

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

J. O. Ramsay, McGill University

February 14, 2005

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 5 |
| 1.1 | The Goals of these Notes | 5 |
| 1.2 | The Two Languages | 5 |
| 1.3 | Object-oriented programming | 6 |
| 1.4 | An Overview of the Steps in an FDA | 8 |
| 1.5 | An Overview of these Notes | 9 |
| 1.6 | An Important Disclaimer | 11 |
| 2 | More on FDA Objects | 12 |
| 2.1 | What is an Object in MATLAB or S-PLUS? | 12 |
| 2.2 | The three essential classes for FDA | 13 |
| 2.2.1 | The <code>basis</code> class | 13 |
| 2.2.2 | The functional data class: <code>fd</code> | 17 |
| 2.2.3 | The bi-variate functional data class: <code>bifd</code> | 20 |
| 2.3 | A warning about variable names | 21 |
| 3 | The more important FDA functions | 22 |
| 3.1 | <code>basis</code> and <code>fd</code> object creation functions | 22 |
| 3.1.1 | <code>create.bspline.basis</code> or <code>create_bspline_basis</code> . . | 23 |
| 3.1.2 | <code>create.fourier.basis</code> or <code>create_fourier_basis</code> . . | 24 |
| 3.1.3 | <code>create.exponential.basis</code> or <code>create_exponential_basis</code> | 25 |
| 3.1.4 | <code>create.power.basis</code> or <code>create_power_basis</code> | 25 |
| 3.1.5 | <code>create.monomial.basis</code> or <code>create_monomial_basis</code> . | 26 |
| 3.1.6 | <code>create.constant.basis</code> or <code>create_constant_basis</code> . | 26 |
| 3.1.7 | <code>create.polygonal.basis</code> or <code>create_polygonal_basis</code> | 27 |
| 3.1.8 | <code>create.basis.fd</code> or <code>create_basis_fd</code> | 27 |
| 3.1.9 | Additional optional arguments for the basis creation functions | 28 |
| 3.1.10 | <code>create.fd</code> or <code>fd</code> | 29 |
| 3.1.11 | <code>create.bifd</code> or <code>bifd</code> | 30 |
| 3.2 | The <code>data2fd</code> function | 30 |
| 3.2.1 | Format of the Data | 31 |
| 3.2.2 | <code>data2fd</code> | 32 |
| 3.3 | Two more classes for smoothing | 33 |
| 3.3.1 | The <code>Lfd</code> class | 34 |

| | | |
|----------|---|-----------|
| 3.3.2 | The <code>fdPar</code> class | 35 |
| 3.4 | Smoothing data using a roughness penalty | 36 |
| 3.4.1 | <code>smooth.basis</code> or <code>smooth_basis</code> | 39 |
| 3.4.2 | <code>smooth.fd</code> for <code>smooth_fd</code> | 41 |
| 3.5 | Functions for Smoothing with Constrained Functions | 41 |
| 3.5.1 | Positive Smoothing with <code>smooth.pos</code> and <code>smooth_pos</code> | 42 |
| 3.5.2 | Monotone Smoothing with <code>smooth.monotone</code> and <code>smooth_monotone</code> | 43 |
| 3.6 | Summary, Evaluation and Plotting Functions | 45 |
| 3.6.1 | <code>plot.fd</code> , <code>plot</code> | 46 |
| 3.6.2 | <code>cycleplot.fd</code> or <code>cycleplot</code> | 47 |
| 3.6.3 | <code>lines.fd</code> or <code>line</code> | 47 |
| 3.6.4 | <code>print.fd</code> or <code>display</code> | 48 |
| 3.6.5 | <code>summary.fd</code> (S-PLUS only) | 48 |
| 3.6.6 | <code>eval.fd</code> or <code>eval_fd</code> | 49 |
| 3.6.7 | <code>eval.bifd</code> or <code>eval_bifd</code> | 49 |
| 3.7 | Data Manipulation Functions | 50 |
| 3.7.1 | Subsets of Functional Data Observations | 50 |
| 3.7.2 | Arithmetical Operations | 51 |
| 3.8 | Registration Functions | 51 |
| 3.8.1 | Landmark Registration | 51 |
| 3.8.2 | A Global Registration Function | 53 |
| 3.9 | Elementary Statistical Functions | 55 |
| 3.9.1 | <code>mean.fd</code> or <code>mean</code> | 55 |
| 3.9.2 | <code>std.fd</code> or <code>std</code> | 55 |
| 3.9.3 | <code>center.fd</code> or <code>center</code> | 56 |
| 3.9.4 | <code>var.fd</code> or <code>var</code> | 56 |
| 3.10 | Principal Components Analysis | 57 |
| 3.10.1 | <code>pca.fd</code> or <code>pca_fd</code> | 57 |
| 3.10.2 | <code>plot.pca.fd</code> or <code>plot_pca</code> | 58 |
| 3.10.3 | <code>varmx.pca.fd</code> or <code>varmx_pca</code> | 59 |
| 3.11 | The Linear Model Functions | 59 |
| 3.11.1 | <code>fRegress</code> or <code>fRegress</code> | 60 |
| 4 | Installation Notes | 62 |
| 4.1 | MATLAB Installation | 62 |
| 4.1.1 | Installing the Matlab functions: | 62 |
| 4.1.2 | Installing the examples: | 63 |

| | | |
|----------|--|-----------|
| 4.2 | S-PLUS Installation | 64 |
| 5 | Some Sample Functional Data Analyses | 66 |
| 5.1 | The Monthly Weather Data | 66 |
| 5.2 | The Lip Data: A Landmark Registration Example | 79 |
| 6 | Some Additional Notes on Monotone Smoothing | 82 |
| 6.1 | Introduction | 82 |
| 6.2 | A Differential Equation for Monotone Functions | 87 |
| 6.3 | Monotone Data Smoothing | 89 |
| 6.3.1 | Basis Function Expansions for w | 90 |
| 6.3.2 | Testing for Non-monotonicity | 91 |
| 6.3.3 | The Growth Data | 92 |
| 7 | Some Additional Notes on Curve Registration | 93 |
| 7.1 | Introduction | 93 |
| 7.2 | Formulation of the Registration Problem | 100 |
| 7.3 | Two Registration Techniques | 103 |
| 7.3.1 | Marker or Landmark Registration | 103 |
| 7.3.2 | Continuous Registration | 107 |
| 7.3.3 | A Test of Continuous Registration | 109 |
| 7.4 | Defining Smooth Warping Functions h | 110 |
| 7.5 | Estimation of Warping Function h_i | 112 |
| 7.5.1 | Roughness Penalties on $w(t)$ | 113 |
| 7.5.2 | The Procrustes Fitting Criterion | 114 |

1 Introduction

1.1 The Goals of these Notes

These notes are designed to accompany the books *Functional Data Analysis* (1997, 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 we 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. We 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 we have tried to keep the modules small and readable, and we 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 we will not distinguish between R and S-PLUS as languages, and will mostly refer to the latter. We 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 we 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 we 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, we 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 pre-specifies 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.

We recommend that users not already familiar with the object-oriented features in these languages first read the few chapters on this topic in references such as Chambers and Hastie (1996) (S-PLUS) or Hanselman and Littlefield (2001) (MATLAB).

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. We assume that there is a one-dimensional argument, that we will denote by t . As a rule functions of t are observed only at discrete sampling values $t_j, j = 1, \dots, n$, 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 $i = 1, \dots, N$. 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, $t_{ij}, j = 1, \dots, n_i$, 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. We provide both a landmark registration algorithm and a continuous registration algorithm to do this.

Figure 1: A flow chart for a typical functional data analysis.

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 (**pda**)
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.

These steps are diagrammed in the flowchart in Figure 1.

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 5 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 6 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 we define the five objects that we shall use in our FDA's. First, we 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 fields: 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 we have used the prefix **create** to indicate such a function. For example, we can create a basis of the Fourier type by us-

ing 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 we only describe the most essential slots or pieces of information. Later we 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, $\phi_k, k = 1, \dots, K$. In this approach, a

functional observation x_i is expressed as

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

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 indicated in Chapter 3, there are many bases possible, and many considerations to take into account. We 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 : $1, t, t^2, t^3, \dots$,
- 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 we 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.

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 fields. 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 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:

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

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, ..., 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.

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 `create_basis_fd` in MATLAB or `create.basis.fd` 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.

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 we use either convention. So from now on, we 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, of 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 expan-

sion 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 `create.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.

2.3 A warning about variable names

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.

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 `create.basis.fd` 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.

We first describe four of the basis-specific functions, and then the generic function.

In each description, we 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, we 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 we 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 `create.basis.fd` or `create_basis_fd`

```
create_basis_fd(type, rangeval, nbasis, params)
create.basis.fd(type, rangeval, nbasis, params)
```

Purpose: The generic function for creating a **basis** object.

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 contains a subset of the indices $1, \dots, \text{nbasis}$ 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 $(1h, 4h, 2h, 4h, \dots, 4h, 2h, 4h, 1h)$ 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 there is a real difference in the syntax for the two languages: MATLAB simply uses the class name to create an object of that class, whereas S-PLUS requires the **create** prefix.

3.1.10 `create.fd` or `fd`

```
fd(coef, basisobj, fdnames)
create.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 `create.bifd` or `bifd`

```
bifd(coef, sbasisobj, tbasisobj, bifdnames)
create.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 `create.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.

This step, 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, we will wait until we 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 is 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 `create.basis.fd`, 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 cubic smoothing spline is defined by the penalty

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

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

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

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

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

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

```
create.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

```
create.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 $y_{ij}, j = 1, \dots, n$ corresponding to a specific function x_i by minimizing

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

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

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

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

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

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 $\lambda \rightarrow 0$, 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

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

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:

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

Integer m is the *order* of the linear differential operator L , and each of the m functions $\beta_j(t)$, $j = 0, \dots, m-1$ 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

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

The reason for considering this wider family of penalties that by the appropriate choice of L , we can force the smooth as $\lambda \rightarrow \infty$ 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 $\beta_0, \beta_1, \dots, \beta_{m-1}$. 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 **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 we 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 \mathbf{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 **R**.

Y2CMAP: The matrix S_λ 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 Smoothing with Constrained Functions

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 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

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

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 β_0 , is permitted to be a linear combination of a set of covariate values in vector \mathbf{z} , in which case β_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.

Wfdbj: 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 **Wfdbj** 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

An 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, monwrdr)
```

```
landmarkreg(fd, ximarks, x0marks=xmeanmarks,  
            WfdParobj, monwrdr=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.

monwrld: (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).

```
registerfd(y0fd, yfd, Wfd0Parobj, periodic, crit,  
          conv, iterlim, dbglev)  
registerfd(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 **y0fd**, 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 `std.fd` or `std`

```
std(fd)
std.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 in 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, we 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, y2cmap, sigmae)
fRegress(yfdPar, xfdcell, betacell, y2cmap, sigmae)
```

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**.

y2cmap: (optional) The matrix mapping from the vector of observed values to the coefficients for the dependent variable. This matrix is output by function **smooth.basis**. If supplied, confidence limits are computed, otherwise not.

sigmae: (optional) Estimate of the covariances among the residuals, required for the estimation of confidence intervals. This can only be estimated from a preliminary analysis. If **sigmae** is not provided, it is estimated.

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

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

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

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.

bvariance: 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.statistics.bristol.ac.uk/~bernard>
<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 "FDAfuncs.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('FDAfuncs.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.

5 Some Sample Functional Data Analyses

In this section we give some examples of functional data analyses, with the goal being to show how to use the functional data objects and the functional data functions described in the previous sections. In each of the analyses shown, the actual MATLAB and S-PLUS commands are given in typewriter font. In each case, it will be assumed that the data arrays have already been input. In the code distributed with these notes, the raw data files and the commands to input the data in them are included, however.

We include both the MATLAB and S-PLUS code, and in that order.

5.1 The Monthly Weather Data

These data involve two samples of functional data, one for temperature and one for precipitation. They are set up as two functional data objects rather than one object containing two functions because the measures involved, degrees Celsius and millimetres, are different. Each set of data is input into a 35 by 12 matrix, there being 35 different weather stations and 12 months. The measures are taken as positioned at the middle of each month. these data are, of course, periodic, and therefore will be expanded in terms of the first 12 fourier series functions. The names of the weather stations, and symbols for the months are input as the `dimnames` attributes of these matrices.

We first read in the temperature and precipitation data from a file. Unfortunately, the data are set up in the file the wrong way, and we need to transpose to have rows corresponding to the monthly sampling points, and the columns to replications. At the same time, we specify the sampling points themselves as the mid-points for the months. Finally, let's assume that the weather station names have already been defined in character array `meteonames`.

MATLAB:

```
fid = fopen('temp.dat','rt');
tempvec = fscanf(fid,'%f');
tempmat = reshape(tempvec, [12, 35]);
fid = fopen('prec.dat','rt');
precvec = fscanf(fid,'%f');
precmat = reshape(precvec, [12, 35]);
```

S-PLUS:

```
tempmat <- t(matrix(scan('temp.dat'), 35, 12, byrow=T))
precmat <- t(matrix(scan('prec.dat'), 35, 12, byrow=T))
weathertime <- seq(0.5, 11.5, 1)
dimnames(tempmat) <- list(months, meteonames)
dimnames(precmat) <- list(months, meteonames)
```

Now we have to set up the Fourier basis object to be combined with the discrete temperature data to make the temperature fd object. The following statement specifies the fourier series basis, sets up the range spanning the interval $[0,12]$, and asks for 12 basis functions. By default the period is set to 12, the range of possible argument values defined by the first argument. Moreover, even though we ask for 12 basis functions, the fourier basis always involves an odd number of basis functions; that is, the constant function plus a number of sine/cosine pairs. We will actually work with 13 basis functions in this problem. This is more than the number of sampling points, but `data2fd` will take care of this. The result of the following statement is a basis object called `monthbasisobj`.

```
monthbasis = create_fourier_basis([0,12], 12);
```

```
monthbasis <- create.fourier.basis(c(0,12), 12)
```

Now we create the temperature and precipitation functional data objects, called `tempfd`, and `precfd`, respectively. The `argnames` argument sets up the labels for the `tt fdnames` member of the object.

```
tempfd = data2fd(tempmat, monthtime, monthbasis);
precfd = data2fd(precmat, monthtime, monthbasis);

tempfd <- data2fd(tempmat, monthtime, monthbasisobj
                  argnames=c('Months', 'Station', 'Deg C'))
precfd <- data2fd(precmat, monthtime, monthbasisobj
                  argnames=c('Months', 'Station', 'mm'))
```

Lets have a look at what we have. First we plot the temperature curves, and then print out a summary. Figure 2 shows the temperature functions resulting from the plot command. Note that the plotting function automatically labels axes using the information in the `fdnames` member of `tempfd`.

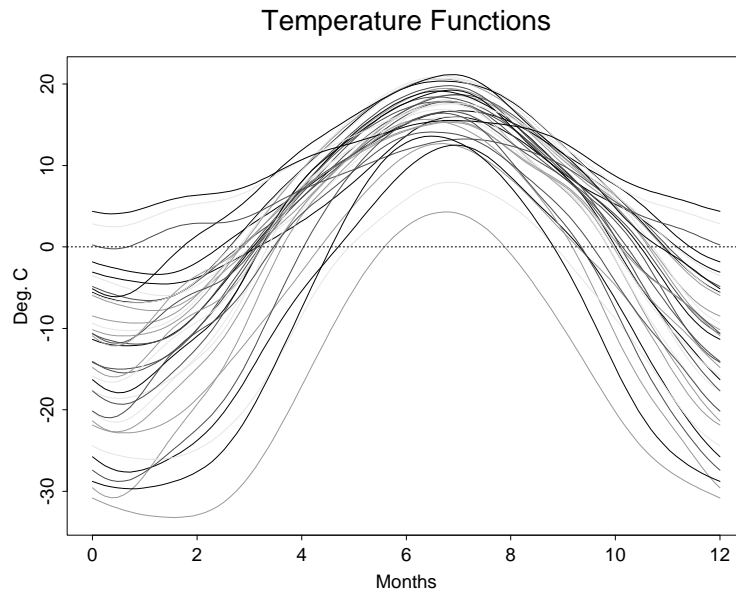


Figure 2: The 35 temperature functions resulting from expanding monthly mean temperatures in terms of the first 12 terms of the fourier series.

```
plot(tempfd);
title('Temperature Functions');
display(tempfd)

plot(tempfd, main="Temperature Functions")
summary(tempfd)
```

The output from the `summary` command in S-PLUS is

```
Dimensions of coefficient matrix:
[1] 12 35
```

```
Basis:
```

```
  Type: fourier
  Range: 0 to 12
  Number of basis functions: 12
  Period: 12
```

Here's a challenge: How can we plot the forcing functions defined by

the linear differential operator $L = (6/\pi)D + D^3$, shown in Chapter 1? We would need to call the `eval.fd` function twice, compute the weighted sum, and plot; four commands in short. But here's another approach. Let's define the linear differential operator itself, since we will need it later. We will call it `harmaccelLfd`, for "harmonic acceleration". It is defined by three weight functions, each weighting one of the derivatives of orders 0, 1, and 2. Since each weight function is constant, we only need the constant basis to set this up. Matrix `Lcoef` contains the coefficients for this simple expansion. It is 1 by 3 since there is a single basis function, 1, and three replications.

```
Lbasis      = create_constant_basis([0,12]);
Lcoef       = [0,(pi/6)^2,0];
harmaccelLfd = fd(Lcoef, Lbasis);

Lbasis      <- create.constant.basis(c(0,12))
Lcoef       <- matrix(c(0, (pi/6)^2, 0),1,3)
harmaccelLfd <- create.fd(Lcoef, Lbasis)
```

Now plotting the forcing functions is simple. MATLAB requires a separate command to add a title, while S-PLUS can include the title as an argument for `plot`.

```
plot(tempfd, harmaccelLfd);
title('Temperature Forcing Functions')

plot(tempfd, harmaccelLfd,
      main="Temperature Forcing Functions")
```

The next phase is to look at some basic statistical descriptions of the data. The following commands set up the two mean functions as functional data objects and plot them. Results are in Figure 3.

```
tempmeanfd = mean(tempfd);
precmeanfd = mean(precfd);
subplot(1,2,1), plot(tempmeanfd), axis('square')
subplot(1,2,1), plot(precmeanfd), axis('square')

tempmeanfd <- mean.fd(tempfd)
```

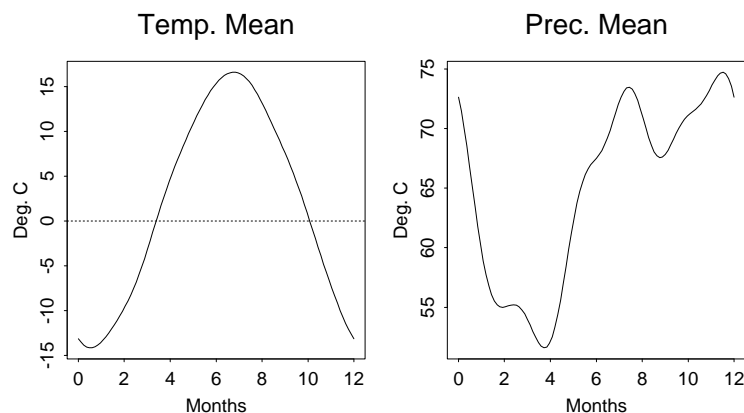


Figure 3: The mean functions for temperature and precipitation for the monthly weather data.

```
precmeanfd <- mean.fd(precfd)
par(mfrow=c(1,2),pty="s")
plot(tempmeanfd)
plot(precmeanfd)
```

Now we compute the variance functions for the two measures, and the covariance between them. Each of these sets up a `bifd` object. We didn't design a plotting function specifically for objects of this nature (there being a lot of different ways to do this), so instead we make tables of each of them using the `eval.bifd` function. Figure 4 plots these tables as contour plots.

MATLAB: (plot the temperature covariance only)

```
tempvarbifd = var(tempfd);
weeks = linspace(0,12,53);
tempvarmat = eval(tempvarbifd, weeks, weeks);
subplot(1,1,1);
surf(tempvarmat);
xlabel('Weeks'), ylabel('Weeks'), zlabel('Covariance')
title('Temperature Variance-Covariance Function')
```

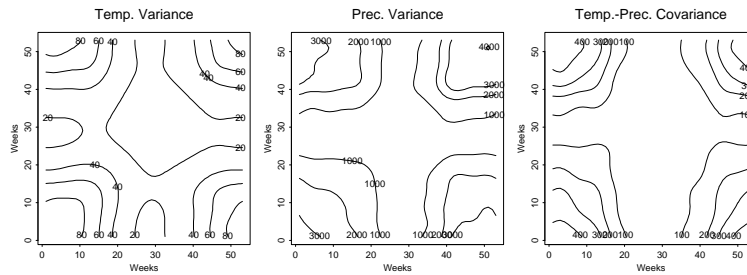


Figure 4: The variance surfaces for temperature and precipitation, and the covariance surface for their relationship.

S-PLUS:

```
tempvarbifd      <- var.fd(tempfd)
precvarbifd      <- var.fd(precfd)
temppreccovbifd <- var.fd(tempfd, precfd)
weeks <- seq(0,12,length=53)
tempvarmat       <- eval.bifd(weeks,weeks,tempvarbifd)
precvarmat       <- eval.bifd(weeks,weeks,precvarbifd)
temppreccovmat  <- eval.bifd(weeks,weeks,temppreccovbifd)
par(mfrow=c(1,3),pty="s")
contour(tempvarmat, xlab="Weeks", ylab="Weeks",
        main="Temp. Variance")
contour(precvarmat, xlab="Weeks", ylab="Weeks",
        main="Prec. Variance")
contour(temppreccovmat, xlab="Weeks", ylab="Weeks",
        main="Temp.-Prec. Covariance")
```

Can you figure out how to plot correlation surfaces using standard S-PLUS functions?

Now we get into some heavier analyses. Our first task is to carry out a principal components analysis of the temperature data. In order to keep the displays simple, we go for the default two harmonics. But a preliminary peek at the results convinces us that a little bit of regularization or smoothing of the eigenfunctions or harmonics would be helpful. We elect to smooth by penalizing the harmonic acceleration, rather than the curvature, since we don't object to simple sinusoidal variation in the harmonics. We also apply a VARIMAX rotation to the harmonics.

```
temppcastr = pca(tempfd, 4, 1e-3, harmaccelLfd);
temppcastr = varmx_pca(temppcastr);

temppcalist <- pca.fd (tempfd, Lfd=harmaccelLfd,
                      nharm=4, lambda=1e-3)
temppcalist <- varmx.pca.fd(temppcalist)
```

The eigenvalue plot in Figure 5 is produced by these following commands. We have found it informative to plot the log eigenvalues, and to fit a line by least squares to the log eigenvalues that we don't intend to look at. This seems to confirm that there are four main components of variation in the data. The first reflects variation in annual temperature, and the second the variation from the average of the difference between the mid-winter and mid-summer temperatures.

```
tempharmeigval = temppcastr.eigvals;
x = ones(8,2); x(:,2) = reshape((5:12),[8,1]);
y = log10(tempharmeigval(5:12));
c = x\y;
plot(1:12, log10(tempharmeigval(1:12)),'-o', ...
     1:12, c(1)+c(2).*(1:12),           ':')
xlabel('Eigenvalue Number'), ylabel('Log10 Eigenvalue')

tempharmeigval <- temppcalist$values
plot(1:12, log10(tempharmeigval[1:12]), type='b',
     xlab='Eigenvalue Number', ylab='Log10 Eigenvalue')
abline(lsfit(5:12, log10(tempharmeigval[5:12])), lty=2)
```


The following plotting commands produce the harmonic scores plotted in Figure 6. We aren't surprised to see the marine stations up in the upper-right "high annual/low variation" quadrant, and Resolute at the lower extreme of the first harmonic.

```
tempharmscr = temppcastr.harmscr;
plot(tempharmscr(:,1),tempharmscr(:,2), 'o')
xlabel('Scores on Harmonic 1')
ylabel('Scores on Harmonic 2')
text(tempharmscr(:,1),tempharmscr(:,2),meteonames)

tempharmscr <- tempharmlist[[3]]
par(mfrow=c(1,1),pty="s")
plot(tempharmscr[,1],tempharmscr[,2],type="n",
      xlab="Harm. 1",ylab="Harm. 2")
text(tempharmscr[,1], tempharmscr[,2],
      dimnames(tempharmscr)[[1]])
```

Now how about a little linear modeling. We start with the simple model in which the temperature functions are predicted by their corresponding weather zones. First we set up the indices for the weather zones:

```
atlindex = [1,2,3,4,5,6,7,8,9,10,11,13,14,16];
pacindex = [25,26,27,28,29];
conindex = [12,15,17,18,19,20,21,22,23,24,30,31,35];
artindex = [32,33,34];

atlindex <- c(1,2,3,4,5,6,7,8,9,10,11,13,14,16)
pacindex <- c(25,26,27,28,29)
conindex <- c(12,15,17,18,19,20,21,22,23,24,30,31,35)
artindex <- c(32,33,34)
```

Next we set up the design matrix. It has five columns, the first corresponding to the mean function, and the remainder corresponding to zone effects.

```
zmat = zeros(35,5);
zmat(:,1) = 1;
```

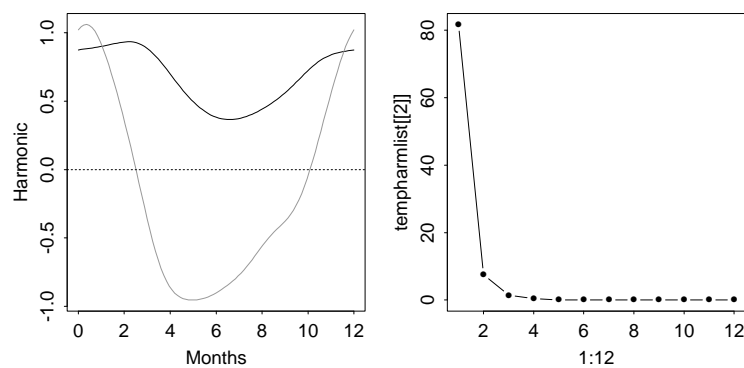


Figure 5: The left panel displays the two principal component functions or harmonics for the monthly temperature data. The right panel displays the twelve eigenvalues in the principal components analysis.

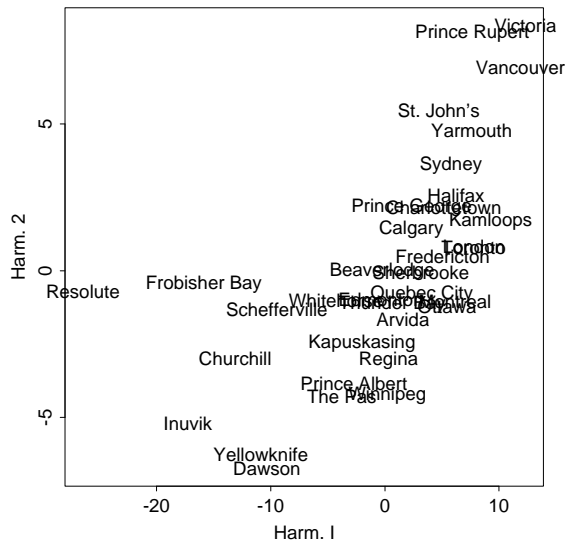


Figure 6: The names of the weather stations are plotted in the positions defined by their scores on the first two harmonics.

```

zmat(atlindex,2) = 1;
zmat(pacindex,3) = 1;
zmat(conindex,4) = 1;
zmat(artindex,5) = 1;

zmat <- matrix(0,35,5)
zmat[,1] <- 1
zmat[atlindex,2] <- 1
zmat[pacindex,3] <- 1
zmat[conindex,4] <- 1
zmat[artindex,5] <- 1
dimnames(zmat) <- list(meteonames,
  c("Mean", "Atlantic", "Pacific", "Continental", "Arctic"))

```

Now we fit the linear model. Note that the design matrix is actually of rank 4, so we have to say so in the function call. The output of the function is a functional data object containing five functions: the mean function, the atlantic zone effect, the pacific zone effect, the continental zone effect, and

the artic zone effect.

```
templinstr = fRegress(zmat, tempfd);  
  
templinlist <- fRegress(zmat, tempfd, zmatrnk=4)
```

We can now use the functional data subscripting feature to plot the mean function first, and then the zone effects. You can see the zone effects in Figure 7.

```
tempregfd = fRegressstr.reg; plot(tempregfd), title('Regression  
Functions') for i=1:4  
  subplot(2,2,j), line(tempregfd[i+1]);  
end  
  
tempregfd <- templinlist[[2]]  
par(mfrow=c(1,1),pty="m")  
plot(tempregfd[1],xlab="Month",ylab="Deg. C", main="Mean Fn.")  
par(mfrow=c(2,2),pty="m")  
for (i in 1:4) plot(tempregfd[i+1],xlab="Month",ylab="Deg. C",  
  main=dimnames(zmat)[[2]][i+1])
```

Next we try a scale dependent variable, log annual precipitation, and use as a functional independent variable temperature.

```
logannprec = log10(sum(precmat)');  
  
logannprec <- as.matrix(log10(apply(precmat,2,sum)))
```

Now we carry out the linear model. As discussed in the book, some smoothing is required in order to not waste too many degrees of freedom, and in order to aid interpretation. We again penalize harmonic acceleration. The smoothing parameter used, after some experimentation, (but cross-validation could also have been used) was 1.

```
fRegressstr = fRegress(tempfd, logannprec, ones(35,1), ...  
  harmaccelLfd, 0, 1, 0);  
  
fRegresslist <- fRegress(tempfd, logannprec,  
  xLfd=harmaccelLfd, xlambda = 1)
```

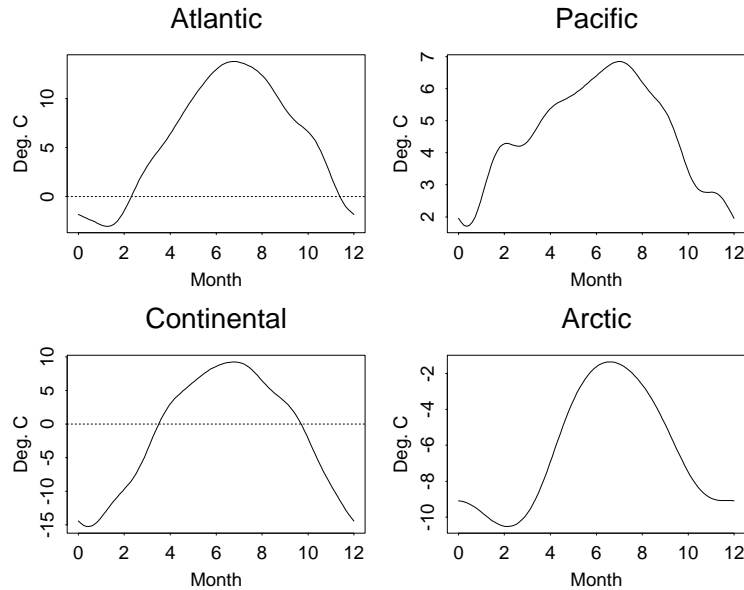


Figure 7: The four zone effects resulting from predicting temperature from climate zone.

The intercept emerged as 3.06, corresponding to just over one meter of annual rainfall. The regression function is shown in Figure 8, where we see that annual temperature seems to be determined by how low the temperature is in February and August, and how high the temperature is in October. In fact, this neatly fits the temperature profile of Prince Rupert, the station with the highest precipitation by far.

```
regressionfd = fRegressstr.reg; plot(regressionfd);
xlabel('\fontsize{16} Month') ylabel('\fontsize{16} Regression
Function Value')

regressionfd <- fRegresslist[[2]] plot(regressionfd, type="l",
cex=1,
      xlab="Month", ylab="Regression Function Value")
```

Our final linear model involves predicting the complete precipitation function from the temperature function. This time the regression function will be bivariate. Some smoothing is required for both arguments in this case.

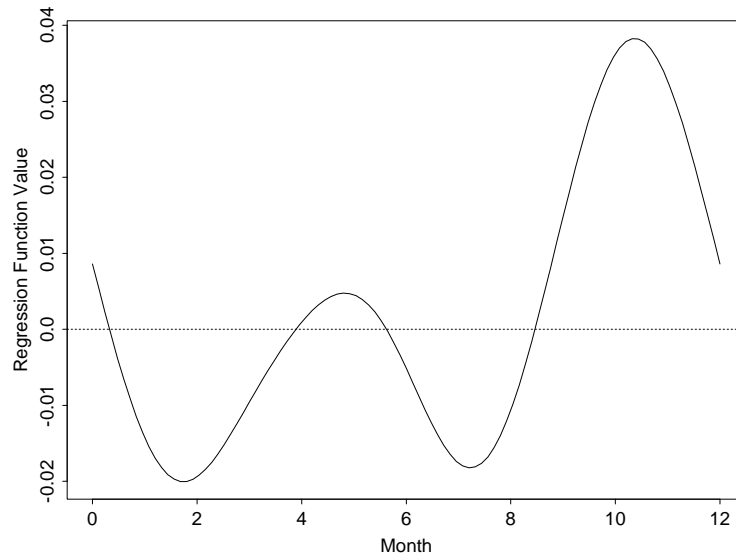


Figure 8: The regression function for predicting log annual precipitation from the temperature function.

the intercept in this case is a regular function. The plots of the intercept function and the bivariate regression function are given in Figure 9.

MATLAB:

```
fRegressstr = fRegress(tempfd, precfd, ones(35,1), ...
    harmacellfd, harmacellfd, 1, 10);
alphafd = fRegressstr.alpha; regbifd = fRegressstr.reg;
subplot(1,2,1) plot(alphafd); subplot(1,2,2) regbifdmat =
eval_bifd(regbifd, weeks, weeks); surf(regbifdmat) xlabel('Week'),
ylabel('Week'), zlabel('Regression Function')
```

S-PLUS:

```
fRegresslist <- fRegress(tempfd, precfd,
    xLfd = harmacellfd, xlambd=1,
    yLfd = harmacellfd, ylambd=10)
alphafd <- fRegresslist[[1]] regbifd <- fRegresslist[[2]]
par(mfrow=c(1,2),pty="s") plot(alphafd, main="Intercept Function")
regbifdmat <- eval.bifd(weeks, weeks, regbifd) persp(regbifdmat,
```

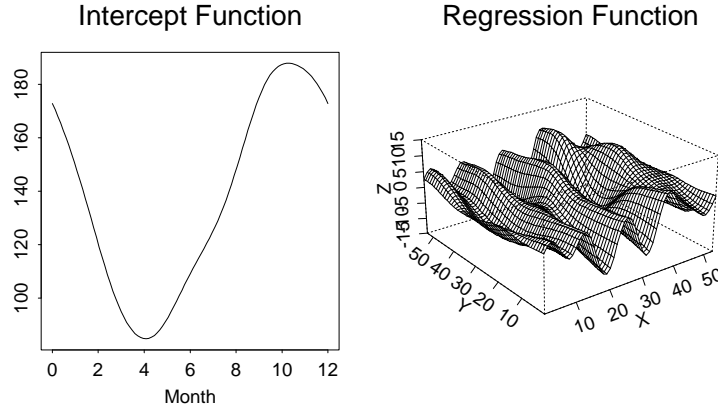


Figure 9: The left panel displays the intercept function resulting from fitting the precipitation functions by a linear model using the temperature functions as the independent variable. The right panel shows the corresponding bivariate regression function.

```
xlab='Weeks', ylab='Weeks') title("Regression Function")
```

Of course, in each of these analyses there is much more work to be done to aid interpretation and evaluate the results. These analyses are designed only to introduce the use of these functional data objects and functions.

5.2 The Lip Data: A Landmark Registration Example

We use these data to illustrate the smoothing and the landmark registration processes. There are 20 curves, each observed at 51 sampling points. We want to use a B-spline expansion that has four derivatives, since we will need to look at acceleration.

First we set up the `basis` object as a B-spline basis with 31 basis functions and order 6. The high order is needed because we are going to estimate the acceleration, and do so by penalizing the size of the fourth derivative.

```
liptime = (0:0.02:1)';
```

```
lipbasis = create_bspline_basis([0,1], 31, 6);

liptime  <- seq(0,1,.02)
lipbasis <- create_bspline_basis([0,1], 31, 6)
```

Now we set up the discrete data and pass them through `data2fd` to get the fd object `lipfd`.

```
fid = fopen('lip.dat','rt');
lipmat = reshape(fscanf(fid,'%f'), [51, 20]);
lipfd = data2fd(lipmat, liptime, lipbasis, ...
               {'Normalized time', 'Replications', 'mm'});

lipmat  <- matrix(scan("lip.dat", 0), 51, 20)
lipfd <- data2fd(lipmat, liptime, lipbasis,
               fdnames=c('Normalized time', 'Replications', 'mm'))
```

An inspection of the second derivatives has already convinced us that a little smoothing would help. We now apply smoothing by penalizing the fourth derivative, so as to have acceleration curves with reasonable curvature.

```
lipfd = smooth(lipfd, lipbasisobj, 4, 1e-12) lipfd <-
smooth.fd(lipfd, lipbasisobj,
          Lfd=4, lambda=1e-12)
```

The curves have two distinctive features: a clearly defined minimum at around $t = 0.4$ and an elbow at about $t = 0.75$. We could perhaps see these more clearly as peaks in the acceleration curve, but for purposes of illustration, let's locate these features by eye in each curve.

In addition, we need to locate these features in the mean curve, since we are going to warp time for each curve so that the location of the features in warped time corresponds to their timings for the mean curve. But in this example we let the registration function do this by using the average landmark timing for each feature.

Here is the code for getting the landmarks for each curve.
MATLAB:


```

nmarks    = 2;
lipmarks = zeros(nobs,nmarks);
index     = zeros(nmarks,1);
subplot(1,1,1)
for i = 1:nobs
    plot(liptime, D2lipmat(:,i), 'o', [0,1], [0,0], ':')
    title(['Curve ',num2str(i)])
    for j = 1:nmarks
        [x y] = ginput(1);
        index(j) = round(x*51);
    end
    lipmarks(i,:) = liptime(index)';
end
lipmeanmarks = mean(lipmarks);

```

S-PLUS:

```

nmarks <- 2
lipmat <- eval.fd(liptime,lipfd)
lipmarks <- matrix(0,20,nmarks)
par(mfrow=c(1,1), pty="m")
for (i in 1:20) {
    plot(liptime, lipmat[,i], main=paste("Curve",i))
    abline(v=lipmarksmean,lty=2)
    index <- identify(liptime, lipmat[,i], n=nmarks)
    lipmarks[i,] <- liptime[index]
}
lipmeanmarks <- apply(lipmarks,2,mean)

```

We have to set up a basis for the warping functions.

```

wbasis = create_bspline_basis([0,1], 4, 6, [0,lipmeanmarks,1]);

wbasis <- create.bspline.basis(c(0,1), 4, 6, c(0,lipmeanmarks,1))

```

Now we can register the curves, calling function `landmarkreg`.

```

lmrkstr = landmarkreg(lipfd, lipmarks, lipmeanmarks, wbasis);
lipregfd = lmrkstr.regfd;

```

```
lipwarpfd = lmrkstr.warpfd;
```

```
lipreglist <- landmarkreg(lipfd, lipmeanfd, lipmarks, wbasis)
lipregfd    <- lipreglist[[1]]
lipwarpfd   <- lipreglist[[2]]
```

Finally, we plot the unregistered and registered curves, as well as the warping functions. These plots are in Figures 21 and 11.

```
subplot(1,2,1)
plot(lipfd, axis('square'))
title('Unregistered')
subplot(1,2,2)
plot(lipregfd, axis('square'))
title('Registered')

par(mfrow=c(1,2), pty="s")
plot(lipfd, main="Unregistered")
abline(v=lipmeanmarks,lty=2)
plot(lipregfd, main="Registered")
abline(v=lipmeanmarks,lty=2)
```

6 Some Additional Notes on Monotone Smoothing

6.1 Introduction

Monotone functions have played an important role in data analysis since the development and statistical application of isotone regression techniques (Bartholomew, 1959; Kruskal, 1965) and the Box-Cox transformation, $f(x) = (x^\lambda - 1)/\lambda$, (Box and Cox, 1964). Wright and Wegman (1980) provide a broad review of the considerations involved in combining monotonicity with smoothness, along with some valuable existence results.

Figure 13 displays a monotone smoothing problem. In a study of the growth of children of 5 to 8 years of age over a 312 day period by Thalange,

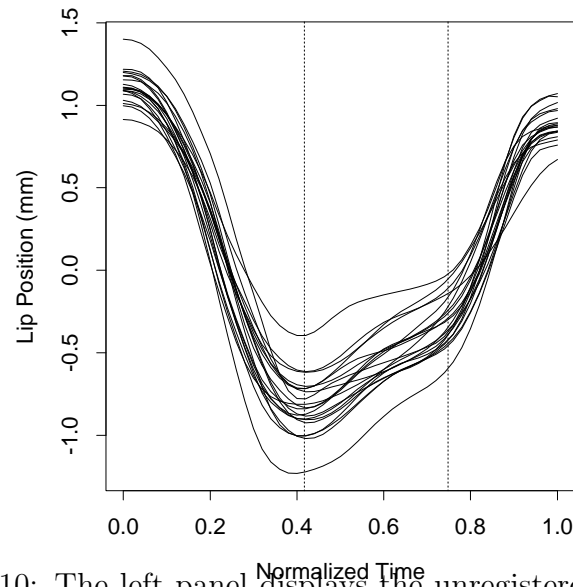


Figure 10: The left panel displays the unregistered lip position curves, and the right panel the registered versions. The vertical dashed lines are the timings of the two landmark features, the minimum and the elbow, for the mean curve.

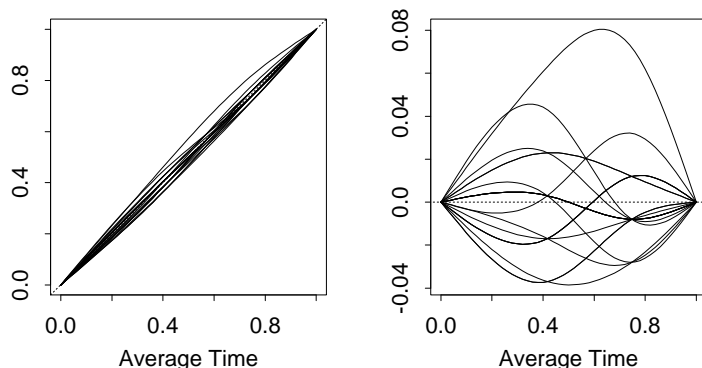


Figure 11: The 20 warping functions estimated by registering the lip curves using two landmark features.

et al (1996), it was observed that growth tends to occur in short bursts rather than continuously. The Figure shows data for a single child, and also contains two smooth curves: A cubic smoothing spline fit using the GCV criterion for bandwidth selection, and the monotone smooth described in this paper. Although it would seem reasonable to assume that height increases monotonically, the smoothing spline is far from monotonic, and especially in the gap in recording at about 100 days due to the Christmas vacation period. On the other hand, the data might seem to call for a reduction in height immediately after Christmas, and thus testing the hypothesis of monotonicity could be useful.

Monotonicity can be a useful way of regularizing or stabilizing estimated functions, since the imposition of monotonicity can remove the small-scale wiggles common near the boundaries or resulting from only lightly smoothing. We see this in Figure 13 where the monotone function behaves more reasonably than the cubic smoothing spline at both the lower boundary and over the Christmas period.

Smoothness in a monotone function can be essential. In the growth curve

Figure 12: The points are heights of a child recorded over a 312 day period. The solid line is a twice-differentiable monotone smooth of the the data using the technique described in this paper, and the dotted curve is a cubic smoothing spline with bandwidth chosen by minimizing the GCV criterion. The gaps in the record at about 100 and 200 days are due to the Christmas and Easter vacations, respectively.

Figure 13: The solid curve is velocity of growth estimated by the first derivative of the monotone smoothing function shown in Figure 13. The dotted lines indicate 95% pointwise confidence limits estimated by bootstrapping residuals.

example, both the first and second derivatives are considered interesting by specialists, and are especially important in characterizing possible spurts in growth. Figure ?? displays the estimated growth velocity for the growth data along with confidence bands estimated by bootstrapping, and shows the burst-like character of short-term growth. In another context, it can be important to use the Jacobian when transforming the dependent variable in a regression or ANOVA problem (Ramsay, 1988), and thus a reasonable first derivative estimate is required. Although isotonic regression (Bartholemew, 1959; Barlow, et al, 1972) provides an elegant and computationally attractive technique for estimating monotone functions, the result is a non-differentiable step function that may not be aesthetically appealing. Finally, it may be that a variable transformation problem requires inversion of the transformation, perhaps to estimate argument values at points not corresponding to the original observations. That is, the fitted function should have a first derivative bounded away from zero.

Various techniques for estimating smooth monotone transformations have been developed. They have tended either to be very complex algorithmically, or to involve some compromise in flexibility. Ramsay (1988), for example, estimated monotone data transformations by taking linear combinations of monotone regression splines. The coefficients were constrained to be nonnegative, implying that the fitting procedure involved optimization under linear inequality constraints. Moreover, constraining coefficients to be positive is sufficient to ensure monotonicity, but not necessary, so that there is always the possibility that the fit to the data could be improved by allowing coefficients to go negative while still preserving monotonicity. Kelly and Rice (1990) developed a related approach involving constrained optimization with respect to linear combinations of regression splines. Bloch and Silverman (1997) apply a monotone transformation technique to the estimation of discriminant functions. Friedman and Tibshirani (1984) considered combining a conventional nonmonotonic smoothing procedure with isotonic regression to improve the smoothness of the result, and Mammen (1991) studied the properties of this technique.

Monotone transformation techniques are especially useful in applications where the observed variables are indicators rather than measurements; that is, they do not have intrinsic metric properties, but can be taken to reflect ordering relationships among objects to which values have been assigned. Indicator variables are prevalent in the social sciences, where there is seldom any strong argument for believing that variables have interval or ratio scale properties, but where there is often some reason to consider only smooth transformations.

The technique discussed in this paper is a computationally convenient procedure for estimating an arbitrary twice-differentiable strictly monotone function defined on an interval closed on the left, that may be taken without loss of generality to be either $[0, \infty)$ or $[0, 1]$. The transformation family can also be conveniently restricted to various special cases such as the Box-Cox transform.

6.2 A Differential Equation for Monotone Functions

The notation D^m will be used to refer to the operation of taking the derivative of order m for $m > 0$, for the identity operator for $m = 0$, and for the partial integration operator $D^{-1}f(t) = \int_0^t f(s) ds$. The class of mono-

tone functions discussed in this paper consists of those functions f for which $\ln Df$ is differentiable and $D \ln Df = D^2 f / Df$ is Lebesgue square-integrable. These conditions ensure both that the function is strictly monotone increasing ($Df > 0$) and that its first derivative is smooth and bounded almost everywhere. The following theorem states that this class is identified with a simple linear differential equation.

Theorem: Every function f of this class is representable as either

$$f = \beta_1 + \beta_2 D^{-1} \exp[D^{-1}w] \quad (8)$$

or as a solution of the homogeneous linear differential equation

$$D^2 f = w Df \quad (9)$$

where w is a Lebesgue square-integrable function and β_1 and β_2 are arbitrary constants.

Proof: If f has the form (8), then $D \ln Df = w$ and therefore $w \in L^2$ by definition of the class; and it is also clear that it satisfies (9) for $w = D \ln Df$. On the other hand, if f has representation (9), then (8) is a solution, and it is a standard result in the theory of linear differential equations that the solution space is linear and of dimension 2, and hence every solution is also of that form.

The coefficient function $w = D \ln Df = D^2 f / Df$ measures the relative curvature of the monotone function in the sense that it assesses the size of the curvature $D^2 f$ relative to the slope Df . The special case of $w = \alpha$ defines $f(x) = \beta_1 + \beta_2 \exp(\alpha x)$, so that exponential functions have constant relative curvature, and $w = 0$ defines a linear function. Thus, small or zero values of $w(x)$ correspond to locally linear functions, while very large values correspond to regions of sharp curvature.

The representation (8) also has an interpretation that may occasionally suggest a plausible model. By expanding $Df(t)$ about a fixed value t_0 , one obtains $Df(t) \approx Df(t_0)[1 + w(t_0)(t - t_0)]$, so $w(t)$ induces a local proportional change in the velocity $Df(t)$ by acting as a multiplier of $t - t_0$. That is, it defines a system in which velocity is constantly being updated by a multiplicative transformation.

The class of functions (8) can be extended to orders of differentiability $j > 0$ by replacing $D^{-1}w$ by $D^{-(j-1)}w$. The operator $M = D^{-1} \exp$ may be

termed the *monotonicity operator*, and operates on a function so as to render it monotone increasing and of one higher order of differentiability.

The Box-Cox transformation is not, strictly speaking, a member of this class since it need not be differentiable or even defined at zero. On the other hand, on the interval $[\epsilon, \infty]$, $\epsilon > 0$, the transform satisfies the differential equation

$$D^2 f(x) = \frac{1 - \lambda}{x} Df(x)$$

and is, therefore, representable in practical terms by this family.

6.3 Monotone Data Smoothing

This subsection considers the smoothing problem motivated by the model $y_i = f(t_i) + \epsilon_i$ where the values ϵ_i are assumed i.i.d. with mean zero and variance σ^2 , and the argument values are within the interval $[0, T]$.

The fitting criterion considered here is

$$F_\lambda(\mathbf{y}|w) = N^{-1} \sum_i [y_i - \beta_1 - \beta_2 m(t_i)]^2 + \lambda \int_0^T w^2(t) dt \quad (10)$$

where

$$m(t) = (D^{-1} \exp D^{-1} w)(t). \quad (11)$$

The first term is the least squares fitting criterion usual in spline smoothing, except that the linear regression parameters β_1 and β_2 are essential because $m(0) = 0$ and $Dm(0) = 1$. The first term in (10) could be generalized to include variable weights for the observations, and other loss functions such as negative log likelihood might be appropriate in certain applications, but for simplicity the exposition is limited to this case.

The second penalty or regularization term has some of the characteristics of the norm of the second derivative used in cubic spline smoothing, but the role played by the denominator in $w = D^2 f / Df$ is important since one wants the regularization to also keep the fitted function away from the boundary condition $Df = 0$. From (8) the limiting case $\lambda \rightarrow \infty$ is a straight line.

An estimate of σ^2 is

$$\hat{\sigma}^2 = (N - 2)^{-1} \sum_i [y_i - \beta_1 - \beta_2 m(t_i)]^2, \quad (12)$$

which discounts N for the two regression parameters, but ignores the impact of adapting w to the data.

The smoothing parameter may be chosen by cross-validation, that is, by removing each observation y_i in turn, minimizing (10) each time, and summing the squared differences between the value left out and the corresponding value predicted by the model. However, this technique can often fail in the sense that an infinitely large smoothing parameter is indicated, as will be seen in the example in Section 3.3. Likewise, confidence bands for the estimated function may be estimated by bootstrapping, using resampling of the residuals from the estimated function. It should be noted that these do not take into account the bias in the estimated monotone function.

6.3.1 Basis Function Expansions for w

The principal advantage brought by representation (8) is the transformation of the estimation problem from one of finding the constrained function f to one of computing the unconstrained function w . Because of this lack of constraint, w may be defined as a linear combination of some set of basis functions $\phi_k, k = 1, \dots, K$, appropriate to the problem at hand. Let $\Phi_k = D^{-1}\phi_k$, and let $\boldsymbol{\phi}$ and $\boldsymbol{\Phi}$ be the vectors $(\phi_1, \dots, \phi_K)^t$ and $(\Phi_1, \dots, \Phi_K)^t$, respectively, so that $w(t) = \mathbf{c}^t \boldsymbol{\phi}(t)$ where \mathbf{c} is the coefficient vector defining the linear combination. Then the fitted function is of the form

$$\hat{y}(t) = \beta_1 + \beta_2 m(t) = \beta_1 + \beta_2 D^{-1} \exp[\mathbf{c}^t \boldsymbol{\Phi}(t)].$$

The criterion (10) must be minimized with respect to the coefficient vector \mathbf{c} and the regression coefficients β_1 and β_2 .

Given the limited accuracy required of f in most statistical applications, the values of $D^{-1}w$ using an arbitrary basis $\boldsymbol{\phi}$ for w may be approximated by the trapezoidal rule with a sufficiently fine equally-spaced mesh of arguments. The author's experience with this simple approach has been quite satisfactory. The M-spline bases and their partial integrals, the I-spline bases, used in Ramsay (1988) are especially convenient. However, if a piecewise constant basis for w is used, f may be expressed analytically, and certain aspects of the computations can be simplified.

The author's experience has been satisfactory with the following iterated two-stage minimization of (10). Beginning with an initial estimate $\mathbf{c}^{(0)}$, which may be a vector of zeros, estimate $\beta_1^{(0)}$ and $\beta_2^{(0)}$ by linear regression. Then,

on any iteration $\nu > 0$ for which $\beta_j^{(\nu-1)}$ and $\mathbf{c}^{(\nu-1)}$ are estimates on the previous iteration, first optimize with respect to \mathbf{c} by the Gauss-Jordan or scoring procedure for nonlinear least squares problems to obtain $\mathbf{c}^{(\nu)}$, and then compute $\beta_j^{(\nu)}$ by linear regression. The Gauss-Jordan procedure requires that the update vector

$$\delta^{(\nu)} = \mathbf{c}^{(\nu)} - \mathbf{c}^{(\nu-1)}$$

be the solution of the linear equation

$$\mathbf{R}^{(\nu-1)} \delta^{(\nu)} = -\mathbf{s}^{(\nu-1)}$$

where

$$\mathbf{R} = N^{-1} \beta_1^2 \mathbf{X}^t \mathbf{X} + \lambda \mathbf{K} ,$$

matrix \mathbf{X} is N by K and has rows

$$\mathbf{x}(t_i) = \frac{\partial m(t_i)}{\partial \mathbf{c}} = \int_0^{t_i} \boldsymbol{\Phi}(s) \exp[\mathbf{c}^t \boldsymbol{\Phi}(s)] ds ,$$

symmetric matrix \mathbf{K} of order K is

$$\mathbf{K} = \int_0^T \boldsymbol{\phi}(s) \boldsymbol{\phi}^t(s) ds , \quad (13)$$

vector \mathbf{s} of length K has values

$$\mathbf{s} = -N^{-1} \beta_1 \mathbf{X}^t \mathbf{r} + \lambda \mathbf{K} \mathbf{c}$$

and, finally, where \mathbf{r} is the vector of length N containing the residuals $r_i = y_i - \beta_1 - \beta_2 \hat{m}(t_i)$. The rate of convergence of these iterations is only linear, but appears to be acceptably fast, and is usually obtained in about four to five iterations.

6.3.2 Testing for Non-monotonicity

The differential operator (9) defining the monotone function f also suggests a technique for assessing whether the data call for a nonmonotone fit. Heckman and Ramsay (1997) describe an $O(N)$ algorithm for fitting an L-spline h , defined as the minimizer of

$$F_\lambda(\mathbf{y}|L) = N^{-1} \sum_i [y_i - h(t_i)]^2 + \gamma \int_0^T (Lh)^2(t) dt . \quad (14)$$

Defining $L = wD - D^2$ for a given fixed w implies that h defined by $\gamma \rightarrow \infty$ will be the monotone function f defined by w and the appropriate values of β_1 and β_2 . Finite values of γ , however, need not correspond to monotone functions, and the limit $\gamma \rightarrow 0$ yields, as usual, an interpolating function. Thus, if the data indicate a finite value for γ corresponding to a nonmonotone function, one may reasonably question whether monotonicity was an appropriate assumption. Bowman, Jones and Gijbels (1997) developed a monotonicity test that is similar in spirit to this approach.

Heckman and Ramsay (1997) may be consulted for the details of implementing this L-spline procedure. A module for Splus (Statistical Sciences, 1995) called L-spline is available in the Statlib library accessible by ftp at stat.cmu.edu. This module requires two matrices as arguments. The first is an N by 2 matrix U containing one's in the first column, and the function values $m(t_i)$ in the second. The second contains the diagonal and first four off-diagonal bands in band-structured mode of the corresponding matrix of reproducing kernel values $K(t_i, t_j)$. These are defined for this problem as follows. Let $\mathbf{u}(t) = (1, m(t))'$, and $\mathbf{v}(t) = (-m(t)/Dm(t), 1/Dm(t))'$. Then

$$K(s, t) = \mathbf{u}'(s) \left[\int_0^{\min(s, t)} \mathbf{v}(z) \mathbf{v}'(z) dz \right] \mathbf{u}(t) .$$

The integral in this expression may be approximated by the trapezoidal rule using a fine grid of values since there is no great need for high accuracy.

A further indicator may be constructed from the F-ratio, here used only as a graphical device for displaying the credence of the monotone fit. Let SSE_∞ be the error sum of squares generated by the monotone fit, and let SSE_γ be that resulting from any specific value of γ . Let k_γ be a measure of equivalent degrees of freedom for the L-spline; this is commonly taken to be the $\text{tr } \mathbf{S}$ where \mathbf{S} is the corresponding linear smoothing matrix. Then the ratio

$$F(\gamma) = \frac{SSE_\gamma - SSE_\infty}{k_\gamma - 2} / \frac{SSE_\gamma}{k_\gamma} \quad (15)$$

should not differ greatly from unity if the L-spline fit defined by γ is not a great improvement on that offered by the monotone fit.

6.3.3 The Growth Data

The growth data in Figure 13 were smoothed using a step-function expansion of w defined by 41 equally-spaced breakpoints. This number was considered

to give enough resolution to allow the monotone function to track curve characteristics lasting a week or so. Many applications will not require nearly this number of break values.

The cross-validation criterion favored a value of λ far too small to give a reasonable estimate of the standard error σ , and consequently the value $10^{-5.5}$ was chosen, giving $\hat{\sigma} = 0.26cm$, a value that is considered reasonable for height measurements.

The confidence bands for the first derivative function displayed in Figure ?? were computed by resampling of the residuals 100 times, recomputing f each time, and finally adding and subtracting at each argument two pointwise standard deviations of the corresponding 100 bootstrapped curve values.

Applying the L-spline procedure for testing monotonicity to these data yielded ∞ as the value of λ minimizing the generalized cross-validation criterion, implying that the monotone fit in Figure 13 was optimal. Moreover, an analysis of 100 sets of data produced by resampling of the residuals yielded finite values of γ 39 times, and of these seven were nonmonotonic.

Figure 14 plots the standard error estimate and the F-ratio (15) for these data over a range of values of γ . The standard error estimate in this case discounts N by the approximate number of degrees of freedom computed by the L-spline smoother, rather than by 2 as in (12). Values of $\log_{10} \gamma < 3$ would give estimates of σ too small to be plausible. The corresponding F-ratio values are not large enough to call into question the monotonicity hypothesis.

7 Some Additional Notes on Curve Registration

7.1 Introduction

Techniques in functional data analysis (FDA) (Ramsay & Silverman 1997) can be employed to study the variation in a sample of functions $x_i, i = 1, \dots, N$, and their derivatives. In practice these functions are often a consequence of a preliminary smoothing process applied to discrete data, and in others the entire functions may be immediately available by on-line recording techniques. In any case, we want to study variation both within and between functions, and will be interested in functional analogs of descriptive

Figure 14: The left panel displays the standard error estimate $\hat{\sigma}$ for a range of smoothing parameter γ values. The F-ratio index (15) is shown in the right panel.

statistics such as means, variances, and correlations, and also in functional counterparts of multivariate methods such as regression, principal components analysis and canonical correlation analysis. The functional context also invites application of methods using derivative information, such as principal differential analysis.

However, functional data tend to show two quite distinct kinds of variation, called here *amplitude variation* and *phase variation*. Figure 15 illustrates the fact that the compounding of these two types can frustrate even the simplest analyses of replicated curves. Ten estimates of the acceleration in height show individually the salient features of growth in children: the large deceleration during infancy is followed by a rather complex but small-sized acceleration phase during late childhood, and then the dramatic acceleration-deceleration pulses of the pubertal growth spurt finally give way to zero acceleration in adulthood. Note that the timing of these salient features obviously varies from child to child. Ignoring this timing variation in computing a cross-sectional mean function (the heavy dashed line in Figure 15) can result in a estimate of average acceleration that does not resemble any of the observed curves: the mean curve has less variation during the puberty than any single curve, and the duration of the mean pubertal growth spurt is rather larger than for any individual curve.

Figure 16 displays a similar problem for mean temperature records of two Canadian cities; the marine climate of St. John’s, Newfoundland, is associated with rather later seasons than is the continental climate of Edmonton, Alberta. Before studying other ways in which the two curves differ, one needs to consider how their seasons can be compared on the same time scale.

The upper left panel of Figure 17 presents a particularly common registration problem. In an experiment described in Ramsay, Wang and Flanagan (1995), the force exerted by the thumb and forefinger was recorded during twenty brief pinches applied to a force meter, with a background force of about two Newtons applied before and after the pinch. The starting time for each record was arbitrary, so that it was essential to find a common origin in time in order to combine information across the records.

Figure 18 presents an even more complex problem. The solid curve displays the horizontal trace of the author printing “fda”, and this event took 2.49 seconds. The dashed line displays the cross-sectional mean of 20 replications, computed by first interpolating the raw data to a fixed number of sampling points. The average and the standard deviations of the printing

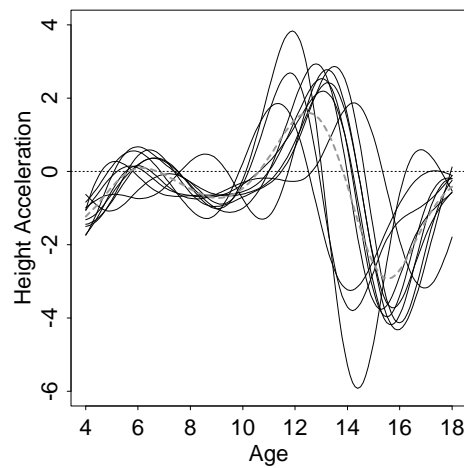


Figure 15: Ten height acceleration curves ($cm/year^2$) for boys estimated by Ramsay, Bock and Gasser (1995). The heavy dashed line is the cross-sectional mean, and illustrates the fact that averaging unregistered curves can result in an average that does not resemble any sample curve. The heavy solid line is the cross-sectional mean after registration of the curves.

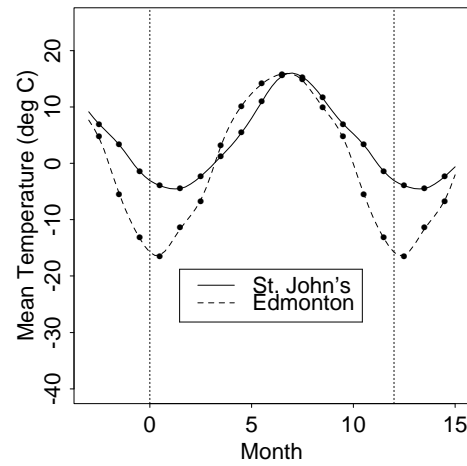


Figure 16: Mean temperature records for Edmonton, Alberta, and St. John's, Newfoundland. The seasons change later in St. John's than they do in Edmonton.

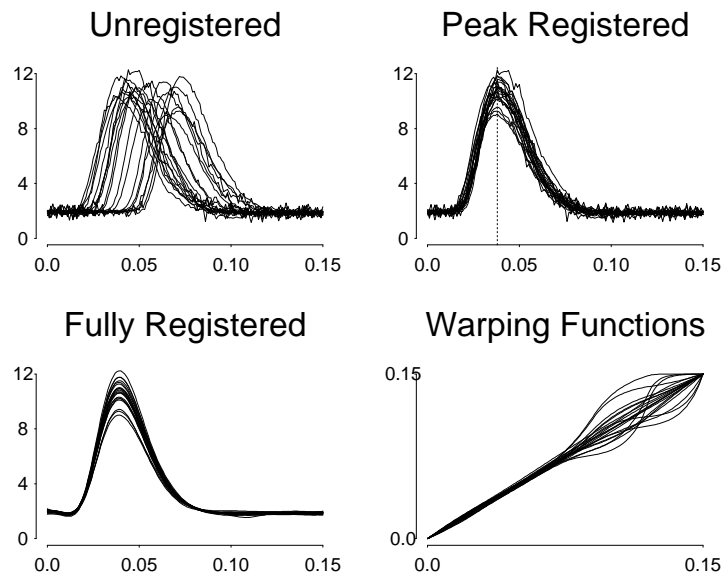


Figure 17: The upper left panel contains 20 records of force (Newtons) exerted by the thumb and forefinger with a maintained background force of 2 Newtons. The starting time (seconds) of each record is arbitrary. The upper right panel contains these records with the times of maximum force (the vertical dotted line) being aligned. The lower left panel shows the completely registered force functions, and the lower right panel displays the time-warping functions that register them.

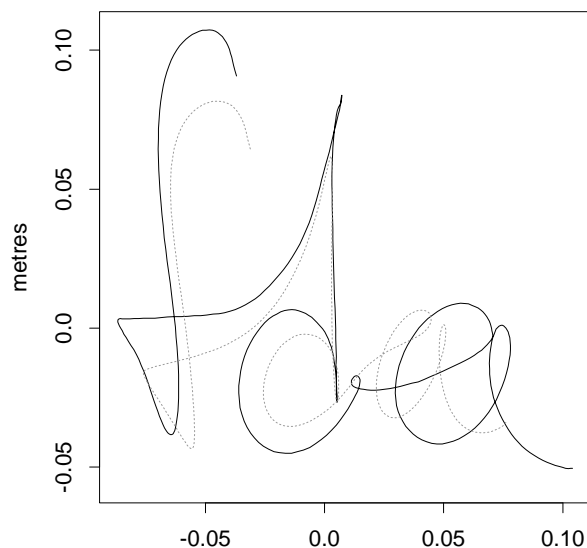


Figure 18: The solid line is a single replication of the author printing “fda”. The dotted line is the mean across 20 replications, where the individual curves vary in starting time and duration of printing.

times for these scripts were 2.35 and 0.145 seconds, respectively. Here both the origin and the scale of time vary from replication to replication, a situation that holds in many applications. Moreover the functions observed, consisting of the X-, Y- and Z-coordinates of pen position, are multivariate. What meaning can we attach to the cross-sectional mean computed in this manner? We see that the mean differs considerably in shape from this replicate, and in fact, the mean curve calculated in this way is smaller than almost all of the individual replicates. Should we convert these scripts to a common time scale? Not if we want to retain the meaning of derivatives of the coordinate functions, since changes in time scale imply changes in the scale of derivatives as well.

These examples illustrate that the rigid metric of physical time may not be directly relevant to the internal dynamics of many real-life systems. Rather, there can be a sort of physiological or meteorological time scale that relates nonlinearly to physical time and varies from case to case. Human growth is, ignoring external factors, largely a consequence of a complex sequence of

hormonal events that do not happen at the same rate from child to child, and also have a variable rate over the growth of a specific child. Weather is driven by ocean currents, reflectance changes for land surfaces and other factors that are timed differently for different spatial locations. And finally muscle contractions do not build up and release at exactly the same rate from one pinch to another.

Put more formally, the values $x_i(t_j)$ of two or more functions may differ because of two types of variation. The first is the more familiar *amplitude variation* or vertical variation due to the fact that two functions x_1 and x_2 may simply differ at points of time at which they can be compared. But they may also exhibit *phase variation* in the sense that x_1 and x_2 should not be compared at a fixed time t , but at times t_1 and t_2 at which the two processes are essentially in comparable states, so that the curves exhibit comparable features at these times. For example, the intensity of the pubertal growth spurts of two children should be compared at their respective ages of peak velocity defined by $D^2x_1(t_1) = D^2x_2(t_2) = 0$, rather than at any fixed age. Here we use the notation D^m to mean the process of taking the m th derivative of a function, so that $D^2x(t)$ is the value of the second derivative or acceleration at time t . As another example we want to compare two scripts at time points at which features such as the cusp in “d” in Figure 18 are being created.

The problem of transforming the arguments of curves so as to align various salient features has a very large literature in many different fields. The problem is referred in this paper and by Silverman (1995) as *curve registration*, the engineering literature tends to the evocative term *time warping* (Sakoe & Chiba, 1978; Wang & Gasser, 1998), and the process of registering curves for purposes of computing average curves is called *structural averaging* by Kneip and Gasser (1988, 1992). Registering outcomes over surfaces and volumes is especially important in medical imaging (Bookstein, 1991).

7.2 Formulation of the Registration Problem

The curve registration problem can be expressed formally as follows. Let N functions x_i be defined on closed real intervals that can be taken without loss of generality as $[0, T_i]$. These functions may be vector-valued, as would be the case, for example, if they indicated positions in two- or three-dimensional space, or simultaneous growth in several aspects of the skeleton. The upper

boundaries may either vary or may be fixed.

In practice the boundaries of the interval are often defined by marker events such as birth and a fixed adult age for the growth data, or by arbitrary values such as midnight on December 31 for the weather data. Or it may be that the interval is simply large enough to include all of the curves of interest plus some tail behavior of little concern. In the event that the functions are periodic with known period, it will be assumed that each x_i is extended beyond $[0, T_i]$ if there is a need to use information beyond the interval. Thus, for periodic data we can also permit the study of the sampled functions over the intervals $[\delta, T_i + \delta]$ for any δ .

Let time interval $[0, T_0]$ be a standard interval. It may, for example, be the average interval $[0, \bar{T}]$. Let $h_i(t)$ be a transformation of time t for case i with domain either $[0, T_0]$ for nonperiodic data, or $[\delta, T_0 + \delta]$ for periodic data, as are displayed in Figure 16. The fact that the timings of events remain the same order regardless of the time scale implies that h_i , the time warping function should be strictly increasing, i.e., $h_i(t_1) > h_i(t_2)$ for $t_1 > t_2$. This strictly increasing condition ensures that the function h_i is *invertible*, meaning that we can always solve the equation $y = h(t)$ for t given value y . We use the notation h_i^{-1} to denote the function such that $h_i^{-1}y = h_i^{-1}[h_i(t)] = t$. Don't confuse this notation with the reciprocal of h , a concept that we won't need in this paper. Invertibility of h_i implies that for a specific event the time points on two different time scales correspond to each other uniquely. In addition, $h_i(t)$ must satisfy the boundary conditions $h_i(0) = 0$ and $h_i(T_0) = T_i$. We may, in addition, require that $h_i(t)$ be a smooth function of t in the sense of being differentiable a certain number of times. This will be important, for example, if we went to use the derivatives of registered functions.

Let $x_0(t)$ be a fixed function defined over $[0, T_0]$ that provides a sort of template for the individual curves x_i in the sense that after registration, the features of x_i will be aligned in some sense to those of x_0 . We can propose, for example, the model

$$x_i[h_i(t)] = x_0(t) + \epsilon_i(t) \quad \text{or} \quad x_i \circ h_i = x_0 + \epsilon_i, \quad (16)$$

where ϵ is small relative to x_i and roughly centered about 0. If, alternatively, the template x_0 may defined by discrete values x_{j0} , $j = 1, \dots, n$, then our model becomes

$$x_i[h_i(t_j)] = x_{j0} + \epsilon_{ij}. \quad (17)$$

Because we assume that ϵ is small relative to x_i , this model postulates that major differences in shape between target function x_0 and specific function x_i are due only to phase variation.

A more complex model for inter-curve variation combining phase and amplitude variation might be

$$x_i[h_i(t)] = A_i(t)x_0(t) + \epsilon_i(t) \quad \text{or} \quad x_i \circ h_i = A_i x_0 + \epsilon_i, \quad (18)$$

where $A_i(t)$ is an amplitude modulation function, probably constrained to be positive. The discrete version of this is

$$x_i[h_i(t_j)] = A_i(t_j)x_{j0} + \epsilon_{ij}. \quad (19)$$

Here, a practical solution to the registration problem will depend on $A_i(t)$ varying rather slowly relative to $x_i(t)$, as well as on the size variation in residual function $\epsilon_i(t)$ being relatively small. One practical way of achieving this is to register a derivative $D^m x_i$ rather than x_i itself, since derivatives tend to vary rather more rapidly than the functions themselves.

Suppose, now, that we have managed to identify these N warping functions $h_i(t)$. We can then calculate the *registered functions* $x_i^*(t)$ as follows:

1. Compute values of the inverse function $h_i^{-1}(t)$ for a fine grid of values of t . This is typically achieved to representing the relationship between the values $h_i(t)$ put on the abscissa of a plot and t put on its ordinate as a smooth function. Note that care must be taken to ensure that this relation is strictly increasing.
2. Compute the values of $x_i(t)$ for the same fine grid of t -values.
3. Represent the relation between $h_i^{-1}(t)$ and $x_i(t)$ as a function using the same techniques used to get the unregistered functions x_i . These new functions are the registered functions $x_i^*(t)$ in the sense that their features will be aligned with those of the template $x_0(t)$, and therefore will tend to occur at the same times across replications.

The registration task, then, is to estimate the time-warping functions h_i so that the de-warped components x_i can be studied separately, along with possible analyses of the functions h_i as well.

7.3 Two Registration Techniques

In this subsection, we look closely at two methods for registering curves. The first, marker registration, requires the identification of the location of a number of visible features in each curve to be registered. Provided that these features are unambiguously localizable, and the number of curves and features is modest, the method is especially easy to use and easy to understand.

The second method involves using the entire curve, rather than just the location of certain features. Although more technical, the method is completely automatic, and especially handy when features are hard to pin down in certain curves and/or a large number of curves has to be processed.

7.3.1 Marker or Landmark Registration

Marker registration is often used in engineering, biology, physiology and other fields. It is the process of aligning curves by identifying the timing of salient features in the curves. These features are often peaks or valleys, but can also be the crossing by a curve of a threshold. The zero of acceleration during the pubertal growth spurt and optimal temperature timings are examples. In fact, if the derivative of a curve can be easily evaluated, both a peak and a valley can be located by finding the time at which the derivative crosses zero.

More complex features are also possible, but what is essential in landmark registration is that the ℓ th feature among L for curve $x_i(t)$ be localizable at a unique time value $t_{i\ell}$. The timing of the corresponding feature in the template function $x_0(t)$ is $t_{0\ell}$. In this notation, we can designate the beginnings and ends of curves as features with timings 0 and T_i , respectively, and indicate the entire sequence of feature timings as $t_{i\ell}, \ell = 0, \dots, L+1$, where L is the number of features within the interior of the interval $[0, T_i]$.

Registration of curve $x_i(t)$ is then a question of computing a strictly monotone function $h_i(t)$, the *warping function*, such that

$$x_i[h(t_{i\ell})] = x_0(t_{0\ell}), \quad \ell = 0, \dots, L+1.$$

Using this strategy, curves are aligned by transforming time so that marker events occur at the same values of the transformed times. Comparisons between marker timings can also be made by using corresponding transformed times.

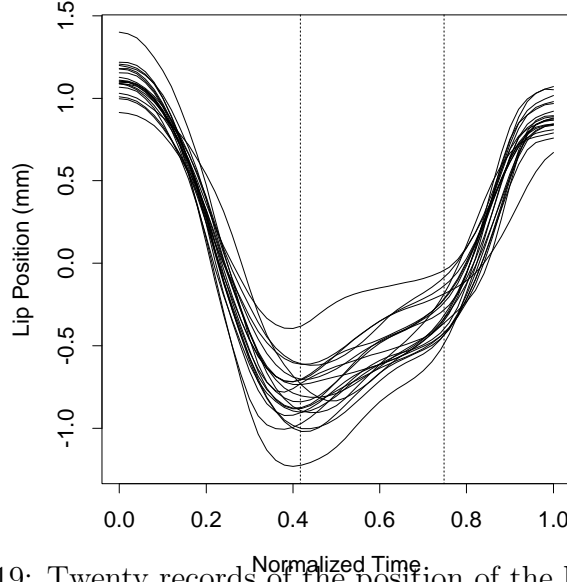


Figure 19: Twenty records of the position of the lower lip during the phrase “bob”. The duration of the syllable has been normalized to the interval $[0,1]$. The vertical dotted lines indicate the average timings of the minimum and the elbow.

Consider as an example the curves displayed in Figure 19. We see two obvious landmarks in each curve: the minimum position and the elbow. The average timings, t_{01} and t_{02} , for these two features are 0.42 and 0.75, respectively, and are indicated in the Figure as vertical dotted lines. But for the first curve, the minimum occurs later at $t_{11} = 0.48$, and the elbow at $t_{12} = 0.82$. This is plotted in Figure 20, where both the timings of the two landmarks and a smooth curve $h(t)$ passing through these points as well as $(0,0)$ and $(1,1)$ is indicated. The curve is above the diagonal line, indicating that timings of features for curve 1 are later than average. For example, indicating the average curve by $x_0(t)$, we have, for the minimum, $x_1(0.48) = x_1(h(0.42)) \approx x_0(0.42)$. This relation is only approximately true because, while $x_1(0.48)$ and $x_0(0.42)$ are both minimum values, these values need not be exactly equal.

Figure 21 displays the twenty curves registered so that the minima and elbows all occur at the same times. In the S-PLUS and MATLAB functions

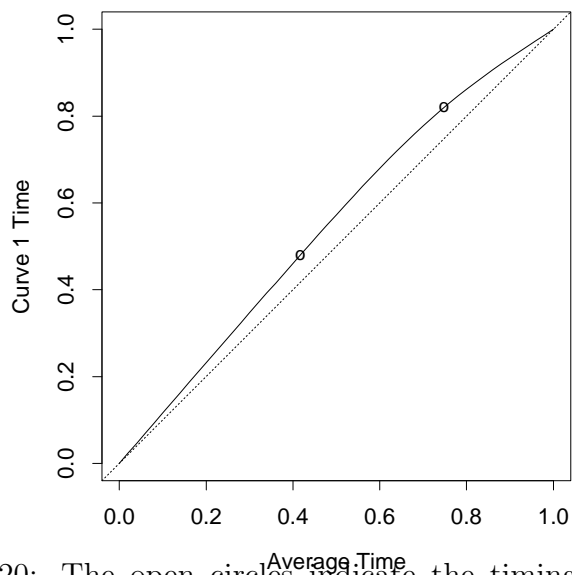


Figure 20: The open circles indicate the timings of the minima and the elbows for the first lip curve in Figure 19 and average lip curve. The smooth curve $h(t)$ has been constructed to as to pass through these points as well as points(0,0) and (1,1).

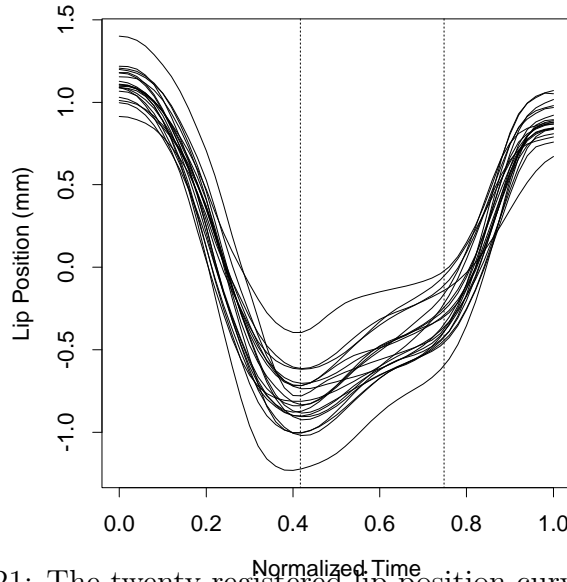


Figure 21: The twenty registered lip position curves. Landmark registration was used to make the timings of the minima and elbows identical.

available by ftp from

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

the function that implements landmark registration is `landmarkreg`.

Sakoe & Chiba (1978) estimated the values of h at marker timings by minimizing the sum of weighted distances of two speech patterns at the marker timings and imposing monotonicity and continuity on h . They solved for the discrete values of h by using a dynamic programming algorithm, and their method is referred to as *dynamic time warping*, especially in the engineering literature. Kneip and Gasser (1988, 1992) described marker registration in detail from a statistical perspective.

However, marker or landmark registration can present some problems: Marker events may be missing from certain curves, marker timing estimates can often be difficult to obtain, and may be ambiguous. Moreover, constructing automatic algorithms to identify landmarks may be a tricky exercise, and users may prefer to identify landmarks by eye. In this case, landmark identification will be a time-consuming and tedious exercise that may not be

practical when large numbers of curves are to be registered. These issues are discussed in the context human growth curves by Ramsay, Bock and Gasser (1995).

7.3.2 Continuous Registration

We may also register two curves by optimizing some measure of similarity of their shapes, and thus use the entire curves in the process. Put another way, the timings of a fixed set of landmarks provide one way of describing how similar the shapes of two curves are, but we can also choose measures which use the whole curves.

Silverman (1995) optimized a global fitting criterion with respect to a restricted parametric family of transformations of time shifts, and applied this approach to estimating a shift in time for each of the temperature functions in 35 Canadian weather stations. He also incorporated this shift into a principal components analysis of the variation among curves, thus explicitly partitioning variation into range and domain components. His measure of shape similarity was the familiar least squares criterion, recast into functional terms as follows:

$$F(h_i) = \int_0^{T_i} \{x_i[h_i(t)] - x_0(t)\}^2 dt . \quad (20)$$

This measure works well enough provided that the warping function h_i is severely restricted in its complexity. However, the measure can run into trouble for more flexible warping functions when $x_i(t)$ and $x_0(t)$ have the same shape but differ in amplitude. Ramsay and Li (1998) offer an example in which it is shown that this criterion has a tendency to “pinch in” the sides of the larger of the two curves in order to make it look more like the smaller.

Suppose that the values of $x_0(t)$ and $x_i(t)$ differ only by a scale factor, so that $x_0(t) = Ax_i(t)$ for some positive constant A . This corresponds to $A_i(t) = A$ and $\epsilon_i(t) = 0$ for all t in model (18). This means that the two functions have essentially the same shape. In this case, if we plot the *values* of one function against the other across for each value t in a fine grid, we will see a straight line passing through the origin, but with slope either A or $1/A$ depending on which curve we assign to which coordinate.

Suppose that we now consider curve values as a set of points to which we might apply principal components analysis. If curve values are proportional,

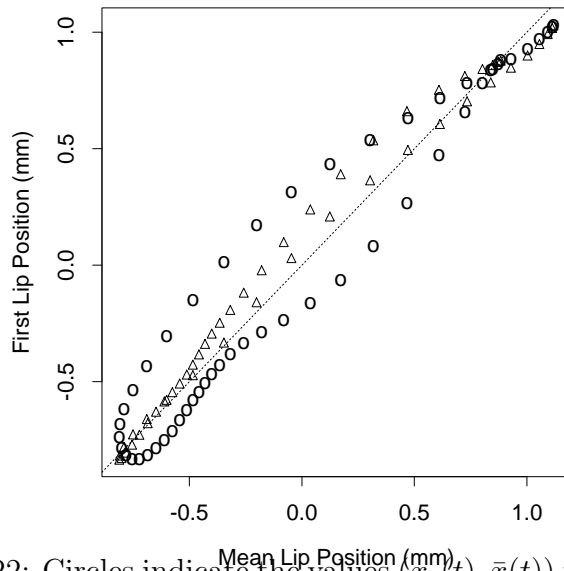


Figure 22: Circles indicate the values $(x_1(t), \bar{x}(t))$ for 51 equally spaced values of t . Triangles indicate the corresponding points after registration.

then such an analysis would yield only one nonzero principal component. That is, only one of the two eigenvalues of the matrix

$$\begin{bmatrix} \int x_0^2(t) dt & \int x_0(t)x_i(t) dt \\ \int x_0(t)x_i(t) dt & \int x_i^2(t) dt \end{bmatrix}$$

is going to be nonzero. The integrals in this matrix expression are appropriate if we have curve values for every value of time t , or if we have curve values for a very fine mesh of values t_j . Otherwise, we may prefer to replace integrals by summations.

Consider, for example, the first lip position curve $x_1(t)$ and the average lip position curve $x_0(t) = \bar{x}(t)$. The 51 observed points are plotted as circles in Figure 22. Because of the later timings of features for curve 1, the values are not proportional, and exhibit a fair amount of what sometimes called *hysteresis*. The ratio of the first eigenvalue to the second is about 62. But if we do the same for the first registered curve, plotted as triangles in the Figure, we now see much more collinearity, and the eigenvalue ratio is now about 206.

This line of reasoning suggests that we might choose warping function $h_i(t)$ to as to minimize the logarithm of the smallest eigenvalue of the cross-product matrix

$$F(h_i) = \log \mu_2 \begin{bmatrix} \int x_0^2(t) dt & \int x_0(t)x_i[h_i(t)] dt \\ \int x_0(t)x_i[h_i(t)] dt & \int x_i^2[h_i(t)] dt \end{bmatrix} \quad (21)$$

In the event that functions are multivariate, such as is the case for the hand-writing data, we shall form a composite criterion by adding this criterion across functions.

Details on how we work with this continuous registration criterion are deferred to later, after we consider how to define a suitable family of warping functions $h_i(t)$ in the next Section. However, we observe here that this criterion tends to work even better if we replace the function values by their first derivatives, or even a higher derivative if it can be estimated stably. This is because derivatives tend to oscillate more rapidly than functions, and also to vary about zero, so that the smallest eigenvalue measure is even more sensitive to whether or not functions differ only by amplitude variation.

7.3.3 A Test of Continuous Registration

We can see how these two techniques work on an artificial example, using an S-PLUS function `registerfd` that will be described in more detail below. Let the target function be $x(t) = \sin(2\pi t)$, and let the function to be registered be $x(t) = \sqrt{2}[\sin(2\pi t) + \cos(2\pi t)]$. These two functions have a phase difference of $1/8$, and $x(t)$ has a maximum of 2 as compared to the maximum of $x_0(t)$ of 1. Otherwise, the two functions have the same shape.

Here is the S-PLUS code to set up 101 discrete values of these two functions for registration.

```
x <- seq(0,1.0,.01) y <- sqrt(2)*(sin(2*pi*x) + cos(2*pi*x)) y0 <- sin(2*pi*x)
```

Now we convert these two sets of discrete values to functional data objects, using a Fourier basis with 101 basis functions.

```
basisobj <- create.fourier.basis(c(0,1),101) fd <- data2fd(y, x, basisobj) fd0 <- data2fd(y0, x, basisobj)
```

Here is the code using S-PLUS function `registerfd` to set up the registration, and to set of the registered curve $x^*(t)$ in functional data object `yregfd` and the warping function $h(t)$ in functional data object `warpfd`. Notice that we must tell the function that these are periodic data, and the constant phase shift is saved as variable `delta`. Also, note that by default `registerfd` uses the minimum eigenvalue criterion.

```
result <- registerfd(fd, fd0, periodic=T) yregfd <- result[[1]] warpfd
<- result[[2]] delta <- result[[4]]
```

The results are shown in the upper two panels of Figure `regist`, where we see that the registered function is a lateral shift by 0.125 of the unregistered function. In the upper right panel, we see as expected that $h(t) \approx t$.

The problem with the least squares criterion (20) is vividly present in the bottom two panels, resulting from running `registerfd` with the optional argument `crit=1`. We see that this criterion is minimized in the presence of considerable amplitude differences by pinching in the larger curve over amplitudes where both the smaller and larger curve have values. The resulting warping function is far from diagonal, and even the lateral shift is poorly estimated, with a value of 0.117.

7.4 Defining Smooth Warping Functions h

The warping functions h are required by most applications to be both monotone and smooth. In the case of landmark registration, if there are several landmarks, and if they are reasonably evenly distributed, the use of a standard smoothing method for the relationship between the timings $t_{i\ell}$ and $t_{0\ell}$ may satisfy these requirements. For example, we may use a B-spline basis for h_i , or use a smoothing spline algorithm such as the `smooth.Pspline` function provided with the FDA functions. These approaches, however, cannot assure monotonicity, and may get us into trouble from time to time.

More generally, and within continuous smoothing algorithms such as that used in the function `registerfd`, assurance that $h_i(t)$ is strictly increasing must be built into the method. We use a method for constructing smooth monotone functions that was developed and described in detail in Ramsay (1998). Here is a brief outline of how these are defined.

Suppose that a function h has an integrable second derivative in addition to being strictly increasing. Then every such function can be described by

Figure 23: regist

The upper two panels show results for an artificial registration problem using the minimum eigenvalue criterion. The dotted curve in the upper right panel is the curve to be registered to the curve indicated by the dashed line. The solid line is the registered curve. The upper right panel contains the warping function. The lower panels show the same results using the least squares criterion.

the homogeneous linear differential equation

$$D^2h = wDh \quad (22)$$

because a strictly monotone function has a nonzero derivative, and hence weight function w is simply D^2h/Dh , or the *relative curvature* of h . This equation, subject to the requirement that $h(0) = 0$ and $h(T_0) = T_i$, has the solution

$$h(t) = C_1 \int_0^t \exp \int_0^u w(v) dv du . \quad (23)$$

We get a bit fancy with notation, and also write $h(t)$ as

$$h(t) = C_1(D^{-1} \exp D^{-1}w)(t) = C_1(MD^{-1}w)(t) .$$

Here the notation D^{-1} means computing the indefinite integral, and is natural since $D^{-1}Dx = x$. Function $W(T) = D^{-1}w(t)$ is often referred to as either the *antiderivative* of $w(t)$, or its *primitive*. The constant C_1 is necessarily $T_i/[D^{-1} \exp D^{-1}w(T_0)]$.

The integration-rectification operator, $M = D^{-1} \exp$, which in this case maps the function $D^{-1}w$ into a twice-differentiable monotone function, may be called the *monotonization operator*. When w is constant, $h(t) = (C_1/w) \exp(wt)$, so that an exponential function has constant relative curvature. A straight line is implied by $w = 0$.

The relative curvature w can also be seen as the rate of the local percentage change in Dh . The Taylor expansion of Dh at t_0 yields

$$Dh(t) \approx Dh(t_0)[1 + w(t_0)(t - t_0)] .$$

Thus $w(t_0)$ is approximately the proportional change in Dh per unit time at $t = t_0$.

Just as using *log* or *exp* functions to eliminate the need of imposing positivity in many situations, using this monotone family eliminates the need of imposing monotonicity on the time transformation functions h by allowing us to estimate the unconstrained function w .

7.5 Estimation of Warping Function h_i

In this subsection we look at some further computational issues and details. We can achieve some simplification of notation by dropping the subscript on the function $x_i(t)$ to be registered as well as the warping function $h_i(t)$, and by indicating the fixed target function $x_0(t)$ as $y(t)$.

7.5.1 Roughness Penalties on $w(t)$

The S-PLUS and MATLAB function `registerfd` offers a choice between the two fitting criteria $F(h)$ defined above. But in addition, the function permits a penalty on the roughness of $w(t)$, or, equivalently, on $W(t) = D^{-1}w(t)$. This is achieved by minimizing

$$F_\lambda(h) = F(t) + \lambda \int [D^m w(t)]^2 dt, \quad (24)$$

For either criterion, if $m = 0$, larger values of smoothing parameter λ shrink the relative curvature $w = D^2 h / Dh$ to zero, and therefore shrink $h(t)$ to t . Moreover, since the relative curvature measure w is scale free, appropriate values of λ tend not to vary much from one application to another. We find, for example, that λ values of 10^{-4} , 10^{-3} , and 10^{-2} have worked well over a range of applications.

Note, however, that if we need to estimate derivatives of $h(t)$, it may be better to work with higher values of m . This can happen, for example, if we want to use derivatives of the registered functions with respect to t , in which case the chain rule will imply that we take the corresponding derivatives of $h(t)$. Specifically, if the first derivative is needed, using $m = 1$ will effectively penalize the total curvature this derivative, and thus keep it as smooth as desired. In the limit $\lambda \rightarrow \infty$, this will ensure that $w(t)$ is a constant.

In the analyses reported in this guide, the function w is represented by a linear combination of B-spline bases

$$w(t) = \sum_{k=0}^K c_k B_k(t). \quad (25)$$

The B-spline bases are of a specified order and defined by a breakpoint sequence $\xi_l, l = 1, \dots, L$. The definition (23) of h involves two partial integrals, and although the use of quadrature schemes even as simple as the trapezoidal rule is quite practical for computational purposes, it would be desirable in many problems to have an explicit expression for h . Accordingly Ramsay and Li (1998) used order 1 B-spline bases for w , since this permits the expression of h in a closed form and leads to relatively fast computation. But function `registerfd` allows for an arbitrary choice of basis in the expansion of $w(t)$.

7.5.2 The Procrustes Fitting Criterion

The Procrustes fitting process, used in many multivariate data analysis problems, involves the alternation between using the data to define a target for defining a particular transformation of each observation, and estimating the transformations themselves. In the applications, the cross-sectional average $\bar{x}^{(0)}(t)$ of the unregistered curves is used as the initial target y for the estimation of each sample warping function $h_i^{(1)}$. If the curves have obvious landmarks, they may also be aligned prior to computing $\bar{x}^{(0)}(t)$.

Once these warping functions have been estimated, an updated cross-sectional average

$$y(t) = \bar{x}^{(1)}(t) = N^{-1} \sum_i x_i[h_i^{(1)}(t)]$$

can be computed and used as a target for computing revised warping functions. Our experience indicates, however, that there is seldom any need for this revision, since the change in the h_i 's from the first to the second iteration tends to be negligible.

References

- Barlow, R., Bartholomew, D., Bremner, J., and Brunk, H. (1972) *Statistical Inference Under Order Restrictions*. New York: Wiley.
- Bartholomew, D. J. (1959) A test of homogeneity for ordered alternatives. *Biometrika*, **46**, 36-48.
- Bloch, D. A. and Silverman, B W. (1997) Monotone discriminant functions and their applications in rheumatology, *Journal of the American Statistical Association*, **92**, 144-153.
- Bookstein, F. L. (1991) *Morphometric Tools for Landmark Data: Geometry and Biology*. Cambridge: Cambridge University Press.
- Bowman, A. W., Jones, M. C. and Gijbels, I. (1997) Testing monotonicity of regression, *Journal of Computational and Graphical Statistics*, to appear.
- Box, G. E. P. and Cox, D. R. (1964) An analysis of transformations, *Journal of the Royal Statistical Society, Series B*, **26**, 211-252.
- Chambers, J. M. and Hastie, T. J. (1996) *Statistical Models in S*. New York: Chapman and Hall.
- Friedman, J. and Tibshirani, R. (1984) The monotone smoothing of scatterplots. *Technometrics*, **26**, 243-250.
- Hanselman, D. and Littlefield, B. (2001) *Mastering MATLAB 6: A Comprehensive Tutorial and Reference*. Upper Saddle River: Prentice Hall.
- Heckman, N. and Ramsay, J. O (1997) Penalized regression with model-based penalties. University of British Columbia: unpublished manuscript.
- Kelly, C. and Rice, J. R. (1990) Monotone smoothing with application to dose response curves and the assessment of synergism. *Biometrics*, **46**, 1071-1085.
- Kneip, A. and Gasser, T. (1988) Convergence and consistency results for self-modeling nonlinear regression. *Annals of Statistics*, **16**, 82-112.

- Kneip, A. and Gasser, T. (1992) Statistical tools to analyze data representing a sample of curves. *Annals of Statistics*, **20**, 1266-1305.
- Kruskal, J. B. (1965) Analysis of factorial experiments by estimating monotone transformations of the data. *Journal of the Royal Statistical Society, Series B*, **27**, 251-263.
- Mammen, E. (1991) Estimating a smooth monotone regression function. *Annals of Statistics*, **19**, 724-740.
- Ramsay, J. O. (1988) Monotone regression splines in action (with discussion). *Statistical Science*, **3**, 425-461.
- Ramsay, J. O. (1998) Estimating smooth monotone functions, *Journal of the Royal Statistical Society, Series B*, **60**, 365-375.
- Ramsay, J. O., Bock, R. D. and Gasser, T. (1995) Comparison of height acceleration curves in the Fels, Zurich, and Berkeley growth data. *Annals of Human Biology*, **22**, 413-426.
- Ramsay, J. O. and Li, X. (1998) Curve registration. *Journal of the Royal Statistical Society, Series B*, **60**, 351-363.
- Ramsay, J. O. and Silverman, B. W. (1997) *Functional Data Analysis*. New York: Springer.
- Ramsay, J. O. and Silverman, B. W. (2002) *Applied Functional Data Analysis*. New York: Springer.
- Ramsay, J. O., Wang, X. and Flanagan, R. (1995) A functional data analysis of the pinch force of human fingers. *Applied Statistics*, **44**, 17-30.
- Roche, A. (1992) *Growth, Maturation and Body Composition: The Fels Longitudinal Study 1929-1991*. Cambridge: Cambridge Press.
- Sakoe, H. and Chiba, S. (1978) Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions, ASSP-26*, **1**, 43-49.

- Silverman, B. W. (1995) Incorporating parametric effects into functional principal components analysis. *Journal of the Royal Statistical Society, Series B*, **57**, 673–689.
- Statistical Sciences (1995) *S-PLUS Guide to Statistical and Mathematical Analysis, Version 3.3*. Seattle: StatSci, a division of MathSoft, Inc.
- Thalange, N. K. S., Foster, P. J., Gill, M. S., Price, D. A., and Clayton, P. E. (1996) Model of normal prepubertal growth. *Archives of Disease in Childhood*, **75**, 1-5.
- Wang, K. and Gasser, T. (1998) Alignment of curves by dynamic time warping. *Annals of Statistics*, to appear.
- Wright, I. and Wegman, E. (1980) Isotonic, convex, and related splines. *Annals of Statistics*, **8**, 1023-1035.