SUPPLEMENTAL FILE 1: TUTORIAL - Combining Geographic Information Systems and Agent-Based Models in Archaeology

This tutorial builds on that presented in Romanowska et al. (2019), and the model is based on the work of Brantingham (2003). The simulation is built using NetLogo version 6.0.2 (Wilensky 1999). It is highly recommended that the previous tutorial be completed before attempting this tutorial. Average completion time for this tutorial is two hours.

Setting up a NetLogo model

After downloading and installing NetLogo, open the program. The initial screen has three tabs, the *Interface*, *Info*, and *Code* tabs. Imagining a NetLogo model to be like a computer, the Interface tab is akin to the screen and keyboard, while the Code tab is similar to the internal components of the machine. The Info tab can be used for model description and documentation.

The NetLogo environment possesses several of the basic characteristics of GIS software, in the sense that it keeps track of spatial data in a systematic way, and can be used to create visualizations of spatial phenomena. NetLogo has a gridded "world" window that can be manipulated by right-clicking on the world window in the NetLogo interface and selecting "Edit", allowing the user to set an origin point, maximum and minimum coordinates, patch (cell) size, and whether or not the world "wraps" as a toroid (ring-shaped) or cylinder surface. This should be considered carefully when adding data, as these will influence how data (such as the resolution of raster cells) is translated to the NetLogo world. For this exercise, the settings for the parameters in Table S1 should be changed from the defaults:

Table S1: *Model settings used in example model*

NetLogo world parameter	**Value setting**
Location of origin	Center
max-pxcor	100
max-pycor	100
Patch size	2

Once this is completed, create two buttons: setup and go. Buttons can be creating by clicking the + button and clicking anywhere on the white background of the Interface tab. In each respective window, enter 'setup' and 'go' in the Commands pane, and tick the 'Forever' box in the 'go' button (Fig. S1). The 'setup' and go procedures are used to initialize the model and iterate through its commands, respectively, and are a convention used in many Netlogo models.

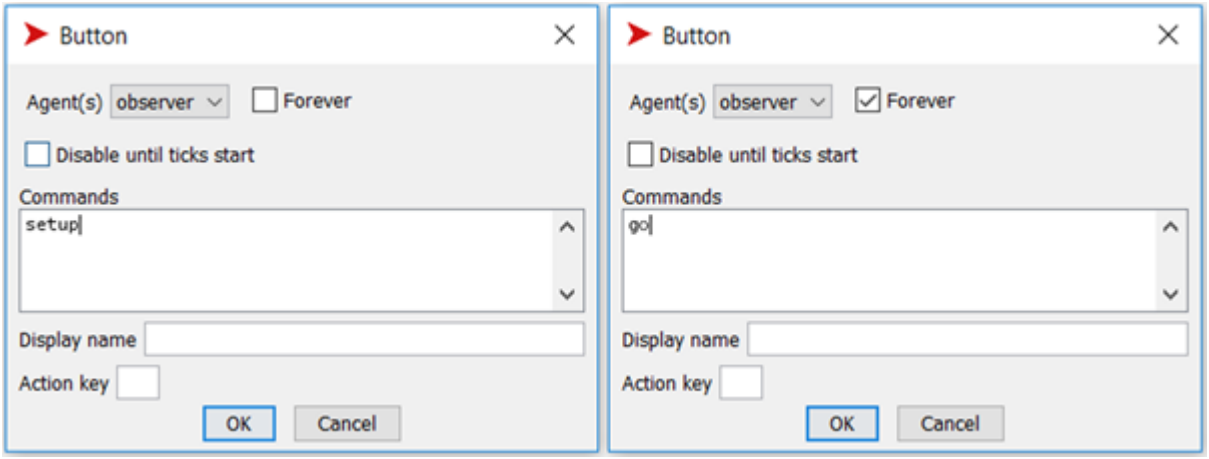

Figure S1: *Creating NetLogo buttons for the 'setup' and 'go' procedures.*

Since we have not yet defined these procedures, the text on these buttons will be red. This will be done in the next section.

Adding GIS data

Support for GIS is available through an extension that comes pre-packaged with NetLogo, requiring no configuration following installation of the core software. The NetLogo GIS extension allows for standard GIS data formats (shapefiles and ASCII rasters) to be added to and manipulated by a NetLogo model. The code and data used in this tutorial are available from an online repository. Make sure that the .nlogo file is saved in the same folder as the GIS data.

The first step toward using the GIS extension is to add the following code to the top of a model's Code tab to access the GIS extension:

```
extensions [ gis ]
```

In order for NetLogo to use GIS data, the data need to be imported into NetLogo. One way to do this is to create a "global" variable (a variable that can be accessed by any other component of the model) where the GIS data can be stored, and then import the data as that variable, as in the following pseudocode:

```
globals [ gis-shape gis-raster ]

to setup
      set gis-shape gis:load-dataset "myshapefiledata.shp"
      set gis-raster gis:load-dataset "myrasterdata.asc"
end
```

In this tutorial, the datasets of interest are locations of quarry sites saved as a point shapefile called "quarries.shp", as well as a digital elevation model (DEM) saved in ASCII format

called “dem.asc”[1]. Save these in the same folder along with the current model. Next, create two new global variables called *quarries* and *elevation*; these will be used to import and store the data in NetLogo in this way:

```
globals [ quarries elevation ]

to setup
      clear-all
      reset-ticks

      set quarries gis:load-dataset "quarries.shp"
      set elevation gis:load-dataset "dem.asc"
      gis:set-world-envelope gis:envelope-of elevation
end
```

First, the global variables are declared. Next, a *setup* procedure is started with the *clear-all* and *reset-ticks* commands used to reset the model and reset the timer, respectively. Next, the *gis:load-dataset* command is used to store the shapefile data in the global variable *quarries*, and the raster data in the variable *elevation*. Finally, the *gis:set-world-envelope* sets the envelope, or maximum extent, of the NetLogo world to match that of the raster dataset. This latter command is important when working with more than one GIS layer as it limits the boundaries of the NetLogo world.

To quickly visualize the data, the following procedure can be used:

```
to show-data
      gis:paint elevation 50
end
```

The *gis:paint command* is used to display raster data shaded from low (dark) to high (light) at a given opacity, where by 0 is opaque and 255 is transparent.Now add the following line at the end of the *setup* procedure:

```
show-data
```

Return to the interface and click the ‘setup’ button. The resulting image in the NetLogo world window displays the raster data, shaded low (dark) to high (light) (Fig. S2). The localities of the quarries can also be added by updating the `show-data` procedure:

```
to show-data
  gis:paint elevation 50
  gis:set-drawing-color red
  gis:draw quarries 3
end
```

[1] The “dem.asc” dataset used in this example is an ASCII raster extracted from tile n34w119 of the USGS National Elevation Dataset, centered on Santa Catalina Island, California. The “quarries.shp” dataset is a shapefile of simulated quarry locations.

The *gis:set-drawing-color* command used here sets the color used to draw the point data, and the *gis:draw* command draws the points in a given size. Now when the 'setup' button is pushed, the red points indicating the locations of the quarries are also displayed (Fig. S2). A visualization like this confirms that the data appear as they should or indicates whether adjustments need to be made to either the size of the world window or the translation of the data in the NetLogo world envelope. Once the data appear as they should, this data can be incorporated into a model by allowing agents and patches to interact with it.

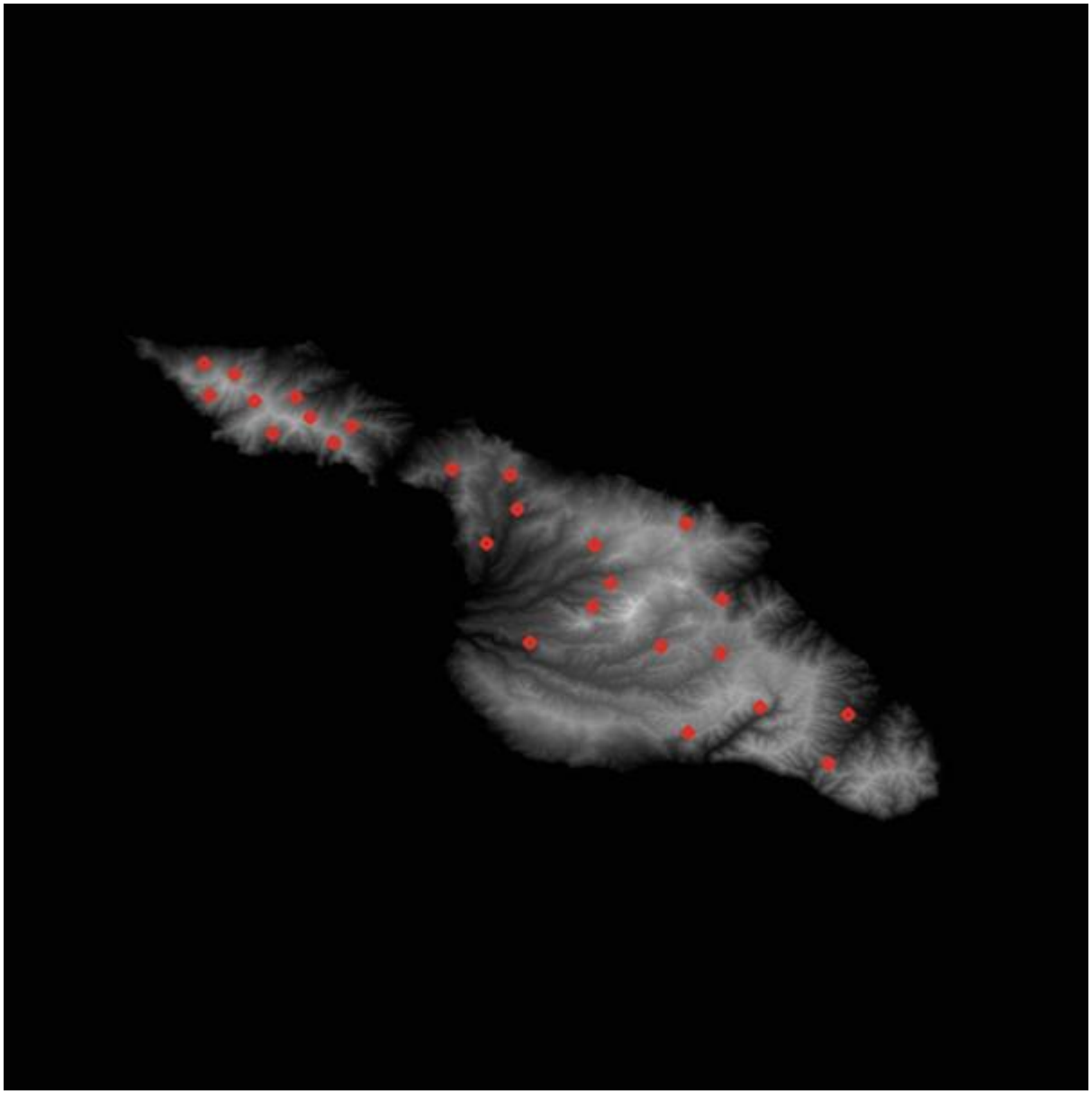

Figure S2: *NetLogo visualization showing imported GIS data. Grayscale background image drawn from ASCII raster digital elevation model. Red dots derived from shapefile simulating locations of raw material quarry sites.*

Getting agents to interact with GIS data

There are two ways to make imported spatial data available to agents in NetLogo: by using GIS data to create or alter entities within the NetLogo world (i.e., agents or patches), or by

using the GIS extension within agent or patch behavioral rules. In the former case, a variable can be created that stores the GIS data within the patches or agents. For example, the raster elevation data from the previous section might be transmitted to the patches in the NetLogo world so that agents can easily determine the elevation at their current location. To this, a 'patch-owned' variable will be needed to store the elevation data for each patch. This can be accomplished by adding the following to the top of the code window, just beneath the extensions:

```
patches-own [ patch-elevation ]
```

In this case, the *patches-own* command declares that any named variable within the following brackets is an attribute of the patches. To translate the GIS elevation data into the individual patches, the following code might be added to the 'setup' procedure:

```
ask patches [
      set patch-elevation ( gis:raster-sample elevation self )
]
```

This uses the *gis:raster-sample* primitive to get raster values from the previously loaded elevation dataset, and the *self* primitive to indicate that patches are to get values from their own locations within the GIS space. This works similarly to extraction tools in a standalone GIS. In this case, the patches extract the value from the raster values occupying the same space. If the patch covers more than one cell in the raster, NetLogo will average the GIS values together. This can be visualized by removing the `show-data` command from the 'setup' procedure and adding the following code to replace it:

```
let mx gis:maximum-of elevation
  ask patches [
      set patch-elevation ( gis:raster-sample elevation self )
      ifelse patch-elevation > 0  [
       set pcolor scale-color green patch-elevation 0 mx
      ]
      [
      set pcolor blue
      ]
  ]
```

The first line of this code creates a temporary variable, *mx*, that holds the maximum value of the GIS elevation data. The rest of the code colors patches with an elevation greater than zero are in shades of green, from darker to lighter with increasing elevation toward the maximum, while patches with an elevation less than or equal to zero are turned blue (in other words, below or at sea level). The visualization resulting from this operation is shown in Figure S3.

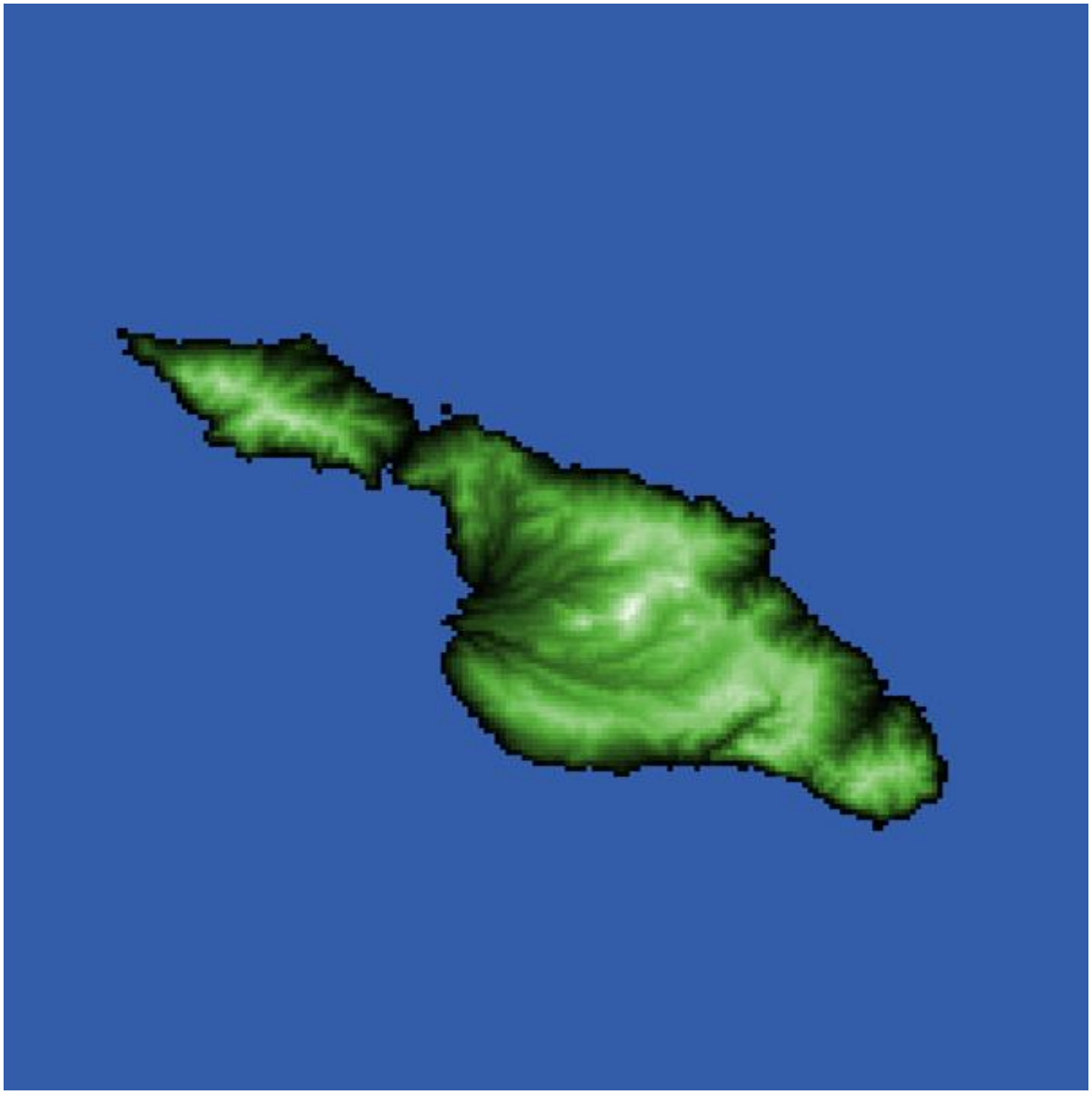

Figure S3: *Visualization of NetLogo example model showing digital elevation model raster values integrated into NetLogo patches. Elevations shaded on a gradient from low (dark) to high (light), between values of 0 and 600m a.s.l.*

To add an agent (turtle) to the example model, the following code can be added to the end of the *setup* procedure:

```
ask one-of patches with [ patch-elevation > 0 ] [
     sprout 1 [
     set size 8
     set shape "person"
     set color red
     set toolkit []
     ]
 ]
```

This code creates an agent on a randomly chosen patch above sea-level (elevation > 0), and sets up attributes such as size, shape, and color. Notice that the agent has an attribute called *toolkit*, which is set up as an empty list using the left and right square brackets. Because the agents (turtles) will be using this attribute individually, it will need to be established as a “turtle-owned” variable by adding the following to the top of the code window:

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

At this point, the agent can now read elevation information directly from the patches and incorporate that information into its behavior. For example, a model could be constructed where the agent moves more slowly through low terrain, which might have implications for the relative amounts of time spent and cultural deposition. First, we can edit the *go* procedure used in the Romanowska et al. (2019) so that agents only move to patches that are above sea-level:

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

In this *go* procedure, once the agent determines to take a step, the agent evaluates the *patch-elevation* of the neighboring patches and moves to one of them with a value greater than 0. Next, agents need to move with more frequency at higher elevations. Something like the following edits to the *go* procedure would have this effect:

```
to go
  ask turtles [
      if random 9 > 0 [
      move-to one-of neighbors with [ patch-elevation > 0 ]
      if [ patch-elevation ] of patch-here > 300 [
      move-to one-of neighbors with [ patch-elevation > 0 ]
      ]
      ]
  ]
  tick
end
```

Here, after the agent makes its first move, it determines if the *patch-elevation* value where it is located is greater than 300 meters and, if so, the agent will move again. In making this determination, and evaluating whether they are above sea-level, agents are drawing on the imported GIS data transmitted to the patches. In doing so, we are implementing a behavioral rule in which the agent uses higher elevation places differently, creating greater probability that the agent will move out of high elevation areas more quickly.

If we are interested in understanding how the threshold between low and high elevation affects the model, the hard-coded value of 300 can be changed to a variable called *elevation-threshold* that will be connected to the Interface using a slider:

```
to go
  ask turtles [
    if random 9 > 0 [
      move-to one-of neighbors with [ patch-elevation > 0 ]
        if [ patch-elevation ] of patch-here > elevation-threshold [
          move-to one-of neighbors with [ patch-elevation > 0 ]
      ]
    ]
  ]
  tick
end
```

Sliders and other Interface tools (e.g. switches, choosers, etc.) allow the user to quickly change the value of a variable without making changes to the code. To create the elevation threshold slider, go to the Interface tab and select “Slider” from the dropdown menu where buttons are generated. Click in the whitespace on the Interface to create the slider, and give this the same name as the variable (e.g. *elevation-threshold*). Finally, select some values for this variable that we might want to explore in terms of a minimum, increment, and maximum (Fig S4).

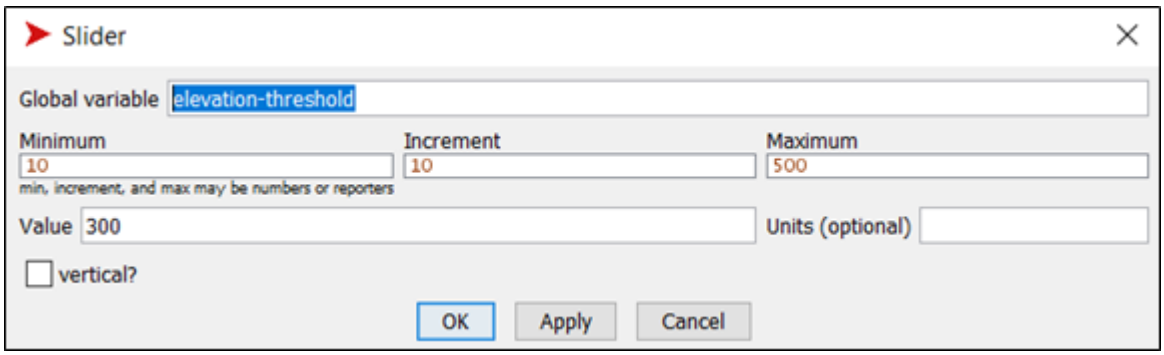

Figure S4: *Creating a NetLogo slider*

Similarly, if we want to examine how the relative degree of mobility between low and high elevation areas affects the model, we can edit the *go* command to use a multiplier variable (or coefficient) called *elevation-multiplier* which will increase the number of times the agent performs the movement process:

```
to go
  ask turtles [
    if random 9 > 0 [
      move-to one-of neighbors with [ patch-elevation > 0 ]
        if [ patch-elevation ] of patch-here > elevation-threshold [
        repeat (elevation-multiplier - 1) [
          move-to one-of neighbors with [ patch-elevation > 0 ]
```

```
        ]
      ]
    ]
  ]
  tick
end
```

The slider created at the front can use any integer values greater than 0 for minimum, maximum, and increment, but for this exercise we'll use 1 for minimum, 1 for increment, and 5 for maximum. The *repeat* command repeats the enclosed code a given number of times, in this case one less than the *elevation-multiplier*. One is subtracted so that it acts as a true multiplier (e.g. if the value of elevation-multiplier were set to 1, then the number of additional moves after the first move would be zero).

It is also possible for agents to interact with GIS data by using commands from the GIS extension within agent. For example, the current model can be adapted so that agents procure stone from quarries when they are nearby. To do this, add the following code to the end of the *go* procedure, just before *tick*:

```
ask turtles [
     if gis:intersects? quarries patch-here [
     reprovision-toolkit
     ]
]
```

Here, the agents uses the *gis:intersects?* command to check whether the quarries dataset (which is a set of GIS points) intersects with the patch where the agent is located. If it does, the agent reprovisions their toolkit. However, as of yet there is no procedure called *reprovision-toolkit*, so it will need to be generated at the bottom of the code window, like so:

```
to reprovision-toolkit
   let t gis:property-value first (filter [ q -> gis:contained-by? q
patch-here] (gis:feature-list-of quarries)) "ID"
     while [ length toolkit < max-carry ] [
     set toolkit lput t toolkit
     ]
end
```

In addition to accessing locations for quarry data, it would useful to know from which quarry the material in the toolkit was procured, so the first line of code within the *reprovision-toolkit* procedure creates a temporary variable called *t* which is used to store the numerical ID value, a property of the shapefile, for the quarry nearest to the agent. To get this value, the *filter* command is used to filter a list of features within the quarries GIS dataset to only those contained by the patch where there agent is located (which is determined using the *gis:contained-by?* command). The *gis:property-value* command is used on the *first* item in the filtered list to pull out the "ID" value for that feature. After this, a *while* loop is used to repeatedly add the ID value *t* to the *toolkit* list until a maximum value (*max-carry*) of items is

reached. The *max-carry* variable, like that featured in the tutorial in Romanowksa et al. (2019), should be instituted as a slider in the model interface using a minimum value of 1, an interval value of 1, and a maximum value of 100.

From here, it is a short step toward a model where agents are generating a simulated archaeological record. First, patches need to be given a patch-owned variable that stores discarded objects, here called ‘assemblage’:

```
patches-own [ patch-elevation assemblage ]
```

Additionally, the following code needs to be added to the *setup* procedure to make *assemblage* an empty list held by each patch:

```
ask patches [ set assemblage [] ]
```

Next, adding the following procedure will allow agents to subtract tools from their kit and add them to the assemblage of the underlying patch:

```
to consume-material
  if length toolkit > 0 [
    let i random length toolkit
    ask patch-here [
      set assemblage lput (item i [ toolkit ] of myself) assemblage
    ]
  set toolkit remove-item i toolkit
  ]
end
```

If the agent has any items in its *toolkit* list, this procedure selects a random item from the list, adds that item to the *assemblage* list, and then removes the item from the *toolkit* list. By adding *consume-material* procedure after each call of *move-to one-of neighbors with [patch-elevation > 0]* within the *go* procedure, agents discard a tool at each stop. At this stage by clicking the ‘setup’ and ‘go’ buttons, this model will run this process without stopping. The *go* procedure should now look like this:

```
to go
  ask turtles [
    if random 9 > 0 [
      move-to one-of neighbors with [ patch-elevation > 0 ]
      consume-material
        if [ patch-elevation ] of patch-here > elevation-threshold [
        repeat (elevation-multiplier - 1) [
          move-to one-of neighbors with [ patch-elevation > 0 ]
          consume-material
        ]
      ]
    ]
  ]
```

```
  ask turtles [
      if gis:intersects? quarries patch-here [
      reprovision-toolkit
      ]
]

  tick
end
```

Adding the following code to the end of the *go* procedure will stop the simulation after 100,000 time steps (ticks), while displaying the relative densities of ‘tools’ in the patch-level ‘assemblages’:

```
if ticks = 100000 [
  let mx max [ length assemblage ] of patches
  ask patches with [ length assemblage > 0 ] [
    set pcolor scale-color red (length assemblage) 0 mx
  ]
  stop
]
```

Here, a temporary variable *mx* is used to hold the size of the largest assemblage (*length assemblage*) among all patches. Patches with any artefacts in their assemblages are then recolored red, shaded from dark (few artefacts) to light (many artefacts) (Fig. S5), with a maximum value equal to *mx* in order to allow for comparisons between between runs when assemblage size might vary widely:

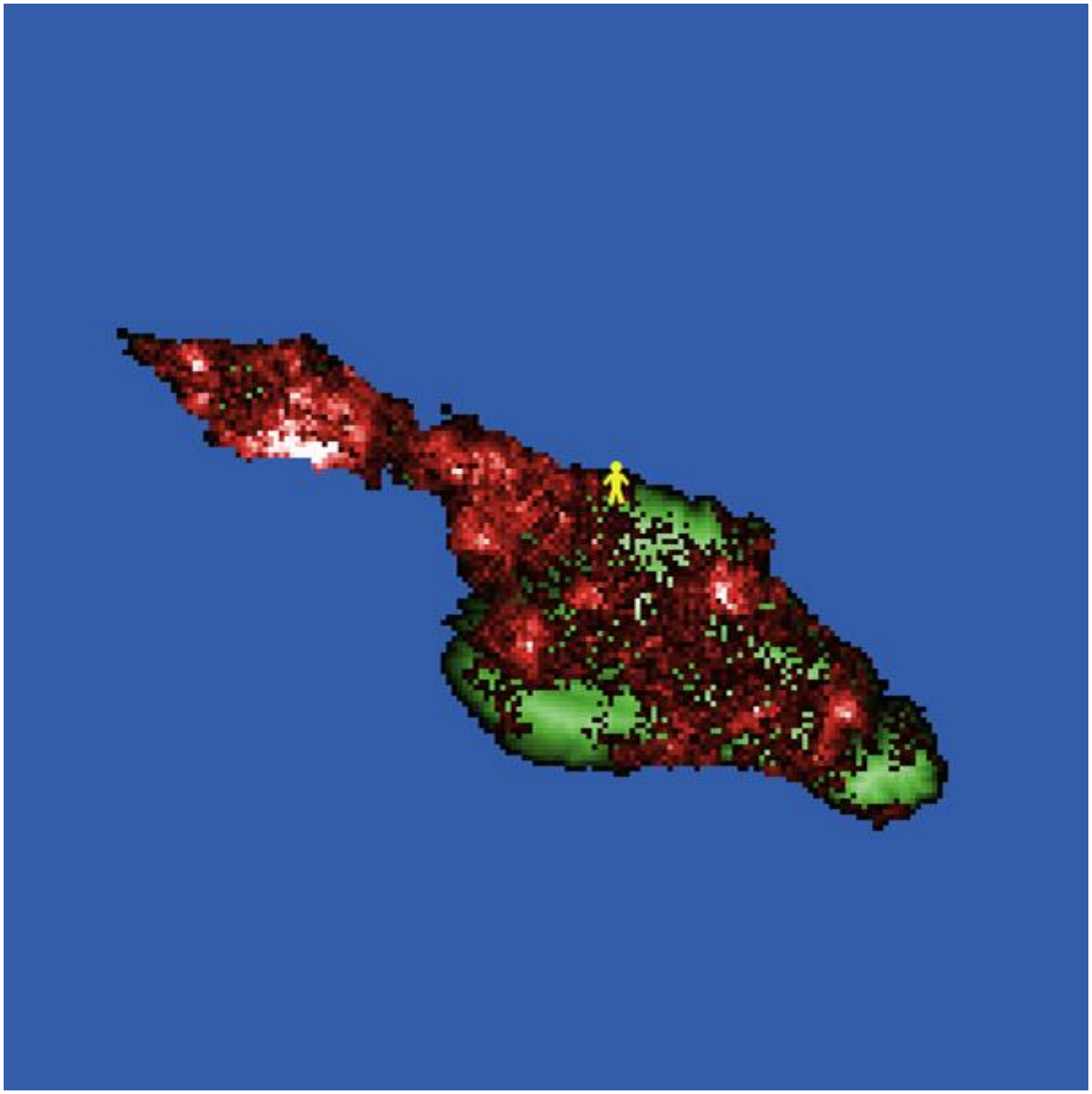

Figure S5: *Visualization of NetLogo example model showing agent (yellow figure) and densities of 'tools' (dark red = low, light red = high) after 100,000 time steps.*

However, if the duration of the simulation was something that we wanted to explore, it can be converted to a variable (*time-limit*) that could be added as a slider in the model interface:

```
if ticks = time-limit [
  let mx max [ length assemblage ] of patches
  ask patches with [ length assemblage > 0 ] [
    set pcolor scale-color red (length assemblage) 0 mx
  ]
  stop
]
```

Finally, we can export data from the model for use in a GIS environment. To do this, first create a new patch-owned variable and call it *diversity*:

```
patches-own [ patch-elevation assemblage diversity ]
```

Next, go to the Interface tab and add a button. Call the button “write raster”, and add the following code:

```
ask patches [ set diversity length remove-duplicates assemblage ]
let raster gis:patch-dataset diversity
gis:store-dataset raster "diversity.asc"
```

When the button is pressed, each patch takes the assemblage list and uses *remove-duplicates* to reduce it to only the unique values in it (for example, a list [3 3 3 3 5 5] would become [3 5]), and records the number of values in that list as *diversity*. These values are then converted into a raster dataset at the same resolution as the NetLogo patches, and these are then saved as an ASCII file called “diversity.asc”. This ASCII file can then be imported into any GIS environment capable of handling this data type (e.g. ArcGIS, QGIS, etc) and used to generate maps (Figure S6) or into other spatial analysis software.

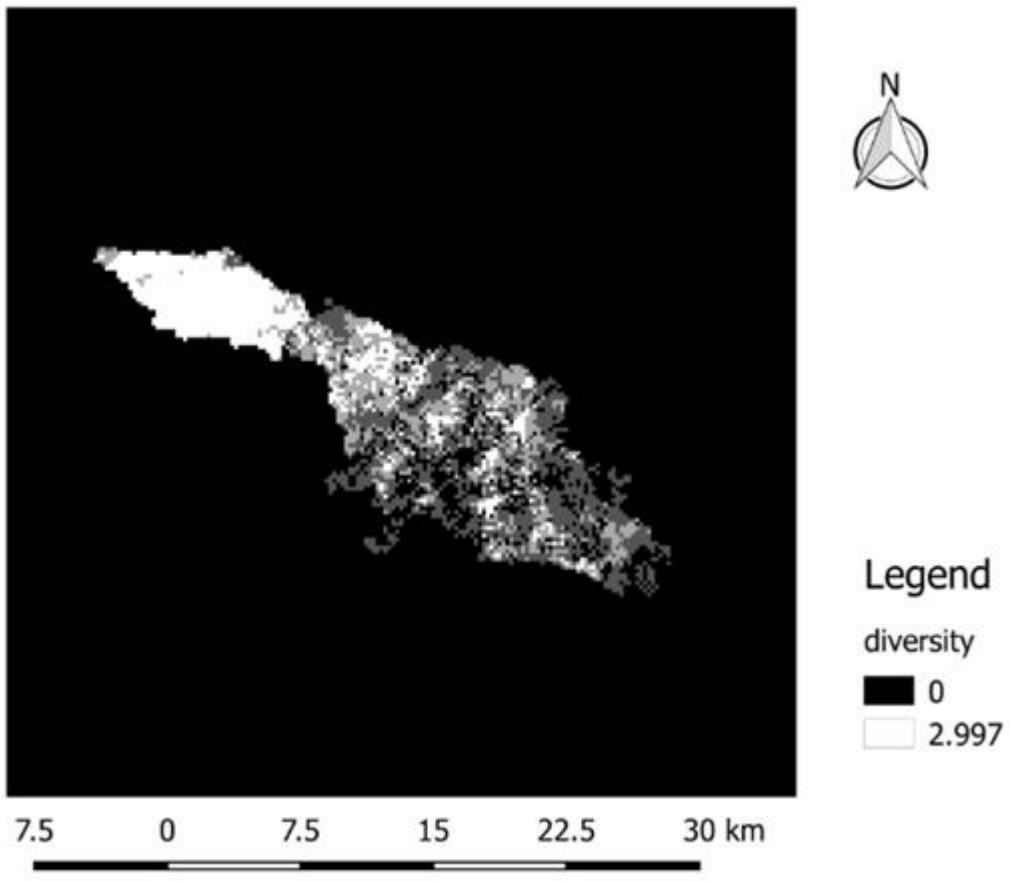

Figure S6: *Visualisation of ‘diversity.asc’ output file as a GIS-generated map*

References

Brantingham, P. Jeffrey
2003 A neutral model of stone raw material procurement. *American Anthropologist* 68(3):487–509.

Romanowska, Iza, Stefani Crabtree, Kathryn Harris, and Benjamin Davies.
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):xxx-xxxx

Wilensky, Uri
1999 *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University (http://ccl.northwestern.edu/netlogo), Evanston, IL.

Code (comments preceded by semi-colons)

```
; ADD GIS EXTENSION
extensions [ gis ]

; GLOBAL VARIABLES
globals [ elevation quarries base orth ]

; AGENT (TURTLE) VARIABLES
turtles-own [ toolkit ]

; PATCH VARIABLES
patches-own [ patch-elevation assemblage material-type diversity]


; SETUP PROCEDURE
to setup

  ; clear model
  clear-all

  ; use coordinate system associated with raster dataset
  gis:load-coordinate-system "dem.prj"

  ; load elevation data from ascii raster
  set elevation gis:load-dataset "dem.asc"
  ;  load lithic source data from point shapefile
  set quarries gis:load-dataset "quarries.shp"
  ; resize the world to fit the patch-elevation data
  gis:set-world-envelope gis:envelope-of elevation

  ; add elevation data to patch data and color accordingly
  let mx gis:maximum-of elevation
  ask patches [
      set patch-elevation ( gis:raster-sample elevation self )
      ifelse patch-elevation > 0  [
      set pcolor scale-color green patch-elevation 0 mx
      ]
      [
      set pcolor blue
      ]
  ]

  ; create an agent with an empty toolkit list
  ask one-of patches with [ patch-elevation > 0 ] [
      sprout 1 [
      set size 8
      set shape "person"
```

```
      set color yellow
      set toolkit []
      ]
  ]

  ;give patches empty assemblage lists
  ask patches [
      set assemblage []
  ]


  ;reset time
  reset-ticks

end

; GO PROCEDURE
to go

  ; move agents based on elevation
  ask turtles [
      if random 9 > 0 [
      move-to one-of neighbors with [ patch-elevation > 0 ]
      if [ patch-elevation ] of patch-here > elevation-threshold [
      repeat (elevation-multiplier - 1) [
            move-to one-of neighbors with [ patch-elevation > 0 ]
      ]
      ]
      ]
  ]

  ; if any quarries nearby, reprovision
  ask turtles [
      if gis:intersects? quarries patch-here [
            reprovision-toolkit
      ]
  ]

  ; if any tools in toolkit, discard
  ask turtles [
      if length toolkit > 0 [
      discard-tools
      ]
  ]

  ; if simulation reaches 100,000 time steps, recolor patches
  ; based on artifact density and stop simulation
  if ticks = time-limit [
```

```
      ask patches with [ length assemblage > 0 ] [
      set pcolor scale-color red length assemblage 0 30
      ]
stop
  ]

  ; advance time
  tick

end


; PROCEDURE FOR REPROVISIONING
to reprovision-toolkit
  ; stores the ID of a nearby quarry as a temporary variable t
  let t gis:property-value first (filter [ q -> gis:contained-by? q
patch-here] (gis:feature-list-of quarries)) "ID"

  ; repeatedly adds value t to toolkit until toolkit is full
  while [ length toolkit < 100 ] [
      set toolkit lput t toolkit
  ]
end

; PROCEDURE FOR DISCARDING TOOLS
to discard-tools
  ; selects random item from toolkit
  let i random length toolkit

  ; adds that item to the local assemblage
  ask patch-here [
      set assemblage lput (item i [ toolkit ] of myself) assemblage
  ]

  ; removes that item from toolkit
  set toolkit remove-item i toolkit
end
```

Model Interface