Model fitting and inference for infectious disease dynamics

Useful **R** commands

Contents

1 Introduction

This document provides a summary of **R** commands that will be useful to learn or refresh in preparation for the course on *Model fitting and inference for infectious disease dynamics*, 8-10 July at the London School of Hygiene & Tropical Medicine. While we expect that you will have some knowledge of **R**, the commands listed here are the ones that we think it would be most useful for you to familiarise yourselves with in order to be able to read the code we will provide for the practical session, and to debug any code you write yourselves during the sessions.

2 Data types

The data types we will be working with in the course are (named) *vectors*, *lists*, and *data frames*. More information on data types in **R** can be found in many places on the web, for example the R programming wikibook.

2.1 Vectors

Vectors are an ordered collection of simple elements such as numbers or strings. They can be created with the `c()` command.

```
a <- c(1, 3, 6, 1)
a
```

```
[1] 1 3 6 1
```

An individual member at position `i` be accessed with `[i]`.

```
a[2]
```

```
[1] 3
```

Importantly, vectors can be named. We will use this to define parameters for a model. For a named vector, simply specify the names

```
b <- c(start = 3,  inc = 2,  end = 17)
```

The elements of a named vector can be addressed both by index

```
b[2]
```

```
inc
  2
```

and by name

```
b["inc"]
```

```
inc
  2
```

To strip the names from an unnamed vector, one can use double brackets

```
b[["inc"]]
```

```
[1] 2
```

```
b[[2]]
```

```
[1] 2
```

or the `unname` function

```
unname(b)
```

```
[1]  3  2 17
```

2.2 Lists

Lists are different from vectors in that elements of a list can be anything (including more lists, vectors, etc.), and not all elements have to be of the same type either.

```
l <- list("string", c(3,4,1))
l
```

```
[[1]]
[1] "string"

[[2]]
[1] 3 4 1
```

Similar to vectors, list elements can be named:

```
l <- list(text = "string", vector = c(3,4,1))
l
```

```
$text
[1] "string"

$vector
[1] 3 4 1
```

The meaning of brackets for lists is different to vectors. Single brackets return a list of one element

```
l["text"]
```

```
$text
[1] "string"
```

whereas double brackets return the element itself (not within a list)

```
l[["text"]]
```

```
[1] "string"
```

More on the meanings of single and double brackets, as well as details on another notation for accessing elements (using the dollar sign) can be found in the R language specification.

2.3 Data frames

Data frames are a 2-dimensional extensions of vectors. They can be thought of as the **R**-version of an Excel spreadsheet. Every column of a data frame is a vector.

```
df <- data.frame(a = c(2, 3, 0), b =c(1, 4, 5))
df
```

```
  a b
1 2 1
2 3 4
3 0 5
```

Data frames themselves have a version of single and double bracket notation for accessing elements. Single brackets return a 1-column data frame

```
df["a"]
```

```
  a
1 2
```

```
2 3
3 0
```

whereas double brackets return the column as a vector

```
df[["a"]]
```

```
[1] 2 3 0
```

3 Functions

Functions are at the essence of everything in **R**. The `c()` command used earlier was a call to a function (called `c`). To find out about what a function does, which parameters it takes, what it returns, as well as to see some examples for use of a function, one can use `?`, e.g. `c` or `data.frame`.

To define a new function, we assign a `function` object to a variable. For example, a function that increments a number by one.

```
add1 <- function(x) {
    return(x + 1)
}
add1(3)
```

```
[1] 4
```

To see what any function does in detail, one can look at its source code by typing the function name:

```
add1
```

```
function(x) {
    return(x + 1)
}
```

3.1 Passing functions as parameters

Since functions themselves are variables, they can be passed to other functions. For example, we could write a function and an argument and applies the function twice to the argument.

```r
doTwice <- function(f, x) {
    return(f(f(x)))
}
doTwice(add1, 3)
```

```
[1] 5
```

3.2 Debugging functions

Writing functions comes with the need to debug them, in case they return errors or faulty results. **R** provides its own debugger, which is started with `debug`:

```r
debug(add1)
```

On the next call to the function, this puts us into **R**'s own debugger, where we can advance step-by-step (with `n`), inspect variables, evaluate calls, etc. `Q` quits the debugger. To stop debugging a function, we can use

```r
undebug(add1)
```

More on the debugging functionalities of **R** can be found on the Debugging in R pages.

An alternative way for debugging is to include printouts in the function, for example using `cat`

```r
add1 <- function(x) {
    cat("Adding 1 to", x, "\n")
    return(x + 1)
}
add1(3)
```

```
Adding 1 to 3
[1] 4
```

4 Probability distributions

Probability distributions are at the heart of many aspects of model fitting. *R* provides functions to both estimate the probability of obtaining a certain value under a given probability distribution and to sample random numbers from the same distribution. The corresponding functions have a common nomenclature, that is `dxxx` for the probability (density) of a given value and `rxxx` for generation of a random number from the same distribution. For example, for a uniform distribution we have `dunif` and `runif`, and to generate a random number between 0 and 5 we can write

```r
r <- runif(n = 1, min = 0, max = 5)
```

This number has density $1/(\max - \min) = 0.2$ within the uniform distribution:

```r
dunif(x = r, min = 0, max = 5)
```

```
[1] 0.2
```

For almost all probability distributions, we can get the logarithm of the probability density by passing `log = TRUE`:

```r
dunif(x = r, min = 0, max = 5, log = TRUE)
```

```
[1] -1.609438
```

A number of probability distributions and their corresponding **R** functions can be found on the R Tutorial web pages.

5 Running dynamic models

R provides packages for running both deterministic and stochastic dynamic models. For deterministic models, the `deSolve` package is a good choice, whereas for stochastic models, `adaptivetau` is recommended.

5.1 Deterministic models

The `deSolve` package can be installed with `install.packages("deSolve")`. Once installed, it is loaded with

```r
library(deSolve)
```

The command for running a model based on a system of differential equations (e.g., the ones of the SIR model), is `ode`. It takes as parameters the initial state (as a named vector), parameters (again, a named vector), the times at which to produce model output, and a model function `func` – for more details, see the deSolve vignette. The `func` argument is for specifying the derivatives in the system of ordinary differential equations. It is a function that takes the current time, the current state of the system and the parameters and returns a list of transition rates. For the SIR model, for example, we could write

```
SIR_ode <- function(time, state, parameters) {

    ## parameters
    beta <- parameters[["R0"]] / parameters[["infectious.period"]]
    gamma <- 1 / parameters[["infectious.period"]]

    ## states
    S <- state["S"]
    I <- state["I"]
    R <- state["R"]

    N <- S + I + R

    dS <- -beta * S * I/N
    dI <- beta * S * I/N-gamma * I
    dR <- gamma * I

    return(list(c(dS, dI, dR)))
}
```

We can plug this into the `ode` function

```
ode(y = c(S = 999, I = 1, R = 0),
    times = 1:10,
    parms = c(R0 = 2, infectious.period = 1),
    func = SIR_ode)
```

```
   time        S          I          R
1     1 999.0000   1.000000   0.000000
2     2 995.5805   2.705097   1.714392
3     3 986.4422   7.232761   6.325009
4     4 962.7806  18.754785  18.464618
5     5 906.1827  45.060377  48.756919
6     6 792.7535  91.625369 115.621083
7     7 626.8686 140.122500 233.008861
8     8 464.1066 152.573180 383.320218
9     9 349.8650 125.531815 524.603145
10   10 283.0729  86.402205 630.524931
```

5.2 Stochastic models

The adaptivetau package can be installed with install.packages("adaptivetau"). Once installed, it is loaded with

```
library(adaptivetau)
```

The adaptivetau package uses a different syntax from the deSolve package. Instead of providing a function to calculate the rates of change at each time point, one specifies a list of *transitions* and their rates. Examples for how this is done can be found in the adaptivetau vignette.

For the SIR model, we could write

```
SIR_transitions <- list(
    c(S = -1, I = 1), # infection
    c(I = -1, R = 1) # recovery
)

SIR_rateFunc <- function(x, parameters, t) {

    beta <- parameters[["R0"]]/parameters[["infectious.period"]]
    nu <- 1/parameters[["infectious.period"]]

    S <- x["S"]
    I <- x["I"]
    R <- x["R"]

    N <- S + I + R

    return(c(
        beta * S * I / N, # infection
        nu * I # recovery
    ))
}
```

To run the stochastic model, we then use the ssa.adaptivetau function, which takes a vector of initial conditions, the list of transitions and rate function, a named vector of parameters, and the final time (with simulations starting at time 0).

```r
run <- ssa.adaptivetau(init.values = c(S = 999, I = 1, R = 0),
                       transitions = SIR_transitions,
                       rateFunc = SIR_rateFunc,
                       params = c(R0 = 2, infectious.period = 1),
                       tf = 10)
head(run)
```

```
          time   S I R
[1,] 0.0000000 999 1 0
[2,] 0.1852643 998 2 0
[3,] 0.3986411 997 3 0
[4,] 0.4152689 996 4 0
[5,] 0.4390597 996 3 1
[6,] 0.5022681 996 2 2
```