香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

人工智能學院
School of Artificial Intelligence

Lab 2:
Simulate a basic city model

(AIE1901)

Course Overview:

- Understanding how individual vs. collective decision rules shape social patterns (e.g., segregation vs. mixing).
- Focus a Python implementation of the Schelling-type model, step by step
- Finally, put all modules together in the notebook to reproduce the key results from the PNAS article (segregation vs. mixing transition).

Learning Objectives:

By the end of this session, you should be able to:

- Explain the role of each Python modules in the simulation pipeline.
- Relate the code functions (Δu, ΔU, α, T) to theoretical concepts from the PNAS paper.
- Run and interpret basic simulations of residential moves.Visualize outcomes (heatmaps, 3D surfaces) and connect them to social science concepts like segregation, cooperation, and externalities.
- Critically discuss: Why self-interested behavior may lead to inefficient collective outcomes; How cooperation (α) and randomness (T) influence system dynamics.

Before starting

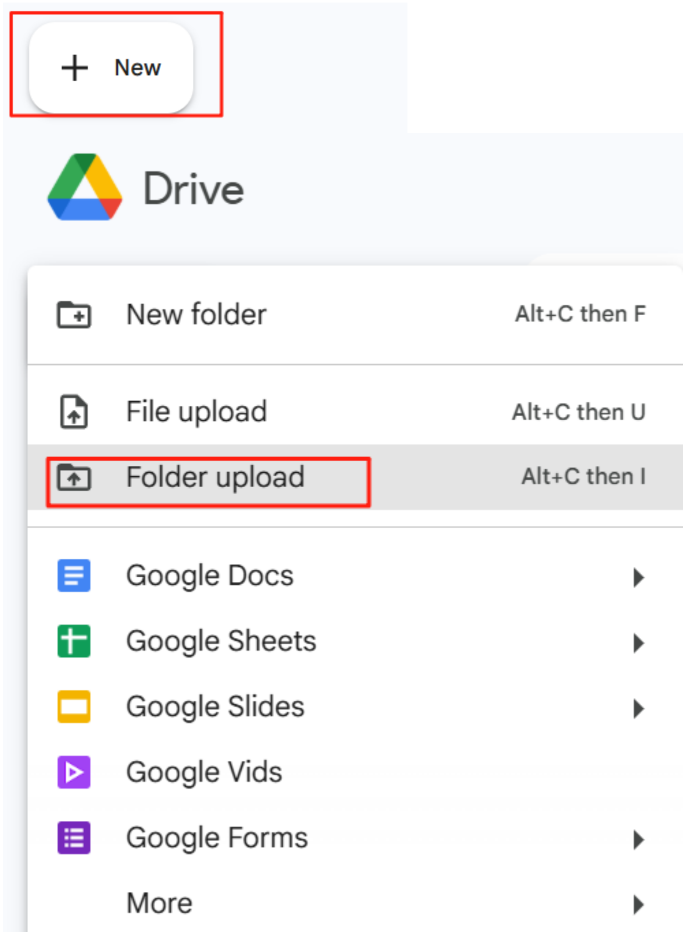
- Please connect to the CUHKSZ Wi-Fi while on campus,
- Please use the school VPN when off campus
- For more details: https://itso.cuhk.edu.cn/node/169

Step 1: Get lab files

- In your browser, go to Blackboard.
- Locate **Lab 2** and download the **Schelling-simulation-for-Lab 2.zip**
- **Unzip this file in your PC (keep the original filenames and file structure).**

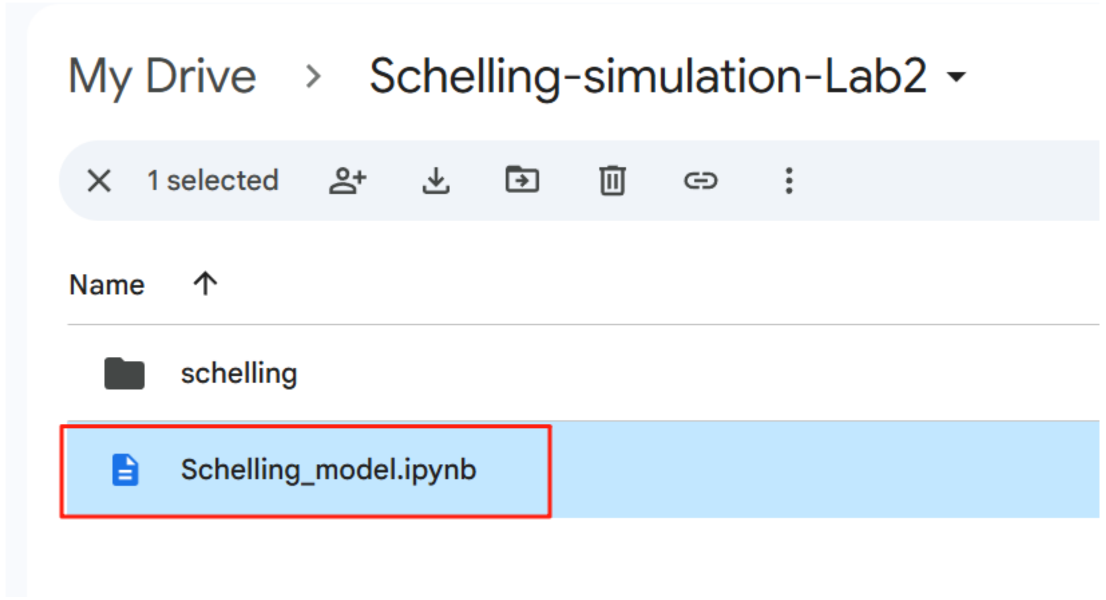
Step 2: Upload Lab files to Google Drive

- Unzip the Schelling-simulation-Lab2.zip file you just downloaded.
- In your browser, go to: https://drive.google.com/
- Click '+ New' in the upper left corner.
- Click 'Folder upload'
- Upload the Schelling-simulation-Lab2 folder you just unzipped.



Step 3: Open the *ipynb* in Google Colab.

- Double-click the ipynb file.
- It will automatically open in Google Colab.

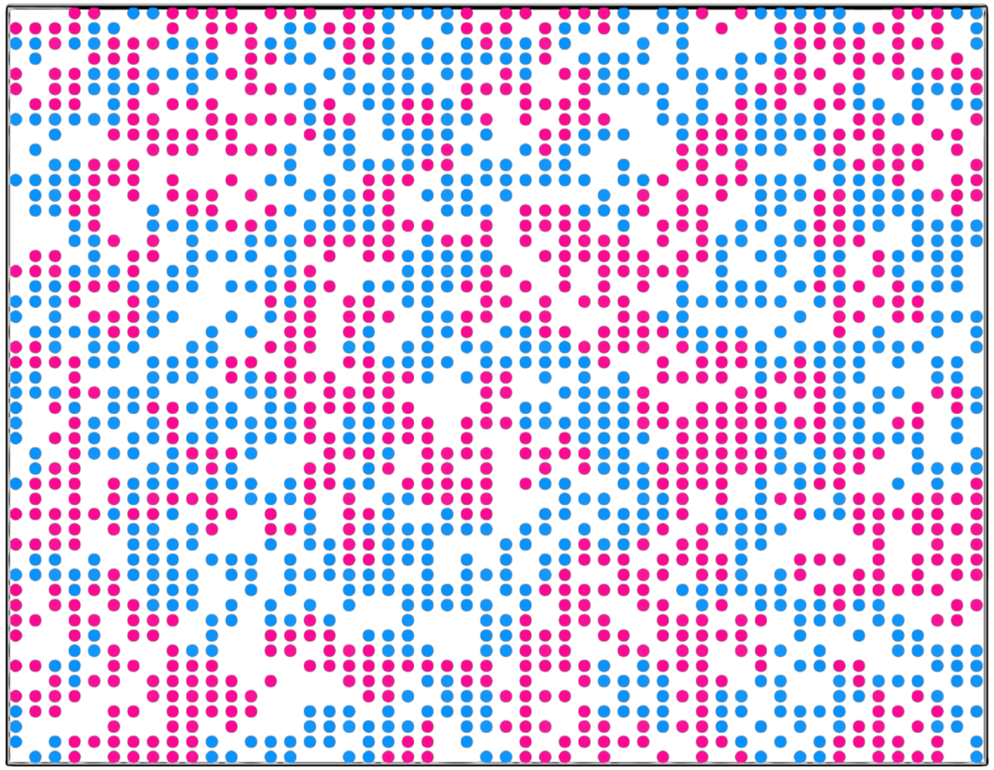

My Drive › Schelling-simulation-Lab2

💡 Tip: Using Google Chrome may provide the smoothest experience when accessing Google services.

The Schelling Model of Segregation

- Introduced by **Thomas Schelling (1970s)**
- A **foundational agent-based model** for studying spatial segregation
- Shows how **mild local preferences** can lead to **large-scale segregation**
- Agents of different types (e.g., colors/groups) occupy a grid
- Move if their neighborhood satisfaction threshold is not met
- Despite its simplicity, reveals **emergent collective behavior**
- Widely applied in **economics, sociology, and urban studies**

Schelling Model with two colours: Final State with Happiness Threshold 30%



Segregated neighbourhoods

💡 **In this lab:** We will implement and explore key variants of the Schelling model

Parameters Recap

Parameter	Explanation / Analogy
Q	Number of neighborhoods (blocks) → city cut into Q small grids
H	Capacity per neighborhood (max residents) → like H apartments per block
ρ (rho)	Density of a block = residents ÷ H → 0 = empty, 1 = full
ρ_0 (rho0)	Average citywide density → e.g., 0.5 = "half full on average"
u(ρ)	Individual utility at density ρ → symmetric: prefer half full; asymmetric: tilted preference
U	Collective utility = sum of all individuals' utility
α (alpha)	Cooperation / altruism → 0 = selfish, 1 = collective, in-between = mixed
T (temperature)	Randomness / noise → higher T = more chance-based, suboptimal moves possible
m	Asymmetry of utility → larger m = more tolerant of high density, needs higher α to mix

Key Components:

- **Initialization strategies**: how the city and agents are initially distributed
  – *e.g.*, random placement, balanced layout, half-full blocks
- **Agent selection**: which agent moves next
  – *e.g.*, random agent, agent with lowest utility
- **Target selection**: where the agent may want to move
  – *e.g.*, random vacant cell, block with highest utility
- **Move acceptance rule**: when a move is allowed
  – *e.g.*, only if personal utility increases, or social welfare increases, or via Metropolis-Hastings
- **Stopping criteria**: when to halt the simulation
  – *e.g.*, fixed number of steps, or no more accepted moves

Modular Code Organization: schelling/

Our implementation is organized as a **modular package**, where each simulation component is placed in its own file.

This makes the code easier to **understand, maintain, and extend.**

- **initialization.py** – how agents are initially placed in the city
- **agent_selector.py** – strategy for choosing which agent moves
- **target_selector.py** – strategy for choosing where the agent might move
- **move_acceptor.py** – rule for deciding whether a move is accepted
- **stop_criterion.py** – when to stop the simulation
- **utils.py** – shared utility functions and plotting tools

💡 Each module provides a **baseline implementation** that is simple but functional, and can be easily replaced to explore new model variants.

initialization.py — Randomly Fill a City to a Given Average Density

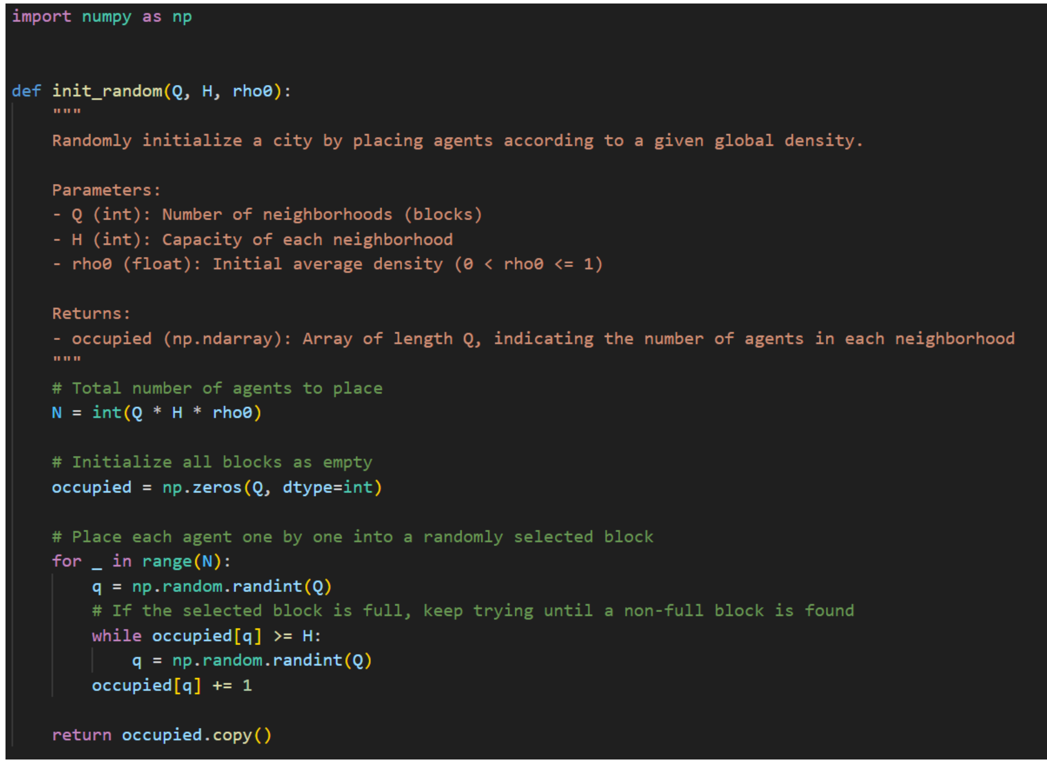
- Given a city divided into Q blocks, each with capacity H, and a target average density ρ_0, the function randomly places people into blocks according to that density.

- Inputs (3 key parameters):
  - Number of blocks Q
  - Capacity per block H
  - Target average density $\rho_0 \in (0,1]$

e.g. $Q = 36$, $H = 100$, $\rho_0 = 0.5 \rightarrow$ Place $36 \times 100 \times 0.5 = 1800$ people

- **Output:** An array occupied of length Q, where each entry records how many people live in that block.

💡 **Why random?** This gives us an unbiased starting point. In later models, people will “move.” Starting from randomness allows us to compare how different movement rules lead cities to spontaneously form mixed vs. segregated patterns.



```python
import numpy as np


def init_random(Q, H, rho0):
    """
    Randomly initialize a city by placing agents according to a given global density.

    Parameters:
    - Q (int): Number of neighborhoods (blocks)
    - H (int): Capacity of each neighborhood
    - rho0 (float): Initial average density (0 < rho0 <= 1)

    Returns:
    - occupied (np.ndarray): Array of length Q, indicating the number of agents in each neighborhood
    """
    # Total number of agents to place
    N = int(Q * H * rho0)

    # Initialize all blocks as empty
    occupied = np.zeros(Q, dtype=int)

    # Place each agent one by one into a randomly selected block
    for _ in range(N):
        q = np.random.randint(Q)
        # If the selected block is full, keep trying until a non-full block is found
        while occupied[q] >= H:
            q = np.random.randint(Q)
        occupied[q] += 1

    return occupied.copy()
```

agent_selector.py — Deciding “Who Moves Next”

Problem it solves: In each simulation step, we must first “pick” one resident. Which block does this resident come from?

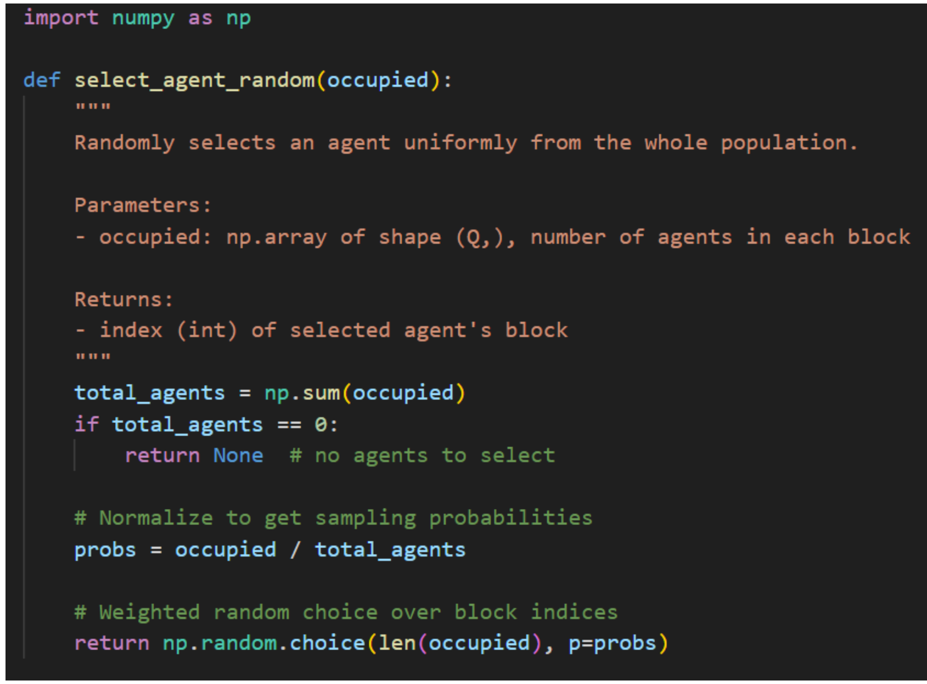
Core logic:

•Blocks with **more people** → higher chance of being selected
•Blocks with **fewer people** → lower chance
•If the city is empty → return None

Input: occupied, an array of length Q, with the number of residents in each block

Output: An **integer index** indicating which block the selected resident comes from

💡 Residents in larger blocks are more likely to be picked to move.



```python
import numpy as np

def select_agent_random(occupied):
    """
    Randomly selects an agent uniformly from the whole population.

    Parameters:
    - occupied: np.array of shape (Q,), number of agents in each block

    Returns:
    - index (int) of selected agent's block
    """
    total_agents = np.sum(occupied)
    if total_agents == 0:
        return None  # no agents to select

    # Normalize to get sampling probabilities
    probs = occupied / total_agents

    # Weighted random choice over block indices
    return np.random.choice(len(occupied), p=probs)
```

香 港 中 文 大 學 (深 圳)
The Chinese University of Hong Kong, Shenzhen
人工智能學院
School of Artificial Intelligence

target_selector.py — Deciding “Where to Move”

Once a resident is chosen to move, we must decide: **Which block do they go to?**

Two strategies:

1. random_block (uniform over blocks):

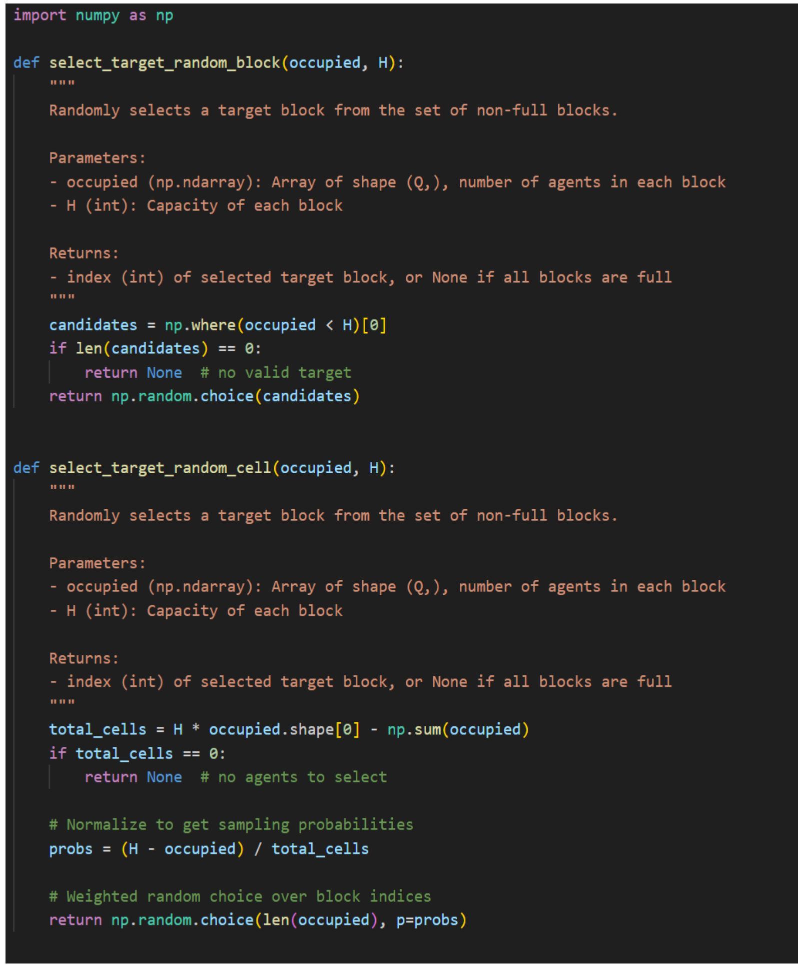
- Randomly pick from all blocks that are **not full**
- Number of empty spots does **not** matter

2. random_cell (weighted by vacancies):

- Every empty house has the same chance of being chosen
- Blocks with more vacancies are **more likely** to be selected

Input: occupied (residents per block)
H (capacity per block)
Output: An **integer index** for the target block, or None if the city is already full

💡 **Choose the destination:** either **pick a block at random**,
or **weight by how many empty houses it has**.



```python
import numpy as np

def select_target_random_block(occupied, H):
    """
    Randomly selects a target block from the set of non-full blocks.

    Parameters:
    - occupied (np.ndarray): Array of shape (Q,), number of agents in each block
    - H (int): Capacity of each block

    Returns:
    - index (int) of selected target block, or None if all blocks are full
    """
    candidates = np.where(occupied < H)[0]
    if len(candidates) == 0:
        return None  # no valid target
    return np.random.choice(candidates)


def select_target_random_cell(occupied, H):
    """
    Randomly selects a target block from the set of non-full blocks.

    Parameters:
    - occupied (np.ndarray): Array of shape (Q,), number of agents in each block
    - H (int): Capacity of each block

    Returns:
    - index (int) of selected target block, or None if all blocks are full
    """
    total_cells = H * occupied.shape[0] - np.sum(occupied)
    if total_cells == 0:
        return None  # no agents to select

    # Normalize to get sampling probabilities
    probs = (H - occupied) / total_cells

    # Weighted random choice over block indices
    return np.random.choice(len(occupied), p=probs)
```

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

人工智能學院
School of Artificial Intelligence

move_acceptor.py — Deciding "Move or Stay"

Three levels of decision rules:

1.Individual version — accept_if_personal_utility_improves

Move only if **personal utility increases**: $u(to) > u(from)$

2. Accelerator — build_marginal_table

Speeds up calculation

3.Comprehensive version — accept_metropolis (with α, T)

Compute personal change Δu and social change ΔU

Composite utility combine into: $G = \Delta u + \alpha(\Delta U - \Delta u)$

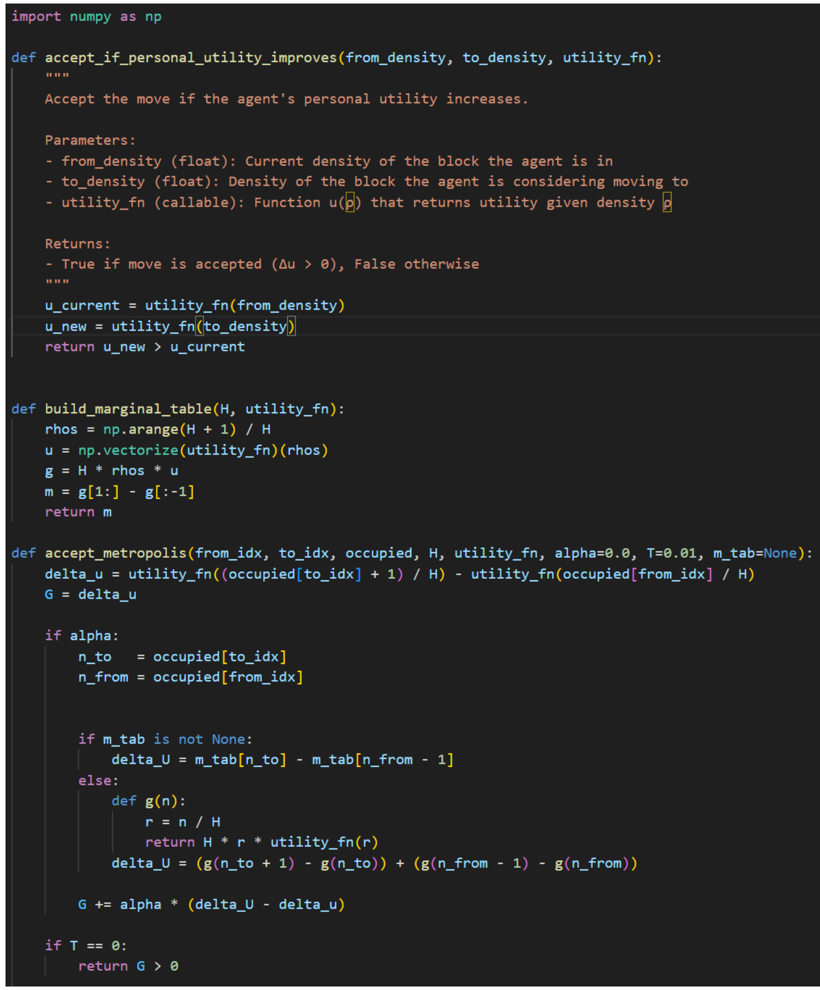
Zero temperature (T=0): move if G > 0

Finite temperature (T>0): move with probability depending on G and T

💡 **Parameters:**

α ("altruism / cooperation"): how much weight to give others' utility

T ("randomness / noise"): higher T → more chance-based moves



```python
import numpy as np

def accept_if_personal_utility_improves(from_density, to_density, utility_fn):
    """
    Accept the move if the agent's personal utility increases.

    Parameters:
    - from_density (float): Current density of the block the agent is in
    - to_density (float): Density of the block the agent is considering moving to
    - utility_fn (callable): Function u(ρ) that returns utility given density ρ

    Returns:
    - True if move is accepted (Δu > 0), False otherwise
    """
    u_current = utility_fn(from_density)
    u_new = utility_fn(to_density)
    return u_new > u_current


def build_marginal_table(H, utility_fn):
    rhos = np.arange(H + 1) / H
    u = np.vectorize(utility_fn)(rhos)
    g = H * rhos * u
    m = g[1:] - g[:-1]
    return m

def accept_metropolis(from_idx, to_idx, occupied, H, utility_fn, alpha=0.0, T=0.01, m_tab=None):
    delta_u = utility_fn((occupied[to_idx] + 1) / H) - utility_fn(occupied[from_idx] / H)
    G = delta_u

    if alpha:
        n_to   = occupied[to_idx]
        n_from = occupied[from_idx]


        if m_tab is not None:
            delta_U = m_tab[n_to] - m_tab[n_from - 1]
        else:
            def g(n):
                r = n / H
                return H * r * utility_fn(r)
            delta_U = (g(n_to + 1) - g(n_to)) + (g(n_from - 1) - g(n_from))

        G += alpha * (delta_U - delta_u)

    if T == 0:
        return G > 0
```

stop_criterion.py — Deciding "When to Stop"

Problem it solves:
The simulation cannot run forever — we need a **stopping condition**.

Strategy used here:

Fixed number of steps

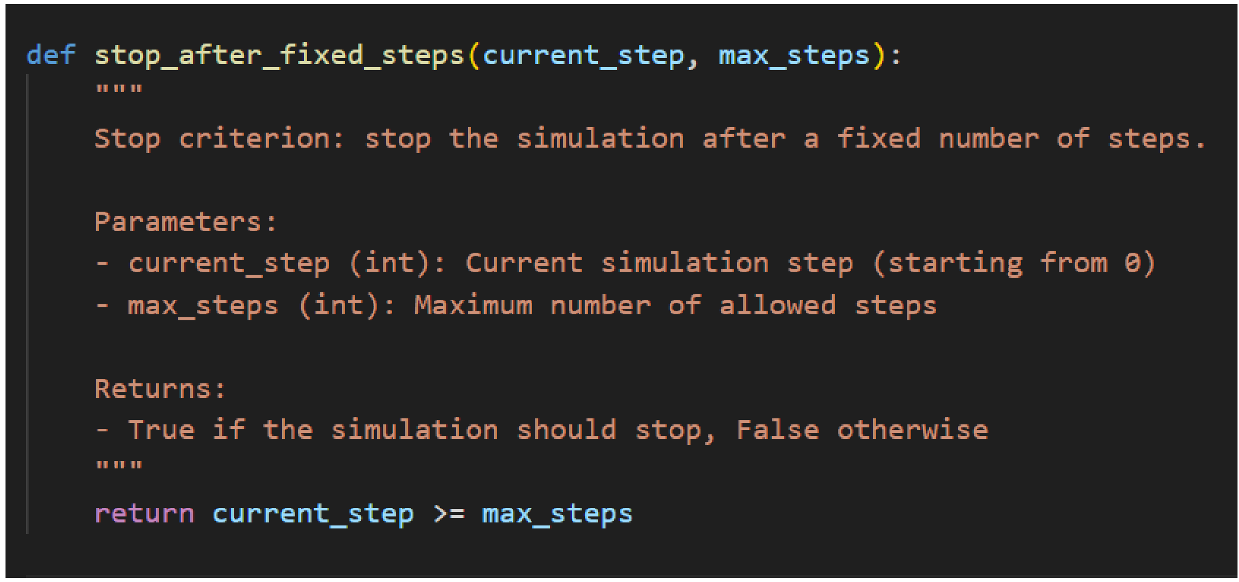
A maximum step count max_steps is given (e.g., 100,000)

At each step:

- If current_step ≥ max_steps → **Stop**
- Else → **Continue**

Input: current_step: current step number;
max_steps: maximum number of steps

Output: True → Stop;
False → Continue simulation



```python
def stop_after_fixed_steps(current_step, max_steps):
    """
    Stop criterion: stop the simulation after a fixed number of steps.

    Parameters:
    - current_step (int): Current simulation step (starting from 0)
    - max_steps (int): Maximum number of allowed steps

    Returns:
    - True if the simulation should stop, False otherwise
    """
    return current_step >= max_steps
```

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen
人工智能學院
School of Artificial Intelligence

utils.py (Part 1)—Utility Functions: Scoring "Preference for Density"

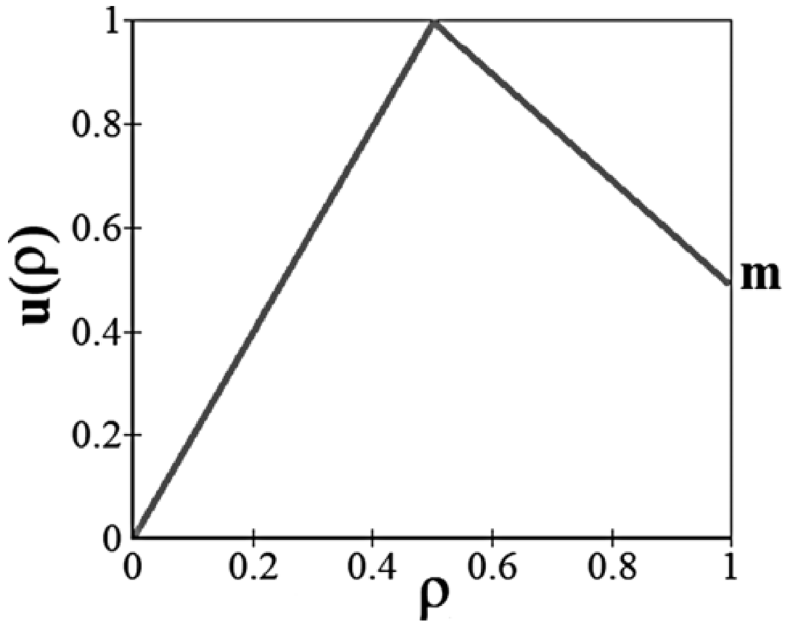
What do we want to express?
For each neighborhood density **ρ**, compute the resident's **satisfaction** *u(ρ)*.

Symmetric triangle — triangle_utility

- Peak at ρ = 0.5
- Closer to 0.5 → happier;
- farther → less happy

Asymmetric triangle — asymmetric_triangle_utility

- Peak position/height adjustable
- End-point values adjustable
- Left slope ≠ Right slope
- Matches **asymmetric preferences** in the paper (e.g., steeper or wider on one side)



💡 **Key idea:** *u(ρ) translates "psychological preferences" into computable numbers.*

香港中文大學(深圳) The Chinese University of Hong Kong, Shenzhen
人工智能學院 School of Artificial Intelligence

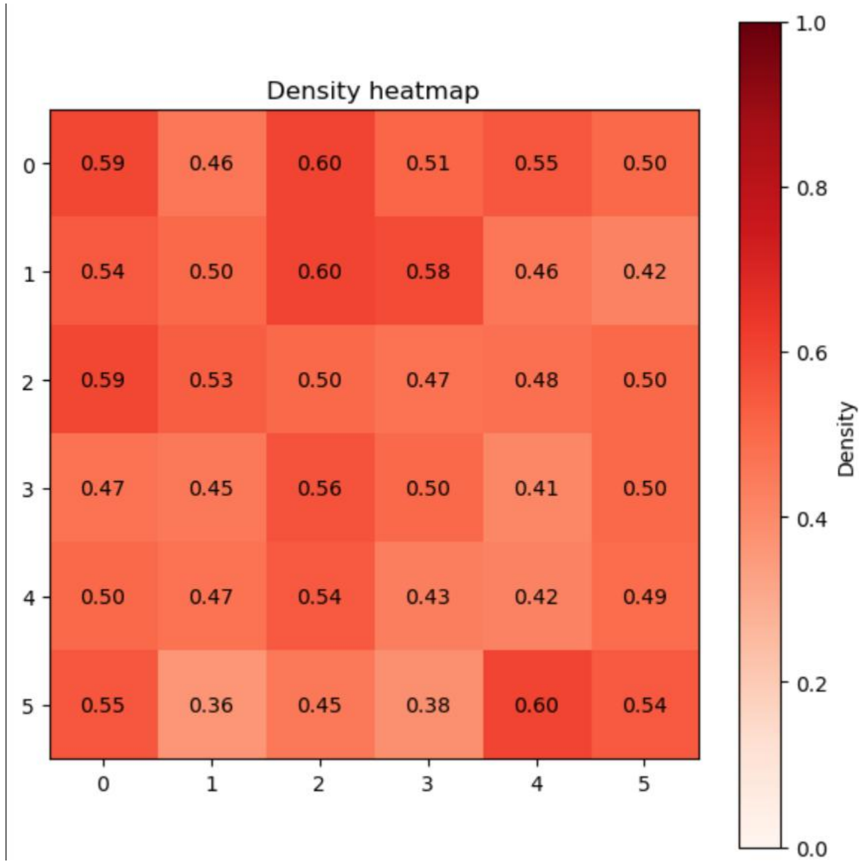
utils.py (Part 2)

—Heatmaps & Line Charts: 2D Views of “Where It’s Crowded, How It Evolves”

1. Heatmap — plot_density_heatmap
Input: grid size (rows × cols) + 1D density array
Output:

- Darker red = more crowded
- Each cell labeled with its value
- Colorbar included

Use case: compare spatial patterns under different **α / T** values



	0	1	2	3	4	5
0	0.59	0.46	0.60	0.51	0.55	0.50
1	0.54	0.50	0.60	0.58	0.46	0.42
2	0.59	0.53	0.50	0.47	0.48	0.50
3	0.47	0.45	0.56	0.50	0.41	0.50
4	0.50	0.47	0.54	0.43	0.42	0.49
5	0.55	0.36	0.45	0.38	0.60	0.54

Now, let's turn to the *.ipynb* notebook file.

Notebook Overview (.ipynb)

Purpose:

Combine all the .py modules and run a **complete simulation** end-to-end

Three main parts:

1.Initialization

Set parameters **Q, H, ρ_0**, etc.

Build an “empty city” and randomly place agents

2.Simulation loop

Repeatedly perform: **select source** → **select target** → **decide move**

3.Results visualization

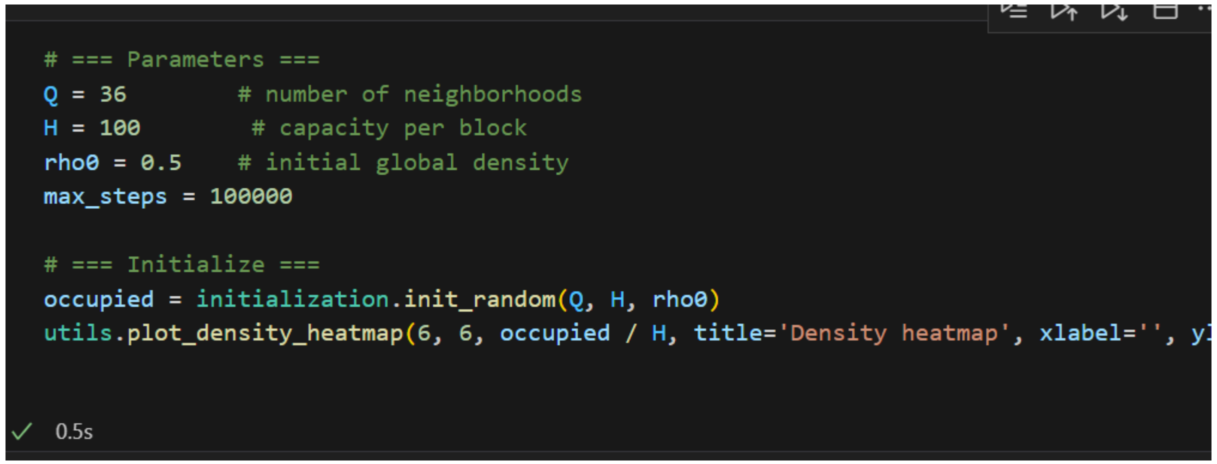
Inspect outcomes with **heatmaps, line charts, phase diagrams (3D surfaces)**

💡 **Notebook = our “laboratory bench”**

City Initialization

What it does:

- **Input parameters:** number of neighborhoods Q, capacity H, average density ρ_0
- Randomly assign residents into neighborhoods



```
# === Parameters ===
Q = 36          # number of neighborhoods
H = 100          # capacity per block
rho0 = 0.5     # initial global density
max_steps = 100000

# === Initialize ===
occupied = initialization.init_random(Q, H, rho0)
utils.plot_density_heatmap(6, 6, occupied / H, title='Density heatmap', xlabel='', y
```

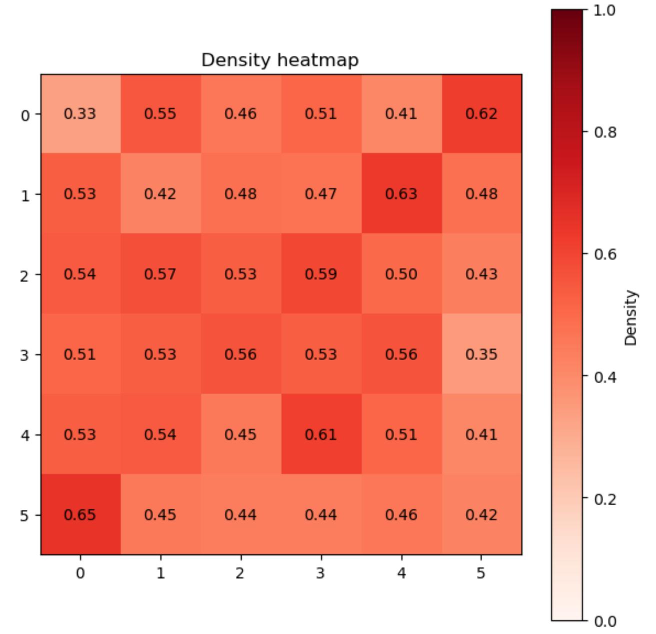
✓ 0.5s

Why do this?

- To give the city an **unbiased starting point**
- Like "scattering beans" randomly into blocks

What can we see?

- Use a **heatmap** to check: which neighborhoods are crowded, which are sparse
- This is the **initial state**, before any dynamic evolution begins



香港中文大學(深圳) The Chinese University of Hong Kong, Shenzhen

人工智能學院 School of Artificial Intelligence

Baseline: Purely Self-Interested

Purpose:

- A baseline experiment where agents care **only about themselves**
- **Rules:** move **only if personal utility improves** (Δu > 0)

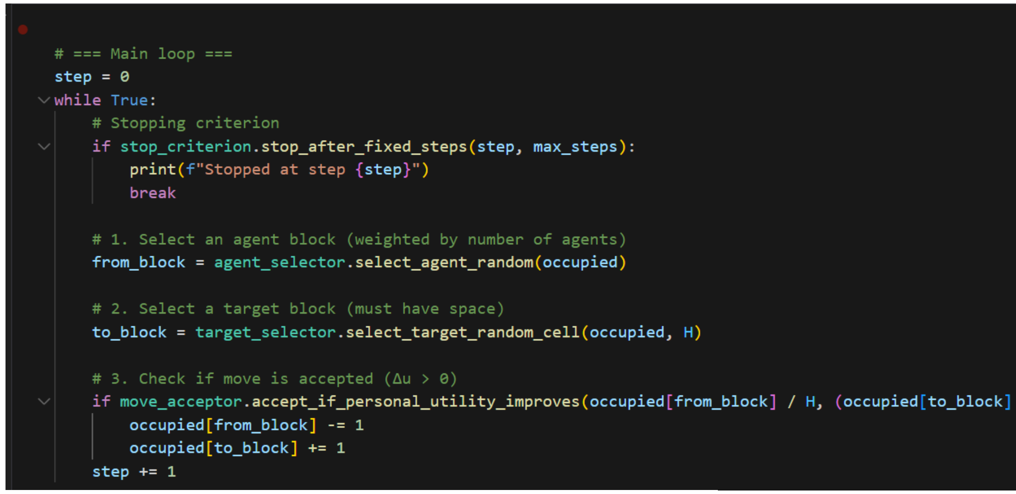
Utility function: symmetric triangle *u(ρ)* (peak at half full)

Observed phenomenon:

- City gradually develops **segregated patterns** (some blocks full, others empty)
- Overall collective utility remains **low**

Corresponding to paper:

- **α = 0** (purely selfish)

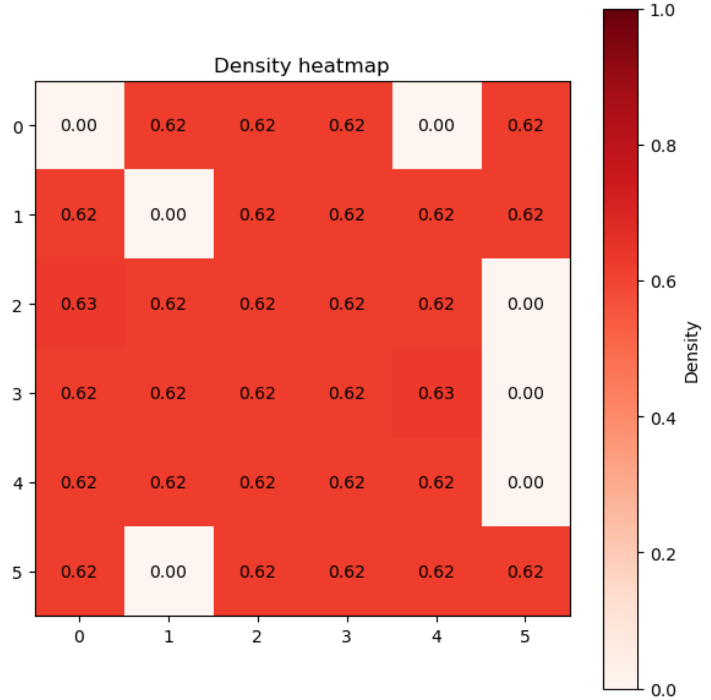


```
# === Main loop ===
step = 0
while True:
    # Stopping criterion
    if stop_criterion.stop_after_fixed_steps(step, max_steps):
        print(f"Stopped at step {step}")
        break

    # 1. Select an agent block (weighted by number of agents)
    from_block = agent_selector.select_agent_random(occupied)

    # 2. Select a target block (must have space)
    to_block = target_selector.select_target_random_cell(occupied, H)

    # 3. Check if move is accepted (Δu > 0)
    if move_acceptor.accept_if_personal_utility_improves(occupied[from_block] / H, (occupied[to_block]
        occupied[from_block] -= 1
        occupied[to_block] += 1
    step += 1
```



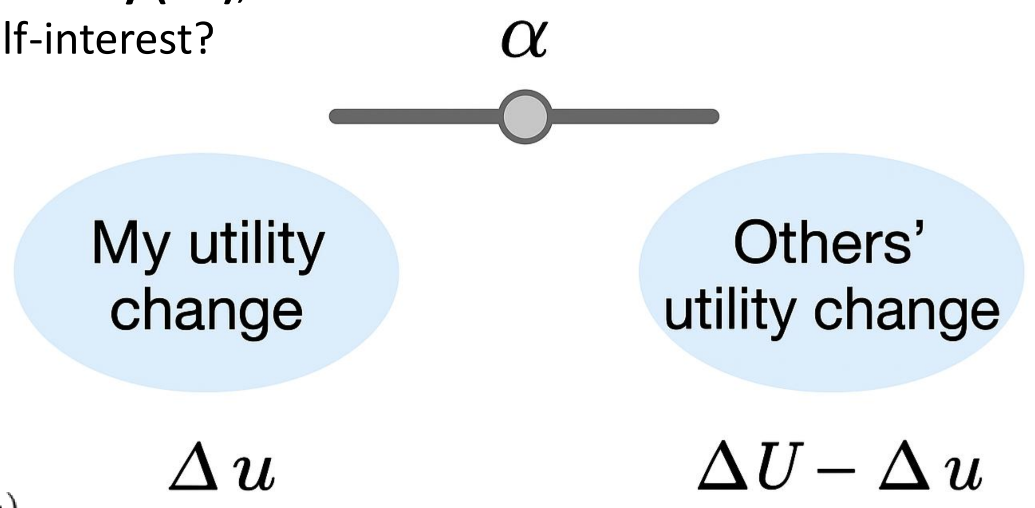
From Purely Selfish to Collective Dynamics

Core question:

•If agents consider **collective utility (ΔU)** in addition to **personal utility (Δu)**,
→ Can we avoid segregation and inefficiency caused by pure self-interest?

Key parameter: α

•**α = 0** → purely selfish
•**α = 1** → fully collective
•**0 < α < 1** → mixed behavior



Connection to the PNAS paper:

•Construct a bridge between Δu and ΔU: $G = \Delta u + \alpha(\Delta U - \Delta u)$

or $G = (1-\alpha)\,\Delta u \;+\; \alpha\,\Delta U$

•Provide a critical threshold formula: $\alpha_c = \dfrac{1}{3-2m}$

•Explain the **phase transition** from segregation → mixing

香港中文大學（深圳） The Chinese University of Hong Kong, Shenzhen
人工智能學院 School of Artificial Intelligence

What is T?

Temperature (**T**) = randomness / noise in real life, Captures hesitation, mistakes, or “going with the flow”

Case 1: T = 0 (strict rule)

- If **G > 0** → always move
- If **G ≤ 0** → never move
- Like a perfectly rational, cold “economic agent”

Case 2: T small (close to 0)

- Most of the time: follow G, occasionally: make “mistakes”:
    - Slightly negative G → might still move (small probability)
    - Slightly positive G → might stay
- Like real people showing hesitation or impulse

Case 3: T large

- Decisions become highly **randomized**
- Move vs. stay ~ like flipping a coin
- Agents barely care about the size of G, often choose randomly

Try it yourself!

Set different parameters
to observe changes in the plot.

Cheat Sheet: Parameter Effects on Heatmaps

Purpose	Setting	Heatmap outcome	Key point to explain
Baseline segregation	$m=0.5$, $\alpha=0$, $T=0$	Mixture of full blocks and empty blocks	Micro-level rationality → macro-level inefficiency
Below threshold	$m=0.5$, $\alpha=0.1$, $T=0$	Segregation	α not beyond α_c
Cross the threshold	$m=0.5$, $\alpha=0.6$, $T=0$	Becomes uniform (≈0.5 each block)	Phase transition: increasing α leads to mixing
Larger m	$m=0.8$, $\alpha=0.3$ vs 0.8, $T=0$	0.3 → segregation, 0.8 → mixing	Higher asymmetry m raises α_c
Increasing T	$\alpha=0$, $T=0/0.1/0.8$	From segregation to more uniform	Noise smooths patterns ≠ higher utility
Different ρ_0	$\rho_0=0.3$, $\alpha=1$, $T=0$	Half-full blocks + empty blocks	With limited population, "best mix" is half-full + empty
System size	$Q=16$ vs $Q=100$	Coarse-grained vs fine-grained patterns	System size changes resolution, not mechanism

Thank You!

Thank you for your attention and participation.

📫 **Contact Information:**

zhanzhanzhao@cuhk.edu.cn (Dr. Zhao Zhanzhan)
225080011@link.cuhk.edu.cn (Jia Xiao)