

## Chapter 2

# Essential Comparisons of the Matlab and R Languages

We assume a working knowledge of either Matlab or R. For either language, there are many books that describe the basics for beginners. However, a brief comparison of the two languages might help someone familiar with one language read code written in the other.

Matlab and R have many features in common. Some of the differences are trivial while others can be troublesome. Where differences are minor, we offer code in only one language, which will be often R.

We will use typewriter font for any text meant to be interpreted as Matlab or R code, such as `plot(heightfd)`.

### 2.1 A Quick Comparison of Matlab and R Syntax

There are similarities and differences in the syntax for Matlab and R.

#### 2.1.1 *Minor Differences*

Here is a quick list of the more commonly occurring differences so that you easily translate a command in one language in that in the other:

- Your code will be easier to read if function names describe what the function does. This often produces a preference for names with words strung together. This is often done in Matlab by connecting words or character strings with underscores like `create_fourier_basis`. This is also acceptable in R. However, it is not used that often, because previous versions of R (and S-Plus) accepted an underscore as a replacement operator. Names in R are more likely to use dots or periods to separate strings, as in `create.fourier.basis` used below.

- The dot or period in Matlab identifies a component of a `struct` array. This is roughly comparable to the use of the dollar sign (\$) in R to identify components of a list, though there are differences, which we will not discuss here.
- Vectors are often defined using the `c()` command in R, as in `rng = c(0,1)`. In Matlab, this is accomplished using square brackets, as in `rng = [0,1]`.
- On the other hand, R uses square brackets to select subsets of values from a vector, such as `rng[2]`. Matlab does this with parentheses, as in `rng(2)`.
- R has logical variables with values either `TRUE` or `FALSE`. Recent releases of Matlab also have logical variables taking values `true` or `false`.
- Previous releases of R, S, and S-Plus allowed the use of `T` and `F` for `TRUE` and `FALSE`. Recent releases of R have allowed users to assign other values to `T` or `F` for compatibility with other languages. This has the unfortunate side effect that R code written using `T` or `F` could throw an error or give a wrong answer without warning if, for example, a user defined `F = TRUE` or `F = c('Do', 'not', 'use', 'F', 'as', 'a', 'logical.')`.
- In both languages, numbers can sometimes be used as logicals; in such cases, 0 is treated as `FALSE` and any nonzero is `TRUE`.
- If a line of code is not syntactically complete, the R interpreter looks for that code to continue on the next line; Matlab requires the line to end in `"..."` if the code is to be continued on the next line.
- Matlab normally terminates a command with a semicolon. If this is not done, Matlab automatically displays the object produced by the command. Lines in R can end in a semicolon, but that is optional.

In this book, where we give commands in both languages, the R version will come first and the Matlab version second. But we will often give only one version; in most such cases, the conversion is just a matter of following these rules.

The matter of the *assignment operator* needs at least a brief comment. In R the correct way to write the transfer of the value produced by the right side of a statement to the object named on the left side is with the two-character sequence `<-`. We like this notation, and prefer to use it in our own work. However, there was from the beginning a resistance among users of R, S and S-PLUS to the use of two characters instead of one. The underscore `_` was allowed but created problems, if only because of incompatibility with many other languages like Matlab that allowed the underscore in names. Recent versions of R allow the use of `=` for replacement in most contexts, but users are warned that there are situations where the code becomes ambiguous and may generate errors that can be hard to trace. With this in mind, we notwithstanding opt for `=` in this book, primarily to keep statements readable and to minimize the differences between R and Matlab. (Matlab uses only `=` for replacement.)

### 2.1.2 Using Functions in the Two Languages

The ways in which arguments are passed to functions and computed results returned is, unfortunately, different in the two languages. We can illustrate the differences by the ways in which we use the important smoothing function, `smooth.basis` in R and `smooth_basis` in Matlab. Here is a full function call in R:

```
smoothlist = smooth.basis(argvals, y, fdParobj,
                          wtvec, fdnames)
```

and here is the Matlab counterpart:

```
[fdobj, df, gcv, coef, SSE, penmat, y2cMap] = ...
  smooth_basis(argvals, y, fdParobj, wtvec, fdnames);
```

An R function outputs only a single object, so that if multiple objects need to be returned, as in this example, then R returns them within a list object. But Matlab returns its outputs as a set of variable(s); if more than one, their names are contained within square brackets.

The handy R feature of being able to use argument names to provide any subset of arguments in any order does not exist in Matlab. Matlab function calls require the arguments in a rigid order, though only a subsequence of leading arguments can be supplied. The same is true of the outputs. Consequently, Matlab programmers position essential arguments and returned objects first.

For example, most of the time we just need three arguments and a single output for `smooth.basis` and its Matlab counterpart, so that a simpler R call might be

```
myfdobj = smooth.basis(argvals, y, fdParobj)$fd
```

and the Matlab version would be

```
myfdobj = smooth_basis(argvals, y, fdParobj);
```

Here R gets around the fact that it can only return a single object by returning a list and using the `$fd` suffix to select from that list the object required. Matlab just returns the single object. If we want the third output `gcv`, we could get that in R by replacing `fd` with `gcv`; in Matlab, we need to provide explicit names for undesired outputs as, `[fdobj, df, gcv]` in this example. R also has the advantage of being able to change the order of arguments by a call like

```
myfdobj = smooth.basis(y=yvec, argvals=tvec,
                       fdParobj)$fd
```

In order to keep things simple, we will try keep the function calls as similar as possible in the examples in this book.

## 2.2 Singleton Index Issues

The default behavior in matrices and arrays with a singleton dimension is exactly the opposite between R and Matlab: R drops apparently redundant dimensions, compressing a matrix to a vector or an array to a matrix or vector. Matlab does not.

For example, `temp = matrix(c(1, 2, 3, 4), 2, 2)` sets up a  $2 \times 2$  matrix in R, and `class(temp)` tells us this is a "matrix". However, `class(temp[, 1])` yields "numeric", which says that `temp[, 1]` is no longer a matrix. If you want a matrix from this operation, use `temp[, 1, drop=FALSE]`. This can have unfortunate consequences in that an operation that expects `temp[, index]` to be a matrix will work when `length(index) > 1` but may throw an error when `length(index) = 1`. If `A` is a three-dimensional array, `A1 = A[, 1, ]` will be a matrix provided the first and third dimensions of `A` both have multiple levels. If this is in doubt, `dim(A1) = dim(A)[-2]` will ensure that `A` is a matrix, not a vector as it would be if the first or third dimensions of `A` were singleton.

Matlab has the complementary problem. An array with a single index, as in `temp = myarray(:, 1, :)`, is still an array with the same number of dimensions. If you want to multiply this by a matrix or plot its columns, the `squeeze()` function will eliminate unwanted singleton dimensions. In other words, `squeeze(temp)` is a matrix, as long as only one of the three dimension of `temp` is a singleton.

A user who does not understand these issues in R or Matlab can lose much time programming around problems that are otherwise easily handled.

## 2.3 Classes and Objects in R and Matlab

Our code uses *object-oriented programming*, which brings great simplicity to the use of some of the functions. For example, we can use the `plot` command in either language to create specialized graphics tailored to the type of object being plotted, e.g., for basis function systems or functional data objects, as we shall see in the next chapter.

The notion of a class is built on the more primitive notion of a *list* object in R and its counterpart in Matlab, a *struct* object. Lists and structs are used to group together types of information with different internal characteristics. For example, we might want to combine a vector of numbers with a fairly lengthy name or string that can be used as a title for plots. The vector of numbers is a *numeric* object in R or a *double* object in Matlab, while the title string is a *character* object in R and a *char* object in Matlab.

Once we have this capacity of grouping together things with arbitrary properties, it is an easy additional step to define a class as a specific recipe or predefined combination of types of information, along with a name to identify the name of the recipe. For example, in the next chapter we will define the all-important class `fd` as, minimally, a coefficient array combined with a recipe for a set of basis functions.

That is, an `fd` object is either a `list` or a `struct`, depending on the language, which contains at least two pieces of information, each prespecified and associated with the class name `fd`. Actually, the specification for the basis functions is itself also a object of a specific class, the `basisfd` class in R and the `basis` class in Matlab, but let us save these details until the next chapter.

Unfortunately, the languages differ dramatically in how they define classes, and this has wide-ranging implications. In Matlab, a class is set up as a folder or directory of functions used to work with objects of that class.

R has two different ways to define classes and operations on objects of different classes, the S3 and S4 systems. The `fda` package for R uses the S3 system. In this S3 standard, R recognizes an object to be of a certain class, e.g., `fd`, solely by the possession of a `'class'` attribute with value `'fd'`. The `class` attribute is used by *generic* functions such as `plot` by *methods dispatch*, which looks first for a function with the name of the generic followed by the class name separated by a period, e.g., `plot.fd` to plot an object of class `fd`.

An essential operation is the extraction of information from an object of a particular class. Each language has simple classes that are basic to its structure, such as the class `matrix` in either language. However, the power of an object-oriented program becomes apparent when a programmer sets up new classes of objects that, typically, contain multiple entities or components that may be named. These components are themselves objects of various classes, which may be among those that are basic to the language or in turn are programmer-constructed new classes.

There are two standards in R for “object oriented” programming, called “S3” and “S4”. The `fda` package for R uses the S3 system, which is described in Appendix A of Chambers and Hastie (1991). (The S4 system is described in Chambers (2008).) In the S3 system, everything is a vector. Basic objects might be vectors of numbers, either double precision or integers. Or they might be a vector of character strings of varying length. This differs from Matlab, where a character vector is a vector of single characters; to store names with multiple characters, you must use either a character matrix (if all names have the same number of characters) or a cell array of strings (to store names of different lengths). A list is a vector of pointers to other objects. In R, component `i` of vector `v` is accessed as `v[i]`, except that if `v` is a list, `v[i]` will be a sublist of `v`, and `v[[i]]` will return that component. Objects in R can also have `attributes`, and if an object has a `class` attribute, then it is an object of that class. (Classes in the S4 system are much more rigidly defined; see Chambers (2008).) If a list `xxx` in R has an attribute `names = “a”, “b”, “c”`, say, these attributes can optionally be accessed via `xxx$a`, `xxx$b`, `xxx$c`. To see the names associated with an object `x`, use `names(x)` or `attr(x, 'names')`. To see all the `attributes` of an object `x`, use `attributes(x)`. To get a compact summary of `x`, use `str(x)`. Or the class name can also be used in the language’s help command: `help myclass` and `doc myclass` in Matlab or `?myclass` and `help(myclass)` in R.

For example, there are many reasons why one would want to get the coefficient array contained in the functional data `fd` class. In Matlab we do this by using functions that usually begin with the string `get`, as in the command `coefmat =`

`getcoef(fdobj)` that extracts the coefficient array from object `fdobj` of the `fd` class. Similarly, a coefficient array can be inserted into a Matlab `fd` object with the command `fdobj = putcoef(fdobj, coefmat)`. In Matlab, all the extraction functions associated with a class can be accessed by the command `methods myclass`.

The procedure for extracting coefficients from an R object depends on the class of the object. If `obj` is an object of class `fd`, `fdPar` or `fdSmooth`, `coef(obj)` will return the desired coefficients. (The `fd` class is discussed in Chapter 4, and the `fdPar` and `fdSmooth` classes are discussed in Chapter 5.) This is quite useful, because without this generic function, a user must know more about the internal structure of the object to get the desired coefficients. If `obj` has class `fd`, then `obj$coefs` is equivalent to `coef(obj)`. However, if `obj` is of class `fdPar` or `fdSmooth`, then `obj$coefs` will return `NULL`; `obj$fd$coefs` will return the desired coefficients.

As of this writing, a “method” has not yet been written for the generic `coef` function for objects of class `monfd`, returned by the `smooth.monotone` function discussed in Section 5.4.2. If `obj` has that class, it is not clear what a user might want, because it has two different types of coefficients: `obj$Wfdobj$coefs` give the coefficients of a functional data object that is exponentiated to produce something that is always positive and integrated to produce a nondecreasing function. This is then shifted and scaled by other coefficients in `obj$beta` to produce the desired monotonic function. In this case, the structure of objects of class `monfd` is described in the help page for the `smooth.monotone` function. However, we can also get this information using `str(obj)`, which will work for many other objects regardless of the availability of a suitable help page.

To find the classes for which methods have been written for a particular generic function like `coef`, use `methods('coef')`. Conversely, to find generic functions for which methods of a particular class have been written, use, e.g., `methods(class='fd')`. Unfortunately, neither of these approaches is guaranteed to find everything, in part because of “inheritance” of classes, which is beyond the scope of the present discussion. For more on methods in R, see Appendix A in Chambers and Hastie (1991).

## 2.4 More to Read

For a more detailed comparison of R and Matlab, see Hiebeler (2009).

There is by now a large and growing literature on R, including many documents of various lengths freely downloadable from the R website:

<http://www.r-project.org>

This includes books with brief reviews and publisher information as well as freely downloadable documents in a dozen different languages from Chinese to Vietnamese via “Documents: Other” both on the main R page and from CRAN. This

is in addition to documentation beyond `help` that comes with the standard R installation available from `help.start()`, which opens a browser with additional documentation on the language, managing installation, and “Writing R Extensions.”



<http://www.springer.com/978-0-387-98184-0>

Functional Data Analysis with R and MATLAB

Ramsay, J.O.; Hooker, G.; Graves, S.

2009, XII, 202 p., Softcover

ISBN: 978-0-387-98184-0