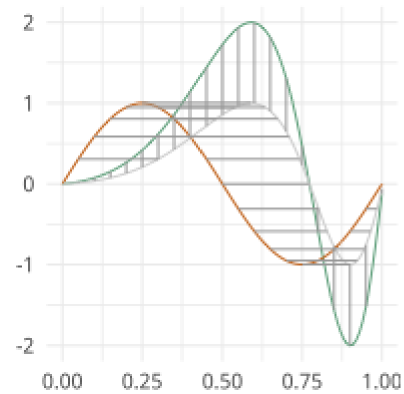
fdasrsf Documentation

Release 2.4.2

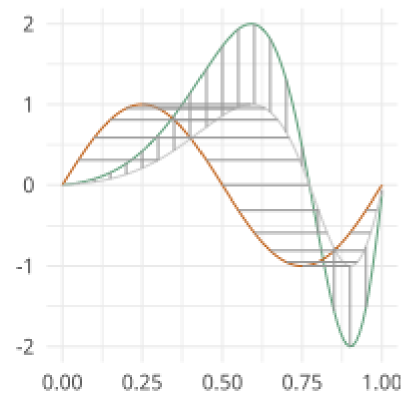
J. Derek Tucker

Jul 21, 2023

CONTENTS



A python package for functional data analysis using the square root slope framework and curves using the square root velocity framework which performs pair-wise and group-wise alignment as well as modeling using functional component analysis and regression.

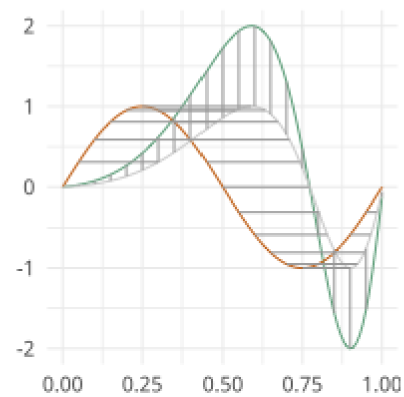


CHAPTER

ONE

USER GUIDE

Contents:



1.1 Elastic Functional Alignment

Otherwise known as time warping in the literature is at the center of elastic functional data analysis. Here our goal is to separate out the horizontal and vertical variability of the functional data

[1]:
```
import fdasrsf as fs
import numpy as np
```

Load in our example data

[2]:
```
data = np.load('../../bin/simu_data.npz')
time = data['arr_1']
f = data['arr_0']
```

We will then construct the `fdawarp` object

[3]:
```
obj = fs.fdawarp(f,time)
```

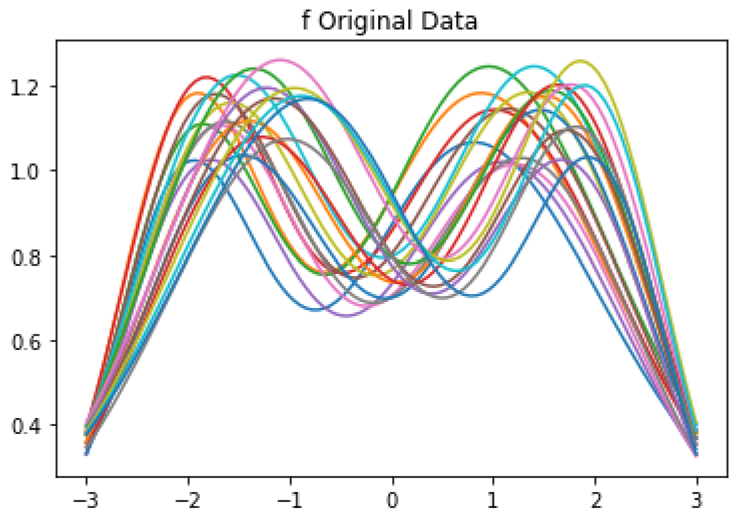
Next we will align the functions using the elastic framework

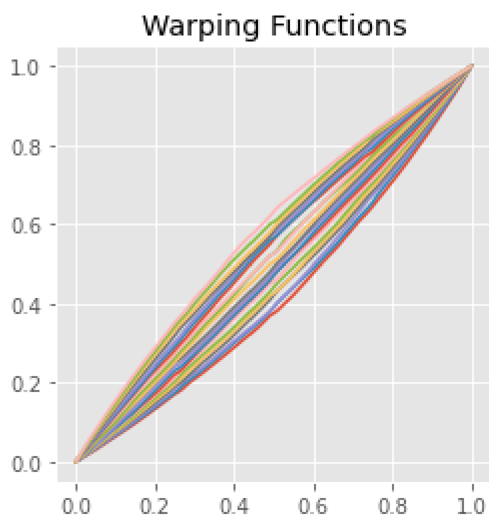
[4]:
```
obj.srsf_align(parallel=True)
```

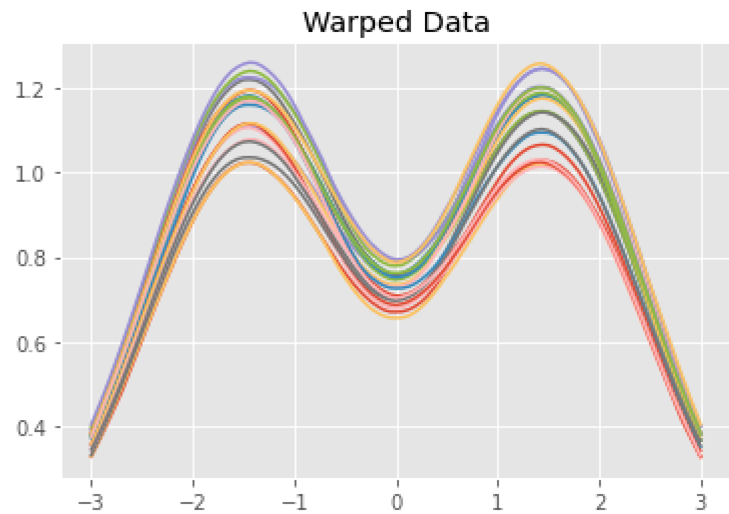
```
Initializing...
Compute Karcher Mean of 21 function in SRSF space...
updating step: r=1
updating step: r=2
```

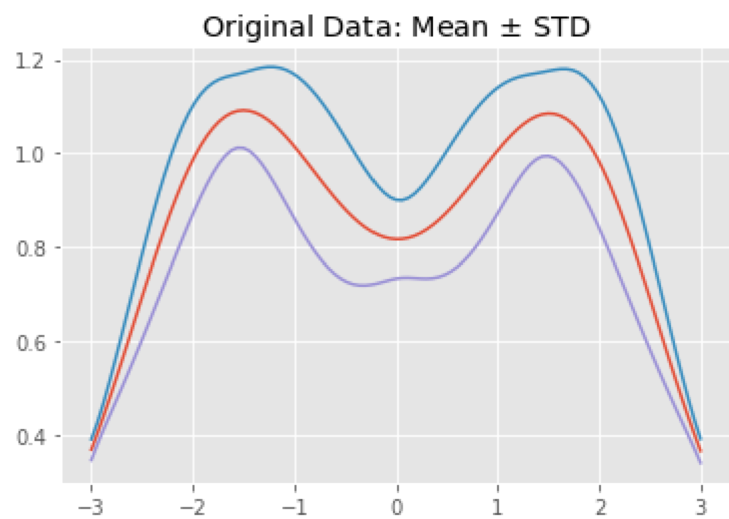
Display plots demonstrating the alignment

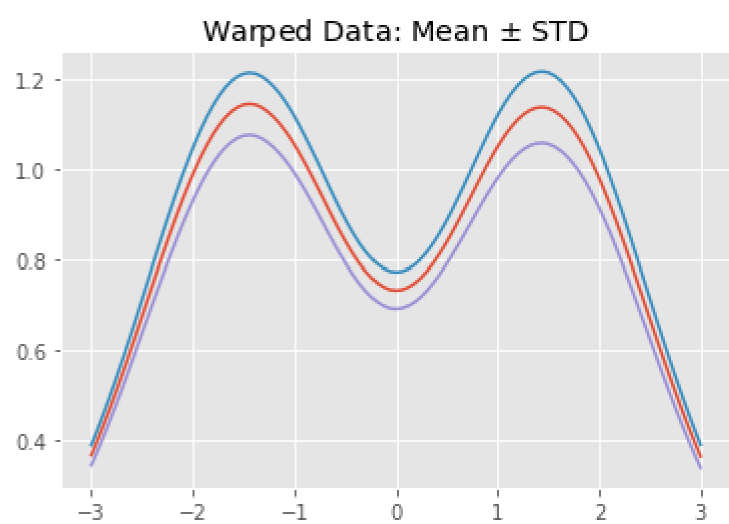
[5]:
```
obj.plot()
```

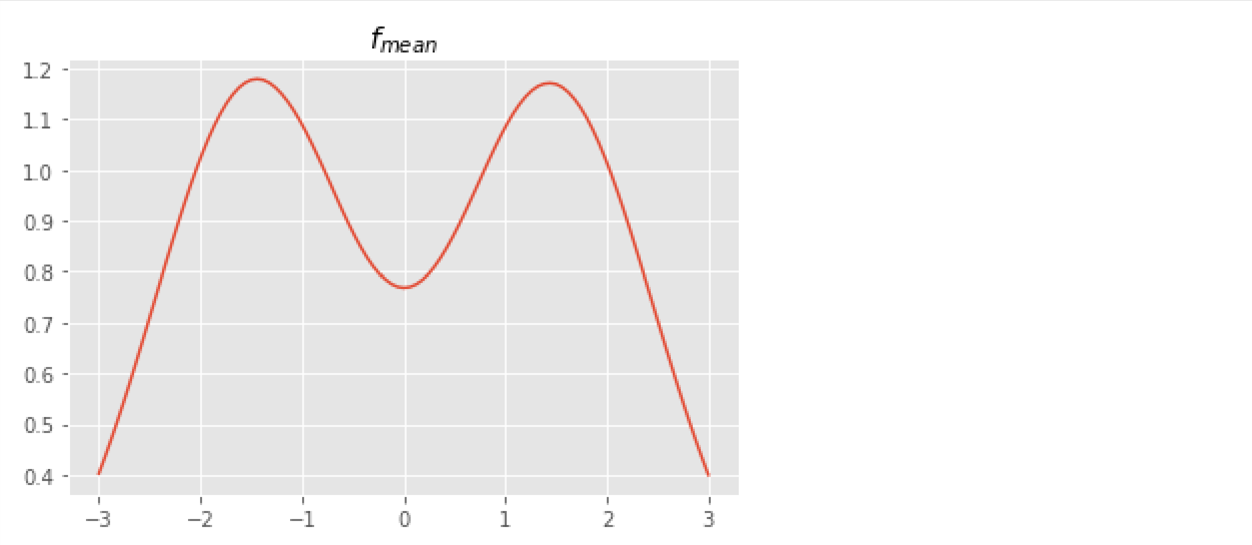


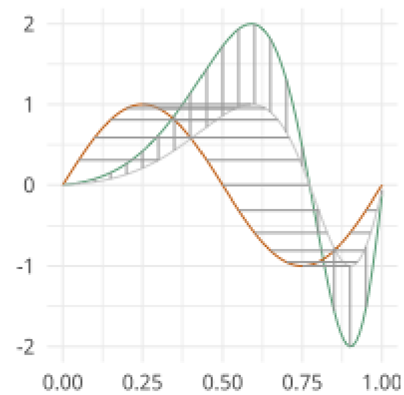



Warped Data
1.2
1.0
0.8
0.6
0.4
−3
−2
−1
0
1
2
3


Original Data: Mean ± STD
1.2
1.0
0.8
0.6
0.4
−3
−2
−1
0
1
2
3


Warped Data: Mean ± STD
1.2
1.0
0.8
0.6
0.4
−3
−2
−1
0
1
2
3





1.2 Elastic Functional Principal Component Analysis

After we have aligned our data we can compute functional principal component analysis (fPCA) on the aligned data, warping functions, and jointly

[1]:
```
import fdasrsf as fs
import numpy as np
```

We will load in our example data again and compute the alignment

[2]:
```
data = np.load('../../bin/simu_data.npz')
time = data['arr_1']
f = data['arr_0']
obj = fs.fdawarp(f,time)
obj.srsf_align(parallel=True)
```

```
Initializing...
Compute Karcher Mean of 21 function in SRSF space...
updating step: r=1
updating step: r=2
```

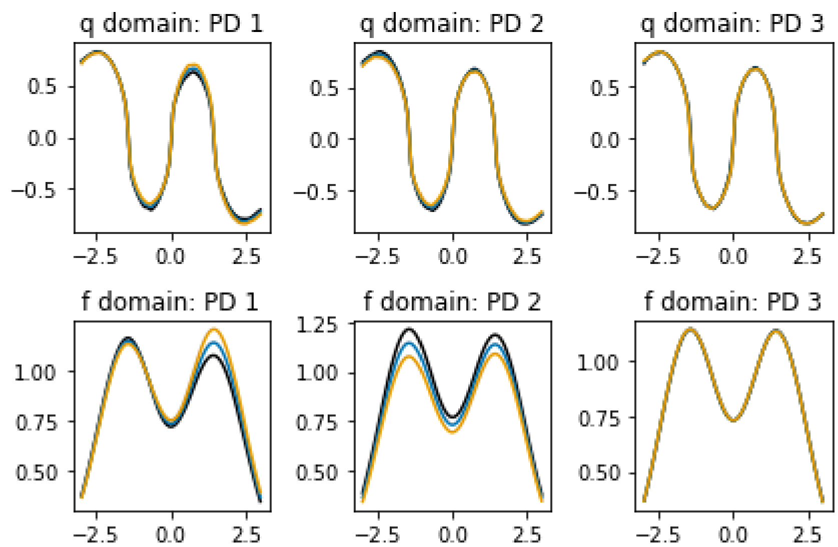
1.2.1 Vertical fPCA

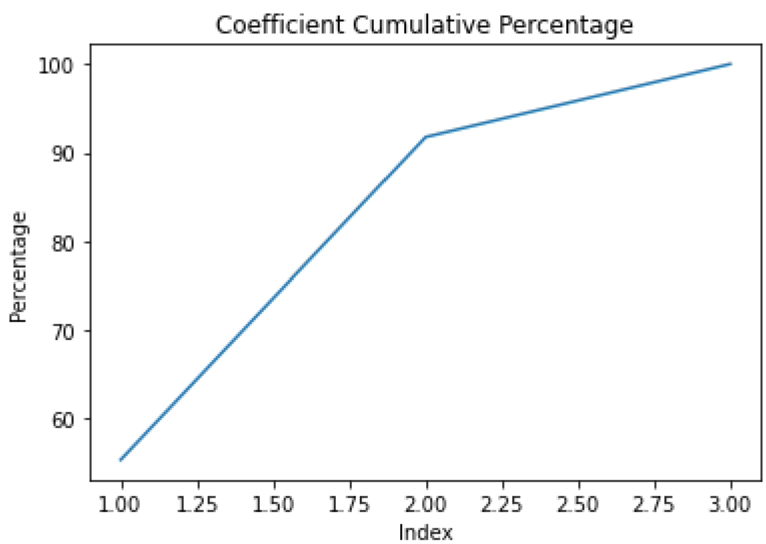
We will first compute fPCA on the aligned functions, by constructing the object and computing the PCA for the number of components, default=3)

[3]:
```
vpca = fs.fdavpca(obj)
vpca.calc_fpca(no=3)
```

We then can plot the principal directions

[4]:
```
vpca.plot()
```





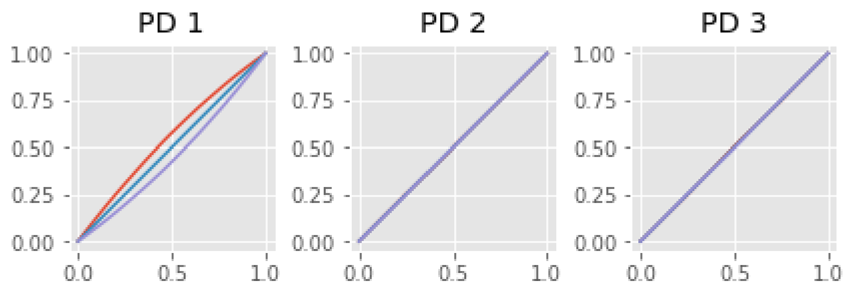
1.2.2 Horizontal fPCA

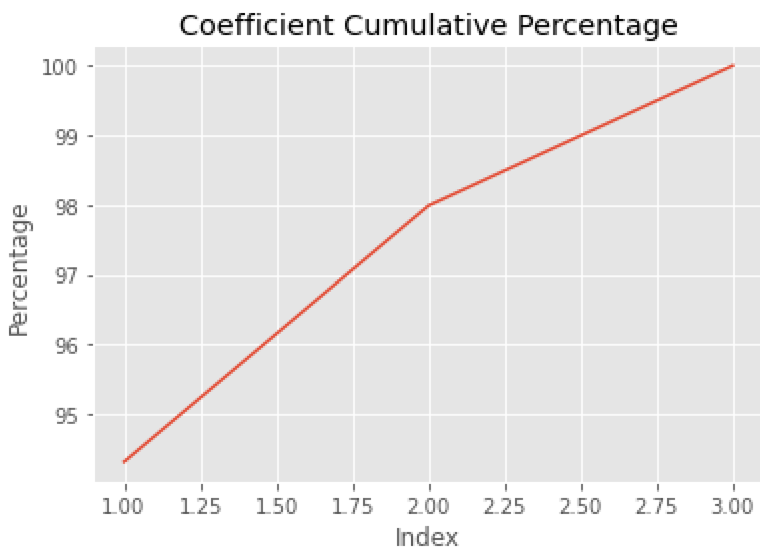
We can then compute PCA on the set of warping functions

[5]:
```
hpca = fs.fdahpca(obj)
hpca.calc_fpca(no=3)
```

We then can plot the principal directions

[6]:
```
hpca.plot()
```





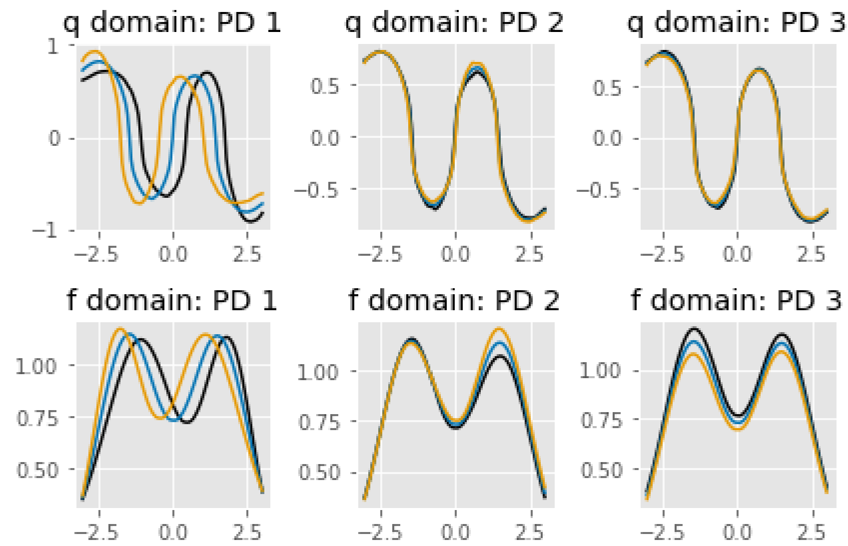
1.2.3 Joint fPCA

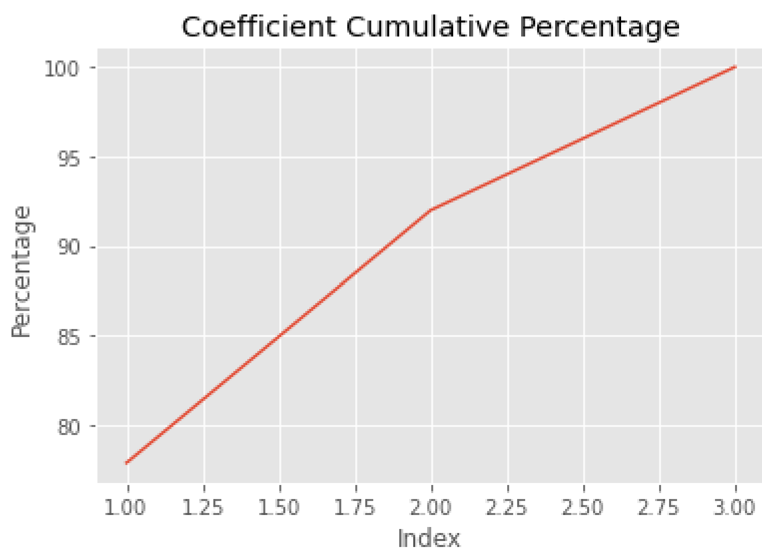
We can also compute the fPCA on jointly on the phase/amplitude space if we feel there is correlation between the variabilities

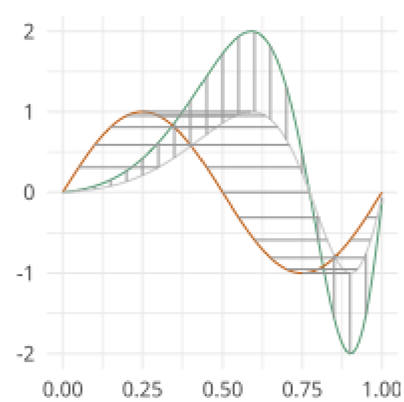
[7]:
```
jpca = fs.fdajpca(obj)
jpca.calc_fpca(no=3)
```

We then can plot the principal directions

[8]:
```
jpca.plot()
```


q domain: PD 1
q domain: PD 2
q domain: PD 3
f domain: PD 1
f domain: PD 2
f domain: PD 3


Coefficient Cumulative Percentage
Percentage
Index



1.3 Elastic Curve Alignment

Otherwise known as time warping in the literature is at the center of elastic functional data analysis. Here our goal is to separate out the horizontal and vertical variability of the open/closed curves

[1]:
```
import fdasrsf as fs
import numpy as np
```

Load in our example data

[2]:
```
data = np.load('../../bin/MPEG7.npz',allow_pickle=True)
Xdata = data['Xdata']
curve = Xdata[0,1]
n,M = curve.shape
K = Xdata.shape[1]

beta = np.zeros((n,M,K))
for i in range(0,K):
    beta[:,:,i] = Xdata[0,i]
```

We will then construct the `fdacurve` object

[3]:
```
obj = fs.fdacurve(beta,N=M)
```

We then will compute karcher mean of the curves

[4]:
```
obj.karcher_mean()
```

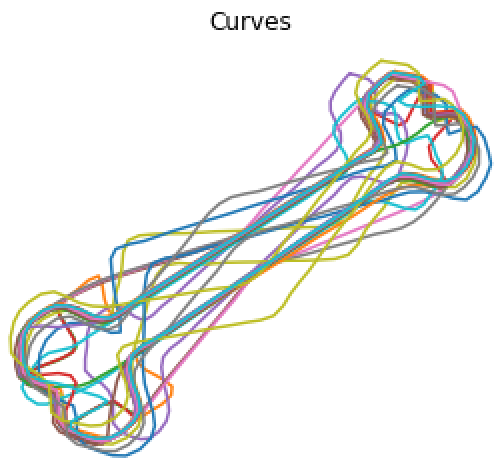
```
Computing Karcher Mean of 20 curves in SRVF space..
updating step: 1
updating step: 2
updating step: 3
updating step: 4
updating step: 5
updating step: 6
updating step: 7
```

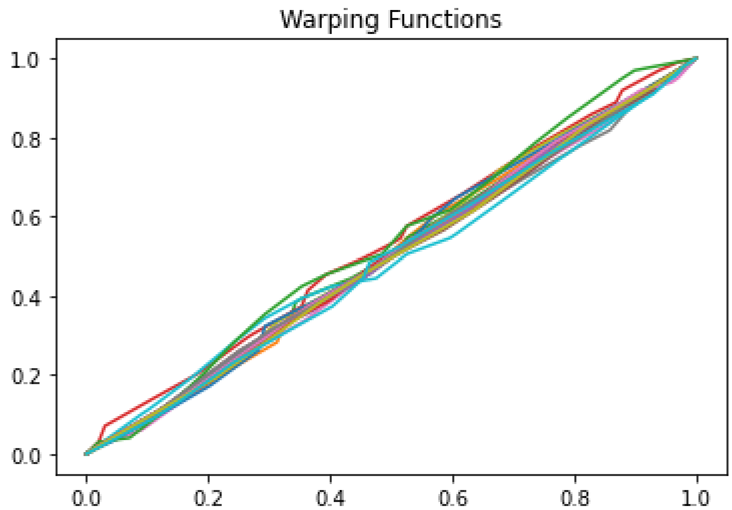
We then can align the curves to the karcher mean

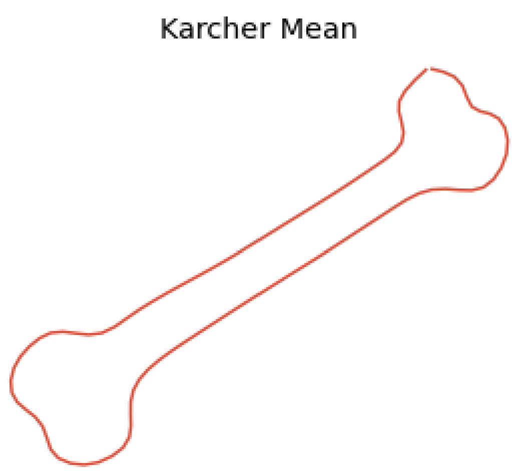
[5]:
```
obj.srvf_align(rotation=False)
```

Plot the results

[6]:
```
obj.plot()
```


Curves


Warping Functions
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Karcher Mean

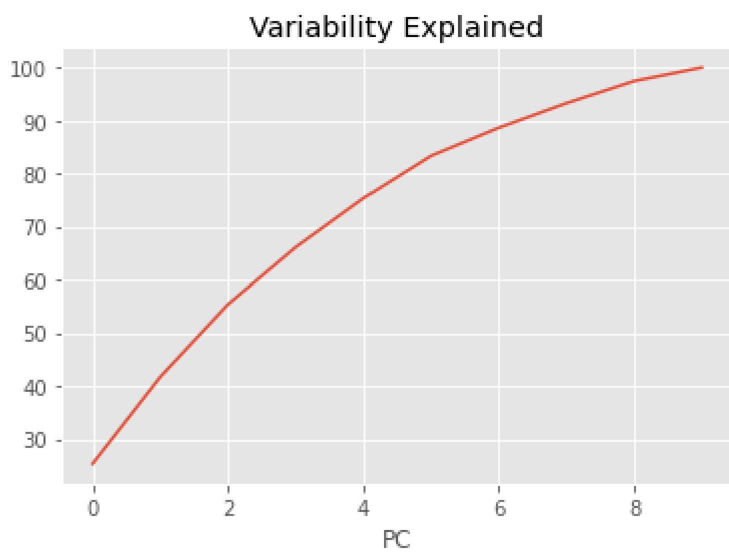
1.3.1 Shape PCA

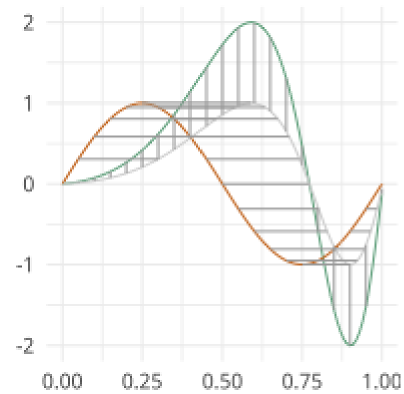
We then can compute the Karcher covariance and compute the shape pca

```
[7]: obj.karcher_cov()
     obj.shape_pca()
```

Plot the principal directions

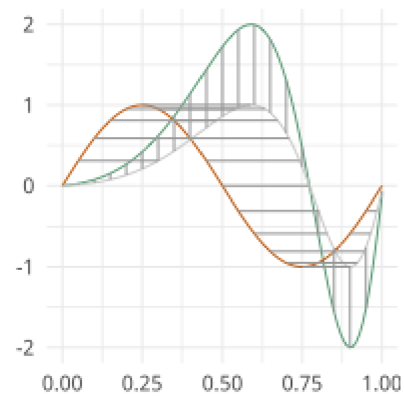
```
[8]: obj.plot_pca()
```




2
1
0
-1
-2
0.00
0.25
0.50
0.75
1.00

CHAPTER TWO

FUNCTIONAL ALIGNMENT



CHAPTER

THREE

FUNCTIONAL PRINCIPAL COMPONENT ANALYSIS

Vertical and Horizontal Functional Principal Component Analysis using SRSF

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

class fPCA.**fdahpca**(*fdawarp*)

This class provides horizontal fPCA using the SRVF framework

Usage: obj = fdahpca(warp_data)

Parameters

- **warp_data** – fdawarp class with alignment data
- **gam_pca** – warping functions principal directions
- **psi_pca** – srvf principal directions
- **latent** – latent values
- **U** – eigenvectors
- **coef** – coefficients
- **vec** – shooting vectors
- **mu** – Karcher Mean
- **tau** – principal directions

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 15-Mar-2018

calc_fpca(*no=3*, *stds=array([- 1, 0, 1])*)

This function calculates horizontal functional principal component analysis on aligned data

Parameters

- **no** (*int*) – number of components to extract (default = 3)
- **stds** – number of standard deviations along gedoesic to compute (default = -1,0,1)

Return type
fdahpca object of numpy ndarray

Return q_pca
srsf principal directions

Return f_pca
functional principal directions

Return latent
latent values

Return coef
coefficients

Return U
eigenvectors

`plot()`
plot plot elastic horizontal fPCA results

Usage: obj.plot()

class `fPCA.fdajpca(`*fdawarp*`)`
This class provides joint fPCA using the SRVF framework

Usage: obj = fdajpca(warp_data)

Parameters

- **warp_data** – fdawarp class with alignment data
- **q_pca** – srvf principal directions
- **f_pca** – f principal directions
- **latent** – latent values
- **coef** – principal coefficients
- **id** – point used for f(0)
- **mqn** – mean srvf
- **U** – eigenvectors
- **mu_psi** – mean psi
- **mu_g** – mean g
- **C** – scaling value
- **stds** – geodesic directions

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 18-Mar-2018

`calc_fpca(`*no=3*, *stds=array([- 1., 0., 1.])*, *id=None*, *parallel=False*, *cores=- 1*`)`
This function calculates joint functional principal component analysis on aligned data

Parameters

- **no** (*int*) – number of components to extract (default = 3)
- **id** (*int*) – point to use for f(0) (default = midpoint)
- **stds** – number of standard deviations along gedoesic to compute (default = -1,0,1)
- **parallel** (*bool*) – run in parallel (default = F)
- **cores** (*int*) – number of cores for parallel (default = -1 (all))

Return type
fdajpca object of numpy ndarray

Return q_pca
srsf principal directions

Return f_pca
functional principal directions

Return latent
latent values

Return coef
coefficients

Return U
eigenvectors

`plot()`

plot plot elastic vertical fPCA result

Usage: obj.plot()

`class fPCA.fdavpca(fdawarp)`

This class provides vertical fPCA using the SRVF framework

Usage: obj = fdavpca(warp_data)

Parameters

- `warp_data` – fdawarp class with alignment data
- `q_pca` – srvf principal directions
- `f_pca` – f principal directions
- `latent` – latent values
- `coef` – principal coefficients
- `id` – point used for f(0)
- `mqn` – mean srvf
- `U` – eigenvectors
- `stds` – geodesic directions

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 15-Mar-2018

`calc_fpca(no=3, id=None, stds=array([- 1, 0, 1]))`

This function calculates vertical functional principal component analysis on aligned data

Parameters

- `no` (*int*) – number of components to extract (default = 3)
- `id` (*int*) – point to use for f(0) (default = midpoint)
- `stds` – number of standard deviations along gedoesic to compute (default = -1,0,1)

Return type
fdavpca object containing

Return q_pca
srsf principal directions

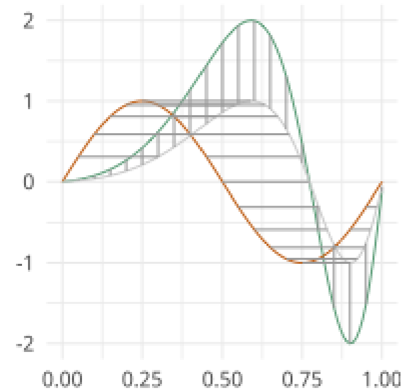
Return f_pca
functional principal directions

Return latent
latent values

Return coef
coefficients

Return U
eigenvectors

plot()
plot plot elastic vertical fPCA result Usage: obj.plot()



CHAPTER FOUR

ELASTIC FUNCTIONAL BOXPLOTS

Elastic Functional Boxplots

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

class `boxplots.ampbox`(*fdawarp*)

This class provides amplitude boxplot for functional data using the SRVF framework

Usage: obj = ampbox(warp_data)

Parameters

- **warp_data** (*fdawarp*) – fdawarp class with alignment data
- **Q1** – First quartile
- **Q3** – Second quartile
- **Q1a** – First quantile based on alpha
- **Q3a** – Second quantile based on alpha
- **minn** – minimum extreme function
- **maxx** – maximum extreme function
- **outlier_index** – indexes of outlier functions
- **f_median** – median function
- **q_median** – median srvf
- **plt** – surface plot mesh

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 15-Mar-2018

construct_boxplot(*alpha=0.05*, *k_a=1*)

This function constructs the amplitude boxplot using the elastic square-root slope (srsf) framework.

Parameters

- **alpha** – quantile value (e.g.,=.05, i.e., 95%)
- **k_a** – scalar for outlier cutoff (e.g.,=1)

plot()

plot box plot and surface plot

Usage: obj.plot()

class boxplots.**phbox**(*fdawarp*)

This class provides phase boxplot for functional data using the SRVF framework

Usage: obj = phbox(warp_data)

Parameters

- **warp_data** (*fdawarp*) – fdawarp class with alignment data
- **Q1** – First quartile
- **Q3** – Second quartile
- **Q1a** – First quantile based on alpha
- **Q3a** – Second quantile based on alpha
- **minn** – minimum extreme function
- **maxx** – maximum extreme function
- **outlier_index** – indexes of outlier functions
- **median_x** – median warping function
- **psi_median** – median srvf of warping function
- **plt** – surface plot mesh

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 15-Mar-2018

construct_boxplot(*alpha=0.05*, *k_a=1*)

This function constructs phase boxplot for functional data using the elastic square-root slope (srsf) framework.

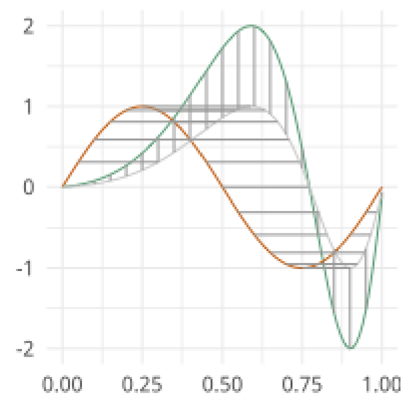
Parameters

- **alpha** – quantile value (e.g.,=.05, i.e., 95%)
- **k_a** – scalar for outlier cutoff (e.g.,=1)

plot()

plot box plot and surface plot

Usage: obj.plot()



CHAPTER FIVE

FUNCTIONAL PRINCIPAL LEAST SQUARES

Partial Least Squares using SVD

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

`fPLS.pls_svd`(*time*, *qf*, *qg*, *no*, *alpha=0.0*)

This function computes the partial least squares using SVD

Parameters

- **time** – vector describing time samples
- **qf** – numpy ndarray of shape (M,N) of N functions with M samples
- **qg** – numpy ndarray of shape (M,N) of N functions with M samples
- **no** – number of components
- **alpha** – amount of smoothing (Default = 0.0 i.e., none)

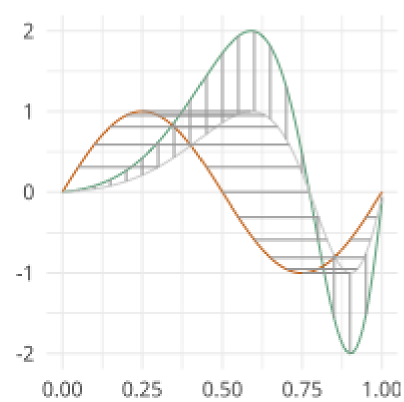
Return type
numpy ndarray

Return wqf
f weight function

Return wqg
g weight function

Return alpha
smoothing value

Return values
singular values



CHAPTER

SIX

ELASTIC REGRESSION

Warping Invariant Regression using SRSF

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

class regression.**elastic_logistic**(*f*, *y*, *time*)

This class provides elastic logistic regression for functional data using the SRVF framework accounting for warping

Usage: obj = elastic_logistic(f,y,time)

Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy array of N responses
- **time** (*np.ndarray*) – vector of size M describing the sample points
- **B** – optional matrix describing Basis elements
- **alpha** – alpha parameter of model
- **beta** – beta(t) of model
- **fn** – aligned functions - numpy ndarray of shape (M,N) of M functions with N samples
- **qn** – aligned srvfs - similar structure to fn
- **gamma** – calculated warping functions
- **q** – original training SRSFs
- **b** – basis coefficients
- **Loss** – logistic loss

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 29-Oct-2021

calc_model(*B=None*, *lam=0*, *df=20*, *max_itr=20*, *cores=- 1*, *smooth=False*)

This function identifies a regression model with phase-variability using elastic pca

Parameters

- **B** – optional matrix describing Basis elements
- **lam** – regularization parameter (default 0)
- **df** – number of degrees of freedom B-spline (default 20)
- **max_itr** – maximum number of iterations (default 20)
- **cores** – number of cores for parallel processing (default all)

predict(*newdata=None*)

This function performs prediction on regression model on new data if available or current stored data in object Usage: obj.predict()

obj.predict(newdata)

Parameters

- **newdata** (*dict*) – dict containing new data for prediction (needs the keys below, if None predicts on training data)
- **f** – (M,N) matrix of functions
- **time** – vector of time points
- **y** – truth if available
- **smooth** – smooth data if needed
- **sparam** – number of times to run filter

class regression.**elastic_mlogistic**(*f*, *y*, *time*)

This class provides elastic multinomial logistic regression for functional data using the SRVF framework accounting for warping

Usage: obj = elastic_mlogistic(f,y,time)

Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy array of N responses
- **time** (*np.ndarray*) – vector of size M describing the sample points
- **B** – optional matrix describing Basis elements
- **alpha** – alpha parameter of model
- **beta** – beta(t) of model
- **fn** – aligned functions - numpy ndarray of shape (M,N) of N functions with M samples
- **qn** – aligned srvfs - similar structure to fn
- **gamma** – calculated warping functions
- **q** – original training SRSFs
- **b** – basis coefficients
- **Loss** – logistic loss

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 29-Oct-2021

calc_model(*B=None*, *lam=0*, *df=20*, *max_itr=20*, *delta=0.01*, *cores=- 1*, *smooth=False*)

This function identifies a regression model with phase-variability using elastic pca

Parameters

- **B** – optional matrix describing Basis elements
- **lam** – regularization parameter (default 0)
- **df** – number of degrees of freedom B-spline (default 20)
- **max_itr** – maximum number of iterations (default 20)

- **cores** – number of cores for parallel processing (default all)

predict(*newdata=None*)

This function performs prediction on regression model on new data if available or current stored data in object Usage: obj.predict()

obj.predict(newdata)

Parameters

- **newdata** (*dict*) – dict containing new data for prediction (needs the keys below, if None predicts on training data)
- **f** – (M,N) matrix of functions
- **time** – vector of time points
- **y** – truth if available
- **smooth** – smooth data if needed
- **sparam** – number of times to run filter

class regression.**elastic_regression**(*f*, *y*, *time*)

This class provides elastic regression for functional data using the SRVF framework accounting for warping

Usage: obj = elastic_regression(f,y,time)

Parameters

- **f** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy array of N responses
- **time** – vector of size M describing the sample points
- **B** – optional matrix describing Basis elements
- **alpha** – alpha parameter of model
- **beta** – beta(t) of model
- **fn** – aligned functions - numpy ndarray of shape (M,N) of M functions with N samples
- **qn** – aligned srvfs - similar structure to fn
- **gamma** – calculated warping functions
- **q** – original training SRSFs
- **b** – basis coefficients
- **SSE** – sum of squared error

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 29-Oct-2021

calc_model(*B=None*, *lam=0*, *df=20*, *max_itr=20*, *cores=- 1*, *smooth=False*)

This function identifies a regression model with phase-variability using elastic pca

Parameters

- **B** – optional matrix describing Basis elements
- **lam** – regularization parameter (default 0)
- **df** – number of degrees of freedom B-spline (default 20)
- **max_itr** – maximum number of iterations (default 20)

- **cores** – number of cores for parallel processing (default all)

`predict`(*newdata=None*)

This function performs prediction on regression model on new data if available or current stored data in object Usage: obj.predict()

obj.predict(newdata)

Parameters

- **newdata** (*dict*) – dict containing new data for prediction (needs the keys below, if None predicts on training data)
- **f** – (M,N) matrix of functions
- **time** – vector of time points
- **y** – truth if available
- **smooth** – smooth data if needed
- **sparam** – number of times to run filter

`regression.logistic_warp`(*beta*, *time*, *q*, *y*)

calculates optimal warping for function logistic regression

Parameters

- **beta** – numpy ndarray of shape (M,N) of N functions with M samples
- **time** – vector of size N describing the sample points
- **q** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy ndarray of shape (1,N) responses

Return type
numpy array

Return gamma
warping function

`regression.logit_gradient`(*b*, *X*, *y*)

calculates gradient of the logistic loss

Parameters

- **b** – numpy ndarray of shape (M,N) of N functions with M samples
- **X** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy ndarray of shape (1,N) responses

Return type
numpy array

Return grad
gradient of logistic loss

`regression.logit_hessian`(*s*, *b*, *X*, *y*)

calculates hessian of the logistic loss

Parameters

- **s** – numpy ndarray of shape (M,N) of N functions with M samples

- **b** – numpy ndarray of shape (M,N) of N functions with M samples
- **X** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy ndarray of shape (1,N) responses

Return type
numpy array

Return out
hessian of logistic loss

`regression.logit_loss(b, X, y)`

logistic loss function, returns Sum{-log(phi(t))}

Parameters

- **b** – numpy ndarray of shape (M,N) of N functions with M samples
- **X** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy ndarray of shape (1,N) of N responses

Return type
numpy array

Return out
loss value

`regression.mlogit_gradient(b, X, Y)`

calculates gradient of the multinomial logistic loss

Parameters

- **b** – numpy ndarray of shape (M,N) of N functions with M samples
- **X** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy ndarray of shape (1,N) responses

Return type
numpy array

Return grad
gradient

`regression.mlogit_loss(b, X, Y)`

calculates multinomial logistic loss (negative log-likelihood)

Parameters

- **b** – numpy ndarray of shape (M,N) of N functions with M samples
- **X** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy ndarray of shape (1,N) responses

Return type
numpy array

Return nll
negative log-likelihood

regression.**mlogit_warp_grad**(*alpha*, *beta*, *time*, *q*, *y*, *max_itr=8000*, *tol=1e-10*, *delta=0.008*, *display=0*)

calculates optimal warping for functional multinomial logistic regression

Parameters

- **alpha** – scalar
- **beta** – numpy ndarray of shape (M,N) of N functions with M samples
- **time** – vector of size M describing the sample points
- **q** – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy ndarray of shape (1,N) responses
- **max_itr** – maximum number of iterations (Default=8000)
- **tol** – stopping tolerance (Default=1e-10)
- **delta** – gradient step size (Default=0.008)
- **display** – display iterations (Default=0)

Return type
tuple of numpy array

Return gam_old
warping function

regression.**phi**(*t*)

calculates logistic function, returns 1 / (1 + exp(-t))

Parameters
t – scalar

Return type
numpy array

Return out
return value

regression.**regression_warp**(*beta*, *time*, *q*, *y*, *alpha*)

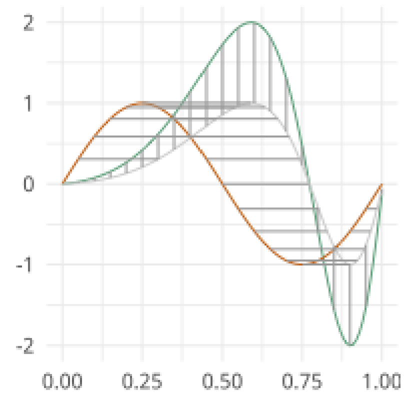
calculates optimal warping for function linear regression

Parameters

- **beta** – numpy ndarray of shape (M,N) of M functions with N samples
- **time** – vector of size N describing the sample points
- **q** – numpy ndarray of shape (M,N) of M functions with N samples
- **y** – numpy ndarray of shape (1,N) of M functions with N samples responses
- **alpha** – numpy scalar

Return type
numpy array

Return gamma_new
warping function


2
1
0
-1
-2
0.00
0.25
0.50
0.75
1.00

CHAPTER

SEVEN

ELASTIC PRINCIPAL COMPONENT REGRESSION

Warping Invariant PCR Regression using SRSF

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

class pcr_regression.**elastic_lpcr_regression**(*f*, *y*, *time*)

This class provides elastic logistic pcr regression for functional data using the SRVF framework accounting for warping

Usage: obj = elastic_lpcr_regression(f,y,time)

Parameters

- **f** – (M,N) % matrix defining N functions of M samples
- **y** – response vector of length N (-1/1)
- **warp_data** – fdawarp object of alignment
- **pca** – class dependent on fPCA method used object of fPCA

:param information :param alpha: intercept :param b: coefficient vector :param Loss: logistic loss :param PC: probability of classification :param ylabels: predicted labels

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 18-Mar-2018

calc_model(*pca_method='combined'*, *no=5*, *smooth_data=False*, *sparam=25*, *parallel=False*)

This function identifies a logistic regression model with phase-variability using elastic pca

Parameters

- **pca_method** – string specifing pca method (options = "combined", "vert", or "horiz", default = "combined")
- **no** – scalar specify number of principal components (default=5)
- **smooth_data** – smooth data using box filter (default = F)
- **sparam** – number of times to apply box filter (default = 25)
- **parallel** – calculate in parallel (default = F)

predict(*newdata=None*)

This function performs prediction on regression model on new data if available or current stored data in object Usage: obj.predict()

obj.predict(newdata)

Parameters

- **newdata** (*dict*) – dict containing new data for prediction (needs the keys below, if None predicts on training data)
- **f** – (M,N) matrix of functions
- **time** – vector of time points
- **y** – truth if available
- **smooth** – smooth data if needed
- **sparam** – number of times to run filter

class pcr_regression.**elastic_mlpcr_regression**(*f*, *y*, *time*)

This class provides elastic multinomial logistic pcr regression for functional data using the SRVF framework accounting for warping

Usage: obj = elastic_mlpcr_regression(f,y,time)

Parameters

- **f** – (M,N) % matrix defining N functions of M samples
- **y** – response vector of length N
- **Y** – coded label matrix
- **warp_data** – fdawarp object of alignment
- **pca** – class dependent on fPCA method used object of fPCA

:param information :param alpha: intercept :param b: coefficient vector :param Loss: logistic loss :param PC: probability of classification :param ylabels: predicted labels

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 18-Mar-2018

calc_model(*pca_method='combined'*, *no=5*, *smooth_data=False*, *sparam=25*, *parallel=False*)

This function identifies a logistic regression model with phase-variability using elastic pca

Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy array of N responses
- **time** (*np.ndarray*) – vector of size M describing the sample points
- **pca_method** – string specifing pca method (options = "combined", "vert", or "horiz", default = "combined")
- **no** – scalar specify number of principal components (default=5)
- **smooth_data** – smooth data using box filter (default = F)
- **sparam** – number of times to apply box filter (default = 25)
- **parallel** – run model in parallel (default = F)

predict(*newdata=None*)

This function performs prediction on regression model on new data if available or current stored data in object Usage: obj.predict()

obj.predict(newdata)

Parameters

- **newdata** (*dict*) – dict containing new data for prediction (needs the keys below, if None predicts on training data)
- **f** – (M,N) matrix of functions
- **time** – vector of time points
- **y** – truth if available
- **smooth** – smooth data if needed
- **sparam** – number of times to run filter

class pcr_regression.**elastic_pcr_regression**(*f*, *y*, *time*)

This class provides elastic pcr regression for functional data using the SRVF framework accounting for warping

Usage: obj = elastic_pcr_regression(f,y,time)

Parameters

- **f** – (M,N) % matrix defining N functions of M samples
- **y** – response vector of length N
- **warp_data** – fdawarp object of alignment
- **pca** – class dependent on fPCA method used object of fPCA
- **alpha** – intercept
- **b** – coefficient vector
- **SSE** – sum of squared errors

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 18-Mar-2018

calc_model(*pca_method='combined'*, *no=5*, *smooth_data=False*, *sparam=25*, *parallel=False*, *C=None*)

This function identifies a regression model with phase-variability using elastic pca

Parameters

- **pca_method** – string specifing pca method (options = "combined", "vert", or "horiz", default = "combined")
- **no** – scalar specify number of principal components (default=5)
- **smooth_data** – smooth data using box filter (default = F)
- **sparam** – number of times to apply box filter (default = 25)
- **parallel** – run in parallel (default = F)
- **C** – scale balance parameter for combined method (default = None)

predict(*newdata=None*)

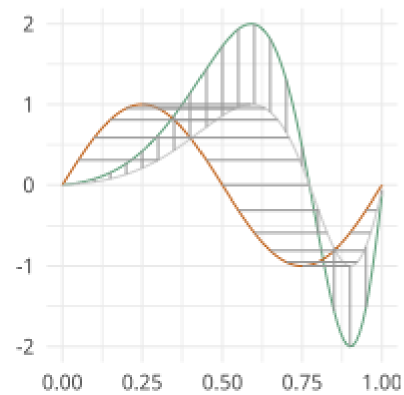
This function performs prediction on regression model on new data if available or current stored data in object Usage: obj.predict()

obj.predict(newdata)

Parameters

- **newdata** (*dict*) – dict containing new data for prediction (needs the keys below, if None predicts on training data)
- **f** – (M,N) matrix of functions

- **time** – vector of time points
- **y** – truth if available
- **smooth** – smooth data if needed
- **sparam** – number of times to run filter



CHAPTER

EIGHT

ELASTIC GLM REGRESSION

Warping Invariant GML Regression using SRSF

moduleauthor:: Derek Tucker <jdtuck@sandia.gov>

class elastic_glm_regression.**elastic_glm_regression**(*f*, *y*, *time*)

This class provides elastic glm regression for functional data using the SRVF framework accounting for warping

Usage: obj = elastic_glm_regression(f,y,time)

Parameters

- **f** – (M,N) % matrix defining N functions of M samples
- **y** – response vector of length N
- **time** – time vector of length M
- **alpha** – intercept
- **b** – coefficient vector
- **B** – basis matrix
- **lambda** – regularization parameter
- **SSE** – sum of squared errors

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 18-Mar-2018

calc_model(*link='linear'*, *B=None*, *lam=0*, *df=20*, *max_itr=20*, *smooth_data=False*, *sparam=25*, *parallel=False*)

This function identifies a regression model with phase-variability using elastic pca

Parameters

- **link** – string of link function ('linear', 'quadratic', 'cubic')
- **B** – optional matrix describing Basis elements
- **lam** – regularization parameter (default 0)
- **df** – number of degrees of freedom B-spline (default 20)
- **max_itr** – maximum number of iterations (default 20)
- **smooth_data** – smooth data using box filter (default = F)
- **sparam** – number of times to apply box filter (default = 25)
- **parallel** – run in parallel (default = F)

predict(*newdata=None*, *parallel=True*)

This function performs prediction on regression model on new data if available or current stored data in object Usage: obj.predict()

obj.predict(newdata)

Parameters

- **newdata** (*dict*) – dict containing new data for prediction (needs the keys below, if None predicts on training data)
- **f** – (M,N) matrix of functions
- **time** – vector of time points
- **y** – truth if available
- **smooth** – smooth data if needed
- **sparam** – number of times to run filter

elastic_glm_regression.**rand**(*d0*, *d1*, *...*, *dn*)

Random values in a given shape.

Note: This is a convenience function for users porting code from Matlab, and wraps *random_sample*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

d0, d1, …, dn

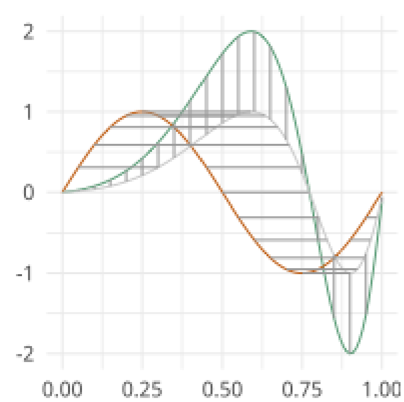
[int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

out

[ndarray, shape `(d0, d1, ..., dn)`] Random values.

random

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```



CHAPTER

NINE

ELASTIC FUNCTIONAL TOLERANCE BOUNDS

Functional Tolerance Bounds using SRSF

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

`tolerance.bootTB`(*f*, *time*, *a=0.05*, *p=0.99*, *B=500*, *no=5*, *parallel=True*)

This function computes tolerance bounds for functional data containing phase and amplitude variation using bootstrap sampling

Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **time** (*np.ndarray*) – vector of size M describing the sample points
- **a** – confidence level of tolerance bound (default = 0.05)
- **p** – coverage level of tolerance bound (default = 0.99)
- **B** – number of bootstrap samples (default = 500)
- **no** – number of principal components (default = 5)
- **parallel** – enable parallel processing (default = T)

Return type
tuple of boxplot objects

Return amp
amplitude tolerance bounds

Rtype out_med
ampbox object

Return ph
phase tolerance bounds

Rtype out_med
phbox object

Return out_med
alignment results

Rtype out_med
fdawarp object

`tolerance.mvtol_region`(*x*, *alpha*, *P*, *B*)

Computes tolerance factor for multivariate normal

Krishnamoorthy, K. and Mondal, S. (2006), Improved Tolerance Factors for Multivariate Normal Distributions, Communications in Statistics - Simulation and Computation, 35, 461–478.

Parameters

- **x** – (M,N) matrix defining N variables of M samples
- **alpha** – confidence level
- **P** – coverage level
- **B** – number of bootstrap samples

Return type
double

Return tol
tolerance factor

`tolerance.pcaTB(f, time, a=0.5, p=0.99, no=5, parallel=True)`

This function computes tolerance bounds for functional data containing phase and amplitude variation using fPCA

Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **time** (*np.ndarray*) – vector of size M describing the sample points
- **a** – confidence level of tolerance bound (default = 0.05)
- **p** – coverage level of tolerance bound (default = 0.99)
- **no** – number of principal components (default = 5)
- **parallel** – enable parallel processing (default = T)

Return type
tuple of boxplot objects

Return warp
alignment data from time_warping

Return pca
functional pca from jointFPCA

Return tol
tolerance factor

`tolerance.rwishart(df, p)`

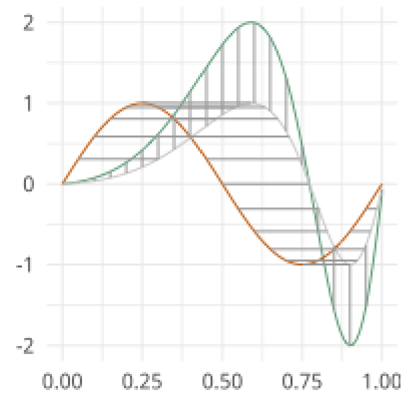
Computes a random wishart matrix

Parameters

- **df** – degree of freedom
- **p** – number of dimensions

Return type
double

Return R
matrix


2
1
0
-1
-2
0.00
0.25
0.50
0.75
1.00

CHAPTER

TEN

ELASTIC FUNCTIONAL CLUSTERING

Elastic Functional Clustering

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

`kmeans.kmeans_align`(*f*, *time*, *K*, *seeds=None*, *lam=0*, *showplot=True*, *smooth_data=False*, *parallel=False*, *alignment=True*, *omethod='DP2'*, *MaxItr=50*, *thresh=0.01*)

This function clusters functions and aligns using the elastic square-root slope (srsf) framework.

Parameters

- `f` – numpy ndarray of shape (M,N) of N functions with M samples
- `time` – vector of size M describing the sample points

:param K number of clusters :param seeds indexes of cluster center functions (default = None) :param lam controls the elasticity (default = 0) :param showplot shows plots of functions (default = T) :param smooth_data smooth data using box filter (default = F) :param parallel enable parallel mode using code{link{joblib}} and

code{doParallel} package (default=F)

:param alignment whether to perform alignment (default = T) :param omethod optimization method (DP,DP2,RBFGS) :param MaxItr maximum number of iterations :param thresh cost function threshold :type f: np.ndarray :type time: np.ndarray

Return type
dictionary

Return fn
aligned functions - matrix (N x M) of M functions with N samples which is a list for each cluster

Return qn
aligned SRSFs - similar structure to fn

Return q0
original SRSFs

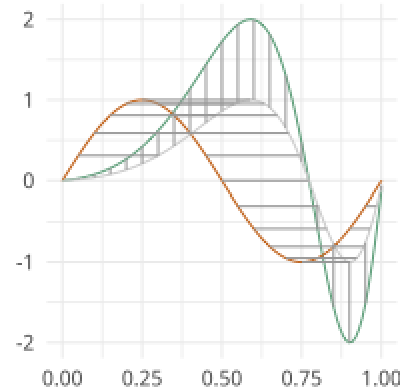
Return labels
cluster labels

Return templates
cluster center functions

Return templates_q
cluster center SRSFs

Return gam
warping functions - similar structure to fn

Return qun
Cost Function



CHAPTER

ELEVEN

ELASTIC IMAGE WARPING

image warping using SRVF framework

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

`image.reparam_image`(*It*, *Im*, *gam=None*, *b=None*, *stepsize=0.0001*, *itermax=20*)

This function warps an image to another using SRVF framework

Parameters

- **Im** – numpy ndarray of shape (N,N) representing a NxN image
- **Im** – numpy ndarray of shape (N,N) representing a NxN image
- **gam** – numpy ndarray of shape (N,N) representing an initial warping function
- **b** – numpy ndarray representing basis matrix

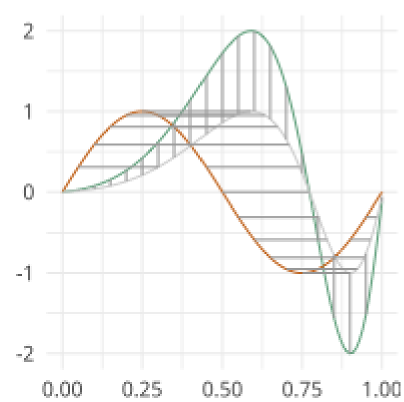
Return type
numpy ndarray

Return gamnew
diffeomorphism

Return Inew
warped image

Return H
energy

Return stepsize
final stepsize



CHAPTER

TWELVE

CURVE REGISTRATION

statistic calculation for SRVF (curves) open and closed using Karcher Mean and Variance

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

class curve_stats.fdacurve(*beta*, *mode='O'*, *N=200*, *scale=False*)

This class provides alignment methods for open and closed curves using the SRVF framework

Usage: obj = fdacurve(beta, mode, N, scale) :param beta: numpy ndarray of shape (n, M, N) describing N curves in R^M :param mode: Open ('O') or closed curve ('C') (default 'O') :param N: resample curve to N points :param scale: scale curve to length 1 (true/false) :param q: (n,T,K) matrix defining n dimensional srvf on T samples with K srvfs :param betan: aligned curves :param qn: aligned srvfs :param basis: calculated basis :param beta_mean: karcher mean curve :param q_mean: karcher mean srvf :param gams: warping functions :param v: shooting vectors :param C: karcher covariance :param s: pca singular values :param U: pca singular vectors :param coef: pca coefficients :param pca principal directions :param qun: cost function :param samples: random samples :param gamr: random warping functions :param cent: center :param scale: scale :param len: length of curve :param len_q: length of srvf :param mean_scale mean length :param mean_scale_q mean length srvf :param E: energy

Author : J. D. Tucker (JDT) <jdtuck AT sandia.gov> Date : 26-Aug-2020

karcher_cov()

This calculates the mean of a set of curves

karcher_mean(*rotation=True*, *parallel=False*, *cores=- 1*, *method='DP'*)

This calculates the mean of a set of curves :param rotation: compute optimal rotation (default = T) :param parallel: run in parallel (default = F) :param cores: number of cores for parallel (default = -1 (all)) :param method: method to apply optimization (default="DP") options are "DP" or "RBFGS"

plot()

plot curve mean results

sample_shapes(*no=3*, *numSamp=10*)

Computes sample shapes from mean and covariance

Parameters

- **no** – number of direction (default 3)
- **numSamp** – number of samples (default 10)

shape_pca(*no=10*)

Computes principal direction of variation specified by no. N is Number of shapes away from mean. Creates 2*N+1 shape sequence

Parameters

no – number of direction (default 3)

srvf_align(*rotation=True*, *parallel=False*, *cores=- 1*, *method='DP'*)

This aligns a set of curves to the mean and computes mean if not computed :param rotation: compute optimal rotation (default = T) :param parallel: run in parallel (default = F) :param cores: number of cores for parallel (default = -1 (all)) :param method: method to apply optimization (default="DP") options are "DP" or "RBFGS"

curve_stats.**randn**(*d0*, *d1*, *...*, *dn*)

Return a sample (or samples) from the "standard normal" distribution.

Note: This is a convenience function for users porting code from Matlab, and wraps *standard_normal*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Note: New code should use the *~numpy.random.Generator.standard_normal* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

If positive int_like arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

d0, d1, ..., dn
[int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Z
[ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

standard_normal : Similar, but takes a tuple as its argument. normal : Also accepts mu and sigma arguments. random.Generator.standard_normal: which should be used for new code.

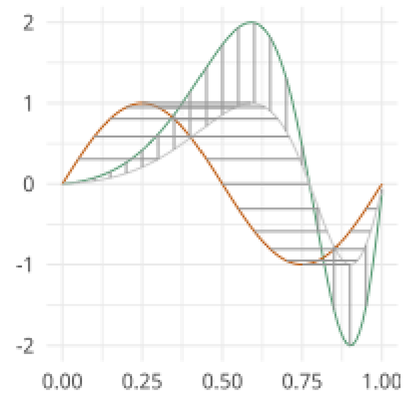
For random samples from the normal distribution with mean `mu` and standard deviation `sigma`, use:

```
sigma * np.random.randn(...) + mu
```

```
>>> np.random.randn()
2.1923875335537315  # random
```

Two-by-four array of samples from the normal distribution with mean 3 and standard deviation 2.5:

```
>>> 3 + 2.5 * np.random.randn(2, 4)
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```


2
1
0
-1
-2
0.00
0.25
0.50
0.75
1.00

CHAPTER

THIRTEEN

SRVF GEODESIC COMPUTATION

geodesic calculation for SRVF (curves) open and closed

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

`geodesic.back_parallel_transport`(*u1*, *alpha*, *basis*, *T=100*, *k=5*)

backwards parallel translates q1 and q2 along manifold

Parameters

- **u1** – numpy ndarray of shape (2,M) of M samples
- **alpha** – numpy ndarray of shape (2,M) of M samples
- **basis** – list numpy ndarray of shape (2,M) of M samples
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy ndarray

Return utilde
translated vector

`geodesic.calc_alphadot`(*alpha*, *basis*, *T=100*, *k=5*)

calculates derivative along the path alpha

Parameters

- **alpha** – numpy ndarray of shape (2,M) of M samples
- **basis** – list of numpy ndarray of shape (2,M) of M samples
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy ndarray

Return alphadot
derivative of alpha

`geodesic.calculate_energy`(*alphadot*, *T=100*, *k=5*)

calculates energy along path

Parameters

- **alphadot** – numpy ndarray of shape (2,M) of M samples

- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy scalar

Return E
energy

`geodesic.`**`calculate_gradE`**(*u*, *utilde*, *T=100*, *k=5*)

calculates gradient of energy along path

Parameters

- **u** – numpy ndarray of shape (2,M) of M samples
- **utilde** – numpy ndarray of shape (2,M) of M samples
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy scalar

Return gradE
gradient of energy

Return normgradE
norm of gradient of energy

`geodesic.`**`cov_integral`**(*alpha*, *alphadot*, *basis*, *T=100*, *k=5*)

Calculates covariance along path alpha

Parameters

- **alpha** – numpy ndarray of shape (2,M) of M samples (first curve)
- **alphadot** – numpy ndarray of shape (2,M) of M samples
- **basis** – list numpy ndarray of shape (2,M) of M samples
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy ndarray

Return u
covariance

`geodesic.`**`find_basis_normal_path`**(*alpha*, *k=5*)

computes orthonormalized basis vectors to the normal space at each of the k points (q-functions) of the path alpha

Parameters

- **alpha** – numpy ndarray of shape (2,M) of M samples (path)
- **k** – number of samples along path (Default = 5)

Return type
numpy ndarray

Return basis
basis vectors along the path

geodesic.**geod_dist_path_strt**(*beta*, *k=5*)

calculate geodisc distance for path straightening

Parameters

- **beta** – numpy ndarray of shape (2,M) of M samples
- **k** – number of samples along path (Default = 5)

Return type
numpy scalar

Return dist
geodesic distance

geodesic.**geod_sphere**(*beta1*, *beta2*, *k=5*, *scale=False*, *rotation=True*, *center=True*)

This function calculates the geodesics between open curves beta1 and beta2 with k steps along path

Parameters

- **beta1** – numpy ndarray of shape (2,M) of M samples
- **beta2** – numpy ndarray of shape (2,M) of M samples
- **k** – number of samples along path (Default = 5)
- **scale** – include length (Default = False)
- **rotation** – include rotation (Default = True)
- **center** – center curves at origin (Default = True)

Return type
numpy ndarray

Return dist
geodesic distance

Return path
geodesic path

Return PsiQ
geodesic path in SRVF

geodesic.**init_path_geod**(*beta1*, *beta2*, *T=100*, *k=5*)

Initializes a path in $\mathcal{C}$. beta1, beta2 are already standardized curves. Creates a path from beta1 to beta2 in shape space, then projects to the closed shape manifold.

Parameters

- **beta1** – numpy ndarray of shape (2,M) of M samples (first curve)
- **beta2** – numpy ndarray of shape (2,M) of M samples (end curve)
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy ndarray

Return alpha
a path between two q-functions

Return beta
a path between two curves

Return O
rotation matrix

`geodesic.init_path_rand(beta1, beta_mid, beta2, T=100, k=5)`

Initializes a path in $\mathcal{C}$. beta1, beta_mid beta2 are already standardized curves. Creates a path from beta1 to beta_mid to beta2 in shape space, then projects to the closed shape manifold.

Parameters

- **beta1** – numpy ndarray of shape (2,M) of M samples (first curve)
- **betamid** – numpy ndarray of shape (2,M) of M samples (mid curve)
- **beta2** – numpy ndarray of shape (2,M) of M samples (end curve)
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy ndarray

Return alpha
a path between two q-functions

Return beta
a path between two curves

Return O
rotation matrix

`geodesic.path_straightening(beta1, beta2, betamid=None, init='rand', T=100, k=5)`

Perform path straightening to find geodesic between two shapes in either the space of closed curves or the space of affine standardized curves. This algorithm follows the steps outlined in section 4.6 of the manuscript.

Parameters

- **beta1** – numpy ndarray of shape (2,M) of M samples (first curve)
- **beta2** – numpy ndarray of shape (2,M) of M samples (end curve)
- **betamid** – numpy ndarray of shape (2,M) of M samples (mid curve Default = None, only needed for init "geod")
- **init** – initialize path geodesic or random (Default = "rand")
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy ndarray

Return dist
geodesic distance

Return path
geodesic path

Return pathsqnc
geodesic path sequence

Return E
energy

`geodesic.plot_geod(path)`

Plots the geodesic path as a sequence of curves

Parameters
path – numpy ndarray of shape (2,M,K) of M sample points of K samples along path

`geodesic.update_path(alpha, beta, gradE, delta, T=100, k=5)`

Update the path along the direction -gradE

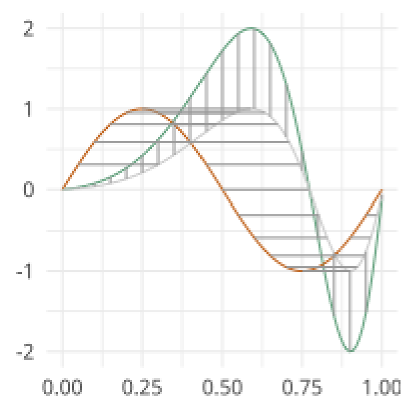
Parameters

- **alpha** – numpy ndarray of shape (2,M) of M samples
- **beta** – numpy ndarray of shape (2,M) of M samples
- **gradE** – numpy ndarray of shape (2,M) of M samples
- **delta** – gradient paramenter
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type
numpy scalar

Return alpha
updated path of srvfs

Return beta
updated path of curves



CHAPTER

FOURTEEN

UTILITY FUNCTIONS

Utility functions for SRSF Manipulations

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

utility_functions.**SqrtMean**(*gam*, *parallel=False*, *cores=- 1*)

calculates the srsf of warping functions with corresponding shooting vectors

Parameters

- **gam** – numpy ndarray of shape (M,N) of M warping functions with N samples
- **parallel** – run in parallel (default = F)
- **cores** – number of cores for parallel (default = -1 (all))

Return type
2 numpy ndarray and vector

Return mu
Karcher mean psi function

Return gam_mu
vector of dim N which is the Karcher mean warping function

Return psi
numpy ndarray of shape (M,N) of M SRSF of the warping functions

Return vec
numpy ndarray of shape (M,N) of M shooting vectors

utility_functions.**SqrtMeanInverse**(*gam*)

finds the inverse of the mean of the set of the diffeomorphisms gamma

Parameters
gam – numpy ndarray of shape (M,N) of N warping functions with M samples

Return type
vector

Return gamI
inverse of gam

utility_functions.**SqrtMedian**(*gam*)

calculates the median srsf of warping functions with corresponding shooting vectors

Parameters
gam – numpy ndarray of shape (M,N) of M warping functions with N samples

Return type
2 numpy ndarray and vector

Return gam_median
Karcher median warping function

Return psi_meidan
vector of dim N which is the Karcher median srsf function

Return psi
numpy ndarray of shape (M,N) of M SRSF of the warping functions

Return vec
numpy ndarray of shape (M,N) of M shooting vectors

`utility_functions.cumtrapzmid(x, y, c, mid)`

cumulative trapezoidal numerical integration taken from midpoint

Parameters

- **x** – vector of size N describing the time samples
- **y** – vector of size N describing the function
- **c** – midpointtic
- **mid** – midpiont location

Return type
vector

Return fa
cumulative integration

`utility_functions.diffop(n, binsize=1)`

Creates a second order differential operator

Parameters

- **n** – dimension
- **binsize** – dx (default = 1)

Return type
numpy ndarray

Return m
matrix describing differential operator

`utility_functions.elastic_depth(f, time, method='DP2', lam=0.0, parallel=True)`

calculates the elastic depth between functions in matrix f

Parameters

- **f** – matrix of size MxN (M time points for N functions)
- **time** – vector of size M describing the sample points
- **method** – method to apply optimization (default=”DP2”) options are “DP”,”DP2”,”RBFGS”
- **lam** – controls the elasticity (default = 0.0)

Return type
scalar

Return amp
amplitude depth

Return phase
phase depth

`utility_functions.elastic_distance(f1, f2, time, method='DP2', lam=0.0)`

" calculates the distances between function, where f1 is aligned to f2. In other words calculates the elastic distances

Parameters

- **f1** – vector of size N
- **f2** – vector of size N
- **time** – vector of size N describing the sample points
- **method** – method to apply optimization (default="DP2") options are "DP","DP2","RBFGS"
- **lam** – controls the elasticity (default = 0.0)

Return type
scalar

Return Dy
amplitude distance

Return Dx
phase distance

`utility_functions.f_K_fold(Nobs, K=5)`

generates sample indices for K-fold cross validation

:param Nobs number of observations :param K number of folds

Return type
numpy ndarray

Return train
train indexes (Nobs*(K-1)/K X K)

Return test
test indexes (Nobs*(1/K) X K)

`utility_functions.f_to_srsf(f, time, smooth=False)`

converts f to a square-root slope function (SRSF)

Parameters

- **f** – vector of size N samples
- **time** – vector of size N describing the sample points

Return type
vector

Return q
srsf of f

`utility_functions.geigen(Amat, Bmat, Cmat)`

generalized eigenvalue problem of the form

max tr L'AM / sqrt(tr L'BL tr M'CM) w.r.t. L and M

:param Amat numpy ndarray of shape (M,N) :param Bmat numpy ndarray of shape (M,N) :param Bmat numpy ndarray of shape (M,N)

Return type
numpy ndarray

Return values
eigenvalues

Return Lmat
left eigenvectors

Return Mmat
right eigenvectors

`utility_functions.gradient_spline(time, f, smooth=False)`

This function takes the gradient of f using b-spline smoothing

Parameters

- `time` – vector of size N describing the sample points
- `f` – numpy ndarray of shape (M,N) of M functions with N samples
- `smooth` – smooth data (default = F)

Return type
tuple of numpy ndarray

Return f0
smoothed functions functions

Return g
first derivative of each function

Return g2
second derivative of each function

`utility_functions.innerprod_q(time, q1, q2)`

calculates the innerproduct between two srsfs

:param time vector descrbing time samples :param q1 vector of srsf 1 :param q2 vector of srsf 2

Return type
scalar

Return val
inner product value

`utility_functions.invertGamma(gam)`

finds the inverse of the diffeomorphism gamma

Parameters
`gam` – vector describing the warping function

Return type
vector

Return gamI
inverse of gam

`utility_functions.optimum_reparam(q1, time, q2, method='DP2', lam=0.0, penalty='roughness', grid_dim=7)`

calculates the warping to align srsf q2 to q1

Parameters

- **q1** – vector of size N or array of NxM samples of first SRSF
- **time** – vector of size N describing the sample points
- **q2** – vector of size N or array of NxM samples samples of second SRSF
- **method** – method to apply optimization (default="DP2") options are "DP","DP2","RBFGS"
- **lam** – controls the amount of elasticity (default = 0.0)
- **penalty** – penalty type (default="roughness") options are "roughness", "l2gam", "l2psi", "geodesic". Only roughness implemented in all methods. To use others method needs to be "RBFGS"
- **grid_dim** – size of the grid, for the DP2 method only (default = 7)

Return type
vector

Return gam
describing the warping function used to align q2 with q1

`utility_functions.`**`optimum_reparam_pair`**`(`*q*, *time*, *q1*, *q2*, *lam=0.0*`)`
calculates the warping to align srsf pair q1 and q2 to q

Parameters

- **q** – vector of size N or array of NxM samples of first SRSF
- **time** – vector of size N describing the sample points
- **q1** – vector of size N or array of NxM samples samples of second SRSF
- **q2** – vector of size N or array of NxM samples samples of second SRSF
- **lam** – controls the amount of elasticity (default = 0.0)

Return type
vector

Return gam
describing the warping function used to align q2 with q1

`utility_functions.`**`outlier_detection`**`(`*q*, *time*, *mq*, *k=1.5*`)`
calculates outlier's using geodesic distances of the SRSFs from the median

Parameters

- **q** – numpy ndarray of N x M of M SRS functions with N samples
- **time** – vector of size N describing the sample points
- **mq** – median calculated using `time_warping.srsf_align()`
- **k** – cutoff threshold (default = 1.5)

Returns
q_outlier: outlier functions

`utility_functions.`**`randomGamma`**`(`*gam*, *num*`)`
generates random warping functions

Parameters

- **gam** – numpy ndarray of N x M of M of warping functions

- **num** – number of random functions

Returns
rgam: random warping functions

utility_functions.**resamplefunction**(*x*, *n*)

resample function using n points

Parameters

- **x** – functions
- **n** – number of points

Return type
numpy array

Return xn
resampled function

utility_functions.**rgam**(*N*, *sigma*, *num*, *mu_gam=None*)

Generates random warping functions

Parameters

- **N** – length of warping function
- **sigma** – variance of warping functions
- **num** – number of warping functions

:param mu_gam mean warping function (default identity) :return: gam: numpy ndarray of warping functions

utility_functions.**smooth_data**(*f*, *sparam=1*)

This function smooths a collection of functions using a box filter

Parameters

- **f** – numpy ndarray of shape (M,N) of M functions with N samples
- **sparam** – Number of times to run box filter (default = 25)

Return type
numpy ndarray

Return f
smoothed functions functions

utility_functions.**srsf_to_f**(*q*, *time*, *f0=0.0*)

converts q (srsf) to a function

Parameters

- **q** – vector of size N samples of srsf
- **time** – vector of size N describing time sample points
- **f0** – initial value

Return type
vector

Return f
function

utility_functions.**update_progress**(*progress*)

This function creates a progress bar

Parameters
`progress` – fraction of progress

utility_functions.**warp_f_gamma**(*time*, *f*, *gam*)

warps a function f by gam

:param time vector describing time samples :param q vector describing srsf :param gam vector describing warping function

Return type
numpy ndarray

Return f_temp
warped srsf

utility_functions.**warp_q_gamma**(*time*, *q*, *gam*)

warps a srsf q by gam

:param time vector describing time samples :param q vector describing srsf :param gam vector describing warping function

Return type
numpy ndarray

Return q_temp
warped srsf

utility_functions.**zero_crossing**(*Y*, *q*, *bt*, *time*, *y_max*, *y_min*, *gmax*, *gmin*)

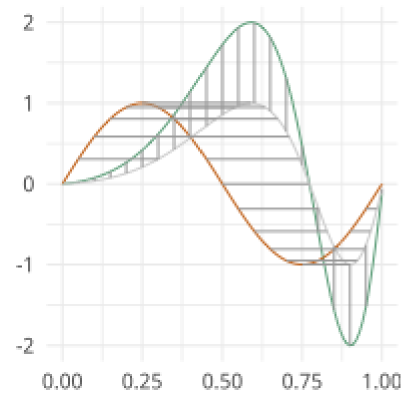
finds zero-crossing of optimal gamma, gam = s*gmax + (1-s)*gmin from elastic regression model

Parameters

- **Y** – response
- **q** – predicitve function
- **bt** – basis function
- **time** – time samples
- **y_max** – maximum repsonse for warping function gmax
- **y_min** – minimum response for warping function gmin
- **gmax** – max warping function
- **gmin** – min warping fucntion

Return type
numpy array

Return gamma
optimal warping function


2
1
0
-1
-2
0.00
0.25
0.50
0.75
1.00

CHAPTER

FIFTEEN

CURVE FUNCTIONS

functions for SRVF curve manipulations

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

`curve_functions.`**`Basis_Normal_A`**(*q*)

Find Normal Basis

Parameters
`q` – numpy ndarray (n,T) defining T points on n dimensional SRVF

:rtype list :return delg: basis

`curve_functions.`**`calc_j`**(*basis*)

Calculates Jacobian matrix from normal basis

Parameters
`basis` – list of numpy ndarray of shape (2,M) of M samples basis

Return type
numpy ndarray

Return j
Jacobian

`curve_functions.`**`calculate_variance`**(*beta*)

This function calculates variance of curve beta

Parameters
`beta` – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return variance
variance

`curve_functions.`**`calculatecentroid`**(*beta*)

This function calculates centroid of a parameterized curve

Parameters
`beta` – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return centroid
center coordinates

`curve_functions.`**`curve_to_q`**(*beta*, *mode='O'*)

This function converts curve beta to srvf q

Parameters

- **beta** – numpy ndarray of shape (2,M) of M samples
- **mode** – Open ('O') or closed curve ('C') (default 'O')

Return type
numpy ndarray

Return q
srvf of curve

Return lenb
length of curve

Return lenq
length of srvf

`curve_functions.`**`curve_zero_crossing`**(*Y*, *q*, *bt*, *y_max*, *y_min*, *gmax*, *gmin*)

finds zero-crossing of optimal gamma, gam = s*gmax + (1-s)*gmin from elastic curve regression model

Parameters

- **Y** – response
- **beta** – predicitve function
- **bt** – basis function
- **y_max** – maximum repsonse for warping function gmax
- **y_min** – minimum response for warping function gmin
- **gmax** – max warping function
- **gmin** – min warping fucntion

Return type
numpy array

Return gamma
optimal warping function

Return O_hat
rotation matrix

`curve_functions.`**`elastic_distance_curve`**(*beta1*, *beta2*, *closed=0*, *rotation=True*, *scale=False*, *method='DP'*)

Calculates the two elastic distances between two curves :param beta1: numpy ndarray of shape (2,M) of M samples :param beta2: numpy ndarray of shape (2,M) of M samples :param closed: open (0) or closed (1) curve (default=0) :param rotation: compute optimal rotation (default=True) :param scale: include scale (default=False) :param method: method to apply optimization (default="DP") options are "DP" or "RBFGS"

Return type
tuple

Return dist
shape distance

Return dx
phase distance

curve_functions.**elastic_shooting**(*q1*, *v*, *mode=0*)

Calculates shooting vector from v to q1

Parameters

- **q1** – vector of srvf
- **v** – shooting vector
- **mode** – closed or open (1/0)

:rtype numpy ndarray :return q2n: vector of srvf

curve_functions.**elastic_shooting_vector**(*q1*, *q2*, *mode=0*)

Calculates shooting between two srvfs

Parameters

- **q1** – vector of srvf
- **q2** – vector of srvf
- **mode** – closed or open (1/0)

:rtype numpy ndarray :return v: shooting vector :return d: distance :return q2n: aligned srvf

curve_functions.**find_basis_normal**(*q*)

Finds the basis normal to the srvf

Parameters
q1 – numpy ndarray of shape (2,M) of M samples

Return type
list of numpy ndarray

Return basis
list containing basis vectors

curve_functions.**find_best_rotation**(*q1*, *q2*, *allow_reflection=False*, *only_xy=False*)

This function calculates the best rotation between two srvfs using procustes rigid alignment

Parameters

- **q1** – numpy ndarray of shape (2,M) of M samples
- **q2** – numpy ndarray of shape (2,M) of M samples
- **allow_reflection** – bool indicating if reflection is allowed (i.e. if the determinant of the optimal rotation can be -1)
- **only_xy** – bool indicating if rotation should only be allowed in the first two dimensions of the space

Return type
numpy ndarray

Return q2new
optimal rotated q2 to q1

Return R
rotation matrix

curve_functions.**find_rotation_and_seed_coord**(*beta1*, *beta2*, *closed=0*, *rotation=True*, *method='DP'*)

This function returns a candidate list of optimally oriented and registered (seed) shapes w.r.t. beta1

Parameters

- **beta1** – numpy ndarray of shape (2,M) of M samples
- **beta2** – numpy ndarray of shape (2,M) of M samples
- **closed** – Open (0) or Closed (1)
- **rotation** – find rotation (default=True)
- **method** – method to apply optimization (default=”DP”) options are “DP” or “RBFGS”

Return type
numpy ndarray

Return beta2new
optimal aligned beta2 to beta1

Return q2best
optimal aligned q2 to q1

Return Rbest
rotation matrix

Return gamIbest
warping function

`curve_functions.`**`find_rotation_and_seed_q`**(*q1*, *q2*, *closed=0*, *rotation=True*, *method='DP'*)

This function returns a candidate list of optimally oriented and registered (seed) srvs w.r.t. q1

Parameters

- **q1** – numpy ndarray of shape (2,M) of M samples
- **q2** – numpy ndarray of shape (2,M) of M samples
- **closed** – Open (0) or Closed (1)
- **rotation** – find rotation (default=True)
- **method** – method to apply optimization (default=”DP”) options are “DP” or “RBFGS”

Return type
numpy ndarray

Return q2best
optimal aligned q2 to q1

Return Rbest
rotation matrix

Return gamIbest
warping function

`curve_functions.`**`find_rotation_and_seed_unique`**(*q1*, *q2*, *closed=0*, *rotation=True*, *method='DP'*)

This function returns a candidate list of optimally oriented and registered (seed) shapes w.r.t. beta1

Parameters

- **beta1** – numpy ndarray of shape (2,M) of M samples
- **beta2** – numpy ndarray of shape (2,M) of M samples
- **closed** – Open (0) or Closed (1)
- **rotation** – find rotation (default=True)
- **method** – method to apply optimization (default=”DP”) options are “DP” or “RBFGS”

Return type
numpy ndarray

Return beta2new
optimal rotated beta2 to beta1

Return O
rotation matrix

Return tau
seed

`curve_functions.`**`gram_schmidt`**`(`*basis*`)`

Performs Gram Schmidt Orthogonlization of a basis_o

param basis
list of numpy ndarray of shape (2,M) of M samples

rtype
list of numpy ndarray

return basis_o
orthogonlized basis

`curve_functions.`**`group_action_by_gamma`**`(`*q*, *gamma*`)`

This function reparamerized srvf q by gamma

Parameters

- **f** – numpy ndarray of shape (2,M) of M samples
- **gamma** – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return qn
reparatermized srvf

`curve_functions.`**`group_action_by_gamma_coord`**`(`*f*, *gamma*`)`

This function reparamerized curve f by gamma

Parameters

- **f** – numpy ndarray of shape (2,M) of M samples
- **gamma** – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return fn
reparatermized curve

`curve_functions.`**`innerprod_q2`**`(`*q1*, *q2*`)`

This function calculates the inner product in srvf space

Parameters

- **q1** – numpy ndarray of shape (2,M) of M samples
- **q2** – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return val
inner product

curve_functions.**inverse_exp**(*q1*, *q2*, *beta2*)

Calculate the inverse exponential to obtain a shooting vector from q1 to q2 in shape space of open curves

Parameters

- **q1** – numpy ndarray of shape (2,M) of M samples
- **q2** – numpy ndarray of shape (2,M) of M samples
- **beta2** – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return v
shooting vectors

curve_functions.**inverse_exp_coord**(*beta1*, *beta2*, *closed=0*, *method='DP'*)

Calculate the inverse exponential to obtain a shooting vector from beta1 to beta2 in shape space of open curves

Parameters

- **beta1** – numpy ndarray of shape (2,M) of M samples
- **beta2** – numpy ndarray of shape (2,M) of M samples
- **closed** – open (0) or closed (1) curve
- **method** – method to apply optimization (default="DP") options are "DP" or "RBFGS"

Return type
numpy ndarray

Return v
shooting vectors

Return dist
distance

curve_functions.**optimum_reparam_curve**(*q1*, *q2*, *lam=0.0*, *method='DP'*)

calculates the warping to align srsf q2 to q1

Parameters

- **q1** – matrix of size nxN or array of NxM samples of first SRVF
- **time** – vector of size N describing the sample points
- **q2** – matrix of size nxN or array of NxM samples samples of second SRVF
- **lam** – controls the amount of elasticity (default = 0.0)
- **method** – method to apply optimization (default="DP") options are "DP" or "RBFGS"

Return type
vector

Return gam
describing the warping function used to align q2 with q1

`curve_functions.parallel_translate(w, q1, q2, basis, mode=0)`

parallel translates q1 and q2 along manifold

Parameters

- **w** – numpy ndarray of shape (2,M) of M samples
- **q1** – numpy ndarray of shape (2,M) of M samples
- **q2** – numpy ndarray of shape (2,M) of M samples
- **basis** – list of numpy ndarray of shape (2,M) of M samples
- **mode** – open 0 or closed curves 1 (default 0)

Return type
numpy ndarray

Return wbar
translated vector

`curve_functions.pre_proc_curve(beta, T=100)`

This function prepcoessed a curve beta to set of closed curves

Parameters

- **beta** – numpy ndarray of shape (2,M) of M samples
- **T** – number of samples (default = 100)

Return type
numpy ndarray

Return betanew
projected beta

Return qnew
projected srvf

Return A
alignment matrix (not used currently)

`curve_functions.project_curve(q)`

This function projects srvf q to set of close curves

Parameters
q – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return qproj
project srvf

`curve_functions.project_tangent(w, q, basis)`

projects srvf to tangent space w using basis

Parameters

- **w** – numpy ndarray of shape (2,M) of M samples
- **q** – numpy ndarray of shape (2,M) of M samples
- **basis** – list of numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return wproj
projected q

`curve_functions.psi(x, a, q)`

This function formats variance output

Parameters

- **x** – numpy ndarray of shape (2,M) of M samples curve
- **a** – numpy ndarray of shape (2,1) mean
- **q** – numpy ndarray of shape (2,M) of M samples srvf

Return type
numpy ndarray

Return psi1
variance

Return psi2
cross variance

Return psi3
curve end

Return psi4
curve end

`curve_functions.q_to_curve(q, scale=1)`

This function converts srvf to beta

Parameters

- **q** – numpy ndarray of shape (n,M) of M samples
- **scale** – scale of curve

Return type
numpy ndarray

Return beta
parameterized curve

`curve_functions.resamplecurve(x, N=100, time=None, mode='O')`

This function resamples a curve to have N samples

Parameters

- **x** – numpy ndarray of shape (2,M) of M samples
- **N** – Number of samples for new curve (default = 100)
- **time** – timing vector (Default=None)
- **mode** – Open ('O') or closed curve ('C') (default 'O')

Return type
numpy ndarray

Return xn
resampled curve

`curve_functions.scale_curve(beta)`

scales curve to length 1

Parameters
beta – numpy ndarray of shape (2,M) of M samples

Return type
numpy ndarray

Return beta_scaled
scaled curve

Return scale
scale factor used

`curve_functions.shift_f(f, tau)`

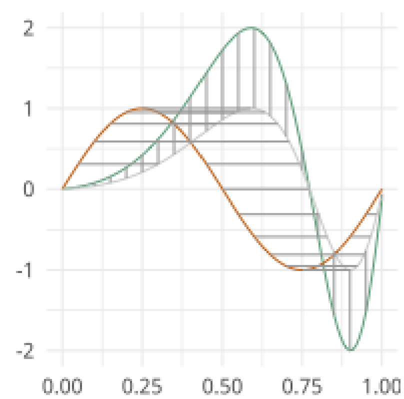
shifts a curve f by tau

Parameters

- **f** – numpy ndarray of shape (2,M) of M samples
- **tau** – scalar

Return type
numpy ndarray

Return fn
shifted curve



CHAPTER

SIXTEEN

UMAP EFDA METRICS

Distance metrics for functions and curves in R^n for use with UMAP (https://github.com/lmcinnes/umap)

moduleauthor:: J. Derek Tucker <jdtuck@sandia.gov>

`umap_metric.efda_distance(q1, q2, alpha=0)`

” calculates the distances between two curves, where q2 is aligned to q1. In other words calculates the elastic distances/ This metric is set up for use with UMAP or t-sne from scikit-learn

Parameters

- **q1** – vector of size N
- **q2** – vector of size N
- **alpha** – weight between phase and amplitude (default = 0, returns amplitude)

Return type
scalar

Return dist
amplitude distance

`umap_metric.efda_distance_curve(beta1, beta2, closed)`

” calculates the distances between two curves, where beta2 is aligned to beta1. In other words calculates the elastic distance. This metric is set up for use with UMAP or t-sne from scikit-learn

Parameters

- **beta1** – vector of size n*M
- **beta2** – vector of size n*M
- **closed** –

(0) if open curves and (1) if closed curves

Return type
scalar

Return dist
shape distance

CHAPTER SEVENTEEN

REFERENCES

Tucker, J. D. 2014, Functional Component Analysis and Regression using Elastic Methods. Ph.D. Thesis, Florida State University.

Robinson, D. T. 2012, Function Data Analysis and Partial Shape Matching in the Square Root Velocity Framework. Ph.D. Thesis, Florida State University.

Huang, W. 2014, Optimization Algorithms on Riemannian Manifolds with Applications. Ph.D. Thesis, Florida State University.

Srivastava, A., Wu, W., Kurtek, S., Klassen, E. and Marron, J. S. (2011). Registration of Functional Data Using Fisher-Rao Metric. arXiv:1103.3817v2 [math.ST].

Tucker, J. D., Wu, W. and Srivastava, A. (2013). Generative models for functional data using phase and amplitude separation. Computational Statistics and Data Analysis 61, 50-66.

J. D. Tucker, W. Wu, and A. Srivastava, "Phase-Amplitude Separation of Proteomics Data Using Extended Fisher-Rao Metric," Electronic Journal of Statistics, Vol 8, no. 2. pp 1724-1733, 2014.

J. D. Tucker, W. Wu, and A. Srivastava, "Analysis of signals under compositional noise With applications to SONAR data," IEEE Journal of Oceanic Engineering, Vol 29, no. 2. pp 318-330, Apr 2014.

Srivastava, A., Klassen, E., Joshi, S., Jermyn, I., (2011). Shape analysis of elastic curves in euclidean spaces. Pattern Analysis and Machine Intelligence, IEEE Transactions on 33 (7), 1415-1428.

S. Kurtek, A. Srivastava, and W. Wu. Signal estimation under random time-warpings and nonlinear signal alignment. In Proceedings of Neural Information Processing Systems (NIPS), 2011.

Wen Huang, Kyle A. Gallivan, Anuj Srivastava, Pierre-Antoine Absil. "Riemannian Optimization for Elastic Shape Analysis", Short version, The 21st International Symposium on Mathematical Theory of Networks and Systems (MTNS 2014).

Cheng, W., Dryden, I. L., and Huang, X. (2016). Bayesian registration of functions and curves. Bayesian Analysis, 11(2), 447-475.

W. Xie, S. Kurtek, K. Bharath, and Y. Sun, A geometric approach to visualization of variability in functional data, Journal of American Statistical Association 112 (2017), pp. 979-993.

Lu, Y., R. Herbei, and S. Kurtek, 2017: Bayesian registration of functions with a Gaussian process prior. Journal of Computational and Graphical Statistics, 26, no. 4, 894–904.

Lee, S. and S. Jung, 2017: Combined analysis of amplitude and phase variations in functional data. arXiv:1603.01775 [stat.ME], 1–21.

J. D. Tucker, J. R. Lewis, and A. Srivastava, "Elastic Functional Principal Component Regression," Statistical Analysis and Data Mining, vol. 12, no. 2, pp. 101-115, 2019.

J. D. Tucker, J. R. Lewis, C. King, and S. Kurtek, "A Geometric Approach for Computing Tolerance Bounds for Elastic Functional Data," Journal of Applied Statistics, 10.1080/02664763.2019.1645818, 2019.

T. Harris, J. D. Tucker, B. Li, and L. Shand, "Elastic depths for detecting shape anomalies in functional data," Technometrics, 10.1080/00401706.2020.1811156, 2020.

M. K. Ahn, J. D. Tucker, W. Wu, and A. Srivastava. "Regression Models Using Shapes of Functions as Predictors" Computational Statistics and Data Analysis, 10.1016/j.csda.2020.107017, 2020.

J. D. Tucker, L. Shand, and K. Chowdhary. "Multimodal Bayesian Registration of Noisy Functions using Hamiltonian Monte Carlo", Computational Statistics and Data Analysis, accepted, 2021.

X. Zhang, S. Kurtek, O. Chkrebtii, and J. D. Tucker, "Elastic k-means clustering of functional data for posterior exploration, with an application to inference on acute respiratory infection dynamics", arXiv:2011.12397 [stat.ME], 2020.

Q. Xie, S. Kurtek, E. Klassen, G. E. Christensen and A. Srivastava. Metric-based pairwise and multiple image registration. IEEE European Conference on Computer Vision (ECCV), September, 2014

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

G

I

K

L

M

O

P

Q

R

S

T

U

W

Z