# Designing and prototyping a Biologically Inspired Swarm Intelligence Systems

Antonio Luca Alfeo – July 2019

## Biologically Inspired Swarm Intelligence.

- **Swarm intelligence** (SI) is the collective behavior of decentralized, self-organized systems, natural or artificial. The concept is employed in work on artificial intelligence. The expression was introduced by Gerardo Beni and Jing Wang in 1989, in the context of cellular robotic systems.
- SI systems consist typically of a population of simple agents **interacting** locally with one another and with their environment. The inspiration often comes from nature, especially biological systems.
- The **agents** follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, local, and to a certain degree random, interactions between such agents lead to the emergence of "intelligent" global behavior, unknown to the individual agents.
- The application of swarm principles to robots is called “swarm robotics”, while “swarm intelligence” refers to the more general set of algorithms, covering from optimization to forecasting problems.

## Example of Application

https://www.youtube.com/watch?v=UtBa9yVZBJM

# Benefits

- **Robust**: Because collective systems are built upon multitudes in parallel, there is **redundancy**. Individuals don't count. Small failures are lost in the hubbub. Big failures are held in check by becoming merely small failures at the next highest level on a hierarchy.

- **Self-organized**: Plain old linear systems can be dramatically affected by feedback loops. But in swarm systems, feedback can results in increasing order. By incrementally extending new structure beyond the bounds of its initial state, a swarm can build its own scaffolding to build further structure. Spontaneous order helps create more order.

- **Adaptable**: It is possible to build a clockwork system that can adjust to predetermined stimuli. But constructing a system that can adjust to new stimuli, or to change beyond a narrow range, requires a swarm. Finally, adaptation over individuals and time leads results in **evolution**.

## Drawbacks

- **Non-optimal**: Because they are redundant and have no central control, swarm systems are inefficient. Emergent controls such as prices in free-market economy tend to dampen inefficiency, and never eliminate such inefficiency as a linear system can.

- **Non-controllable**: There is no authority in charge. Guiding a swarm system can only be done as a shepherd would drive a herd: by applying force at crucial leverage points, and by subverting the natural tendencies of the system to new ends. An economy can't be controlled from the outside; it can only be slightly tweaked from within.

- **Non-predictable**: A causes B, B causes C. This logic does not apply to swarm systems. Swarm systems are oceans of intersecting logic: A indirectly causes everything else and everything else indirectly causes A. It is called horizontal causality. The credit for the true cause (or more precisely the true proportional mix of causes) will spread horizontally through the web until the trigger of a particular event is essentially unknowable.

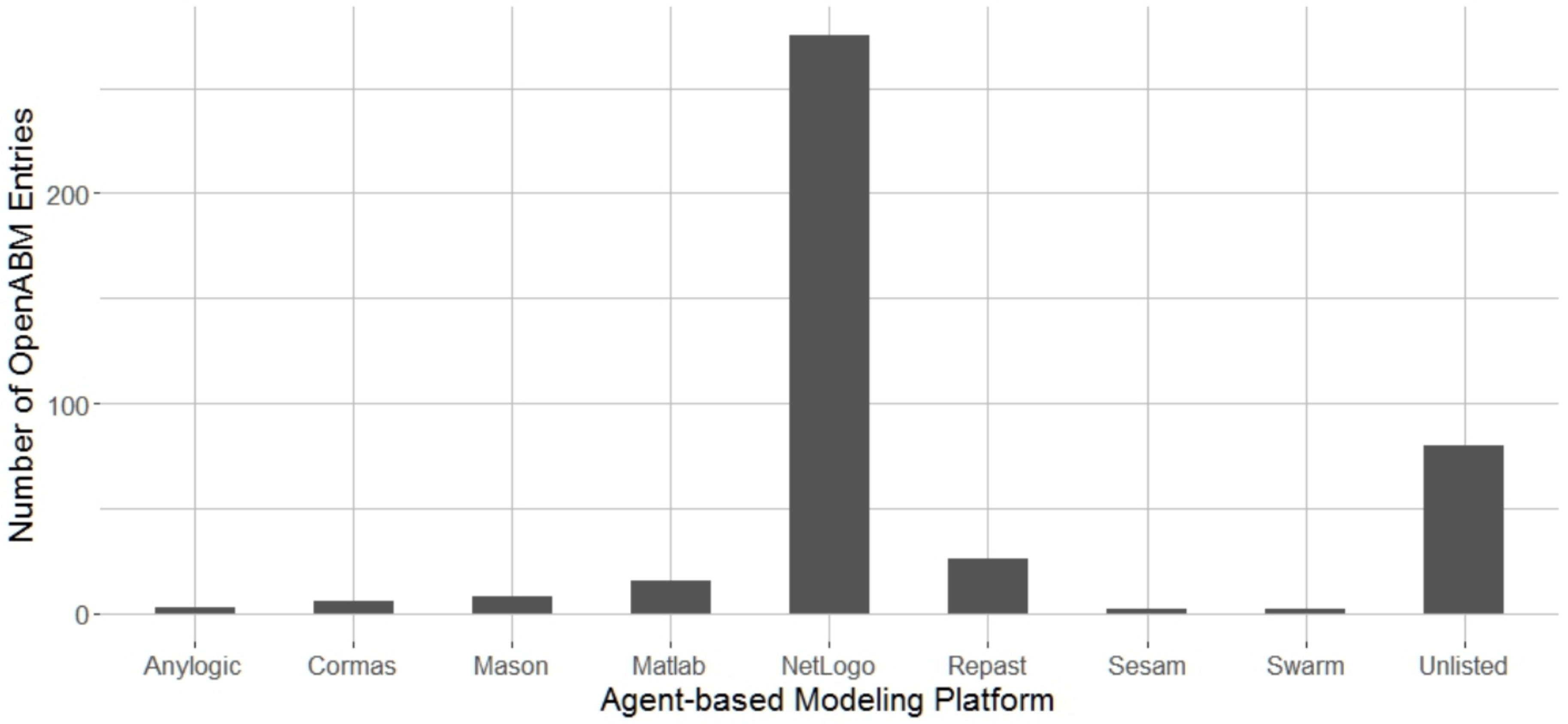
## Simulation-based Design

- The study of real world implementations of swarm systems, are limited by a lack of well-established design methodologies that provide performance guarantees.

- Engineering such systems is a challenging task because of the difficulties to obtain the micro-macro link: a correspondence between the microscopic description of the individual agent behaviour and the macroscopic models that describe the system’s dynamics at the global level.

- As such most of such system are **designed by simulation**.

# Platforms



Platform by Agent-based models on OpenAbm, now CoMSES.net, between 2008 and 2018. NetLogo is clearly dominant [Gunaratne, C., & Garibay, I. (2018). NL4Py: Agent-Based Modeling in Python with Parallelizable NetLogo Workspaces. *CoRR, abs/1808.03292*].

# Netlogo

- **NetLogo** is a programmable modeling environment for simulating natural and social phenomena.
- NetLogo was written in 1999 by Uri Wilensky, based on the programming language Logo (Seymour Papert, MIT), which is a simplified dialect of LISP, and it was designed for modelling complex systems developing over time:
  - You can give instructions to hundreds or thousands of agents all operating independently;
  - Observe the emergence of global behaviors;
  - Connect micro-level behaviors of individuals and macro-level patterns emerging from interactions.
- It runs on the Java Virtual Machine, so it works on all major platforms.

## Netlogo Features

- Extensive documentation and tutorials
  http://ccl.northwestern.edu/netlogo/docs/dictionary.html;
- Models Library, a large collection of pre-written simulations covering topics in:
  - Biology and medicine;
  - Physics and chemistry;
  - Mathematics and computer science;
  - Economics and social psychology.
- Free and open source;
- Fully programmable;
- Double precision floating point math;

# Netlogo Resources

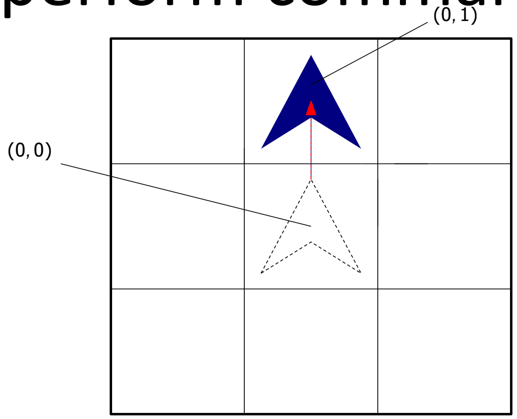
- Download: http://ccl.northwestern.edu/netlogo/download.shtml

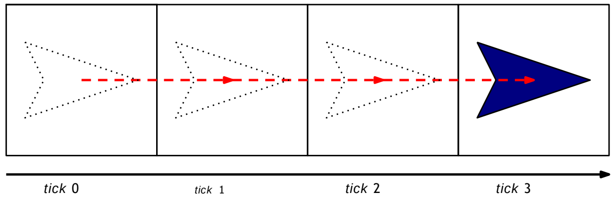
- Web: http://www.netlogoweb.org/launch#http://www.netlogoweb.org/assets/modelslib/Sample%20Models/Social%20Science/Traffic%20Basic.nlogo

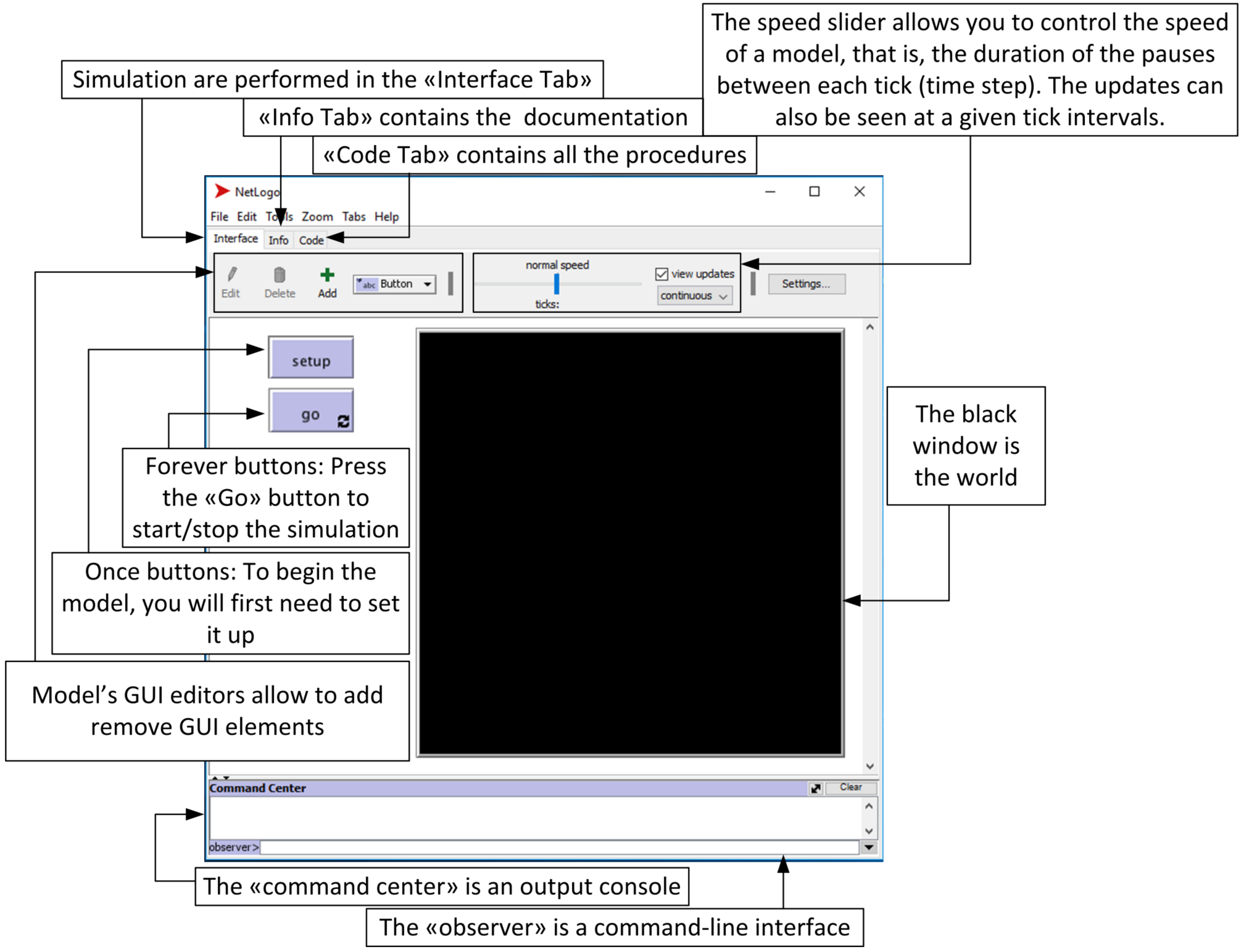
- Documentation: http://ccl.northwestern.edu/netlogo/docs/

# Simulations in Netlogo

- NetLogo simulations are essentially discrete: the world (the space) is discrete, i.e. is a grid whose basic regions are called **patches** and identified by their Cartesian coordinates.
- Time too is discrete, i.e. measured in ticks (retrivable with the primitive `ticks`): each tick the **agents** (**turtles** and **links**), the patches and the world can perform command or procedures.







Simulation are performed in the «Interface Tab»

«Info Tab» contains the  documentation

«Code Tab» contains all the procedures

The speed slider allows you to control the speed of a model, that is, the duration of the pauses between each tick (time step). The updates can also be seen at a given tick intervals.

The black window is the world

Forever buttons: Press the «Go» button to start/stop the simulation

Once buttons: To begin the model, you will first need to set it up

Model's GUI editors allow to add remove GUI elements

The «command center» is an output console

The «observer» is a command-line interface

# First Netlogo Model Run

File > Model Library > Biology > Rabbits Grass Weeds > Wolf sheep Predation

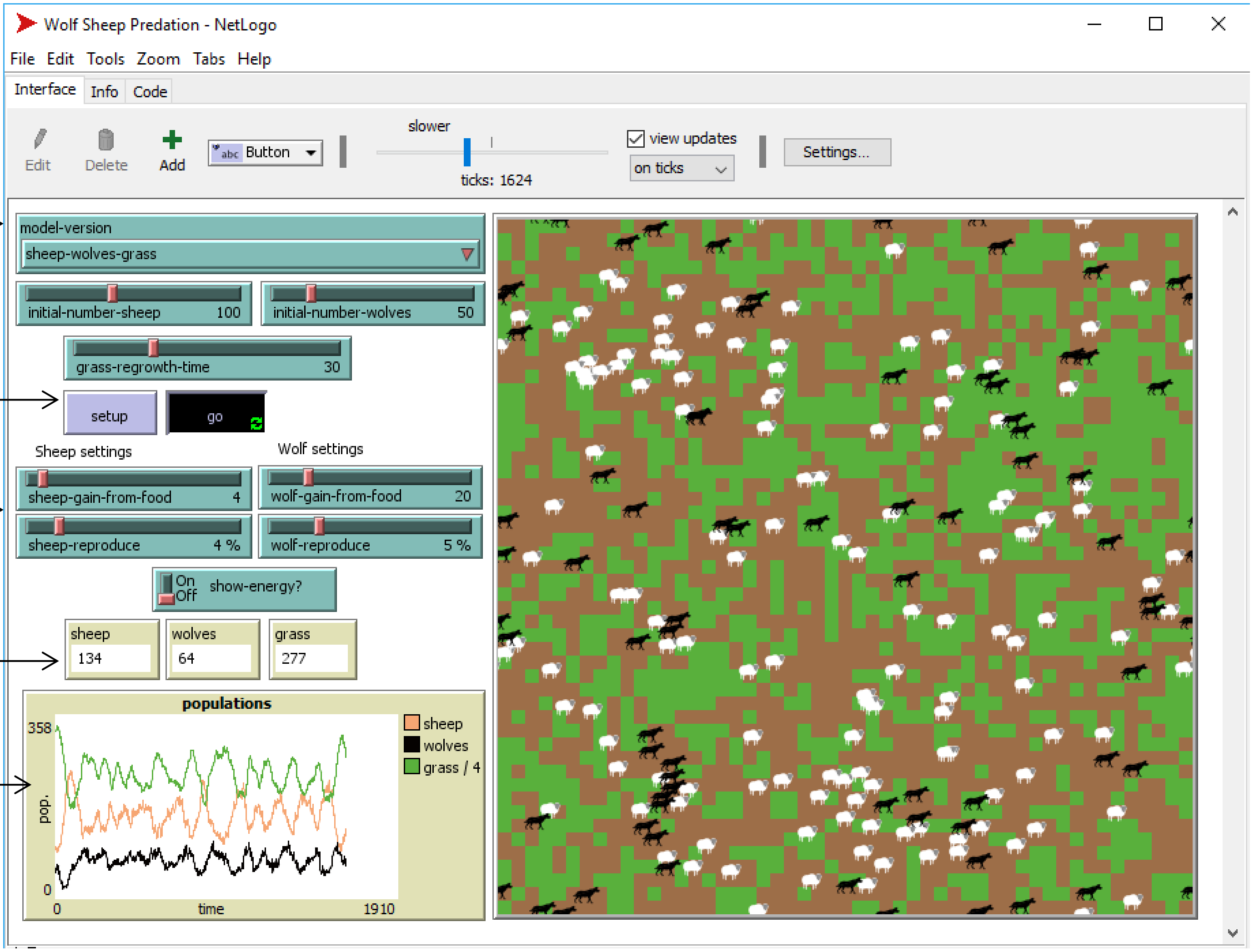
Version

Buttons

Other Parameters

Monitors

Plot



model-version: sheep-wolves-grass

initial-number-sheep 100

initial-number-wolves 50

grass-regrowth-time 30

sheep-gain-from-food 4

wolf-gain-from-food 20

sheep-reproduce 4 %

wolf-reproduce 5 %

On Off show-energy?

| sheep | wolves | grass |
|---|---|---|
| 134 | 64 | 277 |

Plot legend: sheep, wolves, grass / 4; y axis 0 to 358; x axis 0 to 1910

# Studying The Model’s Behavior

How does the model work?

What would happen to the sheep population if there was more initial sheep and less initial wolves at the beginning of the simulation?

- Switch to the "`sheep-wolves-grass`" model.
- Set the "`initial-number-sheep`" slider to 150.
- Set the "`initial-number-wolves`" slider to 25.
- Press "`setup`" and then "`go`".
- Can you kill all the wolves?

## Playing God (from the scratch)

## Setting up a model

- When you open NetLogo the world is empty (no turtle): you have to create turtles, determine their behavior and the characterize the environment they live in;
- There is a special agent, called the **Observer** that take care of  this kind of actions.
- The observer can give **commands**.
- The turtles work with **procedures** and **variables**.
- Those can be encoded in the code tab (remember NetLogo is case insensitive) and linked to GUI elements.

## Day 1: God created the day and the night

# Basic Program Structure

- Every NetLogo program should contain at least the following parts:
  - **Setup part**: this part consists of a single procedure `setup` that  initializes global variables, create agents and does other setup  operations;
  - **Go part**: this part consists of the main procedure `go` that  implements one cycle of the simulation.

Those should be linked to the buttons in the GUI in order to setup each new model run and initiate each running.

# Clearing world

- Procedures such as `clear-all`, `clear-turtles`, `clear-patches`, `clear-output` are typically called at the beginning of a program.
- `Clear-all` resets all global variables to zero; also clears out turtles, patches, and output. Other commands clear subsets of the model.

```
to setup
clear-all ;; clears entire world, usually called at beginning
end
```

## Day 2: God created the sky

# Variables, the volatile part of the model

- The initial part of the program should contain the declaration of global and agent variables (in this order).
- Logo is a dynamically-typed language
- We have the following basic types and operators:
  - **Booleans**: true or false;
  - **Lists**: ordered, immutable collections of objects;
  - **Strings**: immutable sequences of characters; create with double quotes.
  - **Numbers**: floating point numbers;
  - **Numerical operators**: +, -, /, ^;
  - **Relational operators**: >, >=, =, !=, <, <=;
  - **Logical operators**: and, or, xor, not.

**REMARK**: Since all numbers in Netlogo are floating points, we are going to work with approximations! The value of 0.1 + 0.2 is 0.30000000000000004, so the value of 0.1 + 0.2 = 0.3 is false, and the value of 0.1 + 0.2 > 0.3 is true.

# Setting Variables

There are **3 main procedures to output the value of a variable** on the console. The `print` command prints some value to the console, followed by a carriage return. The `type` command also prints a value to the console, but does not include a carriage return. The `show` command also prints a value (followed by a carriage return), but includes the identifying features of the calling agent.

There are 2 procedures aimed at **setting variable's value**, `let` and `set`. The `let` command creates a new local value with a given value. If you want to later change the value of any variable, you use the `set` command. Example:

```
let finished? true
type finished? ;; report value of finished? to output screen
set finished? false
```

## Variables’ Types

- We can divide  variables in:
  - **Global variables**: they have only one value. Every agent can  access it;
  - **Agent variables** (e.g. turtle variables): each agent has its own  value for each agent variable. There are both built-in and  user-defined agent variables.
  - **Local variables**: variables that are defined and used only in the  context of a particular procedure or part of a procedure.

## Global Variables

- Global variables can be read and set by any agent at any time (by default global variables are set to zero). Global variables are declared with the keyword `globals`:

```
globals [ <variableNames> ]
```

- Example

```
globals [ score counter ] ;; declare the global variables score and counter
```

- To set a global variable we use the keyword `set`:

```
set <variableName> <value>
```

# Agent's Variables

- Agent variables can be both built-in and user-defined; Built-in turtle variables are `color`, `xcor`, `heading` etc. To define a new agent variable we use the constructor:

```
<agentType>-own [ <variableName> ]
```

- Where agent types own are `turtles-own`, `patches-own`, `links-own`.

```
turtles-own [ energy ] ;; each turtle has its own
energy
```

# Agent’s variables

- Each agent has direct access to its own variables, both for reading and setting. Other agents can read and set a different agent’s variables using ask:

```
ask turtle 0 [ show color ] ;; print turtle’s 0 color
ask turtle 0 [ set color blue ] ;; turtle 0 becomes blue
```

- An agent can also read a different agent’s variables using of

```
show [ color ] of turtle 0 ;; print turtle’s 0 color
```

## Agentsets

- So far we have asked either to single agents to do something (`ask turtle 0 [ doSomething ]`) or to all agents to do something (`ask turtles [ doSomething ]`);

- We can also ask to specific sets of agents to execute actions. This is done via the so-called **agentsets**;

- An agentset is nothing but a set of agents. An agentset can contain either turtles, patches or links, but not more than one type at once;

- An agentset is not in any particular order. In fact, it’s always in a random order. And every time you use it, the agentset is in a different random order.

# Agentsets

It’s possible to construct **agentsets** that  contain only some agents (e.g. all the red turtles).,e.g. according to a particular attribute:

```
ask one-of turtles [ set color green ] ;; make a randomly chosen
turtle turn green
patches with [ pxcor < 3 ] ;; create the agentset of patches with x
coordinate smaller than 3
to changeColor
let green-turtles turtles with [ color = green ] ;; create the
agenset of green turtles
ask green-turtles [ set color red ] ;; make green turtles red
```

# Local Variables

- A local variable is defined and used only in the context of a particular procedure or part of procedure. To create local variables we use the keyword `let`;

```
let <variableName> <value>
```

- Example

```
to-report sumOfThree [ num1 num2 num3 ] ;; report num1+num2+num3
       let numTemp num1 + num2 ;; store num1 + num2
       report numTemp + num3 ;;report the sum
end
```

## Day 3: God created the earth

# Patches related properties

- Patches are identified by their variables `pxcor` and `pycor`. Other variables are:
- `pcolor`: the color of the patch.
- `plabel` : It may hold a value of any type.
- `plabel-color` : It holds a number greater than or equal to 0 and less than 140. This number determines what color the patch's label appears in (if it has a label).

- Moreover, there are also commands to pull up specific (or random) coordinates. The commands "`max-pxcor`" and "`max-pycor`" return the maximum x and y coordinates, respectively. The commands "`random-pxcor`" and "`random-pycor`" return random x,y coordinates.

# One other patches’ primitive

- `Diffuse`: when studying the spread of some process or resource, it may be useful to use the “`diffuse`” command. This tells each patch to give equal shares of (number * 100) percent of the value of patch-variable to its eight neighboring patches. Its argument should be between 0 and 1. Note that this is an observer command. Example:

```
diffuse pcolor 0.5 ;; each patch diffuses 50% of its variable color to its neighboring 8 patches. Thus, each patch gets 1/8 of 50% of the color from each neighboring patch.
```

## Day 4: God creates all the stars and celestial objects

# Conditionals

- We can use booleans for conditional code execution.

```
if ( <boolCondition> ) [ <commands> ] ;; note parentheses
ifelse ( <boolCondition> )
[ <commands-for-true> ]
[ <commands-for-false> ]
ifelse-value ( <boolCondition> ) ;; allows condition determination of
value
[ <reporter-for-true> ]
[ <reporter-for-false> ]
```

- Example:

```
if ( random-float 1 < 0.5 ) [ show "heads" ] ;; random float num return a
floating number between 0 and num
```

- Example:

```
ask turtles [
set color ifelse-value ( wealth < 0 ) [ red ] [ blue ]
]
```

## Loops

NetLogo offers several looping constructs:

```
loop [ <commands> ]
```

The list of commands <commands> is repeatedly executed (potentially forever). We can use `stop` to exit a loop: if one of the commands eventually calls `stop`, you will exit the loop.
Example:

```
loop [ ifelse ( counter > 100 )   [ stop ]
[ set counter counter + 1 ]
]
```

```
foreach [ item in list ] [<commands> ]
```

Example:

```
let my-turtle-list (list (turtle 1) (turtle 2) (turtle 3) )
foreach my-turtle-list [ ask ? [ fd 2 ] ] ;; ask each turtle in list to
move forward 2 steps
```

# Loops

- We can also use `repeat` and `while` for bounded iteration:

```
repeat <num> [ <commands> ]
```

- The sequence of commands <commands> is executed exactly <num> times.

```
while [ <condition>] [<commands> ]
```

- The sequence of commands <commands> is executed exactly <num> times. Example:

```
while [ ticks < 100 ] ;; while tick count is less than 100
[ go ] ;; run command "go"
```

REMARKS: For most of the tasks we will encounter loops are not needed. In NetLogo, we usually program a single cycle/step of the simulation (via the procedure `go`), and then use a forever button in order to repeat that cycle forever. We can click again on the forever button to exit the loop.

## Instruction

- Instructions tell agents what to do. We can divide instructions according to three criteria:
  - Whether they are built into NetLogo (**primitives**) or whether they are implemented by the user (**procedure**);
  - Whether the instruction produces an output (**reporters**) or not (**commands**);
  - Whether the instruction takes inputs or not.

## Commands

- **Commands** are procedures that do not output and may take inputs:

```
to <commandName> [ <formalParameters> ]
<commands>
end
```

- Examples:

```
to go

clear-all ;; reset the world

create-turtles 100 ;; create 100 turtles
ask turtles [ forward 1 ] ;; all turtles move fd of 1
end
```

```
to createNumTurtles [ num ]

create-turtles num

end
```

# Reporters

- **Reporters** are procedures that compute a result and report it:

```
to-report <reporterName>
<body>
report <reportValue>  end
```

- Example:

```
to-report double [ num ]
report 2 * num ;; note that there is no parentheses
end
```

# More utilities...

- There is a number of more general purpose functions (e.g. `ceiling, floor, int, min, max, mean, standard-deviation, round, random`).
- Check the Netlogo documentation [http://ccl.northwestern.edu/netlogo/docs/dictionary.html](http://ccl.northwestern.edu/netlogo/docs/dictionary.html).

## Day 5 and 6: God created all the living beings

# Creating/Killing agents

There are 2 main agents type: **turtle** (the moving one) and **links** (used with graph and network modeling). In this course we will focus on the turtles. All the following primitives create new turtles, though from different contexts.

- `create` command all called by the observer. Note that you can immediately assign turtle characteristics and issue commands. This is an alternative to the keyword `turtles-own [`*`var1`*` ...]` or *⟨breeds⟩*`-own [`*`var1`*` ...]` that can only be used at the beginning of a program, before any function definitions. It defines the variables belonging to each individual turtle or breed.
- `hatch` command is called by the turtles. Turtles “hatched” have features identical to their predecessor.
- `sprout` command is called by patches.
- `breeds` creates a specific breed.
- `die` removes one or more agents.

# Creating/Killing agents

- Examples:

```
ask n-of 10 patches [ sprout 1 ] ;; ask 10 random patches to sprout 1
turtle

ask turtle 0 [ hatch 1 ] ;; ask turtle 0 to "hatch" a new turtle

create-turtles 10 ;; creates 10 turtles

breed [wolverines wolverine] ;; define a breed called "wolverines"

create-wolverines ;; creates 1 turtles of breed wolverine

create-wolverines 50 [ set color blue ] ;; creates 50 blue-colored
wolverines

ask turtle 4 [ die ]
```

# Ask command

The observer can gives commands to turtles (and patches).

```
create-turtles 1 ;; creates one turtle (who is turtle 0) in  the
centre of the world
inspect turtle whoID ;; shows properties of turtle whoID
```

The ask command specifies command to be run by turtles or patches:

```
ask somebody [ setOfComands ]
ask turtle whoID [ set color red ] ;; set the color of turtle
whoID to red;
ask turtles [ set color red ] ;; set the color of all turtles  to
red;
ask patch 2 3 [ set pcolor green] ;; set the color of the patch
with center (2, 3) to green (note that we used pcolor)
```

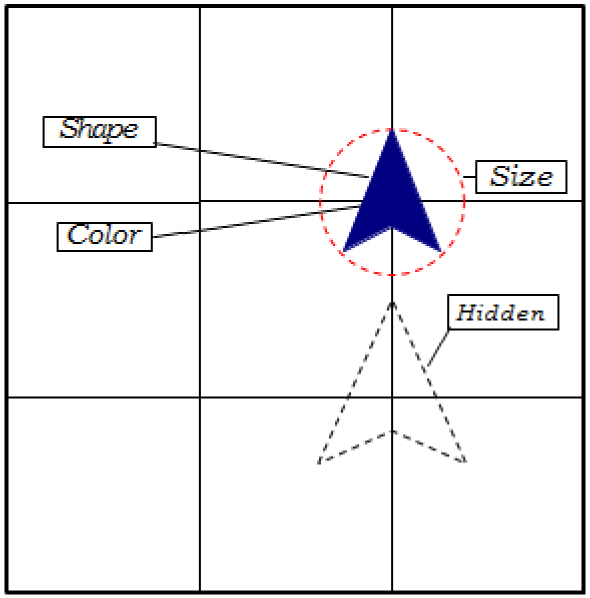
In an ask command, the actions to be executed must be consistent  with the agent we are asking to execute those very  actions.

```
ask turtle whoID [ set color green ];; this makes sense
ask turtle whoID [ create-turtles 1 ];; this doesn't make sense
```

# Turtle Properties

The world is inhabited by agents called turtles, i.e. **movable entities** within the world, having a:

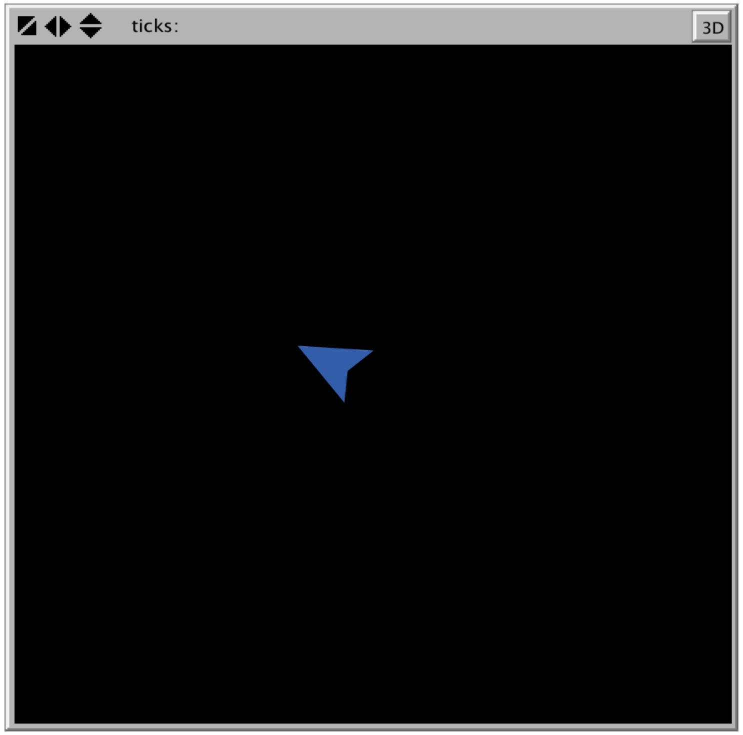
- Unique ID: resulting by the «who» procedure
- Size: size of a turtle w.r.t. the patch size (1 by default).
- Hidden? : boolean properties that makes invisible a given turtle
- Position: grid coordinates (xcor,ycor)
- Heading: degrees (0 north, 90 est,...)
- Shape: an arrow by default
- Color

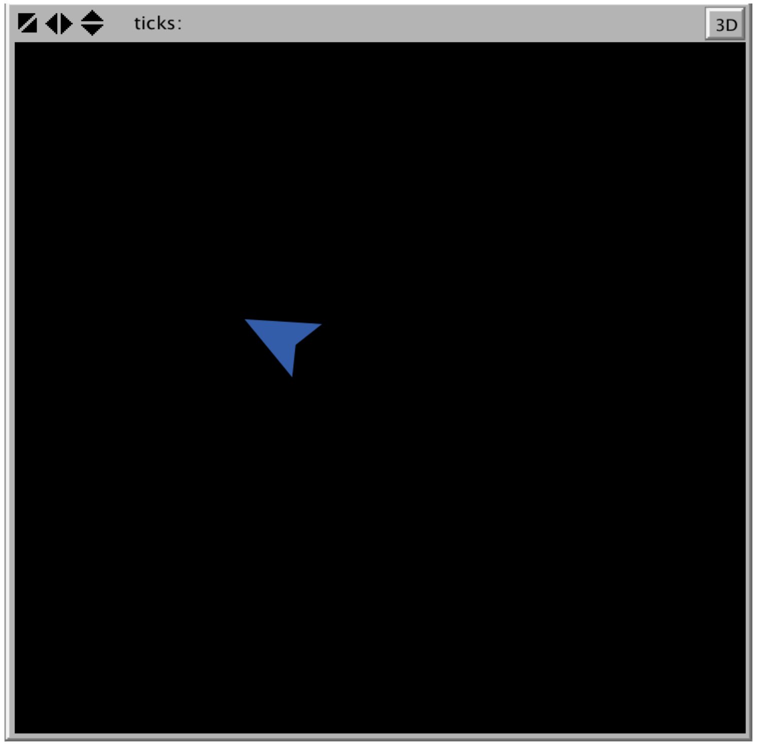


# Moving

We can move turtles forward and backwards with the commands forward <distance> and back <distance>:

```
ask turtle 0 [ forward 2 ]
```

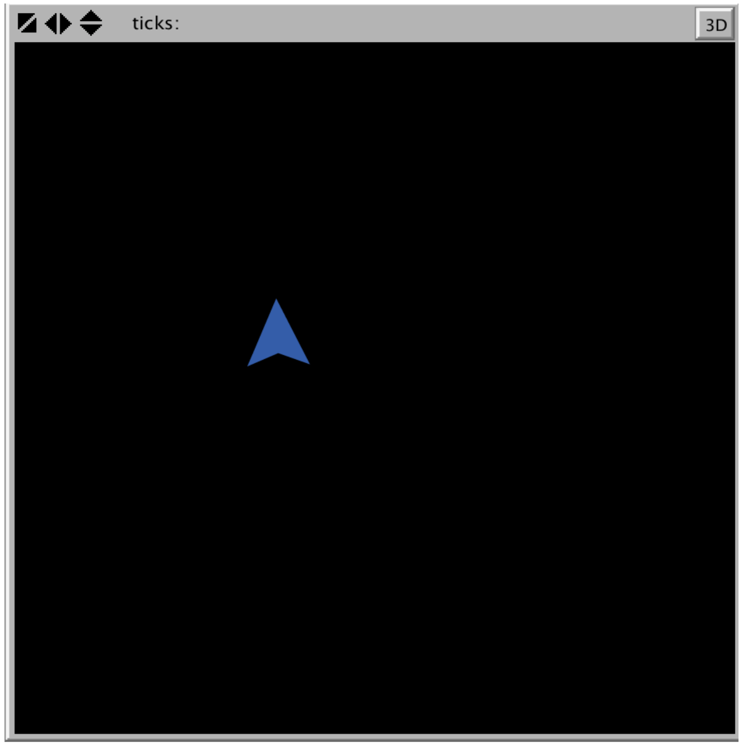


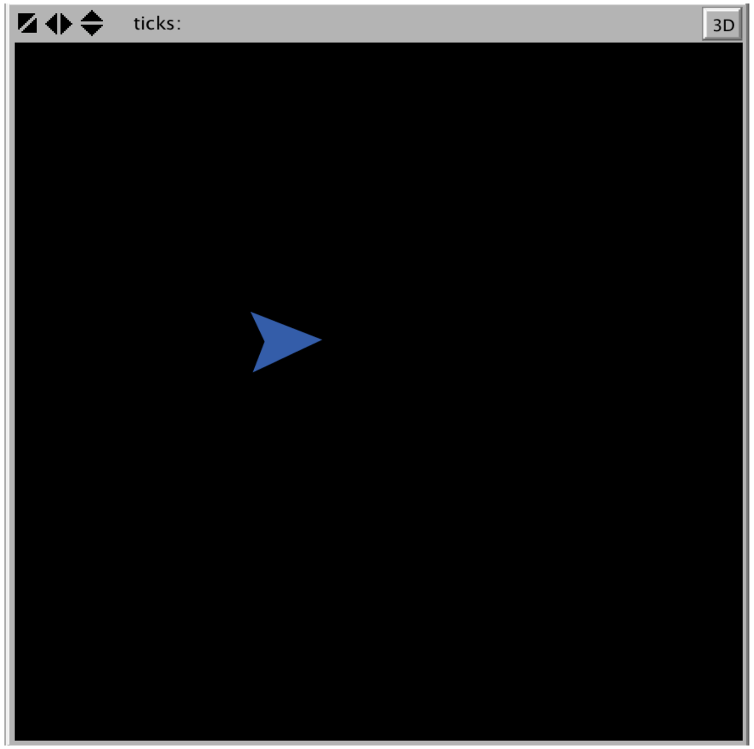


# Turning

Turtles can turn left and right: `left` <degree>, `right` <degree>:

```
ask turtle 0 [ right 90 ]
```





## More movement procedures…

`Back n`, moves backward by *n* steps.

`Setxy x y`, sets its x-coordinate to *x* and its y-coordinate to *y*.

`move-to agent`, sets its x and y coordinates to be the same as the given agent's

`towards`, reports the heading from this agent to the given agent

`towardsxy`, reports the heading from the turtle or patch towards the point (*x*,*y*)

## Checking to the environment

`myself`, "self" and "myself" are very different. "self" is simple; it means "me". "myself" means "the turtle, patch or link who asked me to do what I'm doing right now.“

`nobody`, This is a special value which some primitives such as turtle, one-of, max-one-of, etc. report to indicate that no agent was found. Also, when a turtle dies, it becomes equal to nobody.

`turtles-at` *`dx dy`*, Reports an agentset containing the turtles on the patch (dx, dy) from the caller.

`turtles-on` ***`agent`*****,** reports an agentset containing all the turtles that are on the given patch or patches, or standing on the same patch as the given turtle or turtles.

`turtles-here`, reports an agentset containing all the turtles on the caller's patch (including the caller itself if it's a turtle).

# Checking to the environment

`Neighbors`: Reports an agentset containing the 8 surrounding patches (neighbors) or 4 surrounding patches (neighbors4). Example:

```
show sum [count turtles-here] of neighbors ;; prints the total number of
turtles around this turtle or patch
```

`In-cone`: in-cone reports an agentset that includes only those agents from the original agentset that fall in the cone The cone is defined by the two inputs, the vision distance (radius) and the viewing angle. The viewing angle may range from 0 to 360 and is centered around the turtle's current heading. Example:

```
ask patches in-cone 3 60 [ set pcolor red ]
```

*agentset* `in-radius` *number*: Reports an agentset that includes only those agents from the original agentset whose distance from the caller is less than or equal to *number*. Example:

```
ask patches in-radius 3 [ set pcolor red ]
```

[*reporter*] `of` *agent*: For an agent, reports the value of the reporter for that agent (turtle or patch). Example:

```
show [pxcor] of patch 3 5
```

# Checking to the environment

`[reporter] of agentset`*:* For an agentset, reports a list that contains the value of the reporter for each agent in the agentset (in random order). Example:

```
crt 4
show sort [who * who] of turtles ;; [0 1 4 9]
```

`one-of  agentset` *:* From an agentset, reports a random agent. Example:

```
ask one-of patches [ set pcolor green ] ;; a random patch turns green
```

`Count agentset` : Reports the number of agents in the given agentset. Example:

```
show count patches with [pcolor = red] ;; prints the total number of red patches
```

`agentset with [reporter]:` Takes two inputs: on the left, an agentset (usually "turtles" or "patches"). On the right, a boolean reporter. Reports a new agentset containing only those agents that reported true -- in other words, the agents satisfying the given condition. It can be applyied also as `with-max` and `with-min`. Example:

```
show count patches with [pcolor = red] ;; prints the number of red patches
show count patches with-min [pycor] ;; prints the number of patches on the bottom
edge
```

# Checking the environment

`any? Agentset` reports true if the given agentset is non-empty, false otherwise. Equivalent to "`count` *`agentset`* `> 0`", but more efficient (and arguably more readable). Example:

```
if any? turtles with [color = red] [ show "at least one turtle is red!" ]
```

`all?` *`agentset [reporter]`* reports true if all of the agents in the agentset report true for the given reporter. Otherwise reports false as soon as a counterexample is found. Example:

```
if all? turtles [color = red] [ show "every turtle is red!" ]
```

`n-of size agentset`, from an agentset, reports an agentset of size *size* randomly chosen from the input set, with no repeats. Example:

```
ask n-of 50 patches [ set pcolor green ] ;; 50 randomly chosen patches turn
green
```

`max-n-of`, reports a random agent in the agentset that reports the greater value for the given reporter. Example:

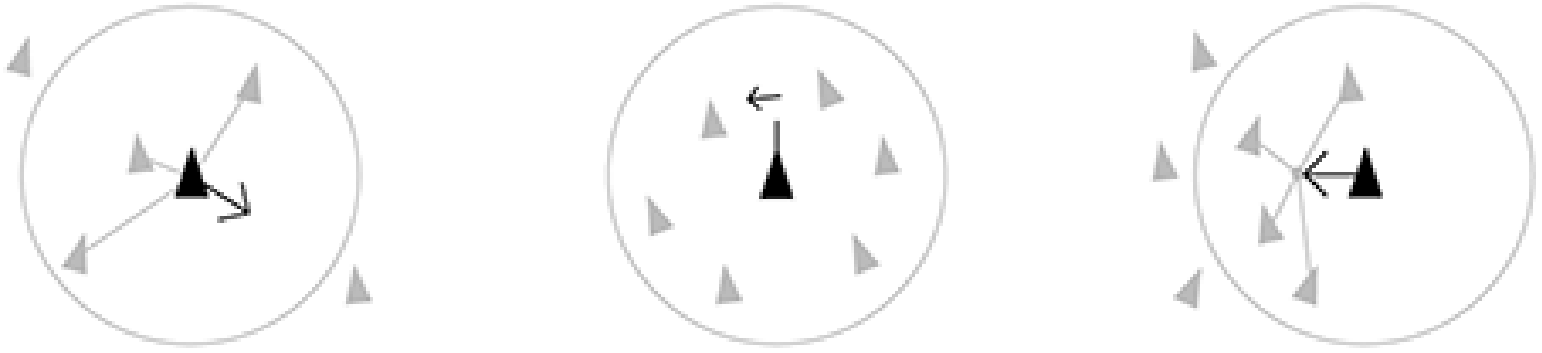
```
show max-one-of turtles [xcor + ycor] ;; reports the first turtle with the
greater sum of coordinates
```

## Day 7: God did rest.

Since you all, as Ph.D. students, are way less divine living being, you may need some «Being-God practice» ...

## Exercize

- Modify the «**Flocking**» model by letting the boids change their colors according to their current mode: alignment (blue), separation (red) or cohere (green).



**Separation:**
Steer to avoid crowding local flockmates

**Alignment:**
Steer toward the average heading of local flockmates

**Cohesion:**
Steer to move toward the average position of local flockmates

## Any question?

# Designing and prototyping a Biologically Inspired Swarm Intelligence System

PART 2

## Local and Global Swarm Features

The behavior of swarm intelligent systems if often said to be an “emergent behavior”

- It does not arise from a rationale choice
- It does not arise from an engineering finalized analysis
- No one and nothing in the system states: I will do that because this will lead to a specific behavior of the system

So, **intelligence seems to magically “emerge”.** Clearly, emergence is in the eye of the observer:

- The individual in the system have no global perspective
- They are not aware of what’s globally happening
- They are not aware they are doing something intelligent

## Local and Global Swarm Features

The “intelligence” is observed in terms of some high level global scale organized behavior:

- Spontaneously emerged in the system
- Spanning outside the typical local capability of sensing/effecting of individuals

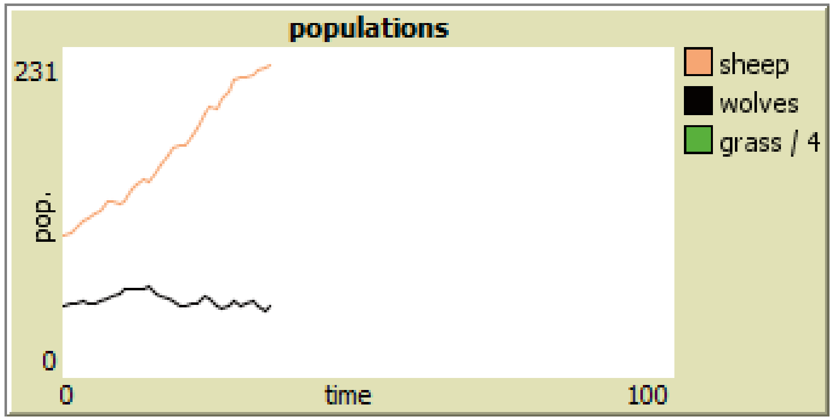
From Local to Global:

- Typically, in swarm systems, interactions and capability are local
- Nevertheless, the behavior observed has some sorts of global organization

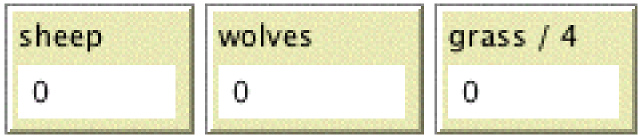
To evaluate the quality of a swarm system we need to build some sort of **local** and **global observers**.

# Monitors And Plotting

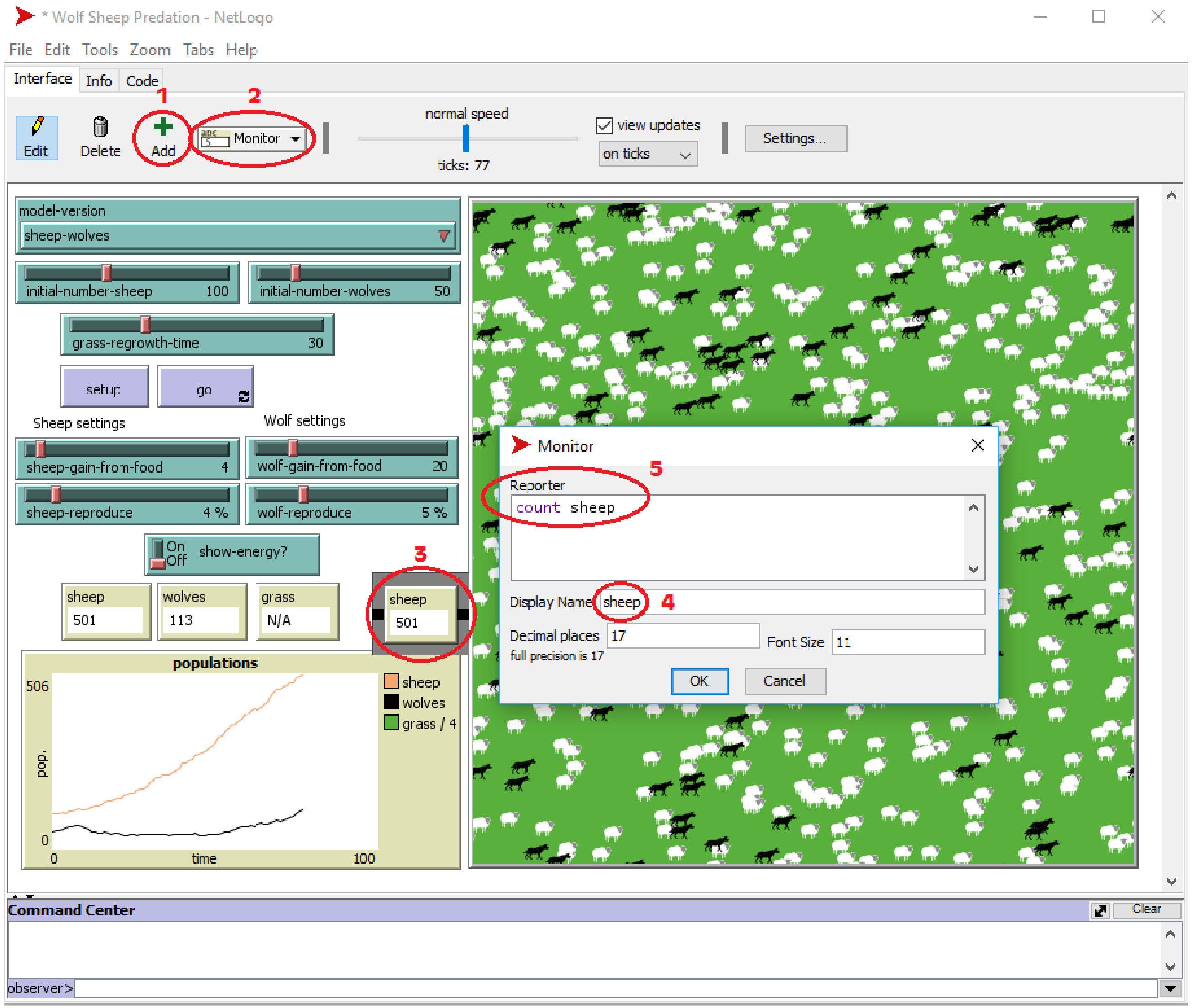
- NetLogo has two main ways of displaying performances and data to the user:
  - **Plots**: show the values of a few variables from the whole model's run



  - **Monitors**: show the value of one variable the model runs.



# Monitors



model-version: sheep-wolves

initial-number-sheep 100
initial-number-wolves 50
grass-regrowth-time 30
sheep-gain-from-food 4
wolf-gain-from-food 20
sheep-reproduce 4 %
wolf-reproduce 5 %

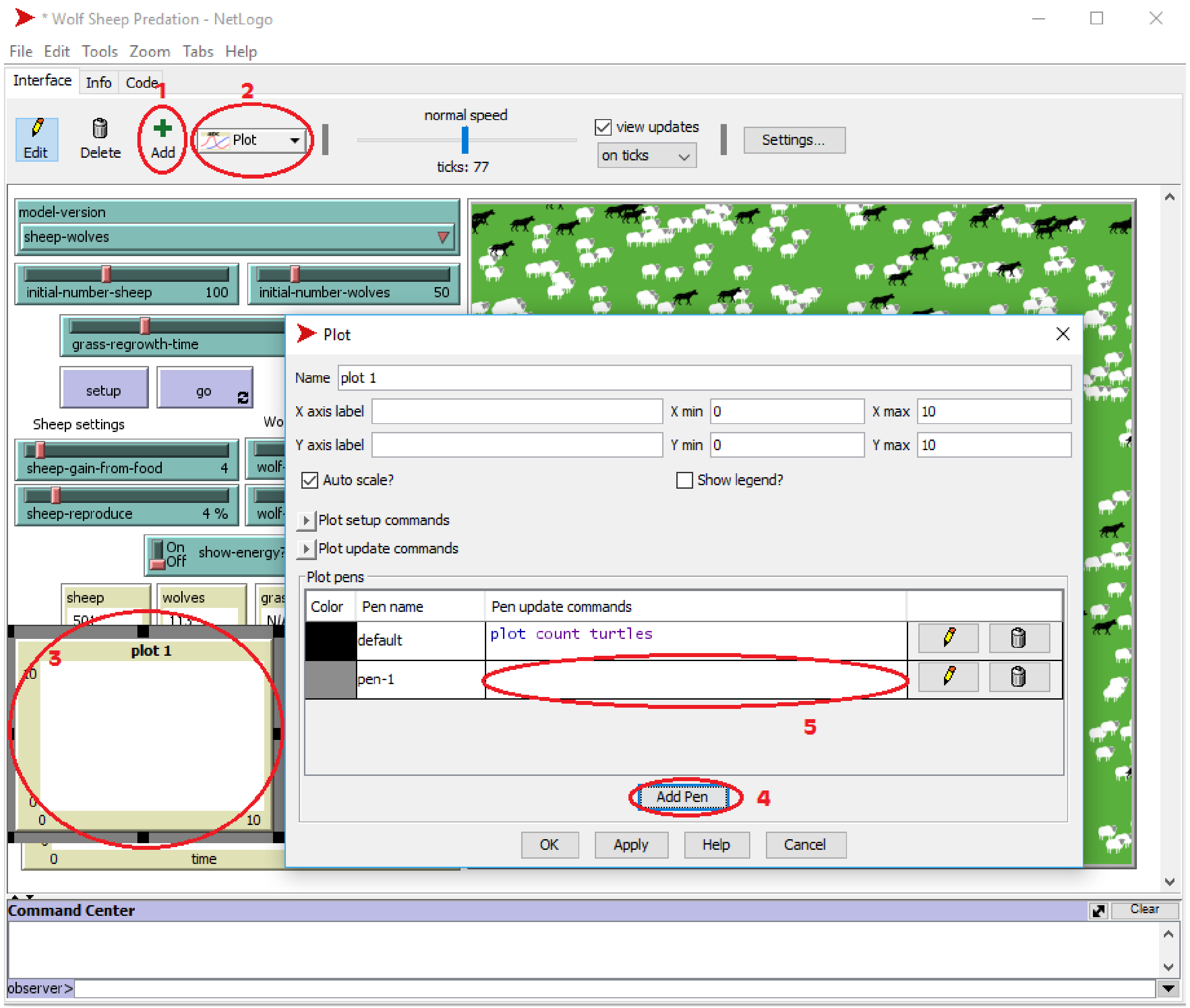
sheep 501
wolves 113
grass N/A
sheep 501

Monitor

Reporter
```
count sheep
```

Display Name: sheep
Decimal places: 17
full precision is 17
Font Size: 11
OK
Cancel

# Plots



model-version: sheep-wolves

initial-number-sheep 100

initial-number-wolves 50

grass-regrowth-time

sheep-gain-from-food 4

sheep-reproduce 4 %

show-energy? On Off

Plot

Name: plot 1

X axis label: | X min: 0 | X max: 10

Y axis label: | Y min: 0 | Y max: 10

[✓] Auto scale?  [ ] Show legend?

| Color | Pen name | Pen update commands |
|---|---|---|
| ■ | default | `plot count turtles` |
| ▨ | pen-1 | |

## Plotting

Of course it is possible to code your plot in the code tab by using primitives such as: `set-current-plot`, `set-current-plot-pen`, `plot`, `histogram`, `plotxy`. Example:

```
to do-plotting
set-current-plot "populations" ;;name of the plot
set-current-plot-pen "sheep" ;; name of the variable
plot count sheep
set-current-plot-pen "wolves"
plot count wolves
set-current-plot "next plot"
end
```

Check the documentation for more details.

## Testing the behavior with a coverage problem

At the beginning of the model “Flocking Vee Formation” add:

```
globals [
visited-count
visited-perc
hits
]

patches-own [
visited?
]
```

# Testing the behavior with a coverage problem

- Add to setup:

```
ask patches [set visited? false]
```

- Replace to go:

```
to go
  ask turtles [
    ask patches in-radius ((vision-distance / 2)  - too-close) [set visited? true]
    ask patches in-radius ((vision-distance / 2)  - too-close) [set pcolor blue]
    set speed base-speed
    set visible-neighbors (other turtles in-cone vision-distance vision-cone)
    ifelse any? visible-neighbors
      [ adjust ]
      [ set happy? true ]
    recolor
    fd speed  ; fly forward!
    if any? turtles in-radius too-close [ set hits hits + 1 ]
  ]
  set visited-count count patches with [visited? = true]
  set visited-perc visited-count / ((max-pycor - min-pycor)*(max-pxcor - min-pxcor))
  if visited-perc > 0.99 [ stop ]
  tick
end
```

# Exploration of the Parameters’ Space

Dramatically change (one by one):

- `Max-turn`
- `Too-close`
- `Speed-change-factor`
- `Updraft-distance`

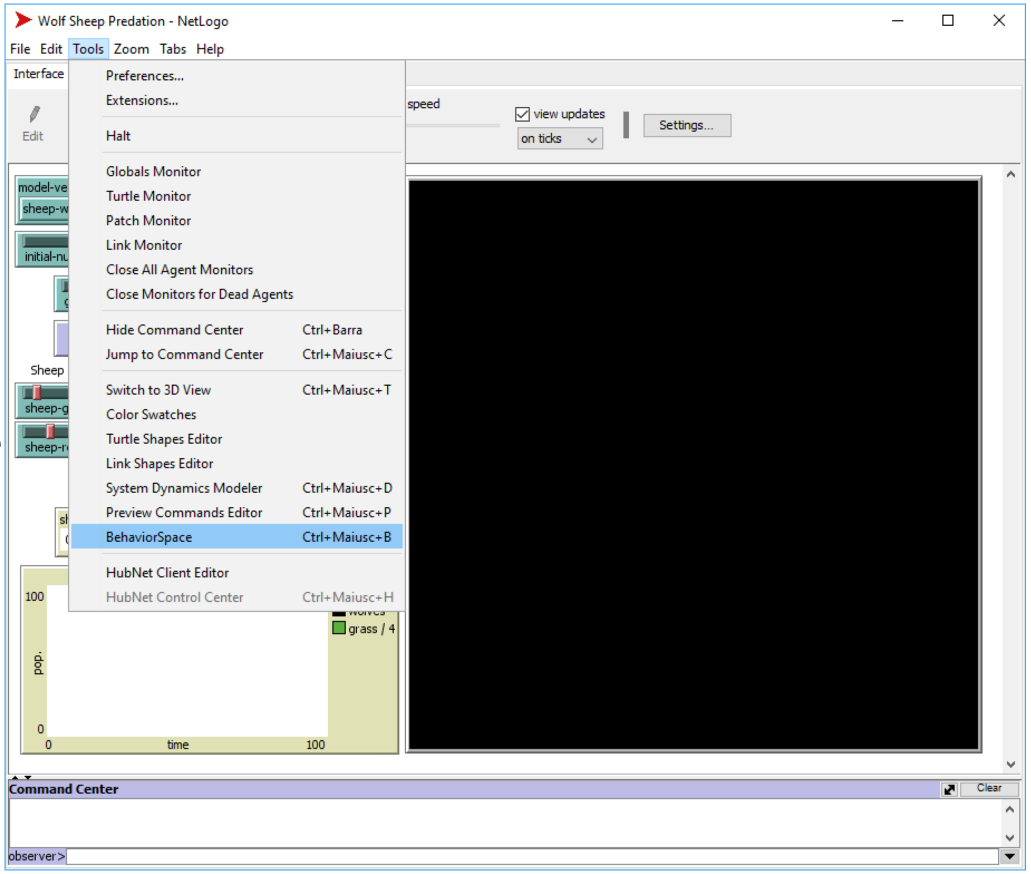
What did you notice?

## The Behavior space

- **BehaviorSpace** is a software tool integrated with NetLogo that allows you to systematically perform experiments with models.
- BehaviorSpace runs a model many times, varying the model's settings and recording the results of each model run. This process is sometimes called "parameter sweeping". It lets you explore the model's "space" of possible behaviors and determine which combinations of settings cause the behaviors of interest.

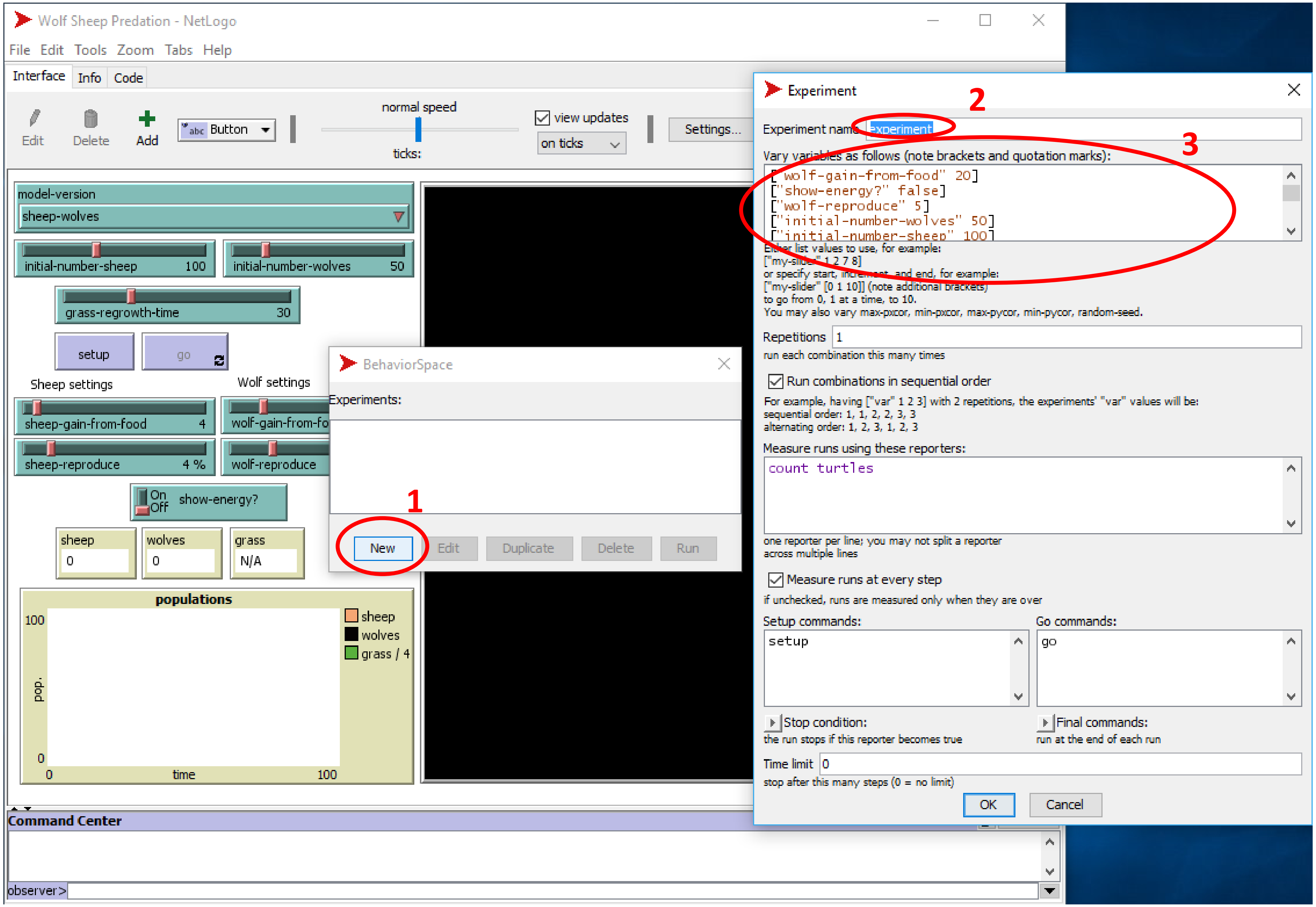
# Using the Behavior space

- To begin using BehaviorSpace, open your model, then choose the BehaviorSpace item on NetLogo's Tools menu.
- The dialog that opens lets you create, edit, duplicate, delete, and run experiment setups. **Experiments** are listed by name and how by model runs the experiment will consist of.
- Experiment setups are considered part of a NetLogo model and are saved as part of the model.


Wolf Sheep Predation - NetLogo
File Edit Tools Zoom Tabs Help
Interface
Edit
Preferences...
Extensions...
Halt
Globals Monitor
Turtle Monitor
Patch Monitor
Link Monitor
Close All Agent Monitors
Close Monitors for Dead Agents
Hide Command Center Ctrl+Barra
Jump to Command Center Ctrl+Maiusc+C
Switch to 3D View Ctrl+Maiusc+T
Color Swatches
Turtle Shapes Editor
Link Shapes Editor
System Dynamics Modeler Ctrl+Maiusc+D
Preview Commands Editor Ctrl+Maiusc+P
BehaviorSpace Ctrl+Maiusc+B
HubNet Client Editor
HubNet Control Center Ctrl+Maiusc+H
speed
view updates
on ticks
Settings...
Sheep
100
pop.
0
0
time
100
grass / 4
Command Center
Clear
observer>

# Setting up the Behavior Space

- To create a new experiment setup, press the “**New**” button
- **Experiment name:** If you have multiple experiments, giving them different names will help you keep them straight.
- **Vary variables as follows:** This is where you specify which settings you want varied, and what values you want them to take. Variables can include sliders, switches, choosers, and any global variables in your model.
- You may specify values either by listing the values you want used, or by specifying that you want to try every value within a given range. For example, to give a slider named number every value from 100 to 1000 in increments of 50, you would enter:
- `["number" [100 50 1000]]`
- Or, to give it only the values of 100, 200, 400, and 800, you would enter:
- `["number" 100 200 400 800]`



model-version: sheep-wolves
initial-number-sheep 100
initial-number-wolves 50
grass-regrowth-time 30
sheep-gain-from-food 4
sheep-reproduce 4 %
wolf-gain-from-fo
wolf-reproduce
On Off show-energy?
sheep 0
wolves 0
grass N/A

BehaviorSpace

Experiments:

New Edit Duplicate Delete Run

Experiment

Experiment name experiment

Vary variables as follows (note brackets and quotation marks):

```
["wolf-gain-from-food" 20]
["show-energy?" false]
["wolf-reproduce" 5]
["initial-number-wolves" 50]
["initial-number-sheep" 100]
```

Either list values to use, for example:
["my-slider" 1 2 7 8]
or specify start, increment, and end, for example:
["my-slider" [0 1 10]] (note additional brackets)
to go from 0, 1 at a time, to 10.
You may also vary max-pxcor, min-pxcor, max-pycor, min-pycor, random-seed.

Repetitions 1
run each combination this many times

☑ Run combinations in sequential order
For example, having ["var" 1 2 3] with 2 repetitions, the experiments' "var" values will be:
sequential order: 1, 1, 2, 2, 3, 3
alternating order: 1, 2, 3, 1, 2, 3

Measure runs using these reporters:

```
count turtles
```

one reporter per line; you may not split a reporter across multiple lines

☑ Measure runs at every step
if unchecked, runs are measured only when they are over

Setup commands:

```
setup
```

Go commands:

```
go
```

Stop condition:
the run stops if this reporter becomes true

Final commands:
run at the end of each run

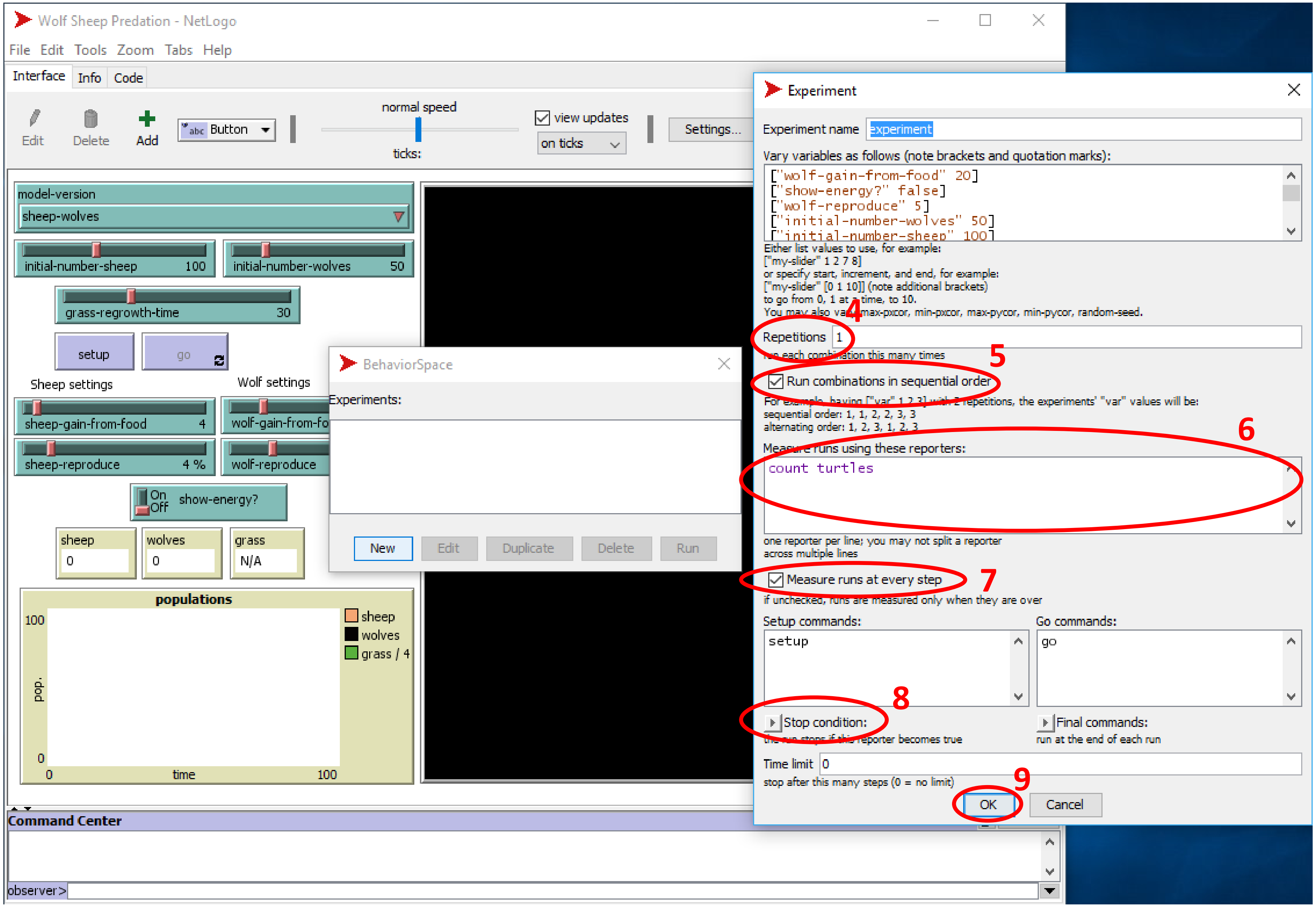
Time limit 0
stop after this many steps (0 = no limit)

OK Cancel

# Setting up the Behavior Space

- **Measure runs at every step:** Normally NetLogo will measure model runs at every step, using the reporters you entered in the previous box. If you're doing very long model runs, you might not want all that data. Uncheck this box if you only want to measure each run after it ends.
- **Setup commands:** These commands will be used to begin each model run. Typically, you will enter the name of a procedure that sets up the model, typically setup. But it is also possible to include other commands as well.
- **Go commands:** These commands will be run over and over again to advance to the model to the next "step". Typically, this will be the name of a procedure, such as go, but you may include any commands you like.
- **Stop condition:** This lets you do model runs of varying length, ending each run when a certain condition becomes true. For example, suppose you wanted each run to last until there were no more turtles. Then you would enter:

```
not any? turtles
```



Experiment name experiment

Vary variables as follows (note brackets and quotation marks):

```
["wolf-gain-from-food" 20]
["show-energy?" false]
["wolf-reproduce" 5]
["initial-number-wolves" 50]
["initial-number-sheep" 100]
```

Either list values to use, for example:
["my-slider" 1 2 7 8]
or specify start, increment, and end, for example:
["my-slider" [0 1 10]] (note additional brackets)
to go from 0, 1 at a time, to 10.
You may also vary max-pxcor, min-pxcor, max-pycor, min-pycor, random-seed.

Repetitions 1
run each combination this many times

☑ Run combinations in sequential order
For example, having ["var" 1 2 3] with 2 repetitions, the experiments' "var" values will be:
sequential order: 1, 1, 2, 2, 3, 3
alternating order: 1, 2, 3, 1, 2, 3

Measure runs using these reporters:

```
count turtles
```

one reporter per line; you may not split a reporter across multiple lines

☑ Measure runs at every step
if unchecked, runs are measured only when they are over

Setup commands:

```
setup
```

Go commands:

```
go
```

Stop condition:
the run stops if this reporter becomes true

Final commands:
run at the end of each run

Time limit 0
stop after this many steps (0 = no limit)

# Running/saving options

- The run options dialog lets you select the formats you would like the data from your experiment saved in. **Data is collected for each run or step**, according to the setting of Measure runs at every step option. In either case, the initial state of the system is recorded, after the setup commands run but before the go commands run for the first time.

- **Table** format lists each interval in a row, with each metric in a separate column. Table data is written to the output file as each run completes. Table format is suitable for automated processing of the data, such as importing into a database or a statistics package.

- **Spreadsheet** format calculates the min, mean, max, and final values for each metric, and then lists each interval in a row, with each metric in a separate column. Spreadsheet data is more human-readable than Table data, especially if imported into a spreadsheet application.

- After selecting your output formats, BehaviorSpace will prompt you for the name of a file to save the results to. The default name ends in “.csv”. You can change it to any name you want, but don’t leave off the “.csv” part; that indicates the file is a Comma Separated Values (**CSV**) file. This is a plain-text data format that is readable by any text editor as well as by most popular spreadsheet and database programs.

# Observing runs

- After you complete the run options dialog, another dialog will appear, titled "**Running Experiment**". In this dialog, you'll see a progress report of how many runs have been completed so far and how much time has passed. If you entered any reporters for measuring the runs, and if you left the "**Measure runs at every step**" box checked, then you'll see a plot of how they vary over the course of each run.
- You can also watch the runs in the main NetLogo window (if the "**Running Experiment**" dialog is in the way, just move it to a different place on the screen.) The view and plots will update as the model runs. If you don't need to see them update, then use the checkboxes in the "**Running Experiment**" dialog to turn the updating off. This will make the experiment go faster.
- If you want to stop your experiment before it's finished, press the "**Abort**" button. Any results generated so far will still be saved.
- When all the runs have finished, the experiment is **complete**.

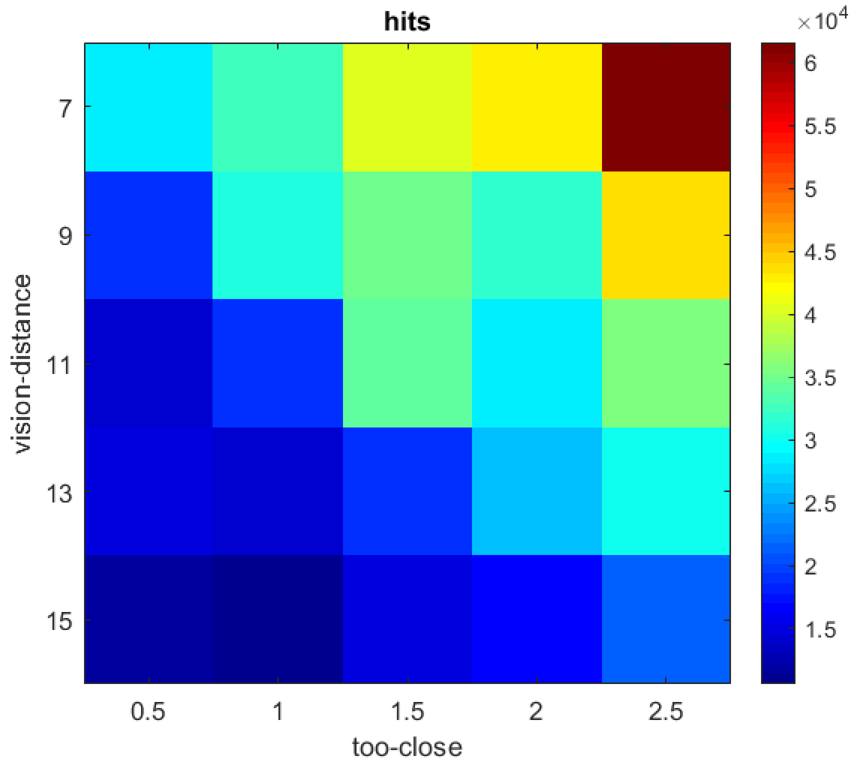
## Simulations’ results

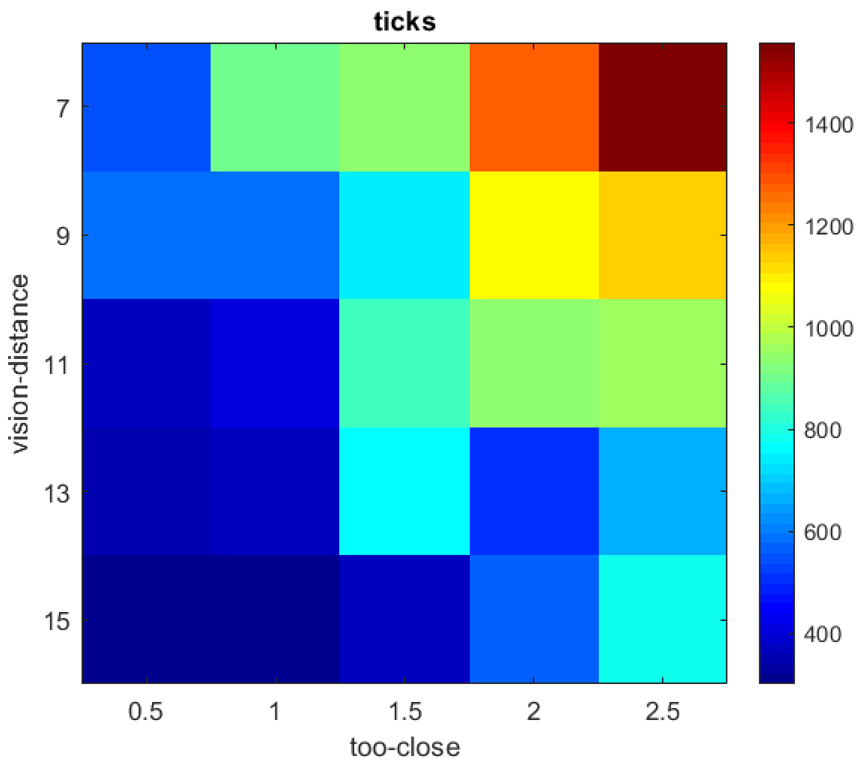
For example, let’s  test the following parametrizations:

- ["vision-distance" [7 2 15]]
- ["too-close" [0.5 0.5 2.5]]

- Which is the best parametrization to ensure good local feature (e.g. time in a safe distance)?
- What about the global feature (e.g. time to cover 90% of the environment)?
- Is there any tradeoff?

# Analyzing the results





# Choosing the variables to test

**Qualitative way**:

1. Run a few simulations "manually" and select the parameters which affect the most the swarm behaviour.
2. Test a range of values for those parameters.

**Quantitative way**:

1. Run multiple simulations systematically, by testing a range of value for each parameters.
2. Perform a regression by using the rescaled input and output model's parameters.
3. Select the parameter with the larger beta coefficients, and test a range of values for those parameters.

**Interpret the results!**

## Exercize

Modify the model “**Flock Vee Formation**” in order to:

- At the setup of the simulation, choose 20 random patches as (non discovered) targets
- A target is **discovered** if a boid **is on top of it**
- Stop the simulation once 19 targets are discovered
- Modify the behavior of the boids in order to minimize the **time to find 19 targets**, by using 20 boids
- Design some protection mechanism to **reduce** as much as possible the time in which each boid is **too-close** to another one.
- Test systematically the parametrizations of the model you made and interpret the results.

# FINAL TEST

Report and present the obtained results. A few guidelines:

The document should  report each choice on the modification of the original Netlogo model, the reasons for this, if these modification are effective and with which parameters settings. Specifically:

- The motivations: i.e. the **reasons** for the modification of the Netlogo model, the performance metrics chosen, and/or the range of parameters chosen.
- The modifications: or explain in concrete terms what **modification** has been carried out and in what way.
- Which **results** produced this modification of the model: use tables/images/graphics depicting the results of the exploration with your model.
- What **conclusions** can be drawn from these results: for example we have satisfied the motivations? If yes, why?  If not, why? how can the results be interpreted?

Any question?

## BACKUP SLIDES

# Lists

Lists are containers of elements in a fixed order. In NetLogo  lists are: Immutable, Ordered, Potentially heterogeneous. We can construct lists with the `list` constructor (note  parentheses):

```
( list ) ;; emtpy list
( list 0 "one" whatever ) ;; list of three items
```

The usual square brackets notation can be used as well.

## files (file-read, file-open, file-write, file-delete, file-close)

- It is often useful to be able to read and write data from files. Netlogo has a number of primitives for managing files. An example follows below.

```
to test-files
    set-current-directory user-directory ;; allow user to select directory for files
    file-open "location.txt" ;; Opening file for writing
    ask turtles
    [ file-write xcor file-write ycor ] ;; ask turtles to write their coordinates
    out to file
    file-close
    file-open "location.txt" ;; Opening file for reading
    ask turtles
    [ setxy file-read file-read ] ;; ask turtles to read their coordinates from file
    file-close
    show file-exists? "location.txt" ;; reports true if file exists
    file-delete "location.txt"
    show file-exists? "location.txt" ;; should now report false
end
```

## **making movies** (movie-start, movie-grab-view, movie-close)

To make a quicktime movie, the bare minimum is that you must name it, tell Netlogo how many “**frames**” of the program to grab, and close it at the end. Note: these Quicktime movies are not compressed, so they can quickly become huge!

```
;; export a 10 frame movie of the view
setup
movie-start "abm.mov" ;; setup movie and name it
movie-grab-view ;; show the initial state
repeat 10 ;; repeat what’s in brackets 10 times
      [
      go ;; run the go procedure
      movie-grab-view
      ]
movie-close
```