Tutorial 1: The base model

This tutorial is based on the Neutral Model of Stone Raw Material Procurement developed by Jeffrey Brantingham. Brantingham's (2003) highly influential article was the first application of agent-based modeling to the topic of procurement, curation and spatial distribution of raw material used for lithics production in prehistory. Brantingham sought to develop from first principles a model of raw material curation, that is, a base model stripped of any behavioral assumptions that could be then compared to the lithics assemblages found at archaeological sites. Essentially, he asked how these assemblages would look if the processes leading to their creation were random (Brantingham 2003: 490).

The main premise behind the model is to establish the pattern of assemblage variability under neutral conditions of no behavioural biases (for example, without a preference for any particular raw material type and no specific type of mobility). In the simulation, a single agent-forager follows a random walk (see O'Sullivan and Perry 2013, ch 4) through a uniform landscape dotted with raw material sources. The agent collects raw material indiscriminately whenever s/he comes across a source. When a raw material source is encountered their toolkit is filled up until it contains 100 pieces and the agent continues their journey. At every step one piece is deposited on the current grid cell. There is no specific foraging/movement strategy or any preference for a particular type of raw material.

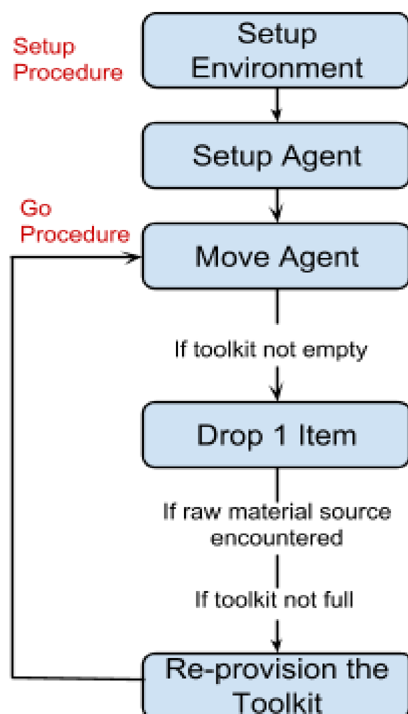

Figure 1.0.1. Flowchart of the Neutral Model. Adapted and simplified from Brantingham 2003, fig. 5.

The outline of the simulated processes is given in Fig. 1. In the setup phase a 500x500 cells world is seeded with 5000 unique raw material sources. Each raw material type is present at only one cell and their distribution is random. In the second phase the agent is initialised with an empty toolkit. Once the setup phase is completed the time clock is started. In each time step the agent either moves to one of the 8 neighbouring cells chosen at random or stays put. If the toolkit is not empty, the agent drops one randomly chosen item. If the cell the agent is on is a raw material source the agent reprovisions the toolkit unless it is full. The cycle move-drop-reprovision repeats until the simulation is stopped. The richness of the toolkit (the number of unique raw materials) as well as the variability of assemblages composed of items dropped by the agent are recorded throughout the simulation.

In this tutorial we will try to replicate Brantingham's model as accurately as possible. It will take approximately two hours to complete. It has been built and tested in NetLogo 6.0.1. For installing instruction and more in-depth explanations see Romanowska et al. 2019 Supplementary Information B.

1.1 NetLogo Interface

The NetLogo interface (Fig. 2) consists of three tabs: **Interface**, **Info** and **Code**. Let's look at them in turn. The Interface window consists of: the **View Panel** for watching the simulation, a few buttons along the top of the window with settings and the **Command Center** towards the bottom of the window, which you can use to directly alter or inspect any element of the simulation.

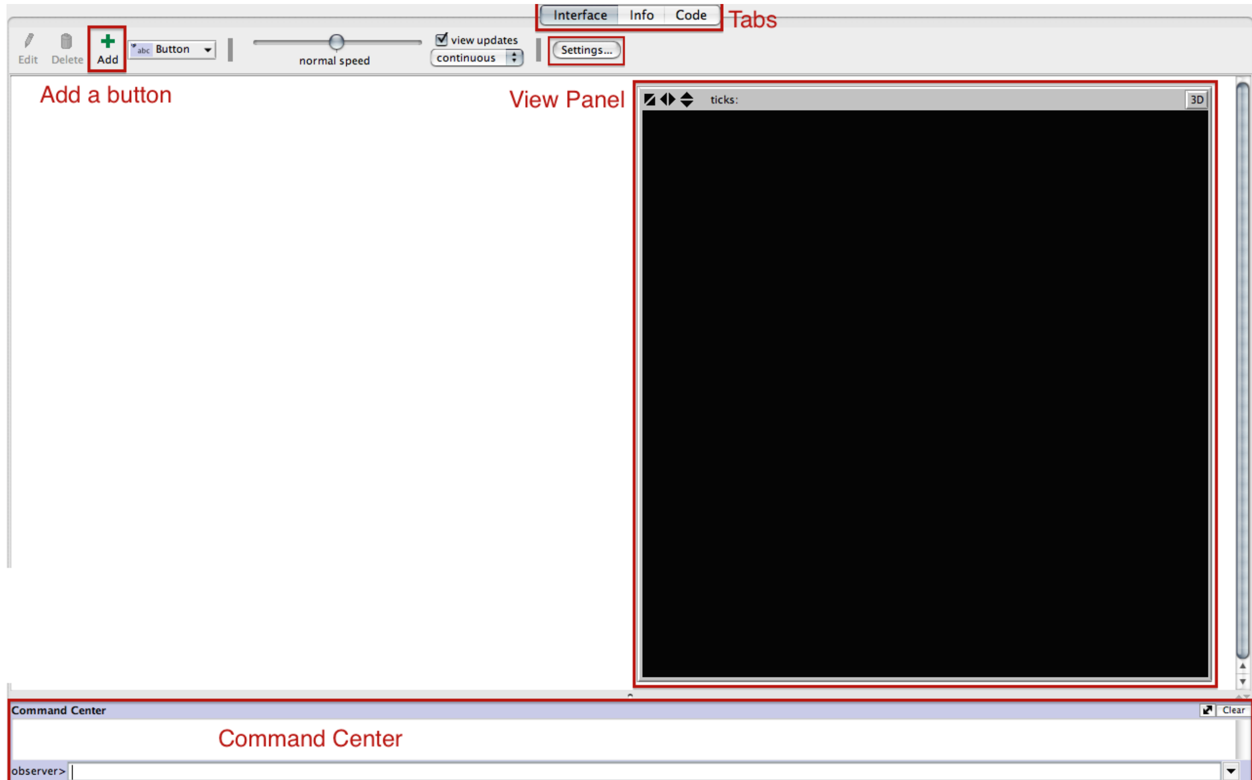

Figure 1.2. The Netlogo Interface.

We are first going to change the size of our simulation window to accommodate a much larger view. Click on the 'Settings' button in the top menu to adjust the view panel. First change the 'Location of origin:' to 'Corner' and choose 'Bottom left' from the drop down menu. We need a much larger area than the default so replace the values in max-pxcor and max-pycor to 499 (the coordinates of the first patch start at 0 0 so setting maximum x and y to 499 gives a 500x500 square). However, this means that if we keep the size of **patches** (grid squares) as large as it is now the screen will be enormous. Change the 'Patch size' to 1.0 and hit 'Apply'. You might have noticed the two tick boxes 'World wraps horizontally' and 'World wraps vertically'. If ticked they provide continuity between the edges of the screen, i.e., if the agent moves right while standing on the right-most patch it will appear on the left-most patch; this is known as a torus world as it doesn't have edges. Check that both tick boxes are ticked and hit 'OK'. This will get you back to the main "home" image. If you do not like the size of the view panel, right click anywhere on it, choose 'Select' from the dropdown menu and drag one of the corners until the size is ok - note that this only changes the size of the patches, their number (500 by 500) remains the same.

1.2 The setup and the go procedures

The backbone of almost all NetLogo simulations are two procedures: 'setup' and 'go'. The **setup procedure** is used to initialise the simulation, i.e., to create the starting population of agents and to their environment. The **go procedure** is the main simulation loop where in each time step the agents and the environment undergo a series of actions.

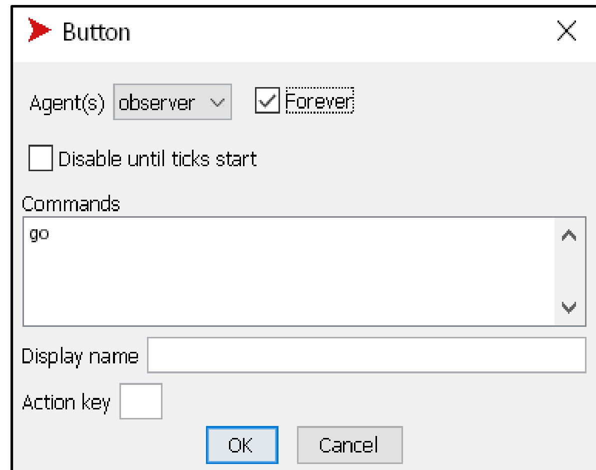

Figure 1.3. The button window.

Click on the 'Add' button and then click anywhere on the white space. A dialogue box (Fig. 3) should pop up. Write `setup` in the 'Commands' box and click OK. Follow the same steps to create a second button and write `go` in the 'Commands' box. This time also tick the 'Forever' box. This means that this action will be repeated until the simulation ends.

You can see that the text on both buttons has instantly turned red, indicating an error - this is because we have not yet defined what we mean by 'setup' and 'go' within the code. Let's move to the Code Tab to fix it. The Code Tab consists mostly of a white text box, the **code box**, and a few buttons towards the top, which we will inspect in a moment. We will start with setting up the two procedures. Type the following in the code box:

```
to setup
end

to go
end
```

The words `to` and `end` delimit all NetLogo procedures. If you now click on the 'Procedures' button at the top of the screen, you will find that 'setup' and 'go' are already there. To the left is the debugger button 'Check' - if you click on it, it will check if the basic syntax of the code is correct.

1.3 The setup procedure

Logo - the language of NetLogo - was developed to resemble a natural language as much as possible, which means that it is very easy to understand the code. It was also developed with educational goals in mind (read: teaching kids), which means that it is equally easy to write. We will start with setting up the environment by asking each patch to set a number of variables: colour, whether they are a raw material source patch or not and the list of dropped lithics it contains. In addition, we will use the standard NetLogo functions that ensure that every time we hit the setup button the remains of the previous runs are removed. All **commands** directed at turtles and patches are initiated by the word `ask` and enclosed in square brackets `[]`, here we will make use

of them for the first time. Type inside the setup procedure (i.e., between `to setup` and `end`), so that it looks like this:

```
to setup
      clear-all
      ask patches[
            set pcolor white
            set source? false
            set assemblage []
      ]
      reset-ticks
end
```

In the first line we use the `clear-all` primitive to wipe clean any remnants of the previous runs. Similarly the last line of the setup procedure is always dedicated to resetting the time counter using the `reset-ticks` primitive. **Primitive** is NetLogo jargon for an in-built function, that is defined in the NetLogo library. Check out the [NetLogo dictionary](https://ccl.northwestern.edu/netlogo/docs/dictionary.html) (https://ccl.northwestern.edu/netlogo/docs/dictionary.html) for a full list of primitives. Coming back to the code, we set up the environment by asking patches: 1) to set their color to white, 2) to set their status as a source of raw material (we will ask them all to be a no-source for now), and 3) to start a list in which we will record whether any artefacts have been dropped on this particular patch during a run. The flow of the program is governed by brackets and it is very easy to lose track of how many you have opened already. To avoid confusion once you open a bracket, immediately close it and write the code inside. The indentation does not matter, but it makes the code easier to read so we recommend using it. Hit the 'Check' button to see if there are any errors.

And there are. There always are. The message: 'Nothing named SOURCE? has been defined.' appeared at the top of the screen. Indeed, we tried to set the variable `source?` to false without defining it first. Congratulations on seeing your first code error!

Variables are often used to describe the characteristics of agents, patches and the world. For example, an agent can have age, gender, energy, cultural marker or any other feature relevant to the model. These variables may change throughout the simulation run (e.g., age, energy) or remain static (e.g., gender, cultural marker). There are two types of variables in NetLogo's syntax: 1) **global variables**, used throughout the code, which must be listed at the beginning of the code or defined using Interface items (we will come back to this), and 2) **local variables**, defined by the `let` primitive, which are only valid within one procedure. We will see the use of local variables later on, but the `source?` is a global one - it is a characteristic of all the patches. We will set its value in the setup procedure and it will remain unchanged throughout the run. The same applies

to the `assemblage` list so we will also define it as a global variable. Type at the beginning of the code (before "`to setup`"):

```
patches-own [ source? assemblage ]
```

And hit the 'Check' button; there should be no errors. If you now go to the Interface tab and click on the 'setup' button, you will see that the screen went white. Right click anywhere within the view panel and choose 'inspect patch ...' from the drop down menu. It will show you (Fig. 4) a list of patch variables, including some of the built-in ones such as the x and y coordinates and the patch color, but also the two we have defined ourselves: `source?` and `assemblage`.

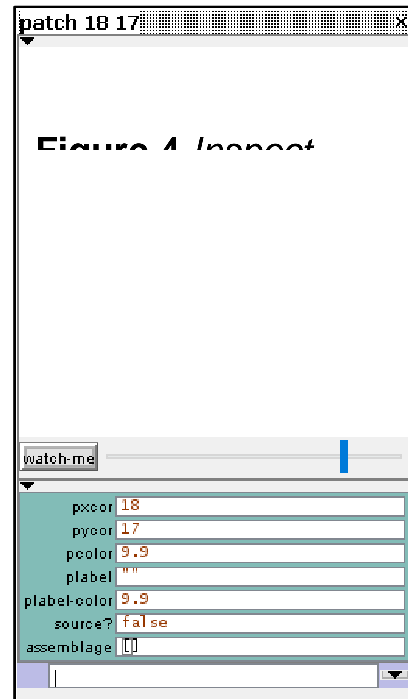

Figure 1.4. Inspect patch window.

The white screen is not very exciting so let's set up the patches that are raw material sources. Because we do not want ALL the patches to be a source we will use the `n-of` (*number of*) primitive. Each raw material source needs to be unique so we will give them a different id. Type inside the setup command, after the first command block but before the `reset-ticks`.

```
let r 1
ask n-of 5000 patches [
            set source? true
            set material_type r
            set r r + 1
            set pcolor black
  ]
```

The first thing we do here is to define a local variable `r` and set it up as 1. We then ask 5000 patches 1) to change their `source?` status to true, 2) to set the `material_type` as the unique id `r`, 3) to then add 1 to `r` so that the next patch gets the next (`r+1`) id, and finally 4) we will change their colour to black to see where the raw material source patches are. Hit the 'Check' button. A familiar error message appears. But this time you know what to do! Add `material_type` in the list of patch variables (`patches-own`) at the beginning of the code:

```
patches-own [ source? assemblage material_type ]
```

Move to the Interface tab and hit the 'setup' button. You can now inspect one of the source patches by right clicking on it and choosing 'Inspect Patch ...' from the drop down menu. You will see in the pop-up window that the value of `source?` is `true`.

We have now created the environment, but not the agents. We actually only need one but we should give her/him quite a few variables such as the initial location and looks as well as a maximum number of lithics s/he can carry and a list to keep track of them. Type after the patches setup procedures but before `reset-ticks`:

```
  crt 1 [
    setxy random max-pxcor random max-pycor
    set color red
    set size 10
    set shape "person"
    set max_carry 100
    set toolkit []
  ]
```

`crt` stands for 'create agents', in our case, one agent. Inside the brackets we set their initial position to a random patch (i.e., with x and y coordinates between 0 and the current maximum - `max-pxcor` and `max-pycor`) and add a few variables: color (notice that `color` applies to agents and `pcolor` to patches), size and shape. We also initiate a list of all the raw material types the agent carries in its toolkit and set up how much s/he can carry at any one time. Just like `source?` or `assemblage` the `toolkit` list and the `max_carry` variable are global variables (you can hit 'Check' if you want to see the error message). However, this time they apply to agents not patches so we need to make an turtle-specific variables lists. Write the following line at the beginning of the code:

```
turtles-own [ toolkit max_carry ]
```

Here, we finally drop the bombshell - in NetLogo jargon agents are called **turtles**. This is the legacy of being originally developed as an educational tool for kids. It makes for a fun code development and all NetLogo developers sooner or later learn to love their turtles.

If you now go to the Interface tab and hit 'setup' you should find a red human-shaped agent on one of the patches (Fig. 5). If you keep on pressing the setup button you will see that each time the simulation is reset, the agent and source patches are initialised at a different (random) location.

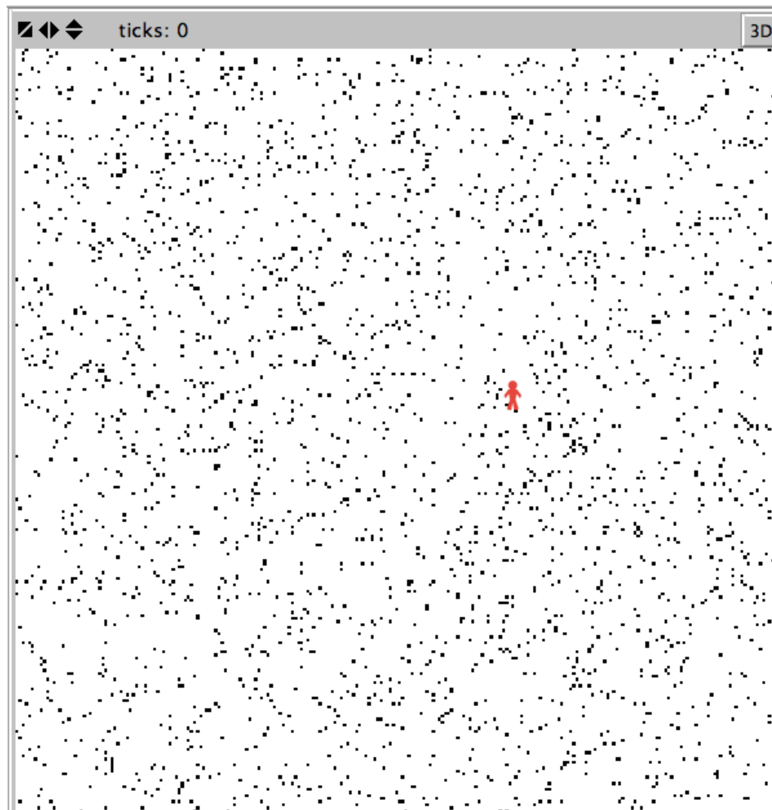

Figure 1.5. The initialized simulation.

1.4 The go procedure

With the setup complete let's move on to the `go` procedure, i.e., the main body of the simulation which will be repeated until you click on the 'go' button again. The first thing we want the agent to do is to move around the landscape. We have established that in each time step the agent will move to one patch in its Moore neighborhood (that is, to the S, N, E, W or SE, SW, NE, NW of the current location) or stay put. You can put it in a more mathematical terms as: the agent has a 1 in 9 chance of staying where it is or moving to any one of the surrounding patches. In order to code the agent's movement, we start the command with the keyword `ask turtles` and enclose a list of functions, e.g., the `move-to` primitive in brackets. Type the following inside the go procedure:

```
to go
      ask turtles [
            if random 9 > 0 [ move-to one-of neighbors ]
      ]
      tick
end
```

Let's look at the code a bit closer. We ask all turtles (in our case there is only one) that if a random integer (whole number) between 0 and 8 is higher than 0 then the agent should perform the functions that are inside the brackets: `move-to one-of neighbors`. If it's 0 then nothing happens - the agent stays put. `move-to`, `one-of` and `neighbors` are all NetLogo primitives and we encourage you to check them in the NetLogo [dictionary](https://ccl.northwestern.edu/netlogo/docs/dictionary.html) (https://ccl.northwestern.edu/netlogo/docs/dictionary.html). We also added the `tick` primitive at the end of the go procedure that moves the time counter by one. Go to the Interface tab and click first on 'setup' and then on the 'go' button. You should see the red agent running around the world like crazy. Use to speed slider at the top of the window to slow it down a bit. You should be able to see that the agent moves by one patch as the time counter underneath the speed slider is ticking forward. Click on the 'go' button to stop the simulation.

The next thing to do according to the flow diagram of the model (Fig. 1) is for the agent to drop one item at each step whenever its toolbox is not empty. We will use an if-loop to check whether there is anything to drop in the toolkit and then update both the assemblage of the patch and the agent's toolkit. Write the following code inside the `go` using another '`ask turtles`' command (after the final closing bracket of the movement function but before `tick`):

```
ask turtles[

   if length toolkit > 0 [
     let i random length toolkit
     ask patch-here [
            set assemblage fput (item i [ toolkit ] of myself)
     assemblage
     ]
     set toolkit remove-item i toolkit
   ]
]
```

Let's go through the code line by line. Like in the previous code snippet we use the 'if' conditional loop. This time we will only perform the functions enclosed by the first set of square brackets if the length of the toolkit list is more than zero, i.e., the toolkit is not empty. If that is the case, we choose at random an item with an index `i` from the toolbox. `i` is an index number between zero and the number of items currently present in the toolkit (`length toolkit`) which denotes its position in the list (as in: first, third, tenth etc.). In the next line we ask the patch on which the agent stands (note the special primitive `patch-here`) to add the item (using the `fput` list primitive) `i` of the `toolkit` of the turtle asking, referred to with the primitives `of myself`, to the patch's list of dropped items - `assemblage`. We then remove the same item from the agent's toolkit.

If you run the simulation and inspect the agent (click on the 'go' button to pause the simulation, then right click on the agent and choose 'inspect turtle 0'), you will notice that despite all the coding we have just done the agent's toolkit remains empty. That's not what we want! But it is easy to understand why. We have no code for picking up raw material! In short, the agent never got a

chance to pick anything up! This cycle of writing small modular code bits and then checking the simulation (by just running it and by inspecting its elements) is the best way of writing code. If you try to write everything at once, chances are there will be errors and it will be much harder to find them.

We recognise patches which contain a raw material source by their variable `source?` set as `true`. Whenever the agent comes across one of them we want it to restock the raw material. We will again use an if-loop to check whether the patch is a 'source patch' and if so fill up the agent's toolkit until it is full (using the `while` loop). Type the following code to make a new `ask turtles` command inside the `go` procedure (after dropping procedure but before `tick`):

```
ask turtles[

      if [ source? ] of patch-here = true [
            let raw_material [ material_type ] of patch-here
            while [ length toolkit < max_carry ] [
                  set toolkit fput raw_material toolkit
            ]
      ]
]
```

In the first line we ascertain that the patch is indeed a source patch. If it is not, the block of code enclosed in the square brackets will be ignored and the program will move to the next statement (in this case: `tick`). Do you remember that each raw material source has a unique ID? We need that ID so that we know what type of raw material is added to the toolkit. The `let` statement sets up a local variable `raw_material` to the same value as the ID (`material_type`) of the patch the agent is standing on (`patch-here`). The variable raw_material is local, meaning it is only recognised inside the `ask turtles` brackets. If you try to use is anywhere else, our favourite error message ('Nothing named raw_material has been defined') would pop up. In the next line a 'while-loop' adds the `raw_material` to the `toolkit` list until the maximum capacity of the agent (`max_carry`) is reached. Note the difference between the if- and while-loops here. If the given condition is fulfilled (e.g., the patch-here is a source or a randomly drawn number is higher than zero) an if-loop will perform the actions defined inside its brackets once. A while-loop will keep on repeating them until the given condition is reached (e.g., the toolkit length is equal to the maximum capacity of the agent).

Go to the Interface tab and run the simulation. Inspect the turtle - if you keep the inspect window open during the run you should be able to see how the toolkit changes every time the agent comes across a raw material source patch.

It is a bit difficult to see the raw material toolkit in the small box, so let's create a plot that will show the changes in the frequencies of different raw material types present in the toolkit. Click on the drop-down list next to the 'Add' button at the top of the Interface tab and choose 'Plot'. Click anywhere on the white area outside of the view panel. A new pop-up window will appear. Write in the 'Name' box: 'Toolkit richness'. The 'Plot pens' box is where we specify what should be

plotted. The default value of `plot count turtles` counts the number of agents and in many cases is very useful but since we only have one agent it does not make much sense. Delete it and type:

```
plot [ length (remove-duplicates toolkit) ] of turtle 0
```

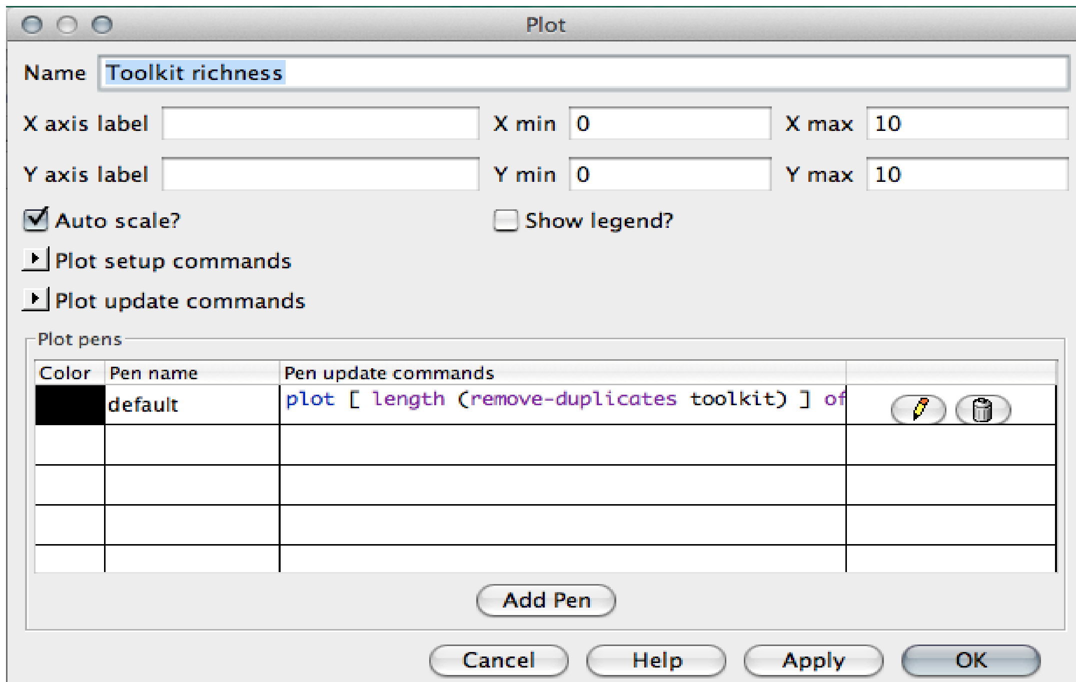

Figure 1.6. The plot interface.

This will plot the size (`length`) of the toolkit list once all duplicates are removed, that is the number of unique raw material sources present in the agent's toolkit. Every turtle has an in-built unique number assigned to it when it is created. Since we only have one agent, its number is zero. We specify this (`of turtle 0`) because otherwise the plotting function would not know which agent's toolkit to plot. Run the simulation (slow it down). If you compare your plot with figure 7 in Brantingham's paper (2003) you will notice that they strongly resemble each other.

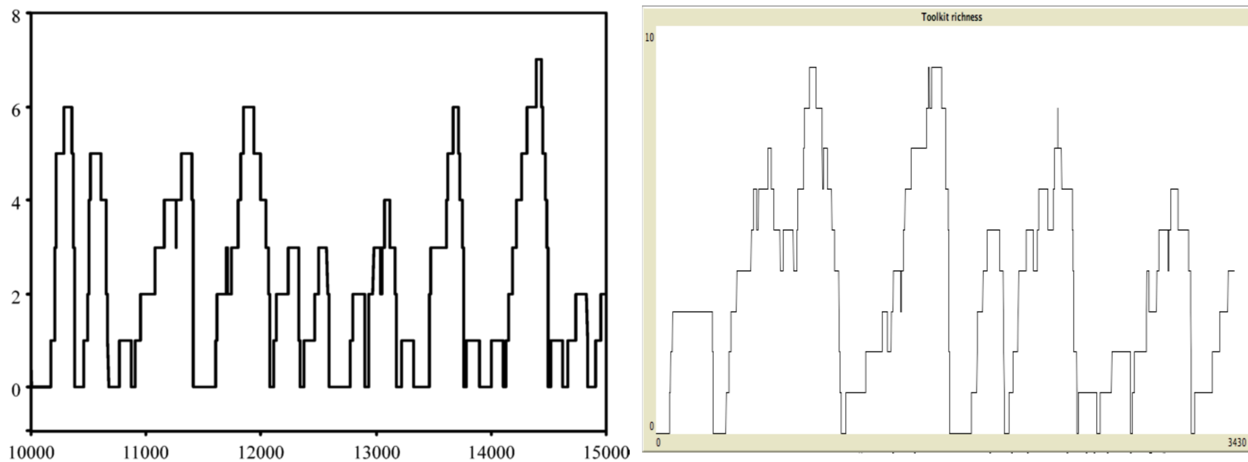

Figure 1.7. The plots, Brantingham versus our model.

Congratulations you have successfully replicated a famous model!

To finish off, we will slightly extend the neutral model. Brantingham notes that the dynamics of the simulation will change if the maximum size of the toolkit that the agent can carry is altered. We will set up a slider to help running a series of experiments that will test it. As mentioned before, global variables can be defined at the beginning of the code in the variables lists (`patches-own`, `turtles-own`). However, you can also define them in the Interface tab by using a slider, a chooser or a box. Go to the interface and click on the drop-down list next to the 'Add' button and choose 'Slider'. Then click anywhere on the white field outside of the view panel. A pop-up window will appear. Type `max_carry` in the 'Global variable' box at the very top. You can leave the rest of the boxes unchanged and hit OK. You immediately get an error message saying that 'There is already a global variable called MAX_CARRY'.

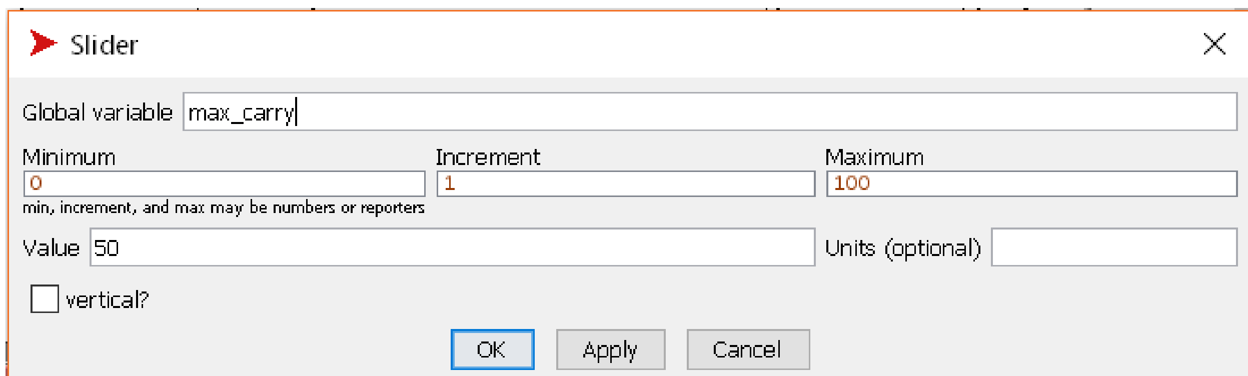

Figure 1.8. The slider variable interface.

This is because we have already defined `max_carry` at the beginning of the code in the `turtles_own` list. Now we have two sources of a variable called `max_carry` (the slider and the `turtles-own` list) and NetLogo does not know which one to use. Simply delete `max_carry` from the `turtles_own` list. It should look like this now:

```
turtles-own [ toolkit ]
```

Also, go to the `setup` procedure and delete the line `set max_carry 100` in the command used to create the agent. If you forget to remove it from the setup procedure what will happen is NetLogo will read from the slider the value of `max_carry` (say 80) and then start executing the setup procedure, but when it comes across the line in which you `set max_carry 100` it will overwrite the value set on the slider (80) to the one in the code (100). This is a common error because it does not produce an error message (since you have not done anything illegal according to the NetLogo syntax). Hit the 'Check' button and there should be no more errors. You can now use the slider to vary how much the agent can carry in each run. Although you can use the slider during the run, it is discouraged as the results will not be replicable. Instead change the value before each run and compare the output of the plot. Have you noticed how the peaks of the toolkit richness are lower and less frequent if the `max_carry` is set to a lower number?

This is the end of Tutorial 1. In the next one (Davies et al. 2019), we will move our model into a real landscape generated from GIS layers. Tutorial 3 (Crabtree et al. 2019) will focus on how to better collect the simulation output (just looking at a plot is not the greatest method) and how to automate running the experiments (so you don't spend days moving sliders).

To cite this document:

Romanowska, I., S. Crabtree, B. Davies, and K. Harris. 2019. "Agent-based Modeling for Archaeologists. A step-by-step guide for using agent-based modeling in archaeological research (Part I of III)." Advances in Archaeological Practice 7 (2).

References cited

Brantingham, P. Jeffrey
2003 A Neutral Model of Stone Raw Material Procurement. *American Antiquity* 68(3): 487–509.

Crabtree, Stefani, Kathryn Harris, Benjamin Davies, Iza Romanowska
2019 "Outreach in Archaeology with Agent-based modeling. A step-by-step guide for using agent-based modeling in archaeological research (Part III of III)." Advances in Archaeological Practice 7 (2).

Davies, Benjamin, Iza Romanowska, Kathryn Harris, Stefani Crabtree
2019 "Combining Geographic Information Systems and Agent-Based Models in Archaeology: A step-by-step guide for using agent-based modeling in archaeological research (Part II of III)." Advances in Archaeological Practice 7 (2).

O'Sullivan, David, and George Perry
2013 *Spatial Simulation: Exploring Pattern and Process*. Wiley-Blackwell, Chichester.

```
; __________________ GLOBAL VARIABLES __________________
turtles-own [ toolkit ]
patches-own [ source? assemblage material_type ]

; __________________ TO SETUP __________________

to setup
 ;;; The setup procedure is run only once at the beginning of each experiment.
 clear-all                        ; remove any residuals of previous experiments

; __________________ 1. Environment Setup __________________
 ;;; setup patches
 ask patches[
  set pcolor white
  set source? false               ; initially all cells are set as having no raw material
  set assemblage []               ; start a list of items that the agent dropped at this location
 ]                                ; (this would be an equivalent to an archaeological 'find spot')

  ;;; setup patches with raw material
  let r 1
  ask n-of 5000 patches [         ; 5000 random patches become raw material sources
    set source? true              ; set the variable source? as true
    set material_type r           ; each will have a different raw material type
    set r r + 1                   ; marked as a number between 1 and 5000
    set pcolor black
 ]
 ; __________________ 2. Agent Setup __________________
  ;;; create the agent and place him on a random patch, set color, size and shape
  crt 1 [
   setxy random max-pxcor random max-pycor
   set color red
   set size 10
   set shape "person"
   set toolkit []                 ; start a list of items that the agent carries

 ]
  reset-ticks                     ; reset the time counter
end                               ; end of the SETUP procedure
```

```
; __________________ TO GO __________________

to go
;;; agent procedure: 1. move to one of 4 neighbouring cells; 2. drop an item from the toolkit if not
empty 3. reprovision the toolkit if a raw material source patch is encountered

  ask turtles [
   ;__________________ 1. Move __________________
    if random 9 > 0 [                    ; if a randomly drawn no between 0 - 9 is higher than 0
      move-to one-of neighbors           ; move to any one of the 8 neighbouring patches
    ]                                    ; otherwise (it is 0) don't do anything
  ]

   ;__________________ 2. Drop a random item from the toolkit __________________
  ask turtles [
    if length toolkit > 0 [              ; if the toolkit is not empty
      let i random length toolkit        ; determine which item (i) will be dropped

      ask patch-here [                   ; add item i to the 'assemblage' of the patch
        set assemblage fput (item i [ toolkit ] of myself) assemblage
    ]

    set toolkit remove-item i toolkit    ; remove the item (i) from the toolkit
   ]
 ]
    ;__________________ 3. Reprovision if on a source patch  __________________
  ask turtles [
    if [ source? ] of patch-here = true [          ; if you come across a patch with raw material
        let raw_material [ material_type ] of patch-here          ; determine the type of raw material
        while [ length toolkit < max_carry ] [     ; while you still have capacity to carry more...
          set toolkit fput raw_material toolkit    ; keep on adding that raw material to your toolkit
      ]
    ]
   ]

  tick                                             ; time + 1

end                                                ; end of the GO procedure
```