Problem set

Problem 1: Design a model

- LOs: #Modeling
- HCs: #emergentproperties
- Words: 300–600

(Adapted from Exercise 2.4 in H. Sayama, *Introduction to the Modeling and Analysis of Complex Systems*, p. 299.)

Write a few paragraphs (no math required) describing a real-world system of your choice with many interacting components/agents.

- At which scale do you choose to describe the microscopic components, and what are those components? Briefly describe other scales (larger and/or smaller) at which you could have chosen to model the microscopic state and why your chosen scale is the most appropriate.
- What states can the microscopic components take on?
- What are the relationships between the components?
- How do the states of the components change over time? Some of these changes may depend on interactions between the components, while others may not.

After answering all of the questions above, predict what kind of macroscopic behaviors would arise if you run a computational simulation of your model.

Topic: Model the societal opinion for China in Taiwan society

- **Scale**: Taiwan's population from 18-80 years old
- **Components**: There are multiple ways to break down society into components. e.g. Parties, schools, income groups, gender groups, education level groups, families, and individuals. To simplify the simulations, we constraint to only family and individual level components.
- **State**: Three simple states for individuals
  - Pro-China opinion

    - Against China opinion
    - Neutral opinion
- **Relationships between components**
    - Before setting the relationships between components, we can start by identifying which features will be crucial to determine an individual's behavior.
        - The initial positive attitude toward China i.e., 0 - 33% is considered against China, 34% - 66% is considered neutral, 66%- 100% is considered pro China
        - Stubbornness tendency: How stubborn is a person holding their prior belief? i.e., 0(will always change) to 1 (never change their opinion)
        - Collectivist tendency: How likely will a person change their opinion by the people around them?
    - To set up the relationship, we can hypothesize some simple rules on how individuals' opinions can be swayed by observing the social phenomenon. For example,
        - The family has a stronger influence on individuals' prior beliefs.
        - Individuals will randomly interact with other individuals in the morning, and spend time with their families in the evening.
        - Individuals might gradually change their beliefs based on their stubbornness tendency and collectivist tendency.
- **Changes over time:** In the beginning, we will expect lots of clusters of the same beliefs that represent different initial family opinions. Over time, individuals will interact with each other which might strengthen or weaken the belief.
- **Prediction:** As time evolves, we can model the percentage of different opinions in Taiwanese society.

Problem 2: Is the `random` module actually *random*?

- LOs: #EmpiricalAnalysis
- HCs: #confidenceintervals (for the optional challenge question)
- Words: 100–200

We check if NumPy's `random` module generates values that appear to be random.[2] If the model is random, we should get integers selected from a range at random approximately equally often. Use the code below to generate an array of 1,000,000 random integers from 0 to 99.

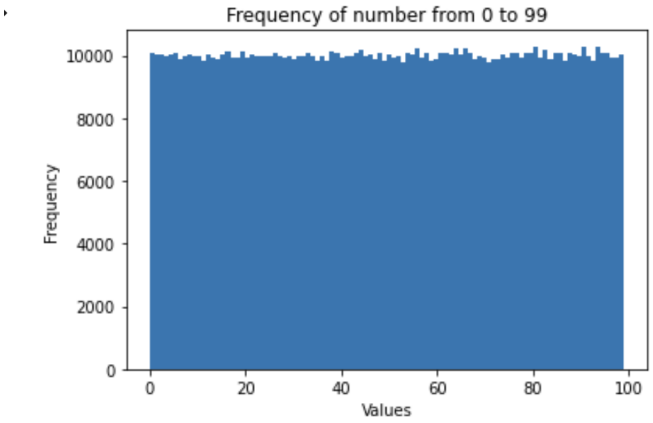
```
import numpy as np
numbers = np.random.randint(100, size=1000000)
```

Then complete the following tasks.

1. Make a properly formatted histogram of the number of times each value appears.
2. Discuss in detail how often you expect each value to appear, how often each value actually appeared, and whether the distribution over the results matches what you would expect according to the Central Limit Theorem.
3. Optional challenge: Derive (analytically) a 95% confidence interval of the mean count of 1,000,000 randomly selected integers from 0 to 99 that equal a particular value. Explain how many of your 100 histogram values should fall within this confidence interval. Count how many of your 100 histogram values do actually fall within this confidence interval. Compare and comment on the results.

```
numbers = np.random.randint(100, size=1000000)
```

```
plt.hist(numbers,  bins = 100)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Frequency of number from 0 to 99")
plt.show()
```



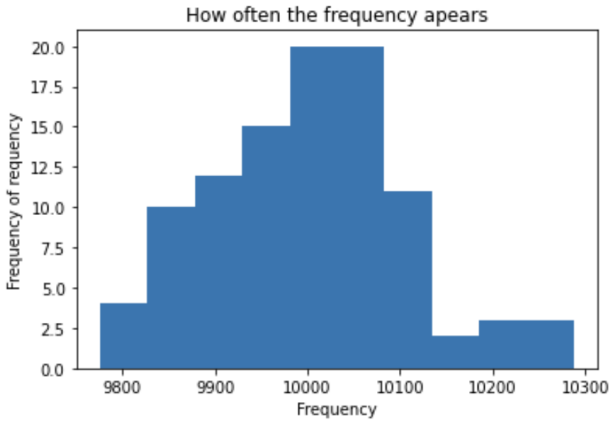

```
exp_freq = 1000000/100
freq_dic = collections.Counter(numbers)
freqs = list(freq_dic.values())
```

```
mean_freq = np.mean(freqs)
std_freq = np.std(freqs)
```

```
print(f'Expected frequency: {exp_freq}')
print(f'Actual frequency mean: {mean_freq}. Actual frequency variance: {round(std_freq,
2)}.')
```

```
Expected frequency: 10000.0
Actual frequency mean: 10000.0. Actual frequency standard deviation: 106.26.
```

```
plt.hist(freqs)
plt.xlabel("Frequency")
plt.ylabel("Frequency of requency")
plt.title("How often the frequency apears")
plt.show()
```



Initially, I expect that every value will have the same frequency by a uniform distribution which is 10000 times. Observing from the simulation, we can see that most numbers appear around 10000 times, ranging from 9600 to 10300. This aligned with CLT because we will probably mostly get 10000 times but with some variation, making the distribution a normal distribution. Based on CLT, if we have even more simulations, we will have a higher peak at 10000 with a small variance.

Optional Challenge

We can assume that we randomly select integers from 0 to 99 that are equal to 0. I will assume that we obtain a binomial distribution because we either succeed at sampling 0 or not. Since we randomly select 1,000,000 times, we should be selecting 1,000,000*1/100 = 10,000 times of 0.

The variance will be 10,000,000*1/100*99/100 = 99,000, and the standard deviation will be 314.6. If we want to compute 95% CI, we will use mean - 1.96 * std/sqrt(n) and mean + 1.96 * std/sqrt(n), we will derive the CI is [9999.38, 10000.62], indicating that 95% of the when we sample, our mean counts should be fall within this value.

Problem 3: Forest fire mean-field approximation

- LOs: #TheoreticalAnalysis
- Words: no specific word count; mostly math

Consider the forest fire model described in Drossel, B., & Schwabl, F. (1994). Formation of space-time structure in a forest-fire model. Each site of a 2-dimensional grid is occupied by a tree, a fire (a burning tree), or it is empty. During each discrete update, the following rules apply:

- fire → empty, with probability 1.
- tree → fire, with probability $1 - g$ if at least one neighbor in its Von Neumann neighborhood is a fire. (Think of g as the immunity of a tree to catching fire.)
- tree → fire, with probability $(1 - g)\, f$ if no neighbor is burning. (Think of f as the probability of a lightning strike or some other cause of a spontaneous fire–a small probability quite close to 0.)
- empty → tree, with probability p.
- Otherwise, the state of the cell remains the same.

Prove the equations for the change of the mean-field densities during one timestep:

$$\Delta\rho_e = \rho_f - p\rho_e$$

$$\Delta\rho_t = p\rho_e - \rho_t(1-g)\Big(f + (1-f)(1-(1-\rho_f)^{2d})\Big)$$

$$\Delta\rho_f = -\rho_f + \rho_t(1-g)\Big(f + (1-f)(1-(1-\rho_f)^{2d})\Big)$$

Note that $d = 2$ is the dimension of the grid.

To construct the equation, we start by defining the variables.

- g : immunity rate
- ρ_e : mean density of empty sites
- ρ_t : mean density of trees
- ρ_f : mean density of burning trees in a steady state
- f : the probability of firing (close to 0)
- p : the probability of tree growing

Here, we know

- The cycle of a land: empty site → tree grow → tree burn → empty site.

- $\rho_e + \rho_t + \rho_f = 1$

Based on the variables and the rules, we know that for the next time step,

- Changes in the mean density of empty sites: Trees die after burning - Trees grow from empty sites
- Changes of the mean density of trees = Trees grow from empty sites - (Trees burn by themselves + Trees burn due to neighbor burning trees)
- Changes in the mean density of burning trees: Trees burn by themselves + Trees burn due to neighbors burning trees - trees die after burning

Translating into math function,

- $\Delta\rho_e$
  - Trees die when they are burning: ρ_f
    - A tree will always die when they burn (the probability of dying is 1), so we multiply 1 by the mean density of burning sites
  - Trees grow from empty sites: $p\rho_e$
    - A tree has a probability p of growing, and we multiply with the mean density of empty sites
  - Conclusion: $\Delta\rho_e = \rho_f + p\rho_e$
- $\Delta\rho_t$
  - Trees grow from empty sites: $p\rho_e$
    - A tree has a probability p of growing, and we multiply with the mean density of empty sites
  - Tree burn by itself when no neighbor is burning: $\rho_t(1-g)f(1-\rho_f)^{2d}$
    - When a tree burns by itself, it is under the condition that no neighbor around it is burning. The probability of no neighbor around it is burning based on Von Neumann neighborhood is $(1-\rho_f)^4$. Since d =2, $(1-\rho_f)^4$ can be denoted as $(1-\rho_f)^{2d}$.
    - The probability of a tree starting to burn through lightning is $(1-g)f$.
    - Hence, the joint probability is $\rho_t(1-g)f(1-\rho_f)^{2d}$

- Tree burn by because at least one neighbor is burning: $\rho_t(1-g)(1-(1-\rho_f)^{2d})$
  - The probability of at least one neighbor is burning is 1 - P(no neighbor is burning) = $1-(1-\rho_f)^{2d}$
  - The probability of the tree starting burning is $(1-g)$.
  - Hence, the joint probability is $\rho_t(1-g)(1-(1-\rho_f)^{2d})$
- Conclusion:$\Delta\rho_t = p\rho_e - \rho_t(1-g)f(1-\rho_f)^{2d} - \rho_t(1-g)(1-(1-\rho_f)^{2d}) = p\rho_e - \rho_t(1-g)(f+(1-f)(1-(1-\rho_f)^{2d}))$

- $\Delta\rho_f$
  - The tree die after burning: ρ_f
  - Tree burn by itself when no neighbor is burning: $\rho_t(1-g)f(1-\rho_f)^{2d}$
  - Tree burn by because at least one neighbor is burning: $\rho_t(1-g)(1-(1-\rho_f)^{2d})$
  - Conclusion: $\Delta\rho_f = -\rho_f + \rho_t(1-g)f(1-\rho_f)^{2d} + \rho_t(1-g)(1-(1-\rho_f)^{2d}) == -\rho_f + \rho_t(1-g)(f+(1-f)(1-(1-\rho_f)^{2d}))$

Problem 4: Wireworld

- LOs: #PythonImplementation
- HCs: #algorithms (for the optional challenge question)
- Words: no specific word count; mostly code

The Wireworld model is a cellular automaton that can be used to simulate simple electrical circuits. Even though these circuits are simple, the Wireworld CA is Turing-complete. This

means you can use it to simulate a general-purpose computer like your laptop–assuming you have enough storage, computation, and patience to run the CA.

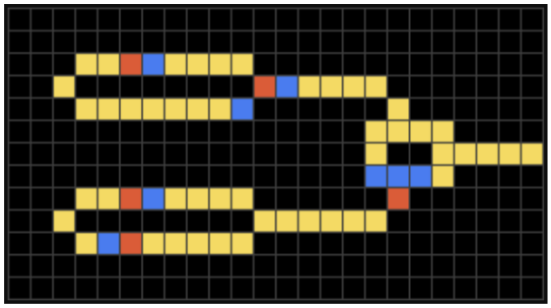


Figure 1. An example of a Wireworld state. Black cells represent the background, yellow cells represent wires, and blue and red cells represent electrons (head and tail, respectively). See the Wikipedia source image for an animation showing how electrons move around the circuit.

Each cell is in 1 of 4 states–

0. background
1. electron head
2. electron tail
3. wire

In each discrete time step, the update rules are as follows.

- *Background* remains *background*.
- *Electron head* changes to *electron tail*.
- *Electron tail* changes to *wire*.
- *Wire* changes to *electron head* if and only if one or two of its neighbors are *electron head*. Otherwise, it remains *wire*. Consider a Moore neighborhood.

```
import matplotlib.pyplot as plt
from matplotlib import animation
import numpy as np
import matplotlib.colors
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
from tqdm import tqdm
import copy
```

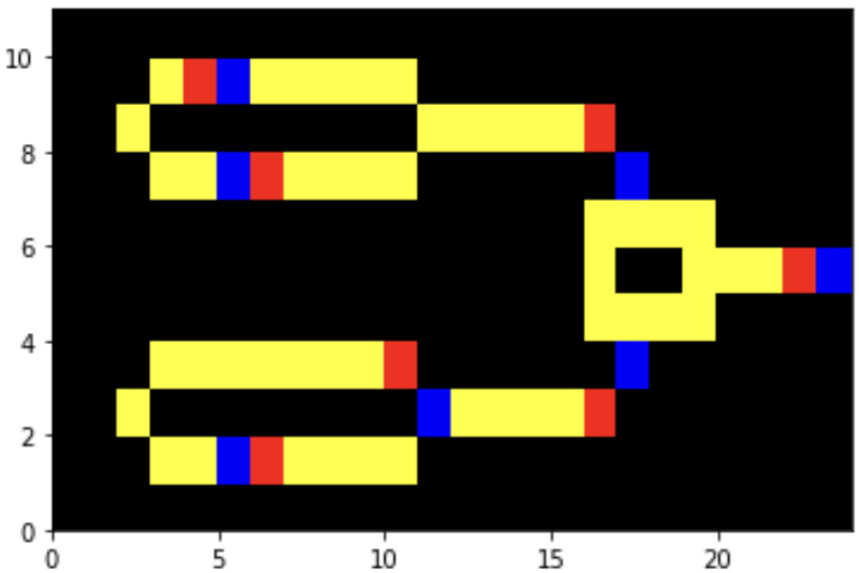
```
class Wireworld:
    '''
    Wireworld simulation

    Parameters
```

```
    ----------
    initial_setup: string
        The initial wire setup


    Attributes
    ----------
    current_frame: list[list[int]]
        The current frame of the simulation
    rows: int
        The width of the simulation space
    cols: int
        The height of the simulation space
    fig, axes: matplotlib.subplot
        The figure and its corresponding axes
    '''
    def __init__(self, initial_setup):
        self.current_frame = initial_setup
        self.rows = len(self.current_frame)
        self.cols = len(self.current_frame[0])
        self.figure, self.axes = plt.subplots()

    def update_neighbors(self, x,y, last_frame):
        '''
        Update the state of a single cell

        Parameters
        ----------
        x: int
            position x of the input cell
        y: int
            position y of the input cell
        last_frame: list
            configeration of the previous frame

        Returns
        ----------
        None
        '''
        # print("initial setup", x, y)
        if self.current_frame[x][y] == 0:
            self.current_frame[x][y] = 0
        elif self.current_frame[x][y] == 1:
            self.current_frame[x][y] = 2
        elif self.current_frame[x][y] == 2:
            self.current_frame[x][y] = 3
        elif self.current_frame[x][y] == 3:
            head_count = 0
            for dx in range(-1, 2):
                for dy in range(-1, 2):
                    # print("x+dx, y+dy", x+dx, y+dy)
                    # print("dx, dy", dx, dy)
                    if (x + dx < 0) or (x + dx >= self.rows) or (y + dy < 0) or (y + dy >=
self.cols):
```

```
                        # print("not running dx, dy", dx, dy)
                        continue
                    if last_frame[x+dx][y+dy] == 1:
                        # print("add headcount")
                        # print("x+dx, y+dy", x+dx, y+dy)
                        head_count += 1
            # print("position", x, y, "head_count", head_count)
            if (head_count == 1) or (head_count == 2):
                # print("convert", x, y, "value",self.initial_setup[x][y])
                self.current_frame[x][y] = 1

    def update_frame(self):
        '''
        Update the state of the entire frame

        Parameters
        ----------
        None

        Returns
        ----------
        None
        '''
        last_frame = copy.deepcopy(self.current_frame)
        for x in range(self.rows):
            for y in range(self.cols):
                self.update_neighbors(x, y, last_frame)


    def create_visualization(self):
        '''
        Create a visualization plot of the state

        Parameters
        ----------
        None

        Returns
        ----------
        plot
            The plot of the current wireworld state
        '''
        # fig, axes = plt.subplots()
        cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", ["black","blue","re
d","yellow"])
        plot = self.axes.pcolor(self.current_frame, cmap = cmap)
        # plot = axes.pcolor(self.initial_setup, cmap = cmap)
        plt.show()
        return plot
```

```
def animate(simulation, total_frames = 100, interval = 200):
    """
    Create a animation from the simulation
    """
    def update(i):
        simulation.update_frame()
        return [simulation.create_visualization()]
    animated_sim = animation.FuncAnimation(simulation.figure, update,
                                           init_func=lambda: [],
                                           frames=total_frames, interval=interval)
    output = HTML(animated_sim.to_html5_video())
    simulation.figure.clf()
    return output
```

```
initial_setup = [[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, 3, 2, 1, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 2, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 0, 3, 3, 1, 2, 3, 3, 3, 3, 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, 3, 3, 3, 3, 0, 0, 0, 0],
                 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 3, 3, 3, 2, 1],
                 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 0, 0, 0, 0],
                 [0, 0, 0, 3, 3, 3, 3, 3, 3, 3, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, 3, 3, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 0, 3, 3, 1, 2, 3, 3, 3, 3, 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, 0, 0, 0, 0, 0, 0, 0, 0]
                 ]
initial_setup.reverse()
wireworld = Wireworld(initial_setup)
wireworld.create_visualization()
```



animation code is adapted from Class code

```
animate(wireworld)
```

You can view the animation: here

From the animation, we can see that it demonstrates the same behavior as the behavior from Wikipedia.