

---

# **fdasrsf Documentation**

***Release 1.2.0***

**J. Derek Tucker**

February 06, 2014



## CONTENTS

<b>1</b>	<b>Functional Alignment</b>	<b>3</b>
<b>2</b>	<b>Functional Principal Component Analysis</b>	<b>7</b>
<b>3</b>	<b>Gaussian Generative Models</b>	<b>9</b>
<b>4</b>	<b>Functional Principal Least Squares</b>	<b>11</b>
<b>5</b>	<b>Elastic Regression</b>	<b>13</b>
<b>6</b>	<b>SRVF Geodesic Computation</b>	<b>19</b>
<b>7</b>	<b>Utility Functions</b>	<b>25</b>
<b>8</b>	<b>Curve Functions</b>	<b>31</b>
<b>9</b>	<b>Indices and tables</b>	<b>35</b>
	<b>Python Module Index</b>	<b>37</b>
	<b>Index</b>	<b>39</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:: Derek Tucker <dtucker@stat.fsu.edu>

`time_warping.align_fPCA(f, time, num_comp=3, showplot=True, smoothdata=False)`

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 M functions with N samples
- **time** (*np.ndarray*) – vector of size N 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)
- **sparam** (*double*) – Number of times to run box filter (default = 25)

**Return type** tuple of numpy array

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

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

**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

`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 M functions with N samples
- **g** (*np.ndarray*) – numpy ndarray of shape (M,N) of M functions with N samples

- **time** (*np.ndarray*) – vector of size N 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 M

functions with N samples :return gn: aligned functions - numpy ndarray of shape (M,N) of M functions with N 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

`time_warping.srsf_align(f, time, method='mean', showplot=True, smoothdata=False, lam=0.0)`

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

#### Parameters

- **f** – numpy ndarray of shape (M,N) of M functions with N samples
- **time** – vector of size N describing the sample points
- **method** – (string) warp calculate Karcher Mean or Median

(options = “mean” or “median”) (default=“mean”) :param showplot: Shows plots of results using matplotlib (default = T) :param smoothdata: Smooth the data using a box filter (default = F) :param lam: controls the elasticity (default = 0) :type lam: double :type smoothdata: bool :type f: np.ndarray :type time: np.ndarray

**Return type** tuple of numpy array

**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 q0: original srvf - similar structure to fn :return fmean: function mean or median - vector of length N :return mqn: srvf mean or median - vector of length N :return gam: warping functions - similar structure to fn :return orig\_var: Original Variance of Functions :return amp\_var: Amplitude Variance :return phase\_var: Phase Variance

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

`time_warping.srsf_align_pair(f, g, time, method='mean', showplot=True, smoothdata=False, lam=0.0)`

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

#### Parameters

- **f** (*np.ndarray*) – numpy ndarray of shape (M,N) of M functions with N samples
- **g** – numpy ndarray of shape (M,N) of M functions with N samples
- **time** (*np.ndarray*) – vector of size N describing the sample points
- **method** – (string) warp calculate Karcher Mean or Median (options = “mean” or “median”) (default=“mean”)
- **showplot** – Shows plots of results using matplotlib (default = T)



- **smoothdata** (*bool*) – Smooth the data using a box filter (default = F)
- **lam** (*double*) – controls the elasticity (default = 0)

**Return type** tuple of numpy array

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

**Return gn** aligned functions - numpy ndarray of shape (M,N) of M functions with N 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 fmean** f function mean or median - vector of length N

**Return gmean** g function mean or median - vector of length N

**Return mqfn** srvf mean or median - vector of length N

**Return mqgn** srvf mean or median - vector of length N

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



## FUNCTIONAL PRINCIPAL COMPONENT ANALYSIS

Vertical and Horizontal Functional Principal Component Analysis using SRSF

moduleauthor:: Derek Tucker <[dtucker@stat.fsu.edu](mailto:dtucker@stat.fsu.edu)>

`fPCA.horizfPCA(gam, time, no, showplot=True)`

This function calculates horizontal functional principal component analysis on aligned data

### Parameters

- **gam** – numpy ndarray of shape (M,N) of M warping functions
- **time** – vector of size N describing the sample points
- **no** (*int*) – number of components to extract (default = 1)
- **showplot** (*bool*) – Shows plots of results using matplotlib (default = T)

**Return type** tuple 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

`fPCA.vertfPCA(fn, time, qn, no=1, showplot=True)`

This function calculates vertical functional principal component analysis on aligned data

### Parameters

- **fn** – numpy ndarray of shape (M,N) of M aligned functions with N samples
- **time** – vector of size N describing the sample points
- **qn** – numpy ndarray of shape (M,N) of M aligned SRSF with N samples
- **no** (*int*) – number of components to extract (default = 1)
- **showplot** (*bool*) – Shows plots of results using matplotlib (default = T)

**Return type** tuple 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

## GAUSSIAN GENERATIVE MODELS

Gaussian Model of functional data

moduleauthor:: Derek Tucker <[dtucker@stat.fsu.edu](mailto:dtucker@stat.fsu.edu)>

`gauss_model.gauss_model` (*fn, time, qn, gam, n=1, sort\_samples=False*)

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

**Parameters** *fn* – numpy ndarray of shape (M,N) of M aligned functions with

N samples :param *time*: vector of size N describing the sample points :param *qn*: numpy ndarray of shape (M,N) of M aligned srvfs with N samples :param *gam*: warping functions :param *n*: number of random samples :param *sort\_samples*: sort samples (default = T) :type *n*: integer :type *sort\_samples*: bool :type *fn*: np.ndarray :type *qn*: np.ndarray :type *gam*: np.ndarray :type *time*: np.ndarray

**Return type** tuple of numpy array

**Return** *fs* random aligned samples

**Return** *gams* random warping functions

**Return** *ft* random samples



## FUNCTIONAL PRINCIPAL LEAST SQUARES

Partial Least Squares using SVD

moduleauthor:: Derek Tucker <[dtucker@stat.fsu.edu](mailto:dtucker@stat.fsu.edu)>

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 M functions with N samples
- **qg** – numpy ndarray of shape (M,N) of M functions with N 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:: Derek Tucker <[dtucker@stat.fsu.edu](mailto:dtucker@stat.fsu.edu)>

`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 M functions with N samples
- **y** – numpy array of labels (1/-1)
- **time** (*np.ndarray*) – vector of size N 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 M functions with N samples
- **y** – numpy array of labels {1,2,...,m} for m classes
- **time** (*np.ndarray*) – vector of size N 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_prediction(f, time, model, y=None, smooth=False)`

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

#### Parameters

- **f** – numpy ndarray of shape (M,N) of M functions with N samples
- **time** – vector of size N 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 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.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 M functions with N samples
- **y** – numpy array of N responses
- **time** (*np.ndarray*) – vector of size N 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 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

**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 M functions with N samples
- **X** – 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

**Return type** numpy array

**Return grad** gradient of logisitc loss

`regression.logit_hessian(s, b, X, y)`  
calculates hessian of the logistic loss

**Parameters**

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

**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 M functions with N samples
- **X** – 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

**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 M functions with N samples
- **X** – 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

**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 M functions with N samples
- **X** – 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

**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 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 max\_itr: maximum number of iterations (Default=8000) :param tol: stopping tolerance (Default=1e-10) :param delta: gradient step size (Default=0.008) :param 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



## SRVF GEODESIC COMPUTATION

geodesic calculation for SRVF (curves) open and closed)

moduleauthor:: Derek Tucker <[dtucker@stat.fsu.edu](mailto:dtucker@stat.fsu.edu)>

`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 geodesic 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)`  
This function calculates the geodesic 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)

**Return type** numpy ndarray

**Return dist** geodesic distance

**Return path** geodesic path

**Return O** rotation matrix

`geodesic.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

`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.parallel_translate(w, q1, q2, basis)`  
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

**Return type** numpy ndarray

**Return wbar** translated vector

`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.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

`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:: Derek Tucker <[dtucker@stat.fsu.edu](mailto:dtucker@stat.fsu.edu)>

`utility_functions.SqrtMean(gam)`

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

**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.cumtrapzmid(x, y, c)`

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` – midpoint

**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_distance(f1, f2, time, 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
- **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 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, lam=0.0)`

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
- **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.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)`  
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.update_progress(progress)`  
This function creates a progress bar

**Parameters** progress – fraction of progress

`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,  $\text{gam} = s * \text{gmax} + (1-s) * \text{gmin}$  from elastic regression model

**Parameters**

- **Y** – response
- **q** – predicitive 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 fuction

**Return type** numpy array

**Return gamma** optimal warping function



## CURVE FUNCTIONS

functions for SRVF curve manipulations

moduleauthor:: Derek Tucker <[dtucker@stat.fsu.edu](mailto:dtucker@stat.fsu.edu)>

`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)`

This function converts curve beta to srvf q

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

**Return type** numpy ndarray

**Return q** srvf of curve

`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)`

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

**Return type** numpy ndarray

**Return q2new** optimal rotated q2 to q1

**Return R** rotation matrix

`curve_functions.find_rotation_and_seed_coord(beta1, beta2)`

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

**Parameters**

- **beta2** – 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.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_q(q1, q2)`

This function calculates the innerproduct 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.optimum_reparam_curve(q1, q2, lam=0.0)`

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)

**Return type** vector

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

`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.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)`

This function converts srvf to beta

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

**Return type** numpy ndarray

**Return beta** parameterized curve

`curve_functions.resamplecurve(x, N=100)`

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)

**Return type** numpy ndarray

**Return xn** resampled curve

`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

References:

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.

Joshi, S.H., Srivastava, A., Klassen, E. and Jermyn, I. (2007). A Novel Representation for Computing Geodesics Between n-Dimensional Elastic Curves. IEEE Conference on computer Vision and Pattern Recognition (CVPR), Minneapolis, MN.

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.

## INDICES AND TABLES

- *genindex*
- *modindex*
- *search*





## **c**

`curve_functions`, 31

## **f**

`fPCA`, 7

`fPLS`, 11

## **g**

`gauss_model`, 9

`geodesic`, 19

## **r**

`regression`, 13

## **t**

`time_warping`, 3

## **u**

`utility_functions`, 25



- align\_fPCA() (in module time\_warping), 3
- align\_fPLS() (in module time\_warping), 3
- back\_parallel\_transport() (in module geodesic), 19
- calc\_alphadot() (in module geodesic), 19
- calc\_j() (in module curve\_functions), 31
- calculate\_energy() (in module geodesic), 19
- calculate\_gradE() (in module geodesic), 20
- calculate\_variance() (in module curve\_functions), 31
- calculatecentroid() (in module curve\_functions), 31
- cov\_integral() (in module geodesic), 20
- cumtrapzmid() (in module utility\_functions), 25
- curve\_functions (module), 31
- curve\_to\_q() (in module curve\_functions), 31
- diffop() (in module utility\_functions), 25
- elastic\_distance() (in module utility\_functions), 26
- elastic\_logistic() (in module regression), 13
- elastic\_mlogistic() (in module regression), 13
- elastic\_prediction() (in module regression), 14
- elastic\_regression() (in module regression), 14
- f\_K\_fold() (in module utility\_functions), 26
- f\_to\_srsf() (in module utility\_functions), 26
- find\_basis\_normal() (in module curve\_functions), 31
- find\_basis\_normal\_path() (in module geodesic), 20
- find\_best\_rotation() (in module curve\_functions), 31
- find\_rotation\_and\_seed\_coord() (in module curve\_functions), 32
- fPCA (module), 7
- fPLS (module), 11
- gauss\_model (module), 9
- gauss\_model() (in module gauss\_model), 9
- geigen() (in module utility\_functions), 26
- geod\_dist\_path\_strt() (in module geodesic), 20
- geod\_sphere() (in module geodesic), 20
- geodesic (module), 19
- gradient\_spline() (in module utility\_functions), 26
- gram\_schmidt() (in module geodesic), 21
- group\_action\_by\_gamma() (in module curve\_functions), 32
- group\_action\_by\_gamma\_coord() (in module curve\_functions), 32
- horizfPCA() (in module fPCA), 7
- init\_path\_geod() (in module geodesic), 21
- init\_path\_rand() (in module geodesic), 21
- innerprod\_q() (in module curve\_functions), 32
- innerprod\_q() (in module utility\_functions), 27
- invertGamma() (in module utility\_functions), 27
- logistic\_warp() (in module regression), 15
- logit\_gradient() (in module regression), 15
- logit\_hessian() (in module regression), 15
- logit\_loss() (in module regression), 15
- mlogit\_gradient() (in module regression), 16
- mlogit\_loss() (in module regression), 16
- mlogit\_warp\_grad() (in module regression), 16
- optimum\_reparam() (in module utility\_functions), 27
- optimum\_reparam\_curve() (in module curve\_functions), 32
- optimum\_reparam\_pair() (in module utility\_functions), 27
- outlier\_detection() (in module utility\_functions), 27
- parallel\_translate() (in module geodesic), 21
- path\_straightening() (in module geodesic), 22
- phi() (in module regression), 16
- pls\_svd() (in module fPLS), 11
- pre\_proc\_curve() (in module curve\_functions), 33
- project\_curve() (in module curve\_functions), 33
- project\_tangent() (in module geodesic), 22
- psi() (in module curve\_functions), 33
- q\_to\_curve() (in module curve\_functions), 33
- randomGamma() (in module utility\_functions), 28
- regression (module), 13
- regression\_warp() (in module regression), 17
- resamplecurve() (in module curve\_functions), 33
- resamplefunction() (in module utility\_functions), 28
- rgam() (in module utility\_functions), 28

`shift_f()` (in module `curve_functions`), [34](#)  
`smooth_data()` (in module `utility_functions`), [28](#)  
`SqrtMean()` (in module `utility_functions`), [25](#)  
`SqrtMeanInverse()` (in module `utility_functions`), [25](#)  
`srsf_align()` (in module `time_warping`), [4](#)  
`srsf_align_pair()` (in module `time_warping`), [4](#)  
  
`time_warping` (module), [3](#)  
  
`update_path()` (in module `geodesic`), [22](#)  
`update_progress()` (in module `utility_functions`), [28](#)  
`utility_functions` (module), [25](#)  
  
`vertfPCA()` (in module `fPCA`), [7](#)  
  
`warp_q_gamma()` (in module `utility_functions`), [28](#)  
  
`zero_crossing()` (in module `utility_functions`), [29](#)