

---

# **fdasrsf Documentation**

***Release 2.3.3***

**J. Derek Tucker**

**Oct 19, 2021**



## CONTENTS

<b>1</b>	<b>Functional Alignment</b>	<b>3</b>
<b>2</b>	<b>Functional Principal Component Analysis</b>	<b>11</b>
<b>3</b>	<b>Elastic Functional Boxplots</b>	<b>15</b>
<b>4</b>	<b>Functional Principal Least Squares</b>	<b>17</b>
<b>5</b>	<b>Elastic Regression</b>	<b>19</b>
<b>6</b>	<b>Elastic Principal Component Regression</b>	<b>25</b>
<b>7</b>	<b>Elastic GLM Regression</b>	<b>29</b>
<b>8</b>	<b>Elastic Functional Tolerance Bounds</b>	<b>31</b>
<b>9</b>	<b>Curve Registration</b>	<b>33</b>
<b>10</b>	<b>SRVF Geodesic Computation</b>	<b>35</b>
<b>11</b>	<b>Utility Functions</b>	<b>39</b>
<b>12</b>	<b>Curve Functions</b>	<b>45</b>
<b>13</b>	<b>UMAP EFDA Metrics</b>	<b>53</b>
<b>14</b>	<b>References</b>	<b>55</b>
<b>15</b>	<b>Indices and tables</b>	<b>57</b>
	<b>Python Module Index</b>	<b>59</b>
	<b>Index</b>	<b>61</b>



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.



## FUNCTIONAL ALIGNMENT

Group-wise function alignment using SRSF framework and Dynamic Programming

moduleauthor:: J. Derek Tucker <[jdtuck@sandia.gov](mailto:jdtuck@sandia.gov)>

`time_warping.align_fPCA(f, time, num_comp=3, showplot=True, smoothdata=False, cores=-1)`  
aligns a collection of functions while extracting principal components. The functions are aligned to the principal components

### 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
- **num\_comp** – number of fPCA components
- **showplot** – Shows plots of results using matplotlib (default = T)
- **smooth\_data** (*bool*) – Smooth the data using a box filter (default = F)
- **cores** – number of cores for parallel (default = -1 (all))

**Return type** tuple of numpy array

**Return fn** aligned functions - numpy ndarray of shape (M,N) of N functions with M samples

**Return qn** aligned srvfs - similar structure to fn

**Return q0** original srvf - similar structure to fn

**Return mqn** srvf mean or median - vector of length M

**Return gam** warping functions - similar structure to fn

**Return q\_pca** srsf principal directions

**Return f\_pca** functional principal directions

**Return latent** latent values

**Return coef** coefficients

**Return U** eigenvectors

**Return orig\_var** Original Variance of Functions

**Return amp\_var** Amplitude Variance

**Return phase\_var** Phase Variance

`time_warping.align_fPLS(f, g, time, comps=3, showplot=True, smoothdata=False, delta=0.01, max_itr=100)`

This function aligns a collection of functions while performing principal least squares

### Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **g** (*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
- **comps** – number of fPLS components
- **showplot** – Shows plots of results using matplotlib (default = T)
- **smooth\_data** (*bool*) – Smooth the data using a box filter (default = F)
- **delta** – gradient step size
- **max\_itr** – maximum number of iterations

**Return type** tuple of numpy array

**Return fn** aligned functions - numpy ndarray of shape (M,N) of N

functions with M samples :return gn: aligned functions - numpy ndarray of shape (M,N) of N functions with M samples :return qfn: aligned srvfs - similar structure to fn :return qgn: aligned srvfs - similar structure to fn :return qf0: original srvf - similar structure to fn :return qg0: original srvf - similar structure to fn :return gam: warping functions - similar structure to fn :return wqf: srsf principal weight functions :return wqg: srsf principal weight functions :return wf: srsf principal weight functions :return wg: srsf principal weight functions :return cost: cost function value

**class** time\_warping.**fdawarp**(*f, time*)

This class provides alignment methods for functional data using the SRVF framework

Usage: obj = fdawarp(f,t)

### Parameters

- **f** – (M,N): matrix defining N functions of M samples
- **time** – time vector of length M
- **fn** – aligned functions
- **qn** – aligned srvfs
- **q0** – initial srvfs
- **fmean** – function mean
- **mqn** – mean srvf
- **gam** – warping functions
- **psi** – srvf of warping functions
- **stats** – alignment statistics
- **qun** – cost function
- **lambda** – lambda
- **method** – optimization method
- **gamI** – inverse warping function
- **rsamps** – random samples
- **fs** – random aligned functions
- **gams** – random warping functions



- **ft** – random warped functions
- **qs** – random aligned srvfs
- **type** – alignment type
- **mcmc** – mcmc output if bayesian

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

**gauss\_model**(*n=1, sort\_samples=False*)

This function models the functional data using a Gaussian model extracted from the principal components of the srvfs

#### Parameters

- **n** (*integer*) – number of random samples
- **sort\_samples** (*bool*) – sort samples (default = T)

**joint\_gauss\_model**(*n=1, no=3*)

This function models the functional data using a joint Gaussian model extracted from the principal components of the srsfs

#### Parameters

- **n** (*integer*) – number of random samples
- **no** (*integer*) – number of principal components (default = 3)

**multiple\_align\_functions**(*mu, omethod='DP2', smoothdata=False, parallel=False, lam=0.0, cores=- 1, grid\_dim=7*)

This function aligns a collection of functions using the elastic square-root slope (srsf) framework.

**Usage:** `obj.multiple_align_functions(mu)` `obj.multiple_align_functions(lambda)`

`obj.multiple_align_functions(lambda, ...)`

#### Parameters

- **mu** – vector of function to align to
- **omethod** – optimization method (DP, DP2, RBFGS) (default = DP)
- **smoothdata** (*bool*) – Smooth the data using a box filter (default = F)
- **parallel** – run in parallel (default = F)
- **lam** (*double*) – controls the elasticity (default = 0)
- **cores** – number of cores for parallel (default = -1 (all))
- **grid\_dim** – size of the grid, for the DP2 method only (default = 7)

**plot()**

plot plot functional alignment results

Usage: `obj.plot()`

**srsf\_align**(*method='mean', omethod='DP2', smoothdata=False, MaxIter=20, parallel=False, lam=0.0, cores=- 1, grid\_dim=7*)

This function aligns a collection of functions using the elastic square-root slope (srsf) framework.

#### Parameters

- **method** – (string) warp calculate Karcher Mean or Median (options = “mean” or “median”) (default=“mean”)

- **omethod** – optimization method (DP, DP2, RBFGS) (default = DP2)
- **smoothdata** (*bool*) – Smooth the data using a box filter (default = F)
- **MaxItr** – Maximum number of iterations (default = 20)
- **parallel** – run in parallel (default = F)
- **lam** (*double*) – controls the elasticity (default = 0)
- **cores** – number of cores for parallel (default = -1 (all))
- **grid\_dim** – size of the grid, for the DP2 method only (default = 7)

Examples >>> import tables >>> fun=tables.open\_file("../Data/simu\_data.h5") >>> f = fun.root.f[:] >>> f = f.transpose() >>> time = fun.root.time[:] >>> obj = fs.fdwarp(f,time) >>> obj.srsf\_align()

**time\_warping.normal**(*loc=0.0, scale=1.0, size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently<sup>2</sup>, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution<sup>2</sup>.

---

**Note:** New code should use the **normal** method of a **default\_rng()** instance instead; please see the random-quick-start.

---

**loc** [float or array\_like of floats] Mean (“centre”) of the distribution.

**scale** [float or array\_like of floats] Standard deviation (spread or “width”) of the distribution. Must be non-negative.

**size** [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (**m**, **n**, **k**), then **m \* n \* k** samples are drawn. If size is **None** (default), a single value is returned if **loc** and **scale** are both scalars. Otherwise, **np.broadcast(loc, scale).size** samples are drawn.

**out** [ndarray or scalar] Drawn samples from the parameterized normal distribution.

**scipy.stats.norm** [probability density function, distribution or] cumulative density function, etc.

Generator.normal: which should be used for new code.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where  $\mu$  is the mean and  $\sigma$  the standard deviation. The square of the standard deviation,  $\sigma^2$ , is called the variance.

The function has its peak at the mean, and its “spread” increases with the standard deviation (the function reaches 0.607 times its maximum at  $x + \sigma$  and  $x - \sigma$ <sup>2</sup>). This implies that normal is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

---

<sup>2</sup> P. R. Peebles Jr., “Central Limit Theorem” in “Probability, Random Variables and Random Signal Principles”, 4th ed., 2001, pp. 51, 51, 125.

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s))
0.0 # may vary
```

```
>>> abs(sigma - np.std(s, ddof=1))
0.1 # may vary
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...          np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Two-by-four array of samples from  $N(3, 6.25)$ :

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

`time_warping.pairwise_align_bayes(f1i, f2i, time, mcmcopts=None)`

This function aligns two functions using Bayesian framework. It will align  $f_2$  to  $f_1$ . It is based on mapping warping functions to a hypersphere, and a subsequent exponential mapping to a tangent space. In the tangent space, the Z-mixture pCN algorithm is used to explore both local and global structure in the posterior distribution.

The Z-mixture pCN algorithm uses a mixture distribution for the proposal distribution, controlled by input parameter `zpcn`. The `zpcn$betas` must be between 0 and 1, and are the coefficients of the mixture components, with larger coefficients corresponding to larger shifts in parameter space. The `zpcn["probs"]` give the probability of each shift size.

**Usage:** `out = pairwise_align_bayes(f1i, f2i, time)` `out = pairwise_align_bayes(f1i, f2i, time, mcmcopts)`

#### Parameters

- **f1i** – vector defining  $M$  samples of function 1
- **f2i** – vector defining  $M$  samples of function 2
- **time** – time vector of length  $M$
- **mcmcopts** – dict of mcmc parameters

default mcmc options: `tmp = {"betas":np.array([0.5,0.5,0.005,0.0001]),"probs":np.array([0.1,0.1,0.7,0.1])}`  
`mcmcopts = {"iter":2*(10**4), "burnin":np.minimum(5*(10**3),2*(10**4)//2),`

`"alpha0":0.1, "beta0":0.1,"zpcn":tmp,"propvar":1, "initcoef":np.repeat(0,20), "npoints":200, "extrainfo":True}`

:rtype collection containing :return `f2_warped`: aligned  $f_2$  :return `gamma`: warping function :return `g_coef`: final `g_coef` :return `psi`: final `psi` :return `sigma1`: final `sigma`

if extrainfo :return accept: accept of psi samples :return betas\_ind :return logl: log likelihood :return gamma\_mat: posterior gammas :return gamma\_stats: posterior gamma stats :return xdist: phase distance posterior :return ydist: amplitude distance posterior)

`time_warping.pairwise_align_bayes_infHMC(y1i, y2i, time, mcmcopts=None)`

This function aligns two functions using Bayesian framework. It uses a hierarchical Bayesian framework assuming measurement error error. It will align f2 to f1. It is based on mapping warping functions to a hypersphere, and a subsequent exponential mapping to a tangent space. In the tangent space, the infHMC algorithm is used to explore both local and global structure in the posterior distribution.

**Usage:** `out = pairwise_align_bayes_infHMC(f1i, f2i, time)` `out = pairwise_align_bayes_infHMC(f1i, f2i, time, mcmcopts)`

#### Parameters

- **y1i** – vector defining M samples of function 1
- **y2i** – vector defining M samples of function 2
- **time** – time vector of length M
- **mcmcopts** – dict of mcmc parameters

default mcmc options: `mcmcopts = {"iter":1*(10**4), "nchains":4, "vpriorvar":1, "burnin":np.minimum(5*(10**3),2*(10**4)//2), "alpha0":0.1, "beta0":0.1, "alpha":1, "beta":1, "h":0.01, "L":4, "f1propvar":0.0001, "f2propvar":0.0001, "L1propvar":0.3, "L2propvar":0.3, "npoints":200, "thin":1, "sampfreq":1, "initcoef":np.repeat(0,20), "nbasis":10, "basis":'fourier', "extrainfo":True}`

Basis can be 'fourier' or 'legendre'

:rtype collection containing :return f2\_warped: aligned f2 :return gamma: warping function :return v\_coef: final v\_coef :return psi: final psi :return sigma1: final sigma

if extrainfo :return theta\_accept: accept of psi samples :return f2\_accept: accept of f2 samples :return SSE: SSE :return gamma\_mat: posterior gammas :return gamma\_stats: posterior gamma stats :return xdist: phase distance posterior :return ydist: amplitude distance posterior)

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.

`time_warping.pairwise_align_functions(f1, f2, time, omethod='DP2', lam=0, grid_dim=7)`

**This function aligns f2 to f1 using the elastic square-root slope (srsf) framework.**

**Usage:** `out = pairwise_align_functions(f1, f2, time)` `out = pairwise_align_functions(f1, f2, time, omethod, lam, grid_dim)`

#### Parameters

- **f1** – vector defining M samples of function 1
- **f2** – vector defining M samples of function 2
- **time** – time vector of length M
- **omethod** – optimization method (DP, DP2, RBFGS) (default = DP)
- **lam** – controls the elasticity (default = 0)
- **grid\_dim** – size of the grid, for the DP2 method only (default = 7)

:rtype list containing :return f2n: aligned f2 :return gam: warping function :return q2n: aligned q2 (srsf)  
`time_warping.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
```



## FUNCTIONAL PRINCIPAL COMPONENT ANALYSIS

Vertical and Horizontal Functional Principal Component Analysis using SRSF

moduleauthor:: J. Derek Tucker <[jdtuck@sandia.gov](mailto: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](mailto: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 geodesic 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 geodesic 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()`



## 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()

## FUNCTIONAL PRINCIPAL LEAST SQUARES

Partial Least Squares using SVD

moduleauthor:: J. Derek Tucker <[jdtuck@sandia.gov](mailto: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



## ELASTIC REGRESSION

Warping Invariant Regression using SRSF

moduleauthor:: J. Derek Tucker <[jdtuck@sandia.gov](mailto:jdtuck@sandia.gov)>

`regression.elastic_logistic(f, y, time, B=None, df=20, max_itr=20, cores=-1, smooth=False)`

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

### Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy array of labels (1/-1)
- **time** (*np.ndarray*) – vector of size M describing the sample points
- **B** – optional matrix describing Basis elements
- **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)

**Return type** tuple of numpy array

**Return alpha** alpha parameter of model

**Return beta** beta(t) of model

**Return fn** aligned functions - numpy ndarray of shape (M,N) of M

functions with N samples :return qn: aligned srvfs - similar structure to fn :return gamma: calculated warping functions :return q: original training SRSFs :return B: basis matrix :return b: basis coefficients :return Loss: logistic loss

`regression.elastic_mlogistic(f, y, time, B=None, df=20, max_itr=20, cores=-1, delta=0.01, parallel=True, smooth=False)`

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

### Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of N functions with M samples
- **y** – numpy array of labels {1,2,...,m} for m classes
- **time** (*np.ndarray*) – vector of size M describing the sample points
- **B** – optional matrix describing Basis elements
- **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)

**Return type** tuple of numpy array

**Return alpha** alpha parameter of model

**Return beta** beta(t) of model

**Return fn** aligned functions - numpy ndarray of shape (M,N) of N

functions with M samples :return qn: aligned srvfs - similar structure to fn :return gamma: calculated warping functions :return q: original training SRSFs :return B: basis matrix :return b: basis coefficients :return Loss: logistic loss

`regression.elastic_prediction(f, time, model, y=None, smooth=False)`

This function performs prediction from an elastic regression model with phase-variability

#### Parameters

- **f** – numpy ndarray of shape (M,N) of N functions with M samples
- **time** – vector of size M describing the sample points
- **model** – identified model from elastic\_regression
- **y** – truth, optional used to calculate SSE

**Return type** tuple of numpy array

**Return alpha** alpha parameter of model

**Return beta** beta(t) of model

**Return fn** aligned functions - numpy ndarray of shape (M,N) of N

functions with M samples :return qn: aligned srvfs - similar structure to fn :return gamma: calculated warping functions :return q: original training SRSFs :return B: basis matrix :return b: basis coefficients :return SSE: sum of squared error

`regression.elastic_regression(f, y, time, B=None, lam=0, df=20, max_itr=20, cores=-1, smooth=False)`

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

#### 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
- **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)

**Return type** tuple of numpy array

**Return alpha** alpha parameter of model

**Return beta** beta(t) of model

**Return fn** aligned functions - numpy ndarray of shape (M,N) of M



functions with N samples :return qn: aligned srvfs - similar structure to fn :return gamma: calculated warping functions :return q: original training SRSFs :return B: basis matrix :return b: basis coefficients :return SSE: sum of squared error

`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  $\text{Sum}\{-\log(\phi(t))\}$

**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 :param alpha: numpy scalar

**Return type** numpy array

**Return gamma\_new** warping function



## 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 specifying 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 :param

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 specifying 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 specifying 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



## 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

:param information :param alpha: intercept :param b: coefficient vector :param B: basis matrix :param lambda: regularization parameter :param 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
```

## ELASTIC FUNCTIONAL TOLERANCE BOUNDS

Functional Tolerance Bounds using SRSF

moduleauthor:: J. Derek Tucker <[jdtuck@sandia.gov](mailto:jdtuck@sandia.gov)>

`tolerance.bootTB(f, time, a=0.5, 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

## 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 `standard_normal` method of a `default_rng()` 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. `Generator.standard_normal`: which should be used for new code.

For random samples from  $N(\mu, \sigma^2)$ , use:

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

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

Two-by-four array of samples from  $N(3, 6.25)$ :

```
>>> 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
```

## SRVF GEODESIC COMPUTATION

geodesic calculation for SRVF (curves) open and closed

moduleauthor:: J. Derek Tucker <[jdtuck@sandia.gov](mailto: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 O** rotation matrix`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, 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 = NULL, only needed for init “rand”)
- **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.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 parameter
- **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

## UTILITY FUNCTIONS

Utility functions for SRSF Manipulations

moduleauthor:: J. Derek Tucker <[jdtuck@sandia.gov](mailto: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 M warping functions with N 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** – midpoint 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","RBFSGS"
- **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","RBFSGS"
- **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

$\max \text{tr } L'AM / \sqrt{\text{tr } L'BL \text{tr } M'CM}$  w.r.t. L and M

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

**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 describing 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, 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 of second SRSF
- **method** – method to apply optimization (default="DP2") options are "DP", "DP2", "RBFGS"
- **lam** – controls the amount of elasticity (default = 0.0)
- **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 of second SRSF
- **q2** – vector of size N or array of NxM 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)`

Generates random warping functions

**Parameters**

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

**Returns** gam: numpy ndarray of warping functions

`utility_functions.smooth_data(f, sparam)`

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** – predictive function
- **bt** – basis function
- **time** – time samples
- **y\_max** – maximum response for warping function  $gmax$
- **y\_min** – minimum response for warping function  $gmin$
- **gmax** – max warping function
- **gmin** – min warping function

**Return type** numpy array

**Return  $gamma$**  optimal warping function



## 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, beta, bt, y_max, y_min, gmax, gmin)`

finds zero-crossing of optimal gamma,  $\text{gam} = s \cdot \text{gmax} + (1-s) \cdot \text{gmin}$  from elastic curve regression model

**Parameters**

- **Y** – response
- **beta** – predicitive 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 fuction

**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)`

Calculates shooting vector from v to q1

**Parameters**

- **q1** – vector of srvf
- **v** – shooting vector

:rtype numpy ndarray :return q2n: vector of 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 rotated beta2 to beta1

**Return O** rotation matrix

**Return tau** seed

`curve_functions.find_rotation_and_seed_q(q1, q2)`

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

**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 beta2new** optimal rotated beta2 to beta1

**Return O** rotation matrix

**Return tau** seed

`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 Orthogonalization 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** orthogonalized basis

`curve_functions.group_action_by_gamma(q, gamma)`

This function reparameterized 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** reparameterized srvf

`curve_functions.group_action_by_gamma_coord(f, gamma)`

This function reparameterized 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** reparameterized 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 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 preprocessed 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





## 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](mailto:jdtuck@sandia.gov)>

`umap_metric.efda_distance(q1, q2)`

” 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

**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  $n \times M$
- **beta2** – vector of size  $n \times M$
- **closed** –  
(0) if open curves and (1) if closed curves

**Return type** scalar

**Return dist** shape distance



## 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.

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### b

boxplots, [15](#)

### c

curve\_functions, [45](#)

curve\_stats, [33](#)

### e

elastic\_glm\_regression, [29](#)

### f

fPCA, [11](#)

fPLS, [17](#)

### g

geodesic, [35](#)

### p

pcr\_regression, [25](#)

### r

regression, [19](#)

### t

time\_warping, [3](#)

tolerance, [31](#)

### u

umap\_metric, [53](#)

utility\_functions, [39](#)





## A

align\_fPCA() (in module *time\_warping*), 3  
 align\_fPLS() (in module *time\_warping*), 3  
 ampbox (class in *boxplots*), 15

## B

back\_parallel\_transport() (in module *geodesic*), 35  
 Basis\_Normal\_A() (in module *curve\_functions*), 45  
 bootTB() (in module *tolerance*), 31  
 boxplots  
     module, 15

## C

calc\_alphadot() (in module *geodesic*), 35  
 calc\_fpca() (fPCA.fdahpca method), 11  
 calc\_fpca() (fPCA.fdajpca method), 12  
 calc\_fpca() (fPCA.fdavpca method), 13  
 calc\_j() (in module *curve\_functions*), 45  
 calc\_model() (elastic\_glm\_regression.elastic\_glm\_regression  
     method), 29  
 calc\_model() (pcr\_regression.elastic\_lpcr\_regression  
     method), 25  
 calc\_model() (pcr\_regression.elastic\_mlpcr\_regression  
     method), 26  
 calc\_model() (pcr\_regression.elastic\_pcr\_regression  
     method), 27  
 calculate\_energy() (in module *geodesic*), 35  
 calculate\_gradE() (in module *geodesic*), 36  
 calculate\_variance() (in module *curve\_functions*),  
     45  
 calculatecentroid() (in module *curve\_functions*), 45  
 construct\_boxplot() (boxplots.ampbox method), 15  
 construct\_boxplot() (boxplots.phbox method), 16  
 cov\_integral() (in module *geodesic*), 36  
 cumtrapzmid() (in module *utility\_functions*), 39  
 curve\_functions  
     module, 45  
 curve\_stats  
     module, 33  
 curve\_to\_q() (in module *curve\_functions*), 45  
 curve\_zero\_crossing() (in module *curve\_functions*),  
     46

## D

diffop() (in module *utility\_functions*), 40

## E

efda\_distance() (in module *umap\_metric*), 53  
 efda\_distance\_curve() (in module *umap\_metric*), 53  
 elastic\_depth() (in module *utility\_functions*), 40  
 elastic\_distance() (in module *utility\_functions*), 40  
 elastic\_distance\_curve() (in module  
     *curve\_functions*), 46  
 elastic\_glm\_regression  
     module, 29  
 elastic\_glm\_regression (class in *elastic\_glm\_regression*), 29  
 elastic\_logistic() (in module *regression*), 19  
 elastic\_lpcr\_regression (class in *pcr\_regression*),  
     25  
 elastic\_mlogistic() (in module *regression*), 19  
 elastic\_mlpcr\_regression (class in *pcr\_regression*),  
     26  
 elastic\_pcr\_regression (class in *pcr\_regression*), 27  
 elastic\_prediction() (in module *regression*), 20  
 elastic\_regression() (in module *regression*), 20  
 elastic\_shooting() (in module *curve\_functions*), 46

## F

f\_K\_fold() (in module *utility\_functions*), 40  
 f\_to\_srsf() (in module *utility\_functions*), 41  
 fdacurve (class in *curve\_stats*), 33  
 fdahpca (class in *fPCA*), 11  
 fdajpca (class in *fPCA*), 12  
 fdavpca (class in *fPCA*), 12  
 fdawarp (class in *time\_warping*), 4  
 find\_basis\_normal() (in module *curve\_functions*), 46  
 find\_basis\_normal\_path() (in module *geodesic*), 36  
 find\_best\_rotation() (in module *curve\_functions*),  
     46  
 find\_rotation\_and\_seed\_coord() (in module  
     *curve\_functions*), 47  
 find\_rotation\_and\_seed\_q() (in module  
     *curve\_functions*), 47

`find_rotation_and_seed_unique()` (in module `curve_functions`), 47

`fPCA`  
module, 11

`fPLS`  
module, 17

## G

`gauss_model()` (*time\_warping.fdawarp method*), 5

`geigen()` (in module `utility_functions`), 41

`geod_dist_path_strt()` (in module `geodesic`), 36

`geod_sphere()` (in module `geodesic`), 36

`geodesic`  
module, 35

`gradient_spline()` (in module `utility_functions`), 41

`gram_schmidt()` (in module `curve_functions`), 48

`group_action_by_gamma()` (in module `curve_functions`), 48

`group_action_by_gamma_coord()` (in module `curve_functions`), 48

## I

`init_path_geod()` (in module `geodesic`), 37

`init_path_rand()` (in module `geodesic`), 37

`innerprod_q()` (in module `utility_functions`), 41

`innerprod_q2()` (in module `curve_functions`), 48

`inverse_exp()` (in module `curve_functions`), 48

`inverse_exp_coord()` (in module `curve_functions`), 48

`invertGamma()` (in module `utility_functions`), 41

## J

`joint_gauss_model()` (*time\_warping.fdawarp method*), 5

## K

`karcher_cov()` (*curve\_stats.fdacurve method*), 33

`karcher_mean()` (*curve\_stats.fdacurve method*), 33

## L

`logistic_warp()` (in module `regression`), 21

`logit_gradient()` (in module `regression`), 21

`logit_hessian()` (in module `regression`), 21

`logit_loss()` (in module `regression`), 21

## M

`mlogit_gradient()` (in module `regression`), 21

`mlogit_loss()` (in module `regression`), 22

`mlogit_warp_grad()` (in module `regression`), 22

module  
boxplots, 15  
curve\_functions, 45  
curve\_stats, 33  
elastic\_glm\_regression, 29

`fPCA`, 11

`fPLS`, 17

`geodesic`, 35

`pcr_regression`, 25

`regression`, 19

`time_warping`, 3

`tolerance`, 31

`umap_metric`, 53

`utility_functions`, 39

`multiple_align_functions()`  
(*time\_warping.fdawarp method*), 5

`mvtol_region()` (in module `tolerance`), 31

## N

`normal()` (in module `time_warping`), 6

## O

`optimum_reparam()` (in module `utility_functions`), 42

`optimum_reparam_curve()` (in module `curve_functions`), 49

`optimum_reparam_pair()` (in module `utility_functions`), 42

`outlier_detection()` (in module `utility_functions`), 42

## P

`pairwise_align_bayes()` (in module `time_warping`), 7

`pairwise_align_bayes_infHMC()` (in module `time_warping`), 8

`pairwise_align_functions()` (in module `time_warping`), 8

`parallel_translate()` (in module `curve_functions`), 49

`path_straightening()` (in module `geodesic`), 37

`pcaTB()` (in module `tolerance`), 32

`pcr_regression`  
module, 25

`phbox` (class in `boxplots`), 15

`phi()` (in module `regression`), 22

`plot()` (*boxplots.ampbox method*), 15

`plot()` (*boxplots.phbox method*), 16

`plot()` (*curve\_stats.fdacurve method*), 33

`plot()` (*fPCA.fdahpca method*), 11

`plot()` (*fPCA.fdajpca method*), 12

`plot()` (*fPCA.fdavpca method*), 13

`plot()` (*time\_warping.fdawarp method*), 5

`pls_svd()` (in module `fPLS`), 17

`pre_proc_curve()` (in module `curve_functions`), 49

`predict()` (*elastic\_glm\_regression.elastic\_glm\_regression method*), 29

`predict()` (*pcr\_regression.elastic\_lpcr\_regression method*), 25

`predict()` (*pcr\_regression.elastic\_mlpcr\_regression method*), 26

`predict()` (*pcr\_regression.elastic\_pcr\_regression*  
*method*), 27  
`project_curve()` (*in module curve\_functions*), 49  
`project_tangent()` (*in module curve\_functions*), 50  
`psi()` (*in module curve\_functions*), 50

## Q

`q_to_curve()` (*in module curve\_functions*), 50

## R

`rand()` (*in module elastic\_glm\_regression*), 30  
`rand()` (*in module time\_warping*), 9  
`randn()` (*in module curve\_stats*), 34  
`randomGamma()` (*in module utility\_functions*), 42  
`regression`  
*module*, 19  
`regression_warp()` (*in module regression*), 22  
`resamplecurve()` (*in module curve\_functions*), 50  
`resamplefunction()` (*in module utility\_functions*), 42  
`rgam()` (*in module utility\_functions*), 43  
`rwishart()` (*in module tolerance*), 32

## S

`sample_shapes()` (*curve\_stats.fdacurve method*), 33  
`scale_curve()` (*in module curve\_functions*), 51  
`shape_pca()` (*curve\_stats.fdacurve method*), 33  
`shift_f()` (*in module curve\_functions*), 51  
`smooth_data()` (*in module utility\_functions*), 43  
`SqrtMean()` (*in module utility\_functions*), 39  
`SqrtMeanInverse()` (*in module utility\_functions*), 39  
`SqrtMedian()` (*in module utility\_functions*), 39  
`srsf_align()` (*time\_warping.fdawarp method*), 5  
`srsf_to_f()` (*in module utility\_functions*), 43  
`srvf_align()` (*curve\_stats.fdacurve method*), 33

## T

`time_warping`  
*module*, 3  
`tolerance`  
*module*, 31

## U

`umap_metric`  
*module*, 53  
`update_path()` (*in module geodesic*), 38  
`update_progress()` (*in module utility\_functions*), 43  
`utility_functions`  
*module*, 39

## W

`warp_f_gamma()` (*in module utility\_functions*), 43  
`warp_q_gamma()` (*in module utility\_functions*), 43

## Z

`zero_crossing()` (*in module utility\_functions*), 44