fdasrsf Documentation

Release 1.5.0

J. Derek Tucker

Aug 07, 2019

CONTENTS

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

FUNCTIONAL ALIGNMENT

Group-wise function alignment using SRSF framework and Dynamic Programming

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

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 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)
- **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 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

time_warping.**srsf_align**(*f*, *time*, *method='mean'*, *omethod='DP'*, *showplot=True*, *smooth-data=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 N functions with M samples

- **time** – vector of size M describing the sample points

- **method** – (string) warp calculate Karcher Mean or Median

(options = "mean" or "median") (default="mean") :param omethod: optimization method (DP, DP2, RBFGS) (default = DP) :param showplot: Shows plots of results using matplotlib (default = T) :param smoothdata: Smooth the data using a box filter (default = F) :param parallel: run in parallel (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 N

functions with M 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 M :return mqn: srvf mean or median - vector of length M :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 N functions with M samples

- **g** – numpy ndarray of shape (M,N) of N functions with M samples
- **time** (*np.ndarray*) – vector of size M 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 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 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 <jdtuck@sandia.gov>

fPCA.**horizfPCA**(*gam*, *time*, *no=2*, *showplot=True*)
    This function calculates horizontal functional principal component analysis on aligned data

> **Parameters**
>
> > - **gam** – numpy ndarray of shape (M,N) of N warping functions
> >
> > - **time** – vector of size M describing the sample points
> >
> > - **no** (*int*) – number of components to extract (default = 2)
> >
> > - **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.**jointfPCA**(*fn*, *time*, *qn*, *q0*, *gam*, *no=2*, *showplot=True*)
    This function calculates joint functional principal component analysis on aligned data

> **Parameters**
>
> > - **fn** – numpy ndarray of shape (M,N) of N aligned functions with M samples
> >
> > - **time** – vector of size N describing the sample points
> >
> > - **qn** – numpy ndarray of shape (M,N) of N aligned SRSF with M samples
> >
> > - **no** (*int*) – number of components to extract (default = 2)
> >
> > - **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=2*, *showplot=True*)

    This function calculates vertical functional principal component analysis on aligned data

        Parameters

- **fn** – numpy ndarray of shape (M,N) of N aligned functions with M samples
- **time** – vector of size N describing the sample points
- **qn** – numpy ndarray of shape (M,N) of N aligned SRSF with M samples
- **no** (*int*) – number of components to extract (default = 2)
- **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

ELASTIC FUNCTIONAL BOXPLOTS

Elastic Functional Boxplots

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

boxplots.**ampbox**(*ft*, *f_median*, *qt*, *q_median*, *time*, *alpha=0.05*, *k_a=1*)

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

> **Parameters**
>
> - **ft** – numpy ndarray of shape (M,N) of N functions with M samples
> - **f_median** – vector of size M describing the median
> - **qt** – numpy ndarray of shape (M,N) of N srsf functions with M samples
> - **q_median** – vector of size M describing the srsf median
> - **time** – vector of size M describing the time
> - **alpha** – quantile value (e.g.,=.05, i.e., 95%)
> - **k_a** – scalar for outlier cutoff (e.g.,=1)
>
> **Return type** tuple of numpy array
>
> **Return fn** aligned functions - numpy ndarray of shape (M,N) of N

functions with M samples :return Q1: First quartile :return Q3: Second quartile :return Q1a: First quantile based on alpha :return Q3a: Second quantile based on alpha :return minn: minimum extreme function :return maxx: maximum extreme function :return outlier_index: indexes of outlier functions :return f_median: median function :return q_median: median srsf :return plt: surface plot mesh

boxplots.**phbox**(*gam*, *time*, *alpha=0.05*, *k_a=1*)

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

> **Parameters**
>
> - **gam** – numpy ndarray of shape (M,N) of N warping functions with M samples
> - **alpha** – quantile value (e.g.,=.05, i.e., 95%)
> - **k_a** – scalar for outlier cutoff (e.g.,=1)
>
> **Return type** tuple of numpy array
>
> **Return fn** aligned functions - numpy ndarray of shape (M,N) of N

functions with M samples :return Q1: First quartile :return Q3: Second quartile :return Q1a: First quantile based on alpha :return Q3a: Second quantile based on alpha :return minn: minimum extreme function :return maxx: maximum extreme function :return outlier_index: indexes of outlier functions :return median_x: median warping function :return psi_median: median srsf of warping function :return plt: surface plot mesh

GAUSSIAN GENERATIVE MODELS

Gaussian Model of functional data

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

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** (*np.ndarray*) – numpy ndarray of shape (M,N) of N aligned functions with M samples
> > - **time** (*np.ndarray*) – vector of size M describing the sample points
> > - **qn** (*np.ndarray*) – numpy ndarray of shape (M,N) of N aligned srvfs with M samples
> > - **gam** (*np.ndarray*) – warping functions
> > - **n** (*integer*) – number of random samples
> > - **sort_samples** (*bool*) – sort samples (default = T)
>
> **Return type** tuple of numpy array
>
> **Return fs** random aligned samples
>
> **Return gams** random warping functions
>
> **Return ft** random samples

gauss_model.**joint_gauss_model** (*fn*, *time*, *qn*, *gam*, *q0*, *n=1*, *no=3*)
This function models the functional data using a joint Gaussian model extracted from the principal components of the srsfs

> **Parameters**
>
> > - **fn** (*np.ndarray*) – numpy ndarray of shape (M,N) of N aligned functions with M samples
> > - **time** (*np.ndarray*) – vector of size M describing the sample points
> > - **qn** (*np.ndarray*) – numpy ndarray of shape (M,N) of N aligned srsfs with M samples
> > - **gam** (*np.ndarray*) – warping functions
> > - **q0** – numpy ndarray of shape (M,N) of N unaligned srsfs with samples
> > - **n** (*integer*) – number of random samples
> > - **n** – number of principal components (default = 3)

> **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 <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:: Derek Tucker <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-variablity 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-variablity 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-variablity

> > **Parameters**

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

> > - **time** – vector of size M describing the sample points

> > - **model** – indentified 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-variablity 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 logisitc loss

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

> calculates hessian of the logistic loss
>
> > **Parameters**
> >
> > - **s** – numpy ndarray of shape (M,N) of N functions with M samples
> >
> > - **b** – numpy ndarray of shape (M,N) of N functions with M samples
> >
> > - **X** – numpy ndarray of shape (M,N) of N functions with M samples
> >
> > - **y** – numpy ndarray of shape (1,N) responses
> >
> > **Return type** numpy array
> >
> > **Return out** hessian of logistic loss

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

> logistic loss function, returns Sum{-log(phi(t))}
>
> > **Parameters**
> >
> > - **b** – numpy ndarray of shape (M,N) of N functions with M samples
> >
> > - **X** – numpy ndarray of shape (M,N) of N functions with M samples
> >
> > - **y** – numpy ndarray of shape (1,N) of N responses
> >
> > **Return type** numpy array
> >
> > **Return out** loss value

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

    calculates gradient of the multinomial logistic loss

        Parameters

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

        Return type numpy array

        Return grad gradient

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

    calculates multinomial logistic loss (negative log-likelihood)

        Parameters

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

        Return type numpy array

        Return nll negative log-likelihood

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

    calculates optimal warping for functional multinomial logistic regression

        Parameters

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

        Return type tuple of numpy array

        Return gam_old warping function

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

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

        Parameters **t** – scalar

        Return type numpy array

        Return out return value

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

    calculates optimal warping for function linear regression

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

responses :param alpha: numpy scalar

> **Return type** numpy array
>
> **Return gamma_new** warping function

ELASTIC PRINCIPAL COMPONENT REGRESSION

Warping PCR Invariant Regression using SRSF

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

pcr_regression.**elastic_lpcr_regression**(*f*, *y*, *time*, *pca_method='combined'*, *no=5*, *smooth_data=False*, *sparam=25*)

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

Parameters

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

- **y** – numpy array of N responses

- **time** (*np.ndarray*) – vector of size M describing the sample points

- **pca_method** – string specifing pca method (options = "combined", "vert", or "horiz", default = "combined")

- **no** – scalar specify number of principal components (default=5)

- **smooth_data** – smooth data using box filter (default = F)

- **sparam** – number of times to apply box filter (default = 25)

Return type tuple of numpy array

Return alpha alpha parameter of model

Return b regressor vector

Return y response vector

Return warp_data alignment object from srsf_align

Return pca fpca object from corresponding pca method

Return Loss logistic loss

Return pca.method string of pca method

pcr_regression.**elastic_mlpcr_regression**(*f*, *y*, *time*, *pca_method='combined'*, *no=5*, *smooth_data=False*, *sparam=25*)

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

Parameters

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

- **y** – numpy array of N responses

- **time** (*np.ndarray*) – vector of size M describing the sample points

- **pca_method** – string specifing pca method (options = "combined", "vert", or "horiz", default = "combined")

- **no** – scalar specify number of principal components (default=5)

- **smooth_data** – smooth data using box filter (default = F)

- **sparam** – number of times to apply box filter (default = 25)

Return type tuple of numpy array

Return alpha alpha parameter of model

Return b regressor vector

Return y response vector

Return warp_data alignment object from srsf_align

Return pca fpca object from corresponding pca method

Return Loss logistic loss

Return pca.method string of pca method

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

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

- **y** – numpy array of N responses

- **time** (*np.ndarray*) – vector of size M describing the sample points

- **pca_method** – string specifing pca method (options = "combined", "vert", or "horiz", default = "combined")

- **no** – scalar specify number of principal components (default=5)

- **smooth_data** – smooth data using box filter (default = F)

- **sparam** – number of times to apply box filter (default = 25)

- **parallel** – run in parallel (default = F)

- **C** – scale balance parameter for combined method (default = None)

Return type tuple of numpy array

Return alpha alpha parameter of model

Return b regressor vector

Return y response vector

Return warp_data alignment object from srsf_align

Return pca fpca object from corresponding pca method

Return SSE sum of squared errors

Return pca.method string of pca method

ELASTIC FUNCTIONAL TOLERANCE BOUNDS

Functional Tolerance Bounds using SRSF

moduleauthor:: Derek Tucker <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
>
> **Return ph** phase tolerance bounds

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

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

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

If positive, int_like or int-convertible 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 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use *numpy.random.standard_normal* instead.

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should be all positive. 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.

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):

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

tolerance.**rwishart**(*df, p*)

Parameters

- **df** – degree of freedom

- **p** – number of dimensions

Return type double

Return R matrix

SRVF GEODESIC COMPUTATION

geodesic calculation for SRVF (curves) open and closed)

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

geodesic.**back_parallel_transport**(*u1*, *alpha*, *basis*, *T=100*, *k=5*)
backwards parallel translates q1 and q2 along manifold

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

geodesic.**calc_alphadot**(*alpha*, *basis*, *T=100*, *k=5*)
calculates derivative along the path alpha

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

geodesic.**calculate_energy**(*alphadot*, *T=100*, *k=5*)
calculates energy along path

> **Parameters**
>
> - **alphadot** – numpy ndarray of shape (2,M) of M samples
> - **T** – Number of samples of curve (Default = 100)
> - **k** – number of samples along path (Default = 5)
>
> **Return type** numpy scalar

Return E energy

geodesic.**calculate_gradE**(*u*, *utilde*, *T=100*, *k=5*)
    calculates gradient of energy along path

> **Parameters**
>
> > - **u** – numpy ndarray of shape (2,M) of M samples
> >
> > - **utilde** – numpy ndarray of shape (2,M) of M samples
> >
> > - **T** – Number of samples of curve (Default = 100)
> >
> > - **k** – number of samples along path (Default = 5)
>
> **Return type** numpy scalar
>
> **Return gradE** gradient of energy
>
> **Return normgradE** norm of gradient of energy

geodesic.**cov_integral**(*alpha*, *alphadot*, *basis*, *T=100*, *k=5*)
    Calculates covariance along path alpha

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

geodesic.**find_basis_normal_path**(*alpha*, *k=5*)
    computes orthonormalized basis vectors to the normal space at each of the k points (q-functions) of the path
    alpha

> **Parameters**
>
> > - **alpha** – numpy ndarray of shape (2,M) of M samples (path)
> >
> > - **k** – number of samples along path (Default = 5)
>
> **Return type** numpy ndarray
>
> **Return basis** basis vectors along the path

geodesic.**geod_dist_path_strt**(*beta*, *k=5*)
    calculate geodisc distance for path straightening

> **Parameters**
>
> > - **beta** – numpy ndarray of shape (2,M) of M samples
> >
> > - **k** – number of samples along path (Default = 5)
>
> **Return type** numpy scalar
>
> **Return dist** geodesic distance

geodesic.**geod_sphere**(*beta1*, *beta2*, *k=5*)
    This function caluclates the geodecis 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.**init_path_geod**(*beta1*, *beta2*, *T=100*, *k=5*)

Initializes a path in cal{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 cal{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 straigtening 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** – initilizae 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 paramenter
- **T** – Number of samples of curve (Default = 100)
- **k** – number of samples along path (Default = 5)

Return type numpy scalar

Return alpha updated path of srvfs

Return beta updated path of curves

UTILITY FUNCTIONS

Utility functions for SRSF Manipulations

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

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.**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** – midpoint

- **mid** – midpiont location

> **Return type** vector

> **Return fa** cumulative integration

utility_functions.**diffop**(*n*, *binsize=1*)
  Creates a second order differential operator

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

> **Return type** numpy ndarray

> **Return m** matrix describing differential operator

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

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

> **Return type** numpy ndarray
>
> **Return values** eigenvalues
>
> **Return Lmat** left eigenvectors
>
> **Return Mmat** right eigenvectors

utility_functions.**gradient_spline**(*time*, *f*, *smooth=False*)
> This function takes the gradient of f using b-spline smoothing

> **Parameters**
>
> > - **time** – vector of size N describing the sample points
> >
> > - **f** – numpy ndarray of shape (M,N) of M functions with N samples
> >
> > - **smooth** – smooth data (default = F)
>
> **Return type** tuple of numpy ndarray
>
> **Return f0** smoothed functions functions
>
> **Return g** first derivative of each function
>
> **Return g2** second derivative of each function

utility_functions.**innerprod_q**(*time*, *q1*, *q2*)
> calculates the innerproduct between two srsfs

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

> **Return type** scalar
>
> **Return val** inner product value

utility_functions.**invertGamma**(*gam*)
> finds the inverse of the diffeomorphism gamma

> **Parameters** **gam** – vector describing the warping function
>
> **Return type** vector
>
> **Return gamI** inverse of gam

utility_functions.**optimum_reparam**(*q1*, *time*, *q2*, *method='DP'*, *lam=0.0*, *f1o=0.0*, *f2o=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
> >
> > - **method** – method to apply optimzation (default="DP") options are "DP", "DP2" and "RBFGS"
> >
> > - **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.**srsf_to_f**(*q*, *time*, *f0=0.0*)

> converts q (srsf) to a function

> > **Parameters**

> > > • **q** – vector of size N samples of srsf

> > > • **time** – vector of size N describing time sample points

> > > • **f0** – initial value

> > **Return type** vector

> > **Return f** function

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

> This function creates a progress bar

> > **Parameters** **progress** – fraction of progress

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

> warps a function f by gam

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

> > **Return type** numpy ndarray

> > **Return f_temp** warped srsf

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

> warps a srsf q by gam

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

> > **Return type** numpy ndarray

> > **Return q_temp** warped srsf

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

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

> > **Parameters**

> > > • **Y** – response

> > > • **q** – predicitve function

> > > • **bt** – basis function

> > > • **time** – time samples

> > > • **y_max** – maximum repsonse for warping function gmax

> > > • **y_min** – minimum response for warping function gmin

- **gmax** – max warping function
- **gmin** – min warping fucntion

Return type numpy array

Return gamma optimal warping function

CURVE FUNCTIONS

functions for SRVF curve manipulations

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

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.**curve_zero_crossing**(*Y*, *beta*, *bt*, *y_max*, *y_min*, *gmax*, *gmin*)
> finds zero-crossing of optimal gamma, gam = s*gmax + (1-s)*gmin from elastic curve regression model

>> **Parameters**

>>> • **Y** – response

>>> • **beta** – predicitve function

>>> • **bt** – basis function

>>> • **y_max** – maximum repsonse for warping function gmax

- **y_min** – minimum response for warping function gmin

- **gmax** – max warping function

- **gmin** – min warping fucntion

> **Return type** numpy array
>
> **Return gamma** optimal warping function
>
> **Return O_hat** rotation matrix

curve_functions.**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**

- **beta1** – numpy ndarray of shape (2,M) of M samples

- **beta2** – 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_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.**gram_schmidt**(*basis*)
>   Performs Gram Schmidt Orthogonlization of a basis_o

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

>>>   **rtype** list of numpy ndarray

>>>   **return basis_o** orthogonlized basis

curve_functions.**group_action_by_gamma**(*q*, *gamma*)
>   This function reparamerized srvf q by gamma

>>   **Parameters**

>>>   • **f** – numpy ndarray of shape (2,M) of M samples

>>>   • **gamma** – numpy ndarray of shape (2,M) of M samples

>>   **Return type** numpy ndarray

>>   **Return qn** reparatermized srvf

curve_functions.**group_action_by_gamma_coord**(*f*, *gamma*)
>   This function reparamerized curve f by gamma

>>   **Parameters**

>>>   • **f** – numpy ndarray of shape (2,M) of M samples

>>>   • **gamma** – numpy ndarray of shape (2,M) of M samples

>>   **Return type** numpy ndarray

>>   **Return fn** reparatermized curve

curve_functions.**innerprod_q2**(*q1*, *q2*)
>   This function calculates the inner product in srvf space

>>   **Parameters**

>>>   • **q1** – numpy ndarray of shape (2,M) of M samples

>>>   • **q2** – numpy ndarray of shape (2,M) of M samples

>>   **Return type** numpy ndarray

>>   **Return val** inner product

curve_functions.**inverse_exp**(*q1*, *q2*, *beta2*)
>   Calculate the inverse exponential to obtain a shooting vector from q1 to q2 in shape space of open curves

>>   **Parameters**

>>>   • **q1** – numpy ndarray of shape (2,M) of M samples

>>>   • **q2** – numpy ndarray of shape (2,M) of M samples

>>>   • **beta2** – numpy ndarray of shape (2,M) of M samples

>>   **Return type** numpy ndarray

>>   **Return v** shooting vectors

curve_functions.**inverse_exp_coord**(*beta1*, *beta2*)
>   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

> **Return type** numpy ndarray

> **Return v** shooting vectors

> **Return dist** distance

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 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.**parallel_translate**(*w*, *q1*, *q2*, *basis*, *mode=0*)
> parallel translates q1 and q2 along manifold

> > **Parameters**

> > > • **w** – numpy ndarray of shape (2,M) of M samples

> > > • **q1** – numpy ndarray of shape (2,M) of M samples

> > > • **q2** – numpy ndarray of shape (2,M) of M samples

> > > • **basis** – list of numpy ndarray of shape (2,M) of M samples

> > > • **mode** – open 0 or closed curves 1 (default 0)

> > **Return type** numpy ndarray

> > **Return wbar** translated vector

curve_functions.**pre_proc_curve**(*beta*, *T=100*)
> This function prepcoessed a curve beta to set of closed curves

> > **Parameters**

> > > • **beta** – numpy ndarray of shape (2,M) of M samples

> > > • **T** – number of samples (default = 100)

> > **Return type** numpy ndarray

> > **Return betanew** projected beta

> > **Return qnew** projected srvf

> > **Return A** alignment matrix (not used currently)

curve_functions.**project_curve**(*q*)
> This function projects srvf q to set of close curves

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

> > **Return type** numpy ndarray

> > **Return qproj** project srvf

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

> projects srvf to tangent space w using basis

> > **Parameters**

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

> > **Return type** numpy ndarray

> > **Return wproj** projected q

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

> This function formats variance output

> > **Parameters**

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

> > **Return type** numpy ndarray

> > **Return psi1** variance

> > **Return psi2** cross variance

> > **Return psi3** curve end

> > **Return psi4** curve end

curve_functions.**q_to_curve**(*q*)

> 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.**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

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, 10.1002/sam.11399, 2018.

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX