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

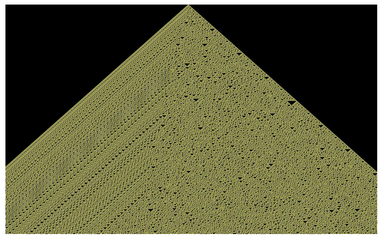


Fig. 1 – Rule 30 with one starting cell

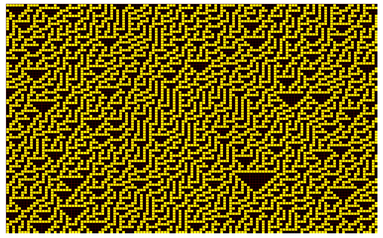


Fig 2 – Close up of a middle section of rule 30

```
for(unsigned int ctr = 0; ctr < num; ctr++)
    {
        for(int ctr2 = 1; ctr2 < 999; ctr2++)
        {
                temp = 100 * a[0][ctr2 - 1] + 10 * a[0][ctr2] + a[0]
[ctr2 + 1];
        if(temp == 0 || temp == 10 || temp == 11 || temp == 101)
                a[1][ctr2] = 1;
                else
                        a[1][ctr2] = 0;
        }
    file_out << a[1][500];
    for(int ctr2 = 0; ctr2 < 1000; ctr2++)
    a[0][ctr2] = a[1][ctr2];
}
```

Fig 3 – The code that generates the random numbers

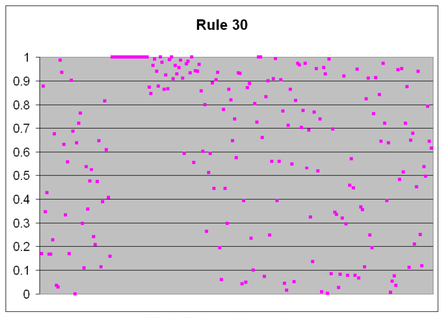


Fig 4 – P-values with wrapping

Fig 5 – P-values without wrapping

Introduction

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 that look at which of its neighbors are alive and which are dead, and also if itself is alive or dead. Many intricate systems can be made with only these simple rules.

The cellular automata used to generate random numbers was one dimensional. Each cell only looked at itself and two neighbors – the cells directly to the left and right. Each new generation of cells is shown underneath the generation that spawned it.

Cellular automata do not have to be one dimensional, as hodgepodge shows. This is an example of cellular automata in which there are many different states, not just alive or dead. The cell can have varying degrees of sickness. 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.

Method

To generate the random numbers, we decided to start with an initial condition of one cell alive in the middle of a row of dead cells, because that would simplify the initialization process. 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 (fig. 3). These middle cells' values would be the random numbers. We chose to only take this single number from each row because the middle value was guaranteed to be more random than other values towards the edges, which, as can be seen in fig. 1 and 2, are less random than the inner cells.

To analyze the numbers, we ran the output through two programs. One program converted the numbers we had generated and stored in a text file into integers stored in a binary file. This format was required by George Marsaglia's program Diehard, which runs fifteen tests for randomness. After running that program, we had the p-values necessary to determine the randomness of our numbers.

Results

The results of the randomness tests ran on our data for both rule 30 and rule 45 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.

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).

P-values should be uniform from 0 to 1. However, the tests used said not to be worried with p-values close to 0 or 1 such as .0012 or .9983. If a test really fails, the p-values will be 0 or 1 to more than six decimal places.

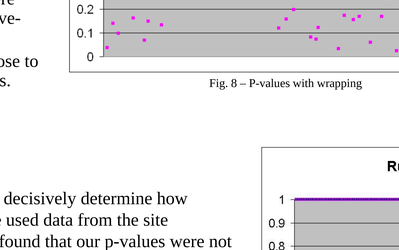


Fig 6 – Rule 45 with one starting cell

Fig 7 – Close up of a middle section of rule 45

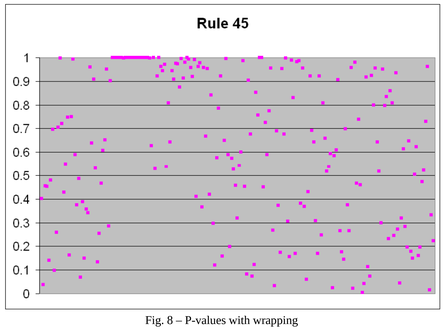


Fig. 8 – P-values with wrapping

Conclusion

From the charts of the p-values, we determined that both rules returned nominally random data. To see more decisively determine how random our numbers were, we compared our p-values to those generated by a truly random set of data. 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, nor were our p-values in the desired range of .05 to .95 as often as random.org's p-values. 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. Both rules are better, by a significant level, than the standard psuedo-random number generator, as shown by our graphs of the C++ rand() function's p-values.

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 20 percent more p-values outside of the acceptable range than does rule 30.

Finally, <<PUT WRAPPING CONCLUSION HERE>>

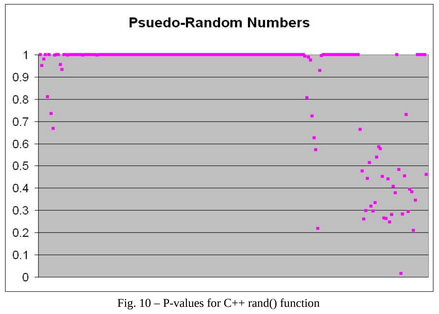


Fig. 10 – P-values for C++ rand() function

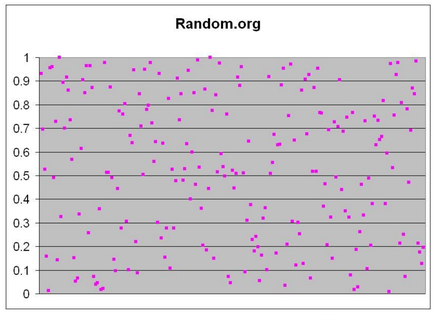
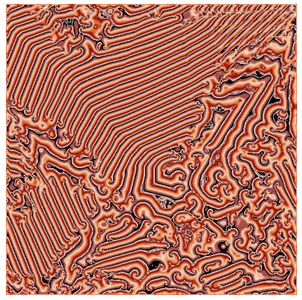


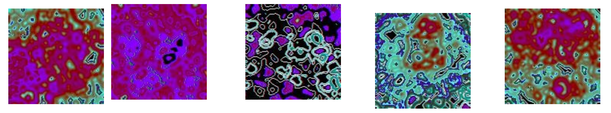
Fig. 11 – P-values for data obtained on 7/17/2003

Fig 9 – P-values without wrapping

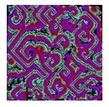


The uses of 2 dimensional cellular automata can be even more extensive than the one dimensional ones we looked at first. The example we looked at, often called the hodgepodge machine, simulates disease and infection rates. It uses a special set of rules that determine, from generation to generation, how sick the cell becomes: the formula $\lfloor \frac{A}{k1} \rfloor + \lfloor \frac{B}{k2} \rfloor$ is used to determine what state *n* a healthy cell will become, where A and B are the numbers of infected and sick cells. The formula $\lfloor \frac{S}{A} \rfloor + g$ 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. 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.

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. 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 value mostly determined the frequency of the waves of illness that are seen. High g values, such as g = 10, create fast, tight waves, while a medium value of 5 creates slower waves, while a low value of 2 creates waves so slow that the entire universe becomes cyclical, appearing much like what occurred when low k values were used.

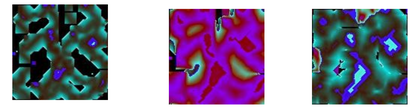


K1 – 1 K2 – 1 G – 10



K1 – 2 K2 – 2 G – 10

K1 – 5 K2 – 5 G – 10



K1 – 2 K2 – 2 G – 2

K1 – 2 K2 – 2 G - 5