Tutorial: R for NetLogo

Author: George van Voorn (Wageningen University & Research)
With: Guus ten Broeke (Wageningen University & Research)

This is part 1 of the teaching module "Sensitivity Analysis for Agent Based Models". For this module we assume you (the student) are working on an Agent Based Model (ABM), for instance in the context of studying the resilience of a Socio-Ecological System. Building an ABM is only part of the modelling work. A model is a way of generating new data, e.g., for understanding how the system works, or for making predictions about system response to interventions. To get information out of your model, you need to analyze it in a rigorous fashion. ABMs are stochastic, rule-based models, and as such cannot be analyzed using the mathematical methodologies that are available for analyzing deterministic models. Moreover, we can expect ABMs to display behaviour typical for complex adaptive systems, including tipping points, system adaptation, and transitional patterns. To analyze your ABM, you will need statistical methods. One way of implementing these is by making use of the statistical facilities of the free software R. In this tutorial we assume you have built your model in NetLogo and will make use of R to analyze it. The interfacing with NetLogo goes through RNetLogo (developed by Jan Thiele). Using RNetLogo helps to automate running of ABMs and collecting simulation data for various parameter settings. This collected data may then be used in the application of statistical methods.

This part of the tutorial is intended to make you acquainted with R and RNetLogo so that you can use it for analyzing your ABM in this course.

Contents

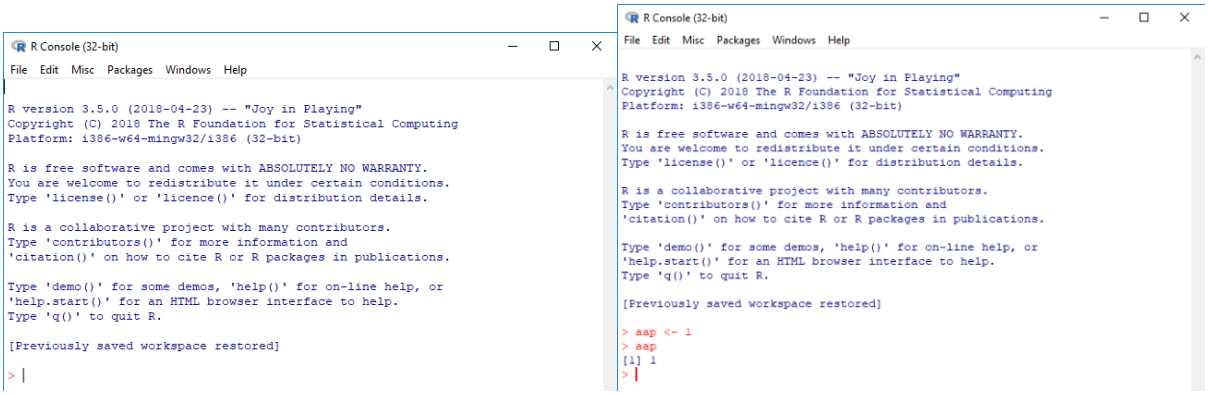
1. Download and install R

You can download R here:
https://cran.r-project.org/bin/windows/base/

Instructions on how to install R (and RStudio, but we will not work with it) can be found in many videos circulating on YouTube, for example. The version of R used when creating this tutorial is version 3.5.0. As always, small deviations might occur when working with a different version. We assume in this tutorial that you, as a student, have little to no experience with using R.

2. Starting R files and basic commands for R

Select R under the Start button, and open up R. You should now have the R console.



Enter a variable. E.g., type

- aap <- 1

Then hit 'Enter'. We have now created a variable named 'aap'. What is put into it is the value 1. You can always check what 'aap' is by typing

- aap

(and then of course 'Enter'). The console responds with

[1] 1

The first element ([1]) is the line number, the second (1) is the value.

We can also enter a vector. Remember your linear algebra? A *vector* is a column of entries, and most data will typically be vector-based. For example, we have

$$v = \begin{pmatrix} 4 \\ 5 \\ 7 \end{pmatrix}$$

These could be the results of running your ABM three times under similar conditions, e.g., in the first run four agents survived in the simulation of your ABM, in the next one five, and in the third one seven. We can also denote this as

$$v^T = (4 \quad 5 \quad 7)$$

We call this 'transposed'. This is a common notation you should get used to, as working with vectors is the basis of understanding how R operates. Type

- noot <- c(4,5,7)

You have now entered a transposed column named 'noot' (again, you can check by typing '>noot' followed by 'Enter').

RNetLogo allows you to store multiple outputs from a single run of your ABM. You may choose to store output on different time points, or multiple outputs, like 'number of agents' and some traits of these agents. You will then probably have a file with multiple rows and columns. Such a file is a *matrix*. Suppose we have

$$M = \begin{pmatrix} 4 & a \\ 5 & b \\ 7 & NA \end{pmatrix}$$

The second column of this matrix indicates some trait, which can have values 'a', 'b', or 'NA' (not available – yes, R can handle those!). Before you can enter a matrix, you will need to set the dimensions. Type

- mies <- matrix(nrow = 3, ncol = 2)

You have created a matrix with three rows and two columns. If you type 'mies', you get

```
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA
[3,]   NA   NA
```

All these entries are empty, but the structure exists now. You can now fill each entry with a value, by typing

- mies[1,1] <- 4
- mies[1,2] <- a

Oops! Error: object 'a' not found

We have not defined 'a' previously. We, however, would like enter strings in this case, as the trait is not a numerical variable, but a categorical one (like 'red' or 'blue'). We type

- mies[1,2] <- "a"

We continue with

- mies[2,1] <- 5
- mies[2,2] <- "b"
- mies[3,1] <- 7

Like we said, do you remember your linear algebra? Element 'mies[i,j]' has indices row 'i' and column 'j'. If you mix these things up, your analysis later on may give totally different results!
We now have

- mies

```
     [,1] [,2]
[1,] "4"  "a"
[2,] "5"  "b"
[3,] "7"  NA
```

You see that R has coerced all values in the same format. We type

- is.numeric(mies[1,1])

The answer is: FALSE, because R has recognized from the "a" and "b" entries that the matrix is not numeric. The lesson is that we can use 'matrix' only for homogeneous data entries.

If we want to enter heterogeneous data, like data containing both numbers and strings (and this is not unlikely when working with an ABM, where we commonly have simulated agents with non-numeric properties), we'd better use a data frame. Under the hood, a data frame is a list of vectors of equal length. Type

- schapen <- data.frame("Number of Agents" = c(4,5,7), "Trait" =c("a", "b", NA))

If we check, we have

```
  Number.of.Agents Trait
1                4     a
2                5     b
3                7  <NA>
```

By default, a data frame coerces strings to factors (which are convenient for statistical analysis like linear regression). If you do not want this, add ", stringsAsFactors = FALSE" between the brackets.

We now already have a great deal of stuff in the memory. It is always good practice to start each session in R by wiping your memory (well, not yours – the computer memory). Type

- ls()

I.e., "list(what to list?)". This command gives the locally defined variables. We will have the output

```
[1] "aap"     "mies"    "noot"    "schapen"
```

To wipe these, type

- rm(list = ls())

I.e., "remove (what to remove? Well, all the listed stuff)". It removes the list containing all pre-defined symbols. Check this by again typing

- ls()

It should be an empty list now.

3. Reading files

Typically you will not enter everything manually, but you will read data from files. Also, you will want to work with a text editor, so you can save a file containing all R commands. Also, we want to set the working directory. If we want to call installed packages, and to interface R and NetLogo later on, we need to have a place from which we start.

Create a new text file in the directory of your choice, say, test.r. Alternatively, you create it using file/New script in the R prompt.
Open this file under file/Open script; of course, it's empty.
You can now type all commands as lines in this text file, so

- ls()
- rm(list = ls())

You can execute a line by placing the cursor at the start of that line, then press Ctrl+R. If you want to execute parts of the programme at once, select the appropriate lines. And if you wish to execute the whole programme, just press Ctrl+A to select all, then Ctrl+R.

Then type

- setwd("THE NAME OF THE DIRECTORY YOU WANT TO WORK FROM")

I.e., "set working directory", and then whatever the location is. It could be something like

- setwd("C:/0Werk/Education/Summer_School")

If you want to check, type

- getwd()

Suppose we have an output file (ending in .dat, say foo.dat), containing the entries of this matrix M

$$\begin{pmatrix} 4 & a \\ 5 & b \\ 7 & NA \end{pmatrix}$$

To read it in as a data table, type

- foo.data <- read.table("foo.dat", header = FALSE)

Of course, "foo.data" can be any name you'd like. Note that R has automatically assigned foo.data as a data frame. Should the data columns have names, you would have header = TRUE. If you check, you see

- foo.data

```
  V1   V2
1  4    a
2  5    b
3  7 <NA>
```

You can select columns. E.g., type

- foo.data[1]

This gives you the first column of the data file.
Type

- foo.data[,1]

This gives you the transposed first column.
Type

- foo.data[1,]

This gives you the first row.
Finally, type

- foo.data[3,2]

This gives you the element in row 3, column 2 (it should read '<NA>').

If you want to select specific rows or columns, you can use ':'. Type

- foo.data[2:3,]

This gives

```
  V1   V2
2  5    b
3  7 <NA>
```

Play around with this to get the hang of it.

4. Simple statistics in R

Now let's move on to use R for what it was intended for in the first place: statistical analysis.

We read in a file included in the zip belonging to this tutorial, '00testfile.dat'. Give it a name, like

- test.data <- read.table("00testfile.dat", header = FALSE)
- test.data

```
  V1  V2
1   1 1.0
2   2 1.5
...
17 17 9.0
```

We consider only the entries in the second column. Type

- X <- test.data$V2

The '$' indicates the column. Since the columns have no names, they are identified by V followed by a number.

We first check the number of entries. This is the length or dimension of the vector. Type

- length(X)

This gives '17'. This is a good first check on your data. If the number of rows (or columns) isn't correct, then something needs to be fixed!

We then calculate the average (also known as mean or expected value). Type

- mean(X)

The answer should be '5'.

The variance is a measure for the spread of the values. Type

- var(X)

The answer should be '6.375'.

5. Plotting in R

The above are all summary statistics. It is good practice to inspect your data using graphical means.

A good first visual data check is to look at the distribution of the data. Data do not usually follow a perfect textbook distribution known as probability density function (PDF), plus the number of entries is limited and the exact distribution is therefore not known. We therefore approximate a PDF by a histogram. The data are assigned to different bins, typically of fixed interval length.

Type

- h <- hist(X, breaks = 20)

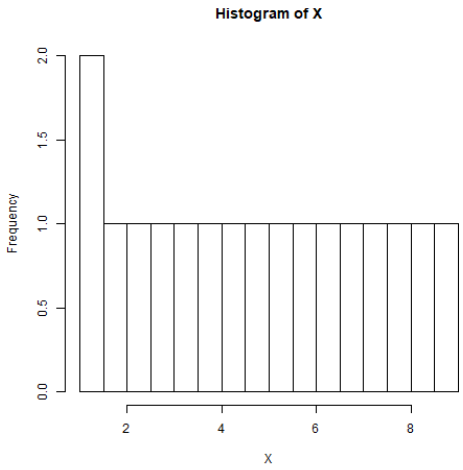
This gives you a histogram with the arbitrary name 'h' consisting of 20 bins. Experiment with the number of bins to see what happens. In this case, not a lot. The 17 entries in the data file are perfectly distributed according to a uniform distribution. But usually your data will look differently!

We can also make graphical files in R (of course). Type

- png(filename = "myhistogram.png")
- plot(h)
- dev.off()

This last command closes the graphical interface, and the png is actually printed. You now have a picture to put into your Powerpoint presentation or paper.

Note, that in this case the left bin has two entries instead of one. This has to do with how you set the lower and upper boundaries.



6. Linear regression in R

We may also be interested in correlating variables.

First open the file '01testfile.r' in the zip in a text editor and manually adjust the line for the working directory. Then, use 'File/Open Script' and select the file '01testfile.r'. The file contains the lines

- ofatr.data <- read.table("ofatr.dat", header=TRUE)
- r <- ofatr.data$r
- ar <- ofatr.data$agents
- fitr <- lm(ar~r,data=ofatr.data)
- summary(fitr)

Select all the text in the file (e.g., by using Ctrl+A when using the default Windows text editor) and execute by running Ctrl+R.

What we do here is give names to the first and second columns, then try to obtain the line that minimizes the sum of square distances, i.e., the regression line. The command for this is

- lm(y ~ x, data = XXX)

Here, variable y is the dependent variable, x is the independent variable, and 'XXX' is the data used in the regression. The summary is to see if this 'fit' is any good. In this case, we get:

```
Call:
lm(formula = ar ~ r, data = ofatr.data)

Residuals:
   Min     1Q  Median     3Q    Max
-32.971  -8.146   3.540  13.116  21.535

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   49.925      4.027   12.40   <2e-16 ***
r            984.176     12.330   79.82   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 15.1 on 88 degrees of freedom
Multiple R-squared:  0.9864,    Adjusted R-squared:
0.9862
F-statistic:  6371 on 1 and 88 DF,  p-value: < 2.2e-16
```

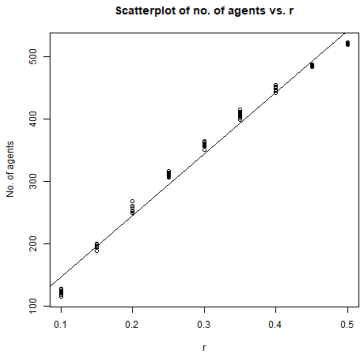
The low p-value < 0.001 (namely, the '***') implies significance following the standard threshold of 5%. In other words, the fit seems very reasonable, and y could depend on x. Although we normally cannot prove a causal relation this way for 'black box' data, it is likely in this case that the causal dependence is real because the model was constructed this way. We can thus use these results in this case as a test of whether we have implemented our model correctly.

Because we commonly expect nonlinear relations in ABMs, we would like to display our data fit in a graphical way to get more information. We type

- png(filename="scatterplotr.png")
- plot(r, ar, main="Scatterplot of no. of agents vs. r", ylab = 'No. of agents')
- abline(fitr)
- dev.off()

The command 'abline()' allows us to plot the fit into the scatterplot.

In your working directory, a png-file should appear that looks like this:



In this case, superficially it seems the linear fit is quite accurate, although a clear curvature is visible. Should you want to further test the quality of your fit, you can always check the residuals with a QQ plot or a Shapiro-Wilks test, for instance.

Look at the data

- ar

This should give:

```
> ar
 [1] 121.284 128.850 120.498 127.340 121.374 116.124 124.668 118.032 115.372
[10] ...
[82] 522.528 520.912 520.774 519.792 522.786 519.058 522.042 519.832 521.786
```

For each 'group' of ten 'observations' (fixed value of r) we can determine whether they would be likely to come from a normal distribution or not. Select for instance the first ten values (for $r = 0.10$), by taking

- ar[1:10]

```
> ar[1:10]
 [1] 121.284 128.850 120.498 127.340 121.374 116.124 124.668 118.032 115.372
[10] 124.668
```

You can do a Shapiro-Wilks test by typing

- shapiro.test(ar[1:10])

```
> shapiro.test(ar[1:10])

        Shapiro-Wilk normality test

data:  ar[1:10]
W = 0.95598, p-value = 0.7392
```

This test assumes as null hypothesis that the distribution (in this case of the values of 'ar' given $r = 0.10$) is a normal distribution. If the p-value is smaller than the chosen alpha level (typically 0.05), this null hypothesis is rejected. In this case, the p-value is clearly larger than alpha, and the null hypothesis cannot be rejected. I.e., we can assume it is a normal distribution. We would like to point out here, that you may need quite a number of samples before you can be reasonably certain of this. In this case, we have used ten samples per parameter value, but some 50 samples might be more appropriate.

You can repeat this test for each of the values of r. However, it is more appropriate to perform this test on the residuals after the fit. The residual of an observed value is the difference between the observed value (or ABM model output, in this case) and the estimated value of the quantity of interest (like the sample mean of the ABM model outputs at the same parameter value, in this case). Do not mix up residual with error, which is the deviation between the observed value and the unobservable true value

(although we might be able to calculate such a true value for simpler models, but this is not likely in the case of ABMs).

You can fetch the residuals by typing
- fitr.res <- residuals(fitr)

Now, when we perform a Shapiro-Wilks test
- shapiro.test(fitr.res)

We get:

```
> shapiro.test(fitr.res)

        Shapiro-Wilk normality test

data:  fitr.res
W = 0.92106, p-value = 4.054e-05
```

Note, that when we look at the residuals of the separate 'groups', like group 1 ($r = 0.10$), we get evidence for a normal distribution (i.e., large p-values) for each 'group'. The statistical inspection of the full set of residuals however seems to suggest that we may not have a straight line. So even if the residuals for each fixed value of r follow a normal distribution, the full set does not. This reveals that the residuals of the linear regression model differ between different values of the parameter r. In other words, there seems to be some trend in the data that is not described by the fitted straight line. Recall that the plot of the regression fit seemed to show some curvature in the dataset. This is consistent with our finding that the residuals are not normally distributed, and likely there is also a nonlinear contribution we should include in our fit.

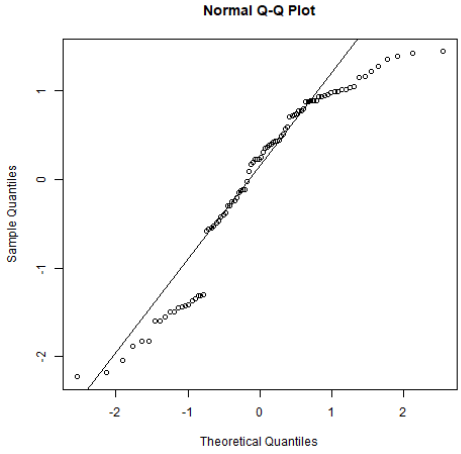
We can also plot the residuals or the standardized residuals in a QQ plot. A QQ (quantile-quantile) plot is a probability plot meant to compare two probability distributions (this could be a histogram made from data and a theoretical distribution). If the two compared distributions are similar, the points in the plot will all lie along the line $y = x$. If the distributions are linearly related, the points will lie along a line, but not necessarily on the line $y = x$.

Type:
- fitr.sres <- rstandard(fitr)

And then plot the QQ plot
- png(filename = "sresiduals.png")
- qqnorm(fitr.sres)
- qqline(fitr.sres)
- dev.off()

(The command 'dev.off()' is meant to close the plot screen again.)

We see something like this:



The results of our analysis do not provide proof, but they do provide insight in whether we maybe should not be blindly accepting a linear regression as a good fit. In this case, both the results of the Shapiro-Wilks test on the residuals and the QQ plot based on the standardized residuals seem to suggest that the relationship between r and ar (the no. of agents) is not a linear one. We would proceed, for instance, by detrending the data using a nonlinear function. From a scientific point of view, you would want to come up with a process or an explanation on which to base this nonlinear relation.

7. Interface RNetLogo

It is now time to start with the interface between R and NetLogo. ***Important***: You should have a folder 'library' in the working directory. This library should contain 'RNetLogo', as well as any other packages you are using. This way you can keep an eye on which packages you use, and you can load them into the text file you are working in by simply calling 'library(NAME_PACKAGE)'.

You can download RNetLogo here:
https://cran.r-project.org/web/packages/RNetLogo/index.html
You can also use 'install.packages()' in the text editor.

Open up a new file '02testfile.r'. The file contains the lines

- .libPaths("library")
- install.packages("RNetLogo")
- library(RNetLogo)

You can out-comment a line by putting a '#' in front; you will only need to use 'install.packages()' once.

We also need to call NetLogo. Type

- nl.path <- "C:/Program Files/NetLogo 6.0.4/app"
- nl.jarname <- "netlogo-6.0.4.jar"
- NLStart(nl.path = nl.path, gui = FALSE, nl.jarname = nl.jarname)

In the example I assume you have installed NetLogo 6.0.4 under Program Files. You will have to change this accordingly if you have installed a different version and/or in a different folder.

8. Playing with RNetLogo: fire model demo

Now, let's load a model to play with. We are going to use the NetLogo fire model. We place a copy of the model from the NetLogo library in the folder 'Model' in the folder we work in. Then type

- NLLoadModel("C:/NAME_FOLDER_YOU_WORK_IN/Model/Fire.nlogo")

RNetLogo can now make an interface to NetLogo, and you can use R to 'control' the NetLogo model without actually opening it (assuming you have first checked it for bugs).

To try, type

- NLCommand("set density", 57)
- NLCommand("setup")
- NLCommand("go")
- burned <- NLReport("burned-trees")
- print(burned)
- NLQuit()

You should get a number, which is the output of the NetLogo simulation after one iteration. The last command ('NLQuit') is to close the NetLogo simulation again, otherwise you will continue with that simulation.

It is of course very cumbersome to execute a whole set of simulations in this way. So we are going to set up a numerical experiment. Type

- data <- c()
- nruns <- 10
- nticks <- 1000

I.e., we have created a column vector containing empty elements to store our simulation results. We have created integers for how many runs we want to do, and how many iterations ('ticks' in NetLogo) they should cover. It is worthwhile to mention to put the integer 'nticks' to a large enough number; the fire model terminates only when no more 'fires are burning', which can take quite a few ticks (but a thousand should be more than enough while ensuring it will not run forever). Alternatively, you may be interested in the output at a particular time point. In that case, put it to the number you desire.

Now, we are using a for-loop. Granted, many R fans do not advocate the use of for-loops; instead, they suggest vectorization. We will address this issue in part 2 of this tutorial, but for now I leave that discussion, because as a practical person I first want things to work; the code cleaning can come later.

The whole reason for setting up this loop is to produce output. Therefore, in the loop we also need commands to write output from the model to an output file.

Type

```
for (i in 1:nruns){
        print(i)
        NLCommand("set density", 57)  # percentage
        NLCommand("setup")
        for(k in 1:(nticks+1)){
                tick <- NLReport("ticks")
                NLCommand("go")
        }
        burned <- NLReport("burned-trees")
        print(burned)
        data[i] <- burned
}
```

Summarizing, we open a for-loop for nruns simulations. For each simulation we print the iteration number 'i', we set the density to a fixed number (57), we set up the initial conditions of the model using the 'setup' command, then run the model for nticks iterations in a separate for-loop, in which the tick is reported, the NetLogo 'go' command is called, and the output 'number of burned trees' is reported. The 'number of burned trees' of the final iteration is stored in the element 'burned', which is stored in the vector 'data' we created earlier as element 'data[i]', i.e., the output at the end of the simulation is stored as the i-th element in the data vector. Be advised, that when 'nruns' is larger it may take a while before the simulations have terminated.

Note, that in order to store our simulation results in a data file, we first have to close NetLogo. Type

```
NLQuit()
```

Then, type

```
write(data, file = "simdatafire.dat", append = FALSE, sep = " ")
```

In the folder you work in, you will see a file 'simdata.dat' has appeared, containing ten values, such as:

```
610 816 795 825 787
854 875 797 644 801
```

This is the output of your numerical experiment you can use in further analysis. We will continue creating and analyzing model output like this in part 2 of this tutorial.

You can now proceed to part 2 of this tutorial.

#