Learning to run in Netlogo

This tutorial is designed for intermediate NetLogo users. As a minimum, please complete the “First steps in NetLogo” tutorial before you start this one (**tinyurl.com/mu9bqev**).

The aim of this tutorial is to make you proficient with the following key aspects of NetLogo:

- Breeds;
- Global and local variables;
- If, while and for- loops;
- Lists.

This tutorial was also designed to increase your familiarity with a number of commonly used primitives, in particular: “of”, “with”, “any?”, “one-of” and “myself”. We will do this using an example of a simple model of exchange, where goods are produced in the centre of the world and then distributed by the agents. We will investigate the pattern of uptake of new goods in areas increasingly further away from the production centre.

Let’s start with breeds. Often you want to make different groups of agents behave differently. You could achieve this simply by giving them a specific variable and then calling them accordingly. In NetLogo lingo, you would say you create ‘an agentset’ on the fly. For example (no need to type it):

```
ask turtles with [color = pink]
      [ move-to one-of neighbor4 ]
```

In this code snippet we created an agentset of all turtles with colour pink (and asked them to move to one of the four neighbouring patches). However, if you want to spare yourself some typing you can predefine such agentsets at the very start of your code. A predefined agentset is what we call a *breed*. Here, we will create a population of producers and a population of vendors. Type the following at the very beginning of the code:

```
breed [ producers producer ]
breed [ vendors vendor ]
vendors-own []
producers-own []
```

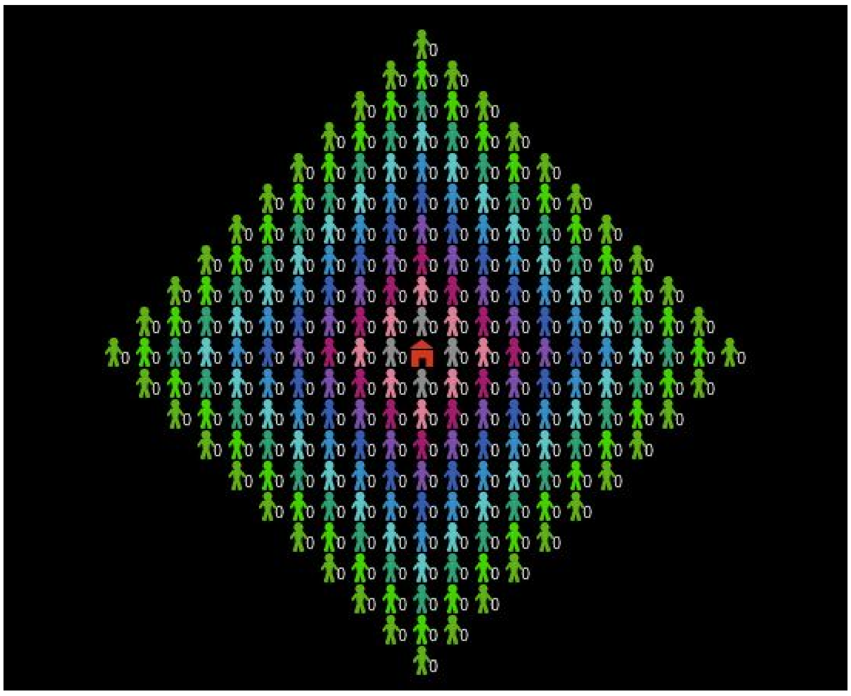
To create a breed you need to specify its singular and plural name (`producers`, `producer`). You can also assign variables to it, just as you would assign variables to turtles or patches (`turtles-own, patches-own`). Remember that from now on, if you want to call only producers you need to use `ask producers`, if you use `ask turtles` you will call all agents - producers AND vendors. Similarly you can use many turtle primitives in the same way as you would if you were calling all turtles, e.g., `create-turtles` or `turtles-here`, can be replaced with `create-producers` and `producers-here`.

Let’s create a production centre and vendors distributed around it. Write a setup procedure with one producer, set its shape to “house” and give it a variable called “goods”. Then create four vendors around the production centre, on the patches directly to the north, south, east and west. Set their shape to “person” and also give them a variable “goods”. Hint: check the primitives `at-points` and `sprout`. When you are done, turn the page to see if you got it right.

```
breed [ producers producer ]
breed [ vendors vendor ]
vendors-own [ goods ]
producers-own [ goods ]

to setup
    __clear-all-and-reset-ticks

    create-producers 1 [
        set color 15
        set shape "house"
        set goods 0
  ]

    ask patches at-points [[0 1] [1 0][-1 0][0 -1]][
      sprout-vendors 1 [
          set color 5
          set shape "person"
          set size 1
          set goods 0
        ]
      ]
end
```

If everything went fine you can see a production centre and four vendors around it when you press on the setup button. However, we will need a few more of them. Also, we are interested in the relationship between the uptake of goods and the distance from the production centre so we need a way of differentiating how far each vendor is from the production centre. To do so give all turtles another variable that will label how far they are from the production centre. Call it 'distance-level' and do not forget to initiate it in the setup (for both vendors and producers).

We will now populate the world with some more vendors. To keep things simple, we will ask vendors to create their neighbours in a centripetal fashion on all neighbouring cells that are not occupied. The end result should look like this:



Each colour denominates a different distance-level so that the green vendors are one step further than the blue vendors, etc. Write the following procedure:

```
to populate
  ask vendors [
    let not-occupied-count count neighbors4 with [ not any? turtles-here ]
    hatch not-occupied-count [
        set color [color] of myself - 10
        set distance-level [distance-level] of myself + 1
        set size 1
        set goods 0
        move-to one-of neighbors4 with [ not any? turtles-here ]
      ]
    ]
end
```

Let’s go through this code step by step. We first determine how many empty patches a vendor has:

```
let not-occupied-count count neighbors4 with [ not any? turtles-here ]
```

Why do we use ‘turtles’ instead of ‘vendors’? It is because we do not want to place new vendors on top of the production centre so we are looking for patches that do not have any agents whatsoever.

Then we hatch the required number of new agents:

```
hatch not-occupied-count
```

And we give them all the necessary variables. Note the use of `myself` for example in

```
set distance-level [distance-level] of myself + 1
```

Here, we initialise a new vendor with the distance-level variable of their parent plus 1 so that each subsequent tier of vendors has a value higher by 1 that the ones closer to the production centre. Remember to add ‘distance-level’ to the list of vendors and producers variables as well as to their initialisation commands!

Finally, we move the new agents to their new locations:

```
move-to one-of neighbors4 with [ not any? turtles-here ]
```

We now need to repeat the process however many “levels” of vendors we want. Start with creating a slider called “level”, with values of 1-6. To repeat the ‘populating’ procedure we will just call it however many distance-levels we need. Add the following line at the end of the setup procedure:

```
repeat (level - 1) [populate]
```

It is `level - 1` because we have already created one distance-level of vendors. If we did not adjust it when setting the level to, say, five on the slider, you would get six tiers of vendors.

Finally, let’s visualise how much goods each vendor has. Add the following line at the end of the setup procedure:

```
ask vendors [set label goods]
```

Your full setup procedure should look like this:

```
breed [ producers producer ]
breed [ vendors vendor ]
vendors-own [ distance-level goods ]
producers-own [ distance-level goods ]

to setup
  __clear-all-and-reset-ticks

  create-producers 1 [
      set color 15
      set shape "house"
      set distance-level 0
      set goods 0
]
  ask patches at-points [[0 1] [1 0][-1 0][0 -1]][
    sprout-vendors 1 [
        set color 5
        set shape "person"
        set size 1
        set distance-level 1
        set goods 0
      ]
    ]

  repeat ( level - 1 ) [ populate ]

  ask vendors
      [ set label goods ]

end
```

In this setup we have created two breeds of turtles: producers and vendors and we have used producers-specific and vendors-specific primitives. But behind the scenes we have also introduced the topic of global and local variables. Can you name the two ways in which you can set a global variable? We have already used both.

First, you can create a slider, chooser, switch or a box in the NetLogo Interface. Just like we have done with the 'level'. Second, you can define a variable using the `globals`, `turtles-own`, `patches-own`, etc. keywords at the beginning of the code. We chose to use `producers-own` and `vendors-own` so that the variables defined using these keywords are only available to members of one of the breeds. Had we used `globals` the variables would have been accessible by all turtles, patches and links.

We have also created a local variable: `not-occupied-count` in the `populate` command. We did it by using the `let` primitive:

```
    let not-occupied-count count neighbors4 with [ not any? turtles-here ]
```

This line says: "create a local variable called `not-occupied-count` which stores the number of (`count`) of all of the patches in the turtle's neighbourhood (`neighbors4`) which do not have any turtles on them ([`not any? turtles-here`])". This variable is only accessible in the ask vendors command within the populate procedure - that is between the brackets we opened after `ask vendors` and closed at the end of this procedure. If you tried to call it from the setup or from any other place NetLogo would prompt you that "nothing named not-occupied-count has been defined" (try and check!). We will define local variables a few more times throughout the code so keep an eye on it and try calling it from various positions in the code to see what happens.

The setup procedure is ready. Check that you get the ‘diamond’ of vendors with a production centre in the middle if you hit the setup button. If so, let’s move to the go procedure.

First of all we need to make the producer generate the goods and distribute them to the four closest markets. Write in the go procedure:

```
  ask producers
      [ set goods production-level ]

  ask producers
      [ trade ]
```

Let’s create a slider `production-level` so that it becomes a user defined variable. Next we need to write the trading procedure.

```
to trade

  let closest-neighbours vendors-on neighbors4
  let next-tier-neighbours closest-neighbours with [distance-level =
[distance-level] of myself + 1]

  while [goods > 0] [
    ifelse any? next-tier-neighbours with [ goods < storage ][
      ask one-of next-tier-neighbours with [ goods < storage ] [
        set goods goods + 1
      ]
      set goods goods - 1
    ]
    [stop]
  ]
end
```

If you hit the ‘check’ button you will get a familiar sight - the yellow warning that nothing named STORAGE has been defined. Create a new slider to deal with it. All together, this new code may look a bit intimidating at the first sight so let’s unpack it line after line.

First we define who we are going to be trading with in the lines:

```
let closest-neighbours vendors-on neighbors4
```

and

```
let next-tier-neighbours closest-neighbours with [distance-level =
[distance-level] of myself + 1]
```

Again, we have created two local variables. First the variable `closest-neighbours`, which stores all the vendors on the neighbouring patches (`vendors-on neighbors4`) and then `next-tier-neighbours` who are simply vendors on the patches around (`neighbours`) whose distance level is higher than the calling agent (`with [distance-level = [distance-level] of myself + 1]`). In sum, each vendor will only trade with their immediate neighbours located further away from the production centre (that is, having a higher `distance-level`).

The next few lines is the meat of this procedure. Let’s express it in pseudocode

```
While you have any goods left,
      if any trading partner has space left in storage,
            ask one of them to increase the amount of goods by one
            and decrease your own amount of goods by one
      If no more trading partners with space left in storage
            stop the procedure
```

I hope this makes it clearer, but let's look at a few potential traps in the code. First, you might have noticed the repetition of the `next-tier-neighbours with [ goods < storage ]`. Can you guess what would have happened if we used this code instead:

```
ifelse any? next-tier-neighbours with [ goods < storage ][
      ask one-of next-tier-neighbours [ set goods goods + 1 ]
```

Although we would ask trading partners to fulfil the condition (empty space in their storage), we would then trade with a random next-tier-neighbour, regardless of their storage situation. In other words, we would ask if there are any trading partners meeting our requirements but then trade with everyone. Thus the repetition of the condition in these two lines.

Second, you may be confused about the brackets and in particular about the `[stop]` line appearing as if from nowhere. This is part of the ifelse loop. The structure of the ifelse loop is the following:

```
Ifelse condition is true:
      [do x]
otherwise
      [do y]
```

So in our case the **x** is `ask one-of…` (which comes with its own set of brackets) and `set goods goods - 1`, while **y** is `stop`. You might have noticed that if you click on a bracket NetLogo highlights its counterpart.

Thankfully, we can reuse this code to model the trade between vendors. Add the following line to the go procedure:

```
ask vendors with [ goods > 0 ]
      [ trade ]
```

Now, why didn't we just ask all vendors? After all, they should all engage in trade. The reason here is performance - if you ask all vendors then, by definition, every one of them will have to perform the trade procedure. Even if each takes only a fraction of a second to complete it, these fractions add up and in the end may significantly slow down your simulation. Instead, we only run the trade procedure on a subset of agents who meet the condition - they have goods. This speeds up things, especially at the beginning of the simulation when only a few agents have any goods at all.

Add a line asking vendors to display the amount of goods they have (same as in the setup procedure) and ask all of them to consume (remove from storage) one item at each time step to account for accidental loss, breakage etc. Finally, do not forget to add `tick` at the end of the procedure. Turn the page to see how the trade function should look like.

```
to go

  ask vendors
      [ set label goods ]

  ask producers
      [ set goods production-level ]

  ask producers
      [ trade ]

  ask vendors with
      [ goods > 0 ] [ trade ]

  ask vendors with [ goods >= 1 ]
      [ set goods goods - 1 ]
  tick

end

to trade

  let closest-neighbours vendors-on neighbors4
  let next-tier-neighbours closest-neighbours with [distance-level =
[distance-level] of myself + 1]

  while [goods > 0] [

    ifelse any? next-tier-neighbours with [ goods < storage ][
      ask one-of next-tier-neighbours with [ goods < storage ] [
        set goods goods + 1
      ]
      set goods goods - 1
    ]
    [stop]
  ]
end
```

One thing that is missing here is the ability of traders to keep some of the goods for their local market. At the moment they trade away everything they have, so it would be more realistic if they only traded a proportion of their goods. To implement this idea we can simply create a global variable 'storage-threshold' by adding a slider (between 0 and 1) in the interface. We also need to modify the code to reflect the change. First of all, this does not apply to the producers - they will distribute all the goods they have. Copy paste the trade procedure (so that you have two) and change its name `to trade-producer`, then in the line when you call the production centre to trade change `trade` to `trade-producer`.

```
  ask producers
      [ set goods production-level ]

  ask producers
      [ trade-producer ]
```

We also need to adjust the trade procedure (the original one, not the trade-producer one).Instead of:

```
while [goods > 0] [
```

We should have:

```
while [goods > storage * storage-threshold] [
```

This means that traders will only trade the goods that exceed the user-defined threshold (say, 10%, 20% or 50% of the total storage) specified in the slider you have just created.

When you change code it is always worth checking if the same piece of code does not appear somewhere else. For example, we now have two trade procedures (one for producers, one for vendors). It would be quite easy to introduce errors if we wanted to change the way agents trade but forgot about one of the trading functions. That is why it is a good idea to familiarise yourself with the search function. Hit ctrl+f or command+f and a search window appears. If you write “goods > 0” in the ‘Find’ window it will take you and highlight all instances of that phrase. It should highlight the line we have just changed and the old version of it in the trade-producer function. But if you keep on pressing ‘Next’ you will find the following line in the trade procedure:

```
 ask vendors with [ goods > 0 ] [ trade ]
```

which could be changed to:

```
 ask vendors with [ goods > storage * storage-threshold ] [ trade ]
```

If you keep on pressing ‘Next’ the find function will take you back to the starting point in the trade-producer procedure. It is surprisingly easy to introduce these kind of errors into the code since they do not trigger the debugger. The error is not in the syntax but in the logic of the simulation. Unfortunately, they are also hard to pick up if you just read through the code. The best solution is to run a series of tests before you complete the simulation. For example you could use the Command Centre to list all trading agents and the amount of goods they have in their storage. You should also familiarise yourself with the `print` primitive - it is really useful for checking whether a command is actually being called.

On the next page there is the pseudocode with all the main elements you should have by now in your code.

```
to setup
      create producers
      create vendors at neighbouring patches
      populate the world with the rest of the vendors
to go
      ask producers -> produce goods
      ask producers -> distribute goods (trade-producer)
      ask vendors -> trade
      ask vendors -> consume one item of the goods
to populate
      hatch agents at neighbouring cells
to trade-producer
      distribute all goods to the neighbouring vendors
to trade
      If more goods in your storage than the storage threshold
            give an item to one of neighbours
            remove an item from own storage
```

The simulation is pretty much ready but we need some kind of output to see how the trends change between different scenarios. What interests us is the differences in amount of goods in assemblages belonging to markets at different distances from the production centre. The stochastic (that is, random) nature of the simulation means that that amount will differ from one market to another, but we can aggregate them to get a mean value for each distance-level. We will do it using Netlogo lists. Lists are what makes a lot of simulations tick. They are also very computationally efficient, which means that you can create more complex simulations or run a much wider parameter sweep. It really is worth spending some time to understand them.

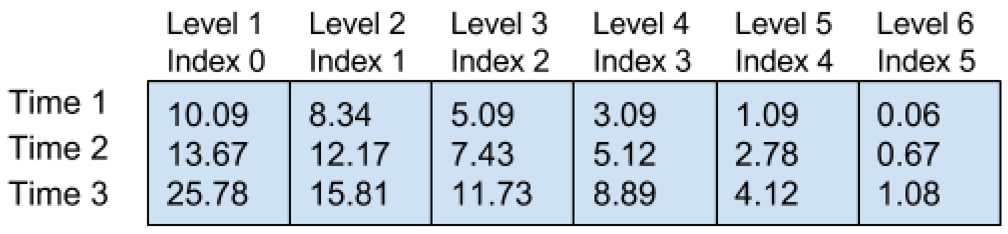
Start with adding a new global variable - `saturations` at the very start of the code; this will be the global list of lists:

```
globals [ saturations ]
```

Then we need to initialise our list of lists in the setup procedure:

```
set saturations (list [] [] [] [] [] [] )
```

Each set of brackets is a separate list that we will use to store the mean volume of goods at one particular distance-level. So the first set of brackets will be used to store volumes from the distance-level 1, the second set of brackets from the distance-level 2, etc. At each time step we will calculate and add a new value to the appropriate set of brackets. We can visualise it as a kind of Excel spreadsheet in which each column contains an ever growing list of numbers.



	Level 1 Index 0	Level 2 Index 1	Level 3 Index 2	Level 4 Index 3	Level 5 Index 4	Level 6 Index 5
Time 1	10.09	8.34	5.09	3.09	1.09	0.06
Time 2	13.67	12.17	7.43	5.12	2.78	0.67
Time 3	25.78	15.81	11.73	8.89	4.12	1.08

Let's create a new type of function - a reporter function. It will calculate the volume of goods of all vendors in one distance-level so that we have the numbers to put into the list. Copy the following code:

```
to-report calculate-volume [a]

  let vendors-tier vendors with [ distance-level = a ]
  let sum-goods sum [ goods ] of vendors-tier
  report sum-goods / count vendors-tier

end
```

A reporter function is a function that "reports", meaning calculating and returning the resulting value. You might have also noticed that we pass a variable here - `a`. This variable is used to recognise which distance-level we are calculating at the moment. We will see how it is passed in a moment. Once we have calculated the value we need to attach it to the right list. To do so, create another function:

```
to iterate-list-of-saturations

  let i 1

  while [ i <= level ] [

    let goods-at-distance calculate-volume i

    let new-list item (i - 1) saturations

    set new-list lput goods-at-distance new-list

    set saturations replace-item (i - 1) saturations new-list

    set i i + 1

  ]

end
```

This may look a bit complicated so let's go through it step by step.
First we initiate the counter, we will use to go through all distance-levels:

```
    let i 1
```

'`i`' is a commonly used token for iterating through lists, it stands for 'index'. Then we construct a while loop.

```
while [ i <= level ] [
```

So the procedure will be going on as long as the counter is lower or equal to the value of `level`, that is the number of the distance levels set by a slider. Next we calculate the volume using the function we have just defined and attach it to a local variable `goods-at-distance`.

```
let goods-at-distance calculate-volume i
```

The counter `i` is passed to it as a variable. Remember the 'a' that we were using in the reporter function? (If you do not remember look for it half a page up). This is that 'a'. Next, we grab the saturation list at a specific index (a column in our 'spreadsheet') and attach it to a local variable called `new-list`:

```
let new-list item (i - 1) saturations
```

The primitive `item` allows us to access an item of the list at a specific intex. We need to specify it as `i - 1` because indexing starts from 0 (check the figure above). So the local variable `new-list` is the set of brackets (the Excel cell) that we want to add the value to. Now we will modify it:

```
set new-list lput goods-at-distance new-list
```

Lists are immutable, which means that if you want to change it in any way, in principle, you need to create a new list. You do it by using the `set` primitive. You may remember that `goods-at-distance` is the calculated volume and we simply add it to the `new-list` (using the primitive `lput`). Now that we have the updated list we simply replace the old version with the new version.

```
set saturations replace-item (i - 1) saturations new-list
```

Finally, we move the counter by one:

```
set i i + 1
```

Let's go through one iteration to see how it works in practice.

1. Initially i = 1 (`let i 1`). So the while loop condition (`while [ i <= level ]`) is not fulfilled as 1 is lower that the value of the level variable. This means that the rest of the code will be executed.
2. Next Line: `let goods-at-distance calculate-volume i.` Here, we call the function `calculate-volume` passing the value 1 to it (because `i` is 1 at the moment) so the mean volume of goods of all agents at distance-level 1 is calculated. We attach the result (say 1.97) to a local variable `goods-at-distance`. So `goods-at-distance` = 1.97 at the moment.
3. Next Line: `let new-list item (i - 1) saturations`. Remember that i = 1? 1-1 = 0, so we grab whatever is the first item of the list saturations, i.e., whatever has index 0 and attach it to a new local variable `new-list`.
4. Next Line: `set new-list lput goods-at-distance new-list`. We create a new `new-list` by adding (`lput`) the previously calculated value of 1.97 to it.
5. Next Line: `set saturations replace-item (i - 1) saturations new-list`. We replace the first item in the list of lists (`saturations`) with the new `new-list`.
6. Finally, we move the counter (`set i i + 1`) so i = 2 now.
7. The loop goes back to beginning and executes itself again if 2 is lower than the level (`while [ i <= level ]`). Voila! This was easy, right?

Do not forget to add the iterate-list-of-saturations command to the go procedure.

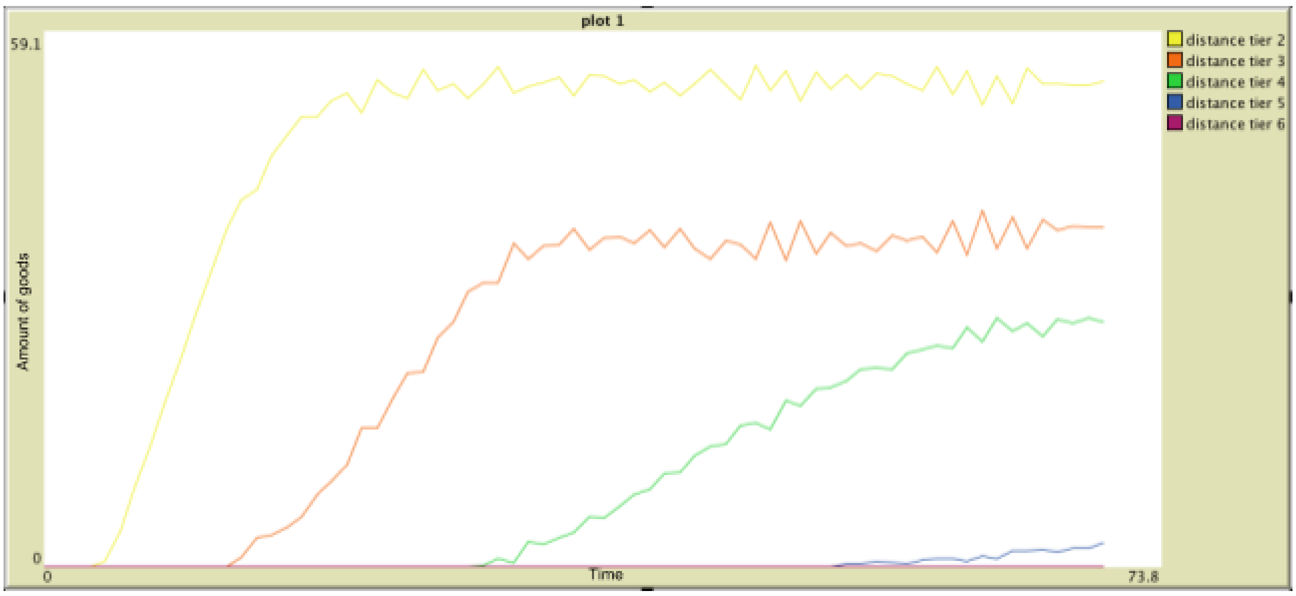
Since we put so much effort into storing the average number of goods in each distance level, let's make a pretty graph to celebrate. Go back to the interface tab, right click anywhere on the white background and choose "plot'.
There is already a pen built for us by default `plot count turtles`. We do not need to know how many agents we have so let's change it. Click on the little icon with a pen next to it and write in the 'Pen update command' the following:

```
let k item 0 saturations
plot last k
```

You may recognise the code here - we grab the first item from the saturations list of lists and we ask Netlogo to plot the last item of it. To plot the second, third, etc. list, that is the volume of goods at distance-level 2, 3, 4… we need to create separate pens. Just hit the 'add pen' button and modify the plotting command. Remember that this time it will be:

```
let k item 1 saturations
plot last k
```

Keep on going until you have all the levels covered. You add a legend by ticking a box and change the colours of the pens by clicking on the color field. If you now run the model you should get a plot similar to this one:



Once this is done for all distance-levels, you are ready to explore the model. Play with the sliders to see how the market saturation curves change depending on the storage capacity of traders, the volume of production of the production centre and the threshold agents use to determine how much of their stored goods they will sell. You can also try to improve the model by:

- Changing the setup with the 'distance levels' to one where the agents calculate their euclidean distance (meaning distance in a straight line) to the production centre.
- If you set the `level` slider at more than 6, you will get an error. Can you figure out why and change the code in such a way that it incorporates any number of distance-levels?
- Change one of the vendors into a second production centre.
- At the moment the number of goods 'consumed' by each market is 1, change that into a user-defined value and explore how the results change.