Increasing the Size of the Potential Neighborhood in the Elementary Cellular Automata: A Case Study

Justin Gabrielle A. Manay[1]

[1] *De La Salle University - Manila*

Abstract

In this paper, we aim to (1) implement the elementary cellular automata in Python, (2) characterize the rules using measures like kinetic energy and spatial entropy and (3) introduce a modification to the rule to determine how it changes the ECA rules based on the characterization measures previously introduced. We find that increasing the size of potential neighborhood increases the spatial entropy for most rules but has no definitive effect on kinetic energy. We posit that this is because increasing the size of the neighborhood injects new information into the matrix, as each cell now has 4 new possible states, instead of one.

Keywords: elementary cellular automata, complex systems

1 Introduction

An *elementary cellular automaton (ECA)* has two possible states (1 or 0) for each cell and the state in the succeeding step is determined by rules concerning each cell's immediate neighbors. There are 256 such rules, and they are determined as follows:

For each triplet, the middle cell will change depending on its surrounding cells. There are $2^3 = 8$ possible configurations, and for each configuration, either 0 or 1 is assigned, thus determining what the middle cell changes to in the next step. For example, given the following pattern to rule mapping,

Table 1: Pattern-to-rule mapping for ECA Rule 30

pattern	111	110	101	100	011	010	001	000
rule	0	0	0	1	1	1	1	0

a 1 surrounded by two 1's changes to 0 in the next time step, a 1 surrounded by a 1 to the left and a 0 to the right changes to 0 in the next time step, etc. The binary number resulting from the rule (in this case, 11110_2) defines the ECA rule; in this case, the above ECA rule is ECA 30.

When plotted in a matrix where cells with 1's are black and cells with 0's are white, the ECA rules form distinct patterns. For example, Figure 1 represents the pattern formed for ECA rule 30, assuming that the initial state is randomized. In particular, each row in the matrix represents the spatial dimension, while each column represents the temporal dimension.

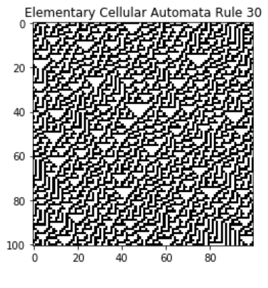

Figure 1: 100 x 101 plot for ECA rule 30, assuming a random initial state

In this paper, we aim to (1) implement the elementary cellular automata in Python, (2) characterize the rules using measures like kinetic energy and spatial entropy and (3) introduce a modification to the rule to determine how it changes the ECA rules based on the characterization measures previously introduced.

2 Coding the Elementary Cellular Automaton

In this section, we implement the elementary cellular automaton using Python.

We begin by assuming that the user will input the rule number. Thus, the program should start with a function that converts decimal numbers to binary (`decimal_to_binary`). The user specifies which ECA rule to output, so the program converts the rule number to a binary number using this function to determine the rule to follow. The function outputs the binary number, represented as a string. Because the pattern-to-rule mapping requires eight digits, we add as much zeroes as needed to produce the required number of digits.

```
4   def decimal_to_binary(num):
5
6       digits = []
7
8       while num >= 1:
9           digits.append(num % 2)
10          num = num // 2
11
12      if len(digits) < 8:
13          digits = [0 for i in range(8 - len(digits))] + digits[::-1]
14      else:
15          digits = digits[::-1]
16
17      return "".join(map(str, digits))
```

After this, the function `eca` goes from one step to the next by using the rule specified by the user. It takes in an initial list `list_init` (list of numbers at the first step) and the rule number `rule`. Using the earlier function `decimal_to_binary`, it converts the rule number to binary, which is then mapped to the different triplets using `rule_dict`. It then uses the rule to append the right numbers to the list in the next time step (`list_new`). `list_init` is treated as cyclic, in which case, we modify `list_init` to append the last element to the first part and the first element to the last part. We can also choose to ignore endpoints, in which case we simply uncomment line 32 of the code below.

```
19  def eca(list_init, rule):
20
21      list_new = []
22
23      rule = decimal_to_binary(rule)
24      rule_dict = {"111" : rule[0], "110" : rule[1], "101" : rule[2],
"100" : rule[3], "011" : rule[4], "010" : rule[5], "001" : rule[6], "000" :
rule[7]}
25
26  #    Treat list_init as cyclic
27      list_init = list_init[-1 : ] + list_init + list_init[0 : 1]
28
29      for i in range(len(list_init) - 2):
30          list_new.append(int(rule_dict["".join(map(str, list_init[i : i +
3]))]))
31
```

```
32      #list_new = list_init[0 : 1] + list_new + list_init[-1 : ]
33
34      return list_new
```

The function `eca_matrix` is now used to apply the rule to lists for all time steps. By default, there will be 101 cells per list, and 100 rows in total. The initial list is generated at random, but to check, the code also allows for an initial list where the middle cell is shaded (at line 39). For each row, the function `eca` is then applied to the preceding list, given the specified rule. This function returns the matrix `eca_matrix` which we will then use for the plot and for the computation of statistical measures in a later section.

```
36  def eca_matrix(rule, length = 101, width = 100):
37      eca_matrix = [[0 for i in range(width)] for j in range(length)]
38
39  #     eca_matrix[0] = [0 for i in range(width // 2)] + [1] + [0 for i in
range(width // 2)]
40
41  #     Randomize list_init
42      np.random.seed(100)
43      eca_matrix[0] = [np.random.randint(0, 2) for i in range(width)]
44
45      for i in range(1, length):
46          eca_matrix[i] = eca(eca_matrix[i - 1], rule)
47
48      return eca_matrix
```

We now use the function `eca_plot` to plot the results of `eca_matrix` in a colormap, where 0 corresponds to the white spaces and 1 corresponds to the dark spaces. The function also asks for `rule` as an argument to properly annotate the figure. We use `eca_plot` to generate the plots for all 256 rules, as shown in Figure 2.

```
82  def eca_plot(matrix, rule):
83
84      plt.imshow(matrix, vmin = 0, vmax = 1, cmap = "binary")
85      plt.title('Elementary Cellular Automata Rule ' + str(rule))
86      plt.show()
```

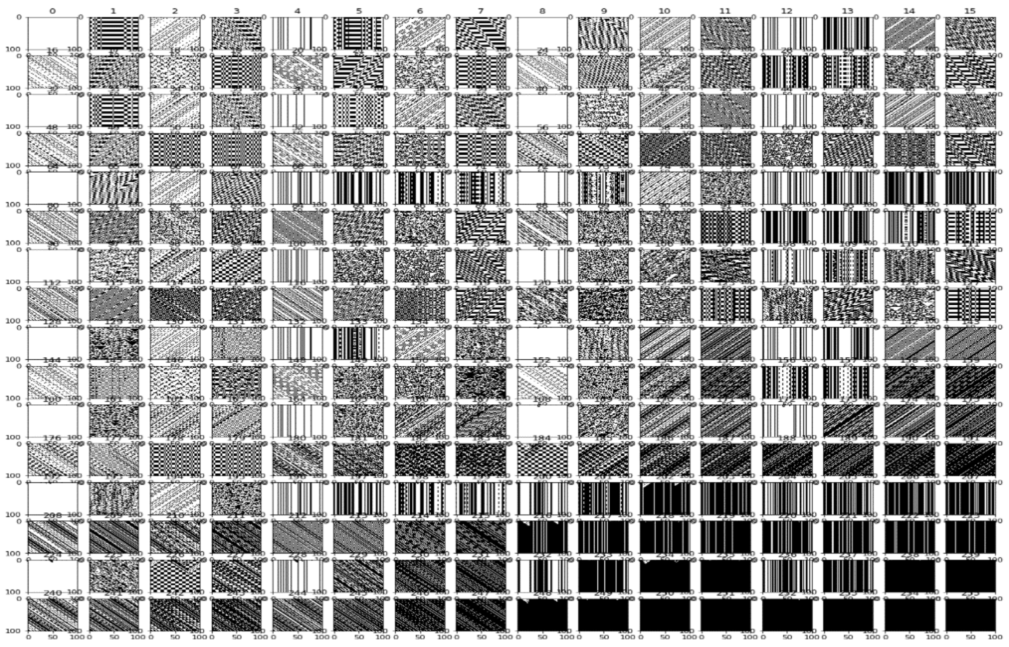

Figure 2: Plot of all 256 ECA rules

3 Characterizing ECA Rules

We use two measures to quantify the spatial and temporal change in the matrix due to the rule: kinetic energy and spatial entropy.

Kinetic energy measures how many times a site has flipped over the course of an iteration and represents the degree of temporal change in an ECA rule. This is given by:

$$K = \frac{1}{L}\sum_{i=1}^{L}\left(\frac{1}{T}\sum_{t=1}^{T}\left|s_i^t - s_i^{t-1}\right|\right) \tag{1}$$

The term inside the inner summation represents the number of times a site has flipped over the entire time T (i.e., across an entire row). We then add this up for all space and average it over the entire one-dimensional grid (i.e., we average it over all rows). In general, a higher K value means that the rows in the rule matrix frequently change. Meanwhile, lower K values mean that the rows are not changing as much.

On the other hand, *spatial entropy* measures how many configurations can be found over time and reresents the degree of spatial change in an ECA rule. This is given by:

$$S = \frac{1}{T}\sum_{t=0}^{T}\left(-C\sum_{n=000}^{111} p_n \ln(p_n)\right)_t \tag{2}$$

The term inside the inner summation represents the Shannon entropy for all possible triplets n at a given time t. To compute for this, for a given time (or for a given row), we compute for p_n by taking the frequency of each possible three-cell combination of occurring, and dividing it by the number of such three-cell combinations in the row. We then take $p_n \ln(p_n)$ and add them up for all eight three-cell combinations. C simply represents a normalization constant. In this case, since we will not be normalizing, we set $C = 1$. We then average over all time.

In general, a higher S value means that the rows in the rule matrix are information-rich. That is, there is a lot of randomness present in the rows, and all possible three-cell combinations are represented almost equally. Meanwhile, a lower S value signifies that the rows tend to be uniform and not as information-dense.

We compute the kinetic energy and spatial entropy for each row using the functions `kinetic_energy` and `spatial_entropy`.

For `kinetic_energy`, we run through each pair of consecutive rows and count the total number of flips, storing the result in `total_flips`. We then divide by the length of the column and add the result to `total_flips_ave`. We do this for all possible pair of consecutive rows and finally divide the result by the length of each row to get the kinetic energy.

```
# ECA Characterization

def kinetic_energy(matrix):

    total_flips_ave = 0

    for row in range(1, len(matrix)):

        total_flips = 0
        for column in range(len(matrix[row])):
            total_flips += abs(matrix[row][column] - matrix[row - 1][column])
        total_flips_ave += total_flips / len(matrix)

    kinetic_energy = total_flips_ave / len(matrix[0])

    return kinetic_energy

def spatial_entropy(matrix):

#     length = len(matrix[0])
#     norm_const = (length / 8) * np.log(1 / length)
    norm_const = 1

    spatial_entropy = 0

    for row in range(len(matrix) - 1):

        shannon_entropy = 0
        prob_dict = {"111" : 0, "110" : 0, "101" : 0, "100" : 0, "011" : 0, "010" : 0, "001" : 0, "000" : 0}
        cyclic_row = matrix[row][-1 : ] + matrix[row] + matrix[row][0 : 1]

        for column in range(len(cyclic_row) - 2):
            triplet = "".join(map(str, cyclic_row[column : column + 3]))
            prob_dict[triplet] += 1 / len(matrix[0])

        for prob in prob_dict.values():
            if prob == 0:
                continue
            else:
                shannon_entropy += prob * np.log(prob)

        spatial_entropy += -norm_const * shannon_entropy / len(matrix)

    return spatial_entropy
```

Meanwhile, for `spatial_entropy`, we run through each row. For each row, we run through three columns at a time to get the string `triplet`. Whenver we see this string, we add 1 over the length of the row to its value in `prob_dict`. After having done this for the entire row, we should now have the probabilities for each possible triplet. We then compute for the inner summation in equation (2). Note that if the probability = 0, we do not consider it in our computation, since ln(0) will throw an error. We then do this for all rows, multiply by the normalization constant and divide by the length of each column to get the spatial entropy.

We implement this for all 256 rules, leading to the plot in Figure 3.

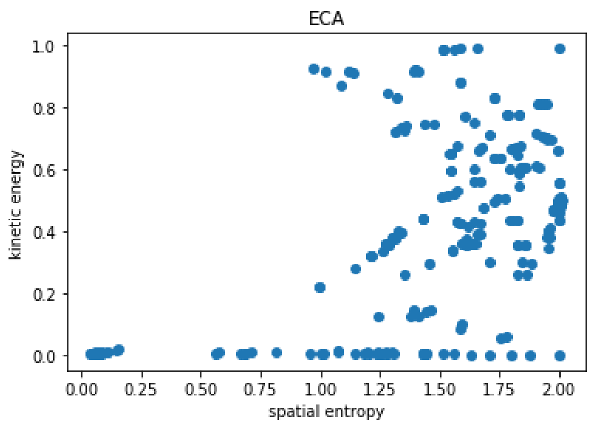

Figure 3: Kinetic energy vs. spatial entropy for all ECA rules

We will compare this plot to other such plots when we modify the ECA in the succeeding section.

4 Modifying the ECA

In the elementary cellular automaton (ECA), each cell determines its state based on its immediate neighbors' states. For this modification, we can increase the scope of the neighborhood in consideration by also taking into account the cells which are two steps to the left and right. We can employ a rule which determines a cell's state based on its four nearest neighbors; however, such a rule would have $2^5 = 32$ unique "quintets." For our convenience, we would like to stick to triplets so as to be able to compare it with the default ECA.

Thus, we can make the cell employ a bias towards its immediate neighbors. Of the two cells to its left and right, the cell in question only chooses one of each. However, it will have a certain bias or preference towards its immediate neighbors.

To implement this, we use the following code:

```python
# MODIFICATION: Choose between one and two steps to the left and between one and two steps to the right with bias bias, towards the immediate neighbors.
def eca_modified(list_init, rule, bias):

    list_new = []

    rule = decimal_to_binary(rule)
    rule_dict = {"111" : rule[0], "110" : rule[1], "101" : rule[2], "100" : rule[3], "011" : rule[4], "010" : rule[5], "001" : rule[6], "000" : rule[7]}

#    Treat list_init as cyclic
    list_init = list_init[len(list_init) - 2 : ] + list_init + list_init[0 : 2]

    for i in range(2, len(list_init) - 2):
        before = [list_init[i - 2], list_init[i - 1]]
        after = [list_init[i + 2], list_init[i + 1]]
        list_new.append(int(rule_dict["".join(map(str, [before[np.random.binomial(1, bias)], list_init[i], after[np.random.binomial(1, bias)]]))]))

    #list_new = list_init[0 : 2] + list_new + list_init[len(list_init) - 2 : ]

    return list_new

def eca_modified_matrix(rule, bias, length = 101, width = 100):
    eca_matrix = [[0 for i in range(width)] for j in range(length)]

#    eca_matrix[0] = [0 for i in range(width // 2)] + [1] + [0 for i in range(width // 2)]

#    Randomize list_init
    np.random.seed(100)
    eca_matrix[0] = [np.random.randint(0, 2) for i in range(width)]
```

Because we now also consider the cells which are two steps to the cell in question's left and right, we now append the last two and first two cells to the left end and right end, respectively, to make the list cyclic. Moreover, we define two new variables `before` and `after`, which give the cells before and after the cell in question which are to be considered to determine the cell's state. We use `np.random.binomial` to choose between 0 and 1, whose probability of success/probability of choosing 1 over 0 is determined by the variable bias. Thus, bias gives us the probability of the cell choosing its immediate neighbor over its neighbor two steps to the left/right. We also use a new function eca_modified_matrix to generate the necessary matrix.

To characterize the rules in the modified ECA, we use the measures discussed in Part 3. We then create a spatial entropy vs. kinetic energy plot for all 256 new rules for varying bias values. The code below is used to generate these plots where the bias values go from 0.1 to 0.9 in multiples of 0.1.

```
for i in range(len(bias_list)):
    kinetic_energy_modified_vals = []
    spatial_entropy_modified_vals = []
    for j in range(256):

kinetic_energy_modified_vals.append(kinetic_energy(eca_modified_matrix(j,
bias_list[i])))

spatial_entropy_modified_vals.append(spatial_entropy(eca_modified_matrix(j,
bias_list[i])))

    row_num = i // 3
    col_num = i % 3
    ax[row_num, col_num].plot(spatial_entropy_modified_vals,
kinetic_energy_modified_vals, ls = "", marker = "o")
    ax[row_num, col_num].set_title("Modified ECA, Bias: " +
str(bias_list[i]))

plt.show()
```

The plots that are generated are shown below in Figure 4.

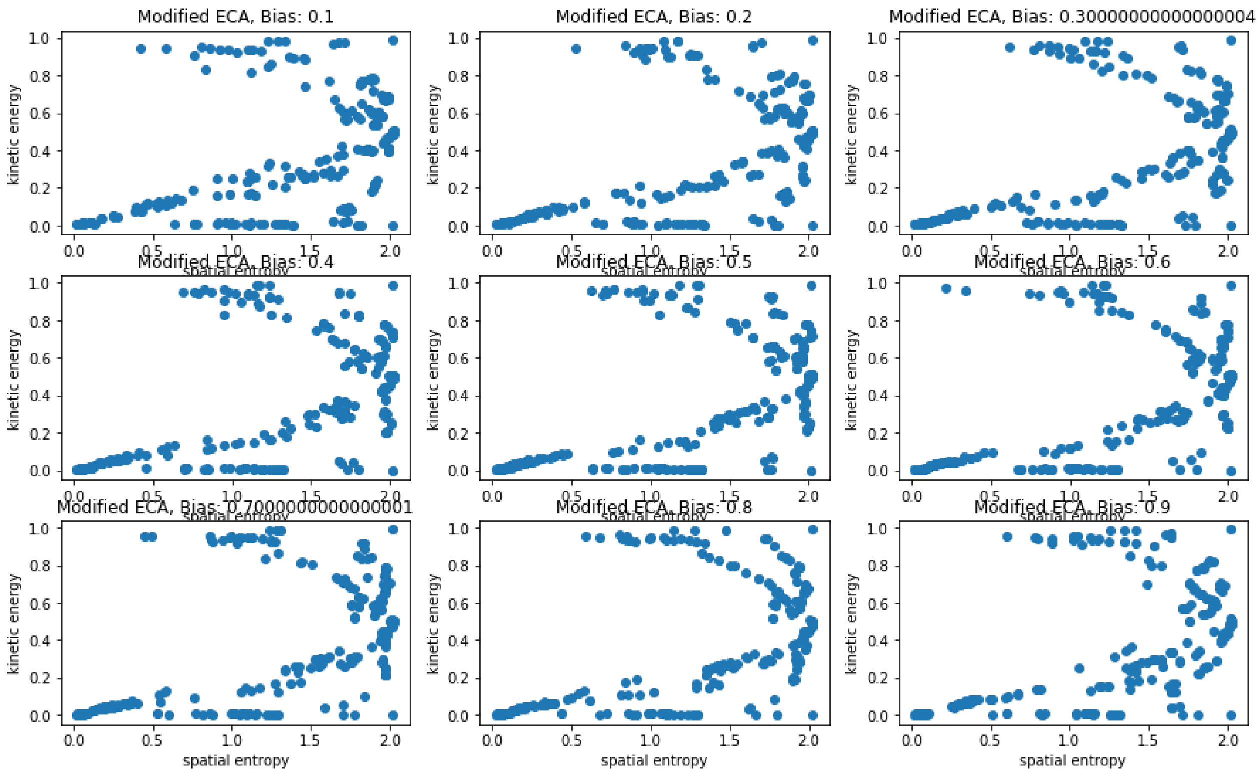

Figure 4: Spatial entropy vs. kinetic energy plot for ECA for different bias values

Notice that as we start introducing a bias towards its immediate neighbors, the ECA plots start to become sharper towards the center right. The points in the center right get pushed, so that some points get pushed downwards (lower spatial entropy and lower kinetic energy) while some points get pushed upwards (lower spatial entropy and higher kinetic energy) to form a parabola-like shape.

A band also forms on the edge of the right side of the plot, indicating that we now have significantly more points for which spatial entropy is high but kinetic energy is low, unlike before. Meanwhile, certain groups of points such as the ones on the edges of the ECA graph from Figure 3 remain the same even as we introduce biasing. These points include points on the lower left (low kinetic energy, low spatial entropy), some points on the lower right (low kinetic energy, high spatial entropy) and a point on the upper right (high kinetic energy, high spatial entropy). After some time, the sharpness in the center right disintegrates until `bias` = 1, represeting the default ECA configuration.

Among these plots, we will compare the plot of the ECA when `bias` = 0.5 (that is, when it considers both neighbors equally) to the default ECA. Both plots are shown side-by-side in Figure 5.

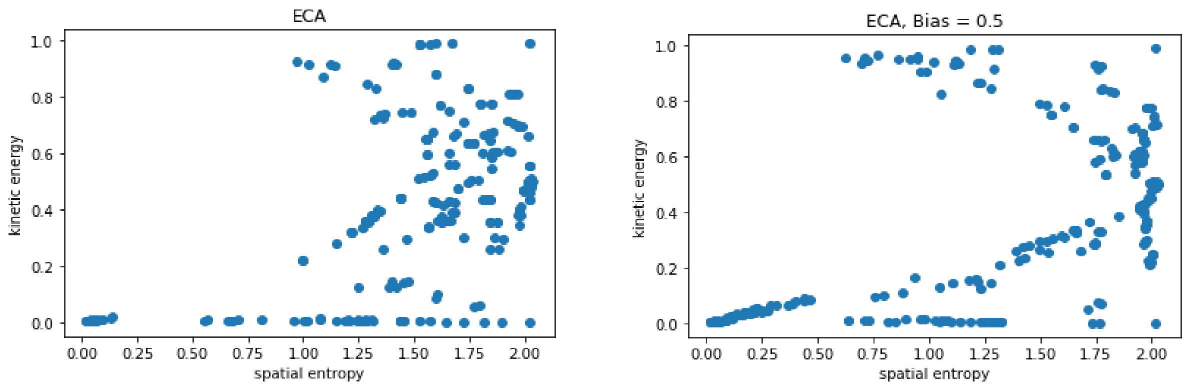

Figure 5: Spatial entropy vs. kinetic energy plot for default ECA and for ECA when bias = 0.5

Using the code below,

```
kinetic_energy_vals = []
spatial_entropy_vals = []
kinetic_energy_vals_equal = []
spatial_entropy_vals_equal = []

for i in range(256):
    kinetic_energy_vals.append(kinetic_energy(eca_matrix(i)))
    spatial_entropy_vals.append(spatial_entropy(eca_matrix(i)))
kinetic_energy_vals_equal.append(kinetic_energy(eca_modified_matrix(i, 0.5)))
spatial_entropy_vals_equal.append(spatial_entropy(eca_modified_matrix(i, 0.5)))

plt.plot(spatial_entropy_vals, kinetic_energy_vals, ls = "", marker = "o")
plt.title("ECA")
plt.xlabel("spatial entropy")
plt.ylabel("kinetic energy")
plt.show()

plt.plot(spatial_entropy_vals_equal, kinetic_energy_vals_equal, ls = "", marker = "o")
plt.title("ECA, Bias = 0.5")
plt.xlabel("spatial entropy")
plt.ylabel("kinetic energy")
plt.show()
```

```

for i in range(256):
    kinetic_energy_diff = [kinetic_energy_vals[i] - kinetic_energy_vals_equal[i] for i in range(256)]
    spatial_entropy_diff = [spatial_entropy_vals[i] - spatial_entropy_vals_equal[i] for i in range(256)]

plt.plot(spatial_entropy_diff, kinetic_energy_diff, ls = "", marker = "o")
plt.title("ECA Default and Modified Rule Differences")
plt.xlabel("spatial entropy")
plt.ylabel("kinetic energy")
plt.show()
```

we monitor the difference in all the rules by computing for `kinetic_energy_diff` and `spatial_entropy_diff`, which we then plot as before, resulting in the plot in Figure 6.

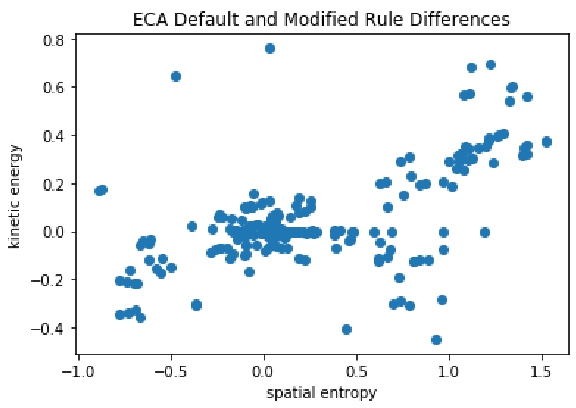

Figure 6: Differences in the default and modified ECA by kinetic energy and spatial entropy

We detail the changes from the default ECA below:

i) Some of the rules experienced very minimal changes in both quantities, resulting in a cluster of rules by the (0, 0) point in the plot.

We expect this to happen to the cluster for which both kinetic energy and spatial entropy are low, since for a majority of these rules, the matrix ends up being a matrix of 1's or 0's at some point in time. Thus, there would be no change in kinetic energy or in spatial entropy even if we were to sample from a larger potential neighborhood.

This also happens to a point in the upper right for which kinetic energy and spatial entropy are at their maximum. The plot for this rule is dynamic, and is characterized by high information density and a large tendency to change. Already being very information dense and very prone to change, a larger potential neighborhood did not change its spatial entropy and kinetic energy at all.

ii) There is a cluster of points for which both kinetic energy and spatial entropy decrease. In Figure 5, this represents points which are pushed to the left and downward in the plot. This represents a decrease in both the information density and the frequency of change.

Notably, this happened to points to the lower right in the center right portion of the original ECA plot. These had relatively low kinetic energy values and relatively high spatial entropy values, representing rules where the plots are information dense initially but do not change that much.

Introducing a larger potential neighborhood decreased information density somewhere along the way (e..g., by making certain triplets more commonly occurring and others less so) and the low kinetic energy simply reinforced this.

iii) On the other hand, there is also a small cluster of points for which spatial entropy increased but kinetic energy decreased. In Figure 5, this represents points which are pushed to the left and upward in the plot. This represents an increase in the information density but a decrease in the frequency of change.

iv) There are a group of points for which both kinetic energy and spatial entropy increased. This represents an increase in both the information density and the frequency of change.

For clusters (iii) and (iv), the explanations are roughly similar to that in (ii).

v) In general, however, there is now a band of rules for which spatial entropy is high. By sampling from a larger potential neighborhood, it is possible that we have injected new information to some of these plots because each cell now has $2^2 = 4$ possible new states, making them more information dense.

5 Conclusions

Increasing the size of the potential neighborhood for ECA does not always guarantee an increase in spatial entropy. We observed instances in which it actually decreased spatial entropy, possibly because certain triplets more commonly occurring and others less so, decreasing information density. For most cases, however, it either increases spatial entropy, however negligible.

As for the kinetic energy, increasing the size of the potential neighborhood for ECA does not seem to have a definitive effect for most cases. It seems to have increased the kinetic energy for some, decreased it for others and not have any effect on some others.

References

[1] Wolfram, S. (2002). A new kind of science (Vol. 5, p. 130). Champaign, IL: Wolfram Media.
[2] Batac, R. (2018). Complexity in Discrete Models: Cellular Automata [Powerpoint slides].

Appendix

<u>Code</u>

```
import numpy as np
import matplotlib.pyplot as plt

def decimal_to_binary(num):

    digits = []

    while num >= 1:
        digits.append(num % 2)
        num = num // 2

    if len(digits) < 8:
        digits = [0 for i in range(8 - len(digits))] + digits[::-1]
    else:
        digits = digits[::-1]
```

```python

    return "".join(map(str, digits))

def eca(list_init, rule):

    list_new = []

    rule = decimal_to_binary(rule)
    rule_dict = {"111" : rule[0], "110" : rule[1], "101" : rule[2], "100" : rule[3], "011" : rule[4], "010" : rule[5], "001" : rule[6], "000" : rule[7]}

#    Treat list_init as cyclic
    list_init = list_init[-1 : ] + list_init + list_init[0 : 1]

    for i in range(len(list_init) - 2):
        list_new.append(int(rule_dict["".join(map(str, list_init[i : i + 3]))]))

    #list_new = list_init[0 : 1] + list_new + list_init[-1 : ]

    return list_new

def eca_matrix(rule, length = 101, width = 100):
    eca_matrix = [[0 for i in range(width)] for j in range(length)]

#    eca_matrix[0] = [0 for i in range(width // 2)] + [1] + [0 for i in range(width // 2)]

#    Randomize list_init
    np.random.seed(100)
    eca_matrix[0] = [np.random.randint(0, 2) for i in range(width)]

    for i in range(1, length):
        eca_matrix[i] = eca(eca_matrix[i - 1], rule)

    return eca_matrix

# MODIFICATION: Choose between one and two steps to the left and between one and two steps to the right with bias bias, towards the immediate neighbors.
def eca_modified(list_init, rule, bias):

    list_new = []

    rule = decimal_to_binary(rule)
    rule_dict = {"111" : rule[0], "110" : rule[1], "101" : rule[2], "100" : rule[3], "011" : rule[4], "010" : rule[5], "001" : rule[6], "000" : rule[7]}

#    Treat list_init as cyclic
    list_init = list_init[len(list_init) - 2 : ] + list_init + list_init[0 : 2]

    for i in range(2, len(list_init) - 2):
        before = [list_init[i - 2], list_init[i - 1]]
        after = [list_init[i + 2], list_init[i + 1]]
        list_new.append(int(rule_dict["".join(map(str, [before[np.random.binomial(1, bias)], list_init[i], after[np.random.binomial(1, bias)]]))]))

    #list_new = list_init[0 : 2] + list_new + list_init[len(list_init) - 2 : ]

    return list_new

```

```python
def eca_modified_matrix(rule, bias, length = 101, width = 100):
    eca_matrix = [[0 for i in range(width)] for j in range(length)]

#    eca_matrix[0] = [0 for i in range(width // 2)] + [1] + [0 for i in range(width // 2)]

#    Randomize list_init
    np.random.seed(100)
    eca_matrix[0] = [np.random.randint(0, 2) for i in range(width)]

    for i in range(1, length):
        eca_matrix[i] = eca_modified(eca_matrix[i - 1], rule, bias)

    return eca_matrix

def eca_plot(matrix, rule):

    plt.imshow(matrix, vmin = 0, vmax = 1, cmap = "binary")
    plt.title('Elementary Cellular Automata Rule ' + str(rule))
    plt.show()

# ECA Characterization

def kinetic_energy(matrix):

    total_flips_ave = 0

    for row in range(1, len(matrix)):

        total_flips = 0
        for column in range(len(matrix[row])):
            total_flips += abs(matrix[row][column] - matrix[row - 1][column])

        total_flips_ave += total_flips / len(matrix)

    kinetic_energy = total_flips_ave / len(matrix[0])

    return kinetic_energy

def spatial_entropy(matrix):

#    length = len(matrix[0])
#    norm_const = (length / 8) * np.log(1 / length)
    norm_const = 1

    spatial_entropy = 0

    for row in range(len(matrix) - 1):

        shannon_entropy = 0
        prob_dict = {"111" : 0, "110" : 0, "101" : 0, "100" : 0, "011" : 0, "010" : 0, "001" : 0, "000" : 0}
        cyclic_row = matrix[row][-1 : ] + matrix[row] + matrix[row][0 : 1]

        for column in range(len(cyclic_row) - 2):
            triplet = "".join(map(str, cyclic_row[column : column + 3]))
            prob_dict[triplet] += 1 / len(matrix[0])

        for prob in prob_dict.values():
            if prob == 0:
                continue
            else:
                shannon_entropy += prob * np.log(prob)

        spatial_entropy += -norm_const * shannon_entropy / len(matrix)

    return spatial_entropy

fig, ax1 = plt.subplots(figsize = (20, 20), nrows = 16, ncols = 16)

for i in range(256):
    row_num = i // 16
    col_num = i % 16
    ax1[row_num, col_num].imshow(eca_matrix(i), vmin = 0, vmax = 1, cmap = "binary")
    ax1[row_num, col_num].set_title(str(i))

plt.show()

kinetic_energy_vals = []
spatial_entropy_vals = []
kinetic_energy_vals_equal = []
spatial_entropy_vals_equal = []

for i in range(256):
    kinetic_energy_vals.append(kinetic_energy(eca_matrix(i)))
    spatial_entropy_vals.append(spatial_entropy(eca_matrix(i)))
    kinetic_energy_vals_equal.append(kinetic_energy(eca_modified_matrix(i, 0.5)))
    spatial_entropy_vals_equal.append(spatial_entropy(eca_modified_matrix(i, 0.5)))

plt.plot(spatial_entropy_vals, kinetic_energy_vals, ls = "", marker = "o")
plt.title("ECA")
plt.xlabel("spatial entropy")
plt.ylabel("kinetic energy")
plt.show()

plt.plot(spatial_entropy_vals_equal, kinetic_energy_vals_equal, ls = "", marker = "o")
plt.title("ECA, Bias = 0.5")
plt.xlabel("spatial entropy")
plt.ylabel("kinetic energy")
plt.show()

for i in range(256):
    kinetic_energy_diff = [kinetic_energy_vals[i] - kinetic_energy_vals_equal[i] for i in range(256)]
    spatial_entropy_diff = [spatial_entropy_vals[i] - spatial_entropy_vals_equal[i] for i in range(256)]

plt.plot(spatial_entropy_diff, kinetic_energy_diff, ls = "", marker = "o")
plt.title("ECA Default and Modified Rule Differences")
plt.xlabel("spatial entropy")
plt.ylabel("kinetic energy")
plt.show()

# Runs very slowly because of nested for loop. Comment out to check plot.

#bias_list = np.arange(0.1, 1, 0.1)
#
#fig, ax2 = plt.subplots(figsize = (15, 9), nrows = 3, ncols = 3)
#plt.setp(ax2.flat, xlabel = "spatial entropy", ylabel= "kinetic energy")

#for i in range(len(bias_list)):
```

```
#    kinetic_energy_modified_vals = []
#    spatial_entropy_modified_vals = []
#    for j in range(256):
# kinetic_energy_modified_vals.append(kinetic_energy(eca_modified_matrix(j, bias_list[i])))
# spatial_entropy_modified_vals.append(spatial_entropy(eca_modified_matrix(j, bias_list[i])))
#
#    row_num = i // 3
#    col_num = i % 3
#    ax2[row_num, col_num].plot(spatial_entropy_modified_vals, kinetic_energy_modified_vals, ls = "", marker = "o")
#    ax2[row_num, col_num].set_title("Modified ECA, Bias: " + str(bias_list[i]))
#
#plt.show()
```