Introduction to R and Rmarkdown for reactive transport modelers

Exercises Accompanying the Course Reaction Transport Modeling in the Hydrosphere

Karline Soetaert and Lubos Polerecky

Januari 2021

This short lecture is an introduction to the programming tools that will be used in the *reactive transport modeling* class. In this class, we will merge text, programming code, and the results of this code in one document. The programming language we use is *R*, the text is written in *markdown*, and the creation of the documents is done by *knitr*. In this document, we show how to work with R and with Rmarkdown, and we introduce the basic features that you will be using in the modelling class. We suggest that you look at the examples given in this document, and recreate some of them. Then proceed to making the exercises.

Introduction: the R-language

Core R software

R can be downloaded from the web site http://www.r-project.org/. Choose the precompiled binary distribution. On this website, you will also find useful documentation.

Other useful software

We will run *R* from within *Rstudio*, which provides an integrated development environment for *R*. It includes R-sensitive syntax and help. *Rstudio* can be downloaded from URL http://rstudio.org.

R packages

Many of the useful functions are not part of the *R* base program, but made available by means of *packages*.

A package in *R* is a collection of files containing many functions that perform related tasks. They are not necessarily made by the *R* core-development team but by experts in other fields, e.g., in the field of modelling.

To use *R* for the examples in this course, two packages need to be downloaded.

- *ReacTran* (Soetaert and Meysman, 2012). This package contains functions for reactive transport modelling. When installing *ReacTran*, the following packages will also be installed:
  - *deSolve* (Soetaert et al., 2010). Performs numerical integration.
  - *rootSolve* (Soetaert, 2009). Finds the root of equations.
- *marelac* (Soetaert and Petzoldt, 2018): Contains functions and constants from the aquatic sciences.

Downloading specific packages can be done within *Rstudio*. Select menu item *packages/install packages*, choose a nearby site (e.g., France Paris) and select the package you need.

Scientific programming practice—Rmarkdown

Scientific research often involves computation, e.g., to convert numbers to different units, to calculate derived quantities, to perform statistical analyses on the data, or to create and solve environmental models. More and more these computations are done by writing computer codes, or scripts. These scripts consist of a sequence of commands that tell the computer what it should do.

It is good programming practice to write computer codes that are easy to understand. This not only facilitates exchange of these codes with other people (e.g., colleagues, your supervisor or lecturer during class), but it will also give you a headstart if you want to use these codes in the future.

In addition, well-documented code is an essential step to reproducible and reusable science.

A first coding attempt

Consider the following piece of an R-code:

```
head(airquality,n=2)
par(mfrow=c(1,2))
plot(airquality$Solar.R,type="l",xlab="time",ylab="Solar.R")
plot(airquality$Wind,type="l",xlab="time",ylab="Wind")
```

While we can more or less guess what this code does, it is not clear where the data come from and what the purpose of this code is. Also, the statements themselves are not legible.

Adding structure and comments in codes

Readability of code can easily be increased by adding *structure* to the code: alignment of comparable sections, use of spaces and more.

One way to document the above code is to add comments to it. In *R*, comments are preceded by a hash-sign ("#").

The following code is self explanatory:

```
# The R dataset "airquality" contains daily air quality measurements in New York,
# from May to September 1973.
# In the code below, we first look at the first two rows of this dataset
# and then plot the solar radiation and wind data, in two figures next to one another
# source code written by Karline Soetaert

head(airquality, n = 2)  # show first two lines
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
```

```
par(mfrow = c(1, 2))     # figures aligned in one row, two columns
plot(airquality$Solar.R, type = "l", xlab = "time", ylab = "Solar.R")
plot(airquality$Wind,    type = "l", xlab = "time", ylab = "Wind"   )
```

If we execute this code, then the first two lines of the dataset will be printed to the console, and the graphs will be created and included in a figure within the document. However, the output will be separate from the code that generated it.

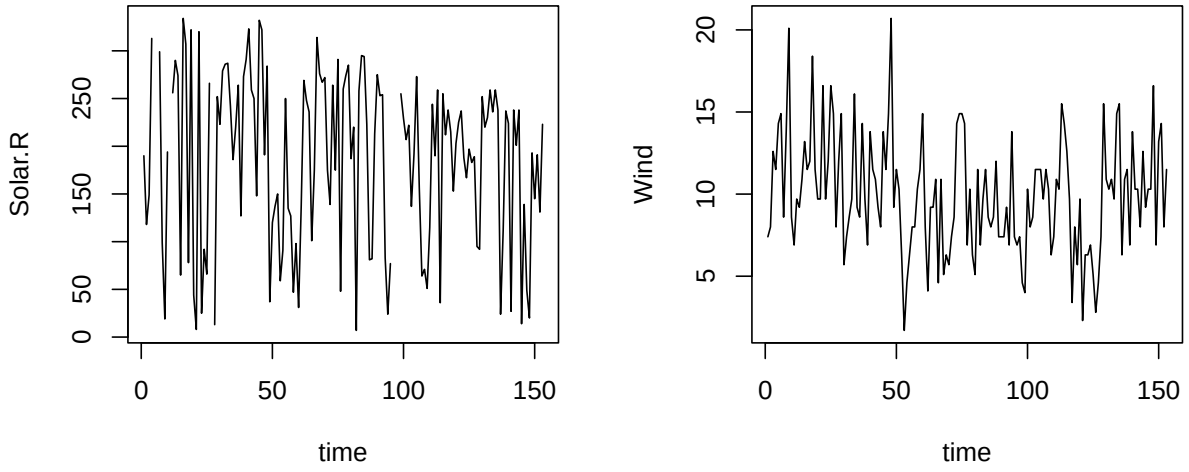


Figure 1: Plotting example.

R plus markdown—creating fully integrated documents

Even better is to have the text, the code and the output in one document. This is what *R plus markdown* does. When the final document is "created", the R-code is executed and the results are merged in the document.

Figure 2 shows how this works. The *Rmarkdown* document (screencapture from Rstudio) is on the left, while the result — after executing this document via *knit* — is on the right.

There are several parts in the *Rmarkdown* document:

- The title section (called *yaml header*) is between the two '- - -' sections at the start of the document. Here you can give a title, specify the author, include a date, abstract, etc.
- Headings are defined with #, or ##, or ### for first, second and third level headings, respectively.
- Text sections can include many features, for instance, lists are created by using an asterisk ('*') for bullets and numbers such as '1', '2', etc., for numbered lists. Figures, tables, equations can be added as well, and so on.
- The *R*-code is embedded between the ```{r} and ``` symbols. On the top-right position of such a section you will find three symbols (of which two are marked with orange circles in figure 2). Pressing the middle symbol will run all code chunks above, while the right symbol will run the current *R*-chunk. If you move with the cursor to a specific line within the *R*-chunk and press Ctrl+Enter, the current line in the chunk will be executed. If there is any output, it will be displayed in the console.

Finally, the "Knit" button (encircled in blue in the figure) will execute the *Rmarkdown* document and generate the document on the right. You can choose between HTML, PDF, WORD or other types of documents.[1] This final document contains both the *R*-code and the results. It fully documents the analysis.

[1]Note that before you can generate a PDF document you will need to have LaTeX installed. LaTeX is a typesetting program widely used in the scientific community. Search the internet to learn how to install it on your computer. Additionally, especially when working under Windows, you may need to install the *R*-package *tinytex* to make the compilation of your *Rmarkdown* documents into PDF work under *Rstudio*. Read carefully the comments and warnings during the installation of *tinytex*.

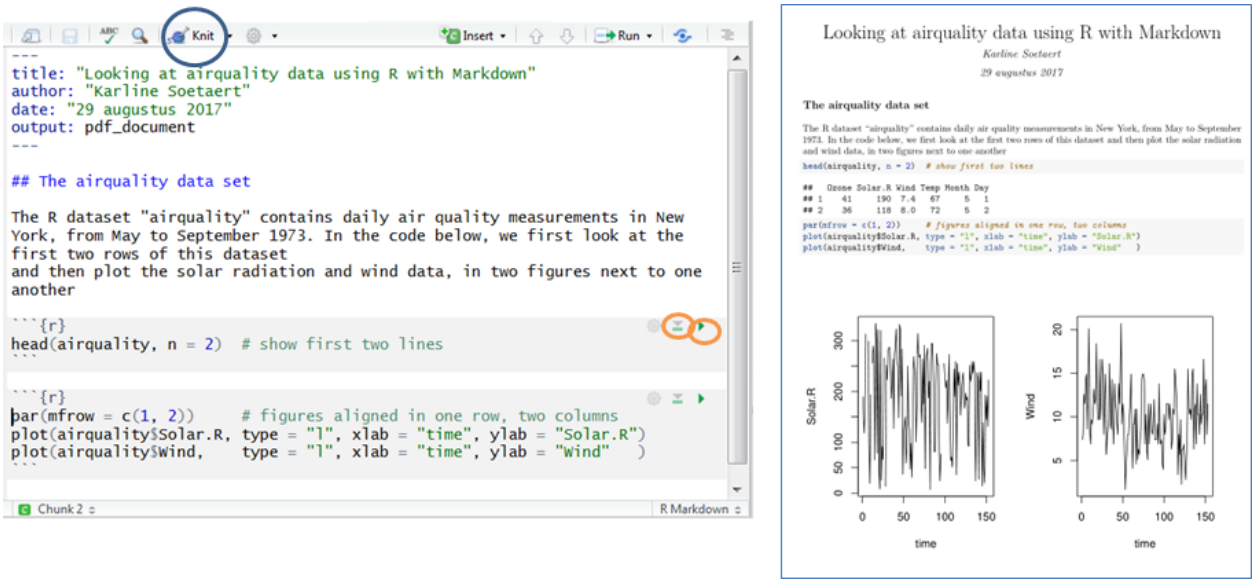


Figure 2: Rmarkdown code (left) and resulting document (right).

Getting started with Rmarkdown

The easiest way to get started is to create a first *R* markdown document using Rstudio: Go to the menu and press "File/new File/R markdown". This will open an *R* markdown file that already contains some *R* code and text. Do this, and look at the contents of this document.

Two useful information sources are accessible from the Rstudio menu: *help/cheatsheets/R markdown Cheat Sheet* and *help/cheatsheets/R markdown Reference Guide*.

Short help on the text format that can be used in markdown documents can be found in *help/Markdown quick reference*.

Tasks

Open the Rmarkdown file called "Rmarkdown_small.Rmd" in Rstudio. The file is available via Blackboard and was used to generate Figure 1.

- Try the different buttons featuring on the top right of the R-code.
- Change the title of the document and make yourself the author.
- Generate a WORD document and a HTML document.[2]
- If you know a bit of *R*, create a second plot that depicts the temperature and the ozone concentration using the same dataset.

[2]HTML is a language used to display webpages.

A quick overview of R and Rstudio

An *R*-code is legible once you realise that:

- <- is the assignment operator (e.g., A <- 1 assigns the value of 1 to variable A).[3]
- everything starting with a hash-sign (#) is considered a comment.
- *R* is case-sensitive: *a* and *A* are two different objects.

Figure 3 shows a screen capture of a typical *Rstudio* session, with the editor (upper left), the R-console (lower left), a help page (upper right) and a plot (lower right). Note the context-sensitive syntax used by the editor (green=comments, blue=reserved words).

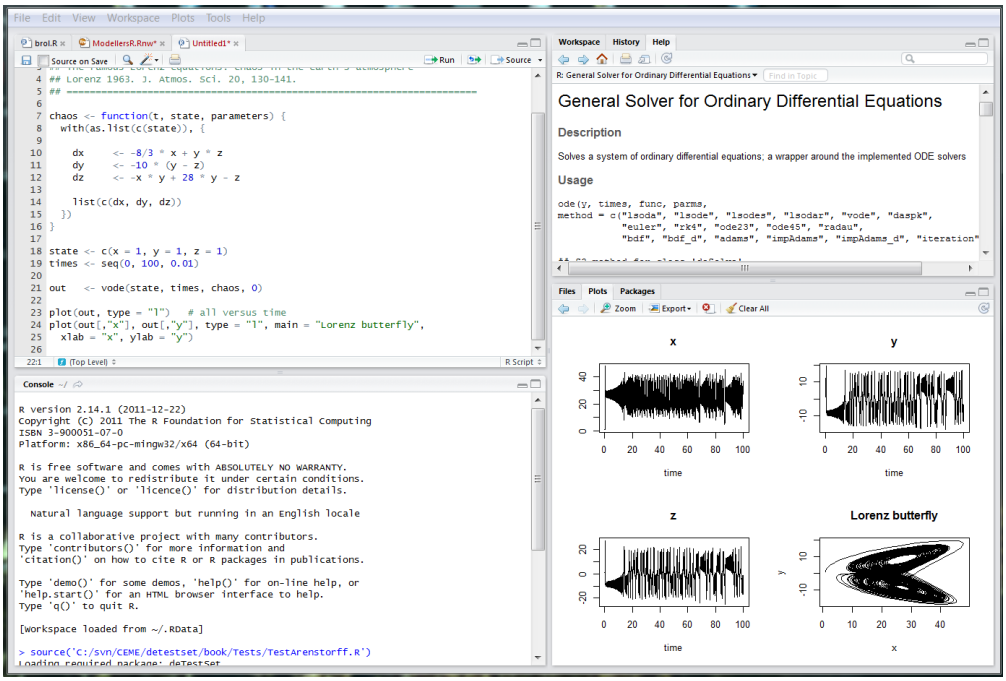


Figure 3: Screen capture of an Rstudio session.

Console versus scripts versus Rmarkdown

There are three ways in which to work with R.

Console

We can type commands into the *R* console window at the command prompt (>) and use *R* as a powerful scientific calculator. For instance, enter in the console window:

```
pi*0.795^2 ; 25*6/sqrt(67) ; log(25) ; log10(25)
```

Here *sqrt*, *log* and *log10* are built-in functions in R, *pi* is a built-in constant, and the semi-colon (;) is used to separate R-commands entered on one line.

[3]Note that within an R-code edited in Rstudio, you can quickly type this operator by pressing a shortcut Alt+minus. This will also include spaces around the operator for increased legibility of the command. This works well for Windows and Linux users. Mac-OS users may need to search the internet for a suitable shortcut.

In the console window, the "UP" and "DOWN" arrow keys can be used to navigate through previously typed commands (command history). Note that typing commands in the console is not the most efficient way of using *R*, especially if you want to debug and save your work for later use (see next).

Scripts

A better way of using *R* is by creating R-scripts in Rstudio's editor and save them in a file (e.g., "MyModel1.R") for later re-use.

R-scripts are sequences of R-commands and expressions. These scripts should be submitted to *R* before they are executed. This can be done in several ways:

- by typing in the R-console window:

```
source ("MyModel1.R")
```

- by opening the file in a text editor, copying the *R*-script to the clipboard (ctrl-C) and pasting it (ctrl-V) into the *R*-console window;
- If you use *Rstudio*, which we recommend, you can either execute the current line (Ctrl+Enter), a section or the entire file (Ctrl+S).

Throughout these notes, the following convention is used:

```
> 3/2
```

denotes an *R*-code (> is the prompt), and

```
[1] 1
```

is an *R* output, as written in the console window.

Rmarkdown

The most integrated way to work with *R* is by using *Rmarkdown*. This is what we will do in the modelling course. This means that you will make, for each modelling project or exercise, a document that contains both the R-code AND the text that provides information on the 'how' and 'why'.

Typically you will use the Rmarkdown document to include everything that you want to show to us, or to keep for later use. You will still use the console to do some quick calculations, or to produce a quick graph, to see whether you are on the right track.

Getting help, examples, demonstrations

R has an extensive help facility. Apart from the Help window launched from the Help menu, it is also available from the command line prompt.

For instance, typing

```
?log
?sin
?sqrt
?round
?Special
```

will explain about logarithms and exponential functions, trigonometric functions, and other functions.

```
?Arithmetic
```

lists the arithmetic operations in R.

```
help.search("steady")
```

will list occurrences of the word "steady" in R-commands.

Sometimes the best help is provided by the very active mailing list. If you have a specific problem, just type *R: your problem* on your search engine. Chances are that someone encountered the problem and it was already solved.

Most of the help files also include examples. You can run all of them by using R-statement *example*.

For instance, typing into the console window:

```
example(pairs)
```

will run all the examples from the pairs help file. (Try this, *pairs* is a very powerful way of visualizing pair-wise relationships.)

Alternatively, you may select one example from the help file, copy it to the clipboard (ctrl-C for windows users) and then paste it (ctrl-V) in the console window. In addition, the *R* main software and many R-packages come with demonstration material. Typing

```
demo()
```

will give a list of available demonstrations in the main software.

```
demo(transport1D)
```

will demonstrate some modelling output generated with package *ReacTran*.

Small things to remember

- Pathnames in *R* are written with forward slashes ("/"). Note that this contrasts to the convention used in Windows, which uses a backslash (\) to separate folders in the paths.

To set a working directory in R, use this command:

```
setwd("C:/R code/")
```

Within *Rstudio*, you can do this via menu *Session/Set Working Directory*.

- If a sentence on one line is syntaxically correct, *R* will execute it, even if the intention was that the sentence proceeds on the next line.

For instance, if we write

```
A <- 3 + cos(pi)
   - sqrt(5)
```

then A will be assigned the value of $3 + cos(\pi)$ and *R* will display the value of $-\sqrt{5}$ (-2.236068) in the console.

In contrast, in the following lines,

```
A <- 3 + cos(pi) -
     sqrt(5)
A
```

R will assign to *A* the value of $3 + cos(\pi) - \sqrt{5}$, as the sentence on the first line was not syntaxically finished. In this case, *R* has (correctly) assumed that the sentence continued on the next line.

Be careful if you want to split a complex statement over several lines! These errors are very difficult to trace, so it is best to avoid this practice.

Exercises—Using R as a calculator

It is very convenient to use *R* as a powerful calculator. This can best be done from within the *R*-console.

Use the console to calculate the value of:

- $(4/6 * 8 - 1)^{2/3}$
- $\ln(20)$
- $log_2(4096)$
- $2 * \pi * 3$
- $\sqrt{2.3^2 + 5.4^2 - 2 * 2.3 * 5.4 * \cos(\pi/8)}$

Hint: you may need to look up some help for some of these functions. Typing

```
?"+"
?log
```

will display help for the common arithmetic operators and about the built-in *R*-function *log*.

R-variables

R calculates as easily with *vectors*, *matrices* and *arrays* as with *single numbers*.

R also includes more complex structures such as *data frames* and *lists*, which allow to combine several types of data.

Learning how to create these variables, how to address them and modify them is essential if you want the make good use of the *R* software.

Numbers, vectors, matrices and arrays

Value assignment

When variables are used, they need to be initialised with numbers. Here is an example.

```
A <- 1
B <- 2
A + B
```

```
## [1] 3
```

R can take as arguments for its functions single numbers, vectors, matrices, or arrays.

```
V <- exp(-0.1)
```

calculates the exponential of -0.1 (e^{-1}). The operator <- assigns the result of this calculation to variable V.

V can then be used in subsequent calculations:

```
log(V)
```

```
## [1] -0.1
```

Note that the assignment of a value to V does not display it.

To display V we simply write:

```
 V
```

```
## [1] 0.9048374
```

Alternatively, we may assign the result of calculations to a variable AND view the results, by embracing the statement between parentheses:

```
 ( X <- sin(3/2*pi) )
```

```
## [1] -1
```

Apart from integers, real and complex numbers, *R* also recognizes infinity (*Inf*) and Not a Number (*NaN*). Try:

```
 1/0
 0/0
 2.3e-8 * 1000
```

Note that *e-8* denotes 10^{-8}.

Vectors

Vectors can be created in many ways; most often we will use:

- The function *c()* combines numbers into a vector[4]
- The operator ":" creates a sequence of values, each by 1 larger (or smaller) than the previous one
- A more general sequence can be generated by R-function *seq*

For instance, the commands

```
c(0, pi/2, pi, 3*pi/2, 2*pi)
seq(from = 0, to = 2*pi, by = pi/2)
seq(0, 2*pi, pi/2)
seq(to = 2*pi, from = 0, by = pi/2)
```

will all create a vector, consisting of: $0, \pi/2, \pi, \ldots, 2\pi$.

Note that R-function *seq* takes as input (amongst others) parameters *from*, *to* and *by* (second line). If the order of these arguments is kept, they do *not* need to be specified by name (third line). But we recommend that you *always* use the names of the input argument if you want to specify their value. In this case, you can change the order of the input arguments as you like (fourth line).

The next command calculates the sine of this vector and outputs the result:

```
sin( seq(0, 2*pi, pi/2) )
```

```
## [1]  0.000000e+00  1.000000e+00  1.224647e-16 -1.000000e+00 -2.449294e-16
```

The next statements

```
 V <- 1:10
 sqrt(V)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

create a sequence of integers between 1 and 10 and take the square root of all of them, displaying the result to the screen. The operator <- assigns the sequence to V.

A peculiar feature of *R* is that the elements of a vector can also be given *names*:

```
( Ocean <- c(total.mass = 1.35e25, volume = 1.34e18, mean.depth = 3690) )
```

```
## total.mass     volume mean.depth
##   1.35e+25   1.34e+18   3.69e+03
```

```
names(Ocean)
```

```
## [1] "total.mass" "volume"     "mean.depth"
```

Matrices

Matrices can also be created in several ways; most often we will use the R-function *matrix*:

The statement:

```
A <- matrix(nrow = 3, data = c(1, 2, 3, 4, 6, 8, 10, 12, 14))
```

creates a *matrix* A, with three rows, and, as there are nine elements, three columns. Note that the *data* are input as a vector (using the c() function), and these values are sorted into the matrix column after column (by default).

```
A
```

[4] This is perhaps THE most important function in R.

```
##      [,1] [,2] [,3]
## [1,]    1    4   10
## [2,]    2    6   12
## [3,]    3    8   14
```

```
sqrt(A)
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.000000 2.000000 3.162278
## [2,] 1.414214 2.449490 3.464102
## [3,] 1.732051 2.828427 3.741657
```

The above statements display the matrix followed by the square root of its elements.

Selecting and extracting elements

To select subsets of vectors or matrices, we can either

- specify the numbers of the elements that we want
- specify the names of the elements that we want
- specify a vector of logical values (TRUE/FALSE) to indicate which elements to include (TRUE) and which not to include (FALSE). This uses logical expressions

Simple indexing

The elements of vectors, matrices and arrays are indexed using the *[]* operator:

To show only the volume of our vector *Ocean*:

```
Ocean["volume"]
```

```
##   volume
## 1.34e+18
```

The following statement takes the elements on the 1^{st} and 3^{rd} row and on the first two columns of matrix A.

```
A[c(1, 3), 1:2]
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    3    8
```

If an index is omitted, then all the rows (1^{st} index omitted) or columns (2^{nd} index omitted) are selected. In the following:

```
A[1:3, ] <- A[1:3, ] * 2
A
```

```
##      [,1] [,2] [,3]
## [1,]    2    8   20
## [2,]    4   12   24
## [3,]    6   16   28
```

the elements on the first three rows of A are multiplied with 2.

Similar selection methods apply to vectors:

```
V[1 : 10]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
V[seq(from = 1, to = 5, by = 2)]
```

```
## [1] 1 3 5
```

Logical expressions

Logical expressions are often used to select elements from vectors and matrices that obey certain criteria.

R distinguishes logical variables *TRUE* and *FALSE*, represented by the integers 1 and 0.[5]

The following will return *TRUE* for values of sequence V that are greater than 1:

```
(V <- seq(from = -2, to = 2, by = 0.5))
```

```
## [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
```

```
 V > 1
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

while

```
V [V > 1]
```

```
## [1] 1.5 2.0
```

will select the values from V that are greater than 1, and

```
V [V > 1] <- 0
```

will set to zero all elements in V that are greater than 1.

Removing elements

When the index is preceded by a "-", the element is removed. For example,

```
 A[ ,-1]
```

```
##      [,1] [,2]
## [1,]    8   20
## [2,]   12   24
## [3,]   16   28
```

will show the contents of matrix *A*, except the first column, while the command

```
V[- V < 0]
```

```
## [1] 0.5 1.0
```

will only show the positive elements of *V*.

More complex data structures

Frequently used data structures that are more complex than vectors or matrices are *data.frames* and *lists*.

[5]Note that variables *T* and *F* are reserved in *R* to represent the logical *TRUE* and *FALSE*, respectively. Therefore, if you want to avoid unexpected consequences, you should *not* use *T* and *F* as variables to which you assign values. For example, it might be tempting to use *T* to denote temperature, or *F* to denote flux. It is recommended that you do *not* do this, but use instead more intuitive names such as *Temp* and *Flux*.

Data-frames

A *data.frame* superficially looks like a *matrix*, but its columns may contain different types of elements (e.g. one column may contain strings, another integers etc. . .).

For instance, the data set *iris* is of class *data.frame*.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

The data set contains strings and numbers. Each column has a name, and can be accessed by its name using the dollar-sign ('$'):

```
mean(iris$Petal.Width)
```

```
## [1] 1.199333
```

Lists

A list is a combination of several objects; each object can be of different length. For instance:

```
 LL <- list(Vector = V, Matrix = A)
```

will combine the previously defined vector V and matrix A in a list called *LL*.

```
names(LL)
```

```
## [1] "Vector" "Matrix"
```

```
LL
```

```
## $Vector
## [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  0.0  0.0
##
## $Matrix
##      [,1] [,2] [,3]
## [1,]    2    8   20
## [2,]    4   12   24
## [3,]    6   16   28
```

Selecting elements from data frames and lists

Lists and data frames can be accessed by their names, or by the square bracket ('[ ]') and double square bracket ('[[ ]]') operators.

Note: The object resulting from a selection using single brackets '[ ]', will be a *data.frame* respectively a *list* itself; with double brackets '[[ ]]', one obtains a *vector* (data.frames) or a variable data-type (lists).

For instance:

```
LL$Vector * 2
```

```
## [1] -4 -3 -2 -1  0  1  2  0  0
```

will multiply all values of element *Vector* with 2.

```
mean(LL[[1]])
```

```
## [1] -0.3888889
```

will calculate the mean of element *Vector*.

Exercises—vectors and sequences

Mean of a vector

- Use *R*-function *mean* to estimate the mean of two numbers, 9 and 17. (You may notice that this is not as simple as you might think! Tip: use the *c()* operator.)

Sediment depth profiles

- For a sediment model, we divide 10 cm of sediment into thin layers, each 1mm thick. These are thin slices, because it is assumed that porosities, concentrations,... remain constant within each layer. We need to know the depth in the centre of each sediment layer. To do so, we need to create a sequence of depth values, extending from 0.05 cm to 9.95 cm, at 1 mm intervals.
  - Create this sequence, put it in a vector called *depth*. Use R-function *seq*.
  - Display this vector.
- Porosity, the volumetric water content of the sediment is often described as an exponentially decreasing function; for instance, following formula generates a typical porosity profile for a deep-sea sediment:

$$porosity = 0.7 + (1 - 0.7) * \exp(-1 * depth)$$

- Calculate the porosity for every value in *depth*. Save the results in a vector called "porosity".
- What is the porosity near the surface (0.05 cm); what is the porosity at 9.95 cm. Put these two values in a vector, called *V*.
- What is the mean porosity in the entire 10 cm?
- What is the mean porosity in the upper cm?
- Now plot porosity versus depth as follows:

```
plot(depth, porosity)
```

Estuarine morphology

- The Westerschelde estuary has a trumpet-shaped morphology, i.e. its cross-sectional area increases in a sigmoidal fashion from Rupelmonde near the river towards Vlissingen near the sea.
- The estuary is 100 km long, and the cross-sectional surface, A, as a function of distance (x), in m^2, can be approximated with the following equation:

$$A(x) = A_r + dA * \frac{x^p}{x^p + {k_s}^p}$$

where $dA = A_s - A_r$, p = 5, ks = 50000 m, $A_r = 4000m^2$, $A_s = 76000m^2$. Here A_r and A_s are the cross-sectional surfaces at the boundary with the *r*iver and the *s*ea respectively.

- The estuary is divided in 200 boxes. Create a sequence of x-values that contain the position in the middle of each box, from the river to the sea.
  - For each box calculate the cross-sectional surface area; put it in a vector called *Area*.

- For modelling purposes, we need the *volume* of each box, rather than the cross-sectional area. Create a vector, called *Volume* that contains the volumes of each box.
- What is the total estuarine volume ? (and what are the units?)
- Plot the cross-sectional surface area versus distance from the river.
- Plot the estuarine volume per box, as a function of the box number.

Plotting observed data

- The following oxygen concentrations were measured, at hourly intervals, starting at 8 o'clock, from the jetty near the NIOZ institute:

(210, 250, 260, 289, 280, 260, 270, 260).

- Make a plot that displays these data. First create a vector containing the hours at which measurements were performed. Then make a vector with the oxygen concentrations.

R functions

One of the strengths of *R* is that one can make user-defined functions that add to *R*'s built-in functions.

Function definition

Typically, complex functions are written in *R* script files or in an R-markdown document, as you will want to use the function several times. For instance,

```
Circlesurface <- function (radius)
    return(pi*radius^2)
```

defines a function (called *Circlesurface*) which takes as input argument a variable called *radius* and which returns the value $\pi * radius^2$ (which is the surface of a circle).

After submitting this function to *R*, we can use it to calculate the surfaces of circles with given radius:

```
Circlesurface(10)
```

```
## [1] 314.1593
```

```
Circlesurface(1:10)
```

```
##  [1]   3.141593  12.566371  28.274334  50.265482  78.539816 113.097336
##  [7] 153.938040 201.061930 254.469005 314.159265
```

the latter statement will calculate the surface of circles with radii 1, 2, . . . , 10.

More complicated functions may return more than one element:

```
Sphere <- function(radius) {
 V  <- 4/3 *pi*radius^3
 S <- 4 *pi*radius^2
 return( list(volume = V, surface = S) )
}
```

Here we recognize

- the function heading (1^{st} line), specifying the name of the function (Sphere) and the input parameter (radius)
- the function specification. As the function comprises multiple statements, the function specification is embraced by curly braces {. . .}.
- The return values (last line). Function *Sphere* will return the volume and surface of a sphere, as a *list*.

The earth has approximate radius 6371 km, so its volume (km3) and surface (km2) are:

```
Sphere(6371)
```

```
## $volume
## [1] 1.083207e+12
##
## $surface
## [1] 510064472
```

The next statement will only display the volume of spheres with radius 1, 2, . . . 5

```
Sphere(1:5)$volume
```

```
## [1]   4.18879  33.51032 113.09734 268.08257 523.59878
```

Sometimes it is convenient to provide default values for the input parameters.

For instance, the next function estimates the density of "standard mean ocean water" (in $kg\ m^{-3}$) as a function of temperature in °C, TC (and for salinity=0 and pressure = 1 atm) (Millero, 1981).

The input parameter TC is by default equal to 20 °C:

```
Rho_W <- function(TC = 20) {
  rho <- 999.842594 + 0.06793952 * TC - 0.00909529 * TC^2 +
         0.0001001685 * TC^3 - 1.120083e-06 * TC^4 + 6.536332e-09 * TC^5
  return(rho)
}
```

Note that, within the function body, we ended the first line with a '+' in order to make clear that the statement is not finished and continues on the next line. It would have been wrong to put the '+' on the next line (and very difficult to trace this error).

Calling the function without specifying temperature (argument T) uses the default value ($TC = 20$):

```
Rho_W()
```

```
## [1] 998.2063
```

```
Rho_W(20)
```

```
## [1] 998.2063
```

```
Rho_W(0)
```

```
## [1] 999.8426
```

```
Rho_W(TC=0)
```

```
## [1] 999.8426
```

Functions in R-packages

For the modelling class you will use many functions from two packages: *marelac* and *ReacTran*.

The marelac package

The R package *marelac* contains many functions useful for the aquatic sciences.

Try:

```
?marelac
```

to see what it contains.

For instance, its function *sw_dens* estimates the density of seawater as a function of salinity (S), temperature(t) and pressure (p) respectively, and using three different funtions (*method*).

To see how it is used type:

```
?sw_dens
```

To estimate the density for a salinity ranging from 0 to 10, and a temperature of 15 dgC, write:

```
library(marelac)
sw_dens(S = 0:10, t = 15)
```

```
##  [1]  999.1026  999.8738 1000.6413 1001.4070 1002.1716 1002.9354 1003.6986
##  [8] 1004.4614 1005.2239 1005.9862 1006.7484
```

The ReacTran package

The *ReacTran* package has been written especially for solving reactive transport models. It contains many funtions to make this type of modelling simple.

For sediment reactive transport modelling for instance, one not only needs a value for the porosity in the *centre* of the boxes, but also at the box *interface*. As this requires quite complicated bookkeeping, R-package *ReacTran* contains two functions that do that for you.

- Function *setup.grid.1D* specifies a 1-Dimensional *grid*, i.e. it will divide a sediment into thin layers, an estuary into 1km thick boxes, a ciliate organism into 1 μm thick concentric spheres, etc.
- Function *setup.prop.1D* calculates a certain *property* on this 1-D grid.

For instance, to subdivide 5 cm of sediment into 10 layers we write:

```
sed <- setup.grid.1D(L = 5, N = 10)
```

This function returns a list, containing many elements that are needed for reactive transport models:

```
sed
```

```
## $x.up
## [1] 0
##
## $x.down
## [1] 5
##
## $x.mid
##  [1] 0.25 0.75 1.25 1.75 2.25 2.75 3.25 3.75 4.25 4.75
##
## $x.int
##  [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
##
## $dx
##  [1] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
##
## $dx.aux
##  [1] 0.25 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.50 0.25
##
## $N
## [1] 10
##
## attr(,"class")
## [1] "grid.1D"
```

It is now simple to define the porosity on this grid: first we define a function that estimates, for a certain sediment depth x the corresponding porosity. Then we use *ReacTran* function *setup.prop.1D* to calculate the porosity on this grid:

```
porfunc <- function (x) return(0.7 + 0.3*exp(-x))
porosity <- setup.prop.1D(func = porfunc, grid = sed)
```

Porosity is now defined, both in the middle of slices and at the slice interfaces:

```
 porosity
```

```
## $mid
##  [1] 0.9336402 0.8417100 0.7859514 0.7521322 0.7316198 0.7191784 0.7116323
##  [8] 0.7070553 0.7042793 0.7025955
##
## $int
##  [1] 1.0000000 0.8819592 0.8103638 0.7669390 0.7406006 0.7246255 0.7149361
##  [8] 0.7090592 0.7054947 0.7033327 0.7020214
##
## attr(,"class")
## [1] "prop.1D"
```

Exercises

R-function to estimate saturated oxygen concentrations

The saturated oxygen concentration in water ($\mu mol\ kg^{-1}$), also called oxygen solubility, can be calculated based on an empirical formula $SatOx = e^A$, where

$$A = -173.9894 + 25559.07/T + 146.4813 \times log_e(T/100) - 22.204 \times T/100 +$$
$$S \times (-0.037362 + 0.016504 \times T/100 - 0.0020564 \times T/100 \times T/100)$$

where T is temperature in Kelvin (Tkelvin = Tcelsius + 273.15) and S is salinity (reported as unitless, but meaning a value in g/kg).

- Make a function that implements this formula; the default values for temperature and salinity are 20°C and 35 respectively.
- What is the saturated oxygen concentration at the default conditions? (A: 225.2346)
- Estimate the saturated oxygen concentration for a range of temperatures from 0 to 30 dgC, and salinity 35.
- Tip: e^A is implemented in R as *exp(A)*.

Molecular diffusion coefficient

Package *marelac* contains a function to calculate molecular diffusion coefficients.

- Estimate the molecular diffusion coefficient for O_2 and CO_2, for salinity = 20 and temperature = 10 dgC. What are the units? Convert to $cm^2 d^{-1}$.

Note: the *diffcoeff* function from *marelac* returns a *data.frame* or a *list*. For plotting, it is easiest to subset this so as to have a vector. So,

```
diffcoeff()$O2
```

will provide the output in a format that is easy to work with.

- Estimate the molecular diffusion coefficient for O_2 and CO_2 for a temperature ranging from 1 to 30. Make a temperature - diffusion coefficient plot for O_2, units of $cm^2 d^{-1}$.
- Add to this plot the temperature-diffusion coefficient relationship for O_2 at salinity = 0. Use the R function *lines* or *points* to add these data.

R-function sphere

Organisms can have many shapes, from spherical to rod-like to amorphous.

In order to create reactive transport models that describe for instance the oxygen concentration in the body of these organisms, we need to estimate the surface area at certain distances from the centre of their body.

Assume a spherical ciliate with a diameter of 100 μm. For modelling purposes, assume that this ciliate consists of concentric spheres, each 1 μm thick.

- What is the area of each of these concentric spheres, in mm^2? Implement as a function.

Porosity profile and estuarine morphology as a function

- Implement the porosity profile of previous exercise as a function that takes as input the sediment depth.
- Implement the estuarine morphology of previous exercise as a function. Return both the cross-section area and volume for each box.

Estuarine morphology using ReacTran

- Use function *setup.grid.1D* to subdivide the estuary in 200 compartments.
- Use function *setup.prop.1D* to calculate the estuarine cross sectional surface on this grid.

Solving ordinary differential equations in R

In the modelling class, you will specify many models as differential equations. Here it is shown how differential equations are implemented and solved in R.

Consider the following set of two ordinary differential equations:

$$\frac{dA}{dt} = r \cdot (x - A) - k \cdot A \cdot B$$

$$\frac{dB}{dt} = r \cdot (y - B) + k \cdot A \cdot B$$

Here, A and B are called the "state variables", $\frac{dA}{dt}$ and $\frac{dB}{dt}$ are the "time derivatives" (also called the rate of change), while r, x, y and k are constant "parameters".

Specifiying the differential equation model

The first step to solving this in *R* is to define the *model function*, which specifies the right-hand side of the differential equations.[6]

This function has three different arguments as input: the actual time (t), the values of the state variables (*state*) and the values of the parameters (*parameters*).

```
model <- function(t, state, parameters) {

with( as.list(c(state, parameters)), {

 dA <- r*(x-A) - k*A*B
 dB <- r*(y-B) + k*A*B
 return (list(c(dA, dB), sum = A+B))

 })
}
```

This function simply calculates the time derivatives of the state variables (*dA* and *dB*) and an output variable called "sum". The derivatives are combined in a vector, and both this vector and the output variable are returned as a list.

The R-statement "with(as.list(c(state,parameters)), {" ensures that the state variables and parameters can be addressed by their names. This statement embraces all other statements in the function—it ends at the line that says "})".

Solving the differential equation model

Before we can actually solve this model, we need to:

- give values to the parameters (*parameters*);
- assign initial conditions to the state variables (*state*);
- generate a sequence of time values at which we want output (*time.seq*).

```
parameters <- c(x = 1, y = 0.1, k = 0.05, r = 0.05)
state      <- c(A = 1, B = 1)
time.seq   <- seq(from = 0, to = 300, by = 1)
```

[6] Note that in *R* we use "*" for multiplication, and not · or × as in the mathematical formalism.

The model can now be solved. To do so, we use the integration routine *ode*, which can be found in R package *deSolve* (Soetaert et al., 2010). This package is loaded first (using function *require*).

```
require(deSolve)
```

The routine *ode* will calculate a value for the state variables A and B at each time value specified in the vector *time.seq*. It does so by numerical integration. The name (ode) hints at the type of differential equations that this function solves, which are *O*rdinary *D*ifferential *E*quations.

The actual numerical solution of our ODE model is obtained within the following single statement:

```
out <- ode(y = state, times = time.seq, func = model, parms = parameters)
```

The output is stored in a matrix, called *out*. All we need to do now is to plot this model output. Before we do so, we can have a look at the first part of the output matrix :

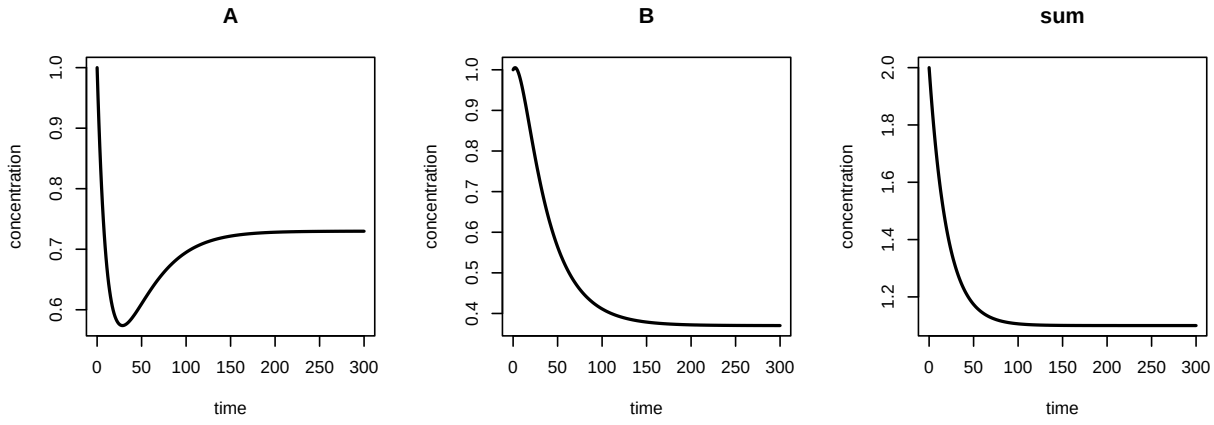
```
head(out)
```

```
##      time         A         B      sum
## [1,]    0 1.0000000 1.0000000 2.000000
## [2,]    1 0.9523189 1.0037869 1.956106
## [3,]    2 0.9090687 1.0052854 1.914354
## [4,]    3 0.8699226 1.0047151 1.874638
## [5,]    4 0.8345728 1.0022854 1.836858
## [6,]    5 0.8027203 0.9982009 1.800921
```

The data are arranged in three columns: first the time, then values of the state variables A and B at each time point, followed by the output variable called "sum".

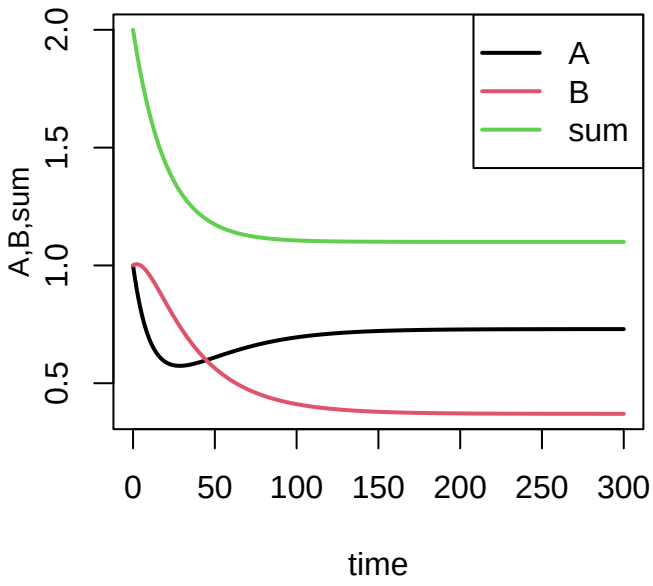
We can plot the output in several ways. The easiest is to plot the entire object at once, either plotting each variable in a separate graph,

```
plot(out, xlab = "time", ylab = "concentration", lwd = 2, type = "l", mfrow=c(1,3))
```



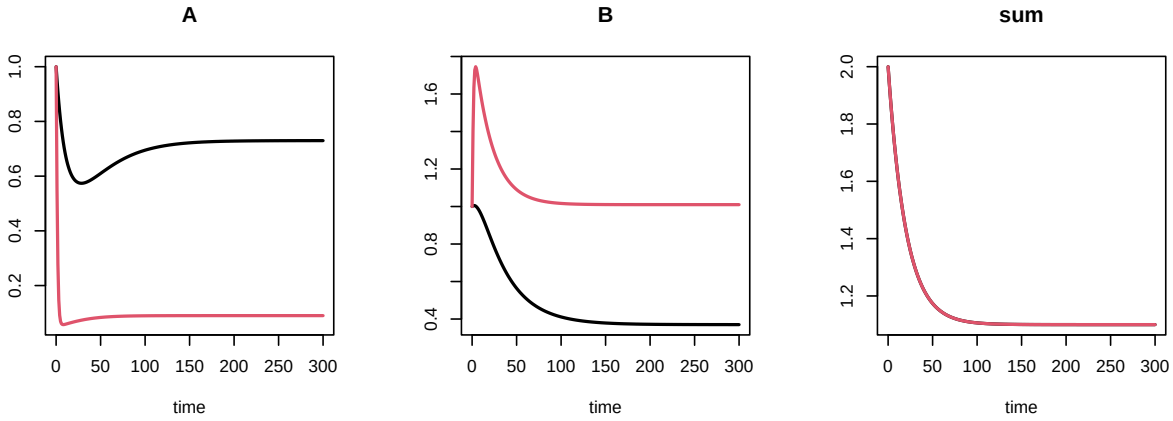
or all variables in one graph.

```
matplot.0D(out, lty = 1, lwd = 2, main=NA)
```



We can run the model with different values of the parameter k, store the output in matrix *out2* and plot the first and second run at the same time. In the code below, we first take a copy of the parameter vector (*parms2*), and then change the parameter named ("k"); we then solve the model, passing the updated parameter vector (*parms = parms2*). We can plot the outcome of the two runs at once (note that by the argument *mfrow = c(1,3)*, we force the output to be in one row and 3 columns - mfrow stands for multiple figures in a row).

```
parms2      <- parameters
parms2["k"] <- 0.5

out2 <- ode(y = state, times = time.seq, func = model, parms = parms2)

plot(out, out2, lty = 1, lwd = 2, mfrow = c(1, 3))
```



For completeness we here list the entire code to solve and plot this differential equation.

```
model <- function(t, state, parameters) {

with (as.list(c(state, parameters)), {

 dA <- r*(x-A) - k*A*B
 dB <- r*(y-B) + k*A*B
 return (list(c(dA, dB), sum = A+B))

 })
}

parameters <- c(x = 1, y = 0.1, k = 0.05, r = 0.05)
state      <- c(A = 1, B = 1)
time.seq   <- seq(from = 0, to = 300, by = 1)

require(deSolve)
out <- ode(y = state, times = time.seq, func = model, parms = parameters)
plot(out, xlab = "time", ylab = "concentration", lwd = 2, type = "l")
```

Steady-state conditions of differential equations.

Sometimes we are interested in the conditions of a differential equation where the state variables do not change anymore. One way to obtain this is to use the function *steady* from the R-package **rootSolve** (Soetaert, 2009):

```
require(rootSolve)
STD  <- steady(y = state, func = model, parms = parameters)
STD$y; STD$sum
```

```
##         A         B
## 0.7298437 0.3701563
```

```
## [1] 1.1
```

Exercises—Solving ordinary differential equations in R

The Lotka-Volterra model

The Lotka-Volterra model is a famous model that either describes predator-prey interactions or competitive interactions between two species. A.J. Lotka and V. Volterra formulated the model almost simultaneously in the 1920's.

- Write a script file that solves the Lotka-Volterra model:

$$\frac{dx}{dt} = a \cdot x \cdot (1 - \frac{x}{K}) - b \cdot x \cdot y$$

$$\frac{dy}{dt} = g \cdot b \cdot x \cdot y - e \cdot y$$

- use for initial values $x = 670$, $y = 610$ and for parameter values: $a = 0.04$, $K = 1000$, $b = 5e\text{-}5$, $g = 0.8$, $e = 0.008$.
- Run the model for 100 days, and call the output *out*.
- Plot the outcome.

- Now run the model with other initial values ($x = 100$, $y = 540$); call the output *out2*. Plot the two runs simultaneously.
- Experiment by running the model for longer intervals (e.g., 1500 days) and by changing the model parameter b in a range between 1e-5 and 10e-5. What do these model predictions tell you about the system?

The Lorenz Butterfly

The Lorenz equations represents the first set of differential equations in which chaotic behaviour was discovered. These three differential equations represent an idealized model for the circulation of air within the atmosphere of the earth.

$$\frac{dx}{dt} = -\frac{8}{3} \cdot x + y \cdot z$$
$$\frac{dy}{dt} = -10 \cdot (y - z)$$
$$\frac{dz}{dt} = -x \cdot y + 28 \cdot y - z$$

- It takes about 10 lines of R-code to generate the solutions and plot them.
- Use as initial conditions $x = y = z = 1$; create output for a time sequence ranging from 0 to 100, and with a time step of 0.005.

References

- R Core Team (2018). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL *https://www.R-project.org/*.
- Soetaert, Karline and Meysman, Filip, 2012. Reactive transport in aquatic ecosystems: Rapid model prototyping in the open source software R Environmental Modelling & Software, 32, 49-60.
- Soetaert K. (2009). rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations. R-package version 1.6
- Karline Soetaert, Thomas Petzoldt, R. Woodrow Setzer (2010). Solving Differential Equations in R: Package deSolve. Journal of Statistical Software, 33(9), 1–25. URL *http://www.jstatsoft.org/v33/i09/* DOI 10.18637/jss.v033.i09
- Karline Soetaert and Thomas Petzoldt (2018). marelac: Tools for Aquatic Sciences. R package version 2.1.7.

Using R as a calculator

Examples of how to use *R* as a powerful calculator. Note the use of spaces around operators such as "+" or "-", or around parentheses. They help to make the code much more legible.

```
(4/6*8-1)^(2/3)
```

```
## [1] 2.657958
```

```
log(20)
```

```
## [1] 2.995732
```

```
log2(4096)
```

```
## [1] 12
```

```
2*pi*3
```

```
## [1] 18.84956
```

```
sqrt( 2.3^2 + 5.4^2 - 2*2.3*5.4 * cos (pi/8) )
```

```
## [1] 3.391288
```

Vectors and sequences

Mean of a vector

```
mean(c(9,17))
```

```
## [1] 13
```

Sediment depth profiles

```
depth <- seq(from = 0.05, to = 9.95, by = 0.1)
depth
```

```
##   [1] 0.05 0.15 0.25 0.35 0.45 0.55 0.65 0.75 0.85 0.95 1.05 1.15 1.25 1.35 1.45
##  [16] 1.55 1.65 1.75 1.85 1.95 2.05 2.15 2.25 2.35 2.45 2.55 2.65 2.75 2.85 2.95
##  [31] 3.05 3.15 3.25 3.35 3.45 3.55 3.65 3.75 3.85 3.95 4.05 4.15 4.25 4.35 4.45
##  [46] 4.55 4.65 4.75 4.85 4.95 5.05 5.15 5.25 5.35 5.45 5.55 5.65 5.75 5.85 5.95
##  [61] 6.05 6.15 6.25 6.35 6.45 6.55 6.65 6.75 6.85 6.95 7.05 7.15 7.25 7.35 7.45
##  [76] 7.55 7.65 7.75 7.85 7.95 8.05 8.15 8.25 8.35 8.45 8.55 8.65 8.75 8.85 8.95
##  [91] 9.05 9.15 9.25 9.35 9.45 9.55 9.65 9.75 9.85 9.95
```

In the following code, note the use of "==". This operator returns a logical value (*TRUE* or *FALSE*) depending on whether or not the value on the left is equal to the value on the right. This logical value is then used within "[]" to select which elements of the vector are to be considered.

```
porosity <- 0.7 + (1-0.7) * exp(-1*depth)
V <- c(porosity[depth == 0.05], porosity[depth == 9.95])
V
```

```
## [1] 0.9853688 0.7000143
```

```
mean(porosity)
```

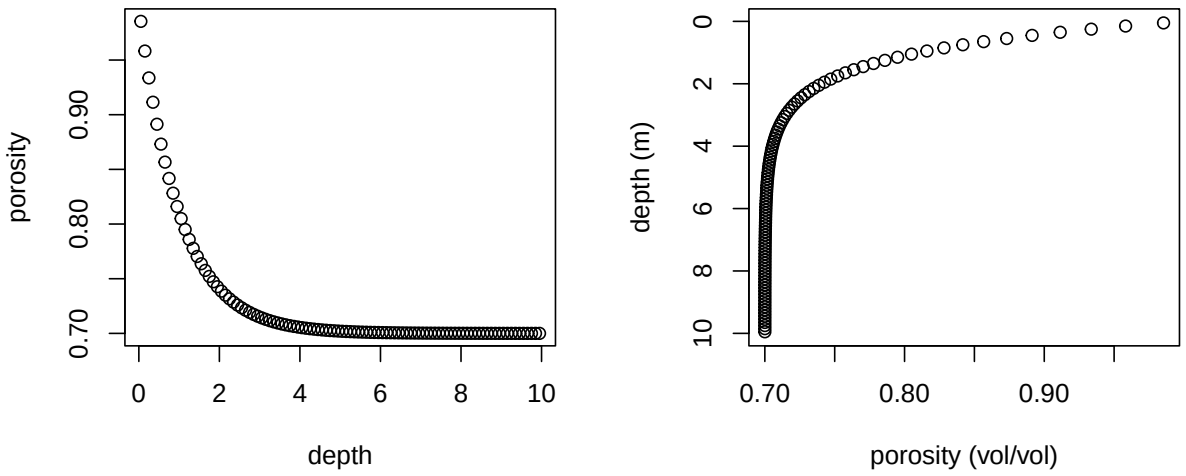
```
## [1] 0.7299861
```

```
mean(porosity[depth <= 1])
```

```
## [1] 0.8895572
```

When plotting a graph, note that by default the first input variable is plotted on the x axis and the second variable is plotted on the y axis. Note that in the second line we change the orientation of the axis by setting the scale in the "opposite" direction, which results in a more natural display of a depth profile.

```
par(mfrow=c(1,2))
plot(depth, porosity)
plot(porosity, depth, ylim=c(10,0), xlab="porosity (vol/vol)", ylab="depth (m)")
```



Estuarine morphology

Note that in the following code, we avoid using constants such as 200 in the line calculating the positions of the middle of the boxes. Instead, we assign this value to a variable (N) and use this variable when necessary and appropriate. Also note that it is not necessary—and in fact rather confusing, as shown on the "confusing line" in the code below—to use parentheses to indicate priority of operators such as "^" and "*". R adheres to priorities that we are used to from algebra: first "^", then "*" or "/", then "+" or "-". Parentheses should, therefore, only be used if you want to *modify* these "natural" priority rules. Otherwise, it's better to avoid them to improve code legibility.

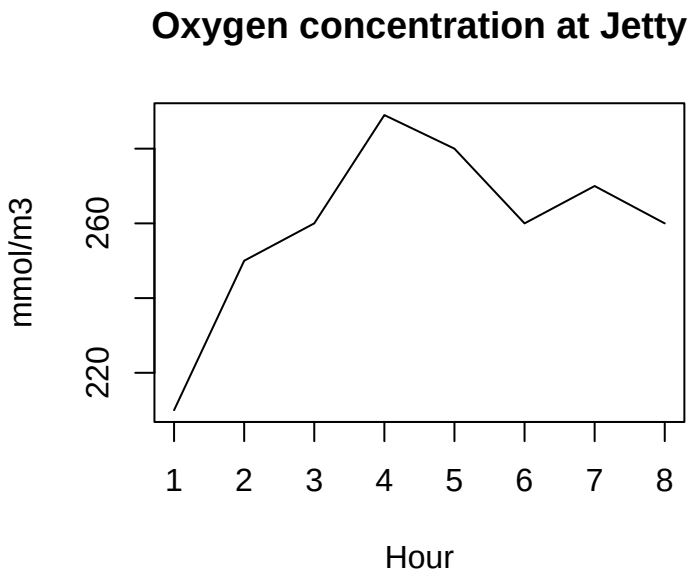
```
L  <- 100000 # metres
N  <- 200
dx <- L/N
x  <- seq(from = dx/2, length.out = N, by = dx)
Ar <- 4000
As <- 76000
p  <- 5
ks <- 50000
Area <- Ar + (As-Ar) * ((x^p)/(x^p+ks^p)) # very confusing, avoid unnecessary ()
```

```
Area <- Ar + (As-Ar) * x^p/(x^p+ks^p) # much easier to read, equal result
Volume <- Area*dx
sum(Volume)  # m3
```

```
## [1] 3807229395
```

Plotting observed data

```
Oxygen <- c(210, 250, 260, 289, 280, 260, 270, 260)
Hour   <- 1:length(Oxygen)
plot(Hour, Oxygen, type = "l", main = "Oxygen concentration at Jetty", ylab = "mmol/m3")
```



R-functions

R-function to estimate saturated oxygen concentrations

```
SatOx <- function(TC = 20, S = 35){
  TK <- TC+273.15 # T in Kelvin
  A <- -173.9894 + 25559.07/TK + 146.4813 * log(TK/100) - 22.204*TK/100 + S *
       (-0.037362 + 0.016504*TK/100 - 0.0020564 * TK/100 * TK/100)
  return(exp(A))
}
SatOx()
```

```
## [1] 225.2346
```

```
SatOx(TC = 0:30)
```

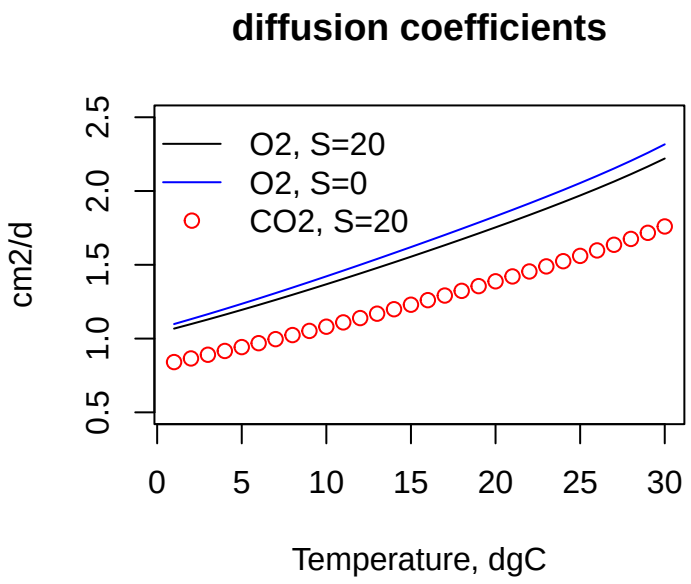
```
##  [1] 349.6542 340.6019 331.9557 323.6924 315.7901 308.2286 300.9890 294.0533
##  [9] 287.4051 281.0288 274.9098 269.0344 263.3897 257.9638 252.7452 247.7235
## [17] 242.8884 238.2306 233.7412 229.4118 225.2346 221.2020 217.3070 213.5431
## [25] 209.9038 206.3833 202.9759 199.6764 196.4796 193.3808 190.3755
```

Molecular diffusion coefficient

```
require(marelac)
DC <- diffcoeff(S=20, t=10, species = c("O2", "CO2"))  #m2/sec
DC*1e4*3600*24                                         #cm2/d
```

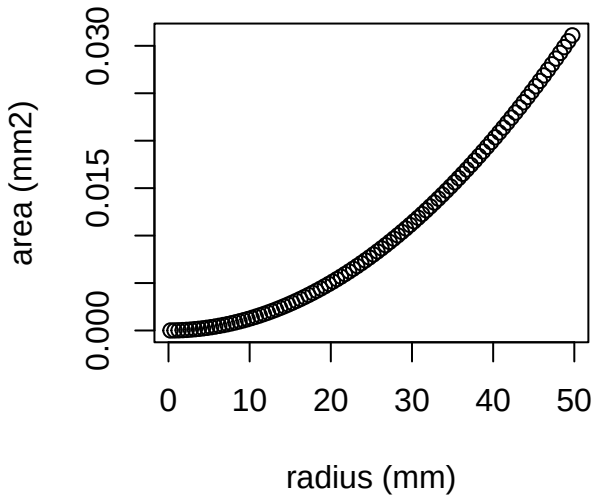
```
##         O2      CO2
## 1 1.368065 1.080572
```

```
t.seq  <- 1:30   # temperature sequence
DC.O2  <- diffcoeff(S=20, t=t.seq)$O2        #m2/sec
DC.CO2 <- diffcoeff(S=20, t=t.seq)[["CO2"]]       #m2/sec (note the alternative to $CO2)
DC.O2.fresh  <- diffcoeff(S=0, t=t.seq)[["O2"]]   #m2/sec

m2_sTOcm2_d <- 1e4*3600*24
# A suitable range for y-axis:
yrange <- c(0.5, 2.5)
plot (t.seq, DC.O2*m2_sTOcm2_d, type = "l", xlab = "Temperature, dgC",
      ylim = yrange, ylab = "cm2/d", main = "diffusion coefficients")
lines(t.seq, DC.O2.fresh*m2_sTOcm2_d, col = "blue")
points(t.seq, DC.CO2*m2_sTOcm2_d, col = "red")
legend("topleft", lty = 1, col = c("black", "blue", "red"), lwd=c(1,1,NA),
       pch=c(NA,NA,1), legend = c("O2, S=20","O2, S=0","CO2, S=20"), bty="n")
```



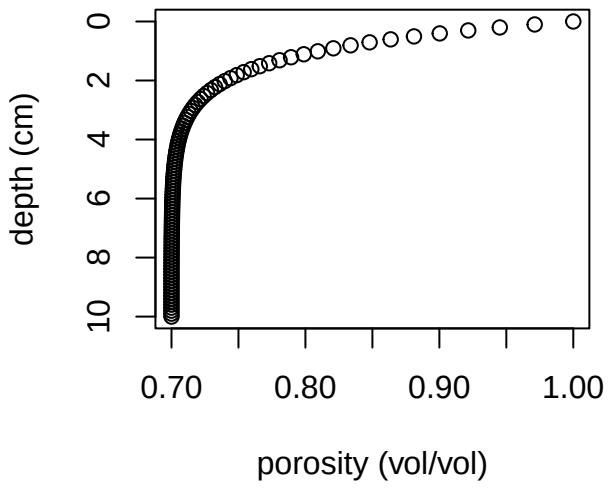
R-function sphere

```
N  <- 100 # number of boxes
L  <- 100/2 # radius, micrometer
dr <- L/N
r  <- seq(from = dr/2, by = dr, length.out = N)
Sphere <- function(r) { return(4*pi*(r/1000)^2) }  # surface area, mm^2
plot(r, Sphere(r), xlab="radius (mm)", ylab = "area (mm2)")
```



Porosity profile and estuarine morphology as a function

```
# porosity as a function of depth
Porfun <- function(depth) {
  return( 0.7 + (1-0.7)*exp(-1*depth) )
}

depth <- seq(from = 0, to = 10, length.out = 100)
plot(Porfun(depth), depth, ylim = c(10,0), xlab = "porosity (vol/vol)", ylab = "depth (cm)")
```



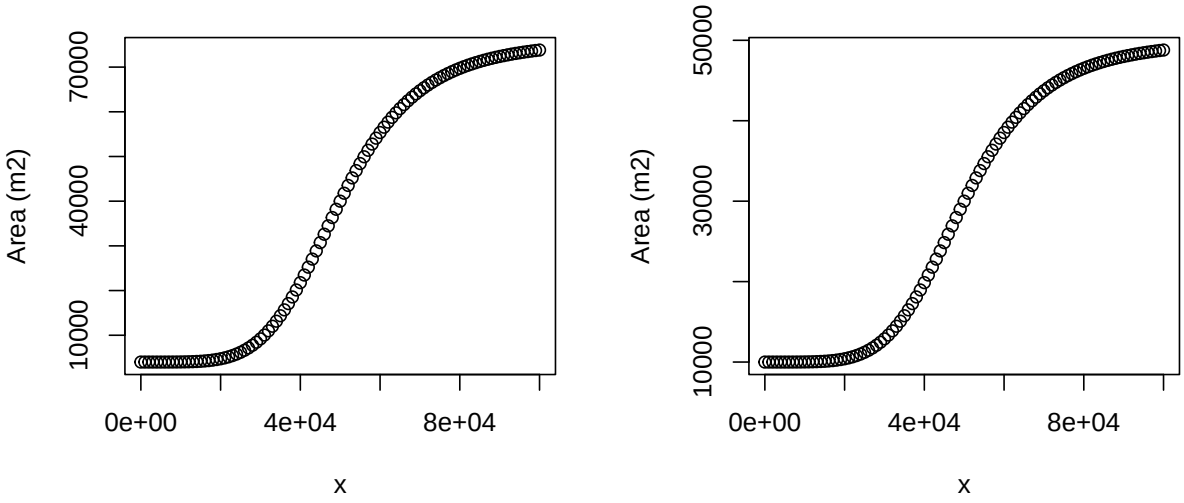

```
# estuarine function, returns the cross-section area and volume as a function of distance
Estfun <- function(x, Ar = 4000, As = 76000, p = 5, ks = 50000, dx = 1000){
  Area <- Ar + (As-Ar) * x^p/(x^p+ks^p)
  Volume <- Area*dx
  return(list(Area = Area, Volume = Volume))
}
dx_new <- 2000
x_new  <- seq(from = dx_new/2, length.out = 50, by = dx_new)
# evaluate for different x and dx input values, the others are kept at default
Estfun(x = x_new, dx = dx_new)
```

```
## $Area
##  [1]  4000.000  4000.056  4000.720  4003.872  4013.602  4037.087  4085.444
##  [8]  4174.536  4325.655  4566.008  4928.838  5453.007  6181.818  7160.853
## [15]  8434.695 10042.576 12013.267 14359.859 17075.406 20130.499 23473.621
## [22] 27034.632 30730.932 34475.210 38183.301 41780.782 45207.419 48419.182
## [29] 51388.104 54100.576 56554.773 58757.782 60722.879 62467.184 64009.793
## [36] 65370.394 66568.298 67621.818 68547.902 69361.963 70077.832 70707.801
## [43] 71262.712 71752.079 72184.216 72566.365 72904.827 73205.076 73471.865
## [50] 73709.318
##
## $Volume
```

```
##  [1]   8000000   8000112   8001440   8007744   8027205   8074174   8170889
##  [8]   8349072   8651311   9132017   9857676  10906014  12363636  14321706
## [15]  16869389  20085152  24026534  28719717  34150813  40260998  46947243
## [22]  54069264  61461864  68950420  76366602  83561564  90414838  96838365
## [29] 102776208 108201153 113109546 117515565 121445759 124934368 128019585
## [36] 130740787 133136596 135243636 137095805 138723926 140155664 141415602
## [43] 142525424 143504159 144368431 145132729 145809654 146410153 146943729
## [50] 147418635
```

Estuarine morphology using ReacTran

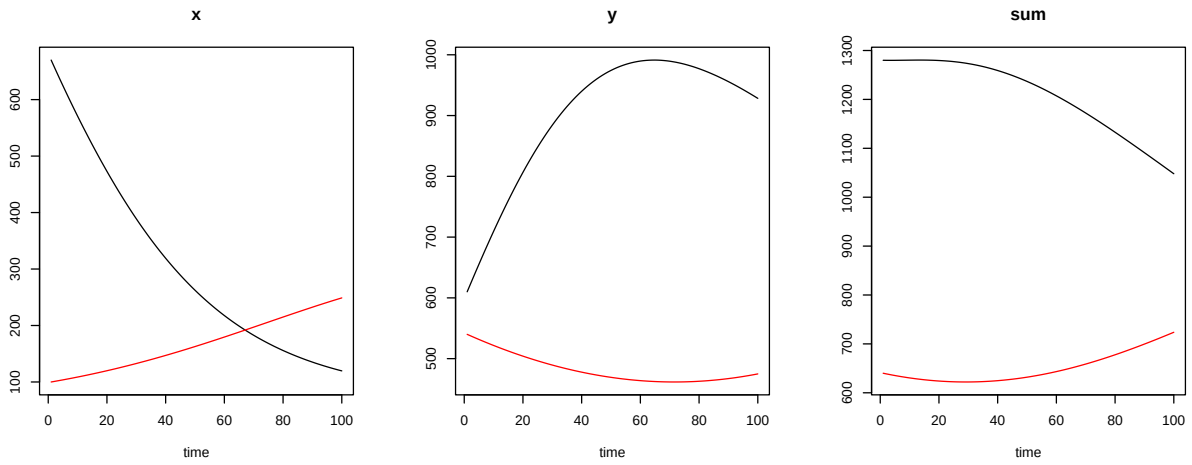
```
Grid <- setup.grid.1D(N = 100, L = 100000)

EstArea <- function(x, Ar = 4000, As = 76000, p = 5, ks = 50000, dx = 1000){
  return( Ar + (As-Ar) * x^p/(x^p+ks^p) )
}

Area <- setup.prop.1D(grid = Grid, func = EstArea)
# example how to pass different input values to the function within setup.prop.1D
Area2 <- setup.prop.1D(grid = Grid, func = EstArea, Ar=10000, As=50000)
par(mfrow=c(1,2))
plot(Area, grid = Grid, ylab = "Area (m2)")
plot(Area2, grid = Grid, ylab = "Area (m2)")
```



Solving differential equations in R

Lotka-Volterra model

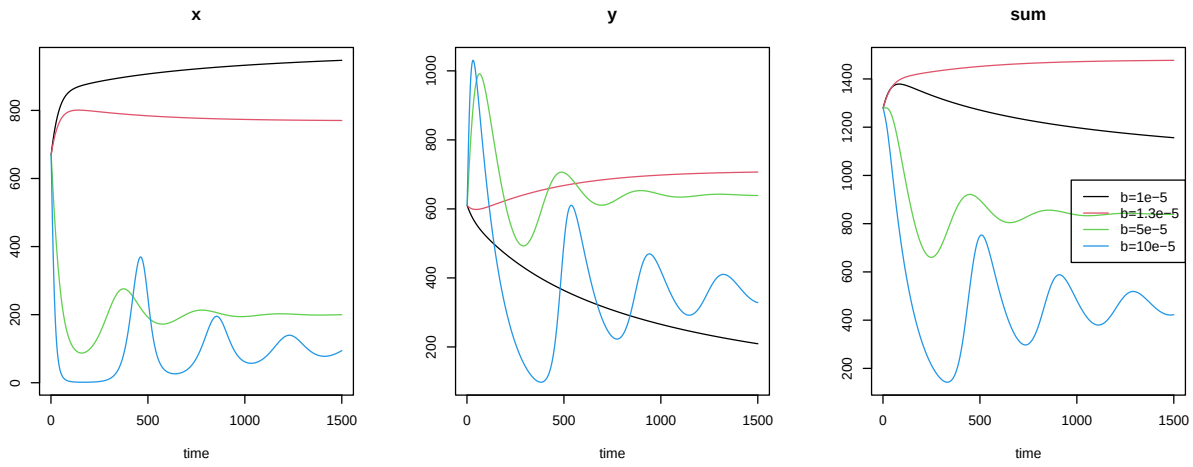
```
LVmodel <- function(t, state, parameters) {
  with (as.list(c(state, parameters)), {

    dx <- a*x*(1-x/K) - b*x*y
    dy <- g*b*x*y - e*y

    return (list(c(dx, dy),
                 sum = x+y))
  })
}

y.ini <- c(x = 670, y = 610)
parms <- c(a = 0.04, K = 1000, b = 5e-5, g = 0.8, e = 0.008)
times <- 1:100
out   <- ode(y = y.ini, func = LVmodel, times = times, parms = parms)
# change initial conditions
y.ini2 <- c(x = 100, y = 540)
out2   <- ode(y = y.ini2, func = LVmodel, times = times, parms = parms)
# plot results in one graph
plot(out, out2, mfrow=c(1,3), col=c("black","red"), lty=c(1,1))
```



Now we experiment a little and explore how the predator-prey dynamics look like at longer time scales and for different values of the parameter "b".

```
times <- 1:1500
parms1 <- parms
parms1["b"] <- 1e-5
out1  <- ode(y = y.ini, func = LVmodel, times = times, parms = parms1)
parms2 <- parms
parms2["b"] <- 1.3e-5
out2  <- ode(y = y.ini, func = LVmodel, times = times, parms = parms2)
parms3 <- parms
parms3["b"] <- 5e-5
```

```
out3  <- ode(y = y.ini, func = LVmodel, times = times, parms = parms3)
parms4 <- parms
parms4["b"] <- 10e-5
out4  <- ode(y = y.ini, func = LVmodel, times = times, parms = parms4)
plot(out1,out2,out3,out4, mfrow=c(1,3), col=c(1,2,3,4), lty=1)
legend("right",legend=c("b=1e-5","b=1.3e-5","b=5e-5","b=10e-5"), col=c(1,2,3,4), lty=1)
```



Lorenz model

```
Lorenz <- function(t, state, parameters) {
  with (as.list(c(state, parameters)), {
    dx <- -8/3*x + y*z
    dy <- -10*(y-z)
    dz <- -x*y + 28*y - z

    return (list(c(dx, dy, dz),
                 sum = x+y+z))
  })
}

parameters <-  NULL
state      <- c(x = 1, y = 1, z = 1)
time.seq   <- seq(from = 0, to = 100, by = 0.005)

#output
out <- ode(y = state, times = time.seq, func = Lorenz, parms=parameters )
head(out)
```

```
##       time         x        y        z      sum
## [1,] 0.000 1.0000000 1.000000 1.000000 3.000000
## [2,] 0.005 0.9920511 1.003193 1.129840 3.125085
## [3,] 0.010 0.9848912 1.012567 1.259918 3.257376
## [4,] 0.015 0.9785610 1.027850 1.391047 3.397458
## [5,] 0.020 0.9731147 1.048823 1.523999 3.545937
## [6,] 0.025 0.9686202 1.075317 1.659515 3.703453
```

```
plot(out, xlab = "time", lwd = 2)
```