# Chapter 4
# How to Build Functional Data Objects

We saw in the last chapter that functions are built up from basis systems $\phi_1(t),\ldots,$ $\phi_K(t)$ by defining the linear combination

$$x(t) = \sum_{k=1}^{K} c_k \phi_k(t) = \mathbf{c}'\phi(t).$$

That chapter described how to build a basis system. Now we take the next step, defining a functional data object by combining a set of coefficients $c_k$ (and other useful information) with a previously specified basis system.

## 4.1 Adding Coefficients to Bases to Define Functions

### *4.1.1 Coefficient Vectors, Matrices and Arrays*

Once we have selected a basis, we have only to supply coefficients in order to define an object of the *functional data* class (with class name `fd`).

If there are $K$ basis functions, we need a coefficient vector of length $K$ for each function that we wish to define. If only a single function is defined, then the coefficients are loaded into a vector of length $K$ or a matrix with $K$ rows and one column. If $N$ functions are needed, say for a sample of functional observations of size $N$, we arrange these coefficient vectors in a $K$ by $N$ matrix. If the functions themselves are multivariate of dimension $m$, as would be the case, for example, for positions in three-dimensional space ($m = 3$), then we arrange the coefficients into a three-dimensional array of dimensions $K$, $N$, and $m$, respectively. (A single multivariate function is defined with a coefficient array with dimensions $K, 1$, and $m$; see Section 2.2 for further information on this case.) That is, the dimensions are in the order "number of basis functions," "number of functions or functional observations" and "number of dimensions of the functions."

DOI: 10.1007/978-0-387-98185-7_4,

Here is the command that creates a functional data object using the basis with name `daybasis65` that we created in the previous chapter, with the coefficients for mean temperature for each of the 35 weather stations organized into the 65 by 35 matrix `coefmat`:

```
tempfd = fd(coefmat, daybasis65)
```

You will seldom need to use the `fd` function explicitly because other functions call it after computing `coefmat` as a representation of functional data in terms of the specified basis set. We will discuss some of these functions briefly later in this chapter and in more detail in the next.

### 4.1.2 Labels for Functional Data Objects

Let us take a moment here to reflect on what functional data objects mean. Functional data objects represent functions, and functions are one-to-one mappings or relationships between values in a *domain* and values in a *range*. In the language of graphics, the domain values are points on the horizontal coordinate or *abscissa*, and the range values are points in a vertical coordinate or *ordinate*. For the purpose of this book, we consider mostly one-dimensional domains, such as time, but we do allow for the possibility that the range space of multidimensional, such as (X,Y,Z) triples for the coordinates of points in a three-dimensional space. Finally, we also allow for the possibility of multiple or replicated functions.

Adding labels to functional data objects is a convenient way to supply the information needed for graphical displays. Specialized plotting functions that the code supplies in either language can look for these labels, and if they are present, place them where appropriate for various kinds of plots. The component for labels for functional data objects is called `fdnames`.

If we want to supply labels, we will typically need three, and they are, in order:

1. A label for the domain, such as `'Time'`, `'Day'`, and so on.
2. A label for the replication dimension, such as as `'Weather station'`, `'Child'`, etc.
3. A label for the range, such as `"Temperature (deg. C)'`, `'Space'`, etc.

We refer to these three labels as the *generic labels* for the functional data object.

In R, we supply labels in a list object of length three. An empty version of such a list can be set up by the command

```
fdnames = vector("list", 3)
```

The corresponding object in Matlab is a cell array of length three, which may be set up by

```
fdnames = cell(1,3)
```

In addition to generic labels for each dimension of the data, we may also want, for the range and/or for the replication dimension, to supply sets of labels, each label applying to a specific dimension or replicate. For example, for the gait data, we may want a label such as “Angle” to be common or generic to the two observed angles, but in addition require two labels such as “Knee” and “Hip” to distinguish which angle is being plotted. Similarly, in addition to “Weather Station” to describe generically the replication dimension for the weather data as a whole, we probably want to supply names for each weather station. Thus, labels for replicates and variables have the potential to have two levels, a generic level and a specific level. Of course, if there is only one dimension for range or only one replicate, a two-level labels structure of this nature would usually be superfluous.

In the simple case where a dimension only needs a single name, labels are supplied as strings having the class `character` in R or `char` in Matlab. For example, we may supply only a common name such as “Child” for the replication dimension of the growth data, and “Height(cm)” for the range, combined with “Age (years)” for the domain. Here is a command that sets up these labels in R directly, without bothering to set up an empty list first,

```
fdnames = list("Age (years)", "Child", "Height (cm)")
```

or, assuming that the empty list has already been defined:

```
fdnames[[1]] = "Age (years"
fdnames[[2]] = "Child"
fdnames[[3]] = "Height (cm)"
```

Since Matlab accesses cell array elements by curly brackets  the expressions are

```
fdnames{1} = 'Age (years)'
fdnames{2} = 'Child'
fdnames{3} = 'Height (cm)'
```

However, when the required label structure for either the replication or the range dimension is two-level, we take advantage of the fact that the elements of a list in R can be character vectors or lists, and entries in cell arrays in Matlab can be cell arrays. We deal with the two languages separately in the following two paragraphs.

In R, generic and specific names can be supplied by a named list. The common or generic label is supplied by the name of the list and the individual labels by the entry of the list, this entry being of either the character or list class. Take weather stations for the weather data, for example. The second element is itself a list, defined perhaps by the commands

```
station = vector("list", 35)
station[[ 1]] = "St. Johns"
                .
                .
                .
station[[35]] = "Resolute"
```

A command to set up a labels list for the daily temperature data might be

```
fdnames = list("Day",
               "Weather Station" = station,
               "Mean temperature (deg C)")
```

Notice that the `names` attribute of a list entry can be a quoted string containing blanks, such as what we have used here. The other two names, `argname` and `varname`, will only be used if the entry is `NULL` or `""` or, in the case of variable name, if the third list entry contains a vector of names of the same length as the number of variables. The code also checks that the number of labels in the label vector for replications equals the number of replications and uses the `names` value if this condition fails.

Matlab does not have an analogue of the `names` attribute in R, but each entry in the cell array of length three can itself be a cell array. If the entry is either a string or a cell array whose length does not match the required number of labels, then the Matlab plotting functions will find in this entry a generic name common to all replicates or variables. But if the entry for either the replicates or variables dimension is a cell array of length two, then the code expects the generic label in the first entry and a character matrix of the appropriate number of rows in the second. The weather station example above in Matlab becomes

```
station=cell(1,2);
station{1} = 'Weather Station';
station{2} = ['St. Johns    ';
              'Charlottetown';
                    .
                    .
                    .
              'Resolute     '];
```

Note that a series of names are stored as a matrix of characters, so that enough trailing blanks in each name must be added to allow for the longest name to be used.

## 4.2 Methods for Functional Data Objects

As for the basis class, there are similar generic functions for printing, summarizing and testing for class and identity for functional data objects.

There are, in addition, some useful methods for doing arithmetic on functional data objects and carrying out various transformations. For example, we can take the sum, difference, power or pointwise product of two functions with commands like

```
fdsumobj = fdobj1 + fdobj2
fddifobj = fdobj1 - fdobj2
fdprdobj = fdobj1 * fdobj2
fdsqrobj = fdobj^2
```

One can, as well, substitute a scalar constant for either argument in the three arithmetic commands. We judged pointwise division to be too risky since it is difficult to detect if the denominator function is nonzero everywhere. Similarly,

```
fdobj^a
```

may produce an error or nonsense if `a` is negative and `fdobj` is possibly zero at some point.

Beyond this, the results of multiplication and exponentiation may not be what one might naively expect. For example, the following produces a straight line from (-1) to 2 with a linear spline basis:

```
tstFn0 <- fd(c(-1, 2), create.bspline.basis(norder=2))
plot(tstFn0)
```

However,

```
tstFn0^2
```

is not a parabola but a straight line that approximates this parabola over `rangeval` using the same linear basis set. We get a similar approximation from `tstFn0*tstFn0`, but it differs in the third significant digit.

What do we get from

```
tstFn0^(-1)?
```

The result may be substantially different from what many people expect. These are known "infelicities" in `fda`, which the wise user will avoid. Using cubic or higher-order splines with basis sets larger than in this example will reduce substantially these problems in many but not all cases.

The mean of a set of functions is achieved by a command like

```
fdmeanobj = mean(fdobj)
```

Similarly, functions are summed by the `sum` function. As the software evolves, we expect that other useful methods will be added (and infelicities further mitigated).

We often want to work with the values of a function at specified values of argument $t$, stored, say, in vector `tvec`. The evaluation function comparable to that used in Chapter 3 for basis functions is `eval.fd` in R and `eval_fd` in Matlab. For example, we could evaluate functional data object `thawfd` at times in vector `day.5` by the R command

```
thatvec = eval.fd(tvec, thawfd)
```

The same command can be used to evaluate a derivative of `thawfd` by supplying the index of the derivative as the third argument. The second derivative of `thawfd` is evaluated by

```
D2thatvec = eval.fd(tvec, thawfd, 2)
```

More generally, if `Lfdobj` is an object of the linear differential operator `Lfd` class, defined in Section 4.4, then

```
Lthatvec = eval.fd(tvec, thawfd, Lfdobj)
```

evaluates the result of applying this operator to `thawfd` at the argument values in `tvec`. (In R, `predict.fd` provides a wrapper for `eval.fd` with a different syntax that is more consistent with the standard R generic `predict` function.)

Plotting functions can be as simple as using the command `plot(tempfd)`. Again, Matlab and R use the class name to find the plotting function that is appropriate to what is being plotted, which in this case is an object of the `fd` class. The functional data version of the `plot` function can also use most of the optional arguments available in the standard plotting function for controlling line color, style and width; axis limits; and so forth.

Here is a set of R commands that plot the mean temperature curves for the Canadian weather data after loading the `fda` package. First, we set up the midday times for each of the days in years that are not leap years.

```
daytime = (1:365)-0.5
```

In this book we will find more of interest in the winter months, so we highlight these by rearranging the standard year to run from July 1 to June 30.

```
JJindex = c(182:365, 1:181)
tempmat = daily$tempav[JJindex,]
```

Next we set up a Fourier basis with 65 basis functions, as we did in Chapter 3.

```
tempbasis = create.fourier.basis(c(0,365),65)
```

Now we use the main smoothing function that we will study in Chapter 5 to set up the functional data object `tempfd`, and install names for the three dimensions of the object.

```
tempfd = smooth.basis(daytime, tempmat, tempbasis)$fd
tempfd$fdnames = list("Day (July 2 to June 30)",
                      "Weather Station",
                      "Mean temperature (deg. C)")
```

Finally we plot the 35 mean temperature functions, shown in Figure 4.1, using the optional standard plotting arguments `col` and `lty` to control the color and line style, respectively.

```
plot(tempfd, col=1, lty=1)
```

Lines defined by functional data objects can be added to an existing plot by the `lines` function in R or the `line` function in Matlab.

### 4.2.1 Illustration: Sinusoidal Coefficients

We pointed out in Chapter 3 that curves defined by B-spline bases tend to follow the same track as their coefficients. Here is an example. This R code sets up a coefficient

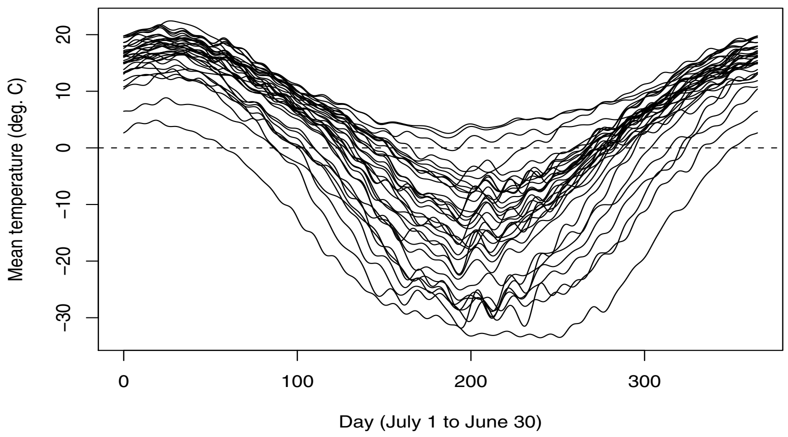

**Fig. 4.1** Mean temperature curves estimated by R command `tempfd = smooth.basis(daytime, tempmat, tempbasis)$fd`, and plotted by command `plot(tempfd)`.

vector of length 13 consisting of values of a sine wave at equally spaced values over its cycle, and then uses these along with the basis system plotted in Figure 3.1 to define a functional data object. Both the defined curve and the sine-valued coefficients are plotted in Figure 4.2. The curve is not a perfect rendition of a spline, but it is surprisingly close.

```
basis13 = create.bspline.basis(c(0,10), 13)
tvec = seq(0,1,len=13)
sinecoef = sin(2*pi*tvec)
sinefd = fd(sinecoef, basis13, list("t","","f(t)"))
op = par(cex=1.2)
plot(sinefd, lwd=2)
points(tvec*10, sinecoef, lwd=2)
par(op)
```

## 4.3 Smoothing Using Regression Analysis

The topic of smoothing data will be taken up in detail in Chapter 5. However, we can sometimes get good results without more advanced smoothing machinery simply by keeping the number of basis functions small relative to the amount of data being approximated.

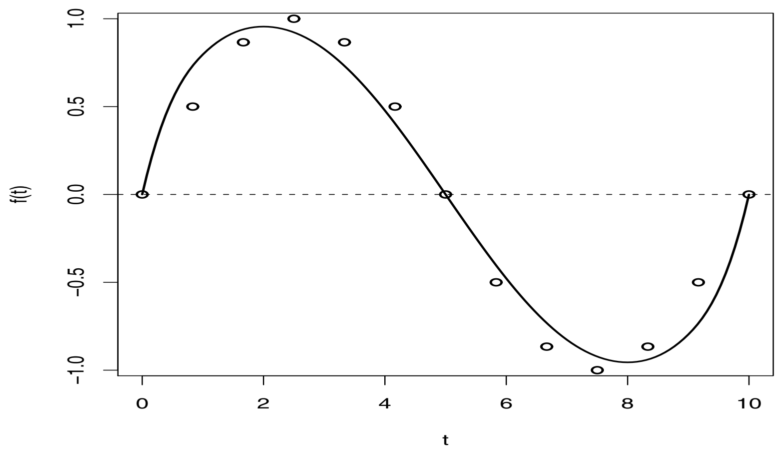

**Fig. 4.2** The 13 spline basis functions defined in Figure 3.1 are combined with coefficients whose values are sinusoidal to construct the functional data object plotted as a solid line. The coefficients themselves are plotted as circles.

### *4.3.1 Plotting the January Thaw*

Canadians love to talk about the weather, and especially in midwinter when the weather puts a chill on many other activities. The January thaw is eagerly awaited, and in fact the majority of Canadian weather stations show clear evidence of these few days of relief. The following code loads 34 years of daily temperature data for Montreal, extracts temperatures for January 16th to February 15th and plots their mean, shown in Figure 4.3.

```
# This assumes the data are in "MtlDaily.txt"
# in the working directory getwd()
MtlDaily = matrix(scan("MtlDaily.txt",0),34,365)
thawdata = t(MtlDaily[,16:47])
daytime  = ((16:47)+0.5)
par(cex=1.2)
plot(daytime, apply(thawdata,1,mean), "b", lwd=2,
     xlab="Day", ylab="Temperature (deg C)")
```

We can fit these data by regression analysis by using a matrix of values of a basis system taken at the times in vector `daytime`. Here we construct a basis system over the interval [16,48] using seven cubic B-splines, and evaluate this basis at these points to produce a 32 by 7 matrix. By default the knots are equally spaced over this interval.

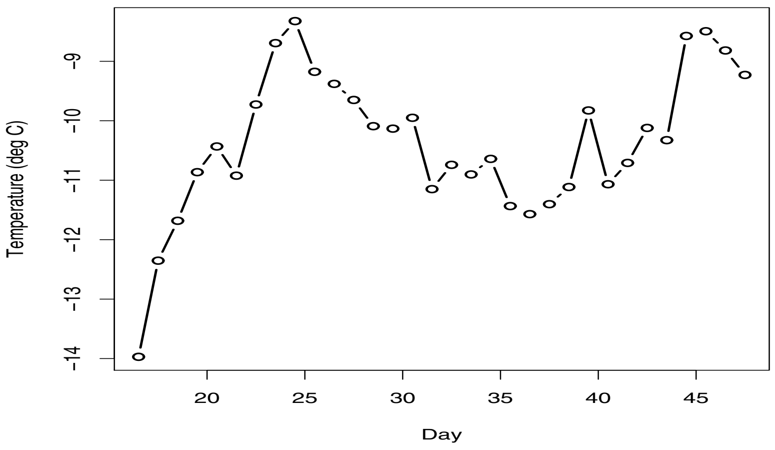

**Fig. 4.3** Temperatures at Montreal from January 16 to February 15 averaged over 1961 to 1994.

```
thawbasis    = create.bspline.basis(c(16,48),7)
thawbasismat = eval.basis(daytime, thawbasis)
```

Now we can compute coefficients for our functional data object by the usual equations for regression coefficients, $\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, and construct a functional data object by combining them with our basis object. A plot of these curves is shown in Figure 4.4 and, sure enough, we do see a fair number of them peaking between January 20 and 25, and a few others with later peaks as well.

```
thawcoef = solve(crossprod(thawbasismat),
                 crossprod(thawbasismat,thawdata))
thawfd = fd(thawcoef, thawbasis,
       list("Day", "Year", "Temperature (deg C)"))
plot(thawfd, lty=1, lwd=2, col=1)
```

We can use these objects to illustrate two useful tools for working with functional data objects. We often want to compare a curve to the data from which it was estimated. In the following command we use function `plotfit.fd` to plot the data for 1961 along with corresponding B-spline fit. The command also illustrates the possibility of using subscripts on functional data objects. The result is shown in Figure 4.5, where the fit suggests a thaw before January 15 and another in early February. The legend on the plot indicates that the standard deviation of the variation of the actual temperatures around the curve is four degrees Celsius.

```
plotfit.fd(thawdata[,1], daytime, thawfd[1],
           lty=1, lwd=2)
```

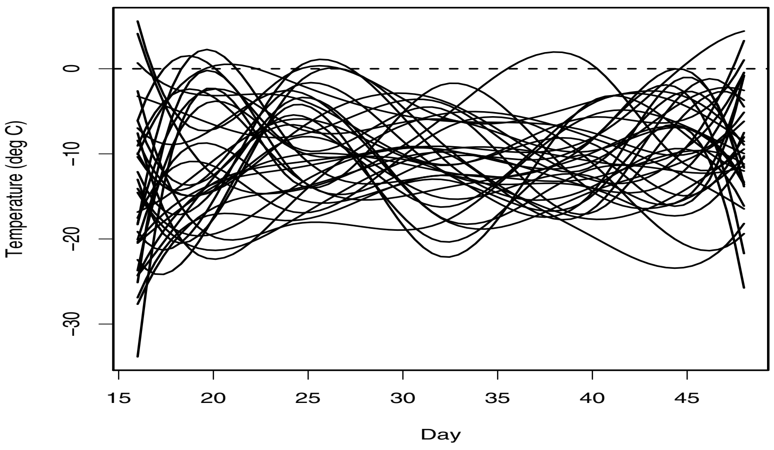

**Fig. 4.4** Functional versions of temperature curves for Montreal between January 16 and February 15. Each curve corresponds to one of the years from 1960 to 1994.

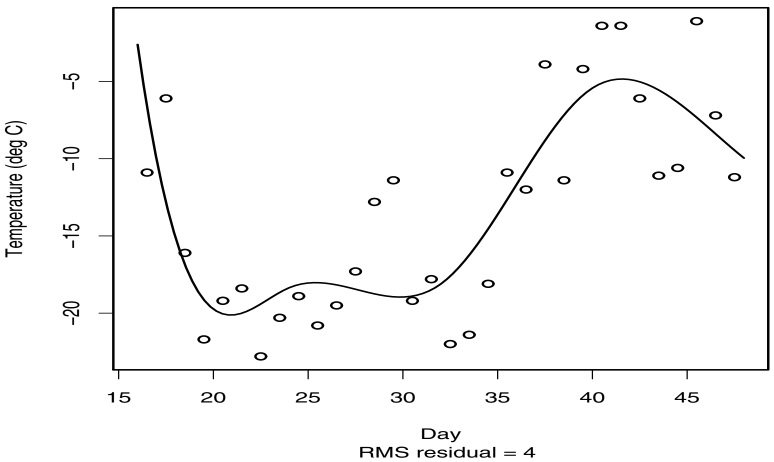

**Fig. 4.5** The temperature curve for 1961 along with the actual temperatures from which it is estimated.

## 4.4 The Linear Differential Operator or `Lfd` Class

We noted in Chapter 1 that the possibility of using a derivative of a function is perhaps the most distinctive feature of functional data analysis. For example, we will take advantage of the information in derivatives in Chapter 5 to customize our definition of what we mean by a "smooth" function. Our discussion in Section 1.4 also implied that the concept of a "derivative" could itself be extended by proposing linear combinations of derivatives, called *linear differential operators*.

Smoothing is supported using the `Lfd` class that expresses the concept of a linear differential operator. An important special case is the *harmonic acceleration* operator that we will use extensively with Fourier basis functions to smooth periodic data.

The notation $Lx$ refers to the application of a linear differential operator $L$ to a function $x$. This might be something as basic as acceleration, $Lx = D^2x$, as moderately sophisticated as harmonic acceleration $L = \omega^2 D + D^3$, or as general as

$$Lx(t) = \beta_0(t)x(t) + \beta_1(t)Dx(t) + ... + \beta_{m-1}(t)D^{m-1}x(t) + D^m x(t) \tag{4.1}$$

where the known *linear differential operator coefficient functions* $\beta_j(t), j = 0,\ldots,$ $m-1$ are either constants or functions.

How do we express this idea in code so as to permit the use of the full potential residing in (4.1)? How do we even do this for harmonic acceleration?

The `Lfd` class is defined by a constructor function `Lfd` that takes as its input two arguments:

`nderiv` The highest order $m$ of the derivative in (4.1).

`bwtlist` A list object in R or a one-dimensional cell array object in Matlab of length $m$. This object contains the coefficient functions $\beta_j$ defining the operator. If a coefficient function varies over $t$, these will be functional data objects with a single replication. But if the coefficient is constant, including zero, the corresponding entry will be that constant.

For example, consider the harmonic acceleration object. Here $m = 3$, and $\beta_0 = \beta_2 = 0$ while $\beta_1 = \omega^2$. In R we could define the harmonic acceleration `Lfd` object `harmaccelLfd` in this way:

```
betalist = vector("list", 3)
betalist[[1]] = fd(0, thawconst.basis)
betalist[[2]] = fd(omega^2, thawconst.basis)
betalist[[3]] = fd(0, thawconst.basis)
harmaccelLfd = Lfd(3, betalist)
```

This is a bit cumbersome for the majority of situations where the differential operator is just a power of $D$ or where all the coefficients $\beta_j$ are constants. Consequently, we have two functions, `int2Lfd` and `vec2Lfd`, to deal with these simpler situations:

```
accelLfd     = int2Lfd(2)
```

```
harmaccelLfd = vec2Lfd(c(0,omega^2,0), c(0, 365))
```

The commands `class(accelLfd)` and `class(harmaccelLfd)` will produce `Lfd` as output.

We are now equipped to evaluate the result of applying any linear differential operator to a functional data object. We can illustrate this by applying an appropriately defined harmonic acceleration operator to temperature curves in functional data object `tempfd`:

```
Ltempmat = eval.fd(daytime, tempfd, harmaccelLfd)
```

This was used to prepare Figure 1.12.

A functional data object for the application of a linear differential operator to an existing functional data object is created by the `deriv.fd` function in R or the `deriv_fd` function in Matlab. The first argument is the functional data object for which the derivative is required and the second is either a nonnegative integer or a linear differential operator object. For example,

```
D2tempfd = deriv.fd(tempfd, 2)
Ltempfd  = deriv.fd(tempfd, harmaccelLfd)
```

## 4.5 Bivariate Functional Data Objects: Functions of Two Arguments

The availability of a sample of $N$ curves makes us wonder how they vary among themselves. The analogue of the correlation and covariance matrices in the multivariate context are the correlation and covariance functions or surfaces, $\rho(s,t)$ and $\sigma(s,t)$. The value $\rho(s,t)$ specifies the correlation between the values $x(s)$ and $x(t)$ over a sample or population of curves, and similarly for $\sigma(s,t)$. This means that we also need to be able to define functions of two arguments, in this case $s$ and $t$. We will need this capacity elsewhere. Certain types of functional regression require bivariate regression coefficient functions.

The bivariate functional data class with name `bifd` is designed to do this. Objects of this class are created in much the same way as `fd` objects, but this now requires two basis systems and a matrix of coefficients for a single such object. In mathematical notation, we define an estimate of a bivariate correlation surface as

$$r(s,t) = \sum_1^K \sum_1^L b_{k,\ell}\phi_k(s)\psi_\ell(t) = \phi'(s)\mathbf{B}\psi(t), \tag{4.2}$$

where $\phi_k(s)$ is a basis function for variation over $s$ and $\psi_\ell(t)$ is a basis function for variation over $t$. The following command sets up such a bivariate functional object:

```
corrfd = bifd(corrmat, sbasis, tbasis)
```

However, situations where you would have to set up bivariate functional data objects are rare, since most of these are set up by R or Matlab functions, `var.fd` and `var_fd` in R and Matlab, respectively. We will use these functions in Chapter 6.

## 4.6 The Structure of the `fd` and `Lfd` Classes

To summarize the most important points of this chapter, we give here the arguments of the constructor function `fd` for an object of the `fd` class.

`coef` A vector, matrix, or three-dimensional array of coefficients. The first dimension (or elements of a vector) corresponds to basis functions. A second dimension corresponds to the number of functional observations, curves or replicates. If `coef` is a three-dimensional array, the third dimension corresponds to variables for multivariate functional data objects.

`basisobj` A functional basis object defining the basis.

`fdnames` A list of length three, each member potentially being a string vector containing labels for the levels of the corresponding dimension of the data. The first dimension is for argument values and is given the default name `"time"`. The second is for replications and is given the default name `"reps"`. The third is for functions and is given the default name `"values"`.

The arguments of the constructor function Lfd for objects of the linear differential operator class are

`nderiv` A nonnegative integer specifying the order $m$ of the highest order derivative in the operator.

`bwtlist` A list of length $m$. Each member contains a functional data object that acts as a weight function for a derivative. The first member weights the function, the second the first derivative, and so on up to order $m-1$.

## 4.7 Some Things to Try

1. Generate a random function using a B-spline basis. Follow these steps:
    a. Decide on the range, such as [0,1].
    b. Choose an order, such as four for cubic splines.
    c. Specify the number of basis functions. The more you specify, the more variability you can achieve in the function. As a first choice, 23 might be reasonable; for order four splines over [0, 1], this places by default knots at $0, 0.05, 0.10, \ldots, 0.90, 0.95$ and 1.

d. Now set up the basis function system in the language you are working with. Plot the basis to see how it looks using the `plot` command (as described in the previous chapter on basis sets).
e. Next define a vector of random coefficients using your language's normal random number generator. These can vary about zero as a mean, but you can also vary them around some function, such as $\sin(2\pi t)$ over [0,1]. If you use a trend, because of the unit sum property of B-splines described above, the function you define will also vary around this trend. You may want to play around their standard deviation as a part of this exercise.
f. Finally, set up a functional data object having a single function using the `fd` command.

2. Plot this function using the `plot` command.
3. Plot both the function and the coefficients on the same graph. To plot the coefficients for order four splines, plot all but the second and third in from each end against knot locations. For example, if you have 23 basis functions, and hence 23 coefficients, plot coefficients 1, 4, 5, and so on up to 20, and then the 23rd. The 21 knots (including end points) are equally spaced by default. At the same time, evaluate the function using the `eval.fd` (R) or `eval_fd` (Matlab) function at a fine mesh of values, such as 51 equally spaced values. Plot these values over the coefficients that you have just plotted. Compare the trend in the coefficients and the curve. If you specified a mean function for the random coefficients, you might want to add this to the plot as well.
4. You might want to extend this exercise to generating $N$ random functions, and plot all of them simultaneously to see how much variation there is from curve to curve. This will, of course, depend on the standard deviation of the random coefficients that you use.
5. Why not also plot the first and second derivatives of these curves, evaluated again using the `eval.fd` function and specifying the order of derivative as the third argument. You might want to compare the first derivative with the difference values for the coefficients.