Institute of Mathematics, University of Copenhagen

Project in Statistics

Misaligned Curve Data: Constructing a Benchmark for Curve Registration and Investigation of Chosen Estimation Procedures

Student: Helene Charlotte Rytgaard

Supervisor: Lars Lau Rakêt

Hand-in Date: Friday, June 19, 2015

Abstract

In this project, we consider misaligned functional data, where the objective is to study sources of patterns of variation among the data. We will identify different types of interesting effects involved in the generation of the underlying data, simulate under the corresponding models and consider different estimation procedures for alignment.

Contents

1 Introduction

In functional data analysis we observe pairs of observations (t_{i_j}, y_{i_j}), $j = 1, ..., m$, $i = 1, ..., n$, where the individual samples $y_i = (y_{i_1}, ..., y_{i_m}) \in \mathbb{R}^m$, for each $i \in \{1, ..., n\}$, are considered a vector of observations from a function, θ, evaluated at the points $t_{i_1}, ..., t_{i_m} \in [0, 1]$.[1] The objective, when analysing the functional data, will be to identify this common underlying function.

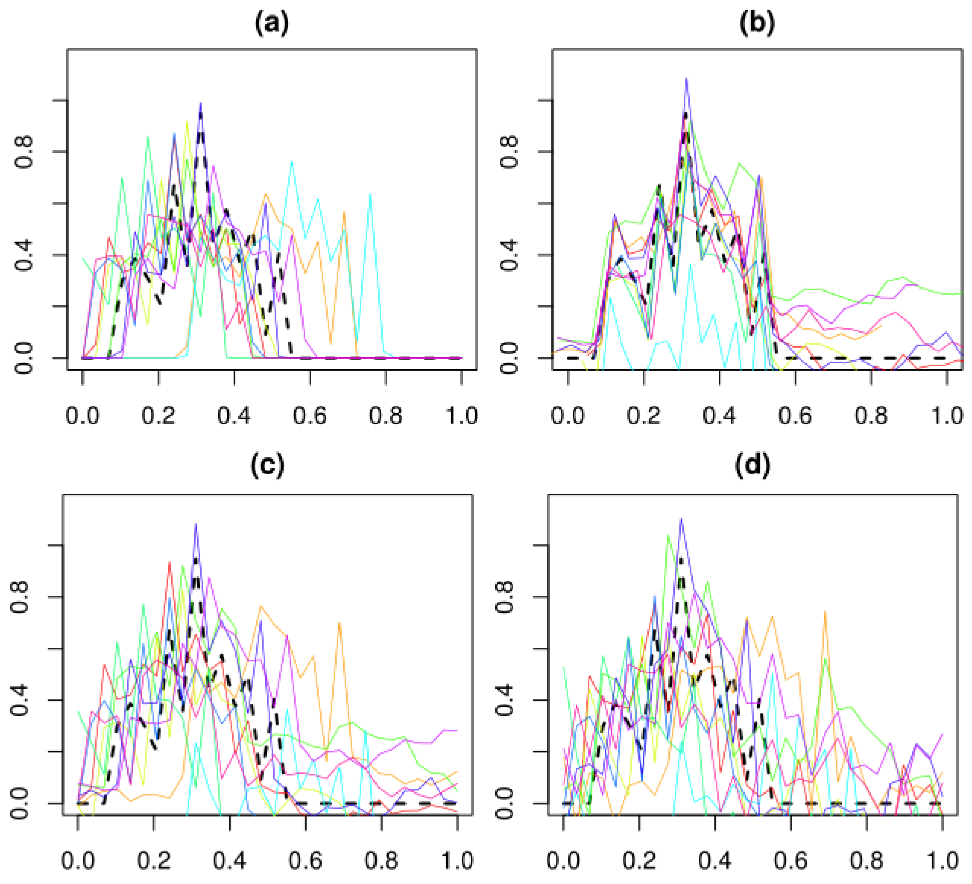

Figure 1: Different types of variation that we could imagine to be found in a functional dataset. The true underlying curve is shown in dashed black. The plot **(a)** shows warped data corresponding to horizontal variation. The plot **(b)** shows the same underlying data, unwarped, but with vertical variation due to individual sample variation. The plot **(c)** shows the two effects together, and in the plot **(d)** random noise is furthermore added.

Figure 1 shows an example of the variations we may encounter when dealing with a real functional data set. The challenge we face is to separate the different effects, hence modelling the individual sample effects and aligning the data properly. Formalizing these variations, one could, initially, based on **(a)** in Figure 1, consider the model,

$$y_i(t_j) = \theta(v(t_j, \boldsymbol{w}_i)), \tag{a}$$

where $v : [0, 1] \times \mathbb{R}^{n_w} \to [0, 1]$ is called the warping function. It could for instance be shifts or linear tranformations of the time axis. This warping function also depends on some $\boldsymbol{w}_i = (w_{i_1}, \ldots, w_{i_{n_w}})$ that is a vector of Gaussian parameters, such that each individual sample in effect has a different, random, warping function.

[1] In this project we constraint ourselves to $t \in [0, 1]$, but handling real data with other t would simply be done by a rescaling of the time measurements.

Handling the possibility of vertical individual variations across the vectors y_i, $i = 1, ..., n$, corresponding to **(b)** in Figure 1, the data could be assumed to be generated from the model,

$$y_i(t_j) = \theta(t_j) + x_i(t_j), \tag{b}$$

where the x_i is some vertical random translation. Combining these two models and their effects, we obtain the model corresponding to **(c)** in Figure 1,

$$y_i(t_j) = \theta(v(t_j, w_i)) + x_i(t_j). \tag{c}$$

Furthermore, adding independent and identically distributed random noise, ϵ_{ij}, we arrive at the model (corresponding to **(d)** in Figure 1),

$$y_i(t_j) = \theta(v(t_j, w_i)) + x_i(t_j) + \epsilon_{ij}, \tag{1.1}$$

for $i \in \{1, ...n\}$, $j \in \{1, ...m\}$. This is the model we will use in this project when dealing with misaligned functional data sets. Summarizing, we have:

- θ: The underlying deterministic mean curve to be estimated.
- v: The warping function. We will in general assume $t \mapsto v$ to be a homeomorphism, a monotone mapping from $[0, 1]$, with $v(0) = 0$ and $v(1) = 1$, which in average is the identity $t = t$ (id).
- w_i: A vector of Gaussian parameters on which the warping function depends.
- x_i: A random vertical translation. In the following we will for instance let this be modelled as a zero-mean Gaussian process, exhibiting some kind of covariance structure.
- ϵ_{ij}: Independent and identically distributed noise, e.g. each a realization of $\mathcal{N}(0, \sigma^2)$, for some variance $\sigma^2 > 0$.

In this project, we will identify and go through fundamentally different classes within each of the effects above, and on the basis hereof we will automatize the process of generating data that imitates some of the same variations that one could imagine finding in real data. The objective will be to look into different methods of handling non-aligned data and to see how these methods actually do compared to each other. The methods considered will mainly be a self-constructed estimation procedure following the lines of [1] and the `R`-function `align_fPCA` from the package `fdasrvf`.

2 Generating Data

2.1 Mean Curves θ

One could imagine studying very different kinds of underlying curves when doing functional data analysis. Real study examples could be that of the hight velocity curves for boys in the age between 0 and 18 years (with a downward sloping curve with several peaks), it could be the mean monthly temperatures for different weather stations (a one-peak curve), which are examples found in [2], or it could be the modelling of some sort of periodic phenomena (a variety of the sine curve), etc.

In `R` we define different types of mean curves, θ, to account for some different underlying mechanisms that we might be interested in modelling. All the curves are simulated from different random distributions.

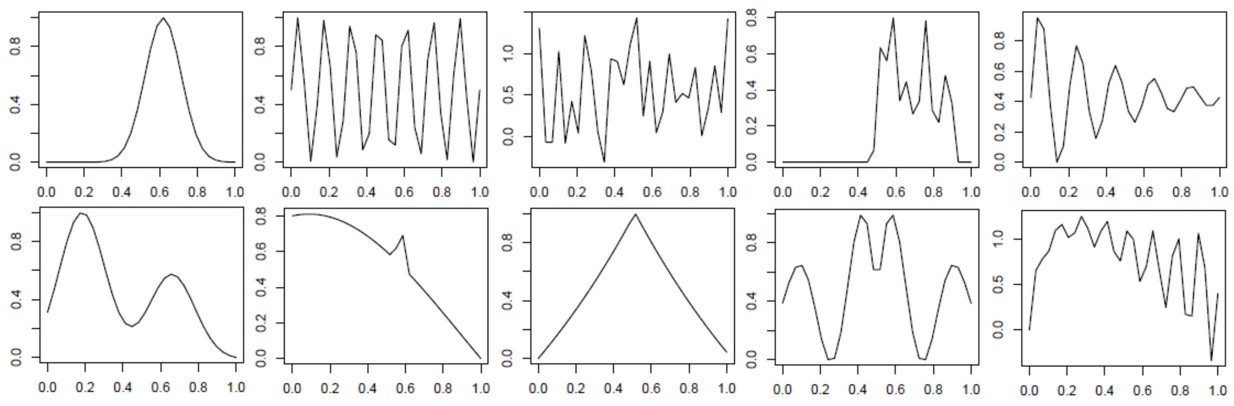

Figure 2: Simulation of different types of mean curves.

2.2 Generating Sample Points

Our data are observations of functions at certain time points. We could image these time points in different ways: They could be equidistant points in $[0, 1]$, or they could be completely randomly chosen (ordered) points in $[0, 1]$. We could also include the possibility of different times of measurement t for the different records, but for now we will not go into that.[2]

2.3 Generating Vertical Variation x_i

We will model the random vertical variation in data x_i using two different types of zero mean Gaussian processes, namely the Gaussian process with Matérn covariance, and the Brownian motion. Where the Brownian motion possesses the characteristics of being nowhere differentiable and having independent increments, we can change the parameters of the Matérn covariance process to achieve different degrees of correlations and smoothness. With these two processes we have a wide spectrum of different possibilities of x_i.

Thus, we assume that $X_i \sim \mathcal{N}(0, \sigma^2)$. Generating the Gaussian processes is achieved by first simulating m iid standard Gaussian variables, $X = (X_1, ..., X_m) = (X(t_1), ..., X(t_m))$, and then realizing that $\Sigma^{\frac{1}{2}} X \sim \mathcal{N}(0_m, \Sigma)$.

Recall now that we define the Brownian motion with zero mean and variance $\sigma^2 > 0$ by the following properties,

1. $W_0 = 0$ P-a.s.
2. For all $t_j \le t_k$, $W(t_k) - W(t_j) \sim \mathcal{N}\left(0, (t_k - t_j)\sigma^2\right)$.
3. The increments for non-overlapping time intervals are independent.

So, if we let

$$\Sigma = \sigma^2 \begin{bmatrix} t_1 & t_1 & t_1 & \dots & t_1 \\ t_1 & t_2 & t_2 & \dots & t_2 \\ t_1 & t_2 & t_3 & \dots & t_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_1 & t_2 & t_3 & \dots & t_m \end{bmatrix},$$

[2]Also, the function align_fPCA can actually not handle this.

then $\Sigma^{\frac{1}{2}}X$ has exactly a finite-dimensional distribution of the Brownian motion with zero mean vector and variance σ^2. This is because, for $t_j \le t_k$,

$$\begin{aligned}\mathrm{Cov}(W(t_j), W(t_k)) &= \mathrm{Cov}\left(W(t_j), W(t_j) + (W(t_k) - W(t_j))\right)\\ &= \mathbb{V}(W(t_j)) + \mathrm{Cov}\left(W(t_j), W(t_k) - W(t_j)\right)\\ &= \mathbb{V}(W(t_j)) = (t_j - 0)\,\sigma^2 = t_j\sigma^2,\end{aligned}$$

obtained simply by using the properties of the Brownian motion.

Similarly, we may generate the Matérn covariance process — here we use the `R` function `Matern` from the package `fields`. The `R`-code for generating the two types of correlation functions are thus as follows, firstly for the Brownian motion:

```
for (i in 1:m) {
  for (j in 1:m) {
    Sigma[i, j] = min(t[i], t[j])
  }
}
```

And then for the Matérn covariance:

```
for (i in 1:m) {
  for (j in 1:m) {
    Matern(abs(t[i] - t[j]), scale = 1, range = r, smoothness = s)
  }
}
```

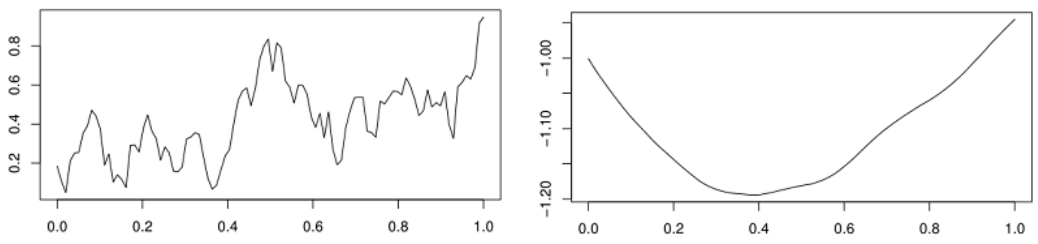

Figure 3: The left curve is an example of the Brownian motian, and the right curve is an example of a process with Matérn covariance.

2.4 Different types of warping functions v.

As a warping function we could consider the most simple one being the shift function on $\mathbb{R}$, following from just adding a (random) real number to the actual time points. Likewise, we could also take into account the possibility of an underlying individual multiplicative effect, which corresponds to multiplying the actual time points with a (random) real number. Generalizing these two, the warping function could be a linear transformation. Note, though, that these examples of warping functions do not fulfil, as functions of t, $v(0) = 0$, $v(1) = 1$.

Adding more generality, we consider piecewise linear transformations or smooth non-linear transformations as warping functions. Thus, we randomly generate intermediate points and then we do linear interpolation (using the `R`-function `approxfun`) or compute a monotone cubic spline (using the `hyman` argument in the `R`-function `splinefun`), respectively, of the points. See Figure 4 for an example that corresponds to a piecewise linear warp and a piecewise smooth warp, both randomly chosen symmetric around $t = t$, with the number of intermediate points chosen randomly from $\{2, 3, ..., 10\}$.

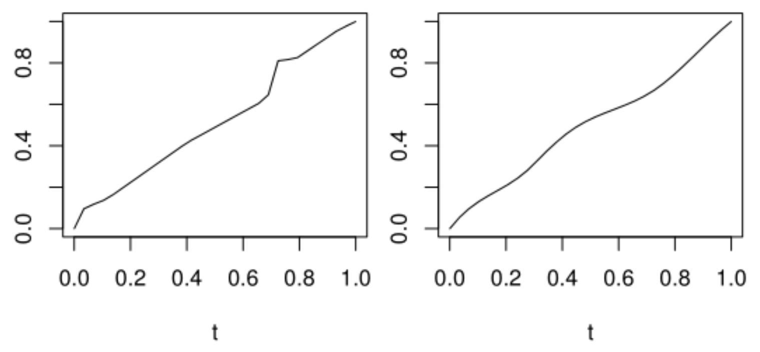

Figure 4: The left curve is an example of a piecewise linear warping effect, and the right curve is an example of a piecewise smooth warping effect. Both curves have a random number of intermediate points, and the intermediate points are also randomly chosen from $[0, 1]$.

2.5 Generating Random Noise

We want to incorporate three different situations of noise: (1) The situation without any noise at all, (2) the classical noise, being iid variables from the standard Gaussian distribution, or (3) noise drawn from the Laplace distribution with an appropriate scaling parameter. Especially (3) is interesting, as a model accommodating Laplace noise is a model reasonably robust to outliers. The code below defines a function that generates noise when choosing from one of these types.

```
generate_noise = function(type = 1) {
  if (type == 1) {
      noise = function(m) {
        rnorm(m, sd = 0.1)
      }
  } else if (type == 2) {
      noise = function(m) {
        rlaplace(m, sigma = 0.1)
      }
  } else if (type == 3) {
      noise = function(m) {
        c(rep(0,m))
      }
  }
return(noise)
}
```

The `laplace` simulation function is made from a transformation of a uniformly distributed variable on $(-\frac{1}{2}, \frac{1}{2})$. We take $U \sim 1_{(-\frac{1}{2},\frac{1}{2})}$ and $h : (-\frac{1}{2}, \frac{1}{2}) \to \mathbb{R}$, $h(u) = \mu - \operatorname{sign}(u)\sigma \log(1 - 2|u|)$, $\sigma > 0$ and $\mu \in \mathbb{R}$, and find the inverse mapping $h^{-1}(y)$,

$$\mu - y = \operatorname{sign}(u)\sigma \log(1 - 2|u|) = \begin{cases} \sigma \log(1 - 2u), & \text{if } u \geq 0, \\ -\sigma \log(1 + 2u), & \text{if } u < 0, \end{cases}$$

note here that $\mu - y \leq 0 \Leftrightarrow u \geq 0$,[3] and thus

$$h^{-1}(y) = \begin{cases} \frac{1}{2}\left(1 - \exp\left(\frac{\mu - y}{\sigma}\right)\right), & \text{if } \mu \leq y, \\ -\frac{1}{2}\left(1 - \exp\left(\frac{y - \mu}{\sigma}\right)\right), & \text{if } \mu > y. \end{cases}$$

[3]Since $u \in (-\frac{1}{2}, \frac{1}{2})$ implies that $1 - 2|u| \leq 1$, i.e. $\log(1 - 2|u|) \leq 0$.

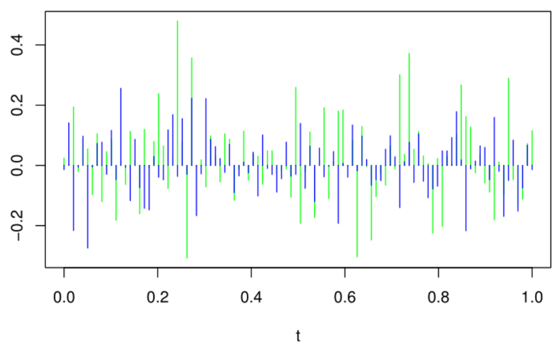

Figure 5: The green lines correspond to Laplacian noise and the blue lines correspond to Gaussian noise, both with the value $\sigma = 1$. It is visible that the Laplacian noise may result in more outlying observations.

Now,

$$\begin{aligned}\left(h^{-1}\right)'(y) &= \begin{cases} \frac{1}{2}\exp\left(\frac{\mu-y}{\sigma}\right), & \text{if } y \geq \mu, \\ \frac{1}{2}\exp\left(\frac{y-\mu}{\sigma}\right), & \text{if } y < \mu. \end{cases} \\ &= \frac{1}{2}\exp\left(-\frac{|y-\mu|}{\sigma}\right).\end{aligned}$$

Thus, $Y = h(U)$ will have the density,

$$f(y) = 1_{(-\frac{1}{2},\frac{1}{2})}\left(h^{-1}(y)\right)\left|\left(h^{-1}\right)'(y)\right| = \frac{1}{2}\exp\left(-\frac{|y-\mu|}{\sigma}\right), \quad y \in \mathbb{R},$$

which we recognize as the Laplace density. We implement this in R with the following code,

```
rlaplace = function(n, mu = 0, sigma = 1){
  U    = runif(n, min = -1/2, max = 1/2)
  mu - sign(U)*sigma*log(1-2*abs(U))
}
```

3 Automatizing the Process of Data Generation

3.1 Creating a Function to Generate Data

We implement in R the following function,

```
generate_data = function(theta_t = 1, t_t = 1, warp_t = 1,
                         x_t = 1, noise_t = 1, plot = TRUE) {
  y      = array(NA, dim = c(m,n))
  w_true = array(NA, dim = c(m,n))
  theta  = generate_theta(type = theta_t)
  t      = generate_t(type=t_t)(m)
  if (plot)  plot(t, theta(t), type='l')
    for (i in 1:n) {
    w_true[, i] = generate_warp(type=warp_t)(t)
    x           = generate_x(type=x_t)
    noise       = generate_noise(type=noise_t)(m)
    y[, i]      = theta(w_true[, i]) + x + noise
    if (plot) lines(t, y[, i], lwd = 0.45, col = rainbow(n)[i])
  }
  attr(y, 'theta')  = theta
  attr(y, 't')      = t
  attr(y, 'w_true') = w_true
  attr(y, 'x')      = x
  attr(y, 'noise')  = noise
  return(y)
}
```

3.2 Examples

We generate different examples of underlying data using the function above, choosing different effects. Just as we did in Figure 1, we add the effects one at a time. The result is seen in Figure 6.

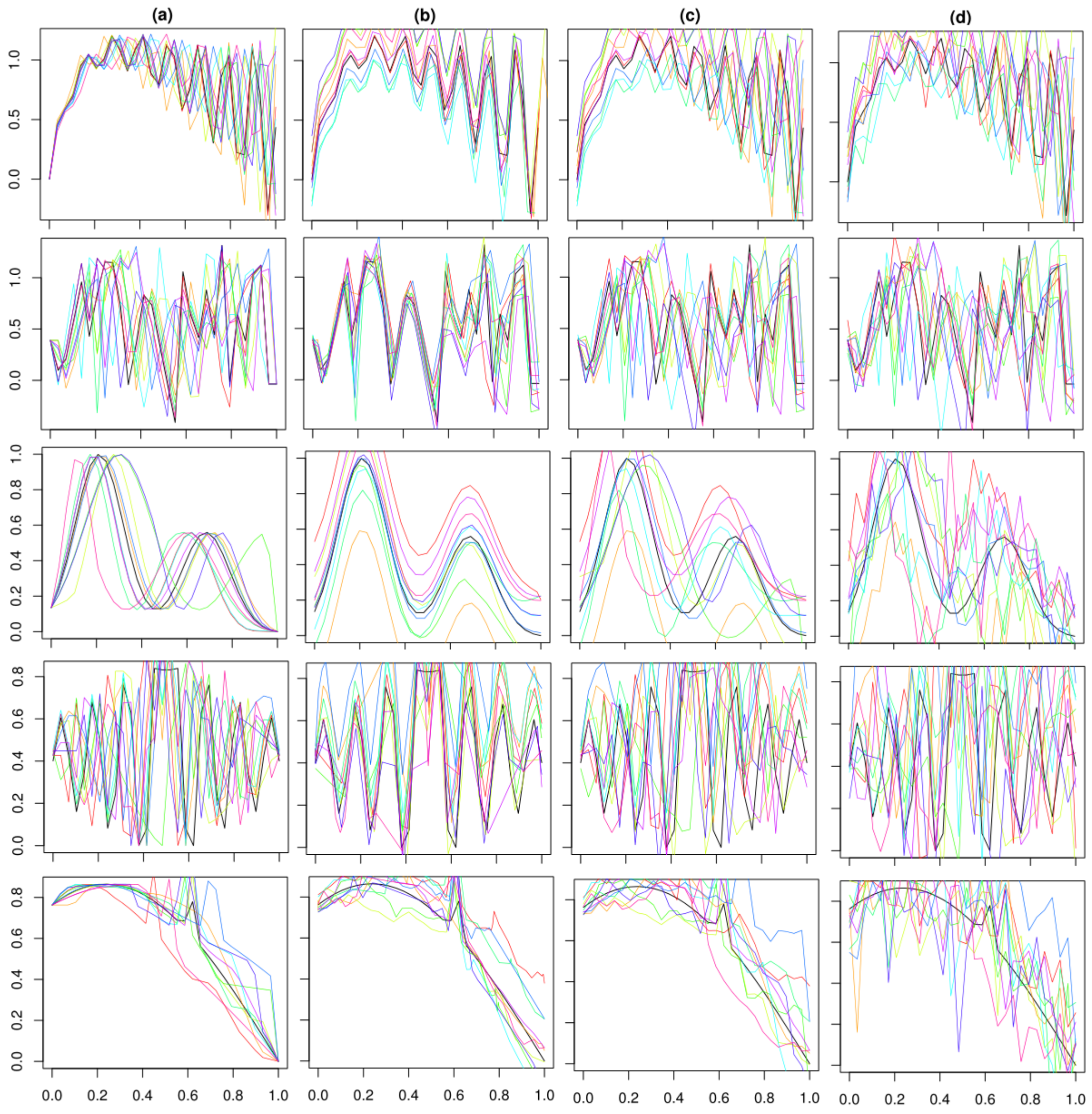

Figure 6: Each row corresponds to each its example of an underlying data set. Just as was done in Figure 1, we add one effect at the time, using different effects. Thus, for each row, the plot **(a)** shows warped data with no noise, plot **(b)** shows the same underlying data, unwarped, but with the addition of some Gaussian process, **(c)** shows the effects from plots (a) and (b) together, and the plot **(d)** is (c) with random noise on top of all the effects.

4 Methods for Aligning Data

As mentioned in the introduction, we consider two different approaches for aligning the data. We will go through both methods to outline each their approach to handling the non-aligned data.

4.1 Estimation Procedure Based on the Fisher-Rao metric

We consider the following variety of the model (1.1),

$$y_i(t_j) = f_i(t_j) = c_i \cdot \theta\left(v_i(t_j)\right) + e_i, \tag{4.1}$$

where $v_i : [0,1] \to [0,1]$ are diffeomorphisms with $v_i(0) = 0$ and $v_i(1) = 1$, and $c_i, e_i \in \mathbb{R}$ are scalings and translations. The following subsections provide a rather informal sketch of the estimation procedure based on this model, following [3]. The point is to get a feeling of the underlying method and assumptions so that we can understand how it compares with our own estimation procedure (to be introduced in Section 4.2). What is essential here is that this method constitutes a very simple mathematical framework for aligning the curves — assuming a nice structure of the underlying data, with no substantial noise.

4.1.1 SRVF, Fisher-Rao Riemannian Metric and the Elastic Distance

Given an absolutely continuous function f on $[0,1]$, we introduce the so-called square-root velocity function (SRVF), $q : [0,1] \to \mathbb{R}$, which is defined by

$$q(t) \equiv \frac{f'(t)}{\sqrt{|f'(t)|}} \cdot 1_{\{|f'(t)| \neq 0\}}.$$

We can think of the space $\mathbb{L}^2([0,1],\mathbb{R})$ to be the set of all SRVFs. For every $q \in \mathbb{L}^2$ there exists a function f such that q is the SRVF of that f. Now, warping the function by v, the SRVF of $f \circ v$ is $(q \circ v)(t)\sqrt{v'(t)} \equiv (q, v)$.

Definition 4.1. *With $\mathcal{F}$ the set of absolutely continuous functions on $[0,1]$ and $T_f(\mathcal{F})$ denoting the tangent space of $\mathcal{F}$ at f, the* ***Fisher-Rao Riemannian metric*** *is defined as the inner product,*

$$\langle\langle v_1, v_2 \rangle\rangle_f = \frac{1}{4}\int_0^1 v_1'(t) v_2'(t) \frac{1}{|f'(t)|} \mathrm{d}t.$$

Lemma 4.2. *Under the SRVF representation, the Fisher-Rao Riemannian metric is the standard $\mathbb{L}^2$-metric.*

The proof of Lemma 4.2 is a matter of simple computations. The point now is that we can then find the distance between any two functions by computing the $\mathbb{L}^2$-distance between the corresponding SRVFs, i.e. $d_{FR}(f_1, f_2) = ||q_1 - q_2||$. A particularly nice property about this distance is that it is invariant to transformation under the same warp, $||(q_1, v) - (q_2, v)|| = ||q_1 - q_2||$. Showing this is even simpler than showing Lemma 4.2, as we have,

$$\begin{aligned} ||(q_1, v) - (q_2, v)||^2 &= \int_0^1 \left(q_1(v(t))\sqrt{v'(t)} - q_2(v(t))\sqrt{v'(t)}\right)^2 \mathrm{d}t \\ &= \int_0^1 \left(q_1(v(t)) - q_2(v(t))\right)^2 v'(t)\mathrm{d}t \\ &= \int_0^1 \left(q_1(u) - q_2(u)\right)^2 \frac{\mathrm{d}u}{\mathrm{d}t}\mathrm{d}t \\ &= ||q_1 - q_2||. \end{aligned}$$

The orbit of an SRVF $q \in \mathbb{L}^2$ is given by $[q] = \text{closure}\{(q,v)\}$, i.e. it is the set of the SRVFs associated with all the warpings of a particular function, including their limit points. We let S be the set of all such orbits, and then we introduce a distance here:

Definition 4.3. *For any $f_1, f_2 \in \mathcal{F}$ and corresponding $q_1, q_2 \in \mathbb{L}^2$, we define the **elastic distance** d on S as,*

$$d([q_1],[q_2]) = \inf_v ||q_1 - (q_2, v)||.$$

So the elastic distance is the smallest $\mathbb{L}^2$-distance between q_1 and the SRVF of $f_2 \circ v$, (q_2, v). Note that the elastic distance between a function f_1 and its warped version will be 0. Note also that this distance is not a proper distance on $\mathbb{L}^2$.

4.1.2 The Karcher Mean

Considering a warping function v, we will represent this using its SRVF, $\psi(t) = \sqrt{v'(t)}$.[4] From the properties of v, the mapping taking us from v to ψ will be bijective, such that we can reconstruct v from ψ by $v(t) = \int_0^t \psi(s)^2 \mathrm{d}s$. Now, considering the elements ψ, we note that

$$||\psi||^2 \overset{\mathbb{L}^2\text{-norm}}{=} \int_0^1 \psi(t)^2 \mathrm{d}t = \int_0^1 v'(t)\mathrm{d}t = v(1) - v(0) = 1 - 0 = 1,$$

such that the ψs will be in the unit sphere in $\mathbb{L}^2$. Now, from this, the Fisher-Rao distance between any two warpings is found as the spherical distance between the corresponding ψs, i.e.

$$d_{FR}(v_1, v_2) = \cos^{-1}\left(\langle\langle \psi_1, \psi_2 \rangle\rangle\right) = \cos^{-1}\left(\int_0^1 \psi_1(t)\psi_2(t)\mathrm{d}t\right),$$

hence, using Lemma 4.2,

$$d_{FR}(v_1, v_2) = \cos^{-1}\left(\int_0^1 \sqrt{v_1'(t)}\sqrt{v_2'(t)}\mathrm{d}t\right). \tag{4.2}$$

Thus, we now have a proper distance on the considered set of warping functions. This will prove useful later.[5]

Definition 4.4. *For a given set of warping functions, $v_1, ..., v_n$, we define the **Karcher mean** to be,*

$$\overline{v}_n = \underset{v}{\text{argmin}} \sum_{i=1}^n d_{FR}(v, v_i)^2.$$

Definition 4.5. *We define the **Karcher mean** $[\mu]_n$ of the $[q_i]$s in the space S as*

$$[\mu]_n = \underset{[q]\in S}{\text{argmin}} \sum_{i=1}^n d([q],[q_i])^2$$

We next want to find a particular element of this $[\mu]_n$.

Definition 4.6. *For a given set of SRVFs $q_1, ..., q_n$ and q we define an element $\tilde{q}$ of $[q]$ as **the center of** $[q]$ **with respect to** $q_1, ..., q_n$ if the warping functions $v_i = \text{argmin}_v ||\tilde{q} - (q_i, v)||$ have the Karcher mean v_{id}.*

[4]Note the simler form, as we have $v' > 0$ (the warping is assumed monotonically increasing).
[5]Later we will also look into this distance by applying it to a few simple examples.

4.1.3 The Estimation Procedure

The estimation procedure now builds on the following steps:

1. For the observed collection of f_i, $i = 1, ..., n$ and their corresponding SRVFs, q_i, the Karcher mean of the $[q_i]$s in S is computed. Denote this by $[\mu]_n$.

2. The center of $[\mu]_n$ with respect to $\{q_i\}$ is found. This is done by for each q_i solving $v_i = \text{argmin}_v \, ||\mu - (q_i, v)||$, and then computing the mean $\bar{v}_n$ of $\{v_i\}$. Then the center of $[\mu]_n$ is given by $\tilde{q} = (q, \bar{v}_n^{-1})$.

3. The warping functions are then estimated by aligning the individual functions to match this $\tilde{q}$. That is, by solving $v_i^* = \text{argmin}_v \, ||\tilde{q} - (q_i, v)||$.

4. These estimated warping functions are then used to align the observed $f_1, ..., f_n$, i.e. $\tilde{f}_i = f_i \circ v_i^*$.

Considering the model as described, we have $y_i(t_j) = f_i(t_j) = c_i \cdot \theta\,(v_i(t_j)) + e_i$. Then the derivatives are given by $f_i'(t_j) = c_i \theta'\,(v_i(t_j))\, v_i'(t_j)$, and the SRVFs q_i, (assuming c_i somewhere around 1, thus $c_i > 0$), $q_i(t_j) = \sqrt{c_i}\sqrt{v_i'(t_j)}\frac{\theta'(v_i(t_j))}{\sqrt{|\theta'(v_i(t_j))|}} = \sqrt{c_i}(q_\theta, v_i)(t_j)$, where q_θ is the SRVF of θ. So the e_is do not effect the estimation procedure. Furthermore, according to step 1 we should find the Karcher mean,

$$\begin{aligned}
[\mu]_n &= \underset{[q]}{\text{argmin}} \sum_{i=1}^{n} d\big([q_i], [q]\big)^2 = \underset{[q]}{\text{argmin}} \sum_{i=1}^{n} d\big([\sqrt{c_i}(q_\theta, v_i)], [q]\big)^2 \\
&= \underset{[q]}{\text{argmin}} \sum_{i=1}^{n} \inf_{v} \big|\big|\sqrt{c_i}(q_\theta, v_i) - (q, v)\big|\big|^2 = \underset{[q]}{\text{argmin}} \sum_{i=1}^{n} \inf_{v} \big|\big|\sqrt{c_i} q_\theta - (q, v \circ v_i^{-1})\big|\big|^2 \\
&= \underset{[q]}{\text{argmin}} \sum_{i=1}^{n} \inf_{v^{(i)}} \big|\big|\sqrt{c_i} q_\theta - (q, v^{(i)})\big|\big|^2 \left(= \underset{[q]}{\text{argmin}} \sum_{i=1}^{n} d\big([\sqrt{c_i} q_\theta], [q]\big)^2\right)
\end{aligned}$$

Now, look at

$$\begin{aligned}
\underset{v^{(i)}}{\text{argmin}} \big|\big|\sqrt{c_i} q_\theta - (q, v^{(i)})\big|\big|^2 &= \underset{v^{(i)}}{\text{argmin}} \int_0^1 \Big(\sqrt{c_i} q_\theta(t) - (q, v^{(i)})(t)\Big)^2 \mathrm{d}t, \quad \Big(\textit{now use } ||(q, v^{(i)})|| = ||q||\Big), \\
&= \underset{v^{(i)}}{\text{argmin}} \left(c_i ||q_\theta||^2 + ||q||^2 - 2\sqrt{c_i} \int_0^1 q_\theta(t)(q, v^{(i)})(t)\mathrm{d}t\right) \\
&= \underset{v^{(i)}}{\text{argmin}} \left(-\int_0^1 q_\theta(t)(q, v^{(i)})(t)\mathrm{d}t\right), \quad \Big(\textit{use same arguments reversed}\Big), \\
&= \underset{v^{(i)}}{\text{argmin}} ||q_\theta - (q, v^{(i)})||^2.
\end{aligned}$$

Now, for any q, we let $v_1^*, \dots, v_n^*$ be warps that solves the optimization $\inf_{v^{(i)}} ||\sqrt{c_i} q_\theta - (q, v^{(i)})||$, then they also solve $\inf_{v^{(i)}} ||q_\theta - (q, v^{(i)})|| = \inf_v ||q_\theta - (q, v)||$. They are thus independent of i, $v_i^* = v^*$, $i = 1, \dots, n$. Then, continuing from above,

$$\sum_{i=1}^{n} \inf_{v^{(i)}} ||\sqrt{c_i} q_\theta - (q, v^{(i)})||^2 = \sum_{i=1}^{n} ||\sqrt{c_i} q_\theta - (q, v^*)||^2,$$

If we let $\bar{c} = \frac{1}{n}\sum_{i=1}^{n}\sqrt{c_i}$, then the mean of $(\sqrt{c_1}q_\theta, \dots, \sqrt{c_1}q_\theta)$ (in $\mathbb{L}^2$) is $\bar{c}q_\theta$, so we have the lower bound,

$$\sum_{i=1}^{n} ||\sqrt{c_i}q_\theta - (q, v^*)||^2 \geq \sum_{i=1}^{n} ||\sqrt{c_i}q_\theta - \bar{c}q_\theta||^2.$$

Thus,

$$\underset{[q]}{\text{argmin}} \sum_{i=1}^{n} d\big([q_i],[q]\big)^2 = \underset{[q]}{\text{argmin}} \sum_{i=1}^{n} ||\sqrt{c_i}q_\theta - (q, v^*)||^2 = [\bar{c}q_\theta] = \bar{c}[q_\theta].$$

Note that not only $\bar{c}q_\theta$ but any element in $[\bar{c}q_\theta]$ will be a solution, since all the arguments above can be repeated with any warped version of q_θ, (q_θ, v). Thus, these calculations indicate that we actually estimate θ exact up to a scaling and translation.

4.2 Self-constructed Estimation Procedure

As mentioned in the introduction, we will construct our own estimation procedure, following the lines of [1], based on the model (1.1),

$$y_i(t_j) = \theta(v(t_j, w_i)) + x_i(t_j) + \epsilon_{ij}. \tag{4.3}$$

The function θ is a fixed effect, $w_i \in \mathbb{R}^{n_w}$ is a vector of Gaussian parameters with covariance function $\sigma^2\lambda^2 C_0$, x_i is the outcome of a zero-mean Gaussian process with covariance function $\sigma^2 S_0$ and ϵ_{ij} is independent and identically distributed random noise with variance σ^2. Opposed to model (4.1), where the data is considered somewhat free of observation noise, this model tries to accommodate the possible random effects and builds on iteratively interchanging between estimating the parameters of the random parts and underlying mean curve and the warps in order to align the data in the best way possible.

4.2.1 Likelihood Estimation with Brownian Bridge (LEBB)

We can linearise model (4.3) with a first order Taylor approximation around w_i^0,

$$\theta(v(t_j, \boldsymbol{w}_i)) \approx \theta(v(t_j, \boldsymbol{w}_i^0)) + \partial_t\theta(v(t_j, \boldsymbol{w}_i^0)) \bigtriangledown_{\boldsymbol{w}} v(t_j, \boldsymbol{w}_i^0)(\boldsymbol{w}_i - \boldsymbol{w}_i^0),$$

which allows us to write our linearised model on vectorized form,

$$y \approx \theta^{\boldsymbol{w}^0} + Z(\boldsymbol{w} - \boldsymbol{w}^0) + \boldsymbol{x} + \boldsymbol{\epsilon}.$$

Simplifying this expression and the estimations we will ignore the contribution from $\boldsymbol{x}$, and the negative log likelihood will thus have the form,

$$l(\sigma^2, C, S) = nm\log\sigma^2 + \log\det V + \sigma^{-2}\big(\boldsymbol{y} - \hat{\theta}^{\boldsymbol{w}^0} + Z(\boldsymbol{w}^0 - \text{id})\big)^T V^{-1}\big(\boldsymbol{y} - \hat{\theta}^{\boldsymbol{w}^0} + Z(\boldsymbol{w}^0 - \text{id})\big). \tag{4.4}$$

In the above, we have

$$\begin{aligned}
\boldsymbol{y} &\approx \theta^{\boldsymbol{w}^0} + Z(\boldsymbol{w} - \boldsymbol{w}^0) + \boldsymbol{\epsilon}, \\
\theta^{\boldsymbol{w}^0} &= \{\theta(v(t_j, \boldsymbol{w}_i^0))\}_{i,j} \in \mathbb{R}^{mn}, \\
Z &= \text{diag}(Z_i)_{1\leq i\leq n}, \\
Z_i &= \{\partial_t\theta(v(t_j, \boldsymbol{w}_i^0)) \bigtriangledown_{\boldsymbol{w}} v(t_j, \boldsymbol{w}_i^0)\}_j \in \mathbb{R}^{m\times n_w},
\end{aligned}$$

$$\boldsymbol{w} = \{\boldsymbol{w}_i\}_i \sim \mathcal{N}_{nn_w}(0, \sigma^2 C),$$

id : the identity mapping $t \mapsto t$ on $[0,1]$.

$$C = \sigma^2\lambda^2 \mathrm{diag}(C_0).$$

$$V = \lambda^2 ZCZ^T + \mathbf{1}_{nm},$$

$$\boldsymbol{\epsilon} \sim \mathcal{N}_{nm}(0, \sigma^2 \mathbf{1}_{nm}).$$

For modelling the distribution of the warps we consider the Brownian bridge. I.e. if $W(t)$ is a standard Brownian motion process, then $B(t) = W(t) - tW(1)$ is a Brownian bridge for $t \in [0,1]$. Thus, the covariance function will be given by, for $t_j \le t_k$,

$$\begin{aligned}
\mathrm{Cov}(B(t_j), B(t_k)) &= \mathrm{Cov}\left(W(t_j) - t_j W(1), W(t_k) - t_k W(1)\right) \\
&= \mathrm{Cov}\left(W(t_j), W(t_k)\right) - \mathrm{Cov}\left(W(t_j), t_k W(1)\right) - \mathrm{Cov}\left(t_j W(1), W(t_k)\right) \\
&\quad + \mathrm{Cov}\left(t_j W(1), t_k W(1)\right) \\
&= \mathrm{Cov}\left(W(t_j), W(t_k)\right) - t_k \mathrm{Cov}\left(W(t_j), W(1)\right) - t_j \mathrm{Cov}\left(W(1), W(t_k)\right) \\
&\quad + t_j t_k \mathrm{Cov}\left(W(1), W(1)\right) \\
&= t_j - t_k t_j - t_j t_k + t_j t_k \\
&= t_j - t_j t_k
\end{aligned}$$

So, in `R` ($C_0 =$ `Sigma`),

```
Sigma = matrix(NA, wn, wn)
cov_fct = function(s, t) {
  ( min(s, t) - s*t )
}
for (i in 1:(wn)) {
  for (j in 1:(wn)) {
    Sigma[i, j] = cov_fct(t[i], t[j])
  }
}
```

In the rest of the project, we will refer to the self-constructed estimation procedure as LEBB.

4.2.2 Outlining the Procedure

In the beginning of the estimation we decide on how many parameters $\boldsymbol{w}$ we want to estimate the warping functions. Per default we set this to `wn=3`. Note that this estimation procedure consists of two parts: Estimating the variance parameter λ and finding the optimal warping functions.

The first actual step of the estimation procedure is then to estimate the mean curve θ at the initial positions t_j, to obtain the estimate $\hat{\theta}^{\boldsymbol{w}^0}$. We do this with a B-spline basis, given $\boldsymbol{y}$. The optimal number of knots will depend on the properties of the underlying data. We make a simple estimate on this: First we fit a spline using 25 knots, then we test how much this spline oscillates — if it changes direction more than a certain number of times, we let the chosen number of knots, `nknots`, be 15, otherwise 6.[6] Then we do the B-spline approximation with this number of knots to obtain $\hat{\theta}^{\boldsymbol{w}^0}$.

[6]This is solely based on experience with the data.

```
thetaest = function(x, wt = wt0) {
  kts        = seq(0, 1, length = nknots)
  bkts       = c(-0.2,1.2)
  basis      = bs(wt, knots = kts, intercept = TRUE, Boundary.knots = bkts)
  c          = ginv(t(basis) %*% basis) %*% t(basis) %*% as.numeric(y)
  basis1     = bs(x, knots = kts, intercept = TRUE, Boundary.knots = bkts)
  return(basis1 %*% c)
}
```

We then want to estimate λ, based on the likelihood (4.4), so we should calculate the relevant derivatives and gradients. With the estimated mean curve, we can find an approximate derivative of this measured for all $t_1, ..., t_m$. In the first step, we will simply let the warping parameters be the $n_w \times n$-matrix with each column being the equidistant points,

```
tn = rep(seq(0, 1, length = wn+2)[2:(wn+1)],m)
```

and with the warping function,

```
v = function(w, tn, t) {
  approx(c(0, tn, 1), c(0, w, 1), xout = t)$y
}
```

we can find the values of these warps for all n and all m, and collect this in the vector with nm elements,

```
w = as.numeric(apply(w_est, 2, function(w) v(w, tn, t)))
```

Then the derivative of the estimated mean curve can be found by:

```
ti     = w[(i*m+1):((i+1)*m)]
dtheta = ( thetaest(ti+10^(-5), wt = w) -
             thetaest(ti-10^(-5), wt = w) ) / (2*10^(-5))
```

Next, we want to find the gradient in the warping parameters, of the warping function. If we let $t_0^w, \dots, t_{n_w+1}^w$ (in R, `tw  = c(0,tn,1)`) be the anchor points of the warping functions, i.e. the points where the warping function is evaluated. Furthermore, we let $w_0, ..., w_{n_w+1}$ be the values of the warpings in the anchor points, $w_0 = 0$, $w_{n_w+1} = 1$. Then we have the following formula for the warping functions,[7]

$$\text{for } t \in [t_j^w, t_{j+1}^w]: \quad v(t, \boldsymbol{w}) = \frac{w_{j+1} - w_j}{t_{j+1}^w - t_j^w} t + \frac{t_j^w w_{j+1} - t_{j+1}^w w_j}{t_{j+1}^w - t_j^w}.$$

Thus, differentiating this with respect to $\boldsymbol{w}$, (the rest being 0),

$$\text{for } t_k \in [t_j^w, t_{j+1}^w]: \quad \frac{\partial}{\partial w_j} v(t_k, \boldsymbol{w}) = \frac{-1}{t_{j+1}^w - t_j^w} t_k + \frac{t_{j+1}^w}{t_{j+1}^w - t_j^w},$$

$$\text{for } t_k \in [t_j^w, t_{j+1}^w]: \quad \frac{\partial}{\partial w_{j+1}} v(t_k, \boldsymbol{w}) = \frac{1}{t_{j+1}^w - t_j^w} t_k - \frac{t_j^w}{t_{j+1}^w - t_j^w}.$$

and evaluating for all t_k, $k = 1, \dots, m$ yields the $m \times n_w$ matrix,

$$\begin{bmatrix} \frac{\partial}{\partial w_1} v(t_1, \boldsymbol{w}) & \frac{\partial}{\partial w_2} v(t_1, \boldsymbol{w}) & \dots & \frac{\partial}{\partial w_{n_w}} v(t_1, \boldsymbol{w}) \\ \frac{\partial}{\partial w_1} v(t_2, \boldsymbol{w}) & \frac{\partial}{\partial w_2} v(t_2, \boldsymbol{w}) & \dots & \frac{\partial}{\partial w_{n_w}} v(t_2, \boldsymbol{w}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial w_1} v(t_m, \boldsymbol{w}) & \frac{\partial}{\partial w_2} v(t_m, \boldsymbol{w}) & \dots & \frac{\partial}{\partial w_{n_w}} v(t_m, \boldsymbol{w}), \end{bmatrix},$$

[7]The slope is easily found as $\frac{w_{j+1} - w_j}{t_{j+1}^w - t_j^w}$, and then we just plug in $t = t_j^w$ and $v(t_j^w, \boldsymbol{w}) = w_j$ to find the intersection with the second axis.

such that each row k corresponds to evaluating the gradient at time t_k, $k = 1, \dots, m$. This matrix is then to be multiplied, row by row, with the derivative of θ, i.e. for each $t_1, \dots, t_m$ we multiply $\partial_t \theta(t_k, \boldsymbol{w}_i)$ with row k of the matrix above. This yields the $m \times n_w$ matrix Z_i specified earlier. In R,

```
for (i in 0:(n-1)) {
  ti     = w[(i*m+1):((i+1)*m)]
  dtheta = ( thetaest(ti+10^(-5), wt = w) -
             thetaest(ti-10^(-5), wt = w) ) / (2*10^(-5))
  for (j in 1:m) {
    Zi[j, ] = dtheta[j]*grad_w[j, ]
  }
  Z[(1+i*m):((i+1)*m),(1+i*(wn)):((i+1)*(wn))] = Zi
}
```

Then writing the likelihood with R follows easily,

```
#-- approximative derivative in t of mean function
thetafitted  = rep(0, m*n)
for (i in 0:(n-1)) {
  ti                              = w[(i*m+1):((i+1)*m)]
  thetafitted[(1+i*(m)):(m+i*(m))] = thetaest(ti, wt = w)
}

#-- setting up dummy vector
r = as.numeric(y) - thetafitted + Z %*% w0_vec

likewarppar = function(warp_par) {
  V        = warp_par^2 * ( Z %*% C %*% t(Z) ) + diag(1,n*m)
  inv_V    = ginv(V)
  sigmahat = 1/(m*n) * t(r) %*% inv_V %*% r
  return( (n*m)*log(sigmahat) +
    determinant(V)$modulus )
}
```

where we have used the estimate $\hat{\sigma}^2 = \frac{1}{nm} r^T V^{-1} r$ for σ^2. Then we use the `optimize` function in R to find the optimal λ,

```
warp_par = optimize(likewarppar, lower = 0, upper = 100)$minimum
```

Now we are ready to define the negative log posterior to estimate the warping parameters, i.e. the values in the inner anchor points of the warping function,

$$l(\boldsymbol{w}_i) = \sum_{i=1}^{n} \Big(y_i - \theta\big(v(t_j, w_i)\big)\Big)^2 + \lambda^{-2} w_i^T C^{-1} w_i. \tag{4.5}$$

In R,

```
Sigma_inv = ginv(Sigma)
wi_like = function(wi, yi, t, tn) {
  return(sum((yi - thetaest(v(wi, tn, t), wt = wt0))^2) +
         warp_par^{-2}*t(wi - tn)%*%Sigma_inv%*%(wi - tn))
}
```

As last part of the first iteration, we optimize over this,

```
for (i in 1:n) {
  w_est[, i] = optim(w_est[, i], wi_like, yi = y[, i], t = t, tn = tn)$par
}
```

Using these estimates to again estimate the mean curve $\hat{\theta}_1$, we continue as outlined above, and again 10 times. Then we take the estimate $\hat{\lambda}$ obtained from this and perform the same iteration procedure again, but this time fixing the estimate $\hat{\lambda}$ and only performing the warping estimation, doing the iterative shifts between (1) estimating the θ and (2) aligning the curves with the estimated warps. We stop the iterations when we reach a predefined maximal number of iterations, or if the mean squared error between two following estimated θs is below some level (0.0001). In short,

1. Estimate mean curve θ using a B-spline.
2. Initialize warping to be the identity.
3. Estimate λ, given the warp, by optimizing the relevant likelihood.
4. Use this λ to predict the warp.
5. Estimate new mean curve.
6. Repeat 3.-5. to obtain optimal $\hat{\lambda}$.
7. Fix $\hat{\lambda}$ in likelihood over the warp, and estimate the warps.
8. Estimate new mean curve.
9. Repeat 7.-8. until stop.

The full R-code can be found in Appendix A.3, where we implement it as a function `iteration_function` with arguments initialized as `(no_iter = 15, wn = 3, reg = 2, warp_par = 1)`. Note here that `reg = 1` corresponds to ignoring the warping contribution to the likelihood, `reg = 2` corresponds to the estimation outlined above, and `reg = 3` corresponds to assuming the warps being independent observations and having variance 1. I.e., in terms of the likelihoods/posteriors,

$$l(\boldsymbol{w}_i) = \sum_{i=1}^{n} \Big(y_i - \theta\big(v(t_j, w_i)\big)\Big)^2, \tag{reg=1}$$

$$l(\boldsymbol{w}_i) = \sum_{i=1}^{n} \Big(y_i - \theta\big(v(t_j, w_i)\big)\Big)^2 + \lambda^{-2} w_i^T C^{-1} w_i, \tag{reg=2}$$

$$l(\boldsymbol{w}_i) = \sum_{i=1}^{n} \Big(y_i - \theta\big(v(t_j, w_i)\big)\Big)^2 + \lambda^{-2} w_i^T w_i. \tag{reg=3}$$

5 Comparing the Methods

We want to see how the methods described in the previous section perform on data generated from our R-function `generate_data`. We should then consider how we want to compare how well the estimations are done.

5.1 Comparing the Mean Curve Estimates

Since the procedures are based on different measures of deviations from the mean curve, we should be careful not to use comparison methods that favours only one of the methods. We will then use two different measures for comparing the estimations of the mean curves: (1) Using the mean squared error, and (2) using the elastic difference. Recall that the elastic distance is the distance between the SRVF of the one function and the warped version of the SRVF of other function, minimized over all increasing diffeomophisms from $[0, 1]$ onto $[0, 1]$. We imagine that maybe (1) favours the LEBB procedure, and that (2) favours the `align_fPCA`, and we should keep this in mind.

We implement in `R`,

```
sq.distance = function(f1,f2) {
  return(mean((f1-f2)^2))
}
```

to compute the mean squared error, and then we use the function `elastic.distance` (from the package `fdarsvf`) to compute the elastic distance.

Furthermore, we will simply plot the estimated mean curves together with the true underlying mean curve.

5.2 Comparing the Estimation of the Warping Functions

We can plot the estimated warping functions against the true warping functions to get a feeling of the performance of our estimation of these. If the estimation is good, we would of course expect these plottings to lie around the identity in $[0, 1]$.

We also want to do a numerical tracking of the quality of our warping function estimation. This can again, as with the mean curves, be done with the mean squared error. But we would also like to have a measure equivalent to the elastic distance. In Section 4.1.2, we argued that we could use the distance given by (4.2) to find the distance between any two warpings. We will now implement this is `R`. First, we find the derivative in t of a given warping. We want to be able to do this based on the warping function's value in each of $t_1, \ldots, t_m$, so we use the approximation,

```
gradt = function(vv, tt) {
  ww  = vv
  w11 = c(ww[-1],0)
  t11 = c(t[-1],0)
  gradv = ((w11-ww)/(t11-t))[-m]
  return(gradv[tt <= t[-1] & tt >= t[-m]])
}
```

Now we can define the distance,

```
dis.warps = function(v1, v2) {
  sqrt_grad1 = function(t) sqrt(ifelse(sapply(t, function(t) gradt(v1, t))<0,0,
                                      sapply(t, function(t) gradt(v1, t))))
  sqrt_grad2 = function(t) sqrt(ifelse(sapply(t, function(t) gradt(v2, t))<0,0,
                                      sapply(t, function(t) gradt(v2, t))))
  sqrt_grad  = function(t) sqrt_grad1(t)*sqrt_grad2(t)
  acos(integrate(sqrt_grad, 0, 1, subdivisions=2000, stop.on.error=FALSE)[[1]])
}
```

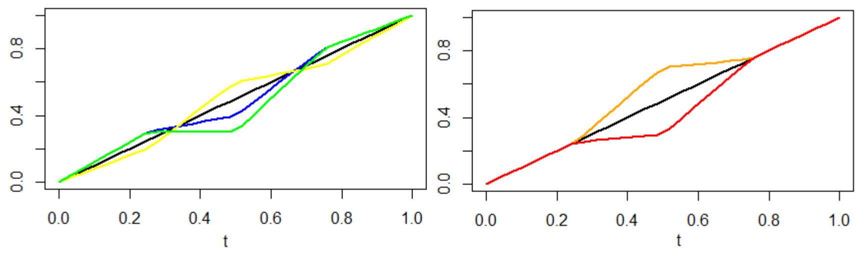

Figure 7: We look at different examples of warping functions, to see if the warping distance function measures in a sensible way for simple cases. The distances between the identity mapping (the black line) and each coloured warping function are as follows: 0.2247, 0.2247, 0.5131, 0.3085 and 0.3085. In the left plot we would expect the blue and the yellow warps to have the same distance to the identity, and we would expect the green to have a greater distance to the identity than the blue. This corresponds with what we see. In the right plot, the distance between the two coloured warps is furthermore 0.6082, which is less than or equal to 0.3085 + 0.3085 = 0.6171, which is also in correspondence with how we would expect a proper measure to behave.

All in all we make a function that takes the mean of the distances for all $i = 1, \dots, n$ between the true warp and the estimated warp. We call this `warp_distance`. The R-code can be found in Appendix A.4.

Having introduced this new measure of distance between warps, we can look at a few simple examples of warping functions and see how the distance behaves. This is done in Figure 7.

6 Testing the Methods for Aligning Data

For applying the method using the Fisher-Rao metric, described in Section 4.1, we install the R-package `fdasrvf` and then use the function `align_fPCA` directly. Note that Section 4.1 actually describes a somewhat simpler estimation procedure, as `align_fPCA` also uses a principal component analysis. Applying our own estimation procedure, we use the `iteration_function` which can be found in Appendix A.3.

6.1 Example From fdasrvf

From the package `fdasrvf` we can run the example data observations, `example(align_fPCA)`. This example is often used to show how well `example(align_fPCA)` works at aligning data. We want to see how good our own method, LEBB, does. The result of the two alignment procedures is shown in Figure 8. We see that the `example(align_fPCA)` does indeed result in some very nicely aligned curves. But our own constructed method looks like working very well too, though it could be argued to align a little less elegantly.[8]

6.2 Own Data Examples

What could be more interesting, would be to see how our own method LEBB and the `align_fPCA` works on data generated from our `generate_data` function. So, we first simply generate 40 completely random data sets, four of every one of the mean curve types and then using different warping effects and noise terms. In Figure 9 the result is shown for one of the four cases for each

[8]We should, however, keep in mind that we do not know how the true underlying functions actually behaves, so it is not impossible that our own method is actually doing a better job at aligning this data.

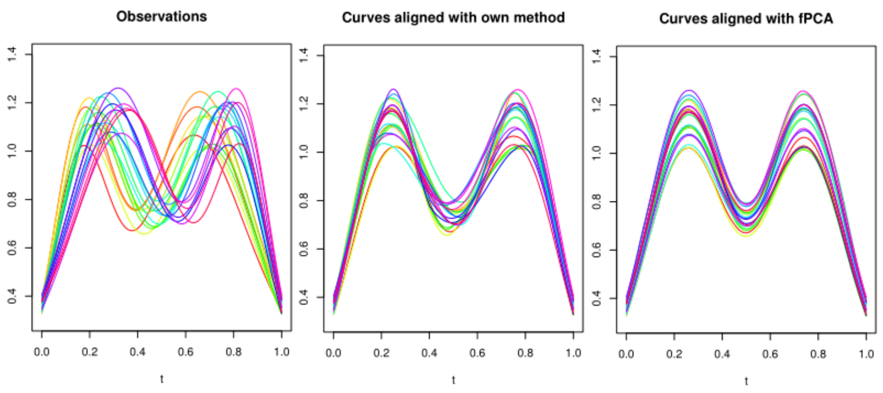

Figure 8: To the left the observations of the fdasrvf package example are plotted. We see that the curves are aligned very nicely using the align_fPCA method (though we do not know the true data), but our own iterative method is not bad either.

of the mean curves. Here, the true underlying data is plotted in the first column, the fitted mean curves in each estimation step of the LEBB procedure is plotted in the second, the aligned data and final estimated mean curve from LEBB is plotted in the third, and the aligned data and final estimated mean curve using `align_fPCA` is plotted in the fourth column. Each row corresponds to each its own example data set.

In Table 1 the mean of the distances described in Section 5.1 for each generated example can be found. Likewise the distances described in Section 5.2 can be found in Table 2. Looking at Table 1 we note that, as we expected, the mean squared error is lower than the elastic distance for the LEBB procedure in more cases than for the `align_fPCA`, and that the elastic distance is lower than the mean squared error in more cases for the `align_fPCA` than for the LEBB. In both tables we see that the LEBB minimizes the distances in all cases. We will make more comments on this later, as we should remain critical to this fact. Note also that especially in Table 1 the standard error is fairly higher for `align_fPCA` than for LEBB, so the fact that the LEBB procedure performs better might not be the case for all of the four simulated datasets. A last observation to make is that the tables provides an indication of which underlying curve structures each method is good at aligning.

Table 1: Mean squared error and elastic distances between the true curves and the estimated ones. The standard error is shown with small font size. The shortest distance is marked with bold in each case.

	Mean Squared Error		**Elastic Distance**	
	LEBB Estimation	`align_fPCA`	LEBB Estimation	`align_fPCA`
Example no. 1	**0.022** 0.030	0.650 0.371	**0.035** 0.035	0.799 0.337
Example no. 2	**0.022** 0.016	0.970 0.290	**0.028** 0.020	1.082 0.416
Example no. 3	**0.036** 0.033	0.713 0.211	**0.045** 0.039	0.701 0.231
Example no. 4	**0.039** 0.062	0.750 0.171	**0.051** 0.082	0.973 0.389
Example no. 5	**0.066** 0.051	1.084 0.464	**0.069** 0.062	1.088 0.389
Example no. 6	**0.062** 0.063	0.949 0.624	**0.085** 0.091	0.903 0.398
Example no. 7	**0.011** 0.010	0.588 0.316	**0.007** 0.005	0.643 0.229
Example no. 8	**0.054** 0.069	0.890 0.315	**0.077** 0.084	0.979 0.327
Example no. 9	**0.033** 0.050	0.757 0.588	**0.044** 0.066	0.952 0.342
Example no. 10	**0.052** 0.037	0.944 0.123	**0.081** 0.038	0.740 0.178

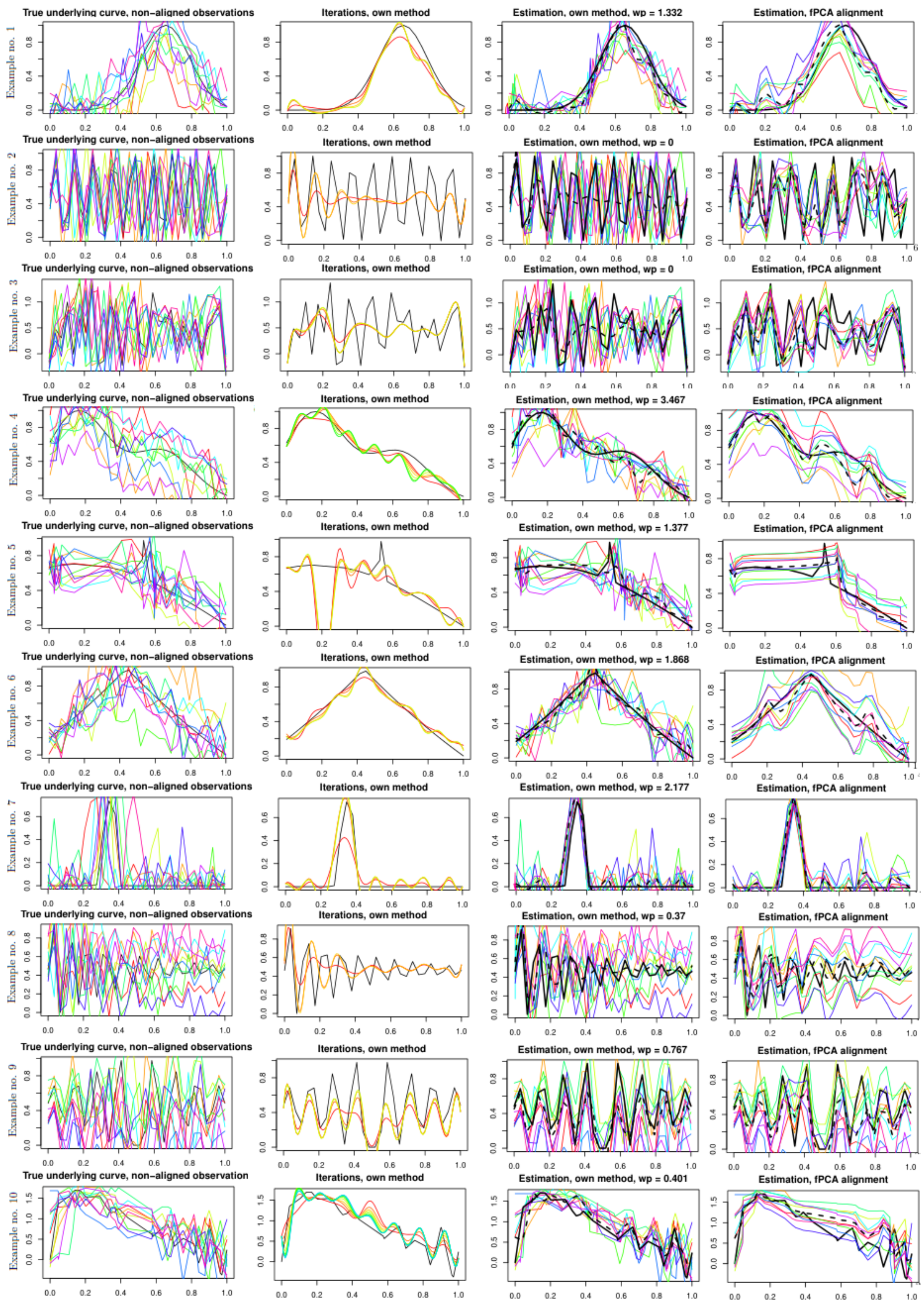

Figure 9: Generating different types of underlying curves and adding different warping and noise effects. The first plot in each row is the observed data. The second plot is the true underlying mean function (black) together with the fitted mean curve in each step of the self-constructed estimation procedure. The third plot is the aligned data using this method, with the true underlying curve (solid black) and the final estimated mean curve (dashed black). The fourth plot is the aligned data using align_fPCA together with, again, the true underlying curve (solid black), and this method's estimated mean curve (dashed black).

Table 2: Mean squared error and self-constructed distance between the true warps and the estimated ones. The standard error is shown with small font size. The shortest distance is marked with bold in each case.

	Mean Squared Error		**Self-Constructed Distance**	
	LEBB Warp	`align_fPCA` Warp	LEBB Warp	`align_fPCA` Warp
Example no. 1	**0.005** 0.007	0.009 0.003	**0.150** 0.072	0.349 0.097
Example no. 2	**0.005** 0.003	0.012 0.008	**0.206** 0.106	0.393 0.048
Example no. 3	**0.006** 0.002	0.013 0.004	**0.275** 0.096	0.446 0.105
Example no. 4	**0.006** 0.003	0.010 0.003	**0.413** 0.082	0.535 0.060
Example no. 5	**0.009** 0.002	0.013 0.002	**0.410** 0.033	0.523 0.052
Example no. 6	**0.006** 0.005	0.010 0.005	**0.197** 0.054	0.363 0.048
Example no. 7	**0.007** 0.006	0.010 0.004	**0.240** 0.109	0.349 0.055
Example no. 8	**0.004** 0.003	0.011 0.005	**0.247** 0.120	0.397 0.132
Example no. 9	**0.006** 0.002	0.014 0.001	**0.364** 0.063	0.491 0.029
Example no. 10	**0.006** 0.002	0.014 0.004	**0.352** 0.057	0.500 0.039

6.3 Systematising the Comparison of the Methods

In this section we will only generate data from one of the underlying mean curve types. We choose the two-peaked one as this is rather simple, and we can easily see the distortion of this. We will then add different effects independently, hoping to get an overview of where the methods perform best, and worst. Especially it is interesting to see how the methods are robust, or non-robust, to the addition of noise. We will consider four different noise set-ups:

1. No noise,
2. Brownian motion process,
3. Laplace iid noise, and
4. Matern process together with normal iid noise.

We will add each of 1.-4. to three different cases of a generated mean curves, corresponding to using (1) a simple linear warp, (2) a piecewise linear warp and (3) a piecewise smooth warp. This gives us 12 different scenarios which we will furthermore repeat five times. Figure 10 shows the resulting graphs of the first repetition, where we have plottet the underlying data, the aligned curves using each method, and, again using each method, the estimated warping functions plotted against the true warps. In Table 3 and Table 4 the mean distances and corresponding standard errors over the five repetitions can then be found, Table 3 for the estimated mean curves and Table 4 for the estimated warping functions.

Looking at Table 3 it again seems as though the self-constructed method of estimation consistently performs better than `align_fPCA`. Moreover, we note that overall both estimation methods are better when there is no noise (with some exceptions for the LEBB procedure). Especially, `align_fPCA` performs a lot worse when adding the different types of noise, whereas the self-constructed estimation appears to be a little less sensitive to this — the distances are for this method not that much higher for the noisy models than for then non-noisy one. From Table 4, we furthermore see that the self-constructed estimation also performs best on the warping estimation in almost all cases, but that `align_fPCA` is actually best in some of the cases of the Matern covariance and iid normal noise set-up.

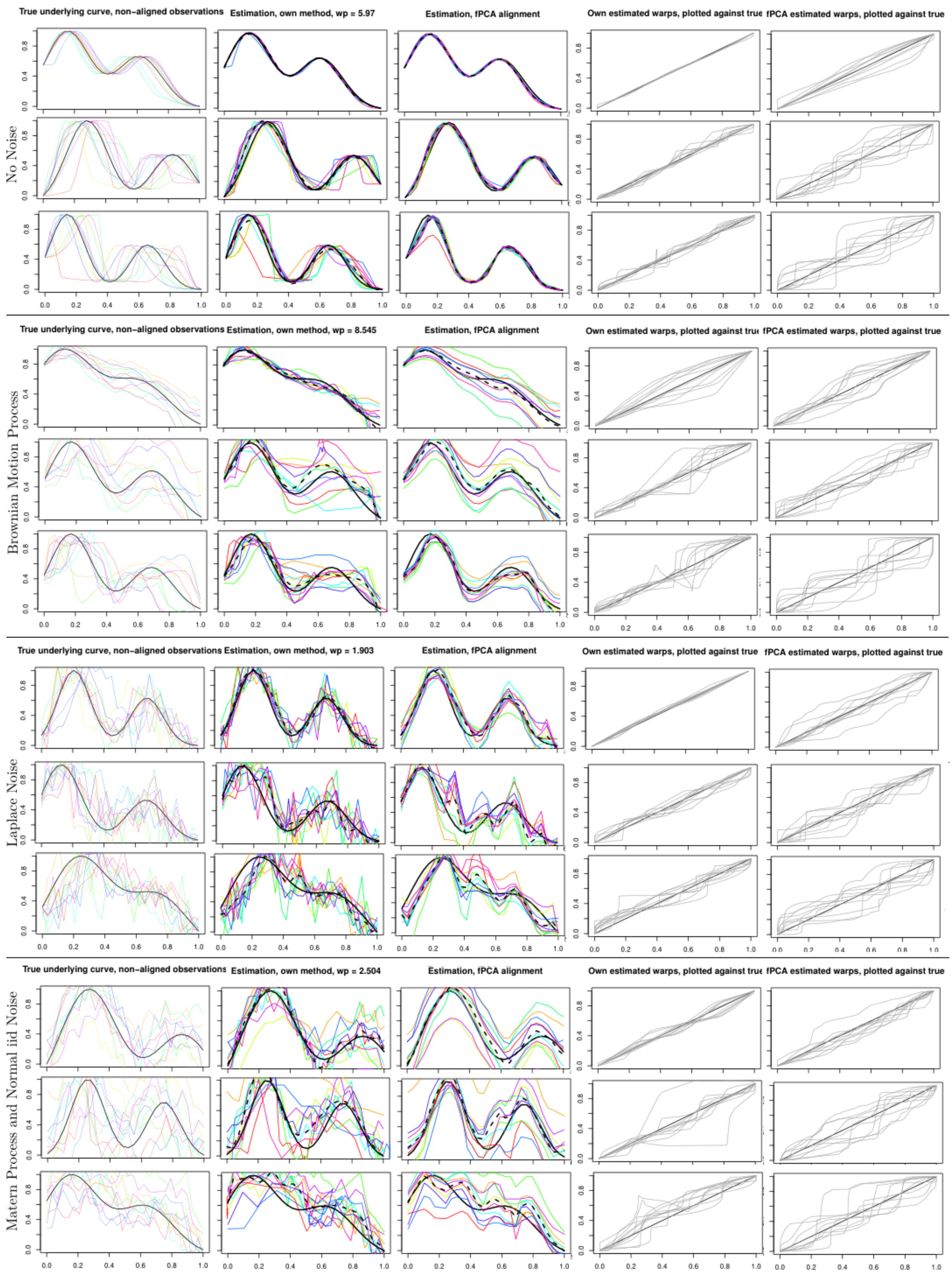

Figure 10: Different warping functions and adding different set-ups of noise, specified to the left and separated by lines. The first row of each case corresponds to a simple linear warp, the second corresponds to a piecewise linear warp, and the last row corresponds to a piecewise smooth warp.

Table 3: Mean squared error and elastic distance between the true curves and the estimated ones. The shortest distance is marked with bold in each case.

		Mean Squared Error		**Elastic Distance**	
		LEBB Estimation	`align_fPCA`	LEBB Estimation	`align_fPCA`
No noise	Linear Warp	**0.002** 0.002	0.142 0.056	**0.001** 0.001	0.120 0.030
	Piecew. Lin. W.	**0.011** 0.016	0.175 0.098	**0.009** 0.011	0.174 0.053
	Piecew. Sm. W.	**0.004** 0.004	0.172 0.074	**0.004** 0.003	0.141 0.046
BM	Linear Warp	**0.005** 0.004	0.165 0.045	**0.004** 0.003	0.157 0.042
	Piecew. Lin. W.	**0.005** 0.004	0.224 0.047	**0.002** 0.001	0.154 0.075
	Piecew. Sm. W.	**0.005** 0.001	0.172 0.076	**0.012** 0.005	0.214 0.023
Lapl.	Linear Warp	**0.009** 0.007	0.549 0.248	**0.011** 0.007	0.772 0.218
	Piecew. Lin. W.	**0.004** 0.002	0.387 0.243	**0.010** 0.006	0.752 0.288
	Piecew. Sm. W.	**0.005** 0.003	0.523 0.204	**0.014** 0.008	0.779 0.199
Ma.+N.	Linear Warp	**0.011** 0.008	0.366 0.013	**0.008** 0.352	0.464 0.500
	Piecew. Lin. W.	**0.017** 0.018	0.521 0.017	**0.015** 0.406	0.644 0.501
	Piecew. Sm. W.	**0.013** 0.021	0.510 0.016	**0.012** 0.482	0.590 0.515

Table 4: Mean squared error and self-constructed distance between the true warps and the estimated ones. The shortest distance is marked with bold in each case.

		Mean Squared Error		**Elastic Distance**	
		LEBB Warp	`align_fPCA` Warp	LEBB Warp	`align_fPCA` Warp
No noise	Linear Warp	**0.001** 0.001	0.013 0.008	**0.229** 0.143	0.426 0.246
	Piecew. Lin. W.	**0.002** 0.002	0.017 0.013	**0.230** 0.125	0.440 0.255
	Piecew. Sm. W.	**0.003** 0.002	0.019 0.012	**0.308** 0.141	0.508 0.248
BM	Linear Warp	**0.007** 0.005	0.015 0.007	**0.334** 0.175	0.484 0.201
	Piecew. Lin. Warp	**0.008** 0.007	0.014 0.010	**0.335** 0.227	0.431 0.265
	Piecew. Sm. Warp	**0.008** 0.008	0.021 0.010	**0.344** 0.117	0.520 0.171
Lapl.	Linear Warp	**0.003** 0.003	0.012 0.003	**0.273** 0.169	0.509 0.128
	Piecew. Lin. W.	**0.003** 0.003	0.019 0.010	**0.239** 0.141	0.499 0.119
	Piecew. Sm. W.	**0.003** 0.003	0.022 0.012	**0.330** 0.151	0.578 0.130
Ma.+N.	Linear Warp	**0.008** 0.008	0.013 0.008	**0.352** 0.143	0.500 0.138
	Piecew. Lin. W.	0.018 0.017	**0.017** 0.006	**0.406** 0.141	0.501 0.142
	Piecew. Sm. W.	0.021 0.008	**0.016** 0.010	**0.482** 0.114	0.515 0.136

7 Conclusion

In this project we have considered the problem of proper registration of functional data observations. We introduced a general set-up and considered different effects causing the misalignment of data observations from the same underlying mean curve. On this basis we constructed a function to use for simulating data imitating real data observations. An important feature about each simulated data set is that it is both misaligned because of a time axis shift, corresponding to the effect of some warping function, and also misaligned because of various types of random noise.

We looked into two methods of aligning functional data. Firstly, we considered the more mathematical approach, where the Fisher-Rao metric was used to minimize specific distances between the mean curve and the observed curves, warped under diffeomorphisms. What was specifically characteristic about this method was that it made no actual assumptions of noise in the underlying data. Secondly, we tried to construct our own estimation procedure. This procedure took a more statistical approach, assuming a model with different kinds of noise terms. The alignment here was done by an iterative maximum likelihood estimation.

We used the simulated data to test the two estimation methods to see if we could find each's strengths and weaknesses. The hypothesis was that the first method would possibly overfit to the noisy parts: First of all because of the lack of noise assumptions, but also because of the way `align_fPCA` is implemented — in this procedure the unknown functions are represented with a parameter per observation point. Second of all, we should also keep in mind that the data simulated and the model assumed in the self-constructed estimation process was based on some of the same ideas, which favours our own method when testing it together with `align_fPCA` on the simulated data. This is, in fact, what we have seen when comparing the two methods. In general, the self-constructed method is in this set-up consistently performing better at aligning the various kinds of simulated data. Based on the arguments above, we should however be very careful to conclude that our method is then better at aligning than `align_fPCA`. There are some very immediate limitations to the examination of the methods that we have performed, including the fact that the `align_fPCA` is per default using many parameters. In reality, we would probably get an even better estimation procedure if we made an adjustment of the `align_fPCA`, and if we tried to implement a combination of this and a modification of our own model. The `align_fPCA` is very powerful as it uses a simple mathematical framework for the alignment, but it has many practical limitations (e.g. the fact that it cannot be used with varying times of measurement) and it also overfits to noisy data. Our own model, on the other hand, is based on one specific way of modelling the data and the noise terms, and also, as now implemented, it is computationally rather inefficient.

Thus, this project provides, most importantly, a way of formalizing the variations of noise that can be found in real data, such that we, and others, can test various methods of curve alignment on the data simulated under these different identified models. Also, it constitutes a starting-point for further development and modification of the current alignment procedures, as we have seen that both methods investigated are on the one hand very strong at aligning data in some set-ups and on the other hand very limited in the way they are implemented.

Appendix

A R Code

A.1 Packages

```
library(fda)
library(fdasrvf)
library(fields)
library(reshape)
library(MASS)
```

A.2 Data Generating Function

```
#---- STEP 1: GENERATE TIME POINTS OF MEASURE ----#

generate_t = function(type = 1) {
  if (type == 1) {
    t = function(m) {
      seq(0, 1, length = m)
    }
  } else if (type == 2) {
    t = function(m) {
      c(0,sort(runif(m-2)),1)
    }
  }
}
tt = generate_t(type=2)

#---- STEP 2: GENERATE TRUE MEAN CURVE ----#

generate_theta = function(type = 1) {
  tx = seq(0, 1, length=100)
  if (type == 1) {
    mu    = runif(1, min = 0.3, max = 0.7)
    sigma = runif(1, min = 0.05, max = 0.15)
    theta = function(t) {
      dnorm(t, mean = mu, sd = sigma) /
        abs(max(dnorm(tx, mean = mu, sd = sigma)))
    }
    attr(theta, 'mu')    = mu
    attr(theta, 'sigma') = sigma
  } else if (type == 2) {
    a     = sample(8,1)+2
    b     = rnorm(1,mean = 0.2, sd = 0.25)
    theta = function(t) {
      ( sin(t*(2*pi*a))*b - min(sin(tx*(2*pi*a))*b,0) ) /
        abs(max(sin(tx*(2*pi*a))*b) - min(sin(tx*(2*pi*a))*b) )
    }
    attr(theta, 'a') = a
    attr(theta, 'b') = b
  } else if (type == 3) {
    n_points   = sample(80, 1) + 15
    tpoints    = sort(runif(n_points))
    ypoints    = runif(n_points, min = -1, max = 1)
    theta      = function(t) {
      approxfun(tpoints, ypoints, rule = 2)(t) -
        min(0, min(approxfun(tpoints, ypoints, rule = 2)(tx))) /
        max( approxfun(tpoints, ypoints, rule = 2)(tx) -
          min(min(approxfun(tpoints, ypoints, rule = 2)(tx))) )
    }
    attr(theta, 'tpoints') = tpoints
    attr(theta, 'ypoints') = ypoints
  } else if (type == 4) {
    mu    = runif(1, min = 0.1, max = 0.3)
    sigma = runif(1, min = 0.1, max = 0.2)
    a     = rnorm(1, mean = 0.5, sd = 0.05)
    b     = rnorm(1, mean = 0.6, sd = 0.05)
    theta = function(t) {
      ( dnorm(t, mean = mu, sd = sigma) + b*dnorm(t, mean = mu+a, sd = sigma) -
         min(dnorm(tx, mean = mu, sd = sigma) + b*dnorm(tx, mean = mu+a,
                                                   sd = sigma)) ) /
        abs(max(dnorm(tx, mean = mu, sd = sigma) + b*dnorm(tx, mean = mu+a,
                                                      sd = sigma) -
                min(dnorm(tx, mean = mu, sd = sigma) + b*dnorm(tx, mean = mu+a,
                                                            sd = sigma))))
    }
    attr(theta, 'mu')    = mu
    attr(theta, 'sigma') = sigma
    attr(theta, 'a')     = a
    attr(theta, 'b')     = b
  } else if (type == 5) {
    mu    = runif(1, min = 0.08, max = 0.25)
    a     = rnorm(1, mean = 0.45, sd = 0.1)
    b     = runif(1, min = 0.005, max = 0.02)
    c     = runif(1, min = 4, max = 6)
    theta = function(t) {
      ( c*dnorm(t, mean = mu, sd = 0.8) + b*dnorm(t, mean = mu+a, sd = 0.01) -
          min(c*dnorm(tx, mean = mu, sd = 0.8) + b*dnorm(tx, mean = mu+a,
                                                    sd = 0.01)) ) /
        abs(max(c*dnorm(tx, mean = mu, sd = 0.8) + b*dnorm(tx, mean = mu+a,
                                                      sd = 0.01) -
                min(c*dnorm(tx, mean = mu, sd = 0.8) + b*dnorm(tx, mean = mu+a,
                                                            sd = 0.01))))
    }
    attr(theta, 'mu') = mu
    attr(theta, 'a')  = a
    attr(theta, 'b')  = b
    attr(theta, 'c')  = c
  } else if (type == 6) {
    c = runif(1, min = 0.2, max = 0.7)
    a = rnorm(1, mean = c, sd = 0.1)
    b = rnorm(1, mean = 1.4, sd = 0.35)
    theta = function(t) {
      ( exp(-abs(t-a)/b)/(2*b) - min(exp(-abs(tx-a)/b)/(2*b)) ) /
        abs(max(exp(-abs(tx-a)/b)/(2*b) - min(exp(-abs(tx-a)/b)/(2*b))))
        # Laplace density
    }
    attr(theta, 'a') = a
    attr(theta, 'b') = b
  } else if (type == 7) {
    n = sample(10,1) + 45
    b = sort(runif(2))
    a = runif(n, min = b[1], max = b[2])
    tpoints = seq(0,1,length = 100)
    thet    = matrix(nrow = length(tpoints), ncol = n)
    ypoints = rep(0,length(tpoints))
    for (i in 1:n) {
      thet[, i] = dnorm(tpoints, mean = a[i], sd = 0.01)
    }
    for (j in 1:length(tpoints)) {
      ypoints[j] = sum(thet[j, ])
    }
    theta = function(t) approxfun(tpoints, ypoints, rule = 2)(t) /
      abs(max(approxfun(tpoints, ypoints, rule = 2)(tx)))
    attr(theta, 'tpoints') = tpoints
    attr(theta, 'ypoints') = ypoints
  } else if (type == 8) {
    a     = sample(8,1)+2
    b     = rnorm(1,mean = 0.2, sd = 0.25)
    theta = function(t) {
      (-min(0,min(exp(-t*2.5)*sin(t*(2*pi*a))*b)) + exp(-t*2.5)*
         sin(t*(2*pi*a))*b) /
            abs(max(-min(0,min(exp(-t*2.5)*sin(t*(2*pi*a))*b)) +
              exp(-tx*2.5)*sin(tx*(2*pi*a))*b))
    }
    attr(theta, 'a') = a
    attr(theta, 'b') = b
  } else if (type == 9) {
    a     = sample(8,1)+2
    b     = rnorm(1,mean = 0.2, sd = 0.25)
    theta = function(t) {
      (-min(exp(-abs(1/2-t)*2.5)*sin(abs(1/2-t)*(2*pi*a))*b) +
         exp(-abs(1/2-t)*2.5)*sin(abs(1/2-t)*(2*pi*a))*b) /
           abs(max(-min(exp(-abs(1/2-t)*2.5)*sin(abs(1/2-t)*(2*pi*a))*b) +
             exp(-abs(1/2-tx)*2.5)*sin(abs(1/2-tx)*(2*pi*a))*b))
    }
    attr(theta, 'a') = a
    attr(theta, 'b') = b
  } else if (type == 10) {
    a     = rnorm(1, mean = 1.5, sd = 0.25)
    b     = rnorm(1, mean = 0.5, sd = 0.25)
    theta = function(t) {
      dgamma(t, shape = a, scale = b) + 1/10*exp(t*2.5)*sin(t*(2*pi*8)) /
```

```
        abs(max(dgamma(tx, shape = a, scale = b) +
                1/10*exp(tx*2.5)*sin(tx*(2*pi*8))))
    }
    attr(theta, 'a') = a
    attr(theta, 'b') = b
  }
  attr(theta, 'tx') = tx
  return(theta)
}

#---- STEP 3: GENERATE GAUSSIAN PROCESSES ----#

generate_x = function(type = 1, r = 0.5, s = 2) {
  t     = seq(0, 1, length = m + 2)[2:(m+1)]
  x0    = rnorm(m, sd=0.2)
  Sigma = matrix(NA, m, m)
  if (type == 1) {
    cov_fct = function(s, t) {
      min(s, t) # BM
    }
    for (i in 1:m) {
      for (j in 1:m) {
        Sigma[i, j] = cov_fct(t[i], t[j])
      }
    }
  }
  else if (type == 2) {
    for (i in 1:m) {
      for (j in 1:m) {
        Sigma[i, j] = Matern(abs(t[i] - t[j]), scale=1, range=r, smoothness=s)
      }
    }
  }
  if (type < 3) {
    Sigma_sqrt = t(chol(Sigma)) # cholesky transform
    x          = Sigma_sqrt %*% x0
  }
  if (type == 3) {
    x = rep(0,m)
  }
  return(x)
}

#---- STEP 4: GENERATE WARPING FUNCTIONS ----#

generate_warp = function(type = 1) {
  if (type == 1) {
    a = rnorm(1, mean = 1, sd = 0.1)
    b = rnorm(1, mean = 0, sd = 0.1)
  } else if (type == 2) {
    a = rnorm(1, mean = 1, sd = 0)
    b = rnorm(1, mean = 0, sd = 0.1)
  } else if (type == 3) {
    a = rnorm(1, mean = 1, sd = 0.1)
    b = rnorm(1, mean = 0, sd = 0)
  }
  v = function(t) {
    a*t + b
  }
  if (type == 4) {
    a = runif(1)
    b = rnorm(1, mean = a, sd = 0.1)
    if (b > 1) b = 1
    if (b < 0) b = 0
    v = approxfun(c(0,a,1), c(0,b,1), rule = 2)
  }
  if (type == 5) {
    n = sample(9,1)+1
    a = sort(runif(n))
    b = rep(0,n)
    for (j in 1:n) {
      b[j] = rnorm(1, mean = a[j], sd = 0.1)
      if (b[j] > 1) b[j] = 1
      if (b[j] < 0) b[j] = 0
    }
    b = sort(b)
    v = approxfun(c(0,a,1), c(0,b,1), rule = 2)
  }
  if (type == 6) {
    a = runif(1)
    b = rnorm(1, mean = a, sd = 0.1)
    if (b > 1) b = 1
    if (b < 0) b = 0
    v = splinefun(c(0,a,1), c(0,b,1), method = "hyman")
  }
  if (type == 7) {
    n = sample(9,1)+1
    a = sort(runif(n))
    b = rep(0,n)
    for (j in 1:n) {
      b[j] = rnorm(1, mean = a[j], sd = 0.1)
      if (b[j] > 1) b[j] = 1
      if (b[j] < 0) b[j] = 0
    }
    b = sort(b)
    v = splinefun(c(0,a,1), c(0,b,1), method = "hyman")
  }
  if (type == 8) {
    v = function(t) t
  }
  if (type != 8) {
    attr(v, 'a') = a
    attr(v, 'b') = b
  }
  return(v)
}

#---- STEP 5: GENERATE NOISE TERMS ----#

rlaplace = function(n, mu = 0, sigma = 1){
  U    = runif(n, min = -1/2, max = 1/2)
  mu - sign(U)*sigma*log(1-2*abs(U))
}

generate_noise = function(type = 1) {
  if (type == 1) {
    noise = function(m) {
      rnorm(m, sd = 0.1)
    }
  } else if (type == 2) {
    noise = function(m) {
      rlaplace(m, sigma = 0.1) ## generate own
    }
  } else if (type == 3) {
    noise = function(m) {
      c(rep(0,m))
    }
  }
  return(noise)
}

#---- STEP 6: SETTING UP FUNCTION TO GENERATE DATA ----#

generate_data = function(theta_t = 1, t_t = 1, warp_t = 1, x_t = 1,
                         noise_t = 1, plot = TRUE) {
  y      = array(NA, dim = c(m,n))
  w_true = array(NA, dim = c(m,n))
  theta  = generate_theta(type = theta_t)
  t      = generate_t(type=t_t)(m)
  if (plot)  plot(t, theta(t), type='l')
    for (i in 1:n) {
    w_true[, i] = generate_warp(type=warp_t)(t)
    x           = generate_x(type=x_t)
    noise       = generate_noise(type=noise_t)(m)
    y[, i]      = theta(w_true[, i]) + x + noise
    if (plot) lines(t, y[, i], lwd = 0.45, col = rainbow(n)[i])
  }
  attr(y, 'theta')  = theta
  attr(y, 't')      = t
  attr(y, 'w_true') = w_true
  attr(y, 'x')      = x
  attr(y, 'noise')  = noise
  return(y)
```

```
}
```

A.3 Iteration Function

```
iteration_function = function(no_iter = 15, wn = 3, reg = 1, warp_par = 2) {

#---- STEP 1: SETTING UP NO OF ITERATIONS AND INITIALIZE MATRICES ETC ----#

wt      = matrix(0, nrow = no_iter, ncol = n*m)
tn      = seq(0, 1, length = wn+2)[2:(wn+1)]
w_est   = array(tn, dim = c(wn, n))

#---- STEP 2A: DEFINING THETA ESTIMATING FUNCTION ----#

thetaest = function(x, wt = wt0) {
  kts        = seq(0, 1, length = 25)
  bkts       = c(-0.2,1.2)
  basis      = bs(wt, knots = kts, intercept = TRUE, Boundary.knots = bkts)
  c          = ginv(t(basis) %*% basis) %*% t(basis) %*% as.numeric(y)
  basis1     = bs(x, knots = kts, intercept = TRUE, Boundary.knots = bkts)
  return(basis1 %*% c)
}

#---- STEP 2B: WHICH.PEAKS FUNCTION TO UPDATE NUMBER OF PEAKS ----#

which.peaks = function(x,partial=TRUE,decreasing=FALSE){
  if (decreasing){
    if (partial){
      which(diff(c(FALSE,diff(x)>0,TRUE))>0)
    }else {
      which(diff(diff(x)>0)>0)+1
    }
  }else {
    if (partial){
      which(diff(c(TRUE,diff(x)>=0,FALSE))<0)
    }else {
      which(diff(diff(x)>=0)<0)+1
    }
  }
}

peaks = length(which.peaks(thetaest(t, wt = rep(t,n))))

if (peaks > 3) nknots = 15
else nknots = 6

#---- STEP 2C: UPDATE THETA ESTIMATING FUNCTION ----#

thetaest = function(x, wt = wt0) {
  kts        = seq(0, 1, length = nknots)
  bkts       = c(-0.2,1.2)
  basis      = bs(wt, knots = kts, intercept = TRUE, Boundary.knots = bkts)
  c          = ginv(t(basis) %*% basis) %*% t(basis) %*% as.numeric(y)
  basis1     = bs(x, knots = kts, intercept = TRUE, Boundary.knots = bkts)
  return(basis1 %*% c)
}

#---- STEP 3: DEFINING FIT OF PIECEWISE LINEAR WARP ----#

v = function(w, tn, t) {
  approx(c(0, tn, 1), c(0, w, 1), xout = t)$y
}

#---- STEP 4A: SIMULATE BROWNIAN BRIDGE FOR FITTING WARP PARAMETERS ----#

Sigma = matrix(0, wn, wn)
cov_fct = function(s, t) {
  ( min(s, t) - s*t )
}
for (i in 1:(wn)) {
  for (j in 1:(wn)) {
    Sigma[i, j] = cov_fct(t[i], t[j])
  }
}

#---- STEP 4B: FIT VARIANCE PARAMETERS ----#

fit_warp_par = function(w_est) {

  #-- values of warps for all t, for all n and m
  w  = as.numeric(apply(w_est, 2, function(w) v(w, tn, t)))

  #-- the gradient of the warping function
  tw     = c(0,tn,1)
  grad_w = matrix(0, nrow = m, ncol = wn)
  for (i in 1:m) {
    for (j in 1:(wn+1)) {
      if ( t[i] <= tw[j+1] &
             t[i] >= tw[j] ) {
        if (j != 1) grad_w[i, j-1] = (-t[i]+tw[j+1]) / (tw[j+1]-tw[j])
        if (j != (wn+1)) grad_w[i, j] = (t[i]-tw[j]) / (tw[j+1]-tw[j])
      }
    }
  }

  #-- setting up dimensions of matrices
  C  = matrix(0, nrow = n*(wn), ncol = n*(wn))
  Z  = matrix(0, nrow = n*m, ncol = n*(wn))
  Zi = matrix(0, nrow = m, ncol = wn)

  #-- covariance matrix for the warpings
  for (i in 0:(n-1)) {
    C[(1+i*(wn)):((i+1)*(wn)),(1+i*(wn)):((i+1)*(wn))] = Sigma
  }

  #-- Z-matrix
  for (i in 0:(n-1)) {
    ti     = w[(i*m+1):((i+1)*m)]
    dtheta = ( thetaest(ti+10^(-5), wt = w) -
                 thetaest(ti-10^(-5), wt = w) ) / (2*10^(-5))
    for (j in 1:m) {
      Zi[j, ] = dtheta[j]*grad_w[j, ]
    }
    Z[(1+i*m):((i+1)*m),(1+i*(wn)):((i+1)*(wn))] = Zi
  }

  #-- vector of current estimated warping deviation values
  w0_vec  = rep(0, (wn)*n)
  for (i in 0:(n-1)) {
    w0_vec[(1+i*(wn)):(wn+i*(wn))] = w_est[, i+1] - tn
  }

  #-- approximative derivative in t of mean function
  thetafitted  = rep(0, m*n)
  for (i in 0:(n-1)) {
    ti                              = w[(i*m+1):((i+1)*m)]
    thetafitted[(1+i*(m)):(m+i*(m))] = thetaest(ti, wt = w)
  }

  #-- setting up dummy vector
  r = as.numeric(y) - thetafitted + Z %*% w0_vec

  likewarppar = function(warp_par) {
    V        = warp_par^2 * ( Z %*% C %*% t(Z) ) + diag(1,n*m)
    inv_V    = ginv(V)
    sigmahat = 1/(m*n) * t(r) %*% inv_V %*% r
    return( (n*m)*log(sigmahat) +
      determinant(V)$modulus )
  }

  warp_par = optimize(likewarppar, lower = 0, upper = 100)$minimum
  V        = warp_par^2 * ( Z %*% C %*% t(Z) ) + diag(1,n*m)
  inv_V    = ginv(V)
  sigmahat = 1/(m*n) * t(r) %*% inv_V %*% r

  return(warp_par)
}

#---- STEP 5: DEFINING LIKELIHOOD TO FIT WARPING PARAMETERS ----#

if (reg == 1) {
  wi_like = function(wi, yi, t, tn) {
    return(sum((yi - thetaest(v(wi, tn, t), wt = wt0))^2))
  }
```

```
}

if (reg == 2) {
  Sigma_inv = ginv(Sigma)
  wi_like = function(wi, yi, t, tn) {
    return(sum((yi - thetaest(v(wi, tn, t), wt = wt0))^2) +
             warp_par^{-2}*t(wi - tn)%*%Sigma_inv%*%(wi - tn))
  }
}

if (reg == 3) {
  wi_like = function(wi, yi, t, tn) {
    return(sum((yi - thetaest(v(wi, tn, t), wt = wt0))^2) + t(wi)%*%wi)
  }
}

#---- STEP 6: DOING THE ITERATIONS TO ESTIMATE WARP VARIANCE ----#

dist = rep(0,10)
warp_par_vec = rep(0,10)

if (reg == 2) {

  for (j in 1:10) {

    if (j == 1) {
      wt[j, ]  = rep(t,n)
      warp_par = fit_warp_par(matrix(seq(0, 1, length = wn+2)[2:(wn+1)],
                                    nrow = wn, ncol = n))
      w_est    = matrix(seq(0, 1, length = wn+2)[2:(wn+1)], nrow = wn, ncol = n)
    }

    else {
      wt[j, ]  = as.numeric(apply(w_est, 2, function(w) v(w, tn, t)))
      warp_par = fit_warp_par(w_est)
    }

    warp_par_vec[j] = warp_par
    wt0 = wt[j, ]

    for (i in 1:n) {
      w_est[, i] = optim(w_est[, i], wi_like, yi = y[, i], t = t, tn = tn)$par
    }

    if (j > 1) {

      w0 = wt[(j-1), ]
      w1 = wt[j, ]

      theta1 = function(t) thetaest(t, wt = w0)
      theta2 = function(t) thetaest(t, wt = w1)
      dist[j]   = sq.distance(theta1(t),theta2(t))
      iter_stop = j

      if (dist[j] < 0.0001) break
    }
  }
}

#---- STEP 7: DOING THE ITERATIONS, USING THE ESTIMATED WARP VARIANCE ----#

dist = rep(0,no_iter)

for (j in 1:no_iter) {

  if (j == 1) wt[j, ] = rep(t,n)
  else        wt[j, ] = as.numeric(apply(w_est, 2, function(w) v(w, tn, t)))

  wt0 = wt[j, ]

  for (i in 1:n) {
    w_est[, i] <- optim(w_est[, i], wi_like, yi = y[, i], t = t, tn = tn)$par
  }

  if (j > 1) {

    w0 = wt[(j-1), ]
    w1 = wt[j, ]

    theta1 = function(t) thetaest(t, wt = w0)
    theta2 = function(t) thetaest(t, wt = w1)

    dist[j]   = sq.distance(theta1(t),theta2(t))
    iter_stop = j

    if (dist[j] < 0.0001) break
  }
}

#---- STEP 8: SETTING UP OUTPUT ----#

warp_est = y

for (i in 1:n) {
  warp_est[, i] = v(w_est[, i], tn, t)
}

wt = wt[1:iter_stop, ]

attr(wt, 'warp_est') = warp_est
attr(wt, 'thetaest') = thetaest
attr(wt, 'nknots')   = nknots
attr(wt, 'warp_par') = warp_par

return(wt)
}
```

A.4 Warp Distance

```
warp_distance = function(vv1, vv2) {

gradt = function(vv, tt) {
  ww  = vv
  w11 = c(ww[-1],0)
  t11 = c(t[-1],0)
  gradv = ((w11-ww)/(t11-t))[-m]
  return(gradv[tt <= t[-1] & tt >= t[-m]])
}

dis.warps = function(v1, v2) {
  sqrt_grad1 = function(t) sqrt(ifelse(sapply(t, function(t) gradt(v1, t))<0,0,
                                       sapply(t, function(t) gradt(v1, t))))
  sqrt_grad2 = function(t) sqrt(ifelse(sapply(t, function(t) gradt(v2, t))<0,0,
                                       sapply(t, function(t) gradt(v2, t))))
  sqrt_grad  = function(t) sqrt_grad1(t)*sqrt_grad2(t)
  acos(integrate(sqrt_grad, 0, 1, subdivisions=2000, stop.on.error=FALSE)[[1]])
}

dis = rep(0, n)

for (i in 1:n) {
  dis[i] = dis.warps(vv1[, i], vv2[, i])
}

distance = mean(dis)
attr(distance, "dis") = dis

return(distance)
}
```

References

[1] Lars Lau Rakêt, Stefan Sommer, and Bo Markussen. A nonlinear mixed-effects model for simultaneous smoothing and registration of functional data. *Pattern Recognition Letters*, 38:1–7, 2014.

[2] James O Ramsay. *Functional Data Analysis*. Wiley Online Library, 2006.

[3] Anuj Srivastava, Wei Wu, Sebastian Kurtek, Eric Klassen, and JS Marron. Registration of functional data using fisher-rao metric. *arXiv preprint arXiv:1103.3817*, 2011.