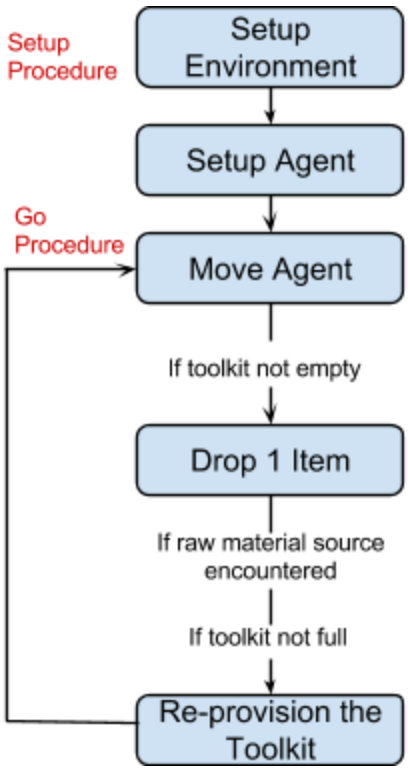
First Steps in NetLogo. Tutorial

This tutorial is based on the 'neutral model' developed by Brantingham. In his 2003 publication 'A Neutral Model of Stone Raw Material Procurement' Brantingham tested a model consisting of one agent moving through a resource landscape collecting raw material and depositing stone tools.

The main premise behind the neutral model is to establish the pattern of assemblage variability under conditions of no behavioural biases (such as preference for certain raw material types or specific type of mobility). The purpose of this paper is to set a benchmark of what we may expect to see in lithic assemblages if the behaviour of their makers was almost entirely random. The agent uses a random walk algorithm to move, collects every type of raw material it encounters and deposits the stone tools indiscriminately. It moves one cell at a time in a Moore radius (cells N, NW, W, SW, S, SE, E, NE of the current cell) and will carry (curate) up to 100 items.

The outline of the simulated processes is given in Fig. 1. In the setup phase a 500x500 cell world is seeded with 5000 different raw material sources. Each raw material type is present at only one cell and their distribution is random--there are 5000 patches with unique raw materials present. 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 can reprovision the toolkit if the toolkit is not full. After all of these actions are completed the cycle (move-drop-reprovision) is repeated until 200 different types of raw material have been encountered. 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.

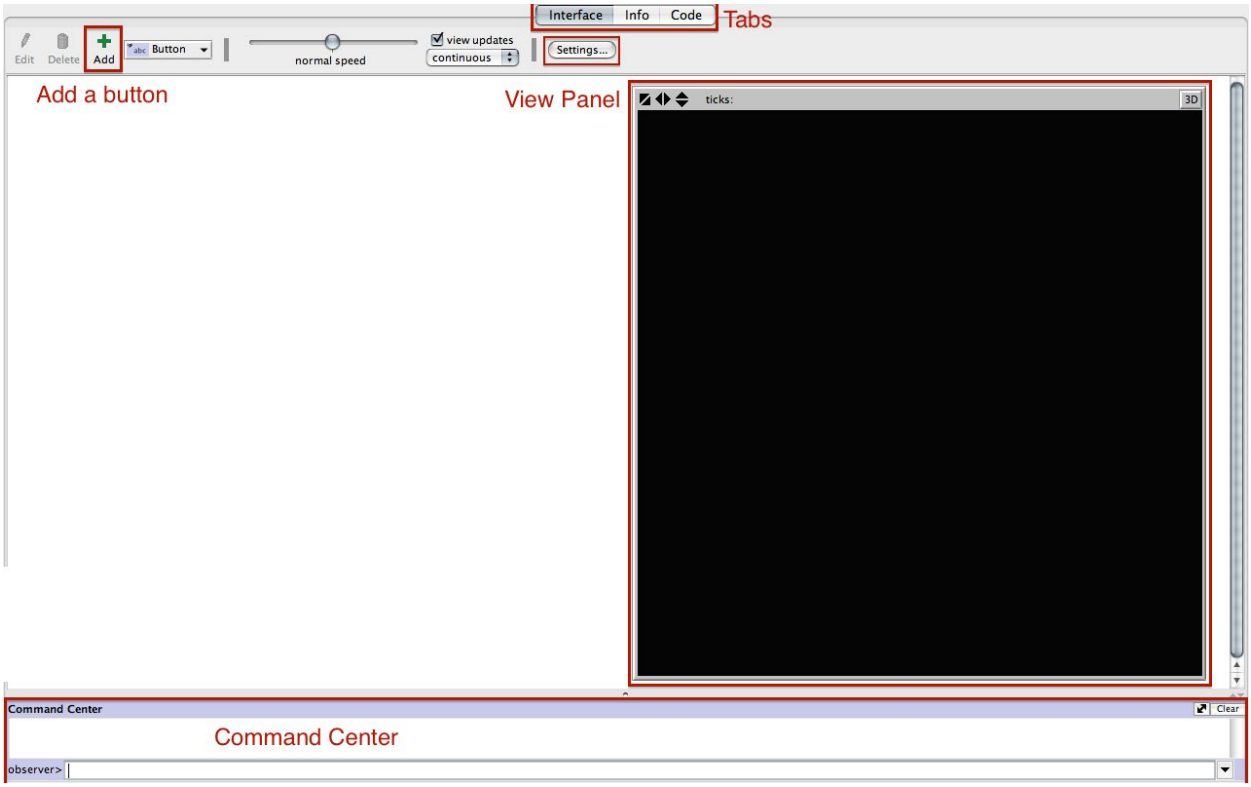


In this tutorial we will try to replicate Brantingham's model as accurately as possible.

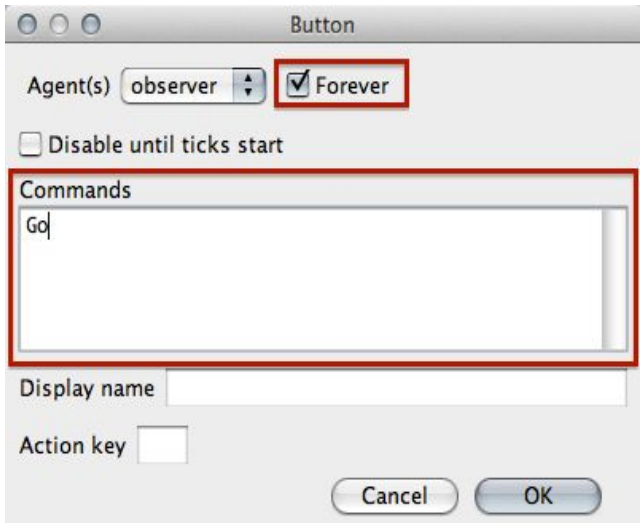
Figure 1 *A schematic outline of the Neutral Model. Adapted and simplified from: Brantingham 2003, fig. 5.*

1. First Steps in NetLogo

The NetLogo interface 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.



We are first going to change the size of our simulation window to accommodate a much larger view than is standard. Click on the 'Settings' button in the top menu. We can setup the view panel here. First change the 'Location of origin:' to 'Corner' and choose 'Bottom left' from the drop down menu--by default the center is the location of origin. 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 are 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. Check that both tick boxes are ticked and hit 'OK'. This will get you back to the main "home" image.



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

Click on the 'Add' button and then click anywhere on the white space. A dialogue box should pop up. Write `setup` in the 'Commands' box and click OK. Follow the

above step to create a second button and write `go` in the 'Commands' box. This time also tick the 'Forever' box. 'Forever' means that this action will repeat until the simulation ends.

You can see that the text on both buttons has instantly turned red - that is because we have not defined what we mean by 'setup' and 'go' yet 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 we will inspect in a moment. Type the following in the code box:

```
to setup
end

to go
end
```

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

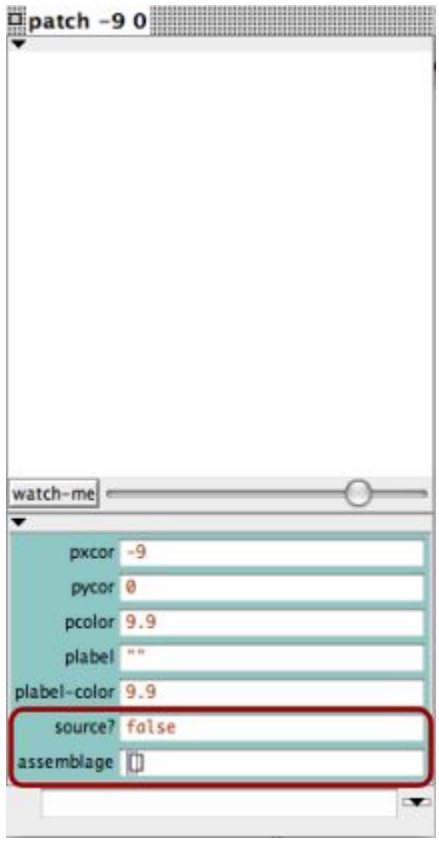
Logo - the language of NetLogo - was developed to resemble a natural language as much as possible, which means that it is very easy to understand the code. It was also developed with educational goals in mind (read: teaching kids), which means that it is equally easy to write. We will start with setting up the environment by asking all patches to set a number of variables. All **commands** in Netlogo are initiated by the word `ask` and enclosed in square brackets `[]`. Type inside the setup procedure (i.e. between `to setup` and `end`**)**:

```
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, meaning it is already defined in the NetLogo library. Check out the NetLogo dictionary (http://ccl.northwestern.edu/netlogo/docs/) 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 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 so the indentation does not matter, but indentation 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. 'Nothing named SOURCE? has been defined.' Indeed, we tried to use a variable `source?` without defining it first. Congratulations on seeing your first error!

There are two types of variables: 1) **global variables**, used throughout the code, which can be simply listed at the beginning of the code (or defined using Interface buttons - we will come back to this), and 2) **local variables**, which are only valid within one procedure, and use the `let` primitive. We will see the use of local variables later on, but the `source?` is a global one - 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 (i.e. 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 the list of patch variables, including some of the built-in ones such as the x and y coordinates, but also the two we have defined ourselves: `source?` and `assemblage`.

It is not very exciting to have a uniform white screen so let's set up the patches that do contain a raw material source. Because we do not want ALL the patches to be a source we will use the `n-of` ('a 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
  ]
```

First thing we do here is to define a local variable `r` and set it up as 1. We then ask 2500 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 the raw material source patches better. Hit the 'Check' button. You may recognise the error message - stick the `material_type` in the list of patch global variables (`patches-own`) at the beginning of the code and move to the Interface tab. Hit the 'setup' button and inspect one of the source patches.

We have now created the environment, but so far we have not created any agents. We actually only need one forager. Type after the patches setup procedures and 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 []
  ]
```

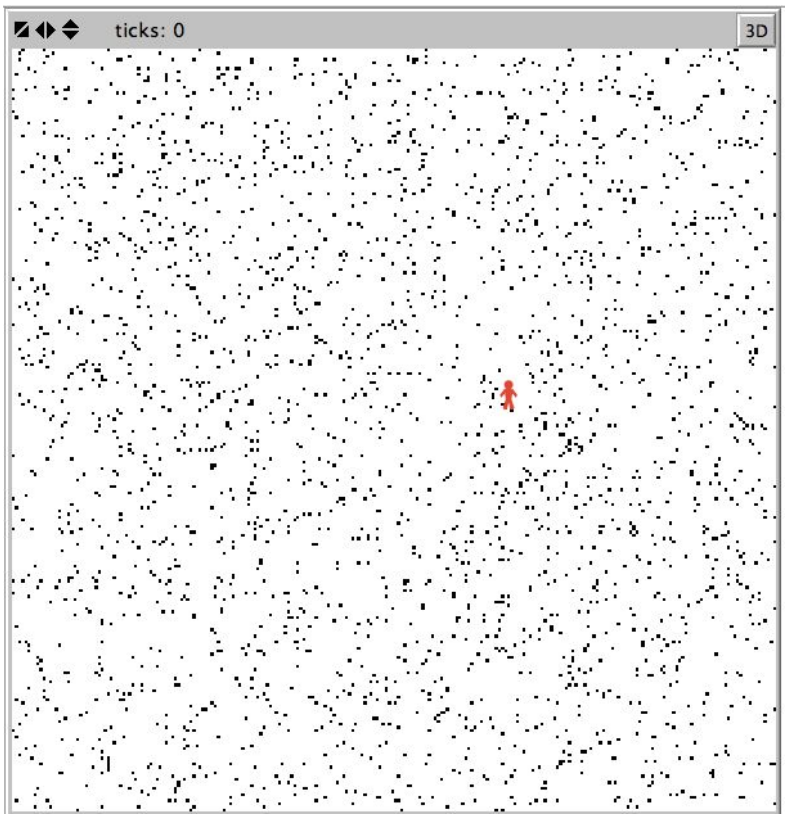
`crt` stands for ‘create a number of agents’, which in our case is one. We then set its initial position in the world to a random cell (i.e., with x and y coordinates between 0 and the current maximum - `max-pxcor` and `max-pycor`) and give it a few variables: color (notice that `color` applies to agents and `pcolor` to patches), size and shape. We also initiate a list of all the stones the agent carries in its toolkit and set up how much the forager 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.

Add the following line at the beginning of the script:

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

Here, we finally dropped the bombshell - in NetLogo jargon agents are called **turtles**. This is the legacy of being primarily 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 the red human-shaped agent on one of the patches. If you keep on pressing it you will see that each time the simulation is restarted, the agent is initialised at a different (random) location.



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 the stop condition is reached (or run forever). 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 by one patch in its Moore radius or stay put. You can put it in a more mathematical term as: the agent has a 1 in 9 chance of staying where it is or moving to any one patch. Similarly to the setup we start the move command with the keyword `ask` and whoever is to perform the task (e.g., all turtles) and enclose a list of functions in brackets. Type inside the go procedure (i.e. between `to go` and `end`):

```
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) to move to one of the neighbours (meaning one of the 8 neighbouring patches) if a random number between 1 and 9 (but not including 9) is drawn. If 0 is drawn, nothing happens - the block of code inside the brackets is not executed, the agent stays put and the program moves on. We also add the tick primitive at the end of the go procedure to move the time counter by one. Go to the Interface tab and click on 'setup' and then on the 'go' button. You should see the red forager 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 does, indeed, move one patch at a time. You might have also notice that the time counter in the top left corner of the view panel 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 (Figure 1) is for the agent to drop one item at each step providing its toolbox is not empty. Write the following code inside the '`ask turtles`' command after the `move` function but before the final closing bracket:

```
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' construction again. This time we will only perform the actions 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 a 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 (`of myself`) to the patch's list of items dropped by the forager - `assemblage`. We then remove the same item from the agent's toolkit. If you run the simulation (make sure you first click the 'setup' button and then the 'go' button) and inspect the agent (to pause the simulation click on the 'go' button, then click it again when you want it to continue), you will notice that despite the heavy coding

we have just done the agent's toolkit remains empty. This is because we have no code to deal with the situation when the agent comes across a patch containing a raw material source. In short, the agent has never got a chance to pick anything up.

We recognise patches which contain a raw material source by their variable `source?` set as `true`. Whenever the agent comes across one of them it can restock the raw material. Type the following code at the end of the `ask turtles` function, but before the last closing bracket:

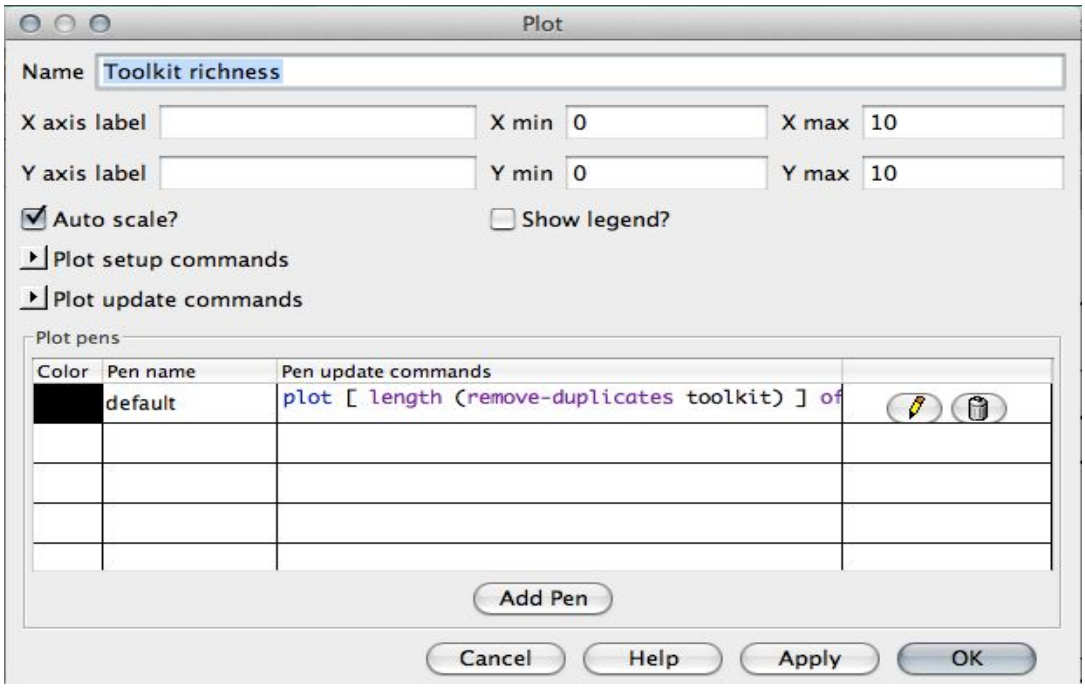
```
if [ source? ] of patch-here = true [
let rm [ material_type ] of patch-here
while [ length toolkit < max_carry ] [
           set toolkit fput rm toolkit
]
]
```

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 defines the `rm` (raw material) as the ID (`material_type`) of the patch the agent is standing on (`patch-here`). We then use a 'while-loop' which adds the `rm` raw material to the `toolkit` list until the maximum capacity (`max_carry`) is reached. Note the difference between the if- and while-loops here. An if-loop will perform an action once if a given condition is fulfilled (e.g., the patch-here is a source or a randomly drawn number is higher than zero). A while-loop will keep on repeating an action until the given condition is reached (e.g., the toolkit length is equal to or exceeds the maximum capacity of the agent).

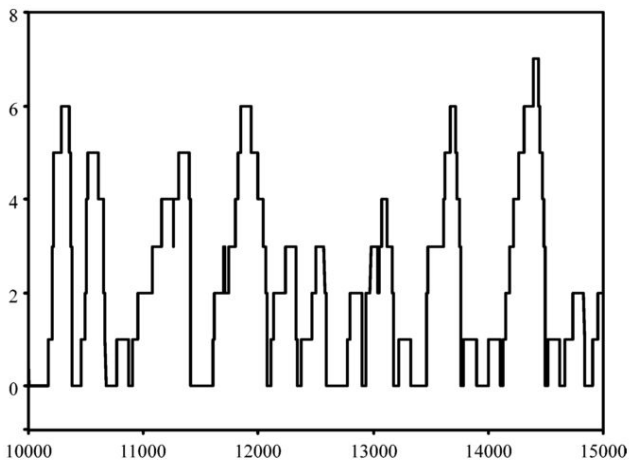
Go to the Interface tab and run the simulation. Inspect the turtle - if you keep the inspect window open during the run you should be able to see how the toolkit changes every time the agent comes across a raw material source patch.

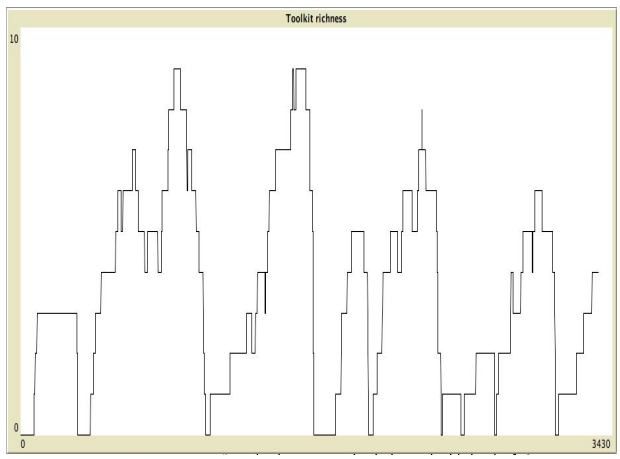
It is a bit difficult to see the raw material assemblage in the small box, so let's create a plot that will show the changes in the frequencies of different raw material types present in the toolkit. Click on the drop-down list next to the 'Add' button at the top of the window and choose 'Plot'. Click 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` is very useful in many cases but since we only have one agent it does not make much sense to use it. Delete the default and type:

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



This will plot the size 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 toolkit of which agent to plot. Run the simulation and see what happens. If you compare your plot with figure 7 in Brantingham 2003 you will notice that they strongly resemble each other. Congratulations, you have successfully replicated the 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. 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'. As mentioned before, global variables can be defined either at the beginning of the script of through the interface. In this case we have already defined `max_carry` at the beginning of the script. Remove `max_carry` from the `turtles_own` globals. Find and

remove it also from the `setup` procedure. If you leave it in the command used to create the agent, it will overwrite the value set on the slider. 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.

Summary of NetLogo

NetLogo is a user-friendly simulation platform commonly used for agent-based modelling in social and natural sciences. It is based on the Logo language originally designed as an educational tool for teaching programming to kids and therefore aims to be a 'low threshold, high ceiling' platform, meaning that it is easy to learn but nevertheless powerful. There are a number of other ABM platforms and simulations can also be built from scratch using any of the general use programming languages. However, NetLogo combines ease of use and quick development with high level capacity and a wide suite of built-in tools commonly used when developing ABMs such as visualizations, automated scenario running, etc. This section will provide a general overview of NetLogo structure.

1.1. Code building blocks

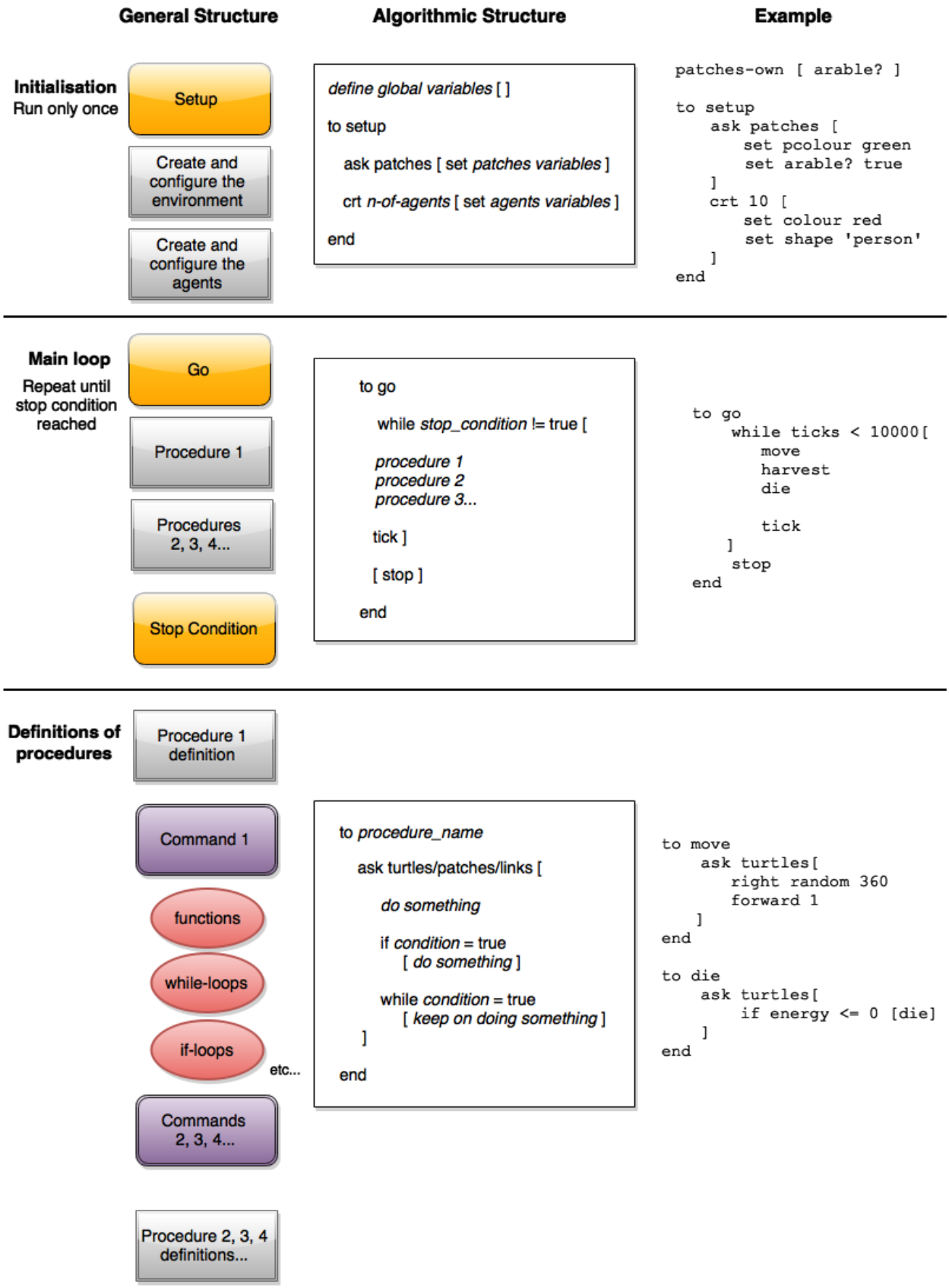
The four main entities in Netlogo are the agents ('turtles'), the grid squares ('patches'), the connections between agents ('links') and the observer - an entity which governs the simulation flow, such as progressing the time counter or scheduling the order of actions. The building blocks in NetLogo consist of procedures, commands and functions (built-in functions are called 'primitives'). Most of NetLogo simulations are composed of two main procedures: **`to setup`** and **`to go`**. In the `setup` procedure the world and the agents are initialised and it is run only once at the beginning of a run. Setup may include loading up the GIS raster, creating the initial population of agents and giving them a set of characteristics (e.g., sex, age, profession). The `go` procedure is the core of a model and consists of a list of actions (often themselves procedures) that are repeated at each time step of the simulation until the stop condition is reached. All procedures come from the observer environment and are defined using the `to` *`procedure_name`* … `end` keywords. In contrast commands are specific to one of the other netlogo entities: turtles, patches or links. Therefore, they are always enclosed between square brackets following the keywords: `ask` *`turtles/patches/links`*.

1.2. Variables

There are two types of variables in NetLogo: global and local variables. Global variables are defined at the beginning of the code and ascribed to one of the NetLogo entities: the observer (`globals []`), agents (`turtles-own[]`), grid squares (`patches-own[]`) or links (`links-own[]`). Buttons, slider, chooser in the Interface tab also define global variables. Global variables can be used throughout the code and accessed by any procedure. In contrast, local variables, defined by the `let` primitive, can only be accessed from within the procedure in has been defined in. For example, the following line of code:

```
let old_turtles turtles with [age > 50]
```

defines `old_turtles` as all turtles whose attribute age is greater than 50.



1.3. Control statements

Similarly to all programming languages NetLogo supports three main types of loops: if-loops, while-loops and for-loops (the latter will be discussed in section 1.4, Lists). If the conditional

statement of an if-loop evaluates to ‘true’ a list of actions specified in the square brackets is performed. For example, the following command:

```
ask turtles[
if energy <= 0 [die]
]
```

can be read as ‘ask each turtle: if your energy is equal or lower than zero - die’. In contrast, the while loop describes a set of actions enclosed by the square brackets to be repeated until the specified condition is reached. For example, this command:

```
ask turtles [
while energy > 0 [move]
]
```

can be read as ‘ask each turtle to move as long as their energy is greater than zero’.

1.4. Lists

A list is an object which stores multiple pieces of information. Lists are useful for keeping track of groups of things where group membership might change over time; for example, a forager’s toolkit. A list can be initiated using the `set` primitive:

```
set example_list [1 2 3]
```

Alternatively, an agentset can be used to construct a list through the primitive `of`, e.g.:

```
set turtle_ages [ age ] of turtles
```

In this case the list `turtle_ages` contains the values of the `age` variable of each turtles. Although lists are immutable, new lists can be created on the basis of existing lists, again using the `set` primitive.

```
set example_list replace-item 2 example_list 5
```

The ‘`2`’ in the example above represents the index of the list, i.e. the position of the item which is to be replaced with the value ‘`5`’. Similarly to most programming languages the indexing in NetLogo starts with zero, i.e., the index of the first element of the list is 0, the second: 1, the third: 2, etc.

The for-loop mentioned above allows the user to perform an action on each element of the list. If we would like to inspect the content of the `example_list` after it was altered we can use:

```
foreach example_list show
```

and the result should be (if you executed the previous example):

```
1
2
5
```

The examples above are intended as a general reference only. We will guide the reader through the process of building a simulation in NetLogo and discuss the code elements in a more comprehensive manner in the tutorial.

1.5. NetLogo resources

There are many freely-available learning resources for ABM and NetLogo on the Internet. NetLogo documentation (NetLogo 2015), which includes tutorials, a programming guide and a full dictionary of NetLogo primitives is usually the first port of call for any technical enquiries. However, there are many other ABM- and NetLogo- dedicated websites, blogs and users groups. **simulatingcomplexity.wordpress.com** run by the authors of this tutorial is a specialized blog on archaeological applications of computational modeling and complexity science. The Special Interest Group in Complex Systems Simulation holds an annual beginner workshop in NetLogo as well as sessions and roundtables at the CAA (Computer Application and Quantitative Methods in Archaeology) conference, while the ESSA (European Social Simulation Association) organizes a week-long summer school in social simulation as a satellite to its annual conference. There are other, domain specific training courses available.

There are a number of university-level textbooks which use NetLogo to show the principles of complex systems modelling in ecology (Railsback and Grimm 2011), geography (O'Sullivan and Perry 2014) and social science (Gilbert and Troitzsch 2005; Miller and Page 2007).

Not all simulations need to be written from scratch. NetLogo comes with a large and growing library of models. With over two hundred ABMs the Models library contains many examples, which, although developed for other disciplines from mathematics and physics to ecology and transport, could be adapted to archaeological research. For instance, epidemiological models simulating the spread of a disease in human populations under different conditions share many characteristics with theoretical models of the diffusion of innovations. In addition, many authors working on archaeological case studies share their model's code after publication in the OpenABM (openabm.org) repository.

To cite this tutorial:

Romanowska, I., Crabtree, S., Davies, B., Harris, K. in prep. 'Agent-based modeling in archaeology: why?, when? and how?' *Advances in Archaeological Practice*.

References

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

Gilbert, Nigel G., and Klaus G. Troitzsch
2005 *Simulation for the Social Scientist*. Open University Press, Maidenhead.

Miller, John H., and Scott E. Page
2007 *Complexity in Social Worlds*. Princeton University Press, Princeton.

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

OpenABM
2014 Open Agent Based Modeling Consortium. A node in the CoMSES Network. https://www.openabm.org

Railsback, Steven F., and Volker Grimm
2011 *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton University Press, Princeton.

Railsback, Steven F., Steven L. Lytinen, and Stephen K. Jackson
2006 Agent-based Simulation Platforms: Review and Development Recommendations. *Simulation* 82(9): 609–623.