German Centre for Integrative Biodiversity Research (iDiv)
Halle-Jena-Leipzig

HELMHOLTZ
Zentrum für Umweltforschung

Individual-based modelling

General concepts and the Agents.jl framework

Daniel Vedder

iDiv is a research centre of the
Deutsche Forschungsgemeinschaft

Types of ecological models*

population models

mathematical

species distribution models

statistical, correlative

individual-based models

rule-based, mechanistic

* non-exclusive list!

“The essence of the individual-based approach is the derivation of the properties of ecological systems from the properties of the individuals constituting these systems.”

Adam Łomnicki, 1992

iDiv

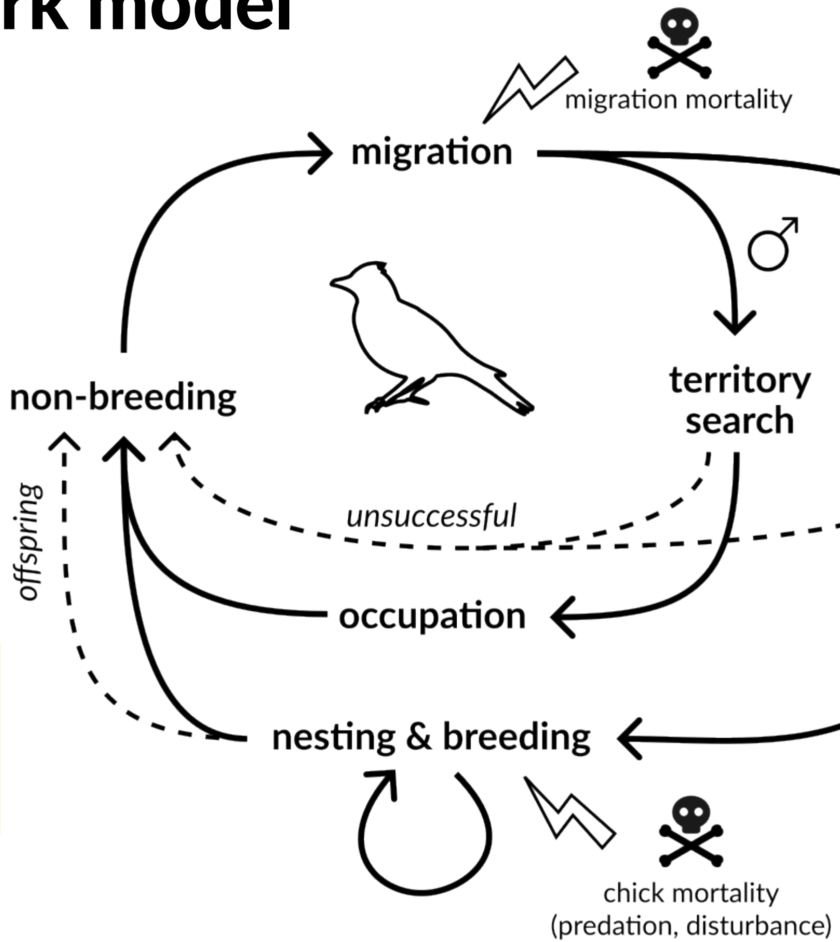
UFZ HELMHOLTZ Zentrum für Umweltforschung

Individual-based ecology

- System properties and dynamics arise from the interactions of individuals with their environment and with each other.
- IBE is based on theory. These theories are models of *individual* behaviour that are useful for understanding *system* dynamics. Theories are developed from both empirical and theoretical ecology.
- Observed patterns are a primary kind of information used to test theories and design models and studies.
- Models are implemented and solved using computer simulation. Software engineering, not differential calculus, is the primary skill needed.

Excerpted from: Grimm, V., & Railsback, S. F. (2005). Individual-based Modeling and Ecology. Princeton University Press. p. 10

Example: Skylark model



***Pattern:** Ecological trap due to preferred nesting in (intensive) grassland*

iDiv UFZ HELMHOLTZ Zentrum für Umweltforschung

Individual-based modelling in Julia

- IBMs can be written in **plain Julia**:
  - Julia is a fully-fledged programming language that is well-suited to constructing larger programs
  - Many useful libraries available, e.g. for visualisation or geographic data
  - Both GeMM and Persefone (medium-high complexity IBMs) were written in plain Julia → **maximum flexibility**
- But, there is also a dedicated ABM/IBM framework available: **Agents.jl**
  - Makes the initial creation of models much simpler
  - Provides many utility functions and visualisation options
  - **→ both powerful and simple (though less flexible)**

“From our perspective, the biggest take-away of this paper is that Agents.jl is a framework that is simple to use, requiring small amount of written code from the user, and overall easy to learn. Despite this, our comparison shows that Agents.jl always exceeds other frameworks in performance, and often also in capacity.”

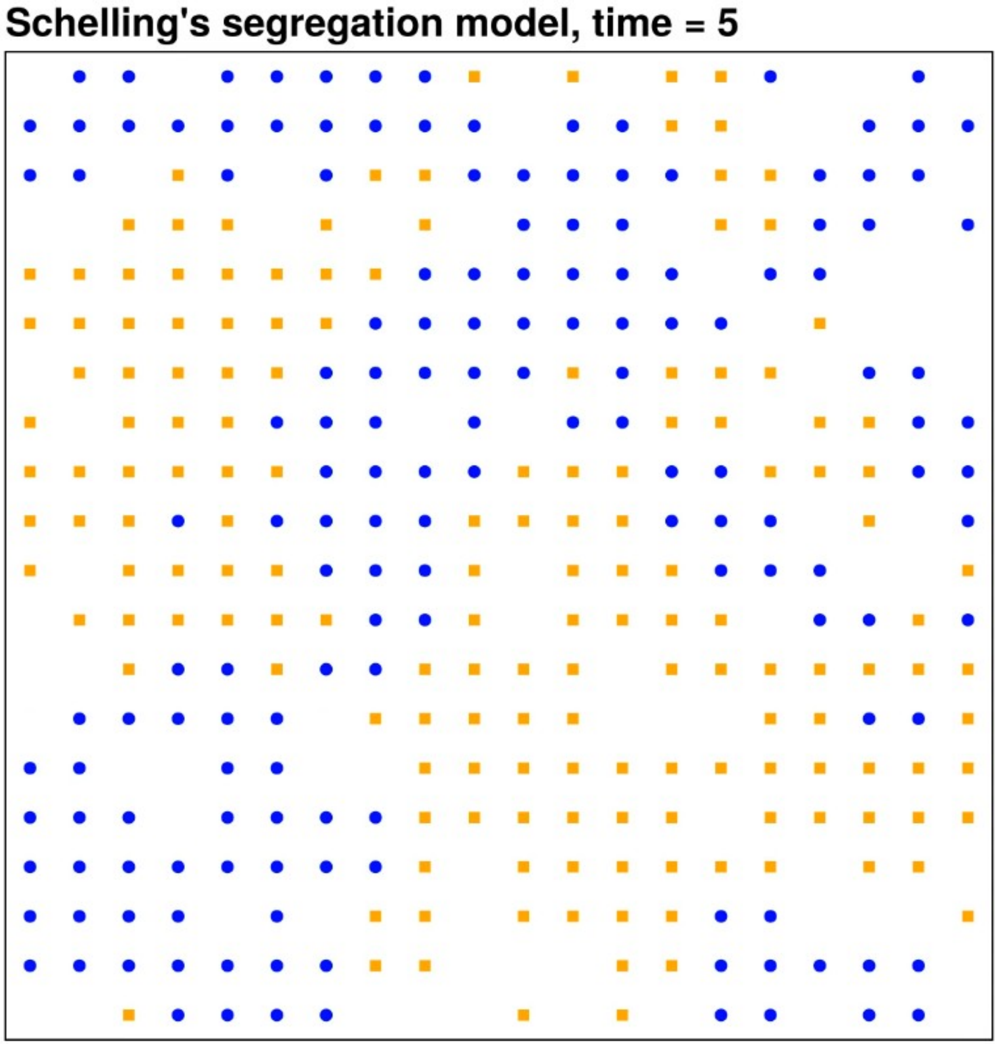
Datseris, G., Vahdati, A. R., & DuBois, T. C. (2022). Agents.jl: A performant and feature-full agent-based modeling software of minimal code complexity. SIMULATION. https://doi.org/10.1177/00375497211068820

Tutorial

Example: Schelling segregation model

Even minimal preferences for similar neighbours can lead to spatial segregation in a city

https://juliadynamics.github.io/Agents.jl/stable/tutorial/



Defining agents

```
using Agents

@agent struct Schelling(GridAgent{2})
    mood::Bool = false
    group::Int
end

function schelling_step!(agent, model)
    minhappy = model.min_to_be_happy
    count_neighbors_same_group = 0
    for neighbor in nearby_agents(agent, model)
        if agent.group == neighbor.group
            count_neighbors_same_group += 1
        end
    end
    if count_neighbors_same_group ≥ minhappy
        agent.mood = true
    else
        agent.mood = false
        move_agent_single!(agent, model)
    end
    return
end
```

Import the Agents.jl library

Create the agent /individual type with its variables and parameters

Write the update function that will be called for each individual every turn

The model object

Define the kind of space used by the model

Additional model variables / parameters

```
space = GridSpace((20, 20))
properties = Dict(:min_to_be_happy => 3)
model = StandardABM(Schelling, space; agent_step! = schelling_step!, properties)

for n in 1:300
    add_agent_single!(model; group = n < 300 / 2 ? 1 : 2)
end

using Statistics: mean
xpos(agent) = agent.pos[1]
adata = [(:mood, sum), (xpos, mean)]

adf, mdf = run!(model, 5; adata)
```

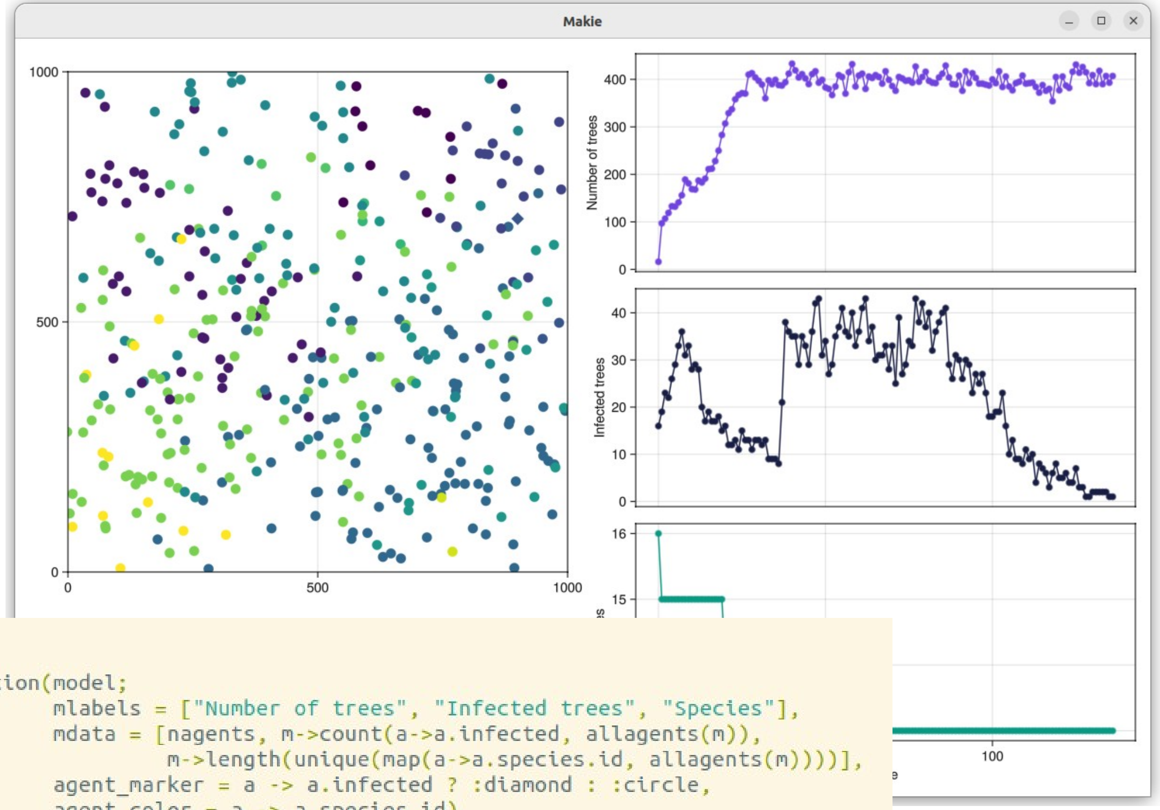
Create the model object

Add agents (=individuals) to the model

Define what data should be collected

Run the model and return the data

Output & visualisation




```julia
function openapp()
    model = initworld()
    fig, abmobs = abmexploration(model;
                                 mlabels = ["Number of trees", "Infected trees", "Species"],
                                 mdata = [nagents, m->count(a->a.infected, allagents(m)),
                                          m->length(unique(map(a->a.species.id, allagents(m))))],
                                 agent_marker = a -> a.infected ? :diamond : :circle,
                                 agent_color = a -> a.species.id)
    @info "Created interface."
    return fig
```

Any questions?

Then let’s get started…