Getting Started with R

Charles D. Canham
Institute of Ecosystem Studies
Millbrook, NY USA

1. INTRODUCTION

R is a powerful programming language/environment that is particularly suited to statistical analysis. It is an implementation of the S programming language, is free for download, and has a large and growing community of users who develop new features (in "packages") that you can download and use. The downside of this free-wheeling and open architecture is that it can be a bit intimidating to get started using R. There are a number of good books on using R, but there is plenty of free documentation and help online through the R site (http://www.r-project.org/) or by searching on the web. We have archived a number of tutorials and help documents on the course website and the course CD in the "R Tutorials" directory.

The most useful of the "official" manuals from the R site are (links below are to files stored in the same directory as this "Getting Started With R" tutorial:

- An Introduction to R (90 pages and worth printing out)
- R Data Import/Export
- R Installation and Administration

You can also find a number of good tutorials online (usually prepared by faculty for their courses). The course CD includes tutorials from

- Steve Ellner (Cornell): an excellent source, with a focus on using R for dynamical modeling
- Dan Stoebel (Stony Brook): a shorter and more basic introduction, [in 4 parts: Part 1, Part 2, Part 3, and Part 4] but with good examples to get you going

There is also a very useful 4-page "Reference Card" produced by Tom Short that gives you a compact list of basic commands and syntax.

The course CD also contains PDFs of several other tutorials on more advanced, statistical topics.

1.1 Basic Installation

Installation of the Windows version is quick and easy. Go to the R site (http://www.r-project.org/) and find the downloadable Windows version and install it. There is also an executable version of the download file on the course CD in the "Downloads" directory.

1.2 Adding Packages

Many of the features in R are available through additional "packages" created by users. The code for both core R and packages is maintained by the Comprehensive R Archive Network (CRAN) (http://cran.us.r-project.org/). There is a very long list of available packages stored at this site. The best way to install these packages is while online and running R (using the methods described in the section on "Package Management" in this tutorial). But first you will need to know the name of the package. In general, you will come across the names of packages as you learn more about R. You can then install them on your own computer for your own use.

We will be using a new package written by Lora Murphy expressly for this course. The package is designed specifically for likelihood calculations of neighborhood models using simulated annealing,

but its functions can also be used in conjunction with other R commands and packages for more general likelihood analyses. The package is called "**neighparam**" (for "parameterization of neighborhood models"). You will be given the package as a zip file on the course CD, and can install it on your own computer.

1.3 Final Note Before You Start

This tutorial covers an eclectic mix of topics that will help you get going. Beyond that you'll find that you need to consult lots of different sources (including the extensive but often cryptic online help files) and make extensive use of examples you find along the way.

2. SYNTAX

This tutorial uses several formatting features:

1. R commands will be on lines beginning with ">", and will be in a `Courier font`
2. Italics within an R command are used to identify terms that are made up by the user (i.e. not a reserved word or formal R term)

R has many syntax rules. Here are two of the most basic and useful:

1. **R is case-sensitive**: upper and lower cases of letters are considered different. Thus "mydata" and "Mydata" will be treated as 2 different things.
2. You can (and should) embed comments in your R scripts by using the **#** character. Anything after a **#** on a line will be ignored by R.

3. GETTING STARTED

As a general rule, you should use a different directory for each project you are working on. R lets you save the status of a session to use at a later time, but this is easiest if you save the session to a directory associated with that project. You can also save all of the data files and scripts you need in that directory.

There are several things that you should remember to do each time you start R:

1. **Set the directory**: **F**ile -> **C**hange dir… this way you won't have to keep specifying annoyingly long Windows path names when you read and write files or load scripts.

Load any packages you will need: **P**ackages -> **L**oad package…

4. GETTING HELP

The main Help menu for R offers a suite of help options: give them a try…

4.1 Viewing Local Help Files in HTML Format

When R was installed, HTML format help files were copied onto your hard drive. There are 3 ways to access these files.

1. The easiest way is to use the R menu choice **H**elp ->**H**tml help…

2. You can also access this same page by issuing the following command from within R:

```
> help.start()
```

this will open up Explorer to exactly the same page described above.

3. You can also set up a bookmark in Explorer. In Explorer, select **F**ile -> **O**pen, then browse to the help files located in the **\R\rw2011\library\stats\html** directory and select the "**rwin.html**" page. Normally the \R directory will be installed in C:\Program Files\, but you may have installed it somewhere else…

Note: help files for the **neighparam** package will be found in the **Packages** section.

Note: help files for the basic statistical functions are located in the **stats** package in the **Packages** section

4.2 Viewing Help within an R Text Window

The standard syntax for requesting help in a window within R is:

```
> help(name)
```

where *name* is an R function or feature (not typed in italics)

Alternatively,

```
> ?(name)
```

does the same thing

4.3 Searching for Help

The HTML Help home page has a link for a "Search Engine and Keywords"

Within R, you can use

```
> help.search("term")
```

to display a list of the help files that contain the string *term*. The display will list both the functions that contain the term, and the package within which the function occurs (in parentheses). If it is not a standard package that you have loaded, you can still get help by typing

```
> help(name, package = package)
```

where name is the name of the function in the specified package…

For more details, try

```
> help(help.search)
```

or

```
> ? help.search
```

4.4 Displaying Help in Windows Format

You can edit the Rprofile file in the \R\rw2011\etc directory and remove the comment character (#) from the line for

```
options(chmhelp=TRUE)
```

Now, when you type "help(*command*)" on the console command line, you will get a Windows-style help file that is easier to use than the simple text windows that will normally open. It will generally open a help file that will allow you to look at all commands that are in the same package as *command*.

5. WORKING WITH SCRIPTS

You can type individual commands directly onto the prompt line in the R console, but a better way to work is to open a "script" file within R (or a simple editor like WordPad) and type your commands there and then copy them to the console. You will make lots of mistakes as you develop a set of commands (a script), and it is easier to edit them in a separate window. You can then store a script in an external file for reference or future use.

If you are doing a lot of work with R, you will probably want to work in a professional text-editing program like ConText (it's free as a download at http://www.context.cx/). You can also download "code templates" for R that will highlight R syntax in ways that make your scripts more readable.

There are two options for loading and running a set of R commands stored in an external file:

1. Use the R menu option to open a **S**cript file. This will open in a wordpad window where you can edit commands, and then use the "**E**dit -> **R**un line or selection" or "**E**dit -> **R**un all" command to submit lines from the script to the console.
2. You can load and run them directly from a stored file (named mycommands.R in this example) using the console command:

    - `source("mycommands.R")`

Note that in this example, the command assumes that you have already set the directory so that it can find this file.

6. DATA INPUT AND EXPORT

6.1 Reading Text Files with Read.Table

The function `read.table` is the most convenient way to read in a rectangular grid of data. Use

`> help(read.table)` to see the syntax and arguments for the function.

The most basic syntax is

`> mydata <- read.table("myfile.dat")` this reads *myfile.dat* and assigns it to the R object named *mydata*

If you set the project directory when you started the session, you don't need to specify a path to the file (assuming the file is in the project directory…).

6.2 Options for Read.Table

Header line : a first line of text will be read as column headings (typically variable names). You can force this by specifying: "**header = TRUE**" as an argument in **read.table**

```
> read.table("myfile.dat", header = TRUE)
```

Separator: Tab-delimited text files are recommended [**sep = "\t"**]. This allows embedded spaces in character strings. `Read.table` will typically determine the field separator to be used, but with white-space separated files there may be a choice between the default `sep = ""` which uses any white space (spaces, tabs or newlines) as a separator, `sep = " "` and `sep = "\t"`. Note that the choice of separator affects the input of quoted strings. If you have a tab-delimited file containing empty fields be sure to use `sep = "\t"`.

```
> read.table("myfile.dat", header = TRUE, sep = "\t")
```

Reading character variables: Unless you take any special action, **read.table** tries to select a suitable class for each variable in the data frame. It tries in turn `logical`, `integer`, `numeric` and `complex`. If all of these fail, the variable is converted to a factor (assumed to be a code for a treatment level). If your character variables represent codes that identify groups, let **read.table** convert them to "factors", i.e. an integer code representing each different code. This facilitates using them as coding variables in analyses. If for some reason you don't want them treated as factors, the argument **as.is** allows you to suppress conversion of character variables to factors and keep them as strings of characters.

```
> read.table("myfile.dat", header = TRUE, sep = "\t", as.is =
TRUE)
```

Missing values By default the file is assumed to contain the character string `NA` to represent missing values, but this can be changed by the argument `na.strings`, which is a vector of one or more character representations of missing values. Empty fields in numeric columns are also regarded as missing values. In numeric columns, the values `NaN`, `Inf` and `-Inf` are accepted.

White space in character fields: If a separator is specified, leading and trailing white space in character fields is regarded as part of the field. To strip the space, use argument **strip.white = TRUE.**

Blank lines: By default, **read.table** ignores empty lines. This can be changed by setting **blank.lines.skip = FALSE**, which will only be useful in conjunction with **fill = TRUE**, perhaps to use blank rows to indicate missing cases in a regular layout.

Comments: By default, **read.table** uses # as a comment character, and if this is encountered (except in quoted strings) the rest of the line is ignored. Lines containing only white space and a comment are treated as blank lines. If it is known that there will be no comments in the data file, it is safer (and may be faster) to use **comment.char = ""**.

```
> read.table("myfile.dat", header = TRUE, as.is = TRUE,
comment.char = "")
```

6.3 Using the Clipboard to Copy Data In and Out of R

You can also use `read.table` to read data from the clipboard. For example, in Excel, select the data you want to transfer to R and copy it to the clipboard (Excel **E**dit -> **C**opy). Then read it into R from the clipboard using

```
> mydataframe <- read.table(file = "clipboard")
```

If the first rows contain column headers that you want to use as variable names, you have to specify this:

```
> mydataframe <- read.table(file = "clipboard", header = TRUE)
```

Remember that if you want to transfer character variables as unique strings (i.e. not treat them as factor codes), you should include the `as.is = TRUE` argument:

```
> mydataframe <- read.table(file = "clipboard", header = TRUE,
as.is = TRUE)
```

This is also an easy way to transfer the contents of a data frame in R to another file (via the clipboard). To write the contents of a dataframe to the clipboard, use

```
> write.table(mydataframe, file = "clipboard", sep = "\t",
row.names = FALSE)
```

In this example, the column separator is a tab (making it easy to paste into columns in Excel. You also need the final argument (`row.names = FALSE`) so that the column names line up with the columns correctly (yes, I mean columns…). After executing this command in R, switch to the other application (Excel, etc.) and paste the data from the clipboard…

6.4 Saving and Loading R Datafiles

You can save a data frame as an R datafile (*.Rdata is the default filetype) and reload it for a later session. The commands are simply `save()` and `load()`.

```
> save(mydataframe, file = "myRdata.Rdata")
```
note that "file =" is required…

```
> load("myRdata.Rdata")
```

6.5 Reading Worksheets from Within Excel Files

The package RODBC will let you read individual worksheets (or portions of worksheets) from within Excel files, without requiring that you export the contents of separate worksheets to individual text files. You will need to install the package (**P**ackages -> **I**nstall Packages…) from either the web or a local zip file, and then you will need to load the package (**P**ackages -> **L**oad packages…) before you use it.

NOTE: RODBC will allow you to retrieve tables from many database programs. The commands and examples below are specific to Excel.

The first step is to create a "channel" and "connect" it to the Excel file:

```
> myexcelchannel <- odbcConnectExcel("myexcelfile.xls")
```

If you want to be able to enter the filename using a Windows dialog, use this:

```
> myexcelchannel <- odbcConnectExcel()
```

Individual worksheets represent individual "tables" accessible through this channel. You can see a list of the worksheets (tables) available through the channel using the command:

```
> sqlTables(myexcelchannel)
```

You then "fetch" the data from a table (worksheet) in the channel:

```
> myexceldata <- sqlFetch(myexcelchannel,"worksheetname")
```

"*worksheetname*" is what RODBC calls a `table_name`, and *myexceldata* is now a data frame usable in R.

NOTE: This works as long as the worksheet name (in Excel) has NO SPACES! If there are spaces in the worksheet name (on the tab at the bottom of the worksheet in Excel), you need to use a weird naming convention:

```
> myexceldata <- sqlFetch(myexcelchannel,"'worksheet name$'")
```

NOTE: the worksheet name is inside double quotes and then inside single quotes too, and the text has a $ at the end (don't ask why this is needed – I don't have a clue, but it works…)

You can add additional arguments to the call to `sqlFetch`. These include:

`as.is = TRUE`	to read character variables properly
`max: n`	to limit the fetch to the first *n* rows

If you are conversant in SQL, the documentation for RODBC claims that you can add SQL queries to the `sqlFetch` command to control what data are retrieved from the database table, but I have not tried this (not being conversant in SQL). See documentation for `sqlQuery` and `sqlGetResults`, or insert the query directly as an argument in the call to `sqlFetch`.

7. SUBSETTING AND SELECTING DATA

7.1 Subsetting Observations and Selecting Variables

You can use the `subset` command to either "subset" the dataset (i.e. specify a subset of observations to use) or to "select" a set of variables (columns in the data frame). The subset command has other uses – see `help(subset)` for details. The basic syntax is:

```
> subset(x, subset, select,…)
```
where *x* is an object (typically a data frame)

`> newdata <- subset(mydata, varname == "varcode")` creates subset of ***mydata*** for which ***varname*** equals ***varcode***. NOTE the use of 2 equals sign to indicate that this is a Boolean ("is equal to…")

`> newdata <- subset(mydata, varname > 50)` creates subset of ***mydata*** for which ***varname*** is greater than 50

`> newdata <- subset(mydata, select = varname)` creates a subset containing only the variable ***varname***

7.2 Dropping variables

You can use `subset` to drop variables from the dataset by putting a minus sign in front of the selected variable name:

```
> newdata <- subset(mydata, select = - varname)
```

7.3 Working "with" data

You can use the `with()` command to evaluate any expression (or run any set of R commands) with a data frame. You can also use `subset` within the `with()` command to evaluate any expression (or run any set of R commands) with just a subset of the data. For example:

```
> with(subset(mydata, species == "ACSA"), {   #begin list of
commands…

          })
```

This will execute the set of commands inside the brackets, using the subset of ***mydata*** for which the value of the ***species*** variable is "ACSA". If you only have 1 command to run, you don't need the brackets.

7.4 Dealing with missing values

You can create a logical vector that can be used to identify cases (observations) in the dataset that have no missing values for any variable using the `complete.cases` command:

```
> is.complete <- complete.cases(mydata)
```

The vector `is.complete` has the same number of elements as the number of observations in mydata, but each element is a Boolean (TRUE or FALSE) specifying whether the observation is complete.

You can easily create a working dataset consisting of only the complete observations, using the `na.omit` command:

```
> newdata <- na.omit(mydata)
```

If you want to create a working dataset with complete observations for just a subset of the variables, use na.omit with the data.frame command to group the selected variables into the new working dataset:

```
> newdata <- na.omit(data.frame(mydata$var1,mydata$var5))
```

8. MANAGING OBJECTS

8.1 Keeping Track Of and Removing Objects During a Session

You will typically create lots of "objects" during a session (data frames, variables, lists, etc.). To see the list of current objects, use the "list" (ls) command

```
>ls()
```

To remove an object, use the "remove" (rm) command:

`> rm(myobject)` where ***myobject*** is the name of one of the objects in your workspace

To remove a list of objects, the syntax is:

- `rm(list = "myobject1","myobject2")`

Note that in this case, the names of the objects have to be embedded inside double quotes.

To remove everything in the current workspace, use:

```
> rm(list = ls())
```

9. MANAGING GRAPHIC WINDOWS

R displays plots in a graphics window. By default, each call to a graphics command that produces a new plot will over-write the plot in the existing window. NOTE: there are many commands that simply add features to an existing plot, such as the commands to add points (`points`), lines (`lines`), or text to a plot. If you want to open a new window for the next plot (leaving the existing graphics windows intact) use the following command before you issue the next plot command:

```
> windows()
```

The next plot will create a separate window, without changing the existing window.

There is a much more elegant solution to this problem, however. When a graphics window is opened and has the focus (i.e. you've clicked on its title bar), you'll see that the menu choices for the main R GUI will change to display options for managing graphics windows. The most important choices are under the **H**istory menu option. If you click on the option for "**R**ecording", new graphs will be placed in the existing graphics window, but the previous ones will be saved as previous "pages". You can scroll back and forth through the set of graphs using the "page up" and "page down" keys (or the corresponding menu choices under **H**istory). You can save the set of graphs to a variable name using the "**S**ave to variable" options (which can then be saved with your workspace). You can then reload the graphs later using the "**G**et from variable" menu option.

In order to turn on recording before issuing your first plot command, create a blank window using windows(), and then click on its title bar to display the menu options, and click on the **R**ecording option under **H**istory. Alternatively, after the graph is displayed, go to **H**istory, select **A**dd to add the plot to the list of recorded graphs, and then check the **R**ecording option to record all subsequent graphs.

NOTE: Don't turn on **R**ecording when doing a simulated annealing with `neighanneal`. That function creates a graphics window to display the progress of the annealing, but essentially creates a new graph every time the window updates. This happens every time a new maximum likelihood is found, so it can create hundreds of windows. If **R**ecord is on, you save each one of them..

If you don't want to automatically "record", then just **A**dd or **R**eplace existing pages after they are created, using those menu choices. This is probably the best option in general, since you will typically modify graphs iteratively until you get them just the way you want them…

10. PACKAGE MANAGEMENT

There is a difference between installing a package and loading a package. Installing a package is like installing any other piece of software: once it is installed it is part of the R system on your machine. You only have to install it once. Loading a package means you have launched it within R. **You have to do this every time you start a new R session.**

10.1 Finding Out What Packages You Have Installed

```
> library()
```

These are the packages that you can then load using **P**ackages -> **L**oad package…

10.2 Package Installation

If you want to install a package available on CRAN, you can use Packages -> Install Package(s)…

For packages such as **likelihood** where you have the install .zip file on your hard drive (this method also works with package installers you download from CRAN), do the following:

```
> install.packages(pkg="c:\\likelihood_0.1.zip" , repos=NULL)
```

(You should type this instead of copying and pasting from Word so that the quotation marks are correct.) Make sure you double the backslashes when typing your path, and make sure the path name is correct. If the package (ZIP) file is located in the current working directory, you don't need to enter the path in the command.

10.3 Uninstalling Packages

```
> remove.packages(pkgs="likelihood")
```