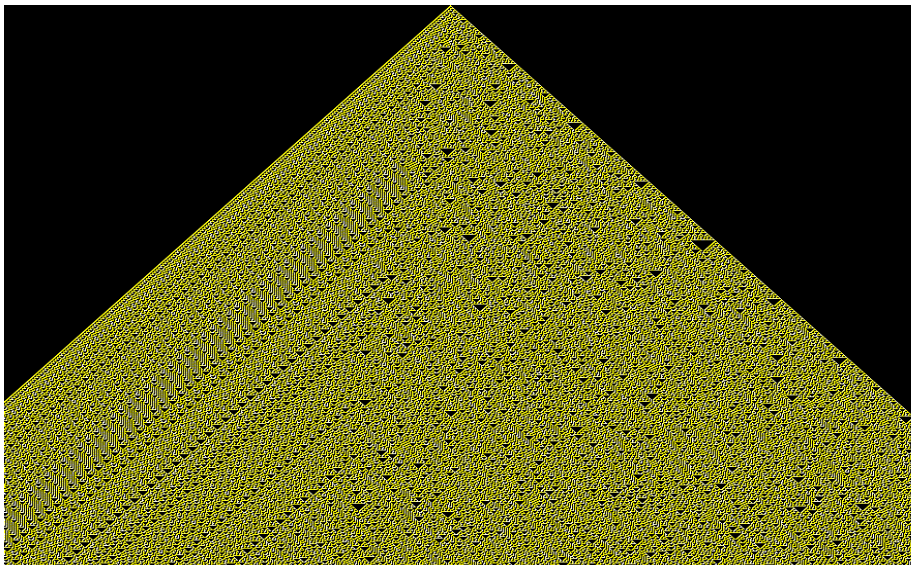
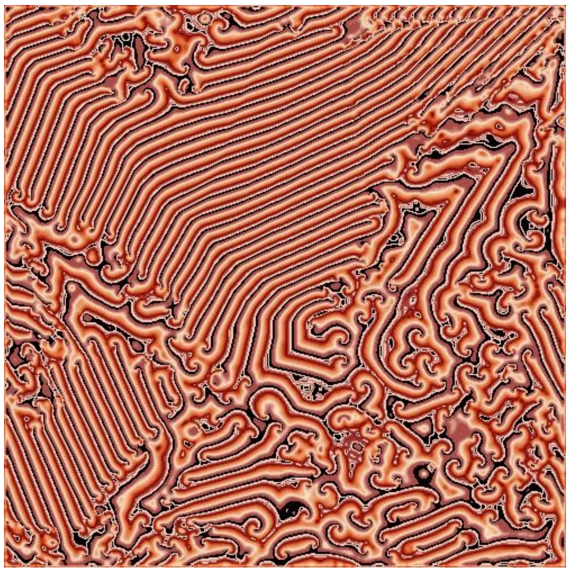
An Investigation into the Randomness and Modeling Potential of Various Cellular Automata

Patrick Neary, Kevin McLeod, Dr. Peterson, Dr. Burkardt
Computational Science and Information Technology, Florida State University





Random Generation

Cellular automata are groups of cells in which each cell's life depends on its surrounding cells. The cell knows what it should be in the next cycle by following simple rules. Most of these rules involve looking at which of its neighbors are alive and which are dead, and also if the cell in question is alive or dead. Many intricate systems can be made with only these simple rules; because of this, cellular automata have many applications in many different fields. These systems have been used as an alternative to differential equations, a simulation of gas behavior, a simulation of crystallization, a simulation of traffic patterns, a model of infection rates, or a generator of random numbers (Rennard 6). It is with the last two applications in which we took interest.

The simplest cellular automata are one-dimensional, where the cells are lined up in a linear universe. If the cell's neighbors are only the two cells directly left and right, then there are a total of 256 different rules to control the life or death of a cell. To see how to easily generate a rule, consider rule 30. First draw the eight possibilities of a cell with its neighbors. If "1" means alive and "0" means dead, these possibilities are 111, 110, 101, 100, 011, 010, 001, and 000. Now write the rule number in binary (30 = 00011110) making sure to fill in extra zeros to the left of the number, for a total of eight digits. Then write each digit of the rule number in binary underneath each of the eight possibilities, in the proper order. So, for rule 30, the neighborhoods 100, 011, 010, and 001 will produce a cell, in the center position in the neighborhood, which is alive. All others will make a dead cell. Each new generation is written under the generation from which it came in order to get a visualization of what exactly is happening.

All of the 256 possible rules can be put into one of four classes. Class I results when a stable, repetitive pattern exists, as in fig. 1. If there are simple, internally repeating structures, such as in fig. 2, it is said to be of Class II. Chaotic, random patterns form class III, as in fig. 3. Finally, Class IV has "structures of substantial spatial and temporal complexity," (Hayes 20) yet is not random. It was Class III rules that we were interested in, because of our interest in the random properties of cellular automata. Thus, we choose rule 30 and 45 to study, since both were reputed to have random properties.

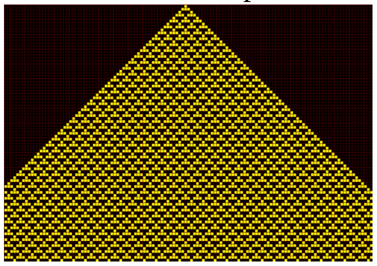


Fig. 1 – Class I: rule 54

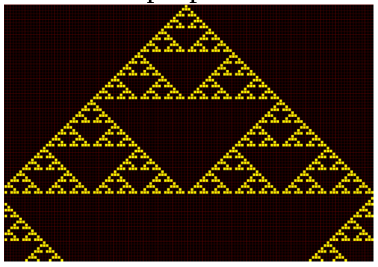


Fig. 2 – Class II: rule 22

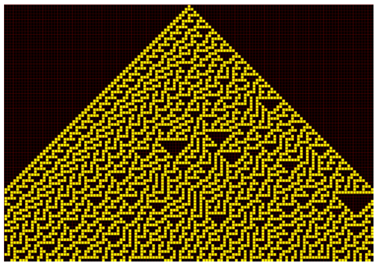


Fig. 3 – Class III: rule 30

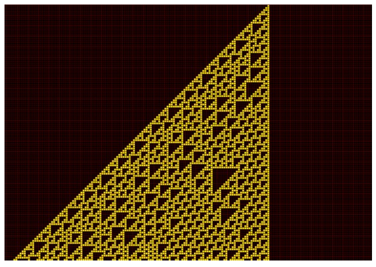


Fig. 4 – Class IV: rule 110

The specific numbers generated came from the states of the cells down the center from the top cell on down. The purpose of only taking this single cell from each row is to avoid any patterns that are seen on the edges of the cellular automata, as is the case with rule 30 (fig. 5). If the cell was alive, we recorded a value of one, and a zero if the cell was dead. However, before we could do this, a number of controls had to be established.

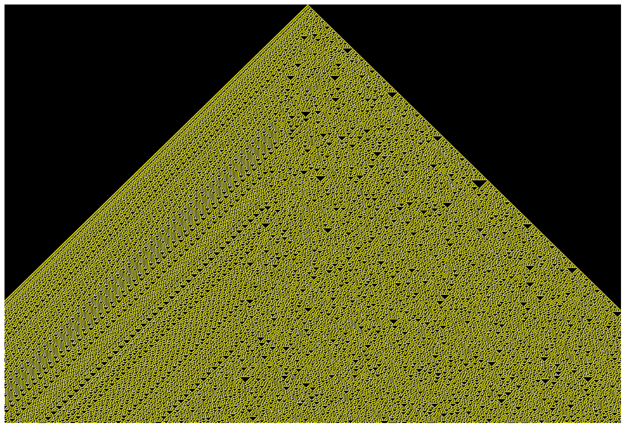


Fig. 5 – Rule 30 with one starting cell

First, we had to decide on the initial condition of the cellular automata. While the original row could be initialized to any collection of dead and living cells, it is more difficult to seed and generate a random assortment of cells than to simply start with one living cell in the middle of a row of dead cells. We then created a program that would generate each new row and store the middle cell's value, either a one or a zero, in a file to be analyzed later. However, a computer would be able to hold only so many cells in its memory, while actual theoretical cellular automata have infinitely long rows, or universes. Thus, we had to arbitrarily limit the size of the universe, which we set at 1000 cells long. This created another variable to be set: whether the sides of the universe should be wrapped around to the other side. When wrapped, the universe would effectively be a circle, while if left unwrapped, the edge cells would never change: they would remain dead forever. We eventually decided to expand the study to also see what happens to the randomness when there is or is not wrapping.

To analyze the numbers, we ran the output through two programs. One program converted the numbers we had generated to a binary file type. The second program, George Marsaglia's Diehard, which runs fifteen tests for randomness, required this format. Diehard would output p-values showing the level of correlation and deterministic qualities of the numbers. These p-values should fall uniformly in the interval [.05, .95], however, due to the extremely large number of p-values returned, some would inevitably be outside this preferred range. The only time a set of numbers should be considered to have truly failed a test is when the p-values returned are zero or one to six or more decimal places.

The results of the randomness tests ran on our data for both rule 30 and rule 45 with wrapping show that the random numbers generated passed twelve out of the fifteen tests. Only the Bitstream test, overlapping pairs sparse occupancy (OPSO) test, and count the one's test returned p-values of one. Overall, the graphs of the p-values show that our random numbers were nominally random (fig. 6 and 7).

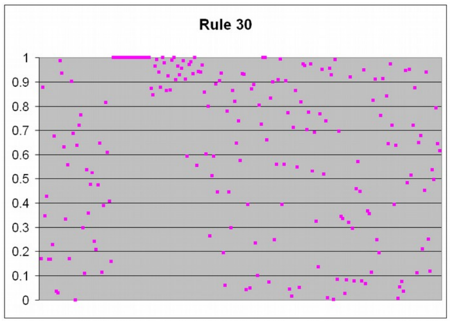


Fig 6 – P-values with wrapping

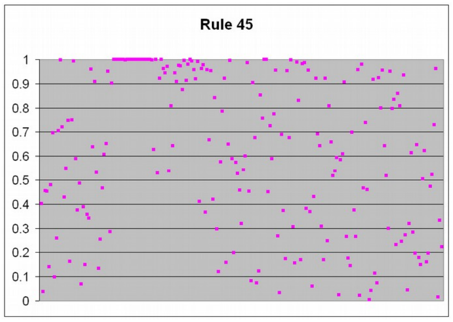


Fig. 7 – P-values with wrapping

The Bitstream test considers twenty-letter words made from only two letters – 0 and 1. The words overlap – the first word is letters 1 through 20, the second 2 through 21 and so on. The test counts the number of missing words (out of a possible 2^20) in a string of 2^21 overlapping words. It should be very close to 141,909. The OPSO test is similar. It counts the number of missing two-letter words. Each letter is determined by ten bits which means there are 2^10 or 1024 possible letters. This should also be close to 141,909. The count the one's test looks at a series of eight numbers. If there are 0, 1, or 2 ones, it is an "A", if there are 3, it is a "B", 4 makes a "C", 5 makes a "D", and 6, 7, or 8 is an "E". The test counts the frequency of five-letter words (a possible 5^5 different words).

We determined that the rules both returned nominally random data by comparing our p-values to those generated by truly random numbers. Thus, we used data from the site random.org, which uses atmospheric noise to generate random numbers. We graphed the resulting p-values and found that our p-values were not as evenly spread as random.org's p-values (fig. 8), nor were our p-

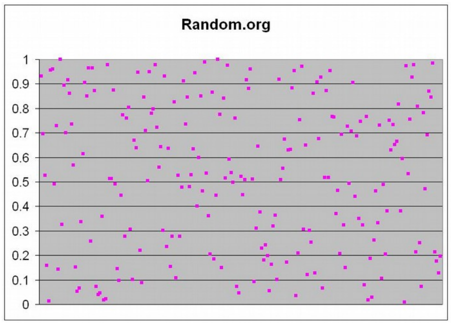


Fig. 8 – P-values for data obtained on 7/17/2003

values in the desired range of .05 to .95 as often as random.org's p-values: our p-values were outside the range 28.6730 percent of the time, while random.org's were outside the range only 13.2701 percent of the time. Thus, we have determined that these two rules are not true sources of randomness. We then went on to determine if these two rules were any better than a standard pseudo-random number generator. Our p-values show that both rules are better, by a significant level, than the standard pseudo-random number generators, as shown by our graphs of the C++ rand() function's p-values, as shown by fig. 9.

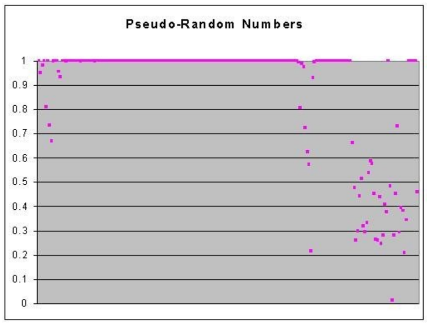


Fig. 9 – P-values for C++ rand() function

We have also found that rule 30 is somewhat better at making random numbers than rule 45. While the difference is slight when looking at the spread of p-values on a graph, rule 45 has 16.0714 percent more p-values outside of the acceptable range than does rule 30.

A very interesting result came from the study of wrapping versus not wrapping the universe. We determined that wrapping the sides around does result in a far greater level of randomness. When the end values of each line are not altered, the numbers generated are in no way random: every p-value was one to more than six decimal places. It may be hypothesized that the non-random tendencies of the edge cells are stronger than the random tendencies coming from the center cells.

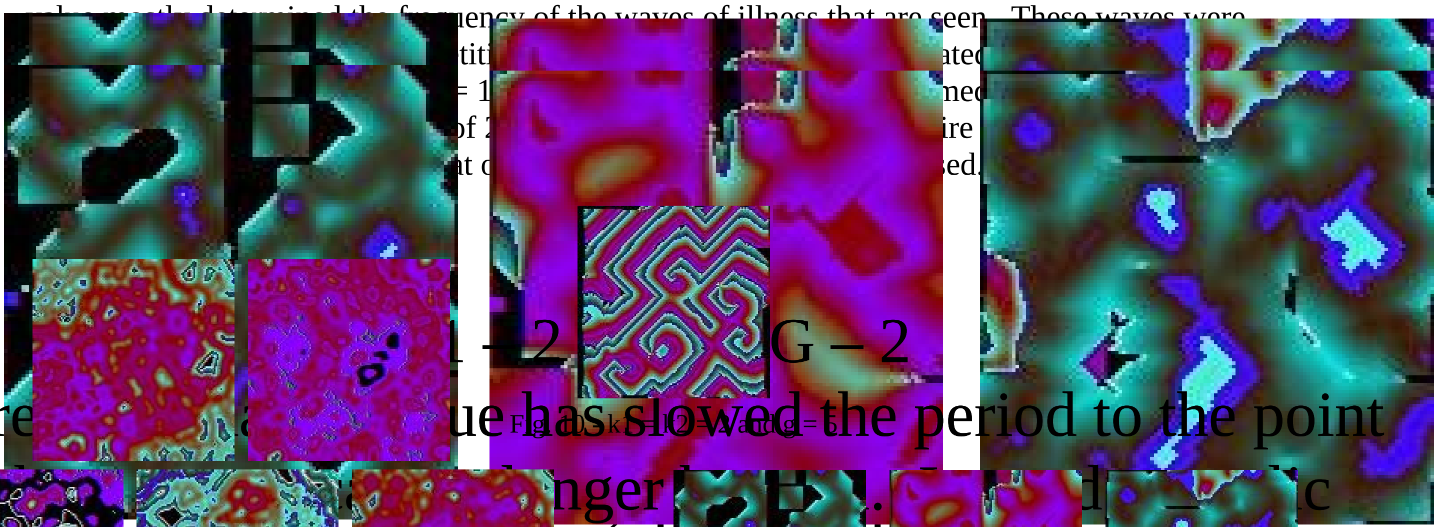
From this research, one obvious application arises: generating random numbers in programs and for other computer-based purposes. While, in many cases, using truly random numbers would be a preferable option, it is not always easy to get access to them. In many cases, external sources of randomness must be found, such as atmospheric noise in the case of random.org. For many users of random numbers, to generate their numbers in such a way would be cumbersome and expensive. Thus, cellular automata could be used. They are completely computer-based, just like other pseudo-random number generators, and yet, they are better than most standard pseudo-random number generators.

Hodgepodge Machine

Cellular automata do not have to be one dimensional, but any dimension desired. The higher-dimension cellular automata we studied for its potential to model illness was the hodgepodge machine. This is also an example of cellular automata in which there are many different states, not just alive or dead, due to the varying degrees of sickness a cell a cell can be. Much like in the one-dimensional cellular automata we studied earlier, a cell is determined to be at a certain state by the states of its neighbors. In this two dimensional example, the cell's neighborhood is the eight cells touching it.

This simulation of disease and infection rates uses a special set of rules that determine, from generation to generation, how sick the cell becomes. The formula $\left\lfloor \frac{A}{k1} \right\rfloor + \left\lfloor \frac{B}{k2} \right\rfloor$ is used to determine what state *n* a healthy cell will become, where A and B are the numbers of infected and sick cells, respectively. The brackets mean to take the greatest integer less than or equal to the ratio in the brackets. The formula $\left\lfloor \frac{S}{A} \right\rfloor + g$ is used to determine the state of an infected cell, where S is the sum of the infected neighbors and A is the number of infected cells. When a cell reaches state *n*, it becomes healthy in the next generation. If at any time a formula would put the cell in a state higher than *n*, the state is simply put into the *n* state (Dewdney 104). We studied generally how the parameters k1, k2, and g affect the infection rates and fluctuations in the number of infected, sick, and healthy cells, because we found ourselves short on time.

To do this, we used a C program called hodge that would allow us to change the parameter values. From this, we were able to determine that the k values mostly determined whether the cells would become sick or not and were responsible for creating long-term cyclical patterns which did not actually precisely repeat, but created what appeared to be random patterns. Unfortunately, we were unable to test these patterns for randomness. At high k values, such as k1 = k2 = 5, every cell eventually become healthy. At very low k-values, such as k1 = k2 = 1, the cells would become sick fast enough that their behavior triggered a cyclical pattern of a very high period of about 8000. From this program, we were also able to determine that the g values mostly determined the frequency of the waves of illness that are seen. These waves were titi ated = 1 med of 2 ire at c sed.



1 – 2 G – 2

Fig. 10 – k1 = k2 = 2 and g = 5

Here the G value has slowed the period to the point where waves can no longer be seen. Instead, a cyclic universe is seen much like when the k parameters were low. There are about 1 generations between each pictures.

Here the G value has slowed the period to the point where waves can no longer be seen. Instead, a cyclic universe is seen much like when the k parameters were low. There are about 1 generations between each pictures.

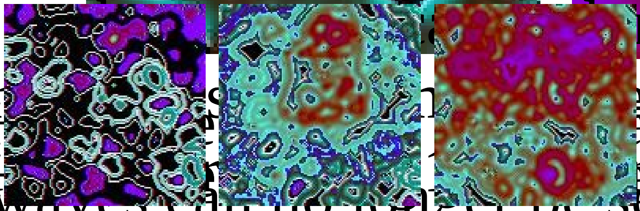


Fig. 11 – k1 = k2 = 1 and g = 10

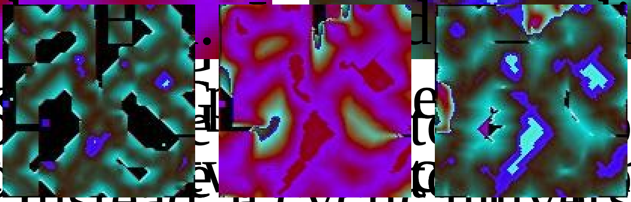


Fig. 12 - k1 = k2 = 2 and g = 2

Unfortunately, we were unable to study the hodgepodge machine any further, due to a lack of time. However, from the limited study we were able to do, we determined that the hodgepodge machine would be able to model, at least to some extent, the spreading factors and behaviors of an illness.

Cited Works:

Dewdney, A. K. *Computer Recreations: The Hodgepodge Machine Makes Waves*. Scientific American: 1988.

Hayes, Brian. *Computer Recreations: The Cellular Automaton Offers a Model of the World and a World Unto Itself.* Scientific American: 1984.

Rennard, Jean-Philippe Ph.D. *Introduction to Cellular Automata*. Rennard.org, accessed 2003.