MATLAB, R and S-PLUS Functions for Functional Data Analysis

J. O. Ramsay, McGill University

November 24, 2005

Contents

1 Introduction

1.1 The Goals of these Notes

These notes are designed to accompany the books *Functional Data Analysis* (2005) and *Applied Functional Data Analysis* (2002) by J. O. Ramsay and B. W. Silverman. They describe some objects and functions that can be used to implement and simplify various types of functional data analysis, and their use is illustrated with some of the data described there. It is hardly necessary to be familiar with all of the book before using this software. Indeed, a useful strategy in developing a feel for how these techniques work might be to read the introductory chapter, with perhaps some browsing elsewhere, and then use this software to perform some of the analyses, before going on to read in detail.

In these notes I will always abbreviate the noun *functional data analysis* by FDA, and the adjective *functional data* by FD.

Our first objective in developing this software was to help a prospective FDA get off the ground. I have not tried to develop sophisticated algorithms able to deal with almost any eventuality, but rather simple methods that invite extension and other modifications. To this end I have tried to keep the modules small and readable, and I have not strayed much beyond the material and examples in the text. However, it must be admitted that each new application seems to call for an extension of some technique, and things do tend to become more complex as time rolls on.

1.2 The Two Languages

The software has been developed for both the MATLAB language and R/S-PLUS family. In general I will not distinguish between R and S-PLUS as languages, and will mostly refer to the latter. I will, however, mention from time to time special issues for R users. For example, in R the `splines` package must be loaded before using a B-spline basis, whereas in S-PLUS nothing particular needs to be done.

We do not wish to discuss any superiority of either programming language over the other, although to be sure each language has strengths that one must work a bit to implement in the other. But on the whole, the two languages are similar, and I have found that moving code from one to the other has not been difficult.

We have tried to use the same names for functions in both languages, but note that compound names are constructed using an underscore in MATLAB, as in `smooth_basis`, and using a period in R/S-PLUS, as in `smooth.basis`.

We will use typewriter script for names of functions and keywords in either language, as I have already done for `splines` and `smooth_basis`.

1.3 Object-oriented programming

Both languages have a capacity for object-oriented programming. This makes both developing and using these functions much easier, and consequently object-oriented programming has been used throughout. However, because the object-oriented programming capability of MATLAB is fairly rudimentary, I have tried not to use any feature in S-PLUS that could not be reproduced in MATLAB.

What is object-oriented programming? Both languages have the capacity to define a single compound variable that can contain several pieces of information of varying types, such as scalars, vectors, names, strings, and so forth. These are `list` variables (S-PLUS) or `struct` variables (MATLAB).

A *class* is a specification or blueprint for one of these variables that prespecifies its structure. That is, a class specifies a name for a blueprint, called its *type*, and also defines a name for each piece of information in the compound variable. Assigning a class name to a list or `struct` variable allows the language's interpreter to know in advance what its internal elements (S-PLUS) or fields (MATLAB) are. These internal elements are often called *slots* in the literature object-oriented programming.

Objects are specific variables created so as to conform to the blueprint or specification in a specific class.

For example, functions are expressed in functional data analysis as linear combinations of basis functions, as in

$$x(t) = \sum_{k=1}^{K} c_k \phi_k(t)$$

A functional data class specifies a `list` variable in S-PLUS for a `struct` variable in MATLAB that contains at a minimum the two essential pieces of information needed to define a function expressed in this way:

- the basis function system defining the basis functions $\phi_k(t)$
- the vector, matrix, or array containing the coefficients c_k

Consequently, the functional data `class` specifies that a functional data object will have at least these two pieces of information:

- an array of numbers that are the coefficients for basis function expansions, and
- another object called a `basis` object that specifies a basis for the expansion.

The name or type of the functional data blueprint is `fd`, and these two members are called `coef` and `basis`, respectively. More will be said about both the `fd` and `basis` classes below, and these classes and objects that they define are introduced here as a preliminary exposure to these essential ideas.

One consequence of using classes and objects is that the same function name can be used for many different types of things. This is called *overloading* a function name. So, for example, we can provide a `plot` function for functional data objects that works because the interpreter is able to recognize that the argument in a call like `plot(fdobj)` is an object of this class, and then will use the appropriate special purpose plotting function. The user doesn't have to know what that special function is, since the interpreter takes care of this. Consequently one can simply use `plot` over and over again, sometimes for regular variables, sometimes for `fd` objects or `basis` objects, and sometimes for other objects as well.

We will also use overloading for such familiar functions as `mean`, `print`, `var`, and even for operations such as `+/-` and for specifying subsets of arrays. See Section 2 for a more detailed description of objects.

In fact, so important is the object concept, that both languages have come to define every variable in terms of a class, so that even user-defined new variables are automatically defined an appropriate default class attribute if the user does not do so.

Fortunately, there are now many fine books to support the use of both languages and to introduce you to object oriented programming. I recommend that Matlab users not already familiar with the object-oriented features in these languages first read the few chapters on this topic in Hanselman and

Littlefield (2005) (MATLAB). On the R and S-PLUS side there are Venables (2004) and Venables and Ripley (2000) and the extensive manuals and documentation accompanying the distribution of these two languages by the R-Project Group and Insightful Corporation, respectively.

Insightful Corporation has also released their own functional data analysis module, which may be downloaded through their website `www.insightful.com`. This module is supported by Clarkson, Fraley, Gu and Ramsay (2005). This module has important features not available in the set of functions described in this manual. For example, the functions are integrated with the *dataframe* class, making it easy to combine multivariate and functional data analyses and to benefit from the other useful features of dataframes. I recommend to S-PLUS users that the functions described here by supplemented by this module.

1.4 An Overview of the Steps in an FDA

A typical FDA tends to include most of the following steps:

1. The raw data are collected, cleaned, and organized. I assume that there is a one-dimensional argument, that I will denote by t. As a rule functions of t are observed only at discrete sampling values $t_j, j = 1, \ldots, n$, and these may or may not be equally spaced. But there may well be more than one function of t being observed, as would be the case for handwriting data, where there are $X(t)$-, $Y(t)$-, and possibly $Z(t)$-coordinate functions.

   We assume that there may also be replications of each function, indexed by $i = 1, \ldots, N$. Each replicate is referred to as an observation, since we want to treat the discrete values as a unitary whole. While most studies will have the same set of argument values or sampling points t_j for all replications, this is not required, and the more general notation for these values, $t_{ij}, j = 1, \ldots, n_i$, might be required.

2. The data are next converted to functional form. By this is meant that the raw data for observation i are used to define a function x_i that can be evaluated at all values of t over some interval. In order to do this, a *basis* must first be specified, which is system of basic functions which are combined linearly to define actual functions. The data are organized

into a *functional data* object, often using the function `data2fd`, or perhaps the function `smooth_basis` (MATLAB) or `smooth.basis` (S-PLUS), and all of these functions require the specification of a *basis* object.

3. Next a variety of preliminary displays and summary statistics are developed. These can be produced by special plotting and summary functions that use functional data objects as input, such as `plot, mean`, and `var.`

4. The functions may also need to be registered or aligned, in order to have important features found in each curve occur at roughly the same argument values. This process is said to separate vertical *amplitude* variation from horizontal or *phase* variation. I provide both a landmark registration algorithm and a continuous registration algorithm to do this.

5. Exploratory analyses are carried out on the registered data. The main techniques discussed in the book are

   - Principal components analysis (`pca`)
   - Canonical correlation analysis (`cca`)
   - Principal differential analysis (`pca`)

6. Models are constructed for the data. These models may in the form of a functional linear model, using `fRegress`, or in the form of a differential equation, using `pda.`

7. The models are evaluated, often with the help of special plotting and summary functions adapted to the particular analysis.

1.5 An Overview of these Notes

Section 2 describes the essential classes needed to use this software. The two most important are:

- the `basis` class used to define the functional data object, and

- the `fd` class defining objects that contain samples of functional observations, and that are the primary input to the various MATLAB and S-PLUS functions defined in the next section.

A `bifd` class is defined for functions of two variables. Two other classes, the `fdPar` and `Lfd` class, are used in more advanced applications.

Section 3 provides details about the functions that will use these objects to do various functional data analyses. These functions

- create functional data objects by smoothing or interpolating the raw discrete data,
- plot and summarize functional data objects,
- align prominent curve features by registration,
- compute functional versions of elementary statistical descriptions,
- perform exploratory analyses, such as principal components analysis (PCA), canonical correlation analysis (CCA), and principal differential analysis (PDA),
- fit linear models where the independent and/or dependent variables are functional, and
- manipulate functional data objects using such basic arithmetical operations as addition, multiplication, square-rooting, exponentiation, as well as selecting subsets, and so forth.

Descriptions of the functions described in this section, as well as other functions in the package, are also available by using `help`, as in `help(data2fd)` in S-PLUS or `help data2fd` in MATLAB. In both languages, the code itself for each function also contains a fair amount of detail at the beginning describing the purpose of the function, each of the arguments, and the results returned.

Section 4 provides some useful information about installing the package.

Section **??** shows how functional data objects and functions are used to carry out some of the analyses that appear in the text. These examples are not at all exhaustive, but only intended to get you started. For further examples in more sophisticated situations, go to the web site `www.functionaldata.org`.

Section **??** offers some notes specific to monotone smoothing.

1.6 An Important Disclaimer

These notes are *not* updated each time a change is made to a function. Although the instructions on how to use the functions that are in these notes won't have changed too much, it is always wise to compare the notes against what is displayed by the `help` command in the language to be sure that the notes are still up to date.

2 More on FDA Objects

In this section I define the five objects that we shall use in our FDA's. First, I go into more detail on the nature of an object.

2.1 What is an Object in MATLAB or S-PLUS?

An object in S-PLUS is a `list` variable having a `class` attribute. In MATLAB, it is a `struct` variable having a `class` attribute.

A `list` or a `struct`, in turn, is a collection of data structures such as scalars, vectors, matrices, other lists, objects, and so forth, referred to as the *members*, *fields*, or *slots* of the `list` or `struct`.

For example, a `fd` object is a `struct` variable in MATLAB that contains at least two slots: a coefficient matrix, and a `basis` object. In S-PLUS, it is a `list` variable that contains these two elements. The type or name for the class is `fd`. A `basis` object is in turn a `list` variable or a `struct` variable depending on the language that contains (i) a string slot for the type of basis, (ii) a vector slot for the range of the argument, (iii) a scalar slot for the number of basis functions, and (iv) a vector slot for the fixed parameters defining the basis.

It is the presence of a class type or name that turns a `list` or a `struct` into an object. This class name is used by MATLAB or S-PLUS to select an appropriate function from among a collection of possibilities. The class name is, effectively, a guarantee to the language that the `list` or `struct` will have a fixed pre-specified internal structure. That is, given its name, the interpreter will know exactly how many slots there are, have a name for each slot, and what the properties of the information in each slot are. Consequently, the language knows exactly what can and cannot be done with the object.

Note that the term *class* refers to the blueprint for the `list` or `struct`, and the term *object* refers to a specific data structure constructed using this blueprint. To illustrate, "hamburger" is a blueprint specifying that ground beef shall be placed between two round breads, and is therefore like a class, whereas the actual hamburger that you are about to eat is like an object. Because you know in advance what the structure of a hamburger is, you won't expect to pour milk on it.

Each object has a function associated with it that creates the object, called its *constructor* function, and I have used the prefix `create` to indicate such a function. For example, we can create a basis of the Fourier type by using the function `create_fourier_basis` in MATLAB or `create.fourier.basis` in S-PLUS. This ensures that the object has the correct structure.

Essentially the creation process is one of organizing the required information into the required `list` structure, and assigning the class name to the `list`. The creation functions also assign names to the members in the `list` for your convenience when you want to get at them specifically rather than at the whole `list`. The object creation functions can also supply some of the members in the `list` by default, so that you need not necessarily provide all the members that the object requires.

2.2 The three essential classes for FDA

Here, then, are the three most important classes that we will need. There are others, but nearly every FDA will make use of these three classes, and certainly of the first two.

Here I only describe the most essential slots or pieces of information. Later I will cover additional slots that may be specified for more advanced applications.

2.2.1 The `basis` class

Before you can convert raw discrete data into a functional data object with these functions, you must specify a *basis*. A basis is a system of primitive functions that are combined linearly to approximate actual functions. An example is successive powers of an argument t, linear combinations of which form polynomials. A more useful example is the unit function 1 and successive pairs of sine and cosine functions with frequencies that are integer multiples of a base period that make up a Fourier series.

The FDA text used this basis expansion method of defining a function exclusively, even though there are certainly other approaches. This was to impose both a uniformity of approach for reasons of simplicity, and to enable us to use roughness penalty methods. These functions continue this strategy, and at this point you may feel like re-reading Chapters 3, 4 and 5 in the FDA text (second edition).

Thus, a function x_i is represented by a basis function expansion, which is defined by a set of basis functions, $\phi_k, k = 1, \ldots, K$. In this approach, a functional observation x_i is expressed as

$$x_i(t) = \sum_k^K c_{ik}\phi_k(t) \ . \tag{1}$$

When these basis functions ϕ_k are specified, then the conversion of the data into a functional data object involves computing and storing the coefficients of the expansion, c_{ik}, into a coefficient matrix.

As was is indicated in Chapter 3, there are many bases possible, and many considerations to take into account. I provide a number of the more common bases:

- the *Fourier basis*, typically used for periodic data,
- the *B-spline basis*, typically used for non-periodic data,
- the *constant basis*, a single basis function whose value is 1 everywhere, used to define constant functions and to convert ordinary univariate scalar observations into functional data form,
- the *exponential basis*, a set of exponential functions, $e^{\alpha_k t}$, each with a different rate parameter α_k,
- the *polygonal basis*, defining a function made up of straight line segments,
- the *polynomial basis*, consisting of the powers of t: $1, t, t^2, t^3, \ldots$,
- the *power basis*, consisting of a sequence of possibly non-integer powers, including negative powers, of an argument t that is usually required to be positive.

Of these basis functions, the first two are by far the most important, and can be used to carry out almost all analyses described in the book. Each of these functions has its own constructor function, such as `create_bspline_basis` in MATLAB or its counterpart `create.bspline.basis` in S-PLUS.

We also hope that users with special bases in mind that I have not provided will discover from the code how they may add their own basis systems.

In specifying a basis, we must specify four things. That is, there are four slots in the `basis` class:

- the *type* of the basis. This is a string such as 'bspline', 'fourier', 'constant' and so on that names the basis. (Note: S-PLUS uses double quotes for strings.)
- the *range* of argument values, specifying the lower and upper limits on argument values,
- the *number* of basis functions, and
- the *parameter* values defining the basis. The number and meaning of the parameter values will depend on the nature of the basis. For example, a Fourier basis requires only a single positive number indicating the base period, a B-spline basis needs a strictly increasing sequence of knot values, but a constant basis doesn't need any parameters at all.

However, a particular call to a `create` function setting up a basis object may not actually specify all four of these pieces of information, and when unspecified, each of them has a default setting that is then automatically applied.

> *An important technical note for R/S-PLUS programmers:* Unfortunately, these two languages already used a class with the name `basis` well before I began to work on these functions. I only discovered this fact in 2005, which certainly suggests that this matter is unlikely to affect your own work with these functions, since it didn't bother mine for eight years! Nevertheless, it is not cool to use the same class name for two quite different ideas, and in the current FDA code for R/S-PLUS, I use the class name `basisfd` rather than `basis`. This will only affect your work if you begin to program your own functions, and then only if you design new types of bases other than the ones described here. If you want to test whether an object is of the `basisfd` class, you can use the function `is.basis()` that returns `T` if it is and `F` if not. Of course, you may also use `inherits(objectname, "basisfd")` for the same purpose. But, unless you get into basis design, you are unlikely to run into trouble. Normally, you will

only create basis objects using the `create` functions. I will continue to use `basis` as the class name for all three languages in these notes.

Details for each type of basis are given below.

Consequently, a basis object in S-PLUS is a `list` variable with four elements, or in MATLAB a `struct` variable with four slots. These slot names are:

type: This is a string such as: `fourier` or `bspline`. A few variants of these strings will also work, such as `fou` or `bsp`.

rangeval: This is a vector containing two values: the initial and final values of t defining the interval over which a functional data object can be evaluated. This interval need not include all the t_j values associated with the discrete data, and it may extend beyond them, but for sure it must contain enough t_j values to define the basis function expansion (1) properly.

nbasis: This is an integer specifying the number of basis functions to be used in the expansion, indicated by K in (1).

params: A vector containing the parameters defining the basis. The contents of this vector depend on the type of basis.

The first three slots don't vary in type or size for different bases, the the last *params* basis is a vector with a length and meaning that has to be specified separately for each basis type. The details are:

Fourier basis: for a fourier basis, the `params` entry contains only the base period T for the sine/cosine series. The basis functions are:

$$1, \sin \omega t, \cos \omega t, \sin 2\omega t, \cos 2\omega t, \ldots$$

where $\omega = 2\pi/T$. Note that because the constant is included, the number of basis functions, `nbasis`, should be odd if you want to completely allow for arbitrary phase variation. In fact, if an even number is specified in the `create.fourier.basis` function, it is changed to the next odd number. By default, if the period T is not specified, the period is set to the width of the interval defined in the `rangeval` entry.

B-spline basis: for a basis of type `bspline`, the `params` vector contains an increasing *knots* or *break points* defining the B-spline functions. The initial and final knots must be equal to the lower and upper limits in `rangeval` entry, respectively. Note that the *order* of a B-spline basis plus the number of *interior* knots equals the number of basis functions, so that the order (degree of the piece-wise polynomials + 1) of these B-splines will be equal to the value of the `nbasis` plus two entry minus the number of elements in the `params` entry. In MATLAB code, `norder = nbasis - length(params) + 2`. For example, if we use 11 break values $0.0, 0.1, 0.2, \ldots, 1.0$ in the `params` slot, and 13 for the `nbasis` slot, this implies that the order of the spline is 13 + 2 - 11 = 4 = 13 - 9. The order must be between 1 and 20. If the `nbasis` and `params` slots determine a value outside of this range, the `create.bspline.fd` S-PLUS function will terminate with an error message, and so will the MATLAB `create_bspline_fd` function. Order 4 is a frequent choice, implying piece-wise cubic polynomials, and this would mean that `nbasis = length(params) + 2`. If no knot or break values are specified, they are set up to partition the interval defined in `rangeval` into equal-sized parts. If only the number of basis functions is specified in addition to the range, the knots are equally spaced and the order is 4. There is room for inconsistency here, of course, when all four arguments are supplied, and if this happens, the `norder` slot value is ignored.

Constant basis: No parameters are required.

Exponential basis: Each basis function is of the form

$$\phi_k(t) = e^{\alpha_k t}$$

and the `params` elements are the rate constants α_k.

Polygonal basis: Polygonal functions are formed from straight line segments. Strictly speaking, these may also be considered as B-splines of order 2. But because they are so handy, we provide a special class for them. The `params` vector contains the junction points for the line segments. These will usually be the sampling values t_j for the raw data.

Monomial basis: Monomials are the integer powers of $t, 1, t, t^2, t^3, \ldots$. The `params` vector contains the sequence of powers to be used, and if not provided, the sequence $0, 1, 2, \ldots$ is used.

Polynomial basis: This is a minor variate of the monomial basis that may occasionally be handy. It defines basis functions $(t-c)^j, j = 0, 1, 2, \ldots$ where c is a constant that shifts t to a new center. This can be important for fitting polynomials to data where the origin is a long way from the data. The `params` vector just contains the shift constant c.

Power basis: This is designed for positive argument values t only. The parameters are a sequence of powers, which need not be integers and may be negative.

A `basis` object can be set up by calling the generic function `basis` in MATLAB or `basisfd` in S-PLUS described in detail in the next section. But special purpose functions such as `create.bspline.basis` and `create.fourier.basis` (S-PLUS) are generally more convenient, and as the technical note above indicates, it is unlikely that you will have to use these primitive basis creation functions unless you want to program new basis types.

Okay, by now you've figured out that in S-PLUS function names can be divided into sections with a period, and that in MATLAB this is done using an underscore. So you can make the translation into the other system yourself whenever I use either convention. So from now on, I will only give a function name for one of the two languages.

2.2.2 The functional data class: fd

With a basis in hand, we are now ready to actually set up functional data object, or, to keep it short, `fd` objects. An `fd` object consists of a sample of N FD observations. An FD observation, in turn, consists of one or more functions. That is, an FD observation is either scalar- or vector-valued function according to the nature of the data. For example, the Berkeley growth data for girls are a sample of 53 scalar functions, and the gait data involve $N = 39$ bivariate FD observations, each consisting of a function for knee angle and another for hip angle. We speak of the corresponding `fd` object as having two functions, although it has 39 replications, each consisting of two functions. When the function is multivariate, we normally expect that all the function values are measurements of the same quantity, such as angle for the gait data or position for the handwriting data.

An `fd` object is a `list` (S-PLUS) or a `struct` (MATLAB) with the class attribute `fd` that contains three named slots. They are as follows, the names being shown in bold type:

coef: This is either a 1-, 2- or 3-dimensional array depending on whether the functions are scalar- or vector-valued and whether there is only one or more than one replication. The dimensions of this array have meanings as follows:

1. The first dimension corresponds to basis functions in the `basis` entry described below. The length of this dimension must therefore be equal to K in (1) and to the `nbasis` slot in the basis object. (The `basis` object has already been described above). That is, for a specific replicate and function, there must be a coefficient for each basis function.
2. The second dimension corresponds to replications. The length of this dimension is N, the sample size. This may be 1, f course, if there is only a single functional observation involved.
3. The third dimension, if required, corresponds to functions when there are multiple functions of t involved. For example, for the handwriting data in the book, the length of this dimension would be 2 since these data have X- and Y-coordinates.

So, for example, if we use 7 fourier basis functions for the monthly temperature data for the 35 Canadian weather stations described in the text, the `coef` array will be 7 by 35. On the other hand, for the 20 samples of handwriting data, each described by an X- and a Y coordinate, and where we are using 23 B-spline basis functions (say with order 4 and 19 interior knots), the `coef` array will be 23 by 20 by 2.

basisobj: The second slot is the name of a `basis` object that has already been set up by calling one of the `create` functions to provide the expansion or representation of the functions, and that is described already above. *Note: earlier versions of the class used* ***basis*** *as the name of this slot.*

fdnames: This is a `list` in S-PLUS or a `cell` array in MATLAB with three members, each being a string. These members provide labels that are used in plotting and other routines to describe the arguments, the replications, and the function values. The members are:

1. A string used to describe arguments. It might be something like `'time'`, for example. The value of the first member, if provided, would be a character vector providing names for each argument value.
2. A string describing replications. For the weather data, we might use `'Weather stations'`.
3. A string to describe the values of the functions, such as `'Deg C'` for the temperature data. When the functions are multivariate, it is still the case that only a single string is required, since we are usually working with functions that all reflect the same quantity.

A functional data analysis usually begins by constructing a `fd` object by inputting raw discrete data along with the sampling argument values t_j or t_{ij} into the function `data2fd` or the function `smooth_basis` described in the next section. However, other ways of constructing functional data objects are also described there.

In general, vector-valued `fd` objects are only used when the values of the functions all mean the same things, such as angle for the gait data, or spatial coordinates for functions defining spatial position. When functions have different units, use multiple `fd` objects rather than a single vector-valued object.

Our use of the term "object" and "observation" is consistent with how multivariate data are described by software packages such as SAS and SPSS, where an observation can be scalar or multivariate, and corresponds to the row of a data matrix.

2.2.3 The bi-variate functional data class: `bifd`

We will sometimes need to set up functions of two variables. For example, a variance function $v(s,t)$ or a correlation function $r(s,t)$ are functions of two variables. So is the regression function $\beta(s,t)$ in a linear model where both the independent and dependent variables are functions.

An `bifd` object is defined by a single coefficient array, and a `basis` object for each argument s and t. Consequently, the class defining an `fd` object has four slots:

coef: This is either a 2-, 3-, or 4-dimensional array depending on whether the functions are scalar- or vector-valued. The meanings of the dimensions are the same as for `fd` objects, except that the first *two* dimensions now correspond to basis functions. Thus, the dimensions are:

1. The first two dimensions corresponds basis functions in the 'sbasisobj' and 'tbasisobj' members for `bifd list` described below. The length of each dimension must therefore be equal to `nbasis` entry in the corresponding basis object named in the second and third arguments.
2. The third dimension corresponds to replications. The length of this dimension is N, the sample size. This may be, of course, 1, and if there is no third dimension, this is assumed to be the case.
3. The fourth dimension, if required, corresponds to functions when there are multiple functions of s and t. See the description of the **var.fd** function for an example. If there are only either two or three dimensions for the `coef` array, then only one variable is assumed.

sbasisobj: The name of a `basis` object for the first argument s.

tbasisobj: The name of a `basis` object for the second argument t.

bifdnames: A `list` in S-PLUS or a `cell` array in MATLAB with three members with the same specifications as above for the `fd` object.

> *It is unwise to use the name of a class as the name of a variable, and especially in MATLAB. For example, if you use one of the*

"create" functions described in the next section to create a basis object with the name `basis`*, you can make it impossible for MATLAB to find the basis class and consequently to create new basis objects. The error messages that result will not be helpful, either. Similarly, don't use* `fd`*,* `fdPar` *or* `Lfd` *as names for variables.*

3 The more important FDA functions

This section describes a variety of MATLAB and S-PLUS functions that do useful things during the course of an FDA. It is natural to classify these functions in rough correspondence with the steps in an FDA described in Section 1.4:

Object creation functions: These, already alluded to in the previous section, create the four types of objects used as inputs to other functions. Included in these are also two functions for actually computing basis function values.

Data plotting and summary functions: These are for the most part simple; they have the same names as the functions used elsewhere in S-PLUS and MATLAB: `display, plot, print, summary` along with the subset selection operator [ ]. But what they actually do depends on the nature of the object supplied as an argument.

Smoothing functions: These functions smooth a functional data object of the `fd` class. They include functions for estimating strictly positive, strictly monotonic and probability density functions from data.

Registration functions: These are used to register or align functions prior to a subsequent analysis.

FDA functions: These actually perform FDAs such as principal components analysis, linear modeling, canonical correlation analysis, and principal differential analysis.

We now detail the functions used to create the three types of objects defined in Section 2.

3.1 `basis` and `fd` object creation functions

The first group of functions are for creating a `basis` object. The function `basisfd` in S-PLUS or `create_basis_fd` in MATLAB is a generic function for creating any basis, but it is usually more convenient to use one of the specialized basis functions designed to create a basis of a specific type. I first describe four of the basis-specific functions, and then the generic function.

In each description, I first specify the call to the function in the two instances. The first will be for MATLAB, and second for S-PLUS. (So as not to seem discriminating, I shall reverse this order in the subsection title.) In each call, the initial arguments are required, but some of the later ones may be optional. Note that S-PLUS syntax permits the specification of the default value for these optional arguments in the function call, while MATLAB does not. This is one area where S-PLUS is better.

The argument call is followed by a description of the function. This description is broken down into parts, in much the same manner as the documentation conventions used in S-PLUS. These parts are: Purpose, Arguments, Returns, and Examples.

We will not attempt to describe the default values for all the arguments that are optional in order to keep the summary both simple and useful at the same time. To find out the last word, you should look at the code for the function itself, and read the initial comment lines.

Note that where I use single quotes, as in `'bspline'`, for strings, the S-PLUS language officially uses double quotes, `"bspline"`, but actually, in fact, works correctly with single quotes as well.

Keep in mind that the first function name in typewriter font specifies the MATLAB call, and the second the S-PLUS call.

3.1.1 `create.bspline.basis` or `create_bspline_basis`

We begin with the most complex creation function, that for a B-spline basis. If you can wade through this, the other functions will be easy. But then, B-splines are complex structures, and it is precisely this complexity that gives them their versatility and ensures an honoured place in our lexicon of bases.

```
create_bspline_basis(rangeval, nbasis, norder, breaks)
create.bspline.basis(rangeval, nbasis, norder=4, breaks=NULL)
```

Purpose: Create a B-spline `basis` object.

Arguments: **rangeval:** (required) A vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated.

nbasis: (required) An integer variable specifying the number of basis functions

norder: (optional) An integer specifying the order of B-splines. The order of a B-spline is one higher than the degree of its piecewise polynomial segments. The default order is 4, and this defines splines that are piecewise cubic.

breaks: (optional) A vector specifying the break points defining the B-spline. Also called *knots*, these are an increasing sequence of junction points between piecewise polynomial segments. They must satisfy `breaks[1] = rangeval[1]` and `breaks[nbreaks] = rangeval[2]`, where `nbreaks` is the length of `breaks`. There must be at least 3 values in `breaks`.

There is a potential for inconsistency among arguments `nbasis`, `norder`, and `breaks`. It is resolved as follows: If `breaks` is supplied, `nbreaks = length(breaks)`, and `nbasis = nbreaks + norder - 2`, no matter what value for `nbasis` is supplied. If `breaks` is not supplied, but `nbasis` is, `nbreaks = nbasis - norder + 2`, and if this turns out to be less than 3, an error message results. If neither `breaks` nor `nbasis` is supplied, `nbreaks` is set to 21.

Some applications may call for *coincident* knots; that is, a sequence of identical values in `breaks`. For each repeated knot value, a degree of continuity is lost in the spline function at that value.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively. The `params` slot contains the values in the `breaks` argument of the function.

3.1.2 `create.fourier.basis` or `create_fourier_basis`

```
create_fourier_basis(rangeval, nbasis, period)
create.fourier.basis(rangeval, nbasis, period)
```

Purpose: Create a fourier `basis` object.

Arguments: **rangeval:** (required) A vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated.

nbasis: (required) An integer variable specifying the number of basis functions. If this is even, it will be increased by one, since the fourier basis will always involve the constant function plus a number of sine/cosine pairs.

period: (optional) The period of the most slowly varying sin and cosine functions. By default, this is the difference, called `width`, between the values in `rangeval`.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively.

3.1.3 `create.exponential.basis` or `create_exponential_basis`

```
create_exponential_basis(rangeval, nbasis, rate)
create.exponential.basis(rangeval, nbasis, rate=width)
```

Purpose: Create an exponential `basis` object.

Arguments: **rangeval:** (required) A vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated.

nbasis: (required) An integer variable specifying the number of basis functions.

rate: (required) A vector of `nbasis` rate constants λ_j for the basis functions $e^{\lambda_j t}$.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively.

3.1.4 `create.power.basis` or `create_power_basis`

```
create_power_basis(rangeval, nbasis, power)
create.power.basis(rangeval, nbasis, power)
```

Purpose: Create an power `basis` object.

Arguments: **rangeval:** (required) A vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated.

nbasis: (required) An integer variable specifying the number of basis functions.

power: (required) A vector of `nbasis` powers λ_j for the basis functions t^{λ_j}. These powers need not be integers, and may be negative, since it is assumed that t will only be positive.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively.

3.1.5 `create.monomial.basis` or `create_monomial_basis`

```
create_monomial_basis(rangeval, nbasis, power)
create.monomial.basis(rangeval, nbasis, power=0:(nbasis-1))
```

Purpose: Create an monomial `basis` object.

Arguments: **rangeval:** (required) A vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated.

nbasis: (required) An integer variable specifying the number of basis functions.

power: (optional) A vector of `nbasis` nonnegative integer powers j for the basis functions t^j. The default is the power sequence `0, 1, ..., nbasis-1`.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively.

3.1.6 `create.constant.basis` or `create_constant_basis`

```
create_constant_basis(rangeval)
create.constant.basis(rangeval)
```

Purpose: Create a constant `basis` object, containing only one basis function, whose value is always one.

Arguments: rangeval: (required) A vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively.

3.1.7 `create.polygonal.basis` or `create_polygonal_basis`

```
create_polygonal_basis(argvals)
create.polygonal.basis(argvals)
```

Purpose: Create a polygonal `basis` object. This is used where all the information in the original discrete data must be preserved, but converted to functional form. Such a function is a polygonal line, with vertices at the sampling points t_j, and heights at these vertices equal to the corresponding observed values y_j.

Arguments: argvals: (required) A vector of defining the points at which the line segments are joined. These will usually be the sampling points for the raw data.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively.

3.1.8 `basisfd` (R/S-PLUS) or `basis` (Matlab)

```
basis(type, rangeval, nbasis, params)
basisfd(type, rangeval, nbasis, params)
```

You might want to review the "Important technical note for programmers" before reading on here.

Purpose: The generic function for creating a `basis` object. You are not likely to need these functions, unless you get into programming your own basis functions.

Arguments: **type:** (required) A character variable with one of the values `fourier`, `bspline`, `const`, `expon`, `polyg`, `poly`, or `power` indicating the nature of the basis. Some variants of these spellings are allowed.

rangeval: (required) A vector of length 2 containing the initial and final values of argument t defining the interval over which the functional data object can be evaluated.

nbasis: (required) An integer variable specifying the number of basis functions

params: (required) A vector containing the parameters defining the basis functions. See Section 2.2.1 for details.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `basis` class attribute with members as above having names `type`, `rangeval`, `nbasis`, and `params`, respectively.

3.1.9 Additional optional arguments for the basis creation functions

These additional arguments come after the arguments specified above, and extend the ways in which basis objects can be created and used.

The arguments are in same order as they are described here. The first argument `dropind` is used to delete unwanted basis functions from a basis system. The next two permit the use of numerical quadrature approximation of integrals in situations where these integrals must be evaluated many times.

dropind: An integer array that conains a subset of the indices $1, \ldots,$ `nbasis` that indicates basis functions that are to be dropped or not included in the final basis system. This provision is especially helpful for the B-spline basis where a function is desired which takes the value zero at one or both boundaries. The respective boundary splines with indices 1 and `nbasis` are the only basis functions that are nonzero at the boundaries, and dropping them ensures that the resulting spline will go to zero where required. If more than one boundary spline is dropped, the number of derivatives which are also zero can also be controlled. Likewise, if a Fourier basis is desired that is centered on zero, then one can drop the initial constant basis function.

quadvals: A matrix with two columns. The first column contains argument values that are used for computing a numerical quadrature approximation to an integral. The second column contains quadrature weights. For example, Simpson's rule is defined by an odd number (≥ 5 as well) of equally spaced quadrature points and quadrature weights $(1h, 4h, 2h, 4h, ...4h, 2h, 4h, 1h)$ where h is the spacing between adjacent quadrature points.

values: A cell array where the first cell contains the matrix of basis functions evaluated at the quadrature weights in `quadvals`, the second the values of the first derivative of the basis functions, and so on. That is, if m is the highest order of derivative required, the cell array has $m + 1$ cells.

We now turn to two functions that create `fd` objects, one for regular objects, and the other for functions of two arguments, or bivariate fd objects. Here I simply use the class name to create an object of that class.

3.1.10 `fd`

```
fd(coef, basisobj, fdnames)
fd(coef, basisobj, fdnames=defaultnames)
```

Purpose: to create an `fd` object containing functional observations. Note that one would normally do this by a call to the `smooth_basis` and `data2fd` function described below, so that this function may not be needed very often.

Arguments: **coef:** (required) A 2- or 3-dimensional array, the first dimension corresponding to basis functions, the second to replications, and the third, if present, to functions.

basisobj: (required) An object of the `basis` class.

fdnames: (optional) A `list` in S-PLUS or a `struct` in MATLAB of length 3, each member being a string vector containing labels for the levels of the corresponding dimension of the discrete 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'`, and the third is for functions, and is given the default name `'values'`. These default names are assigned in function tt data2fd, which also assigns default string vectors by using the `dimnames` attribute of the discrete data array.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `fd` class attribute containing the coefficient array with the name `coefs`, a basis object with the name `basis`, and a `list` with the name `fdnames`.

3.1.11 `bifd`

```
bifd(coef, sbasisobj, tbasisobj, bifdnames)
bifd(coef, sbasisobj, tbasisobj,
            bifdnames = list(NULL, repnames, NULL ))
```

Purpose: to create a `bifd` object containing bivariate functional observations. This function is not normally needed; it is called within other functions to create bivariate functions such as covariance functions and bivariate regression functions.

Arguments: **coef:** (required) A 2-, 3-, or 4-dimensional array, the first two dimensions corresponding to basis functions, the third to replications, and the fourth, if present, to functions.

sbasisobj: (required) An object of the `basis` class for the first argument.

tbasisobj: (required) An object of the `basis` class for the second argument.

fdnames: (optional) A `list` in S-PLUS or a `struct` in MATLAB of length 3 containing dimension names. See `fd` above for details.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `bifd` class attribute containing the coefficient array with the name `coefs`, a basis object with the name `sbasis`, a `basis` object with the name `tbasis`, and a `list` with the name `fdnames`.

3.2 The `data2fd` function

We now describe a simple technique for creating a functional data object from discrete data. This step represents the discrete data associated with each replication by one or more functions, defined by a basis for the expansion of the functions, along with the coefficients determining the expansion. The result is a set of functions that can be evaluated for any argument value, and which can be manipulated in various ways, such as computing inner products, taking derivatives and so on.

Function `data2fd`, on the other hand, is not primarily intended to smooth the data. This may be left to two more flexible functions,

- `smooth.basis` in S-PLUS or `smooth_basis` in MATLAB that uses a roughness penalty to smooth data. This is the most versatile smoothing function, and intended for more high-end applications where the nature and amount of smooth must be carefully controlled.

- `smooth.fd` in S-PLUS or `smooth_fd` in MATLAB, that has as an argument an FD object that has already been computed.

However, I will wait until I have defined the new `Lfd` and `fdPar` classes before describing these two functions.

Indeed, our philosophy has been to leave considerable roughness in the data, but to apply smoothing methods to quantities that are estimated from the functions, such as eigenfunctions or harmonics in principal components analysis, regression functions in linear modeling, and canonical weight functions in canonical correlation analysis.

First we need to consider how the data corresponding to the discrete sampling times t_j should be set up for input to function `data2fd`.

3.2.1 Format of the Data

The first step in an analysis is to collect, clean and organize the raw data. We assume that the observed data are functions of a one-dimensional argument t, which for ease of reference we shall call “time”. Each function is observed at discrete values t_i, which may or may not be equally spaced. There may well be more than one function of t being observed, for example the separate coordinates of the handwriting data. In any case, there will be replications of the observed function(s).

We shall assume, therefore, that our data are given in the form of a one- two- or three-dimensional array `Y` of data values, and a vector or matrix `argvals` of values of t. If `argvals` is a vector, then it is assumed that all the replications are observed at the same time points. Thus, if only one function is being observed, then, using S-PLUS syntax, `Y[i,j]` contains the value of replication j at time point `argvals[i]`. If multiple functions are observed, then `Y[i,j,k]` contains the value for replication j of function k at time `argvals[i]`. Thus, the first dimension of `Y` corresponds to discrete times of observation, the second, if required, to replications, and the third, if required, to functions or variables. For example, if we set up the gait data in this way, where there are 20 sampling times, `Y` will be 20 by 39 by 2.

If not all replications are observed at every time point, then missing values can be coded as `NA` (S-PLUS) or `NaN` (MATLAB). If the replications are observed at varying time points, then `argvals` should be supplied as a matrix, with `argvals[i,j]` being the time point at which `Y[i,j]` or `Y[i,j,k]` is observed. If the number of argument values varies from one replication to another, the rows of `argvals` should be padded out with `NA`s in S-PLUS or `NaN`'s in MATLAB. If any `argvals[i,j]` is coded as missing, then the corresponding entry or members in `Y` is not used.

Names can be supplied for each for each dimension of the data. By default, these are the strings `time`, `replications` and `variables`.

3.2.2 `data2fd`

```
data2fd(y, argvals, basis, fdnames)
data2fd(y, argvals, basis, fdnames = defaultnames)
```

Purpose: This function converts an array 'y' of function values plus an array 'argvals' of argument values to a functional data object. This a function that tries to do as much for the user as possible. This includes selecting a basis, if one is not provided.

Arguments: **y:** (required) An array containing sampled values of curves. If y is a vector, only one replicate and variable are assumed. If y is a matrix, rows must correspond to argument values and columns to replications or cases, and it will be assumed that there is only one variable per observation. If y is a three-dimensional array, the first dimension (rows) corresponds to argument values, the second (columns) to replications, and the third (layers) to variables within replications. Missing values are permitted, and the number of values may vary from one replication to another. If this is the case, the number of rows must equal the maximum number of argument values, and columns having fewer values must be padded out with missing value codes.

argvals: (required) A set of argument values. If this is a vector, the same set of argument values is used for all columns of y. If 'argvals' is a matrix, the columns correspond to the columns of y, and contain the argument values for that replicate or case.

basisobj: (required) Either: A `basis` object created by function `basis` in Matlab or `basisfd` in R/S-PLUS, or a missing value, in which case a `basis` object is set up by the function using the values of the next three arguments.

fdnames: (optional) A `list` in S-PLUS or a `cell` array in MATLAB of length 3, each member being a string vector containing labels for the levels of the corresponding dimension of the discrete data. The first member is a name is for argument values, and is given the default value `'time'`, the second is for replications, and is given the default name `'reps'`, and the third is for functions, and is given the default name `'values'`. These default names are assigned in function `data2fd`, but the S-PLUS version can also assign default string vectors by using the `dimnames` attribute of the discrete data array.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `fd` class attribute containing the coefficient array with the name `coef`, a `basis` object with the name `basis`, and a `list` with the name `fdnames`. If `periodic` is T, the basis is of type `fourier`, otherwise it is either of type `bspline` or `polygonal`. It is of `polygonal` type if `nresol=length(argvals)` and `nderiv=0`; otherwise it is of type `bspline`.

3.3 Two more classes for smoothing

Many applications require more control over the smoothing process than the function `data2fd`, which simply uses ordinary least squares approximation to estimate coefficients.

Our preferred approach for more sophisticated smoothing is the *roughness penalty* or *regularization* method. This method requires the definition of a penalty on the roughness of a smooth and a smoothing parameter that controls the degree of smoothing.

First, we have to define a class that is used to define a flexible family of roughness penalties.

3.3.1 The `Lfd` class

The most common type of penalty is defined in terms a integrated squared derivative. The classic *c*ubic smoothing spline is defined by the penalty

$$\int [D^2 x(t)]^2 \, dt.$$

This defines roughness as the total squared curvature of the fitting function x. More generally, we may use

$$\int [D^m x(t)]^2 \, dt, \quad m \geq 2,$$

if we want to define roughness as the total squared curvature of the derivative of order $m - 2$.

However, even more sophistication in the definition of roughness can be obtained by defining a *linear differential operator* of the form

$$Lx = \beta_0 x + \beta_1 Dx + \ldots + \beta_{m-1} D^{m-1} x + D^m x$$

where the m *weight functions* $\beta_j, j = 0, \ldots, m - 1$, may be either constants or themselves functions. The textbook and the examples that are distributed with this code make heavy use of the operator

$$Lx = \omega^2 Dx + D^3 x$$

for situations where the data are periodic with period $2\pi/\omega$ and we want to smooth towards a vertically shifted sinusoid.

The Lfd class is created with the following functions

```
Lfd(m, bwtlist)
Lfd(m, bwtcell)
```

Purpose: This function defines a linear differential operator object.

Arguments: **m:** (required) A nonnegative integer specifying the degree of the operator. This is the highest order of derivative used in the operator.

bwtlist or bwtcell: (optional) A list in S-PLUS or a cell array in MATLAB of length m containing either

- functional data objects or
- functional parameter objects (described next).

These objects define the weight function β_j that are used to define the operator. If this argument is not present, the operator is simply D^m.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `Lfd` class attribute containing the specification of the linear differential operator.

3.3.2 The `fdPar` class

In functional data analysis, functions are often the result of smoothing data where a roughness penalty is employed, or are a result of estimating a functional parameter where a roughness penalty is used to control its smoothness. In fact, a function for smoothing data is just a special case of a functional parameter.

When we estimate functional parameters, we need to specify at least some of the following four characteristics:

- The functional data object itself, consisting of its basis and a coefficient vector. The latter is important where the estimation is iterative and an initial value of the functional parameter is required to start off the iterations.
- The linear differential operator defining the roughness penalty.
- The smoothing parameter λ.

- A binary variable indicating whether the functional parameter is to be estimated (1), or is to be held fixed (0). For example, we may want to estimate some of the regression functions in a functional linear regression, and hold others fixed.

We can see that the functional parameter class *inherits* from the functional data class by adding to it extra slots or attributes.

The `fdPar` class is created with the following functions

```
fdPar(fdobj, Lfdobj=0, lambda=0, estimate=1)
fdPar(fdobj, Lfdobj, lambda, estimate)
```

Purpose: This function defines a functional parameter object.

Arguments: **fdobj:** (required) A functional data object.

Lfdobj: (optional) An object of the `Lfd` class defining a linear differential object. Alternative, a nonnegative integer may be supplied. An integer is not a `Lfd` object, but it is converted to the D^m operator inside the function. If this argument is not supplied, zero is assumed.

lambda: (optional) A nonnegative real number that defines the smoothing parameter λ that multiplies the roughness penalty and controls the degree of smoothness. If not supplied, zero is assumed.

estimate: (optional) If `T` in S-PLUS or a positive number in MATLAB, the function is to be estimated. If `F` in S-PLUS or zero in MATLAB, the function is to be held fixed. The default is `T` or 1, respectively.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the `fdPar` class attribute containing the specification of the functional parameter object.

3.4 Smoothing data using a roughness penalty

The roughness penalty method or regularization is used for the smoothing process by the two functions in this section. At this point, it would be worthwhile going through Chapters 4 and 5 in the text to appreciate the concepts involved.

The function `data2fd` computes the least squares approximation to the data $y_{ij}, j = 1, \ldots, n$ corresponding to a specific function x_i by minimizing

$$\texttt{SMSSE}(\mathbf{y}_i, \mathbf{c}) = \sum_{j=1}^{n} [y_{ij} - \sum_{k=1}^{K} c_{ik}\phi_k(t_j)]^2. \tag{2}$$

In this expression the coefficients c_{ik} determine the expansion, and the fitting criterion `SMSSE` is minimized with respect to these. The expression also has the possibility of weighting data values differently through a choice of weights w_j. We can control the smoothness of the fit by our choice of K; the smaller K, the smoother the fit, and the larger K, the closer the fit will be to the data. The functional observation is then

$$x_i(t) = \sum_{k=1}^{K} c_{ik}\phi_k(t).$$

However, there are several important advantages to further smoothing or regularizing the function x_i by attaching to the least squares fitting criterion an additional term that controls the roughness of some derivative of the fit, a process called *regularization.*

We regularize the fit to the data vector $\mathbf{y}$ by minimizing the criterion

$$\texttt{PENSSE} = \texttt{SMSSE}(\mathbf{y}, \mathbf{c}) + \lambda\texttt{PEN}(x) \tag{3}$$

where the second term on the right side penalizes some form of roughness in x. For example, we can use the criterion

$$\texttt{PEN}(x) = \int [D^2 x(t)]^2 \, dt, \tag{4}$$

which measures the roughness of the function x by integrating the square of its second derivative $D^2 x$, called the total curvature of x. The more wiggly x is, the larger this term will be.

The smoothing parameter λ plays a key role. The larger λ, the more heavily roughness in x is penalized, and ultimately as λ increases without limit, x is forced towards a straight line, for which the second derivative is everywhere 0. On the other hand, as λ is reduced to zero, the roughness of x matters less and less, and finally when $\lambda \to 0$, x will be just as rough as $\mathbf{y}$ since it will pass exactly through the data points.

Why consider regularization? First, it gives us much finer control over the smoothness of fit. We can even use more basis functions than data values, and still achieve a smooth fit! Without regularization, on the other hand, a smooth fit often means sacrificing important variation in x in places where it is needed.

Also, we may want to get a good derivative estimate, a critical consideration for a number of the displays and analyses described in the book. For this purpose, we may choose to penalize a higher order derivative. For example, if we wanted to get a good acceleration estimate (D^2x), we might penalize the size of D^4x, thereby controlling the curvature in the acceleration function. Getting a good derivative estimate can be difficult without regularization.

It is shown in Chapter 5 that an equivalent expression for the penalty term, PEN, is

$$\texttt{PEN} = \lambda \mathbf{c}'\mathbf{R}\mathbf{c}.$$

The order K matrix $\mathbf{R}$ is called the *penalty matrix*, and, as before, vector $\mathbf{c}$ contains the coefficients of the basis expansion.

The FDA functions permit wider range of roughness penalties than the two mentioned above, namely integrating the square of D^2x or of D^4x. We can also penalize the square of the result of applying any *linear differential operator* L to x. A linear differential operator is a weighted combination of derivatives, and has the following structure:

$$Lx(t) = \beta_0(t)x(t) + \beta_1(t)Dx(t) + \ldots + \beta_{m-1}(t)D^{m-1}x(t) + D^m x(t)\ . \quad (5)$$

Integer m is the *order* of the linear differential operator L, and each of the m functions $\beta_j(t), j = 0, \ldots, m-1$ apply a weight that may vary over argument t to the derivative of order j. We see that $Lx = D^2x$ is a special case in which the order is 2 and the two weight functions are $\beta_0 = \beta_1 = 0$.

The regularization penalty (4) then becomes

$$\texttt{PEN}(x) = \int [Lx(t)]^2\, dt. \quad (6)$$

The reason for considering this wider family of penalties that by the appropriate choice of L, we can force the smooth as $\lambda \to \infty$ to be toward a linear combination of m functions u_j that we choose. The choice $L = D^2$ smooths toward a linear combination of $u_1 = 1$ and $u_2 = t$, for example. One might call this wider choice of penalties a *designer smooth* in the sense that we customize what we choose to call *smooth*. Examples are given in the book, and more technical detail is available in Heckman and Ramsay (2000).

Now when we look at the structure of (5), we see that it can be defined by a functional data object having m replications, with observations $\beta_0, \beta_1, \ldots, \beta_{m-1}$. To define the operator, all we have to do is to choose a suitable basis for expanding these functions to the desired level of accuracy, set up a matrix $\mathbf{Y}$ of values of these functions at a fine mesh of sampling points t_j, and input this matrix, this set of sampling points, and the basis into function `data2fd`.

Thus, in all functions using a roughness penalty, the argument `fdParobj` appears. One of the members of the `fdPar` class is an object of the class `Lfd`, and this is allowed to be of two types: an integer such as 2, in which case the penalty is defined to be of the form (4), or a `Lfd` object, in which case the penalty is of the form (6). The order m of the differential operator is then determined by the number of functions in the `Lfd` object in argument `fdParobj`. As I indicated earlier, the `fdParobj` also contains the smoothing parameter λ as one of its members.

We now give the specifications for the smoothing functions.

3.4.1 `smooth.basis` or `smooth_basis`

The following function permits the direct smoothing of the raw discrete data, if this seems desirable. It also offers the possibility of variable weighting of the discrete observations. However, it lacks the capability of dealing with missing data or with argument values that vary from observation to observation that is available in `data2fd`.

```
smooth_basis(argvals, y, fdParobj, wtvec, fdnames)
smooth.basis(argvals, y, fdParobj, wtvec=rep(1,n),
             fdnames=list(NULL, dimnames(y)[2], NULL))
```

Purpose: Smooths the discrete data in argument `y`, sampled at argument values in `argvals`, and returns a functional data object containing the smooth functions.

Arguments: **argvals:** (required) A set of argument values, assumed to be common to all replicates.

y: (required) An array containing values of curves. If the array is a matrix, rows must correspond to argument values and columns to replications, and it will be assumed that there is only one variable per observation. If `y` is a three-dimensional array, the first dimension corresponds to argument values, the second to replications, and the third to variables within replications. If `y` is a vector, only one replicate and variable are assumed.

fdParobj: (required) Either

- object of the `fdPar` class containing as a member an `fd` object which in turn contains the basis to be used for expanding the functions, or
- a `basis` object that directly specifies the basis to be used. In this case, λ will be taken to be zero, and the coefficients will be estimated by least squares estimation.

wtvec: (optional) A vector of positive weights for the discrete values.

fdnames: (optional) A `list` in S-PLUS or a `struct` in MATLAB of length 3 with members containing 1. a single name for the argument domain, such as "Time" 2. a vector of names for the replications or cases 3. a name for the function, or a vector of names if there are multiple functions.

Returns: A `list` in S-PLUS or a `struct` in MATLAB object in S-PLUS containing the following information:

fd: An FD object

df: A degrees of freedom measure

gcf: A generalized cross-validation measure of lack of fit that discounts fit for the degrees of freedom used to achieve it. It is often suggested that a good value of the smoothing parameter is one that minimizes this measure. A GCV value is returned for each curve that is smoothed, and for each function as well if the curves are multivariate.

coef: The estimated coefficient vector, matrix, or array depending on the number of curves and whether or not the functions are multivariate.

SSE: The error sum of squares summed across sampling points and, if more than one curve is involved, across curves, and, if more than one function is involved, also across functions.

PENMAT: The penalty matrix $\mathbf{R}$.

Y2CMAP: The matrix $\mathbf{S}_\lambda$ mapping the data to the coefficients for each curve.

3.4.2 `smooth.fd` for `smooth_fd`

This is a function designed to smooth a set of functional data objects. That is, the discrete data have typically already been processed by `data2fd` or `smooth_basis` to produce a `fd` object, and now one wants to impose additional smoothness on the objects. The smoothed versions of these objects may retain the same basis as the originals, or they may use a new basis.

```
smooth_fd(fd, fdParobj, rebase)
smooth.fd(fd, fdParobj, rebase = T)
```

Purpose: Smooth the functions in a functional data object by the roughness penalty or regularization method, and return a functional data object containing the smooth functions. The functional data objects to be smoothed will usually have already been created by `data2fd`.

Arguments: **fd:** (required) A functional data object to be smoothed.

fdParobj: (required) Either

- object of the `fdPar` class containing as a member an `fd` object which in turn contains the basis to be used for expanding the functions, or
- a `basis` object that directly specifies the basis to be used. In this case, λ will be taken to be zero, and the coefficients will be estimated by least squares estimation.

rebase: (optional) If T or nonzero, and the basis type is `polygonal`, then the basis is changed to a cubic bspline basis before smoothing.

Returns: A functional data object.

3.5 Functions for Constrained Smoothing

It can happen that we require a smoothing function to satisfy certain constraints. Among these are: (1) that the function be strictly positive, (2) that the function be strictly increasing or monotonic, and (3) that the function be a probability density function (i. e. strictly positive and unit area under the function.) Just using the standard smoothing functions above will often not work because there is no provision in them for forcing the functions to be constrained in any way.

In each of these cases, we have a special purpose smoothing function that smooths the discrete data with a function that satisfies these constraints. Each each case, also, the constrained function is defined by a transformation of an unconstrained function that is a standard `fd` object. That is, that is represented by a basis function expansion. Moreover, each of these objects can also be smoothed by applying a roughness penalty.

Because the actual fit to the data is no longer a linear combination of known basis functions, but rather a transformation of a `fd` object, the computation requires iterative methods for optimizing a measure of fit. This inevitably implies considerably more computation time. It also implies that an initial estimate of the `fd` object must be supplied as an argument. This can usually be an object that has all coefficients equal to zero.

Note: Each of these functions smooths only a single set of discrete data, and returns a `fd` object that is a single observation. When multiple observations are involved, these functions must be called repeatedly. These functions are not designed for multivariate functional observations.

3.5.1 Positive Smoothing with `smooth.pos` and `smooth_pos`

In this case the fit to the data is of the form $x(t) = \exp[W(t)]$ where $W(t)$ is represented by a `fd` object. The following functions do the job.

```
smooth_pos(argvals, y, fdParobj, wt, conv, iterlim, dbglev)
smooth.pos(argvals, y, fdParobj, wtvec=rep(1,n),
          conv=1e-4, iterlim=20, dbglev=1)
```

Purpose: Smooths the discrete data in argument `y` sampled at argument values in `argvals`, and returns a functional data object `Wfd`.

Arguments: **argvals:** (required)A set of argument values.

y: (required) An array containing values to be smoothed for a single functional observation.

fdParobj: (required) A functional parameter or `fdPar` object that contains an initial estimate `Wfd0` of the function $W(t)$ that is exponentiated to produce the positive smoothing function. This will often be the zero function. `fdParobj` may also contain a linear differential operator object and a smoothing parameter value to control the roughness of $W(t)$.

wt: (optional) A vector of positive weights for the discrete values. By default these values are all one's.

conv: (optional) A small positive constant that controls the level of convergence of the fitting criterion that is required in the numerical optimization. The default is 0.0001.

iterlim: (optional) The maximum number of iterations allowed. The default is 20.

dbglev: (optional) An integer controlling the amount of information displayed for each iteration. By default only the iteration number, fitting criterion value, and the gradient length are displayed.

Returns: A `list` in S-PLUS or a `struct` in MATLAB object in S-PLUS containing the following information. For MATLAB, each of the objects is returned separately and in the following order.

Wfdobj: The functional data object defining converged estimate of the function $W(t)$. Remember that the fit to the data is defined by $\exp(W(t)$, so that object `Wfdobj` is in fact the natural logarithm of the fit.

Flist: List object in S-PLUS or a struct object in MATLAB containing

1. `Flist$f`, the final log likelihood ,
2. `Flist$norm`, the final norm of gradient.

iternum: the number of iterations.

iterhist: , a matrix containing results for each iteration.

3.5.2 Monotone Smoothing with `smooth.monotone` and `smooth_monotone`

In this case the fit to the data is of the form

$$x(t) = \mathbf{z}'\boldsymbol{\beta}_0 + \beta_1 \int_0^t \exp[W(u)]\,du \tag{7}$$

where $W(t)$ is represented by a `fd` object. That is, a monotone smooth can be represented as the indefinite integral of a positive function, defined by $\exp[W(t)]$, multiplied by a nonzero constant β_1, plus a constant. The constant term, defined by $\boldsymbol{\beta}_0$, is permitted to be a linear combination of a set of covariate values in vector $\mathbf{z}$, in which case $\boldsymbol{\beta}_0$ is a vector of the same length containing the regression coefficients.

The following functions do the job. They are set up in very much the same way as the functions for positive smoothing given above.

```
smooth_monotone(argvals, y, fdParobj, zmat, wt,
                conv, iterlim, dbglev)
smooth.monotone(argvals, y, fdParobj,
                zmat=matrix(1,n,1), wt=rep(1,n),
                conv=1e-4, iterlim=20, dbglev=1)
```

Purpose: Smooths the discrete data in argument `y` sampled at argument values in `argvals`, and returns a functional data object defining a strictly positive function that fits the data.

Arguments: argvals: (required)A set of argument values.

- **y:** (required) An array containing values to be smoothed for a single functional observation.
- **wt:** (optional) A vector of positive weights for the discrete values. By default these values are all one's.
- **fdParobj:** (required) A functional parameter or `fdPar` object that contains an initial estimate `Wfd0` of the function $W(t)$ that is exponentiated to produce the positive smoothing function. This will often be the zero function. `fdParobj` may also contain a linear differential operator object and a smoothing parameter value to control the roughness of $W(t)$.

zmat: (optional) A matrix of covariate values with a row for each discrete value to be smoothed and a column for each covariate. By default this is a column of one's.

wt: (optional) A vector of positive weights for the discrete values. By default these values are all one's.

conv: (optional) A small positive constant that controls the level of convergence of the fitting criterion that is required in the numerical optimization. The default is 0.0001.

iterlim: (optional) The maximum number of iterations allowed. The default is 20.

dbglev: (optional) An integer controlling the amount of information displayed for each iteration. By default only the iteration number, fitting criterion value, and the gradient length are displayed.

Returns: A `list` in S-PLUS or a `struct` in MATLAB object in S-PLUS containing the following information. For MATLAB, each of the objects is returned separately and in the following order.

Wfdobj: The functional data object defining converged estimate of the function $W(t)$. Remember that the fit to the data is defined by $\exp(W(t)$, so that object `Wfdobj` is in fact the natural logarithm of the fit.

Flist: List object in S-PLUS or a struct object in MATLAB containing

1. `Flist$f`, the final log likelihood ,
2. `Flist$norm`, the final norm of gradient.

iternum: the number of iterations.

iterhist: , a matrix containing results for each iteration.

3.6 Summary, Evaluation and Plotting Functions

We now detail the functions used to display and summarize functional data objects.

These *inherit* the possible arguments of their more generic counterparts. For example, we have a function called `plot.fd`, to be described below, but in fact, you only need to type `plot(fd)` to invoke this special-purpose function for plotting functional data objects in the `fd` class. This means that you don't have to remember the extension following the ".", and you can expect these functions to do pretty much the same thing as their more familiar counterparts. Moreover, optional arguments such as "type, lty, xlab, ylab, main," and etc. in S-PLUS can also be included in the call.

Also provided is function `eval.fd` for evaluating a functional data object at specified argument values. This can be useful for customizing plots and other applications where these plotting functions don't do the job required.

3.6.1 `plot.fd`, `plot`

These functions are designed to plot functional data objects or their derivatives, either replication by replication, or all replications simultaneously.

```
plot(fd, Lfd, matplt, href, nx)
plot.fd(fd, Lfd=0, matplt=T, href=T, nx=101, ...)
```

Purpose: To plot a functional data object, or one of its derivatives.

Arguments: fd: (required) A functional data object; that is, a `list` with the `fd` class attribute.

Lfd: (optional) Either an integer of value 0 or higher, or an `fd` object. If an integer, it specifies the order of derivative to be evaluated, 0 meaning the functions themselves. If it is a functional data object, the functions are taken to be weight functions defining a linear differential operator, and the order of the operator is equal to the number of functions.

matplt: (optional) A logical variable. If the value is `T` in S-PLUS or nonzero in MATLAB, all the functions are plotted simultaneously using the function `matplot`. If the value is `F` or zero, respectively, the plot is interactive: each function is plotted in turn, and a mouse-click is required to advance to the next plot.

href: A logical variable. If the value is `T` in S-PLUS or nonzero in MATLAB, a horizontal dotted line is plotted through 0 on the ordinate.

nx: The number of points at which the functions are to be evaluated for plotting. For fairly smooth functions, 101 values are usually enough, but for functions with a lot of fine detail, this may need to be increased.

...: (S-PLUS only) The additional arguments for controlling the plot available in the regular function `plot`.

Returns: none

3.6.2 `cycleplot.fd` or `cycleplot`

```
cycleplot(fd, matplt, nx)
cycleplot.fd(fd, matplt=T, nx=101, ...)
```

Purpose: Plot a periodic bivariate functional data object, or one of its derivatives, as a set of cycles. item[Arguments:]

fd: (required) A functional data object containing bivariate functions, that is, taking on two types of values. The basis must be of type `fourier`.

matplt: (optional) A logical variable. If the value is `T`, all the functions are plotted simultaneously using the function `matplot`. If the value is `F`, the plot is interactive: each function is plotted in turn, and a mouse-click is required to advance to the next plot.

nx: (optional) The number of points at which the functions are to be evaluated for plotting. For fairly smooth functions, 101 values are usually enough, but for functions with a lot of fine detail, this may need to be increased.

...: S-PLUS only) The additional arguments for controlling the plot that are available in the regular function `plot`.

Returns: none

3.6.3 `lines.fd` or `line`

```
line(fd, Lfd, nx)
lines.fd(fd, Lfd=0, nx=101, ...)
```

Purpose: Similar to `plot.fd` or `plot`, but this adds function plots to an existing plot.

Arguments: **fd:** (required) A functional data object; that is, a `list` with the `fd` class attribute.

Lfd: (optional) Either an integer of value 0 or higher, or an `Lfd` object. If an integer, it specifies the order of derivative to be evaluated, 0 meaning the functions themselves. If it is an `Lfd` object, it defines a linear differential operator.

nx: (optional) The number of points at which the functions are to be evaluated for plotting.

...: (S-PLUS only) The additional arguments for controlling the plot available in the regular function `lines`.

Returns: none

3.6.4 `print.fd` or `display`

```
display(fd)
print.fd(fd, ...)
```

Purpose: Print a functional data object. The usual method for printing an array is used for the "coefs" argument, and the characteristics of the basis also printed.

Arguments: **fd:** (required) A functional data object; that is, a `list` with the `fd` class attribute.

...: (S-PLUS only) The additional arguments for controlling the plot available in the regular function `print`.

Returns: none

3.6.5 `summary.fd` (S-PLUS only)

```
summary.fd(fd,...)
```

Purpose: Summarize a functional data object. The dimensions of the "data" array are printed, along with the characteristics of the `basis` object.

Arguments: **fd:** (required) A functional data object; that is, a `list` with the `fd` class attribute.

Returns: none

3.6.6 `eval.fd` or `eval_fd`

These functions evaluate a functional data object for each of a strictly increasing set of values.

```
eval_fd(evalargs, fd, Lfd)
eval.fd(evalargs, fd, Lfd=0)
```

Purpose: To evaluate a functional data object at specified argument values.

Arguments: Note that the first two arguments may be interchanged.

- **evalargs:** (required) A vector of argument values at which the functions in the functional data object are to be evaluated.
- **fd:** (required) A functional data object; that is, a `list` with the `fd` class attribute.
- **Lfd:** (optional) Either an integer of value 0 or higher, or an `fd` object. If an integer, it specifies the order of derivative to be evaluated, 0 meaning the functions themselves. If it is a functional data object, the functions are taken to be weight functions defining a linear differential operator, and the order of the operator is equal to the number of functions.

Returns: An array of 2 or 3 dimensions containing the function values. The first dimension corresponds to the argument values in "evalargs", the second to replications, and the third if present to functions.

There are also functions `eval.monfd` and `eval_monfd` that are set up in the same way that will evaluate a strictly monotonic function defined by a functional data object. These functions do not apply the multiplier β_1 or the constant term defined by β_0 in (7), however.

3.6.7 `eval.bifd` or `eval_bifd`

```
eval_bifd(sevalarg, tevalarg, bifd, sLfd, tLfd)
eval.bifd(sevalarg, tevalarg, bifd, sLfd = 0, tLfd = 0)
```

Purpose: To evaluate a bivariate functional data object at specified argument values s and t.

Arguments: Note that the first three arguments may also occur in the order `bifd, sevalarg, tevalarg`.

sevalarg: (required) A vector of argument values for the first argument s of the functions in the functional data object that are to be evaluated.

tevalarg: (required) A vector of argument values for the second argument t of the functions in the functional data object that are to be evaluated.

bifd: (required) A bivariate functional data object; that is, a `list` (S-PLUS) or `struct` (MATLAB) with the `bifd` class attribute.

sLfd: (optional) An integer of value 0 or higher, or a linear differential operator or `Lfd` object. This specifies the order of derivative with respect to the first argument s that is to be evaluated, 0 meaning the functions themselves.

tLfd: (optional) The same as argument sLfd, but that specifies the linear differential operator for the second argument.

Returns: An array of 2, 3, or 4 dimensions containing the function values. The first dimension corresponds to the argument values in “sevalarg”, the second to argument values in “tevalarg”, the third if present to replications, and the fourth if present to functions.

3.7 Data Manipulation Functions

In addition to the display and summary functions mentioned above, it is also possible to perform various manipulations of functional data objects. These include subsetting, the elementary arithmetic operations, and taking derivatives.

3.7.1 Subsets of Functional Data Observations

If you want to plot, print, or summarize only a portion of the data, you will want to select a subset of the replications or variables, just as you can do for rows and columns of matrices. Note that there are either 1 or 2 indices in the function call depending on the number of dimensions of the "coefs" array. For example, if there are multiple replications and multiple functions, `fd[2,]` in S-PLUS or `fd(2,:)` in MATLAB selects the second replication and all functions, and `fd[,1:2]` or `fd(:,1:2)` selects all replications and the first two functions in S-PLUS and MATLAB, respectively. If there is only one function, then `fd[1:10]` or `fd(1:10)` would select the first ten replications.

3.7.2 Arithmetical Operations

Functional data objects can be added, subtracted, multiplied, and divided. Moreover, for each of these operations, either argument may be a scalar rather than a functional data object. Thus, arithmetic for functional data objects behaves much like that for matrices.

Indeed, adding and subtracting involve just adding and subtracting the coefficient matrices. This means that the `fd` objects must have the same basis.

In the case of multiplication and division, this is performed by evaluating the two objects on a fine grid, performing the operation on the values, and creating a new functional data object from these values. The basis used for the first argument is used for the result. The operations

```
sqrt.fd(fd), deriv.fd(fd), fd^power
```

are also constructed in this way. It is up to the user to ensure that these operations can actually be carried out. For example, you must be sure that the denominator `fd` object is nowhere zero.

3.8 Registration Functions

In important initial step in a functional data analysis can be the lining up of salient features of the functions, a process called registration. This is described in Chapter 7.

3.8.1 Landmark Registration

The simplest registration process to understand and to implement is landmark registration, in which we specify the argument values associated with each feature for each curve. In addition, we specify the same values for some standard or reference curve. This is often the mean curve, but it may be some specific curve judged to be especially typical that we want to serve as our “gold standard”.

In landmark registration, we warp time for each curve so that, with respect to this warped time, denoted by $h(t)$, the timing of the features are identical to those for the reference curve. That is, if t_{0f} indicates the reference curve timing for landmark number f, and t_{if} is the corresponding timing for curve i, then we require that

$$h_i(t_{0f}) = t_{if}$$

where h_i is the warping function for this curve.

We assume here that these landmark timings are all in the interior of the interval over which the curves are observed. The ends of the interval serve automatically as landmarks, and do not have to be included.

The following function `landmarkreg` has as arguments an `fd` object for the curves, an `fd` object for the reference curve, a matrix with a row for each curve and a column for each landmark containing the landmark values t_{if} for the curves, and a vector containing the landmark timings for the reference curve.

The function estimates the warping functions using the S-PLUS standard function `smooth.spline`. A fifth optional parameter is available to control the amount of smoothing used in the spline fitting. Note that it is vital that the warping functions be strictly monotonic, and if any estimated warping function fails this condition, a warning message is output. In this event, the registration should be repeated with a larger value of the smoothing parameter.

```
landmarkreg(fd, ximarks, x0marks, WfdParobj, monwrd)

landmarkreg(fd, ximarks, x0marks=xmeanmarks,
            WfdParobj, monwrd=F)
```

Purpose: To register curves using landmarks.

Arguments: **fd:** A functional data object for the curves to be registered.

fd0: A functional data object for the reference curve.

ximarks: A matrix with a row for each curve and a column for each landmark containing the landmark timings t_{if}.

x0marks: A vector containing landmark timings t_{0f} for the reference curve. By default these are the average timings.

WfdParobj: (required) A functional parameter or `fdPar` object that defines the strictly monotone warping function. The object can also define a `Lfd` object and a smoothing parameter for controlling the smoothness of the warping function.

monwrd: (optional) If T in S-PLUS or 1 in MATLAB, the warping function is estimated using a monotone smoothing method; otherwise, a regular smoothing method is used, which is not guaranteed to give strictly monotonic warping functions. However, using monotone smoothing will substantially increase the amount of computation required.

Returns: **regfd:** A functional data object for the registered curves.

warpfd: A functional data object defining the warping functions.

Wfd: A functional data object for function $W(t)$ defining the warping functions.

3.8.2 A Global Registration Function

This type of registration uses the whole curve, and does not require the estimation of landmarks. The technique is described in Ramsay and Li (1998).

```
register_fd(y0fd, yfd, Wfd0Parobj, periodic, crit,
            conv, iterlim, dbglev)
register.fd(y0fd, yfd, Wfd0Parobj, periodic=F, crit=2,
            conv=1e-2, iterlim=10, dbglev=1)
```

Purpose: Registers the curves in argument `yfd` to the target function in argument `yfd0`, and returns a functional data object defining a set functions that define the strictly monotonic warping functions that register the curves to the target. The warping functions are strictly monotonic, so these estimated functions define these warping functions in the same way as for monotone smoothing functions, defined in (7).

Arguments: **y0fd:** (required) A functional data object defining the target. It must be univariate and it must define a single functional observation.

yfd: (required) A functional data object defining the functions to be registered to `yfd0`. Multiple functions are permitted.

WfdParobj: (required) A functional parameter or `fdPar` object that defines the strictly monotone warping function. The object can also define a `Lfd` object and a smoothing parameter for controlling the smoothness of the warping function.

conv: (optional) A small positive constant that controls the level of convergence of the fitting criterion that is required in the numerical optimization. The default is 0.0001.

iterlim: (optional) The maximum number of iterations allowed. The default is 20.

dbglev: (optional) An integer controlling the amount of information displayed for each iteration. By default only the iteration number, fitting criterion value, and the gradient length are displayed.

periodic: (optional) A logical variable in S-PLUS or a variable taking only 0 or 1 in MATLAB. If T or 1, the functions are considered to be periodic, in which case a constant can be added to all argument values after they are warped. Otherwise the functions are assumed to be non-periodic, and the arguments are not shifted. The default is F or 0.

crit: (optional) An integer that is either 1 or 2 that indicates the nature of the continuous registration criterion that is used. If 1, the criterion is least squares, and if 2, the criterion is the minimum eigenvalue of a cross-product matrix. In general, criterion 2 is to be preferred. The default is 2.

Returns: A `list` in S-PLUS or a `struct` in MATLAB containing the following information. For MATLAB, each of the objects is returned separately and in the following order.

regfd: A functional data object for the registered curves.

Wfd: A functional data object for function $W(t)$ defining the warping functions.

Flist: List object containing (1) `Flist$f`, the final log likelihood , (2) `Flist$norm`, the final norm of gradient.

iternum: the number of iterations.

iterhist: , a matrix containing results for each iteration.

3.9 Elementary Statistical Functions

These are functions that compute functional versions of elementary statistical descriptions such as means, standard deviations, variances, covariances, and correlations. A function to subtract the mean function from the each curve is also provided.

3.9.1 `mean.fd` or `mean`

```
mean(fd)
mean.fd(fd)
```

Purpose: To evaluate the point-wise mean of a set of functions in a functional data object.

Arguments: **fd:** A functional data object.

Returns: A functional data object with a single replication that contains the mean of the one or several functions in the `fd` object.

3.9.2 `stddev.fd` or `stddev`

```
stddev(fd)
stddev.fd(fd)
```

Purpose: To evaluate the point-wise standard deviation of a set of functions in a functional data object.

Arguments: **fd:** A functional data object.

Returns: A functional data object with a single replication that contains the standard deviation of the one or several functions in the `fd` object.

3.9.3 `center.fd` or `center`

```
center(fd)
center.fd(fd)
```

Purpose: To subtract the pointwise mean from each of the functions in a functional data object; that is, to center them on the mean function.

Arguments: **fd:** A functional data object.

Returns: A functional data object with same dimensions as "fd" that contains the centered versions of the functions in the object fd.

3.9.4 `var.fd` or `var`

```
var(fdx, fdy)
var.fd(fdx, fdy = fdx)
```

Purpose: To compute the variance and covariance functions for functional data.

Arguments: **fdx:** (required) A functional data object.

fdy: (optional) An optional second functional data object.

Returns: A bivariate functional data object that contains the variance and, if there are more than one function in "fd", or if there is more than argument in the call to `var.fd`, the covariance functions. Results differ according to the number of arguments in the call.

- One argument: If "fdx" contains only replications of a single function, the coefficient matrix for the `bifd` object has two dimensions. If "fdx" contains function replications for more than one function, the `bifd` object has four dimensions. The third dimension has length 1, and the fourth dimension has length equal to the number of possible pairs of functions. Pairs are enumerated (1,1), (2,1), (2,2), (3,1) ... as is usual for the lower triangle of a symmetric matrix. For each pair the corresponding coefficients for the covariance function (or variance function if the two functions in the pair are the same), are given.

- Two arguments: If both arguments are functional data objects containing replications of a single function, then the covariance function is returned. If not, an error message is returned.

3.10 Principal Components Analysis

We now turn to principal components analysis, an exploratory analysis that tends to be an early part of many projects. The `pca.fd` function S-PLUS or `pca` function in MATLAB describes below computes the principal component functions, eigenvalues, and principal component scores described in Chapter 6, and also incorporates the regularization concept described in Chapter 7. The adjective phrase "principal component" being somewhat unwieldy, I opt for the term "harmonic" in the following description

3.10.1 `pca.fd` or `pca_fd`

```
pca_fd(fdobj, nharm, harmfdParobj, centerfns)
pca.fd(fdobj, nharm = 2, harmfdParobj, centerfns = T)
```

Purpose: To compute the harmonics, the eigenvalues, and harmonic scores for functional data. If more than one function is found, these are combined into a composite function.

Arguments: **fdobj:** (required) A functional data object.

nharm: (optional) The number of harmonics or principal components desired. The default is two.

harmfdParobj: (optional) A functional parameter of `fdPar` object defining the eigenfunctions harmonics that are to be estimated. The object may also define a `Lfd` object and a smoothing parameter for controlling the smoothness of the estimated eigenfunctions. The default is to use the same basis that defines the functions in argument `fdobj` and not to use smoothing.

centerfns: (optional) A logical variable. If `T`, the pointwise mean function is subtracted from each function before computing the harmonics. The default is `T`.

Returns: In S-PLUS a `list`, with the following members, and in MATLAB the members are returned directly.

harmfd: A functional data object containing the "nharm" harmonic, principal component, or eigenfunctions. If there is more than one variable in the "fd" argument, there is a harmonic corresponding to each function, and in this case the coefficient matrix has three dimensions.

values: The complete set of eigenvalues, equal in number to the number of basis functions, in the PCA.

scores: The principal component scores for each replication and harmonic.

varprop: The proportion of variance accounted for by each harmonic.

meanfd: A `fd` object for the mean function.

3.10.2 `plot.pca.fd` or `plot_pca`

```
plot_pca(pcastr, nx, pointplot, harm, expand, cycle)
plot.pca.fd(pcalist, nx = 128, pointplot = T, harm = 0,
            expand = 0, cycle = F, ...)
```

Purpose: Plots the harmonics of a functional principal component analysis.

Arguments: pcafd: (S-PLUS) or **pcastr:** (MATLAB) (required) In S-PLUS an object of class `pcafd` containing the results of a call to `pca.fd`. In MATLAB, a `struct` object containing the results of a call to `pca`.

nx: (optional) The number of points at which the functions are to be evaluated for plotting. For fairly smooth functions, 101 values are usually enough, but for functions with a lot of fine detail, this may need to be increased.

pointplot: (optional) If pointplot=T, then the harmonics are plotted as + and - otherwise lines are used.

harm: (optional) If harm = 0 (the default) then all the computed harmonics are plotted. Otherwise those in 'harm' are plotted.

expand: (optional) If expand =0 then effect of +/- 2 standard deviations of each principal component are given otherwise the factor expand is used.

cycle: (optional) If cycle=T and there are 2 variables then a cycle plot will be drawn. If the number of variables is anything else, cycle will be ignored.

...: (optional) (S-PLUS only) The additional arguments for controlling the plot available in the regular function `plot`.

Returns: none

3.10.3 `varmx.pca.fd` or `varmx_pca`

```
varmx_pca(pcastr, nharm, nx)
varmx.pca.fd(pcafdlist, nharm = scoresd[2], nx=50)
```

Purpose: Apply varimax rotation to the first `nharm` components of a 'pca.fd' object.

Arguments: **pcafdlist:** (S-PLUS) or **pcastr:** (MATLAB) (required) In S-PLUS an object of class `pcafd`, and in MATLAB a `struct`, containing the results of a call to `pca`.

nharm: (optional) The number of harmonics to be rotated. The default is the number available in `pcafdlist` for `pcastr`.

nx: (optional) The number of points at which the functions are to be evaluated for plotting. For fairly smooth functions, 101 values are usually enough, but for functions with a lot of fine detail, this may need to be increased.

Returns: In both languages the return is the same structure and class as for the principal components analysis function itself, but with the principal components and scores rotated.

3.11 The Linear Model Functions

The linear model function described below fits the three types of linear models described in Chapters 12 to 16. At this point the function can only handle a single functional independent variable.

3.11.1 `fRegress` or `fRegress`

```
fRegress(yfdPar, xfdcell, betacell)
fRegress(yfdPar, xfdlist, betalist)
```

Purpose: To fit a concurrent or point-wise functional linear model. A functional dependent variable is fit by the concurrent or point-wise functional linear model with one or more functional independent variables. Any of the variables, whether dependent or independent, may be univariate or scalar, in which case the variables are converted to functional data objects using the constant basis. A functional data object with a constant basis is in every way equivalent to a univariate or scalar variable.

Arguments: **yfdPar:** (required) A `fdPar` object, a `fd` object or a numerical vector. This is the dependent functional variable. If `yfdPar` is numeric, corresponding a univariate scale variable, then it is converted to a `fd` object using a constant basis.

xfdlist: (required) A list in S-PLUS or a cell array in MATLAB, where it is named `xfdcell`. Each member of the list contains a `fdPar` object, a `fd` object or a numerical vector corresponding to an independent variable. As for `yfdPar`, scalar or numeric vectors are converted to `fd` objects with a constant basis.

betalist: (required) A list in S-PLUS or a cell array in MATLAB, where it named `betacell`. Each member contains a functional parameter or `fdPar` object defining a regression coefficient function corresponding to an independent variable in `xfdlist`. Each member may also define a `Lfd` object and a smoothing parameter value to control the roughness of the estimated regression coefficient function. Moreover, if a regression coefficient function is to be kept at a fixed value rather than estimated, the `estimate` member of the `fdPar` object may be set to `F` or `0`.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the following members:

yfdPar: the first argument of the call to `fRegress`.

xfdlist: the second argument of the call to `fRegress`.

betalist: the second argument of the call to `fRegress`.

betaestlist: A list in S-PLUS or a cell array in MATLAB containing the estimated regression coefficient functions. Each member is a `fdPar` object.

yhatfdobj: A functional data object for the estimated dependent variable.

Cmatinv: Te inverse of the coefficient matrix, needed for function `fRegress.stderr` that computes standard errors.

3.11.2 `fRegress.stderr` or `fRegress_stderr`

```
fRegress_stderr(fRegressCell, y2cMap, SigmaE)
fRegress.stderr(fRegressList, y2cMap, SigmaE)
```

Purpose: To compute estimates of the pointwise standard errors of regression coefficient functions estimated in a call to function `fRegress`.

Arguments: **fRegressList:** (required) A `list` object in S or a cell object in Matlab that has been computed by a previous call to function `fRegress`.

Y2cMap: (required) The matrix mapping from the vector of observed values to the coefficients for the dependent variable. This matrix is output by function `smooth.basis`.

SigmaE: (required) Estimate of the covariances among the residuals, required for the estimation of confidence intervals. This can only be estimated from a preliminary analysis.

Returns: A `list` in S-PLUS or a `struct` in MATLAB with the following members:

betastderrlist: A list in S-PLUS or a cell array in MATLAB containing the estimated standard error functions for the regression coefficient functions. These are only estimated if the `y2cmap` argument is supplied.

bvar: the symmetric matrix of sampling variances and covariances for the matrix of regression coefficients for the regression functions.

These are stored column-wise in defining BVARIANCE. This is only computed if the `y2cmap` argument is supplied.

C2bMap: The matrix mapping from response variable coefficients to coefficients for the regression functions.

4 Installation Notes

If you are reading this, you probably have already visited one of our web sites at

```
http://www.psych.mcgill.ca/faculty/ramsay.html
http://www.functionaldata.org
```

4.1 MATLAB Installation

To install the Matlab functions and sample analyses in a Windows 98/NT/2000/XP system, using Matlab Version 5 through 7, follow these instructions, which will install the Matlab functions, the sample analyses and the data that are analyzed.

4.1.1 Installing the Matlab functions:

1. Create a directory to contain the Matlab functions. For example, on my system the directory is

   ```
   c:\Matlab\fdaM
   ```

2. Put the file "Matlabfunctions.zip" into this directory.

3. Extract the function files, each with extension `.m`, from this`.zip` file using a utility such as WinZip (available on the Web). Each of these function files will contain a function with the same name as the file, and possibly some supporting functions only used by this function. Documentation on the use of the functions is found in the leading lines of the file.

4. Within this directory, create a subdirectory with the name "@fd". That is, on my system, this would have the path

   ```
   c:\Matlab\fdaM\@fd
   ```

This subdirectory will contain functions that are process objects of the "fd" class. Move the file "@fd.zip" into this directory, and extract the function files as you did in step 3.

5. Repeat step 4 with subdirectory names

   - @basis containing .zip file basis.zip
   - @bifd containing .zip file bifd.zip
   - @fdPar containing .zip file fdPar.zip
   - @Lfd containing .zip file Lfd.zip

   Thus, on my system, these five directories have paths

   ```
   c:\Matlab\fdaM\@basis
   c:\Matlab\fdaM\@bifd
   c:\Matlab\fdaM\@fd
   c:\Matlab\fdaM\@fdPar
   c:\Matlab\fdaM\@Lfd
   ```

   and each subdirectory should now contain the unzipped functions appropriate to that directory.

4.1.2 Installing the examples:

There are currently nine sample analyses bundled with the data that are analyzed:

- the gait data, in file `gait.zip`
- the nondurable goods index, in file `goodsindex.zip`
- the growth data, in file `growth.zip`
- the handwriting data, in file `handwrit.zip`
- the lip movement data, in file `lip.zip`
- the melanoma data, in file `melanoma.zip`

- the pinch force data, in file `pinch.zip`
- the refinery data, in file `refinery.zip`
- the monthly and daily weather data, in file `weather.zip`

You might consider setting up a separate subdirectory for each of these analyses, perhaps within a directory "examples" in the directory containing the functions set up above.

For each of these analyses and data, move the `.zip` file with appropriate name into the appropriate subdirectory. Then extract the files in this file using WinZip or some other utility.

To run a sample analysis, start Matlab. At the top of each sample analysis file, with the extension `.m`, you will find two `addpath` commands that attach, respectively, the `functions` directory, and the directory containing the sample data. The paths in these commands are what I use in my system, and you may have to change them to what is appropriate for your system. For example, at the top of the `monthly.m` file, you will find the two commands

```
addpath('c:\\Matlab\\fdaM')
addpath('c:\\Matlab\\fdaM\\examples\\weather')
```

that add the needed paths on my system.

4.2 S-PLUS Installation

The S-PLUS software described in these notes is available at either of these sites.

You must obtain the files named "FDAfuns.s", contained the actual S-PLUS functions.

The first time that you invoke S-PLUS to use these functions, you must use the command

```
source('FDAfuns.s')
```

to set up the S-PLUS FDA functions.

Note that you may also want to use the Pspline module for estimating derivatives by spline smoothing. This can be obtained from the web site

```
http://www.stat.cmu.edu
```

or from Jim Ramsay's web site, or by ftp from

`ego.psych.mcgill.ca/pub/ramsay`

The same instructions apply for the Lspline module.

References

Clarkson, D. B., Fraley, C., Gu, C. C. and Ramsay, J. O. (2005) *S+Functional Data Analysis User's Guide.* New York: Springer.

Hanselman, D. and Littlefield, B. (2005) *Mastering Matlab 7.* Upper Saddle River, NJ: Prentice-Hall.

Ramsay, J. O. and Silverman, B. W. (2002) *Applied Functional Data Analysis.* New York: Springer.

Ramsay, J. O. and Silverman, B. W. (2005) *Functional Data Analysis*, Second Edition. New York: Springer.

Venables, W. N. (2002) *Modern Applied Statistics with S*, Fourth Edition. New York: Springer.

Venables, W. N. and Ripley, B. D. (2000) *S Programming.* New York: Springer.