 4, focusing on rules in which the **a** and **b** values never actually approach the maximum value of 4 closely enough to require clamping. We use a helper variable **bSafe** to prevent division by zero, along with a number of parameters that are named in Table 5.

Symbol	**Meaning**	**Comment**
Variables:		
a	Concentration of the activator.	Typical range 0 to 4.
b	Concentration of the inhibitor.	Typical range 0 to 4.
Safety Variable:		
bSafe	Concentration of the inhibitor, corrected to be above **bMin**.	Typical range 0.001 to 4. Use to avoid division by 0.
Equation Parameters:		
sDensity	Source density, akin to a reaction rate.	Range 0 to 1. Small 0.01 for Turing, medium 0.5 for Zhabo.
aDiffuse	Diffusion rate of the activator.	Range 0 to 1. Needs to be close to **bDiffuse** for Zhabo.
bDiffuse	Diffusion rate of the inhibitor.	Range 0 to 1. Needs to be much larger than **aDiffuse** for Turing.
aBase	Basic activator production rate.	Small 0.01 for Turing, medium 0.3 for Zhabo.
bBase	Basic inhibitor production rate.	Small, about 0.004.
aDecay	Activator removal rate.	Large 0.5 for Zhabo, small 0.01 for Turing.
bDecay	Inhibitor removal rate.	Large 0.3 for Zhabo, small 0.01 for Turing.
aSaturation	Slows down the rate of activator production as **a** increases.	Need this to get good Turing patterns. Low values like 0.04 give dots, high values like 0.2 give stripes.
Simulation Parameters:		
Neighborhood	Dimensionality and neighborhood.	In 2D we get the best-looking, smoothest results with a Moore neighborhood with edges weighted slightly more than corners.
bMin	Minimum inhibitor in rule.	Small 0.001 for Turing, medium 0.5 for Zhabo.
abMax	Maximum value of activator or inhibitor in rule.	4 works well.

Table 5: The variables and parameters for the activator-inhibitor-diffusion rule.

Meinhardt's activator-inhibitor equations have this form.

(Activator) $\mathbf{a_t = aDiffuse * \nabla^2 a +}$
+ sDensity * a^2/(b * (1 + aSaturation*a*a))
+ sDensity * aBase
- aDecay * a.
(Inhibitor) $\mathbf{b_t = bDiffuse * \nabla^2 b}$
+ sDensity * a^2
+ bBase
- bDecay * b.

To model these as CA schemes, we treat the time-step as 1 and use updates of the form $\mathbf{aNew = a + a_t}$. The full activator-inhibitor CA rule takes the following form.

(Avoid division by 0) **IF (b > bMin) THEN bSafe = b ELSE bSafe = bMin.**
(Activator) **aNew = a + aDiffuse * (aNabeAvg - a)**
+ sDensity*a*a/(bSafe * (1 + aSaturation*a*a))
+ sDensity * aBase
- aDecay * a.
(Inhibitor) **bNew = b + bDiffuse * (bNabeAvg - b)**
+ sDensity*a*a
- bDecay * b.
(Clamp) **Clamp both a and b to be in the range [0, abMax].**

Figure 3 shows an example of one of the self-organizing scroll CAs of type IIIa called Zhabo Worms, while Figures 4 and 5 show examples of stable Turing pattern CAs of type IIa called Turing Leopard and Turing Stripes. The parameter values used for these three rules appear in Table 6.

Figure 3: Zhabo Worms. This is an activator-inhibitor diffusion rule. It slowly self-organizes scrolls from a random start.[Zhabo Worms B&W.tif]

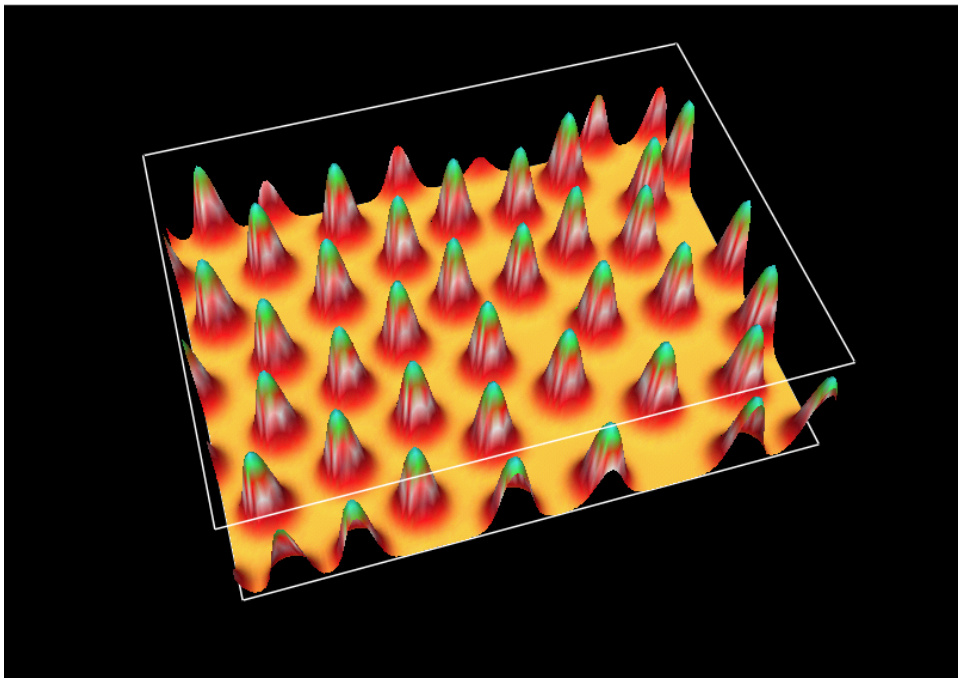

Figure 4. (Color Plate 2.) A three-dimensional view of the two-dimensional Turing Leopard rule. The heights and colors both represent the activator value. Starting from a random start, this rule's values drift down towards zero and then some stable peaks of activation develop and grow to a medium height. Convergence is rapid. [Turing Leopard 3D.tif]

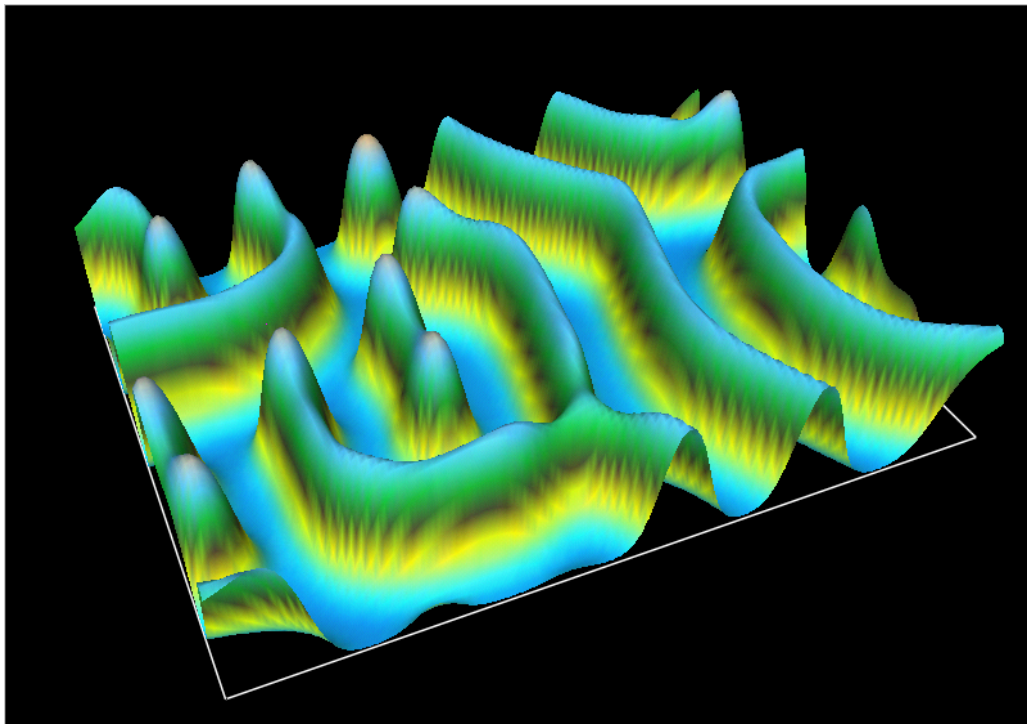

Figure 5. (Color Plate 3.) A three-dimensional view of the two-dimensional Turing Stripes rule. The heights and colors both represent the activator value, and the inhibitor values are closely similar. This pattern self-organized from a random start. It is fully stable, and has been run for over 100,000 updates. [Turing Stripes 3D.tif]

	Zhabo Worms	**Turing Leopard**	**Turing Stripes**
Neighbors	2D Moore	2D Moore	2D Moore
Complexity	IIIa	IIa	IIa
sDensity	0.52	0.011	0.015
aDiffuse	0.0975	0.0399	0.049
bDiffuse	0.04375	0.99995	0.99995
aBase	0.256	0.01	0.01
bBase	0.004	0.0055	0.0055
aDecay	0.52	0.015	0.01
bDecay	0.3	0.01	0.015
aSaturation	0.0	0.04	0.2
bMin	0.52	0.001	0.001
abMax	4.0	4.0	4.0

Table 6. The parameters for the three activator-inhibitor-diffusion rules of Figures 2, 3, and 4.

5. Wave Equations

The two-dimensional wave equation rules give patterns very similar to the surface

of pan of water, although our use of periodic boundary conditions means that the waves don't reflect off the edges. Figure 6 shows an image of a randomly perturbed two-dimensional wave schema rule.

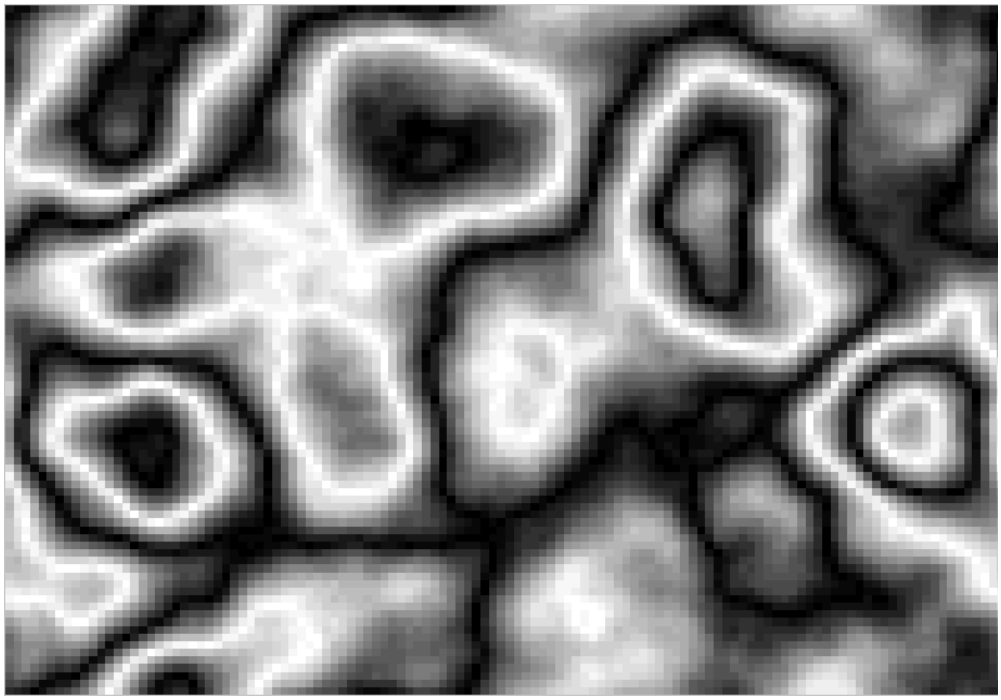

Figure 6. A two-dimensional wave schema being run with a **Wave** constant of 0.694 and a **uMax** of 3.0. The pattern started as a two-dimensional sine wave and was repeatedly perturbed with random conical "dings". It will continue sloshing around like this indefinitely. [Wave B&W.tif]

One might be tempted to say that the wave-based rules have complexity type IIIb, in that the individual wave patterns behave somewhat like gliders that move around. On the other hand, since the wave equation is linear, the wave crests cross each other without interacting. And the interaction of gliders is really the essence of what we think of as complexity type IIIb, for one expects a complexity type IIIb rule to appear as if it may be capable of simulating a universal computer.

In order to have wave-like rules in which the individual wave-patterns interact, we need a non-linear wave equation along the lines mentioned in Section 2. We have worked with three nonlinear two-dimensional wave schemes. Two are fairly straightforward: a quadratic and a cubic nonlinear wave. The third is more complicated, it's a cubic nonlinear wave that's has a "homeostatic" tweak designed to prevent it from becoming unstable.

The first quadratic and cubic wave rules are based on a von Neumann neighborhood. The homeostatic cubic wave rule uses the von Neumann neighborhood for the "wave mode" of its updates and uses the Moore neighborhood when it enter an "averaging mode" to smooth out instabilities. Recalling that we use **uE**, **uN**, **uW**, and **uS** to stand for the intensity values of the update cell's four von Neumann neighbors, we can write the quadratic and cubic wave schemes as follows

(Two-Dimensional Quadratic Wave Scheme) **uNew =**
(2*u - uPast) + 2 * Wave * (uNabeAvg - u +
Nonlinearity*($(uE-u)^2$ - $(u-uW)^2$ + $(uN-u)^2$ - $(u-uS)^2$))
(Two-Dimensional Cubic Wave Scheme) **uNew =**
(2*u - uPast) + 2 * Wave * (uNabeAvg - u +
Nonlinearity*(($(uE-u)^3$ - $(u-uW)^3$ + $(uN-u)^3$ - $(u-uS)^3$))
(Clamp) **Clamp u to be in the range [-uMax, uMax].**

The third nonlinear wave is a "homeostatic cubic wave" scheme which we'll discuss below. In Figure 7 we show our four kinds of waves side by side.

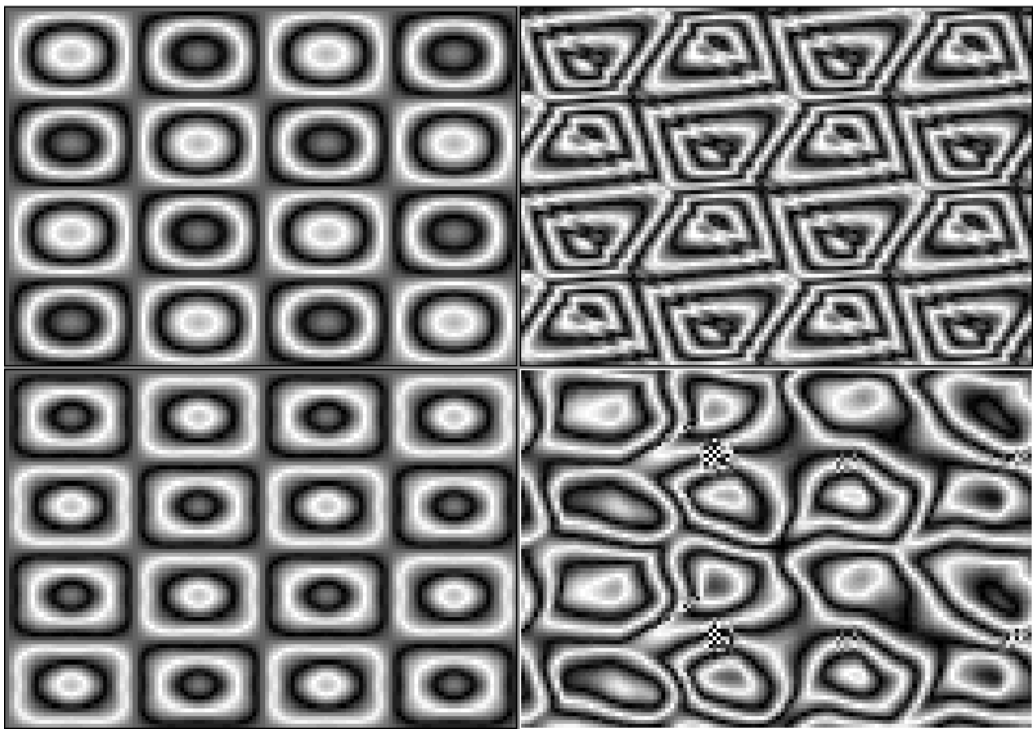

Figure 7. A view of four kinds of two-dimensional waves. From the left, the top row has a linear wave and a quadratic wave, and the bottom row has a cubic wave and a homeostatic cubic wave. Each rule was seeded with a four full cycles of a sine-wave pattern and was run for about 500 updates. In the linear wave this pattern simple oscillates forever, making "sushi" patterns that are displayed by showing the intensities by different shades of black and white. In the quadratic wave, the peaks become asymmetric, and in the cubic wave the peaks become more angular. The flaws on the cubic homeostasis wave are locations where the wave has become unstable and has intensity values that are being clamped to the maximum or minimum allowable value. All the rules are being run with a **Wave** constant of 0.694 and a **uMax** of 3.0. The **Nonlinearity** values of the quadratic and cubic, waves are, respectively 0.5, 3. The **Nonlinearity** in the homeostatic cubic wave varies from cell to cell, ranging from 0.001 to 1000. [Four Waves B&W.tif]

The nonlinear wave schemes easily go unstable, especially the cubic one. In these waves, instability will mean that the intensity values grow without bound. Thanks to the clamping step, the values then get stuck at maximum or a minimum value. In the case of the two-dimensional quadratic wave, instability can lead to a certain kind of interesting

structures. But in the cubic case, an instability is typically a checkerboard of alternating maximum and minimum-valued cells which grows to fill the simulation space.

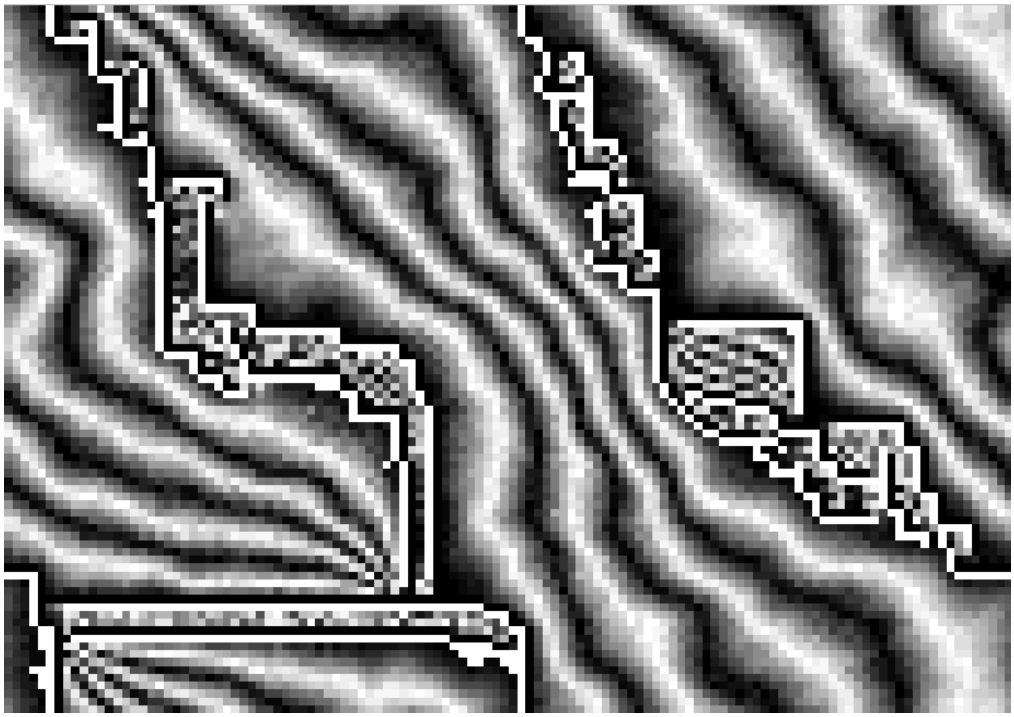

Figure 8. A quadratic wave scheme with **Wave** of 0.25, **Nonlinearity** of 0.15, and **uMax** of 3.0. The pattern was seeded with all u values of 1.5 with a conical bump in one location. The cone tip produced an instability which propagated along a closed "fault line". (Recall that this is a toroidal space.) The pattern is now stable and will remain like this indefinitely. Note that small structures are able to move along within the "wave-guide" pieces of the fault.[Quadratic Walls B&W.tif]

Our "homeostatic cubic" rule has an ad hoc technique for taming the cubic instabilities. The idea is to run an unstable cubic wave, and to let the nonlinearity of the wave be determined locally. This gives an interesting effect showing Zhabotinsky patterns moving about in a wave medium.

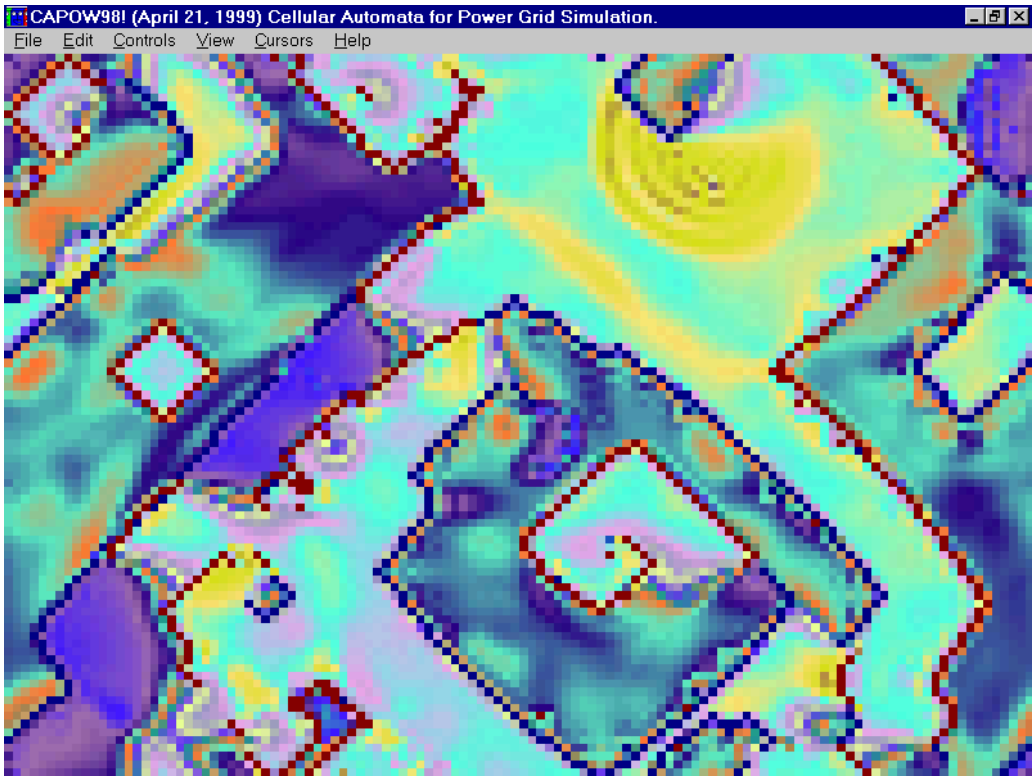

Figure 9. (Color Plate 4.) Homeostatic Cubic Zhabo, based on the homeostatic cubic wave scheme. Notice the circular wave patterns as well as the diamond-shaped Zhabotinsky spiral. See Figure 8 for the parameters used in this rule. [Homeostatic Cubic Zhabo.tif]

For the homeostatic cubic wave, each cell holds an intensity **u** and a nonlinearity multiplier called **LocalNonlinearity**. In addition the rules use a helper variable TooBig and some additional parameters as shown in the table.

Symbol	**Comments**	**Homeostatic Cubic Zhabo**
Variables:		
u	Intensity. Ranges from 0 to **uMax**	
LocalNonlinearity	Each cell has its own value for this; it starts at **MinNonlinearity** and drifts up towards **MaxNonlinearity.**	
TooBig	Boolean helper variable to signal when the rule has become unstable at a given cell.	
Parameters:		
Wave	Normally ranges from 0 to 1, although could be larger as we expect this rule to become unstable anyway.	0.5
MaxNonlinearity	The **LocalNonlinearity** values in	100.0

	the individual cells move towards this. Can be any positive value.	
GrowFactor	The multiplier which moves the cells' **LocalNonlinearity** value towards **MaxNonlinearity**. Should be slightly larger than 1.	1.05
MinNonlinearity	This needs to be larger than 0 because we increase the **LocalNonlinearity** by repeated multiplications by **GrowFactor**.	0.01
WaveThreshold	This should be bigger than **MinNonlinearity**. When the **LocalNonlinearity** lies between **MinNonlinearity** and **WaveThreshold** we update **u** with an averaging rule, otherwise we use a cubic wave rule.	0.015
uMax	We clamp **u** to be in the symmetric range (**-uMax**, **uMax**).	1.0

Table 7. The variables and parameters for the homeostatic cubic wave scheme, along with the values used for the Homeostatic Cubic Zhabo in Figure 8.

The update process for this rule has two parts: the computation of the **uNew** value and the computation of the **LocalNonlinearityNew** value.

At the first stage of the update, we compute the **uNew** value in one of two possible ways, depending on the size of the **LocalNonlinearity** variable. If **LocalNonlinearity** is small, that's an indication that we're in a zone that was recently unstable, and we update **uNew** by a simple averaging scheme. If **LocalNonlinearity** is larger, we update **uNew** by a cubic wave scheme, and we use the **LocalNonlinearity** as the cubic wave's **Nonlinearity** parameter. Finally, after updating **uNew**, we clamp uNew to lie in the range (-uMax, uMax). If uNew was indeed out of range, we set my **TooBig** helper variable to **true**, otherwise we set **TooBig** to false.

At the second stage of the update, we compute the **LocalNonlinearityNew** value in one of two possible ways, depending on whether **TooBig** is **true** or **false**. In the **TooBig** case, we let **LocalNonlinearityNew** collapse to **MinNonlinearity**. This has the dual effect of damping the unstable cubic rule and of signaling the cell to use an averaging rule for its next update. When **TooBig** is false, we multiply **LocalNonlinearityNew** by **GrowFactor** and clamp **LocalNonlinearityNew** to lie in the range (**MinNonlinearity**, **MaxNonlinearity**).

The uNew update.

(Cubic Wave Option) **IF (LocalNonlinearity >= WaveThreshold) THEN**

uNew = 2*u - uPast + Wave*(uNabeAvg - u

+ LocalNonlinearity* ($(uE-u)^3$ - $(u-uW)^3$ + $(uN-u)^3$ + $(u-uS)^3$));

(Averaging Option) **IF (LocalNonlinearity < WaveThreshold) THEN**

uNew = uNabeAvg;
(Clamp and set **TooBig**) Clamp **uNew** to lie in the range (**-uMax**, **uMax**). If **uNew** was outside the range set **TooBig** to **true**, otherwise set **TooBig** to **false**.

The LocalNonlinearityNew update.
(Collapse option) IF (TooBig) THEN LocalNonlinearityNew = MinNonlinearity
(Growth option)IF (NOT TooBig) THEN
LocalNonlinearityNew = **GrowFactor* LocalNonlinearity**
(Clamp) Clamp **LocalNonlinearityNew** to be in the range
(**MinNonlinearity**, **MaxNonlinearity**);

This rule readily falls into a Zhabotinsky-style pattern of complexity type IIIa. The Zhabotinsky spirals are driven by the behavior of the **LocalNonlinearity** parameter, which grows to a maximum value and then drops abruptly.

6. Reaction Wave Systems.

We've also done some preliminary work in trying to put a wave term in place of the diffusion term in one or both of the two equations in our activator-inhibitor systems. We've tried various ways of doing this, but none has been an outstanding success in terms of producing really interesting behaviors. A typical scheme we've tried has the following form. In this scheme we don't use an **aSaturation** term.

(Avoid division by 0) **IF (-bMin < b < bMin) THEN b = Sign(b)*bMin**
ELSE bSafe = b.
(Activator) **aNew = 2*a - aPast + Wave * (aNabeAvg - a)**
+ sDensity*a*a/bSafe
+ sDensity * aBase
- aDecay * a.
(Inhibitor) **bNew = 2*b - bPast + Wave * (bNabeAvg - b)**
+ sDensity*a*a
- bDecay * b.
(Clamp) **Clamp both a and b to be in the range [-abMax, abMax].**

This rule produces patterns resembling clouds.

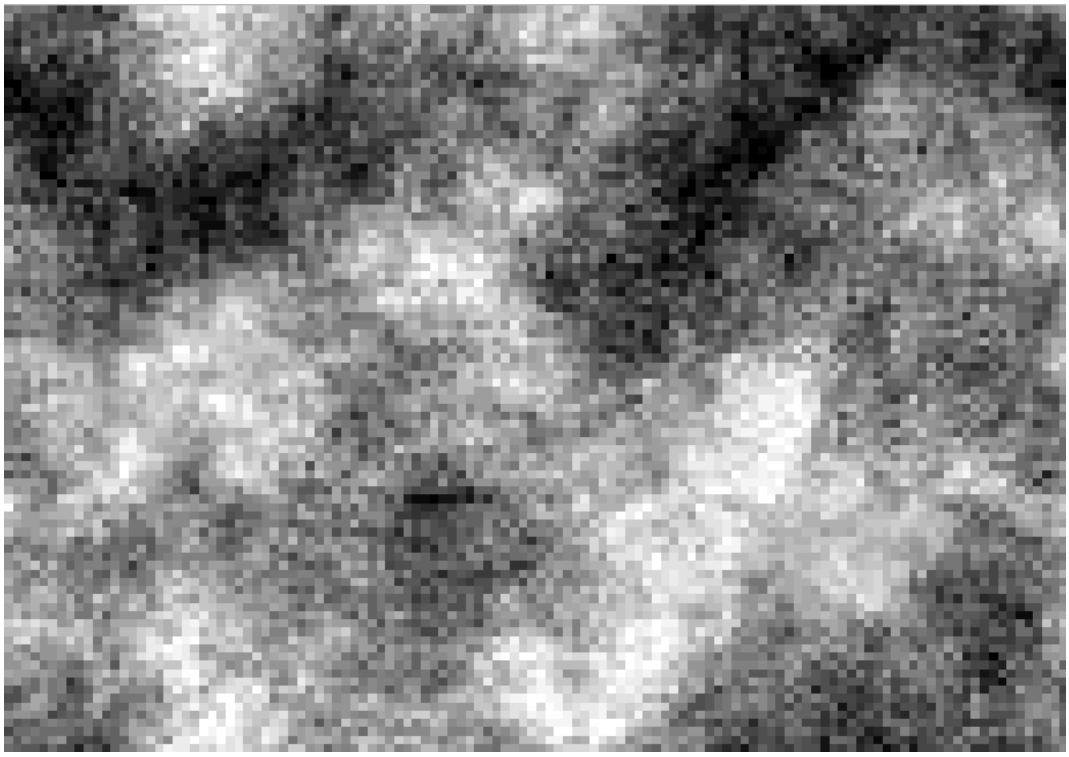

Figure 10: A cloud-like pattern formed by an activator inhibitor rule with wave terms in place of the customary diffusion terms. This rule converges quickly to this behavior from a random start. The rule is shown using only one band of color, that is, black in the minimum intensity and white is the maximum. [Reaction Wave Cloud B&W.tif]

Name	**Clouds**
Neighbors	2D Von Neumann
Complexity	IV
Wave	0.25
sDensity	0.01
aBase	0.01
bBase	0.0055
aDecay	0.01
bDecay	0.015
bMin	0.001
abMax	32.0

Table 8. The parameter values used for the Cloud rule.

7. Suggestions for Further Work

In [9] Meinhardt investigates other, more complicated, kinds of one-dimensional activator-inhibitor systems, and it would interesting to cast more of these into two-dimensional form. It would be of particular interest to see Turing patterns which move about; that is, it would be nice to see crawling dots and writhing stripes. One result of

this might be that our CAs could begin to model the motions of extended objects.

Another, related, goal is to find some continuous-valued CAs with more purely type IIIb behaviors. That is, one would like to see glider-like patterns moving about and interacting.

It might also be useful to base some rules on third-order differential equations; presumably using something like a **uSecondNabeAvg - 2* uNabeAvg + u** term, where the **uSecondNabeAvg** would be computed from neighbors two cells away. Perhaps some of these rules could exhibit solitons that might play the role of information-bearing gliders.

Finally, there is still the open frontier of three-dimensional CAs. Certainly it would be nice generalize the simple activator inhibitor schemes to three-dimensions so as to produce three-dimensional Turing patterns.

References:

[1] Tommaso Toffoli and Norman Margolus, *Cellular Automata Machines* (MIT Press, 1987).

[2] Rudy Rucker et. al., *CAPOW!* software available for free download at http://www.rudyrucker.com/capow, 1999, also updated in 2007.

[3] Daniel Ostrov and Rudy Rucker, "Continuous-Valued Cellular Automata For Nonlinear Wave Equations," *Complex Systems,* **10** (1996) 91-119.

[4] E. Fermi, J. Pasta, and S. Ulam, "Studies of Nonlinear Problems," originally in Los Alamos Report LA-1940, 1955, later in S. Ulam, *Sets, Numbers and Universes*, MIT Press: Cambridge, 1974, pp. 491-501

[5] Thomas Weissert, *The Genesis of Simulation in Dynamics: Pursuing the Fermi-Pasta-Ulam Problem* (Springer-Verlag 1997).

[6] Richard Dawkins, "The Evolution of Evolvability," in C. Langton, *Artificial Life* (Addison-Wesley 1989), p. 201.

[7] Alan Turing, "The Chemical Basis of Morphogenesis," *Philosophical Transactions of the Royal Society* B 237, p. 37.

[8] Philip Ball, *The Self-Made Tapestry: Pattern Formation in Nature*, (Oxford University Press 1999).

[9] Hans Meinhardt, *The Algorithmic Beauty of Shells* (Springer-Verlag, 1995).