Information Theoretic and Complexity Considerations in Four Classes of Cellular Automata

A. Alavizadeh, J. Curry, C. Kelleher, C. Lewiston, A. Resnick, E. Solis, Y. Bar-Yam*

NECSI One-Week Intensive Course in Complex Systems, 9-13 January 2006

Contents

*President, New England Complex Systems Institute

1 Introduction

Originally considered by John von Neumann, Stanislaw Ulam, and others in the mid-20th c., cellular automata blossomed in the early 1980s when Wolfram *et al* brought to the attention of the scientific community the capacity that cellular automata have to generate a broad range of interesting, complex behavior based upon a small set of simple rules. Presently, we will leave the definition of a complex system to the reader's intuition, to return to the issue shortly.

The potential of the field of cellular automata has been of considerable interest of late due to their surprisingly powerful capacity to model complex systems. From snowflakes to ethnic violence, the world is full of systems exhibiting complex behavior. These systems have, in general, refused to yield to traditional approaches. In most cases, they are far too complex, involving many more components (both distinguishable and indistinguishable) interacting at much more complex levels than those to which historical, first-principle approaches are suited. A number of very surprising applications and characteristics of cellular automata have been discovered, ranging from the modeling of partial differential equations and Markov random field to Turing machines and finite difference equations.

Given the richness of this tool, it is a natural impulse to investigate the basic structure of the configuration space of CA so that we might better understand this tool and its applications. The most ocherent attempt to accomplish this thus far has been made by Wolfram with his four classification schemes [3]; although, the adequacy of the resolution of Wolfram's classification scheme has recently come under question [2].

In the context of this desire to generalize the field, the question remains what more general context or discipline is sufficiently broad and powerful to impose an interpretable structure on the configuration space. Historically, the answer has been posited as information theory, a response which we insist is incomplete in its traditional forms. Namely, we feel that the burgeoning field of complexity and complex systems has much to offer.

Although the field of complexity is fundamentally linked to information theory, there are implications and tools associated with a complex systems approach which allow for some dramatic conclusions to be put into a very natural context.

2 Foundations

While the fields of cellular automata and information theory are five decades old, it behooves us to review the most pertinent details and place it in the specific context of our investigation.

2.1 Cellular Automata

The use of cellular automata (CA) is one among many, various tools used in the study of complex systems. Cellular automata are a useful mathematical tool for investigating the

behavior of connected, simple, indistinguishable elements whose interactions over time form a complex system. In many cases, deceptively simple rules for governing the development of a particular CA model give rise to unexpectedly complex, powerful behavior.

In its most general articulation, a cellular automaton comprises an n-dimensional matrix of cells which can take on either discrete or continuous values. These values are updated in a sequence of discrete time steps per a well-defined rule or rules interrelating any number of cells separated by any distance.

Empirically defining the class of cellular automata with which we will concern ourselves, we can note several, definite characteristics of such pertinent cellular automata and their associated behavior [3],

- **Spatial discreteness** - The CA operates on a discrete, spatial grid
- **Temporal discreteness** - The CA operates over a discrete time steps
- **Discrete state space** - The CA comprises cells that can adopt only a discrete number of values
- **Cellular and structural homogeneity** - The cells constituting the matrix over which the CA acts are arranged regularly, and are each indistinguishable from any other cell
- **Synchronous update** - The CA acts on all cells simultaneously, discretely
- **Deterministic update** - The CA updates each cell deterministically
- **Spatial localization of structure** - The CA update rule depends only on other cell states
- **Temporal localization of structure** - The CA update rule relies on a finite number of past states

In light of this success, it becomes interesting to ask what distinguishes various CA models. Wolfram has proposed the division of CA models into four, distinct classes, based upon the nature of their long-term behavior [1]:

I. CA that evolve to fixed homogeneous state: This class evolves to a homogeneous state after finite number of time steps. Its evolution destroys any information on the initial state.

II. CA that evolve into fixed inhomogeneous states or cycles.

III. CA that evolve into chaotic or aperiodic behavior

IV. CA that evolve to complex localized structures

Wolfram proposes that these four classes are sufficient to provide structure to the configuration space for cellular automata as a result of the existence of attractors which contain a relatively small fraction of possible states, suggesting that only a few aspects of a particular CA are relevant to its long-range behavior.

In the case of Class I, the final state is completely determined. Class II allows for meaningful statements about particular sites' end behavior based upon initial information about a limited region of sites. Class III leads to chaotic behavior which can nonetheless be transparently calculated algorithmically. However, in considering Class IV, we find that there may be no alternative to running the simulation, which is to say no condensation of the information contained in the CA can be achieved through a coherent description of structure. It has been suggested that these CA are capable of universal computation, which if confirmed, would mean that their development could depend on any finite algorithm.

With these universality classes in mind, we are seeking to investigate the development of information theoretic measures of complexity over time within and between these classes of rules. Considering the information about the behavior of the CA which we can derive from these simple, qualitative descriptions immediately suggests this information theoretic approach.

To accomplish this, we rely on the traditional information theoretic idea of entropy in the variation of intensity throughout the space of the image, which is to say the spatial arrangement of the cellular automata at a specific time.

2.2 Information Theory

One important aspect of cellular automata is the concept of information. The foundations of information (also known as communications theory) were laid by Ronald Fisher in Great Britain and by Claude Shannon and Norbert Wiener in the United States.

The most relevant result from this field is Shannon's source coding theorem, which equates information content and entropy. This connection seems to be initially disparate; however, when one considers that the less order exists in a sample or system, the more information that is required to completely specify the state of the system, this definition begins to make sense. The natural language for speaking about information is the *bit*, or binary digit. Although the mathematics of information theory can frequently be simplified by using *nats*, or natural digits, we will use the bit since we are not extending the mathematical formalism or conclusions of classical information theory, but instead interfacing it with technology from which bits are particularly easy to obtain, namely computers.

Shannon's source coding theorem states that N independent identically-distributed random variables each with entropy $H(X)$ as given by,

$$H(X) = \sum_{i=1}^{n} p_i * \log_2 \tag{1}$$

can be compressed into more than $NH(X)$ bits with negligible risk of information loss, as N tends to infinity; conversely, if they are compressed into fewer than $NH(X)$ bits it is virtually certain than information will be lost.

More rigorously, we can restate Shannon's coding theorem by letting X be an ensemble with entropy $H(X) = H$ bits. Given $\epsilon >)$ and $0 < \delta < 1$, there exists a positive integer N_0 such that $\forall N > N_0$,

$$|\frac{1}{N} H_\delta(X^N) - H| < \epsilon \tag{2}$$

where $H_\delta(X) = \log_2 |S_\delta(X)|$ and $S_\delta(X)$ is the smallest subset of values of X such that the probability that x is not in S_δ is less than δ.

With this in mind, it becomes clear that $H(X)$ is the minimum information content necessary to specify a variable. Equating this to the idea of complexity is a short step to realizing that the factor which distinguishes a complex element from a simple is the difficulty encountered in handling and describing that element.

This is closely related to the asymptotic equipartition property and the notion of a typical set.

The asymptotic equipartition property (AEP) concerns the output of stochastic sources and is a result of the weak law of large numbers and ergodic theory. Simply put, if a range of values $X_1, X_2, X_3, \ldots, X_n$ is drawn (each independent of the other) from a random variable X per some probability distribution $P(x)$, then the probability of drawing the set tends toward

$$P(X_1, X_2, X_3, \ldots, X_n) = \lim_{n\to\infty} -\frac{1}{n} \log_2 P(X_1, X_2, X_3, \ldots, X_n) = H(X) \tag{3}$$

This explains the definition of the typical set, which is defined as a set for which the $P(X_1, X_2, X_3, \ldots, X_n)$ for which

$$2^{-n(H(X)-\epsilon)} \geq P(X_1, X_2, X_3, \ldots, X_n) \leq 2^{-n(H(X)-\epsilon)} \tag{4}$$

which is to say, that the set follows the prediction of the AEP.

The concepts of entropy and its relationship to complexity are the primary tools necessary to full appreciate this project's aims and results. With this in mind, let us take a closer look at complexity in context of this project and its connection to information theory.

2.3 Complexity

Complexity is defined as the amount of information needed to describe a system [1]. Historically an ill-defined term, for our purposes we will restrict our meaning to this well-posed idea of requisite information content.

The complexity profile is a powerful tool introduced by Bar-Yam [1] to understand the full nature of the complexity of the system (specifically, as this complexity relates to

scale). The complexity profile reveals the nature of the relationship between the observed complexity of a system and the scale at which it is observed.

As covered earlier, complexity will be considered equivalent as the information required to define, determine, and describe the system. In relating this to the scale of observation, one can develop a consistent language which generalizes our discussion of complex systems. Generally, complex systems involve a very large number of component which are indistinct at some useful scale (even if individually, they are quite distinct at the smallest scale). The fine scale parameters and behaviors are rarely, if ever, our concern. Instead, we are most often concerned with macroscopic qualities, trends, and behaviors—otherwise, it would be contrary to our interests to articulate our situation in a systems vocabulary.

This immediately suggests that our fundamental analysis should **not** be scale -restricted. The scale at which the system is observed is important. The information necessary to describe the system from a macroscopic perspective is different from the microscopic one. Typically, this is not the case where observation is concerned—microscopic observers can make many distinctions and collect many data that the macroscopic observer cannot. In concert with the counterintuitive results of the information theoretic/entropic view of complexity (namely the association of disorder with complexity and information content, this discussion makes clear the necessity of assessing complexity in the context of scale.

An issue fundamental to the study of complex systems is the question of understanding the relationship between complexity and scale. This is one of the primary considerations in the formation of our question as to the interaction between the specific system of CA and the general informative nature of the complexity and scaling patterns present in the system.

3 Methods

Modeling general CA systems necessitated clear articulation of the parameters over which the general behavior interested us. To address this need, we chose Wolfram's four class taxonomy of CA as a context for the project. The part of the modeling and simulation to which this is most relevant is the choice of data sets. The descriptions of Wolfram's classes are primarily qualitative, since specific rules and systems can move across classes, through the configuration space, due to their sensitivity to intiial conditions. Thus, a single element in the configuration space (and in our data sets) comprises not only a CA rule, but the triplet of the rule, the initial conditions, and the long term behavior of the system.

In confronting this question, it becomes necessary to develop primarily two classes of tools. The first of these would ideally allow us to simulate the systems in which we are interested with minimal knowledge about the underlying nature of the system's behavior. The second of these tools would handle the analysis by which we hope to extract the pertinent information from the output of the simulations, allowing us to (potentially) draw significant conclusions about the modeled systems.

3.1 Modeling and Simulation

In the context of our question, the primary modeling tool use was the well-defined tool of cellular automata implemented in Java. For our purposess, a cellular automaton was a matrix holding binary values upon which we could define discretely acting rules whose iterative application is responsible for the intermediate behavior and final state of the CA. In particular, a two-dimensional Moore cellular automata was used to investigate the implement entropy measure to calculate the complexity of the four classes of CA.

We implemented this system in Java in such a way that each iteration produced an image in PNG format for analysis. In creating the data sets for analysis, it was necessary to construct sets which were believed to be in separate regions of the configuration space, which is to say, in different Wolfram classes. This unavoidably involved the subjective assessment of the long term behavior of the various rules implemented in Java to explore the configuration space for appropriate rules. Eventually, we discovered four rules such that we could reliably reproduce behavior which could be (subjectively) clearly and consistently placed into one of Wolfram's four classes.

Class I. If less than five (5) neighboring cells are on, turn current cell off, otherwise turn it on. (80 $\times$ 80 grid, random seeding with sixty percent (60%) bias, one-hundred (100) time steps)

Class II. If less than 4 neighboring cells are on, turn current cell off, otherwise turn it on. (80 $\times$ 80 grid, random seeding with sixty percent (30%) bias, one-hundred (100) time steps)

Class III. **For a space that is populated**

- Each cell with one or no neighbors dies, as if by loneliness.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.

For a space that is unpopulated, each cell with three neighbors becomes populated. [1] (40 $\times$ 40 grid, random seeding with sixty percent (30%) bias)

Class IV. If number of neighboring cells that are on is even, turn the current cell on, if odd, turn current cell off (47 $\times$ 47 grid, random seeding with sixty percent (50%) bias)

With this data in hand, we turned to the issue of analysis so that we could address some of the more subtle details of our question.

[1] Standard rules for John Conway's *Game of Life*

3.2 Analysis

The extensive, visual data collected required the implementation of an analysis algorithm capable of quantifying the information content of an image. Per Shannon's definition, we calculated the entropy of intensity images using the `entropy` function of the MATLAB image processing toolbox. We later found that the `entropyfilt` function provided another valuable tool, which we will discuss shortly.

Most frequently used for the characterization of texture in a sample image (due to its association with the randomness in the image), entropy is defined by MATLAB per Equation 1. The probability element p is constructed from the histogram counts associated with the intensity distribution of the image. By default, `entropy` uses two bins for logical arrays and `256` bins other arrays.

Thus, for a specific time series of images, the entropy function is used to create a plot of complexity (as measured by information content, or entropy $H(t)$) versus the progress of the CA iterations.

entropyfilt1 returns an array, each element of which corresponds to the local entropy around a corresponding input pixel in the original image selected by the user-defined neighborhood matrix.

The entropyfilt1 function provides us with an even more powerful tool. The ability to define a neighborhood over which the entropy of the histogram of the image's intensity is calculated makes it very easy to speak about the relationship between complexity (entropy) and scale in the image, allowing for a rich expression of the complexity of the CA in an extension of Bar-Yam's complexity profile [1].

This extension comprises a simple visualization of the progression of the image's information content, its complexity, in terms of scale. In concert with the time series images produced by the CA and output by the Java script, the `entropyfilt` function makes it possible to create a complexity contour, of sorts, which communicates the nature of the progression of the CA complexity.

Thus, a three-dimensional phase space can be easily constructed and visualized wherein a single point represents a triplet of information: the progression of the CA through time, the scale at which the CA is being investigated, and the measure of complexity that is determined by these two parameters.

4 Results

4.1 Class I Cellular Automata

Given that the long-term behavior of Class I comprises a fixed, homogeneous state, one would immediately anticipate that the complexity of the CA would go to zero when its (constant) long range behavior is reached. This is, in fact, the observed behavior in the

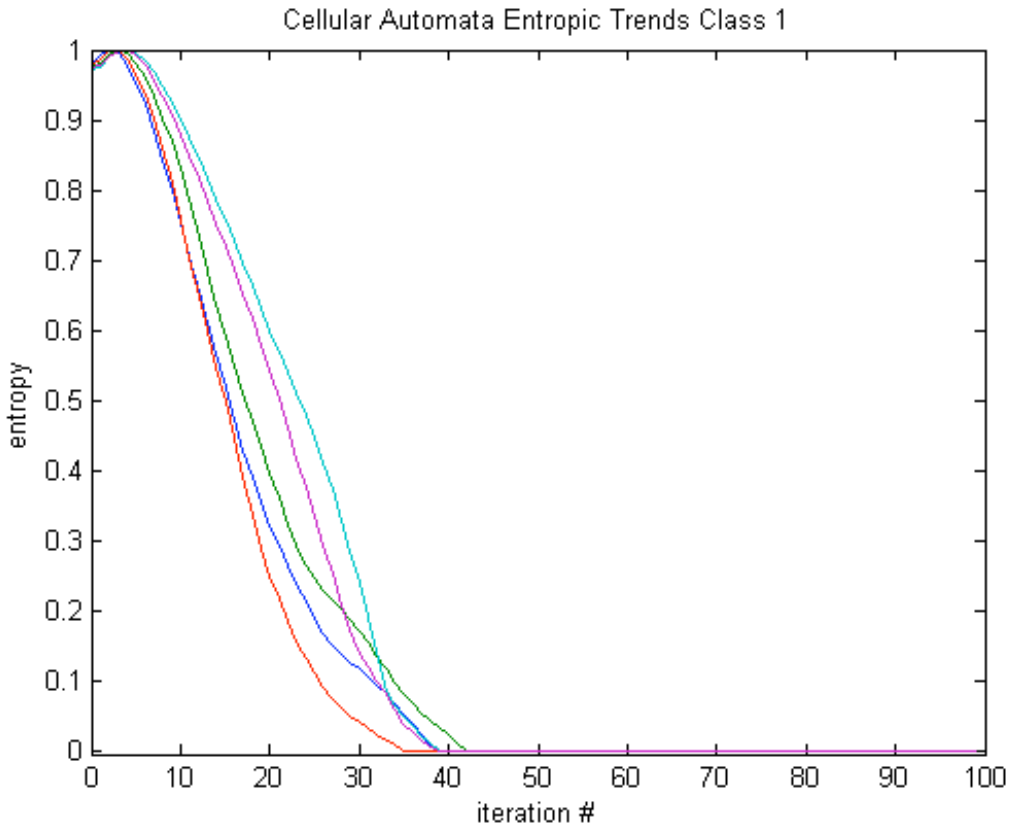

Figure 1: Class I Entropic Time Series

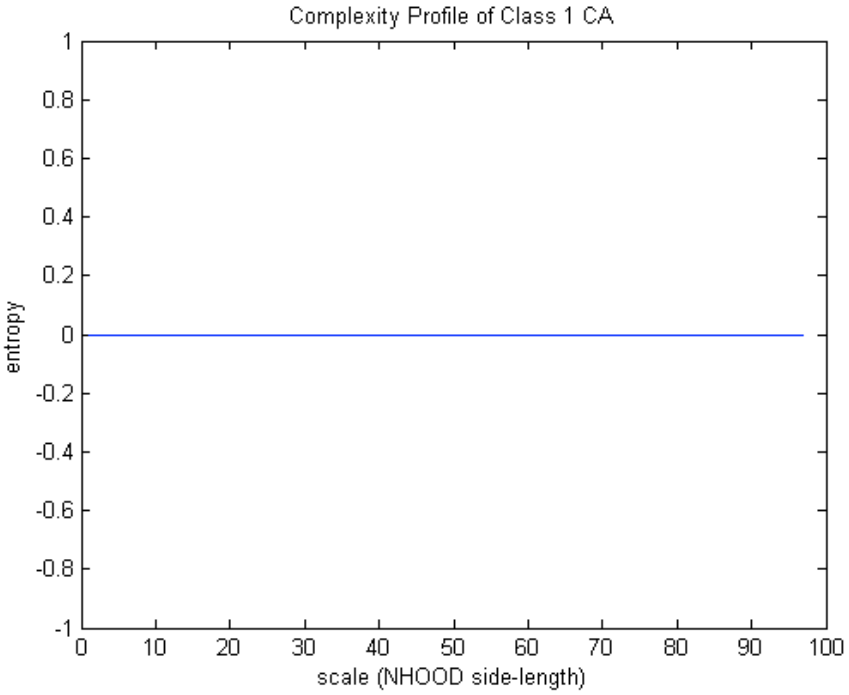

Figure 2: Typical Class I Long Range Complexity Profile

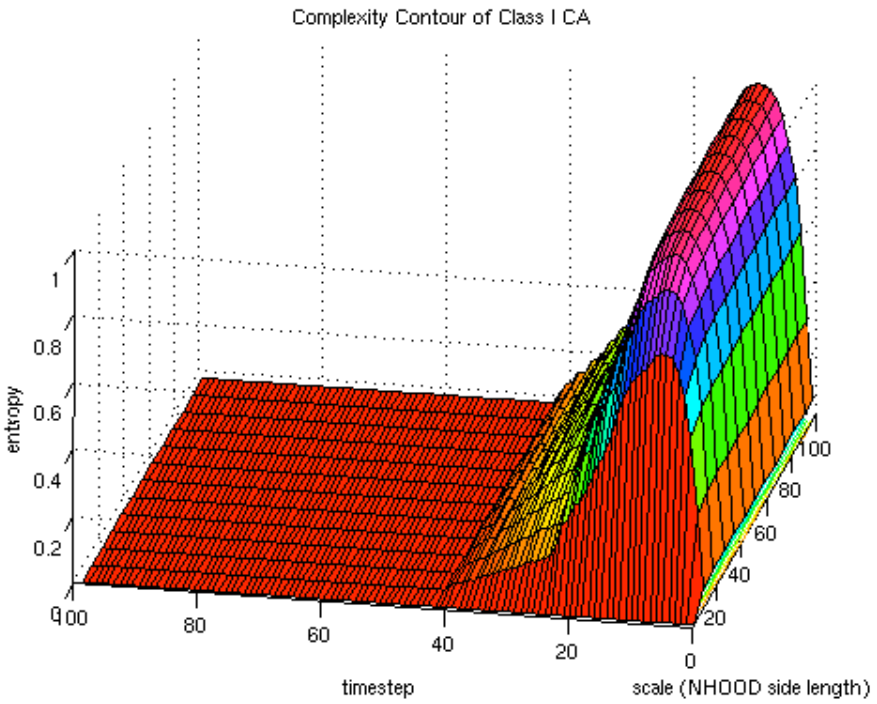

Figure 3: Class III Complexity Contour

complexity profile and entropic measurements; however, the explanation behind the entirety of the behavior, particularly the initial behavior, is not so transparent.

The reason behind the initial increase in entropy becomes clear when we look at the initialization of the CA matrix. The important characteristic of the initialization process to note is its bias—cell values are not initialized randomly, and this bias imposes some structure on the system. Furthermore, this structure is neither guaranteed to be preserved or propagated under the CA iterative map.

If, as is very likely, this structure is not preserved, its destruction increases the complexity (*i.e.*, entropy) of the system initially, returning the system to disorder before imposing the structure associated with the CA rule/class.

This is, in fact, exactly what we see; our intuition is confirmed by cursory inspection of the image sequence itself.

4.2 Class II Cellular Automata

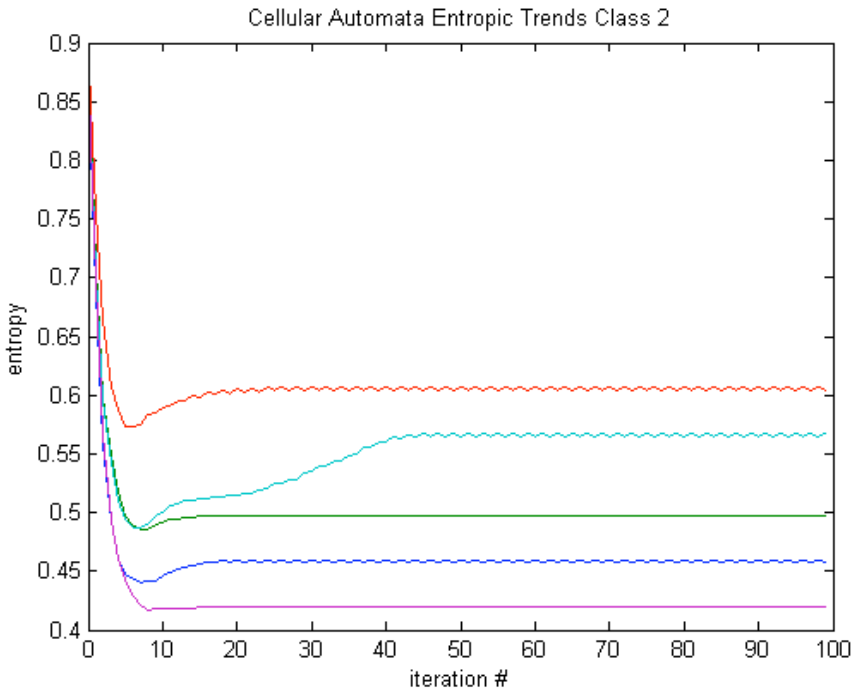

Figure 4: Class II Entropic Time Series

Evolving to a fixed or cyclic, inhomogeneous state, it is immediately clear that the entropy of Class II CAs will be nonzero in its long range behavior. We see here the same destruction of sstructure and commensurate increase in entropy preceding the imposition

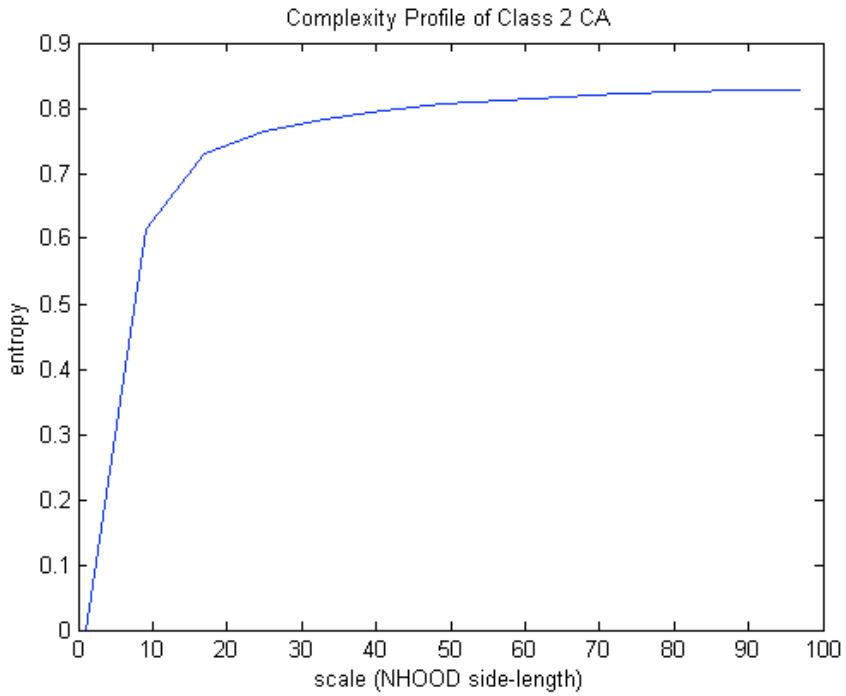

Figure 5: Typical Class II Long Range Complexity Profile

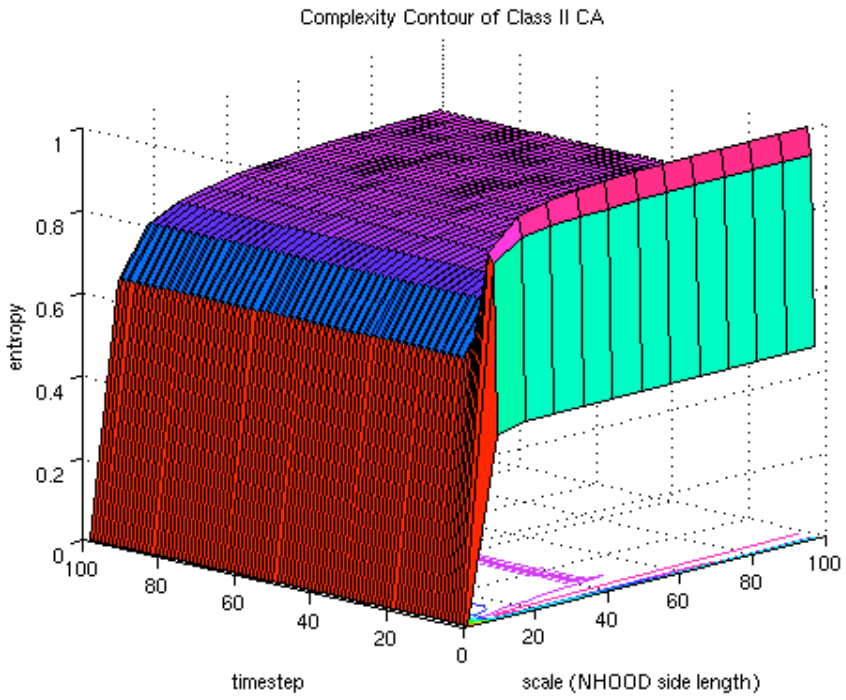

Figure 6: Class III Complexity Contour

of the rule's structure on the matrix. Moreover, we located an of Class II behavior that created both cyclic and fixed inhomogeneity, a fact readily and dramatically observable in the profiles of the entropic progression of the system with time wherein some continue very small oscillations around some value ϵ indefinitely and some reach a constant entropy.

4.3 Class III Cellular Automata

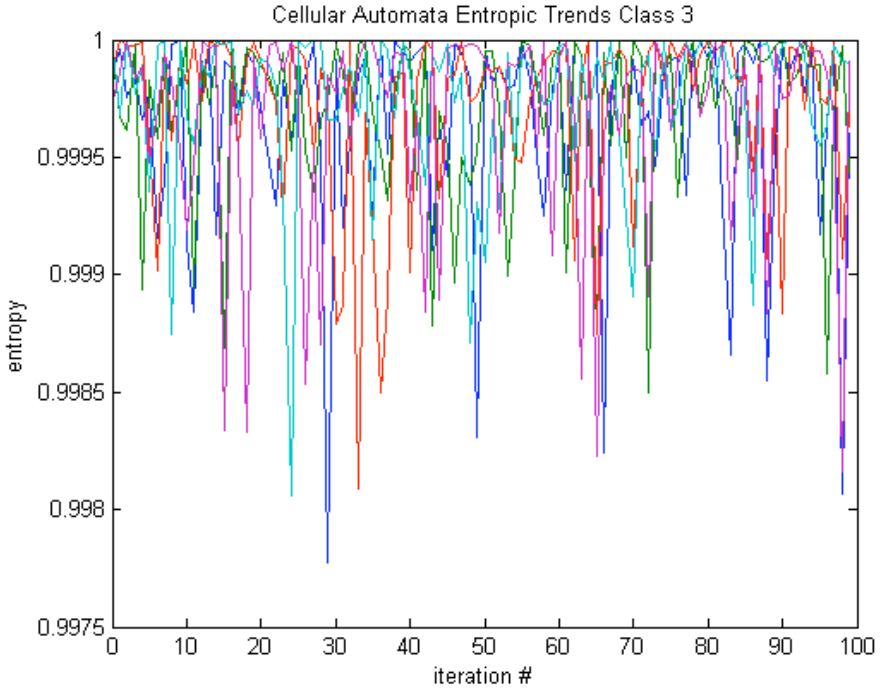

Figure 7: Class III Entropic Time Series

Class III is characterized by chaotic, aperiodic behavior on a large scale. Even from this exceedingly qualitative description, it is possible to extract a surprising amount of information about the nature of the CA.

Consider for a moment that the information stored in the initial configuration of the CA. In moving to chaotic behavior, we see a move to randomness, which can be intuitively equated to an increase in entropy, and thus complexity.

The difference in this case is that the long-range behavior is itself chaotic. *This* is the structure which the CA imposes, rather than the structures we've seen in Classes I and II. As such, one would expect to see a very high entropy which is *not* constant, but does remain within a certain ϵ, and this is precisely the behavior we observed.

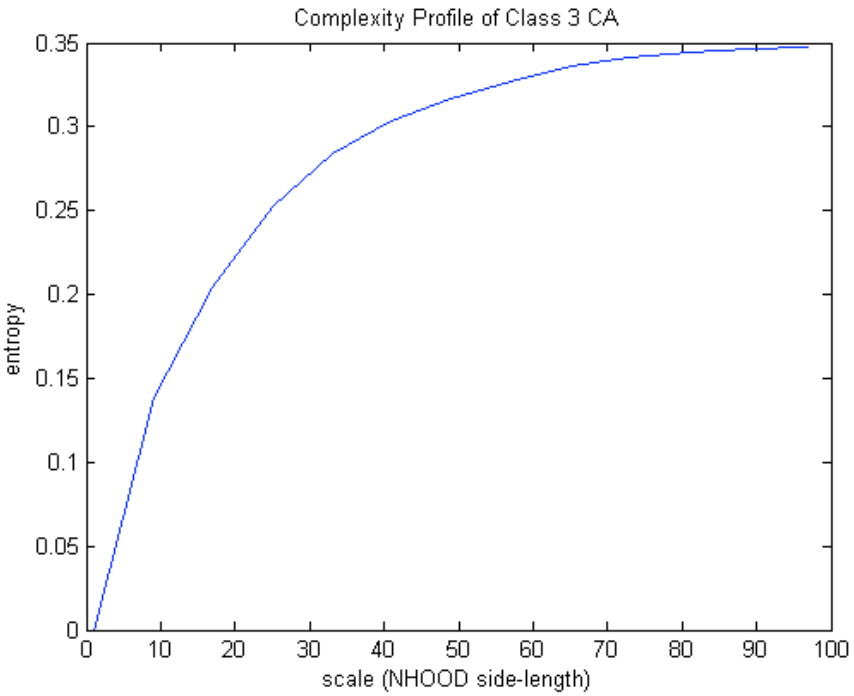

Figure 8: Typical Class III Long Range Complexity Profile

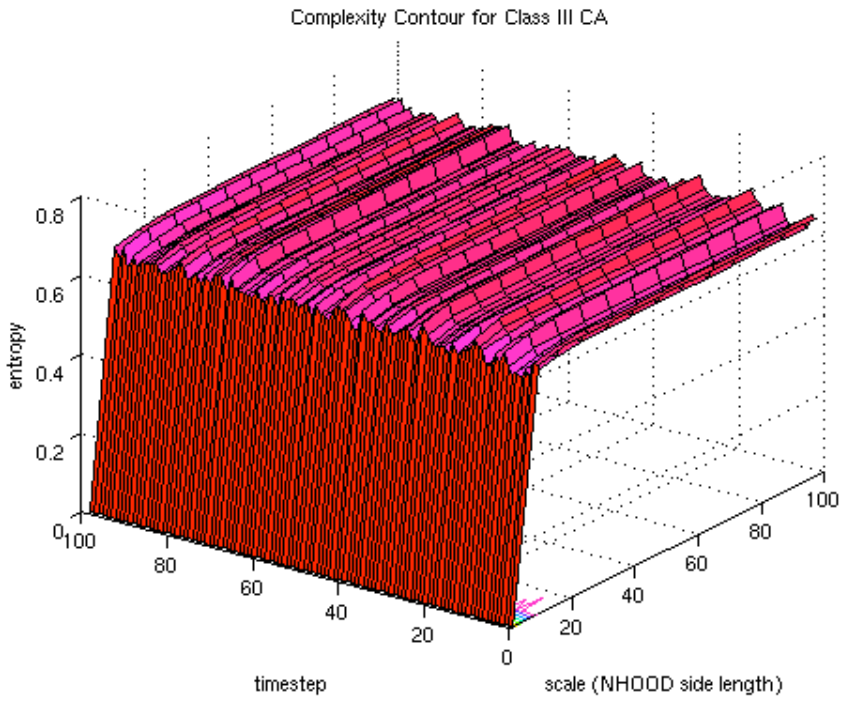

Figure 9: Class III Complexity Contour

4.4 Class IV Cellular Automata

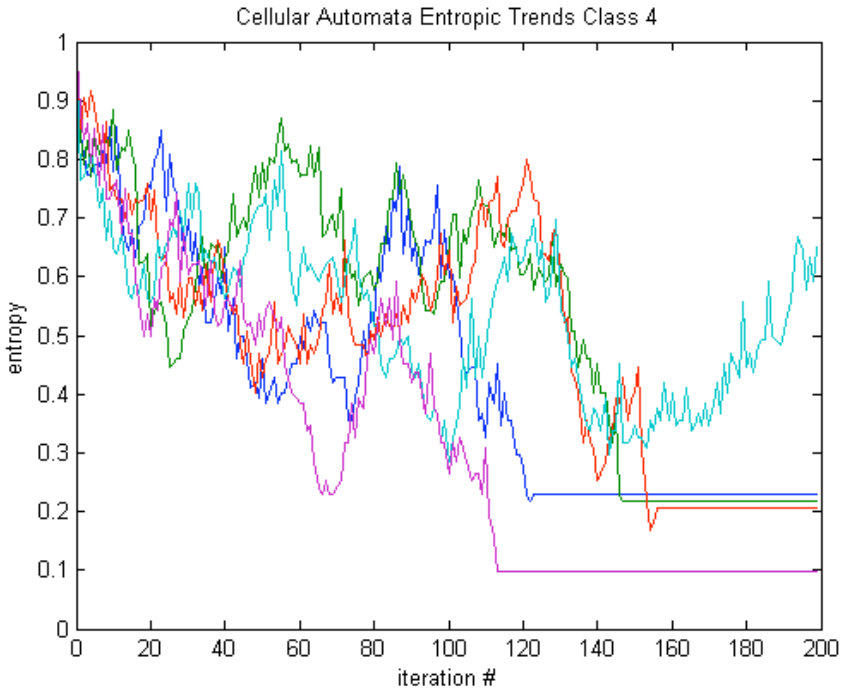

Figure 10: Class IV Entropic Time Series

Though Class IV is the least well-defined of the CA classes, it is also the most intriguing. Class IV is often described has having "complex, localized structure." From this description, it is not nearly as immediately obvious what the behavior of the complexity profile will be.

One may take a first clue from the idea of *localized structure*. This would indicate that at some scale (determined by the necessary localization) structure would emerge; however, global complexity would also prevent meaningful prediction of overall structure. This would imply that at larger scales there *may* be localized (in the scaling sense) instances of structure emergence (rare, nonetheless, given its requirement of spatial coincidence of fine-grain structure), but globally there would be no guarantee of such structure despite the well-defined structure at smaller scales.

This is precisely the behavior we see modeled in the complexity contour. Qualitatively, the complexity profiles at various timesteps are very similar to those we saw in Class III; however, over time as local structure emerges and disappears, we see a corresponding translation of the quasi-Class III complexity profile, giving rise to the periodic surface one sees in the complexity contour until the long range behavior stabilizes.

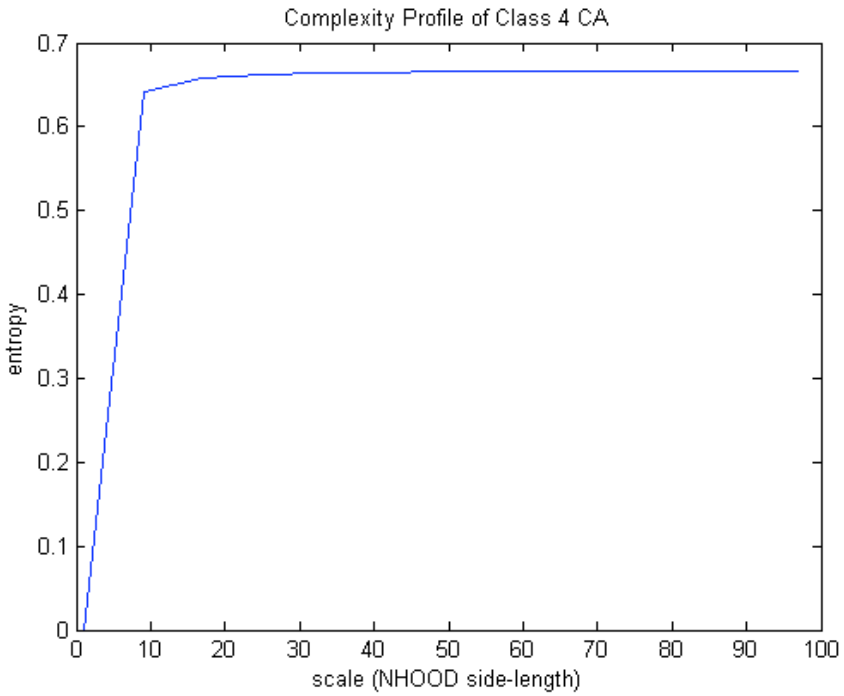

Figure 11: Typical Class IV Long Range Complexity Profile

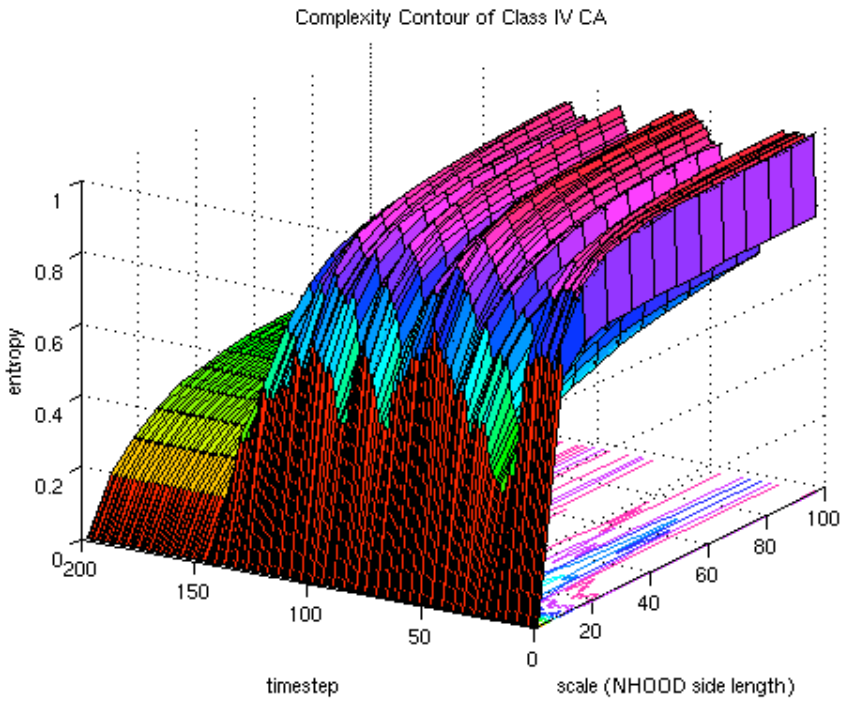

Figure 12: Class IV Complexity Contour

5 Discussion and Conclusion

Most significantly, we have established not only the efficacy of the complexity profile/contour in assessing the nature of the relationship between scale and complexity, but the importance of this measurement itself. Furthermore, the wide range of systems which can be modeled by cellular automata implies that the primacy of the relationship between complexity and scale in characterizing a system generalizes, meaning that an assessment of a system's complexity is insufficient to handle the system in an intelligent manner.

Additionally, our results indicate an intimate link between scale and information that is not obvious in any way, while affirming the intimate connection between information, entropy, and complexity. The scaling behavior of these quantities, however, holds promise as a deep and rich relationship for speaking meaningfully about the fundamental nature of the system.

Incidentally, these results further support Wolfram's initial proposal for a classification scheme; although, we must be careful in separating his classification system from the acquired data, given their *a priori*, systemic connection. Nonetheless, the tools and techniques developed here provide for a strong foundation from which to analyze recent claims of Wolfram's scheme's inadequacy. Beyond this, we see that it is easy (with the tool of the complexity profile/contour in hand) to make Wolfram's taxonomy more precise by speaking about the scaling behavior of structure as well as the qualitative impact of the system.

Thus, in the brief time we had, we were able to use Bar-Yam's complexity profile and extend it to observe the behavior of a system over time, in the process generating a very powerful tool for speaking about the nature of a system's complexity and development from a functional standpoint.

There are still many questions to be explored and endeavors to be undertaken to refine this set of ideas and methods, including the establishment of mathematical rigor in connecting the structural development of information content with the complexity profile as it has been calculated, refining the interpretation of complexity profiles/contours to allow for the meaningful and careful application of the concept to modeled systems, and eventually connecting the algebraic structure of the configuration space with invariants associated with each class's complexity profile or contour, among many other directions which we hope will be pursued in the coming time.

References

[1] Yaneer Bar-Yam. *Dynamics of complex systems*. Addison-Wesley, Reading, Mass., 1997. Yaneer Bar-Yam. ill. ; 24 cm. Studies in nonlinearity.

[2] Yan Guangwu. Sub-classes and evolution stability of wolfram's classes in the total-rule cellular automata. *Progress in Natural Science*, 14(12):1122–1125, 2004.

[3] Stephen Wolfram. *Theory and applications of cellular automata : including selected papers 1983-1986*. World Scientific, Singapore ; Philadelphia, 1986. [edited by] Stephen Wolfram. ill. 24 cm. World Scientific advanced series on complex systems ; vol. 1 Reprints of articiles from vaious journals originally published 1983-1985 Includes index.

A Data Collected

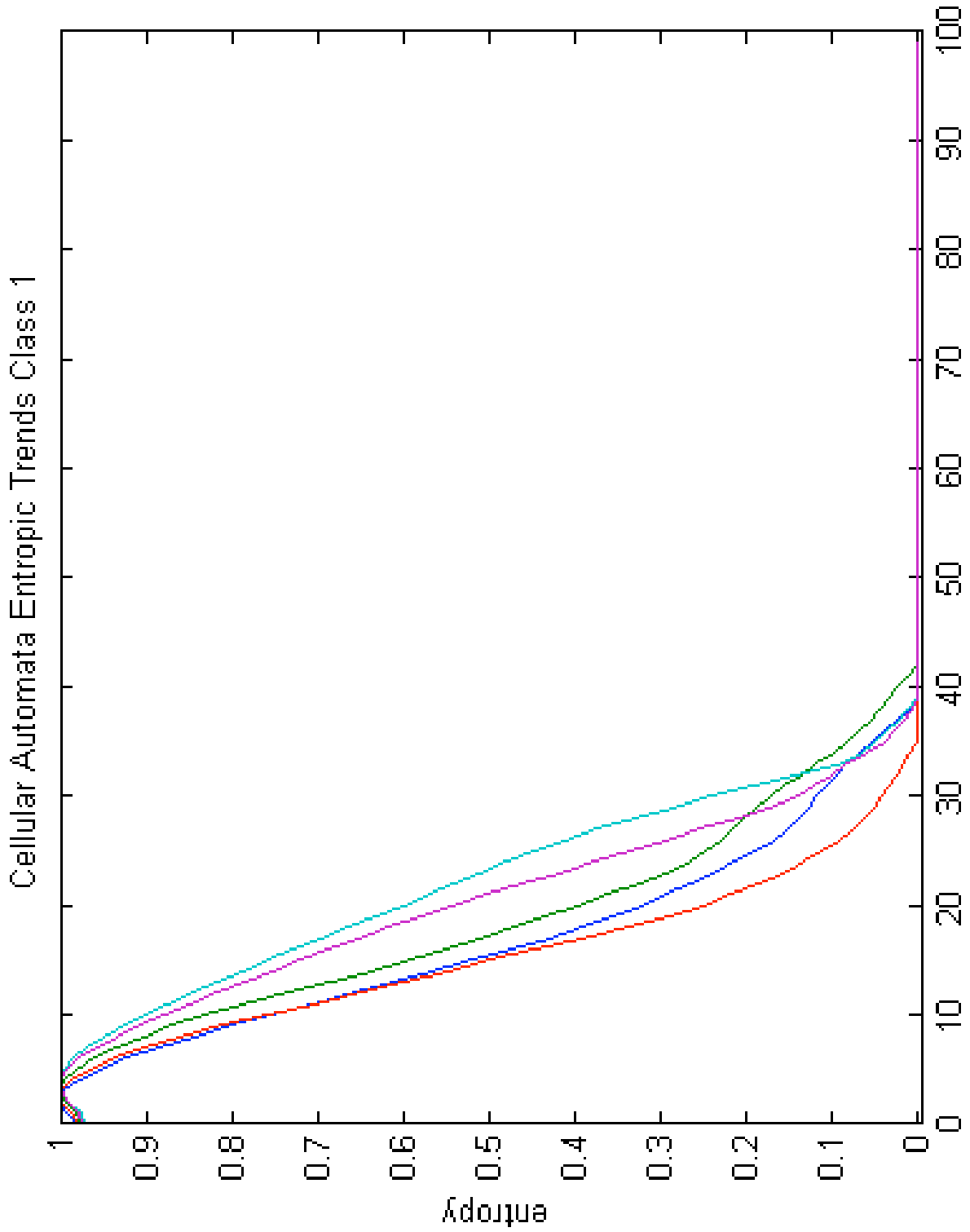

Figure 13: Class I CA Entropic Time Series

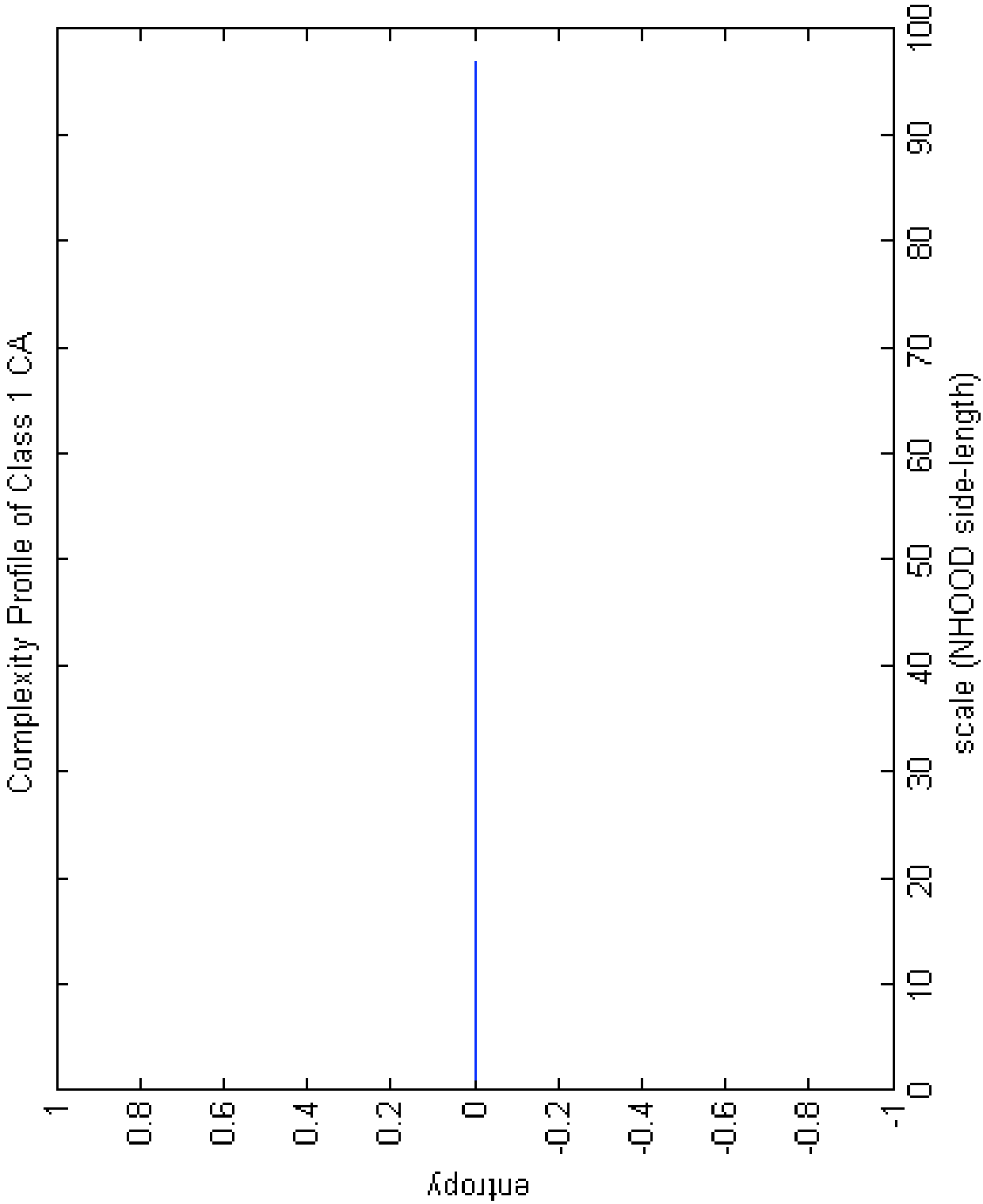

Figure 14: Class I CA Complexity Profile

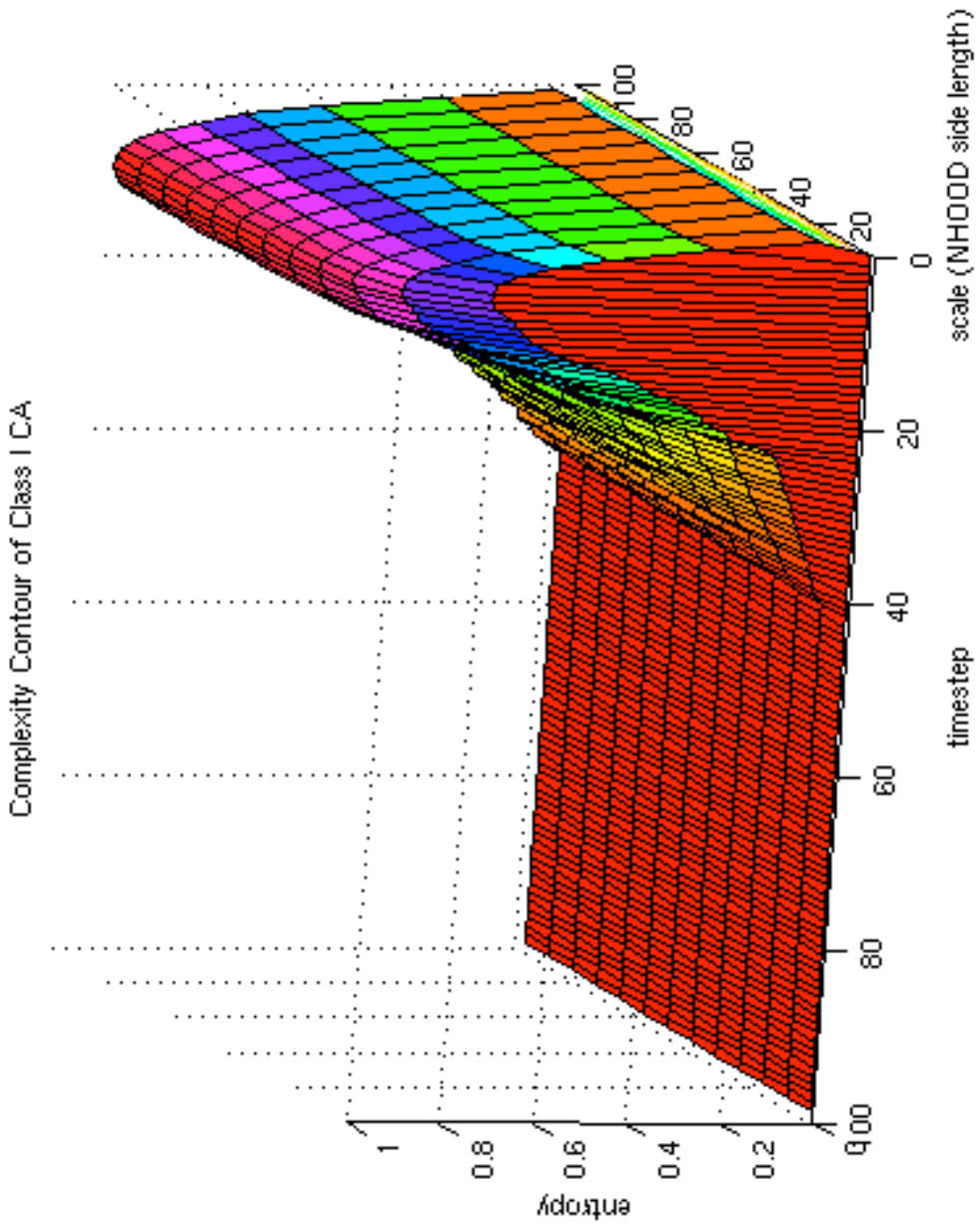

Figure 15: Class I CA Complexity Contour

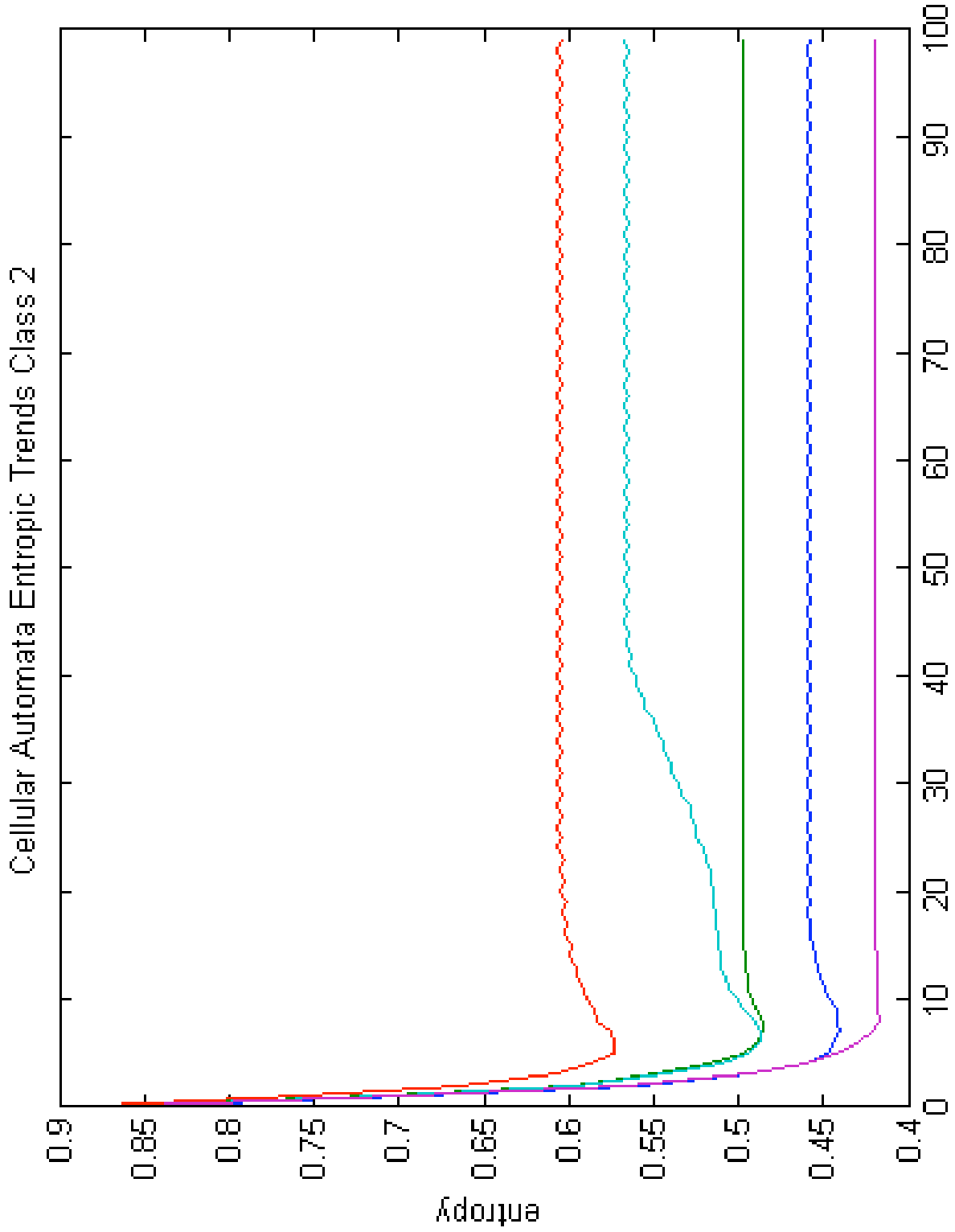

Figure 16: Class II CA Entropic Time Series

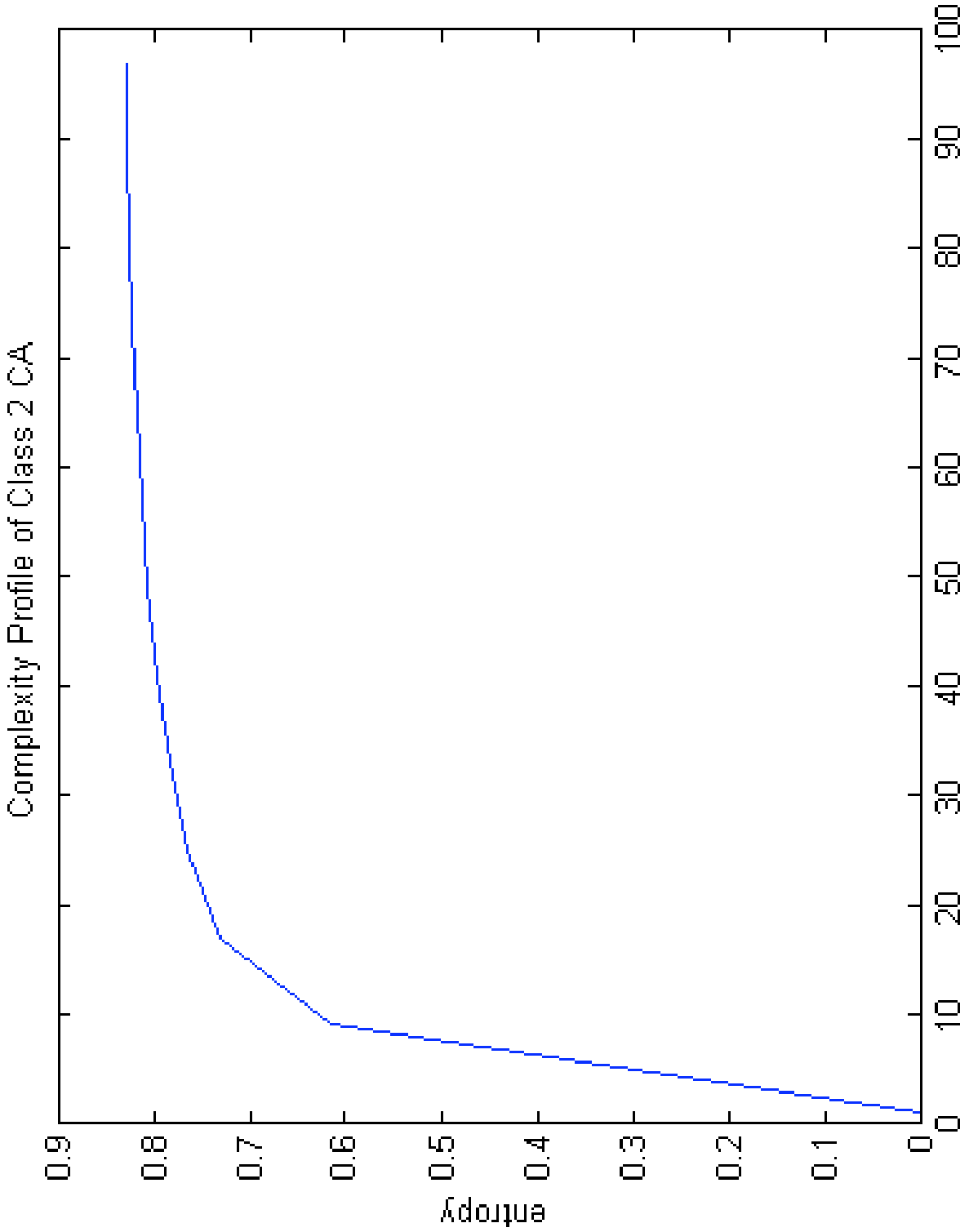

Figure 17: Class II CA Complexity Profile

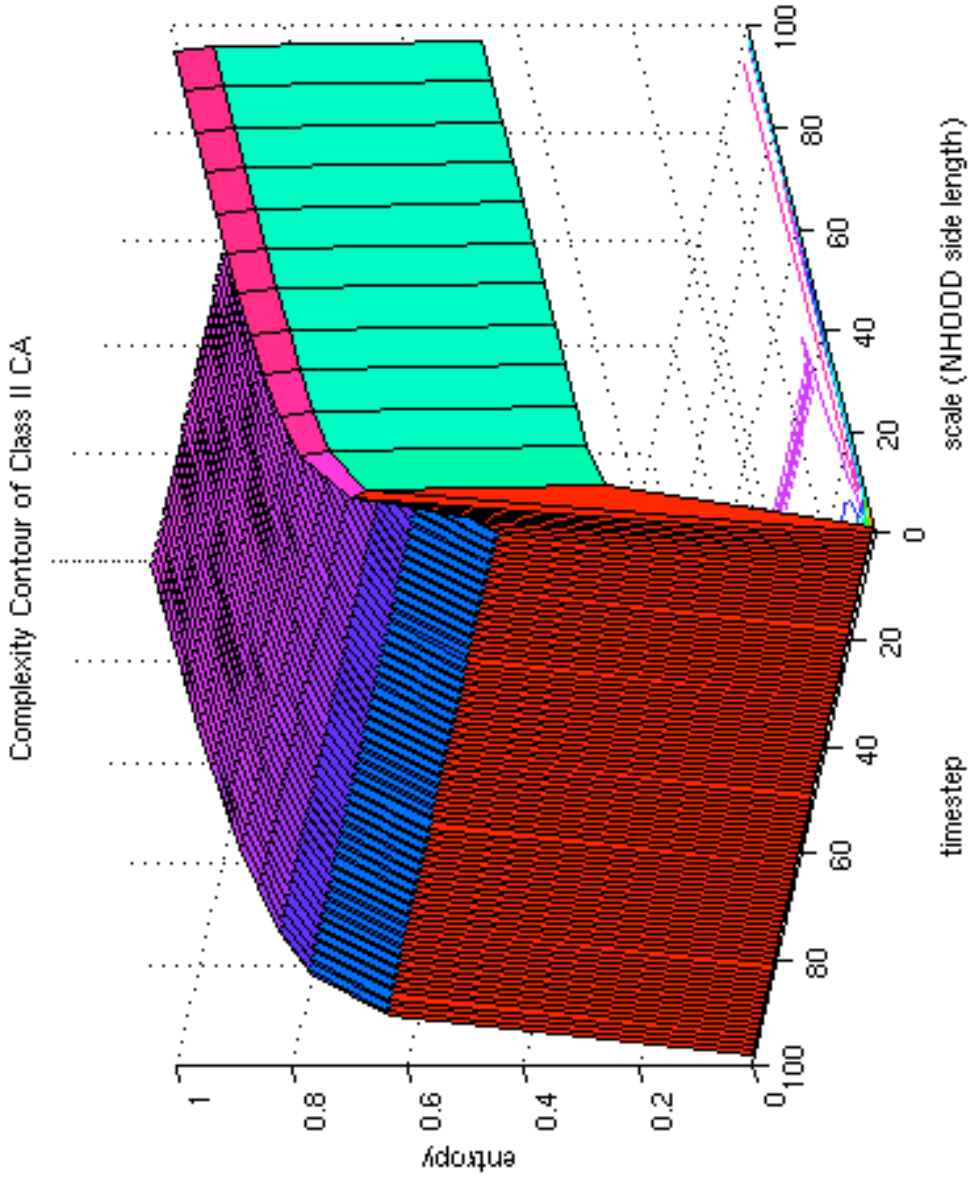

Figure 18: Class I CA Complexity Contour

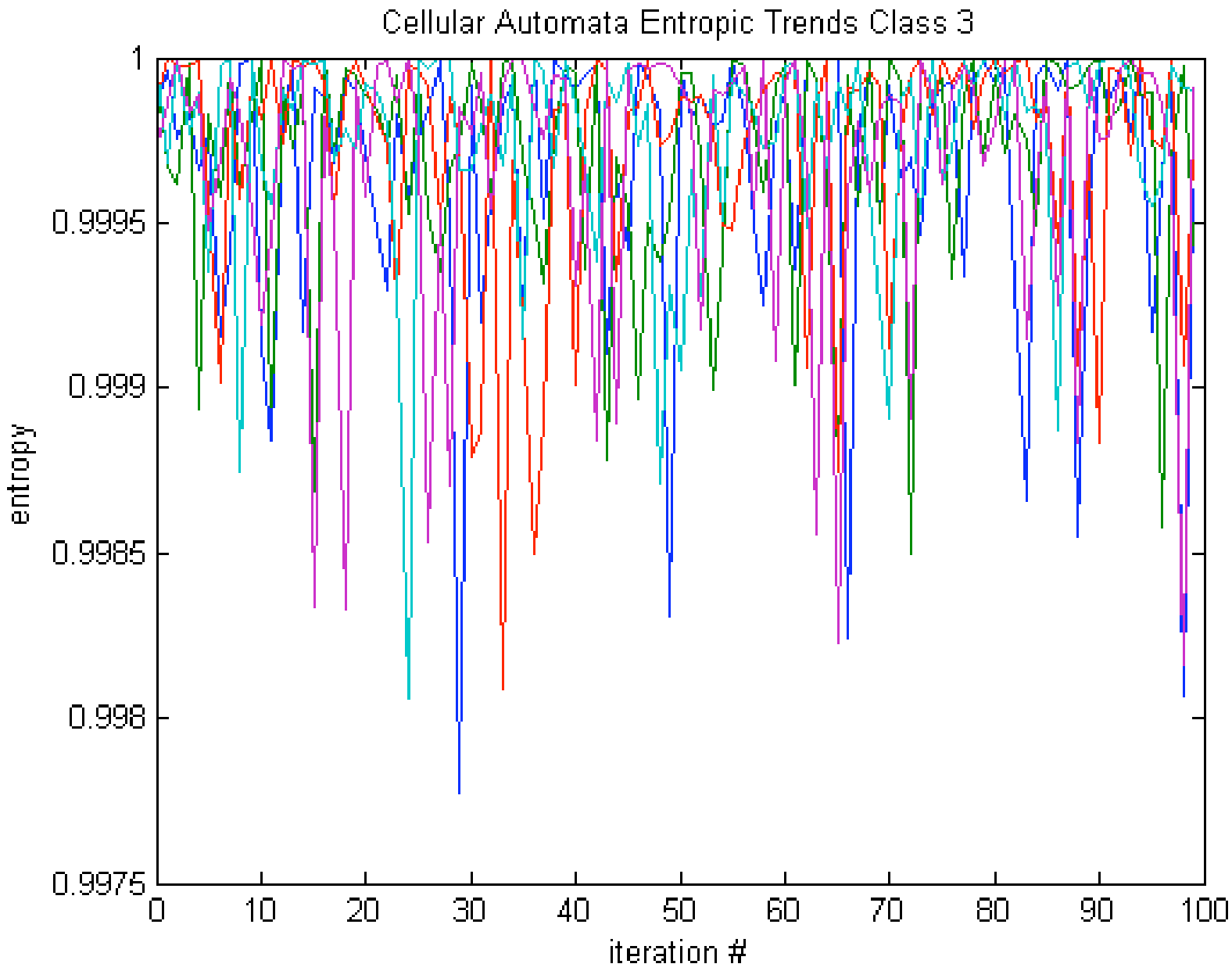

Figure 19: Class III CA Entropic Time Series

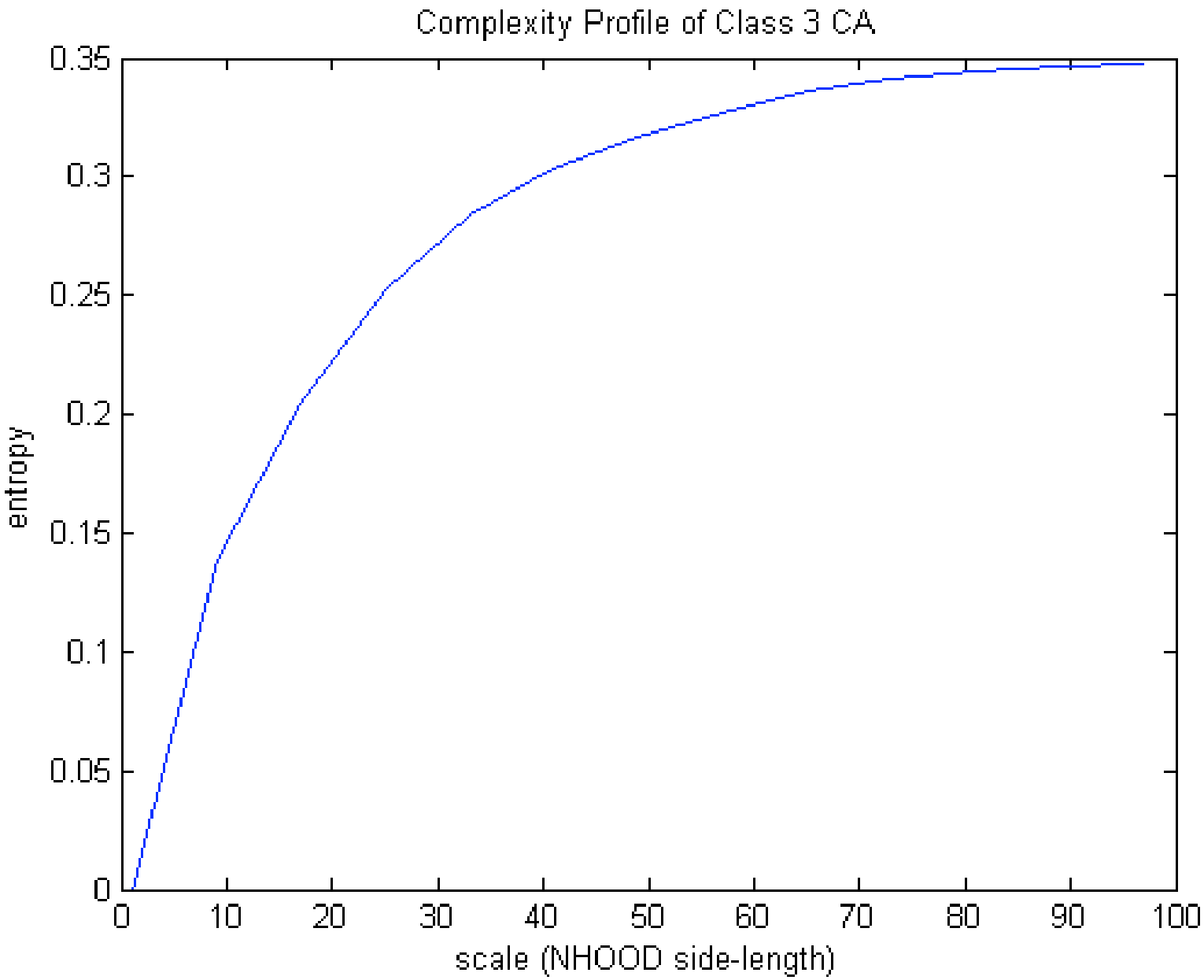

Figure 20: Class III CA Complexity Profile

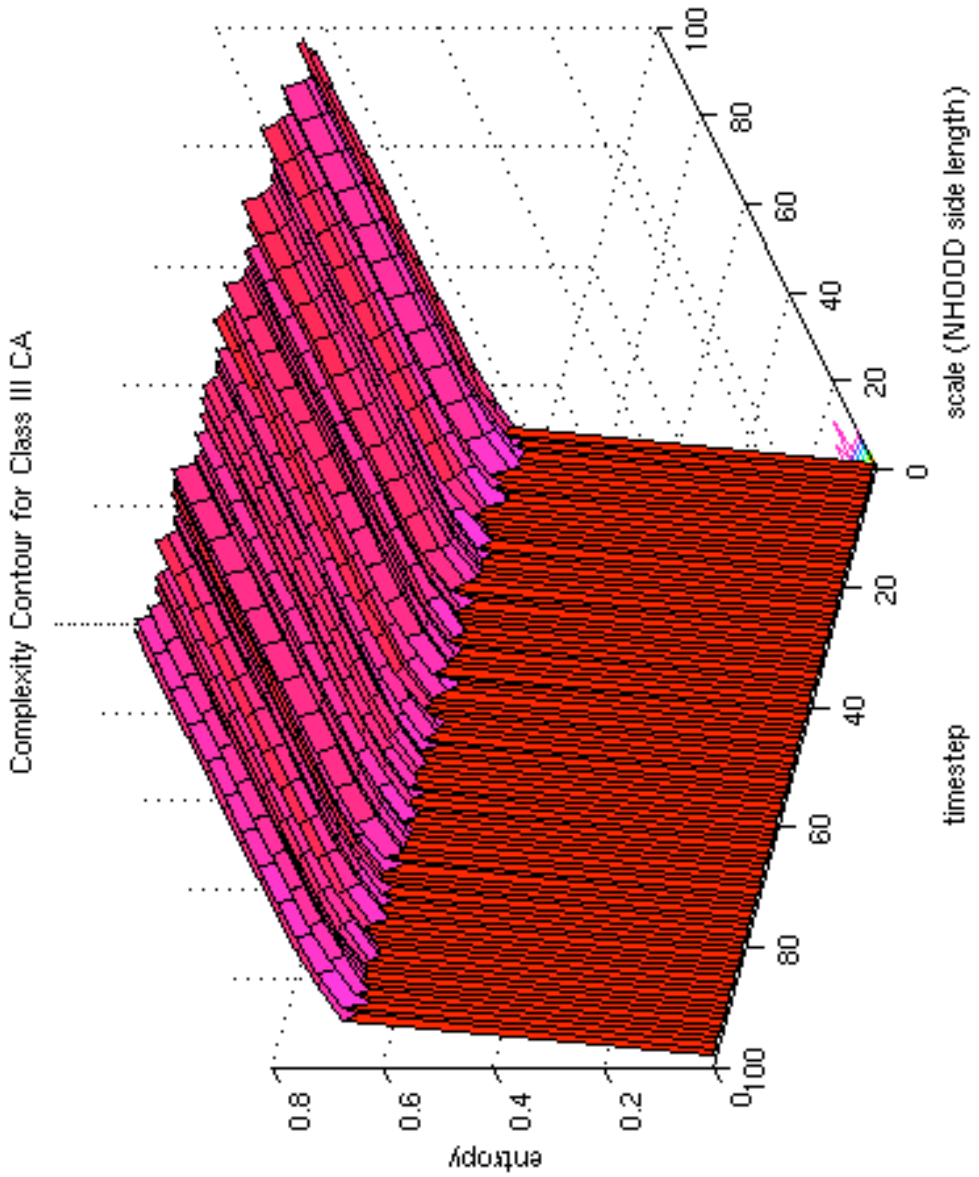

Figure 21: Class I CA Complexity Contour

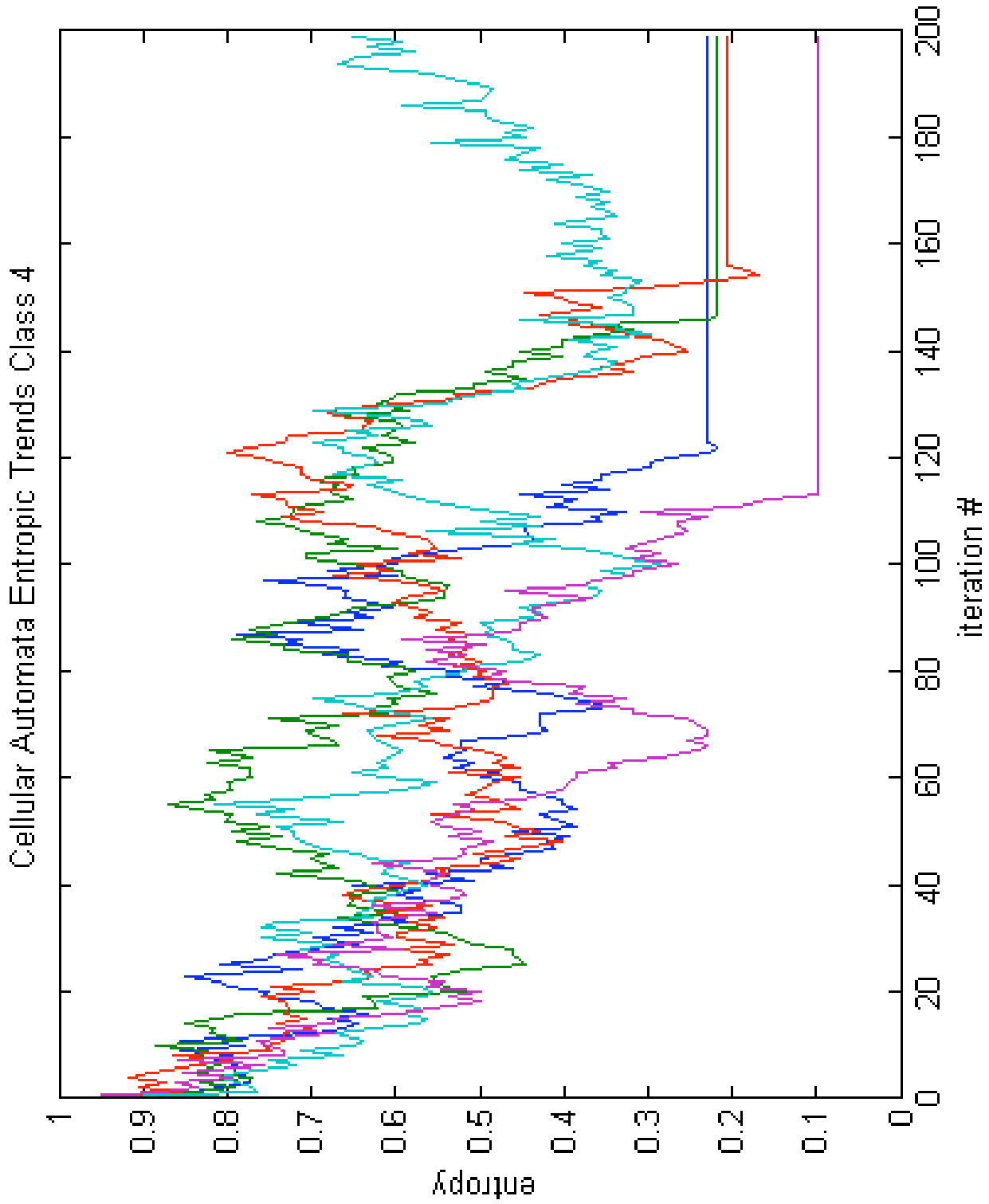

Figure 22: Class IV CA Entropic Time Series

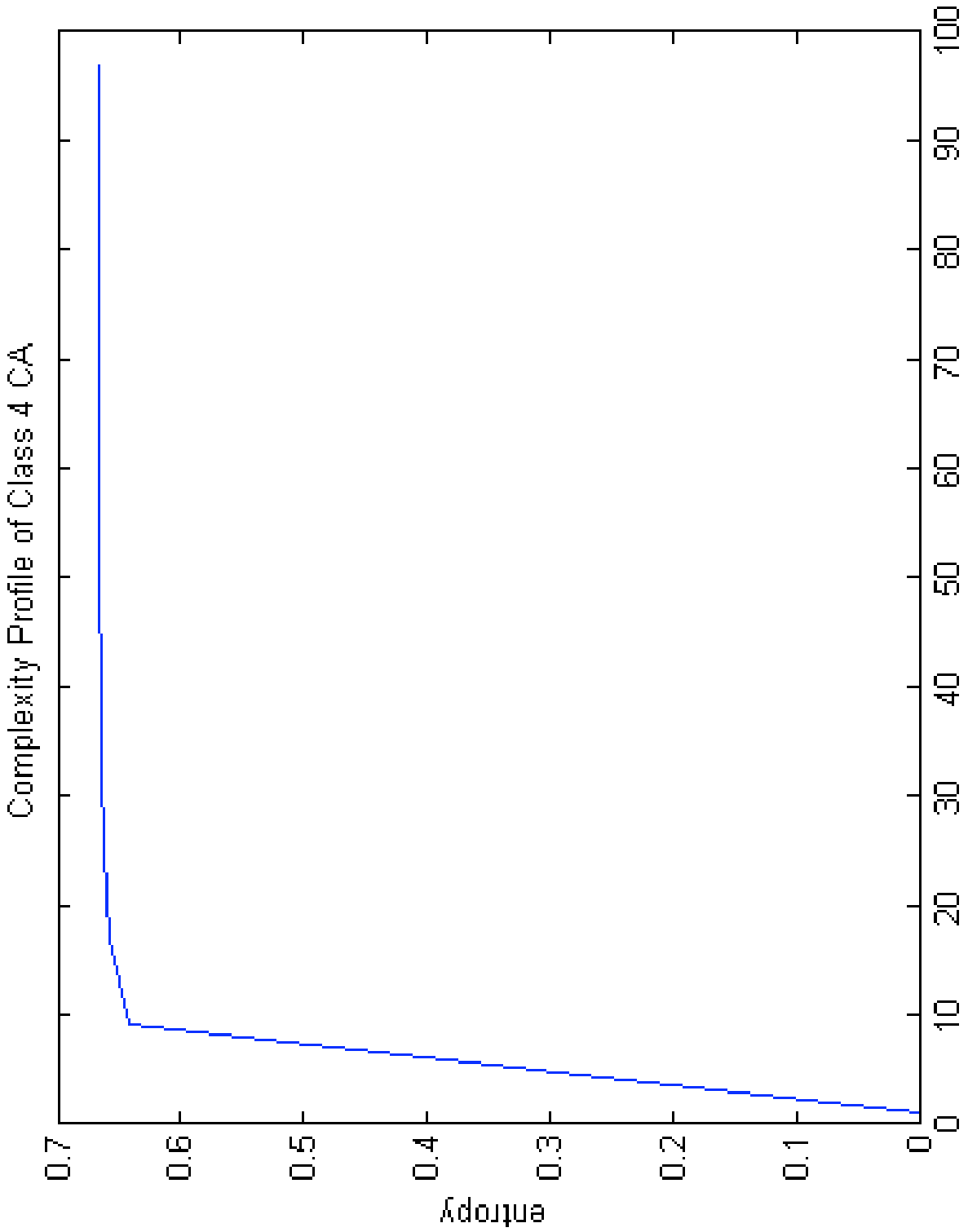

Figure 23: Class IV CA Complexity Profile

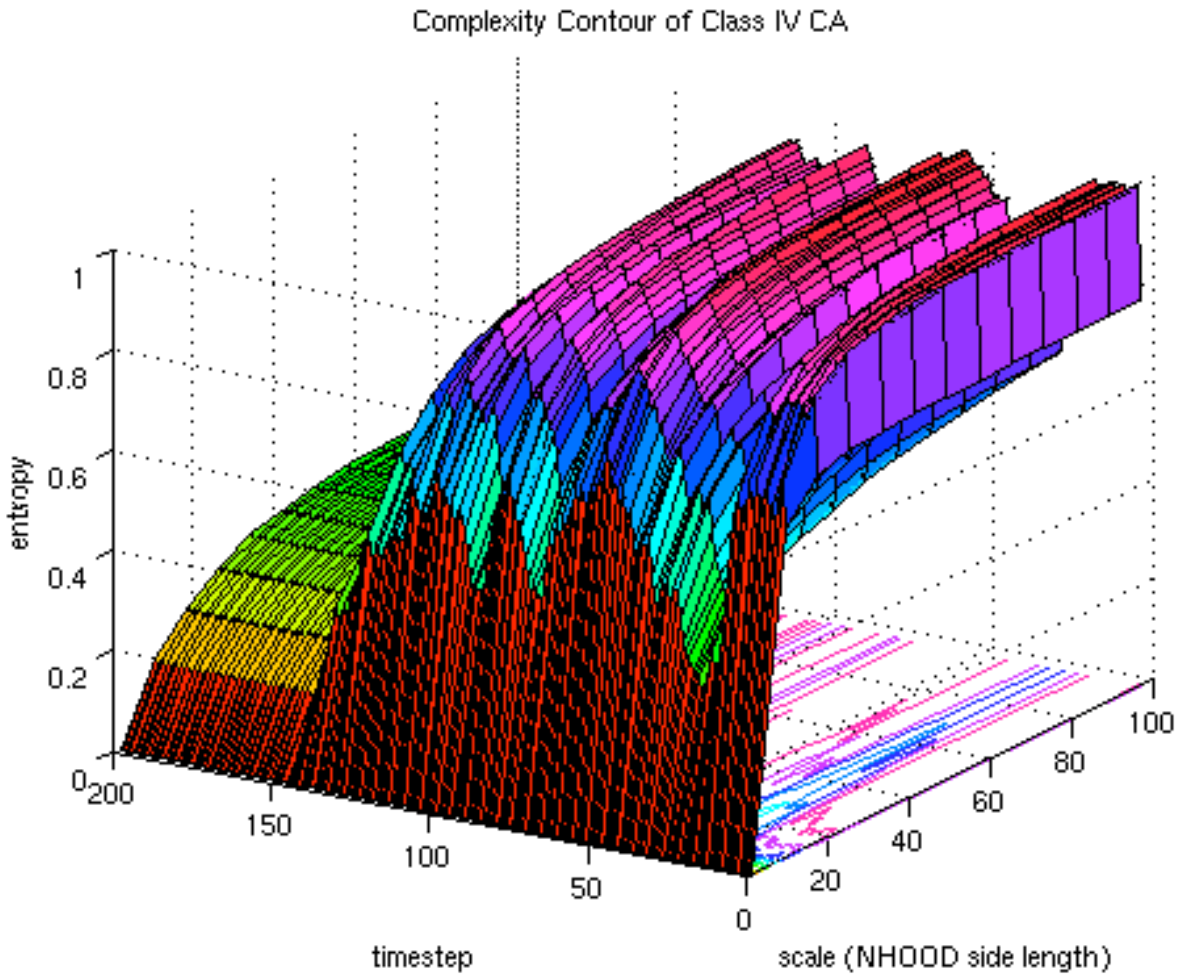

Figure 24: Class I CA Complexity Contour

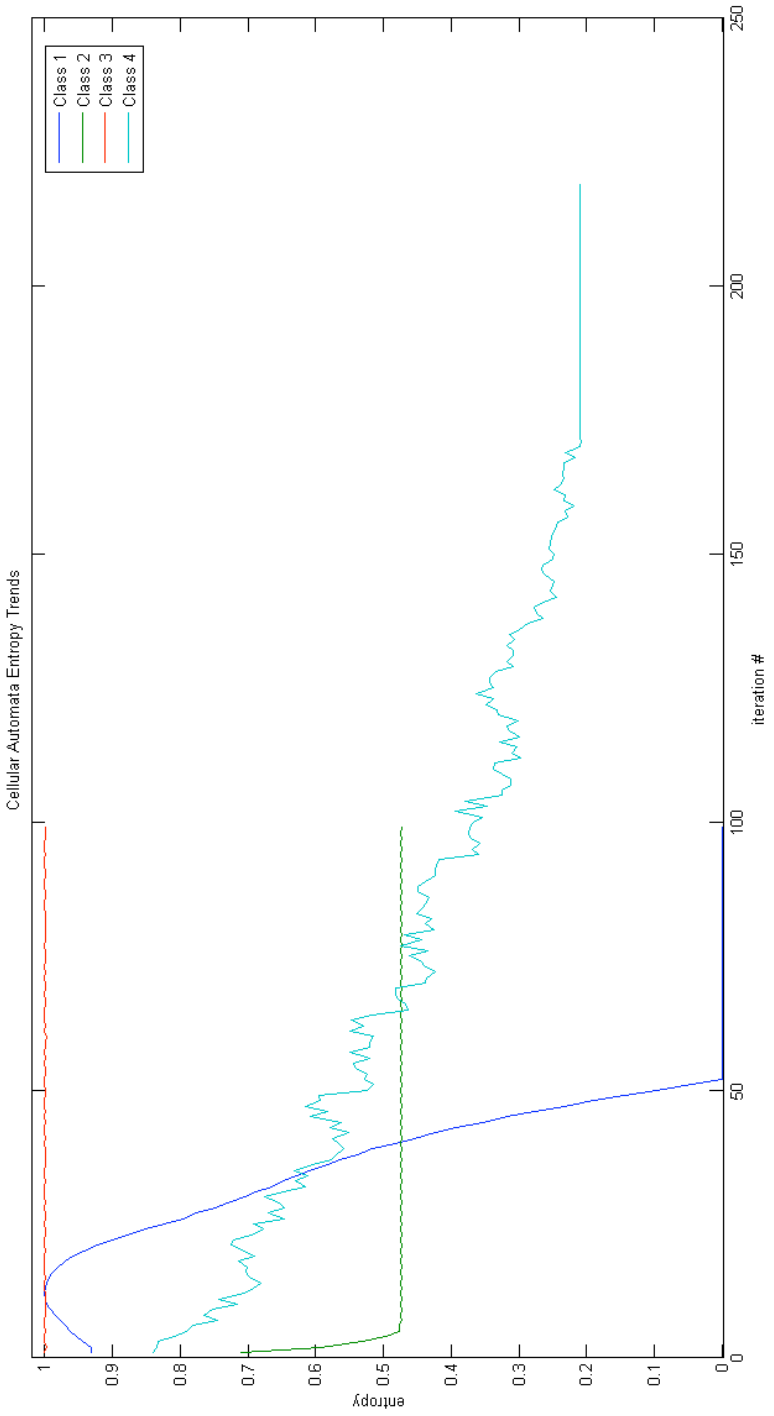

Figure 25: example caption

B Java Simulation Code

B.1 CAApplet.java

```
 import java.applet.Applet;
public class CAApplet extends Applet
{
public void init()
{
CellularAutomata.startCellularAutomata();
}
}
```

B.2 FileOperations.java

```
 import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.imageio.*;
import java.awt.image.BufferedImage;
import java.text.*;
import java.beans.XMLEncoder;
import java.beans.XMLDecoder;

class FileOperations {
public void saveInitialConditionsTemplate()
{
try
{
int[][] initCond = new int[Parameters.sideCells][Parameters.sideCells];
for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
initCond[i][j] = -1;
encodedInitialConditions EIC = new encodedInitialConditions();
            EIC.setInitialConditions(initCond);


File outputFile = new File("C:\\Java\\CellularAutomata\\initialConditions.txt");

XMLEncoder e = new XMLEncoder(
                      new BufferedOutputStream(
```

```
                        new FileOutputStream(outputFile)));

            e.writeObject(EIC);
        e.close();
    }
    catch(Exception e)
    {
     e.printStackTrace();
    }
}
}
```

B.3 Cell.java

```
 import java.awt.*;

public class Cell
{
public final static int N=0,NE=1,E=2,SE=3,S=4,SW=5,W=6,NW=7;
Polygon cellBox;
Color cellColor;
Cell[] surroundingCells;
double state = 0;
double futureState = 0;
public Cell(int xPix,int yPix,int W,int H)
{
int[] xpoints = {xPix,xPix+W,xPix+W,xPix};
int[] ypoints = {yPix,yPix,yPix+H,yPix+H};

cellBox = new Polygon(xpoints,ypoints,4);
}

public Cell(Cell c)
{
cellBox = c.getRectangle();
setState(c.getState());
}

public void setState(double s)
{
```

```
double min=-1,max=1;
state = s;
double c = (s-min)/(max-min);
//System.out.println("c = "+c);
int a = (int)(c*255.0);
//a = 255-a;
cellColor = new Color(a,a,a);
}
public double getState()
{
return state;
}

public void setSurroundingCells (Cell[] surroundings)
{
surroundingCells = surroundings;
}
public Polygon getRectangle()
{
return cellBox;
}
public Color getColor()
{
//cellColor = new Color((int)((double)Math.random()*255),(int)((double)Math.random()*255),(
return cellColor;
}

public void applyRuleVirtually()
{
futureState = Parameters.applyRule(this,surroundingCells);

}

public void actuallyApplyFutureState()
{
setState(futureState);
}
}
```

B.4 CAController.java

```
 import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
class CAController extends JPanel{
public CAController()
{
super();
JButton stopButton = new JButton("Stop");
stopButton.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent a)
{ Parameters.stopRunningFlag = true; }});
add(stopButton);

JFrame f = new JFrame("CA Control");
f.setBounds(600,300,100,100);
f.getContentPane().add(this);
f.setVisible(true);
}
}
```

B.5 Cellular Automata.java

```
 /**
 * @(#)CellularAutomata.java
 * @author Curran Kelleher
 * @version 1.00 06/01/09
 */

import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.imageio.*;
import java.awt.image.BufferedImage;
import java.text.*;
import java.beans.XMLEncoder;
import java.beans.XMLDecoder;
```

```
import java.util.*;

public class CellularAutomata extends JPanel implements Runnable {

static int ApW = 500;//width and height of the applet
static int ApH = 500;

int PWidth = 500;//width and height of the cell grid
int PHeight = 500;
int margin = 0;//distance from the side of the screen

static int runNumber = 0;
static boolean running = false;

static int minScale = 46;
static int maxScale = 1000;
static int numTrialsPerScale = 1;
static int currentDataPointIndex = 0;
static int[] periodDataPoints = new int[(maxScale-minScale)*numTrialsPerScale];
static int[] sizeDataPoints = new int[(maxScale-minScale)*numTrialsPerScale];

File outputFile = new File("C:\\Java\\CellularAutomata\\data.csv");
PrintWriter CSVout;


Cell[][] cells = new Cell[Parameters.sideCells][Parameters.sideCells];

BufferedImage bufferImg;
Graphics buffer;

public static void main(String[] args)
{
startCellularAutomata();
}

public static void startCellularAutomata()
{
// FileOperations fOp = new FileOperations();
// fOp.saveInitialConditionsTemplate();
//
CAController control = new CAController();
```

```
JFrame f = new JFrame("Cellular Automata");
CellularAutomata c = new CellularAutomata();
f.getContentPane().add(c);
f.setSize(ApW+50,ApH+100);
f.setVisible(true);

if(Parameters.mode == Parameters.MODE_OUTPUTIMAGES)
{
for(int i=0;i<Parameters.numRuns;i++)
{
c.startRunThread();
running = true;
while (running)
{
try{Thread.currentThread().sleep(Parameters.runWaitTime);}catch(Exception e){}
System.out.println("waiting on run #"+runNumber);
}
runNumber++;
}
}
else if(Parameters.mode == Parameters.MODE_RENDERONSCREEN)
{
c.startRunThread();
}
else if(Parameters.mode == Parameters.MODE_GENERATESCALEPERIODDATA)
{
// c.startRunThread();
for(double s=minScale;s<maxScale;s+=1.0/(double)numTrialsPerScale)//s = #of side cells
{
Parameters.sideCells = (int)s;
c.startRunThread();
running = true;
while (running)
{
try{Thread.currentThread().sleep(1000);}catch(Exception e){}
System.out.println("waiting on run with "+s+" side cells");
}
}

}
```

```
}

public void addPeriodDataPoint(int currentPeriod)
{
if(CSVout==null)
try
{
CSVout = new PrintWriter(new BufferedWriter(new FileWriter(outputFile)));
}
catch(Exception e)
{
e.printStackTrace();
}

periodDataPoints[currentDataPointIndex] = currentPeriod;
sizeDataPoints[currentDataPointIndex] = Parameters.sideCells;

CSVout.println(sizeDataPoints[currentDataPointIndex]+","+periodDataPoints[currentDataPointI
CSVout.flush();

currentDataPointIndex++;

//System.out.println("data so far...");
//System.out.println("sideCells\tperiod");
for(int l = 0;l< (maxScale-minScale)*numTrialsPerScale;l++)
{
//System.out.println(sizeDataPoints[l]+"\t\t"+periodDataPoints[l]);
}

Parameters.stopRunningFlag = true;//stop the current CA, move on
}

public CellularAutomata()
{
super();
initCells();
setVisible(true);
bufferImg = new BufferedImage(ApW, ApH, BufferedImage.TYPE_INT_RGB);
buffer = bufferImg.createGraphics();
```

```
}

public void initCells()
{
cells = new Cell[Parameters.sideCells][Parameters.sideCells];

//create the cells
for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
{
int W = (int)((double)(i+1)/Parameters.sideCells*PWidth+margin);
W -= (int)((double)i/Parameters.sideCells*PWidth+margin);
int H = (int)((double)(j+1)/Parameters.sideCells*PWidth+margin);
H -= (int)((double)j/Parameters.sideCells*PWidth+margin);

cells[i][j] = new Cell( (int)((double)i/Parameters.sideCells*PWidth+margin),
(int)((double)j/Parameters.sideCells*PHeight+margin),W,H);

cells[i][j].setState(Parameters.generateInitialCellState(i,j));
}

//interconnect the cells
int I,J;
for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
{
Cell N,NE,E,SE,S,SW,W,NW;

//N
J = j-1;
if(J<0)
J+=Parameters.sideCells;
N = cells[i][J];

//NW
I = i-1;
if(I<0)
I+=Parameters.sideCells;
NW = cells[I][J];

//NE
```

```
I = i+1;
if(I>=Parameters.sideCells)
I-=Parameters.sideCells;
NE = cells[I][J];

//E
E = cells[I][j];

//W
I = i-1;
if(I<0)
I+=Parameters.sideCells;
W = cells[I][j];

//SW
J = j+1;
if(J>=Parameters.sideCells)
J-=Parameters.sideCells;
SW = cells[I][J];

//S
S = cells[i][J];

//SE
I = i+1;
if(I>=Parameters.sideCells)
I-=Parameters.sideCells;
SE = cells[I][J];

Cell[] cellss = {N,NE,E,SE,S,SW,W,NW};
cells[i][j].setSurroundingCells(cellss);
}
}
public void initCellStates()
{
for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
cells[i][j].setState(Parameters.generateInitialCellState(i,j));
}
public void startRunThread()
{
```

```
Thread t = new Thread(this);
t.start();
}
public void run()
{
ArrayList listOfPastStates = new ArrayList();
ArrayList listOfPastStepNumber = new ArrayList();
//DecimalFormat f = new DecimalFormat("000");
initCells();
initCellStates();//init the cells with randomized initial conditions
for(int t=0;t<Parameters.numIterations;t++)
{
if(Parameters.stopRunningFlag)
{
Parameters.stopRunningFlag = false;
running = false;
return;
}

if(Parameters.mode == Parameters.MODE_RENDERONSCREEN)
try{Thread.currentThread().sleep(Parameters.waitTime);}catch(Exception e){}
else
{

}
//System.out.println("    calculating iteration #"+t);
buffer.setColor(Color.black);
buffer.fillRect(0,0,600,600);
//draw
if(Parameters.mode == Parameters.MODE_RENDERONSCREEN || Parameters.mode == Parameters.MODE_
for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
{
buffer.setColor(cells[i][j].getColor());
buffer.fillPolygon(cells[i][j].getRectangle());
}

if(Parameters.mode == Parameters.MODE_OUTPUTIMAGES)
saveImage(Parameters.classNumber+"CA"+runNumber+"CA"+t);

if(Parameters.mode == Parameters.MODE_RENDERONSCREEN || Parameters.overrideShowOnScreen)
```

```
repaint();

for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
cells[i][j].applyRuleVirtually();

for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
cells[i][j].actuallyApplyFutureState();

if(Parameters.mode == Parameters.MODE_GENERATESCALEPERIODDATA)
{
if(!listOfPastStates.isEmpty())
{
boolean statesAreTheSame = true;
int numPastStates = listOfPastStates.size();
for(int n=0;n<numPastStates && statesAreTheSame;n++)//for each past state
{

Cell[][] pastCells = (Cell[][])listOfPastStates.get(n);
for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
if(cells[i][j].getState() != pastCells[i][j].getState())
{
statesAreTheSame = false;
//System.out.println("Iter "+t+" is not the same as iter "+((Integer)listOfPastStepNumber.g
}

if(statesAreTheSame)
{
int periodLength = t-((Integer)listOfPastStepNumber.get(n)).intValue();
System.out.println("Period detected!: of length "+periodLength);
addPeriodDataPoint(periodLength);
}
}
}

if(t%500000 == 1)
{
Cell[][] copy = new Cell[Parameters.sideCells][Parameters.sideCells];
```

```
for(int i=0;i<Parameters.sideCells;i++)
for(int j=0;j<Parameters.sideCells;j++)
{
copy[i][j] = new Cell(cells[i][j]);
}
listOfPastStates.add(copy);
listOfPastStepNumber.add(new Integer(t));
System.out.println("adding to list of states at iter "+t);
}
}
}
running = false;
}

public void saveImage(String fileName)
{
File imageOutputFile = new File("C:\\Java\\CellularAutomata\\Images\\"+fileName+".png");
try
    {
        ImageIO.write(bufferImg, "png", imageOutputFile);
    } catch (IOException e){}
}

public void paintComponent(Graphics g)
{
g.drawImage(bufferImg, 0, 0, this);
}
}
```

B.6 Parameters.java

```
 public class Parameters
{
public static final int MODE_OUTPUTIMAGES = 0, MODE_RENDERONSCREEN = 1;
public static final int MODE_GENERATESCALEPERIODDATA = 2;

public static int numRuns = 5;
public static int classNumber = 3;
public static int numIterations = 20000;//Integer.MAX_VALUE;
```

```
public static int mode = MODE_RENDERONSCREEN;
public static int sideCells = 40;//number of cells on one side of the grid
public static double bias = .5;//percent of cells that are initially on (==1,==white)
public static int waitTime = 10;//ms of wait time between iteration updates on screen
public static int runWaitTime = 1000;//ms of wait time for next run

public static boolean stopRunningFlag = false;//a flag to stop the run ( is reset automatic
public static boolean overrideShowOnScreen = true;//in MODE_OUTPUTIMAGES, this turns on (wh



private Parameters(){}
public static double generateInitialCellState(int i,int j)
{
return getRandomWithBias();
// double result = -1;
// if(i == 20 && j == 20)
// result = 1;
//
// return result;

}

public static double getRandomWithBias()
{
double s = Math.random()-1+bias;
if(s<0)
s = -1;
else
s = 1;
return s;
}

public static double applyRule(Cell currentCentralCell,Cell[] surroundingCells)
{
double futureState = 0;

//game of life rule
// int temp = 0;
// for(int i=0;i<8;i++)
// {
```

```
// temp+=(surroundingCells[i].getState()+1)/2;//count the number of live cells
// //System.out.println("State: "+surroundingCells[i].getState());
// }
//
// if(temp == 3)
// futureState = 1;
// else if(temp == 2 && currentCentralCell.getState() == 1)
// futureState = 1;
// else
// futureState = -1;
//
// return futureState;

//"chaotic"
// int temp = 0;
// for(int i=0;i<8;i++)
// {
// temp+=(surroundingCells[i].getState()+1)/2;//count the number of live cells
// //System.out.println("State: "+surroundingCells[i].getState());
// }
//
// if (temp % 2 == 1)
// futureState = 1;
// else
// futureState = -1;
//
// return futureState;

//class 1 & 2
// int temp = 0;
// for(int i=0;i<8;i++)
// {
// temp+=(surroundingCells[i].getState()+1)/2;//count the number of live cells
// //System.out.println("State: "+surroundingCells[i].getState());
// }
//
// if(temp < 4)
// futureState = -1;
// else
// futureState = 1;
//
```

```
// return futureState;

//mating
int i = (int)(Math.random()*8);

if(surroundingCells[i].getState() == currentCentralCell.getState())//same succeeds
futureState = 1;
else
futureState = -1;
return futureState;

}
}
```

B.7 encodedInitialConditions.java

```
class encodedInitialConditions {
int[][] initialConditions;

public encodedInitialConditions()
{
initialConditions = new int[Parameters.sideCells][Parameters.sideCells];
}

public void setInitialConditions(int[][] ic)
{
initialConditions = ic;
}

public int[][] GetInitialConditions()
{
return initialConditions;
}
}
```

C MATLABImage Analysis Code

C.1 entropy.m

```
 function E = entropy(varargin)
%ENTROPY Entropy of an intensity image.
%   E = ENTROPY(I) returns E, a scalar value representing the entropy of an
%   intensity image.  Entropy is a statistical measure of randomness that can be
%   used to characterize the texture of the input image.  Entropy is defined as
%   -sum(p.*log(p)) where p contains the histogram counts returned from IMHIST.
%
%   ENTROPY uses 2 bins in IMHIST for logical arrays and 256 bins for
%   uint8, double or uint16 arrays.
%
%   I can be multidimensional image. If I has more than two dimensions,
%   it is treated as a multidimensional intensity image and not as an RGB image.
%
%   Class Support
%   -------------
%   I must be logical, uint8, uint16, or double, and must be real, nonempty,
%   and nonsparse. E is double.
%
%   Notes
%   -----
%   ENTROPY converts any class other than logical to uint8 for the histogram
%   count calculation so that the pixel values are discrete and directly
%   correspond to a bin value.
%
%   Example
%   -------
%       I = imread('circuit.tif');
%       E = entropy(I)
%
%   See also IMHIST, ENTROPYFILT.

%   Copyright 1993-2004 The MathWorks, Inc.
%   $Revision: 1.1.8.1 $  $Date: 2004/08/10 01:39:21 $

%   Reference:
%      Gonzalez, R.C., R.E. Woods, S.L. Eddins, "Digital Image Processing
%      using MATLAB", Chapter 11.
```

```
I = ParseInputs(varargin{:});

if ~islogical(I)
  I = im2uint8(I);
end

% calculate histogram counts
p = imhist(I(:));

% remove zero entries in p
p(p==0) = [];

% normalize p so that sum(p) is one.
p = p ./ numel(I);

E = -sum(p.*log2(p));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function I = ParseInputs(varargin)

iptchecknargin(1,1,nargin,mfilename);

iptcheckinput(varargin{1},{'uint8','uint16', 'double', 'logical'},...
              {'real', 'nonempty', 'nonsparse'},mfilename, 'I',1);

I = varargin{1};
```

C.2 entropyfilt.m

```
 function J = entropyfilt(varargin)
%ENTROPYFILT Computes the local entropy of an intensity image.
%   J = ENTROPYFILT(I) returns the array J, where each output pixel contains the
%   entropy value of the 9-by-9 neighborhood around the corresponding
%   pixel in the input image I. I can have any dimension.  If I has more than
%   two dimensions, ENTROPYFILT treats it as a multidimensional intensity
%   image and not as a truecolor image.  The output image J is
%   the same size as the input image I.
%
%   For pixels on the borders of I, ENTROPYFILT uses symmetric padding.  In
%   symmetric padding, the values of padding pixels are a mirror reflection
```

```
%   of the border pixels in I.
%
%   J = ENTROPYFILT(I,NHOOD) performs entropy filtering of the input
%   image I where you specify the neighborhood in NHOOD.  NHOOD is a
%   multidimensional array of zeros and ones where the nonzero elements specify
%   the neighbors.  NHOOD's size must be odd in each dimension.
%
%   By default, ENTROPYFILT uses the neighborhood true(9). ENTROPYFILT
%   determines the center element of the neighborhood by
%   FLOOR((SIZE(NHOOD)+1)/2). To specify the neighborhoods of various shapes,
%   such as a disk, use the STREL function to create a structuring element
%   object and then use the GETNHOOD function to extract the neighborhood from
%   the structuring element object.
%
%   Class Support
%   -------------
%   I can be logical, uint8, uint16, or double, and must be real and
%   nonsparse. NHOOD can be logical or numeric and must contain zeros and/or
%   ones. The output array J is double.
%
%   Notes
%   -----
%   ENTROPYFILT converts any class other than logical to uint8 for the histogram
%   count calculation so that the pixel values are discrete and directly
%   correspond to a bin value.
%
%   Example
%   -------
%       I = imread('circuit.tif');
%       J = entropyfilt(I);
%       imshow(I);
%       figure, imshow(J,[]);
%
%   See also ENTROPY, RANGEFILT, STDFILT, IMHIST.

%   Copyright 1993-2003 The MathWorks, Inc.
%   $Revision.1 $  $Date: 2004/08/10 01:39:22 $

%   Reference:
%      Gonzalez, R.C., R.E. Woods, S.L. Eddins, "Digital Image Processing
%      using MATLAB", Chapter 11.
```

```
[I, h] = ParseInputs(varargin{:});

% Convert to uint8 if not logical and set number of bins for the class.
if islogical(I)
  nbins = 2;
else
  I = im2uint8(I);
  nbins = 256;
end

% Capture original size before padding.
origSize = size(I);

% Pad array.
padSize = (size(h) -1) / 2;
I = padarray(I,padSize,'symmetric','both');
newSize = int32(size(I));  %Cast is necessary for MEX-file.

% Calculate local entropy using MEX-file.
J = entropyfiltmex(I,newSize,h,nbins);

% Append zeros to padSize so that it has the same number of dimensions as the
% padded image.
ndim = ndims(I);
padSize = [padSize zeros(1,(ndim - ndims(padSize)))];

% Extract the "middle" of the result; it should be the same size as
% the input image.
idx = cell(1, ndim);
for k = 1: ndim
  s = size(J,k) - (2*padSize(k));
  first = padSize(k) + 1;
  last = first + s - 1;
  idx{k} = first:last;
end
J = J(idx{:});

if ~isequal(size(J),origSize)
  %should never get here
  eid = sprintf('Images:%s:internalError',mfilename);
```

```
  msg = 'Internal error: J is  not the same size as original image I.';
  error(eid,'%s',msg);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [I,H] = ParseInputs(varargin)

iptchecknargin(1,2,nargin,mfilename);

iptcheckinput(varargin{1},{'uint8','uint16','double','logical'},...
              {'real', 'nonempty','nonsparse'},mfilename, 'I',1);
I = varargin{1};

if nargin == 1
  H = true(9);

else
  H = varargin{2};

  % Check H.
  iptcheckinput(H,{'logical','numeric'},{'nonempty','nonsparse'},mfilename, ...
                'NHOOD',2);

  eid = sprintf('Images:%s:invalidNeighborhood',mfilename);

  % H must contain zeros and or ones.
  bad_elements = (H ~= 0) & (H ~= 1);
  if any(bad_elements(:))
    msg = 'This function expects NHOOD to contain only zeros and/or ones.';
    error(eid,'%s',msg);
  end

  % H's size must be odd (a factor of 2n-1).
  sizeH = size(H);
  if any(floor(sizeH/2) == (sizeH/2))
    msg1 = 'This function expects NHOOD to have a size that ';
    msg2 = 'is odd in each dimension.';
    msg = sprintf('%s\n%s',msg1,msg2);
    error(eid,'%s',msg);
  end
```

```
  % Convert H to a logical array.
  if ~islogical(H)
    H = H ~= 0;;
  end

end
```

C.3 entropy_plot.m

```
function[] = entropy_plot(n,iter,start,types)
% n = total number of images
% iter = breakdown of timesteps
% start = timestep count start value
% types = number of rule types to loop through

for k = 1:3:4
    t = start:iter:n; % time array
    S = zeros(1,length(t)); % entropy

    for i = 1:length(t)
        nm = [num2str(k),'CA',num2str(t(1,i)),'.png'];
        I = imread(nm);
        S(1,i) = entropy(I);
    end

    set(gca,'LineStyleOrder',{'-','--',':','-.'})
    plot(t,S);
    hold all;
end

t = start:iter:99; % time array
S = zeros(1,length(t)); % entropy

for i = 1:length(t)
    nm = [num2str(3),'CA',num2str(t(1,i)),'.png'];
    I = imread(nm);
    S(1,i) = entropy(I);
end

set(gca,'LineStyleOrder',{'-','--',':','-.'})
plot(t,S);
```

```
hold all;

t = start:iter:219; % time array
S = zeros(1,length(t)); % entropy

for i = 1:length(t)
    nm = [num2str(0),'CA',num2str(t(1,i)),'.png'];
    I = imread(nm);
    S(1,i) = entropy(I);
end

set(gca,'LineStyleOrder',{'-','--',':','-.'})
plot(t,S);
hold all;

title('Cellular Automata Entropy Trends');
legend('Class 1','Class 2','Class 3','Class 4');
xlabel('iteration #');
ylabel('entropy');
return;
```

C.4 entropyFiltAv.m

```
function[E_1,E_2,E_3,E_4] = entropyFiltAv()


for k2 = 1:4 % class type
[k2],
    if k2 == 3
        t=0:199; %Longer time run necessary to reach equilibrium
    else
        t=0:99;
    end

    E = zeros(length(t),length(1:8:99)); %Initialization of entropy matrix
    for i=1:length(t)

    % perform this per image
        nm = [num2str(k2),'CA',num2str(1),'CA',num2str(t(1,i)),'.png'];
        I = imread(nm);
        I = mean(I,3); %Bypasses three-dimensionality of RGB matrix created by imread
```

```
        for k=1:8:99;
            NHOOD=ones(k,k); %Defines neighborhood
            %[k,i], %Optional counter output
            E(i,floor(k/8)+1)= mean(mean(entropyfilt(I,NHOOD))); %Computes average entropy

        end

    end
    figure;
    surfc(1:8:99,t,E);
    colormap hsv
end

return;
```