

NetLogo Basics

EES 4760/5760

Agent-Based and Individual-Based Computational Modeling

Jonathan Gilligan

Class #3: Wednesday, August 28 2024

Agenda

1. Understanding structure of NetLogo models
 2. Elementary NetLogo commands
 3. Some principles of good programming
 4. Overview of agent-based modeling
-

Start NetLogo on the computer in front of you.

Remember: All slides from class are available at <https://ees4760.jgilligan.org>

Fundamentals of NetLogo

Four Fundamental Code Parts

```
globals []  
turtles-own []  
breed [wolves wolf]
```

1. Declaration of variables and collectives

```
to setup  
  clear-all  
  ask patches [ do-something ]  
  ask turtles [ do-something ]  
  reset-ticks  
end
```

2. Model initialization

```
to go  
  ask patches [ do-something ]  
  ask turtles [ do-something ]  
  tick  
end
```

3. Scheduled actions (tick)

```
to do-something-special  
  ...  
end  
  
to do-something-boring  
  ...  
end
```

4. Submodels (science and cosmetics)

Objects (Nouns)

Categories of objects:

1. Turtles

- Turtles are any kind of agent
- Turtles can move around
- Turtles have properties (`turtles-own`)

```
turtles-own [ age height hunger ]
```

- Your model can have more than one “breed” of turtle (e.g., wolves and sheep)

```
breed [wolves wolf]  
breed [sheeps sheep] ; names must be distinct
```

2. Patches

- Patches represent the environment in which turtles live
- Patches are always square and stationary
- Like turtles, patches can have properties `patches-own`

```
patches-own [ elevation fertility ]
```

3. Links: Connect pairs of turtles

4. Agentsets: groups of agents (turtles, patches, links)

5. The Observer: The global context in which the other objects exist.

Why “Turtles”?



- The Logo language originated before graphical displays were common.
- Seymour Papert invented a robot turtle that would roll around under computer control
 - It had a pen or pencil that it could raise or lower to trace out its path on a piece of paper.
- NetLogo agents share behavior with the original turtle:
 - Turn left or right
 - Move forward
 - Raise or lower a “pen” to trace their paths

Operations (Verbs)

Two kinds of operations:

1. Procedures

- Do things (eat, move, grow, buy, sell, ...)
- Defined using `to`:

```
to wander  
  right random 360  
  forward random 5  
end
```

2. Reporters

- Calculate something and return a value
- Defined using `to-report`:

```
to-report turtles-nearby  
  report count turtles-on neighbors  
end
```

Let's Build A Model!

A Simple Model of an Ecosystem

- The landscape is initialized with random amounts of sugar on each patch
- 100 turtles live on the landscape
- At each tick:
 - Every patch adds 0.05 to its sugar up to a maximum of 100
 - Every turtle's hunger increases by 1, up to a maximum of 10
 - Every turtle eats sugar until it is no longer hungry, or the sugar on that patch runs out
 - The turtle decides whether it wants to move:
 - If there are other turtles on the patch, or if there is not enough sugar on the patch to satisfy its hunger, then the turtle will move to the neighboring patch with the largest amount of sugar.

Create a New Model

- In NetLogo, open **File** menu and choose **New**
- Add three buttons:
 - “setup” (type “setup” in “Commands” space)
 - “go” (check the “forever” button)
 - “step” (type “go” in “Commands” space and “step” in “Display Name”)
- Go to the code tab and type this:

```
globals  
[  
  max-sugar  
  sugar-growth  
]  
turtles-own [ hunger ]  
patches-own [ sugar ]
```

Model Initialization (**setup**)

Example:

Include **only things done once**
to initialize the model

- *Note: anything after a semicolon is a comment and NetLogo ignores it.*
- All code exists in specific *contexts*: turtles, patches, observer, etc.
- If you write code for *turtles* in a *patch* context, it will cause an error
- Brackets ([...]) delimit *contexts*
 - The brackets under **create-turtles** contain a *context* with commands for each turtle that's created.
- **clear-all** and **reset-ticks** are called in the *observer* context.

```
to setup
  clear-all

  ; <set up patch variables>
  ; <paint patches in neat colors>

  create-turtles 100 ; number of turtles to
create
  [
    ; <set up turtle variables, etc.>
  ]

  ; <plot initial model state (histograms,
etc.)>

  reset-ticks
end
```

Initialize Your Model

Type this into the code tab for your model:

```
to setup
  clear-all

  set sugar-growth 0.050
  set max-sugar 100

  ask patches
  [
    ; this is a patch context
    set sugar random max-sugar
    update-pcolor
  ]

  create-turtles 100
  [
    ; this is a turtle context
    setxy random-xcor random-ycor
    set hunger 5
    update-color
  ]
  reset-ticks
end

to update-pcolor
end

to update-color
end
```

Scheduled Actions (go)

Type this into your model:

- “go” is repeated over and over to execute model.
- Include **only** stuff to be executed **each time step**
- Keep the “go” procedure simple and neat
 - For complicated stuff, use submodels
- Include termination point

```
to go
  tick
  ask turtles [
    if hunger < 10 [ set hunger hunger + 1 ]
    eat
    move
    update-color
  ]
  ask patches
  [
    if sugar < max-sugar [ set sugar sugar + sugar-growth ]
    update-pcolor
  ]
  if ticks > 2000 [ stop ]
end

; Submodels

to eat
end

to move
end

to update-pcolor
end

to update-color
end
```

Tricky Things

tick vs. ticks

- `tick` — (verb) increments time by one period
- `ticks` — (noun) measures the time elapsed since the start
- More technical explanation:
 - NetLogo has an internal tick counter
 - `tick` increments the tick counter
 - `ticks` reports the current value of the tick counter

tick vs. ticks

Good code:

```
to go
  tick

  if ticks > max-ticks
    [stop]
  ...
  ask turtles [set age ticks]
end
```

Bad code:

```
to go
  ticks

  if tick > max-ticks
    [stop]
  ...
  ask turtles
    [
      tick
      set age ticks
    ]
end
```


Elementary NetLogo Commands

Elementary NetLogo Commands

- a. Searching NetLogo dictionary
- b. Working with `agentsets`
- c. Working with variables
- d. Code branching (conditional statements)
- e. Working with stochasticity
- f. Working with graphics
- g. How to make your code legible to others (documentation, comments, and tabbing)

Searching NetLogo Dictionary

- NetLogo dictionary is a web page
- Use “Find on this page” in your web browser.

Working with agentsets (*ask*)

- An agentset is a group of zero or more turtles, patches, etc.
 - Plural nouns (*turtles*, *patches*) refer to agentsets.
 - Singular nouns (*turtle*, *patch*) refer to individual agents.
- “*ask*” tells an agent or all members of an agentset to do the code in the following brackets:

```
ask turtles [ forward 5 ]
```
- All members of the agentset do the code, one at a time, in random order.
 - The commands are executed in each agent’s context
- Be careful not to put anything in the brackets that should not be repeated for each member of the agentset!

How are these different?

```
ask turtles
[
  buy
  sell
  update-bank-account
]
```

```
ask turtles [buy]
ask turtles [sell]
ask turtles [update-bank-account]
```

Working with agentsets (`with`)

- `turtles` is an agentset of all turtles.
- “`with`” is one of many primitives that create a subset of an agentset:
`ask turtles with [color = blue] [move]`
- Similar keywords for subsetting:
 - `with-min`, `with-max`
 - `n-of`, `max-n-of`, `min-n-of`,
 - `one-of`, `max-one-of`, `min-one-of`
- Use the dictionary to look up correct syntax.

Working with agentsets (*of*)

- “*of*” provides a *list* of the values of an *–own* variable
`set happiness min [happiness] of turtles-on neighbors`
 - In a patch context, `neighbors` makes an agentset of the neighboring patches.
 - `turtles-on` makes an agentset of all the turtles on `neighbors`
 - `[happiness]` gets the *turtles–own* variable `happiness` in the context of each turtle in `turtles-on neighbors`
- More generally, “*of*” is a primitive for getting a value from *another* agent or agentset: if `friend` is another turtle,
`set happiness [happiness] of friend`
- Use the dictionary to look up correct syntax.

Add Movement to Our Model

```
to move
  if hunger > sugar
    [
      move-to max-one-of neighbors [ sugar ]
    ]
end
```

If there isn't enough sugar to satisfy the turtle, it moves to the neighboring patch with the most sugar.

- We run `move` in the context of a turtle:
 - For each neighbor of the patch the turtle is on, look up `sugar` in the context of that neighbor
 - The brackets `[...]` signal the change of context.
 - Find the neighbors with the maximum `sugar`, and pick one of them at random
 - In the context of the original turtle, move to that neighboring patch

Working with agentsets (`=`, `set`)

- Two fundamental kinds of operations:
 - Changing the value of a variable:
 - Assignment operations (`set`)
 - Checking to see whether a value satisfies some condition:
 - Conditional operations (`=`, also `>`, `<`, `>=`, `<=`, `!=`)

Equals or no equals?

Assignment statements

- Wrong:

```
happiness = ([happiness] of a-neighbor-turtle)
```

- Right:

```
set happiness ([happiness] of a-neighbor-turtle)
```

Conditional statements (Boolean: yes or no)

```
if happiness = 3  
  [stop]  
  
if happiness <= 3  
  [stop]  
  
if happiness != 5 or ticks > 17  
  [stop]
```

Working with variables: `set` vs. `let`

- Global variables (known to all procedures)
- Local variables (known only to one procedure)
- Use `let` to *create and set the value* of a new local variable:

```
let mean-neighbor-size mean [size] of turtles-on neighbors
```

- Use `set` to change the value of an existing variable (global, local, patch, turtle, etc.)

```
set wealth wealth * 1.1  
  
set hypotenuse sqrt (a ^ 2 + b ^ 2)
```

Working On Our Model

Type this into “code” tab to update `to eat` and `to move` in our model:

```
to eat
  ifelse hunger > sugar
  [
    ; Use set to change an existing variable "hunger"
    set hunger hunger - sugar
    set sugar 0
  ]
  [
    set sugar sugar - hunger
    set hunger 0
  ]
end

to move
  if hunger > sugar or any? turtles-here
  [
    ; Use let to create a new variable "dest"
    let dest max-one-of neighbors [ sugar ]
    move-to dest
  ]
end
```

Working with variables: Giving a value to another agent

- How does one patch (or turtle) give the value of one of its variables to other patches?
 - There are two ways to do this.

```
ask neighbors [set pcolor [pcolor] of myself]
```

```
let my-color pcolor  
ask neighbors [set pcolor my-color]
```

- Turtles implicitly access `patches-own` variables (e.g., `pcolor`, `sugar`) of the patch they're on as though they were `turtles-own`
- Converse is not true: Patches don't automatically see `turtles-own`
- Why?
 - A turtle can only be on one patch at a time,
 - But a patch may have multiple turtles.

Code branching (conditional statements)

```
ifelse (boolean condition)
[
  ; Do this if condition is true ...
]
[ ;else
  ; Do this if condition is false
]
```

Working with stochasticity (randomness)

- Uniform distribution of random numbers between a and b :

```
a + random (b-a)
```

- Normal distribution with mean m and std. deviation s :

```
random-normal m s
```

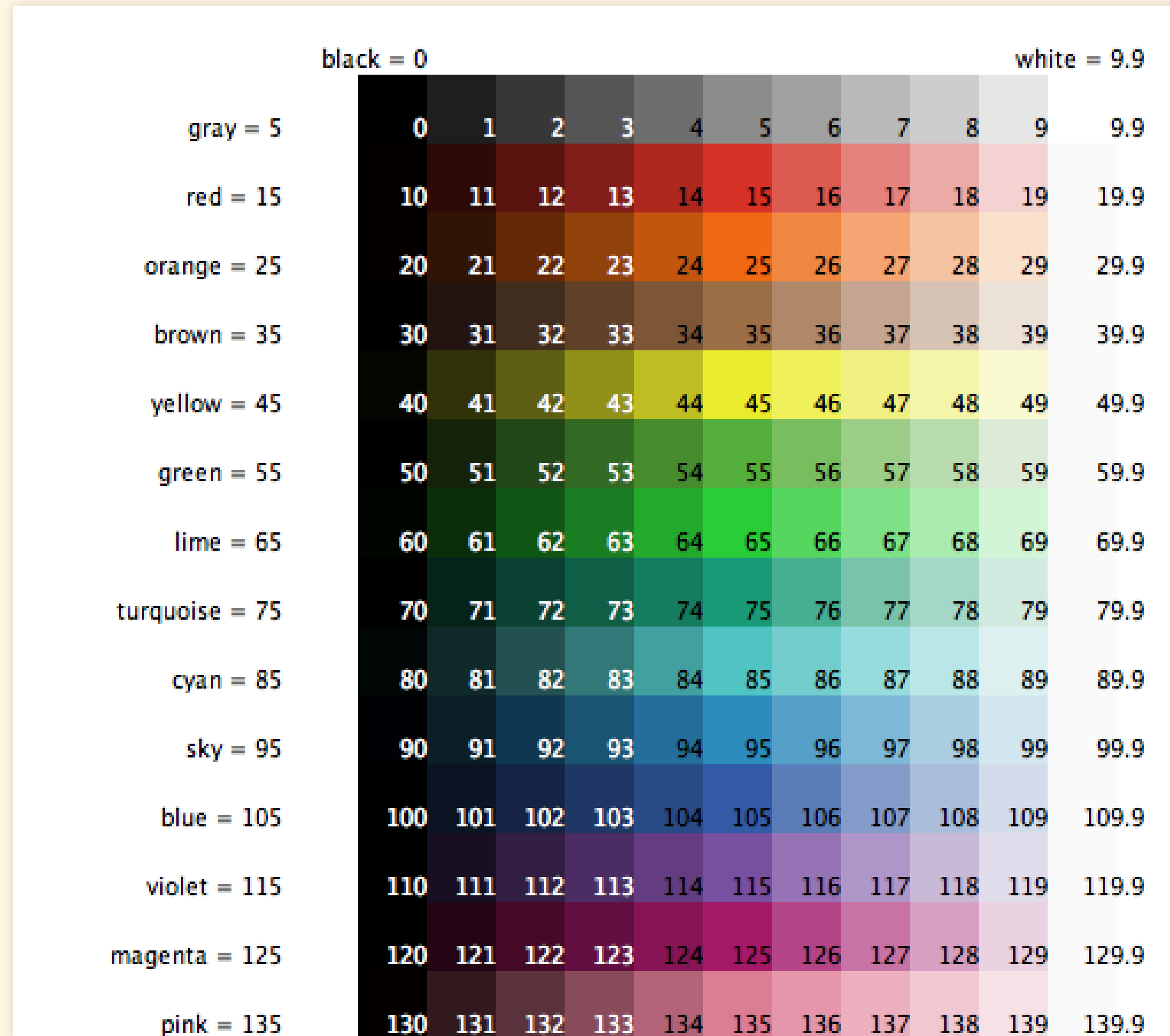
- Selecting one patch at random and turn it green

```
ask one-of patches [set pcolor green]
```

- Selecting one agent at random from an agentset and turn it right 5 degrees:

```
ask one-of turtles [right 5]
```

Working with graphics



Updating Our Model

Type the following into the “code” tab to update the procedures `update-pcolor` and `update-color`

```
to update-pcolor
  set pcolor scale-color yellow sugar 0 (2 * max-sugar)
end

to update-color
  ifelse hunger > 5
  [
    set color scale-color red hunger 15 5
  ]
  [
    set color scale-color green hunger 5 -5
  ]
end
```

- `scale-color color number range1 range2` sets the lightness of the color. Higher numbers = lighter, lower = darker.
- If `range1 > range2`, light and dark are reversed.

Running Our Model

- Press “Check” and make sure there are no syntax errors
- Go to “Interface” tab
- Click on “setup”
- Click on “go”
- You can download a copy of the model from
https://ees4760.jgilligan.org/models/class_03/class_03_example.nlogo

Monitoring and Interacting with a Model

On the “interface” tab:

- Right click and add a Plot
 - Name the plot “Hunger”
 - Set X max to 10 and Y max to 100
 - Type “Hunger” for “X axis label” and “# Turtles” for “Y axis label”
 - Click on the pencil icon under “default” pen
 - Choose “Bar” for “Mode”
 - In “Pen update commands” type `histogram [hunger] of turtles`
 - Press “OK”
- Right click and add a Slider
 - Type “sugar-growth” into “Global Variable”
 - Set minimum to 0, increment to 0.005, maximum to 0.1, and value to 0.050
- Open the code tab and comment out definition and initialization of `sugar-growth`

```
globals
[
  max-sugar
  ; sugar-growth
]
...
to setup
  clear-all
  ; set sugar-growth 0.050
  ...
```

Play with the model

- Do interesting things happen for different values of `sugar-growth`?
- It might be fun to comment out the line in `to go` that stops the model after 2000 ticks

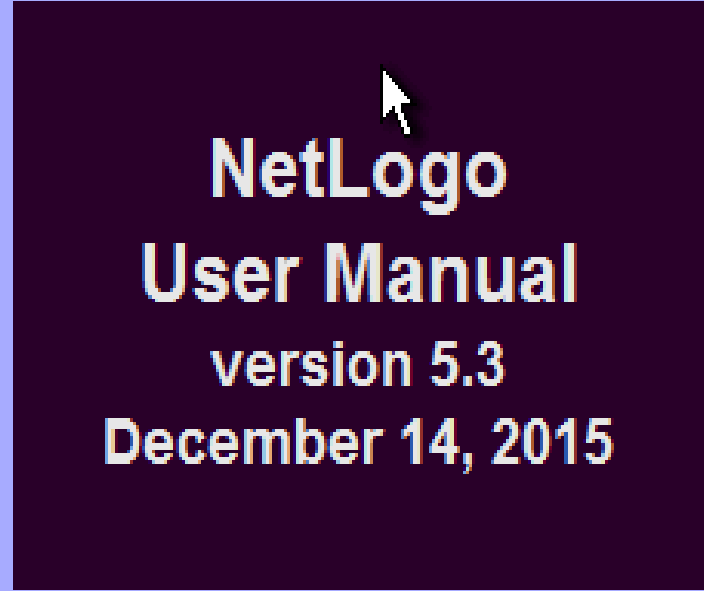
```
; if ticks > 2000 [ stop ]
```

Good Practices for Programming

Making your code legible to other people

1. Comment, comment, comment.
 - Variable declaration: purpose and legal values
 - Procedure: purpose and description
 - Submodel equations: explain and cite the source
2. Indent code so it shows clear blocks
3. After you're finished coding, take time to write detailed documentation (Chapter 3 will describe documentation)

When in doubt, use this.



NetLogo
User Manual
version 5.3
December 14, 2015

Release Notes
[What's New?](#)
[System Requirements](#)
[Contacting Us](#)
[Copyright / License](#)

Introduction
[What is NetLogo?](#)
[Sample Model: Party](#)

Learning NetLogo
[Tutorial #1: Models](#)
[Tutorial #2: Commands](#)
[Tutorial #3: Procedures](#)

Reference
[Interface Guide](#)
[Info Tab Guide](#)
[Programming Guide](#)
[Transition Guide](#)
[NetLogo Dictionary](#)

NetLogo Dictionary

NetLogo 5.3 User Manual

Alphabetical: [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [L](#) [M](#) [N](#) [O](#) [P](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [?](#)

Categories: [Turtle](#) - [Patch](#) - [Agentset](#) - [Color](#) - [Task](#) - [Control/Logic](#) - [World](#) - [Perspective](#)
[Input/Output](#) - [File](#) - [List](#) - [String](#) - [Math](#) - [Plotting](#) - [Links](#) - [Movie](#) - [System](#) - [HubNet](#)

Special: [Variables](#) - [Keywords](#) - [Constants](#)

Categories

This is an approximate grouping. Remember that a turtle-related primitive might still be used by patches or the observer, and vice versa. To see which agents (turtles, patches, links, observer) can actually run a primitive, consult its dictionary entry.

Turtle-related

[back](#) ([bk](#)) [<breeds>-at](#) [<breeds>-here](#) [<breeds>-on](#) [can-move?](#) [clear-turtles](#) ([ct](#)) [create-<breeds>](#) [create-ordered-<breeds>](#) [create-ordered-turtles](#) ([cro](#)) [create-turtles](#) ([crt](#)) [die](#) [distance](#) [distancexy](#) [downhill](#) [downhill4](#) [dx](#) [dy](#) [face](#) [facexy](#) [forward](#) ([fd](#)) [hatch](#) [hatch-<breeds>](#) [hide-turtle](#) ([ht](#)) [home](#) [inspect](#) [is-<breed>?](#) [is-turtle?](#) [jump](#) [layout-circle](#) [left](#) ([lt](#)) [move-to myself](#) [nobody](#) [no-turtles](#) [of](#) [other](#) [patch-ahead](#) [patch-at](#) [patch-at-heading-and-distance](#) [patch-here](#) [patch-left-and-ahead](#) [patch-right-and-ahead](#) [pen-down](#) ([pd](#)) [pen-erase](#) ([pe](#)) [pen-up](#) ([pu](#)) [random-xcor](#) [random-ycor](#) [right](#) ([rt](#)) [self](#) [set-default-shape](#) [__set-line-thickness](#) [setxy](#) [shapes](#) [show-turtle](#) ([st](#)) [sprout](#) [sprout-<breeds>](#) [stamp](#) [stamp-erase](#) [stop-inspecting](#) [subject](#) [subtract-headings](#) [tie](#) [towards](#) [towardsxy](#) [turtle](#) [turtle-set](#) [turtles](#) [turtles-at](#) [turtles-here](#) [turtles-on](#) [turtles-own](#) [untie](#) [uphill](#) [uphill4](#)

Agent-Based Models

Agent-based models

- Agents/Individuals are discrete, unique, and autonomous entities.
 - Discrete entities: Important at low densities
 - Unique: Individuals, even of same age and species, can be **different**
 - Individuals have a **life history**
- Interactions among individuals are usually **local**, not global
- Individuals make decisions, which can be **adaptive**
- Ecology or society **emerges** from individual behavior (bottom-up)

Why agent-based models?

1. Individuals/agents are unique and different from one another
2. Individuals/agents interact locally
3. Individuals/agents show adaptive behavior

Why agent-based models?

Use ABM if *one or more* of the following are **essential** to your research question:

1. Individual variability
2. Local interactions
3. Adaptive behavior

- ABMs that include all three elements can be called **full-fledged**.
- Most ABMs focus on only one or two elements.

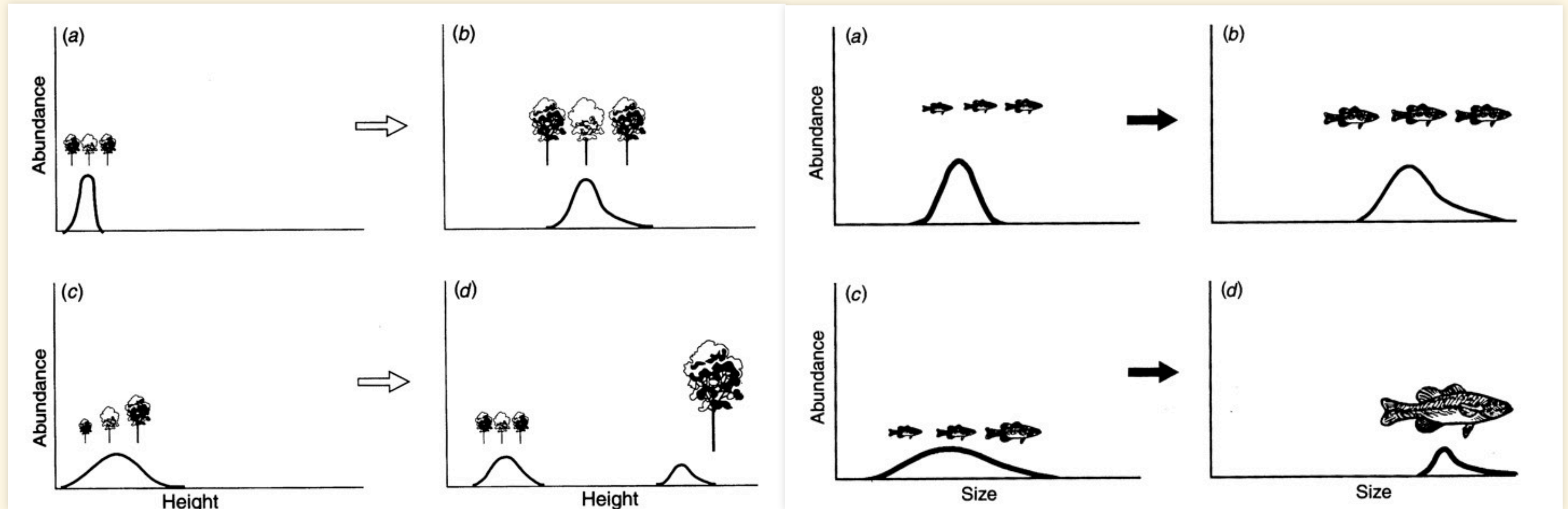
Why **not** agent-based models?

- Too complex
- Too data hungry.
- Too many parameters unknown.
- Too much uncertainty in model structure.
- Hard to test.
- Require too much person and computer power.

When ABMs are too hard, use aggregated modeling techniques:

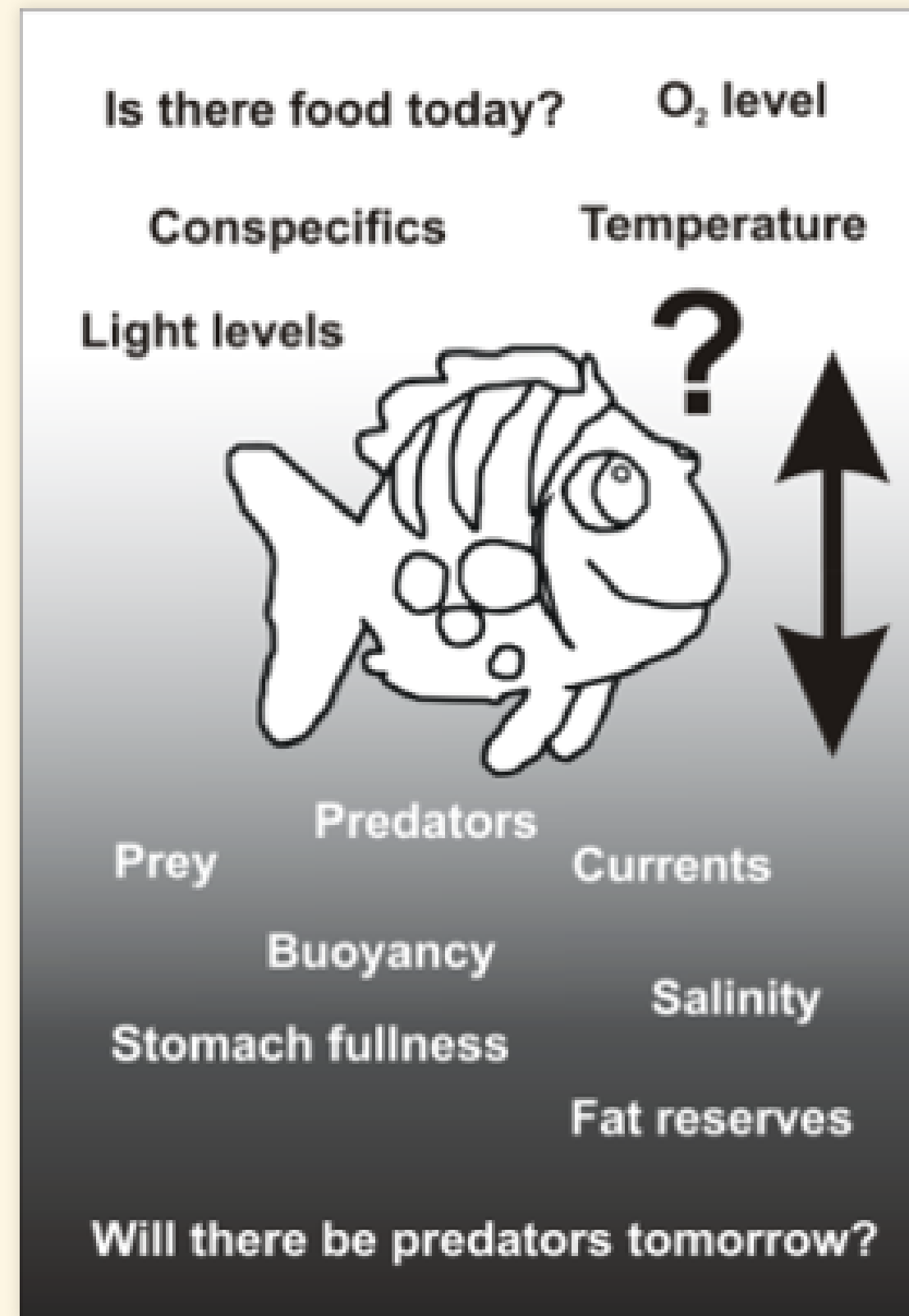
- Microeconomics looks at aggregate supply and demand;
 - does not model individual consumers and producers.
- Biology can use population dynamics without looking at individuals
- Chemists model chemical reactions with rate constants,
 - not individual atoms and molecules.

Individual variability



From Huston, M., *et al.*, BioScience **38**, 682 (1988)

Adaptive behavior



Adaptive behavior: Characteristic patterns in trout habitat selection



Adaptive behavior: Characteristic patterns in trout habitat selection

- Habitat:
 - Use shallow habitat when small
 - Avoid aquatic predators
 - Use deep habitat when big
 - Avoid terrestrial predators
 - Shift when predators, larger competitors are introduced
- Hierarchical feeding: big fish get the best spots
- Move to margins during floods
- Seek slower, quieter habitat when water is turbid
- Seek slower flow when water is cold

Source: S.F. Railsback & B.C. Harvey. 2002. *Ecology* **83**, 1817–1830. doi: 10.1890/0012-9658(2002)083[1817:AOHSRU]2.0.CO;2

Example: flocks of starlings

- Thousands of individuals
 - unique and different
 - interact locally
 - show adaptive behavior

Behavioral Ecology
doi:10.1093/beheco/arq149

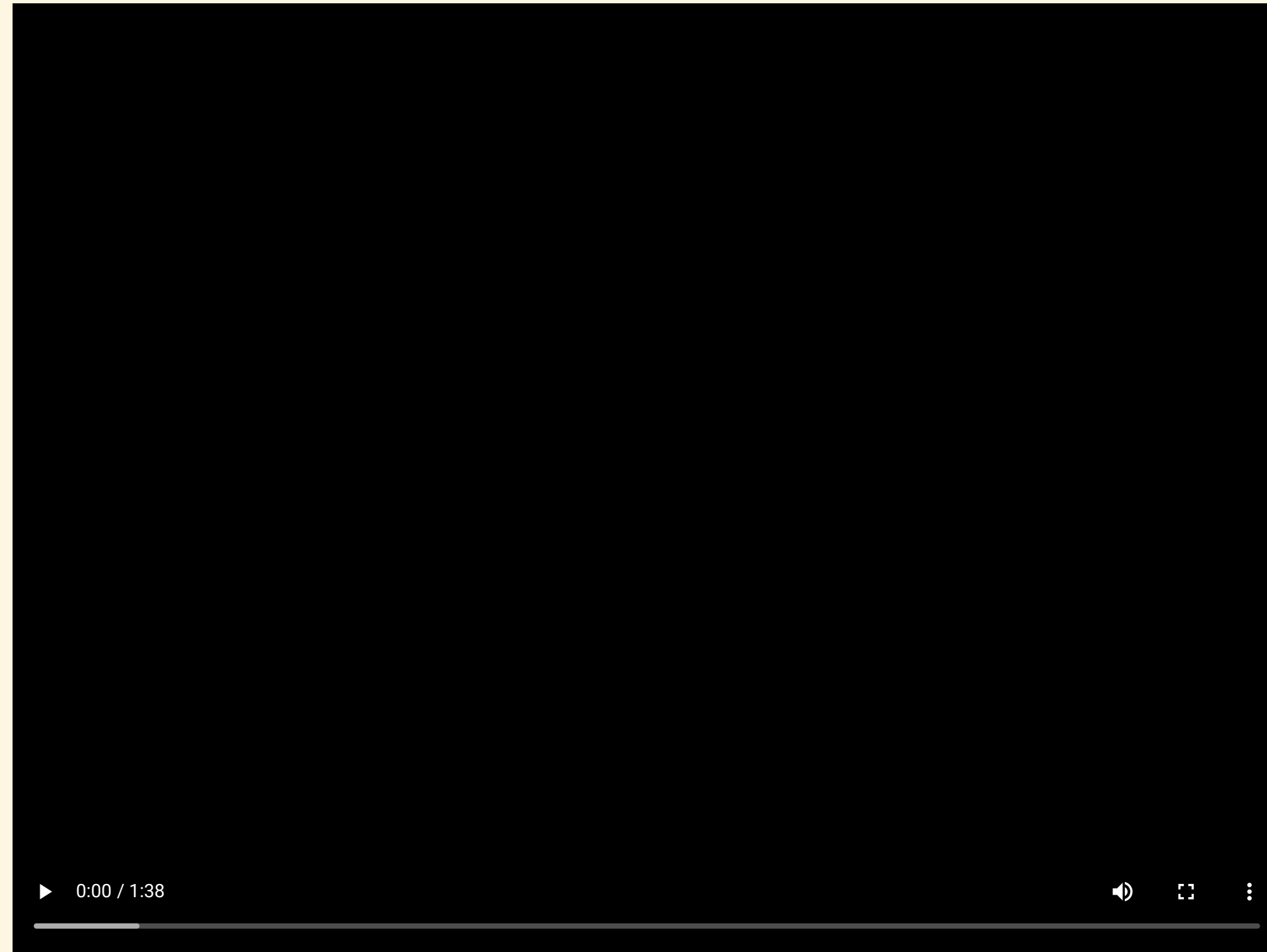
Self-organized aerial displays of thousands of starlings: a model

H. Hildenbrandt,^a C. Carere,^{b,c} and C.K. Hemelrijk^a

^aTheoretical biology, Behavioural Ecology and Self-organisation, Centre for Ecological and Evolutionary Studies, University of Groningen, PO Box 14, 9750 AA, Haren, The Netherlands, ^bCNR-INFM, Dipartimento di Fisica, Università di Roma La Sapienza, P.le A. Moro 2, 00185 Roma, Italy, and ^cDipartimento di Ecologia e Sviluppo Economico Sostenibile Università degli Studi della Tuscia, Viterbo, Italy

Through combining theoretical models and empirical data, complexity science has increased our understanding of social behavior of animals, in particular of social insects, primates, and fish. What are missing are studies of collective behavior of huge swarms of birds. Recently detailed empirical data have been collected of the swarming maneuvers of large flocks of thousands of starlings (*Sturnus vulgaris*) at their communal sleeping site (roost). Their flocking maneuvers are of dazzling

Starling murmuration



By Liberty Smith & Sophie Windsor Clive, Islands and Rivers, <https://vimeo.com/31158841>

Flock of thousands of starlings



Simulated flock of thousands of starlings



Simulated flock of thousands of starlings

