Agent Based Modelling using NetLogo

A few apprentice pieces to get started with

Daniel M. Mayerhoffer

University of Bamberg
daniel.mayerhoffer@uni-bamberg.de

SICSS 2019, Bamberg

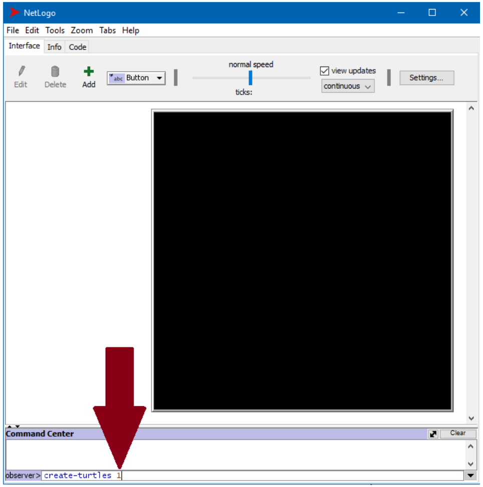
“Hello Wolrd!” and giving birth to your first agent

Type into the **Command Center** (at the bottom of your NetLogo window):

```
"Hello World!"
```

```
create-turtles 1
```

What happens for you?



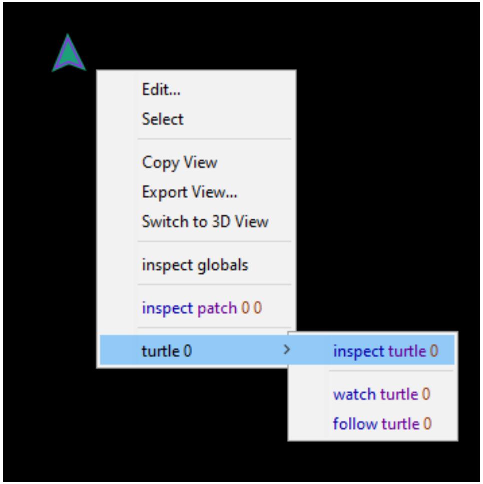
```
observer> create-turtles 1
```

Inspecting agents

Now right-click on the newly created turtle and inspect it. Can you guess what the shown *attributes* mean?

Create another turtle to see which attributes are random.

Also inspect a patch.



Telling agents what to do

Now, let‘s alter some attributes:

```
ask turtles [ setxy random-xcor random-ycor]
```

(Feel free to try this out multiple times. To do so without working your fingers to the bone use the *arrow up* key.)

```
ask turtles [ set color yellow ]
ask turtles [ set size 2 ]

ask patches [ set pcolor pink ]
```

It is also possible to address a single agent:

```
ask turtle 0 [ set color sky ]
```

Letting turtles walk around

With the `setxy` command, turtles "beam" to their new patch. But we can also make them move in a way that resembles walking:

```
ask turtles [ forward 1 ]
```

Turtles who just walk straight all the time are a bit boring but it is possible to let them change their direction, too:

```
ask turtles [ set heading (random 360) ]
```

To observe our turtles' walking path, have them draw it:

```
ask turtles [ pen-down ]
```

Smoothing the turns

Turtles sometimes turn around sharply instead of walking in a smooth curve. To change this, we have to modify the way they change their heading:

```
ask turtles [ set heading (heading +
                          (random 181) - 90) ]
```

Now, that we are happy with turtles‘ walking and turning, we can turn their path drawing off again:

```
ask turtles [ pen-up ]
```

Do you have any questions so far?

The Code tab

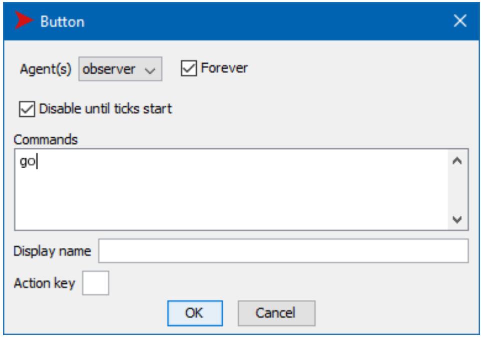
Commanding agents by hand can be (and usually is) quite impractical or even annoying. Especially, it is impossible to run a model multiple times in order to retrieve valuable output data.

Hence, it is usually better to write code in the *Code* tab and trigger its execution by a button (whose pushing can be automated as well) or other code snippets.

The go button

Create a button by right-clicking in the Interface background or **Add** in the toolbar.

In the commands input field type go.
Be sure to tick **Forever** and **Disable until ticks start**!

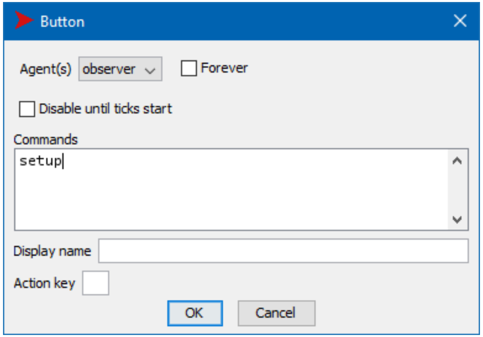


Agent(s): observer
[✓] Forever
[✓] Disable until ticks start
Commands: go
Display name:
Action key:

The setup button

Create another button the same way as the first one.

In the commands input field type `setup`.
This time, do not tick the two boxes!



Agent(s) observer
☐ Forever
☐ Disable until ticks start
Commands

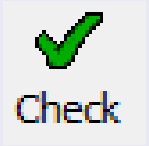
```
setup
```

Display name
Action key

Defining the setup and go procedures

```
to SETUP
 reset-ticks
end ;; SETUP

to GO
 tick
end ;; GO
```

Check your code!



Try it!

First click `setup` .

Now, go should be clickable. Click it, too.

What is happening?

Also try clicking the buttons again.

Uncheck **Forever** in the go (via right-click) and see the impact.

Automatising the walking process

```
to GO
 ask turtles [
   forward 1
   set heading (heading + (random 181) - 90)
 ]
 tick
end ;; GO
```

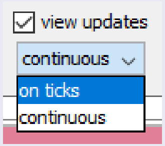
Watch your marvellous busy turtles!

Check your code and try the model.

What happens now?

(Isn't it beautiful?)

Set view updates to on ticks



Building a “wall” that turtles cannot cross

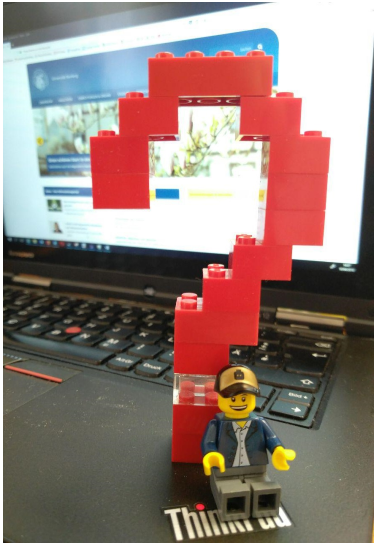
Now, we build a wall on the right of the world that sends turtles stepping onto it towards the centre.
Right click on a patch on the right edge to find its x-coordinate.

```
to SETUP
 reset-ticks
 ask patches with [ pxcor = 16 ] [
   set pcolor brown
 ]
end ;; SETUP
```

Making the wall functional

```
to GO
 ask turtles [
   forward 1
   if ([ pcolor ] of patch-here = brown) [
     facexy 0 0
     forward 10
   ]
   set heading (heading + (random 181) - 90)
 ]
 tick
end ;; GO
```

Do you have any questions about the basic commands?



The first really emergent model: Heroes and Cowards

Save the model and open a new one!

After being familiar with the basic commands in NetLogo, let's now turn to a first model featuring emergent behaviour: The heroes and cowards game.

My introduction and slides on this model a largely based on the ones created by Nicolas Payette (a great modeller and NetLogo expert). Thanks to him for sharing them!

The “Heroes and Cowards” game

Each agent has **one friend** and **one enemy**.
(Those relationships don’t have to be reciprocal.)

- If you are **brave**, you try to put yourself between your friend and your enemy.
- If you are a **coward**, you try to put your friend between yourself and your enemy.

What do you think will happen?
Let‘s try it with you as agents! :-)

Defining the setup and go procedures

This time, we include clear-all in the setup procedure, as it is commonly the case for NetLogo models.

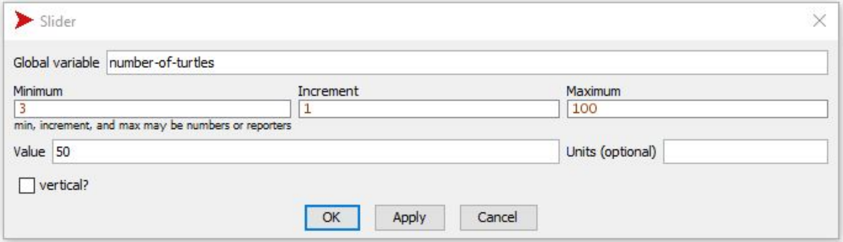
```
to SETUP
 clear-all
 reset-ticks
end ;; SETUP

to GO
 tick
end ;; GO
```

As before. . .

- . . . check your code!
- . . . create the corresponding buttons on the Interface.
  - go: Tick both boxes.
  - setup: Tick no boxes.

Let's create a slider for the new variable `number-of-turtles`

Create the slider the same way you created the buttons (using right-click or Add in the toolbar). On the Interface, be sure to set the **minimum** to **3**:


Slider

Global variable: number-of-turtles

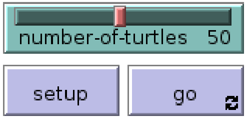
Minimum: 3 | Increment: 1 | Maximum: 100

min, increment, and max may be numbers or reporters

Value: 50 | Units (optional):

vertical? ☐

Result:

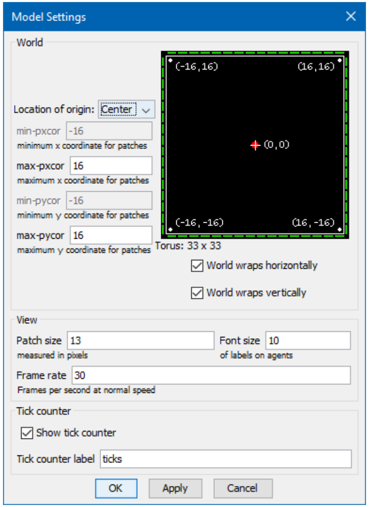


Let's create turtles in the setup procedure!

For the last model, we created the turtles manually. But this is also possible in the setup procedure. Thereby, we can also tell them to move away from the centre (feel free to try it out!):

```
to SETUP
 clear-all
 create-turtles number-of-turtles [
   setxy random-xcor random-ycor
 ]
 reset-ticks
end ;; SETUP
```

Undoughnutize the world!


Model Settings

World

Location of origin: Center

min-pxcor -16
minimum x coordinate for patches

max-pxcor 16
maximum x coordinate for patches

min-pycor -16
minimum y coordinate for patches

max-pycor 16
maximum y coordinate for patches

Torus: 33 x 33

☑ World wraps horizontally

☑ World wraps vertically

View

Patch size 13
measured in pixels

Font size 10
of labels on agents

Frame rate 30
Frames per second at normal speed

Tick counter

☑ Show tick counter

Tick counter label ticks

When playing the game ourselves, the room walls worked as borders of our ‘model world’; but in NetLogo the model world is a doughnut-shaped planet by default. So, we have to change that.
For doing so, click on **Settings. . .** or right click on the model world on the Interface tab.
There, uncheck **World wraps horizontally** and **World wraps vertically**.

Turtles need a friend and an enemy

To protect their friend or flee their enemy, turtles first must know them.
At the top of the code tab (before `setup`), add:

```
turtles-own [
 friend
 enemy
] ;; turtles-own
```

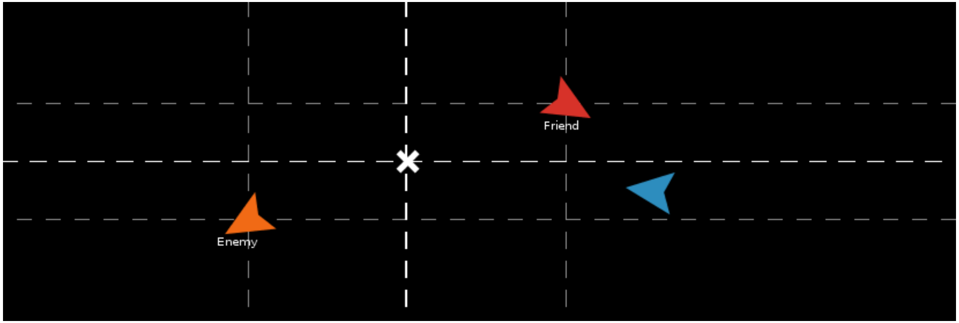
Assign the friends and enemies in setup

```
to SETUP
 clear-all
 create-turtles number-of-turtles [
   setxy random-xcor random-ycor
   set friend one-of other turtles
   set enemy one-of other turtles
 ]
 reset-ticks
end ;; SETUP
```

What is it like, to act bravely?

A brave agent aims for the midpoint between her friend and enemy:

- x-coordinate: $\frac{x_{friend}+x_{enemy}}{2}$
- y-coordinate: $\frac{y_{friend}+y_{enemy}}{2}$



Implement this in NetLogo

```
to ACT-BRAVELY
 let x ([ xcor ] of friend + [ xcor ] of
   enemy) / 2
 let y ([ ycor ] of friend + [ ycor ] of
   enemy) / 2
 facexy x y
end ;; ACT-BRAVELY
```

Link the procedure insetup to make everyone brave!

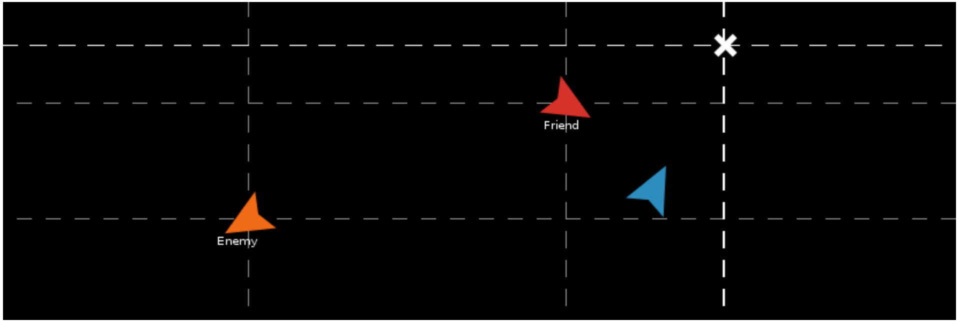
```
to GO
 ask turtles [
   ACT-BRAVELY
   forward 0.1
 ]
 tick
end ;; GO
```

Check it & try it!

What is it like, to act bravely?

A coward feels safest when she hides half the distance between her friend and her enemy behind the friend:

- x-coordinate: $x_{friend} + \frac{x_{friend} - x_{enemy}}{2}$
- y-coordinate: $y_{friend} + \frac{y_{friend} - y_{enemy}}{2}$



Implement this in NetLogo

```
to ACT-COWARDLY
 let x [ xcor ] of friend + ([ xcor ] of
   friend - [ xcor ] of enemy) / 2
 let y [ ycor ] of friend + ([ ycor ] of
   friend - [ ycor ] of enemy) / 2
 facexy x y
end ;; ACT-COWARDLY
```

Link the procedure insetup to make everyone a coward!

```
to GO
 ask turtles [
   ACT-COWARDLY
   forward 0.1
 ]
 tick
end ;; GO
```

Check it & try it!

But we wanted a mixed population of heroes *and* cowards. . .

Assign colours in `setup` to indicate heroic or coward behaviour.

```
to SETUP
 clear-all
 create-turtles number-of-turtles [
   setxy random-xcor random-ycor
   set color one-of [ yellow sky ]
   set friend one-of other turtles
   set enemy one-of other turtles
 ]
 reset-ticks
end ;; SETUP
```

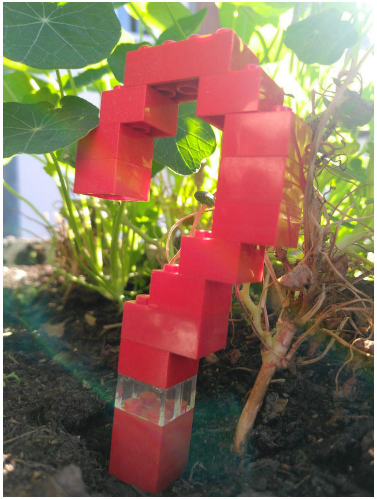
. . . and have them act accordingly!

```
to GO
 ask turtles [
   if color = sky [ act-bravely ]
   if color = yellow [ act-cowardly ]
   forward 0.1
 ]
 tick
end ;; GO
```

That‘s the whole model- Feel free to play with it!

Which cool patterns can you find?

Do you have any questions about this model?



A model on networks

Save the model and open a new one!

Social interaction happens on network structure. Therefore, social networks often research phenomena in networks; we are going to build a (simple) one that does this.

Alex (alexander.brand@uni-bamberg.de) is an expert for simulating network formation and interactions on networks in NetLogo. He also taught how to do that in my last NetLogo course. Hence, I cannot provide any slides here but if you have any questions, do not hesitate to contact either him or me.

Cellular Automata

Save the model and open a new one!

Another important class of models are Cellular Automata (Hegselmann & Flache 1998):

- Cells are arranged in a *regular D-dimensional grid*.
- Every cell adopts a state from a *finite* set of states.
- Time is *discrete*.
- **Cells change their states according to local rules.**
- The same transition rule applies to *all* cells.
- In each period cells are *updated* (simultaneously or sequentially).

Our Cellular Automaton

We want to model adaptation to one‘s immediate (social) context.

- Agents (i.e. cells/patches in this case) can have two colours
- Each step, they pick the majority colour in their neighbourhood.
- Later, we will detail this process by extending the basic model.

What can this model describe?

- Competing social norms
- Opinion development (on a yes-no-question)
  - On a geographical region or in social proximity
  - Evaluation of tipping points for change/revolution
- Moving into or out of property
- Spread of a disease
- . . .

Defining the setup and go procedures

That's a piece of cake for you by now, isn't it?

```
to SETUP
clear-all
reset-ticks
end ;; SETUP

to GO
tick
end ;; GO
```

A bit different from before. . .

- . . . check your code!
- . . . create the corresponding buttons on the Interface.
  - go: Tick both boxes.
  - go-once: Do not tick **Forever**
  - setup: Tick no boxes.

Patches need a pre-set colour initially

We tell the patches to randomly pick their initial colour from a list (of two) in `setup` procedure:

```
to SETUP
 clear-all
 ask patches [
   set pcolor one-of (list blue orange)
 ]
 reset-ticks
end ;; SETUP
```

How a patch sets its colour

Patches are called sequentially (in random order) and each individually perform the following steps:

1. Count neighbouring patches that are orange and store the result temporarily.
2. Count neighbouring patches that are blue and store the result temporarily.
3. Compare the results and set my own colour accordingly.

Implementation of setting the patch colour

```
to GO
 let oranges 0
 let blues 0
 ask patches [
   set oranges count neighbors with [pcolor = orange]
   set blues count neighbors with [pcolor = blue]
   set pcolor ifelse-value (oranges > blues)
                                                [orange] [blue]
 ]
 tick
end ;; GO
```

Feeling blue?

Check your code and try the model a couple of times.

Hint: You do not have to stop the model to reset it. Just hit `setup` again while `go` is active. This may feel more convenient.

What happens?

Why did this happen?

```
to GO
 ...
 ask patches [
   ...
   set pcolor ifelse-value (oranges > blues)
                                     [orange] [blue]
 ]
 tick
end ;; GO
```

If there are four blue and four orange neighbours, the condition is false and hence the patch becomes blue. This slight bias towards blue in case of a tie is sufficient to tip the whole model outcome towards blue.

Making things right

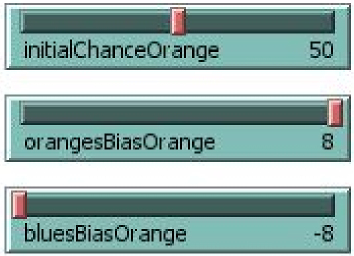
```
to GO
 ...
 ask patches [
   ...
   ifelse (pcolor = orange) [
     set pcolor ifelse-value (oranges < blues)
                                       [blue] [orange]
   ] [
     set pcolor ifelse-value (oranges > blues)
                                       [orange] [blue]
   ]
 ...
end ;; GO
```

Bring back a bias

Maybe we should bring back the bias intentionally to study it: They can play a role at the tipping process (for orange or blue cells) or for the initial distribution.

Create three **Sliders** on the Interface tab

- **initialChanceOrange**: ranging from 0 to 100, with an increment of 10
- **orangesBiasOrange**: ranging from -8 to +8, with an increment of 1
- **bluesBiasOrange**: ranging from -8 to +8, with an increment of 1



Implementing the bias at setup

At `setup`, we now want cells to become orange if a random number (between 0 an 100) is smaller than our specified chance:

```
to SETUP
 clear-all
 ask patches [
   set pcolor ifelse-value (random 100 <
      initialChanceOrange) [orange] [blue]
 ]
 reset-ticks
end ;; SETUP
```

Implementing the bias at go

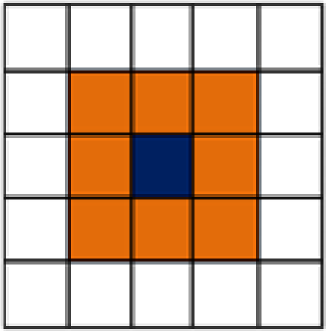
```
to GO
 ...
 ask patches [
   ...
   ifelse (pcolor = orange) [
     set pcolor ifelse-value (oranges +
     orangesBiasOrange < blues) [blue] [orange]
   ] [
     set pcolor ifelse-value (oranges +
     bluesBiasOrange > blues) [orange] [blue]
   ]
 ...
end ;; GO
```

Different neighbourhood sizes

So far, cells have always considered the 8 cells surrounding them as their neighbours. However, this neighbourhood size may encompass theoretical assumptions that we are not sure of.

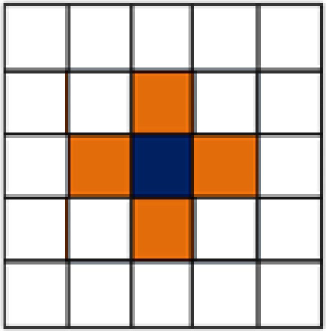
Moore

NetLogo: `neighbors`



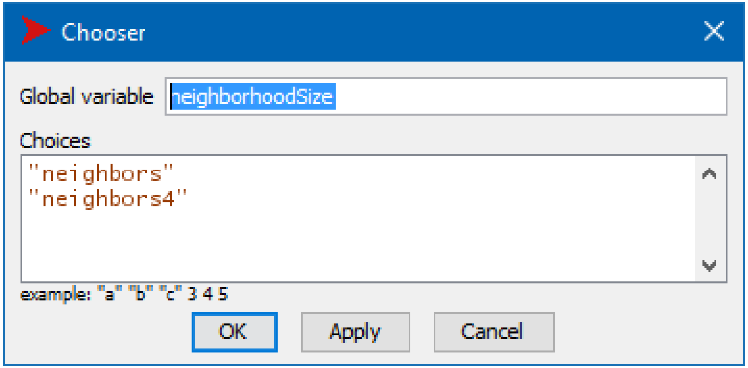
von Neumann

NetLogo: `neighbors4`



Choosing between neighbourhood sizes

Create a **Chooser** on the Interface (the same way you created sliders and buttons) and allow users to switch between the two neighbourhood definitions:



Global variable neighborhoodSize

Choices

```
"neighbors"
"neighbors4"
```

example: "a" "b" "c" 3 4 5

Making the Chooser meaningful

To implement the choice of neighbourhood size, we use the (very slow) command `runresult` that treats a string (that we specified in the Chooser) as command.

```
to GO
  ...
  ask patches [
    set oranges count (runresult neighborhoodSize)
                                   with [pcolor = orange]
    set blues count (runresult neighborhoodSize)
                                     with [pcolor = blue]
  ...
end ;; GO
```

Preparing the model for serious analysis

‘Analysing’ the model by just looking at it running for multiple times is a good start. But at some point, we may want to retrieve data in a more systematic way. To do so, we first have to tweak the model a bit:

1. A stop condition (i.e. no patch changing its colour) so that the model does not run unnecessarily long and that we also get information on how long a certain setting takes to stabilise.
2. A pre-specifiable random-seed that allows us to recreate a certain model run.

Implementing a stop condition. . .

. . . at top of our code. . .

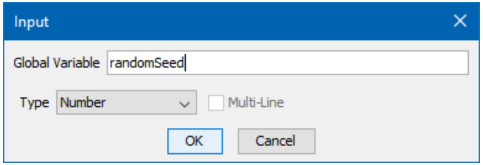
```
patches-own [
 pcolor-old
] ;; patches-own
```

. . . and in the go procedure

```
to GO
 ...
 ask patches [
   set pcolor-old pcolor
   ...
 ]
 if (all? patches [pcolor = pcolor-old]) [stop]
 tick
end ;; GO
```

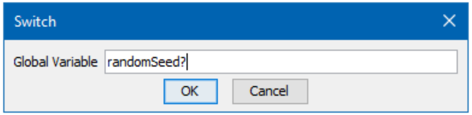
Defining a random seed

On the Interface create an **Input** to specify the random-seed and a **Switch** to activate it.



Global Variable: randomSeed
Type: Number

Make sure to choose Type **Numerical** because otherwise a number entered is treated as a string



Global Variable: randomSeed?

Question marks usually indicate boolean type (true/false) variables.

Implementing the seed at setup

The command random-seed is used to specify the internal seed used in NetLogo. We want it to be either our specified seed or a standard one (created based on the current date and time), depending on the switch:

```
to SETUP
 clear-all
 random-seed ( ifelse-value randomSeed?
                  [randomSeed] [new-seed] )
  ...
end ;; SETUP
```

A parameter for observation

Obviously, one is likely primarily interested in the ratio of blue and orange cells. If we want to calculate this, we first must define it at top of our code as global variable which is stored centrally (and not as agent property):

```
globals [
 shareOranges
 shareBlues
] ;; globals
```

(Yes, one rate would be enough but we will need the second one for visualisation.)

Calculate the parameter values

Code looks way cleaner if statistical calculations are put into a separate procedure.

```
to STATISTICS
 set shareOranges (count patches with
     [pcolor = orange] / count patches)
 set shareBlues (1 - shareOranges)
end ;; STATISTICS
```

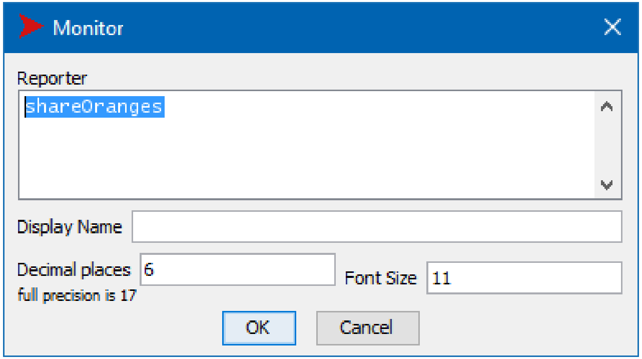
Moreover, this procedure can be called multiple times in the rest of your code. . .

Call STATISTICS in setup and go

```
to SETUP
 ...
 STATISTICS
 reset-ticks
end ;; SETUP

to GO
 ...
 STATISTICS
 if (all? patches [pcolor = pcolor-old]) [stop]
 tick
end ;; GO
```

Observing the parameter: Monitor

The simplest way of observing a variable in your model is a **Monitor**. Create one on the Interface for **shareOranges**. Feel free to play around with Display Name, Decimal places and Font Size:

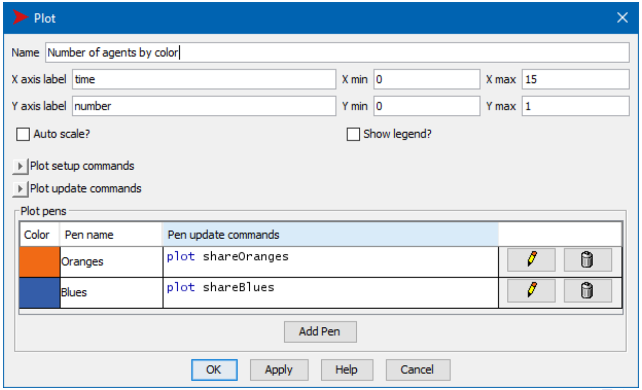


```
shareOranges
```

Display Name
Decimal places 6
full precision is 17
Font Size 11

Observing the parameter: Plot

Create a plot which allows monitoring graphically and over time. Use `plot shareOranges` to plot the rate of orange cells and select a pen colour and name accordingly. Add a Pen for `plot shareOranges`. Set **Y-max** to 1, **Y-max** to 15 and un-check **Auto scale?**.



Plot

Name: Number of agents by color

X axis label: time — X min: 0 — X max: 15

Y axis label: number — Y min: 0 — Y max: 1

☐ Auto scale? ☐ Show legend?

Plot setup commands

Plot update commands

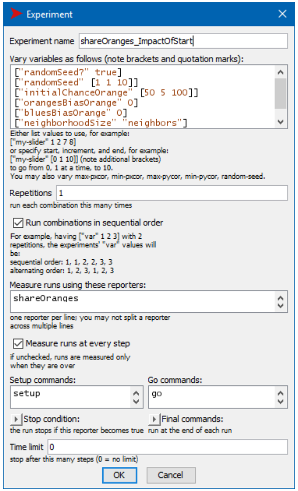
Plot pens

Color	Pen name	Pen update commands
🟧	Oranges	`plot shareOranges`
🟦	Blues	`plot shareBlues`

Experiments in the Behavior Space

Hitting the buttons all the time when running the model multiple times would be annoying and comparing outputs from different runs. Therefore, NetLogo provides a tool to automatise this: The **Behavior Space** is accessible via Tools in the Menu Bar. Create a new experiment and specify parameters (leaving all others unchanged):

- Vary variables as follows:
  - `["randomSeed?" true`
  - `["randomSeed" [1 1 10]]`
  - `["initialChanceOrange" [50 5 100]]`
  - `["orangesBiasOrange" 0]`
  - `["bluesBiasOrange" 0]`
  - `["neighborhoodSize" "neighbors"]`
- Measure runs using these reporters: `shareOranges`
- un-check **Measure runs at every step**



Experiment

Experiment name shareOranges_ImpactOfStart

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

```
["randomSeed?" true]
["randomSeed" [1 1 10]]
["initialChanceOrange" [50 5 100]]
["orangesBiasOrange" 0]
["bluesBiasOrange" 0]
["neighborhoodSize" "neighbors"]
```

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:

```
shareOranges
```

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`

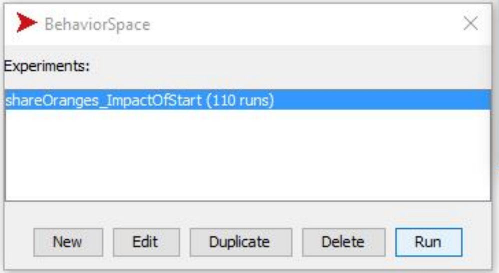
Stop condition:
the run stops if this reporter becomes true

Final commands:
run at the end of each run

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

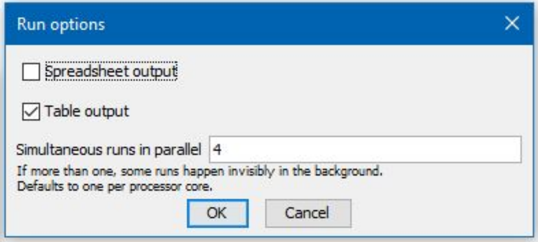
Run the experiment

Save your settings by clicking **OK** and then **Run** the experiment from Behavior Space. Select **Table Output** and if you want to see it also **Spreadsheet Output**. Specify directory and file name. The result is a CSV file.


BehaviorSpace

Experiments:

shareOranges_ImpactOfStart (110 runs)


Run options

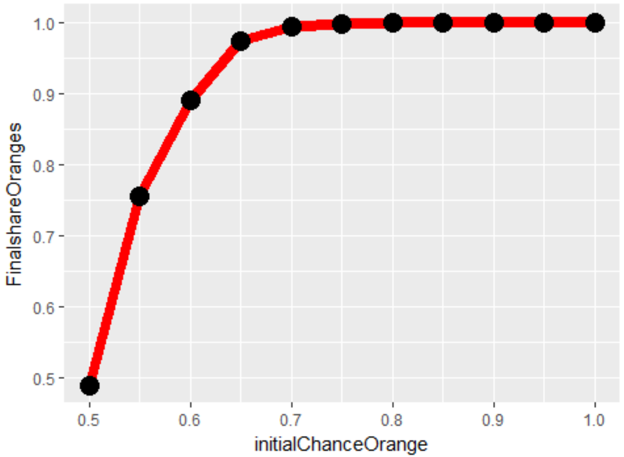
☐ Spreadsheet output

☑ Table output

Simultaneous runs in parallel: 4

If more than one, some runs happen invisibly in the background.
Defaults to one per processor core.

Analyse the output using your favourite statistics software



Do you have any questions for today?