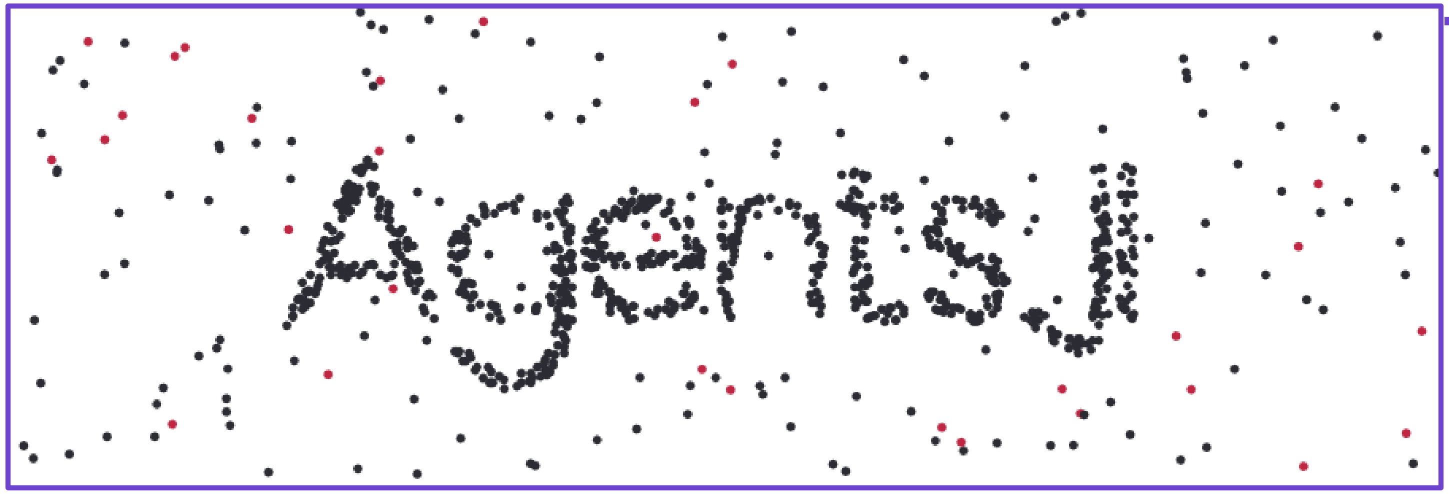
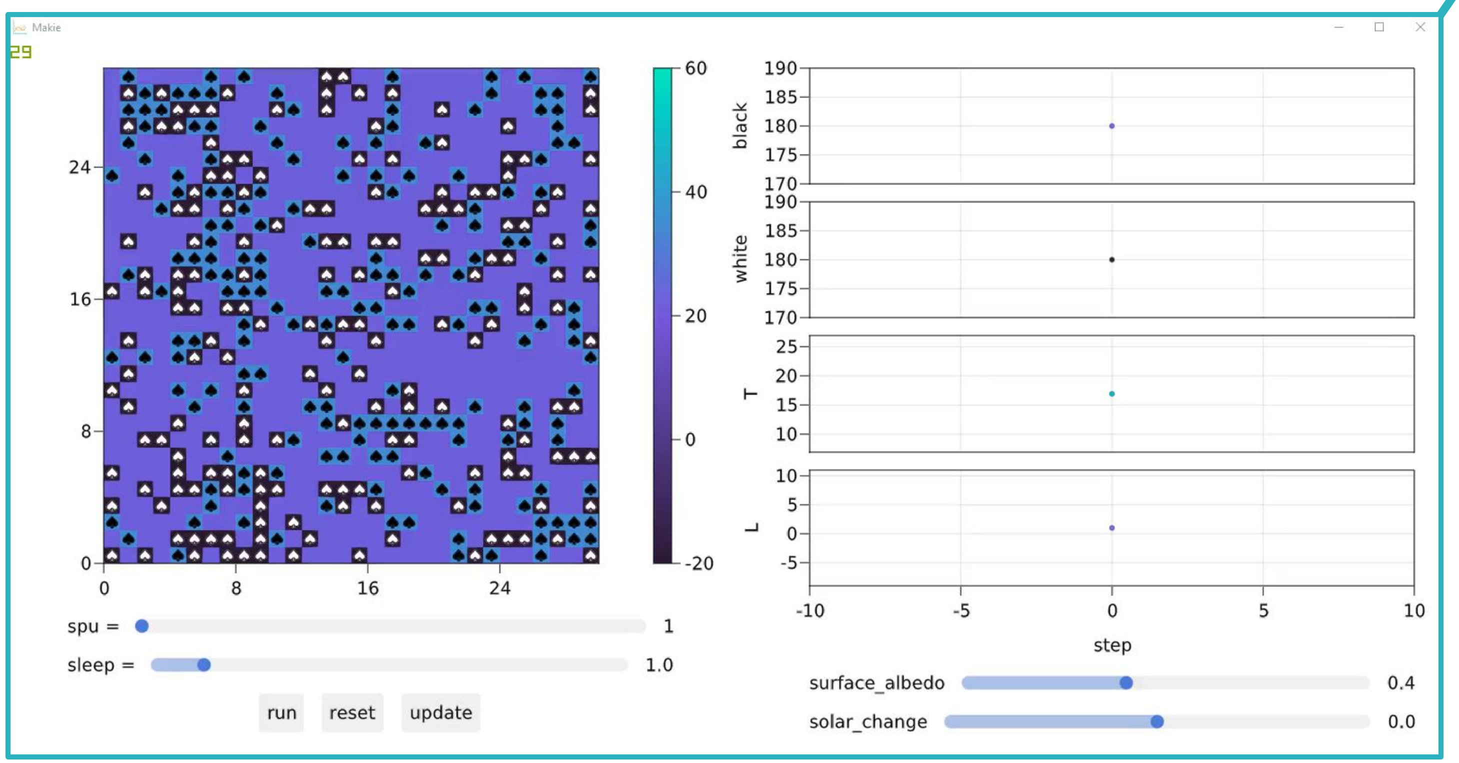


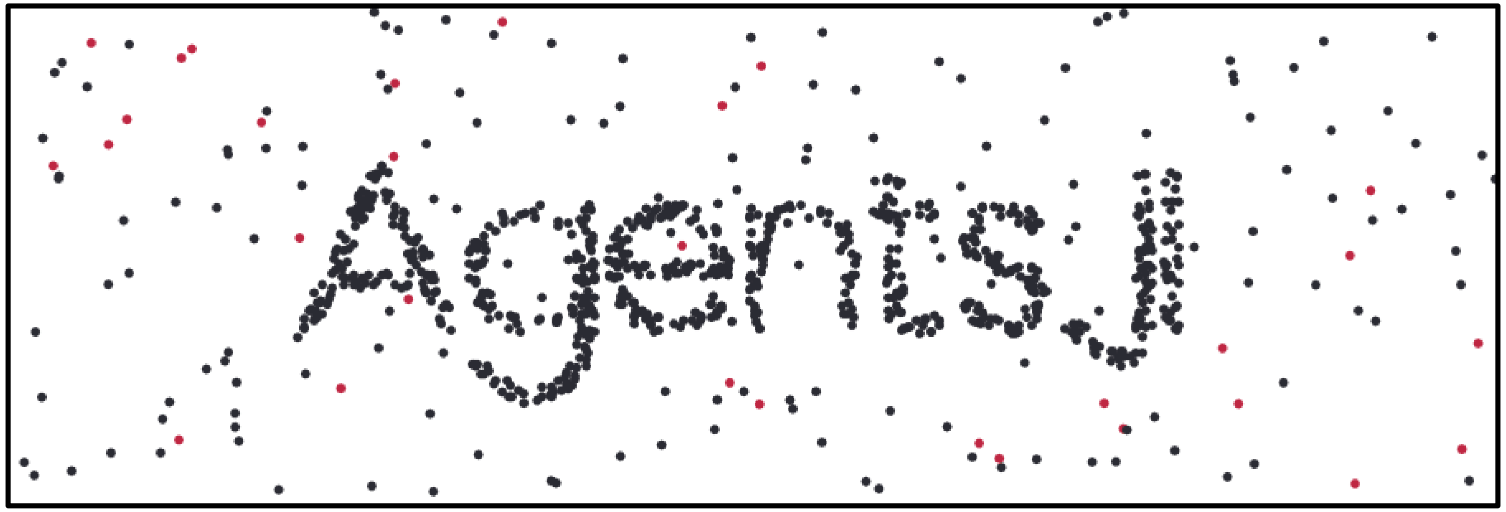
Agents.jl logo is an SIR (virus spreading) simulation in a continuous space

Interactive app with the Daisyworld simulation in discrete space

a zombie outbreak simulation on an Open Street Map



North (m)
East (m)
2000
1000
0
−1000
−2000
−4000
−2000
0
2000
4000



Feature-full

- All standard functionality
- Four kinds of spaces
- Flexible data collection, parameter scanning, check-pointing
- Multi-agent support
- Many more…[1]

Simple

- Harmonious, general API
- Little lines of code, short learning curve
- Simpler than NetLogo, Mesa, …[2]

Performant

- Written purely in Julia, has been optimized
- Allows for distributed computing
- Faster than NetLogo, Mesa, …[2]

Interactive

- Julia is dynamic, Agents.jl is as well
- Visualizations & interactive applications
- API is extendable and adaptable

Integrated

- Integrates with the entire Julia ecosystem
- Main output is a `DataFrame`
- Integrations with DifferentialEquations.jl, LightGraphs.jl, BlackBoxOptim.jl, …[1]

[1] docs: https://juliadynamics.github.io/Agents.jl/stable/ [2] paper: https://arxiv.org/abs/2101.10072

Session 1 exercises

1. Introduce third group to the Schelling model. Re-initialize the model with `N` agents, having `N/3` agents of each type. Animate the model evolution using `abm_play` so that each agent group is plotted with a different color and marker.

2. Continue with the Schelling model. Introduce a agent property `stationary` that counts how many steps has an agent remained in the same location. Then, collect agent happiness only for agents that are stationary for `x` steps.

3. Modify the model properties to have a dictionary that maps agent group to a "segregation" counter, an integer. Then create a `model_step!` function that at its start it sets all segregations to 0. Then, it loops over agents. Each agent that has all 8 neighbors be of the same group, increments their group segregation property by 1.

4. Then, use `run!` to collect model data. Specifically, collect the segregation counter of each group as an individual column of the output dataframe. Plot each segregation curve versus the step number, using the same colors you used to plot agents in `abm_play`.

Where can I get support for Agents.jl? (1) read online documentation, (2) chat with us in the channel #dynamics-bridged in the Julia Slack: https://julialang.org/slack/ , (3) post a question in the Julia discourse: https://discourse.julialang.org/ in the category "Modelling and simulations", using "agent" as a tag!

Session 2 exercises – part 1: Extensions of the Schoolyard example

1. Modify the `agent_step!` function so that the `new_pos` variable is clamped into the size of the schoolyard (the continuous space), because if the `new_pos` exceeds the limits of a space an error is thrown. Hint: what's `size(model.space)` ?
2. Modify the `schoolyard` function so that students have not just one friend and one foe, but 8 of each.
3. Modify the `agent_step!` function such that the `force` value of each loop in the social network adherence section takes into account not only buddiness, but also `class`. The `schoolyard` function could pass in two additional keyword arguments: `a_force = 0.5` and `b_force = -0.5` to store as model properties for example. Update the `sliders` dictionary such that `a_force` and `b_force` can be altered in the data exploration window.

Session 2 exercises – part 2: Create a new ABM from scratch!

Implement a rock-paper-scissors model. Agents exist on an `MxM` square grid, and have a `type` property which can be one of: `:rock, :paper, :scissors`. Populate the grid randomly with `N < M^2` agents (1/3 of each kind). Use a random scheduler. The model dynamics are as follows: each agent first pick a random nearby position (use distance with 4 neighbors). Then, they roll a three-sided dice and perform one of the following 3 actions:

1. Fight: Fight your neighbor according to the rules of rock-paper-scissors. The loser is killed with `kill_agent!`. Do nothing if the chosen position is empty or if there is a tie.
2. Reproduce: If the chosen position is empty, it becomes a new agent with same type via `add_agent!`.
3. Move: Move to the nearby position using `move_agent!`. If another agent was already there, that agent moves to the first agent's original position (that has been left empty).