R meets NetLogo: Introduction to the RNetLogo Package

Jan C. Thiele
Department of
Ecoinformatics, Biometrics
and Forest Growth
University of Göttingen
Germany

Abstract

The RNetLogo package delivers an interface to embed NetLogo into the R environment with headless (no Graphical User Interface) and interactive GUI mode. It provides functions to load models, execute commands, push values and to get values from NetLogo reporters. The interface is mostly equivalent to NetLogos' Mathematica Link `http://ccl.northwestern.edu/netlogo/docs/mathematica.html`.

Keywords: NetLogo, R, agent based modelling, abm, individual based modelling, ibm.

1. What is it all about?

GNU R (R Development Core Team 2011) is a well-known and established language and environment for statistical computing and graphics with lots of user-contributed packages. Therefore, it is the perfect tool for the analysis of agent-based models (ABM). This leads to the desire of a full-fledged integration of agent-based modeling tools and R. NetLogo (Wilensky 1999) is such an agent-based modeling tool. It is a software platform especially developed for agent-based modeling which comes with an integrated development and simulation environment. It is very easy to learn and is a potential candidate to become standard in prototyping and communicating ABMs.

With the `RNetLogo` package in R we present a tool for the integration of NetLogo in R. In short, while using R you can load models, execute commands, push data to and report back data from NetLogo. This opens a large variete of applications.

The `RNetLogo` package is inspired by NetLogo's Mathematica Link and has a very similar syntax (Bakshy and Wilensky 2007).

If you (just) want to integrate R calculations into NetLogo (the other way around!) you should have a look at the `R-Extension` to NetLogo at `http://netlogo-r-ext.berlios.de/`.

For an introduction into NetLogo see the documentation `http://ccl.northwestern.edu/netlogo/docs/` of NetLogo. An introduction into agent-based modeling using NetLogo can be found for example in (Railsback and Grimm 2012) or (Wilensky and Rand in press).

For all NetLogo users how are newbies in R: R is very well documented, see for exam-

ple the R language definition http://cran.r-project.org/doc/manuals/R-lang.html and the Wiki http://rwiki.sciviews.org/doku.php. From an R terminal you can execute `help(<command>)` to open the manual page of a command. Furthermore, lots of tutorials can be found in the web, for example (Maindonald 2008), (Venables and Team 2011), (Kabacoff 2011) or (Owen 2010) and many books are available, for example (Zuur, Ieno, and Meesters 2009), (Crawley 2005), (Kabacoff 2010) or (Venables and Ripley 2002).

2. Installation

There is nothing special to do. You must have `GNU R` (http://cran.r-project.org/) and NetLogo (http://ccl.northwestern.edu/netlogo/download.shtml) installed. The `RNetLogo` package will be installed like any other R package, see http://cran.r-project.org/doc/manuals/R-admin.html#Add_002don-packages for information on how to install a package. But `RNetLogo` requires the `rJava` package (Urbanek 2010). It could be that you have to reconfigure Java after installing `rJava` on Unix machines. This topic was discussed several times, see for example http://r.789695.n4.nabble.com/install-rJava-in-linux-td1579395.html.

3. Loading NetLogo

If we want to use the `RNetLogo` package the first time in the current R session we have to load the package, like any other packages, with

```
> library(RNetLogo)
```

or if you are working with the `RGui` on Windows by clicking on "Packages -> Load Package...". See R's FAQ page at http://cran.r-project.org/doc/FAQ/R-FAQ.html#\How-can-add_002don-packages-be-used_003f for details.

When loading `RNetLogo` it will automaticly try to load `rJava`. If this runs without any error we are ready to start NetLogo (if not, see section 2). To do so, we have to know where NetLogo is installed. What we need is the path to the folder that contains the `NetLogo.jar` file. On Windows machines this could be "C:/Program Files/NetLogo 5.0.4/".

Now, we have to decide, whether we want to run NetLogo in the background without seeing the Graphical User Interface (GUI) and control NetLogo completly from R or if we want to see and use the NetLogo GUI. In the latter case, we can use NetLogo as it was started independently, i.e. can load models, change the source code, click on buttons, see the NetLogo View, inspect agents and so on, but have also controll over NetLogo from R. But the disadvantage by starting NetLogo with GUI is that you cannot run multiple instances of NetLogo in one R session. This is just possible in the so called headless mode, i.e. running NetLogo without GUI (see section 9 for details). Since the NetLogo Controlling API changes with the NetLogo version, you have to use an extra parameter `nl.version` to start RNetLogo for NetLogo version 4 (nl.version=4 for NetLogo 4.1.x, nl.version=40 for NetLogo 4.0.x). The default value of `nl.version` is 5, which means, that we don't have to submit this parameter when using NetLogo 5.0.x.

To keep it simple and comprehensible we start NetLogo with GUI by typing

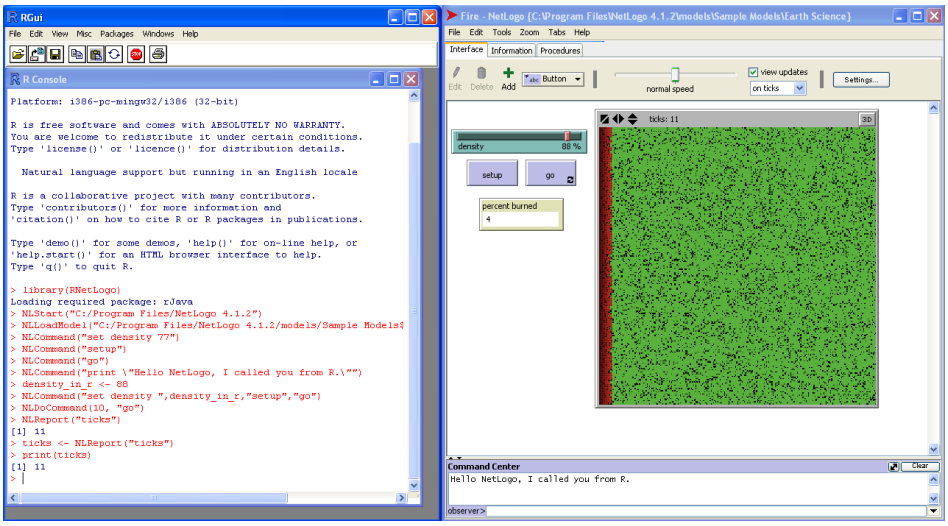

Figure 1: NetLogo (on the right) started and controlled from R (on the left).

```
> nl.path <- "C:/Program Files/NetLogo 5.0.4"
> NLStart(nl.path)
```

If everything goes right, a NetLogo Window will be opened. Please note that R cannot use single backslashes in paths. You should use either forwardslashes or doubled backslashes. As described, we can use the NetLogo Window as we know it from independend startups, with the exception, that we cannot close the Window by clicking. On Windows, NetLogo appears in the same program group at the taskbar as R. For Mac OS and Linux users it can be necessary to run RNetLogo from within JGR (an java-based R GUI available as R package, see http://cran.r-project.org/web/packages/JGR/index.html) when using the GUI mode. If possible, we should arrange the R and NetLogo window so that we have them side by side, as shown in Fig 1, and can see what is happening in NetLogo if we submit the following examples.

4. Loading a model

We could now open a NetLogo model by just clicking on "File -> Open..." or choosing one of the sample models by clicking on "File -> Model Library". But, to learn controlling NetLogo from R, as we would need to do when starting NetLogo in headless mode, we type in R:

```
> model.path <- "/models/Sample Models/Earth Science/Fire.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep=""))
```

The forest fire model (Wilensky 1997a) should be loaded. If we want, we could now change the initial tree density be using the slider on the Interface Tab and run the simulation by clicking on the setup button first and then on the go button.

Note: You can copy and paste the R code in this tutorial from the PDF or, preferably, Sweave file.

Now we will do the same by controlling NetLogo from R in the next section.

5. Principles of controlling a model

In a first step we will change the density value, i.e. the position of the density slider, by submitting the following statement in R:

```
> NLCommand("set density 77")
```

The slider goes immediatly in the position of 77 percent. We could now execute the `setup` procedure to initialize the simulation. We just submit in R:

```
> NLCommand("setup")
```

And again, the command is executed immediatly. The tick counter is reset to 0, the View is green and first fire turtles are found on the left side of the View. Please notice, that the `NLCommand` function does not press the `setup` button, but calls the `setup` procedure. In the forest fire example it makes no difference as the setup button also just calls the `setup` procedure, but it is possibe to add more code to a button than just calling a procedure. But we could copy and paste such code into the `NLCommand` function as well. Now that we know this, we will go ahead with our simulation.

After we have setup our simulation by now, we want to run one simulation step by executing the `go` procedure. This is nothing new; we just submit in R:

```
> NLCommand("go")
```

We see, that the tick counter was incremented by one and the red line of the fire turtles on the left of the View extended to the next patch.

As we have seen, the `NLCommand` function can be used to execute any command which could be typed into NetLogo's Command Center. We could, for example, print a message into NetLogo's Command Center with the following statement:

```
> NLCommand("print \"Hello NetLogo, I called you from R.\"")
```

The backslashes in front of the quotation marks are used to "mask" the quotation marks, otherwise R would think that the command string ends after the `print` and would be confused. Furthermore, it is possible to submit more than one command at once and in combination with R variables. We could change the density slider and execute `setup` and `go` with one `NLCommand` call like this:

```
> density.in.r <- 88
> NLCommand("set density ",density.in.r,"setup",
+           "go")
```

Let us come back to our forest fire simulation. In most cases, we do not want to execute a go procedure only a single time but run it for, say ten times. With the `RNetLogo` package we can do this with:

```
> NLDoCommand(10, "go")
```

By now, we ran the simulation eleven times and we are maybe interested to have this information in R. Therefore, we execute:

```
> NLReport("ticks")
```

```
[1] 11
```

As you might expect, we can save this value in an R variable by typing:

```
> ticks <- NLReport("ticks")
> print(ticks)
```

```
[1] 11
```

Now, you know the basic principles of the functionality of the `RNetLogo` package. The things presented in the following are mostly modifications and/or extensions to this basic functionality.

NetLogo users should note, that there is no "forever button". To run a simulation for several time steps use one of the loop function (`NLDoCommand`, `NLDoCommandWhile`, `NLDoReport`, `NLDoReportWhile`) or write a custom procedure in NetLogo to be called by R.

6. Quit NetLogo session

To quit a NetLogo session, i.e. to close a NetLogo instance, you have to use the `NLQuit` function. If you used the standard GUI mode without submitting an user-defined variable to save the NetLogo reference in, you can write:

```
> NLQuit()
```

Otherwise, you have to specify, which NetLogo instance you want to close by giving the R variable which stores the NetLogo reference as an argument to the function. Please note, that there is currently no way to close the GUI mode completely. That is why you cannot run `NLStart` again in the same R session when started with GUI.

7. Advanced controlling functionalities

In the second last section we used the `NLDoCommand` function to run the simulation for ten times. Here, we will run the model for ten times as well, but we will collecte the percentage of burned trees after every simulation step automaticly. Try this:

```
> NLCommand("setup")
> burned <- NLDoReport(10,
+                      "go",
+                      "(burned-trees /
+                        initial-trees) * 100")
> print(burned)
```

```
[[1]]
[1] 0.4122295

[[2]]
[1] 0.7666022

[[3]]
[1] 1.128207

[[4]]
[1] 1.504276

[[5]]
[1] 1.855033

[[6]]
[1] 2.222062

[[7]]
[1] 2.594514

[[8]]
[1] 2.954311

[[9]]
[1] 3.30326

[[10]]
[1] 3.684753
```

This code ran the simulation for ten ticks and wrote the result of the given reporter (the result of the calculation of the percentage of burned trees) after every tick into the R-List `burned`.

If we would like to run the simulation until no trees are left, while we want to know the precentage of burned-trees in every simulation step we could excute (the result is shown in figure 2):

```
> NLCommand("setup")
> burned <- NLDoReportWhile("any? turtles",
+                           "go",
+                           c("ticks","(burned-trees /
+                             initial-trees) * 100"),
+                           as.data.frame=TRUE,
+                           df.col.names=c("tick","burned"))
> plot(burned, type = "s")
```

In this short example we find some new things. The first argument of the function takes

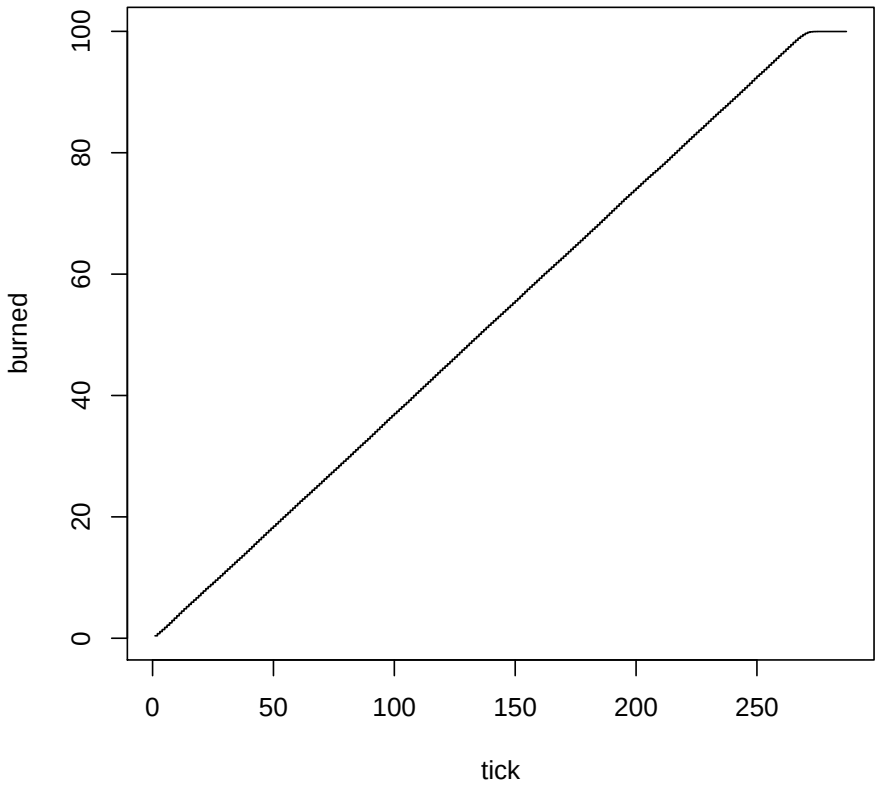

Figure 2: A plot of the percentage of burned trees over time as the result of NLDoReportWhile, which runs as long as there are turtles (any? turtles).

a NetLogo reporter. Here, the `go` procedure will be executed while there are turtles in the simulation, i.e. `any? turtles` reports true. Moreover, we used not just one reporter (third argument) but used a vector of two reporters. One returning the current time and a second with the percentage of burned trees. Furthermore, we have defined that our output should be saved as a data.frame instead of a list and we have defined the names of the columns of the data.frame by using a vector of strings in correspondence to the reporters we have given. At the end, the R variable `burned` is of type data.frame and contains two columns. One with the simulation time and a second with the corresponding percentage of burned trees. By using the standard plot function, we will get a graph with the development of the percentage of burned trees over time.

8. Further examples

For the demonstration of the `NLGetAgentSet` function, we will use a different model. Therefore, we load the tumor model from NetLogo's Model Library, set it up and run it for 20 steps.

```
> model.path <- "/models/Sample Models/Biology/Tumor.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep=""))
> NLCommand("setup")
> NLDoCommand(20, "go")
```

After we have run 20 time steps we will load the x and y positions of all cells (i.e. turtles) into a data.frame and show them in a plot. But before we call the plot function, we will get the spatial extension of the NetLogo World to use it for the plot window (the resulting plot is shown in figure 3).

```
> cells <- NLGetAgentSet(c("xcor","ycor"),
+                        "turtles")
> x.minmax <- NLReport("(list min-pxcor max-pxcor)")
> y.minmax <- NLReport("(list min-pycor max-pycor)")
> plot(cells, xlim=x.minmax, ylim=y.minmax, xlab="x", ylab="y")
```

In a second step, we will get only the metastatic cells and plot them again (the result is shown in figure 4:

```
> cells.metastatic <- NLGetAgentSet(c("xcor","ycor"),
+                                   "turtles with [metastatic? = True]")
> plot(cells.metastatic, xlim=x.minmax, ylim=y.minmax, xlab="x", ylab="y")
```

We can use the `NLGetAgentSet` function to get patches and links as well. But there is a special function for patches, called `NLGetPatches`, which makes life easier. We will test this function by using the Fur model about patterns on animals' skin self-organization and plot the result in a simple raster image (see figure 5).

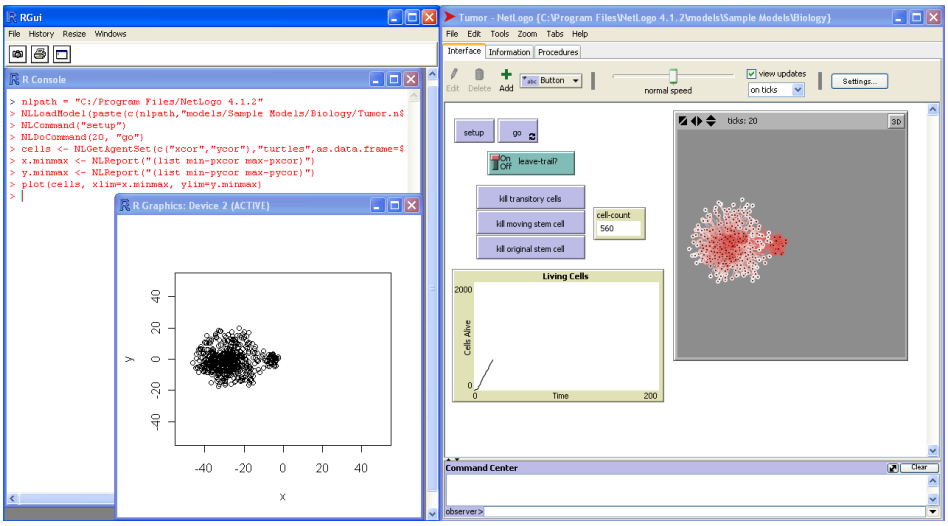

Figure 3: A visualization of the turtles by getting them via NLGetAgentSet. On the right hand side the original NetLogo simulation and on the left hand side the RGUI with the visualization of the reported turtles.

```
> model.path <-"/models/Sample Models/Biology/Fur.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep=""))
> NLCommand("setup")
> NLDoCommand(5, "go")
> # get patches as matrix
> patches.matrix <- NLGetPatches("pcolor","patches",
+                                as.matrix=TRUE)
> # rotate matrix to fit into image function
> patches.matrix.rot <- t(patches.matrix)
> patches.matrix.rot <- as.data.frame(patches.matrix.rot)
> patches.matrix.rot <- rev(patches.matrix.rot)
> patches.matrix.rot <- as.matrix(patches.matrix.rot)
> # set colors for image
> col <- c("black", "white")
> # get x and y limits
> x.minmax <- NLReport("(list min-pxcor max-pxcor)")
> y.minmax <- NLReport("(list min-pycor max-pycor)")
> # draw matrix as image
> image(x.minmax[1]:x.minmax[2], y.minmax[1]:y.minmax[2],
+       patches.matrix.rot, col=col, xlab="", ylab="")
```

The code produced a simple raster image from the patches. It is also possible to create a spatial object from the result of `NLGetPatches` as we see in the next example (the result is shown in figure 6, Package gstat (Pebesma 2004) and sp (Pebesma and Bivand 2005) are used):

```
> # the following operations require the gstat
> # and sp package
```

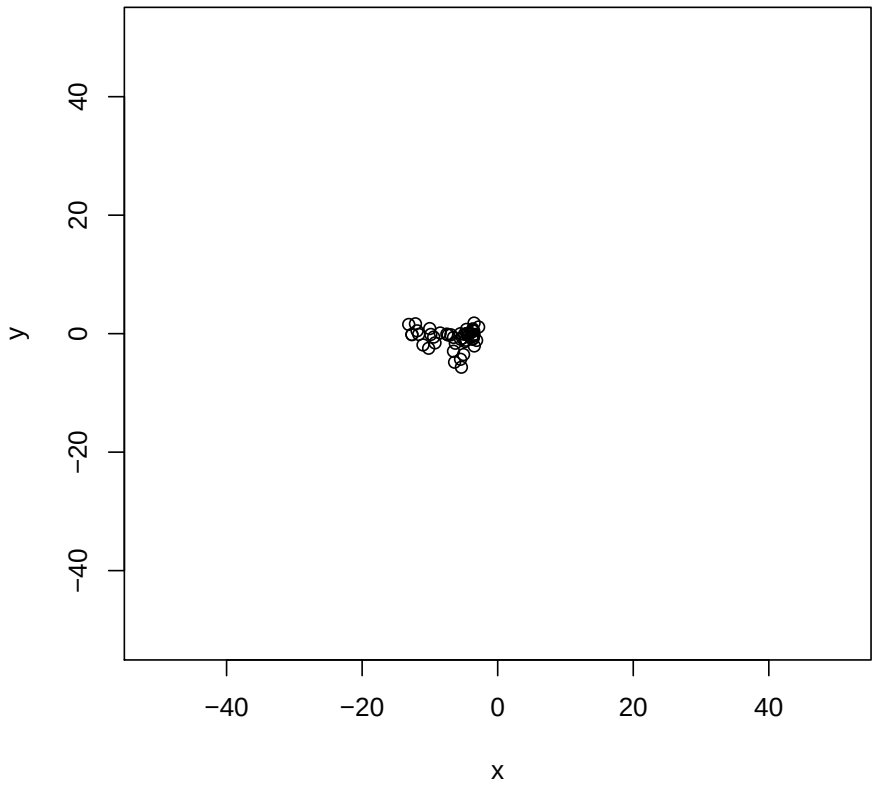

Figure 4: Same as in figure 3 but with a subset of turtles that fulfill a condition (are metastatic cells).

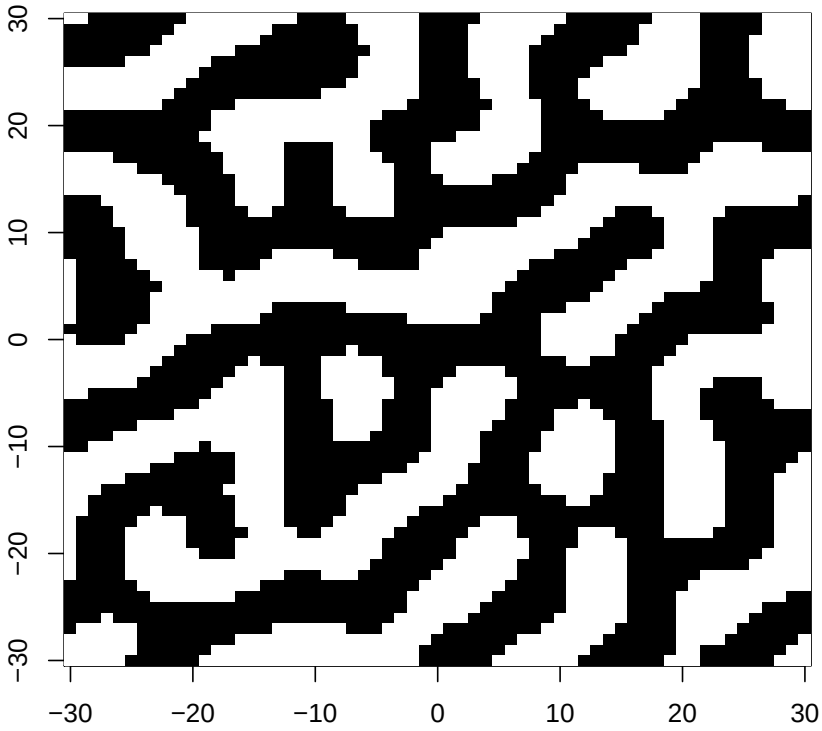

Figure 5: A simple visualization of the result of NLGetPatches as an image.

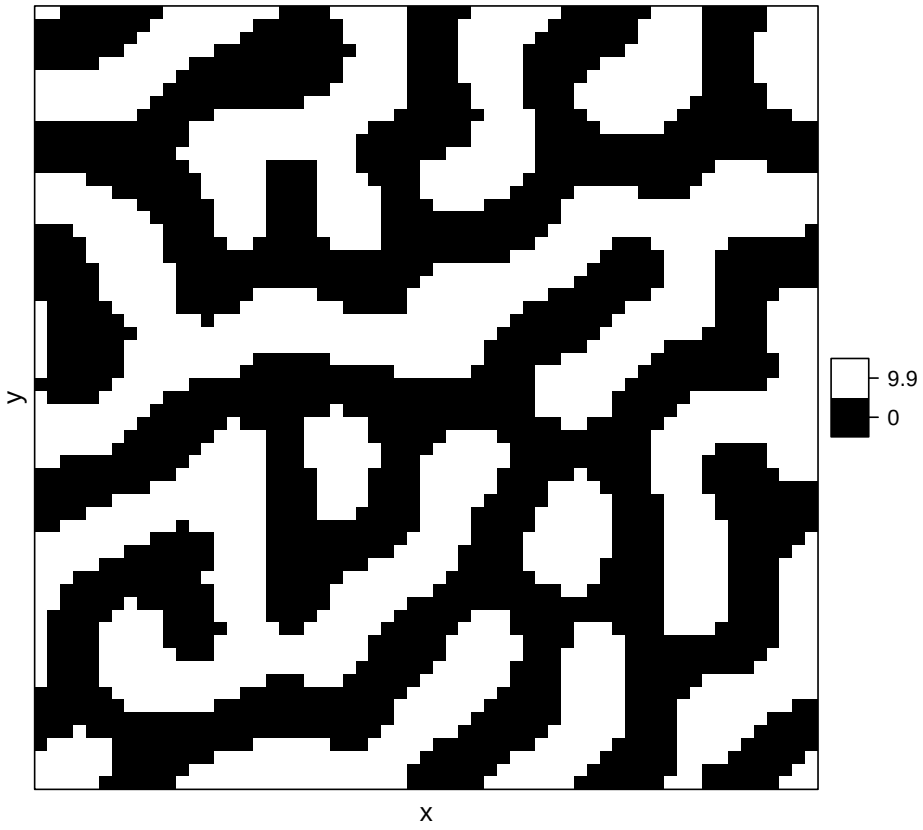

Figure 6: Same as in figure 5 but the result of NLGetPatches is converted to a spatial pixels data.frame and plotted with spplot from package sp.

```
> library(sp)
> library(gstat)
> # get patches
> patches <- NLGetPatches(c("pxcor","pycor","pcolor"),
+                         "patches")
> # convert to SpatialPointsDataFrame
> coordinates(patches) <- ~ pxcor + pycor
> # convert to SpatialPixelsDataFrame
> gridded(patches) <- TRUE
> # convert pcolor to factor
> patches$pcolor <- factor(patches$pcolor)
> # set colors for plot
> col <- c("black", "white")
> # create and show plot
> spplot(patches, 'pcolor', col.regions=col, xlab="x", ylab="y")
```

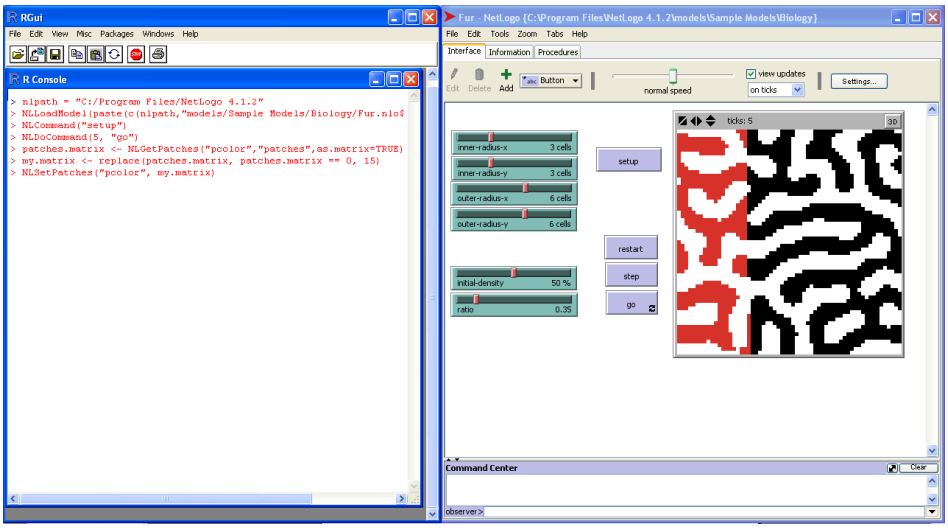

Figure 7: A screenshot while NLSetPatches is executed. The color of the NetLogo patches on the right hand side are changed gradually from black to red.

We see, that it is possible to get the whole NetLogo View. As you can see in the documentation, it is possible to save the result of `NLGetPatches` into a list, matrix or, like here, into a data.frame. Furthermore, we can reduce the patches to a subset, e.g. all patches that fullfill a condition, like we have done in the `NLGetAgentSet` example.

There is another function, which will do the otherway around. With `NLSetPatches` we can push an R matrix into the NetLogo patches. But this function works only if we fill all patches, i.e. if we use a matrix which has the dimension of the NetLogo World. We cannot fill just a subset of patches (see the Code Examples for hints how to do such things). Have a look on the following example. The result is shown in figure 7.

```
> # reuse the patches.matrix from NLGetPatches and
> # change values from 0 (black) to 15 (red)
> # (attention: this is relatively slow due to the drawing procedure)
> my.matrix <- replace(patches.matrix,
+                      patches.matrix == 0,
+                      15)
> # use this matrix as input for "pcolor"
> NLSetPatches("pcolor", my.matrix)
```

The `NLGetGraph` function makes it possible to get a NetLogo network build by NetLogo links into an `igraph` network. This function requires that we have the R package `igraph` installed. As an example we could use the Small World model from NetLogo's Model Library. We will build the NetLogo link network and transform it into an igraph network and finally plot it (see figure 8).

```
> model.path <- "/models/Sample Models/Networks/Small Worlds.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep=""))
> NLCommand("setup","rewire-all")
```

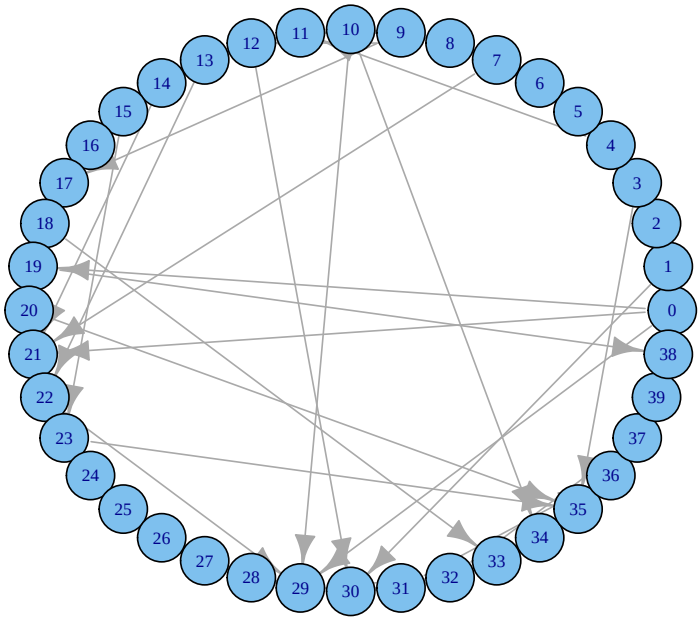

Figure 8: A visualization of the graph using the igraph package. The graph was generated by NetLogo links and send to R via NLGetGraph.

```
> my.network <- NLGetGraph()
> # plot the directed network graph
> plot(my.network, layout=layout.circle,
+      vertex.label=V(my.network)$name,
+      vertex.label.cex=0.7,
+      asp=FALSE)
```

There are two further functions, which will not be presented here in detail. The first one is the `NLSourceFromString` function, which enables you to create or append model source code from strings in R. See the code sample folder (No. 14) for an example. Another helper function to send a data.frame into NetLogo lists is delivered with the `NLDfToList` function. The column names of the data.frame have to be equivalent to the names of the lists in the NetLogo model. See the code sample folder (No. 9) for an example.

9. Headless mode/Multiple NetLogo instances

As mentioned above, it is possible to start NetLogo in background (headless mode) without a Graphical User Interface (GUI). For this, we have to execute the `NLStart` function with a second argument (This will fail, if you do not open a new session, because, as mentioned above, you cannot start several NetLogo sessions if you have already started one session in GUI mode.):

```
> NLStart(nl.path, gui = FALSE)
```

Please note that it is not possible to start more than one instance in GUI mode.

This will save the NetLogo object reference in a variable `_nl.intern_` in the local environment `.rnetlogo`. If we want to work with more than one NetLogo model/instance at once, we can specify an own variable name (as a string) where to save the NetLogo instance in the third argument of `NLStart` like this:

```
> # a first NetLogo instance
> # (beside the one with the default variable name (_nl.intern_) )
> my.netlogo1 <- "my.netlogo1"
> NLStart(nl.path,
+         gui=FALSE,
+         nl.obj=my.netlogo1)
> # a second NetLogo instance
> my.netlogo2 <- "my.netlogo2"
> NLStart(nl.path,
+         gui=FALSE,
+         nl.obj=my.netlogo2)
> # a third instance
> my.netlogo3 <- "my.netlogo3"
> NLStart(nl.path,
+         gui=FALSE,
+         nl.obj=my.netlogo3)
```

All functions, presented above, take as last (optional) argument the referece name of the NetLogo instance (nl.obj). Therefore, we can specify which instance we want to use. When working in headless mode, the first thing to do is always to load a model. Executing a command or reporter without loading a model in headless model will result in an error. Let us load a model into all instances.

```
> # load model in the first instance (my.netlogo1)
> model.path <- "/models/Sample Models/Earth Science/Fire.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep=""),
+             nl.obj=my.netlogo1)
> # in the second instance (my.netlogo2)
> NLLoadModel(paste(nl.path,model.path,sep=""),
+             nl.obj=my.netlogo2)
> # a third instance
> NLLoadModel(paste(nl.path,model.path,sep=""),
+             nl.obj=my.netlogo3)
```

Now, we will setup and run the models over different simulation times.

```
> # setup and run the model in the first instance (my.netlogo1)
> # for 25 time steps
> NLCommand("setup", nl.obj=my.netlogo1)
> NLDoCommand(25,"go", nl.obj=my.netlogo1)
> # in the second instance (my.netlogo2) for 15 time steps
> NLCommand("setup", nl.obj=my.netlogo2)
> NLDoCommand(15,"go", nl.obj=my.netlogo2)
> # a third instance for 5 time steps
> NLCommand("setup", nl.obj=my.netlogo3)
> NLDoCommand(5,"go", nl.obj=my.netlogo3)
> # show number of burned-trees in the different instances
> NLReport("burned-trees", nl.obj=my.netlogo1)

[1] 1557

> NLReport("burned-trees", nl.obj=my.netlogo2)

[1] 921

> NLReport("burned-trees", nl.obj=my.netlogo3)

[1] 525

> # quit NetLogo sessions
> NLQuit(nl.obj=my.netlogo3)
> NLQuit(nl.obj=my.netlogo2)
> NLQuit(nl.obj=my.netlogo1)
> NLQuit() # the standard session as well, if open
> # alternatively we can call:
> # NLQuit(all=TRUE)
```

10. Pitfalls

10.1. Amount of data

Please note, that you are not able to stop an execution of a NetLogo command, whitout closing your R session. Therefore, it is a good idea to think about the amount of data which should be transformed. For example, if you use the `NLGetPatches` function with the standard settings of the Fire.nlogo model from NetLogo's Model Library, you are requesting 63001 patch values. If you ask for the pxcor, pycor and pcolor values than you request for 63001 * 3 = 189003 values. All these values have to be transformed from NetLogo data type to Java and from Java to R. This may take a while. For technical reasons you are not informed about the

process of data transformation. Therefore, it looks like the programm crashed, but if you are patient, the programm will return with the result after some time. That's why it is always a good idea to test your code with a very small example (i.e. small worlds, low number of agents etc.). You should know, that NetLogo 5.0 is extremly faster in transfering data than NetLogo 4.x.

10.2. Endless loops

If you use the functions `NLDoCommandWhile` and `NLDoReportWhile` please double check your while-condition. Are you sure, that the condition will be met some time? Use this carefully, otherwise it will end up in an endless loop and you will wait for the end of the execution as long as you wish.

10.3. Data structure

You should always think about the requested data structure before you call a function. Have a look on the following examples:

```
> # load the well-known Fire model
> model.path <- "models/Sample Models/Earth Science/Fire.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep="/"))
> # initialize the model
> NLCommand("setup")
> # run the model for 10 times and report two values as a list after every
> #step and save them in a data.frame (1. NLDoReport call)
> df1 <- NLDoReport(10,"go","(list count fires count embers)",
+                   as.data.frame=T)
> str(df1)
> NLCommand("setup")
> # 2. NLDoReport call
> df2 <- NLDoReport(10,"go",c("count fires","count embers"),as.data.frame=T)
> str(df2)
> NLCommand("setup")
> # 3. NLDoReport call
> df3 <- NLDoReport(10,"go",c("count turtles",
+                   "(list count fires count embers)"),as.data.frame=T)
> str(df3)
> NLCommand("setup")
> # 4. NLDoReport call
> df4 <- NLDoReport(3,"go","[(list who xcor ycor)] of turtles",
+                   as.data.frame=T,df.col.names=c("who","xcor","ycor"))
> str(df4)
```

The first `NLDoReport` call results in a data.frame with two columns and ten rows. Because we requested a NetLogo list, the package transforms this list to an R vector. If you put an R vector to a data.frame it will be automaticly splitted into several columns. This result is equivalent to the result of the second `NLDoReport` call, where we used an R vector as request. But in the third call of `NLDoReport`, the NetLogo list will not resolved to different columns,

because it is not directly accessible but nested in the result with another value. It will not result in an error. But you see, that the column X2 contains a list instead a vector. If you try, for example, to use the `write.table` function on this data.frame, if will raise an error. In pure R, you would have to use the I(x) function (see `help(I)`) to create such a data.frame. Therefore, you have to add this property to the X2 column. This could be done by executing `df3$X2 <- I(df3$X2)`. If we expected with the fourth call of `NLDoReport` to get a data.frame with three columns containing the who, xcor and ycor variables, we will be disappointed. If you copy and paste the reporter `[(list who xcor ycor)] of turtles` into the NetLogo Command Center, you will see that it creates a NetLogo List with nested lists fo each turtle. This NetLogo list structure is transformed into R lists. So far, for every simulation step the `NLDoReport` creates an R list with a nested R list for each turtle. What the as.data.frame in `NLDoReport` tries to do is to append the result of one iteration (here one simulation step) to a data.frame, i.e. it creates one row of a data.frame from the R list which contains the nested R lists for each turtle. By doing this, the base R list will be resolved to columns and each column will contain three rows (the variables who, xcor and ycor of the turtle). Because this structure is not appendable, it will be overwritten in every iteration. At the end, we have this (maybe unexpected) data.frame with a column for each turtle of the last iteration. Remember: The as.data.frame variable transform the last result of a function to a data.frame. Think about the expected structure and decide if the input data for the data.frame are suffient.

10.4. Data type

Please think about the data type you are trying to combine. For example, an R vector takes values of just one data type (e.g. String, Numeric/Double or Logical/Boolean) unlike a NetLogo list, where you can combine different data type in one list. Try this:

```
> # a NetLogo list of numbers
> NLReport("(list 24 23 22)")
> # a NetLogo list of strings
> NLReport("(list \"foo1\" \"foo2\" \"foo3\")")
> # a NetLogo list of combined numbers and string
> NLReport("(list 24 \"foo\" 22)")
```

The first two calls of `NLReport` will run as expected but the last call will throw an error, because `NLReport` tries to create a NetLogo into an R vector, which will fail due to the mixed data types. This is also very relevant for the columns of data.frames.

10.5. Working directory

You should avoid to manually change the working directory of R, because NetLogo need to have the working directory pointed to its installation path. As the R working directory and the Java working directory depend on each other, changing the R working directory can result in unexpected behavior of NetLogo. There, you should use absolute paths for I/O processes in R instead of submitting setwd(...). Note, that the RNetLogo package changes the working directory automatically when loading NetLogo and changes back to the former working directory when submitting `NLQuit`.

10.6. Combinations with the R-Extension

If you want to use the R-Extension within your NetLogo model which should be controlled via RNetLogo, please note that this will crash as it is not possible to load the JRI-library when rJava is acitve. But to give you the opportunity to have access to R from the NetLogo side, the Rserve-Extension was created. It has the same syntax as the R-Extension (with the exception that you have to change the r:... calls to rserve:...) but uses the Rserve server technology. The Rserve-Extension is available for download at the Berlios repository (http://netlogo-r-ext.berlios.de/). See the documentation of the Rserve-Extension for details on how to get it running.

11. Application example

The following examples are (partly) taken from the examples given for NetLogo's Mathematica Link (see (Bakshy and Wilensky 2007)). These are all one directional examples (from NetLogo to R), but the package opens the possiblity, to let NetLogo and R interact with eachother and and send back results from R (e.g. statistical analysis) to NetLogo and let the model react on these results. Even the manipulation of the model source by using the `NLSourceFromText` function is possible

11.1. Analytical comparison

In the application example of (Bakshy and Wilensky 2007) they compare an agent-based model of gas particles with velocity distributions found in analytical treatments of ideal gases. For this, we use the Free Gas model (Wilensky 1997b) of the GasLab model family from NetLogo's Model Library. In this model, gas particles are moving and colliding with each other without external constraints. (Bakshy and Wilensky 2007) compared this model with classical Maxwell-Boltzmann distribution. R itself is not a symbolic mathematical software but there are packages available which let us integrate such software. Here, we use the Ryacas packages (Goedman, Grothendieck, Højsgaard, and Pinkus 2010) which is an interface to the opensource Yacas Computer Algebra System (Pinkus and Niesen 2007).

We start with the agent-based model simulation. Since this model is based on random numbers we run repeated simulations. The resulting probability distribution of particle speeds is shown in figure 9.

```
> # load RNetLogo package (if not already done)
> library(RNetLogo)
> # path to NetLogo installation
> nl.path <- "C:/Program Files/NetLogo 5.0.4"
> # initialize NetLogo (if not already done)
> NLStart(nl.path, gui=FALSE)

> # load Gas Lab model from model library
> model.path1 <- "models/Sample Models/Chemistry & Physics/GasLab"
> model.path2 <- "GasLab Free Gas.nlogo"
> NLLoadModel(paste(nl.path,model.path1,model.path2,sep="/"))
> # initialize simulation
```

```
> NLCommand("set number-of-particles 500", "no-display", "setup")
> # run simulation for 40 times of 50 steps (= 2000 simulation steps)
> # save speed of particles after every 50 simulation step interval
> particles.speed <- NLDoReport(40, "repeat 50 [go]", "[speed] of particles")
> # flat the list of lists (one list for each of the 40 runs)
> # to one big vector
> particles.speed.vector <- unlist(particles.speed)
```

To calculate the analytical distribution, we have to calculate the following equation:

$$B(v) = v * e^{\frac{-mv^2}{2kT}} \tag{1}$$

$$normalizer = \int_{\infty}^{0} B(v)\ dv \tag{2}$$

$$B(v)_{normalized} = \frac{B[v]}{normalizer} \quad for\ v = [0, max(speed)] \tag{3}$$

Now, Yacas/RYacas will be used. We will first define the equation B(v) with the mean energy derived from the NetLogo simulation. We then define the normalizer integral and solve it numerically.

```
> # load the RYacas pacakge
> library(Ryacas)
> # install Yacas, if currently not installed
> # (just for Windows - see Ryacas documentation for other systems)
> #yacasInstall()
> # get mean energy from NetLogo simulation
> energy.mean <- NLReport("mean [energy] of particles")
> # definition of function B
> B <- function(v, m=1, k=1) v * exp((-m*v^2)/(2*k*energy.mean))
> # register function B in Yacas
> yacas(B)

[1] "Starting Yacas!"
expression(TRUE)

> # define integration function B from 0 to endless
> B.integr <- expression(integrate(B,0,Infinity))
> # register intergration expression in Yacas
> yacas(B.integr)

expression(defint(integer_interval(0, Inf), lambda(x, B(x))))

> # calculate a numerical approximation using Yacas function N()
> normalizer.yacas <- yacas(N(B.integr))
```

```
> # get result from Yacas in R
> normalizer <- Eval(normalizer.yacas)
> # the numeric result is in column value
> print(normalizer$value)

[1] 50
```

In a second step, we calculate the theoretical probability values of particle speeds using the equation `B(v)`. We do this from 0 to the maximum speed observed in the NetLogo simulation.

```
> # get max. speed from NetLogo simulation
> maxspeed <- max(particles.speed.vector)
> # create a sequence vector from 0 to maxspeed + stepsize, by stepsize
> stepsize <- 0.25
> v.vec <- seq(0, maxspeed, stepsize)
> # calculate the theoretical value at the points of the sequence vector
> theoretical <- B(v.vec) / normalizer$value
```

At the end, we plot the empirical/simulation distribution together with the theoretical distribution of particle speeds, shown in figure 9.

```
> hist(particles.speed.vector, breaks=max(particles.speed.vector)*5,
+      freq=FALSE, xlim=c(0,as.integer(maxspeed)+5),
+      main="Histogram of empirical speed together \n
+      with theoretical Maxwell-Boltzmann distribution",
+      xlab="speed of particles")
> lines(v.vec, theoretical, lwd=2, col="blue")
```

11.2. Exploratory analysis

With the RNetLogo package it is relatively simple to explore parameters spaces and save as well as analyse the results. Such technics are very relevant in agent-based modelling regarding exploratory analysis, parameter fitting and sensitivity and robustness analysis (Grimm and Railsback 2005). Here, we will use Fire model (Wilensky 1997a) from NetLogo's Model Library and explore the effect of the density of trees in the forest on the percentage of burned trees as described in (Bakshy and Wilensky 2007). We start with initialization of NetLogo.

```
> # load RNetLogo package (if not already done)
> library(RNetLogo)
> # path to NetLogo installation
> nl.path <- "C:/Program Files/NetLogo 5.0.4"
> # initialize NetLogo (if not already done)
> NLStart(nl.path, gui=FALSE)

> model.path <- "models/Sample Models/Earth Science/Fire.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep="/"))
```

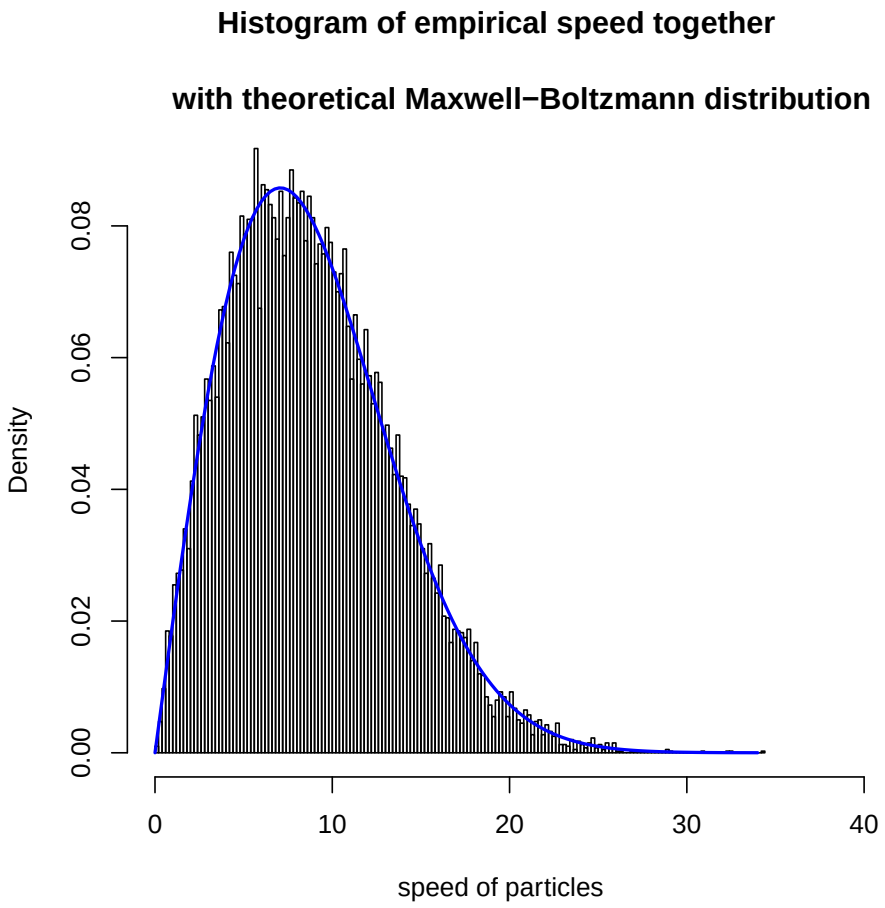

Figure 9: Empirical probability distribution of particle speeds generated by agent-based model (bars) with theoretical Maxwell-Boltzmann distribution (blue line).

Next, we define a function which sets the density of trees, exectues the simulation until no turtles are left and reports back the percentage of burned trees.

```
> # function to simulate with specific density
> sim <- function(density) {
+     NLCommand("set density ", density, "setup")
+     NLCommand("while [any? turtles] [go]");
+     ret <- NLReport("(burned-trees / initial-trees)")
+     return(ret)
+ }
```

We run the simulation for density value between 1 and 100 with a stepsize of 1, for identifying the phase transition (see figure 10).

```
> # perform simulation with density: 1-100, stepwidth: 1, (i.e.
> # call function sim for all values in sequence from 1 to 100)
> # and plot the result
> plot(d <- seq(1,100,1),
+      sapply(d, function(dens) {sim(dens)}),
+      xlab="density", ylab="percent burned")
```

As we know the region of phase transition (between a density of 45 and 70 percent), we can explore this region more precisely. As the Fire model uses random numbers, it is interesting to find out how much variation occur in this region. Therefore, we define a function, which will repeat the simulations with one density several times.

```
> # function to perform replicated simulations
> rep.sim <- function(density, rep) {
+   return(
+     lapply(density, function(dens) {
+       replicate(rep, sim(dens))
+     })
+   )
+ }
```

To get a rough overview we use this new function for densities between 45 and 70 percent with a stepsize of 5 and 10 replications each. The resulting boxplots are shown in figure 11.

```
> # density: 45-70, stepwidth: 5, 10 replications
> d <- seq(45,70,5)
> res <- rep.sim(d,10)
> boxplot(res,names=d, xlab="density", ylab="percent burned")
```

Now, we have seen that the variation of burned trees at densities below 55 and higher 65 is low. As a result, we can skip these values and will have a deeper look into the region of density values between 55 and 65. Therefore, we perform a simulation experiment for this value range with a smaller stepsize of 1 percent and a higher amount of replication of 20 per density value.

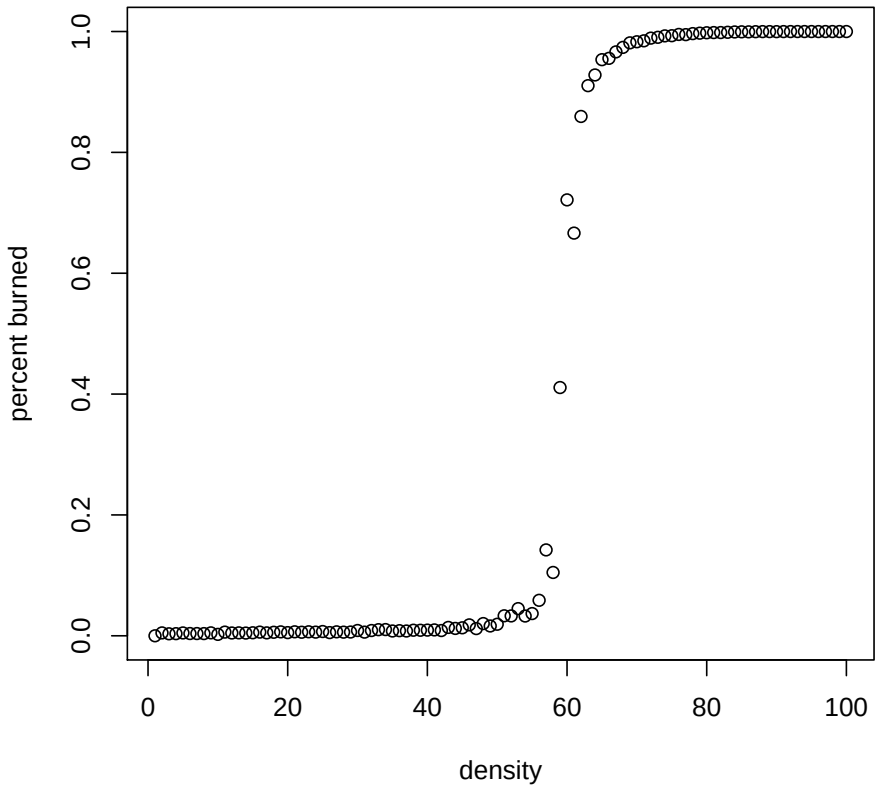

Figure 10: Simulations with the Fire model with varying density of trees and the percentage of burned trees after no burning patches (i.e. no turtles) were left in the simulation.

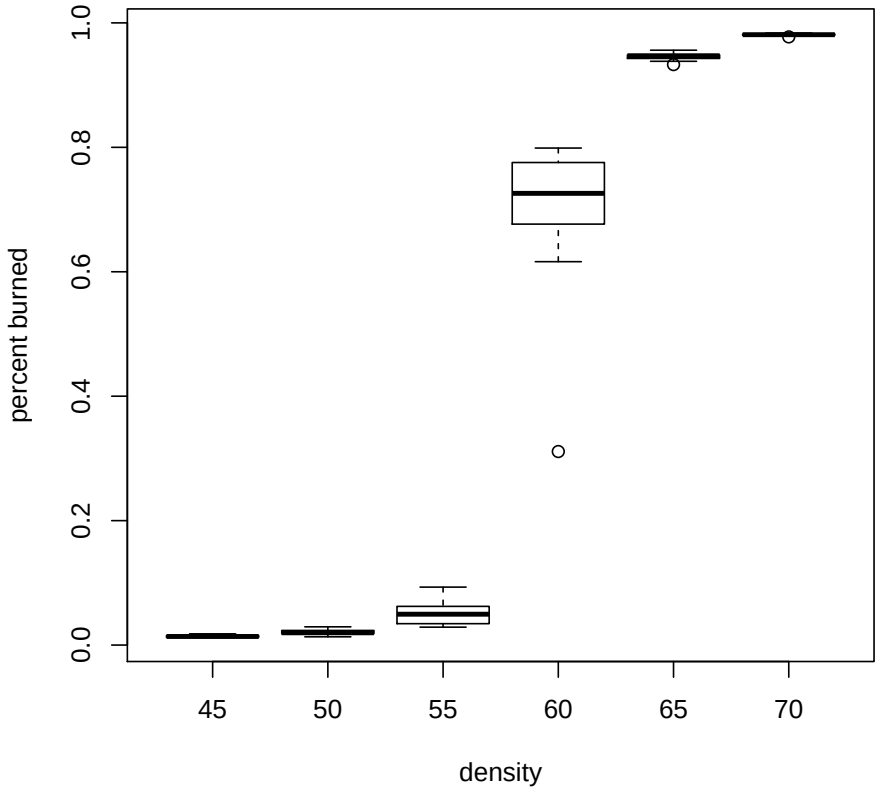

Figure 11: Boxplot of repeated simulations (10 replication) with the Fire model with varying density (45-70 percent) of trees and the percentage of burned trees after no turtles were left in the simulation.

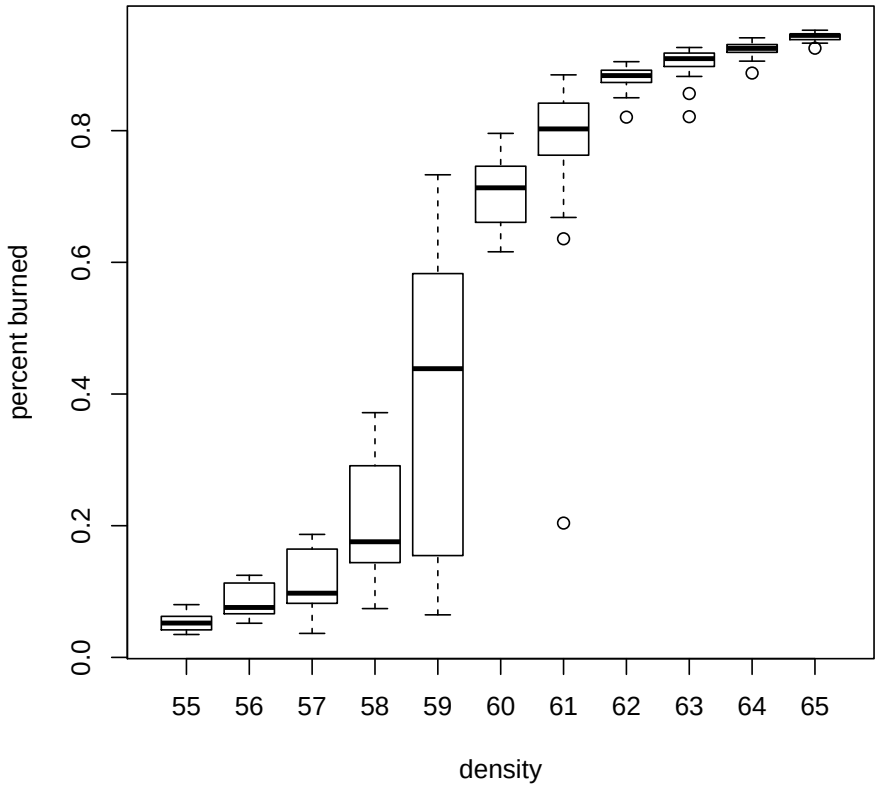

Figure 12: Boxplot of repeated simulations (20 replication) with the Fire model with varying density (55-65 percent) of trees and the percentage of burned trees after no turtles were left in the simulation.

```
> # density: 55-65, stepwidth: 1, 20 replications
> d <- seq(55,65,1)
> res <- rep.sim(d,20)
> boxplot(res,names=d, xlab="density", ylab="percent burned")
```

Note, that if you are not interested in further (statistical) analysis of the results of repeated simulations, you can perform similar experiments with NetLogo's Behavior Space.

11.3. Database connection

Since there are packages available to connect all common database management systems (e.g. RMySQL, RPostgreSQL, ROracle, RJDBC, RSQLite or RODBC), the RNetLogo package opens the possibility to store the simulation results into a database.

In the following example we will use the RSQLite package, which provices a connection to SQLite database, because this is a very easy-to-use database in a single file.

In a first step we have to setup the connections to NetLogo (if not already done) and to our test database.

```
> # load RNetLogo package (if not already done)
> library(RNetLogo)
> # path to NetLogo installation
> nl.path <- "C:/Program Files/NetLogo 5.0.4"
> # initialize NetLogo (if not already done)
> NLStart(nl.path, gui=FALSE)

> library(RSQLite)
> # load database driver
> m <- dbDriver("SQLite")
> # define path and filename for the test database
> database.path = "C:/test_netlogo.db"

> # create connection to the database
> # (if the database does not exist,
> #  this will create a file test_netlogo.db)
> con <- dbConnect(m, dbname = database.path)
> # load a NetLogo Model
> model.path <- "models/Sample Models/Earth Science/Fire.nlogo"
> NLLoadModel(paste(nl.path,model.path,sep="/"))
> # setup the model
> NLCommand("setup")
> # run the model for 10 time steps and save the results
> # (ticks and burned-trees) in table "Burned1" of database
> dbWriteTable(con, "Burned1",
+              NLDoReport(10,"go",c("ticks","burned-trees"),
+              as.data.frame=T,df.col.names=c("time","burned")),
+              row.names=F, append=F)

[1] TRUE

> # the first query: how many lines has the new table?
> dbGetQuery(con, "select count(*) from Burned1")[[1]]

[1] 10

> # the second query: select all row from table Burned10
> # where time is more than 5
> rs <- dbSendQuery(con, "select * from Burned1 where time > 5")
> # get the result of the query
> data <- fetch(rs, n = -1)
> # show the result
> print(data)
```

```
  time burned
1    6    459
2    7    513
3    8    564
4    9    613
5   10    657

> # clear query
> dbClearResult(rs)

[1] TRUE

> # append further results to existing table
> dbWriteTable(con, "Burned1",
+              NLDoReport(10,"go",c("ticks","burned-trees"),
+              as.data.frame=T,df.col.names=c("time","burned")),
+              row.names=F, append=T)

[1] TRUE

> # show the content of our table
> dbGetQuery(con, "select * from Burned1")

   time burned
1     1    127
2     2    205
3     3    278
4     4    345
5     5    407
6     6    459
7     7    513
8     8    564
9     9    613
10   10    657
11   11    705
12   12    746
13   13    782
14   14    826
15   15    872
16   16    916
17   17    971
18   18   1027
19   19   1068
20   20   1109

> # create a second table and save the result of 10 repeated simulation
> # of 20 simulation steps each
```

```
> for (x in 1:10)
+ {
+   NLCommand("setup")
+   dbWriteTable(con, "Burned2",
+                NLDoReport(20,"go",c("ticks","burned-trees"),
+                as.data.frame=T,df.col.names=c("time","burned")),
+                row.names=F, append=T)
+ }
> # calculate the mean of burned trees (out of the 10 repetitions)
> # for each time
> rs <- dbSendQuery(con, "select avg(burned) as mean_burned
+                         from Burned2 group by time")
> # get the result of the query
> data <- fetch(rs, n = -1)
> # show the result
> print(data)

   mean_burned
1        140.5
2        220.4
3        296.1
4        364.3
5        425.0
6        483.7
7        538.3
8        591.0
9        638.7
10       687.3
11       735.0
12       780.2
13       824.9
14       868.6
15       915.5
16       960.4
17      1003.3
18      1046.8
19      1092.3
20      1136.7

> # clear query
> dbClearResult(rs)

[1] TRUE

> # clean up
> dbDisconnect(con)
```

```
[1] TRUE
```

Note that there are also attempts to connect databases directly from NetLogo via an extension (see http://code.google.com/p/netlogo-sql/).

11.4. Advanced plotting functionalities

R and the additional packages deliver a wide variety of plotting capabilites. As an example, we will present three-dimensional plot in combination with contour map. We use the Urban Site - Sprawl Effect model (Felsen and Wilensky 2007) from NetLogo's Model Library. This model simulates the growth of cities and urban sprawl. Seekers (agents) looking for patches with high attractiveness and also increase the attractiveness the patch they are staying. Therefore, a state variable of the model is the attractiveness of the patches, which can be plotted in R. First, we initialize the simulation, run it for 150 time steps and collect the patch state after this time.

```
> # load RNetLogo package (if not already done)
> library(RNetLogo)
> # path to NetLogo installation
> nl.path <- "C:/Program Files/NetLogo 5.0.4"
> # initialize NetLogo (if not already done)
> NLStart(nl.path, gui=FALSE)

> model.path <- "models/Curricular Models/Urban Suite"
> model.name <- "Urban Suite - Sprawl Effect.nlogo"
> NLLoadModel(paste(nl.path,model.path,model.name,sep="/"))

> NLCommand("resize-world -20 20 -20 20 ")
> NLCommand("set smoothness 10",
+           "set max-attraction 5",
+           "set population 500",
+           "set seeker-search-angle 200",
+           "set seeker-patience 15",
+           "set wait-between-seeking 5")
> NLCommand("setup")
> NLDoCommand(150,"go")
> attraction <- NLGetPatches("attraction",as.matrix=T)
> pxcor <- NLReport(c("min-pxcor","max-pxcor"))
> pycor <- NLReport(c("min-pycor","max-pycor"))
```

Now, we define the advanced plotting function with a three-dimensional plot and a contour map. At the end, we execute this function for the collected data. The resulting plot is shown in figure 13.

```
> # adapted from
> # \url{http://addictedtor.free.fr/graphiques/RGraphGallery.php?graph=1}
> # author: Romain Francois
```

```
> kde2dplot <- function(d,
+                       ncol=50,
+                       zlim=c(0,max(z)),
+                       nlevels=20,
+                       theta=30,
+                             phi=30)
+ {
+   z   <- d$z
+   nrz <- nrow(z)
+   ncz <- ncol(z)
+   couleurs  <- tail(topo.colors(trunc(1.4 * ncol)),ncol)
+   fcol      <- couleurs[trunc(z/zlim[2]*(ncol-1))+1]
+   dim(fcol) <- c(nrz,ncz)
+   fcol      <- fcol[-nrz,-ncz]
+
+   par(mfrow=c(1,2),mar=c(0.5,0.5,0.5,0.5))
+   persp(d,col=fcol,zlim=zlim,theta=theta,phi=phi,
+         zlab="attraction",xlab="x",ylab="y")
+
+   par(mar=c(2,2,2,2))
+   image(d,col=couleurs)
+   contour(d,add=T,nlevels=nlevels)
+   box()
+ }
> # merge the data
> d <- list(x=seq(pxcor[[1]],pxcor[[2]]),
+      y=seq(pycor[[1]],pycor[[2]]),
+      z=attraction)
> # execute the plot function
> kde2dplot(d)
```

11.5. Time sliding visualization

As agent-based models are often very complex, more than the three dimensions (x-cor.,y-cor.,color) could be relevant for the analysis. With the RNetLogo package it is possible to save the output of a simulation in R for every time step and then click through, or animate, the time series of these outputs, for example a combination of the model's View and distributions of state variables. As a prototype we wrote a function to implement a timeslider to plot turtles. This function can be extended to visualize a panel of multiple plots per time step. With a slider we can browse through the simulation steps. To give an example, we used the Virus model (Wilensky 1998) from NetLogo's Model Library to visualize the spatial distribution of infected and immune agents as well as boxplots of the time periode of infection and the age in one plot panel.

We first load the required packages (rpanel (Bowman, Crawford, Alexander, and Bowman 2007)) and define a helper function to set the plot colors for the logical variables (sick, immune) of the turtles.

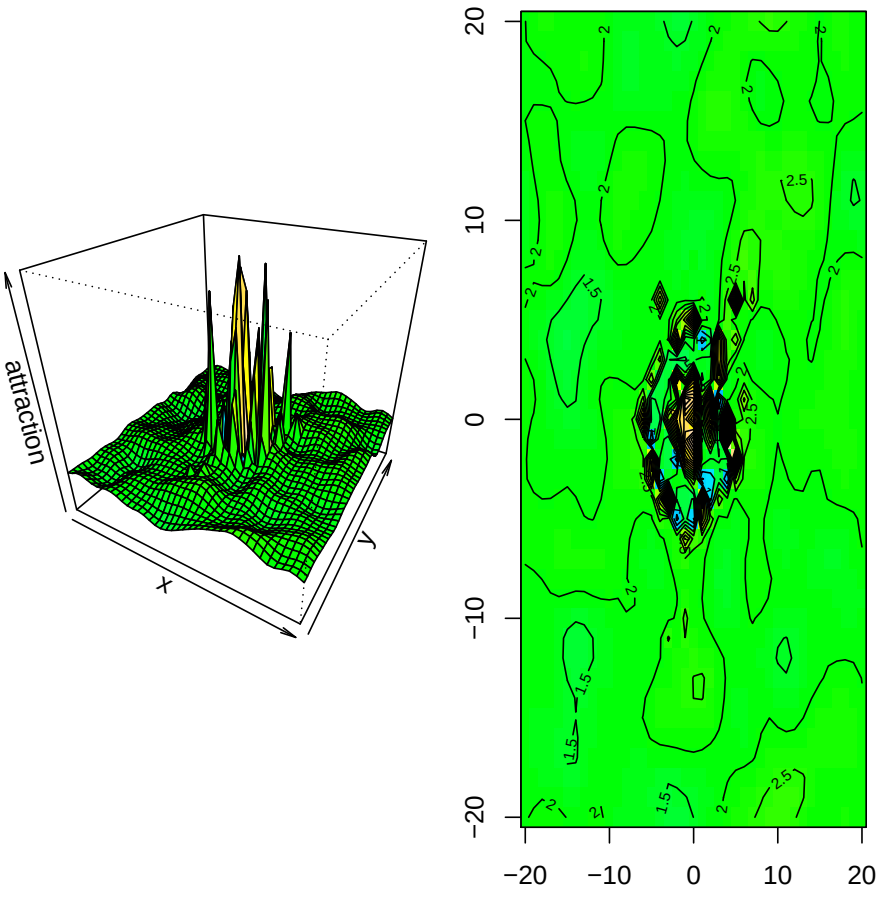

Figure 13: Spatial distribution of attractiveness of patches after 150 simulation steps. 3D plot (left) and contour plot (right).

```
> # load the rpanel pacakge
> library(rpanel)
> # define a function to set a logical variable to colors
> color.func <- function(color.var,colors,timedata) {
+   color <- NULL
+   if (!is.null(color.var)) {
+     index.color <- which(names(timedata)==color.var)
+     color <- timedata[[index.color]]
+     color[color==F] <- colors[1]
+     color[color==T] <- colors[2]
+   }
+   return(color)
+ }
```

Next, we define the main function containing the slider and what to do, if we move the slider. The input is a list containing data.frames for every time step. When the slider is moved, we send the current position of slider (i.e. the requested time step) to the plotting function, extract the corresponding data.frame from the timedata list and draw a panel of four plots using this data.frame.

```
> # define a custom plot function using rp.slider of the rpanel package
> # to browse through the plots
> plottimedata <- function(timedata.list,x.var,y.var,boxplot.var1,
+                          boxplot.var2,color.var1=NULL,colors1="black",
+                          color.var2=NULL,colors2="black",
+                          mains = NULL, ...)
+ {
+
+    # the drawing function, called when the slider position is changed
+    timeslider.draw <- function(panel) {
+      index.x <- which(names(timedata.list[[panel$t]])==x.var)
+      index.y <- which(names(timedata.list[[panel$t]])==y.var)
+      index.b1 <- which(names(timedata.list[[panel$t]])==boxplot.var1)
+      index.b2 <- which(names(timedata.list[[panel$t]])==boxplot.var2)
+
+      # if a color variable (logical) is given set the colors
+      # using function defined above
+      color1 <- color.func(color.var1,colors1,timedata.list[[panel$t]])
+      color2 <- color.func(color.var2,colors2,timedata.list[[panel$t]])
+
+      # 4 figures arranged in 2 rows and 2 columns with one title text line
+      par(mfrow=c(2,2),oma = c( 0, 0, 1, 0 ))
+      # create current plot
+      plot(timedata.list[[panel$t]][[index.x]],
+           timedata.list[[panel$t]][[index.y]],
+           col=color1, main=mains[1], ...)
+      plot(timedata.list[[panel$t]][[index.x]],
```

```
+           timedata.list[[panel$t]][[index.y]],
+           col=color2, main=mains[2], ...)
+      boxplot(timedata.list[[panel$t]][[index.b1]], main=mains[3])
+      boxplot(timedata.list[[panel$t]][[index.b2]], main=mains[4])
+      title( paste("at time ",panel$t), outer = TRUE )
+      panel
+    }
+
+    # create a control panel (hosting the slider)
+    panel <- rp.control()
+
+    # create a slider to switch the plot data
+    rp.slider(panel, resolution=1, var=t, from=1, to=length(timedata.list),
+              title="Time", showvalue=TRUE, action = timeslider.draw)
+ }
```

In the third step, we initialize and run the NetLogo simulation and collect the results into the timedate list. As mentioned above, we push a data.frame containing the results of one simulation step into the timedata list. Here, we run 100 simulation steps and use the `NLGetAgentSet` to collect data from the turtles.

```
> library(RNetLogo)
> # initialize NetLogo
> nl.path <- "C:/Program Files/NetLogo 5.0.4"
> model.path <- "/models/Sample Models/Biology/Virus.nlogo"
> NLStart(nl.path)
> # load the Tumor model
> NLLoadModel(paste(nl.path,model.path,sep=""))
> # initialize the model
> NLCommand("setup")
> # run the model for 100 time steps and save the turtles of
> # every step in one entry of the timedata list
> nruns <- 100
> timedata <- list()
> for(i in 1:nruns) {
+   NLCommand("go")
+   timedata[[i]] <- NLGetAgentSet(c("who","xcor","ycor","age",
+                                    "sick?","immune?","sick-count"),
+                                  "turtles")
+ }
```

In the last step, we collect the dimension of the NetLogo World to use it for the axis extend of the plot and define the colors to use for the variables sick (green=FALSE, red=TRUE) and immune (red=FALSE, green=TRUE). At least, we call the above defined `plottimedata` function to create the timeslider. Then, we can move the slider and the plot is updated immediately as shown in figure 14.

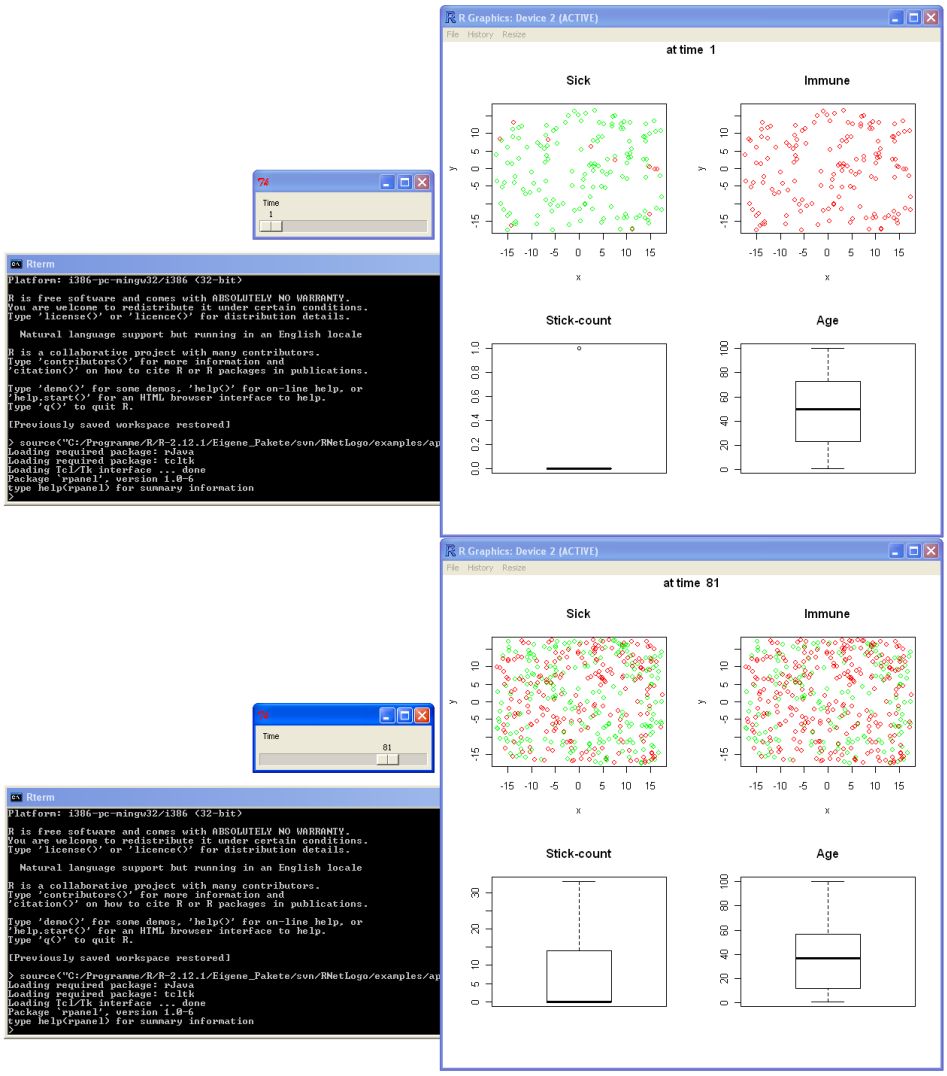

Figure 14: Timeslider example using the Virus model.

```
> # get the world dimension to use for the plot
> world.dim <- NLReport(c("(list min-pxcor max-pxcor)",
+                         "(list min-pycor max-pycor)"))
> # define colors to be used for turtle visualization
> colors1 <- c("green","red")
> colors2 <- c("red","green")
> # call the plottimedata function to brwose through the timedata list
> plottimedata(timedata.list=timedata, x.var="xcor", y.var="ycor", xlab="x",
+              ylab="y", color.var1="sick?", color.var2="immune?",
+              boxplot.var1="sick-count", boxplot.var2="age",
+              colors1=colors1, colors2=colors2,
+              mains=c("Sick","Immune","Stick-count","Age"),
+              xlim=world.dim[[1]], ylim=world.dim[[2]])
```

12. Concluding remark

Now, we think you know everything you need to know to start with your own project. There are so many interesting packages available in R with connections to many other programs - it is really amazing what this connections offers to both: R users and NetLogo users!

Note that there are code samples for all functions in the example folder (RNetLogo/examples/code samples) of the `RNetLogo` package. Furthermore, there are (just) some example applications in the example folder.

A last hint: In case you don't know, there is a program available which comes with eavery R installation, called `Sweave`, which enables you to write a combined LaTex and R document (see `http://www.stat.uni-muenchen.de/~leisch/Sweave` for further details). You can embed your R code in your LaTex text document. When compiling the Sweave document, the R code is evaluated and the results (not only numberic but also images) can be embedded automaticaly into the LaTex document. The `RNetLogo` package opens the possibility to embed not only results of R but also the result of a NetLogo simulation. I.e. you can create a self-documented report with NetLogo simulations and R analytics (with or without source code). For an example see the code of this tutorial (`tutorial.Rnw`). It is completly created with `Sweave`. Similar things can be done with OpenOffice-Writer using the odfWeave package (Kuhn, Weston, Coulter, Lenon, and Otles 2010) and with Microsoft Word using the SWord program (Baier 2009).

Feedback and comments are very welcome. Write to Jan C. Thiele (`jthiele@gwdg.de`).

Many thanks for using the RNetLogo Package.

13. Acknowledgement

I would like to thank Volker Grimm for his comments on this tutorial and his motivating words all over the time.

References

Baier T (2009). *SWordInstaller: SWord: Use R in Microsoft Word (Installer)*. R package version 1.0-2, URL `http://CRAN.R-project.org/package=SWordInstaller`.

Bakshy E, Wilensky U (2007). "Turtle histories and alternate universes: Exploratory modeling with NetLogo and Mathematica." In "Agent 2007 Conference : Complex Interaction and Social Emergence. Evanston, IL. November 15-17," .

Bowman A, Crawford E, Alexander G, Bowman R (2007). "rpanel: Simple interactive controls for R functions using the tcltk package." *Journal Of Statistical Software*, **17**(9), 1–23. URL `http://eprints.gla.ac.uk/13510/`.

Crawley M (2005). *Statistics: An Introduction using R*. Wiley. ISBN 0-470-02297-3, URL http://www.bio.ic.ac.uk/research/crawley/statistics/.

Felsen M, Wilensky U (2007). "NetLogo Urban Suite - Sprawl Effect model." URL http://ccl.northwestern.edu/netlogo/models/UrbanSuite-SprawlEffect.

Goedman R, Grothendieck G, Højsgaard S, Pinkus A (2010). *Ryacas: R interface to the yacas computer algebra system*. R package version 0.2-10, URL http://CRAN.R-project.org/package=Ryacas.

Grimm V, Railsback S (2005). *Individual-based modeling and ecology*. Princeton University Press, Princeton.

Kabacoff R (2010). *R in Action*. Manning. URL http://www.manning.com/kabacoff.

Kabacoff R (2011). "Quick-R: acccessing the power of R." Last accessed: 2011-07-14, URL http://www.statmethods.net/.

Kuhn M, Weston S, Coulter N, Lenon P, Otles Z (2010). *odfWeave: Sweave processing of Open Document Format (ODF) files*. R package version 0.7.17, URL http://CRAN.R-project.org/package=odfWeave.

Maindonald J (2008). "Using R for Data Analysis and Graphics: Introduction, Code and Commentary." URL http://cran.r-project.org/doc/contrib/usingR.pdf.

Owen W (2010). "The R Guide." URL http://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf.

Pebesma E, Bivand R (2005). "Classes and methods for spatial data in R." *R News*, **5 (2)**. URL http://cran.r-project.org/doc/Rnews/.

Pebesma EJ (2004). "Multivariable geostatistics in S: the gstat package." URL http://www.gstat.org/.

Pinkus A; Winitzki S, Niesen J (2007). "YACAS Computer Algebra System." URL http://yacas.sourceforge.net/homepage.html.

Railsback S, Grimm V (2012). *Agent-based and Individual-based Modeling: A Practical Introduction*. Princeton University Press.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Urbanek S (2010). "rJava: Low-level R to Java interface." R package version 0.8-8, URL http://CRAN.R-project.org/package=rJava.

Venables WN; Smith D, Team RC (2011). "An Introduction to R." URL http://cran.r-project.org/doc/manuals/R-intro.pdf.

Venables W, Ripley B (2002). *Modern Applied Statistics with S. Fourth Edition*. Springer, New York. ISBN 0-387-95457-0, URL http://www.stats.ox.ac.uk/pub/MASS4/.

Wilensky U (1997a). "NetLogo Fire model." URL http://ccl.northwestern.edu/netlogo/models/Fire.

Wilensky U (1997b). "NetLogo GasLab Free Gas model." URL http://ccl.northwestern.edu/netlogo/models/GasLabFreeGas.

Wilensky U (1998). "NetLogo Virus model." URL http://ccl.northwestern.edu/netlogo/models/Virus.

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

Wilensky U, Rand W (in press). *An introduction to agent-based modeling: Modeling natural, social and engineered complex systems with NetLogo.* MIT Press.

Zuur A, Ieno E, Meesters E (2009). *A Beginner's Guide to R.* Use R. Springer. ISBN: 978-0-387-93836-3.

Affiliation:

Jan C. Thiele
Department of Ecoinformatics, Biometrics and Forest Growth
University of Göttingen
Büsgenweg 4
37077 Göttingen, Germany
E-mail: jthiele@gwdg.de
URL: http://www.uni-goettingen.de/en/72779.html