Cs620 final term topic 126 to 160

BS Computer science (Virtual University of Pakistan)

Collections of Agents

Types of Agents

Agents are typically divided into three main types:

mobile agents that can move about the landscape;

stationary agents that cannot move at all; and

connecting agents that link two or more other agents.

NetLogo

In NetLogo, turtles are mobile agents, patches are stationary agents, and links are connecting agents. From a geometric standpoint, turtles are generally treated as shapeless area-less points, although they may be visualized with various shapes and sizes). Thus, even if a turtle appears to be large enough to be on several patches at the same time, it is only contained by the patch at the turtle ' s center (given by XCOR and YCOR). Patches are sometimes used to represent a passive environment and are acted upon by the mobile agents; other times, they can take actions and perform operations.

Patches and Turtles

A primary difference between patches and turtles is that patches cannot move. Patches also take up a defined space/area in the world; thus, a single patch can contain multiple turtle agents on it.

Links

Links are also unable to move themselves. They connect two turtles that are the " end nodes " of the link, but the visual representations of the links do move when their end nodes move.

Links are often used to represent the relationship between turtles. They can also represent the environment — e.g., by defining transportation routes that agents can move along, or by representing friendship/communication channels. Despite these differences, all three share the ability to behave, that is, to take actions and perform operations on themselves.

Breeds of Agents

Besides these three predefined agent-types, modelers can also create their own types of agent called breeds. The breed of an agent designates the category or class to which the agent belongs. In the Traffic Basic model, all of the agents are of the same type, so it is not necessary to distinguish between different agent breeds. Instead, we used agents of the " turtles " breed, the default breed in NetLogo. Different breeds of agents are required if different agents have different properties or actions.

Wolf and sheep

we had two breeds of agents, the " wolf " the " sheep " breeds. Even though these two breeds had the same set of properties, it was useful to create two breeds because

each had different characteristic actions — wolves eat sheep, while sheep eat grass. We defined the breeds at the beginning of the model:

```
breed [sheep a-sheep]
breed [wolves wolf]
```

At the same time, we can define the properties that breeds have:

```
turtles-own [energy]
```

→ Syntax

In this case, both the wolves and the sheep have the same property of " energy, " but we could also give them separate properties. For instance, we could give the wolves a " fangstrength " property and the sheep a " wooliness " property. The fang-strength could be used to determine how successful a wolf is at killing sheep. Wooliness, by contrast, could be an indication of how well the sheep was able to fend off the wolves ' attack, if we assume that the wolves sometimes get a mouthful of wool instead of flesh. If this were the case, we would add two additional lines:

```
sheep-own [ wooliness ]
wolves-own [ fang-strength ]
```

These properties are in addition to the properties that all agents have, so that the sheep would have “ wooliness ” and “ energy ” just as the wolves have “ fang-strength ” and “ energy. ”

Sets of Agents Breeds are particular collections of agents where the collections are defined by the kinds of properties and actions of their ag

ents. We can also define collections of agents in other ways. NetLogo uses the term agentset to designate an unordered collection of agents, and we will also use this term/definition throughout this textbook. Usually we construct agentsets either by collecting agents that have something in common (e.g., agent location or other properties) or by randomly selecting a subset of another agentset.

In the Traffic Basic model, we could create a set of all the agents that have a speed over 0.5. In NetLogo, this is usually done with the WITH primitive. Here is an example of how we could create such an agentset that we could insert into the GO procedure in this model:

```
let fast-cars turtles with [speed > 0.5]
```

However, we usually want to ask these turtles to do something specific. For instance,

We can ask all of the turtles that have a high speed to set their size bigger so they are easier to see.

```
let fast-cars turtles with [speed > 0.5]
ask fast-cars [
  set size 2.0
]
```

The “ let ” statement above collects the fast cars into an agentset. If we don’t plan to “ talk to ” this agentset again, we can do without the LET and instead construct the agentset “ on the fly ” and ask the turtles directly:

```
ask turtles with [speed > 0.5] [
    set size 2.0
]
```

If you insert this code into the model, you will soon see that all of the cars are big. This is because we never told the turtles to set their size to small again, once their speed has dropped below 0.5. To do that we would need to add some more code. One way to accomplish this would be to use another ask.

```
ask turtles with [speed > 0.5] [
    set size 2.0
]
ask turtles with [speed <= 0.5] [
    set size 1.0
]
```

Another way to accomplish the same goal, without creating agentsets, is to ask all the turtles to do something, but choose what actions they take based on their properties. In the previous example, all of the faster turtles took their actions before any of the slower turtles took their actions. In the following example, all of the agents are asked with the same ASK command, so the order in which the turtles take actions will be random. In this case the result of running the code will be the same, but depending on what actions the agents take (for instance, if they were to change their speed, instead of their size), the results could be different.

In contrast to lists, which can hold any type of item, an can hold only agents.

```
ask turtles [
    ifelse speed > 0.5 [
set size 2.0
]
[
    set size 1.0
    ]
]
```

Another method for creating agentsets is on the basis of their location. The Traffic Basic model does this in the GO procedure:

```
let car-ahead one-of turtles-on patch-ahead 1
        ifelse car-ahead != nobody
            [ slow-down-car ]
```

TURTLES-ON PATCH-AHEAD 1 creates an agentset of all of the cars in the patch in front of the current car. It there is at least one such car, it then selects one of these cars at random, using the ONE-OF primitive, and sets the speed of the current car to a little less than the speed of that car ahead. There are many other ways to access a collection of agents based on their location, such as using NEIGHBORS, TURTLES-AT, TURTLES-HERE, and IN-RADIUS.

Agentsets and Lists Throughout our examples, we have used both agentsets and lists,

Sometimes explicitly and sometimes implicitly. This may be a good time to stop and clarify what they are, how they work, how they differ, and under what circumstances we would use one over the other. Agentsets and lists are both variables that can contain one or many other variables. We can specify that a variable is an agentset or a list when we create them, by using their respective constructor reporters. If we want to create an empty list, we can write

```
let a-list []
```

If we want, we can then add numbers, strings, and even turtles to the list.

```
;; we put items at the start of the list
set a-list fput 1 a-list
set a-list fput "and" a-list
set alist fput turtle 0 a-list
show a-list
;; prints [(turtle 0) "and" 1
```

In contrast to lists, which can hold any type of item, an agentset can hold only agents.

Moreover, it can hold only agents of the same type, such as turtles, patches, or

NetLogo has special reporters for empty agentsets, no-turtles, no-patches, and no-links.

```
;; this creates an empty agentset of turtles
let an-agentset no-turtles
```

No-turtles is an empty turtle agentset (i.e., an agentset containing no turtles), so by

Setting an-variable to no-turtles, we specify that this variable is of the type agentset.

Now that we have an empty agentset, we can then add turtles to it by using the turtle-set reporter:

```
;; this adds turtle 0 to the agentset
set an-agentset (turtle-set turtle 0)
;; this adds turtle 1 and turtle 2 to the agentset
set an-agentset (turtle-set an-agentset turtle 1 turtle 2)
show an-agentset
;; prints (agentset, 3 turtles)
```

We can only put agents (turtle-breeds, patches and links) in agentsets, but we can put

Anything we want, including agents, in lists. There are two important properties that set agentsets and lists apart from each other.

First, we can “ ask ” agentsets to do things, but we cannot ask lists to do things. So for instance, when we use

```
ask turtles [] ;; do stuff
```

What we are really doing is first creating an agentset of all turtles, and then asking all those turtles, in a random order, to do whatever we put in the brackets. If we try this with lists, we will simply get an error.

Second, agentsets are unordered. This means that whenever we invoke them, they will output a list of agents in a random order. Notice how showing a list shows both what is inside the list and the order in which it appears, but showing an agentset shows only what is inside the agentset.

So when we want to interact with turtles, when would we use one or the other? Unless

We have a very good reason to, we always use agentsets. The primary reason is that we usually want our turtles to do things in a random order. That is because there could be threats to model validity if the agents always execute in the same order. For instance, in the Wolf Sheep Predation model, some sheep would have an unfair advantage if they are always first to check if there is grass on their patches.

But sometimes we do want to determine the order in which turtles do things. For instance, it could be that we want some turtles to have an advantage. In that case we would need to create an ordered list, and order them by whatever parameter we think determines their advantage. If turtles have different information, we might, for example, want the turtles with more information to take their turns later.

```
turtles-own [information] ;; information determines how late they get their turn
;; (later is better because then they will know what everyone else did
;; before making their decision)

;; create an empty list
let a-list []
;; sort-on reports a list containing turtles sorted by the parameter specified
;; in the brackets
set a-list sort-on [information] turtles
```

a-list now contains all turtles sorted by information in ascending order. But, as we just discussed, we cannot ask lists of agents to do things. Rather, we must iterate through the lists, and ask each agent in the list to do what we ask. For this we can use the foreach command:

B

```
foreach a-list [
    ask ? [ ] ;; do stuff here
]
```

The " ? " is a special variable that takes on the value of each element of the list. So this code will iterate through each turtle in the list, and ask each in that particular order, to do what we specify in the brackets.

Similarly we can create lists of patches. Suppose we want to label the patches with numbers in left-to-right, top-to-bottom order. We can create a sorted list of patches and then label them using the code below:

```
;; patches are labeled with numbers in left-to-right,
;; top-to-bottom order
let n 0
foreach sort patches [
    ask ? [
       set plabel n
       set n n + 1
    ]
]
```

Agentsets and Computation Before we end our discussion of agentsets, it should be noted that once you ask an agentset to perform an action, all the agents collected at that moment (and only those agents) will perform the action. If one agent ' s (agent A) action causes another agent (agent B) in the agentset to no longer satisfy the collection criteria, B will still perform the action. Likewise, if A ' s action causes agent C, which is not in the agentset, to meet the collection criteria, C will still not perform the action.

Agent-set Ordering model

As an illustration, let ' s consider this code snippet from the Agentset Ordering model in the chapter 5 subfolder of the IABM Textbook folder in the models library:

```
to setup
    clear-all
    create-turtles 100 [
       set size random-float 2.0
       forward10
    ]
end

to go
    ask turtles [
       set color blue
       ]
    ask turtles with [ size < 1.0 ] [
       ask one-of turtles with [size > 1.0] [
           set size size - 0.5
       ]
       set color red
    ]
    print count turtles with [ color = red ]
    print count turtles with [ size < 1.0 ]
end
```

The bolded code has the potential to change the set of agents with size < 1.0 by making some agents smaller, thereby changing the agentset defined in the first ASK.

Agent-set Efficiency model

```
;; GO-1 sets red patch labels to a small random number and
;; green patch labels to a larger random number
to go-1
    if any? patches with [ pcolor = red] [
       ask patches with [ pcolor = red] [
              set plabel random 5 ;; red patches are labeled 0-4
       ]
    ]
    if any? patches with [ pcolor = green] [
       ask patches with [ pcolor = green] [
              set plabel 5 + random 5 ;; green patches are labeled 5-9
       ]
    ]
    tick
end
```

This code computes each of the two agentsets PATCHES WITH [PCOLOR = RED] and PATCHES WITH [PCOLOR = GREEN] twice.

```
;; GO-2 has the same behavior as go-1 above. But it is more efficient as it
computes each of the patch agentsets only once.
to go-2
 let red-patches patches with [ pcolor = red ]
 let green-patches patches with [ pcolor = green ]
 if any? red-patches [
     ask red-patches [
    set plabel random 5
    ]
]
if any? green-patches [
    ask green-patches [
    set plabel 5 + random 5
    ]
]
 tick
end
```

The procedure GO-2 below has the same behavior as GO-1, but it is more efficient in that it computes each of those agentsets only once.

A more pernicious issue arises if we change just one line of GO-1 and

GO-2:

```
;; GO-3 shows what happens if patch colors are changed "on the fly"
to go-3
    if any? patches with [ pcolor = red] [
       ask patches with [ pcolor = red] [
              set pcolor green
       ]
    ]
    if any? patches with [ pcolor = green] [
       ask patches with [ pcolor = green] [
              set pcolor red
       ]
    ]
    tick
end
```

If you run this code, you might expect to get a picture that looks like figure 5.4; that is, you expect red patches to turn green and vice versa.

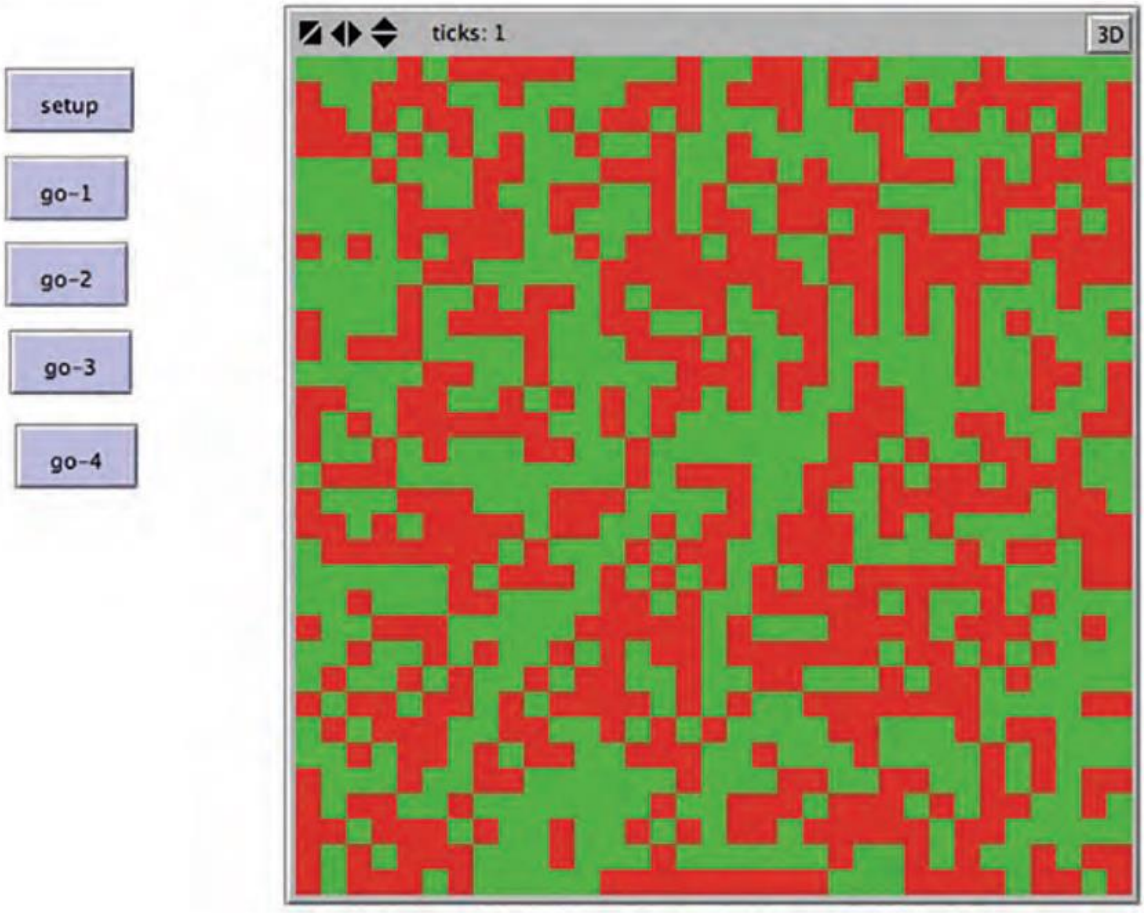


Figure 5.4
Constructing agentsets at the outset results in expected behavior.

In fact, this is not what happens. Instead, this code will result in a picture like figure 5.5, where all the patches are red. Why does this unexpected behavior happen? This is another example of the ordering issue we just discussed above. The first " if statement " turns the patches green, so by the time the second " if statement " is executed, all the patches are green and therefore then turn red.

By computing the agentsets ahead of time, as in GO-4, you are not only using more efficient code but are also ensuring that the green-patches agentsets you ask to execute the instructions are the same green-patches agentsets as at the start of the procedure and not the set of green patches that result from the first " if-statement, " resulting in the expected picture of figure 5.4.

```
;; GO-4 shows what happens if you keep track, at the outset, of which patches
;; are red and which are green
to go-4
        let red-patches patches with [pcolor = red]
        let green-patches patches with [pcolor = green]
        if any? red-patches [
            ask red-patches [
                set pcolor green
            ]
        ]
        if any? green-patches [
            ask green-patches [
                set pcolor red
            ]
        ]
    tick
end
```

Lectures 127

Click here to download .pptx

The Granularity of an Agent

One of the first considerations when designing an agent-based model is at what granularity should you create your agent — at what level of complexity is the agent you are modeling.

Level of Complexity for the Agents

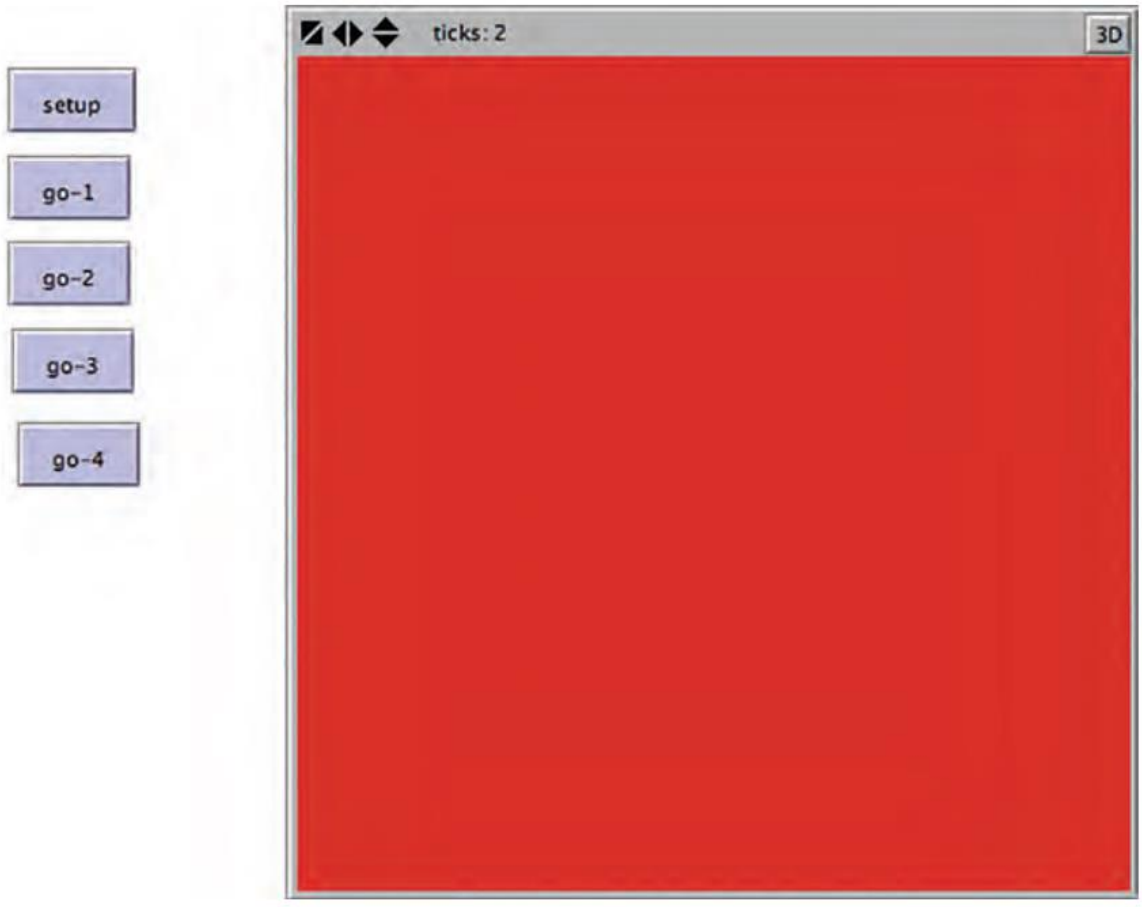


Figure 5.5
Constructing agentsets after the model state has changed, results in unintended outcome.

So how do you choose the level of complexity for the agents in your model? The guideline is to choose agents such that they represent the fundamental level of interaction that pertains to your question about the phenomenon.

Tumor Model

For instance, if you look at the Tumor model (shown in figure 5.6) in the Biology section of the NetLogo models library, the agents are cells within the human body, since the research question the model examines is how tumor cells spread.

NetLogo Tumor model

Lectures 128

Agent Cognition

As we have described, agents have different properties and behaviors. Still, how do agents examine their properties and the world around them to decide what actions to take? This question is resolved by a decision-making process called agent cognition.

Types Of Agent Cognition

We will discuss several types of agent cognition:

- Reflexive agents,
- Utility-based agents,
- Goal-based agents, and

Adaptive agents (Russell & Norvig, 1995).

Often, these types of cognition are thought of in increasing order of complexity, with reflexive agents being the simplest, and adaptive agents the most complex. In reality, though this is not a strict hierarchy of complexity.

Instead, these are descriptive terms that help us talk about agent cognition and can be mixed and matched; for instance, it is possible to have a utility-based adaptive agent.

Reflexive

Reflexive agents are built around very simple rules. They use if-then rules to react to inputs and take actions (Russell & Norvig, 1995). For instance, the cars in the Traffic Basic model are reflexive agents. If you look at the code that controls their actions, it looks like this:

```
let car-ahead one-of turtles-on patch-ahead 1  ;;choose a car on the patch ahead
ifelse car-ahead != nobody [  ;; if there is a car ahead
    slow-down-car car-ahead ;; set car speed to be slower than car ahead
]
[   ;; otherwise, speed up
    speed-up-car ;; increase speed variable by the acceleration
]
;; ….
fd speed
```

Put in words, this code says that if there are cars ahead, then slow down below the speed of the car in front; if there are not any cars ahead, then speed up. This is a reflexive action based on the state of the program, hence the name reflexive agent. This is the most basic form of agent cognition (and often a good starting point), but it is possible to make the agent cognition more sophisticated.

Utility-based Form of Agent Cognition

For instance, we could elaborate this model by giving the cars gas

Tanks and fuel efficiencies based on the speed they are going. We could then have the cars change their speeds in order to improve their fuel efficiencies. As a result, the agents might have to speed up and slow down at different times than they currently do to minimize their gas usage while still not causing accidents. Giving the agents this type of decision-making process would give them a utility-based form of agent cognition in which they attempt to maximize a utility function — namely, their fuel efficiency (Russell & Norvig, 1995).

Implementation Of Model Of Agent Cognition

To implement this model of agent cognition, we need to start by replacing our SPEED-UP-CAR procedure with a new procedure that accounts for the car ' s fuel efficiency.

```
;; choose a car on the patch ahead
let car-ahead one-of turtles-on patch-ahead 1
ifelse car-ahead != nobody [ ;; if there is a car ahead
    slow-down-car car-ahead  ;; set car speed to be slower than car ahead
]
[ ;; otherwise, adjust speed to find ideal fuel efficiency
    adjust-speed-for-efficiency ]
```

We want the ADJUST-SPEED-FOR-EFFICIENCY procedure to maintain the same

Speed if the car is at the maximally fuel efficient speed. If it is not at its most efficient

Speed, the logic in this procedure should have the car speed up if it is moving too slow and slow down if the car is moving too fast. Additionally, the car would still need to slow down if it was about to crash into the car in front of it. As such, the true utility function includes an exception that gives a utility of 0 to any action that results in a crash. As a result, we can leave the first part of the code that slows the car before it crashes and just add ADJUST-SPEED-FOR_EFFICIENCY when there is no car directly ahead, as illustrated in the code below from the Traffic Basic Utility model, of the NetLogo models library.

```
;; car procedure
to adjust-speed-for-efficiency
 if (speed != efficient-speed) [ ;; if car is at efficient speed, do nothing
        if (speed + acceleration < efficient-speed) [
;; if accelerating will still put you below the efficient speed then accelerate
            set speed speed + acceleration
        ]                                          ;;
;; if decelerating will still put you above the efficient speed then decelerate
        if (speed - deceleration > efficient-speed) [
            set speed speed - deceleration
        ]
    ]
end
```

Utility Functions

In the language of utility functions, each car agent is minimizing a function f,

Defined by:

$$f(v) = |v - v^*|$$

Where v is the current velocity of the car and v* is the most efficient velocity. The constraint that is described in the code is that the v cannot be adjusted arbitrarily, but instead, can only be changed by a limited increment every time step. By trying different values for EFFICIENT-SPEED, you can obtain quite different traffic patterns from the original model. For low values of EFFICIENT-SPEED, the system can achieve a free-flow state (with no jams) on a consistent basis.

Goal-based Agents

The third type of agent cognition we will discuss is goal-based cognition. Imagine that each car in the Traffic Grid model (see figure 5.8) from the social sciences section of the NetLogo models library has a home and a place of work, with the goal of moving from home to work in a reasonable amount of time. Now, not only do the agents have to be able to speed up and slow down, but they also have to be able to turn left and right. In this version of the model, the cars are goal-based agents, since they have a goal (getting to and from work) that they are using to dictate their actions.

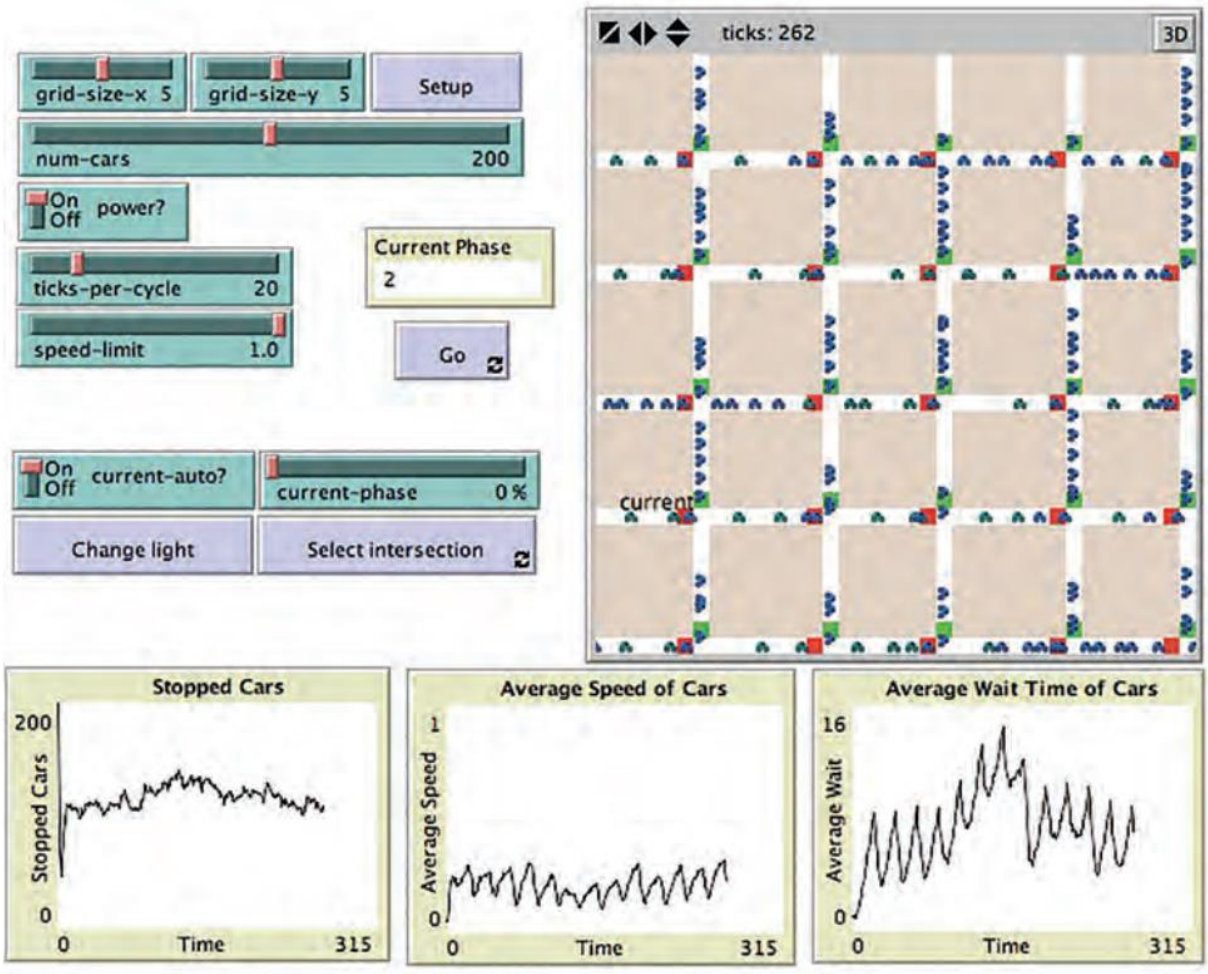


Figure 5.8
Traffic Grid model. http://ccl.northwestern.edu/netlogo/models/TrafficGrid (*Wilensky, 2002b*).

Implementing The Goal-based Version of the Model

To implement the goal-based version of the model, we start by defining the procedure that the cars will use to navigate between their two desired destinations. Instead of just moving straight all the time, as with the cars in the original model, the new model should have the car at each step deciding which of its neighboring road patches is the closest to its destination, subsequently proceeding in that direction. We do this in the procedure called NEXT-PATCH. First, each car checks if it has arrived at its goal, and if so, switches its goal:

```
;; if I am going home and I am on the patch that is my home
;; I turn around and head towards work (my goal is set to "work")
    if goal = house and patch-here = house [
       set goal work
    ]
;; if I am going to work and I am on the patch that is my work
;; I turn around and head towards home (my goal is set to "home")
    if goal = work and patch-here = work [
       set goal house
    ]
```

The code above doesn ' t quite work. That ' s because the house and work are off-road and the cars remain on the road, so the condition " patch-here = house " or " patch-here = work " will never be satisfied. As we placed the house and work adjacent to the road, we can fix this by having the car check if it is right next to its goal.

Lectures 129

Click here to download .pptx

Other Kinds of Agents

We have already discussed breeds and various types of agents, but there are two other special kinds of agents that deserve at least a brief mention.

1) The first type are meta-agents, agents made up of other agents;

2) The second is proto-agents, placeholder agents that allow you to define interactions for your fully defined agents with other entities that have not been fully developed.

Meta-Agents Many things that we would consider agents in reality are actually composed of other agents. In reality, all " agents " are composed of other agents; to cite the oft-told parable, " It ' s turtles all the way down. " In other words, there is always a lower level of detail that you can use to describe an agent in your model, at least until you reach the most basic level so far described by physics. For instance, if we have selected a human to be the agent, he or she is actually composed of many subagents, with these subagents in turn being different depending on how you view the human. You could view the subagents of a human as the systems of the body — e.g., the immune and the respiratory systems — or you could view the subagents of a human as psychological aspects like the intellect and emotion. Moreover, these subagents are not the most basic level; in our example, the systems of the body are made up of organs, tissues, and cells.

Meta-agents and Sub-agents

At each of these levels, we are describing the relationship between meta-agents (agents composed of other agents) and subagents (agents which compose other agents). Still, agents can also be both meta-agents and subagents at the same time. For instance, organs are composed of cells and are meta-agents. However, they also play a compositional role in the systems of the human body and are thus subagents.

We can use meta-agents in our ABMs by defining the subagents that make up our agents and providing them with their own actions and properties.

Meta-agent Appears to be a Single Agent

From the perspective of another meta-agent, a meta-agent appears to be a single

Agent. If a person meets another person, they do not (usually) directly interface with the heart or the lungs (unless the meeting takes place at the surgical table). Instead, they interface with the person as a whole.

Modeling Agents and Their Interactions

When we think of modeling agents and their interactions, we have to determine what level of granularity we want to describe the agent behaviors. However, we always have the option of refining our model by converting agents into meta-agents that describe the agents that constitute other agents. Sometimes, it can be useful to represent the agents in our models not as autonomous individuals, but instead, as meta-agents composed of other agents. NetLogo doesn ' t include explicit language support for these meta-agents, though there are commands for locking together the movement of several agents (e.g., the TIE command) that may be useful in some circumstances. However, there is nothing to prevent you from designing models that have groups of agents representing single agents.

Proto-Agents

To truly be an agent within the agent-based modeling framework, an entity must have its own properties or actions. Sometimes, though, it can be desirable to create agents that, rather than having their own properties or behaviors, are instead placeholders for future agents. We call these agents proto-agents, and their primary purpose is to enable us to specify how other agents would interact with them if they

Were fleshed out into full agents.

Example of Proto-agent

For instance, if you were creating a model of residential location decision making, you would have residents as agents; however, since where a resident lives is greatly influenced by places of employment and services (e.g., grocery stores and restaurants) you might also want to include these " service centers " as agents as well. The same level of detail necessary for the residents, who are the focal agents of concern, may not be necessary for the service centers. Instead, they might be rendered as placeholders to represent where residents might potentially find jobs and transact business. However, as you continue to refine the model, you might give the service centers additional decision-making abilities. For instance, they might have a more elaborate model of market demand and decide where to locate using their own properties and beliefs about the future growth of the world. Keeping these agents as proto-agents early on means that when you add in the more richly detailed versions to the model, you do not have to go back and revise your resident agents. Instead, you can use the interaction events that they had already been using to interface with the service center proto-agents.

Lectures 130

Click here to download .pptx

Environments

Another early and critical decision is how to design the environment of the agent-based model. The environment consists of the conditions and habitats surrounding the agents as they act and interact within the model. The environment can affect agent decisions, and, in turn, can be affected by agent decisions.

Environment Can Affect Agent

For instance, in the Ants model from chapter 1, the ants leave pheromone in the environment that changes the environment, and in turn changes the behavior of the ants. There are many different kinds of environments that are common in ABMs. In this section, we will discuss a few of the most common types of environments.

Implementation of an Environment

Before we discuss the types of environments, it is important to mention that the environment itself can be implemented in a variety of ways. First, the environment can be composed of agents such that each individual piece of the environment can have a full set of properties and actions.

NetLogo

In NetLogo, this is the default view of the environment — the environment is represented by the agentset of patches. This allows different parts of the environment to have different properties and act differently based on their local interactions.

Lectures 131

Spatial Environments

Spatial environments in agent-based models generally have two variants:

Discrete spaces and

Continuous spaces.

Continuous Spaces: In a mathematical representation, in continuous spaces between any pair of points, there exists another point,

Discrete Spaces: While in discrete spaces, though each point has a neighboring point, there do exist pairs of points without other points between them, so that each point is separated from every other point.

ABM Implementation

However, when implemented in an ABM, all continuous spaces must be implemented as approximations, so that continuous spaces are represented as discrete spaces where the spaces between the points are very small. It should be noted that both discrete and continuous spaces can be either finite or infinite.

Discrete Spaces: The most common discrete spaces used in ABM are lattice graphs (also sometimes referred to as mesh graphs or grid graphs), which are environments where every location in the environment is connected to other locations in a regular grid.

جس میں آپ اوپر نیچے آگے پیچھے ہر جگہ جا سکتے ہیں۔

Toroidal Square Lattice

For instance, every location in a toroidal square lattice has a neighboring location up, down, to the left, and to the right. As mentioned, the most common representation of the environment in NetLogo is patches, which are located on a 2D lattice underlying the world of the ABM (see figure 5.9 for a colorful pattern of patches whose code is simply ASK PATCHES [SET PCOLOR PXCOR * PYCOR]). This uniform connectivity makes them different from network-based environments.

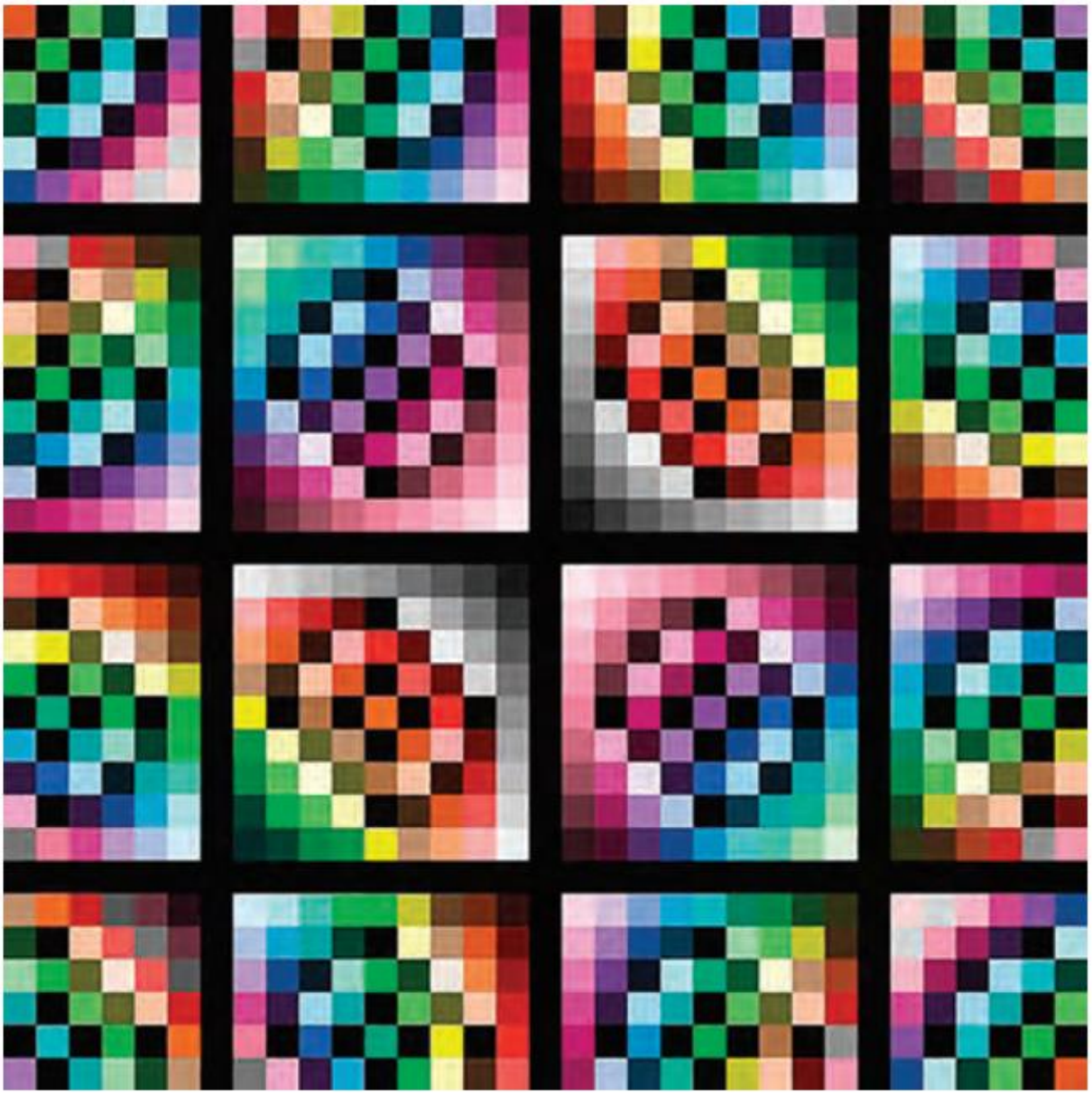


Figure 5.9
Patches displaying a range of colors.

The two most common types of lattices are:

- Square Lattices
- Hexagonal Lattices

Square Lattices The square lattice is the most common type of ABM environment. A square lattice is one composed of many little squares, akin to the grid paper used in mathematics classrooms.

There are two classical types of

Neighborhoods on a square lattice:

The von Neumann neighborhood, consisting of four

Neighbors located in the cardinal directions (see figure 5.10a); and

The Moore neighborhood,

Comprising the 8 adjacent cells (See figure 5.10b).

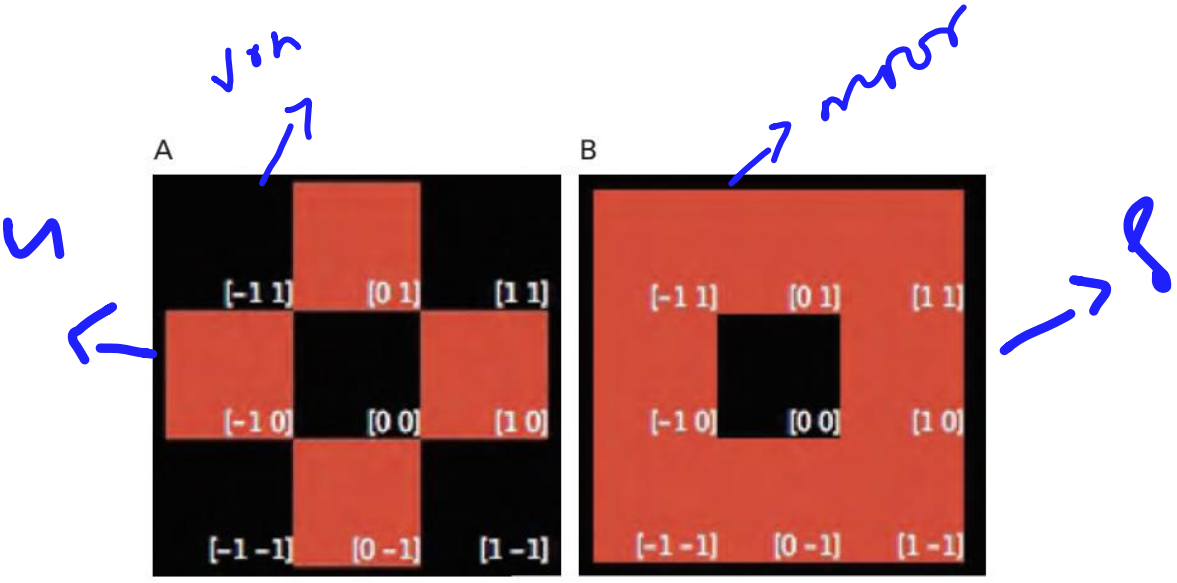


Figure 5.10
(a) Von Neumann Neighborhood. (b) Moore Neighborhood. The black cell in the center is the focal cell, and the red cells comprise its neighborhood.

Von Neumann Neighborhood

A von Neumann neighborhood (named after John von Neumann, a pioneer of cellular automata theory among other things) of radius 1 is a lattice where each cell has four neighbors: up, down, left, and right.

Moore Neighborhood

A Moore neighborhood (named after Edward F. Moore, another pioneer of cellular automata theory) of radius 1 is a lattice in which each cell has eight neighbors in the eight directions that touch either a side or a corner: up, down, left, right, up-left, up-right, down-left, and downright. In general, a Moore neighborhood gives you a better approximation to movement in a plane, and since many ABMs model phenomenon where planar movement is common, it is often the preferred modeling choice for discrete motion.

Hex Lattices: A hex lattice has some advantages over square lattices. The center of a cell in a square grid is farther from the centers of some of the adjacent cells (the diagonally adjacent ones) than other such cells. However, in a hex lattice, the distance between the center of a cell and all adjacent cells is the same. Moreover, hexagons are the polygons with the most edges that tile the plane, and for some applications, this makes them the best polygons to use. Both of these differences (equidistance between centers and the number of edges) mean that hex lattices more closely approximate a continuous plane than square lattices. But because square lattices match more closely a Cartesian coordinate system, a square lattice is a simpler structure to work with; even when a hexagonal lattice

Would be superior, many ABMs and ABM toolkits nevertheless employ square lattices.

However, with a little effort, any modern ABM environment can simulate a hexagonal lattice in a square lattice environment. For instance, you can see a hex lattice environment in the NetLogo Code examples, Hex Cells example, and Hex Turtles example (see figures 5.11 and 5.12).

Lectures 132

Click here to download .pptx

Network-Based Environments

Link: A link is defined by the two ends it connects, which are frequently referred to as nodes. However, we will use the network/node/link vocabulary throughout this module.

In NetLogo, links are their own agent-type.

Lattice networks

In fact, lattice graphs can also be called lattice networks, with the property that each position in the network looks exactly like every other position in the network. However, ABM environments usually do not implement lattice environments as networks for both conceptual and efficiency reasons. Additionally, using patches as the default topology allows for either discrete or continuous representations of the space, whereas a network is always discrete.

Random Networks

Network-based environments have been found useful in studying a wide variety of phenomena, such as the spread of disease or rumors, the formation of social groups, the structure of organizations, or even the structure of proteins. There are several network topologies that are commonly used in ABM. Besides the regular networks described earlier, the three most common network topologies are random, scale-free, and small-world.

In random networks, each individual is randomly connected to other individuals. These networks are created by randomly adding links between agents in the system. For example, if you had a model of agents moving around a large room and connected every agent in the room to another agent based on which agent had the next largest last two digits of their social security number, you would probably create a random network. We show one simple methods for creating a random network. This code is also in the Random Network model in the chapter 5 subfolder of the IABM Textbook folder of the NetLogo models library

Code for creating a random network

Lectures 133

Click here to download .pptx

Special Environments

The two methods we have demonstrated for defining the environment, two-dimensional (2D) grid-based (i.e., lattice) and network-based, are both instances of " interaction topologies. " Interaction topologies describe the paths along which agents can communicate and interact in a model. Besides the two interaction topologies we have looked at so far, there are several other standard topologies to consider. Two of the most interesting topologies involve the use of 3D worlds and Geographic Information Systems (GIS). 3D worlds allow agents to move in a third dimension as well as the two dimensions in traditional ABMs. GIS formats enable the importation of layers of real-world geographical data into ABMs. We will discuss each of these in turn.

3D Worlds

3D environments enable model developers to explore complex systems which are irreducibly bound up with a third dimension as well as sometimes increasing the apparent physical realism to their models.

There is a version of NetLogo called NetLogo 3D (it is a separate application in the NetLogo folder) that allows modelers to explore ABMs in three dimensions. There are

Many implementations of classic ABMs that have been developed for this environment in the NetLogo 3D models library (Wilensky, 2000). For instance, there is a three-dimensional version of the Percolation model (see figure 5.16). click here to download .pptx

Special Environments

The two methods we have demonstrated for defining the environment, two-dimensional (2D) grid-based (i.e., lattice) and network-based, are both instances of " interaction topologies. " Interaction topologies describe the paths along which agents can communicate and interact in a model. Besides the two interaction topologies we have looked at so far, there are several other standard topologies to consider. Two of the most interesting topologies involve the use of 3D worlds and Geographic Information Systems (GIS). 3D worlds allow agents to move in a third dimension as well as the two dimensions in traditional ABMs. GIS formats enable the importation of layers of real-world geographical data into ABMs. We will discuss each of these in turn.

3D Worlds

3D environments enable model developers to explore complex systems which are irreducibly bound up with a third dimension as well as sometimes increasing the apparent physical realism to their models.

There is a version of NetLogo called NetLogo 3D (it is a separate application in the NetLogo folder) that allows modelers to explore ABMs in three dimensions. There are many implementations of classic ABMs that have been developed for this environment in the NetLogo 3D models library (*Wilensky, 2000*). For instance, there is a three-dimensional version of the Percolation model (see figure 5.16).

Lectures 13 4

Interactions

Now that we have discussed both agents and the environments in which they exist, we will look at how agents and environments interact.

There are five basic classes of interactions that exist in ABMs:

Agent-self

Environment-self

Agent-agent

Environment-environment

Agent-environment

We will discuss each in turn along with some examples of these common interactions.

1) Agent-Self Interactions Agents do not always need to interact with other agents or the environment.

In fact, a lot of agent interaction is done within the agent.

For instance, most of the examples of advanced cognition that we discussed in the Agent section involve the agent interacting with itself.

The agent considers its current state and decides what to do. One classic type of self-agent interaction is birth.

Birth events are a typical event in ABMs

In these, one agent creates another agent. Below is the birth routine:

```
;; check to see if this agent has enough energy to reproduce
to reproduce
    if energy > 200 [
       set energy energy - 100  ;; reproduction costs energy to the parent
       hatch 1 [ set energy 100 ] ;; which is transferred to the offspring
   ]
end
```

As you can see, the agent considers its own state and, based on this state, decides whether or not to give birth to a new agent. It then manipulates its state, lowering its energy and creating the new agent. This is the typical way of having agents reproduce.

2 Environment-Self Interactions Environment-self interactions are when areas of the Environment alter or change themselves. For instance, they could change their internal

State variables as a result of calculations. The classic example of an environment-self interaction is when the grass regrows:

```
;; regrow the grass
to regrow-grass
    ask patches [
       set grass-amount grass-amount + grass-regrowth-rate
       if grass > 10 [
           set grass 10
                          ]
recolor-grass
    ]
end
```

Each patch is asked to examine its own state and increment the amount of grass it has, but if it has too much grass then it is set back to the maximum value it can contain.

3) Agent-Agent Interactions Interactions between two or more agents are usually the most important type of action within agent-based models. We saw a canonical example of agent-agent interactions in the Wolf Sheep Predation model when the wolves consume the sheep:

```
;; wolves eat sheep
to eat-sheep
    if any? sheep-here [ ;; if there are sheep here then eat one
       let target one-of sheep-here
       ask target [
           die
       ]
       ;; increase the energy by the parameter setting
       set energy energy + energy-gain-from-sheep
    ]
end
```

In this case, one agent is consuming another agent and taking its resources, whereby the wolf always eats the sheep. However, it is also possible to add competition or flight

To this model, where the wolf gets a chance of eating the sheep and the sheep gets a chance to flee. Competition is another example of agent-agent interaction.

Communication

A final example of agent-agent interaction is communication. Agents can share information about their own state as well as that of the world around them. This type of interaction allows agents to gain information to which they might not have direct access.

4) Environment-Environment Interactions: Interactions between different parts of the Environment are probably the least commonly used type of interaction in agent-based Models. However, there are some common uses of environment-environment interactions: one of these is diffusion. In the Ants model discussed in chapter 1, the ants place a pheromone in the environment, which is then diffused throughout the world via an environment-environment interaction. This interaction is contained in the following piece of code in the main GO procedure:

```
diffuse chemical (diffusion-rate / 100)
ask patches
    [ set chemical chemical * (100 - evaporation-rate) / 100
;; slowly evaporate chemical
       recolor-patch ]
```

3) Agent-Environment Interactions: Agent-environment interactions occur when the agent manipulates or examines part of the world in which it exists, or when the environment in some way alters or observes the agent. A common type of agent- environment interaction involves agents observing the environment. The Ants model demonstrates this kind of interaction when the ants examine the environment to look for food and sense pheromone:

```
to look-for-food ;; turtle procedure
    if food > 0
    [ set color orange + 1      ;; pick up food
        set food food - 1               ;; and reduce the food source
        rt 180                                          ;; and turn around
    stop ]
    ;; face in the direction where the chemical smell is strongest
    if (chemical >= 0.05) and (chemical < 2)
    [ uphill-chemical ]
end
```

In the Ants model, the patches contain food and chemical, so the first part of this code checks to see if there is any food in the current patch. If there is food, the ant then picks up the food and turns around back to the nest, and the procedure stops.

Otherwise, the ant checks to see if there is chemical, wherein it follows the chemical in that direction.

Agent Movement

Another common type of agent-environment interaction is agent movement. In some ways, movement is simply an agent-self interaction, since it only alters the current agent's state.

In the Ants model the ants move around by " wiggling ".

Lectures135

Click here to download .pptx

Observer/User Interface

Now that we have talked about the agents, the environments, and the interactions that occur between agents and environmental attributes, we may discuss who controls the running of the model.

The observer Is a high-level agent that is responsible for ensuring that the model runs and proceeds according to the steps developed by the model author.

The observer issues commands to agents and the environment, telling them to manipulate their data or to take certain actions. Most of the control that model developers have with an ABM is mediated through the observer. However, the observer is a special agent. It does not have many properties, though it can access global properties like any agent or patch can.

Properties Specific to the observer

The only properties that one could consider to be specific to the observer are those relating to the perspective from which the modeled world is viewed. For instance, in NetLogo the view may be centered on a specific agent, or focusing a highlight on a certain agent, using the FOLLOW, WATCH, or RIDE commands (See figure 5.21).

User Input and Model Output

We have already made use of many of the standard ways

To interface with an ABM when we extended m

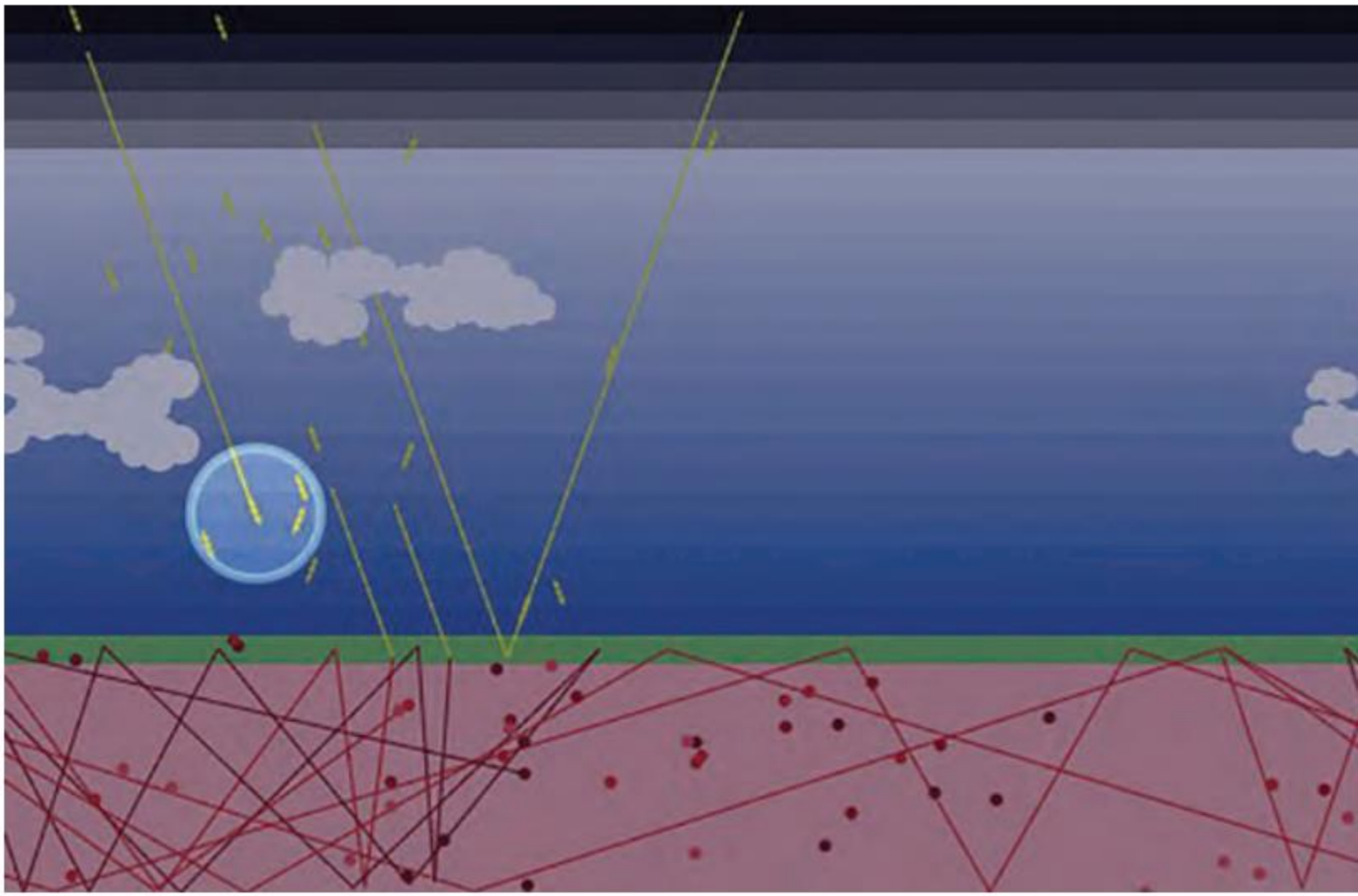


Figure 5.21
Climate Change model (with a sun ray turtle being watched, which is shown by the transpar
it). http://ccl.northwestern.edu/netlogo/models/ClimateChange (*Tinker & Wilensky, 2007*).

odels in chapter 3 and when we built our first model in chapter 4, but it is worth recapping some of them herein. ABMs require a control interface or a parameter group that allows the user to setup different parameters and settings for the ABM. The most common control mechanism is a button, which executes one or more commands within the model; if It is a forever button it will continue to execute those commands until the user presses the button again.

Command Center

A second way that the user can request that actions be performed within the ABM is via the command center (along with the mini – command centers within agent monitors). The command center is a very useful feature of NetLogo, as it allows the user to interactively test out commands, and manipulate agents and the environment.

Sliders, Switches & Input Boxes

Sliders enable the model user to select a particular value from a range of numerical Values. For instance, a slider could range from 0 to 50 (by increments of 0,1) or 1 to 1,000 by increments of 1. In the Code tab the value of a slider is accessed as if it were a global variable. Switches enable the user to turn various elements of a model off or on. In the Code tab, they are also accessed as global variables, but they are Boolean variables. Choosers enable a model user to select a choice from a predefined drop-down menu that the modeler has created. Again these are accessed as global variables in the Code tab, but they are variables that have strings as their values, these strings being the various choices in the chooser. Finally, input boxes are more free-form, allowing the user to input text that the model can use.

Monitors , Plots & Notes

As for output controls, monitors display the value of a global variable or calculation Updated several times a second. They have no history but show the user the current

State of the system. Plots provide traditional 2D graphs enabling the user to observe the change of an output variable over time. Output boxes enable the modeler to create freeform text-based output to send to the user. Finally, notes enable the modeler to place text information on the Interface tab (for example, to give a model user directions on how to use the model). Unlike in monitors, the text in notes is unchanging (unless you manually edit them).

Visualization

Visualization is the part of model design concerned with how to present the data contained in the model in a visual way. Creating cognitively efficient and aesthetic visualizations can make it much easier for model authors and users to understand the

Model. Though there is a long history of work on how to present data in static images

(Bertin, 1967; Tufte, 1983, 1996), there is much less work on how to represent data in

Real-time dynamic situations. However, attempts have been made to take current static

Guidelines and apply them to dynamic visualizations (Kornhauser, Rand & Wilensky,

2007). In general, there are three guidelines that should be kept in mind whenever designing the visualization of an ABM: simplify, explain, and emphasize.

Guidelines for Designing Visualization

Simplify the visualization Make the visualization as simple as possible so that anything that does not present additional usable information (or that is irrelevant to the current point

being explained) has been eliminated from the visualization. This prevents the model user from being distracted by unnecessary “ graph clutter ” (Tufte, 1983).

Explain the components If there is an aspect of the visualization that is not immediately obvious then there should be some quick way to determine what that visualization is illustrating, such as a legend or description. Without clear and direct descriptions of what is going on the model user may misinterpret what the model author is attempting to portray. If a model is to be useful it is necessary that anyone viewing the model can easily understand what it is saying.

Emphasize the main point Model visualizations are themselves simplifications of all the possible data that a model could present to the model user. Therefore, a model visualization should emphasize the main points and interactions that the model author wants to explore, and, in turn, to communicate to end users. By exaggerating certain aspects of the visualization they can draw attention to these key results.

NetLogo Ethnocentrism model

Every NetLogo agent has a shape and color, and by selecting appropriate shapes and colors we can highlight some agents while backgrounding others. For instance, to simplify visualization if agents in our model are truly homogenous, like the ants, then we might make all the agents the same color, as in the Ants model. To explain the visualization we might change their shape to indicate something about their properties. For instance, in the Ethnocentrism model based on Hammond and Axelrod ’ s model (2003) agents employing the same strategies have the same shape, even if they are different colors.

click here to download .pptx

Schedule

The *schedule* is a description of the order in which the model operates. Different ABM toolkits can have more or less explicit representations of the schedule. In NetLogo, there is no single identifiable object that can be identified as " the schedule. " Rather, the schedule is the order of events that occur within the model, which depends on the sequence of buttons that the user pushes, and the code/procedures that those buttons run. We will first discuss the common SETUP/GO idiom which is employed in almost all agent-based models, and then move on to discuss some of the subtler issues concerning scheduling in ABMs.

SETUP and GO First of all, there is usually an initialization procedure that creates the agents, initializes the environment, and readies the user interface. In NetLogo this procedure is usually called SETUP and it executes whenever a user presses the SETUP button on a NetLogo model. The SETUP routine usually starts by clearing away all the agents and data related to the pervious run of the model. Then it examines how the user has manipulated the various variables controlled by the user interface creating new agents and data to reflect the new run of the model. For instance, in Traffic Basic the SETUP procedure looks like this:

```
to setup
    clear-all
    ask patches [ setup-road ]
    setup-cars
    watch sample-car
end
```

The other main part of the schedule is what is often called the main loop, or in NetLogo, the GO procedure. The GO procedure describes what happens in one time unit (or tick) of the model. Usually that involves the agents being told what to do, the environment changing if necessary, and the user interface updating to reflect what has happened. In Traffic Basic the GO procedure looks like this:

```
to go
;; if there is a car right ahead of you, slow down to a spee
    ask turtles [
        let car-ahead one-of turtles-on patch-ahead 1
        ifelse car-ahead != nobody
            [   slow-down-car car-ahead]
            ;; otherwise, speed up
            [ speed-up-car ]
    ;; don't slow down below speed minimum or speed up beyon
        if speed < speed-min [ set speed speed-min ]
        if speed > speed-limit [ set speed speed-limit ]
        fd speed ]
    tick
end
```

Asynchronous vs. Synchronous Updates

If a model uses an ***Asynchronous*** update schedule, this means that when agents change their state, that state is immediately seen by other agents.

In a ***Synchronous*** update schedule changes made to an agent are not seen by other agents until the next clock tick — that is, all agents update simultaneously.

Sequential vs. Parallel Actions

Within the realm of asynchronous updating, agents can act either sequentially or in parallel.

Sequential actions involve only one agent acting at a time while ***Parallel*** actions are those in which all agents act independently. In NetLogo (versions 4.0 and later), sequential action is the standard behavior for agents.

Moreover, for the agents to act truly in parallel, you would need parallel hardware so that the actions of each agent would be carried out by a separate processor.

However, there is an intermediate solution. ***Simulated concurrency*** uses one processor to simulate many agents acting in parallel.

Types of Measurements

Heretofore, we have examined ABMs, modified them, built them from scratch, and analyzed their behavior. In this module, we will learn to employ ABM to produce new and interesting results about the domain that we are investigating. What kinds of results can ABMs produce? There are many different ways of examining and analyzing ABM data.

Choosing just one of these techniques can be limiting; therefore, it is important to know the advantages and disadvantages of a variety of tools and techniques. It is often useful to consider your analysis methods *before* building the ABM, to enable you to design output that is conducive to your analysis.

Modeling the Spread of Disease

If someone catches a cold and is coughing up a storm, he might infect others. Those that he comes into contact with — his friends, co-workers, and even strangers — may catch the cold. If a cold virus infects someone, that person might spread that disease to five other people (six now infected) before they recover. In turn, those five other people might spread the cold to five more people each (thirty-one are now infected), and those twenty-five people might spread the cold to five additional people (a hundred and fifty-six people are now infected). In fact, the rate of infection initially rises exponentially.

However, since this infection count grows so quickly, any population will eventually reach the limit of the number of people who can be infected. For instance, imagine that the 156 people mentioned above all work for the same company of 200 individuals. It is impossible for the remaining 125 people to each infect five new people, and thus the number of infected people will tail off because there is no one left to infect. As we have described it so far, this simple model assumes that each person infects the same number of people, which is manifestly not the case in real contexts. As a person moves through

their workspace, it might be the case that, they happen to not see many people in one day, whereas another individual might see many people. Also, our initial description assumes that if one person infects five people and another person infects five, there will not be any overlap. In reality, there is likely to be substantial overlap. Thus, the spread of disease in a workplace is not as straightforward as our initial description suggests. Suppose that we are interested in understanding the spread of disease, and we want to build an ABM of such a spread. How should we go about doing it?

First, we need some agents that keep track of whether they are infected with a cold or not. Additionally, these agents need a location in space and the ability to move. Finally, we need the ability to initialize the model by infecting a group of individuals. That is exactly how the NetLogo model we will be discussing in this module behaves (See figure 6.1). Individuals move around randomly on a landscape and infect other individuals whenever they come into contact with them.

یعنی اگر لوگ زیادہ ہونگے تو infected بڑھنے کا خطرہ بھی زیادہ ہوگا۔

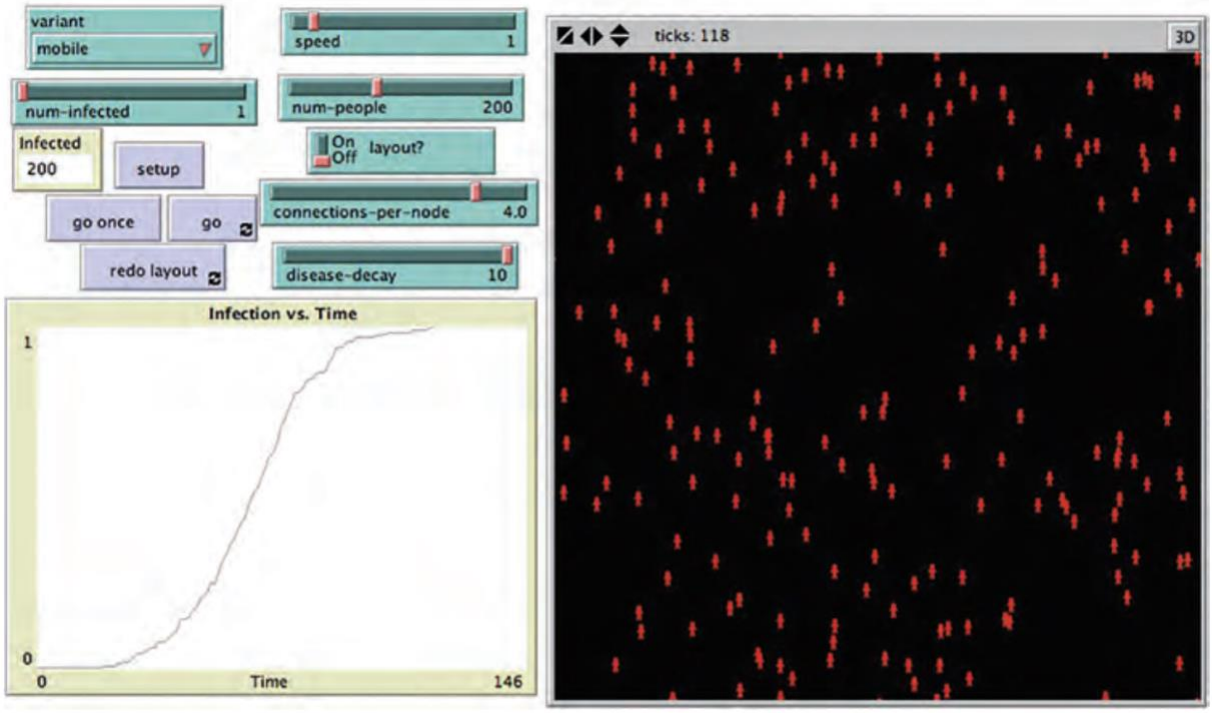


Figure 6.1
Spread of Disease model.

Though this model is simple, it exhibits interesting and complex behavior. For instance, what happens if we increase the number of people in the model? Does the disease spread quicker throughout the population, or does it take a longer time because there are more people? Let us run the model at population sizes of 50, 100, 150, and 200, and examine the results. We will keep the size of the world constant, so that, as we increase the number of individuals, we are also increasing the population density. Along the way we will write down at what time the entire population becomes infected (see table 6.1).

click here to download ppt

click here to download video

اسکا حل یہ ہے کہ آپ statistical کو بھی کریں analysis

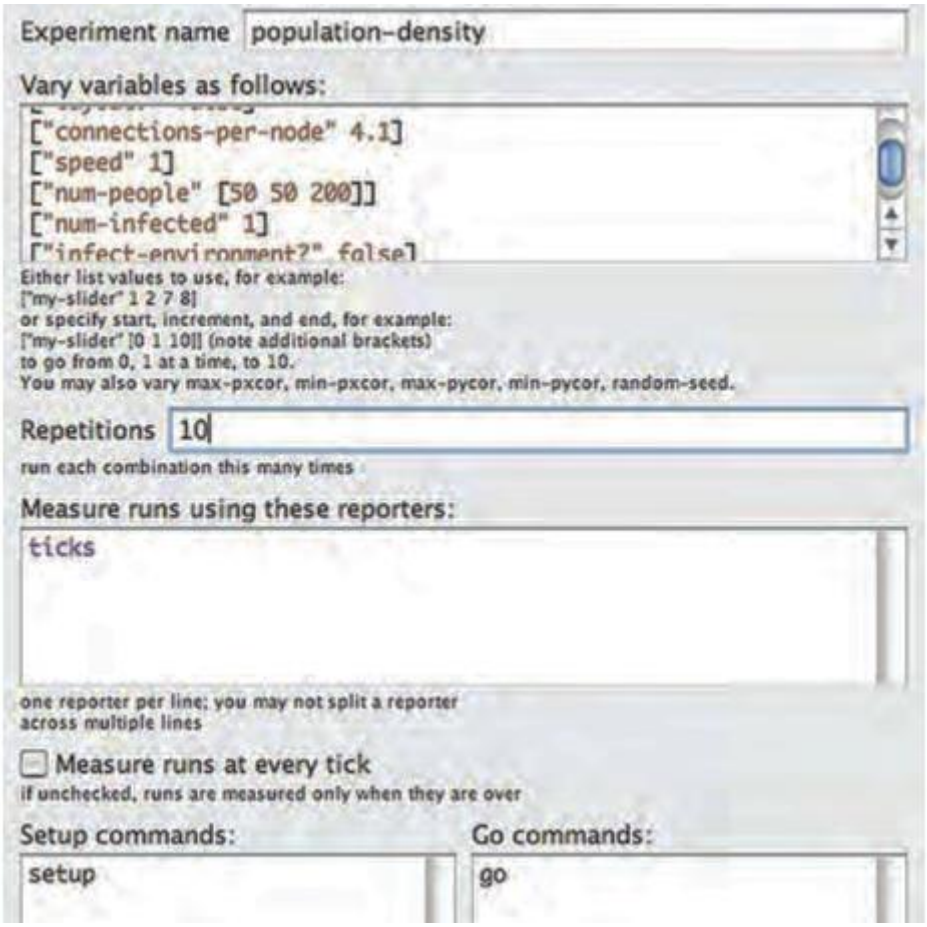
Here, we will understand about the different types of measurements in an ABM. Statistical Analysis of ABM:

- Moving beyond Raw Data

- Statistical results are the most common way of looking at any kind of scientific data
- The general methodology behind descriptive statistics is to provide numerical measures These measures summarize a large data set
- They also describe the data set in such a way that it is not necessary to examine every single value.

Summary Statistics

Population	Mean	Std. Dev.
50	366.8	47.39385802
100	213.8	27.40154091
150	144.4	17.65219533
200	118.7	12.12939497



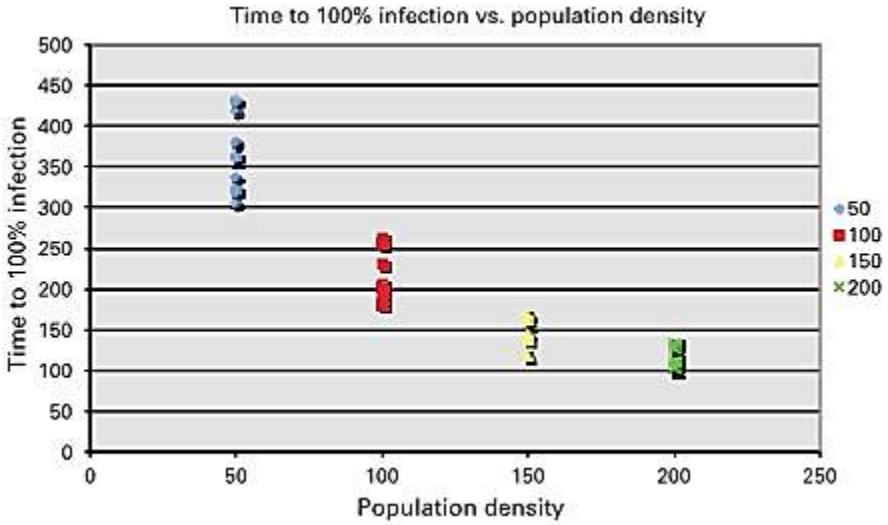
BehaviorSpace Data Imported into a Spreadsheet

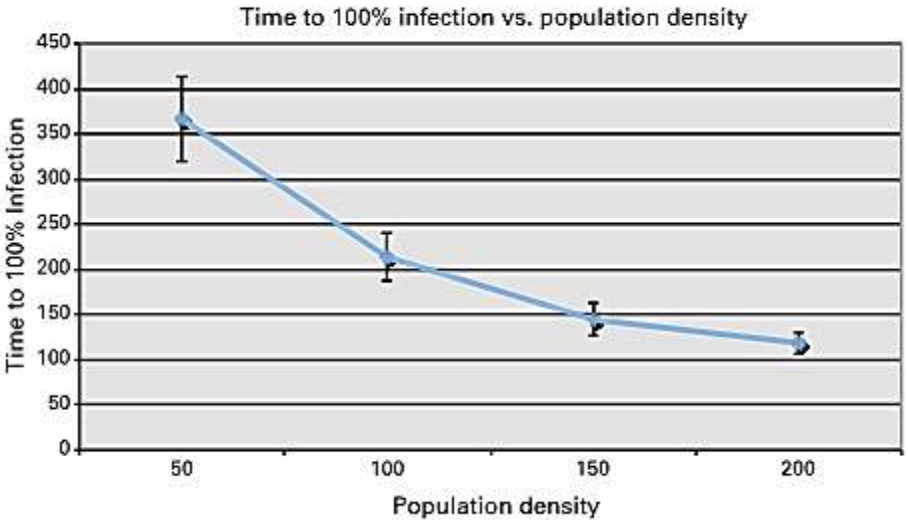
BehaviorSpace Table data

population-density

DATE
TIME

[run number]	network?	layout?	connections-per-node	speed	num-people	num-infected	infect-environment?	[tick]	ticks
1	FALSE	FALSE	4.1	1	50	1	FALSE	299	299
2	FALSE	FALSE	4.1	1	50	1	FALSE	432	432
3	FALSE	FALSE	4.1	1	50	1	FALSE	444	444
4	FALSE	FALSE	4.1	1	50	1	FALSE	400	400
5	FALSE	FALSE	4.1	1	50	1	FALSE	467	467
6	FALSE	FALSE	4.1	1	50	1	FALSE	397	397
7	FALSE	FALSE	4.1	1	50	1	FALSE	337	337





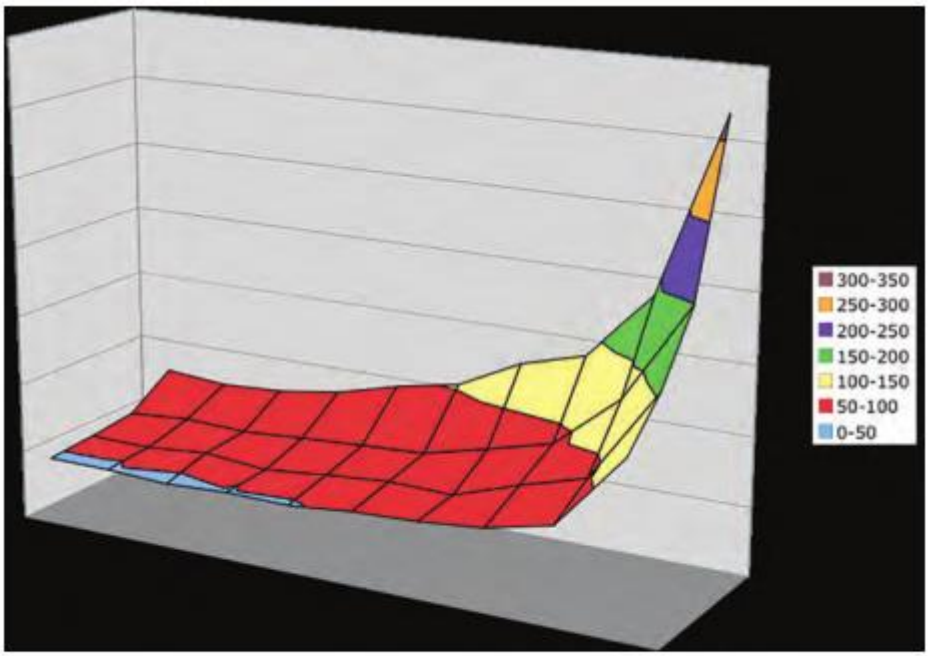


Figure 6.9

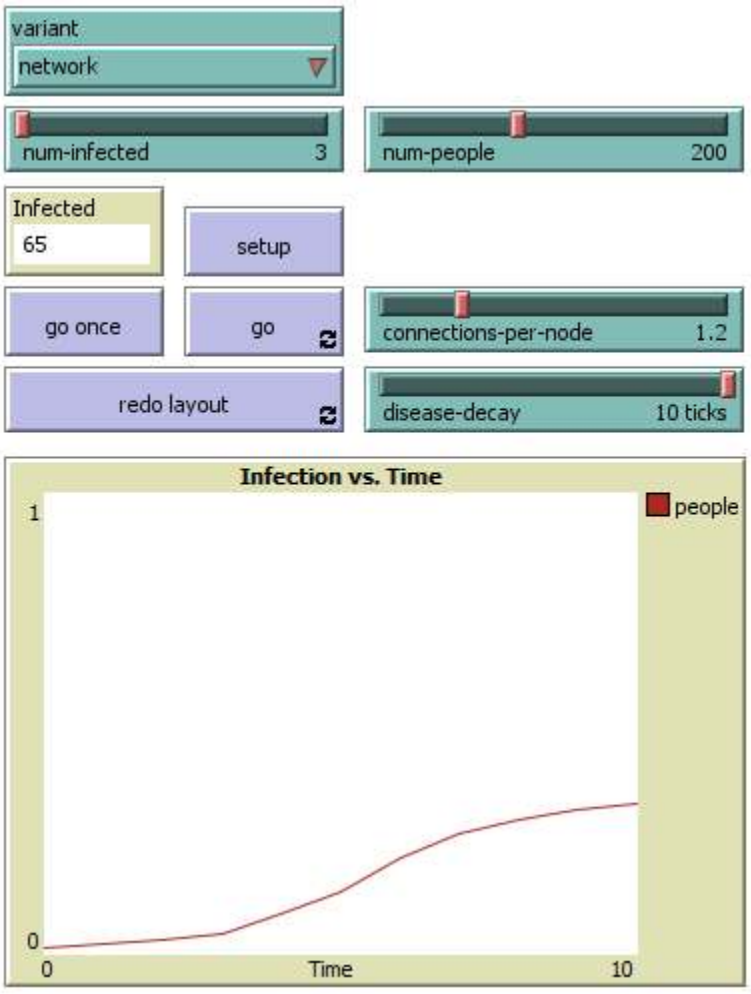
3D Chart of NUM-PEOPLE and DISEASE-DECAY versus time to 100 percent infection.

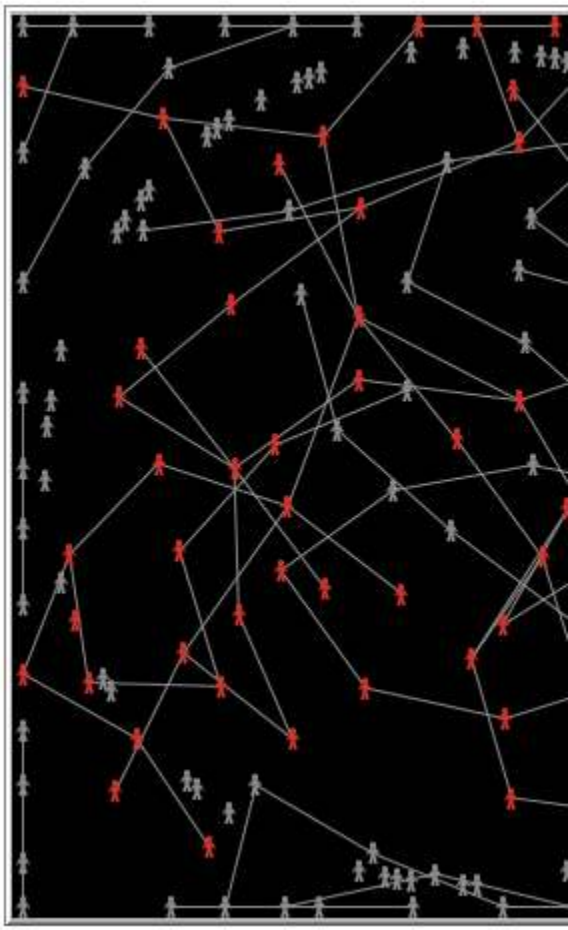
- We have learnt about the types of measurements in the analysis of an ABM.
- Credits: Uri Wilensky book.

click here to download .pptx

Here we will Model the spread of disease in detail.

- If a cold virus infects someone, that person might spread that disease to five other people (six now infected) before they recover.
- In fact, the rate of infection initially rises exponentially.
- Suppose that we are interested in understanding the spread of disease, and we want to build an ABM of such a spread. How should we go about doing it?
- First, we need some agents that keep track of whether they are infected with a cold or not
- We need the ability to initialize the model by infecting a group of individuals
- Individuals move around randomly on a landscape and infect other individuals whenever they come into contact with them.





- We conclude that as the population density increases, the time to full infection dramatically decreases.

- In the beginning, when the first person becomes infected, if there are not many other people around, the person has no one to infect, and thus the infection rate increases slowly.
- However, if there are many people around then there will be plenty of infection opportunities
- Moreover, at the end of the run, when there are only one or two uninfected agents, they will be more likely to run into someone with an infection if the population count is high.
- This is true despite the fact that the total number of people that need to be infected increases.
- To describe these patterns of behavior it makes sense to turn to some statistics
- Credits : Uri Wilensky

140

click here to download .pptx

Here we will learn about the general methodology behind descriptive statistics that is to provide numerical measures that summarize a large data set and describe the data set in such a way that it is not necessary to examine every single value.

- Suppose we are interested in determining whether a coin is fair. (i.e., it is as likely to turn up heads when flipped as it is to turn up tails) then we can conduct a series of experiments where we flip the coin and observe the results.
- It is much easier to look at means and standard deviations than it is to examine large series of data
- (e.g., for HHHHTTHTTT, the observed probability is 0.5, and the expected outcome for ten trials is to observe five heads with a standard deviation of 1.58).
- To apply this technique to our Spread of Disease model in more depth, we can create summary statistics

Summary Statistics

Population	Mean	Std. Dev.
50	366.8	47.39385802
100	213.8	27.40154091
150	144.4	17.65219533
200	118.7	12.12939497

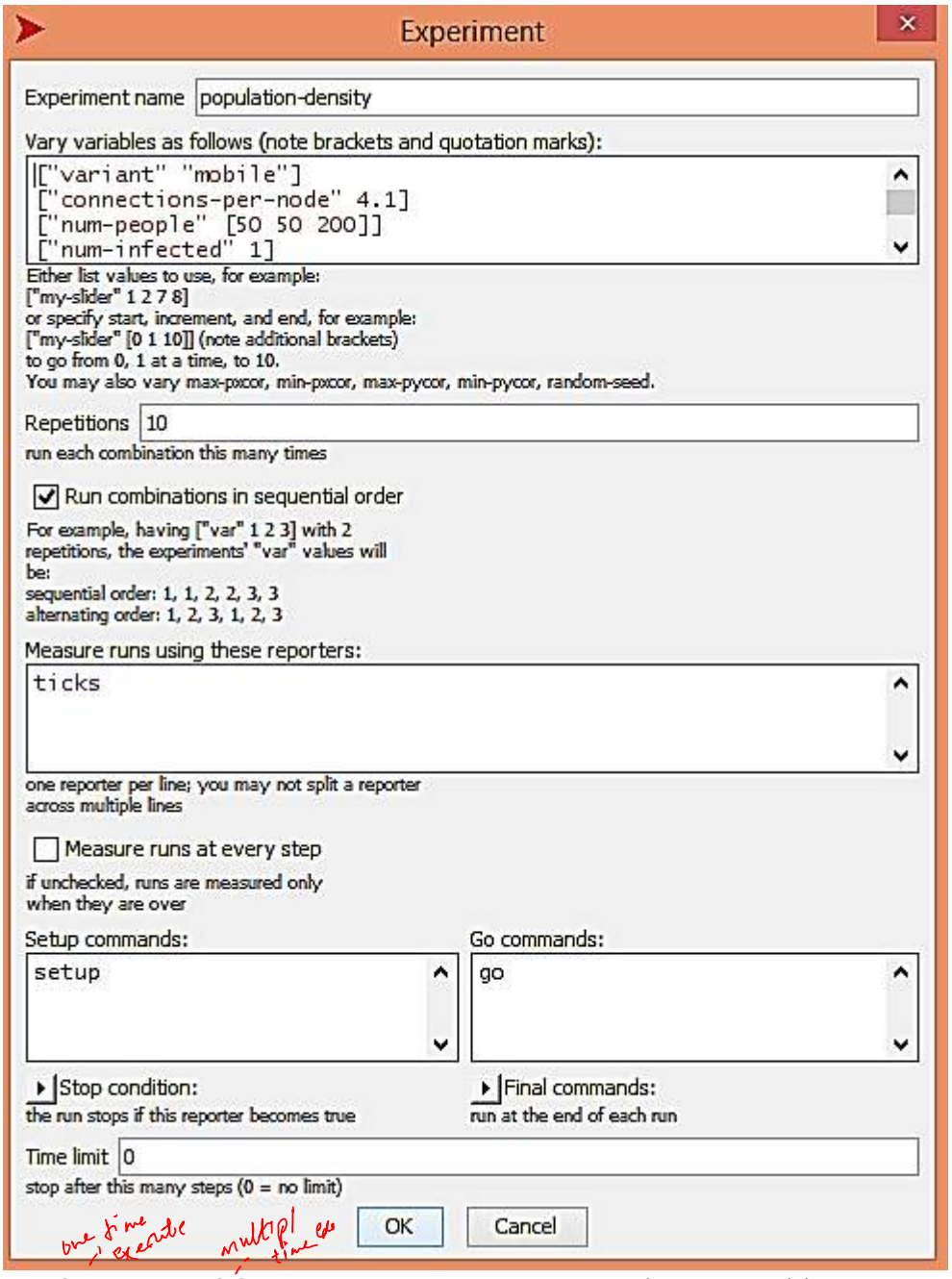
- We see that the mean time to 100 percent infection declines as the population density increases.

- Another interesting result is that as the population density goes up, the standard deviation goes down.
- This means that the data is less varied.
- These results seem to confirm our original hypothesis that as population density increases the mean time to infection declines.
- Within ABM, statistical analysis is a common method of confirming or rejecting hypotheses
- ABMs create large amounts of data (the Spread of Disease model is just a small example), and if we can summarize that data we can examine large amounts of output in an efficient manner.
- Most ABM toolkits give you the basic ability to carry out simple statistical analysis within the package itself
- (e.g., in NetLogo there are MEAN and STANDARD-DEVIATION primitives). Thus, while the model is running, the ABM itself can generate summary statistics.
- The NetLogo R extension can be used to conduct analyses with the R statistical package
- Credits : Uri Wilensky

141

click here to download .pptx

- When you are trying to collect statistical results from an ABM you should run the model multiple times and collect different results at different points.
- Here we will understand how ABM toolkits will provide you with a way to collect the data from these runs automatically.
- In NetLogo there is a tool called BehaviorSpace. ABM toolkits are often full-featured programming languages, allowing you to write your own tools for creating experiments to produce the data sets you want to analyze.
- These tools will automatically run the model multiple times with multiple different settings and collect the results in some easy to use format like the CSV files mentioned earlier.
- Let us call this experiment " population density"



- The SETUP and GO commands allow you to specify any additional NetLogo code that you need to make the model start and go.
- " Stop condition " allows you to specify special stop conditions for each run, and
- " Final commands " allows you to insert any commands that you want executed between runs of the model
- In general in ABM, it is important to carry out multiple runs of your experiments so that you can determine if some result is truly a pattern or just a one-time occurrence.

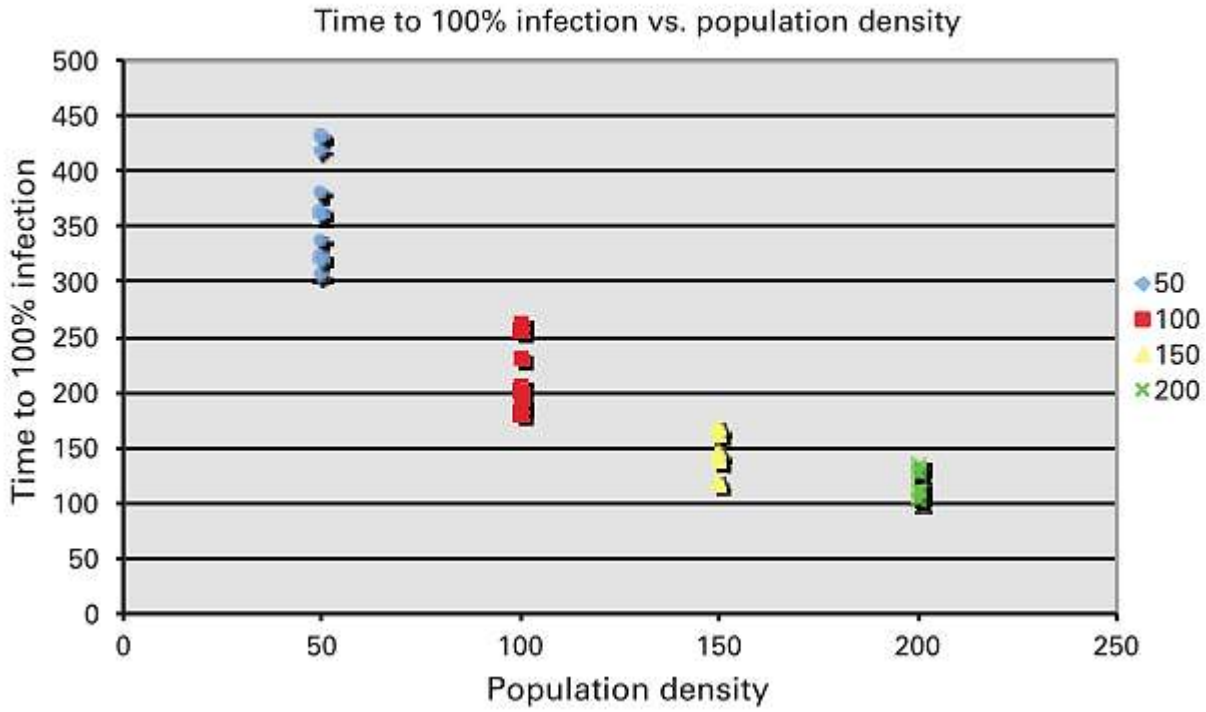
- One common way is to start by manually running your model multiple times, but to get a better sense of the results it is usually much easier to use a batch experiment tool.
- We have illustrated the BehaviorSpace tool, which is the batch experiment tool for NetLogo.
- Credits : Uri Wilensky

142

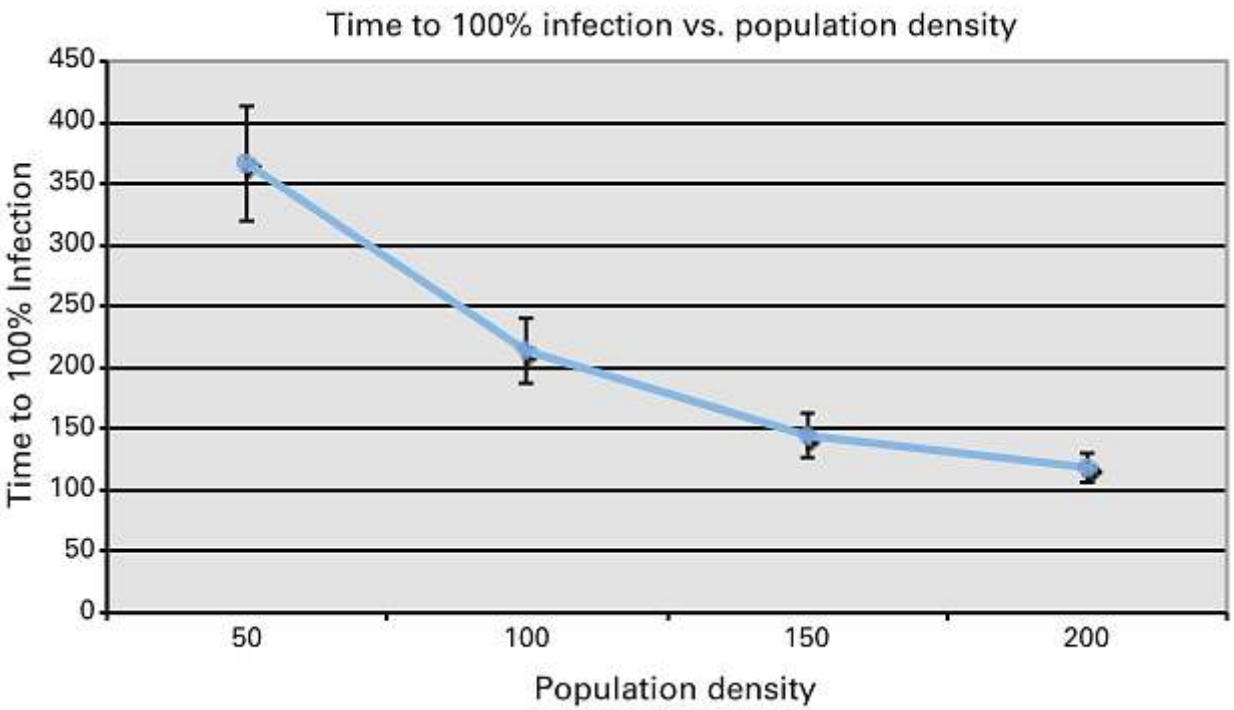
click here to download .pptx

In NetLogo creating simple graphs is easy to do and will often suffice for simple data analysis.

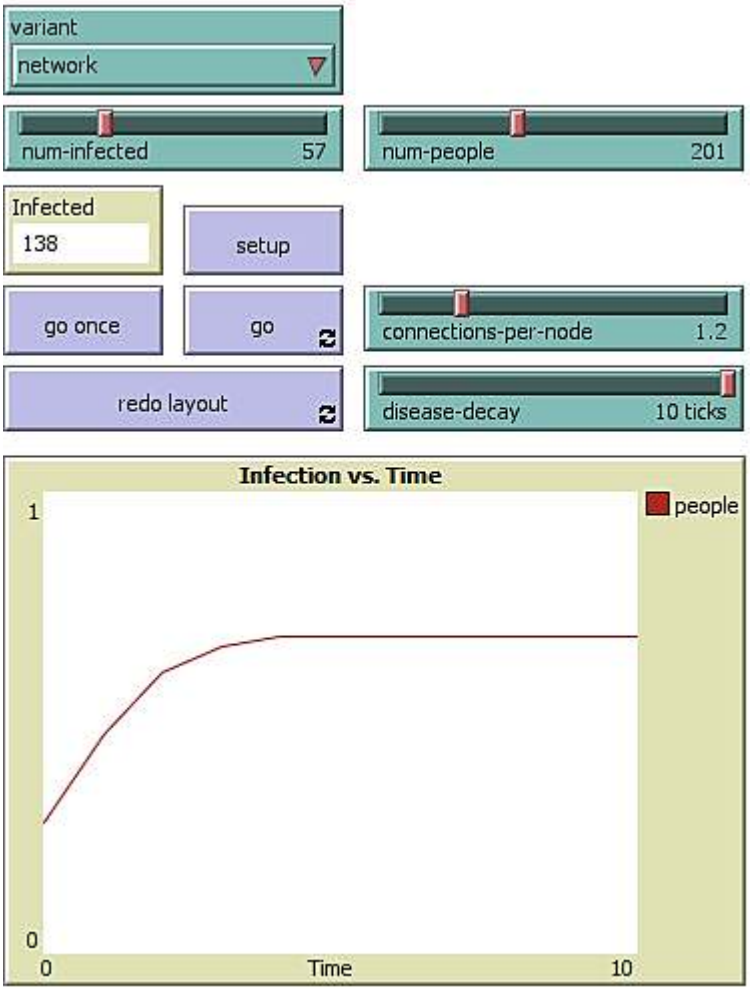
But with complex data sets, designing a useful and immediately informative graph can be challenging and is the subject of an extensive body of literature.

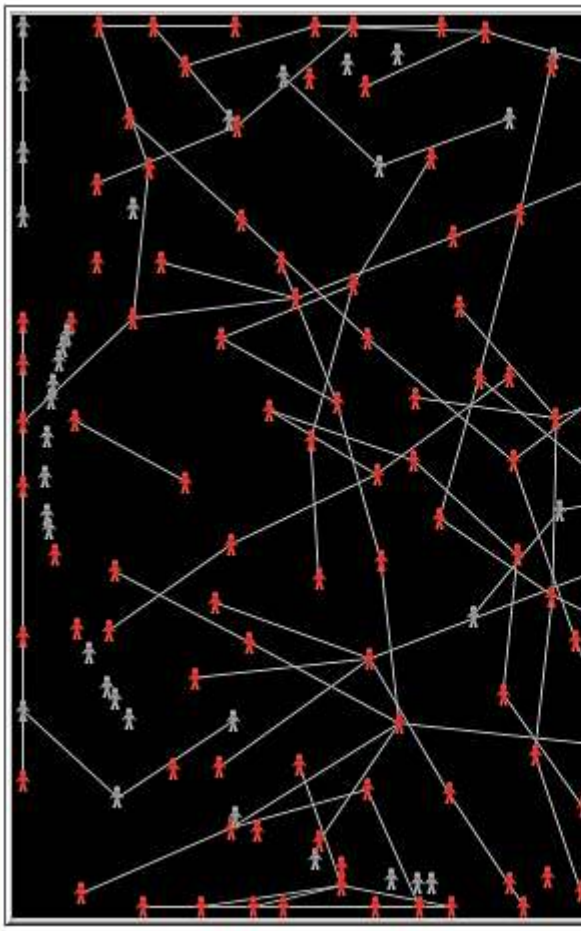


- We can quickly see how the data is distributed and how the data changes with population density



- This new figure might not be easier to understand than previous figure, but if there were one hundred data points in previous figure rather than ten data points, then a figure like this one might be very helpful.
- Many ABM toolkits include capabilities for continually updating graphs and charts during the running of a model and thus enable you to see the progress of the model temporally
- For instance, in the Spread of Disease model there is a graph that illustrates the change in the fraction of infected agents as time proceeds





-
- This is one example of how we can use time series to help understand the behavior of a model.
- Summary: We have seen the use of graphs (plots) in simulation software
- Credits: Uri Wilensky

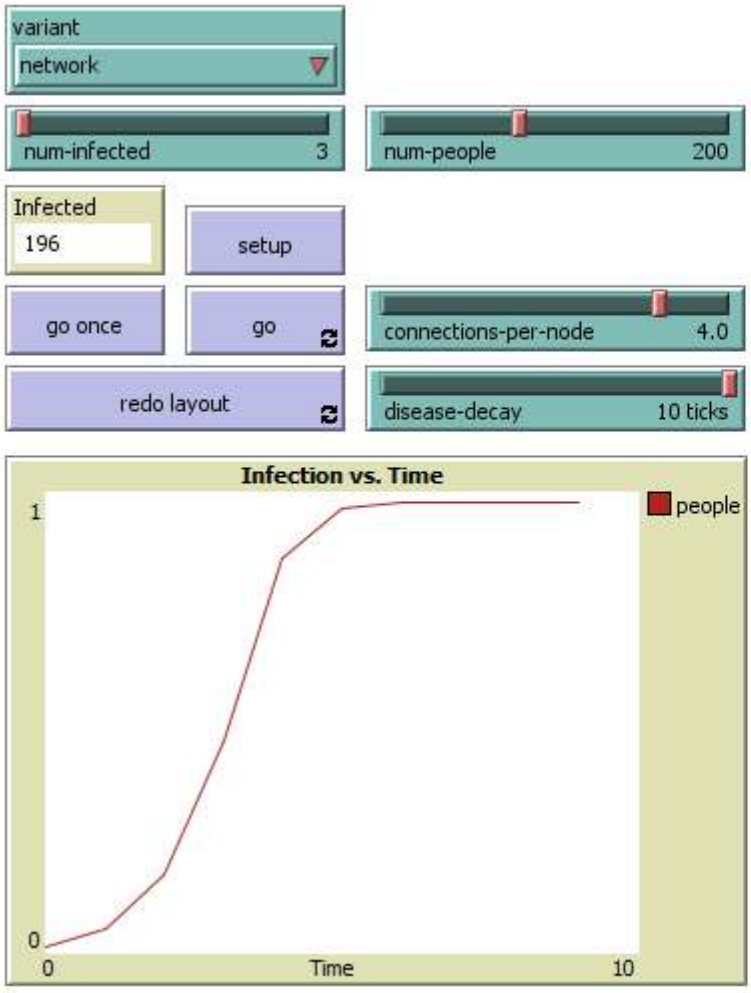
143

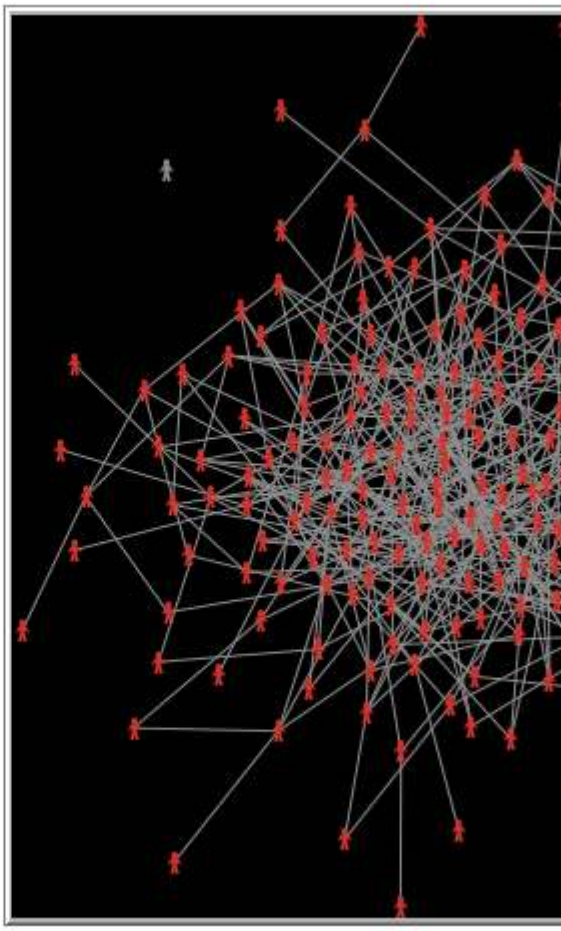
click here to download .pptx

Here, we understand the analysis of "networks within ABM."

- Interactions do not occur in physical space. But rather they can also occur across social networks.
- E.g. some diseases spread only through certain kinds of social networks. If we set the chooser to "network" we can explore this further.
- The property which determines how many connections.

- Each individual has to other individual in the network.
- Known as “connections-per-node”.





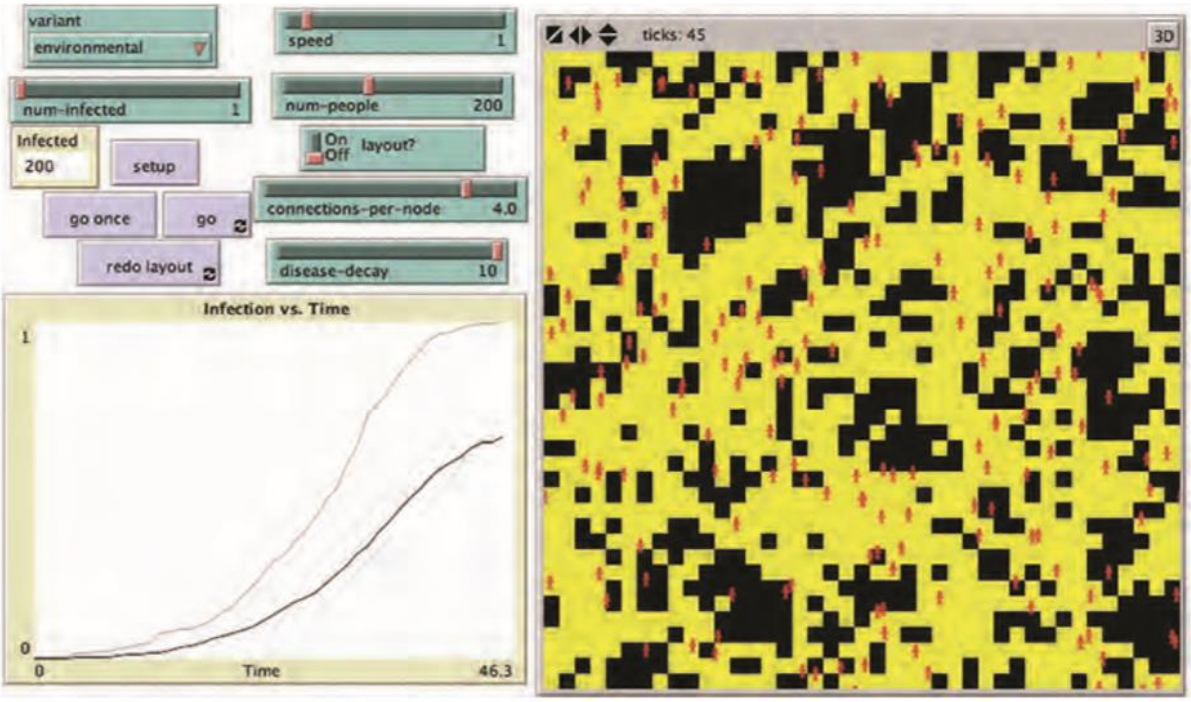
-
- A well known property of random graph is the average number of individuals infected.
- Grows substantially
- The connections-per-node exceeds 1.0
- Forms a giant component in the network.
- The property of average path length, which measure the distance between any two nodes in the network.
- Affect the spread of disease in the network.
- Another property is clustering co-efficient.
- Different properties and tools are there to measure and analyze a wide variety of metrices
- Associated with social network analysis (SNA).
- To summarize, each of these network properties can be analyzed as to their effect on the spread of disease.
- Reports and toolkits like UCInet can be used for further examination.

- Credits: Uri Wilensky book.

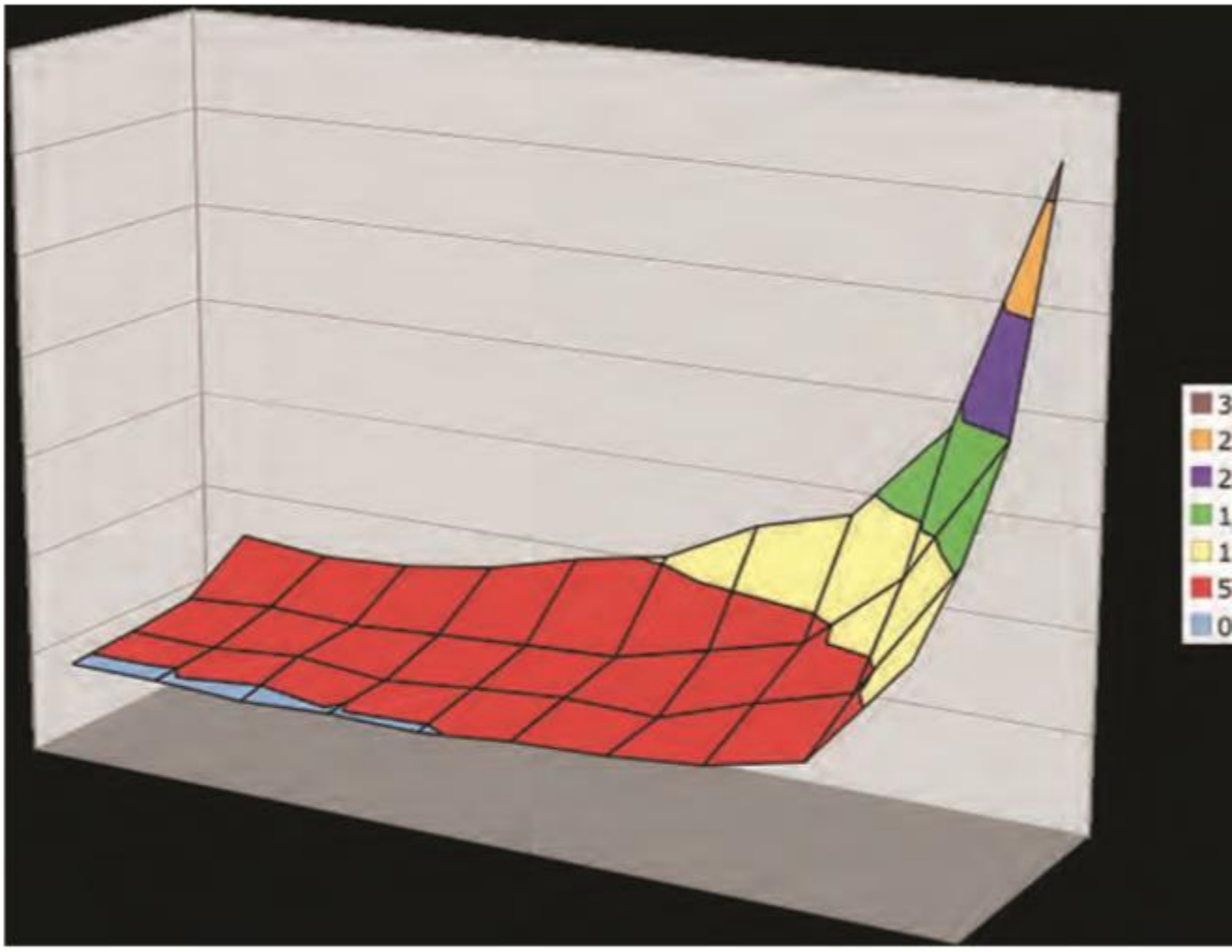
144

In this section we are going to see agent-to-environment and environment-to-agent transmission.

- The key areas to understand are:
  - Spread of disease model for examining.
  - Environmental interaction effect.



- The path below any agent will become yellow.
- Change the rate if DISEASE-DECAY 0 to 10 at a single time intervals
- A long DIESEASE-DECAY might have negligible over having a small DISEASE-DECAY.
- Investigating using behaviour space, we notice that a powerful aspect of ABMs is that it also shows us the pattern of infection.



-
- Leaving long stringy patterns of environmental infection.
- To summarize, we have seen the working of environmental data and ABMs.
- Credits: Uri Wilensky book

145

click here to download .pptx

- Here, we understand the about the “correctness of a model”. Output relating the concerned issue must be accurate.
- Model accuracy is evaluated through validation, verification and replication.
- Implemented model corresponds to, and explains, some phenomenon in the real world.
- Model verification: determining whether an implemented model corresponds to the target conceptual model.

- Make sure that the model has been implemented correctly.
- Model replication is the implementation by one researcher or group of researchers of a conceptual model previously implemented by someone else.
- Set of results from a model that corresponds to the real world is not sufficient.
- Multiple runs are often needed to confirm that a model is accurate.
- Verification, validation, and replication collectively underpin the correctness, and thus utility, of a model.
- Credits: Uri Wilensky book.

146

click here to download .pptx

- Here, we understand the role of “Verification” in Correctness of a Model.
- In larger models, the code can be difficult to understand as it evolves over time.
- Verification ensures the elimination of “bugs” from the code.
- The process of debugging becomes more difficult for complex models.
- Our goal, however, is to keep the process easy and simple.
- Build the model simply to begin with – it will be easier to verify. Expand the complexity of the model as necessary. This incremental approach makes it easy to verify the additional components.
- Even if all the components are verified, it is still possible that the system is not. Complications may arise from the interaction between model components.
- In this section we have examined the issue of verification of a model in the context of a simple ABM.
- Credits: Uri Wilensky book

147

click here to download .pptx

Here, we understand need of “communication” for the correctness of a model.

- Sometimes a team of people builds a model. Other team members actually implement the model. Therefore, verification becomes critical.
- Communication is critical to ensure that the implemented model is correct.
- It is essential.
- E.g. In the voting model.

- Political scientists differentiate between Moore and Van Neumann neighborhoods.
- Small world network and hexagonal versus a rectangular grid.
- In an ideal situation, the model author and the implementer is the same person which averts the sort of communication errors.
- But when it is not, then there is often room for human error and misunderstanding.
- In the past it was difficult to be expert model implementer and model author.
- However, low-threshold ABM languages, such as NetLogo is narrowing the gap between author and implementer.
- We have seen how communication plays an important role in the correctness of a model.
- How communication can fill the gap between the implementer and the author of the model.
- Credits: Uri Wilensky book.

148

click here to download ppt

Here, we will understand how "Describing Conceptual Models" plays a role in analyzing correctness of a model.

- Implementing the voting model, we may realize subsequently that we never understand the idea completely. This could happen if we talked with a political scientist.
- Describing how we plan to implement the model is a specific type of document.
- We and the political scientist should have the same "conceptual model" in our mind.
- Describe the model in more formal terms. This includes a pictorial description of the model using flowcharts.
- Subsequently, we can convert the flowcharts into pseudo-code
- The goal of pseudo-code is to serve as a midway point between natural language and programming language.
- Pseudocode:

```
Voters have votes = {0, 1}

For each voter:
    Set vote either 0 or 1, chosen with equal probability
Loop until election
    For each voter
       If majority of neighbors' votes = 1 and vote = 0 then set vote 1
       Else If majority of neighbors' votes = 0 and vote = 1 then set vote 0
       If vote = 1: set color blue
       Else: color = green
    Display count of voters with vote = 1
    Display count of voters with vote = 0
End loop
```

- Other methods are UML, choosing a language similar to pseudo-code and NetLogo.
- We have learnt about processes involved in describing conceptual model which include flowcharts, pseudo-code, UML, and NetLogo.
- Credits: Uri Wilensky book

149

click here to download here

Here, We will discuss and understand how we can use "Verification Testing" for finding out about the correctness of a model.

- When implementing the design into code we need to follow ABM core design principles.

```
patches-own
[
    vote ;; my vote (0 or 1)
    total ;; sum of votes around me
]

to setup
    clear-all
    ask patches [
           if (random 2 = 0) ;; half a chance of this
           [ set vote 1 ]
       ]
    ask patches [
       if (random 2 = 0) ;; half a chance of this
           [ set vote 0 ]
    ]
    ask patches [
       recolor-patch
    ]
end

to recolor-patch ;; patch procedure
    ifelse vote = 0
       [ set pcolor green ]
       [ set pcolor blue ]
end
```

-

```
to check-setup
    let diff abs ( count patches with [ vote = 0 ] - count patches
with [ vote = 1 ] )
    if diff > .1 * count patches [
       print "Warning: Difference in initial voters is greater than 10^4%
    ]
end
```

-

```
to setup
    clear-all
    ask patches [
           set vote random 2    ;; 0 or 1, with equal probability
    ]
    ask patches [
       recolor-patch
    ]
    check-setup
end
```

-

- This verification technique is a form of unit testing. We can modify the code without disrupting previous code.

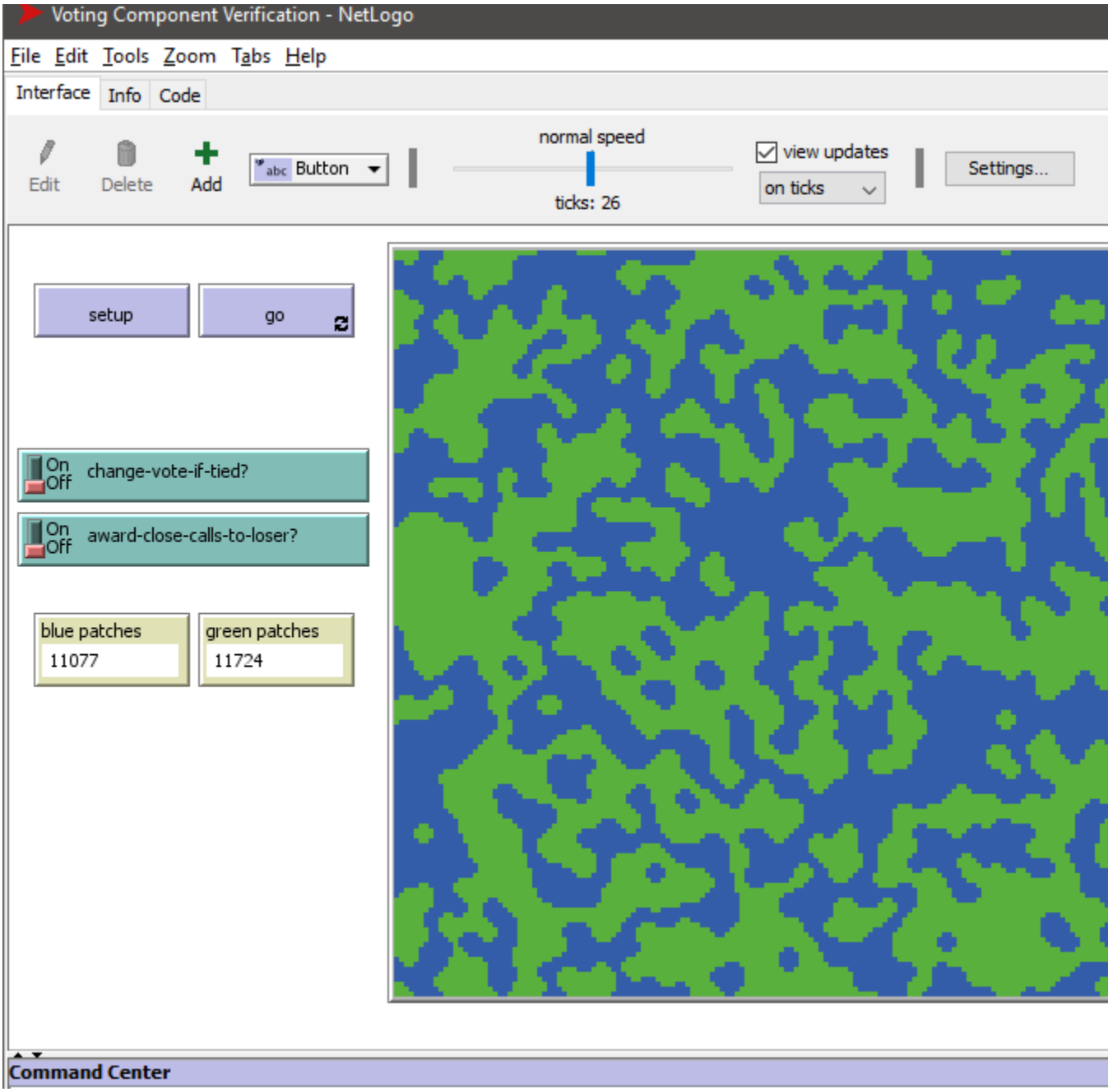
```
to go
    ask patches [
       set total (sum [vote] of neighbors)
    ]
    ;; this is equivalent to count neighbors with [vote = 1]
    ;; use two ask patches blocks so all patches compute "total"
    ;; before any patches change their votes
    ask patches [
       ifelse vote = 0 and total >= 4 [
           set vote 1
       ]
           [if vote = 1 and total <= 4 ] [
               [set vote 0
           ]
           recolor-patch
       ]
       tick
       end
```

- We have understood how an incremental approach makes it easy to implement the model correctly.
- We are then able to write our code into NetLogo for verification purpose.
- Credits: Uri Wilensky book

click here to dowload ppt

Here, we will learn about going beyond verification using agent-based modeling.

- Sometimes results are not produced to what the implementers and authors hypothesized.
- Results of modeling can therefore end up causing further confusions for the scientists.
- Jagged edges can occur due to ties in votes.
- Design the model that neighbors do not change their votes if tied.
- Switch called CHANGE VOTE IF TIED.

- 

```
patches-own [
  vote   ;; my vote (0 or 1)
  total  ;; sum of votes around me
]

to setup
  clear-all
  ask patches [
    set vote random 2 ;; set vote to either 0 or 1
    recolor-patch
  ]
  reset-ticks
  check-setup
end

to go
  ;; keep track of whether any patch has changed their vote
  let any-votes-changed? false
  ask patches [
    set total (sum [ vote ] of neighbors)
  ]
  ;; use two ask patches blocks so all patches compute "total"
  ;; before any patches change their votes
  ask patches [
    let previous-vote vote
    if total < 3 [ set vote 0 ] ;; if majority of your neighbors vote 0, set your vote to 0
    if total = 3 [
      ifelse award-close-calls-to-loser?
```

```
        [ set vote 1 ]
        [ set vote 0 ]
    ]
    if total = 4 and change-vote-if-tied? [
      set vote (1 - vote) ;; invert the vote
    ]
    if total = 5 [
      ifelse award-close-calls-to-loser?
        [ set vote 0 ]
        [ set vote 1 ]
    ]
    if total > 5 [ set vote 1 ] ;; if majority of your neighbors vote 1, set your vote to 1
    if vote != previous-vote [ set any-votes-changed? true ]
    recolor-patch
  ]
  ;; if the votes have stabilized, we stop the simulation
  if not any-votes-changed? [ stop ]
  tick
end

to recolor-patch  ;; patch procedure
  ifelse vote = 0
    [ set pcolor green ]
    [ set pcolor blue ]
end

;; This procedure checks to see if the SETUP procedure sets up the model with roughly
;; equal numbers of blue and green patches
to check-setup
  ;; count the difference between the number of green and the number of blue patches
  let diff abs (count patches with [ vote = 0 ] - count patches with [ vote = 1 ])
  if diff > .1 * count patches [
    print "Warning: Difference in initial voters is greater than 10%."
  ]
end


; Copyright 2008 Uri Wilensky.
```

- Switch AWARD CLOSE CALLS TO LOSER?
- With both switches on, we have a different outcome.

```
to go
  ask patches
    [ set total (sum [vote] of neighbors) ]
  ;; use two ask patches blocks so all patches compute "total"
  ;; before any patches change their votes
  ask patches
    [ if total > 5 [ set vote 1 ]
      if total < 3 [ set vote 0 ]
      if total = 4
        [ if change-vote-if-tied?
          [ set vote (1 - vote) ] ] ;; switch vote
      if total = 5
        [ ifelse award-close-calls-to-loser?
          [ set vote 0 ]
          [ set vote 1 ] ]
      if total = 3
        [ ifelse award-close-calls-to-loser?
          [ set vote 1 ]
          [ set vote 0 ] ]
      recolor-patch ]
  tick
end
```

-
- We have seen the details of how to go beyond verification.
- Credits: Uri Wilensky book

151

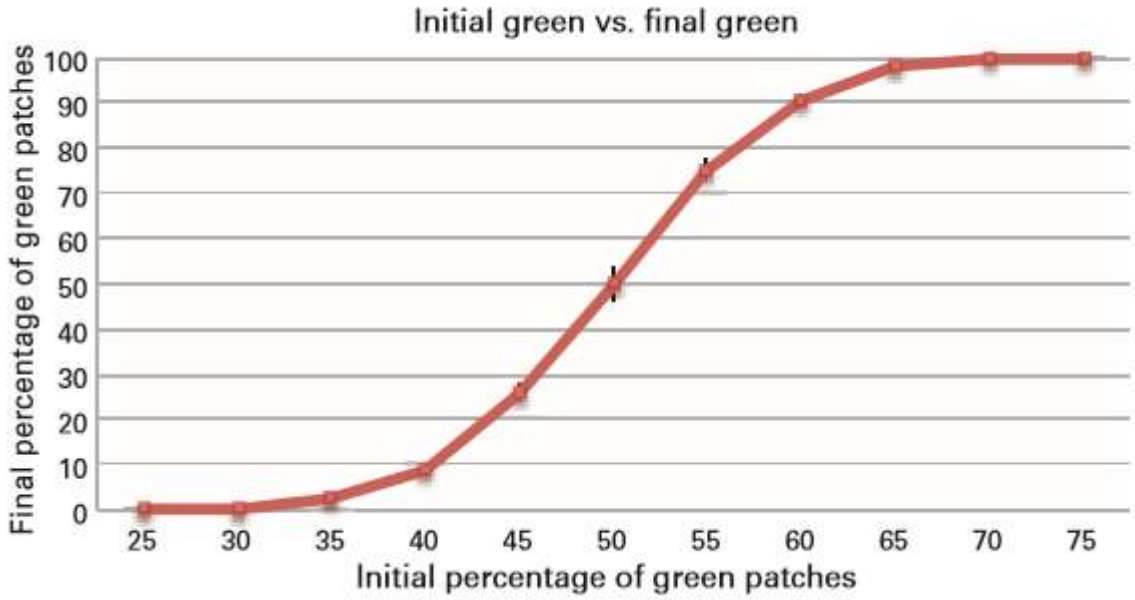
click here to download ppt

Here, we will discuss sensitivity analysis and robustness.

- Creating the parameter to test the hypothesis of initial balance in one direction.
- Using BehaviorSpace, we can run the experiment varying from 25 to 75 percent increment.
- Two conditions:
- If no voter switch vote in the last step the model will stop.
- The model will stop after one hundred times the steps have executed.

```
patches-own
[
  vote   ;; my vote (0 or 1)
  total  ;; sum of votes around me
]

to setup
  clear-all
  ask patches [
    ifelse random 100 < initial-green-pct
      [ set vote 0 ]
      [ set vote 1 ]
    recolor-patch
  ]
  reset-ticks
  check-setup
end

to go
  ;; keep track of whether any patch has changed their vote
  let any-votes-changed? false
  ask patches [
    set total (sum [ vote ] of neighbors)
  ]
  ;; use two ask patches blocks so all patches compute "total"
  ;; before any patches change their votes
]
;; use two ask patches blocks so all patches compute "total"
;; before any patches change their votes
ask patches [
  let previous-vote vote
  if total < 3 [ set vote 0 ] ;; if majority of your neighbors vote 0, set your vote to 0
  if total = 3 [
    ifelse award-close-calls-to-loser?
      [ set vote 1 ]
      [ set vote 0 ]
  ]
  if total = 4 and change-vote-if-tied? [
    set vote (1 - vote) ;; invert the vote
  ]
  if total = 5 [
    ifelse award-close-calls-to-loser?
      [ set vote 0 ]
      [ set vote 1 ]
  ]
  if total > 5 [ set vote 1 ] ;; if majority of your neighbors vote 1, set your vote to 1
  if vote != previous-vote [ set any-votes-changed? true ]
  recolor-patch
]
```

```
  ;; if the votes have stabilized, we stop the simulation
  if not any-votes-changed? [ stop ]
  tick
end

to recolor-patch  ;; patch procedure
  ifelse vote = 0
    [ set pcolor green ]
    [ set pcolor blue ]
end

;; This procedure checks to see if the SETUP procedure sets up the model with
;; roughly expected numbers, given the value of the initial-green-pct slider
to check-setup
  let expected-green (count patches * initial-green-pct / 100)
  let diff-green (count patches with [ vote = 0 ]) - expected-green
  if diff-green > (.1 * expected-green) [
    print "Initial number of green voters is more than expected."
  ]
  if diff-green < (- .1 * expected-green) [
    print "Initial number of green voters is less than expected."
  ]
end


; Copyright 2008 Uri Wilensky.
; See Info tab for full copyright and license.
```



- Past research methodologies for sensitivity analysis include Active Nonlinear Testing.
- We have seen the details of Sensitivity Analysis and Robustness
- Credits : Uri Wilensky book

152

click here to download ppt

Here, we are going to examines benefits and issues related to verification.

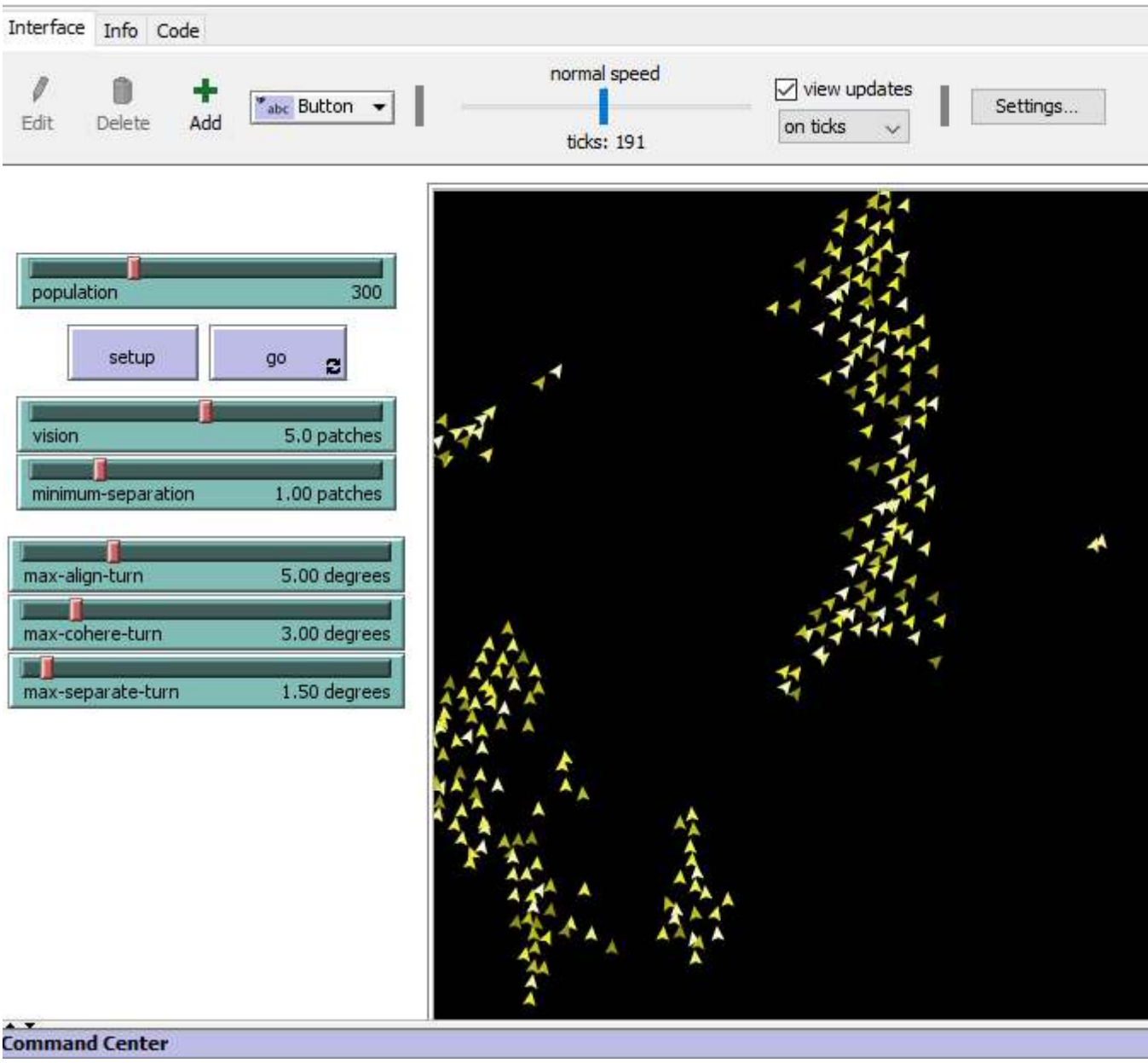
- We need to develop an understanding the cause of unexpected outcomes and the impact of small changes.
- Implemented model needs to correspond well with the conceptual model.
- This all depends on the model's low level rules as well as an understanding of the mechanisms involved. A bug in the code can produce surprising results.
- Understanding the operation of the model is therefore quite essential. It is also important to note that the verification process is not binary.
- We have discussed verification benefits and issues.
- Credits: Uri Wilensky book

153

click here to download ppt

We will discuss the validation of agent based modeling.

- Validation involves corresponding between implemented model and reality.
- The various topics related to validation include:
- Two axis.
- Macrovalidation.
- Face validation.
- Flocking model.
- A classical agent based model.



```
turtles-own [
  flockmates         ;; agentset of nearby turtles
  nearest-neighbor   ;; closest one of our flockmates
]

to setup
  clear-all
  create-turtles population
    [ set color yellow - 2 + random 7  ;; random shades look nice
      set size 1.5  ;; easier to see
      setxy random-xcor random-ycor
      set flockmates no-turtles ]
  reset-ticks
end

to go
  ask turtles [ flock ]
  ;; the following line is used to make the turtles
  ;; animate more smoothly.
  repeat 5 [ ask turtles [ fd 0.2 ] display ]
  ;; for greater efficiency, at the expense of smooth
  ;; animation, substitute the following line instead:
  ;;   ask turtles [ fd 1 ]
  tick
end
```

```
to flock  ;; turtle procedure
  find-flockmates
  if any? flockmates
    [ find-nearest-neighbor
      ifelse distance nearest-neighbor < minimum-separation
        [ separate ]
        [ align
          cohere ] ]
end

to find-flockmates  ;; turtle procedure
  set flockmates other turtles in-radius vision
end

to find-nearest-neighbor ;; turtle procedure
  set nearest-neighbor min-one-of flockmates [distance myself]
end

;;; SEPARATE

to separate  ;; turtle procedure
  turn-away ([heading] of nearest-neighbor) max-separate-turn
end

;;; ALIGN

to align  ;; turtle procedure
  turn-towards average-flockmate-heading max-align-turn
end
```

```
;;; ALIGN

to align  ;; turtle procedure
  turn-towards average-flockmate-heading max-align-turn
end

to-report average-flockmate-heading  ;; turtle procedure
  ;; We can't just average the heading variables here.
  ;; For example, the average of 1 and 359 should be 0,
  ;; not 180.  So we have to use trigonometry.
  let x-component sum [dx] of flockmates
  let y-component sum [dy] of flockmates
  ifelse x-component = 0 and y-component = 0
    [ report heading ]
    [ report atan x-component y-component ]
end

;;; COHERE

to cohere  ;; turtle procedure
  turn-towards average-heading-towards-flockmates max-cohere-turn
end

to-report average-heading-towards-flockmates  ;; turtle procedure
  ;; "towards myself" gives us the heading from the other turtle
  ;; to me, but we want the heading from me to the other turtle,
  ;; so we add 180
  let x-component mean [sin (towards myself + 180)] of flockmates
  let y-component mean [cos (towards myself + 180)] of flockmates
  ifelse x-component = 0 and y-component = 0
    [ report heading ]
    [ report atan x-component y-component ]
end
```

```
;;; HELPER PROCEDURES

to turn-towards [new-heading max-turn]  ;; turtle procedure
  turn-at-most (subtract-headings new-heading heading) max-turn
end

to turn-away [new-heading max-turn]  ;; turtle procedure
  turn-at-most (subtract-headings heading new-heading) max-turn
end

;; turn right by "turn" degrees (or left if "turn" is negative),
;; but never turn more than "max-turn" degrees
to turn-at-most [turn max-turn]  ;; turtle procedure
  ifelse abs turn > max-turn
    [ ifelse turn > 0
        [ rt max-turn ]
        [ lt max-turn ] ]
    [ rt turn ]
end


; Copyright 1998 Uri Wilensky.
; See Info tab for full copyright and license.
```

- Here, we have discussed the validation of agent based models.
- Credits: Uri Wilensky book.

click here to download ppt

Here, we will understand the difference between “macrovalidation and microvalidation“ and examine their role in the validation of a model.

- ABMs are built from agents hence we can directly compare them with those that exist in the real world.
- For instance, we can ask whether the agents have properties similar to real birds in flocking model.
- There are some limitation e.g. real birds can fly in three dimensions but our bird agents move in two dimension only. However, these limitations does not make our model invalid.
- We can also build the flocking model in three dimensions and examine the resultant flocks to see if they are relevantly different from the 2D flocks.
- The other major avenue of validation is to investigate the relationship between the global properties of the model and the flocking patterns of real birds, a process called macrovalidation.

- By showing that our model corresponds to the macro-level phenomenon, we further validate that our model is descriptive of real world systems.
- Macrovalidation tells you if you have captured the important parts of the system, whereas microvalidation tells you if you 've captured the important parts of the agent's individual behavior.

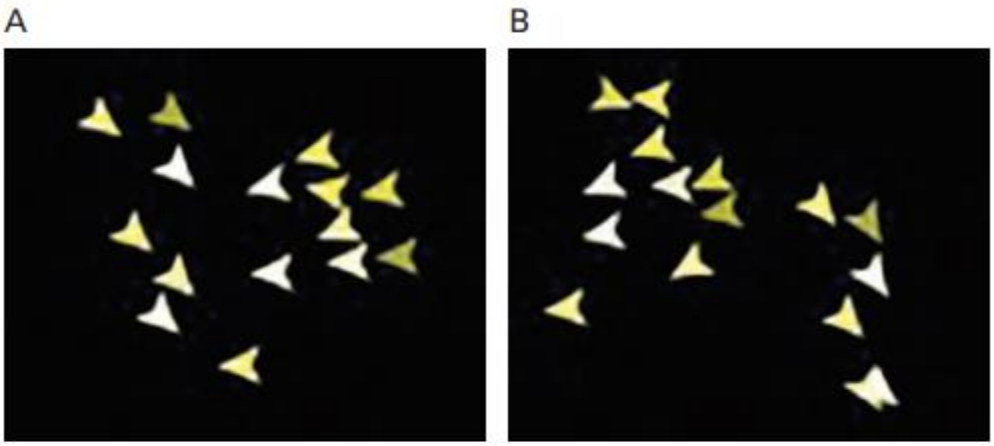


Figure 7.9
Two mini-flocks (A) before and (B) after a collision.

- We now understand how macrovalidation and microvalidation compares to each other.
- Credits: Uri Wilensky book.

click here to download ppt

Here we will discuss how "face validation" is different from "empirical validation".

- Face validation is the process of showing that the mechanisms and properties of the model look like mechanisms and properties of the real world.
- Empirical validation is making sure that the model generates data. It corresponds to similar patterns of data in the real world.
- Face validity can exist at both micro and macro levels.
- Determining whether the birds in the model correspond to real birds is face microvalidity.
- While determining if the flocks correspond to the appearance of real flocks is face macrovalidity.
- Empirical validation sets a higher bar. Data produced by the model must correspond to empirical data derived from the real-world.

- Measures and numerical data generated both by the model and the actual phenomenon.
- Inputs and outputs in " the real world " are often poorly defined. It is hard to isolate and measure parameters from a real-world phenomenon.
- The process of finding the parameters and initial conditions that cause the model to match with real-world data is called calibration.
- We are now able to understand how these four types of validation (micro-face, macro-face, micro-empirical, and macro-empirical) characterize the majority of validation efforts carried out.
- Credits: Uri Wilensky book

156

click here to download ppt

Here we will understand why it is important to validate a model.

- A valid model can be useful for extracting general principles about the world.
- Changing the mechanisms and parameters can often help predict what might occur in the real world.
- A model is not either valid or invalid:
- A model can be said to be more or less valid based upon how closely it has been compared to the real process it is modeling.
- A model is never inherently valid.
- Its validity comes from the context of what it is being used for.
- Something in the model corresponds to something in reality.
- The user compare his observation's with real world and use it as the basis of his validation.
- Here we have understood the benefits of validation and question it arises.
- Credits: Uri Wilensky book.

157

Here, we will understand about the "Replication".

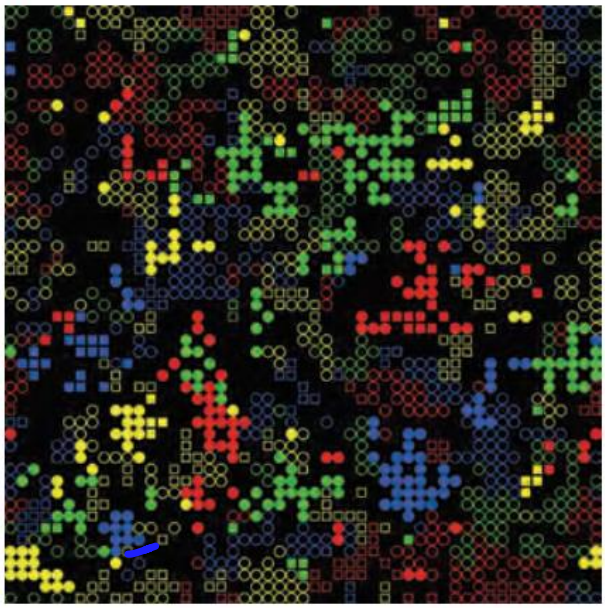
Replication

- Model replication is the implementation by one researcher or group of researchers of a conceptual model previously implemented by someone else.
- Scientists must publish the details of how the experiment was conducted.
- Then subsequent teams of scientists carry out the experiment themselves to ascertain.

- Replicating a physical experiment strengthens original results.
- Comparing both the experimental setup and ensuing results.
- Replicating a computational model serves this same purpose.
- Replicating a computational model increases our confidence.
- New implementation of the model has yielded the same results as the original.

A

B



- This model includes a mechanism for inheritance of strategies.
- The model suggests that " ethnocentric " behavior can evolve under a wide variety of conditions.
- Even when there are no native " ethnocentrics ".
- Here we understand about the basic concepts of replication and how it is useful for the correctness of a model.
- Credits: Uri Wilensky book

1588

click here to download ppt

Here, we will understand about the "Replication of Computational Models: Dimensions and Standards".

Replication

Replication refers to the creation of a new implementation of a conceptual model based on the previous implementation.

Dimensions

- An original model and replicated model can differ across at least six dimensions.
- Time
- Hardware
- Languages
- Toolkits
- Algorithms
- Authors

Successful Replication

- A successful replication is one in which the replicators are able to establish that the replicated model creates outputs sufficiently similar to the outputs of the original.

Replication Standard

- The criterion by which the replication 's success is judged is called the replication standard.
- Different replication standards exist for the level of similarity between model outputs.

Categories of Replication Standard

- There are three categories of replication standard.
- Numerical identity
- Distributional equivalence.
- Relational alignment.

Data

- ABMs usually produce large amounts of data.
- much of which is usually irrelevant.
- Data that are central to the conceptual model should be measured and tested during replication.

Summary

- Here, we understand about dimensions of conceptual model like time, hardware, toolkit, language, algorithms and author.
- Then we discussed about replication standard and its categories.
- Credits: Uri Wilensky book

159

click here to download ppt

Overview

- We are going to see the benefits of replication in this section.

Benefits of Replication

- Advances scientific knowledge.
- Model verification.
- Model validation.
- Shared understanding.
- Shared understanding.
- Creation of sets of terms, idioms and best practices.
- Communicate about model.
- Model verification.
- Distinct implementations. producing same results.
- Conceptual model grows by capturing confidence.
- Correction made if there is any difference found in implemented model and replication.
- Model validation.
- Correspondence between the outputs.
- Validation of a model on the basis of output closer to real-world.
- Reevaluating the original mapping.
- Researchers investing and getting interested in it through replication
- Describing the modeling process through developing a language.
- Culture of replication fosters.
- "mean" and "standard deviation" applying them to tests, overtime, replication of ABM experiment to understand "time step" and "shuffled lists".

Summary

- Here, we understood the benefits of replication in modeling.
- Credits: Uri Wilensky book.

160

click here to download ppt

Overview

- Here, we will learn about recommendations for model replicators.

Recommendations for model replicators

- RS is to produce the level of precision to establish hypothesis regularity.
- Examples : “numerical identity”, “distributional equivalence” and “relational alignment”.
- How detailed the description of the conceptual model is in the original paper or to communicate with authors or original model.
- Beneficial to delay the contact with original author until after first attempt to recreate the original model.
- Differences between public conceptual model and an implementation can be interesting and resulting in new discoveries.
- Becoming familiar with the toolkit in which original model was written.
- Better understanding of original model.
- Replicator understands the subtler working.
- Deliberately implementing a strategy.
- Obtain the source code of original model.
- Effective for illuminating discrepancies in the two model implementations.
- Groupthink
- Unconsciously adopting some of the practices of the original model developer.

Table 7.1
Details to be included in published replications. For each category, sample options to choose from are listed.

Categories of Replication Standards:
Numerical Identity, Distributional Equivalence, Relational Alignment
Focal Measures:
Identify particular measures used to meet goal
Level of Communication:
None, Brief email contact, Rich discussion and personal meetings
Familiarity with Language/Toolkit of Original Model:
None, Surface understanding, Have built other models in this language/toolkit
Examination of Source Code:
None, Referred to for particular questions, Studied in-depth
Exposure to Original Implemented Model:
None, Reran original experiments, Ran experiments other than original ones
Exploration of Parameter Space:
Examined results from original paper, Examined other areas of the parameter space

Summary

- We have understood the recommendation for model replicators in modeling.
- Credits: Uri Wilensky book.