Introduction to NetLogo

Institut für Technische Informatik

14.10.2018

Why NetLogo?

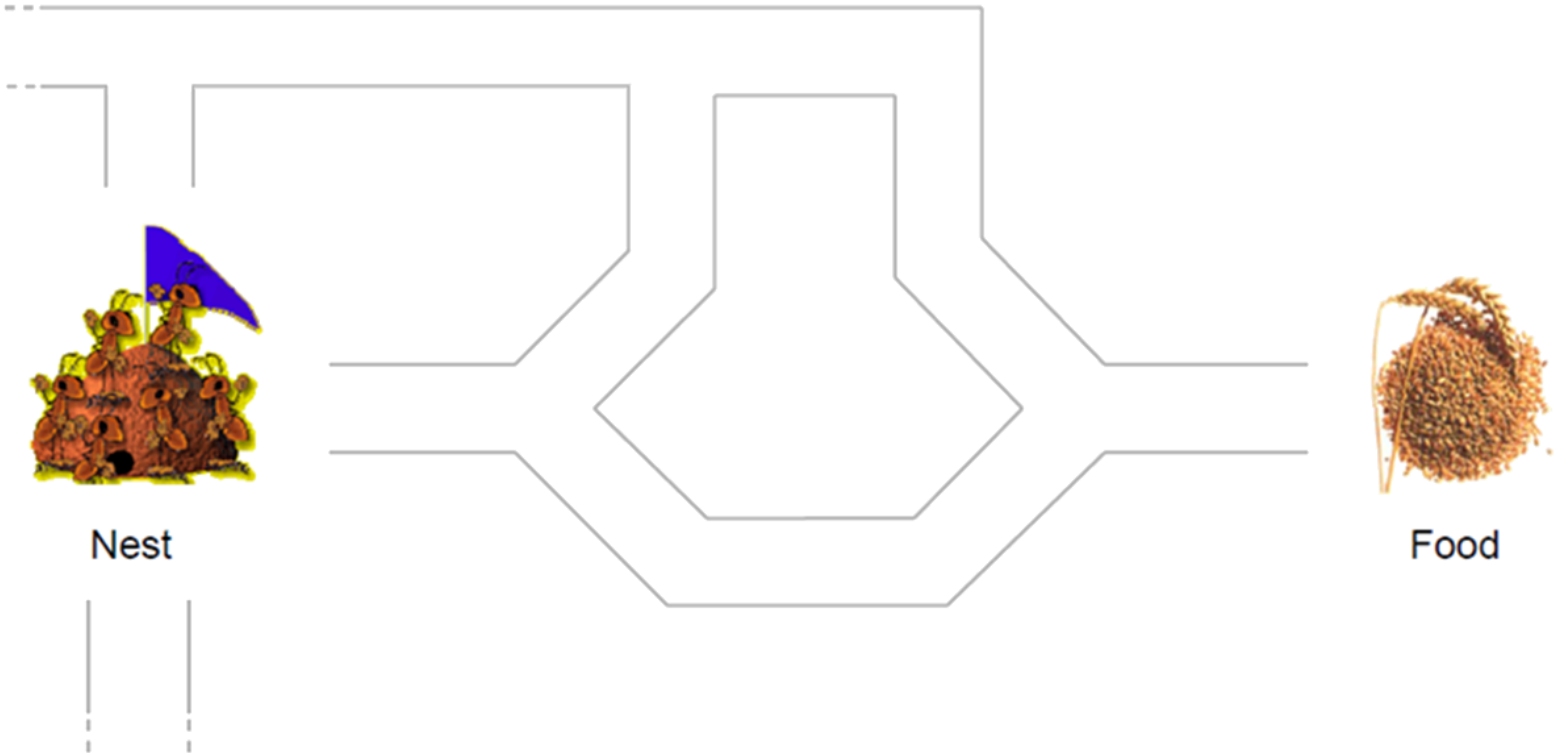
- Understanding complex systems evolving over time can be quite challenging
- Modelling these systems may be necessary
  - How?

→ NetLogo is a simulation environment to study complex systems operating in dynamic context

- Large-scale simulations (thousands of independent agents operating concurrently)
- Can be used to study how to make a system aware of its context so that it can automatically adapt to changes
- Both natural and technical systems

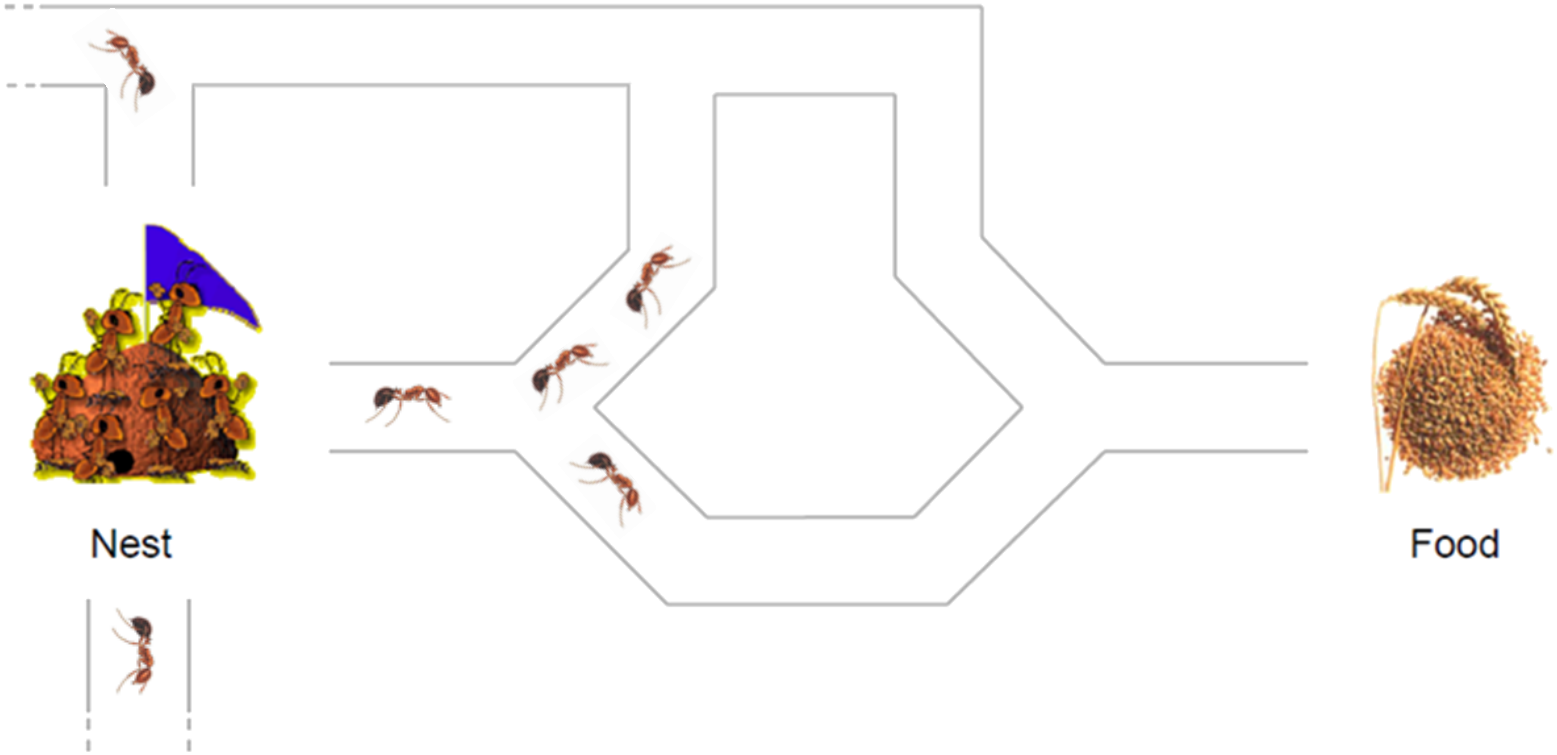
A simple example: Ants foraging food

- NetLogo can be used to
  - Model the micro-level behavior of ants
  - Study the macro-behavior emerging from their interaction



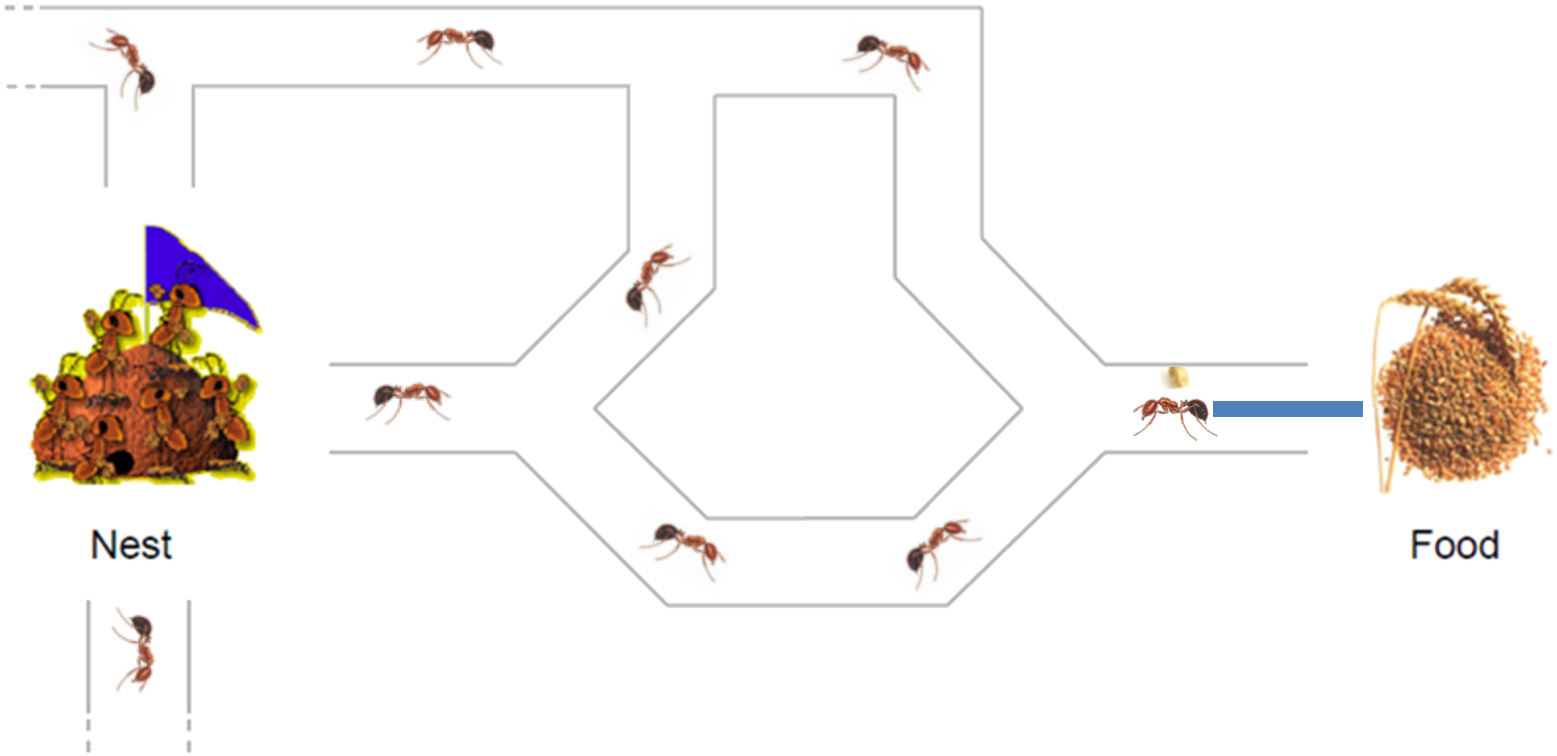
A simple example: Ants foraging food

- Ants are short-sighted
  - Location and distance to the food is unknown
  - But they often know exactly where to go…



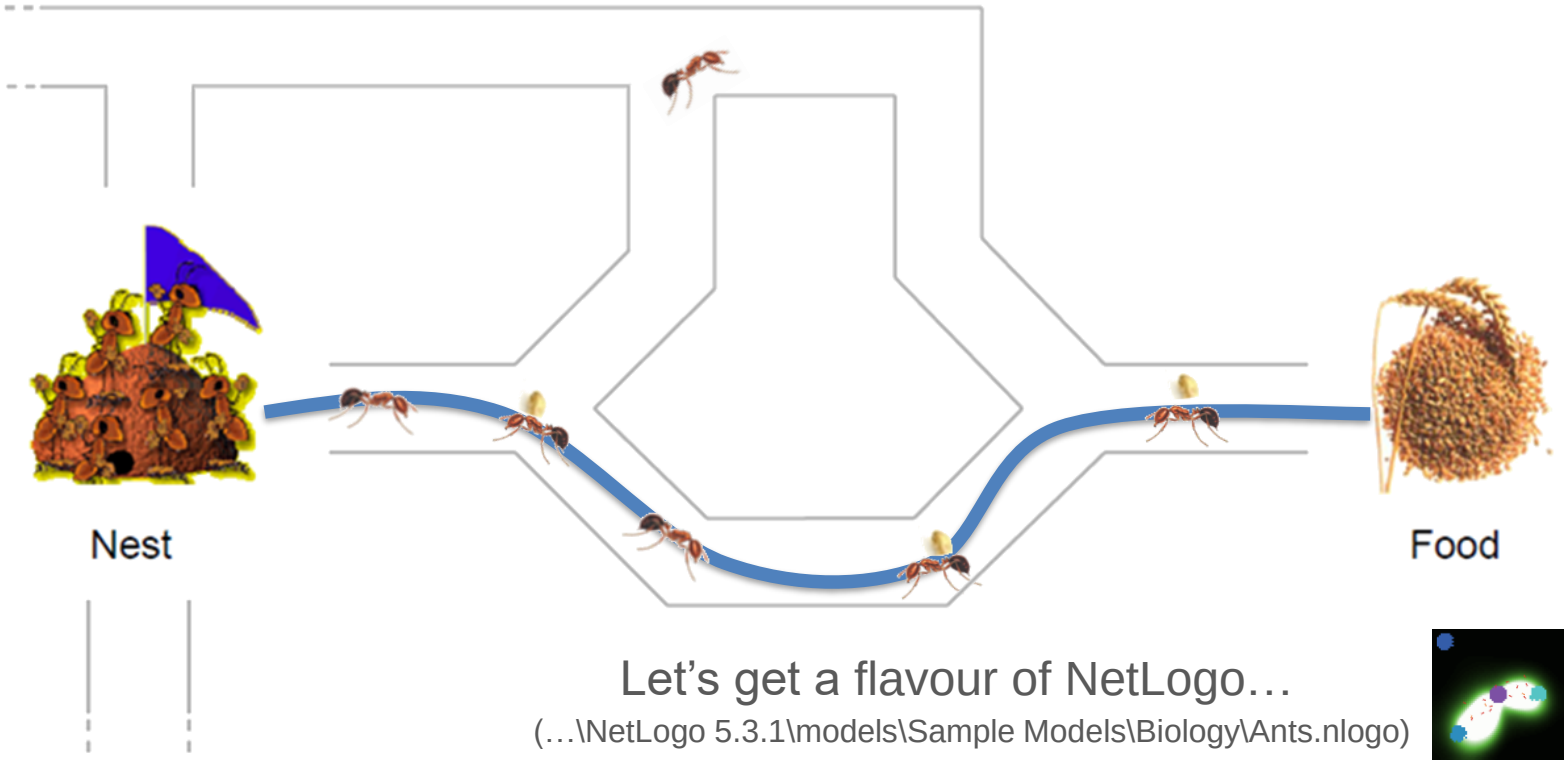
A simple example: Ants foraging food

- While wandering, ants deposit pheromone
  - They decide how much pheromone to deposit
  - Pheromone evaporates



A simple example: Ants foraging food

- Ants follow trails with high pheromone concentration
- Outcome: ants always prefer the shortest path!

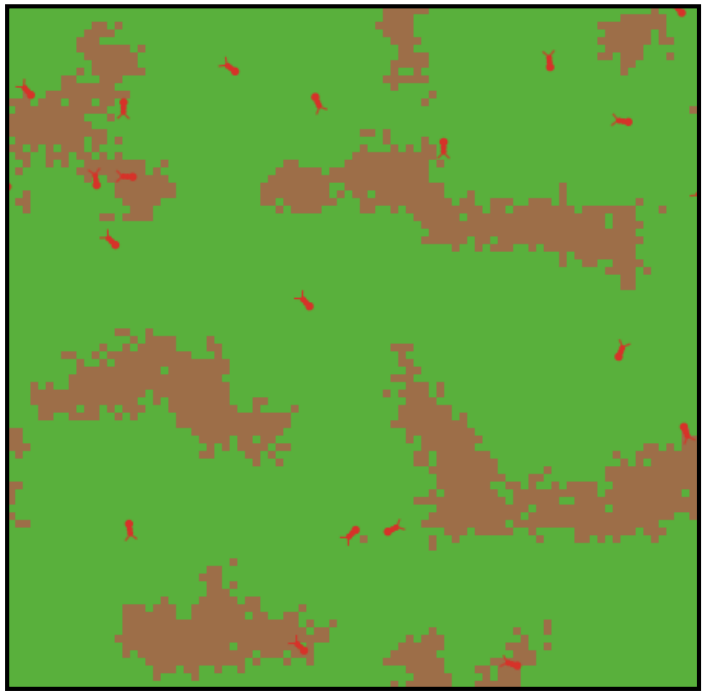


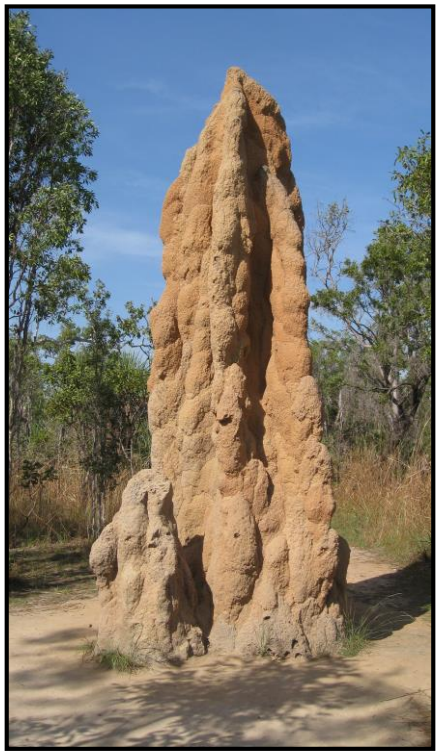
Let's get a flavour of NetLogo…

(…\NetLogo 5.3.1\models\Sample Models\Biology\Ants.nlogo)

Another example: termites clustering wood

- Termites can build enormous mounds
  - Elaborate structures made using a combination of soil, mud, and chewed wood/cellulose





Installation

- NetLogo download webpage
  (http://ccl.northwestern.edu/netlogo/download.shtml)
- Select the most recent version
  the recommended download version is 6.0.4
- Available for Windows,
  Mac OS X, and Linux
  - Simple installer
  - Linux users check FAQ
    (troubles with OpenJDK)

Short History

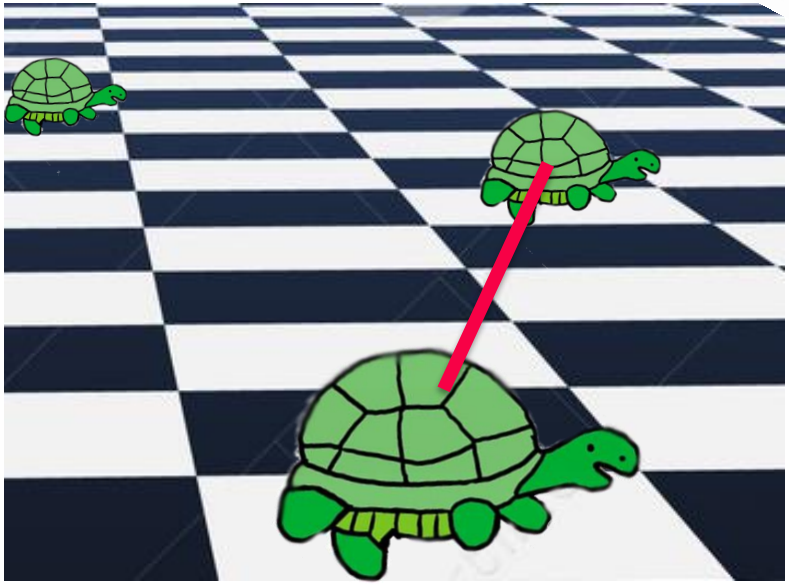
- LOGO (Papert & Minsky, 1967)
  - Simple language derived from LISP (1958)
  - The Children's Machine: Rethinking School in the Age of the Computer
  - Turtle graphics and exploration of "microworlds"

- StarLogo (Resnick, 1991)
  - Agent-based simulation language
  - Exploring decentralized systems through concurrent programming of turtles

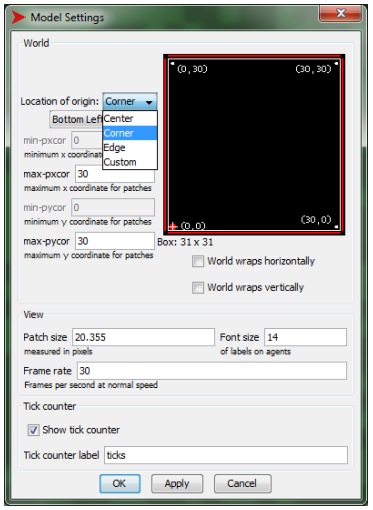
- NetLogo (Wilensky, 1999)
  - Extended StarLogo with many libraries
  - HubNet (participatory simulation tool)

Basic Concepts (Agents)

- NetLogo is a world made of four kinds of agents
  - Patches: make up the background or landscape
  - Turtles: move around on top of the patches
  - Observer: oversees everything going on in the world
  - Links: connection between two turtles

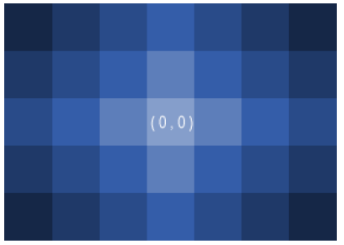


Basic Concepts (Agents)



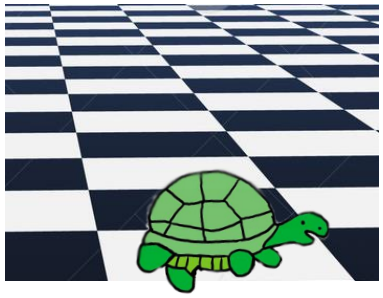
- NetLogo is a 2D grid of patches that have integer coordinates (*pxcor, pycor*)
  - Grid settings can be changed (size, location of origin, world wrap)
  - The origin (0,0) is typically the patch in the middle

- Patches
  - A patch corresponds to a pixel of the virtual world
  - Essentially, a static squared piece of ground over which turtles can move
  - Patches can contain n turtles and k state variables
  - Useful variables: *world-width, world-height*
    max-pxcor, max-pycor
    min-pxcor, min-pycor



Basic Concepts (Agents)

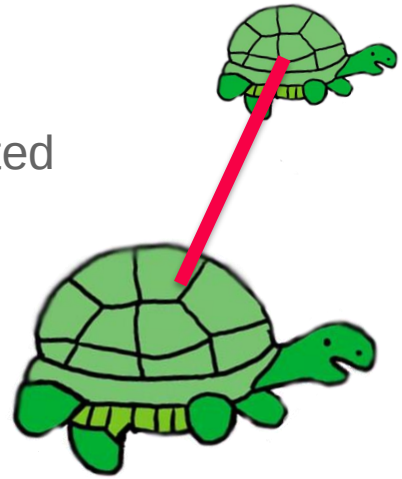
- Turtles
  - Turtles are agents that can move around in the virtual world (on top of the patches)
  - Turtles have decimal coordinates (*xcor*, *ycor*)
  - Turtles have an orientation (*heading*)
  - Each turtle has an unique ID (*who*)
  - Always shown in the center of a patch (but it can be at any point within the patch)
  - Turtles can die (*die*)
  - Turtles can have variables
  - Turtles have a shape (*shape*)
  - Turtles have a color (*color*)
  - Turtles have a size (*size*)



Basic Concepts (Agents)

- Observer
  - The “God” looking out over the virtual world
  - It can create new turtles and patches
  - It has read/write access to all agents and variables
  - It does not have a location in the virtual world

- Links
  - An agent connecting two turtles
  - Can be directed (towards a parent) or undirected
  - Drawn as a line between two turtles
  - They do not have a location



Basic Concepts

Commands and reporters tell agents what to do.

- A command is an action for an agent to carry out, resulting in some effect.
- A reporter is instructions for computing a value, which the agent then “reports” to whoever asked it.
- Primitives are build-in commands or reporters
- Procedures are user-made commands or reporters

Basic Concepts

- Primitives
  - NetLogo language keywords (built-in commands or reporters)
  - Can have a long form or an abbreviated form
  - Example:

    create-turtles 10 (crt 10)
    clear-all (ca)

- Commands
  - Instructions you can give to the agents
  - You can see it as a void function
  - Example:

    ask turtle 5 [fd 1]

ask turtle 5 [<commands>]

Basic Concepts (Procedures)

- Procedures
  - User-made commands or reporters
  - A procedure has a name and it is enclosed between the keywords *to* and *end*
  - Example:

```
to setup
    ca
    crt 10
    fd 1
end
```

- Procedures with inputs
  - Include a list of input names in square brackets after the procedure name
  - Example:

```
to draw-polygon [num-sides size]
    …
end
```

Basic Concepts (Procedures)

- Reporter procedures
  - Carry out some operations and report a value
  - *to-report* keyword instead of to at the beginning
  - *report* keyword (pretty much like return in C)
  - Example:

```
to-report abs [number]
        ifelse number >= 0
        [report number]
        [report 0 - number]
end
```

ifelse reporter [command1] [command2]
(If reporter reports true, runs command1, else command2)

- Procedures with local variables
  - *let* keyword to add a variable
  - Example:

```
to swap [a b]
        let temp a
        set a b
        set b temp
end
```

Basic Concepts (Procedures)

- Anonymous procedures are created with ->
  - [-> fd 1] creates an anonymous command
  - [[x] -> fd x] anonymous command with arguments
  - Example: ask turtle 5 [-> fd 1]

```
foreach mylist [ [ x ] -> print x ]
```

  - [[x] -> x] anonymous reporter with one input
  - [[x y] -> x + y] anonymous reporter with two input variables

Basic Concepts (Variables)

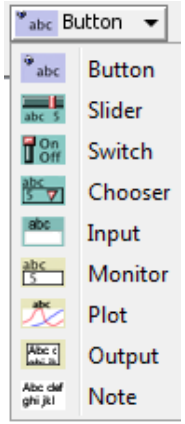
- Global variables
  - Can be read and set anytime by any agent
  - Example: *globals [clock]*
- Turtle & Patch variables
  - Each individual turtle or patch has its own value
  - Turtles can read and set patch variables on which they stand on
  - Example: *turtles-own [energy]*
  - Example: *patches-own [friction]*
- Local variables
  - Defined and accessible only inside a procedure
  - Local scope (the procedure itself or the narrowest square brackets)
  - Example: let localvar

Basic Concepts (Variables)

- Built-in NetLogo variables
  - Built-in turtle variables (color, xcor, ycor, heading, …)
  - Built-in patch variables (pcolor, pxcor, pycor, …)
- How to set variables?
  - *set* command *[set variable value]*
  - Default value is zero!
  - Also used to modify local variables
    (the *let* command is only used to declare them)

```
to swap [a b]
        let temp a
        set a b
        set b temp
end
```

Basic Concepts



- The *ask* command
  - Specifies commands to be run by (all) turtles, patches, links

    ask <agents> [<commands>]
  - Observer code (e.g., *crt 100*) cannot run inside *ask* blocks
  - Can be factored out in buttons
- Examples
  - *Setting the color of all turtles: ask turtles [set color red]*
  - *Setting the color of a single turtle: ask turtle 5 [set color red]*
  - *Setting the color of all patches: ask patches [set pcolor red]*
  - *Setting the color of a single patch: ask patch 2 3 [set pcolor red]*
  - *Setting the color of the patches under the turtles:*
    ask turtles [set pcolor red]
  - *Print the ID of all turtles: ask turtles [print who]*

Basic Concepts

- Examples
  - Setting the color for a specific set of patches:
    ask patches [if(pxcor > 0) [set pcolor green]]
  - Setting the patch to the east of the first turtle to become red:
    ask turtle 0 [ask patch-at 1 0 [set pcolor red]]
  - *patch-at dx dy*: reports the patch at (dx, dy) from the calling agent, i.e., the patch at dx patches east and dy patches north

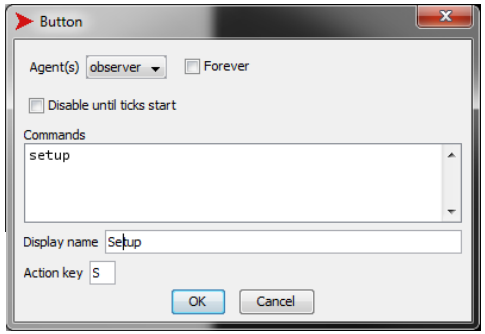
Programming Examples

- Color randomly all the patches in the virtual world
  - User-made command called setup
  - All patches set their color to a random value

```
to setup
    ask patches [
        set pcolor random 110
    ]
end
```

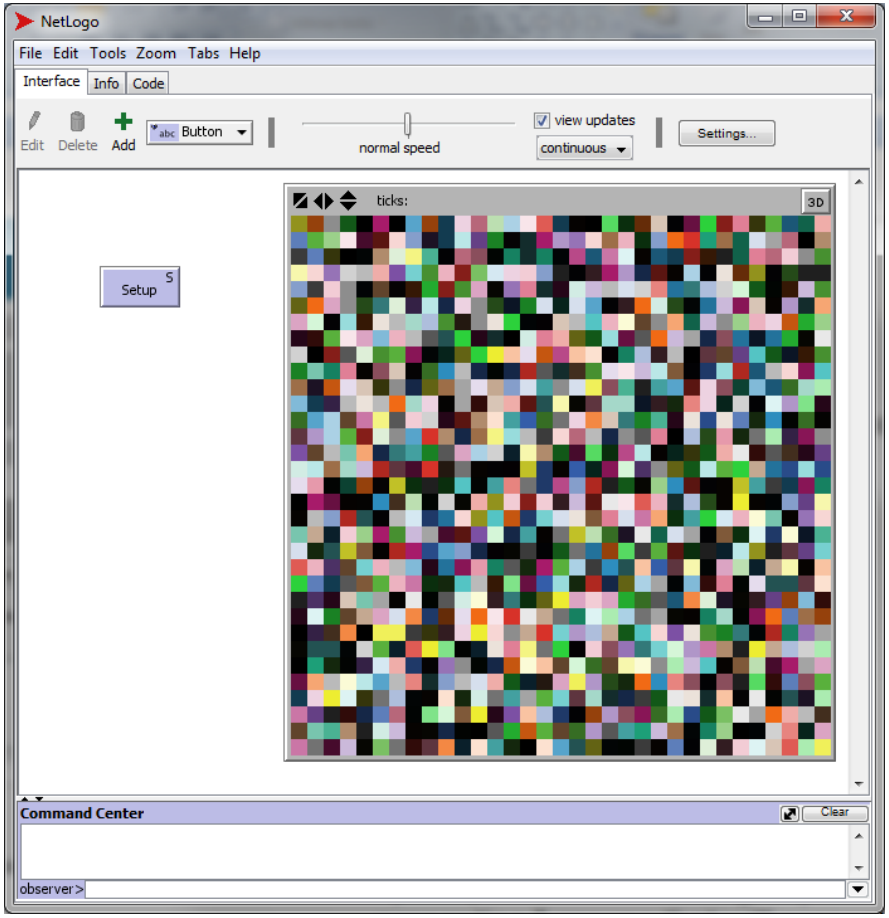
  - We can let the observer run this code by typing it manually

  - We can factor this code into a button →



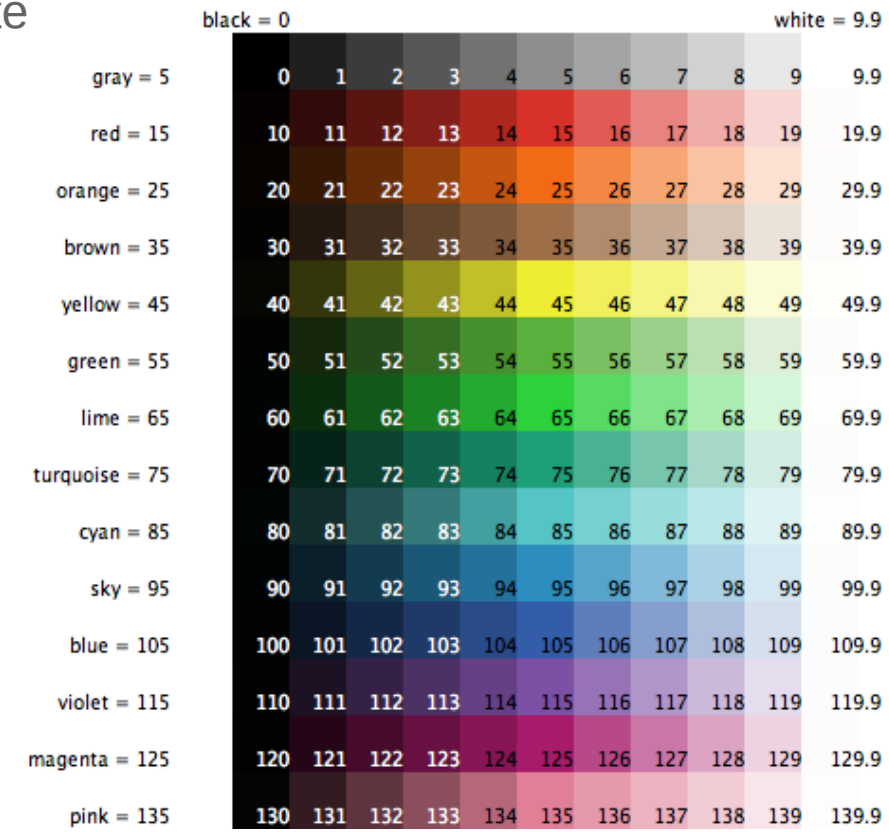
Programming Examples

- Color randomly all the patches in the virtual world



Basic Concepts

- Colors in NetLogo
  - Numeric value in the range 0-139.9
  - 0 is black, 9.9 is pure white



Tip:

wrap-color number: procedure that wraps any number to the color range 0-140 by repeatedly adding or subtracting 140 from the given number until it is in the 0-140 range

show wrap-color 150 → 10

show wrap-color - 10 → 130

Basic Concepts

- Agentsets
  - Non-ordered set of agents containing either turtles, patches, or links
  - All agents in an agentset must be of the same type
  - Used by *ask* or as an input for a reporter (e.g., *ask <agentset> […]*)
  - Subset with *<agentset> with [<condition>]*
    - Example*: ask turtles with [xpos > 10] [set color 10]*
- Commands for agentsets (i)
  - Check if every agent in the set satisfies a given condition:
    if all? *<agentset> [<condition>] [<commands>]*
    - Example: *if all? turtles [color = red] [print "every turtle is red!"]*
  - Check if an agent is member of an agentset: *member?*
    - Example: *show member? turtle 0 turtles → true*
  - Check if the agentset is empty: *any? <agentset>*
    - Example: *ifelse any? turtles with [color = red] [print "at least one red turtle!"] [print "no red turtle"]*

Basic Concepts

- Command for agentsets (ii)
  - Check if two sets are equal: = or *!=*
  - Number of agents in a set: *count*
  - Random agent from an agentset: *one-of*
  - A random agent that has the highest/lowest value for a reporter: *max-one-of* *< agentset > [<reporter>]*
    - Example: *ask (max-one-of turtles [wealth]) [donate]*
  - All red turtles: *turtles with [color = red]*
  - All red turtles on the patch of the current caller: *turtles-here with [color = red]*
  - All agents having the highest/lowest value for a reporter: *with-max*
  - Manually specifying elements: *turtle-set turtle 0 turtle 2 turtle 9*
  - All turtles less than 3 patches away: *turtles in-radius 3*
  - Contiguous patches: *neighbors4* or *neighbors* (all 9 patches)

Basic Concepts

- Lists (i)
  - Pieces of information in a single variable
  - Each value in a list can be anything: a number, a string, an agent, an agentset, or another list
  - Create a list:

    let my-list [2 4 6 8]
  - Create a list from a reporter:

    let my-random-list (list (random 10) (random 20)) → [4 19]
  - Create a list from agentset (e.g., a list that contains the colors of each turtle, in random order):

    let color-list (list [color] of turtles)

Basic Concepts

- Lists (ii)
  - Create lists with the construct *n-values size <reporter>*

list n-values 5 [1]
(creates the list [1 1 1 1 1])

list n-values 5 [[x] -> x]
(creates the list [0 1 2 3 4])

list n-values 5 [[x] -> x x]*
(creates the list [0 1 4 9 16])

list n-values 3 [[x] -> turtle x]
(creates the list [(turtle 0) (turtle 1) (turtle 2)])

  - Create lists with the construct *sentence value_1 value_2 … value_n*

list sentence [1 2] 3
(creates the list [1 2 3])

list (sentence [1 2] 3 [4 5] (3 + 3) 7)
(creates the list [1 2 3 4 5 6 7])

Basic Concepts

- Lists (iii)
  - Add elements to the list at specific positions

let mylist [5 7 10]
set mylist fput 2 mylist
(creates the list [2 5 7 10])

let mylist [2 7 10 "Bob"]
set mylist lput 42 mylist
(creates the list [2 7 10 "Bob" 42])

  - Replace the second item with 6 so that the list becomes [2 6 6 8]:
    set my-list replace-item 1 my-list 6
  - Insert item at index 3 so that the list becomes [2 6 7 6 8]:

    set my-list insert-item 2 my-list 7

Basic Concepts

- Lists (iv)
  - Eliminate elements to the list at specific positions

let mylist [2 4 6 5]
set mylist but-first mylist
(creates the list [4 6 5])

let mylist [2 4 6 5]
set mylist but-last mylist
(creates the list [2 4 6])

show but-first "string"
(prints "tring")

  - Arithmetic expressions on lists

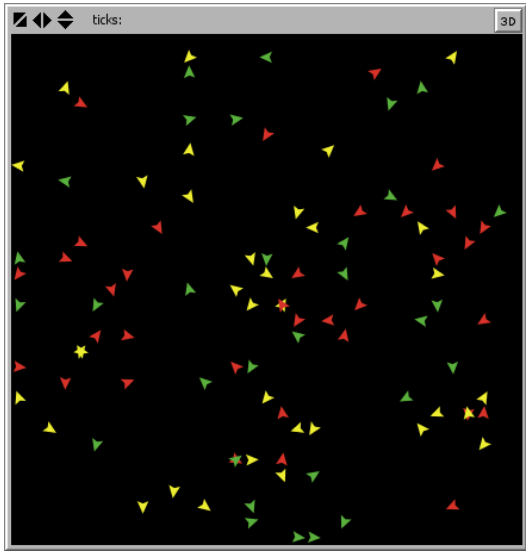
show min [xcor] of turtles
(prints the x coordinate of the turtle which is farthest left in the world)

show mean [xcor] of turtles
(prints the average of all the turtles' x coordinates)

Programming Examples

- Spread randomly 100 turtles over the world
  - Select their color between red, yellow, and green
  - Embed the code in a user button called *setup*

```
to setup
    ca
    crt 100 ; create 100 turtles
    ask turtles [
      set xcor random max-pxcor
      set ycor random max-pycor
      set color one-of [red yellow green]
    ]
end
```

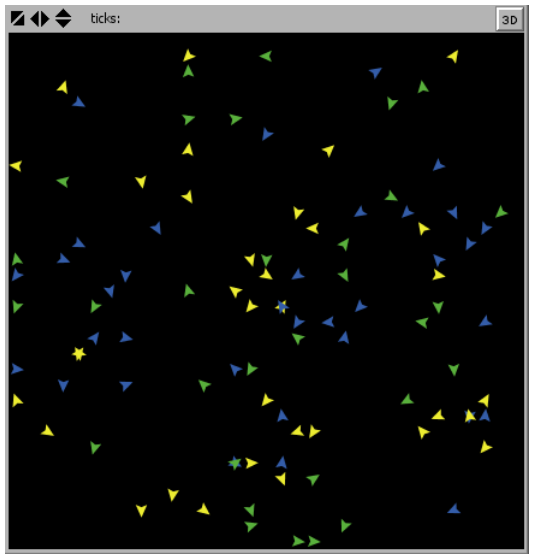



```
crt 100 ; this is a comment
```

Programming Examples

- Change the color of the red turtles to blue
  - Use another user button *color-blue* to do that
  - Nothing else should change

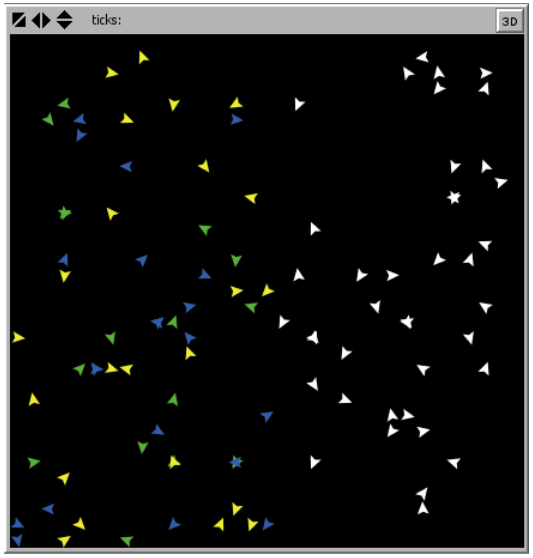
```
to color-blue
      ask turtles with [ color = red ] [
        set color blue
      ]
end
```



Programming Examples

- Change the color of the turtles on the right side of the screen to white
  - Use another user button *color-white* to do that
  - Nothing else should change

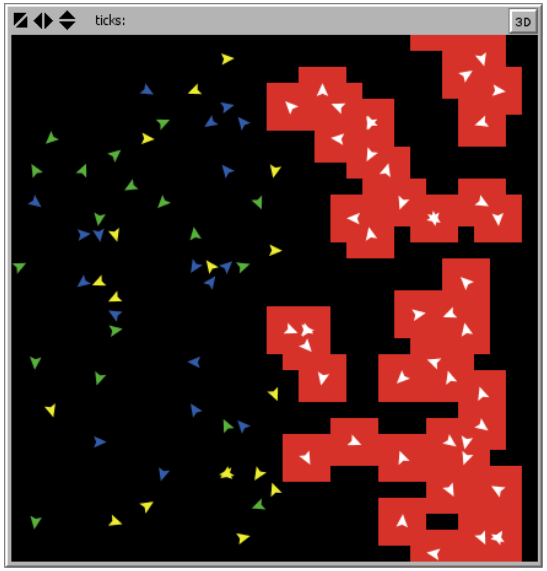
```
to color-white
      ask turtles with [ xcor > (max-pxcor / 2) ] [
            set color white
       ]
end
```



Programming Examples

- Color the adjacent patches to the white turtles of red
  - Extend the previous user button *color-white*
  - Color of red also the patch in which the white turtles are

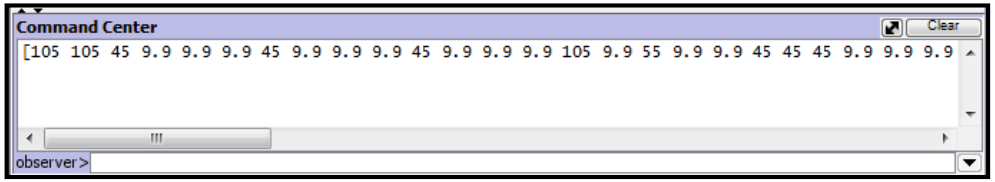
```
to color-white
    ask turtles with [ xcor > (max-pxcor / 2) ] [
        set color white
        set pcolor red
        ask neighbors [
            set pcolor red
        ]
    ]
end
```



Programming Examples

- Color the adjacent patches to the white turtles of red
  - Extend the previous user button *color-white*
  - Color of red also the patch in which the white turtles are

```
to color-white
    ask turtles with [ xcor > (max-pxcor / 2) ] [
        set color white
        set pcolor red
        ask neighbors [
            set pcolor red
        ]
    ]
    let color-list ([color] of turtles)
    print color-list
end
```



Basic Concepts

- Iterators
  - *foreach [2 4 6] [*

    [x] ->

    crt x ; Use x to refer to the current item of list

    show(word "created " x " turtles")

    ]

    (created 2 turtles, created 4 turtles, created 6 turtles)

  - *loop [<commands>]* construct: the commands run forever until the current procedure exits through the use of *stop* or *report*

  - *map [<reporter>] list* construct (reporter runs for each item in the list)

    show map [[x] -> round x] [1.1 2.2 2.7]

    (creates the list [1 2 3])

    show (map [[x y] -> x + y] [1 2 3] [2 4 6])

    (creates the list [3 6 9])

Basic Concepts

- Iterators
  - *repeat* construct

    repeat number [<commands>]

    repeat 36 [fd 1] ; fd = forward = move the turtle by 1 step

    repeat 36 [fd -1] ; negative fd = moves backwards

  - *while [<reporter>] [<commands>]* construct

    while [any? other turtles-here] [fd 1]

    (turtle moves until it finds a patch that has no other turtles on it)

Basic Concepts

- Nested loops
  - If you have nested loops, the *stop* command will also break the outer loop (i.e., to be safe it is better to use the *while* iterator)

```
to test-loop
  print "Repeat loop:"
  let cnt 0
  repeat 3 [
    loop [
      set cnt (cnt + 1)
      print (cnt)
      if (cnt >= 5) [
        stop
      ]
    ]
  ]
end
```

```
observer> test-loop
Repeat loop:
1
2
3
4
5
```

```
to test-while
  print "Repeat while:"
  repeat 3 [
    let cnt 0
    while [cnt <= 5] [
      set cnt (cnt + 1)
      print (cnt)
    ]
  ]
end
```

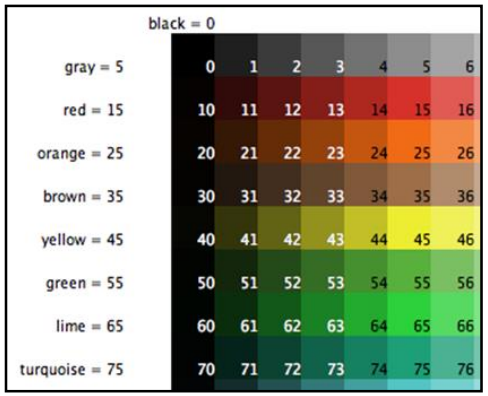
```
observer> test-while
Repeat while:
1
2
3
4
5
6
1
2
3
4
5
6
1
2
3
4
5
6
```

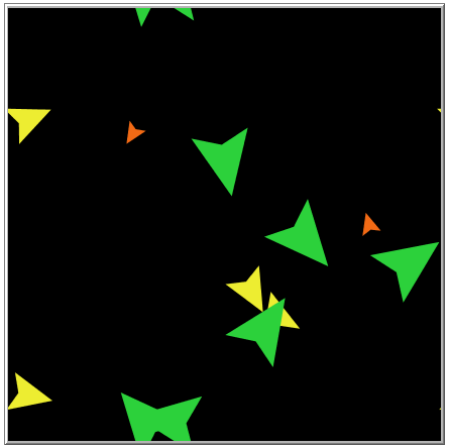
http://stackoverflow.com/questions/20274193/netlogo-nested-loops

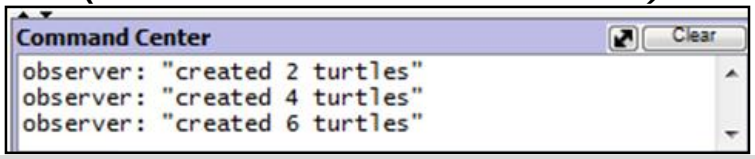
Programming Examples

- Using iterators to create and color turtles
  - Create 2 orange turtles, 4 yellow turtles, and 6 green turtles
  - Using the *foreach* construct

```
to setup
      ca
      foreach [2 4 6] [
       [ x ] ->
        crt x  [
           set size x
           set color (5 + (10 * x))
           set xcor random max-pxcor
           set ycor random max-pycor
         ]
         show(word "created " x " turtles")
      ]
end
```

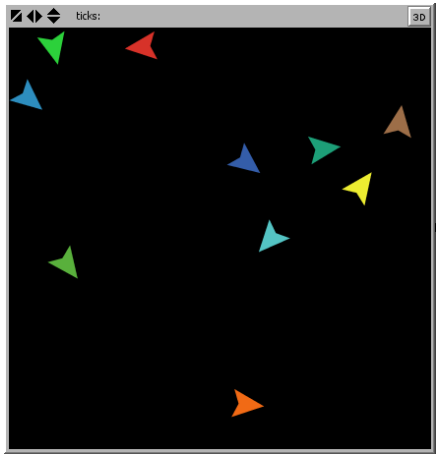






Programming Examples

- Using iterators to create and color turtles
  - Create 10 turtles, each of a different color
  - Using the *loop* construct



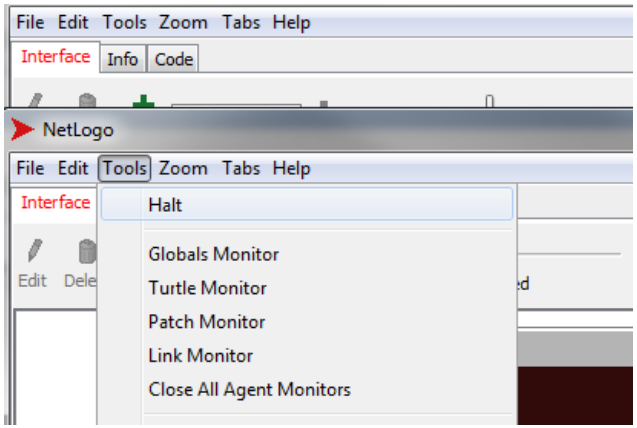

```
to setup
    ca
    let temp 1
    loop [
      crt 1  [
        set size 3
        set color (5 + (10 * temp))
        set xcor random max-pxcor
        set ycor random max-pycor
      ]

      if temp > 10 [ stop ]
       show(word "created " temp " turtles")
    ]
end
```

What is the outcome of this code?

Programming Examples

- Using iterators to create and color turtles
  - Create 10 turtles, each of a different color
  - Using the *loop* construct

```
to setup
      ca
      let temp 1
      loop [
        crt 1  [
          set size 3
          set color (5 + (10 * temp))
          set xcor random max-pxcor
          set ycor random max-pycor
        ]
        set temp (temp + 1) ; if you forget this →
        if temp > 10 [ stop ]
        show(word "created " temp " turtles")
      ]
end
```



Basic Concepts

- Strings
  - Enclosed between double quotes (" … ")
  - Comparable using =, !=, <, >, <=, and >= operators
  - Can be concatenated with the + operator
  - Special characters: \n for newline, \t for tab, \" for double quote, \\ for backslash
  - Length can be retrieved using the command *length* (length "string" will return 6)
  - Several nice functions:
    - *first* "string" *returns "s", last* "string" *returns "g"*
    - *item 2* "string" *returns "r"*
    - *empty?* "" *returns true, empty?* "string" *returns false*
    - *butfirst* "string" *returns "tring", butlast* "string" *returns "strin"*
    - *member?* "rin" "string" *returns true, position* "rin" "string" *returns 2*
    - *remove* "s" "strings" *returns "tring"*
    - *reverse* "string" *returns "gnirts"*

Basic Concepts

- Breeds
  - Specific species of turtles
  - Also used as specific kind of agentsets

    breed [*<breeds> <breed>]*
    breed [cats cat]
    breed [dogs dog]
  - Specific breed-commands

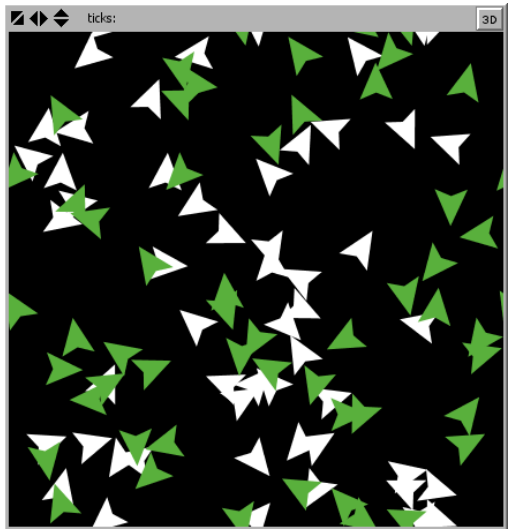
    create-<breed> (create-dogs)
    <breed>-here (Reports all the agents on a patch)
    <breed>-at
  - Breeds are variables and can be changed anytime by a turtle agent

    ask turtle 5 [set breed sheep]

Programming Examples

- Using breeds
  - Create two type of turtles: mice and frogs

```
breed [mice mouse]
breed [frogs frog]

to setup
  clear-all
  create-mice 50
  ask mice [
    set size 3
    set color white
    set xcor random max-pxcor
    set ycor random max-pycor
  ]
  create-frogs 50
  ask frogs [
    set size 3
    set color green
    set xcor random max-pxcor
    set ycor random max-pycor
  ]
end
```

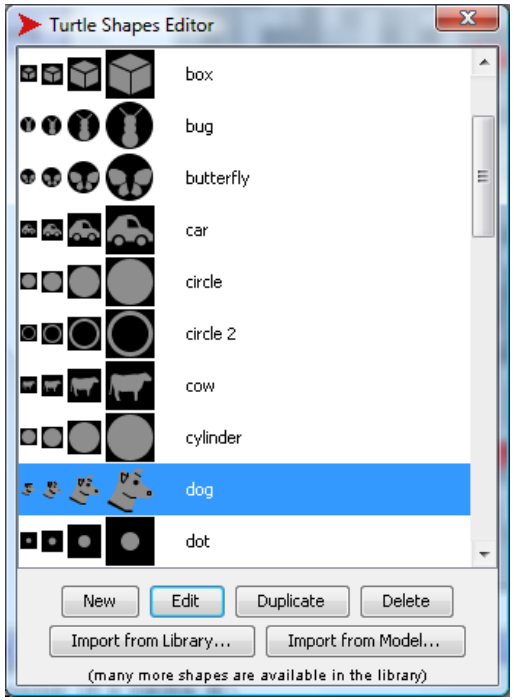



```
show mouse 1 ; prints (mouse 1)
show frog 51 ; prints (frog 51)
show turtle 51 ; prints (frog 51)
```

*Compared to print, show also displays the agent *before the value*

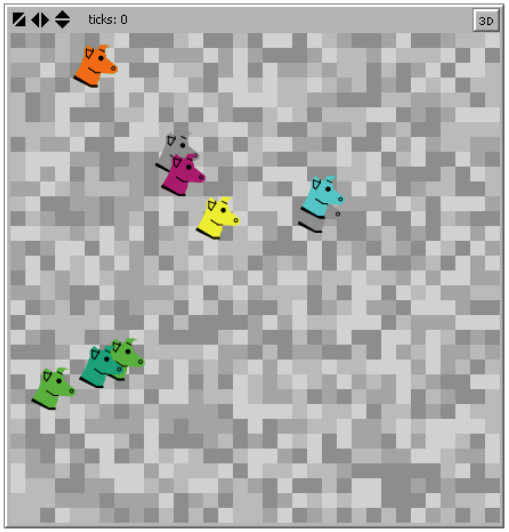
Programming Examples

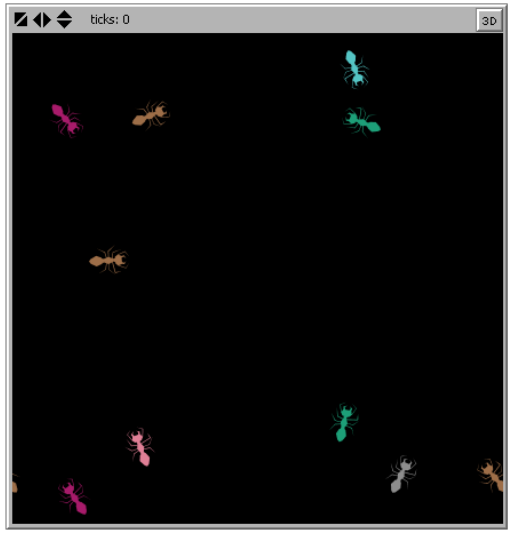
- Changing turtles
  - Many shapes available under Tools\Turtle Shapes Editor
  - Import from library many more shapes
  - Shapes can be changed directly in the code



set shape "dog"

set shape "ant"





Synchronization

- Turtle commands are executed asynchronously
  - Each turtle does its list of commands as fast as it can
  - However, at the end of an *ask* block, the turtles wait until all are finished before proceeding
- Example: these two steps are NOT synchronized

```
ask turtles [
    fd random 10
    do-calculation
]
```

- Example: these two steps are synchronized

```
ask turtles  [ fd random 10 ]
ask turtles  [ do-calculation ]
```

Elapsing time

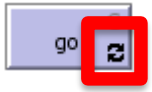
- Time elapses in discrete steps called *ticks*
  - NetLogo has a built-in tick counter, so one can keep track of how many ticks have elapsed
  - The *tick* command advances the tick counter by 1
  - *clear-all* and *reset-ticks* commands reset the tick counter to 0

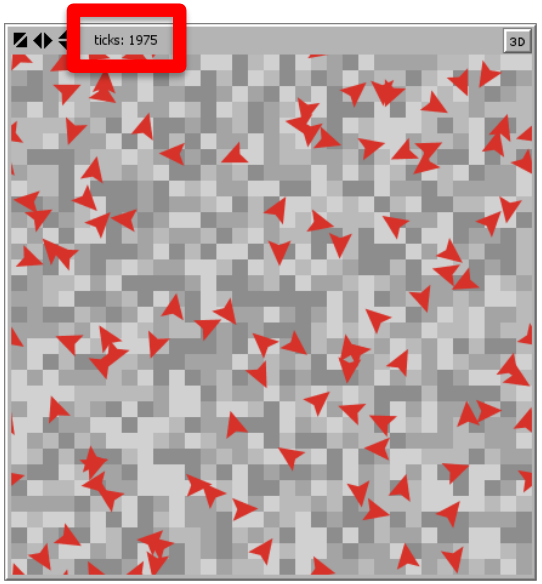
Programming Examples

- Continuous movements
  - Every turtle moves of one unit every tick on a random direction

```
to setup
  ca
  crt 100
  reset-ticks
  ask turtles [
    set color red
    set size 2
  ]
  ask patches [
    set pcolor (5 + random 4)
  ]
end
```

```
to walk_around
  fd 1 ; forward
  rt random 50 ; right
  lt random 50 ; left
end

to go
  ask turtles [
     walk_around
  ]
  tick
end
```



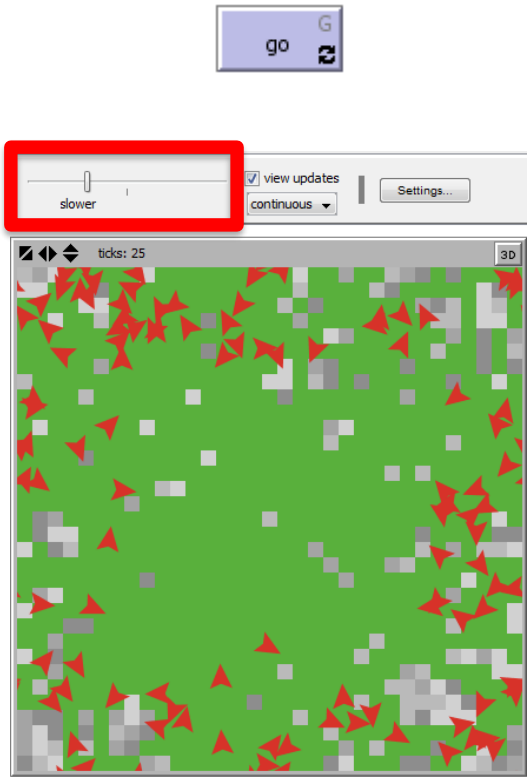


Programming Examples

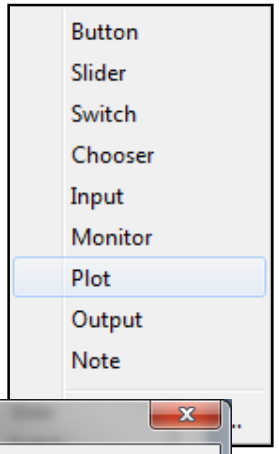
- Continuous movements
  - Every turtle colors of green the patch where it sits on

```
to setup
  ca
  crt 100
  reset-ticks
  ask turtles [
    set color red
    set size 2
  ]
  ask patches [
    set pcolor (5 + random 4)
  ]
end
```

```
to walk_around
  fd 1 ; forward
  rt random 50 ; right
  lt random 50 ; left
end

to go
  ask turtles [
     walk_around
     set pcolor green
  ]
  tick
end
```

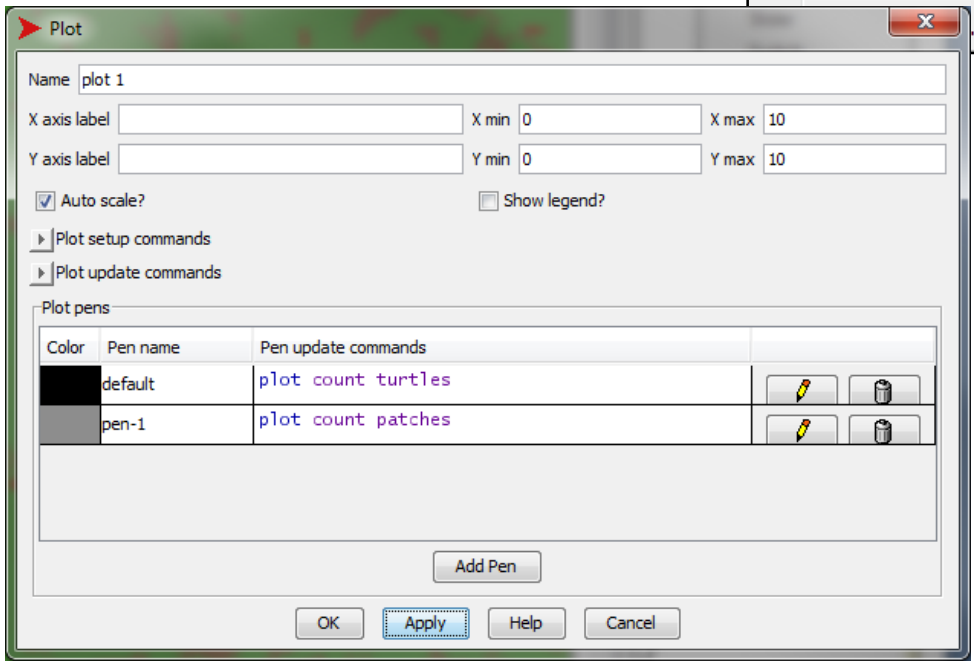


Drawing Plots



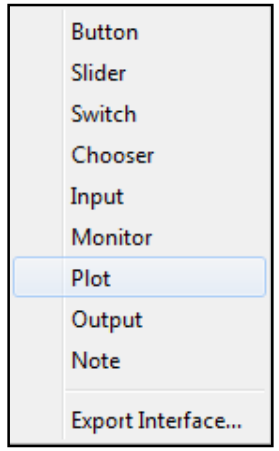
- NetLogo can display plots during execution
  - Select “Plot” when right-clicking with the mouse
  - Specify the rule to be shown in the plot

```
to setup

  …
  reset-ticks
end

to go
  ask turtles [
    …
    tick
  ]
end
```



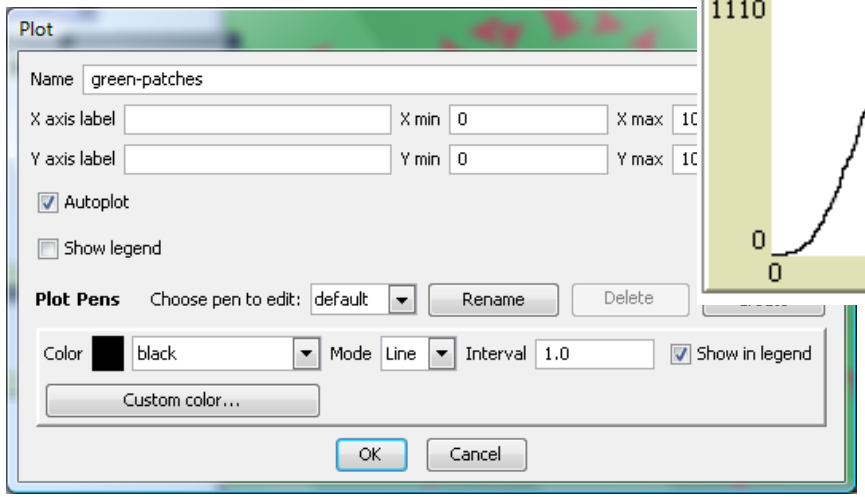

```
plot count patches with ; count reports the number of agents in an agentset
    [ count neighbors with [ pcolor = green] = 8 ]
```

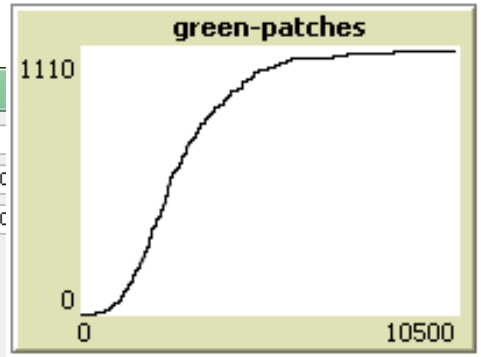
Drawing Plots



- NetLogo can display plots during execution
  - Alternative using old syntax

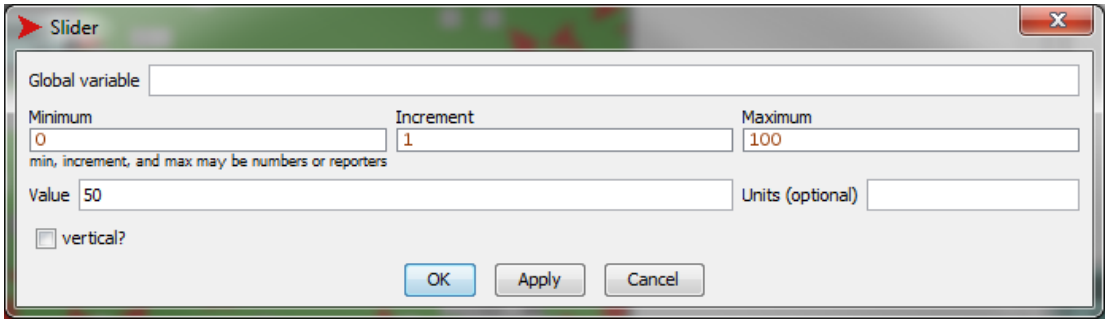
```
to draw-plot
  set-current-plot "green-patches"
  plot count patches with ; count reports the number of agents in a given agentset
     [ count neighbors with [ pcolor = green] = 8 ]
end

to go
  ask turtles [
     …
     draw-plot
]
end
```



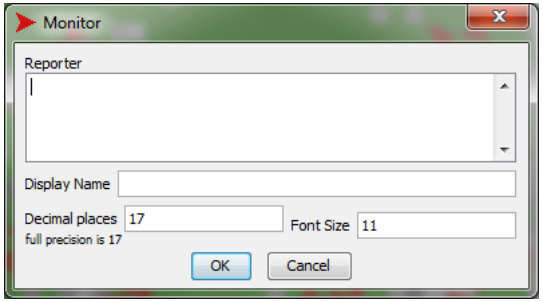


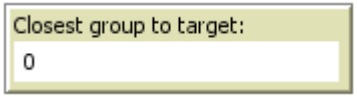
Sliders and Monitors

- Sliders have an adjustable range of numeric values
  - Can be used to let the user control the value of a variable



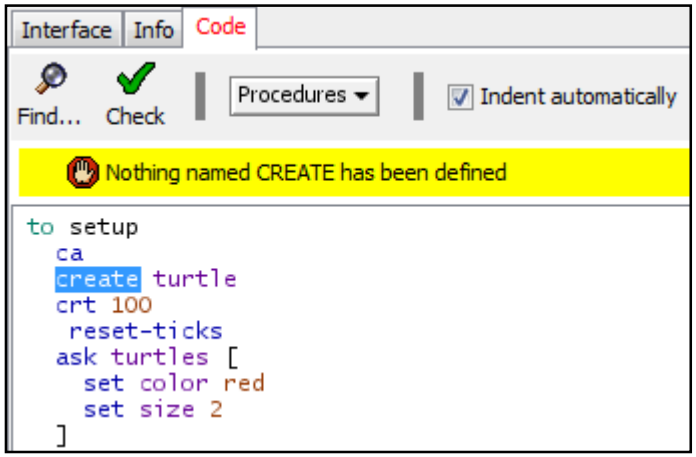
- Monitors show the results of a reporter
  - Can be used to monitor the value of a variable



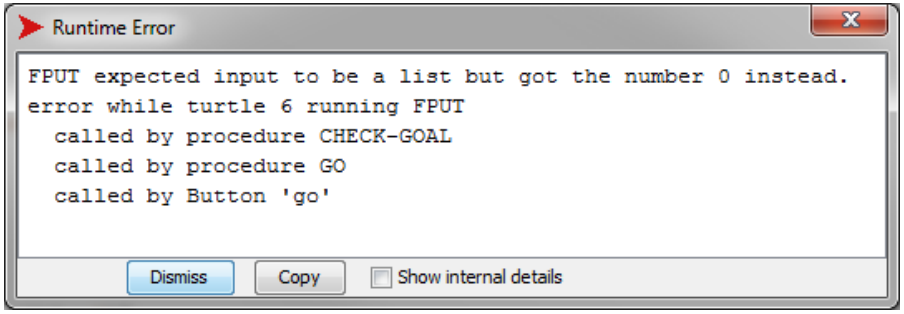


Errors

- Program can be compiled with the check button
  - Most syntactic errors will be highlighted there

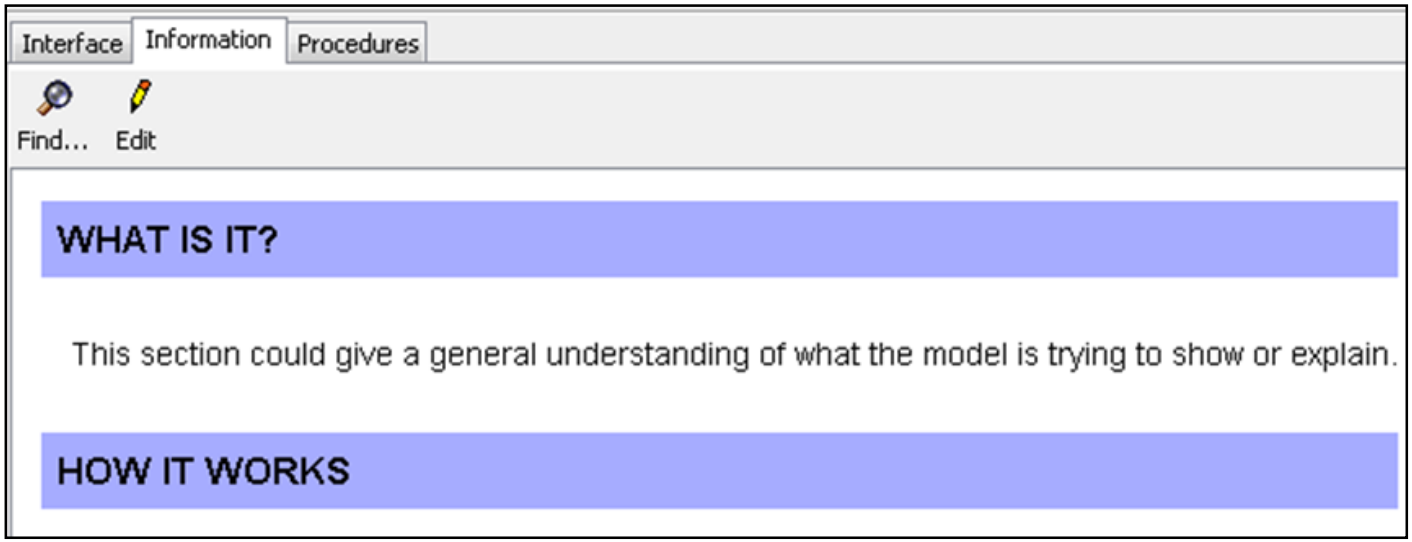


- Runtime errors can occur during program execution



Information Tab

- Make your model self-explanatory
  - Always comment your code!
  - NetLogo models are typically exported as Java applets, so it is important to give a proper description for first-time users!



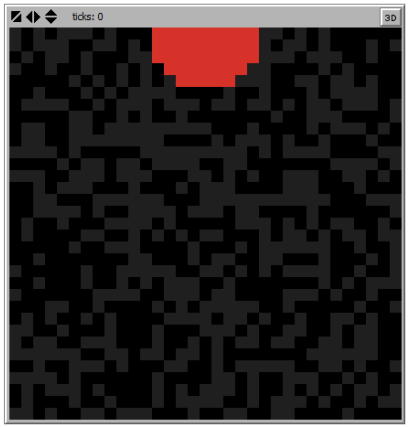
Documentation

- Extensive number of tutorial and examples on the Web
  - Programming is not intuitive, but easy, just dedicate some time!
  - Complete manual with a full reference of existing commands: http://ccl.northwestern.edu/netlogo/docs/NetLogo%20User%20Manual.pdf

- These slides were inspired by:
  - http://ccl.northwestern.edu/netlogo/resources/NetLogo-4-0-QuickGuide.pdf
  - http://library.iscpif.fr/files/CSSS2011_Rene_Doursat-A_Tour_of_Complex_Systems_5.pdf

Wrapping-up: ants returning back to the nest

- Ants are short-sighted
  - We hypothesize that ants are not able to see their nest from far-away, and will only see it if they end-up in the right patch




```
patches-own [
  nest?
]
```

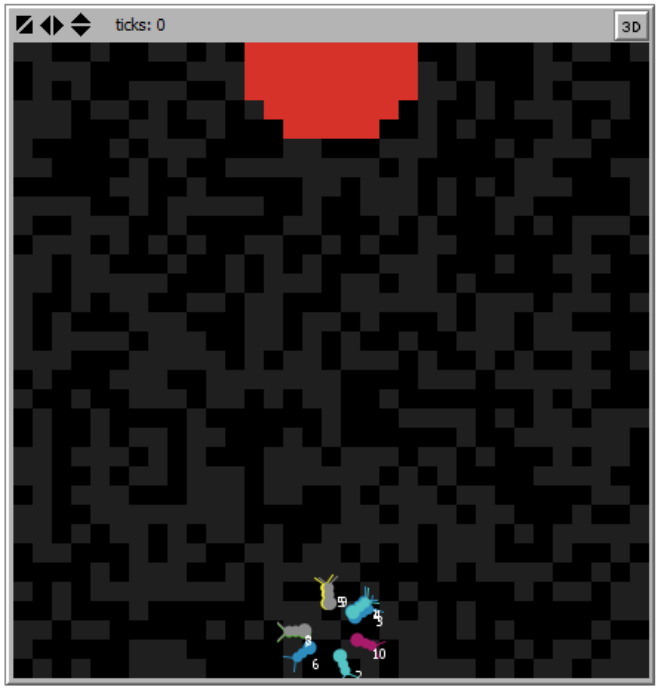
```
to setup
  ca
  ; Color patches of (more or less) the same color
  ask patches [
    set pcolor (0 + random 2)
  ]
  ; Create nest
  ask patches with [(distancexy (max-pxcor / 2) max-pycor) < 5] [
    set pcolor red
    set nest? true
  ]
  ; Create turtles
  setup-turtles
  ; Stop the ticks
  reset-ticks
  print "--------------------"
  print (word "List of the ants that returned to the nest:")
end
```

Wrapping-up: ants returning back to the nest

- Ants are short-sighted
  - We hypothesize that ants are not able to see their nest from far-away, and will only see it if they end-up in the right patch


```
to setup-turtles
  ; Create a number of turtles (can also be done with crt)
  create-turtles number-of-ants [
    set xcor (max-pxcor / 2)
    set ycor 2
    ]

  ask turtles [
    rt random 180
    lt random 180
    fd 2
    set shape "bug"
    set size 2
    set in-nest? false
    set label (who + 1);
  ]
end
```

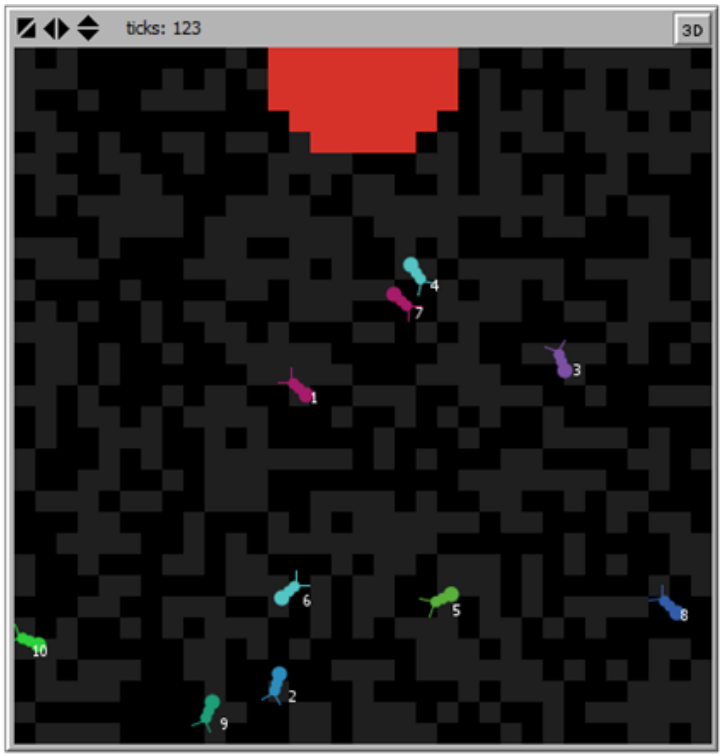
```
turtles-own [
  in-nest?
]
```



Wrapping-up: ants returning back to the nest

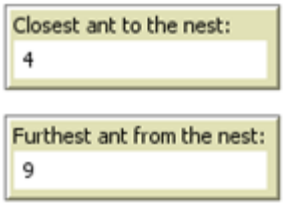
- Ants are short-sighted
  - We hypothesize that ants are not able to see their nest from far-away, and will only see it if they end-up in the right patch


```
to go
   if count turtles > 0 [
      move-turtles
      tick
      check-goal
   ]
end

to move-turtles
  ask turtles with[in-nest? = false] [
    fd 1
    rt random 180
    lt random 180
    set label (who + 1)
  ]
end
```



Wrapping-up: ants returning back to the nest

- Ants are short-sighted
  - We hypothesize that ants are not able to see their nest from far-away, and will only see it if they end-up in the right patch




```
to check-goal
  ask patch (max-pxcor / 2) max-pycor [
    ; Check which is the closest ant to the nest
    let closest min-one-of other turtles with [in-nest? = false] [distance myself]
    ask closest [
        set closest-ant-to-nest (who + 1)
    ]
    ; Check which is the furthest-away ant from nest
    let furthest max-one-of other turtles with [in-nest? = false] [distance myself]
    ask furthest [
        set furthest-ant-from-nest (who + 1)
    ]
  ]

…
```

```
globals [
  closest-ant-to-nest
  furthest-ant-from-nest
]
```

Wrapping-up: ants returning back to the nest

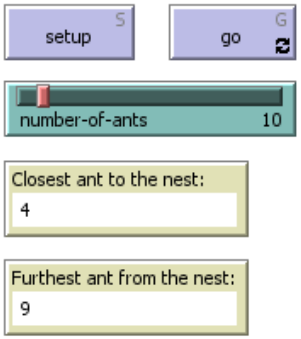
- Ants are short-sighted
  - We hypothesize that ants are not able to see their nest from far-away, and will only see it if they end-up in the right patch

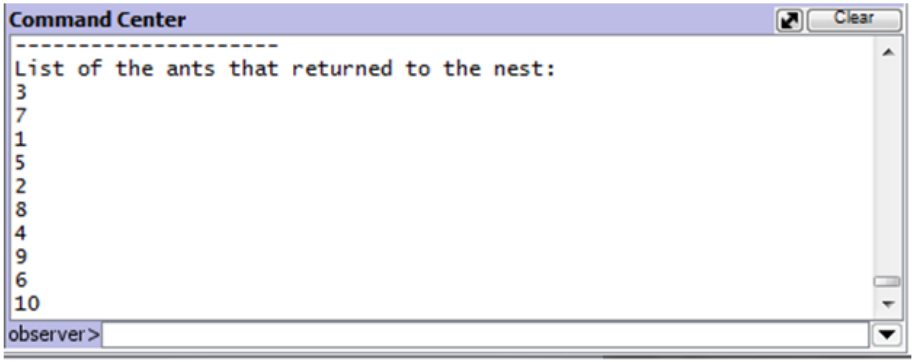
```
to check-goal

  …

  ; Check if the ant reached the nest
  ask turtles with [nest? = true and in-nest? = false] [
    set in-nest? true
    set xcor (max-pxcor / 2)
    set ycor max-pycor
    print (who + 1)
    die
  ]
end
```

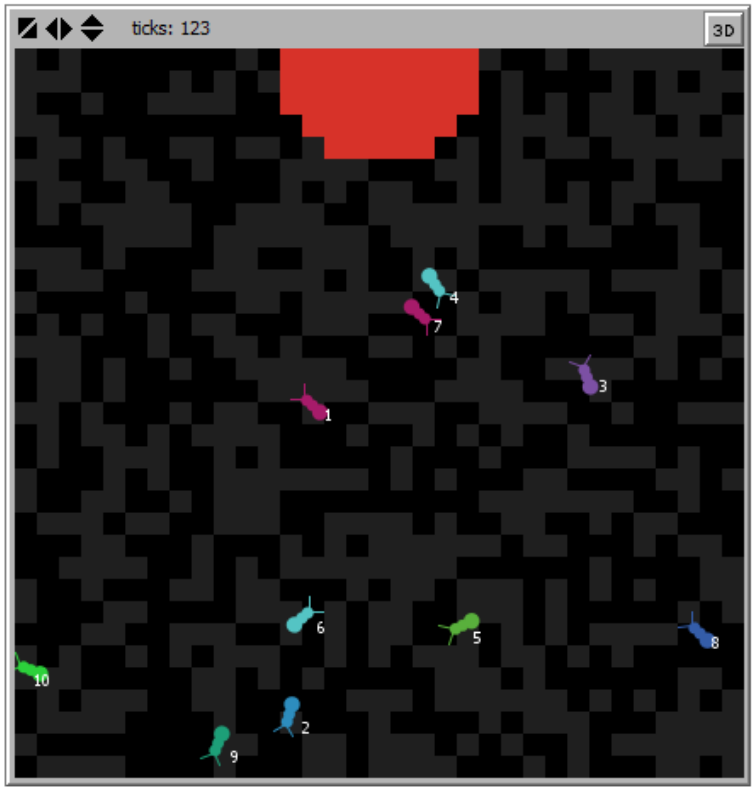
```
globals [
  closest-ant-to-nest
  furthest-ant-from-nest
]
```

Wrapping-up: ants returning back to the nest

- Ants are short-sighted
  - We hypothesize that ants are not able to see their nest from far-away, and will only see it if they end-up in the right patch

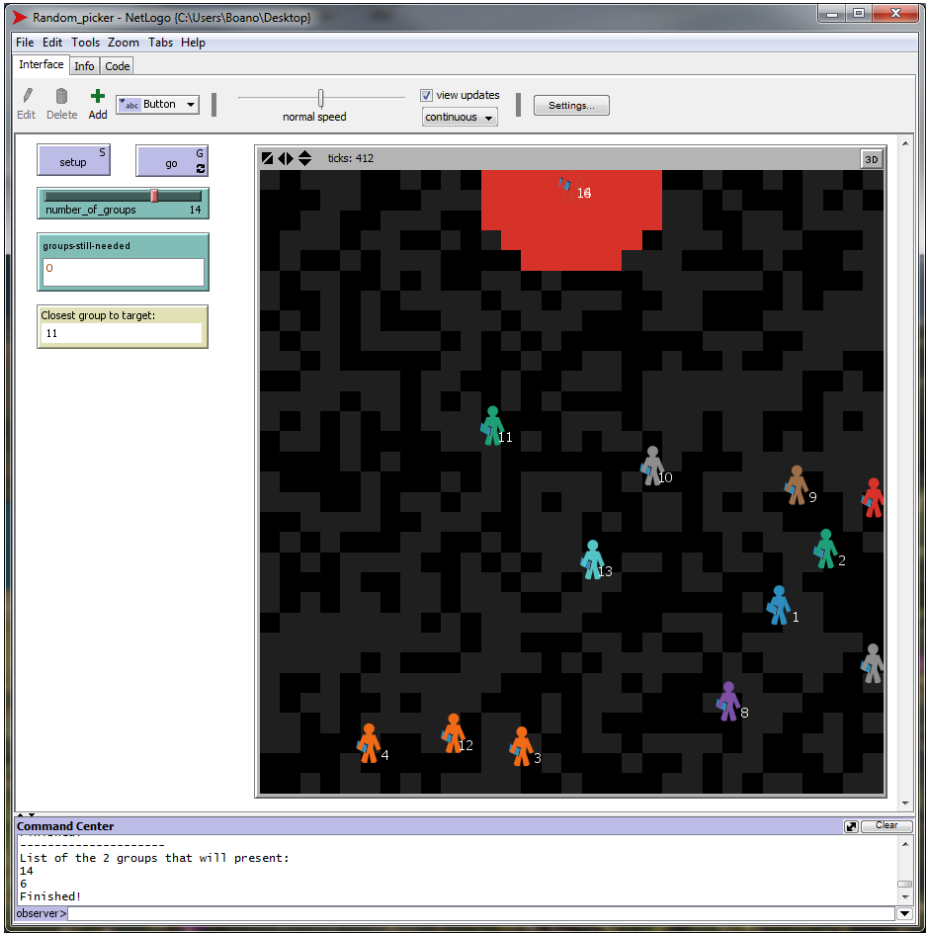






Now imagine ants to be students…

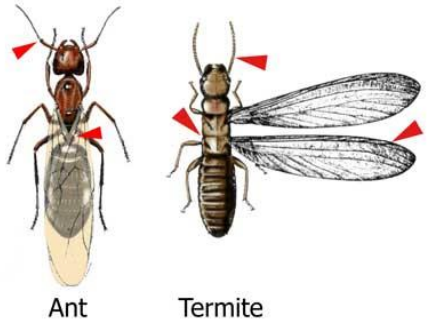
- Instead of the nest we put a whiteboard
- And we let NetLogo pick which student groups will present their solution… ☺



Exercise 1

Exercise 1: Modeling Termites

- Build a NetLogo model that represents the behavior of termites gathering wood chips into piles
- Termites follow a set of simple rules
  - If they bump into a wood chip, they pick the chip up, and continue to wander randomly
  - When they bump into another wood chip, they find a nearby empty space and put the wood chip down
  - With these rules, the wood chips eventually end up in a single pile

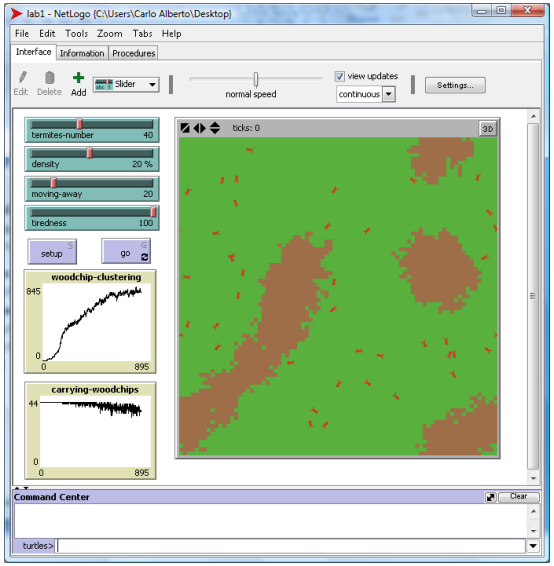


Exercise 1: Modeling Termites

- Generate an initial scenario (*setup* procedure)
  - Define an area with obstacles
  - Generate wood chips
  - Generate termites
- Define local rules (*go* procedure)
  - Termites look around for a wood chip to pick up, but if it takes too much, they stop and rest
  - Termites look around for a pile of wood chips, they found an empty slot next to the pile and drop off the chip
- Simulate and see the emergent behavior
  - Build monitors and plots to check the system behavior
  - Change the parameters and investigate their impact

Exercise 1: Modeling Termites

- Get used to NetLogo first!
  - Run some examples that have been shown in this tutorial, and get a feeling of how NetLogo works
- Understand the existing termites model
  - Extend it, starting with shapes, and other graphical aspects
  - Add input variables through sliders
- Follow the instructions carefully
  - Change the behavior of the termites
  - Add monitors and plots as required
- Deadline
  - **04.11.2018, 23:59 CET**
  - Next lecture: 05.11.2018



Questions?