

## Network structures and assembling code in Netlogo, Tutorial

By Tom Brughmans

First version: Summer 2018

This version created 01/09/2018

Netlogo version used: 6.0.1

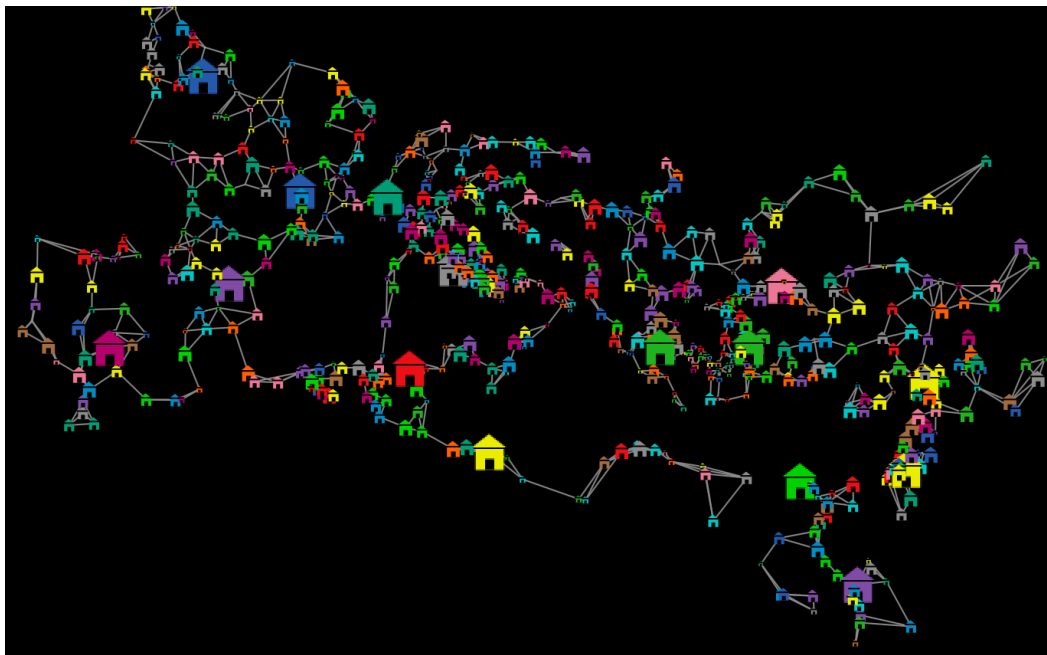
Extension used: nw (pre-packaged with Netlogo 6.0.1)

<https://ccl.northwestern.edu/netlogo/6.0-BETA1/docs/nw.html>

Cite this tutorial as:

Brughmans, T. (2018). Network structures and assembling code in Netlogo, Tutorial,

<https://archaeologicalnetworks.wordpress.com/resources/#structures> .



## 1. Introduction

This tutorial provides an introduction to finding and assembling pre-existing code to quickly create complex models. It uses code and data linked to in the <https://projectmercury.eu> pages. We will create a Roman transport network by reusing existing code that draws on the open access ORBIS dataset (<http://orbis.stanford.edu/>), we will create alternative network structures by reusing existing code, and we will explore the impact these different network structures have in light of simple economic processes. This tutorial will reveal the importance of not reinventing the wheel, of searching for appropriate existing code and letting your model building be inspired by others' previous work.

## 2. Conventions, tips and assumed knowledge

This tutorial assumes basic knowledge of Netlogo and of simulation. It is recommended to walk through the introductory tutorials on the Netlogo website or the tutorial on Netlogo for archaeologists on the [Simulating Complexity blog](https://simulatingcomplexity.files.wordpress.com/2014/07/dispersal_tutorial.pdf):

[https://simulatingcomplexity.files.wordpress.com/2014/07/dispersal\\_tutorial.pdf](https://simulatingcomplexity.files.wordpress.com/2014/07/dispersal_tutorial.pdf)

<https://ccl.northwestern.edu/netlogo/docs/>

This tutorial will also refer to some network science jargon, concepts and techniques. You can get a basic definition of all of these from the glossary on my blog:

<https://archaeologicalnetworks.wordpress.com/resources/#glossary>

Good introductions to network science include the following:

Brandes, U., Robins, G., McCranie, A., & Wasserman, S. 2013. What is network science? *Network Science* 1(01): p.1–15.

Newman, M.E.J., 2010. *Networks: an introduction*, Oxford: Oxford University Press.

Code written in this tutorial will be formatted as in the following example:

```
breed [nodes node]
```

Save your project! Do this regularly and use multiple versions throughout the tutorial so that you can fall back on an earlier version at any time. I will remind you regularly throughout the tutorial to save your project.

**If you are familiar with Netlogo: skip to section 6!**

### 3. Download and install Netlogo

Netlogo is open source software and can be downloaded free of charge for Windows, Mac OS X and Linux.

Go to <https://ccl.northwestern.edu/netlogo/download.shtml> .

Download the Netlogo installer (this tutorial uses version 6.0.1).

Run the installer and install Netlogo.

### 4. Netlogo resources, manual and interface

Netlogo has great documentation about all its features and code in its user manual:

<https://ccl.northwestern.edu/netlogo/docs/>

This manual includes tutorials, a reference to the software functions, a dictionary to its programming language, documentation of its extensions and much more.

Additional external resources can be found on the resources page:

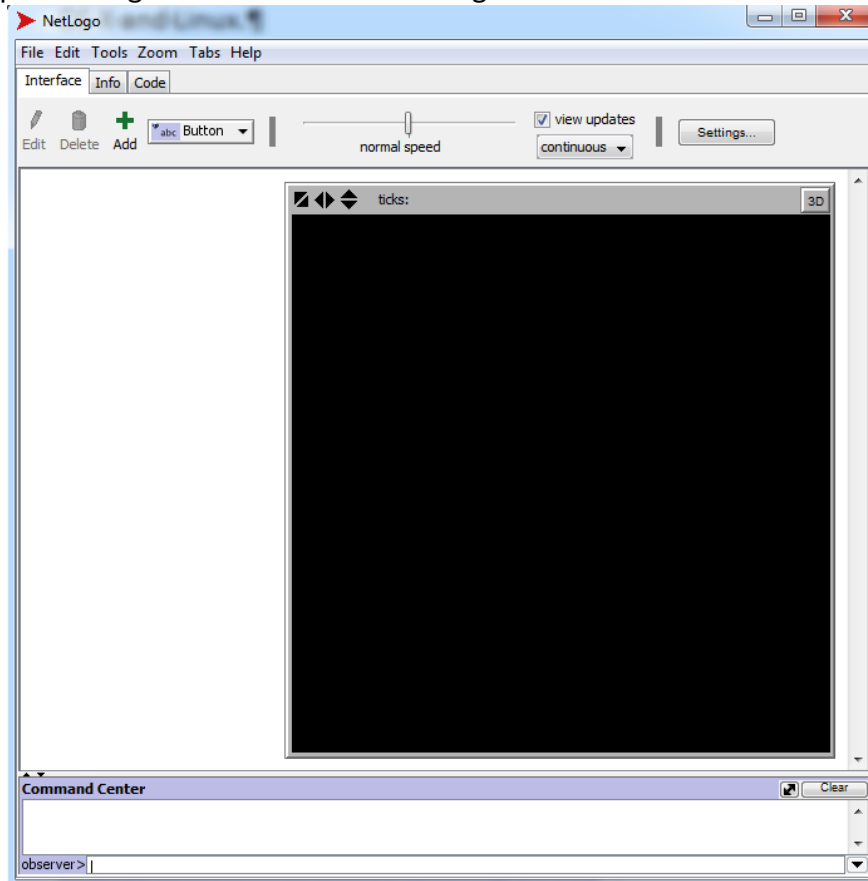
<https://ccl.northwestern.edu/netlogo/resources.shtml>

A detailed reference to Netlogo's Interface can be found here:

<https://ccl.northwestern.edu/netlogo/docs/>

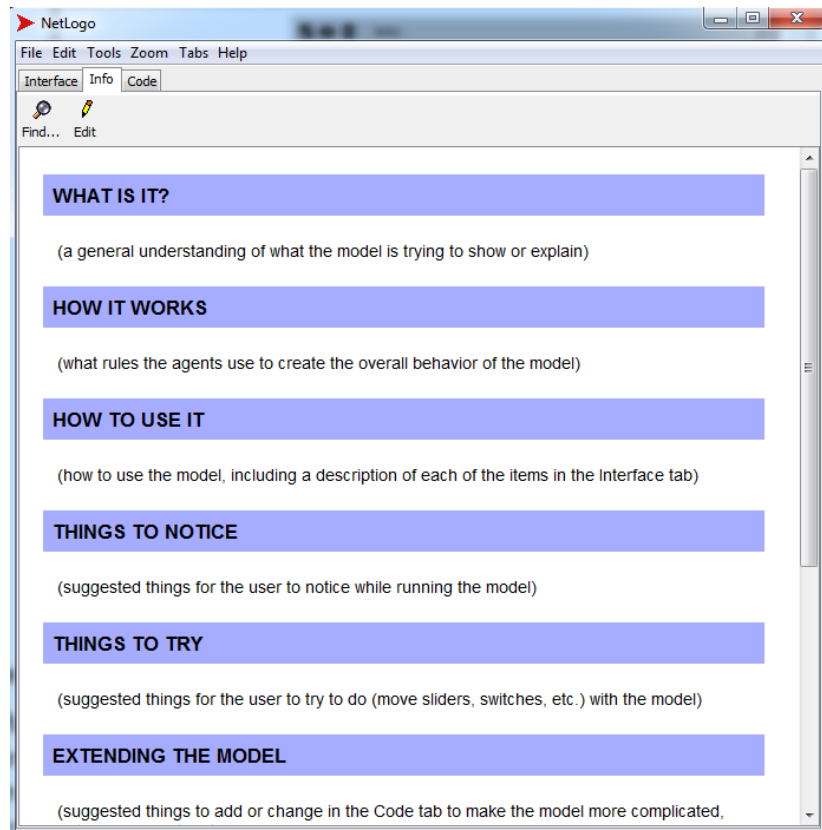
This tutorial will only give a very brief introduction to the key elements of the Netlogo interface you will be using throughout the tutorial.

When you open Netlogo it should look something like this:

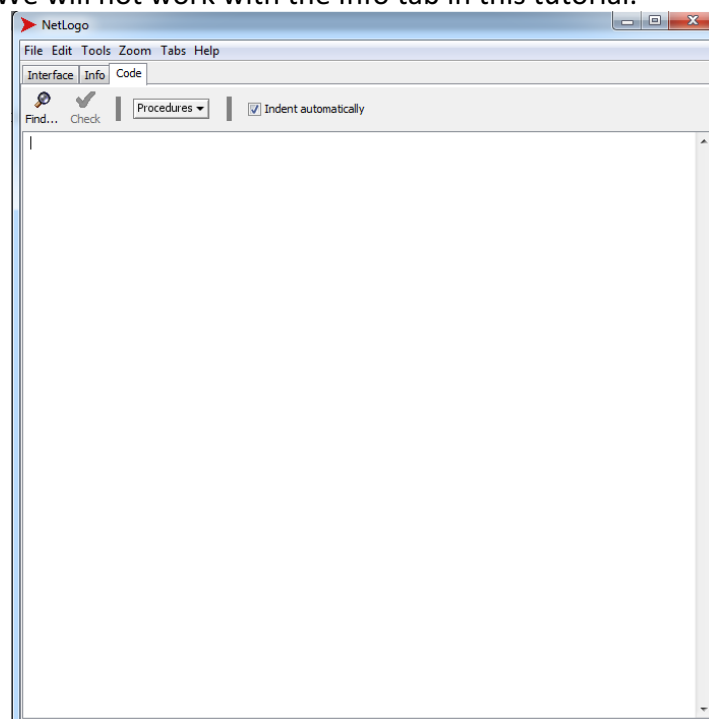


It has three tabs: Interface, Info, Code.

The **Interface tab** is where you watch your model run. Throughout the tutorial you will add buttons and sliders to control the variables of your model, and you will add monitors and plots to inspect what your model is doing. You can speed up or slow down the simulation using the speed slider at the top of the interface. The command center at the bottom will display messages you ask the model to produce, and you can also use it to give to commands to the model from the interface tab.



The **Info tab** is where you describe your model using a standardized set of questions. Adding this information when you share your model with other is crucial to enable them to work with your model. We will not work with the Info tab in this tutorial.



The **Code tab** is where you write and store the code for the model. In this tutorial we will be mainly working in the Code tab. A useful feature is the Check function at the top of this tab: click this to let NetLogo check your code for errors. If it finds errors then you will be guided to the error and asked to resolve it before you can continue, if it does not find errors then you can proceed with coding or viewing your model in action. NOTE: this error checker only

checks whether the primitives used and the order of the code comply with the Netlogo rules (i.e. the code's vocabulary and grammar). It will not check whether the code does what you want it to do, so getting no errors is no guarantee that the code works the way it should or the way you think it does. The error checker will always need to be used alongside other error checking techniques, like reporting variable values and checking them against expectations, or testing submodels independently.

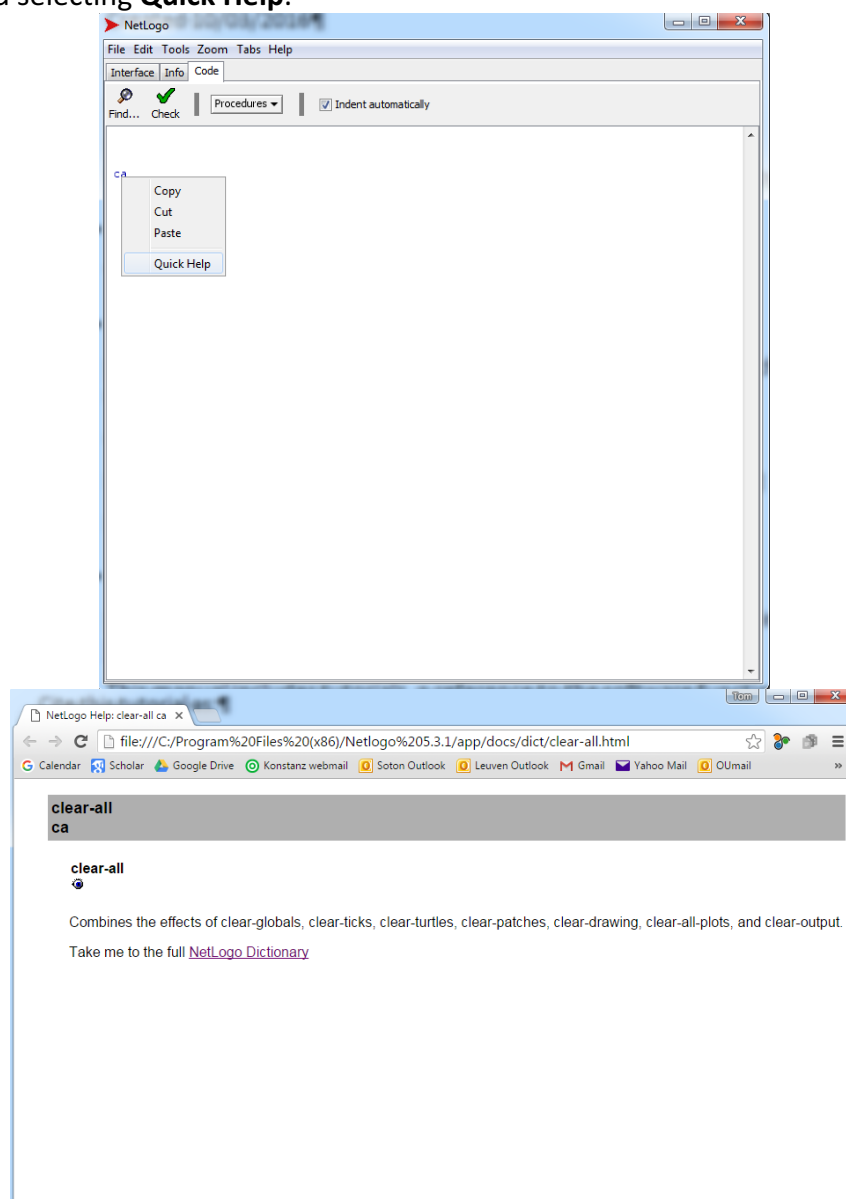
## 5. Netlogo dictionary

A crucial resource when coding in Netlogo is its dictionary:

<https://ccl.northwestern.edu/netlogo/docs/>

For this tutorial, you will find the section on 'Links' in the Netlogo dictionary particularly useful, as well as the documentation of the 'nw' extension.

You can get direct access (even offline) to the entry about a particular primitive by right-clicking it and selecting **Quick Help**.



## 6. Create a working folder

Make a folder on your computer called 'network\_structures'. All code and data we will use and the model we will create will be saved in this folder. When importing data into Netlogo models, as we will do in this tutorial, it is crucial that all data used in the model is saved in the same location as the model itself.

## 7. Why assemble code?

When writing an academic paper, the act of copying entire paragraphs from other scholars' papers into your own paper is called plagiarism. In software development, however, it is standard practice to share snippets of computer code or entire software programmes, and allow colleagues to copy parts of it to make new original software. This process allows software developers to quickly overcome technical code-related roadblocks, to develop new software really fast and to build huge programming resources as a community effort. Of course, it is also considered good practice to reference important bits of copied code in your own code, and to make your own code openly accessible to the community.

This tutorial aims to teach you how this can be usefully done for the case of creating agent-based models of the Roman economy, by using resources from <https://projectmercury.eu>.

Formal modelling in Roman economy studies is not common practice and the community of coders in this subdiscipline is rather small. The development of critical, useful and more models would be seriously enhanced if code sharing, reuse and accreditation become more common practice. Moreover, our small community can draw on the many resources created by other communities within which agent-based modelling is more common practice, such as behavioural economics and the social sciences in general. Project MERCURY hosts a model library listing formal modelling code in Roman studies, and a particularly useful resource is the COMSES OpenABM repository of social science models (which includes a few archaeology models and a huge number of economics models):

- <https://projectmercury.eu/model-library/>
- <https://www.comses.net>

It is crucial to realise that in computational modelling of Roman economy hypotheses the aim is not to develop 100% original and elegant computationally efficient code. We are not computer scientists, but archaeologists and historians: we aim to make appropriate formal computational representations of key hypotheses in Roman studies. Being able to do this faster and more critically on a technical level by drawing on the resources of a coding community is invaluable in achieving this aim. It allows Romanists to focus on their expertise, which is the appropriate representation and study of Roman data and theories, rather than being bogged down in the technicalities of writing good code.

## 8. Our hypothesis: Roman networks

For this tutorial we will study the following abstract hypothesis:

The Roman imperial transport system structured the flow of goods in such a way that wealth inequality between urban communities throughout the empire was enforced.

This hypothesis is formulated with the aim of better understanding the effects of the physical routes connecting urban settlements during the Roman Empire on the flow of goods. Did some urban settlements occupy a position on the Roman transport system that allowed their communities to access and aggregate far more goods than other communities? Is this phenomenon of wealth inequality a particular feature of the Roman transport system structure, or can other network structures enforce more extreme wealth inequality much faster?

Because this is only a tutorial, we will not specify or debate this hypothesis any further here. However, it is crucial to realise that in actual research of the topic such hypotheses would emerge through literature review and data analysis. It is always crucial to provide arguments why a hypothesis is feasible and how it differs from other previously formulated hypotheses. The results of formal computational modelling studies of such hypotheses will always need to be reinserted into these substantive debates, otherwise we are just making computational models for the sake of it.

For now, consider this an interesting hypothesis that lends itself well to being explored through formal computational modelling. We will focus the rest of this tutorial on how the hypothesis can be quickly and appropriately represented using the code and data accessible via <https://projectmercury.eu>.

## 9. A Roman transport network

The first thing we need to do is identify a dataset we can use to represent the Roman imperial transport system. We will use ORBIS, the Stanford Geospatial Network Model of the Roman World (<http://orbis.stanford.edu>). It is a coarse-grained representation of the major terrestrial and maritime routes connecting the major settlements across the Roman world. Check out the ORBIS website if you want to learn more about it, or read the following publications:

- [http://orbis.stanford.edu/assets/Scheidel\\_64.pdf](http://orbis.stanford.edu/assets/Scheidel_64.pdf)
- [http://orbis.stanford.edu/assets/Scheidel\\_59.pdf](http://orbis.stanford.edu/assets/Scheidel_59.pdf)

What is most crucial for this tutorial is that the ORBIS model can be reused with accreditation for research purposes. It offers a great starting point for studies of the Roman world that concern flows of goods, ideas and people.

In this tutorial we will use a network data representation of ORBIS in the .graphml format. It was created from the data made openly accessible by the ORBIS team, listed here: <https://projectmercury.eu/datasets/#orbis>

## 10. Netlogo code implementing ORBIS

ORBIS is the only digital data representation of the entire Roman world during imperial times that can be easily reused for computational modelling work. As such, it has been widely used by Romanists, in particular for agent-based models coded in Netlogo. There is therefore no need for us to figure out from scratch how to import the model into Netlogo, it would be much easier to find a pre-existing Netlogo model that does this, critically assess it and copy the relevant code.



In this tutorial we will use code created by project MERCURY. Go to the MERCURY model library and navigate to the mode 'ORBIS: importing a Roman transport system':

<https://projectmercury.eu/model-library/>

This will take you to the COMSES OpenABM model repository where this code archived.

Before downloading this resource, it is crucial to check whether any information is provided about the reuse license. Scroll down and click on the 'Details' tab. You will see the license is specified as CC-BY-3.0 which means you can copy, redistribute, remix, transform, and build upon the material for any purpose: as long as you give appropriate credit.

<https://creativecommons.org/licenses/by/3.0/>

Download the model:

The screenshot shows a model page with the following details:

- Title:** Importing a Roman transport network
- Submitted by:** Tom Brughmans
- Platform:** netlogo 6.0.1
- Version:** 1.0.0
- Language:** NetLogo
- Tags:** roman archaeology, complex networks
- Summary:** A draft model teaching how a Roman transport model can be imported into Netlogo, and the issues confronted when importing and reusing open access Roman datasets. This model is used for the tutorial.
- Release Notes:** Draft version of model for the following tutorial: Brughmans, T. (2018). Importing a Roman Transport network with Netlogo, Tutorial, <https://archaeologicalnetworks.wordpress.com/resources/#transport>.
- Metadata:**
  - Authors:** Tom Brughmans
  - Publish Date:** Sunday, September 30, 2018
  - Last Updated:** Sunday, September 30, 2018
  - Downloads:** 1
  - Status:** Not reviewed
- Action:** A yellow 'Download' button is highlighted with a red oval.

You will download a folder which holds a few subfolders. Copy the following files into your working folder 'network\_structures':

- From the folder 'code' copy 'Roman-transport.nlogo'. This is the model from which we will copy code.
- From the folder 'data' copy 'orbis.graphml'. This is the orbis network dataset which is required as input data for the model to work.

Now that you've downloaded some code and data, it's time to put your academic hat back on and critically assess the information you downloaded. Open the model 'Roman-transport.nlogo' in Netlogo and familiarize yourself with the code. Check what procedures are included and how they work, identify which procedures you need to copy and check that there are no errors, explore how this code will need to be modified to work for your current purposes. Apply the same scrutiny to the downloaded dataset 'orbis.graphml'. What source was the data derived from, can it be reused, how was it manipulated, does it represent the kind or information you need?

We will assume for the sake of this tutorial that the code and data are appropriate and useful resources for making your own model. So now you have your first building blocks, let's start building your model!

## 11. Copying network import procedures

Create a new model in Netlogo and save it in your folder as 'network\_structures.nlogo'.

Open the model 'Roman-transport.nlogo' you just downloaded in Netlogo (note that if you are working on a Windows machine you can simply open multiple instances of Netlogo simultaneously to compare code. However, if you are working on a Mac you will need to have multiple instances of the Netlogo application installed in order to open multiple models at the same time. To do this, go to the applications folder and copy/paste the folder 'Netlogo 6.0.1'. You can now open both the original application of Netlogo and the copied application).

We will now review the code and list all elements that we need for our own model:

- The first few lines from extensions to turtles-own all list code that is required to import the network dataset, so we will need to copy this.
- The setup procedure includes the actual code that imports the dataset, we obviously need that. We definitely need the procedure 'data-correction'. We don't necessarily need the procedure 'visualise' which is called in the setup procedure and the code of which is written further down in the model, but it might be useful to copy this anyways to make the imported model look more comprehensible.
- All other procedures further down in the code are for the analysis and visualisation of the network, and are therefore not crucial for our current aims.

Copy all the code you need and paste it into your new model. Your model should now look like this:

```

extensions [nw]

directed-link-breed [routes route]

routes-own [ days expense km route-type ]

turtles-own [ node-id x y modern province rank town-name ]

to setup
  clear-all
  nw:set-context turtles routes
  nw:load-graphml "orbis.graphml"
  data-correction
  ask turtles
  [
    setxy x y
  ]
  visualise
  reset-ticks
end

to data-correction
  ask turtles with [node-id = "50317"] [set x 12.258 set y 41.78 set label "Portus" set modern "Italy" set province "Italia" set rank 90]
  ask turtles with [node-id = "50522"] [set x 16.21 set y 41.36 set label "Aufidus" set modern "Italy" set province "Italia" set rank 60]
  ask turtles with [node-id = "50572"] [set x 25.213 set y 37.412 set label "Rheneia" set modern "Greece" set province "Graecia" set rank 60]
  ask turtles with [node-id = "50457"] [set x 23.589 set y 35.232 set label "Kriou Metopon" set modern "Greece" set province "Crete" set rank 60]
  ask turtles with [node-id = "50786"] [set x 25.75 set y 36.75 set label "Kerea" set modern "Greece" set province "Graecia" set rank 60]
  ask turtles with [node-id = "50789"] [set x 26.459 set y 37.805 set label "Lebinthos" set modern "Greece" set province "Graecia" set rank 60]
  ask turtles with [node-id = "50790"] [set x 23.625 set y 37.875 set label "Leros" set modern "Greece" set province "Graecia" set rank 60]
  ask turtles with [node-id = "50792"] [set x 25.37 set y 37.44 set label "Mykonos" set modern "Greece" set province "Graecia" set rank 60]
end

to visualise
  ask turtles [set town-name label set label "" set shape "house"]
  ask turtles with [rank = 6] [set size 0.2]
  ask turtles with [rank = 60] [set size 0.3]
  ask turtles with [rank = 70] [set size 0.4]
  ask turtles with [rank = 80] [set size 0.7]
  ask turtles with [rank = 90] [set size 0.9]
  ask turtles with [rank = 100] [set size 2]

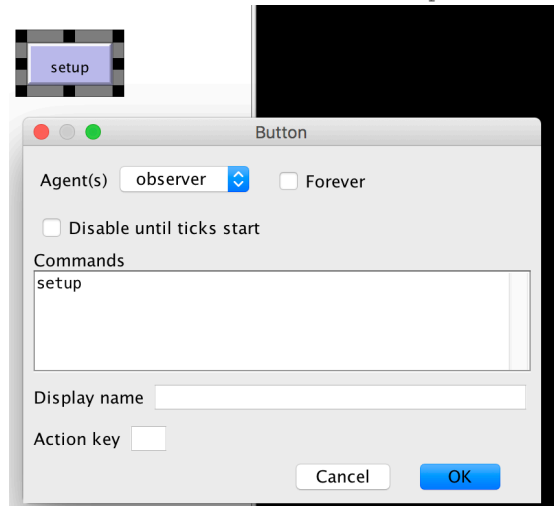
  ask routes with [route-type = "road"] [set color 3]
  ask routes with [route-type = "coastal" or route-type = "overseas" or route-type = "slowcoast" or route-type = "ferry" or route-type = "slowover"] [set color sky]
  ask routes with [route-type = "fastdown" or route-type = "fastup" or route-type = "downstream" or route-type = "upstream"] [set color white]
end

```

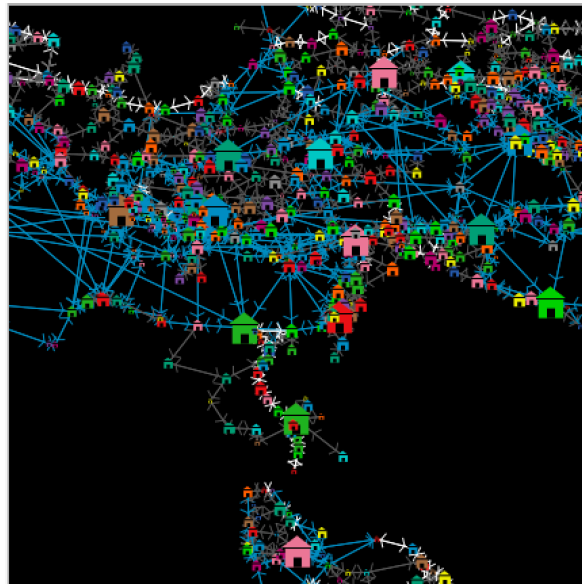
### Save your model!

There should be no errors when you press the 'check' button. So we can go ahead and modify the interface of our model to check whether our code works.

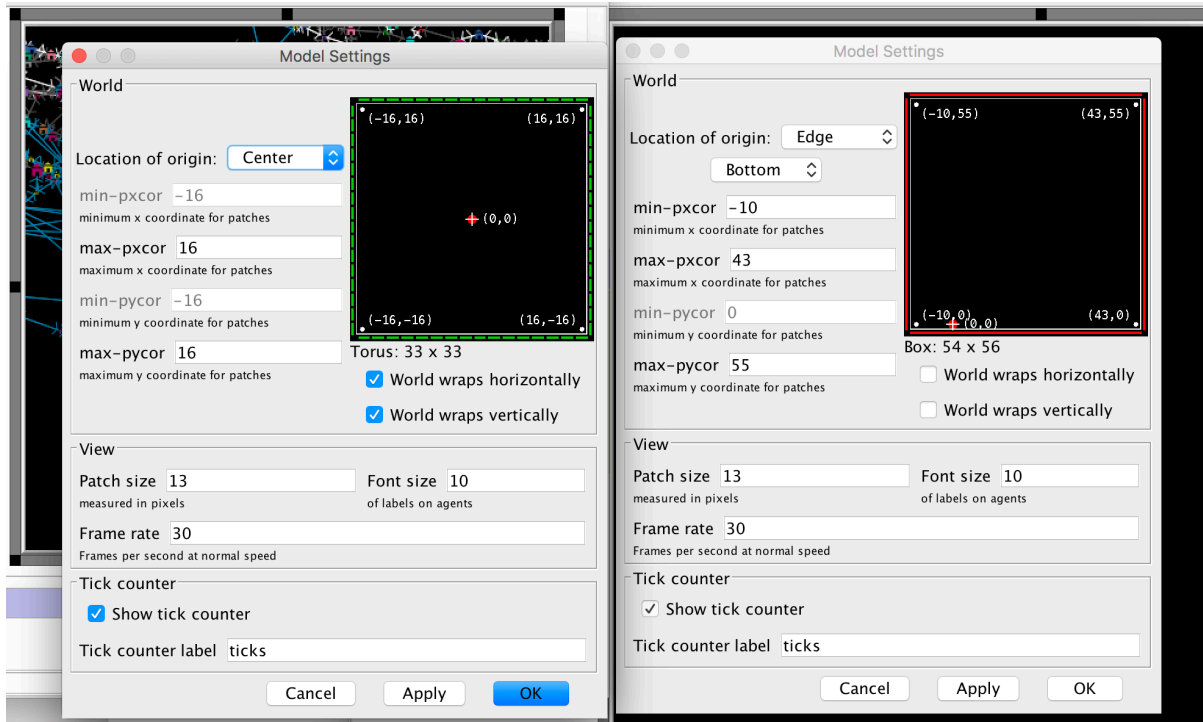
Go to the Interface tab and create a button with the `setup` command:



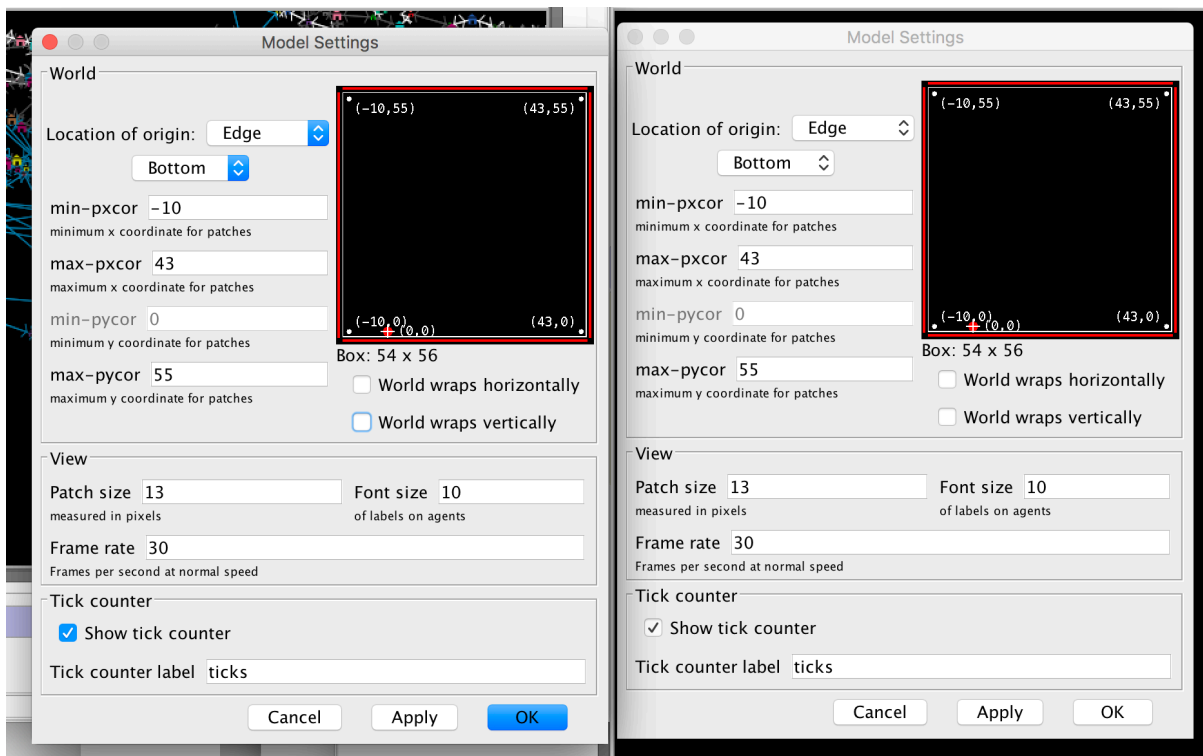
When you click the setup button for the first time you will notice that all data is correctly imported, but also that there is something wrong with the Netlogo world shown in the screen:



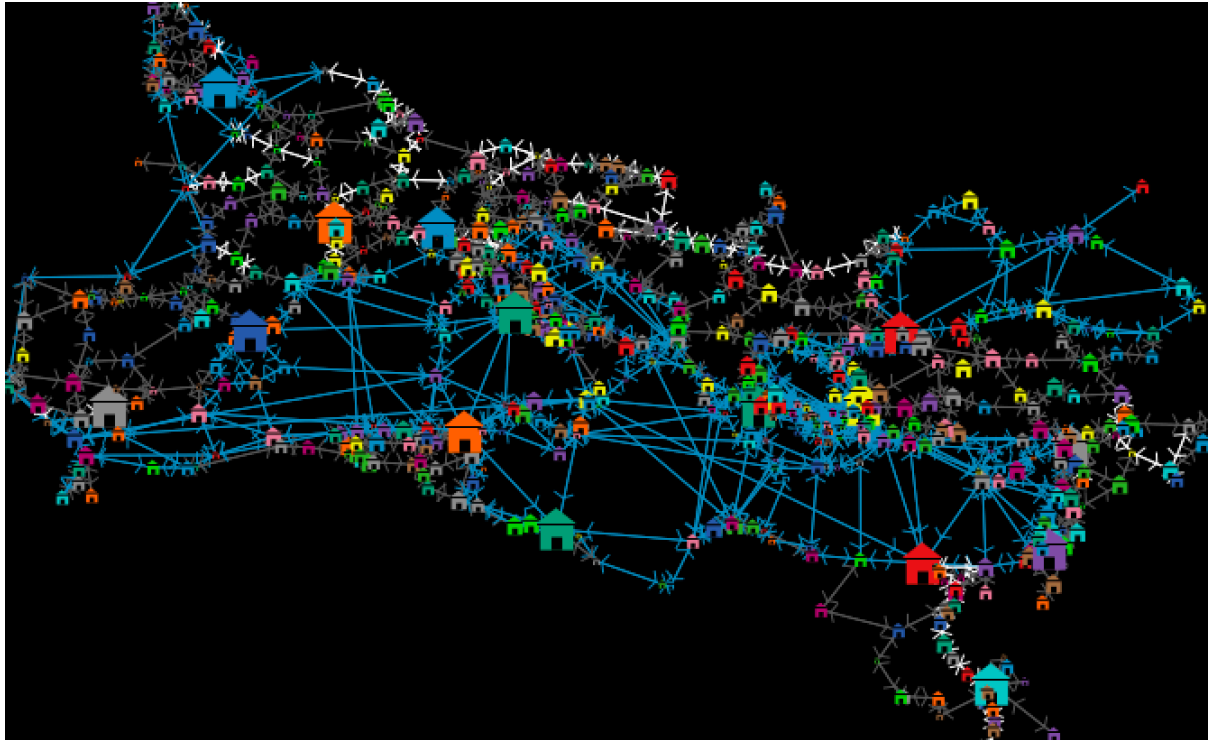
This issue is caused by us not having copied the settings of world settings from the original model. To do this we will open the world settings in both the old and new model, and ensure they are the same. Right click on the world in the old model and select edit, do the same for the new model, and position the two side by side on your screen:



Modify these two screens such that the settings are now the same:



Click ok and click the setup button again. Your network should now be displayed correctly:



You have successfully imported the ORBIS dataset! Surely this was much faster than having to figure out all of this for yourself. Now we can continue to add a simple economic process to explore our hypothesis.

**Save your model!**

## 12. Trading pots over the network

We will now write a very abstract and simple economic process that takes place on this network. We are interested in exploring what the structuring effect of the network is on such economic processes.

We will assume each urban settlement in the ORBIS transport system is a market that behaves as an individual economic actor and can trade pottery with other actors. Although the model we create in this tutorial is so generic that the goods could be called anything, we choose to call them pottery to illustrate how we can simulate artificial archaeological data that can be compared with actual observed archaeological data in future work.

We will give all turtles (which is what the settlements are called in our code) a certain amount of pots and every tick (Netlogo jargon for time step) all pots in the model will be considered for trade. To keep it simple, we will just assume that all turtles want to sell the pots they own, want to obtain new items, and that a transaction is successful with a certain probability.

First, we will create a variable for the turtles called `pots` by adding it at the top of the code:

```
turtles-own [ pots node-id x y modern province rank town-name ]
```

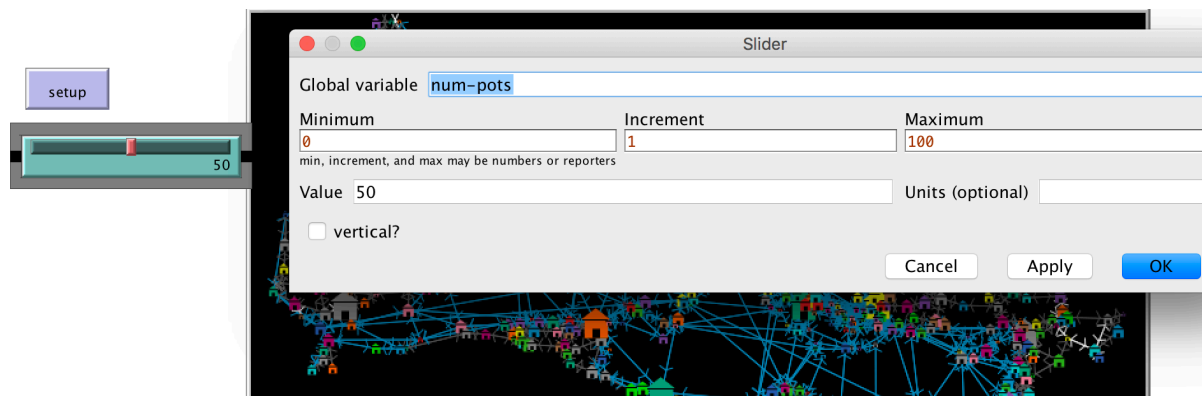
Second, we will give all turtles pots during the setup procedure by adding the following code at the end of the `setup` procedure between `visualise` and `reset-ticks`:

```
visualise
ask turtles [set pots round ( num-pots * size )]
reset-ticks
end
```

This will give each settlement a number of pots proportional to its rank according to the ORBIS dataset (each settlement has a different rank of importance in ORBIS (taken from the Barrington Atlas), and in the visuase procedure the size of each settlement is made proportional to their rank).

Note that we just mentioned a new variable called `num-pots`, so Netlogo will give us an error and we will need to add a slider to control this variable in the **Interface tab**:

- Select Slider from the dropdown box.
- Click the 'Add' button.
- Click anywhere in the white space. A new window will appear where you can determine what the slider will do.
- Write `num-pots` in the 'Global variable' box, set the minimum to 0, the increment to 1, the maximum to 100 and the value to 50. Click **OK**.



Now when you click the setup button all turtles will be given as many pots as defined by the variable `num-pots`

Third, we will create a new procedure that determines what processes will take place when the model runs. So far we have restricted ourselves to determining how the model is set up, and now we will determine what happens after that for every tick (time step).

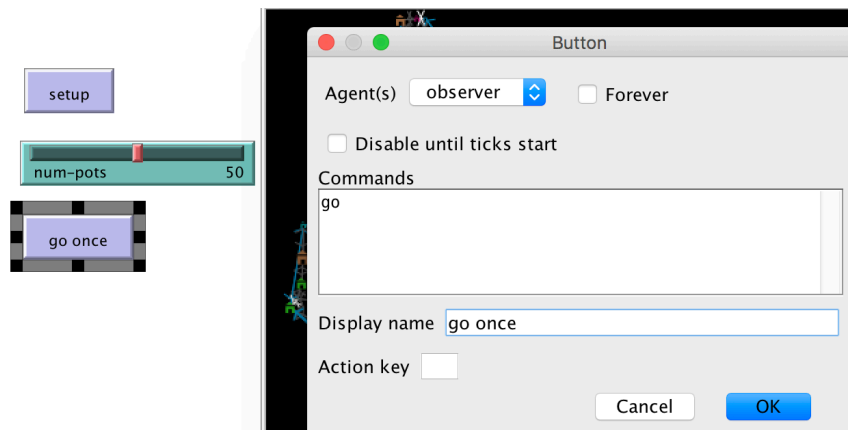
Add a procedure called `go` by writing in your code below the `setup` procedure the following:

```
to go
  tick
end
```

The command `tick` will increase the number of time steps of the model by one every time the `go` procedure happens. To use the `go` procedure we will add a button for it on the **Interface tab**:

- Select Button from the dropdown box.
- Click the 'Add' button.
- Click anywhere in the white space. A new window will appear where you can determine what the button will do.

- Write `go` in the 'Commands' box and 'go once' in the 'Display name' box. Click **OK**.



You have just added a button that will run the `go` procedure once every time you click it. If you do so, you will notice that the number of ticks in the Interface window will increase by one each time. You might also want to create a second button that runs the `go` procedure continuously until you ask it to stop:

- Select Button from the dropdown box.
- Click the 'Add' button.
- Click anywhere in the white space. A new window will appear where you can determine what the button will do.
- Write `go` in the 'Commands' box and 'go' in the 'Display name' box. Make sure the 'Forever' box is ticked. Click **OK**.

You have now created a button that will continuously run the `go` procedure.

Fourth, we will start adding more interesting trade-related commands to the `go` procedure, beginning with a count of the total number of pots during any one tick. In this model this sum will be the same for every tick within a single setup, but we will add this sum in case you decide to modify the model by allowing pots to be added or removed from the model. Write the following in the `go` procedure before `tick`:

```
let total-pots sum [pots] of turtles
```

Fifth, now that we know how many pots there are in this tick we can repeat trade as many times as there are pots by adding the following code in the `go` procedure immediately after the sum of all pots:

```
repeat total-pots
[
]
```

Every command that we now add within these brackets will be repeated as many times as there are pots at the start of that tick.

Sixth, we will randomly select a turtle which has at least one pot, i.e. a turtle that can trade:

```
repeat total-pots
[
  ask one-of turtles with [pots > 0]
  []
]
```

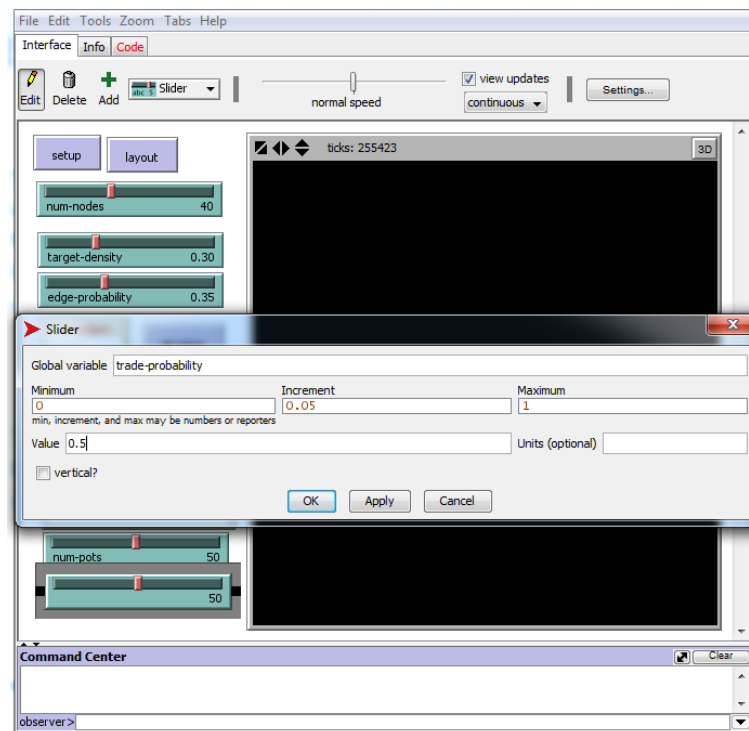


Seventh, we will ask this node to perform a transaction (sell a pot to another node it is connected to) with a certain probability. We will use the same technique as we used earlier for implementing probability: checking whether a randomly selected number is higher or lower than a variable representing the trade probability. Add the following to the `ask` command we just wrote:

```
ask one-of turtles with [pots > 0]
  [if random-float 1 < trade-probability
    []
  ]
```

This code will perform an action only if a randomly selected floating point number between 0 and 1 is smaller than the value we set for the variable `trade-probability`. Let's create a slider for this new variable in the **Interface tab**:

- Select Slider from the dropdown box.
- Click the 'Add' button.
- Click anywhere in the white space. A new window will appear where you can determine what the slider will do.
- Write `trade-probability` in the 'Global variable' box, set the minimum to 0, the increment to 0.05, the maximum to 1 and the default value to 0.5. Click **OK**.



Finally, we will add the transaction that will take place if a randomly selected number is lower than the threshold determined by the `trade-probability` variable. Add the following to the `if` command we just wrote:

```
[if random-float 1 < trade-probability
  [set pots pots - 1
    ask one-of route-neighbors
    [set pots pots + 1]
  ]
]
```

This code will perform a transaction by first reducing the number of pots of the seller (the randomly selected turtle), then it will randomly select one of the other turtles this node is



connected to (this second turtle becomes the buyer), and finally it will increase the number of pots the buyer owns by 1.

Your complete go procedure should now look like this:

```
to go
  let total-pots sum [pots] of turtles
  repeat total-pots
  [
    ask one-of turtles with [pots > 0]
    [if random-float 1 < trade-probability
      [set pots pots - 1
        ask one-of route-neighbors
        [set pots pots + 1]
      ]
    ]
  ]
  tick
end
```

### Save your model!

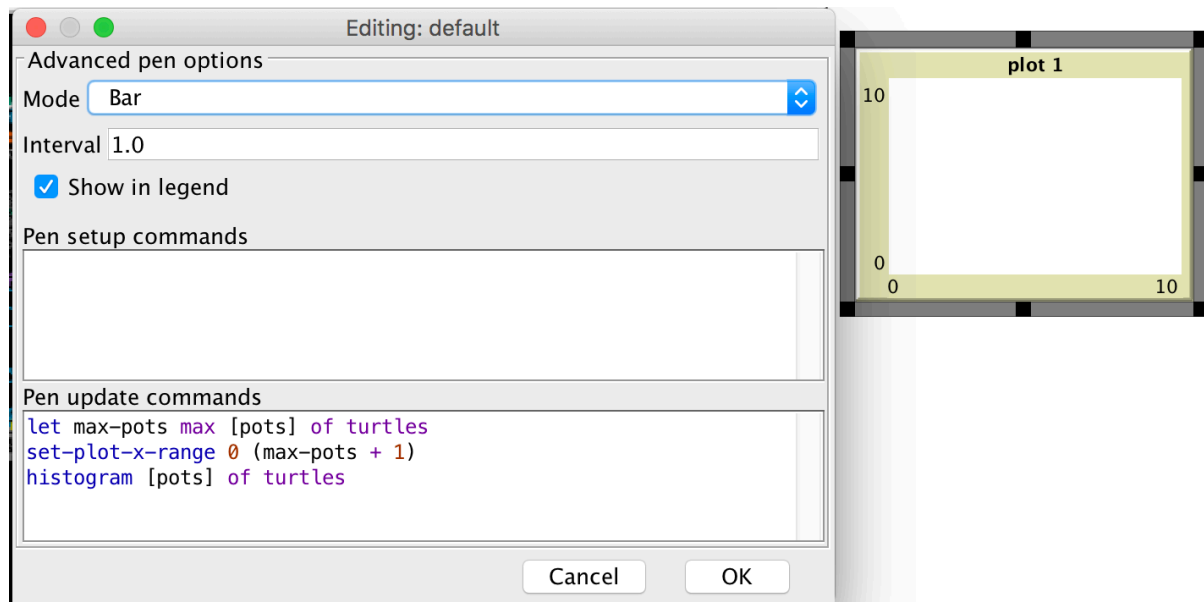
At the moment we cannot explore the trade procedure we just created very easily, because we do not have anything that reports its effects. Let's add a plot that tells us something about the trade processes.

One of the most interesting things to explore would be to see how many pots each turtle has: the distribution of pots. We will create a plot that shows us this in the **Interface tab**:

- Select Plot from the dropdown box.
- Click the 'Add' button.
- Click anywhere in the white space. A new window will appear where you can determine what the Plot will do.
- Write 'Distribution of pots' in the 'name' box.
- Click on the yellow pencil to modify the 'default pen'. A new window will open where you can determine what this pen will plot.
- Set the 'Mode' dropdown box to 'Bar', set the 'Interval' to 1.0, and write the following code in the 'Pen update commands' box:

```
let max-pots max [pots] of turtles
set-plot-x-range 0 (max-pots + 1)
histogram [pots] of turtles
```

- Click **OK** twice.



This plot will create a histogram, displaying the distribution of pots among turtles. On the X axis we see the number of pots, and on the Y axis we see the number of turtles that own a certain number of pots.

Finally, we will add another way of visualising the differences in the number of pots that different turtles own by updating the size of turtles relative to the number of pots they own.

At the end of the go procedure, just before `tick`, we will call a new procedure `update-size`:

```
...
]
  update-size
  tick
end
```

At the end of all the code we will specify what this new procedure does. Copy and paste the following:

```
to update-size
  let max-pots [pots] of one-of turtles with-max [pots]
  ask turtles
  [
    set size ( pots / max-pots ) * 3
  ]
  ask one-of turtles with-max [pots] [ show town-name ]
end
```

This procedure will change the size of each settlement at the end of each time step, making the settlements with a lot of pots larger and the settlements with few pots smaller. The settlement with the highest number of pots will be displayed in the command centre at the bottom of the screen.

**Save your model!**

### 13. The effect of the Roman transport system on wealth inequality

Now we can finally start exploring the first aspect of our hypothesis: the effect of the Roman transport network on the distribution of wealth (as represented by our simple process of trade in pots).

Setup your model and let it run for a few time steps. Explore how the plot of pot distribution and the size of settlements changes. Try changing the sliders and setting up and running the model again. How do the distributions and settlement sizes change over time? What is the effect of the `trade-probability` variable? What is the effect of the `num-pots` variable? Is there strong inequality in the distribution of pots?

Try to identify how a settlement's "wealth" in pots correlates with its rank and its number of connections in the ORBIS network.

### 14. Other network structures

There is an aspect of our hypothesis that we cannot address with our current model: Is the phenomenon of wealth inequality a particular feature of the Roman transport system structure, or can other network structures enforce more extreme wealth inequality much faster?

To explore this question, and to see how wealth inequality does or does not arise depending on network structure, we will again search for pre-existing code that allows us to create different network structures in our model. Luckily, such code is available via the project MERCURY website.

Navigate to the MERCURY model library and follow the links to the 'Network structures' tutorial:

<https://projectmercury.eu/model-library/>

Click 'Download' and from the 'code' folder inside the downloaded folder copy the model 'network-structures.nlogo' and paste it into your working folder 'network\_structures'.

Open the model you just downloaded. This model simply allows you to create a set of nodes and connect them according to a particular network structure (see the network science glossary for more information about the different types of networks

<https://archaeologicalnetworks.wordpress.com/resources/#glossary> ). Let's have a look at the different procedures to identify what we need to copy and what not:

- The `create-new-network` procedure will create a completely new set of nodes and connect them with a new set of links according to 8 different types of network structure. Because we do not want to change the set of nodes (the settlements in the ORBIS network) we will not use this procedure.
- The `same-nodes/new-links` procedure will keep the existing set of nodes in the model and connect them with a new set of links according to four different network structures. Because we want to keep the ORBIS nodes we can use the code in this procedure.

- The remaining code is used to change the visual layout of the network and to report statistical network properties. We will not copy this code.

Copy the entire `same-nodes/new-links` procedure and paste it into your own model at the very end.

Now we need to iron out a few issues to ensure this new code works with our current model. What we want to achieve is that when we setup the model, it either shows us the ORBIS network or an alternative network structure that we select.

First, let's change the name of this procedure to something more descriptive and appropriate for our own model. Replace `to same-nodes/new-links` for `to alternative-network`.

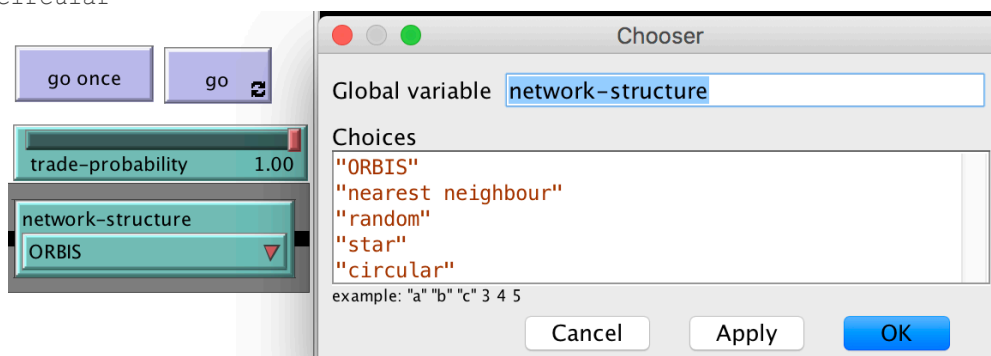
Second, to use this new procedure we can include it in the setup procedure just before

```
reset-ticks:
  ask turtles [set pots round ( num-pots * size )]
  alternative-network
  reset-ticks
end
```

Third, we should change the variable `existing-network-structure` into something more appropriate for our own model. Go to the `alternative-network` procedure in your code and change every `existing-network-structure` to the following new variable: `network-structure`

Fourth, we need to create this new variable `network-structure` in the interface of our model. It will take the form of a drop-down window that allows us to select what network structure we want to create for a given experiment. Go to the interface tab (ignore all error warnings) and add a chooser drop-down window to the interface for the global variable `network-structure` with the following five choices:

```
"ORBIS"
"nearest neighbour"
"random"
"star"
"circular"
```



Now that we have made the necessary changes to incorporate the copied code into our own model, we can turn our attention to the remaining error messages.

- Nothing named `nearest-neighbours` exists. This is a variable that is needed for creating the "nearest neighbour" network structure (it determines to how many of a nodes nearest neighbours each node is connected). Go to the interface and add a

slider for the variable `nearest-neighbours` with a minimum of 1 and a maximum of 10, an increment of 1 and a default value of 3.

- Nothing named `probability` exists. This is a variable needed for the “random” network structure (it determines the probability with which a link is created between a pair of nodes). Go to the interface and add a slider for the variable `probability` with a minimum of 0 and a maximum of 0.3, an increment of 0.01 and a default value of 0.01.

Once you made these changes the model does not report any errors any more. Put this to the test by clicking the setup button and changing the setting of the `network-structure` variable. Can you create each of the five different network structures correctly?

In fact, there is still one issue we need to iron out. If you click the `go` button, you will get an error because our links are no longer known as routes (the breed we needed to import the ORBIS network) but they are now called links. To solve this issue, change “route” in the `go` procedure to “link”:

```
to go
  let total-pots sum [pots] of turtles
  repeat total-pots
  [
    ask one-of turtles with [pots > 0]
    [if random-float 1 < trade-probability
      [set pots pots - 1
        ask one-of link-neighbors
        [set pots pots + 1]
      ]
    ]
  ]
  update-size
  tick
end
```

## Save your model!

Your setup procedure should now look like this:

```
to setup
  clear-all
  nw:set-context turtles routes
  nw:load-graphml "orbis.graphml"
  data-correction
  ask turtles
  [
    setxy x y
  ]
  visualise
  ask turtles [set pots round ( num-pots * size )]
  alternative-network
  reset-ticks
end
```

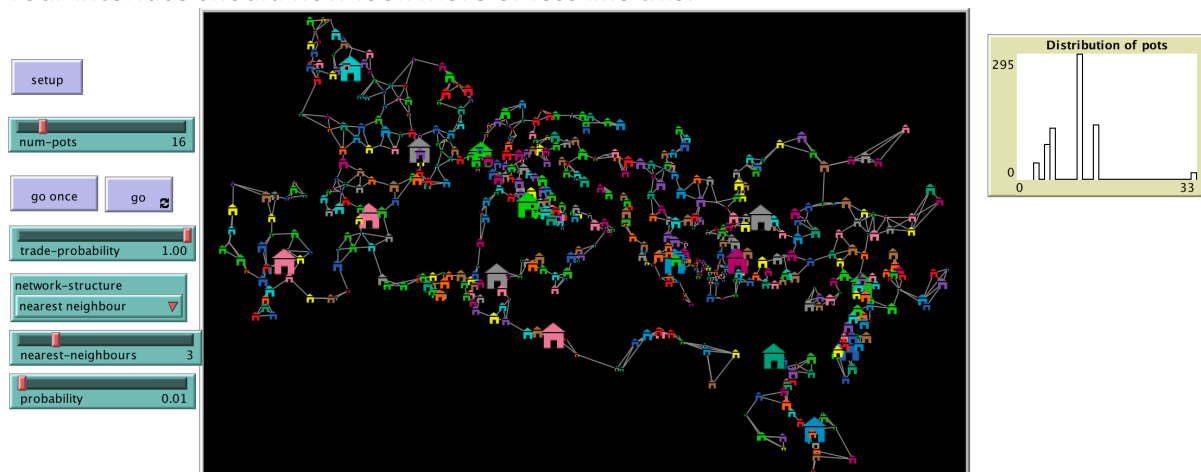
Your alternative-network procedure should now look like this:

```

to alternative-network
  if network-structure = "nearest neighbour"
  [
    ask links [die]
    ask turtles
    [
      create-links-with min-n-of nearest-neighbours other turtles [distance myself]
    ]
  ]
  if network-structure = "random"
  [
    ask links [die]
    ask turtles
    [
      ask other turtles
      [
        if random-float 1 < probability
        [
          create-link-with myself
        ]
      ]
    ]
  ]
  if network-structure = "star"
  [
    ask links [die]
    ask one-of turtles [ create-links-with other turtles ]
  ]
  if network-structure = "circular"
  [
    ask links [die]
    let n [who] of one-of turtles with-max [who]
    while [n >= 0]
    [
      if n = 0
      [
        ask turtle n
        [
          create-link-with one-of other turtles with-max [who]
        ]
        stop
      ]
      if n != 0
      [
        ask turtle n
        [
          create-link-with one-of other turtles with [who = (n - 1)]
          set n n - 1
        ]
      ]
    ]
  ]
]
end

```

Your interface should now look more or less like this:



## 15. Network structures and wealth inequality

Now we can explore the second part of our hypothesis: to what extent is the effect of wealth inequality a result of the transport network structure?

First explore the distribution of pots and the top scoring settlements when only using the ORBIS network. Then start exploring in turn the other four network structures. Try identifying the differences for the distribution of pots for each network structure using the following questions:

- Are pots distributed very unequally?
- With what network structure is the distribution of pots most unequal?
- With what network structure is the inequality of pot distributions at its most extreme?
- With what network structures does it take more time steps to obtain very strong inequalities in pot distributions?
- Are pots ever normally distributed? Can you identify conditions under which you can generate non-extreme wealth inequalities?

You will notice that most of the alternative network structures do not make any sense in real terms: they blatantly ignore geographical reality and known socio-political realities of the Roman world. However, it is crucial to keep in mind that they do so in a very productive way. They represent alternative ways in which transactions between urban settlements in the Roman world could have taken place. What if every place was only connected to two other places (circular structure)? What if these places were the geographically closest towns (nearest neighbour structure)? What if all trade had to take place through one single urban settlement (star structure)? What if interaction opportunities of a pair of settlements was entirely a result of chance (random structure)?

Obviously these network structures represent extreme alternative scenarios, they have nothing to do with the Roman reality. It is very clear that the precise number of simulated pots at particular sites like Rhodes or Rome is not informative at all. But what *is* informative is the general trends that recur between different setups of a simulation experiment: is one town always the most wealthy or does it share this position with a handful of others; is the wealthiest town 10 times wealthier than the majority or 1000 times? As extreme theoretical scenarios, these alternative network structures help us outline the realm of possible past realities. Most crucially, they help us to narrow down where we should position credible theories of a Roman phenomenon by comparing simulated data patterns with observed data patterns. How close are the archaeologically and historically observable information to the artificial information that results from these extreme scenarios?