ABM for Ecological Dynamics

Lab 3

Kwabena Owusu, Peter Steiglechner

29 Jan 2025

[Advertisement] Complexity Science Hub

Complexity Science*Hub

*

- Foundations of Complex Systems

Societies Under Stress

- Collective Minds
- Human Migration
- Social Complexity & Collapse
- Crime in the Digital Age

Societies in Transition

- Urban Sustainability
- Transforming Economies
- Algorithmic Fairness
- AI & Machine Learning

Metabolism of Societies

- Digital Currency Ecosystems
- Supply Chain Science
- Healthcare & Medicine
- Health Across Species

- Data Analytics

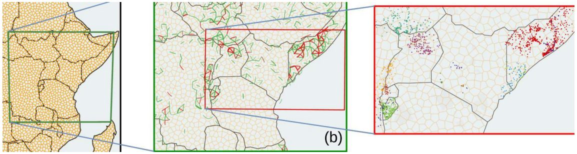


Figure: Kushwaha, Niraj and Lee, Eddie (2023) 'From scaling to microscopic mechanism of armed conflict.' *PNAS Nexus*.

ABMs for simulating predator-prey systems

- preys grow, but get eaten by predators
- predators grow if they get the preys but otherwise they die off.

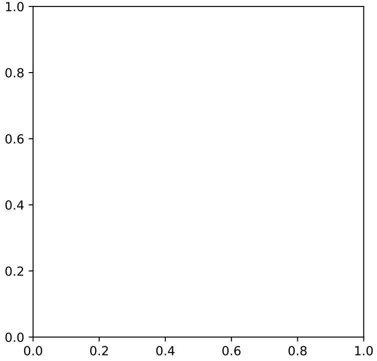
Today: Foxes (predator) vs. Rabbits (prey) on a squared area.

- Now at an ABM, the ecological dynamics are described **at the level of individual agents** and not at an aggregated population level.

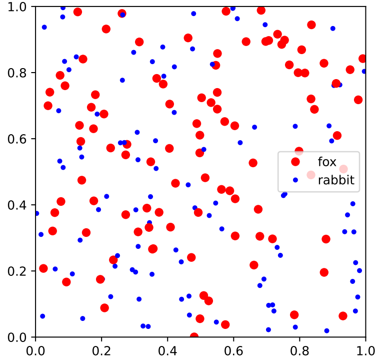
Designing the ABM

1. Design the data structure to store the attributes of the agents.
2. ~~Design the data structure to store the states of the environment.~~
3. ~~Describe the rules for how the environment behaves on its own.~~
4. ~~Describe the rules for how agents interact with the environment.~~
5. Describe the rules for how agents behave on their own.
6. Describe the rules for how agents interact with each other.

1. Attributes of the agents



1. Attributes of the agents



- type of agents: predators (foxes) and prey (rabbits).
- spatial location (because we simulate their interactions in a space)

Structure of ABMs

```
class agent:
  pass

def initialize():
  global agents
  ... # initialize a list of agents = [ag1, ag2,...]

def update():
  global agents
  ... # update the attributes of the agents

def observe():
  ... # plot/store the results

t=0
np.random.seed(2025)
initialize()
observe()
for t in range(1, T):
  update()
  observe()
```

The rough structure is **always** the same.

1. Attributes of the agents — code

```python
class agent:
  pass

def initialize():
  global agents
  agents = []
  # initialize the rabbits
  for i in range(rabbits_init):
    ag = agent()
    ag.type = 'rabbit'
    ag.x = np.random.random()
    ag.y = np.random.random()
    agents.append(ag)

  # initialize the foxes
  for i in range(foxes_init):
    ag = agent()
    ag.type = 'fox'
    ag.x = np.random.random()
    ag.y = np.random.random()
    agents.append(ag)
  return
```

`global agents` means that `initialize()` changes a global variable `agents`. It is available to us even when we leave the function.

1. Attributes of the agents — code

```
class agent:
  pass

def initialize():
  global agents
  agents = []
  # initialize the rabbits
  for i in range(rabbits_init):
    ag = agent()
    ag.type = 'rabbit'
    ag.x = np.random.random()
    ag.y = np.random.random()
    agents.append(ag)

  # initialize the foxes
  for i in range(foxes_init):
    ag = agent()
    ag.type = 'fox'
    ag.x = np.random.random()
    ag.y = np.random.random()
    agents.append(ag)
  return
```

`ag = agent()` creates the agent. Then we add some attributes: `type`, `x`, `y`.

1. Attributes of the agents — code

```
class agent:
  pass

def initialize():
  global agents
  agents = []
  # initialize the rabbits
  for i in range(rabbits_init):
    ag = agent()
    ag.type = 'rabbit'
    ag.x = np.random.random()
    ag.y = np.random.random()
    agents.append(ag)

  # initialize the foxes
  for i in range(foxes_init):
    ag = agent()
    ag.type = 'fox'
    ag.x = np.random.random()
    ag.y = np.random.random()
    agents.append(ag)
  return
```

`agents.append(ag)` adds the agent `ag` we have just created to our list of agents.

Designing the ABM

1. Design the data structure to store the attributes of the agents.
2. ~~Design the data structure to store the states of the environment.~~
3. ~~Describe the rules for how the environment behaves on its own.~~
4. ~~Describe the rules for how agents interact with the environment.~~
5. Describe the rules for how agents behave on their own.
6. Describe the rules for how agents interact with each other.

5.+6. Rules for agent behaviour and agent interactions

- Different rules have to be designed for prey and predator agents.
- **Prey agents**: Each individual agent reproduces at a certain reproduction rate. We assume a logistic growth. If a prey agent meets a predator agent, it dies with some probability because of predation. Death can be implemented simply as the removal of the agent from the agents list.
- **Predator agents**: If a predator agent cannot find any prey agent nearby, it dies with some probability because of the lack of food. But if it can consume a prey, it can also reproduce at a certain reproduction rate.
- **Both agents**: Agents move in space and we assume they do a random walk (but with different 'speeds').

5.+6. Agent interactions — conceptual code

```
def update_one_agent():
  global agents

  if agents == []:
    return

  # choose a random agent
  ag = agents[np.random.randint(len(agents))]

  # simulating random movement
  ...

  # detecting collision and simulating death or birth
  if ag.type == 'rabbit':
      # if there are foxes nearby, die with probability
          death_r
      # reproduce via logistic growth
  else: # fox
    # if there are no rabbits nearby, die with probability
        death_f
    # if there are rabbits nearby, reproduce
```

5.+6. Agent interactions — code

```
def update_one_agent():
  global agents

  if agents == []:
    return

  # choose a random agent
  ag = agents[np.random.randint(len(agents))]

  # simulating random movement
  move_radius = moveradius_r if ag.type == 'r' else moveradius_f
  ag.x = ag.x + np.random.uniform(-move_radius, move_radius)
  ag.y = ag.y + np.random.uniform(-move_radius, move_radius)

  # ensure that agents do not leave the environment.
  ag.x = np.clip(ag.x, 0, 1)
  ag.y = np.clip(ag.y, 0, 1)

  ...
```

5.+6. Agent interactions — code continued

```
def update_one_agent():
  ...
  # detecting collision and simulating death or birth
  neighbors = []
  for ag2 in agents:
      if (ag2.type != ag.type) and ((ag.x - ag2.x)**2 + (ag.y - ag2.y)**2 <
           hunt_radius**2):
          neighbors.append(ag2)

  if ag.type == 'rabbit':
      if len(neighbors) > 0: # if there are foxes nearby, die with probability
           death_r
        if np.random.random() < death_r:
            agents.remove(ag)
            return
      # reproduce via logistic growth
      rabbit_pop = len([ag for ag in agents if ag.type=="rabbit"])
      if np.random.random() < repro_r*(1 - rabbit_pop/carrying_cap):
        agents.append(cp.copy(ag))
  else:
      if len(neighbors) == 0: # if there are no rabbits nearby, die with
           probability death_f
        if np.random.random() < death_f:
            agents.remove(ag)
            return
      else: # if there are rabbits nearby, reproduce
        if np.random.random() < repro_f:
            agents.append(cp.copy(ag))
  return
```

The `copy` function of package `copy` (`cp`) creates a copy of the object.

Updating

What is the issue with varying number of agents?

- Consider we have 10 agents in the beginning. Simply calling `update_one_agent()` will update an agent every 10 steps.

Updating

What is the issue with varying number of agents?

- Consider we have 10 agents in the beginning. Simply calling `update_one_agent()` will update an agent every 10 steps.
- Consider now that the number of agents has grown to 1000. If we simply call `update_one_agent()` each agent updates every 1000 steps. It seems like each fox/rabbit has become "slower".

Solution $\rightarrow$ sligthly more complicated updating scheme (next slide)

Updating

- Synchronous Updating: All agents update at the same time

Updating

- Synchronous Updating: All agents update at the same time (e.g. Election, Betting)

Updating

- Synchronous Updating: All agents update at the same time (e.g. Election, Betting)
- Asynchronous Updating: Agents update after one another.

Updating

- Synchronous Updating: All agents update at the same time (e.g. Election, Betting)
- Asynchronous Updating: Agents update after one another. (e.g. Board game, Predator-prey(?))

Updating

- Synchronous Updating: All agents update at the same time (e.g. Election, Betting)
- Asynchronous Updating: Agents update after one another. (e.g. Board game, Predator-prey(?))

- Here: Asynchronous updating and assuming that each agent updates once per time step on average.

Updating

- Synchronous Updating: All agents update at the same time (e.g. Election, Betting)
- Asynchronous Updating: Agents update after one another. (e.g. Board game, Predator-prey(?))

- Here: Asynchronous updating and assuming that each agent updates once per time step on average.

```
def updaet_one_time_step():
  global agents
  for i in range(len(agents)):
    update_one_agent()
```

Structure of ABMs

```
class agent:
  pass

def initialize():
  global agents
  ... # initialize a list of agents = [ag1, ag2,...]

def update():
  global agents
  ... # update the attributes of the agents

def observe():
  ... # plot/store the results

t=0
np.random.seed(2025)
initialize()
observe()
for t in range(1, T):
  update()
  observe()
```

The rough structure is **always** the same.

Observation

```
# lists to store the rabbit/fox population over time
rabbit_pop = []
fox_pop = []

...

def observe():
  global agents

  # count foxes at current time t
  # count rabbits at current time t

  # 1. plot foxes and rabbits as dots on our map
  # 2. plot the number of foxes and rabbits over time (from
      time 0 to t)
```

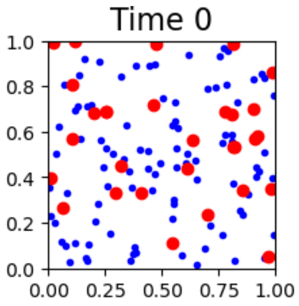
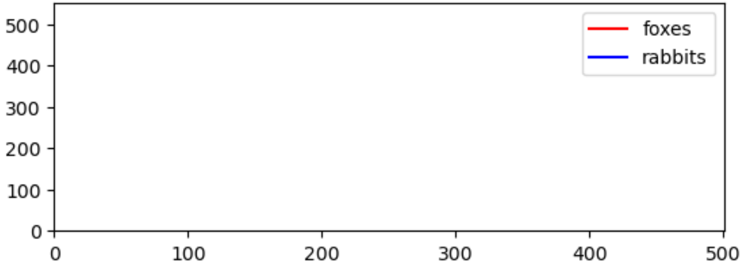
We skip the details here.
Note, for counting the foxes/rabbits, we use *list comprehension*, which is a short-cut for-loop (see extra material)

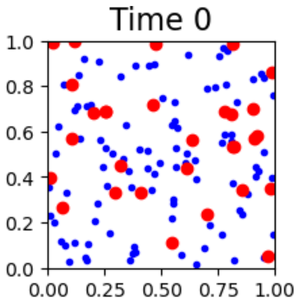
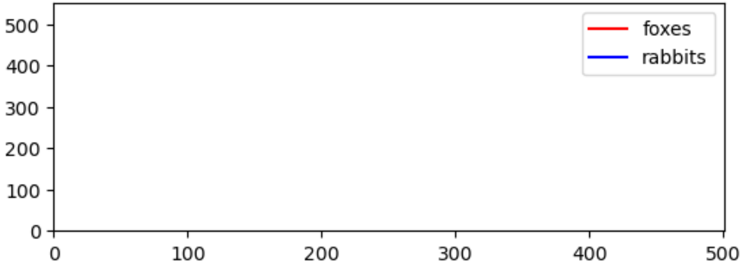
Structure of ABMs

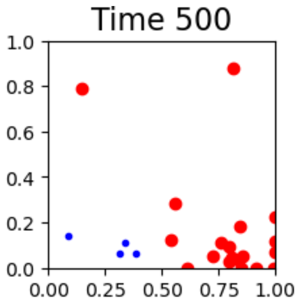
```
class agent:
  pass

def initialize():
  global agents
  ... # initialize a list of agents = [ag1, ag2,...]

def update():
  global agents
  ... # update the attributes of the agents

def observe():
  ... # plot/store the results

t=0
np.random.seed(2025)
initialize()
observe()
for t in range(1, T):
  update()
  observe()
```

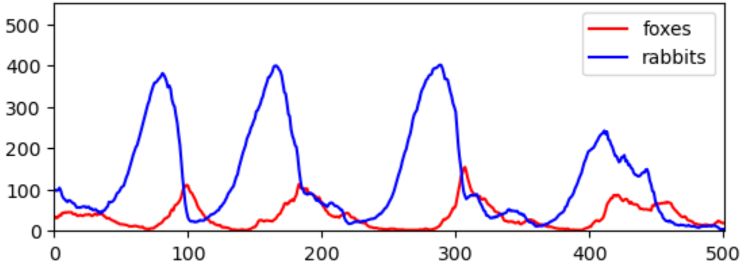
Note: `np.random.seed(seed)` will enforce the same sequence of random events/numbers.

Let's go! Yanko!


Time 0
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00

foxes
rabbits
500
400
300
200
100
0
0
100
200
300
400
500


Time 0
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00

foxes
rabbits
500
400
300
200
100
0
0
100
200
300
400
500


Time 500
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00


foxes
rabbits
500
400
300
200
100
0
0
100
200
300
400
500

The populations of the two species are definitely showing oscillatory dynamics, yet the oscillations are different than the regular cyclic ones predicted by differential equation models. Instead, there are significant fluctuations and the period of the oscillations is not regular.
Spatial extension, discreteness of individual agents, and stochasticity in their behaviors all contribute in making the results of agent-based simulations more dynamic and "realistic".

Modelling Lab

Modelling lab

Numerical experiments

Use the code in the Google Drive to model the predator-prey dynamics as an ABM. **(1)** Run the simulation with different random seeds and look at the change in the figures. Can you crash one of the populations? **(2)** Change the initial conditions. What do you observe? **(3)** Change the parameters of the model. What do you observe? **(4)** [Advanced] After the simulation finishes, plot the phase diagram, i.e. a plot that shows the population of rabbits on the x-axis and the population of foxes on the y-axis. Make sense of that plot and again change the parameters. **(5)** [Advanced] Modify a process of your choice.

Extra

List comprehension

List comprehensions are short-cut for loops. It starts with `[` and ends with `]`, to help you remember that the result is going to be a list.

- `[x**2 for x in range(0,5)]` in normal language means 'a list with x^2 for all x in $\{0, 1, \ldots 4\}$'.
- `[ag.x for ag in agents if ag.type=="rabbit"]` in normal language means 'a list with all x-positions of the agents that are rabbits'.

The basic syntax is: 'result = [transform iteration filter]'
This is equivalent to:

```
result = [ ]
for ag in agents:
  if ag.type == "rabbit":
    result.append(ag.x)
```