YaRI - Yet another R Introduction
A self-help guide

Written by Andreas Handel (ahandel@uga.edu) with the help of other sources
See the acknowledgment section for more details

Last updated on August 6, 2012
Permanently under construction

Contents

1 Preface

1.1 Acknowledgments

A lot of this document is based on "An introduction to R for ecological modeling" by Benjamin Bolker, which in turn is based on "An introduction to R for dynamic models in biology" by Stephen Ellner and John Guckenheimer. Ben Bolker's notes are a supplement to his textbook *Ecological models and data in* R. Stephen Ellner and John Guckenheimer wrote their notes as a supplement to their textbook *Dynamic Models in Biology*. You can find the original notes, links to these very good textbooks, and additional information and resources on the respective webpages for Ben Bolker (`http://www.math.mcmaster.ca/~bolker/`) and Steve Ellner (`http://www.eeb.cornell.edu/Ellner/`). Thanks to those authors for generously providing the source code for their notes on their webpages. In turn, this document, with its source code, lives at `http://ahandel.myweb.uga.edu/resources.htm`. Feel free to use it to create your own remix.

If you use this tutorial and find any mistakes, confusing parts, suggestions for improvements, etc., let me know. I very much appreciate feedback. Email me at ahandel@uga.edu.

1.2 Purpose of this guide

The goal of this document is to give you a very rapid start with R. This guide is very short, shorter than most other similar tutorials, including the two documents on which it is based. The hope is that you can work through it reasonably quickly. If you have no previous R experience, you should work through the tutorial from beginning to end – even if you have programmed before. The focus is on getting you started, showing you how to take the first steps with R and providing you with the tools to embark on a self-learning journey – that's why I call it a self-help guide. As you continue to use and learn R, you will need to consult other sources; this document is only the start. Also, to keep it short, I often only show one way of doing things. A nice feature of R is that it is highly flexible and there are many ways of doing certain tasks. You will soon develop your own style of programming, and you might start to do things differently than explained here – which is absolutely fine.

1.3 A note for non-Windows users

This document is written with Windows users in mind. Sometimes, differences for OS X are mentioned – but this is not consistent. I ignored issues specific to Linux/Unix. I assume if

you know how to use Linux, you are computer literate enough to figure out how to get things working if the Windows way is not applicable.

2 Getting started with R

2.1 What is R?

R is a "high-level" programming language that includes many built-in and user-contributed functions. It has very powerful statistical capabilities and is also great for producing high-quality graphics. Thanks to the many available add-on packages, almost anything goes. R is an open source project, available for free. R runs on many different operating systems. Originally a research project in statistical computing, it is now managed by a development team that includes a number of well-regarded statisticians, and it is widely used by statistical researchers (plus a growing number of modelers in all kinds of disciplines) as a platform for making new methods available to users. More about R, it's history, the people behind it, and lots of other useful information can be found on the R webpage at `http://www.r-project.org`.

2.2 Installing and starting R

The main source for R is the CRAN home page `http://cran.r-project.org`. You can get the source code, but most users will prefer a precompiled version. To get one of these from CRAN, click on the link for your OS. For Windows, click on `base` and then download R. It won't hurt to read some of the installation instructions and FAQs. Install R by launching the downloaded file and following the on-screen instructions. You should have administrator rights on the computer you are trying to install R to. If you don't, ask your administrator to install R for you. Alternatively, you can try to install R as user. It might work, but I'm not sure about it. If you have enough space on your computer, choose "full install" to install the complete R with all the manuals and extras. It's not strictly necessary but good to have them. Other than that, accept the default settings suggested by the installer. At the end you'll have an R icon on your desktop that can be used to launch the program. Just click on the icon on your desktop, or in the `Start` menu (You should have allowed the Setup program to make either or both of these). When you start R it opens the **console** window. The console has a few basic menus at the top; check them out on your own. The console is also where you enter commands for R to execute *interactively*, meaning that the command is executed and the result is displayed as soon as you hit the `Enter` key. We will do that in the next section.

2.3 Installing R packages

The standard distribution of R includes several *packages*, user-contributed suites of add-on functions. As you start using R more, you will likely want to install more packages. You can find all the available R packages at `http://cran.r-project.org/web/packages/`. To install a package, either type `install.packages('packagename')` at the console, or use the `Packages` menu. You need to be connected to the internet for this to work. You should also have

administrator privileges on the computer. If you don't have those, you can still try to install the package, it might work best using the `Packages` menu. Let's try this. At the command prompt, type `install.packages('deSolve')` - it should look like this:

```
> install.packages('deSolve')
```

and hit `Enter` (You always execute a statement by pressing `Enter`, I will from now on not state that every time). Make sure you spell it exactly `deSolve`, with a capital `S`, since R distinguishes between upper and lower case letters. If you misspell, R will tell you that it can't find the package. You will be asked to select a mirror, choose one close to you. This will then install the package `deSolve` which can solve differential equations for you and which we will use a lot.

Once you have installed a package, it is available to R. But before you use it, you need to load it, which is done with the commands `library('packagename')` or `require('packagename')`. We will come back to that shortly. You can see which packages are already installed by typing `library()` at the R prompt.

2.4 Stopping R code and ending an R session

If you want to quit R, you can do so from the `File` menu, by closing the R window or you can type `q()` at the command prompt. When you quit, R will ask you if you want to save the workspace (that is, all of the variables you have defined in this session); for now (and in general), say "no" in order to avoid clutter.

Should an R command seem to be stuck or take longer than you're willing to wait, click on the stop sign on the menu bar or hit the `Escape` key.

2.5 Exercises

Exercise 2.1 Go to `http://cran.r-project.org/web/packages/` and take a look at the available packages. If you click on a package, you get more information. Notice the `Depends` entry. Sometimes one package uses the contents of another package to work. Then you need to install all of the dependencies. Often, there is also link to manual in pdf form. Take a look at a few of those. Now install a package of your choice.

Exercise 2.2 Install the package `lattice`. This package provides more graphical abilities in addition to the built-in R commands. Load the package into R. Run the command `>demo(lattice)`. This will show you some of the graphs you can do with functions contained in this package. You need to click on the graphs or press `Enter` to go from one to the next.

Exercise 2.3 Close R. Now re-start it. Right-click somewhere inside the R Console window and de-select `Buffered output` (only applies to R on Windows). This will make sure that the output produced by the commands below will be printed on the screen while the program runs. If you leave `Buffered output` selected, R will only print the output once the whole program has finished – which would be a problem here... Now, type

```
> n=1; while (1<2) {cat(n,"\n"); n=n+1;}
```

What is going on here? How long will the program run? Once you saw enough, use what you learned in the previous section and move on.

3 Getting help and learning R

The following section will contain arguably the most important information, namely how to teach yourself R beyond this tutorial.

3.1 R help files

R comes with rather complete documentation and a good built-in help system. You can get to it through the `Help` entry in the menu bar. If you installed the complete R, you will have both the usual help file that explains specific functions and several tutorials/manuals in pdf form. Try it out. Alternatively, you can type at the console `help(functionname)` or `?functionname` to open the help entry for a specific function. Try by typing `help(mean)` at the command prompt. This will give you information about the function `mean`, which – you guessed it – computes the mean of several numbers. If some of the explanation seems too technical/incomprehensible, don't worry. You will understand it soon enough. Often, the examples at the end of the documentation are very useful in understanding how things work.

Some more tips on the help system:

- `help.start()` fires up a web browser pointing at all of the help files
- `help()` only searches through functions in the *currently loaded* packages
- `apropos('somestring')` looks through all accessible R objects, which means it will match names of functions containing a given string
- if you're connected to the Web, you can use the `RSiteSearch('somestring')` command (from the command line or the Help menu) to do a full-text search of all R documentation and the mailing list archives
- `example(function)` will run all of the examples in the help page, if any, for function `function`
- `demo(topic)` runs demonstration code on topic `topic`. Type `demo()` by itself to list all available demos

3.2 R manuals and tutorials

There are plenty of R resources on the web. Some of them are available through the `Help` menu. Also check out the links under `Documentation` at `http://cran.r-project.org/`. You will find small and large manuals, lots that are introductory, some that focus on certain topics (e.g. statistics or graphics). There are also more and more R-books published as a quick search on Amazon will confirm. Further tutorials and manuals exist that are not listed on the R

webpage. Do an online search for R `tutorial` or R `+ topic` and you will likely find something useful. Here's a list of free resources I found rather useful:

- An Introduction to R - the "official" beginner guide. Available online on the R webpage, and you should also have it as part of your installation. Go to `Help` → `Manuals` and it should be the first entry. If you read and work through that guide, you will have a pretty good understanding of most of the basic R functionality.
- The notes by Ben Bolker and Steve Ellner mentioned above. They contain additional information that I ommited here, it's definitely worth checking them out.
- Matlab/R Reference by David Hiebeler: `http://germain.its.maine.edu/~hiebeler/comp/matlabR.html`. Especially useful if you know Matlab well. But even if not, it's a great collection of tables, sorted by topics. The first column describes what you might want to do, and the last one shows you how to do it in R.
- Producing Simple Graphs with R by Frank McCown: `http://www.harding.edu/fmccown/R/`. Nice quick intro to basic plotting.
- R Graph Gallery: `http://addictedtor.free.fr/graphiques/`. A large collection of mostly fancy R graphs. Shows you some of the graphics capabilities of R and R packages. You can also see the source code that lead to these figures.
- R bloggers (http://www.r-bloggers.com/) collection of blog posts on all kinds of R topics
- R-seek (http://www.rseek.org) is a Google-type search engine that searches through online R materials

4 Using R

4.1 Interactive sessions and why you should avoid them

We already used R interactively when we installed the package `deSolve` above. As another example, at the command prompt, type in `2+2` and hit `Enter`; you will see

```
> 2+2
[1] 4
```

Now try something more complicated. Type `a=runif(10)`, hit `Enter`, type `mean(a)`. You should see the following (the number you get will be different):

```
> a=runif(10)
> mean(a)
[1] 0.517583
```

Here, is what you did: the first command called the function `runif(n)`, which produces n random numbers between 0 and 1. These numbers were stored (all 10 at once!) in the variable a. You can guess what the function `mean(a)` did. Another way of doing exactly the same thing is to type

```
> a=runif(10); mean(a)
```

or

```
> mean(runif(10))
```

Try it - but notice that the answers might be different, since every time you get different random numbers. If you don't like this, replace the command `runif(10)` by `seq(1,10)` and try all three versions again. Try to guess what `seq(1,10)` does, check by for instance typing `help(seq)` at the command line. You might also want to learn more about `runif` and `mean`, do so by typing `?runif` and `?mean`.

Built-in functions like `runif` or `mean` are one of the great advantages of R. Instead of writing your own function, quite often R has one that does it for you. These functions sometimes do relatively simple things like calculating the mean or more complicated tasks like robustly integrating differential equations. This saves you a lot of time because you don't have to write your own algorithms. Note that while in principle you don't need to know how certain routines work, it is useful to understand the basic concepts, otherwise you might get results that are nonsensical and you don't know it. ("The computer told me so" is not a valid excuse for wrong results).

Now that you have seen how to use the R command line to do calculations, here's the next useful thing to learn: Don't use it! This comes with qualifiers of course, it is often immensely helpful to type a few quick commands and see what happens – as you will be doing throughout the rest of this tutorial – or quickly install a new package like we did above. R remembers the results from what you typed in before (try by typing in `a`, R should return the values stored in the variable `a`). So in principle, you could type one command after the other until you did everything you wanted. However, things quickly get messy. Even for relatively simple projects, you usually want to execute a number of commands. Typing each one on the console is tedious and you might forget what you already did and what you did not. It is way more efficient if you write all of your commands into a file (a script) and then execute the script. Always work with script files, even on small projects. You will quickly learn to appreciate them. The next section tells you how this works.

4.2 Using a text editor to write and run scripts in R

To write and edit scripts, you need a text editor. The Windows and MacOS versions of R both have basic script editors but they are not too great. A good editor - and in fact a whole integrated development environment (IDE) - is R Studio. It runs on Windows, OS X and Linux and is freely available at `http://rstudio.org/`. Another good option for Windows users is Tinn-R, freely available at `http://sourceforge.net/projects/tinn-r/`.

There are other editors out there for all the different operating systems, you can find additional information at `http://www.sciviews.org/_rgui/projects/Editors.html`

A note: It's important that scripts are preserved as *plain text* files. If you try to edit a script file with a text editor like MS-Word, OpenOffice Writer or similar, you might end up saving the file in a format that R can't read. So don't use these programs as editors.

If you haven't already done so, now is a good time to install a text editor. Next, get the files `yari-example1.r` and `yari-example2.r` from `http://ahandel.myweb.uga.edu/resources.htm`, and open either one with the text editor. Now, open R. (Note: If you use R-Studio, it automatically opens Rin a window, so no need to start Rseparately). In R, change your working directory to wherever the file you just opened is located using `Change dir` in the `File` menu (On Mac it's `Misc/Change Working Directory`). Now you are ready to run say the first file by typing

```
> source('yari-example1.r')
```

A graph should appear on the screen. Note that if you used R-studio, you will have the text editor and R in one window, both open. To run the script, simply hit the `Source` button and the script will be executed.

If you go back to the text editor and look at the file, you will see the collection of commands that are executed to run a simulation and produce the graph. The R script `yari-example1.r` simulates a single outbreak of an infectious disease (e.g. influenza) in a population, the script `yari-example2.r` simulates a simple acute viral infection (e.g. influenza) within an infected host. In both cases, we run an ordinary differential equation model to simulate the dynamics and then plots the result. These are already a somewhat advanced programs, since they use functions, the `deSolve` package, and some other things that you might not fully understand right away. The many comments should nevertheless allow you to roughly follow what's going on. By the end of this tutorial, you should be able to fully understand these programs – and write similar ones yourself.

In general, you want your scripts to be somewhat structured for easier understanding. Most programs for working with models or analyzing data follow a simple pattern of program parts:

1. "Setup" statements: Clear old variables from workspace, load needed packages, etc.
2. Load data from an external file and/or define data/model parameters.
3. Carry out the calculations that you want. This is often split into a main program and several functions that are called by the main program. They can all live inside the same script, or you can save functions as their own files. The latter is an especially good idea if you have a function that you use often with different projects. If you place functions inside the main script, it is a good idea to put them all at the beginning.
4. Print or graph the results, save the graph to a file for inclusion in a publication/report, save results into a file for further processing with another program.

4.3 A comment on comments

As you can see, the two example scripts you played with in the last section include a lot of comments. These comments don't do anything, other than help you understand what is going on. It is a very good idea to always put lots of comments into your code. In my opinion, for a code that is very well documented, at least 50% of your program should be comments. As you start writing programs, you might find this to be overkill. That might be true initially. But

your programs will quickly become more complicated. And you might use specific functions for one program, and then forget how exactly that function worked. What looked obvious to you back when you wrote it might be completely puzzling to you when you revisit your code several weeks or months later. And it gets even worse if you want to share code with colleagues. So it's good practice to document your program thoroughly by adding lots of comments. As you can see from the example, comments start with the # symbol. R ignores everything on the line that follows such a symbol.

4.4 Exercises

Exercise 4.1 Copy the file `yari-example1.r` and save it under a different name. If you save it in a different folder than the original file, go to that folder. Then run your newly created file using the `source` command.

Exercise 4.2 Open your newly created file with your editor. Change some of the parameter values and re-run the program. Often, when playing around with changing values, it is a good idea to keep the old ones in the program, otherwise you might forget them. Copy the line for the parameter b and duplicate it under the original line. Then turn one of the lines into a comment using the # symbol. Now you can change the other value around and won't forget what your initial value was. Don't forget to save the script after you made changes, otherwise R won't run the new version. Watch what happens to the simulation/graph.

5 A quick overview of specific topics

The previous sections were meant to get you quickly started with R and have your first success by running and modifying a program. The following sections briefly discuss important building blocks that are part of most programs. As soon as you start working on your own problems, or even doing the exercises in this guide, you will find that you need to get additional information to learn how to do things - that's the time when you revisit Section 3.

5.1 Naming variables

To do anything complicated, the results from calculations have to be stored in (*assigned to*) variables. For example above we stored 10 data values in the variable `a` by typing

```
>a=runif(10)
```

Variable names in R must begin with a letter, followed by alphanumeric characters. You can break up long names with a period, as in `very.long.variable.number.3`, but you can't use blank spaces in variable names. R is case sensitive: `Abc` and `abc` are different variables. Make variable names long enough to remember, short enough to type. `N.per.ha` or `pop.density` are better than `x` and `y` (too short) or `available.nitrogen.per.hectare` (too long). Avoid `c`, `l`, `q`, `t`, `C`, `D`, `F`, `I`, and `T`, which are either built-in R functions or hard to tell apart.

5.2 Arithmetic operations

R does calculations with variables as if they were numbers. It uses `+`, `-`, `*`, `/`, and `^` for addition, subtraction, multiplication, division and exponentiation, respectively. For example type the following commands into the R console:

```
> x=5; y=2
> z1=x*y; z2=x^y
> z1
> z2
```

Even though R did not display the values of `x` and `y`, it "remembers" that it assigned values to them. Just type `x` or `y` to see the values stored in those variables. Typing `z1` and `z2` into the command line makes R display these values.

Now let's try doing the same with a script. Open your R editor, start a new file, and copy the commands into the file, without the `>`. You should have four lines that look like

```
x=5; y=2
z1=x*y; z2=x^y
z1
z2
```

Now save the file (give it any name and save it anywhere you like). Next, go back to R, make sure you switched to the directory in which you saved the file, and run it using the `source` command. What happened? Probably nothing. If you run a program as a script, R does not print values that it might print if you type it into the console! So if we want to see the values for `z1, z2`, we need to tell R. This can be done by replacing `z1` with `cat(z1,"\n")` and the same for `z2`. Save the file and run it again. You should now see the values for $z1$ and $z2$ displayed. The `cat` function (short for "concatenate") for printing results to the console window. `cat` converts its arguments to character strings, concatenates them, and then prints them. The `"\n"` argument is a line-feed character so that each entry is put on a separate line. Use `?cat` to learn more about this function. Since most of the examples here are very short, I assume that you will type them straight into the console. But as mentioned in Section 4.1, you should use scripts as soon as your code goes beyond a few simple lines.

You can combine several operations in one calculation:

```
> A=3
> B=(A+2*sqrt(A))/(A+5*sqrt(A)); B
```

(Recall that R cares about capitalization. The variable `A` is different from `a`). Parentheses specify the order of operations. The command

```
> B=A+2*sqrt(A)/A+5*sqrt(A)
```

is not the same as the one above; rather, it is equivalent to `> B=A + 2*(sqrt(A)/A) + 5*sqrt(A)`.

The default order of operations is: (1) parentheses; (2) exponentiation, or powers, (3) multiplication and division, (4) addition and subtraction.

`abs()`	absolute value
`cos()`, `sin()`, `tan()`	cosine, sine, tangent of angle x in radians
`exp()`	exponential function, e^x
`log()`	natural (base-e) logarithm
`log10()`	base-10 logarithm
`sqrt()`	square root

Table 1: Some of the built-in mathematical functions in R. You can get a more complete list from the Help system: `?Arithmetic` for simple, `?log` for logarithmic, `?sin` for trigonometric, and `?Special` for special functions.

x < y	less than
x > y	greater than
x <= y	less than or equal to
x >= y	greater than or equal to
x == y	equal to
x ! = y	*not* equal to

Table 2: Some comparison operators in R. Use `?Comparison` to learn more.

```
> a = 12-4/2^3     gives  12 - 4/8 = 12 - 0.5 = 11.5
> a = (12-4)/2^3   gives  8/8 = 1
> a = -1^2         gives  -(1^2) = -1
> a = (-1)^2       gives  1
```

In complicated expressions it's best to use parentheses to specify explicitly what you want. A few extra sets of parentheses never hurt and might help you avoid doing the wrong computation by accident.

R also has many built-in mathematical functions that operate on variables (Table 1 shows a few). Check how these functions work by typing `help(functionname)` or `?functionname`.

5.3 Logical operators

Often, you will want to compare variables with each other. The operators shown in Table 2 do that. They return a logical value of `TRUE` or `FALSE`. For example, try:

```
> a=1; b=3;
> x1=a<b; x1
> x2=(a>b); x2
```

The parentheses around (`a>b`) are optional but do make the code easier to read. One special case where you *do* need parentheses (or spaces) is when you make comparisons with negative values; `a<-1` will surprise you by setting `a=1`, because `<-` (representing a left-pointing arrow) is equivalent to `=` in R. Use `a< -1`, or more safely `a<(-1)`, to make this comparison.

Note that `=` and `==` are not the same! Try the following and make sure you understand what happens here

```
> a=1:3; a
> b=2:4; b
> a==b
> a=b
> a==b
```

Exclamation points `!` are used in R to mean "not"; `!=` means "not equal to" (note that there is only a single equal sign).

R also does arithmetic on logical values, treating `TRUE` as 1 and `FALSE` as 0. For instance

```
> a=1:3; b=1:3;
> c=(a==b); c
> sum(c)
```

returns the value 3 since the 3 `TRUE` entries are interpreted as ones. Note that we did something in this snippet of code that you shouldn't do. Do you know what? (Hint: Re-read Section 5.1).

One issue you should always keep in mind with comparisons is that computers make round-off errors. For instance if you try

```
> exp(-1000)==0
```

R will claim that these two expressions are equal. They are of course not, the exponential expression is slightly larger than zero. But it's so close to zero that R can't distinguish it from zero. Equally, you might sometimes get two results that you consider to be equal, but since R arrived at them through computation and rounding, they are not strictly the same. For instance try

```
> sin(pi)
> sin(pi)==0
> sin(pi/2)==1
```

The first expression, `sin(pi)`, should be equal to zero and therefore the second should return a `TRUE`, but it does not. Interestingly, the last expression does give an exact value for `sin(pi/2)` and therefore returns `TRUE` as it should. To get around these rounding issues it is more robust to write comparisons like this: `> abs(a-b)<eps` where you set `eps` to some small number for which you consider that `a==b` is satisfied for all practical purposes. For instance `abs(sin(pi)-0)<1e-10` works.

More complicated conditions are built by connecting different expressions with AND or OR. In R you write `&`, or `&&` for AND and `|` or `||` for OR. The two forms of AND and OR differ in how they handle vectors. The shorter one does element-by-element comparisons; the longer one only looks at the first element in each vector. More about vectors in the next section.

Try the following examples and make sure you understand what happens. Then try a few more on your own.

```
> a=c(1,2,1,4)
```

```
> b=c(1,1,4,5)
> (a<b) & (b>4)
> (a<b) & (b<4)
> (a<3) & (b<5)
> (a<3) && (b<5)
> (a<b) | (b>3)
> (a<b) || (b>3)
```

5.4 Exercises

Exercise 5.1 Try the following commands

```
> 12^0-1/(log(1)+3)
> (12^0-1)/(log(1)+3)
> 12^0-1/log(1)+3
> (12^0-1)/log(1)+3
```

and explain the different results. Take a peek at Section 5.9 if you don't understand some of the answers.

5.5 Vectors

R has several different data types. Vectors and matrices (1- and 2-dimensional rectangular arrays of numbers) are some of the most useful ones for scientific computing. We will discuss those next.

5.5.1 Constructing vectors

`seq(from,to,by=n)`	Vector of evenly spaced values, increment = n)
`seq(from, to, length.out=n)`	Vector of n evenly spaced values)
`c(u,v,...)`	Combine a set of numbers and/or vectors into a single vector
`rep(a,b)`	Create vector by repeating elements of `a` by amounts in `b`

Table 3: Some important R functions for creating and working with vectors. Many of these have other optional arguments; use the help system (e.g. `help(rep)`) for more information.

Some functions for creating and working with vectors are listed in Table 3. Try them out by typing the following commands

```
> seq(1,3,length=6)
> seq(1,10,by=2)
> c(1:8)
> rep(3,5)
```

Try some more commands. You can also combine commands, for instance try

```
> c(rep(2,4),seq(1,5,length=2))
> c(seq(1,5),seq(1,5,by=2))
```

Keep playing around some more to understand how these commands work. The worst thing that can happen is that R complains and doesn't do anything useful. But you can't break anything.

You may be wondering if vectors in R are row vectors (lying vectors with 1 row and many columns) or column vectors (standing vectors with one column and many rows). The answer is "both and neither". Vectors are printed out as row vectors, but if you use a vector in an operation that succeeds or fails depending on the vector's orientation, R*will assume that you want the operation to succeed and will proceed as if the vector has the necessary orientation.* For example, R will let you add a vector of length 5 to a 5×1 matrix or to a 1×5 matrix, in either case yielding a matrix of the same dimensions. The fact that R wants you to succeed is both good and bad – good when it saves you needless worry about details, bad when it masks an error that you would rather know about. We will come back to that below when we talk about matrices.

5.5.2 Addressing vectors

Often it is necessary to extract a specific entry or other part of a vector. This is done using subscripts, for example

```
> q=c(1,3,5,7,9,11); q[3]
```

`q[3]` extracts the third element in the vector `q`. Note that these are square brackets, unlike for instance in Matlab, where the brackets are round. R **uses square brackets to address vectors, matrices, etc. and round brackets to indicate functions, e.g.** `c()` **is a function creating a vector.**

You can also access a block of elements from a vector by writing for instance

```
v=q[2:5]; v
```

This extracted 2^{nd} through 5^{th} elements in the vector. If you enter `v=q[seq(1,5,2)]`, what will happen? Try it and see, and make sure you understand what happened. Extracted parts of a vector don't have to be regularly spaced. For example try

```
> v=q[c(1,2,5)]; v
```

If you want the last element of a vector, you need to first find its length. You can do so with

```
> q[length(q)]
```

A more flexible way of getting the `n` first or last elements from a vector such as `q` is with the commands `head(q,n)` and `tail(q,n)`. Check the help file to see how they work.

Addressing is also used to set specific values within a vector. For example,

```
> q[1]=12
```

changes the value of the first entry in `q` while leaving all the rest alone, and

```
> q[c(1,3,5)]=c(22,33,44)
```

changes the 1^{st}, 3^{rd}, and 5^{th} values.

We can also use *logical* vectors (lists of `TRUE` and `FALSE` values) to pick elements out of vectors. For instance, try the following

```
> a=runif(20)
> a.low=a[a<0.5]; a.low
> a.high=a[a>=0.5]; a.high
```

What is happening here is that `a<0.5` generates a logical vector the same length as `a` which is then used to select the appropriate values and stick them into a new variable, `a.low`. Type `a<0.5` to see this logical vector. Make sure you understand how this whole approach works.

If you want the positions at which `a` is lower than 0.5, you could say `(1:length(a))[a<0.5]`, but you can also use a built-in function: `which(a<0.5)`. If you wanted the position at which the maximum value of `a` occurs, you could say `which(a==max(a))`. (This normally results in a vector of length 1; when could it give a longer vector?) There is even a built-in command for this specific function, `which.max()` (although `which.max()` always returns just the *first* position at which the maximum occurs).

(What would happen if instead of setting `a.low` you replaced `a` by saying `a=a[a<0.5]`? Why would that be the wrong thing to do?)

We can also combine logical operators (making sure to use the element-by-element `&` and `|` versions of AND and OR):

```
> a[a>0.5 & a <= 0.8]
> a[a>0.5 | a <= 0.8]
```

Finally, it is sometimes handy to be able to drop a particular set of elements, rather than taking a particular set: you can do this with negative indices. For example, `x[-1]` extracts all but the first element of a vector.

5.5.3 Manipulating vectors

A vector can be used in calculations as if it were a number (more or less). Try the following

```
> v1=c(1, 3, 5, 7, 9, 11)
> v2=v1+1; v3=sqrt(v1); v2; v3;
```

Notice that the operations were applied to every entry in the vector. Similarly, commands like `v1-5, 2*v1, v1/10` apply subtraction, multiplication, and division to each element of the vector. The same is true for `v1^2`. Try it.

In R the default is to apply functions and operations to vectors in an *element by element* manner; anything else (e.g. matrix multiplication) is done using special notation (discussed below). This also holds for comparisons of two vectors or a number with a vector. For example,

```
> x=1:5; x
> b=(x<=3); b
```

compares the value 3 with all elements of the vector x and returns either `TRUE` or `FALSE` depending on if the condition is met or not. So if `x` and `y` are vectors (of the same length), then `(x==y)` will return a vector of values giving the element-by-element comparisons. If you want to know whether `x` and `y` are identical vectors, use `identical(x,y)` which returns a single value: `TRUE` if each entry in `x` equals the corresponding entry in `y`, otherwise `FALSE`.

Often, you might want to do something more interesting than adding 10 to a collection of numbers that you stored in a vector. R has many built-in functions that let you do cool stuff very rapidly. Try the following

```
> v1=rnorm(100)
> mean(v1)
> sort(v1)
> plot(v1)
> plot(sort(v1))
> sqrt(v1)
> mean(sqrt(v1))
```

The second to last command might produce a warning and the last command likely produced an error message. Make sure you understand what's going on. We will come back to NaN and similar issues in Section 5.9. We will also discuss the plot command in Section 5.11.

Most of these functions you just used have a number of **optional arguments**, which means in practice that you can choose to specify their value, or if you don't a default value is assumed. For example, `x=rnorm(100)` generates 100 random numbers with a Normal (Gaussian) distribution having mean=0, standard deviation=1. But `rnorm(100,2,5)` yields 100 random numbers from a Gaussian distribution with mean=2, standard deviation=5.

In the R documentation and help pages, the existence of default values for some arguments of a function is indicated by writing (for example) `rnorm(n, mean=0, sd=1)`. Since no default value is given for n, the user must supply one: `rnorm()` gives an error message.

5.5.4 Exercises

Exercise 5.2 Create a vector `v=(1 5 9 13)` using `seq`. Create a vector going from 1 to 5 in increments of 0.2 first by using `seq`, and then by using a command of the form `v=1+b*c(i:j)`.

Exercise 5.3 Execute the following code

```
> c=3; c=c(c,c,c(c,c)); c
```

Explain why it's a really bad piece of code and come up with a better way to produce the same result.

Exercise 5.4 Use `runif(n)` to create a vector of 20 numbers, then find the subset of those numbers that is less than the mean.

Exercise 5.5 Find the *positions* of the elements that are less than the mean of the vector you just created (e.g. if your vector were `(0.1 0.9. 0.7 0.3)` the answer would be `(1 4)`).

Exercise 5.6: Find a way to take only the elements in the odd positions (first, third, ...) of a vector of arbitrary length.

Exercise 5.7 Generate a vector of 1000 random numbers from a Gaussian distribution with mean=3, standard deviation=2. Use the functions `mean, sd` to compute the sample mean and standard deviation of the values in the vector, and `hist` to visualize the distribution. If you can't get these functions to work right away, read the help files (e.g. `help(hist)`).

5.6 Matrices

A matrix is a two-dimensional array of numbers. While one could simply consider a vector as a matrix that happens to have either a single row or a single column, R does treat matrices and vectors a bit differently – though there are overlaps. Just keep in mind that sometimes, certain functions produce different results depending on if they act on matrices or vectors.

5.6.1 Constructing matrices

One way to produce a matrix is to first produce a vector of the matrix entries, and then reshape it to the desired number of rows and columns using the `matrix` function. For example

```
> X=matrix(c(1,2,3,4,5,6),2,3); X
```

takes the values 1 to 6 and reshapes them into a 2 by 3 matrix. Note that values in the data vector are put into the matrix column-wise, by default. You can change this by using the optional parameter `byrow`). For example

```
> A=matrix(1:9,3,3,byrow=T); A
```

Rwill re-cycle through entries in the data vector, if need be, to fill out a matrix of the specified size. So for example `matrix(1,5,5)` creates a 5×5 matrix of all 1's.

Another useful function for creating matrices is `diag`. `diag(v,n)` creates an $n \times n$ matrix with data vector v on its diagonal. So for example `diag(1,5)` creates the 5×5 *identity matrix*, which has 1's on the diagonal and 0 everywhere else.

If their sizes match, vectors can be combined to form matrices, and matrices can be combined with vectors or matrices to form other matrices. The functions that do this are **cbind** and **rbind** (which stand for column-bind and row-bind).

`cbind` binds together columns of two objects. One thing it can do is put vectors together to form a matrix:

```
> v1=seq(1,5); v2=rep(3,5);
> A=cbind(v1,v2); A
```

`matrix(v,m,n)`	$m \times n$ matrix using the values in v
`diag(v,n)`	diagonal $n \times n$ matrix with v on diagonal, 0 elsewhere
`cbind(a,b,c,...)`	combine compatible objects by binding them along columns
`rbind(a,b,c,...)`	combine compatible objects by binding them along rows
`outer(v,w)`	"outer product" of vectors `v,w`: the matrix whose $(i,j)^{th}$ element is `v[i]*w[j]`
`dim(X)`	dimensions of matrix X. `dim(X)[1]`=# rows, `dim(X)[2]`=# columns
`apply(A,MARGIN,FUN)`	apply the function `FUN` to each row of A (if MARGIN=1) or each column of A (if MARGIN=2). See ?apply for details and examples.

Table 4: Some important functions for creating and working with matrices. Many of these have additional optional arguments; use the Help system for full details.

Now try

```
> v1=seq(1,5); v2=rep(3,6);
> A2=cbind(v1,v2); A2
```

and make sure you understand the warning message and what R is doing when it produces the matrix `A`.

Remember that R interprets vectors as row or column vectors according to what you're doing with them. Here it treats them as column vectors so that columns exist to be bound together. On the other hand,

```
> v1=seq(1,5); v2=rep(3,5);
> B=rbind(v1,v2); B
```

treats them as rows. You can also combine matrices, e.g. try `cbind(A,A)`. Table 4 lists several useful functions for dealing with matrices.

5.6.2 Addressing matrices

Matrix addressing works like vector addressing except that you have to specify both the row and column, or range of rows and columns. Try the following:

```
> A=matrix(2:10,3,3)
> A[2,3]
> A[2,2:3]
> A[2:3,2]
> B=A[2:3,1:2]; B
> C=A[2:3,c(1,3)]; C
```

There is an easy shortcut to extract entire rows or columns: leave out the limits.

```
> first.row=A[1,]; first.row
> second.column=A[,2]; second.column
```

As with vectors, addressing works in reverse to assign values to matrix entries. For example,

```
A[1,1]=12; A
```

The same can be done with blocks, rows, or columns, for example

```
> A[1,]=runif(3); A
```

5.6.3 Matrix operations and matrix-vector multiplication

As with vectors, a numerical function applied to a matrix acts element-by-element.

```
> A=matrix(c(1,4,9,16),2,2); A
> sqrt(A);
```

The same is true for scalar multiplication and division. Try `2*A` and `A/3` and see what you get. If two matrices (or two vectors) are the same size, then you can do element-by-element addition, subtraction, multiplication, division, and exponentiation: (`A+B, A-B, A*B, A/B, A^B`). Matrix $\times$ matrix and matrix $\times$ vector multiplication (when they are of compatible dimensions) is indicated by the special notation `%*%`. Remember, **element-by-element is the default in R**. This requires some attention, because R's eagerness to make things work can sometimes let errors get by without warning. So for example

```
> A=matrix(c(1,4,9,16),2,2);
> v=1:2; A*v
```

A is a 2×2 matrix, and v is a vector of size 2, so the matrix-vector product Av is legitimate. However, Av should be a vector, not a matrix. Since you (incorrectly) "asked" for element-by-element multiplication, that's what R did, "recycling" through the elements of v when it ran out of entries in v before it ran out of entries in A. What you should have done is

```
> A%*%v
```

5.6.4 Exercises

Exercise 5.8 Create matrices `A` and `B` with dimensions 2×3 and 3×3 (fill them with numbers any way you want). Try to use `rbind` and `cbind` to build larger matrices by combining the matrices with themselves and with each other. Some of the combinations should not work. Read about the transpose command (`help(t)`), try it and see if you can use it to get those non-working combinations to work. Play with it and look at the matrices you create until you understand what exactly is going on.

Exercise 5.9 Use **runif** to construct a 5×5 matrix of random numbers with a uniform distribution between 0 and 1. Extract from it the second row; the second column; and the 3×3 matrix of the values that are not at the margins (i.e. not in the first or last row, or first or last column). Use `seq` to replace the values in the first row by `2 5 8 11 14`.

5.7 Other variable types

R apart from vectors and matrices, R has additional types of variables. I will only mention them rather briefly. Once you start using them, you will need to teach yourself more about those data types.

5.7.1 Strings

You likely won't need strings as often as the numerical data types. However, sometimes they are useful. Strings are made up of characters. Use `character` to learn more about this data type. Some useful commands for manipulating or printing characters are `paste, substr, cat, print, sprintf`. We have already used `cat` above. Look up the other functions to learn how to use them. Here are a few simple examples

```
> n=4;
> print(sprintf('The number is %d',n));
> cat("It is really",n,"\n");
> paste('b',n,' the dawn',sep = "");
```

5.7.2 Lists

Vectors and matrices have to contain elements that are all the same type: lists in R can contain anything — vectors, matrices, other lists ... Indexing is a little different too, use `[[ ]]` to extract an element of a list by number or name, or `$` to extract an element by name only. Try:

```
> x=c(5,9,1); y=matrix(rep(5,6),nrow = 2, ncol = 3); z=c("one","two","three");
> L = list(L1=x,L2=y,L3=z)
> L$L1
> L[["L3"]]
> L[[2]]
```

5.7.3 Data frames

Data frames are the solution to the problem that vectors and matrices (which might seem to be the most natural way to store data) can only contain a single type of data, but most data sets have several different kinds of variables for each observation. Data frames are a hybrid of lists and vectors; internally, they are a list of vectors that can be of different types but all have to be the same length, but you can do most of the same things with them (e.g., extracting a

subset of rows) that you can do with matrices. You can index them either the way you would index a list, using `[[ ]]` or `$` — where each variable is a different item in the list — or the way you would index a matrix. Use `as.matrix()` if you have a data frame (where all variables are the same type) that you really want to be a matrix, e.g. if you need to transpose it (use `as.data.frame()` to go the other way).

5.8 Reading and writing data

Once you start doing serious work with R, you might often want to work with data that has been previously recorded. You can relatively easily import data into R. The best way to do so is to have the data in a plain text table, with entries separated by tabs or semicolons or similar field delimiters. You can easily read and write data using `read.table, read.csv, write.table, write.csv` and similar variants. Check the help file. There are ways to read and write Excel files, but the best procedure is to first convert any Excel file to a CSV (comma-separated-values) type file, which you can easily do in Excel. If you absolutely need to read/write Excel files, search online for help. Also note that you need to make sure that R is looking for the data file in the right place. Either move the data file to your current working directory, or change the line so that it points to the actual location of the data file.

5.9 Special values

R has a number of "special values". These are `NA, NaN, Inf, -Inf`. The last two should be obvious, they arise for instance if you do `1/0, -1/0`. You can compute with `Inf`, try

```
> Inf > 1E10
> Inf - 1E10
```

But something like `0/0` or `Inf/Inf` is ill defined, and therefore you get `NaN` (= not a number). Check `?NaN` for more information on those three. There are special functions, such as `is.nan` that allow you to manipulate such values. The value `NA` stands for "not available" and often represents a missing value in a dataset. There is no fixed rule on how functions that you apply to a dataset which contains NA behave. Some functions have built-in options for dealing with NA. Try for instance

```
> n=c(2,3,NA)
> mean(n)
> mean(n,na.rm=TRUE)
```

Re-visit the help entry for `mean` to understand what is happening and `?NA` to learn more. In general, you need to be careful with any of these special values. While they can often be used to your advantage, make sure you carefully check all your computations, otherwise things can go badly wrong.

5.10 Exercises

Exercise 5.10 Create a matrix, fill it with any numbers you like. Then save the matrix to a CSV file, using `write.csv()`. Use Excel or OpenOffice Calc to open the file you created. Delete

a random entry in the table. Then save it again. Make sure you keep the CSV format and don't change to the native format of whatever program you are using. Now use `read.csv()` to read the file back into R (e.g. `M=read.csv('test.csv')`). Take a look at the new matrix. It should contain a `NA` value. (I tested this with Excel 2007 and OpenOffice 3, I hope it also works with other spreadsheet programs).

Exercise 5.11 Create any matrix filled with numbers. Set at least one entry in the matrix to NA. Compute the mean over all rows, over all columns and over the whole matrix. Make sure you deal properly with the `NA` value. Read `?mean` to see how to deal with `NA` values and how to do means of rows or columns.

Exercise 5.12 Write an R script that automatically creates a 4x5 matrix, writes it to a file, reads it from the file, replaces an entry with `NA`, computes the mean and sum of the matrix, and prints the values onto the screen. Note that you might want to use `write.table` and `read.table` instead of write/read CSV. If you use the latter, you might have to manipulate the matrix a bit once you load it.

Exercise 5.13 Modify your script such that a **random** entry is replaced with `NA`. Hint: Think about a way to produce 2 random integers between 1 and the number of columns/rows of the matrix. Then use those 2 numbers in `B[rownum,colnum]=NA`. There are many different possibilities to solve this.

5.11 Plotting and saving figures

R comes with a wealth of graphical capabilities, both in the base package and several packages you can download and install. A very versatile function is `plot`. Here is a simple example of a plot

```
> x=seq(1,21,by=2);
> y=rnorm(length[x]);
> plot(x,y);
```

The above snippet of code contains a mistake! I stuck it in on purpose. You should by now know how to fix it.

There are myriad options for plot. You can set pretty much anything in the figure, the line or marker type, style, thickness, color, you can change axes, add legends, etc. Use ?plot to learn about all those options. Some other useful types of plots are `points, lines, barplot, matplot, hist, persp, image`. Check out the R help files or the resources mentioned in Section 3.2 for more on plotting. Further manual and books can be found on the R webpage or Amazon. A few specific resources that might be worth checking out:

- Producing Simple Graphs with R by Frank McCown: `http://www.harding.edu/fmccown/R/`. Nice quick intro to basic plotting.
- R Graph Gallery: `http://addictedtor.free.fr/graphiques/`. A large collection of mostly fancy R graphs. Shows you some of the graphics capabilities of R and R packages. You can also see the source code that lead to these figures.

R plotting is so flexible that you can produce (almost) any plot you like – it's just a matter of finding out how to do it.

To learn how to save figures you created, see the last exercise in the next section.

5.12 Exercises

Exercise 5.14 The axes in plots are scaled automatically, but the outcome is not always ideal (e.g. if you want several graphs with exactly the same axes limits). You can control scaling using the `xlim` and `ylim` arguments, e.g. `plot(x,y,xlim=c(x1,x2))` will draw the graph with the x-axis running from `x1` to `x2`, and using `ylim=c(y1,y2)` within the `plot()` command will do the same for the y-axis. Create two vectors `x, y`, plot them, and play around with the x and y-axes.

Exercise 5.15 Several graphs can be placed within a single figure by using the `par` function (short for "parameter") to adjust the layout of the plot. For example the command `par(mfrow=c(m,n))` divides the plotting area into m rows and n columns. As a series of graphs is drawn, they are placed along the top row from left to right, then along the next row, and so on. `mfcol=c(m,n)` has the same effect except that successive graphs are drawn down the first column, then down the second column, and so on. Save any of the example files (`yari-example1.r` or `yari-example2.r`) with a new name and modify the program as follows. Before the first `plot` command, write `par(mfcol=c(2,1))`. Also replace one of the `lines` commands with a `plot` command. Run the program using `source`. Do the same again, using `mfcol=c(1,2)`. Also play with axes limits and similar options to make the plots look nicer.

Exercise 5.16 Use `?par` to read about other plot control parameters that can be set using `par()` (feel free to just skim — this is one of the longest help files in the whole R system). Then draw a 2×2 set of plots, each showing the line $y = 5x + 3$ with x running from 3 to 8, but with 4 different line styles and 4 different line colors. Make sure you are plotting lines, not markers/symbols!

Exercise 5.17 Modify one of the example scripts so that at the very end it saves the plot as a .png file. In Windows you can do this with `savePlot()`, otherwise with `dev.print()`. Use `?savePlot`, `?dev.print` to read about these functions.

5.13 Flow control in R

When you write longer programs, you sometimes want to execute or not execute certain commands, depending on the current status of things. For instance you might want to do one thing if `a>b` and another if `a<b`. In addition, you might often want to do the same thing many times. To get your program to do such things, i.e. to control the flow of your code, R has several basic ways of achieving this. We will briefly discuss them now.

5.13.1 For-loops

Loops make it easy to do the same operation over and over again. A **for** loop runs for a specified number of steps and is written as

```
for (var in seq)
{
    commands
}
```

Here's an example. Note that it might be a good idea to write this example (and the ones that follow below) into an R script, give it a filename, and then run it with `source`.

```
#create vector of length 10 filled with zeros. Set first element to 1.
popsize=rep(0,10); popsize[1]=1;

# calculate population size at times 2 through 10, write to Command Window
for (n in 2:10)
{
      popsize[n]=2*popsize[n-1];
      x=log(popsize[n]);
      cat(n,x,"\n");
}
plot(1:10,popsize,type="l");
```

The first time through the loop, n=2. The second time through, n=3. When it reaches n=10, the for-loop is finished and R starts executing any commands that occur after the end of the loop. The result is a table of the log population size in generations 2 through 10.

Several `for` loops can be nested within each other, which is sometimes useful for working with matrices as in the example below. It is important to notice that the second loop is **completely** within the first. Loops must be either **nested** (one completely inside the other) or **sequential** (one starts after the previous one ends).

```
A=matrix(0,3,3);
for (row in 1:3)
{
for (col in 1:3)
    {
       A[row,col]=row*col
    }
}
```

Type this into a script file and run it; It seems like nothing happens, but if you now type `A` into the console, you should see

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9
```

Line 1 creates A as a matrix of all zeros - this is an easy way to create a matrix of whatever size you need, which can then be filled in with meaningful values as your program runs. Then two nested loops are used to fill in the entries. Line 2 starts a loop over the rows of A, and immediately in line 3 a loop over the columns is started. To fill in the matrix we need to consider all possible values for the pair (row, col). So for row=1, we need to consider col=1,2,3. Then for row=2 we also need to consider col=1,2,3, and the same for row=3. That's what the nested for-loops accomplish. For row=1 (as requested in line 2), the loop in lines 3-5 is executed until it ends. Then we get to the end in line 6, at which point the loop in line 2 moves on to row=2, and so on.

Also note in the above example how the code is written, with opening and closing parentheses in the same position. This is not strictly necessary, but writing code like this helps you to quickly see which blocks of statements belong together and makes reading and programming much easier.

If this discussion of looping doesn't make sense to you, **stop now and get help**. Loops are an essential component of programming in R or other languages. In fact, the "getting help" applies to any topic we have covered so far and will continue to cover. Make sure you are comfortable with everything explained here and that you fully understand it. If not, keep practicing and asking for clarification.

5.13.2 While-loops

A `while` loop lets an iteration continue until some condition is satisfied. For example, we can solve a model until some variable reaches a threshold. The format is

```
while(condition)
{
      commands
}
```

The loop repeats as long as the condition remains true. Write the following lines into an R script and run them:

```
n=1
while (n<10)
{
  n=2*n
}
print(sprintf("n=%d",n))
```

Note the double-command `print(sprintf())` which has a smiliar effect to `cat` used above, but differs in the details. Learn about it by reading the help file. Move the `print(sprintf())` command into the loop and run it again. What would happen if you replace the statement inside the loop with $n = n^2$? If you don't know, try it. And you might want to revisit the discussion about how to interrupt R...

Within a while-loop it is often helpful to have a **counter** variable that keeps track of how many times the loop has been executed. In the following code, the counter variable is `ct`:

```
ct=1;
while(condition)
{
    commands
    ct=ct+1;
}
```

The result is that `ct=1` is true while the `commands` (whatever they are) are being executed for the first time. Afterward `ct` is set to 2, and this remains true during the second time that the commands are executed, and so on. One use of counters is to store a series of results in a vector or matrix: on the ct^{th} time through the commands, put the results in the ct^{th} entry of the vector, ct^{th} row of the matrix, etc.

5.13.3 Branching

Logical conditions also allow the rules for "what happens next" in a model to depend on the current values of state variables. The **if** statement lets us do this; the basic format is

```
if(condition)
{
    some commands
}
else
{
    some other commands
}
```

More complicated decisions can be built up by nesting one `if` block within another, i.e. the "other commands" under `else` can include a second `if` block. Here is an example where a population grows, and the growth tails off in several steps as the population size increases.

```
rm(list=ls()) #this clears the workspace
graphics.off(); #close all graphics windows
popnow=10;
popsize=popnow;
for (i in 1:50)
{
```

```
    if(popnow<250)
    {
        popnow=popnow*2;
    }
    else
    {
        if(popnow<500)
        {
            popnow=popnow*1.5
        }
        else
        {
            popnow=popnow*0.95
        }
    }
   popsize=c(popsize,popnow);
}  #this ends the for loop
plot(popsize,type="b") #plots with both lines and symbols
```

What does this accomplish?

- If `popnow` is still < 250, then growth by a factor of 2 occurs. Since the `if` condition was satisfied, the entire `else` block isn't looked at.
- If popnow is not < 250, R moves on to the `else` statement, and immediately encounters another `if`.
- If popnow is < 500 the growth factor of 1.5 applies.
- If neither of the two `if` conditions is satisfied, the final `else` block is executed and population declines by 5% instead of growing.

The final command inside the `for-loop` adds the current population size, `popnow`, to the vector `popsize`, which therefore keeps growing. For a vector or matrix to grow like this, the variable needs to exist. Test this by removing the `popsize=popnow` command and run it again. In general, these growing arrays make the code run slower compared to creating an initial array of all-zeros and filling it. But sometimes you don't know how many numbers your code produces, and then these growing arrays can be useful.

Also note the comment behind the last }. Such comments can be very useful if you have long blocks of code that stretch over pages and you can't see where the closing parentheses belong to.

5.13.4 Exercises

Exercise 5.18 : For the code in the last section, replace the growing array `popsize` with a fixed array that is initially empty, and then fill the array with the results as the simulation runs.

Exercise 5.19 : Imagine that while doing fieldwork in some distant land you and your friend have picked up a parasite that grows exponentially until treated. Your case is more severe than your friend's: on return there are 400 of them in you, and only 120 in your friend. However, your field-hardened immune system is more effective. In your body the number of parasites grows by 10 percent each day, while in your friend's it increases by 20 percent each day. Write a script file that uses a for-loop to compute the number of parasites in your body and your friend's over the next 30 days, and draws a single plot of both with the y-axis on log-scale.

Exercise 5.20 Write a script file that uses a while-loop to compute the number of parasites in your body and your friend's so long as you are sicker than him (i.e. so long as you have more parasites) and stops when your friend becomes sicker than you. Then print out the day at which that happens.

Exercise 5.21 : Write a script file that uses for-loops to create the following 5×5 matrix A.

0	1	2	3	4
0.1	0	0	0	0
0	0.2	0	0	0
0	0	0.3	0	0
0	0	0	0.4	0

Exercise 5.22 Write a script file that creates 1000 random numbers. Write this in several ways:
1) Use a while-loop and dynamically add the newly created random number to a vector.
2) same as 1), but use a for-loop.
3) same as 1, but pre-allocate the result vector (i.e. create a vector of length 1000 that is filled with zeros) and then write the newly created random number into the appropriate place in the vector.
4) Use the vectorization ability of R, i.e. the fact that R can produce as many random numbers as you want with a single command.

Exercise 5.23 It is often important to write code that runs as fast as possible. R has some built-in functions that can help you test the speed of your code. For the script in the last exercise, add commands that compute the time R needs for each of the different ways of creating the random numbers. You can for instance use a function such as `proc.time()` or similar.

5.14 Functions in R

Functions (often called subroutines in other computing languages) allow you to break a program into subunits. This makes complex problems easier to program, and helps us (and others) to understand the logical flow of programs. Each function is an independent little program, performing one task or a few related tasks, and returning the results. A program can then be written to call on various functions to perform different tasks. Each function can be written and tested independently, which leads naturally to the generally recommended modular style of program construction.

Schematically, a function looks like

```
function.name=function(argument1,argument2,...)
{
     command;
     command;
       ...
     command;
     return(value)
}
```

In a somewhat oversimplified explanation, you send something into a function, the function performs some operations on whatever you sent it, and gives you back a result. The result is usually returned using the `return()` function. You have already used a lot of built-in functions in previous sections. For instance the function `mean` takes a collection of numbers, computes the mean, and returns it to you as an answer. We can write our own function to do a mean, here it is

```
mymean=function(v)
{
sumv=0;
for (n in (1:length(v)))
{
sumv=sumv+v[n];
}
meanv=sumv/length(v);
    return(meanv)
}
#end function, start main program
u=runif(100);
meanvalue=mymean(u) #this is the line which calls the function
cat("Mean is", meanvalue,"\n")
```

The main program creates a vector and stores it in the variable `u`. This is sent into the function `mymean`. The function "picks up" `u` and stores it internally in the variable `v`. It then computes the mean (note that we make use of another built-in R function, namely `length()`). The variable `v` is *internal* to the function; if you use `mymean` in a program, the program won't "know" the value of `v`. Also note that you could have also saved the function into its own script and then called it by its file name.

You can in principle place your functions somewhere in the script file, as long as it is before the line where you want to call the function. But it is usually better to place them all at the beginning of the script, this gives your program a somewhat less cluttered structure.

Functions can return several different values. For that, you need to combine them into a list with named parts. For instance in the example scripts, the function called `odeequations` returns the variables that are being integrated (`dSdt, dIndt, dRdt` or `dUtcdt, dItcdt, dVirdt`).

Exercise 5.24 Write a script that defines a function `mymax` which takes as input a single matrix, and returns a vector with components that contain the maximum value in each column

of the matrix. Hint: You could use a loop or you might find the command `apply()` helpful.

Exercise 5.25 Extend your function from the last exercise so that one can specify if it should return the maximum for each column or each row of the matrix. Hint: Write a function that has as input the matrix and another variable which defines what procedure should be done. E.g. `mymax(M,cl)`, where `cl=1` or `cl=TRUE` returns the maximum of each column, and if `cl=0` or `cl=FALSE`, the maximum of each row is returned. Inside the function, you need statements of the form `if (cl==1)` to distinguish between the cases.

Exercise 5.26 Extend your function further such that it tests if the first entry is really a matrix, and the second entry is really either 0/1 or TRUE/FALSE. If not, make the function print an error message.

5.14.1 Functions for solving systems of differential equations

R built-in functions make it relatively easy to do some complicated things. One important application for us is finding numerical solutions for a system of differential equations. While the core packages that come with the R installation do not include ODE solvers, several packages are available. One such package is `deSolve`, which we installed previously. This package contains a number of different solvers. It is worth reading through the documentation to familiarize yourself a bit with those solvers. For most problems, you can likely use the solver called `lsoda` without much difficulty. But sometimes this solver might fail. If you get weird results that don't make sense, it's time to read the help files for `lsoda` and the `deSolve` package. And if you start using ODE solvers more frequently, you should learn a bit more about how they work, what stiff and non-stiff problems are, etc. Wikipedia has some good basic information and the "Numerical Recipes" books are very good as well (an older version of Numerical Recipes is available online for free).

To use the ODE solvers, the first step is to write a function that specifies the right side of the set of differential equations *in the specific format required by the solvers.* Here's what that looks like for the differential equation used in `yari-example2.r`

```
odeequations=function(t,y,parameters)
{
   Utc=y[1]; Itc=y[2]; Vir=y[3];
   b=parameters[1]; delta=parameters[2]; p=parameters[3]; clear=parameters[4];

   #these are the 3 differential equations
   dUtcdt=-b*Vir*Utc;
   dItcdt=b*Vir*Utc-delta*Itc;
   dVirdt=p*Itc-clear*Vir;

   return(list(c(dUtcdt,dItcdt,dVirdt)));
}
```

The call of this function in the main text is done by

```
odeoutput=lsoda(Y0,timevec,odeequations,parvec);
```

Note that `lsoda` requires that your ODE function (here `odeequations`) has input arguments `(t,y,parameters)`, even if only the state vector `y` is used to compute the vector field. Here `t` is time, `y` is the state vector, and `parameters` is a vector of parameter values for the function. Note that the name of the variables does not matter. It is customary to call time `t` and the vector of variables `y`. The variable for parameters is often called `params, pars, p` or something else entirely. That might likely be confusing initially, but you'll get used to it. The solver `lsoda` has many options. At some point, read the help file and try to understand some of them. For instance changing `atol, rtol` can often be very useful.

5.15 Other stuff that might be useful

As you saw in the example scripts, we cleared the workspace before running the script, using

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

It's a good idea to use that at the beginning of your program.

You will soon spend a fair amount of time trying to find out why your program produces errors or otherwise doesn't perform as it should. During this bug-hunting season, a useful command is `browser()`. Just stick it anywhere into your script. R will then run up to the point where it encounters this command, then stop and turn the console over to you. Sticking in the command at various places in your script can help you better localize the source of your error. Also, once you have the console, you can check the values of all current variables and thereby test if some are not what they are supposed to be. You can exit the browser command by either typing `c`, which continues execution of your script, or `Q`, which exits the script.

6 A small project

Once you worked through the manual this far, and are able to fully understand all what you read and did, you are well on your way to becoming a proficient R user. This final section will push you further in that direction by making you write a complete simulation project.

Write a script file that simulates a simple model for density-independent population growth with spatial variation. The model is as follows. The *state variables* are the numbers of individuals in a series of $L = 20$ patches along a line (L stands for "length of the habitat") .

1	2	3	4	...					...	L-1	L

Let $N_j(t)$ denote the number of individuals in patch j ($j = 1, 2, \ldots, L$) at time t ($t = 1, 2, 3, \ldots$), and let λ_j be the geometric growth rate in patch j. The *dynamic equations* for this model consist of two steps:

1. Geometric growth within patches:

$$M_j(t) = \lambda_j N_j(t) \quad \text{for all } j. \tag{1}$$

2. Dispersal between neighboring patches:

$$N_j(t+1) = (1-2d)M_j(t) + dM_{j-1}(t) + dM_{j+1}(t) \quad \text{for } 2 \le j \le L-1 \tag{2}$$

where $2d$ is the "dispersal rate". We need special rules for the end patches. For this exercise we assume *reflecting boundaries*: those who venture out into the void have the sense to come back. That is, there is no leftward dispersal out of patch 1 and no rightward dispersal out of patch L:

$$\begin{aligned} N_1(t+1) &= (1-d)M_1(t) + dM_2(t) \\ N_L(t+1) &= (1-d)M_L(t) + dM_{L-1}(t) \end{aligned} \tag{3}$$

Write your script to start with 5 individuals in each patch at time t=1, iterate the model up to t=50, and graph the log of the total population size (the sum over all patches) over time. Use the following growth rates: $\lambda_j = 0.9$ in the left half of the patches, and $\lambda_j = 1.2$ in the right.

Notes and hints:

1. Write your program so that d and L are parameters, in the sense that the first line of your script file reads `d=0.1; L=20;` and the program would still work if these were changed other values.
2. You start with a founding population at time 1, and use a loop to compute successive populations at times 2,3,4, and so on.
3. The population is described by a vector rather than a number. Therefore, to store the population state at times $t = 1, 2, \cdots, 50$ you will need a matrix `njt` with 50 rows and L columns. Then `njt(t,:)` is the population state vector at time t.
4. Vectorize! Vector/matrix operations are <u>much</u> faster than loops. Set up your calculations so that computing $M_j(t) = \lambda_j N_j(t)$ for $j = 1, 2, \cdots, L$ is a `one-line` statement of the form `a=b*c` . Then for the dispersal step: if $M_j(t), j = 1, 2, \ldots, L$ is stored as a vector `mjt` of length L, then what (for example) are $M_j(t)$ and $M_{j\pm1}(t)$ for $2 \le j \le (L-1)$?

Next, modify the model to ask how the spatial arrangement of good versus bad habitat patches affects the population growth rate. For example, does it matter if all the good sites ($\lambda > 1$) are at one end or in the middle? What if they are not all in one clump, but are spread out evenly (in some sense) across the entire habitat? Notes and hints:

1. Patterns will be easiest to see if good sites and bad sites are very different from each other.
2. Patterns will be easiest to see if you come up with a nice way to compare growth rates across different spatial arrangements of patches.
3. Don't confound the experiment by also changing the proportion of good versus bad patches at the same time you're changing the spatial arrangement.

7 Next steps

You reached the end of this short tutorial. You have learned quite a bit. But as you start using R you will find very soon (probably within minutes) that you encounter a problem that this guide does not answer. Now it's self-learning time. I hope that by going to the resources listed in this guide, you will be able to find the answer in a short time.