Chapter 19
Bonus Chapter: The R Language

R is a free programming language that traditionally focuses on statistical computing. It was developed in the early 1990s by Ross Ihaka and Robert Gentleman to teach introductory courses in statistics at the University of Auckland, New Zealand. The name of the language derives from the fact that it is a successor to an earlier language S and the shared initials of the authors.

Although R is still used mainly for statistical analysis and graphical representation, it is rapidly becoming more suitable for mathematical computing as well. Areas where significant progress has been made recently are the development of linear algebra tools and the solution of differential equations.

In this chapter we give an introduction to the language, in particular with respect to its use in scientific computing. For general background information and further reading, the book [3] can be recommended. A free online version can be found at `r4ds.hadley.nz`. Very helpful in preparing this chapter was [2], also available online at `adv-r.had.co.nz`.

Programming Environment

R is an interpreted language. It is currently available on major platforms such as Linux, Windows and macOS. Probably the most comfortable programming environment for R is the RStudio environment, which is very similar to the Spyder environment in Python. Installation instructions can be found at `rstudio-education.github.io/hopr`. This site also provides a good starting point for learning programming in R.

All programs in this chapter have been tested for R version 4.3.1 in RStudio on macOS 15.

19.1 Basics

The easiest way to get started with R is to use it like a calculator. Launch RStudio. Then, in the console pane on the left, at the '>' prompt, type the following:

```
> 1 + 2  # my first R program
```

The interpreter then immediately responds with the result

```
[1] 3
```

In R, '#' is the comment symbol. The exact meaning of the output marker '[1]' will be explained later. We often use '##' instead and often write the output on the same line after the input.

Remark Note that numeric values are by default automatically interpreted as float values in machine precision, i.e. as 'double':

```
> typeof(3)  ## "double"
```

Only occasionally you will see a letter L appended to a whole number, such as 3L. This means that the number is stored as a 32-bit integer:

```
> typeof(3L)  ## "integer"
```

We do not normally distinguish between floats and integers and refer to both types generically as "numeric".

Arithmetic

R has all the standard arithmetic operators like addition '+', subtraction '-', multiplication '*', division '/', and exponentiation '^'. R also has operators for integer division '%/%' and remainder in integer division '%%', also known as modulo arithmetic.

Elementary Functions

R offers a large collection of elementary mathematical functions, including the square root `sqrt`, the exponential function `exp`, several variants of the logarithm, and common trigonometric functions. Example:

```
> sin(pi/2)  ## 1
```

Notice that '`pi`' denotes the constant π.

Each R function comes with its own help page, that can be accessed with a preceeding question mark, such as

```
> ?sin  # or, equivalent: help(sin)
```

Then look for the pages to appear in the Help tab of RStudio's bottom-right pane. Actually, help pages are also provided for many other types of R objects.

Comparisons, Boolean Operators

The standard comparison operators are written as in Python and C. They return one of the Boolean values `TRUE` or `FALSE`. As in C, Boolean expressions can be combined with the logical operators `||` (or) and `&&` (and). The operator '`!`' (not) negates a logical value.

Variables

Values are generally assigned to variables using the assignment operator '`<-`'. As in Python and MATLAB, a variable has no particular data type. Unlike many other languages, in R, the dot character '`.`' can be used in variable names.

19.2 Control Structures

R knows the frequently used control structures such as while and for loops and conditional if-else statements.

Example 19.1 (Machine Epsilon) As an example of a while loop, consider the calculation of the machine epsilon in Sect. 3.4 in the Python chapter. Remember that the idea is to find the minimum number n such that for $x_0 = 1$, $x_{n+1} = x_n/2$, the number $x_n + 1$ is indistinguishable from 1 within machine precision.

In R that number n can be computed with

```
1 > eps <- 1; n <- 0
2 > while (1 + eps > 1) {
3 +   n <- n + 1
4 +   eps <- eps / 2 }
5 > n  ## 53
```

Note in line 1 that multiple commands can be put on one line when separated by a semicolon. In line 2, note that the condition is enclosed in parentheses. As in C, the statement block to be executed when the condition is met is enclosed in braces. In lines 3 and 4, the line continuation character '`+`' indicates that the lines still belong to the statement block started in line 2.

Interlude: Scripts

As indicated in the example, direct command-line programming can be cumbersome for program structures that contain blocks of multiple statements. Alternatively, as in the Python Spyder environment, we can write programs in a script file which can then be executed as a whole.

To do this, open the Script Editor by clicking the File menu and choosing File > New File > R Script. When you click the Run button in the script pane, R executes the line of code your cursor is on. If you have highlighted a selection, R will execute the highlighted code. Alternatively, you can run the entire script by clicking the Source button. The script can also be saved to a file for later use.

A note of caution: When you run the script by clicking the Source button, you must explicitly specify output commands. For example, the implicit print command in line 5 in Example 19.1 must then be replaced with `print(n)`.

We leave it to the reader to decide which method to use. From now on, however, we often dispense with the explicit prompt symbol '>'.

Example 19.2 Continuing with Example 19.1, now that we know the number $n = 53$ we can check the result using a for loop:

```
eps <- 1; n <- 52
for (i in 1:n) eps <- eps/2
eps + 1 > 1  ## TRUE
```

The term '`1:n`' in the for loop represents the sequence $1, 2, \ldots, n$. We will return to the formal definition later.

Another iteration gives

```
eps <- eps/2
eps + 1 > 1  ## FALSE
```

and shows that $n = 53$ is indeed minimal.

Note that, as in C, a statement block constisting of a single statement does not need to be enclosed in curly braces.

Finally, we mention that, as in other languages, loops can be terminated with the `break` command. To end only the current iteration, use the `next` command.

Example 19.3 (Collatz Problem) To illustrate the use of if-else statements we return to our standard Collatz problem:

```
1 n <- 100
2 while (n > 1) {
3   if (n %% 2 == 0) n <- n/2
4   else n <- 3*n + 1 }
5 print("reached 1")
```

Notice again that the condition in the if expression in line 3 is in parentheses.

The single statements to be executed in lines 3 and 4 do not need to be enclosed in braces.

As in other languages, the if-else construction in lines 3 and 4 can be abbreviated with a ternary operator, here as

```
n <- ifelse(n %% 2 == 0, n/2, 3*n + 1)
```

19.3 Functions

We have already seen that R provides a large collection of built-in functions. Below we discuss how you can write functions yourself.

Example 19.4 We illustrate the idea with a basic factorial function `fac` to compute $n!$ for integer numbers $n \geq 1$:

```
1 fac <- function(n) {
2   res <- 1;
3   for (i in 1:n) res <- i*res
4   return(res) }
5 fac(5)  ## 120
```

In line 4, the return statement can be replaced by the simple expression '`res`'. If, as here, the execution order is clear from the context, the value of the last expression is automatically returned.

For completeness we mention that in R, functions generally use call-by-value semantics to pass parameters, so that changing the value of a passed argument within a function does not affect the value of the variable in the environment.

Note that function definitions in R, as in Example 19.4, consist of two separate parts, the specification itself and the assignment to a variable. It is quite possible to use the specification alone in the sense of an anonymous function or closure.

Example: The function specification in lines 1-4 can be directly applied to an argument as follows:

```
(function(n) {res <- 1; for (i in 1:n) res <- i*res; res}) (5)
## 120
```

Just notice that when using a closure as an anonymous function, the definition must be enclosed in parentheses.

Functions as Arguments and Return Values

In R it is easy to define functions that take a function as an argument and/or return a function as a value.

Example 19.5 As an example we define an operator that for a function f returns the reciprocal function $1/f$.

```
1 recip <- function (f) { function(x) { 1/f(x) } }
```

We use it to compute $r(x) := 1/e^x$, and evaluate it for $x = 1$:

```
2 r <- recip(exp)
3 r(1)  ## 0.3678794
```

Recursive Functions

A recursive function can only be specified in named form, since it must be referenced within the function body.

Example 19.6 Here is a recursive version of the factorial function in Example 19.4:

```
fac_rec <- function(n) ifelse(n == 0, 1, n * fac_rec(n-1))
fac_rec(5)  ## 120
```

19.4 Vectors and Lists

A *vector* is the most common data structure in R. Central to our purposes are atomic numeric vectors for representing numbers. These correspond to the one-dimensional arrays in Numerical Python. In the following we use the term "vector" mostly in the sense of an atomic numeric vector.

Occasionally we come across other atomic vector types, used for example to collect character strings. The important thing about an atomic vector is that all its components are of a homogeneous basic type.

In contrast, *lists*, sometimes called generic vectors, can contain any mix of data types, even other lists. In this sense, they are similar to lists in Python.

Vectors

The basic method of creating a vector is concatenating individual elements with the operator '`c`'. Example:

```
v <- c(3, 1, 4, 1, 5); v  ## 3 1 4 1 5
```

Components are accessed using subscript or bracket notation in the form `v[i]`, with indices starting with 1. The length of the vector is returned by `length(v)`.

A character vector is created in the same way. Example:

```
v <- c("a", "character", "vector"); v  ## "a" "character" "vector"
```

As mentioned, in this chapter we will almost exclusively use numeric vectors.

For completeness, we mention that R uses a *copy-on-modify* resource management technique to store vectors. When assigning `w <- v` between vectors, only a reference to `w` is passed. First when one of the vectors is changed is it copied to its own memory area. A more detailed explanation can be found in Sect. 18.3 in the Swift chapter.

Empty vectors can be created with

```
v <- c(); v  ##  NULL
```

Note that `NULL` stands for "undefined" or "no value at all". It plays a similar role as the keyword `None` in Python. Also in R, a variable declaration without initialization is not possible, so we need a special object to denote undefinedness.

The concatenation operator can also be used to combine given vectors:

```
v <- c(1, 2, 3); w <- c(4, 5)
c(v, w)  ## 1 2 3 4 5
```

Note that the concatenation is again flattened to a linear sequence.

Another way to create a vector is to use the `seq` function, which in its basic form behaves like the colon operator in MATLAB. Example:

```
seq(from = 1, to = 0, by = -0.3)  ## 1.0 0.7 0.4 0.1
```

Named arguments such as `from`, `to`, `by` can be entered in any order. Often, as here, it is possible to omit the names. Then we have to rely on positional arguments, where of course the order plays a role: `seq(from=1,to=0,by=-0.3)` is equivalent to `seq(1,0,-0.3)`.

If the step is not specified, it is assumed to be 1, as in

```
seq(from = 2, to = 4)  ## 2 3 4
```

Remark For the interested reader. There is a curious difference between the two notations:

```
typeof(seq(2, 4, 1))  ## "double"
typeof(seq(2, 4))     ## "integer"
```

This confirms that the user should not normally be concerned with number storage issues.

An even shorter form that is often used is the notation `x:y` as an abbreviation for `seq(x,y)`. Example:

```
2:5  ## 2 3 4 5
```

The `seq` function can also be used to generate a sequence of equally spaced points, similar to the `linspace` constructor in Python and MATLAB:

```
seq(-1, 1, length = 5)  ## -1.0 -0.5  0.0  0.5  1.0
```

Constant vectors can be created with the '`rep`' function:

```
rep(1, times = 3)  ## 1 1 1
```

Vector Functions

There are several types of vector functions. First there are the scalar functions which, when applied to a vector, are applied component-wise and combine the results to form a new vector.

Example:

```
> exp(1:8)
[1]     2.718282     7.389056    20.085537    54.598150
[5]   148.413159   403.428793  1096.633158  2980.957987
```

Here we can explain the output markers of the form `[i]`. For a longer vector output that spans multiple lines, the output marker indicates the index of the first component in that line.

Then there are functions that are applied to a vector `v` as an object, such as the `length` function shown above. Another typical example is `sum` with its obvious meaning:

```
v <- 1:10
sum(v)  ## 55
```

Still other vector functions return vectors. A typical example is the `rev` function, which returns a vector in reverse order:

```
v <- 1:5
rev(v)  ## 5 4 3 2 1
```

As mentioned earlier, vectors in R are value types. This includes the fact that functions applied to vectors also use call-by-value semantics, i.e. vector functions do not change arguments.

Vectors vs. Loops

As mentioned, R is a vector based language. Consequently, vector operations are implemented to perform efficiently.

Example 19.7 We compute the sum $\sum_{i=1}^{n} i$ for $n = 10^6$ using a for loop:

```
1 s <- 0
2 t0 <- Sys.time()
3 for (i in 1:1e6) s <- s + i
4 t1 <- Sys.time()
5 t1 - t0  ## time difference of 0.0188272 secs
```

The execution time for the loop in line 3 is measured in the enclosing lines 2, 4 and 5.

Alternatively, instead of the loop, we can use the built-in function `sum` to add all elements:

```
3 s <- sum(1:1e6)
```

On our test machine we then get

```
5 t1 - t0  ## Time difference of 0.0004777908 secs
```

i. e. a speed increase by a factor of about 40.

Subsetting

In R we can use bracket notation in several ways to extract subvectors from given vectors.

The easiest way is to set the positions of the components to select:

```
1 v <- c(6, 2, 8, 2, 9)
2 v[c(1, 3, 5)]  ## 6 8 9
```

In line 2, the `v` components at positions 1, 3 and 5 are selected.

Conversely, we can specify the components to *exclude* by marking the positions with a prefixed '-' symbol:

```
3 v[c(-2, -4)]   ## 6 8 9
4 v[-c(2, 4)]    #  equivalent
```

Note that mixed include/exclude specifications like `v[c(-1,2)]` are not permitted.

In case of a single-element selector, we can again omit the concatenation operator:

```
5 v[-3]          ## 6 2 2 9
```

The most versatile way to construct subvectors is to use a Boolean function as a filter. Example:

```
6 v[v > 4]       ## 6 8 9
```

What happens here is that the Boolean expression `v>4` is evaluated component-wise, and then the position numbers `i` where `v[i]>4` are combined to yield the selection vector `c(1,3,5)`.

Finally, it should be noted that an empty selection returns the entire vector:

```
7 v[]            ## 6 2 8 2 9
```

This is not very useful for individual vectors, but it is in the context of row and column vectors in matrices.

Subassignment

All subsetting operators can be combined with assignment to change selected values of an input vector: this is called subassignment. Example:

```
8 v[c(1, 5)] <- c(11, 55); v  ## 11  2  8  2 55
9 v[c(2, 4)] <- 0; v          ## 11  0  8  0 55
```

In line 9 note that the scalar value 0 is assigned to both selected positions.

Applications

We show two example programs that illustrate the use of the vector operations described above.

Example 19.8 (Sieve of Eratosthenes) Here is an R implementation of the sieve of Eratosthenes, which has been discussed several times.

```
1 n <- 100
2 L <- 2:n; P <- c()
3 while (length(L) > 0) {
4   p <- L[1]; P <- c(P, p)
5   L <- L[L %% p != 0] }
6 print(P)
7  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
8 [21] 73 79 83 89 97
```

In line 2, the vector `P` is initialized as empty. In line 4, the first element `p` still in `L` is appended to `P`, and then in line 5 all multiples of `p` are removed from `L`.

The marker '`[21]`' in line 8 again indicates that the first element 73 corresponds to position 21 in `P`.

The following example illustrates the use of recursive functions applied to vectors:

Example 19.9 (Quicksort) Here is an R implementation of the Quicksort algorithm explained in Example 3.13 in the Python chapter:

```
1 qsort <- function(v) {
2   if (length(v) <= 1) return(v)
3   p <- v[1]; v <- v[-1]
4   sml <- v[v < p]
5   grt <- v[v >= p]
6   c(qsort(sml), p, qsort(grt)) }
```

In line 3 the first element in `v` is chosen as pivot and removed from `v`. In lines 4 and 5, the remaining elements are distributed to `sml` and `grt` according to their size relation to `p`. In line 6, the parts are sorted recursively and then concatenated with `p` in the middle.

We test the `qsort` function:

```
v <- c(4, 5, 7, 3, 8, 3)
qsort(v)  ## 3 3 4 5 7 8
```

Lists

A list can be created with the `list` function. As a simple example, we define a list `lst` that contains two vectors as components:

```
lst <- list(1:3, 7:9)
```

Components can be accessed using *double bracket notation*:

```
lst[[1]]  ## 1 2 3
lst[[2]]  ## 7 8 9
```

Note that in lists, single bracket notation selects the *sublist* of length one that contains the component, not the component itself. In atomic vectors, a vector of length one is identified with its content component, but in lists this is not the case.

It should be mentioned that lists, like vectors, exhibit copy-on-modify behavior.

Named Lists

A commonly used technique is to assign names to list members. The members can then be referenced directly by name using the '`$`' operator instead of numeric indices.

In the example above, assume that `lst` is defined as

```
lst <- list(low = 1:3, high = 7:9)
```

Then we can access the members with

```
lst$low   ## 1 2 3
lst$high  ## 7 8 9
```

19.5 Matrices

Matrices can be explicitly created from a vector of data elements by specifying the division into the number of rows and columns:

```
> A <- matrix(data = 1:6, nrow = 2, ncol = 3); A
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

The matrix entries are accessed with `A[i,j]`, where *indices start at 1*. The number of rows and columns is returned using `nrow(A)` and `ncol(A)`, the vector consisting of both row and column numbers is returned with `dim(A)`.

If, as here, the data vector is explicitly defined, it is sufficient to specify either the number of rows or the number of columns, as the other can then directly be calculated from the length of the data vector.

Also, the argument names `data`, `nrow`, `ncol` can be dropped if the arguments are given in that order.

As before, we usually skip the input prompt '`>`' and instead specify the output by preceeding it with double comment symbols '`##`'. Also, we often omit row and column markers of the form `[i,]` and `[,j]`.

In R, matrices are given in column-major form by default. To define them in row-major order, we need to state it explicitly:

```
matrix(1:6, nrow = 2, byrow = TRUE)
##   1   2   3
##   4   5   6
```

If '`data`' is a single scalar value, all positions in the matrix are assigned this value. Thus, a zero matrix can be defined, for example, by

```
matrix(data = 0, nrow = 2, ncol = 3)  # zero matrix
matrix(0, 2, 3)                       # equivalent
```

Another way to build matrices is to combine column vectors with the function `cbind`, short for column-bind. For example, the initial example matrix on page 479 above can be created with

```
A <- cbind(1:2, 3:4, 5:6); A  # same as above
```

Similarly, matrices can be created by combining row vectors with `rbind`.

Diagonal matrices can be generated directly using the R function `diag`. In its basic form, it constructs a square matrix with a given vector `v` on the diagonal and 0s elsewhere:

```
diag(1:3)
##   1   0   0
##   0   2   0
##   0   0   3
```

The `diag` function also allows creating $n \times n$ diagonal matrices with constant entries '`a`' in the diagonal with `diag(a, nrow = n)`, or shorter with `diag(a, n)`.

The special form `diag(n)` assumes that the constant `a` is 1, and returns the identity matrix.

Submatrices

As in NumPy and MATLAB, any subblock in a matrix can be accessed, not just individual elements. It is sufficient to choose subsettings of row- and column-indices, and then create the intersection:

```
A <- matrix(1:16, nrow = 4, byrow = TRUE)
A[c(1, 4), c(2, 3)]
##    2    3
##   14   15
```

As in MATLAB, an entry of the form `A[,v]` selects all rows, `A[w,]` all columns.

If one of the selector vectors `v`, `w` in `A[v,w]` consists of a single number, the result is returned as a vector:

```
A[c(1, 4), 2]  ## 2 14
```

Conversely, as with vectors, we can also specify rows or columns to be *excluded*:

```
A[-c(2, 3), c(2, 3)]  # same result as in lines 3-4
```

As with vectors, selections can also be used in subassignments to replace selected entries. Example:

```
A <- matrix(1:12, nrow = 3)
A[c(1, 3), c(2, 4)] <- matrix(42:45, nrow = 2); A
##    1   42    7   44
##    2    5    8   11
##    3   43    9   45
```

The diagonal in a square matrix A can be accessed with `diag(A)` and, if desired, entries can be changed. Example:

```
A <- matrix(1:4, nrow = 2)
diag(A)  ## 1 4
diag(A) <- 8:9; A
##   8   3
##   2   9
```

Example 19.10 We show how to create a matrix with 1s on the first super- and subdiagonal and 0s otherwise. This corresponds to the construction of `sD` in Sect. 4.1 in the Scientific Python chapter.

```
n <- 3
sD <- matrix(0, n, n)
diag(sD[-n, -1]) <- 1
diag(sD[-1, -n]) <- 1
```

We start with a zero matrix in line 2. Then in line 3 the submatrix without row n and column 1 is selected. The diagonal in this sumbmatrix is then set to 1. But obviously the diagonal in the submatrix is actually the first superdiagonal in `sD` itself.

Dually, the construction in line 4 inserts the 1 entries into the subdiagonal.

Another useful method is to select lower and upper triangular matrix parts using the `lower.tri` and `upper.tri` functions.

Example 19.11 As an example we show how to split the matrix

$$A = \begin{pmatrix} 4 & 3 & 0 \\ 3 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix}$$

into a sum $A = L + U$, where L is a lower triangular matrix including the diagonal and R is srict upper triangular:

```
A <- matrix(c(4, 3, 0, 3, 4, -1, 0, -1, 4), nrow=3, byrow=TRUE)
L <- A; U <- A
L[upper.tri(L, diag = FALSE)] <- 0
U[lower.tri(U, diag = TRUE)] <- 0
```

In line 2 we define two copies `L` and `U` of `A`. Then in line 3 all entries that belong to the strict upper part of `L` are selected and replaced with 0. Conversely, in line 4, all entries belonging to the lower part, including the diagonal, are replaced with 0.

Sparse Matrices

As we have seen several times, it is often advantageous for performance reasons to use sparse matrices in linear algebra calculations. Base R does not provide suitable tools,

so we have to rely on an external source. A prominent example that we will use here is the `Matrix` library, which contains a comprehensive collection of efficient tools for working with both dense and sparse matrices.

The `Matrix` package usually comes with every R implementation, and if not, can be installed with

```
> install.packages("Matrix")
```

After installation, the package can then be loaded by entering

```
> library(Matrix)
```

An object of type `Matrix` can now be created as before, just replacing the `matrix` constructor with `Matrix` (with capitalized first letter):

```
1 A <- Matrix(1:6, nrow = 2); A
2 2 x 3 Matrix of class "dgeMatrix"
3      [,1] [,2] [,3]
4 [1,]    1    3    5
5 [2,]    2    4    6
```

Here `dgeMatrix` denotes the standard default type for dense matrices in `Matrix`.

Alternatively we could convert a given `matrix` object:

```
a <- matrix(1:6, nrow = 2)
A <- Matrix(a)  # same output
```

with the same output as in lines 2-5 above.

The specialty of `Matrix` is of course sparse matrices. Here is a simple example:

```
Matrix(c(1, 2, 3, 0, 0, 6), nrow = 2, sparse = TRUE)
##   1  3  .
##   2  .  6
```

The output is marked as a sparse matrix of class `dgCMatrix` to be stored in compressed column format, the standard for sparse matrices in `Matrix`. Note that the 0 values are not stored. This is indicated by the dots in the corresponding positions.

The use of sparse matrices becomes particularly relevant, when the number of 0s is large. By default, the `Matrix` constructor returns a sparse matrix if more than half of the entries are 0. Example:

```
Matrix(c(1, 0, 0, 0, 0, 6), nrow = 2)
##   1  .  .
##   .  .  6
```

The identity matrix is of course a good candidate for a sparse representation. It can be created by converting a `matrix` object generated with the `diag` function:

```
Matrix(diag(1, 3))
##   1   .   .
##   .   1   .
##   .   .   1
```

To generate it directly, we use the `Matrix` function `Diagonal`:

```
Diagonal(3, 1)  # same result as above
```

Note the reverse order of the arguments.

Example 19.12 We construct a sparse variant of the matrix `sD` in Example 19.10. One option is to simply convert the `matrix` object:

```
Matrix(sD)
##   .   1   .
##   1   .   1
##   .   1   .
```

A more elegant, and in larger contexts, more efficient way is to start with a sparse representation of the zero matrix:

```
sD <- Matrix(0, n, n)
diag(sD[-n, -1]) <- 1
diag(sD[-1, -n]) <- 1
sD  # same output as in lines 2-4
```

19.6 Linear Algebra

Numeric vectors directly also take on the role of mathematical vectors. They can be multiplied with a scalar:

```
2 * 1:3    ## 2 4 6
```

and vectors of the same length can be added:

```
1:3 + 3:1  ## 4 4 4
```

In fact, all of the arithmetic operations introduced previously can be applied to vectors. Operations with scalars are mapped over the vectors, operations with two vectors of the same length are applied component by component.

In linear algebra we also need the dot product between vectors. It can be computed by combining an element-wise multiplication with an addition of the products:

```
v <- 1:10; w <- 10:1
sum(v * w)  ## 220
```

Matrix Operations

Matrices are based on linear vectors. Therefore, standard operations such as addition and scalar multiplication are adopted immediately. Here is a list of common matrix operations:

```
A <- matrix(1:4, nrow = 2); B <- matrix(5:8, nrow = 2)
t(A)     # transpose of A
2 * A    # scalar multiplication
A + B    # addition
A * B    # component-wise multiplication
A %*% B  # matrix multiplication
```

Note that all operations refer to `matrix` objects in Base R. However, they also apply to `Matrix` objects created with the `Matrix` library presented above. To see this, it is sufficient to replace the lowercase initials '`m`' in line 1 with the uppercase letter '`M`'. The technical details behind this kind of operator overloading are explained in Sect. 19.9.

Matrix multiplication can also be used to compute the dot product between vectors:

```
v <- 1:10; w <- 10:1
as.vector(v %*% w)  ## 220
```

Note, however, that the result is returned as a 1×1 matrix, which is then converted to a vector of length 1 using the function `as.vector`.

Often useful is the Kronecker product. For background see e.g. Sect. 4.1 in the SciPy chapter. In R it is denoted by `%x%`:

```
A <- matrix(c(1, 3, 2, 4), nrow = 2)
B <- matrix(c(7, 9, 8, 0), nrow = 2)
A %x% B
##    7    8   14   16
##    9    0   18    0
##   21   24   28   32
##   27    0   36    0
```

The Kronecker product can also be applied to objects of the class `Matrix`.

Example 19.13 We can use the Kronecker product to define sparse versions of the Poisson matrix. We base the construction on the sparse matrix `sD` in Example 19.12, here for $n = 3$:

```
n <- 3
# include lines 5-7 from Example 19.12
I <- Diagonal(n, 1)
D <- 4*I - sD
A <- I %x% D + sD %x% -I  # returns sparse 9 x 9 Poisson matrix
```

Linear Equations

Linear equations are solved using the `solve` function:

Example 19.14 Consider the equation system $Ax = b$ with A as in Example 19.11 and $b = (24, 30, -24)$:

```
A <- matrix(c(4, 3, 0, 3, 4, -1, 0, -1, 4), nrow=3, byrow=TRUE)
b <- c(24, 30, -24)
x <- solve(A, b); x  ## 3  4 -5
as.vector(A %*% x)   # returns b
```

If the second argument `b` is missing, `solve` defaults to assuming that it is the identity matrix corresponding to `A`, so `solve(A)` then returns the inverse of `A`. Example:

```
A <- matrix(1:4, nrow = 2, byrow = TRUE)
solve(A)
## -2.0  1.0
##  1.5 -0.5
```

Example 19.15 Continuing with Example 19.14, we solve $Ax = b$ with A and b as there, now however with the Gauss-Seidel method.

For an explanation of the method we refer to Exercise 4.1 in the SciPy chapter and the discussion at the end of Sect. 18.6 in the Fortran chapter.

We start the approximation with the initial guess $x = (3, 3, 3)^T$.

Here is the R program:

```
# include lines 1 and 2 from Example 19.14
# include lines 2-4 from Example 19.11
Linv <- solve(L)
x <- c(3, 3, 3)
for (i in 1:10) x <- Linv %*% (b - U %*% x)
as.vector(x)  ## 2.974898  4.020918 -4.994770
```

Example 19.16 As a more computationally demanding example, we solve $Ax = b$ for the $n^2 \times n^2$ Poissson matrix A with $n = 100$ and the all-ones vector b of length n^2. We use a sparse variant here because, as already mentioned, this achieves significantly better performance.

For convenience, we present the entire construction:

```
1  library(Matrix)
2  n <- 100
3  sD <- Matrix(0, n, n)
4  diag(sD[-n, -1]) <- 1
5  diag(sD[-1, -n]) <- 1
6  I <- Diagonal(n, 1)
7  D <- 4*I - sD
8  A <- I %x% D + sD %x% -I  # sparse Poisson matrix
9  b <- rep(1, n^2)
10 x <- solve(A, b)          # specific solver for sparse matrices
```

Note in line 10 that the `solve` function here uses a method specifically suited for sparse matrices. How the choice of solver methods is controlled is explained in Sect. 19.9.

In fact, the performance is comparable to that in Julia: 0.053 seconds in R compared to 0.021 in Julia. The execution time is computed by including line 10 in a pair of time measurements, as in Example 19.7.

Matrix Decompositions

We discuss R implementations of the standard decompositions related to solving systems of linear equations: LU, Cholesky and QR. For background information we refer to Sect. 4.4 in the Scientific Python chapter.

LU Decomposition

The `solve` function in R uses an LU decomposition. The decomposition itself is strangely not available in Base R. However, it is provided by the `Matrix` package.

Example 19.17 We consider the same example as in the SciPy chapter:

```
1 library(Matrix)
2 A <- matrix(c(1, 2, 3, 1, 1, 1, 3, 3, 1), nrow = 3, byrow = TRUE)
3 LU <- lu(A)    # lu function from Matrix package
4 dec <- expand2(LU)
5 P <- dec$P; L <- dec$L; U <- dec$U
6 P %*% L %*% U  # returns A as "dgeMatrix"
```

The `Matrix` function `lu` in line 3 returns a solution based on the `Lapack` function `dgetrf` already seen in the Fortran and Swift chapters. The solution contains three components: a superposition of the `L` and `U` components in linear form, the matrix dimensions and a vector to construct the permutation matrix.

In line 4, this raw solution is converted into a named list `dec`, which stores the matrices `P`, `L`, `U`. In line 5, the list components are selected by name. Line 6 checks whether they return the original matrix.

For the curious reader it might be interesting to see how the factor matrices are represented:

`P` is stored as a sparse matrix of class `pMatrix`:

```
##   .  |  .
##   .  .  |
##   |  .  .
```

`L` as a `dtrMatrix (unitriangular)`:

```
## 1.000000e+00            .            .
## 3.333333e-01 1.000000e+00            .
## 3.333333e-01 5.551115e-17 1.000000e+00
```

and finally `U` as a `dtrMatrix`:

```
##  3.0000000 3.0000000 1.0000000
##          . 1.0000000 2.6666667
##          .         . 0.6666667
```

Explanations of the involved matrix classes can be found by entering `?"pMatrix-class"` and `?"dtrMatrix-class"` in the console.

Cholesky Decomposition

Recall that the Cholesky decomposition factors a symmetric positive definite matrix A into an upper triangular matrix U and its transpose, such that $A = U^T U$.

Example 19.18 Here is the same example matrix A as in the SciPy chapter:

```
1 A <- matrix(c(11,12,7,12,14,10,7,10,11), nrow = 3, byrow = TRUE)
2 isSymmetric(A)   ## TRUE
3 eigen(A)$values  ## 31.71590754  4.25444827  0.02964419
```

In lines 2 and 3, we first verify that there is a Cholesky decomposition at all by establishing that A is symmetric and all eigenvalues are real and positive.

Then we compute the decomposition itself:

```
4 U <- chol(A)
5 isTriangular(U, upper = TRUE)  ## TRUE
6 t(U) %*% U  # returns A
```

The solution in line 4 corresponds to the one determined in Python. In line 5 we confirm that `U` is upper triangle and in line 6 that $A = U^T U$.

Example 19.19 (Method of Least Squares) We consider the same overdetermined linear equation system $Ax = b$ as in Example 4.4 in the SciPy chapter and write a program to solve it using Cholesky decomposition:

```
A <- matrix(c(3, -6, 4, -8, 0, 1), nrow = 3, byrow = TRUE)
b <- c(-1, 7, 2)
B <- t(A) %*% A               # symmtric positive definite
U <- chol(B)                  # upper triangle
z <- solve(t(U), t(A) %*% b)  # forward substitution
x <- solve(U,z)               # backward substitution
as.vector(x)  ## 5 2
```

QR Decomposition

Recall that the QR decomposition factorizes a matrix A into a product $A = QR$, where Q is a matrix consisting of orthonormal column vectors and R is an upper triangle matrix.

Example 19.20 Here is the same example as in the SciPy chapter:

```
A <- matrix(c(12,-51,4,6,167,-68,4,-24,-41), nrow=3, byrow=TRUE)
qr_decomp <- qr(A)
Q <- qr.Q(qr_decomp)
R <- qr.R(qr_decomp)
Q %*% R     # returns A
Q %*% t(Q)  # returns identity matrix
```

The Gaussian Normal Equation

In the SciPy chapter we used QR decomposition to solve the Gaussian normal equation $A^TAx = A^Tb$ for an overdetermined linear equation system $Ax = b$. The solution was based on a QR decomposition $QR = A^TA$ with a square matrix R, since then $x = R^{-1}Q^Tb$. Actually, in R this is the default behavior for `qr(A)` for $n \times m$ matrices A with $n \geq m$.

Example 19.21 We consider the same overdetermined system as in Example 19.19 above:

```
A <- matrix(c(3, -6, 4, -8, 0, 1), nrow = 3, byrow = TRUE)
b <- c(-1, 7, 2)
qr_decomp <- qr(A)
Q <- qr.Q(qr_decomp)     # returns 3 x 2 matrix
R <- qr.R(qr_decomp); R  # returns 2 x 2 square matrix:
##   -5   10
##    0   -1
gne <- as.vector(solve(R) %*% t(Q) %*% b); gne  ## 5 2
```

19.7 Calculus

In this section we consider real-valued functions of one variable. We begin by introducing basic tools for function visualization. Then we discuss different approaches to differentiation and integration.

Graph Plotting

The basic plotting tool in R is the function `plot`.

Example 19.22 As an example we plot the square function $x \mapsto x^2$ together with the derivative $x \mapsto 2x$ over the unit interval:

```
1 x <- seq(0, 1, length = 100)
2 plot(x, x^2, type = "l", xlab="", ylab="")
3 lines(x, 2*x)
```

In line 2, the argument `type = "l"` means that the graph should be drawn as a line, the last two arguments that we do not want axis lablels.

In line 3, we add the second graph to the first. Here we cannot use the `plot` function because it would overwrite the existing plot. Note that the function name `lines` already indicates that it draws a line. The result is shown on the left in Fig. 19.1.

It may not be quite as expected. What happens, is that the y-range is already automatically fixed by the original plot in line 2 and the larger range for the derivative is ignored.

There are several ways to get around this. Here we use the `matplot` function, which is designed to process multiple curves. The function values are expected to be represented as column vectors in a matrix.

Example 19.23 Continuing with Example 19.22, we replace lines 2 and 3 with:

```
matplot(x, cbind(x^2, 2*x), type = "l", xlab="", ylab="")
```

Note that the function values are represented by vectors of the same length as `x`, i.e. 100. They are then assembled into a 100×2 matrix using the function `cbind`, which was explained in Sect. 19.5.

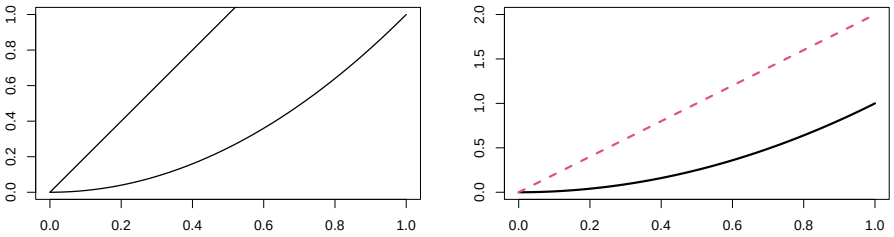

Fig. 19.1 Functions $x \mapsto x^2$ (curved lines) and $x \mapsto 2x$ (straight lines). On the left using `plot` and `lines` according to Example 19.22. On the right using `matplot` according to Example 19.23

Differentiation

As already mentioned in Sect. 19.3, it is quite possible in R to create functions that return functions.

Definition 19.24 We use this to define a differential operator `ddx` that, for a given function f, provides an approximation to the derivative function f':

```
ddx <- function(f) {
  h <- 1e-6
  function(x) (f(x+h) - f(x)) / h }
```

We test the operator for the function $f\colon x \mapsto x^2$:

```
f <- function(x) x^2
dfdx <- ddx(f)
dfdx(1) ## 2.000001
```

Example 19.25 As a slightly larger example, let us consider the function `atan` and compute the first and second derivatives:

```
1 d_atan <- ddx(atan)
2 d2_atan <- ddx(d_atan)
```

We draw the basis function as well as the first and second derivatives over the interval $[-4, 4]$. For this we use the function `matplot` introduced in Example 19.23:

```
x <- seq(-4, 4, length = 100)
matplot(x, cbind(atan(x), d_atan(x), d2_atan(x)),
        type = "l", lty = 1, xlab="", ylab="")
```

The `lty = 1` argument specifies that all lines are drawn as solid lines. The plot is shown in Fig. 19.2.

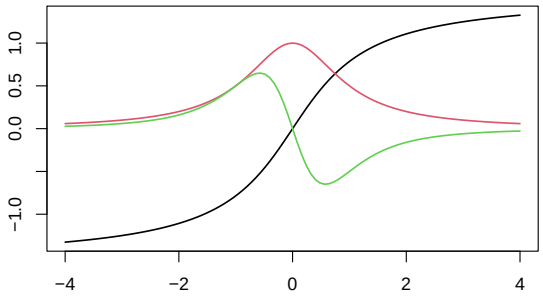

Fig. 19.2 Functions `atan` (black), first derivative `d_atan` (red), and second derivative `d2_atan` (green) evaluated over the interval $[-4, 4]$

Symbolic Differentiation

R also provides an operator called `D` for symbolic differentiation. Similar to the `diff` operator in Symbolic Python and Maple, it works with term expressions.

Example 19.26 As an example, we consider the same `atan` function as before. First, we encode the term `atan(x)` in unevaluated form into an expression object that can be processed by `D`. This is done using the `expression` function, which is similar to the `sympify` function in Symbolic Python:

```
expr <- expression(atan(x))
```

The `D` operator then converts the expression `expr` into an expression for the derivative, where `'x'` denotes the variable with respect to which the derivative is computed:

```
d_expr <- D(expr, "x"); d_expr  ## 1/(1 + x^2)
```

As we can see, the operation returns the correct expression for the derivative.

To evaluate an expression, we use the function `eval`, which assumes that the variables of the expression, here just `x`, have been assigned values in the environment. Example:

```
> x <- 0         # sets the environment
> eval(d_expr)  ## 1
```

Using `eval` we can convert expressions into functions, similar to `lambdify` in SymPy or `unapply` in Maple.

```
3 df <- function(x) eval(d_expr)
```

Applying `df` to an argument value creates an environment in which `x` is assigned that value, and the expression `d_expr` is then evaluated in that environment. Example:

```
  > df(0)  ## 1
```

The `D` operator can also be used for second derivatives:

```
4 d2_expr <- D(d_expr, "x"); d2_expr  ## -(2 * x/(1 + x^2)^2)
5 d2f <- function(x) eval(d2_expr)
```

Putting it all together, we can now plot all involved functions:

```
4 x <- seq(-4, 4, length = 100)
5 matplot(x, cbind(atan(x), df(x), d2f(x)),
6         type = "l", lty = 1, xlab="", ylab="")
```

The plot looks exactly as the one in Fig. 19.2.

Integration

With the function `integrate`, R offers a general-purpose operator for the numerical integration of functions of one variable over an interval $[a, b]$.

Example 19.27 As an example, consider the evaluation for the integrand function $f\colon x \mapsto 1/(1+x^2)$ over $[0, 1]$. Recall that the exact solution is $\pi/4$:

```
1 f <- function(x) 1/(1 + x^2)
2 I <- integrate(f, lower = 0, upper = 1)
3 I$value      ## 0.7853982
4 I$abs.error  ## 8.719671e-15
```

In line 2, `integrate` returns a named list that contains, among other things, the computed value shown in line 3 and an upper bound for the error in line 4.

Example 19.28 (Antiderivative) The `integrate` function can also be used to compute the antiderivative of a given function f:

```
1 aderiv <- function(f) { function(x) {integrate(f, 0, x)$value } }
```

Recall that the antiderivative is unique only up to an additive constant. Therefore, the lower integration limit 0 can be chosen arbitrarly.

We illustrate the use of `aderiv` for the function $f\colon x \mapsto 1/(1+x^2)$:

```
2 f <- function(x) 1/(1 + x^2)
3 F <- aderiv(f)
4 F(1) ## 0.7853982
```

The antiderivative `F` cannot yet be applied to a vector of arguments, as we would need for plotting, for example. To make this possible, we convert it like this:

```
F <- Vectorize(F)
F(-1:1)  ## -0.7853982  0.0000000  0.7853982
```

To check our construction we confirm the fundamental theorem of calculus:

```
g <- ddx(F)      # ddx as in Definition 19.24
x <- seq(-4, 4, length = 100)
plot(x, g(x), type = "l", xlab="", ylab="")
x <- seq(-4, 4, length = 21)
points(x, f(x))  # adds selected points to plot
```

In line 7 we compute F' and plot it over the interval $[-4, 4]$ in line 9. In line 11 we insert the values of the original integrand for selected arguments.

The result is shown in Fig. 19.3.

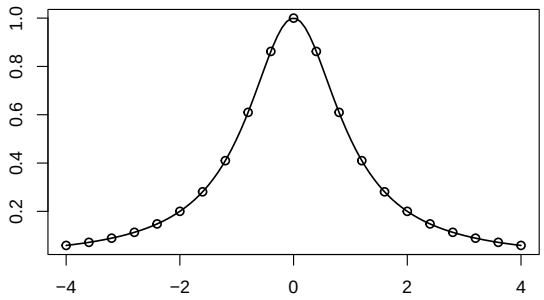

Fig. 19.3 Solid line shows F' for antiderivative F of $f\colon\ x \mapsto 1/(1+x^2)$, points indicate $f(x)$ for selected points

Composite Midpoint, Trapezoid and Simpson's Rules

As an alternative to the built-in function `integrate`, we develop a generic tool that allows us to apply general Newton-Cotes rules, here shown for the usual midpoint, trapezoid and Simpson's rules.

Our basic rules for the integration of a function f over an interval $[a, b]$ are:

```
midpoint <- function(f, a, b) { (b - a) * f((a + b)/2) }
trapeze <- function(f, a, b) { (b - a) * (f(a) + f(b))/2 }
simpson <- function(f, a, b) {
  (b - a) / 6*(f(a) + 4*f((a + b)/2) + f(b)) }
```

We can then define a common operator that can apply composite versions of each of these rules to any equidistant subsivision of the integration interval into a number n of subintervals:

```
my_integrate <- function(rule, f, a, b, n) {
  pts <- seq(a, b, length = n+1)
  int <- 0
  for (i in 1:n) int <- int + rule(f, pts[i], pts[i+1])
  return(int) }
```

We test the operator for the integrand function $f\colon x \mapsto 1/(1+x^2)$ over the unit interval $[0,1]$:

```
f <- function(x) 1/(1 + x^2)
my_integrate(midpoint, f, 0, 1, 50)  ## 0.7854065
my_integrate(trapeze, f, 0, 1, 50)   ## 0.7853815
my_integrate(simpson, f, 0, 1, 50)   ## 0.7853982
```

Recall that the exact solution is $\pi/4$.

19.8 Differential Equations

Base R does not provide any built-in solvers for differential equations. However, at least for ordinary equations, there are several external packages that fill this gap. For background information the book [1] can be recommended.

Initial Value Problems IVPs

A popular choice for IVPs, which we also use below, is the `deSolve` package. It can be installed with

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

After installation it can be loaded with

```
> library(deSolve)
```

Example 19.29 We illustrate the general idea with the equation $u'(x) = x - u(x)$, which we already discussed in Example 4.13 in the SciPy chapter. Except for the R-specific syntax, the program corresponds directly to that in Python:

```
du <- function (x, u, p) { list(x - u) }
xvals <- seq(0, 5, length = 100)
u0 <- 1
sol <- ode(u0, xvals, du)
plot(sol, xlab="", main="")
```

The function in line 1 computes $x - u_e(x)$ based on the current estimate u_e of u. The value is returned in the form of a one-component list. This list can contain additional components, but we don't need that here. As in Julia, the argument `p`, which can contain specific parameter values, cannot be omitted, even if it is empty.

The solver `ode` in line 4 then iteratively modifies the estimator, such that $u_e'(x)$ converges to $x - u_e'(x)$. By default, `ode` uses a general method called `lsoda`. If a different method is desired, e.g. `ode45` as in MATLAB, it can be specified by inserting `method = "ode45"` into the argument list.

The solution `sol` is returned as a 100×2 matrix, where `sol[,1]` contains the grid points in `xvals` and `sol[,2]` the corresponding `u` values.

The solution is plotted in line 5. It looks the same as in the Python example and is therefore not repeated here.

Example 19.30 (Equation System) As an example of a *system* of equations, we consider the same Lotka-Volterra problem as in Example 8.25 in the Julia chapter:

```
1 lotvol <- function (t, u, p) {
2   du1 <-  p[1]*u[1] - p[2]*u[1]*u[2]
3   du2 <- -p[3]*u[2] + p[4]*u[1]*u[2]
4   list(c(du1, du2)) }
5 p = c(1, 0.2, 0.5, 0.2)          # parameter values
6 u0 = c(1, 2)                     # initial values for u1 and u2
7 tspan = seq(0, 4*pi, length = 100)
8 sol = ode(u0, tspan, lotvol, p)  # here parameters needed
9 matplot(sol[,1], sol[,c(2, 3)], type = "l", xlab="", ylab="")
```

In line 4, the values `du1` and `du2` are concatenated into a vector, which is then embedded in a one-component list.

The solution `sol` in line 8 is a 100×3 matrix, where the number of rows is equal to the number of elements in `tspan`. Column 1 contains the values in `tspan`, columns 2 and 3 the function values of the respective functions u_1 and u_2.

Example 19.31 (Second Order IVP) In R, second-order initial value problems are also routinely reduced to a first-order system of equations. As an example, consider the pendulum equation in Example 4.15 in the SciPy chapter:

```
pend <- function(t, u, p) {
  du <- u[2]
  d2u <- -p[1]*u[2] - p[2]*sin(u[1])
  list(c(du, d2u)) }
p <- c(0.25, 5)
u0 <- c(pi - 0.1, 0)
tspan <- seq(0, 10, length = 100)
sol <- ode(u0, tspan, pend, p)
matplot(sol[,1], sol[,c(2, 3)],
        type = "l", lty = 1, xlab="", ylab="")
```

Boundary Value Problems BVPs

The `bvpSolve` package provides tools for dealing with boundary value problems. It can be installed and loaded in a similar manner to the `deSolve` package previously.

Example 19.32 (Bratu Equation) As an example we consider the Bratu equation. Here is the R version of the Python program from Example 4.18:

```
1 bratu <- function(x, u, p) {
2   du <- u[2]
3   d2u <- -exp(u[1])
4   list(c(du, d2u)) }
5 xvals <- seq(0, 1, length = 100)
6 uinit <- c(0, NA); uend <- c(0, NA)
7 sol <- bvpcol(uinit, xvals, bratu, uend)
8 plot(sol, which = 1, xlab="", main="")
```

In line 6 we set values for the functions u and u' at the two boundary points 0 and 1. Actually, we only specify the values for the solution function u itself. The `NA` entry in the second components for u' simply says that the value is not available. For a more detailed explanation see the remark below.

That's all we need to apply the solver `bvpcol` in line 7. Note that the solution is returned for both u and u'. In line 8 we use a special plotting method for BVPs. For details on the selection mechanism, we again refer to Sect. 19.9. The `which=1` statement tells the `plot` command to plot the graph for `u` only. As you can see in Fig. 19.4, this corresponds to the blue curve in Fig. 4.8, whose peak value is about 0.14.

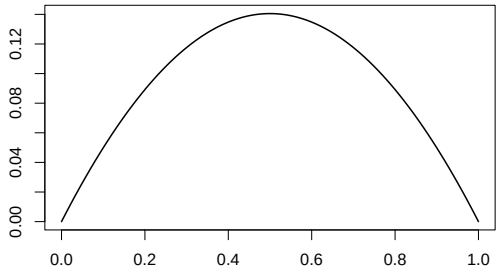

Fig. 19.4 Solution for the Bratu equation $u'' = e^{-u}$, $u(0) = u(1) = 0$

To get the other solution, which peaks at 4.09, we need to run the solver with an initial guess. A rough approach, but one that already works in this case, is to simply include

```
xguess = c(0, 0.5, 1), yguess = matrix(4, nrow = 2, ncol = 3)
```

in the argument list in line 7. The guess assumes the value 4 for $x = 0, 0.5, 1$ for both function u and the derivative u'.

Remark 19.33 Both `NULL` in Sect. 19.4 and `NA` in line 6 in Example 19.32 indicate missing values. The difference is that `NULL` means that a value simply does not exist, whereas `NA` means that a value does exist, but it is unknown.

The difference becomes clear when we use `NULL` and `NA` in vector creation:

```
c(NULL, 22)  ## 22
c(NA, 22)    ## NA 22
```

Example 19.34 (Shooting Method) The `bvpSolve` package also provides a method for solving boundary value problems using the shooting method that "shoots" out trajectories in different directions from one boundary until one finds the trajectory that "hits" the other boundary condition.

In our Bratu equation, we try to find the correct value for u' at the left boundary 0, starting with an initial guess $u'(0) = 0$.

To do this, in the program in Example 19.32 we replace the solver function `bvpcol` in line 7 with `bvpshoot` and add `guess = 0` to the arguments:

```
7 sol <- bvpshoot(uinit, xvals, bratu, uend, guess = 0)
```

This returns the solution, whose peak value is 0.14. The correct value for $u'(0)$ is about 0.55.

To get the solution with a peak value of 4.09 it is enough to set the initial guess to, for example, 6. The actual value is around 10.8.

Partial Differential Equations

As a simple example, consider the following Laplace equation, which has already been discussed several times: Determine a function $u\colon \Omega \to \mathbb{R},\ \Omega := [0,1] \times [0,1]$, such that

$$(*) \qquad \begin{aligned} \Delta u &= 0 \quad \text{in } \Omega, \\ u &= g \quad \text{on the boundary } \partial\Omega, \end{aligned} \qquad \text{where} \quad g(x,y) := \begin{cases} x\,(1-x) & \text{if } y = 0 \\ 0 & \text{else.} \end{cases}$$

Here we solve it via a linear equation system based on the Poisson matrix. To illustrate, consider an interpolation over an $n \times n$ grid for $n = 10$.

For convenience we repeat the construction of the Poisson matrix:

```
1 n <- 10
2 sD <- diag(0, n); diag(sD[-n, -1]) <- 1; diag(sD[-1, -n]) <- 1
3 I <- diag(1, n)
4 D <- 4*I - sD
5 A <- I %x% D + sD %x% -I  # Poisson matrix
```

Note that for small problem sizes as here, we assume dense matrices for simplicity.

Then we prepare the matrix equation. This consists mainly of incorporating the boundary condition into the right-hand side:

```
6 x <- seq(0, 1, length = n)
7 F <- matrix(0, nrow = n, ncol = n)
8 F[,1] <- x*(1 - x)
```

The matrix `F` must be flattened into a vector:

```
9 b <- as.vector(F)
```

The equation can then be solved:

```
solVec <- solve(A, b)
```

and the solution converted back into a matrix:

```
solMat <- matrix(solVec, nrow = n)
```

To visualize the solution, we use the function `persp` that draws perspective plots of a surface over the x-y plane:

```
persp(solMat, theta=30, phi=30, expand=0.5, ticktype="detailed",
      xlab="", ylab="", zlab="")
```

The result is shown in Fig. 19.5.

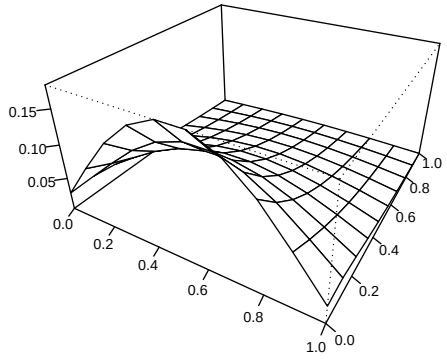

Fig. 19.5 Solution of Laplace equation (∗)

19.9 Object-Oriented Programming I: Generic Functions

The simplest form of object-oriented programming in R is based on the concept of multiple methods in functions, which was already discussed in the Julia chapter.

Example 19.35 (Fractions) We start with an example of how fractions can be represented:

```
fract <- function(num, den) {
  structure(list(num = num, den = den), class = "fract") }
```

An object of class `fract` can then be instantiated with:

```
> a <- fract(3, 4)
```

Note that `a` is actually a `list` in which the components can be accessed by name `a$num` and `a$den`. The function `structure` returns the list with an additional attribute `class = "fract"`.

Just like the underlying lists, `fract` objects have value type semantics. In this respect, they are more comparable to C structures than, for example, to class instances in Python.

The `class` attribute can be used to select specific methods in generic functions to be applied to objects of that class. To illustrate, we add a `print.fract` method to the generic function `print`:

```
print.fract <- function(a) cat(a$num, "/", a$den, sep = "")
```

Now when the `print` function is applied to an object with class attribute `fract`, it actually uses the method `print.fract`:

```
> print(a)  ## 3/4
```

Note that the function `cat` in the definition of `print.fract` first evaluates the expressions as in f-strings in Python or in string interpolation in Julia and then outputs the objects, concatenating the representations.

We can also extend arithmetic operators with class-specific methods. However, note that operator names must then be enclosed in quotation marks:

```
"+.fract" <- function(a, b) {
  fract(a$num * b$den + b$num * a$den,  a$den * b$den) }
"*.fract" <- function(a, b) fract(a$num * b$num, a$den * b$den)
"==.fract" <- function(a, b) a$num * b$den == a$den * b$num
```

We test the new operator methods:

```
> b <- fract(2, 3); c <- fract(1, 2)
> print(a*(b + c))        ## 21/24
> a*(b + c) == a*b + a*c  ## TRUE
```

Inheritance

The concept of generic functions can also be used to emulate subclassing and inheritance. As an example, we consider a class of polynomials together with a sublcass of parabolas.

Example 19.36 (Polynomial Class) Similar to the previous example, we start with a list and a class specifier:

```
pol <- function(c) structure(list(coeff = c), class = "pol")
```

As before, the idea is to represent a polynomial by a vector of coefficients. e.g.:

```
> p <- pol(1:3); p$coeff  ## 1 2 3
```

Then we add an addition method:

```
"+.pol" <- function(p, q) {
  print("polynomial addition")
  v <- p$coeff; w <- q$coeff
```

```
5   n <- max(length(v), length(w))
6   v <- c(v, rep(0, n - length(v)))
7   w <- c(w, rep(0, n - length(w)))
8   pol(v + w) }
```

We use it to add another polynomial in coefficient form to `p` above:

```
> q <- pol(1:4)
> (p + q)$coeff  ## "polynomial addition"  ##  2 4 6 4
```

Note that we combine two outputs into one line to save vertical space.

Example 19.37 (Parabola Subclass) We turn to the construction of the subclass to represent parabolas:

```
9  par <- function(c) {
10   stopifnot(length(c) == 3)  ## abort if no parabola
11   structure(list(coeff = c), class = c("par", "pol")) }
```

In line 10, further processing is aborted if the argument `c` does not have length 3, i.e. does not represent a parabola. Otherwise, line 11 returns the coefficient vector together with a character vector consisting of the class attributes `par` and `pol`. These are used to select function methods. (Incidentally, this is the only place in this chapter where we explicitly use a character vector.)

We add two `par` objects:

```
> p <- par(1:3); q <- par(3:1)
> (p + q)$coeff  ## "polynomial addition"  ## 4 4 4
```

The R interpreter first looks for a specific addition method `+.par` for `par` objects, corresponding to the first entry in the `class` vector in line 11. Since it does not find one, it goes on to the next position `pol` and then finds one for it, namely `+.pol`, which it then uses.

Let us now introduce a specific `par` method:

```
12 "+.par" <- function(p, q) {
13   print("parabola addition")
14   v <- p$coeff; w <- q$coeff
15   par(v + w) }
```

Now this one overrides the `pol` method:

```
> (p + q)$coeff  ## "parabola addition"  ## 4 4 4
```

Example 19.38 (Generic Evaluation Function) We still need a function to evaluate polynomials that are given as coefficient vectors. It should implement a method for general polynomials and a specific method for the subclass of parabolas.

As a preparatory step, we first need to introduce the generic function

```
16 evalp <- function(p, ...) { UseMethod("evalp", p) }
```

This is the formal syntax for defining a function that can contain different methods for different classes. Which method to use is specified by the class attribute of the argument `p`. The ellipsis means that other arguments are passed directly to each method.

Then we can specify a method adapted to `pol` objects:

```
17 evalp.pol <- function(p, x) {
18   print("evalp.pol")
19   res <- 0
20   for (c in rev(p$coeff)) res <- x * res + c
21   return(res) }
```

In lines 19-21, the polynomial is evaluated according to Horner's method. For a detailed explanation, we refer to Examples 18.19 and 18.25 in the Swift chapter.

We test it:

```
> x <- 4
> p <- pol(1:3); evalp(p, x)  ## "evalp.pol"  ## 57
> q <- par(1:3); evalp(q, x)  ## "evalp.pol"  ## 57
```

Note that here the `pol` method is also used for the `par` object `q`, since there is no specific method for parabola evaluation.

But we can define one that then overrides the `evalp` method `evalp.pol`:

```
22 evalp.par <- function(p, x) {
23   print("evalp.par")
24   v <- p$coeff
25   v[1] + v[2]*x + v[3]*x*x }
```

Then we get:

```
> evalp(q, x)  ## "evalp.par"  ## 57
```

Remark 19.39 What we have discussed so far is the class framework known as S3. Objects can be assigned a class attribute, and this attribute can be used to select specific computation methods provided by a generic function.

For the sake of completeness, we mention that there is also a slightly more sophisticated framework called S4. A key difference is that in S4, generic functions can be dispatched to methods based on any number of arguments, not just one like in S3.

Actually, we have implicitly already used a library that is based on the S4 framework, namely the library `Matrix` presented in Sect. 19.5.

19.10 Object-Oriented Programming II: Encapsulated Functions

The distinguishing feature of object-oriented programming is often seen in the bundling of data structures together with processing methods, in technical jargon: encapsulation of methods in classes.

Below we discuss a prominent approach in R, called R6. R6 classes are similar to classes in typical object-oriented languages such as Python and C++. On the other hand, R6 works well with the generic functions approach in S3.

The R6 package can be installed and then loaded with

```
> install.packages("R6")
> library(R6)
```

Example 19.40 (R6 Polynomial Class) As an example we recast the S3 discussion and now create a basic R6 class `Polynomial` to represent polynomials:

```
1  library(R6)
2  Polynomial <- R6Class(
3    classname = "Polynomial",
4    public = list(
5      coeff = NULL,
6      initialize = function(coeff) { self$coeff <- coeff },
7      evalp = function(x) {
8        res <- 0
9        for (c in rev(self$coeff)) res <- x * res + c
10       cat("Polynomial evaluation:", res) } ) )
```

The class name in line 3 is useful primarily for S3 method dispatch, as we will see below.

A list of instance variables and methods follows, here `coeff` for storing a polynomial as a coefficient vector, an initialization method `initialize` that is called automatically when a new class object is instantiated, and a method `evalp` that evaluates a polynomial given as a coefficient vector `coeff` at a point `x`.

The value of `coeff` is initially undefined, indicated by the special value `NULL`. It is then set by the initializer. Note that in methods, members are accessed as '`self`' followed by the '`$`' operator and the member's name, here `self$coeff`. The `evalp` method is again based on Horner's rule.

The member list is declared as `public`, which means that all members can be accessed from outside the class definition.

To instantiate a class object, we use the class method '`new`', which is appended to the class name using the operator '`$`':

```
> p <- Polynomial$new(1:3)
```

The '`new`' method creates the object and calls the method `initialize`, here with the vector '`1:3`' as an argument. It replaces the `NULL` value in `coeff`.

Once the object is instantiated, you can access members using the `$` operator:

```
> p$coeff     ## 1 2 3
> p$evalp(4)  ## Polynomial evaluation: 57
```

Remark Note that unlike S3, R6 objects have reference semantics:

```
> q <- p
> p$coeff[1] <- 4  # modified value is passed also to q:
> q$coeff  ## 4 1 2
```

We add an addition method for `Polynomial` objects. Actually, we don't include it in the class definition, but rely on the generic S3 function concept. The definition is a slight modification of the S3 version:

```
"+.Polynomial" <- function(p, q) {
  print("Polynomial addition")
  # include lines 4-7 from '+.pol' definition in Example 19.36
  Polynomial$new(v + w) }
```

We test it:

```
> p <- Polynomial$new(1:3)
> (p + p)$coeff  ## "Polynomial addition"  ## 2 4 6
```

Subclassing and Inheritance

In R6, a subclass is specified by providing the superclass object to the `inherit` argument and a member list, potentially overriding members in the superclass.

Example 19.41 (R6 Parabola Subclass) Continuing with the `Polynomial` class, we define a subclass `Parabola`:

```
Parabola <- R6Class(
  classname = "Parabola",
  inherit = Polynomial,
  public = list(
    initialize = function(coeff) {
      stopifnot(length(coeff) == 3)
      super$initialize(coeff) } ) )
```

In line 16, `Parabola` is declared as a subclass of `Polynomial`. Lines 19–21 define an initializer that overrides the one in `Polynomial`: It rejects input vectors that are not of length 3. Otherwise, the input is passed to the initializer in the `Polynomial` superclass.

We try it:

```
> Parabola$new(1:4)  # returns error message
> p <- Parabola$new(1:3)
> p$evalp(4)  ## Polynomial evaluation: 57
```

The evaluation is carried out using the general superclass method.

We now integrate a `Parabola` specific evaluation method '`evalp`' into the class. This can be done either by inserting it into the original `Parabola` class definition above, or, as we do here, by adding it using the class method '`set`':

```
22 Parabola$set("public", "evalp",
23   function(x) {
24     v <- self$coeff
25     res <- v[1] + v[2]*x + v[3]*x*x
26     cat("Parabola evaluation:", res) } )
```

What happens here is that the method defined in lines 23-26 is added to the `public` list under the name of `evalp`.

Since it has the same name as the `evalp` method in `Polynomial`, it overrides it:

```
> p <- Par$new(1:3); p$evalp(4)  ## Parabola evaluation: 57
```

To override the addition method, we again resort to generic S3 functions:

```
27 "+.Parabola" <- function(p, q) {
28   print("Parabola addition")
29   v <- p$coeff; w <- q$coeff  # line 14 in Example 19.37
30   Parabola$new(v + w) }
```

We test it:

```
> (p + p)$coeff  ## "Parabola addition"  ## 2 4 6
```

Linked Lists

As a final example, we show that we can define linked lists in R6. The construction is similar to that in Example 18.27 in the Swift chapter, so the reader is encouraged to consider this example as background.

We first introduce a class to represent nodes:

```
1 Node <- R6Class(
2   public = list(val = NULL, nxt = NULL,
3     initialize = function(val) self$val <- val ) )
```

Individual nodes can then be connected to form a linked list:

```
4 LinkedList <- R6Class(
5   classname = "LinkedList",
6   public = list(head = NULL, tail = NULL,
7     addNode = function(node) {
8       if (is.null(self$head)) self$head <- node
9       else self$tail$nxt <- node
10      self$tail <- node } ) )
```

Note that the `LinkedList` class does not require an initializer. Each assignment to the instance variables is done using the `addNode` method.

More precisely, `addNode` appends a new node to the end of the list. Initially the list is empty, represented by the node `head` being `NULL`. In this case, `node` is assigned to `head` in line 8. Otherwise it is appended to the current end node `tail` in line 9. In line 10, it is then set as the new end node.

As an example, we create a linked list consisting of the numbers $1, 2, \ldots, 10$:

```
> lst <- LinkedList$new()
> for (i in 1:10) lst$addNode(Node$new(i))
```

We still need a method to print `LinkedList` objects. Here we again use the generic S3 function `print` and add a specific method to it:

```
print.LinkedList <- function(lst) {
  node <- lst$head; w <- c()
  while (!is.null(node)) {
    w <- c(w, node$val)
    node <- node$nxt }
  print(w) }
```

Example:

```
> print(lst)  ## 1  2  3  4  5  6  7  8  9 10
```

References

1. K. Soetaert, J. Cash, F. Mazzia *Solving Differential Equations in R*. (Springer-Verlag, 2012)
2. H. Wickham *Advanced R, 2nd Edition.* (Chapman & Hall/CRC, 2019)
3. H. Wickham, M. Çetinkaya-Rundel, G. Grolemund *R for Data Science, 2nd Edition.* (O'Reilly Media, 2023)