Introduction to R and R Markdown for Reaction-Transport Modellers

Teaching material accompanying the course
Reaction Transport Modelling in the Hydrosphere

Karline Soetaert and Lubos Polerecky, Utrecht University

July 2021

Abstract

This short reader introduces you to the programming tools that we will use in the *Reaction-Transport Modelling* class. It will show you how to 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 *R Markdown*, and the creation of the documents is done by *knitr*. We recommend that you recreate the examples provided in this document and then solve the exercises. In this way you will start gradually building your knowledge of R and *R Markdown*, which will help you understand more efficiently the material covered in the course.

Contents

1 If you are a total beginner and need to get started from scratch

You will need to install couple of programs on your computer to be able to do all the learning, modelling and reporting work in the course. The software we will use is free of charge and used by millions of people all around the world. In this section we will explain step by step how to get started with the installation to make sure that everything works on your computer as it should.

We assume here that you use Microsoft Windows. The sequence of steps for Linux or Mac-OS users is the same, although the actual execution of the steps may be different.

1.1 Core R software

R is the programming language that we will use in the course. It can be downloaded from http://www.r-project.org/. Click on the *download R* link, choose a mirror, and download the precompiled binary distribution for your system. On this website, you will also find useful documentation.

First, install the *base* R (e.g., installation file `R-4.1.0-win.exe` or some later version). Subsequently, install *Rtools* (e.g., installation file `rtools40v2-x86_64.exe`). The later step will allow you to compile R-packages on your computer, which will be quite important later on. Ensure that you complete the instructions described in the section *Putting Rtools on the PATH* after you have installed *Rtools*.

1.2 Rstudio

To execute R code in a nice and comprehensive environment, we will use *Rstudio*. Go to http://rstudio.org, choose *Download*, choose *Rstudio Desktop Free*, download the installation package for your system (e.g., `RStudio-1.4.1717.exe`), and install it.

Open *Rstudio* and test whether you can compile an R-package from source, as described in the instructions for installing *Rtools* (see above).

1.3 R packages

Many of the useful functions are not part of the R base program, but are available as *R packages*. They are not necessarily made by the R core-development team, but are developed and shared with the R community by experts in other fields, e.g., in the field of modelling.

To install an R-package in *Rstudio*, click on the *Packages* tab (lower-right window), click on *install*, type the package name, make sure you have checked *install dependencies*, and click *install*. When asked about compilation for packages that need to be installed from sources, click *yes*. Be patient, the package compilation step may take a while.

In this course, we will use several packages. We recommend that you install them in this order:

- `learnr` — This package will allow you to view *tutorials*, which are an essential part of the course. More information about tutorials is provided later in this document (section RTM package).
- `deSolve` (Soetaert et al., 2010) — This package contains methods to perform numerical integration. Thus, it allows you to solve 0-, 1- and 2-dimensional differential equations, which will be dealt with in the first part of the course.
- `ReacTran` (Soetaert and Meysman, 2012) — This package allows you to solve differential equations describing reaction-transport models in 1D and 2D. This is the core R-package that we will use in the second part of the course.
  - When installing `ReacTran`, the package `rootSolve` (Soetaert, 2009) will also be installed. It is a package that allows you to find the root of functions.
- `marelac` (Soetaert and Petzoldt, 2018) — This package contains functions and constants from the aquatic sciences.
- `devtools` — This package allows you to develop R packages. We will use it to install the R package `RTM` from the github repository.
- `RTM` — This package contains much of the teaching material for the course. This package is currently available from a Github repository `dynamic-R/RTM` (https://github.com/dynamic-R/RTM). You can install it by typing the following command in the R-console:

  ```
  devtools::install_github("dynamic-R/RTM", depend=TRUE)
  ```

  More information about the RTM package is provided at the end of this document (section *Learning reaction-transport modelling using the package RTM*).

1.4 LaTeX distribution

We will use the typesetting program LaTeX to produce beautifully typeset documents. This program is extremely powerful and versatile, and it is widely used especially by people who want to write equations in their documents. In this course we will use it to convert documents written in R Markdown into PDF. To install LaTeX, go to https://miktex.org/, choose *Download*, download the installation file (e.g., `basic-miktex-21.6-x64.exe`), and install it. During the installation process, select *yes* when asked *Install missing packages on-the-fly*. It is handy to check for updates just after the installation has finished, to be sure that all LaTeX packages are up-to-date.

Restart *Rstudio* if it was running before LaTeX installation.

1.5 Check whether everything is working

To check whether everything is working at this point, open *Rstudio*, and create a new R Markdown file. This is done from the menu by selecting File → New file → R Markdown → Document. Type the title, your name

in the author box, check the `PDF` checkbox, and click *Ok*. If everything goes as expected, your *Rstudio* session should look similar as shown in Figure 1.

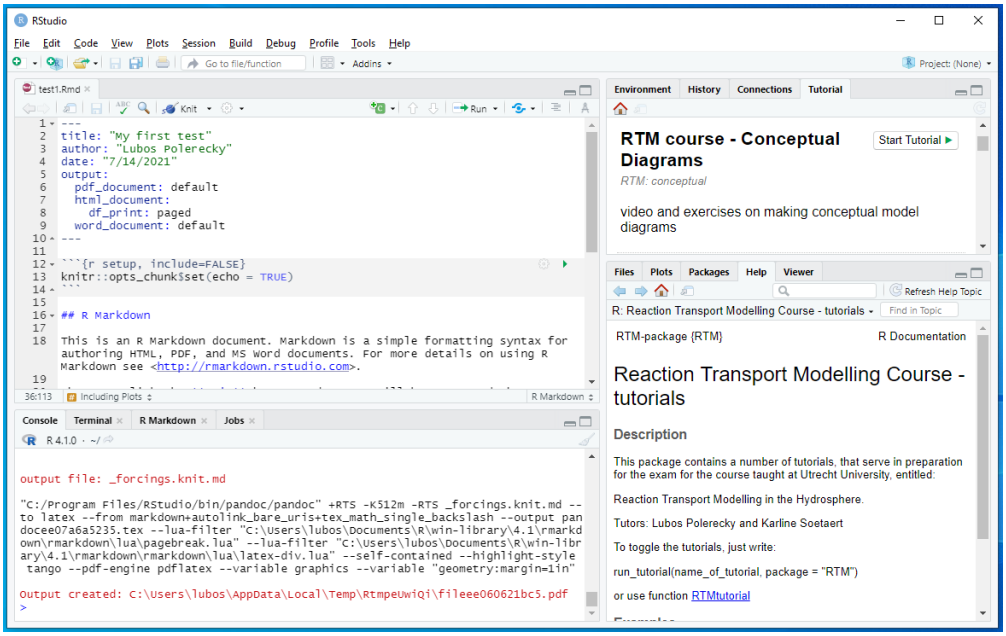


Figure 1: Screenshot of Rstudio with the default Rmd file.

The top-left panel contains the editor, where you will write your text and R-code. In the top-right panel, you can view several things, including the "Environment" (which shows values of your R-variables) and "Tutorials". The bottom-left panel contains the R-console, where you can type R-commands, and the R Markdown console, where you can see the progress of your Rmd file compilation (see next). Finally, the bottom-right panel allows you to manage your R-packages, view help files, or see plots.

Once you create a new Rmd file, it is best to first save it under a name of your choice (e.g., `test1.Rmd`). Then, after you are satisfied with its content, click on the *Knit* button in the top part of the editor panel (useful short-cut: press `Ctrl+Shift+K`). This will start a compilation process, which will convert the Rmd file into a nicely typeset document in one of three possible formats: PDF, DOC, or HTML (Figure 2).

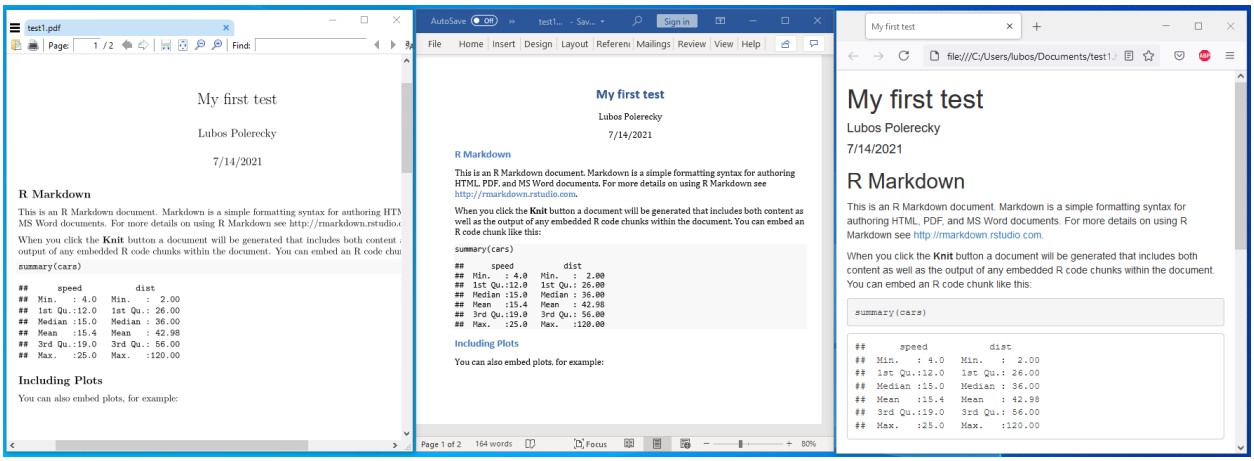


Figure 2: Nicely formatted output produced from the Rmd file. Left: PDF, viewed in SumatraPDF. Middle: DOC, viewed in Microsoft Word. Right: HTML, viewed in Firefox.

Note that if you are converting the Rmd file into PDF for the very first time, the compilation may take a while. This is because *MikTeX* is installing extra LaTeX packages, which are likely missing on your computer as they have not been installed by default during the installation process.[1] Be patient and wait until the compilation process is finished.[2] If everything goes well, you should find a file `test1.pdf` in the same folder as the `test1.Rmd`. You can open it in any PDF reader.[3] After the lengthy first-time compilation, the subsequent compilations will only take a few seconds.

2 Scientific programming practice

Scientific research often involves computation, e.g., to convert numbers to different units, to calculate derived quantities, to perform statistical analyses on data, or to create and solve models. More and more of 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 a 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), but it will also give you a head-start if you want to use these codes in the future. In addition, a well-documented code is an essential step to reproducible and reusable science.

2.1 The 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.

2.2 Adding structure and comments in the code

Legibility of the code can easily be increased by adding a *structure* to the code: alignment of comparable sections, use of spaces, etc.

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

The following code is self explanatory. 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.

[1] This assumes that you have selected *yes* when asked *Install missing packages on-the-fly* during *MikTeX* installation. If you forgot to do that, you need to stop the *knitting* process by clicking the red stop button, and go to the folder of your `test1.Rmd` file. There should be a file called `test1.tex`. Open it in the native TeX editor that comes with the *MikTeX* distribution (most likely `TeXworks`), and compile it with pdfLaTeX from there. In this process, you should be prompted to install missing LaTeX packages.

[2] At the time of writing this reader, it took about two minutes until the first-time *knitting* and LaTeX compilation were complete.

[3] Note that *Acrobat reader*, which is a commonly used PDF reader, may not be the best choice. This is because the PDF file is *locked* for writing when open by *Acrobat reader*. This means that if you want to re-knit your Rmd file to a PDF file, you may end up with a writing error rather than an updated PDF file. To avoid this, you will need to close the PDF file in *Acrobat reader* before clicking the *knit* button in *Rstudio*. We recommend a better solution: use a PDF viewer that does *not* lock the viewed PDF file. One such program is called *SumatraPDF*, which can be downloaded from http://www.sumatrapdfreader.org.

```r
# 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 of the dataset
```

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

```r
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"   )
```

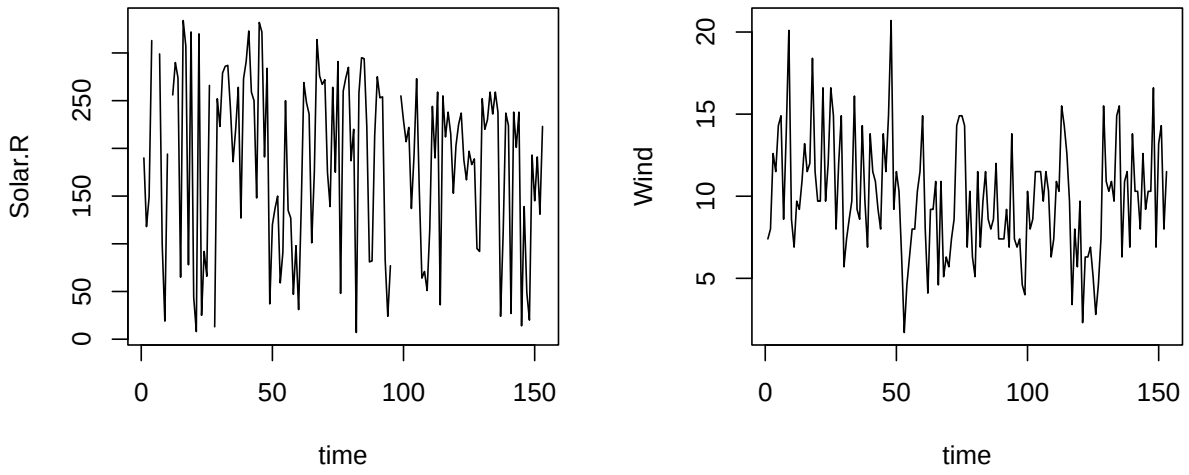


Figure 3: Example of a plot generated by an R-code.

3 R Markdown — creating fully integrated documents

Even better is to have the text, the code and the output in one document. This is what *R Markdown* does. When the final document is "created", the R-code is executed and the results are merged in the document. Figure 4 illustrates how this works in *Rstudio*.

There are several parts in the *R Markdown* 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.

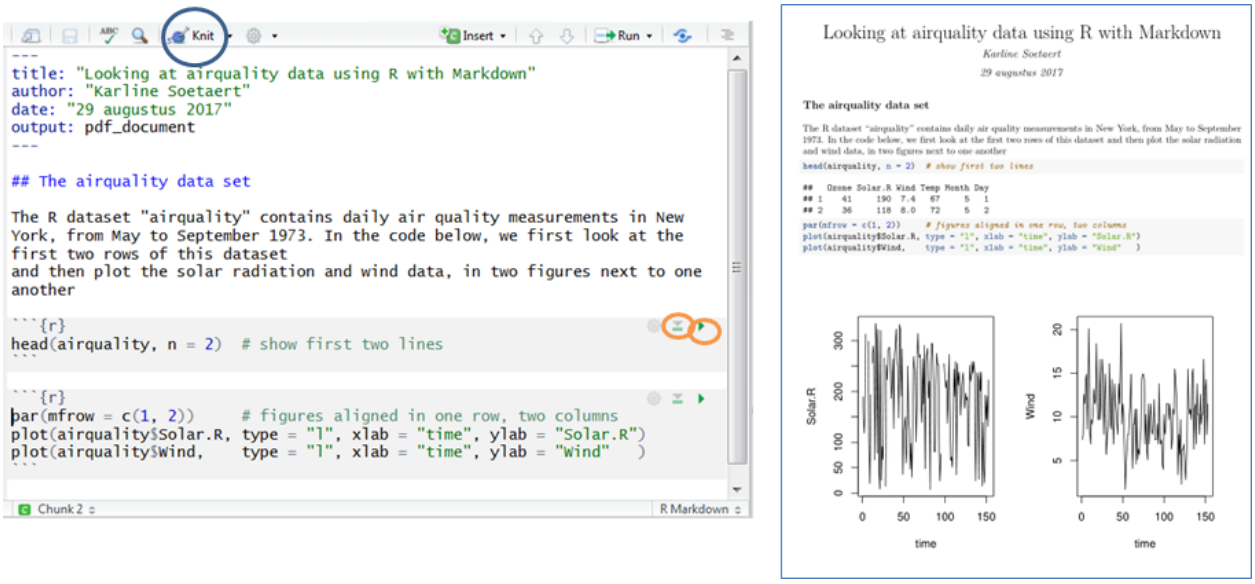


Figure 4: Screenshot of *Rstudio* showing an R Markdown code (left) and the resulting document produced via *knit* (right).

- The R-code is embedded between the ‘‘‘{r} and ‘‘‘ symbols. These are so-called code-chunks. In the top-right position of each code-chunk you will find three symbols (two of them are marked with an orange circle in Figure 4). Pressing the middle symbol will run all code-chunks above, while the right symbol will run the current code-chunk. If you move with the cursor to a specific line within the code-chunk and press `Ctrl+Enter`, the code on that line will be executed. If there is any output, it will be displayed in the console.

Finally, the *Knit* button (encircled in blue in Figure 4) will execute the *R Markdown* document and generate the document on the right. This final document contains both the R-code and the results. It fully documents the analysis.

3.1 Getting started with R Markdown

The easiest way to get started is to create a first *R Markdown* document using *Rstudio*. From the menu choose 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*.

3.2 Exercises — Creating and exporting an Rmd document

Open the *R Markdown* file called `Rmarkdown_small.Rmd` in *Rstudio*. The file is available via Blackboard and was used to generate Figure 4.

- 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, HTML and PDF document.

- If you know a bit of R, create a second plot that depicts the temperature and the ozone concentration using the same dataset.

4 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 the variable A).[4]
- everything starting with a hash-sign (`#`) is considered a comment.
- R is case-sensitive: a and A are two different objects.

4.1 Console versus R-scripts versus R Markdown

There are three ways in which to work with R.

4.1.1 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.

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).

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.

4.1.2 R-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:

```
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;

[4] 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.

- if you use *Rstudio*, which we recommend, you can either execute the current line (Ctrl+Enter), a section, or the entire file (Ctrl+S).

4.1.3 R Markdown

The most integrated way to work with R is by using *R Markdown*. 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.

4.2 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*.

4.3 Small things to remember

4.3.1 Slash vs. backslash

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*.

4.3.2 Finished vs. unfinished R-commands

If an R-command on one line is syntactically correct, R will execute it, even if the intention was that the command 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 command on the first line was not syntactically finished. In this case, R has (correctly) assumed that the command 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.

4.4 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:

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

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*.

5 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.

5.1 Numbers, vectors, matrices and arrays

5.1.1 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 in the console:

```
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}.

5.1.2 Vectors

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

- The function `c()` combines numbers into a vector.[5]
- 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"
```

This will be a very useful feature in our models.

5.1.3 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).

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

```
A
```

```
##      [,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.

5.2 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.

5.2.1 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
```

5.2.2 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.[6]

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.

5.3 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.

[6] 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*.

5.4 More complex data structures

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

5.4.1 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
```

5.4.2 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
```

5.4.3 Selecting elements from data frames and lists

Lists and data frames can be accessed by their names (using `$`), or by the square bracket (`[ ]`) or 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*.

5.5 Exercises — Vectors and sequences

5.5.1 Mean of a vector

- Use the 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()` function.)

5.5.2 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, etc., 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 the 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:

$$porosity = 0.7 + (1 - 0.7) \times e^{-depth}$$

  - 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)
```

5.5.3 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 (in m^2), as a function of the distance, x (in m), can be approximated with the following expression:

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

  where $dA = A_s - A_r$, p = 5, ks = 50000 m, $A_r = 4000\ m^2$, $A_s = 76000\ m^2$. Here A_r and A_s are the cross-sectional surfaces at the boundary with the river and the sea, 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.

5.5.4 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.

6 R-functions

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

6.1 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 an input argument a variable called *radius* and 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 a 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 an approximate radius of 6371 km, so its volume (km^3) and surface (km^2) 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 the temperature (input argument TC) 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
```

6.2 Functions in R-packages

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

6.2.1 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), and using three different functions (*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 °C, 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
```

6.2.2 The ReacTran package

The *ReacTran* package has been written specifically for solving reaction-transport models. It contains many functions to make this type of modelling simple.

For reaction-transport modelling in sediments, 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 a complicated book-keeping, the R-package *ReacTran* contains two functions that do that for you.

- Function *setup.grid.1D* specifies a 1-Dimensional *grid*, i.e., it divides a sediment column into thin layers, an estuary into 1 km 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 a 5 cm thick sediment column into 10 layers, we write:

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

This function returns a list that contains many elements that are needed in reaction-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 the *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"
```

6.3 Exercises — R-functions

6.3.1 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

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

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 °C, and salinity 35.
- Tip: e^A is implemented in R as `exp(A)`.

6.3.2 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 °C. What are the units? Convert to $cm^2\ d^{-1}$.

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

```
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 °C. Make a temperature vs. diffusion coefficient plot for O_2 (in units of $cm^2\ d^{-1}$).
- Add to this plot the temperature vs. diffusion coefficient relationship for O_2 at salinity = 0. Use the R function *lines* or *points* to add these data.

6.3.3 R-function sphere

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

In order to create reaction-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 (in mm^2) of each of these concentric spheres? Implement as an R-function.

6.3.4 Porosity profile and estuarine morphology as a function

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

6.3.5 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.

7 Solving ordinary differential equations in R

Throughout the modelling course, you will specify many models in the form of 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 *rates of change*), while r, x, y and k are constant parameters.

7.1 Specifying the differential equation model

The first step to solving this set of differential equations in R is to define the *model function*, which specifies the right-hand side of the differential equations.[7]

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

 })  # end of the *with* function
}
```

This function simply calculates the time derivatives of the state variables (`dA.dt` and `dB.dt`) 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 `})`.

7.2 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 the output (*time.seq*).

```
parameters <- c(x = 1, y = 0.1, k = 0.05, r = 0.05)
state      <- c(A = 1, B = 1)  # the same order as dA.dt and dB.dt in the model function!!
time.seq   <- seq(from = 0, to = 300, by = 1)
```

The model can now be solved. To do so, we use the integration routine *ode*, which can be found in the 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 Ordinary Differential Equations.

The actual numerical solution of our ODE model is obtained by the following single R-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 top few rows of the output matrix:

```
head(out)
```

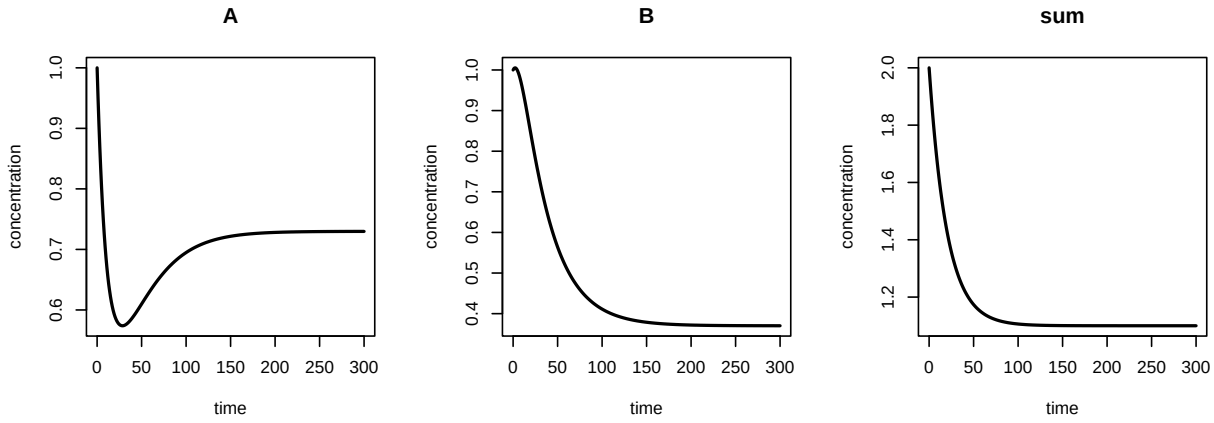
[7] Note that in R we use "*" for multiplication, and not $\cdot$ or $\times$ as in the mathematical formalism.

```
##      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*.

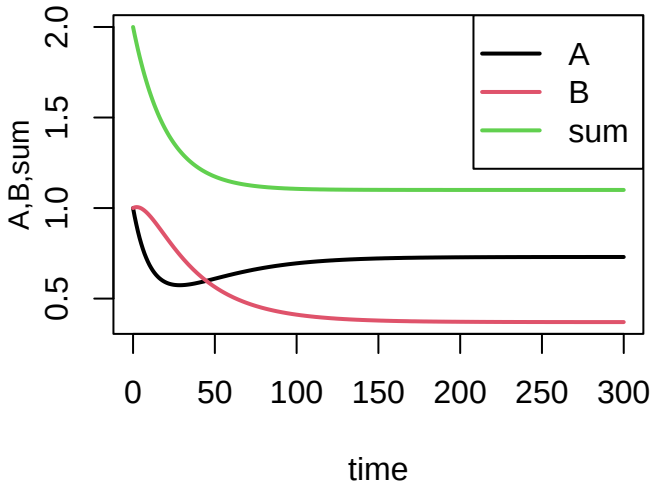
We can plot the output in several ways. The easiest is to plot the entire object at once, either plotting each variable in a separate graph,

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



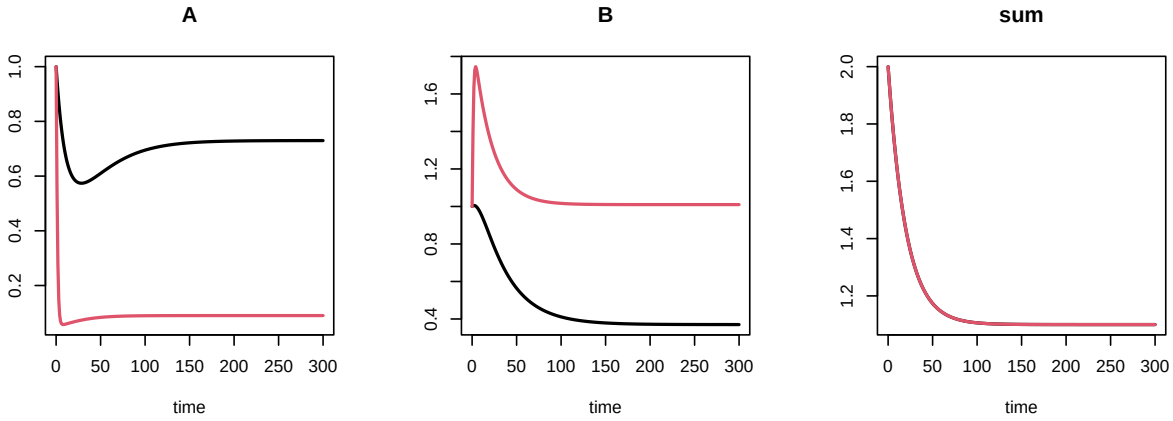
or all variables in one graph.

```
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 a 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 R-code that solves and plots the above set of differential equations.

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

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

 dA.dt <- r*(x-A) - k*A*B
 dB.dt <- r*(y-B) + k*A*B
 return (list(c(dA.dt, dB.dt), 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")
```

7.3 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
```

7.4 Exercises — Solving ordinary differential equations in R

7.4.1 The Lotka-Volterra model

The Lotka-Volterra model is a famous model that 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 an R-script that solves the Lotka-Volterra model

$$\frac{dX}{dt} = a \cdot X \cdot (1 - \frac{X}{K}) - b \cdot X \cdot Y$$

$$\frac{dY}{dt} = g \cdot b \cdot X \cdot Y - e \cdot Y$$

  for the initial values of $X = 670$ and $Y = 610$, and for the parameter values of $a = 0.04$, $K = 1000$, $b = 5 \times 10^{-5}$, $g = 0.8$, and $e = 0.008$.
- Run the model for 100 days, and store the output in a variable *out*.
- Plot the outcome.
- Now run the model with other initial values ($X = 100$, $Y = 540$); store the output in the variable *out2*. Plot the two model 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 1×10^{-5} and 10×10^{-5}. What do these model predictions tell you about the system?

7.4.2 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 an R-code to generate the solutions and plot them. Try it out.
- Use the 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.

8 Learning reaction-transport modelling using the package RTM

Now that you have successfully learned the basics of R, you can proceed by studying other material prepared for the reaction-transport modelling course. This material is put together in the R-package *RTM*. You can install this package in *Rstudio* by typing in the R-console:

```
devtools::install_github("dynamic-R/RTM", depend=TRUE)
```

The RTM package contains *tutorials*, *exercises* and extra *readers*. This division reflects the main course activities, as described in the following.

8.1 RTM tutorials

The *tutorials* contain short videos about a specific modelling topic. The topics range from basic principles (e.g., conceptual diagrams, mass balance equations) to more advanced (e.g., rate laws) and very advanced topics (e.g., pH modelling, reaction-transport modelling in porous media). The tutorials also contain simple tasks and exercises, which allow you to test your knowledge gained from the videos and prepare you for the exam.

We recommend that you study the tutorials in advance, i.e., *before the class*. You try to learn as much as possible by watching the short videos (once, but preferably multiple times), consult the textbook, and solve the tasks and exercises within the individual tutorials. In the process, you note questions about topics that you have not fully understood. You will be able to consult these questions with your peers and with the lecturers during the class.

If you installed the *RTM* package, you will find the tutorials in the *Tutorial* tab of the *Rstudio* program (Figure 5). Run the *tutorial* by clicking the *Start Tutorial* button.

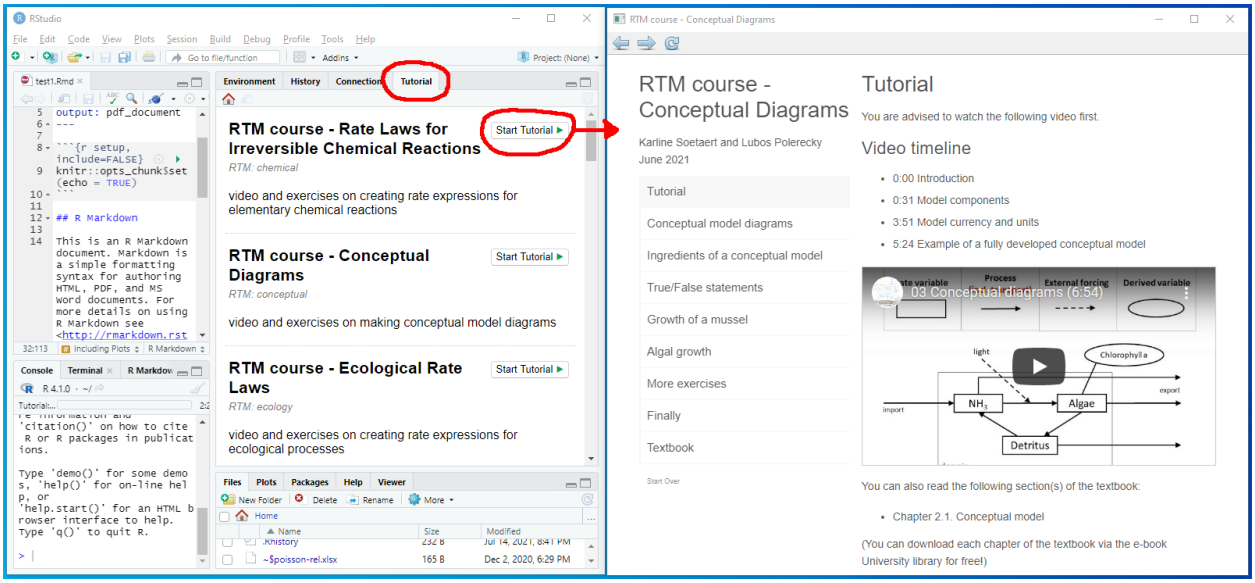


Figure 5: Left: Rstudio with the Tutorial tab selected. Right: RTM tutorial *conceptual* run as a web-based app, created using the *learnr* package.

Alternatively, type in the R-console

```
RTMtutorial("?")
```

to see the complete list of available tutorials (see table below).

```
##                      x                                      description
## 1         introduction                     About the course at Utrecht
## 2                  why                         Why modelling is useful
## 3           conceptual                       Making conceptual models
## 4          massbalance                 Creating mass balance equations
## 5              Rmodels              Dynamic modelling in the R language
## 6           largescale        Large-scale models (e.g. earth's C-cycle)
## 7             chemical                   Elementary chemical reactions
## 8          equilibrium      Equilibrium (reversible) chemical reactions
## 9            enzymatic                               Enzymatic reactions
## 10        partitioning   Chemical reactions partitioning between phases
## 11             ecology                               Ecological reactions
## 12  transportProcesses                 The general transport equation
## 13     transportFluxes               Advection and diffusion/dispersion
## 14     transportPorous               Reaction transport in porous media
## 15 transportBoundaries          Boundary conditions in transport models
## 16          transportR Modelling Reaction tranport in porous media in R
```

Then, choose a tutorial you want to view (e.g., `conceptual`), and start it by typing in the R-console

```
RTMtutorial("conceptual")
```

or

```
RTMtutorial(3)
```

Note that the tutorials are sorted from basics to intermediate and advanced topics. By studying the tutorials in the same order, you will be able to build up your knowledge of modelling in R in a gradual way. We will adhere to the same order during the course.

8.1.1 RTM exercises

Each tutorial comes with a set of *exercises*, which contain more complicated problems that you should be able to solve once you have learned the basics from the tutorial. These exercises will be solved *during the class*. This means that you will be able to consult your questions, ideas, and solutions with the lecturers in a direct and interactive way.

The exercises recommended for a given tutorial are listed at the end of the tutorial (section *More exercises*). For example, the tutorial on *Reversible reactions and equilibrium chemistry* recommends three additional exercises (*equilibriumNH3*, *equilibriumHCO3*, *equilibriumOMD*) (Figure 6).

Type in the R-console

```
RTMexercise("?")
```

to see the complete list of available exercises (see table below).

```
##                      x                                              description
## 1            modelersR                          Learning R for modellers
## 2           conceptual      Translating problems into a conceptual scheme
## 3          massbalance                 Creating mass balance equations
## 4  massbalance_ecology        Creating mass balance equations in ecology
## 5          carbonCycle          An earth-system box model of the C-cycle
## 6                ozone                 Ozone dynamics in the troposphere
## 7        dissolutionSi              Dissolution kinetics of Si particles
## 8       equilibriumNH3            Equilibrium chemistry - ammonium/ammonia
## 9      equilibriumHCO3         Equilibrium chemistry - the carbonate system
## 10      equilibriumOMD  Equilibrium chemistry - impact of mineralisation on pH
```

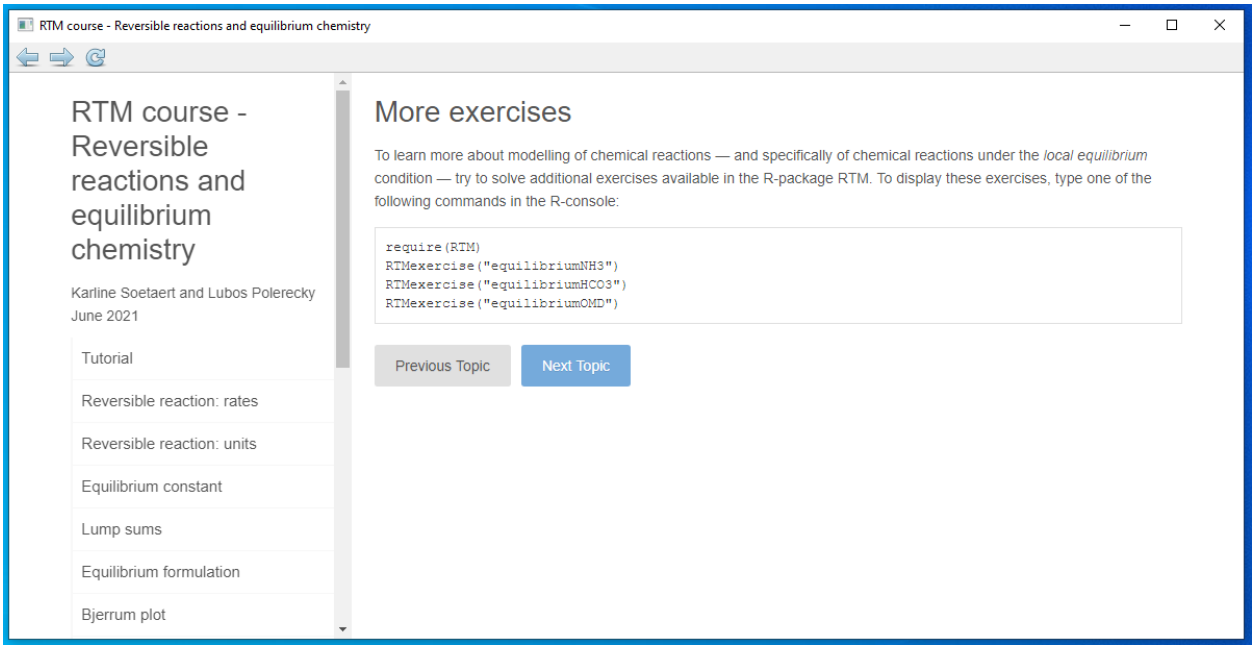


Figure 6: Screenshot of an RTM tutorial *Reversible reactions and equilibrium chemistry*, showing the suggestions for extra exercises to solidify the knowledge gained in the tutorial.

```
## 11            detritus           Bacterial decay of detritus (biogeochemistry)
## 12               COVID                The COVID pandemic (population dynamics)
## 13               virus                                Virus dynamics in the ocean
## 14                npzd                        NPZD model (marine ecosystem model)
## 15   plant_coexistence     Competition and coexistence of plants in grasslands
## 16         crops_weed      Crops and weed competition (agriculture-economics)
## 17      hyacinth_algae Competition between floating plants and algae in a lake
## 18         riverAnoxia     Anoxia in an estuary (1-D reaction transport model)
## 19         Pdiagenesis        Simple phosphorus diagenesis in marine sediments
## 20          diagenesis    Complex diagenesis in marine sediments (C, N, O2, S)
```

Then, choose an exercise you want to solve (e.g., `diagenesis`), and generate the corresponding PDF file by typing in the R-console

```
RTMexercise("diagenesis", type="PDF")
```

or

```
RTMexercise(20, type="PDF")
```

If you want to export the exercise in a HTML or DOC format, change the value of the parameter *type* to `HTML` or `DOC`, respectively.

8.1.2 RTM readers

Finally, the *RTM* package contains extra readers. These readers explain topics that are not part of the course but are nevertheless useful in the context of reaction-transport modelling. They explain, e.g., how to visualise multi-dimensional data generated by the models, implement events and forcing functions based on data, fit data with a model, create interactive model applications, analyse the response times of perturbed systems, or model pH gradients.

The complete list of extra readers is obtained by typing in the R-console

```
RTMreader("?")
```

x	description
events	Events in dynamic models developed in R
forcings	Forcing functions based on data in models developed in R
observations	Showing observed data alongside model results in R
fitting	Fitting a 1D reaction-transport model to data in R
visualisation	Visualising outputs from a 1D reaction-transport model in R
pHprofiles	Estimating pH in a 1D reaction-transport model in R
perturbation_I	Response of systems to a perturbation from equilibrium - Part I
perturbation_II	Response of systems to a perturbation from equilibrium - Part II
interactive	Interactive applications in R
numericalR	Numerical methods used for reaction-transport modelling in R
git_sharing_code	Git, GitLab/Github and RStudio - sharing code with the world

To generate a PDF file of the reader, type in the R-console

```
RTMreader("visualisation", type="PDF")
```

or

```
RTMreader(5, type="PDF")
```

9 References

R Core Team (2018). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. 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. 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.

10 Solutions to exercises

10.1 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 make the code much more legible.

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

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

```
( 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
```

10.2 Vectors and sequences

10.2.1 Mean of a vector

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

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

10.2.2 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. (Similar effects are achieved using the logical operators `<=`, `>=`, `<`, `>`).

```
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)
```

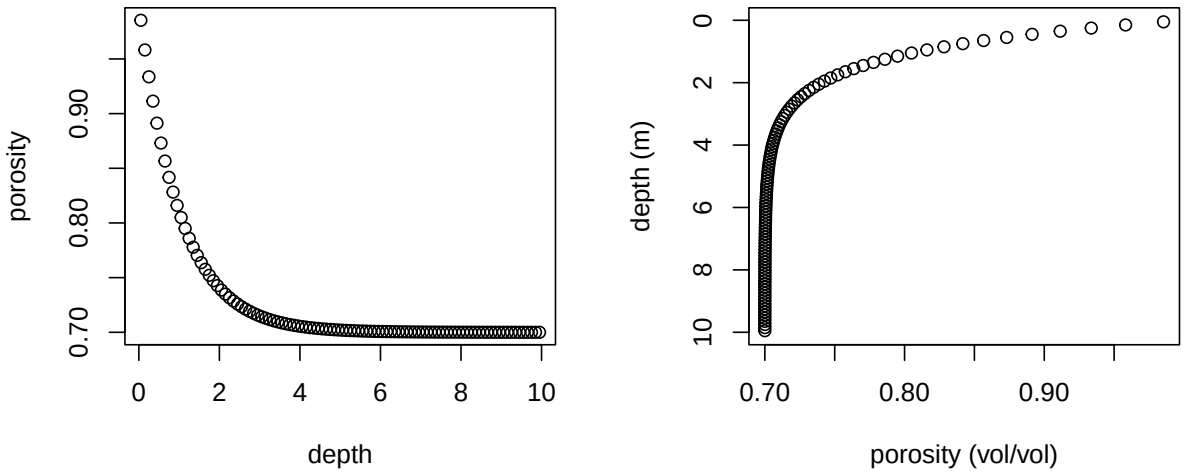
```
## [1] 0.7299861
```

```
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. On the third 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)")
```



10.2.3 Estuarine morphology

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 conventional priority rules. Otherwise, it's better to avoid them to improve code legibility.

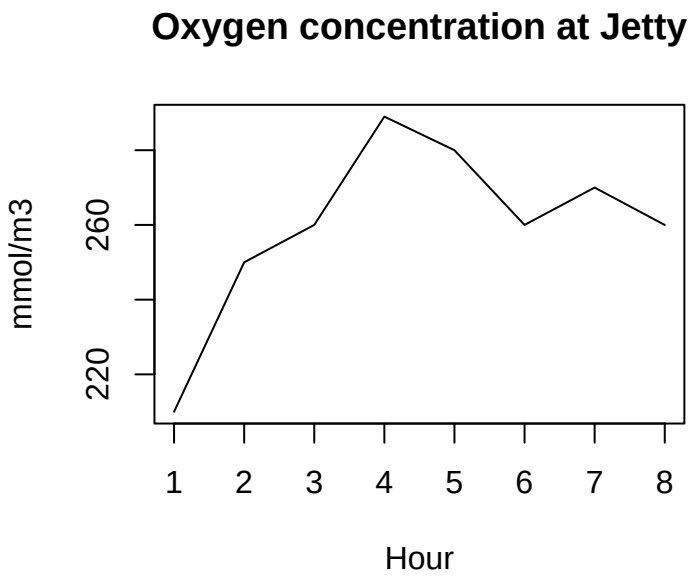
```
L  <- 100000 # metres
N  <- 200
dx <- L/N
x  <- seq(from = dx/2, length.out = N, by = dx)
```

```
Ar <- 4000
As <- 76000
p  <- 5
ks <- 50000
Area <- Ar + (As-Ar) * ((x^p)/(x^p+ks^p)) # very confusing, too many 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
```

10.2.4 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")
```



10.3 R-functions

10.3.1 R-function to estimate saturated oxygen concentrations

```
SatOx <- function(TC = 20, S = 35){ # TC in deg-Celsius
  TK <- TC+273.15                   # TK 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)
```

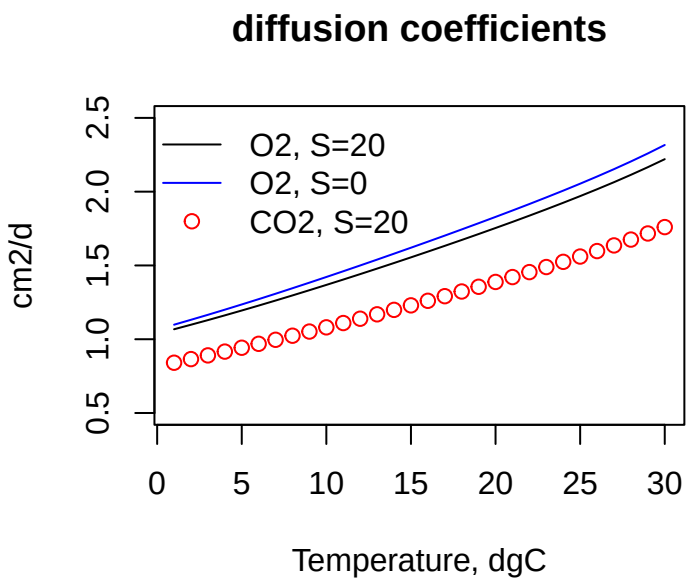
```
##  [1] 349.6542 340.6019 331.9557 323.6924 315.7901 308.2286 300.9890 294.0533
##  [9] 287.4051 281.0288 274.9098 269.0344 263.3897 257.9638 252.7452 247.7235
## [17] 242.8884 238.2306 233.7412 229.4118 225.2346 221.2020 217.3070 213.5431
## [25] 209.9038 206.3833 202.9759 199.6764 196.4796 193.3808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

10.3.2 Molecular diffusion coefficient

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

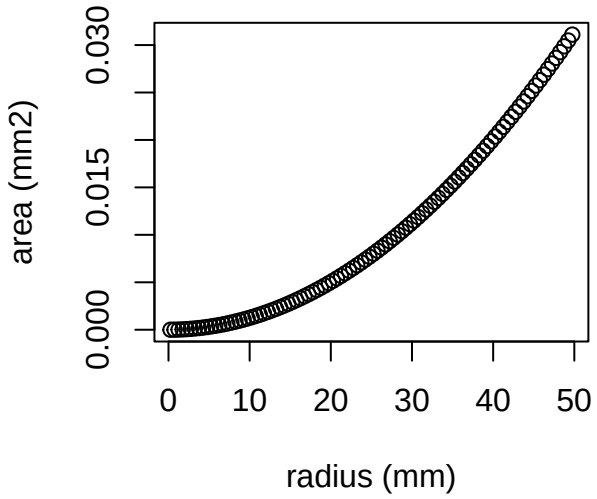
```
##         O2      CO2
## 1 1.368065 1.080572
```

```
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
yrange <- c(0.5, 2.5)                           # A suitable range for y-axis
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")
```



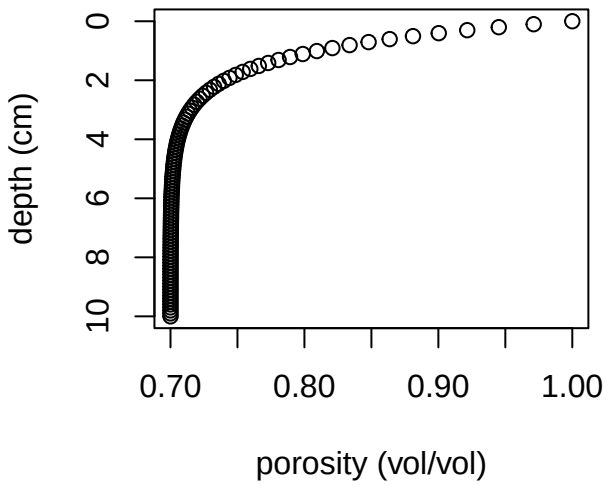
10.3.3 R-function sphere

```
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)")
```



10.3.4 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)")
```

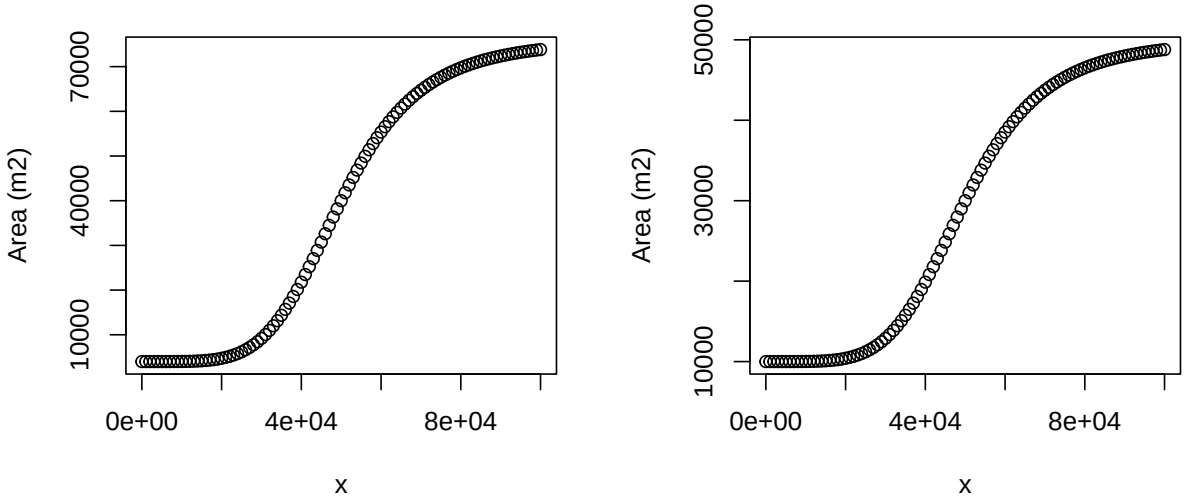



```
# estuarine function, returns the cross-section area and volume as a function of distance
Estfun <- function(x, Ar = 4000, As = 76000, p = 5, ks = 50000, dx = 1000){
  Area <- Ar + (As-Ar) * x^p/(x^p+ks^p)
  Volume <- Area*dx
  return(list(Area = Area, Volume = Volume))
}
dx_new <- 2000
x_new  <- 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
```

10.3.5 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)")
```

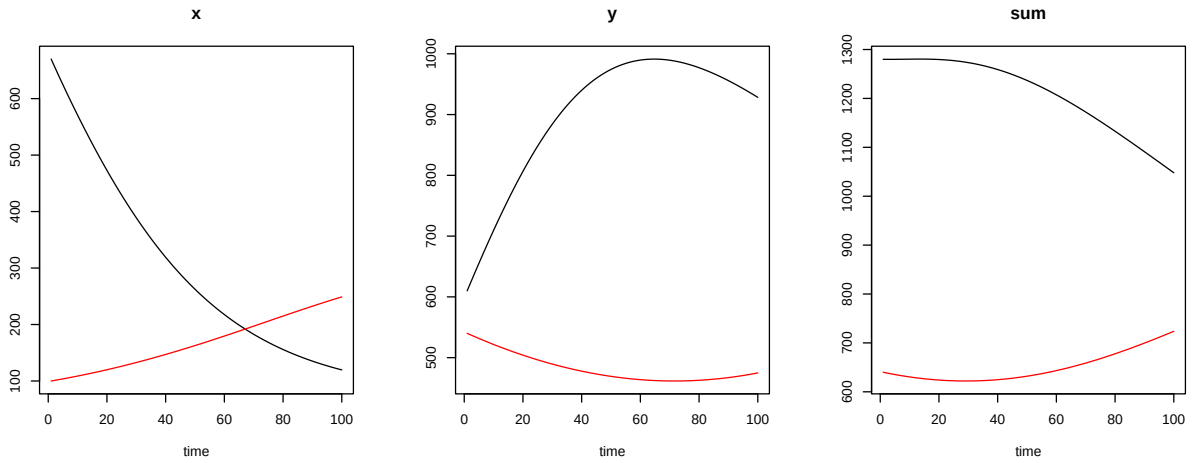


10.4 Solving differential equations in R

10.4.1 Lotka-Volterra model

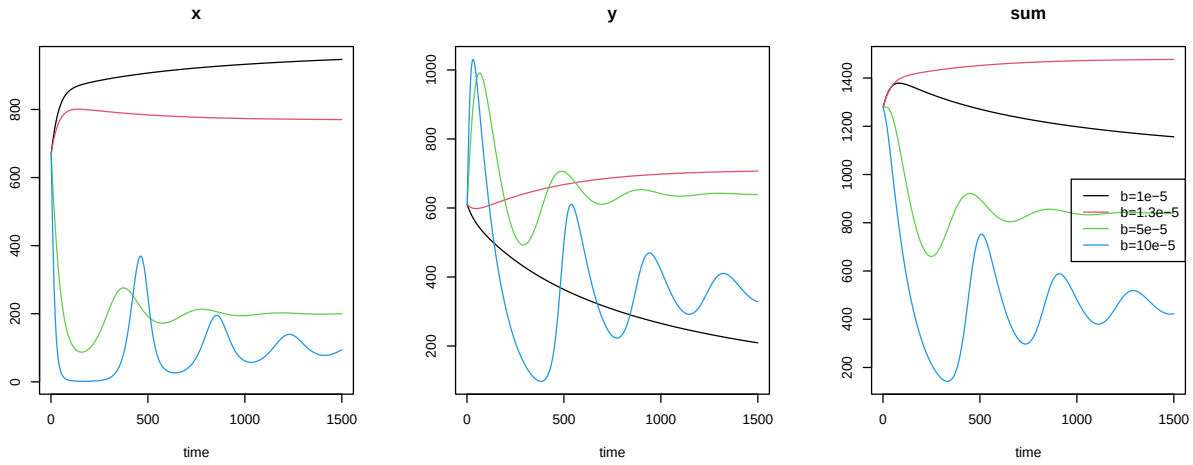
```
LVmodel <- function(t, state, parameters) {
  with (as.list(c(state, parameters)), {

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

    return (list(c(dx.dt, dy.dt),
                 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)
```

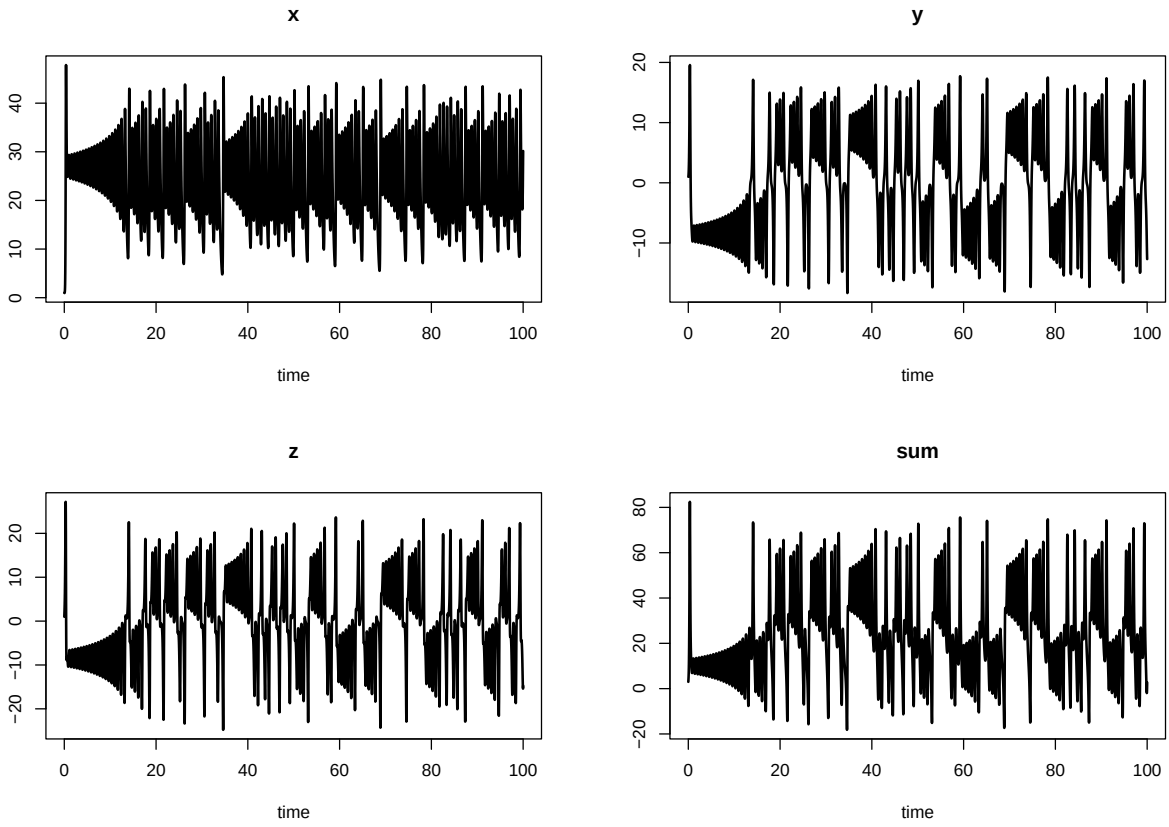


10.4.2 Lorenz model

```
Lorenz <- function(t, state, parameters) {
  with (as.list(c(state, parameters)), {
    dx.dt <- -8/3*x + y*z
    dy.dt <- -10*(y-z)
    dz.dt <- -x*y + 28*y - z
    return (list(c(dx.dt, dy.dt, dz.dt),
                 sum = x+y+z))
  })
}
parameters <-  NULL
state      <- c(x = 1, y = 1, z = 1)
time.seq   <- seq(from = 0, to = 100, by = 0.005)
# solve it
out <- ode(y = state, times = time.seq, func = Lorenz, parms=parameters )
head(out, n=4)
```

```
##       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
```

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


x
y
z
sum
time
time
time
time
0
20
40
60
80
100
0
10
20
30
40
−10
−20
0
10
20
40
60
80