Automata and Life

Patrick Harrison 30023631 & Craig Michie 30001523

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import scipy as sc
import phys481_game_of_life as gol
%matplotlib inline
```

Introduction

Python is a imperitive paradigm programing language. This means that the programmer will tell the machine how to change it's state. Up to this point we have only looked at the procedural nature in python, where a line of code s a procedure to exicute. In an object oriented nature, instructions are grouped together with the state they operate on.

The game of life is a good way to explore pythons object oriented nature. We can create a cellular automata (1-dimension) to analyze the steady state behaviour of a certain rules and how entropy changes with the change in rules.

A 1-dimensional automata is a grid of cells in a binary state that evolve through a certain number of steps. The way this grid of cells evolve is by looking at the state of it's neighbouring cells. In the 1-dimensional case, the state of cell n in the grid wil depend on cell $n+1$, $n-1$ and It's self.

To establish a rule for a cellular automata, we only have to look at the 3-bit number

$$\left[\text{left cell} = 1,0, \quad \text{cell} = 1,0, \quad \text{right cell} = 1,0\right]$$

since this is a binary number, we need to decribe the states of cells as binary numbers. to do this in python we can use numpy as follows

In [2]:
```
for number in range(0,4):
    print('The binary representation of',number,' is', np.unpackbits(np.uint8(number)))
```

```
The binary representation of 0  is [0 0 0 0 0 0 0 0]
The binary representation of 1  is [0 0 0 0 0 0 0 1]
The binary representation of 2  is [0 0 0 0 0 0 1 0]
The binary representation of 3  is [0 0 0 0 0 0 1 1]
```

This is the bits of the numbers 0 to 3 sotered as a numpy array. This form is convenient rather then a string or a list for ease of computation later. The downside is that it is a very expensive method to convert as it calls a function within a function.

for this 3-bit number there are 2^3 events possible for each cell in the next step. Conway's game of life lists those in the order:

$$[1,1,1] \quad [1,1,0] \quad [1,0,1] \quad [1,0,0] \quad [0,1,1] \quad [0,1,0] \quad [0,0,1] \quad [0,0,0]$$

Where for each rule, the 8-bit number, for example rule 1, has binary form 00000001, each bit corrisponds respectivly to what each different state will give.

$$[1,1,1]=0 \quad [1,1,0]=0 \quad [1,0,1]=0 \quad [1,0,0]=0 \quad [0,1,1]=0 \quad [0,1,0]=0 \quad [0,0,1]=0 \quad [0,0,0]=1$$

For each bit in an 8-bit rule, gives a answer to each possible state for a cell to have with it's neighbours.

Write a program to generate a sequence on $N = 64$ grid cells (1-D) and an arbitrary rule

To create the $N = 64$ grid we will use a numpy array filled with zeros to start except for an initial point somewhere near the middle of the grid. This will act as our 64-bit integer state.

In [3]:
```
def grid (size,initial_points):
    '''
    Creates a numpy array to act as a grid with a given length

    parameters:
        size -- length of the 1D grid
        initial_points -- specified initial cells to turn on (=1) in a list

    returns:
        state -- 1D numpy array of binary state dtype 8 bit int.

    '''

    ncells = size #ncell is size of bit number
    state = np.zeros(ncells, dtype=np.int8 )
    state=np.array(state) # put into an array for ease of computation

    # initialize cell True for each n.
    for n in initial_points:
        state[n] = 1

    return state
```

Test the function to make sure it works

In [4]:
```
grid(64,[32])
```

Out[4]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
      dtype=int8)
```

Take a function that will take a state and use a rule to evolve the state. ech rule is based on the binary form of an 8 bit integer. This cellular step function will wrap around from edge to edge.

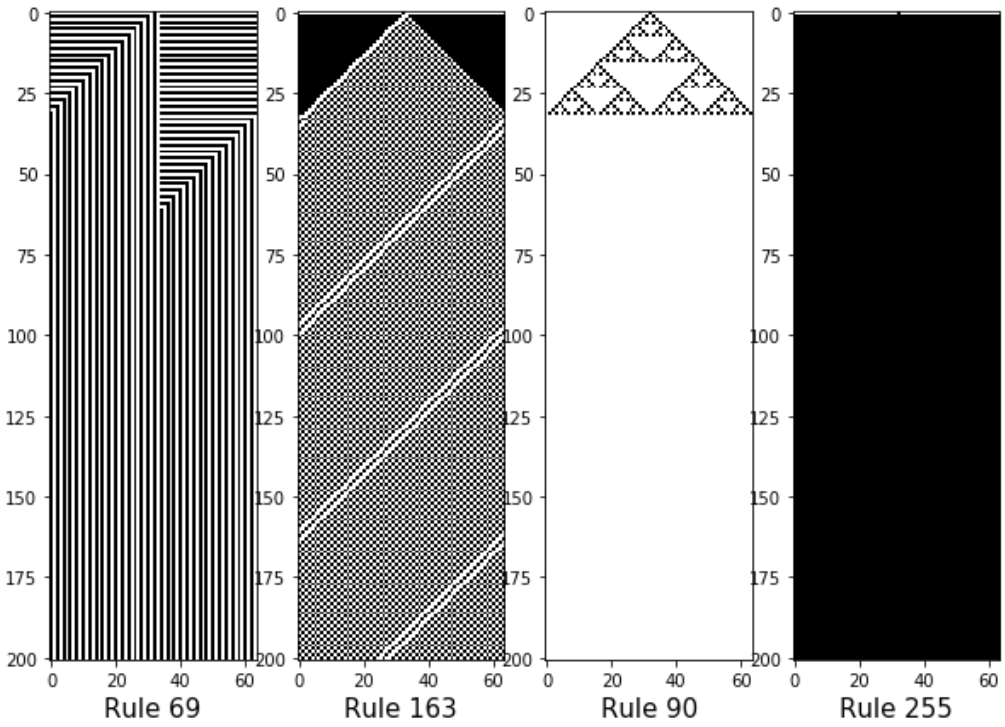
In [5]:
```
def cellular_step(value, rule_number=110):
    '''
    evolves a state n to state n+1 by unpacking bits and rolling them back up
    into a different configuration based on the rule.

    args:
        value       -- bits to evolve
        rule_number -- interger rule
    returns:
        lookup[triple] -- new bits for evolved state

    '''

    lookup = np.unpackbits( np.uint8(rule_number) )[::-1]
    triple = np.roll(value,+1)*4 + value*2 + np.roll(value,-1)

    return lookup[triple]
```

we can then take a function that will calculate the state evolution for a given number of itterations and plot it. This is what gives the cellular automata and we can start to see patterns occuring

In [6]:
```
def cellular_automata (value,nsteps,rule,plot=False):
    '''
    uses the cellular_step to evolve a value nsteps and may return a plot.

    args:
        value -- initail bits
        nsteps -- number itterations
        rule -- 8bit number specifying with rule to use
        plot -- if True will return a plot

    returns:
        states -- a 2 dimensional array giving each state for each step

    '''

    states = [value]
    state = np.ndarray( [nsteps, len(value)], dtype=np.int8)
    for n in range(nsteps):
        value = cellular_step(value, rule)
        states.append(value)
        state[n,:] = value

    if plot:
        fig, ax = plt.subplots(figsize=(6,6))
        title = 'Game of Life for N=', len(value) ,' and rule number',rule
        ax.set_title(str(title))
        ax.set_xlabel('state of cell')
        ax.set_ylabel('steps')
        ax.imshow(state,cmap='Greys')

    return states
```

Test the above function to calculate the cellualr automata for rule 69 for 200 steps with a single intial point in the middle of the grid.

In [7]:
```
ca_rule69 = cellular_automata (grid(64,[32]),nsteps=200,rule=69,plot=False)
ca_rule163 = cellular_automata (grid(64,[32]),nsteps=200,rule=163,plot=False)
ca_rule90 = cellular_automata (grid(64,[32]),nsteps=200,rule=90,plot=False)
ca_rule255 = cellular_automata (grid(64,[32]),nsteps=200,rule=255,plot=False)

fig = plt.figure(figsize=(10,10)) # Set plot size
ax1 = plt.subplot(1,4,1) # Subplot 1
ax1.set_xlabel('Rule 69', fontsize=15)
plt.imshow(ca_rule69, cmap="Greys")

ax2 = plt.subplot(1,4,2) # Subplot 2
ax2.set_xlabel('Rule 163', fontsize=15)
plt.imshow(ca_rule163, cmap="Greys")

ax3 = plt.subplot(1,4,3) # Subplot 3
ax3.set_xlabel('Rule 90', fontsize=15)
plt.imshow(ca_rule90, cmap="Greys")

ax4 = plt.subplot(1,4,4) # Subplot 4
ax4.set_xlabel('Rule 255', fontsize=15)
plt.imshow(ca_rule255, cmap="Greys")

plt.show()
```



We can see that for rule 69, it evolves into a static steady state after ~55 to ~60 steps. Rule 163 on the other hand go to a non-static steady state. Instead of using the cellular automata fucntion every itteration, it will be easier to compute the states for each rule in 200 steps and store it in a catalog of values. The keys will be the rule and the values a list of lists for each state.

In [8]:
```
def get_rule_catalog(igrid,steps):
    ''' puts initail grid into a catalog for an evolution of nsteps'''
    rule_catalog ={}
    for rule in range(0,256):
        rule_catalog[rule]=cellular_automata(grid(64,[32]),nsteps=200,rule=rule,plot=False)
    return rule_catalog
```

Using a function makes it easy to go back and change the number of steps or the grid later if we need to without having to go through alot of code.

In [9]:
```
rule_catalog = get_rule_catalog(grid(64,[32]),200)
```

Analyze steady state entropy

Assuming that the steady state will occur after 100 steps in evolution, we can take the entropy from steps 100 to 200 to get the steady sate entropy. Each line will have it's own entropy bit by bit. Taking the sum of the entropy for each step in an assumed steady state, It is possible to gain the average by then dividing by the number of steps you summed over.

$$\bar{x} = \frac{\sum_i^n x_i}{n}$$

We define the entropy as the same as the previous assignment.

$$S = -k_B \sum_i p_i \log_2 p_i$$

Start by getting the probability of getting a 1 for each step in a cellular automata for each rule.

In [10]:
```
def get_prob(rule_catalog,rule,nsteps):
    '''
    gets the probability of getting a 1 for each step in a cellular automata in a given rule.

    args:
        rule_catalog -- A catalog of keys being rules and values of the resulting nsteps as a 2
d list
        rule         -- rule to evolve step over
        nsteps       -- how many steps to evolve over

    returns:
        probability_list -- a 1d list of probabilities for given rule

    '''

    probability_list=[]
    for i in range(len(rule_catalog[0])): # all number of steps assumed to be the same so takin
g first index.
        y = rule_catalog[rule][i]
        prob = np.count_nonzero(y == 1)/len(rule_catalog[rule][i])
        probability_list.append(prob)

    return probability_list
```

Check to see if the function works for rule 255 as this is easy one to visulize as it goes to all 1 after at least 1 step.

In [11]:
```
print(get_prob(rule_catalog,rule=255,nsteps=200))
```

```
[0.015625, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.
0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

In [12]:
```
def get_prob_catalog (rule_catalog,nsteps):
    '''puts probability into a catalog for each rule'''
    prob_catalog={}
    for rule in range(0,256):
        prob_catalog[rule]=get_prob(rule_catalog,rule,nsteps=200)
    return prob_catalog

prob_catalog = get_prob_catalog(rule_catalog,nsteps=200)
```

This is a dictionay in which the keys are rules and the values are a list of probabilites for each step in the evolution. We will now finally define a function to calculate the entropy for each step and average a certain number of steady state evolutions. This gives a better entropy then a striaght up calculation of the numbers or a difference. A striaght calculation does not take into effect the non-static steady state case. The difference method also does not properly take into account if the difference is not a proper magnitude.

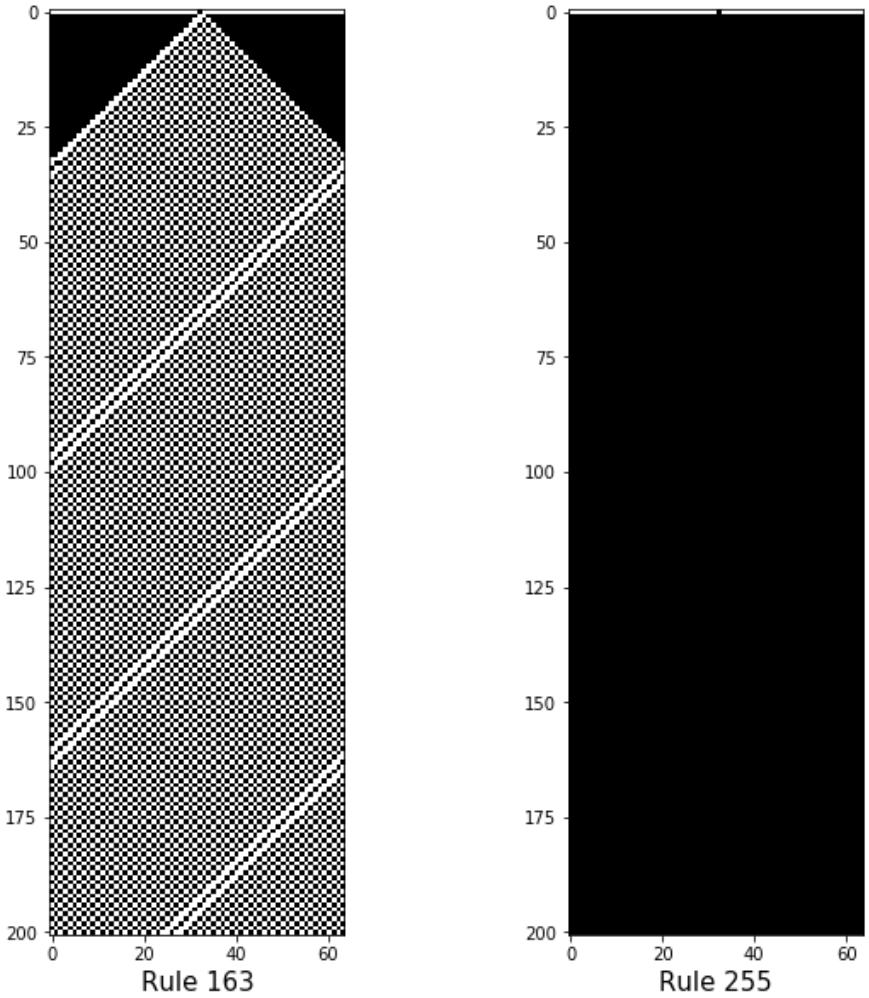
In [13]:
```
def get_entropy (prob,rule,nsteps,steps):
        '''
        Calculate the entropy for steady state for a given rule

        args:
            prob -- probability catalog
            rule -- rule to evolve by
            nsteps -- number of steps to evolve over
            steps -- how many steps to average over (last number of steps)

        returns:
            entropy -- the average entropy for a rule for the last step number of nsteps

        '''

        e = 0.0
        for n in range(nsteps-steps,nsteps):
            if prob[rule][n] != 0:
                #print(prob[rule][n])
                e += np.log2(prob[rule][n]) * prob[rule][n]
                #print ('e',e)
        entropy = -1*e /steps


        return entropy
```

In [14]:
```
entropy_163=get_entropy(prob_catalog,rule=163,nsteps=200,steps=10)
entropy_255=get_entropy(prob_catalog,rule=255,nsteps=200,steps=10)
print('The entropy of rule 163 is:', entropy_163)
print('The entropy of rule 255 is:', entropy_255)
```

```
The entropy of rule 163 is: 0.5065611621563574
The entropy of rule 255 is: -0.0
```

This also makes sense for this rule as entropy can be considered a measure of how unexpected a value is. Since rule 255 is very ordered, we can say that entropy will be very low for this rule. Rule 163 however does have more of an unordered stucture to it.

In [15]:
```
ca_rule163 = cellular_automata (grid(64,[32]),nsteps=200,rule=163,plot=False)
ca_rule255 = cellular_automata (grid(64,[32]),nsteps=200,rule=255,plot=False)

fig = plt.figure(figsize=(10,10)) # Set plot size

ax1 = plt.subplot(1,2,1) # Subplot 1
ax1.set_xlabel('Rule 163', fontsize=15)
plt.imshow(ca_rule163, cmap="Greys")

ax2 = plt.subplot(1,2,2) # Subplot 2
ax2.set_xlabel('Rule 255', fontsize=15)
plt.imshow(ca_rule255, cmap="Greys")

plt.show()
```



In [16]:
```
def get_entropy_catalog (entropy,nsteps):
    '''put entropy into a catalog for ease'''
    entropy_catalog={}
    for rule in range(0,256):
        entropy_catalog[rule]=get_entropy(prob_catalog,rule,nsteps=200,steps=10)
    return entropy_catalog
```

In [17]:
```
entropy_catalog = get_entropy_catalog (get_entropy,nsteps=200)
print(entropy_catalog.get(255))
```

```
-0.0
```

Now we can easily get an entropy for a given rule in it's steady state. This also makes it easy to get find the 5 maximum entropy rules.

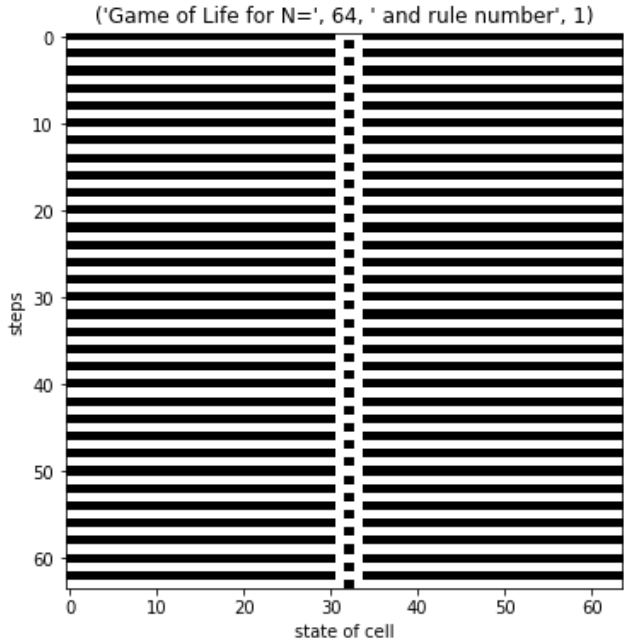
In [18]:
```
def get_max (entropy_catalog):
    ''' finds and returns largest five maximum values for entropy in a list'''
    maximum=sorted(entropy_catalog, key=entropy_catalog.__getitem__)[-5:][::-1]
    return maximum
```

In [19]:
```
for max_entropy in get_max(entropy_catalog):
    print('rule',max_entropy,'has an entropy of:',entropy_catalog[max_entropy])
```

```
rule 145 has an entropy of: 0.5306258201732998
rule 131 has an entropy of: 0.5306258201732998
rule 193 has an entropy of: 0.5209840033604419
rule 137 has an entropy of: 0.5209840033604419
rule 73 has an entropy of: 0.5133127931807003
```

To see the difference between the average antropy calculation and the difference in entropy calculation we can calculate the entropy similar to before and subtract a slightly higher step. This will help with calculating entropy for a evolution rule such as rule 1. visually seen below, the bits alternate in a non-static steady state. By taking a rows entropy difference with a row two steps above it we can find a good approximation to the entropy of the cellular atuomata.

In [20]:
```
ca_rule1=cellular_automata (grid(64,[32]),nsteps=64,rule=1,plot=True)
```



In [21]:
```
def entropy_diff (prob,rule,nsteps=200):
    '''
    calculates entropy difference for rows a distance 2 apart.

    args:
        prob -- probability catalog
        nsteps -- number of steps to evolve over
    returns:
        entropy_catalog -- an entropy catalog for a difference in entropy calculation
    '''
    e = 0.0
    for n in range(100,len(prob)-100):
        if prob[rule][n] and prob[rule][n+2] !=0:
            e += np.abs((np.log2(prob[rule][n]) * prob[rule][n])\
                        -(np.log2(prob[rule][n+2]) * prob[rule][n+2]))
    entropy = -1*e

    return entropy_catalog
```

In [22]:
```
ent_diff_cat = entropy_diff (prob_catalog,rule=2,nsteps=200)
print('entropy difference for rule 2 using average method:',ent_diff_cat.get(2))
print('entropy difference for rule 163 using average method:',ent_diff_cat.get(163))
print('entropy difference for rule 69 using average method:',ent_diff_cat.get(69))
```

```
entropy difference for rule 2 using average method: 0.09375
entropy difference for rule 163 using average method: 0.5065611621563574
entropy difference for rule 69 using average method: 0.5
```

compared to the average entropy calculation of:

In [23]:
```
print('entropy difference for rule 2 using difference method:',entropy_catalog.get(2))
print('entropy difference for rule 163 using difference method:',entropy_catalog.get(163))
print('entropy difference for rule 69 using difference method:',entropy_catalog.get(69))
```

```
entropy difference for rule 2 using difference method: 0.09375
entropy difference for rule 163 using difference method: 0.5065611621563574
entropy difference for rule 69 using difference method: 0.5
```

Both methods give the same answer for rule 2 and 163 which are non-static steady states and rule 69 which is a static state. a farther test is to see if the difference method gives the same five maximum entropy rules.

In [24]:
```
for max_entropy in get_max(ent_diff_cat):
    print('rule',max_entropy,'has an entropy of:',ent_diff_cat[max_entropy])
```

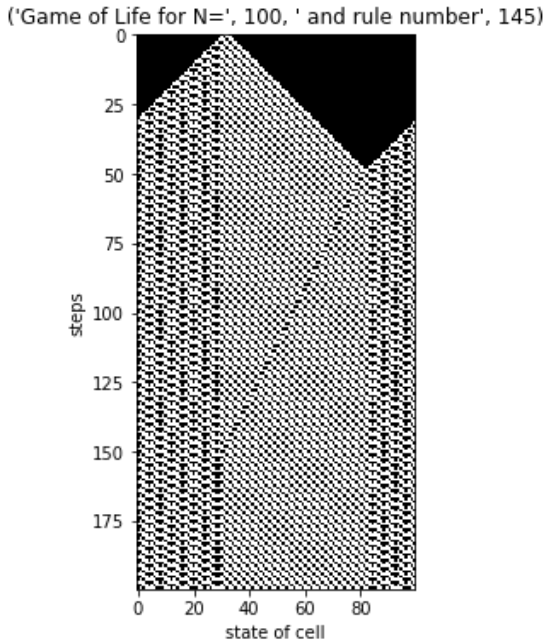
```
rule 145 has an entropy of: 0.5306258201732998
rule 131 has an entropy of: 0.5306258201732998
rule 193 has an entropy of: 0.5209840033604419
rule 137 has an entropy of: 0.5209840033604419
rule 73 has an entropy of: 0.5133127931807003
```

It is found for both methods that the max five entropy rules are the same. rule 143, rule 131, rule 193, rule 137, and rule 73, are the max five (most pseudorandom) rules.

Generate a pseudorandom sequence 64-bit integer based on a cellular automaton using the highest entropy rule.

Using the highest entropy rule (rule 145), we can create a pseudorandom sequence of 64bit integers. The highest entropy rule will give the most random appearance as it has the highest entropy.

In [25]:
```
x=cellular_automata (grid(100,[32]),nsteps=200,rule=145,plot=True)
```



In [26]:
```
def state_to_binary(initial_point,nsteps=200,rule=94):
    '''
    takes an inital point and calculates a given cellular automata up to an amount of steps and
returns each
    step as it's integer number.

    args:
        inital_point -- a list of inital points in the 64-bit grid
        nsteps -- number of steps to calculate over
        rule -- which rule to evolve for nsteps over
    returns:
        list of psuedorandom numbers generated from the cellualr automata

    '''
    numbers=[]
    for step in range(nsteps):
        number=(cellular_automata(grid(64,initial_point),nsteps,rule,plot=False)[step])
        str_number= ''.join(str(e) for e in number)

        numbers.append(str_number)
    for number in range(len(numbers)):
        numbers[number]=int(str(numbers[number]),2)

    return numbers
```

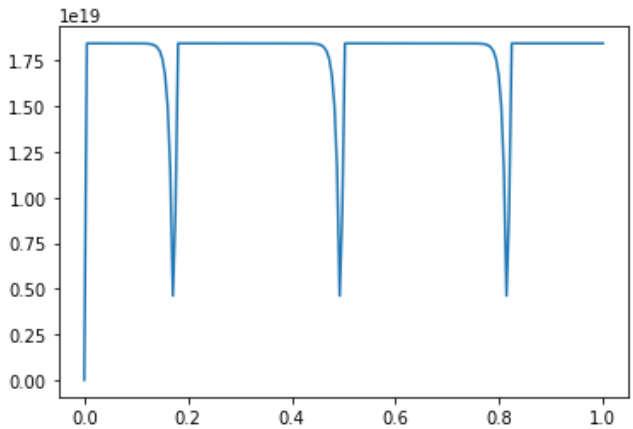
In [27]:
```
random_numbers=state_to_binary(initial_point=[32],nsteps=200,rule=143)
print(random_numbers[:35])
```

```
[2147483648, 18446744072635809791, 18446744070488326143, 18446744067267100671, 184467440608246
49727, 18446744047939747839, 18446744022169944063, 18446743970630336511, 18446743867551121407,
18446743661392691199, 18446743249075830783, 18446742424442109951, 18446740775174668287, 184467
37476639784959, 18446730879570018303, 18446717685430484991, 18446691297151418367, 184466385205
93285119, 18446532967477018623, 18446321861244485631, 18445899648779419647, 184450552238492876
79, 18443366373989023743, 18439988674268495871, 18433233274827440127, 18419722475945328639, 18
392700878181105663, 18338657682652659711, 18230571291595767807, 18014398509481983999, 17582052
945254416383, 16717361816799281151, 14987979559889010687, 11529215046068469759, 46116860184273
87903]
```

This gives a decent sequence of apparent random numbers though some values are repeated often in a row.

In [28]:
```
x=np.linspace(0,1,len(random_numbers))
y=random_numbers
plt.plot(x,y)
```

Out[28]:
```
[<matplotlib.lines.Line2D at 0x21e4009deb8>]
```



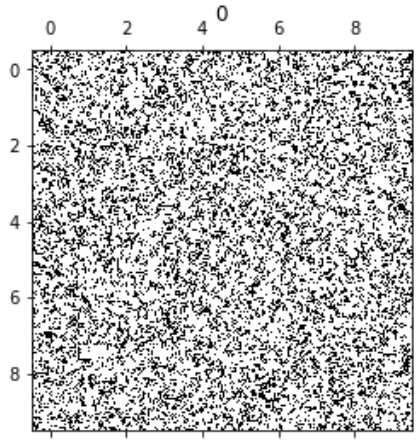
plotting these numbers, it does not appear they are very random.

Implimentations of different stepper functions

Start by creating a random grid of on off states with a probability p

In [29]:
```
def TwoDGrid (xsize,ysize,p):
    ''' returns a grid of xsize by ysize wtih a probability of cells turned on of p'''
    return np.random.rand(xsize,ysize) > p
```

In [30]:
```
p = plt.matshow(TwoDGrid(10,10,0.5),cmap='Greys')

#show it
final_grid = gol.life_generation_stepper(TwoDGrid(300,300,0.5),nsteps=1,plot=p)
```



Performing time trials for each stepper, we can compare the efficiency for each stepper. This is done by performing timeit for each stepper for a random 50x50 grid. Firstly, to create a random 50x50 grid.

In [31]:
```
def stepystep_test (grid):
    '''
    times the stepper functions for a given grid
    '''

    print('\tStepper0')
    %timeit gol.stepper0(grid)
    print('\tStepper1')
    %timeit gol.stepper1(grid)
    print('\tStepper2')
    %timeit gol.stepper2(grid)
    print('\tStepper3')
    %timeit gol.stepper3(grid)
    print('\tStepper4')
    %timeit gol.stepper4(grid)
```

In [32]:
```
stepystep_test(TwoDGrid(50,50,0.5))
```

```
        Stepper0
151 ms ± 59.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
        Stepper1
111 ms ± 36.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
        Stepper2
2.62 ms ± 630 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
        Stepper3
1.32 ms ± 161 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
        Stepper4
1.07 ms ± 7.85 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [33]:
```
def call_stepper (stepper,igrid,nsteps,plot):
    '''
    calls the steper functions
    '''

    if stepper == 'stepper0':
        new_grid = gol.stepper0(igrid, nsteps, plot=None)
    elif stepper == 'stepper1':
        new_grid = gol.stepper1(igrid, nsteps, plot=None)
    elif stepper == 'stepper2':
        new_grid = gol.stepper2(igrid, nsteps, plot=None)
    elif stepper == 'stepper3':
        new_grid = gol.stepper3(igrid, nsteps, plot=None)
    elif stepper == 'stepper4':
        new_grid = gol.stepper4(igrid, nsteps, plot=None)
    elif stepper == 'life_generation_stepper':
        new_grid = gol.life_generation_stepper(igrid, nsteps, plot=None)

    if plot:
        fig, ax = plt.subplots()
        ax.imshow(new_grid)

    return new_grid
```

From these timestamps, it can be observed that they get faster down the list above. This makes sense, as in the game of life code where the steppers are defined it is observed that stepper0 has the most amount of for loops, if, elif and else statements and the use of them decreases through to stepper4 (which doesn't use any). We can conclude that limiting the use of loops and statements, increases the speed of the stepper.

Analyze steady state behaviour

We will look only at the static case of steady state behaviour for now. That is, if the rule does not make any changes to the new grid from the initial grid

In [34]:
```
def get_steady_state(igrid,nsteps,stepper,plot=False):
    '''
    calculates each step in Conway's game of life and stops when it finds a static steady state

    args:
        igrid -- initial grid
        nsteps -- number of steps to evolve over
        stepper -- stepper function to use
        plot -- if true will return a plot
    returns:
        new_grid -- last grid computed once static steady state is reached
        ev -- the evolution step the rule entered a static steady state

    '''

    new_grid=igrid
    for steps in range(nsteps):
        new_grid = call_stepper(str(stepper), igrid=igrid, nsteps= 1, plot=None)

        if np.array_equal(new_grid,igrid):
            ev=steps
            break
        else:
            ev=steps

        igrid=new_grid


    if ev == nsteps-1:
        ev = 'Never reached a static steady state'
    if plot:
        fig, ax = plt.subplots()
        ax.imshow(new_grid)

    return new_grid, ev
```

In [35]:
```
ngrid, step= get_steady_state(igrid=TwoDGrid(10,10,0.5), nsteps=10000, stepper='stepper3', plot
=False)
print(step)
```

```
141
```

It is sometimes seen that the grid never reaches a static steady state in a resonable amount of time. This case will be ommitted from the mean and standard deviation below.

In [36]:
```
steps=[]
for trial in range(5):
    new_grid, step = get_steady_state(igrid=TwoDGrid(10,10,0.5), nsteps=90000, stepper='stepper
3', plot=False)
    steps.append(step)

print(steps)
```

```
[36, 34, 35, 60, 32]
```

we will now calculate the mean and standard deviation as below.

In [37]:
```
def  mean (steps):
    '''calculates mean for given values (steps) in list form'''
    totaltrials=len(steps)
    tsum=0
    for n in range(len(steps)):
        if isinstance(steps[n], int): #removes the case on non-static steady state
            tsum += steps[n]
        else:
            totaltrials = totaltrials -1

    return tsum/totaltrials
```

In [38]:
```
print('The mean value is:',mean(steps))
```

```
The mean value is: 39.4
```

In [39]:
```
def sd(steps):
    '''
    calculates standard deviation ommitting nonstatic cases
    '''
    totaltrials=len(steps)
    for n in range(len(steps)):
        if not isinstance(steps[n], int): #removes the case on non-static steady state
            totaltrials = totaltrials -1
            print(len(steps))
            steps.pop(n)

    avg = [mean(steps) for x in range(len(steps))]
    steps = np.array(steps)
    return np.sqrt(np.sum(steps - avg)**2 / totaltrials)
```

In [40]:
```
print('The standard deviation is:',sd(steps))
```

```
The standard deviation is: 3.1776437161565094e-15
```

Examine steady state behaviour of inital random densities ranging from $p = 0$ to $p = 1$

use again the grid function from earlier.

In [41]:
```
steps = []
for p in range(0,11):
    p = p/10
    new_grid, step = get_steady_state(TwoDGrid (50,50,p),nsteps=5000,stepper='stepper3',plot=Fa
lse)
    steps.append(step)
    if isinstance(step, int): #removes the case on non-static steady state
        print('for the case of p=',p,'it reaches its static steady state after',step,'steps')
    else:
        print('for the case of p=',p,'it never reaches its static steady state')
```

```
for the case of p= 0.0 it reaches its static steady state after 1 steps
for the case of p= 0.1 it reaches its static steady state after 1 steps
for the case of p= 0.2 it never reaches its static steady state
for the case of p= 0.3 it never reaches its static steady state
for the case of p= 0.4 it never reaches its static steady state
for the case of p= 0.5 it never reaches its static steady state
for the case of p= 0.6 it never reaches its static steady state
for the case of p= 0.7 it never reaches its static steady state
for the case of p= 0.8 it never reaches its static steady state
for the case of p= 0.9 it never reaches its static steady state
for the case of p= 1.0 it reaches its static steady state after 0 steps
```

Conclusion

Using the cellular automata to get a sequence of random numbers is not a very good random number generator as a pattern starts to appear after a large number of steps. This is also implies that it is a non-static steady state. The highest entropy rule was calculated both by an average over a steady state and a ΔS for two different entropies of different rows in a steady state. Just plugging in the numbers should give results of zero or close to as each steady state is not completely suprising.

in the game of life, we only looked at the static steady state case. This was for ease of computation, to determine other static states, we could look at the fraction of on and off cells. if the fraction ramins constatnt for a certain number of steps, it may be in a steady state.

For probabilities of $0, 0.1, 0.2, \ldots, 1$ we see that the more evenly disperssed on off cells (0.4,0.5,0.6) we see that it either takes an incredible amount of steps or never reaches a static steady state.