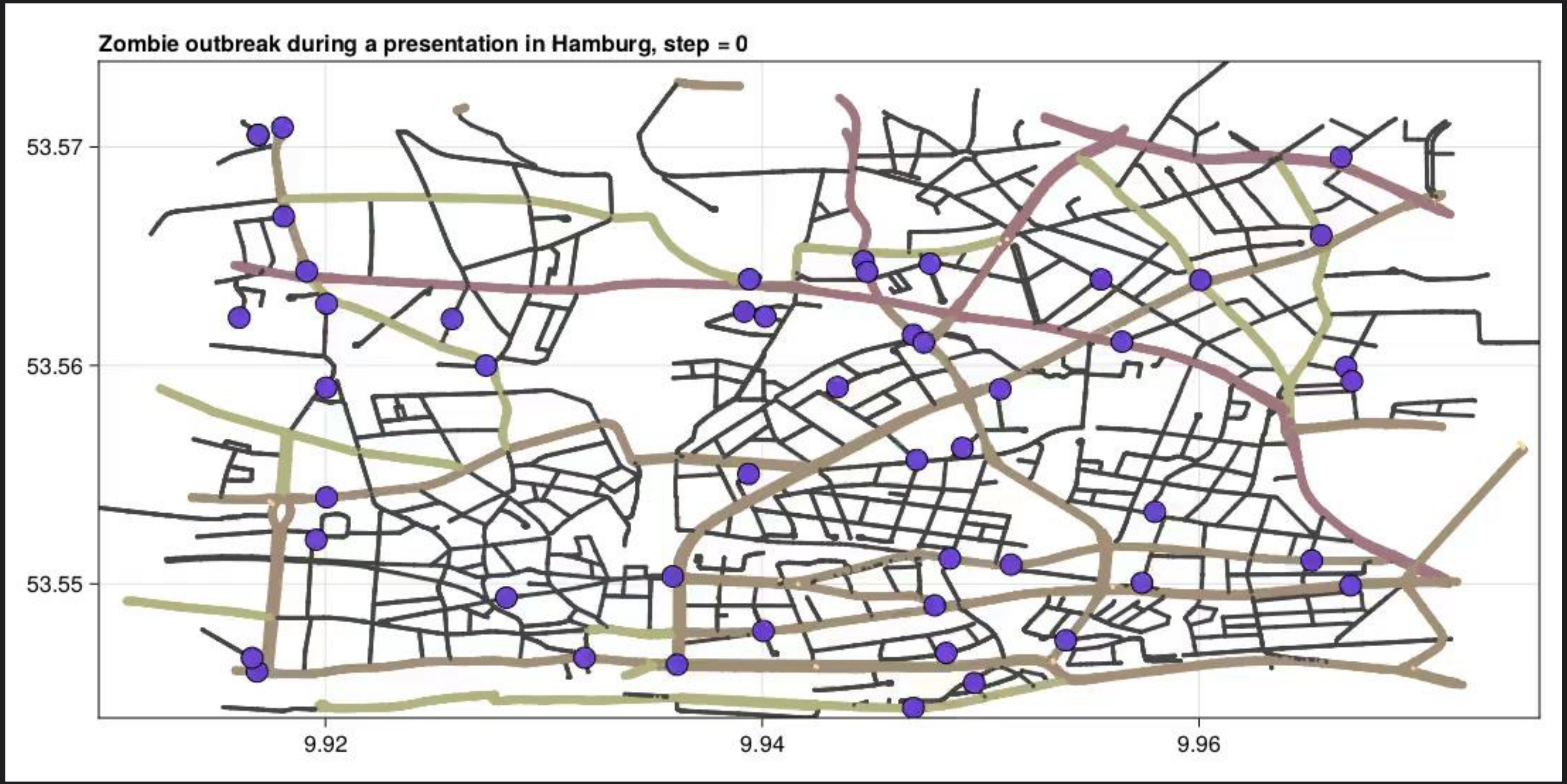
Why you should do your agent based modelling in Julia with Agents.jl

George Datseris,
Max Planck Institute for Meteorology

JuliaDynamics


Zombie outbreak during a presentation in Hamburg, step = 0
53.57
53.56
53.55
9.92
9.94
9.96

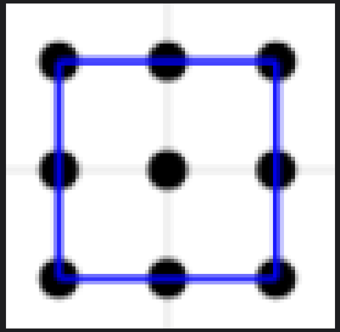
What's Agent Based Modelling?

- Simulation where autonomous agents react to their environment and interact with each other given a predefined set of rules. These rules are formulated based on explicit statements rather than mathematical equations (such as: "If condition X is fulfilled, do action Y, and then perform operation Z on all nearby agents").
  - From *Nonlinear Dynamics*, Datseris & Parlitz, Nature-Springer Textbooks

- Useful in socioeconomic and complexity sciences because
  - Can capture emergent phenomena from agent interactions
  - Provides natural/intuitive description of the system
  - Flexible, very easy to change the rules of the simulation

Schelling's segregation model

- Archetypical ABM example, showing the emergent phenomenon of segregation, in an unexpected scenario where agents care about having <u>only a few</u> of their neighbors to be the "same"



- Rules of the game:
  - Agents belong to groups enumerated by the integers (0, 1, 2, ...)
  - Agents live in a 2D grid with integer positions. Each position = 1 agent max!
  - At each step, agents look at their 8 closest neighbors on the grid
  - If at least `k` of them are same group, agent becomes happy, and stays put
  - If not, agent is unhappy and moves to a random unoccupied position
  - The amazing thing is that this simulation shows segregation already with `k=3`
- Interactive simulation with Julia and Agents.jl

Flocking

- Birds can create flocks without direct directive to flock

- Rules of the game:
  - birds live in continuous space, and have continuous velocity and orientation
  - they maintain a minimum distance from other birds to avoid collision
  - they fly towards the average position of neighbors
  - they fly in the average direction of neighbors

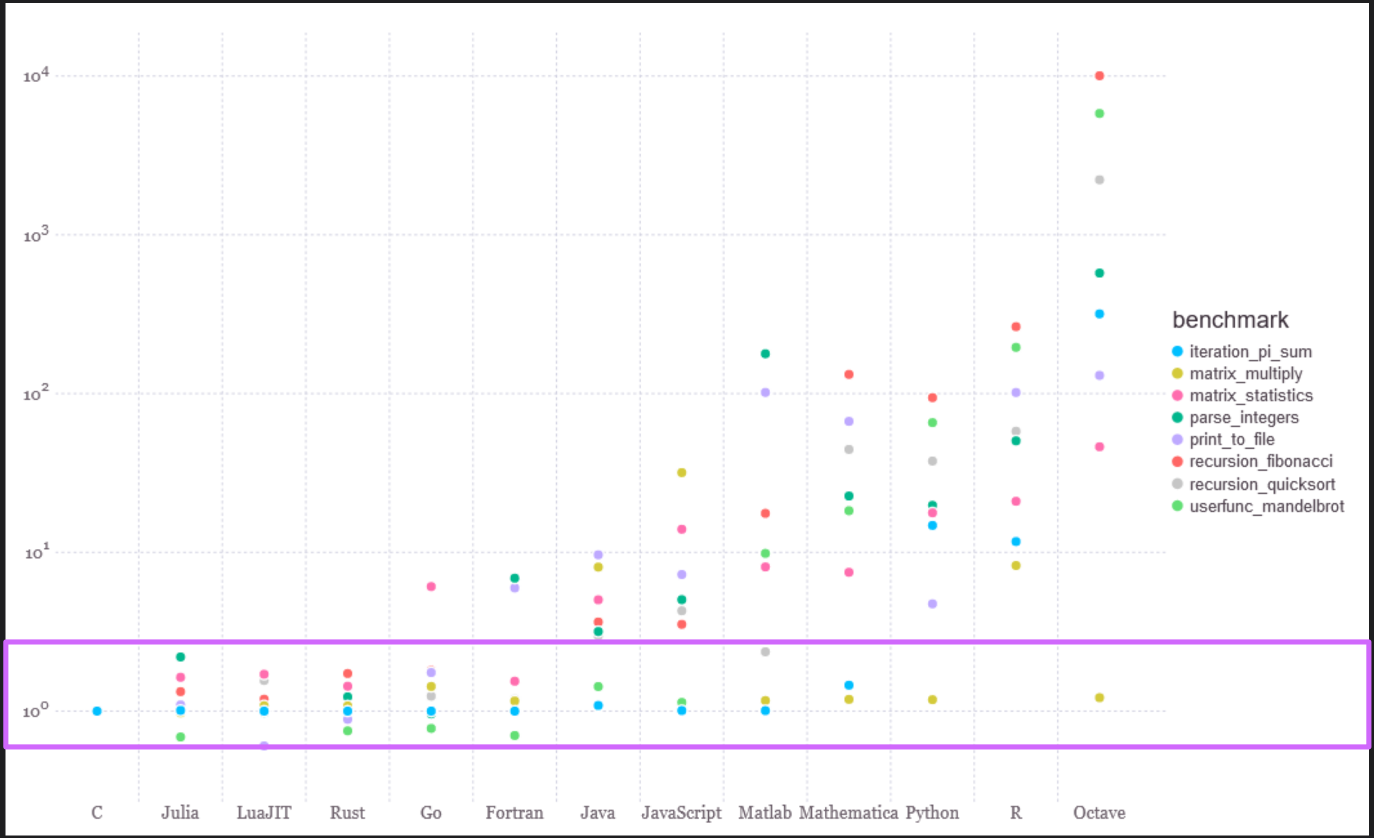
- Interactive simulation with Julia and Agents.jl

julia

- ...is a general-purpose programming language
- ...is dynamic and interactive like Python/Matlab, ***allowing fast prototyping***
- ...is performant like C/Fortran, ***allowing fast execution***
- ...***has high level, intuitive syntax that leads to clean, concise code***
- ...has a thriving ecosystem of cutting edge, high-quality packages
- ...has an exceptional packages & binaries management system
- ...***makes package composability seamless*** due to multiple dispatch
- ...***is completely free***, open source, with the most permissive license
- ...is the most suitable language for science and has the ***shortest time to first science™***

Julia code is fast by itself!

- Julia has been designed for hardcore scientific computing from start



Julia microbenchmarks (from [official site]())

- User-written code can be as fast as internal! Very important for ABM!

Let’s make an ABM in pure Julia!

- The high-levelness of Julia allows one to write ABMs from scratch rather straightforwardly!

- To prove the point, let’s go through a barebones pure Julia implementation of the Schelling model.

- The code, while super high level, is still very fast!
  - If we have time at the end of the presentation, I’ll give a rough idea of why Julia is so fast!

Enter Agents.jl

- The Julia Agent Based Modelling package, with ~3,000 users

Feature-full
- All standard features you'd expect
- Five kinds of spaces
- Flexible data collection, parameter scanning, check-pointing
- Multi-agent support
- And many more…[1]
- More than NetLogo, Mesa, etc.[1]

Simple
- Harmonious, general API
- Little lines of code, short learning curve
- Modular design allows user to focus on composition, not making it work
- Simpler than NetLogo, Mesa, …[2]

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

Performant
- Written purely in Julia & strongly optimized
- Allows for distributed computing
- Faster than NetLogo, Mesa, etc.[2]

Integrated (or, composable)
- Integrates with the entire Julia ecosystem
- Main output is a DataFrame
- Integrations with DifferentialEquations.jl, Graphs.jl, BlackBoxOptim.jl, …[1]

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

Re-writing an ABM in Agents.jl

- Let's see how Schelling would be written in Agents.jl

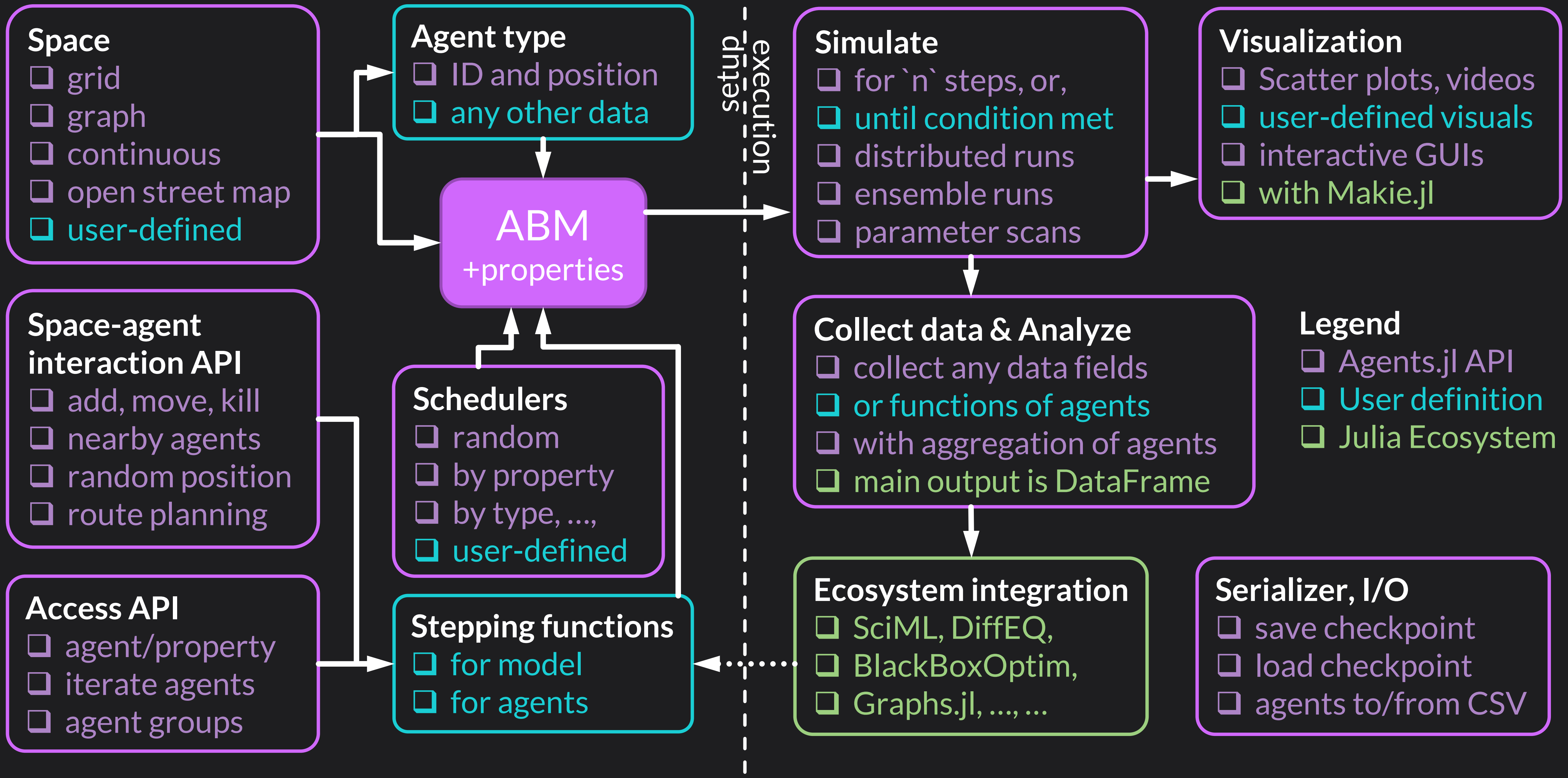
- The boost in performance is good!
  - Becomes much, much larger in actually realistic models
  - And even more so in any other non-trivial space

- But the most important part: the gain in brain time as you don't have to care about the implementation details of an ABM, and can focus on the composition, i.e., the science
  - This is true even though this is the most trivial possible example of an ABM (one only agent per position in a grid space)

Zombie outbreak in a city

- Showcases the huge gain in brain time for a model of even medium complexity
  - agents exist in an open street map representation of Hamburg-Altona and are restricted to walk in streets only. The distance between agents is the road length needed to be travelled to reach an agent (in kilometers).
  - they go to a given coordinate where an event happens but one of the agents is carrying a deadly zombie virus that goes “live” after some countdown, turning “patient zero” into a zombie
  - zombies look around for humans and they pick the closest one to hunt down; if they reach the human they turn it into zombie. The zombification process costs some time to the hunting zombie and the victim.
  - (“zombies look around” in an open street map would take weeks to implement)
  - humans start running towards random locations in the city. However, they keep track of the amount of kilometers run. If this exceeds a threshold, they have to stop running and rest for a while.
  - zombie agents have reduced vision but increased speed
- Let’s go through the code of the opening simulation step-by-step

Agents.jl modular design



Integration example: Differential equations

- You can integrate differential equations solving inside your ABM, or you can integrate an existing ABM into a differential equations solver!
  - Many potential advantages, such as accelerating simulation, increasing stability of discretized time processes, enabling ABM-event-driven differential equations…

- How simple would it be to couple DiffEq solvers to an ABM…?
  - You literally add a DiffEq integrator as a model property, just like you would do any other parameter/option/quantity related to the ABM

- https://juliadynamics.github.io/Agents.jl/stable/examples/diffeq/

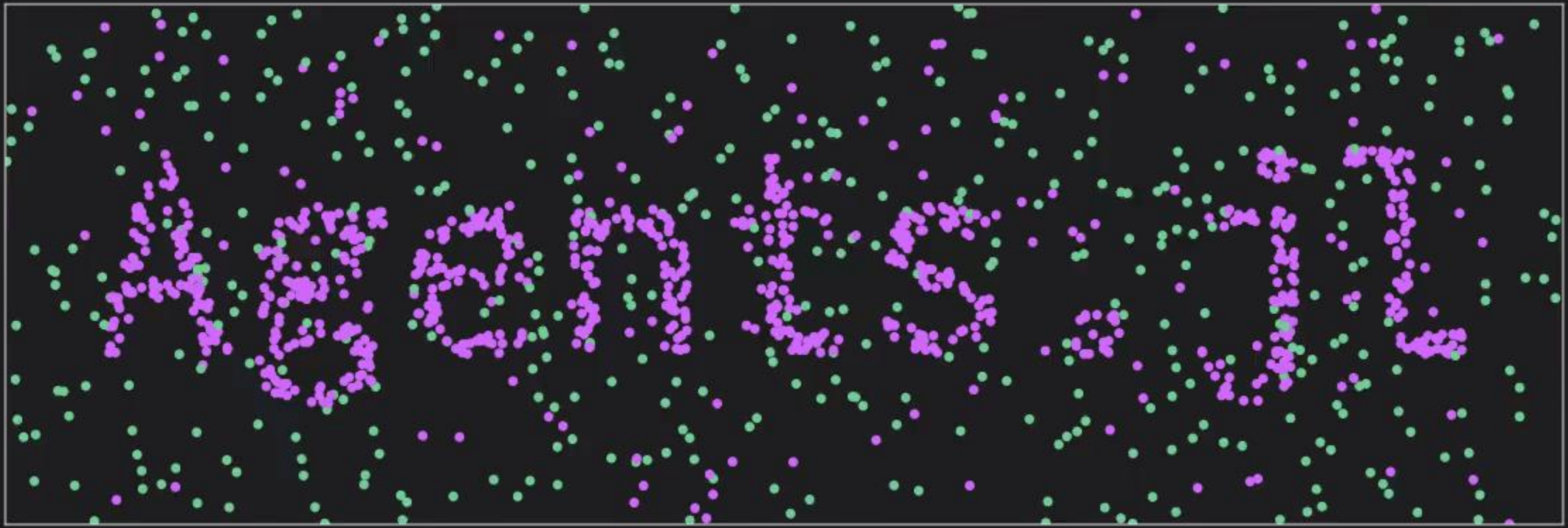
Brief look into ABM-relevant packages

- DataFrames.jl for data analysis
- Graphs.jl for social networks or generally weighted/directional graphs
- Optimization.jl for optimization (minimizing cost functions)
- Surrogates.jl for developing surrogate models of ABMs
- Distributions.jl for random number distributions

- And some other Julia packages that do ABM-related simulations:
  - ReinforcementLearning.jl
  - POMDPs.jl (partially observable Markov decision processes)
  - DynamicGrids.jl

Super cool stuff bro, where do I sign up?

- Intense workshop for learning Julia:
  - https://www.youtube.com/watch?v=Fi7Pf2NveH0
- Agents.jl video tutorial:
  - https://www.youtube.com/watch?v=fgwAfAa4kt0
- Julia discussion forum (also where people ask questions about Julia or Agents.jl):
  - https://discourse.julialang.org/
- Frederik Banning blog posts for doing ABM in Julia:
  - https://forem.julialang.org/fbanning/series/1
- Textbook on ABMs:
  - https://www.railsback-grimm-abm-book.com/
- Complexity Explorer courses on ABMs and complexity:
  - https://www.complexityexplorer.org/courses/
- Introductory/review article on ABMs:
  - https://link.springer.com/article/10.1057/jos.2010.3



Thank you ♡

all presented material and code are on GitHub:

https://github.com/Datseris/WhyDoABMwithAgents.jl

“How do I do X in Agents.jl?” Just ask!

Why is Julia fast?

- While it looks like Python, Julia is actually an ***interactive compiled language***
- First, every object existing in Julia has an associated “Type”

```
typeof(0.1)    typeof("test")    typeof(Dict(1 => :a, 2 => :b))    typeof(random_agent(model))
# Float64      # String          # Dict{Int64, Symbol}             # SchellingAgent
```

Here’s what happens when you run some code...

some function definition

```
function add_agent_single!(agent, model)
    position = random_empty(model)
    isnothing(position) && return nothing
    agent.pos = position
    add_agent_pos!(agent, model)
    return agent
end
```

function called at runtime 1st time

```
model = graph_space_model
agent = some_appropriate_agent
add_agent_single!(agent, model)
```

Julia performs type inference

```
model::ABM{GraphSpace, A}
agent::A # specific fields!
# infers positions are `Int`
position::Int
# hence, infers:
random_empty → rand(1:N) → ...
# similarly, this:
add_agent_pos!(...)
# becomes integer insertion
# into integer vector, ...
```

Sends concretely typed code to LLVM (/.//)

LLVM compiles optimized machine code because it has complete information (all types are known)

compiled code is stored in the function’s method table

method matching input arguments is called at runtime

Multiple Dispatch, pt. 1

- Multiple dispatch is a central programming concept in Julia
- Remember, every object existing in Julia has an associated “Type”

```
typeof(0.1)    typeof("test")    typeof(Dict(1 => :a, 2 => :b))    typeof(random_agent(model))
# Float64      # String          # Dict{Int64, Symbol}             # SchellingAgent
```

  - *such internal type representation exists in all languages; in Julia it is pronounced & at the forefront due to what comes in the next slides…*

- In common object oriented languages, e.g., Python, function behavior depends on only the first argument (which is the “self”). E.g.:

```
array.set_size(*args)
axis.set_size(*args)
```

here `array` could be an instance of something from numpy while `axis` could come from matplotlib. The language “dispatches” the function `set_size`, depending on the first argument, which is `array` or `axis`. It is important to note that in most object oriented languages, the **method is a property of the type**. This is a huge deficit for the language composability.

Multiple Dispatch, pt. 2

- In Julia, the behavior of a function (i.e., the "dispatch"), depends on the input type of **all** arguments

```
set_size(a::Array, args...) = ...
set_size(a::Axis, args...) = ...
set_size(s, a::Array, args...) = ...
set_size(a::Array, b::Vector) = ...
set_size(a::Array, x::Real, y::Real, z::Real) = ...
```

- And an actual example now:

```
f(a,b) = "generic method"
f(a::Int, b::Int) = a + b
f(a::String, b::String) = "wow i got $a and $b"
f(a::AbstractVector) = sum(a)
f(a, b, c::Number) = c
f(a, b, c::Float64) = c^2
```

```
julia> f(1, 2)
3

julia> f("apples", "bananas")
"wow i got apples and bananas"

julia> f([1,2,3])
6

julia> f(1, 2, 42)
42

julia> f("x", "y", 0.1)
0.010000000000000002

julia> f(Dict(), Dict())
"generic method"
```

Multiple Dispatch, pt. 3

- Why is this system **absolutely awesome?** Because:
  - You can define new types to which existing operations apply
  - You can define new operations which apply to existing types

- I understand that this may mean nothing to you at the moment. But…
- This leads to an astonishing amount of code-reuse and composability out-of-the-box between different packages of the Julia ecosystem!
  - much more than happens in any of the other popular high-level languages

- We won't really use Multiple Dispatch in the rest of the talk, but I strongly recommend the talk "The Unreasonable Effectiveness of Multiple Dispatch" by Stefan Karpinski (co-creator of Julia) if you want to learn more!