11.3: Simulating Cellular Automata

Despite their capability to represent various complex nonlinear phenomena, CA are relatively easy to implement and simulate because of their discreteness and homogeneity.

![Typical behavior of the most well-known binary CA, the Game of Life.](image)

There are existing software tools and online interactive demonstrations already available for cellular automata simulation, but it is nonetheless helpful to learn how to develop a CA simulator by yourself. Let’s do so in Python, by working through the following example step by step. The CA model we plan to implement here is a binary CA model with the droplet rule [4]. Its state-transition function can be understood as a model of panic propagation among individuals sitting in a gym after a fire alarm goes off. Here is the rule (which uses the Moore neighborhoods):

- A normal individual will get panicky if he or she is surrounded by four or more panicky individuals.
- A panicky individual will remain panicky if he or she is surrounded by three or more panicky individuals. Otherwise he or she will revert back to normal.

Note that this rule can be further simplified to the following single rule:

- If there are four or more panicky individuals within the neighborhood, the central cell will become panicky; otherwise it will become normal.

Here are other model assumptions:
- Space: 2-D, \((n \times n)\) (\(n = 100\) for the time being)
- Boundary condition: periodic
- Initial condition: Random (panicky individuals with probability \(p\); normal ones with probability \(1-p\))

We will use pycxsimulator.py for dynamic CA simulations. Just as before, we need to design and implement the three essential components—initialization, observation, and updating.

To implement the initialization part, we have to decide how we are going to represent the states of the system. Since the configuration of this CA model is a regular two-dimensional grid, it is natural to use Python’s `array` data structure. We can initialize it with randomly assigned states with probability \(p\). Moreover, we should actually prepare two such arrays, one for the current time step and the other for the next time step, in order to avoid any unwanted conflict during the state updating process. So here is an example of how to implement the initialization part:

```
Code 11.1:

n = 100 # size of space: n x n
p = 0.1 # probability of initially panicky individuals

def initialize():
    global config, nextconfig
    config = zeros([n, n])
    for x in xrange(n):
        for y in xrange(n):
            config[x, y] = 1 if random() < p else 0
    nextconfig = zeros([n, n])
```

Here, `zeros([a, b])` is a function that generates an all-zero array with `a` rows and `b` columns. While `config` is initialized with randomly assigned states (1 for panicky individuals, 0 for normal), `nextconfig` is not, because it will be populated with the next states at the time of state updating.

The next thing is the observation. Fortunately, pylab has a built-in function `imshow` that is perfect for our purpose to visualize the content of an array:

```
Code 11.2:

def observe():
    global config, nextconfig
    cla()
    imshow(config, vmin = 0, vmax = 1, cmap = cm.binary)
```

Note that the `cmap` option in `imshow` is to specify the color scheme used in the plot. Without it, pylab uses dark blue and red for binary values by default, which are rather hard to see.

The updating part requires some algorithmic thinking. The state-transition function needs to count the number of panicky individuals within a neighborhood. How can we do this? The positions of the neighbor cells within the Moore neighborhood around a position \((x,y)\) can be written as

\[
\{(x',y'): x-1 \leq x' \leq x+1, y-1 \leq y' \leq y+1\}, \text{label}\{(11.3)\}
\]

This suggests that the neighbor cells can be swept through using nested `for` loops for relative coordinate variables, say, \(\langle dx\rangle\) and \(\langle dy\rangle\), each ranging from \(-1\) to \(+1\). Counting panicky individuals (1’s) using this idea can be implemented in Python as
Here, \( dx \) and \( dy \) are relative coordinates around \((x,y)\), each varying from \(-1\) to \(+1\). They are added to \( x \) and \( y \), respectively, to look up the current state of the cell located at \((x + dx, y + dy)\) in \( \text{config} \). The expression \((...) \% n\) means that the value inside the parentheses is contained inside the \([0, n - 1]\) range by the mod operator (\%). This is a useful coding technique to implement periodic boundary conditions in a very simple manner.

The counting code given above needs to be applied to all the cells in the space, so it should be included in another set of nested loops for \( x \) and \( y \) to sweep over the entire space. For each spatial location, the counting will be executed, and the next state of the cell will be determined based on the result. Here is the completed code for the updating:

```
Code 11.3:

count = 0
for dx in [-1, 0, 1]:
    for dy in [-1, 0, 1]:
        count += config[(x + dx) % n, (y + dy) % n]
```

```
nextconfig[x, y] = 1 if count >= 4 else 0
config, nextconfig = nextconfig, config
```

Note the swapping of \( \text{config} \) and \( \text{nextconfig} \) at the end. This is precisely the same technique that we did for the simulation of multi-variable dynamical systems in the earlier chapters.

By putting all the above codes together, the completed simulator code in its entirety looks like this:

```
Code 11.4:

def update():
    global config, nextconfig
    for x in xrange(n):
        for y in xrange(n):
            count = 0
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    count += config[(x + dx) % n, (y + dy) % n]

            nextconfig[x, y] = 1 if count >= 4 else 0
    config, nextconfig = nextconfig, config
```

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 100 # size of space: \( n \times n \)
p = 0.1 # probability of initially panicry individuals

def initialize():
    global config, nextconfig
    config = zeros((n, n))
    for x in xrange(n):
        for y in xrange(n):
```

UC Davis ChemWiki is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License.
When you run this code, you see the results like those shown in Fig. 11.3.2. As you can see, with the initial configuration with \( p = 0.1 \), most of the panicky states disappear quickly, leaving only a few self-sustaining clusters of panicky people. They may look like condensed water droplets, hence the name of the model (droplet rule).

```
config[x, y] = 1 if random() < p else 0
nextconfig = zeros([m, n])

def observe():
global config, nextconfig
clear()
imshow(config, vmin = 0, vmax = 1, cmap = cm.binary)

def update():
global config, nextconfig
for x in xrange(n):
    for y in xrange(n):
        count = 0
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                count += config[(x + dx) % n, (y + dy) % n]

        nextconfig[x, y] = 1 if count >= 4 else 0
        config, nextconfig = nextconfig, config

import pycsimulator
pycsimulator.GUI().start(func=(initialize, observe, update))
```

Figure \( \PageIndex{2} \): Visual output of Code 11.5. Left: Initial configuration with \( p = 0.1 \). Right: Final configuration after 20 time steps.

Exercise \( \PageIndex{1} \):

Modify Code 11.5 to implement a simulator of the Game of Life CA. Simulate the dynamics from a random initial configuration. Measure the density of state 1’s in the configuration at each time step, and plot how the density changes over time. This can be done by creating an empty list in the `initialize` function, and then making the measurement and appending the result to the list in the `observe` function. The results stored in the list can be plotted manually after the simulation, or they could be plotted next to the visualization using `pylab`’s `subplot` function during the simulation.

Exercise \( \PageIndex{2} \):

Modify Code 11.5 to implement a simulator of the majority rule CA in a two-dimensional space. Then answer the following questions by conducting simulations:

- What happens if you change the ratio of binary states in the initial condition?
• What happens if you increase the radius of the neighborhoods?

• What happens if you increase the number of states?

The second model revision in the previous exercise (increasing the radius of neighborhoods) for the majority rule CA produces quite interesting spatial dynamics. Specifically, the boundaries between the two states tend to straighten out, and the characteristic scale of spatial features continuously becomes larger and larger over time (Fig. 11.3.3). This behavior is called coarsening (to be more specific, non-conserved coarsening). It can be seen in many real-world contexts, such as geographical distributions of distinct cultural/political/linguistic states in human populations, or incompatible genetic types in animal/plant populations.

Figure \(\PageIndex{3}\): Coarsening behavior observed in a binary CA with the majority rule in a \(100 \times 100\) 2-D spatial grid with periodic boundary conditions. The radius of the neighborhoods was set to 5. Time flows from left to right.

What is most interesting about this coarsening behavior is that, once clear boundaries are established between domains of different states, the system’s macroscopic behavior can be described using emergent properties of those boundaries, such as their surface tensions and direction of movement[41,42]. The final fate of the system is determined not by the relative frequencies of the two states in the space, but by the topological features of the boundaries. For example, if a big “island” of white states is surrounded by a thin “ring” of black states, the latter minority will eventually dominate, no matter how small its initial proportion is (even though this is still driven by the majority rule!). This is an illustrative example that shows how important it is to consider emergent macroscopic properties of complex systems, and how counter-intuitive their behaviors can be sometimes.

Here is one more interesting fact about CA dynamics. The droplet/panic model discussed above has an interesting property: When you increase the initial density of panicky people (e.g., \(p = 0.3\)), the result changes dramatically. As seen in Fig. 11.3.4, the initially formed clusters tend to attach to each other, which makes their growth unstoppable. The whole space will eventually be filled up with all panicky people, which could be a disaster if this was a real situation.

Figure \(\PageIndex{4}\): Another visual output of Code 11.5. Left: Initial configuration with \(p = 0.3\). Right: Configuration after 40 time steps.

You can explore the value of \(p\) to find out that the transition between these two distinct behaviors takes place at a rather
narrow range of \( p \). This is an example of a phase transition, which is defined informally as follows:

A phase transition is a transition of macroscopic properties of a collective system that occurs when its environmental or internal conditions are varied

A familiar example of phase transitions is the transition between different phases of matter, i.e., solid, liquid, and gas, which occur when temperature and/or pressure are varied. Phase transitions can be characterized by measuring what physicists call order parameters\(^4\) that represent how ordered a macroscopic state of the system is. A phase transition can be understood as a bifurcation observed in the macroscopic properties (i.e., order parameters) of a collective system.

Exercise \( \PageIndex{3} \): Implement an interactive parameter setter for \( p \) in Code 11.5. Then conduct systematic simulations with varying \( p \), and identify its critical value below which isolated clusters are formed but above which the whole space is filled with panic.

\(^2\) Most notable is Golly (http://golly.sourceforge.net/).

\(^3\) For example, check out Wolfram Demonstrations Project(http://demonstrations.wolfram.com/) and Shodor.org’s interactive activities (http://www.shodor.org/interactivate/activities/).

\(^4\) Note that the word “parameter” in this context means an outcome of a measurement, and not a condition or input as in “model parameters.”