Chapter 3
How to Specify Basis Systems for Building Functions

We build functions in two stages:

1. First, we define a set of functional building blocks ϕ_k called *basis functions*.
2. Then we set up a vector, matrix, or array of coefficients to define the function as a linear combination of these basis functions.

This chapter is primarily about setting up a basis system. The next chapter will discuss the second step of bundling a set of coefficient values with the chosen basis system.

The functions that we wish to model tend to fall into two main categories: periodic and nonperiodic. The Fourier basis system is the usual choice for periodic functions, and the spline basis system (and bsplines in particular) tends to serve well for nonperiodic functions. We go into these two systems in some detail, and the spline basis especially requires considerable discussion. These two systems are often supplemented by the constant and monomial basis systems, and other systems are described more briefly.

A set of functions in both languages are presented for displaying, evaluating and plotting basis systems as well as for other common tasks.

3.1 Basis Function Systems for Constructing Functions

We need to work with functions with features that may be both unpredictable and complicated. Consequently, we require a strategy for constructing functions that works with parameters that are easy to estimate and that can accommodate nearly any curve feature, no matter how localized. On the other hand, we do not want to use more parameters than we need, since doing so would greatly increase computation time and complicate our analyses in many other ways as well.

We use a set of functional building blocks $\phi_k, k = 1, \ldots, K$ called *basis functions*, which are combined linearly. That is, a function $x(t)$ defined in this way is expressed in mathematical notation as

J.O. Ramsay et al., *Functional Data Analysis with R and MATLAB*, Use R,
DOI: 10.1007/978-0-387-98185-7_3,

$$x(t) = \sum_{k=1}^{K} c_k \phi_k(t) = \mathbf{c}'\boldsymbol{\phi}(t), \tag{3.1}$$

and called a *basis function expansion*. The parameters $c_1, c_2, \ldots, c_K$ are the *coefficients* of the expansion. The matrix expression in the last term of (3.1) uses $\mathbf{c}$ to stand for the vector of K coefficients and ϕ to denote a vector of length K containing the basis functions.

We often want to consider a sample of N functions, $x_i(t) = \sum_{k=1}^{K} c_{ik}\phi_k(t), i = 1, \ldots, N$, and in this case matrix notation for (3.1) becomes

$$\mathbf{x}(t) = \mathbf{C}\boldsymbol{\phi}(t), \tag{3.2}$$

where $\mathbf{x}(t)$ is a vector of length N containing the functions $x_i(t)$, and the coefficient matrix $\mathbf{C}$ has N rows K columns.

Two brief asides on notation are in order here. We often need to distinguish between referring to a function in a general sense and referring to its value at a specific argument value t. Expression (3.1) refers to the basis function expansions of the value of function x at argument value t, but the expansion of x is better written as

$$x = \sum_{k=1}^{K} c_k \phi_k = \mathbf{c}'\boldsymbol{\phi}. \tag{3.3}$$

We will want to indicate the result of taking the mth derivative of a function x, and we will often refer to the first derivative, $m = 1$, as the *velocity* of x and to the second derivative, $m = 2$, as its *acceleration*. No doubt readers will be familiar with the notation

$$\frac{dx}{dt}, \frac{d^2x}{dt^2}, \ldots, \frac{d^m x}{dt^m}$$

used in introductory calculus courses. In order to avoid using ratios in text, and for a number of other reasons, we rather prefer the notation Dx and D^2x for the velocity and acceleration of x, and so on. The notation can also be extended to zero and negative values of m, since $D^0x = x$ and $D^{-1}x$ refers to the indefinite integral of x from some unspecified origin.

The notion of a basis system is hardly new; a polynomial such as $x(t) = 18t^4 - 2t^3 + \sqrt{17}t^2 + \pi/2$ is just such a linear combination of the *monomial* basis functions $1, t, t^2, t^3$, and t^4 with coefficients $\pi/2, 0, \sqrt{17}, -2$, and 18, respectively. Within the monomial basis system, the single basis function 1 is often needed by itself, and we call it the *constant* basis system.

But polynomials are of limited usefulness when complex functional shapes are required. Therefore we do most of our heavy lifting with two basis systems: *splines* and *Fourier series*. These two systems often need to be supplemented by the *constant* and *monomial* basis systems. These four systems can deal with most of the applied problems that we are see in practice.

For each basis system we need a function in either R or Matlab to define a specific set of K basis functions ϕ_k's. These are the `create` functions. Here are the calling statements of the `create` functions in R that set up constant, monomial, Fourier

and spline basis systems, omitting arguments that tend only to be used now and then as well as default values:

```
basisobj = create.constant.basis(rangeval)
basisobj = create.monomial.basis(rangeval, nbasis)
basisobj = create.fourier.basis(rangeval, nbasis,
                                period)
basisobj = create.bspline.basis(rangeval, nbasis,
                                norder, breaks)
```

We will take each of these functions up in detail below, where we will explain the roles of the arguments. The Matlab counterparts of these `create` functions are:

```
basisobj = create_constant_basis(rangeval);
basisobj = create_monomial_basis(rangeval, nbasis);
basisobj = create_fourier_basis(rangeval, nbasis, ...
                                period);
basisobj = create_bspline_basis(rangeval, nbasis, ...
                                norder, breaks);
```

In either language, the specific basis system that we set up, named in these commands as `basisobj`, is said to be a *functional basis object* with the class name `basis` (Matlab) or `basisfd`(R). Fortunately, users rarely need to worry about the difference in class name between Matlab and R, as they rarely need to specify the class name directly in either language.

However, we see that the first argument `rangeval` is required in each `create` function. This argument specifies the lower and upper limits of the values of argument t and is a `vector` object of length 2. For example, if we need to define a basis over the unit interval $[0, 1]$, we would use a statement like `rangeval = c(0,1)` in R or `rangeval = [0,1]` in Matlab.

The second argument `nbasis` specifies the number K of basis functions. It does not appear in the constant basis call because it is automatically 1.

Either language can use the class name associated with the object to select the right kind of function for operations such as plotting or to check that the object is appropriate for the task at hand. You will see many examples of this in the examples that we provide.

We will defer a more detailed discussion of the structure of the `basis` or `basisfd` class to the end of this chapter since this information will only tend to matter in relatively advanced uses of either language, and we will not, ourselves, use this information in our examples.

We will now look at the nature of each basis system in turn, beginning with the only mildly complicated Fourier basis. Then we will discuss the more challenging B-spline basis. That will be followed by more limited remarks on constant and monomial bases. Finally, we will mention only briefly a few other basis systems that are occasionally useful.

3.2 Fourier Series for Periodic Data and Functions

Many functions are required to repeat themselves over a certain period T, as would be required for expressing seasonal trend in a long time series. The Fourier series is

$$\begin{aligned}\phi_1(t) &= 1\\ \phi_2(t) &= \sin(\omega t)\\ \phi_3(t) &= \cos(\omega t)\\ \phi_4(t) &= \sin(2\omega t)\\ \phi_5(t) &= \cos(2\omega t)\\ &\vdots\end{aligned} \tag{3.4}$$

where the constant ω is related to the period T by the relation

$$\omega = 2\pi/T.$$

We see that, after the first constant basis function, Fourier basis functions are arranged in successive sine/cosine pairs, with both arguments within any pair being multiplied by one of the integers $1,2,\ldots$ up to some upper limit m. If the series contains both elements of each pair, as is usual, the number of basis functions is $K = 1+2m$. Because of how we define ω, each basis function repeats itself after T time units have elapsed.

Only two pieces of information are required to define a Fourier basis system:

- the number of basis functions K and
- the period T,

but the second value T can often default to the range of t values spanned by the data. We will use a Fourier basis in the next chapter to smooth daily temperature data. The following commands set up a Fourier basis with $K = 65$ basis functions in R and Matlab with a period of 365 days:

```
daybasis65 = create.fourier.basis(c(0,365), 65)
daybasis65 = create_fourier_basis([0,365],  65);
```

Note that these function calls use the default of $T = 365$, but if we wanted to specify some other period T, we would use

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

in R.

In either language, if K is even, the `create` functions for Fourier series add on the missing cosine and set $K = K+1$. When this leads to more basis functions than values to be fit, the code takes steps to avoid singularity problems.

There are situations where periodic functions are defined in terms of only sines or only cosines. For example, a pure sine series will define functions that have the value 0 at the boundary values 0 and T, while a pure cosine series will define functions

with zero derivatives at these points. Bases of this nature can be set up by selecting only the appropriate terms in the series by either subscripting the basis object or by using a component of the class called `dropind` that contains a vector of indices of basis functions to remove from the final series. For example, if we wanted to set up a Fourier basis for functions centered on zero, we would want to not include the initial constant term, and this could be achieved by either a command like

```
zerobasis = create.fourier.basis(rangeval, nbasis,
                                 dropind=1)
```

or, using a basis object that has already been created, by something like

```
zerobasis = daybasis65[2:65]
```

Here is the complete calling sequence in R for the `create.fourier.basis` in R:

```
create.fourier.basis(rangeval=c(0, 1), nbasis=3,
  period=diff(rangeval), dropind=NULL, quadvals=NULL,
  values=NULL, basisvalues=NULL, names=NULL,
  axes=NULL)
```

A detailed description of the use of the function can be obtained by the command

```
help(create.fourier.basis) or ?create.fourier.basis
help create_fourier_basis  or doc create_fourier_basis
```

in R and Matlab, respectively.

3.3 Spline Series for Nonperiodic Data and Functions

Splines are piecewise polynomials. Spline bases are more flexible and therefore more complicated than finite Fourier series. They are defined by the range of validity, the knots, and the order. There are many different kinds of splines. In this section, we consider only B-splines.

3.3.1 Break Points and Knots

Splines are constructed by dividing the interval of observation into subintervals, with boundaries at points called *break points* or simply *breaks*. Over any subinterval, the spline function is a polynomial of fixed degree or order, but the nature of the polynomial changes as one passes into the next subinterval. We use the term *degree* to refer the highest power in the polynomial. The *order* of a polynomial is one higher than its degree. For example, a straight line is defined by a polynomial of degree one since its highest power is one, but is of order two because it also has a constant term.

We will assume in this book that the order of the polynomial segments is the same for each subinterval.

A spline basis is actually defined in terms of a set of *knots*. These are related to the break points in the sense that every knot has the same value as a break point, but there may be multiple knots at certain break points.

At each break point, neighboring polynomials are constrained to have a certain number of matching derivatives. The number of derivatives that must match is determined by the number of knots positioned at that break point. If only one knot is positioned at a break point, the number of matching derivatives (including the function value itself) is two less than its order, which ensures that for splines of more than order two the join will be seen to be smooth. This is because a function composed of straight line segments of order two will have only the function value (the derivative or order 0) matching, so the function is continuous but its slope is not; this means that the joins would not be seen as smooth by most standards.

3.3.2 Order and Degree

Order four splines are often used, consisting of cubic polynomial segments (degree three), and the single knot per break point makes the function values and first and second derivative values match.

By default, and in the large majority of applications, there will be only a single knot at every break point except for the boundary values at each end of the whole range of t. The end points, however, are assigned as many knots as the order of the spline, implying that the function value will, typically, drop to zero outside of the interval over which the function is defined.

3.3.3 Examples

Perhaps a couple of simple illustrations are in order. First, suppose we define a function over [0,1] with a single interior break point at, say, 0.5. The cubic spline basis set up in the simplest and most usual way has knots $(0,0,0,0,0.5,1,1,1,1)$ because a cubic spline has order four (degree three), so the end knots appear four times each. Similarly, a linear spline has order two, so a single interior break point at 0.5 translates into knots $(0,0,0.5,1,1)$.

Now, suppose that we want an order two polygonal line, but we want to allow the function value to change abruptly at 0.5. This would be achieved by a knot sequence $(0,0,0.5,0.5,1,1)$. Alternatively, suppose we want to work with cubic splines, but we want to allow the first derivative to change abruptly at 0.5 while the function remains continuous. The knot sequence that does this has three knots placed at 0.5. An illustration of such a situation can be seen in the oil refinery Tray 47 function in Figure 1.5.

You will not have to worry about those multiple knots at the end points; the code takes care of this automatically. You will be typically constructing spline functions where you will only have to supply break points, and if these break points are equally spaced, you will not even have to supply these.

To summarize, spline basis systems are defined by the following:

- the break points defining subintervals,
- the degree or order of the polynomial segments, and
- the sequence of knots.

The number K of basis functions in a spline basis system is determined by the relation

$$number\ of\ basis\ functions = order + number\ of\ interior\ knots. \tag{3.5}$$

By *interior* here we mean only knots that are placed at break points which are not either at the beginning or end of the domain of definition of the function. In the knot sequence examples above, that would mean only knots positioned at 0.5.

3.3.4 B-Splines

Within this framework, however, there are several different basis systems for constructing spline functions. We use the most popular, namely the *B-spline basis system*. Other possibilities are M-splines, I-splines, and truncated power functions. For a more extensive discussion of splines, see, e.g, de Boor (2001) or Schumaker (1981).

Figure 3.1 shows the 13 order four B-splines corresponding to nine equally spaced interior knots over the interval $[0, 10]$, constructed in R by the command

```
splinebasis = create.bspline.basis(c(0,10), 13)
```

or, as we indicated in Chapter 2, by the Matlab command

```
splinebasis = create_bspline_basis([0,10], 13);
```

Figure 3.1 results from executing the command `plot(splinebasis)`.

Aside from the two end basis function, each basis function begins at zero and, at a certain knot location, rises to a peak before falling back to zero and remaining there until the right boundary. The first and last basis functions rise from the first and last interior knot to a value of one on the right and left boundary, respectively, but are otherwise zero. Basis functions in the center are positive only over four intervals, but the second and third basis functions, along with their counterparts on the right, are positive over two and three intervals, respectively. That is, all B-spline basis functions are positive over at most four adjacent intervals. This *compact support* property is important for computational efficiency since the effort required is proportional to K as a consequence, rather than to K^2 for basis functions not having this property.

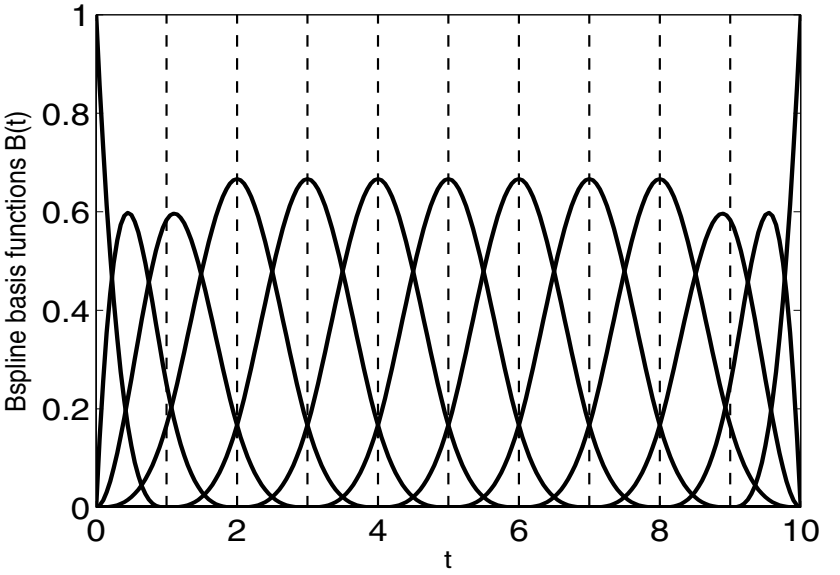

Fig. 3.1 The 13 spline basis functions defined over the interval [0,10] by nine interior boundaries or knots. The polynomial segments are cubic or order four polynomials, and at each knot the polynomial values and their first two derivatives are required to match.

The role of the order of a spline is illustrated in Figure 3.2, where we have plotted linear combinations of spline basis functions of orders two, three and four, called *spline functions*, that best fit a sine function and its first derivative. The three R commands that set up these basis systems are

```
basis2 = create.bspline.basis(c(0,2*pi), 5, 2)
basis3 = create.bspline.basis(c(0,2*pi), 6, 3)
basis4 = create.bspline.basis(c(0,2*pi), 7, 4)
```

Recall from relation (3.5) that, using three interior knots in each case, we increase the number of basis functions each time that we increase the order of the spline basis.

We see in the upper left panel the order two spline function, a polygon, that best fits the sine function, and we see how poorly its derivative, a step function, fits the sine's derivative in the left panel. As we increase order, going down the panels, we see that the fit to both the sine and its derivative improves, as well as the smoothness of these two fits. In general, if we need smooth and accurate derivatives, we need to increase the order of the spline. A useful rule to remember is to *fix the order of the spline basis to be at least two higher than the highest order derivative to be used.* By this rule, a cubic spline basis is a good choice as long as you do not need to look at any of its derivatives.

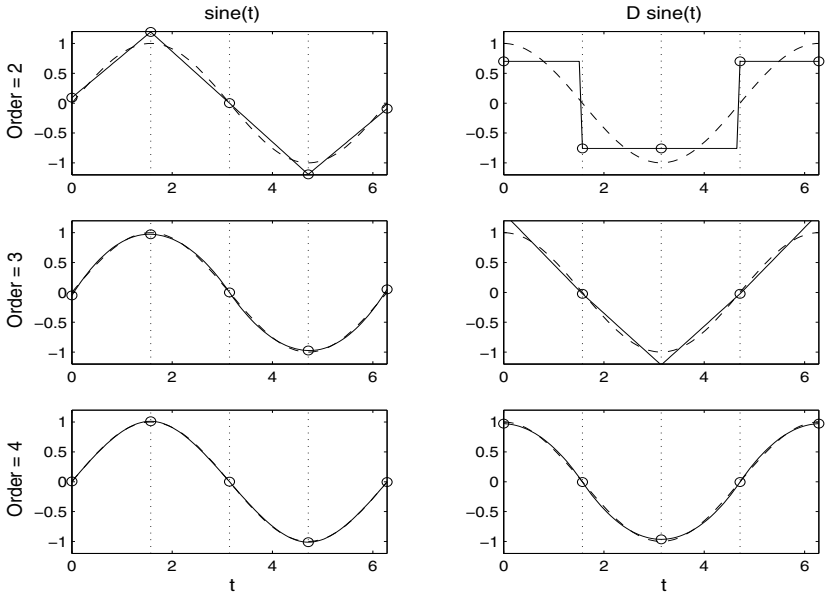

Fig. 3.2 In the left panels, the solid line indicates the spline function of a particular order that fits the sine function shown as a dashed line. In the right panels, the corresponding fits to its derivative, a cosine function, are shown. The vertical dotted lines are the interior knots defining the splines.

The order of a spline is four by default, corresponding to cubic polynomial segments, but if we wanted a basis system with the same knot locations but of order six, we would use an additional argument, as in

```
splinebasis = create.bspline.basis(c(0,10), 15, 6)
```

If, in addition, we wanted to specify the knot locations to be something other than equally spaced, we would use a fourth argument in the function call, with a command such as `create.bspline.basis(c(0,10), nbasis, norder, knotvec)`.

Notice in Figure 3.1 that any single spline basis function is nonzero only over a limited number of intervals, a feature that can be seen more clearly if you use the command `plot(splinebasis[7])` (in R) to plot only the seventh basis function. You would then see that an order four spline basis function is nonzero over four subintervals; similarly, an order six spline is nonzero over over six subintervals. Change 7 to 1 or 2 in this `plot` command to reveal that the end splines are nonzero over a smaller number of intervals.

The B-spline basis system has a property that is often useful: *the sum of the B-spline basis function values at any point t is equal to one*. Note, for example, in Figure 3.1 that the first and last basis functions are exactly one at the boundaries. This is because all the other basis functions go to zero at these end points. Also, because each basis function peaks at a single point, it follows that the value of a

coefficient multiplying any basis function is approximately equal to the value of the spline function near where that function peaks. Indeed, this is exactly true at the boundaries.

Although spline basis functions are wonderful in many respects, they tend to produce rather unstable fits to the data near the beginning or the end of the interval over which they are defined. This is because in these regions we run out of data to define them, so at the boundaries the spline function values are entirely determined by a single coefficient. This boundary instability of spline fits becomes especially serious for derivative estimation, and the higher the order of the derivative, the wilder its behavior tends to be at the two boundaries. However, a Fourier series does not have this problem because it is periodic; in essence, the data on the right effectively "wrap around" to help estimate the curve at the left and vice versa.

Let us set up a spline basis for fitting the growth data by the methods that we will use in Chapter 5. We will want smooth second derivatives, so we will use order six splines. There are 31 ages for height measurements in the data, ranging from 1 to 18, and we want to position a knot at each of these sampling points. Relation (3.5) indicates that the number of basis function is $29+6=35$. If the ages of measurement are in vector `age`, then the command that will set up the growth basis in Matlab is

```
heightbasis = create_bspline_basis([1,18], 35,6,age);
```

As with the Fourier basis, we can select subsets of B-spline basis functions to define a basis by either dropping basis functions using the `dropind` argument or by selecting those basis functions that we want by using subscripts.

Here is the complete calling sequence in R for the `create.bspline.basis` in R:

```
create.bspline.basis(rangeval=NULL, nbasis=NULL,
  norder=4, breaks=NULL, dropind=NULL, quadvals=NULL,
  values=NULL, basisvalues=NULL,
  names="bspl", axes=NULL)
```

A detailed description of the use of the function can be obtained by the commands

```
help(create.bspline.basis)
help create_bspline_basis
```

in R and Matlab, respectively.

3.3.5 Computational Issues Concerning the Range of t

We conclude this section with a tip than can be important if you are using large numbers of spline basis functions. As with any calculation on a computer where the accuracy of results is limited by the number of bits used to express a value, some accuracy can be lost along the way. This can occasionally become serious. A spline is constructed by computing a series of differences. These are especially prone to

rounding errors when the values being differenced are close together. To avoid this, you may need to redefine t so that the length of each subinterval is roughly equal to one. For the gait data example shown in Figure 1.6, where we would construct 23 basis functions if we placed a knot at each time of observation, it would be better, in fact, to run time from 0 to 20 than from 0 to 1 as shown. The handwriting example is even more critical, and by changing the time unit from seconds to milliseconds, we can avoid a substantial amount of rounding error.

On the other hand, computations involving Fourier basis functions tend to be more accurate and stable if the interval $[0,T]$ is not too different from $[0,2\pi]$. We have encountered computational issues, for example, in analyses of the weather data when we worked with $[0,365]$. Once results have been obtained, it is usually a simple matter to rescale them for plotting purposes to a more natural interval.

3.4 Constant, Monomial and Other Bases

3.4.1 The Constant Basis

Different situations call for different basis systems. One such case leads to the simplest basis system. This is the *constant basis*, which contains only a single function whose value is equal to one no matter what value of t is involved. We need the constant basis surprisingly often. For example, we will see in functional regression and elsewhere that we might need to compare an analysis using an unconstrained time-varying function (represented by a functional data or functional parameter object discussed in Chapters 4 and 5, respectively) with a comparable analysis using a constant. We can also convert a conventional scalar variable into functional form by using the values of that variable as coefficients multiplying the constant basis.

The constant basis over, say $[0,1]$, is constructed in R by

```
conbasis = create.constant.basis(c(0,1))
conbasis = create_constant_basis([0,1]);
```

in R and Matlab, respectively.

3.4.2 The Monomial Basis

Simple trends in data are often fit by straight lines, quadratic polynomials, and so on. *Polynomial regression* is a topic found in most texts on the linear model or regression analysis, and is, along with Fourier analysis, a form of functional data analysis that has been used in statistics for a long time. As with constant functions, these may often serve as benchmark or reference functions against which spline-based functions are compared.

The basis functions in a monomial basis are the successive powers of t : $1, t, t^2, t^3$ and so on. The number of basis functions is one more than the highest power in the sequence. No parameters other than the interval over which the basis is defined are needed. A basis for cubic polynomials is defined over $[0, 1]$ in R by

```
monbasis = create.monomial.basis(c(0,1), 4)
monbasis = create_monomial_basis([0,1], 4);
```

Be warned that beyond `nbasis = 7`, the monomial basis system functions become so highly correlated with each other that near singularity conditions can arise.

3.4.3 Other Basis Systems

Here are other basis systems available at the time of writing:

- The *exponential basis*, a set of exponential functions, $\exp(\alpha_k t)$, each with a different rate parameter α_k, and created with function `create.exponential.basis`.
- The *polygonal basis*, defining a function made up of straight line segments, and created with function `create.polygonal.basis`.
- The *power basis*, consisting of a sequence of possibly noninteger powers and even negative powers, of an argument t. These bases are created with the function `create.power.basis`. (Negative powers should be avoided if `rangeval`, the interval of validity of the basis set, includes zero.)

Many other basis systems are possible, but so far have not seemed important enough in functional data analysis to justify writing the code required to include them in the `fda` package. However, a great deal of use in applications is made of bases defined *empirically* by the *principal components analysis* of a sample of curves. Basis functions defined in this way are the most compact possible in the sense of providing the best possible fit for fixed K. If one needs a low-dimensional basis system, this is the way to go. Because principal components basis functions, which we call *harmonics*, are also orthogonal, they are often referred to in various fields as *empirical orthogonal functions* or "*eofs*". Further details are available in Chapter 7.

3.5 Methods for Functional Basis Objects

Common tasks like `plot` are called `generic functions`, for which `methods` are written for object of different classes; see Section 2.3. In R, to see a list of generic functions available for basis objects, use `methods(class='basisfd')`.

Once a basis object is set up, we would like to use some of these `generic functions` via `methods` written for objects of class `basisfd` in R or `basis`

in Matlab. Some of the most commonly used generic functions with methods for functional basis objects are listed here. Others requiring more detailed treatment are discussed later. The R function is shown first and the Matlab version second, separated by a `/`.

In R, the actual name of the function has the suffix `.basisfd`, but the function is usually used with its initial generic part only, though you may see some exceptions to this general rule. That is, one types `print(basisobj)` to display the structure of functional basis object `basisobj`, even though the actual name of the function is `print.basisfd`. In Matlab, however, the complete function name is required.

`print/display` The type, range, number of basis functions, and parameters of the functional basis object are displayed. Function `print` is used in R and `display` in Matlab. These are invoked if the object name is typed (without a semicolon in Matlab).

`summary` A more compact display of the structure of the basis object.

`==/eq` The equality of two functions is tested, and a logical value returned, as in `basis1 == basis2` in R or `eq(basis1,basis2)` in Matlab.

`is/isa_basis` Returns a logical value indicating whether the object is a functional basis object. In R the function `inherits` is similar.

In R, we can extract or insert/replace a component of a basis object, such as its `params` vector, by using the component name preceded by `$`, as in `basisobj$params`. This is a standard R protocol for accessing components of a list. In Matlab, there is a separate function for each component to be extracted. Not all components of an object can be changed safely; some component values interlock with others to define the object, and if you change these, you may later get a cryptic error message or (worse) erroneous results. But for those less critical components, which include *container components* `dropind`, `quadvals`, `basisvalues` and `values`, the R procedure is simple. The object name with the `$` suffix appears on the left side of the assignment operator. In Matlab, each reasonable replacement operation has its own function, beginning with `put`. The first argument in the function is the name of the basis object, and the second argument is the object to be extracted or inserted. The names of these extractor and insertion functions are displayed in Table 3.1 in Section 3.6.

It is often handy to set up a matrix of basis function values, say for some specialized plotting operation or as an input into a regression analysis. To this end, we have the basis *evaluation* functions

```
basismatrix = eval.basis(tvec, mybasis)
basismatrix = eval_basis(tvec, mybasis)
```

where argument `tvec` is a vector of n argument values within the range used to define the basis, and argument `mybasis` is the name of the basis system that you have created. The resulting `basismatrix` is n by K. One can also compute the derivatives of the basis functions by adding a third argument that specifies the degree of the derivative, as in

```
Dbasismatrix = eval.basis(tvec, mybasis, 1)
Dbasismatrix = eval_basis(tvec, mybasis, 1)
```

Warning: Do not use the command `eval` without its suffix; this command is a part of the core system in both languages and reserved for something quite different. For example, in R `print(mybasis)` does "methods dispatch," passing `mybasis` to function `print.basisfd`. However, `eval(tvec, mybasis)` does not invoke `eval.basis(mybasis)`.

An alternative in R is the generic `predict` function. With this, the previous two function calls could be accomplished as follows:

```
basismatrix = predict(mybasis, tvec)
Dbasismatrix = predict(mybasis, tvec, 1)
```

In the execution of these two commands, the (`S3`) "methods dispatch" in R searches for a function with the name of the generic combined with the name of the class of the first argument. In this case this process will find `predict.basisfd`, which in turn is a wrapper for `eval.basis`.

There are `predict` methods written for many different classes of objects, which makes it easier to remember the function call. Moreover, for objects with similar functionality but different structure, a user does not have to know the exact class of the object. We use this later with objects of class `fd` and `fdSmooth`, for example.

3.6 The Structure of the `basisfd` or `basis` Class

All basis objects share a common structure, and all of the `create` functions are designed to make the call to the function `basisfd` in R or `basis` in Matlab more convenient. Functions like these two that set up objects of a specific class are called *constructor* functions. The complete calling sequence for `basisfd` in R is

```
basisfd(type, rangeval, nbasis, params,
  dropind=vector("list", 0),
  quadvals=vector("list", 0),
  values=vector("list", 0),
  basisvalues=vector("list", 0))
```

The equivalent Matlab calling sequence lacks specification of default values:

```
basis(basistype, rangeval, nbasis, params, dropind,
      quadvals, values, basisvalues)
```

We include a brief description of each argument here for R users, but you should use the `help` command in either language to get more information.

`type` A character string indicating the type of basis. A number of character sequences are permitted for each type to allow for abbreviations and optional capitalization.

rangeval A vector of length two containing the lower and upper boundaries of the range over which the basis is defined. If a positive number if supplied instead, the lower limit is set to zero.

nbasis The number of basis functions.

params A vector of parameter values defining the basis. If the basis type is "fourier", this is a single number indicating the period. That is, the basis functions are periodic on the interval (0,PARAMS) or any translation of it. If the basis type is bspline, the values are interior knots at which the piecewise polynomials join.

dropind A vector of integers specifying the basis functions to be dropped, if any. For example, if it is required that a function be zero at the left boundary, this is achieved by dropping the first basis function, the only one that is nonzero at that point.

The final three arguments, quadvals, values, and basisvalues, are used to store basis function values in situations where a basis system is evaluated repeatedly.

quadvals A matrix with two columns and a number of rows equal to the number of argument values used to approximate an integral (e.g., using Simpson's rule). The first column contains the argument values. A minimum of five values is required. For type = 'bspline', this is used in each interknot interval, the minimum of 5 values is often enough. These are typically equally spaced between adjacent knots. The second column contains the weights. For Simpson's rule, these are proportional to 1, 4, 2, 4, ..., 2, 4, 1.

values A list, with entries containing the values of the basis function derivatives starting with 0 and going up to the highest derivative needed. The values correspond to quadrature points in quadvals. It is up to the user to decide whether or not to multiply the derivative values by the square roots of the quadrature weights so as to make numerical integration a simple matrix multiplication. Values are checked against quadvals to ensure the correct number of rows, and against nbasis to ensure the correct number of columns; values contains values of basis functions and derivatives at quadrature points weighted by square root of quadrature weights. These values are only generated as required, and only if the quadvals is not matrix("numeric",0,0).

basisvalues A list of lists. This is designed to avoid evaluation of a basis system repeatedly at a set of argument values. Each sublist corresponds to a specific set of argument values, and must have at least two components, which may be named as you wish. The first component in an element of the list vector contains the argument values. The second component is a matrix of values of the basis functions evaluated at the arguments in the first component. Subsequent components, if present, are matrices of values of their derivatives up to a maximum derivative order. Whenever function getbasismatrix is called, it checks the first list in each row to see first if the number of argument values corresponds to the size of the first dimension, and if this test succeeds, checks that all of the argument values match.

The names of the suffixes in R or the functions in Matlab that either extract or insert component information into a basis object are shown in Table 3.1.

Table 3.1 The methods for extracting and modifying information in a `basisfd` (R) or `basis` (Matlab) object

R suffix	Matlab function	
`$nbasis`	`getnbasis`	`putnbasis`
`$dropind`	`getdropind`	`putdropind`
`$quadvals`	`getquadvals`	`putquadvals`
`$basisvalues`	`getbasisvalues`	`putbasisvalues`
`$values`	`getvalues`	`putvalues`

3.7 Some Things to Try

1. Work the examples in the help page for `create.fourier.basis` and `create.bspline.basis`. What do these examples tell you about these alternative basis systems?
2. Generate a B-spline basis. Follow these steps:
   a. Decide on the range, such as perhaps [0,1].
   b. Choose an order, such as four.
   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, this places by default knots at $0, 0.05, 0.10, \ldots,$ $0.90, 0.95$ and 1 over [0,1].
   d. Plot the basis to see how it looks using the `plot` command.
   e. Now evaluate and plot a few derivatives of the basis functions to see how their smoothness diminishes with each successive order of derivative.