spscicomp Documentation

Release beta

The Project Group

February 13, 2015

CONTENTS

Contents:

CHAPTER

ONE

COMMON MODULES

Currently there is one common module for all algorithms, namely the data importer module. It provides the following classes for importing numerical data:

1.1 common_data_importer

class `common_data_importer.`**`CommonBinaryFileDataImporter`**(*filename*)

Import data from a binary file. The file format should be as generated by numpy.save.

`get_data`(*size*)

Return a numpy array of floats where each data point occupies one row of the array. The data is read from the current position of the pointer onwards. If the pointer reaches the end of the file, an array of all data points up to the end of the file is returned and the hasMoreData flag is set to False.

Parameters **size** – Number of data points to be returned.

Returns A numpy array of data points.

`get_outData`(*size*)

Return a numpy array of floats where each data point occupies one row of the array. The data is read from the current position of the pointer onwards. If the pointer reaches the end of the file, an array of all data points up to the end of the file is returned and the hasMoreData flag is set to False.

Parameters **size** – Number of data points to be returned.

Returns A numpy array of data points.

`has_more_data`()

Test if the pointer is at the end of the file or not.

Returns True if there is more data after the pointer, and False if not.

`init_file_input_stream`()

Create a numpy array object which reads from the binary file using a memmap.

`rewind`()

Reset the file pointer to the beginning and set the hasMoreData flag to True.

`rewindPatch`()

Reset the file pointer to the beginning and set the hasMoreData flag to True.

`write_data`(*i_data*)

Assumption: i_data has same size like data of the last get_data() call

class `common_data_importer.`**`CommonDataImporter`**

This is an abstract data importer class. Implementations are expected to override the get_data and has_more_data methods.

class `common_data_importer.`**`CommonFileDataImporter`**(*filename*)

Import data from a text file. The data structure should be as follows: One point occupies one line. Each point consists of several floats with space as a separator.

`close_file`()

Close the file handle if it is open.

`get_data`(*size*)

Return a numpy array of floats where each data point occupies one row of the array. The data is read from the current position of the pointer onwards. If the pointer reaches the end of the file, an array of all data points up to the end of the file is returned, the file is closed and the hasMoreData flag is set to False.

Parameters **size** – Number of data points to be returned.

Returns A numpy array of data points.

`has_more_data`()

Test if the pointer is at the end of the file or not.

Returns True if there is more data after the pointer, and False if not.

`init_file_input_stream`()

Initialize the file input stream, that is, open the file and create the iterator on the file's lines.

`rewind`()

Reset the file pointer to the beginning, that is, initialize the file and set the hasMoreData flag to True.

class `common_data_importer.`**`CommonSimpleDataImporter`**(*data*)

"Import" data from a given data array.

`get_data`(*size*)

Return all available data regardless of the requested size.

Parameters **size** – Size of data which is to be returned. This parameter is disregarded as all data is returned.

Returns All data.

`has_more_data`()

Return if there is any more data. As all data is returned when using get_data, this function always returns False.

Returns False since there never is any more data.

CHAPTER

TWO

THE K-MEANS ALGORITHM

The implementation of the k-means algorithm consists of the following modules:

2.1 kmeans_main

`kmeans_main.`**`kmeans`**(*k*, *importer=None*)

Initialize and run the k-means algorithm. If any of the optimized implementations (CUDA, OpenCL, C extension) are available, they are selected and initialized automatically in the above order. Then the respective `kmeans.Kmeans.calculate_centers()` method is called and the output is returned.

Parameters

- **k** (*int*) – Number of cluster centers to compute.
- **importer** (`CommonDataImporter`) – A `CommonDataImporter` object to be used for importing the numerical data.

Returns An array of integers $[c(x_i)]$ where x_i is the i-th data point and $c(x_i)$ is the index of the cluster center to which x_i belongs.

Return type int[]

2.2 kmeans

class `kmeans.`**`DefaultKmeans`**(*metric=<spscicomp.kmeans.kmeans_metric.EuclideanMetric object at 0x7f24d5fe96d0>*, *importer=None*, *chunk_size=1000*, *max_steps=100*)

Default implementation of the k-means algorithm. Once supplied with an `CommonDataImporter` object, use the calculate_centers method to compute k cluster centers.

Parameters

- **metric** (`KmeansMetric`) – A `KmeansMetric` object to be used for calculating distances between points. The default is the `EuclideanMetric`.
- **importer** (`CommonDataImporter`) – A `CommonDataImporter` object to be used for importing the numerical data.
- **chunk_size** (*int*) – The number of data points to be imported and processed at a time.
- **max_steps** (*int*) – The maximum number of steps to run the algorithm for. If the iteration did not converge after this number of steps, the algorithm is terminated and the last result returned.

`calculate_centers` (*k*, *initial_centers=None*, *return_centers=False*, *save_history=False*)

Main method of the k-means algorithm. Computes k cluster centers from the data supplied by a `CommonDataImporter` object.

Parameters

- **k** (*int*) – Number of cluster centers to compute.
- **initial_centers** (*numpy.array*) – Array of cluster centers to start the iteration with. If omitted, random data points from the first chunk of data are used.
- **return_centers** (*bool*) – If set to True then the cluster centers are returned.
- **save_history** (*bool*) – If this and return_centers is set to True then the cluster centers in each iteration step are returned.

Returns An array of integers $[c(x_i)]$ where x_i is the i-th data point and $c(x_i)$ is the index of the cluster center to which x_i belongs.

Return type int[]

Returns An array of the computed cluster centers.

Return type np.array

Returns A list of arrays of the cluster centers in each iteration step.

Return type np.array[]

class `kmeans.`**`Kmeans`**(*metric=<spscicomp.kmeans.kmeans_metric.EuclideanMetric object at 0x7f24d5fe90d0>*, *importer=None*)

Abstract k-means algorithm. Implementations are expected to override the calculate_centers method.

2.3 c_kmeans

class `spscicomp.kmeans.extension.c_kmeans.`**`CKmeans`**(*metric=<spscicomp.kmeans.kmeans_metric.EuclideanMetric object at 0x7f24d5f04f10>*, *importer=None*, *chunk_size=1000*, *max_steps=100*)

An implementation of the k-means algorithm in C. Refer to the `DefaultKmeans` class for parameters and public methods.

static PyObject* **`cal_chunk_centers`**(PyObject **dummy*, PyObject **args*)

Main function of the C extension.

Parameters

- **args** (*PyObject**) – Pointer to parameters transported from Python.
- **dummy** (*PyObject**) – Not used here.

Returns The new chunk centers.

Return type PyObject*

Raises TypeError Python Arguments parse error!

void **`initkmeans_c_extension`**()

Initialize the extension module

PyMethodDef **`kmeans_c_extensionMethods`**

Variable which stores the maps between functions in C and Python int **closest_center** (PyArrayObject **data*, int *data_lab*, PyArrayObject **centers*, int *cluster_size*, int *dimension*)

Given the centers and one point and return which center is nearest to the point.

Parameters

- **data** (*PyArrayObject**) – One point with related dimension.
- **data_lab** (*int*) – Index of the point.
- **centers** (*PyArrayObject**) – Current centers.
- **cluster_size** (*int*) – Number of clusters.
- **dimension** (*int*) – Dimension of each point and center.

Returns The index of the nearest center.

Return type int

PyObject* **kmeans_chunk_center** (PyArrayObject **data*, PyArrayObject **centers*, PyObject **data_assigns*)

Record the nearest center of each point and renew the centers.

Parameters

- **data** (*PyArrayObject**) – Pointer to the point set to be calculated.
- **centers** (*PyArrayObject**) – Current centers.
- **data_assigns** (*PyObject**) – For each point record the index of the nearest center.

Returns The updated centers.

Return type PyObject*

Raises ValueError Parameters are of the wrong sizes.

Raises MemoryError RAM allocate error. The imported data chunk may be too large.

Raises MemoryError RAM release error.

Raises MemoryError Error occurs when creating a new PyArray

2.4 cuda_kmeans

class `cuda.cuda_kmeans.`**CUDAKmeans** (*metric=EuclideanMetric()*, *importer=None*, *chunk_size=1000*, *max_steps=100*)

An implementation of the k-means algorithm in CUDA. Refer to the `DefaultKmeans` class for parameters and public methods.

static PyObject* **cal_chunk_centers** (PyObject **dummy*, PyObject **args*)

Refer to the `cal_chunk_centers()` in c_kmeans.

void **initkmeans_c_extension_cuda** ()

Refer to the `initkmeans_c_extension()` in c_kmeans.

PyMethodDef **kmeans_c_extensionMethods**

Refer to the `kmeans_c_extension_cudaMethods` in c_kmeans.

_global__ void **`chunk_centers_sum_cuda`**(double **cu_data*, double **cu_centers*, int* *cu_centers_counter*, double* *cu_new_centers*, int* *cu_data_assigns*, int* *cluster_size*, int **dimension*, int **chunk_size*)

Divide the whole data set into several parts, each part is calculated by a Block in cuda. After calculating the index of the nearest center, select a thread to add up the related centers in one Block.

Parameters

- **cu_data** (*double**) – A chunk of points, which are given pointwise.
- **cu_centers** (*double**) – Current centers.
- **cu_centers_counter** (*int**) – Count how many points are nearest to a given center, count blockwise.
- **cu_new_centers** (*double**) – Calculate the sum of the points which are nearest to a given center, add blockwise.
- **cu_data_assigns** (*int**) – The index of the center which is nearest to a given point.
- **cluster_size** (*int**) – Number of clusters
- **dimension** (*int**) – Dimension of the points.
- **chunk_size** (*int**) – Number of points in the chunk.

Return chunk_centers_sum_cuda Summation of nearest centers in one block.

Return type double*

PyObject* **`kmeans_chunk_center_cuda`**(PyArrayObject **data*, PyArrayObject **centers*, PyObject **data_assigns*)

Record the nearest center of each point and renew the centers.

Parameters

- **data** (*PyArrayObject**) – Pointer to the point set to be calculated.
- **centers** (*PyArrayObject**) – Current centers.
- **data_assigns** (*PyObject**) – For each point record the index of the nearest center.

Returns The updated centers.

Return type PyObject*

Raises Exception No available device detected.

Raises Exception Compute compacity of the graphic card is not enough.

Raises Exception Only 1 device is supported currently.

Raises ValueError Parameters are of the wrong sizes.

Raises MemoryError RAM allocate Error. The imported data chunk may be too large.

Raises MemoryError RAM release error.

Raises MemoryError Graphic card RAM allocate error.

Raises MemoryError Graphic card RAM release error.

Raises MemoryError Error occurs when creating a new PyArray

2.5 opencl_kmeans

class `opencl.opencl_kmeans.`**`OpenCLKmeans`**(*metric=EuclideanMetric()*, *importer=None*, *chunk_size=1000*, *max_steps=100*)

An implementation of the k-means algorithm in OpenCL. Refer to the `DefaultKmeans` class for parameters and public methods.

2.6 kmeans_data_generator

class `kmeans_data_generator.`**`KmeansDataGenerator`**

Abstract data generator. Implementations are expected to override the generate_data method.

class `kmeans_data_generator.`**`KmeansRandomDataGenerator`**(*size*, *dimension*, *centers_count*)

Generate a test dataset for the k-means algorithm. The centers are generated uniformly. The other points are produced randomly near one of the centers with normal distribution.

Parameters

- **size** (*int*) – Number of data points to generate.
- **dimension** (*int*) – Dimension of the euclidean space the data points will belong to.
- **centers_count** (*int*) – Number of cluster centers around which the data points are to be generated.

`get_centers`()

Return the generated cluster centers.

Returns A list of numpy arrays representing the cluster centers.

Return type np.array[]

`get_data`()

Return the generated data points.

Returns A numpy array of size *size*x*dimension*.

Return type np.array

`to_binary_file`(*filename*)

Save the generated data to a binary file using `numpy.save()` which can be read later using the respective `CommonDataImporter` object.

Parameters **filename** (*str*) – The file name.

`to_file`(*filename*)

Save the generated data to a text file using `numpy.savetxt()` which can be read later using the respective `CommonDataImporter` object.

Parameters **filename** (*str*) – The file name.

CHAPTER THREE

THE HMM ALGORITHM

The implementation of the Hidden Markov Model algorithm consists of the following modules:

3.1 algorithms

`algorithms.`**`baum_welch`**`(`*ob, A, B, pi, accuracy=0.001, maxit=1000, kernel=<module 'spscicomp.hmm.kernel.python' from '/home/florian/PyCharmProjects/spscicomp/hmm/kernel/python.pyc'>, dtype=<type 'numpy.float32'>*`)`

Perform an optimization iteration with a given initial model.

Locally maximize P(O|A,B,pi) in a neighborhood of (A,B,pi) by iterating *update*. Stops if the probability does not change or the maximal iteration number is reached.

Parameters

- **ob** (*numpy.array shape (T)*) – observation sequence
- **A** (*numpy.array shape (N,N)*) – initial transition matrix
- **B** (*numpy.array shape (N,M)*) – initial symbol probabilities
- **pi** (*numpy.array shape (N)*) – initial distribution
- **accuracy** (*float, optional*) – ending criteria for the iteration
- **maxit** (*int, optional*) – ending criteria for the iteration
- **kernel** (*module, optional*) – module containing all functions to make calculations with
- **dtype** (*{ numpy.float32, numpy.float32 }, optional*) – datatype to be used for the matrices.

Returns

- **A** (*numpy.array shape (N,N)*) – new transition matrix
- **B** (*numpy.array shape (N,M)*) – new symbol probabilities
- **pi** (*numpy.array shape (N)*) – new initial distribution
- **new_probability** (*dtype*) – log P(O | A,B,pi)
- **it** (*int*) – number of iterations done

See also:

kernel.python, kernel.c, kernel.fortran : possible kernels baum_welch_multiple : perform optimization with multiple observations.

3.2 kernel

3.2.1 python

Python implementation of Hidden Markov Model kernel functions

This module is considered to be the reference for checking correctness of other kernels. All implementations are being kept very simple, straight forward and closely related to Rabiners [1] paper.

`kernel.python.`**`backward`**(*A*, *B*, *ob*, *scaling*, *dtype=<type 'numpy.float32'>*)

Compute all backward coefficients. With scaling!

Parameters

- **A** (*numpy.array of floating numbers and shape (N,N)*) – transition matrix of the hidden states
- **B** (*numpy.array of floating numbers and shape (N,M)*) – symbol probability matrix for each hidden state
- **ob** (*numpy.array of integers and shape (T)*) – observation sequence of integer between 0 and M, used as indices in B

Returns **beta** (*np.array of floating numbers and shape (T,N)*) – beta[t,i] is the ith forward coefficient of time t. These can be used in many different algorithms related to HMMs.

See also:

backward_no_scaling : Compute backward coefficients without scaling

`kernel.python.`**`backward_no_scaling`**(*A*, *B*, *ob*, *dtype=<type 'numpy.float32'>*)

Compute all backward coefficients. No scaling.

Parameters

- **A** (*numpy.array of floating numbers and shape (N,N)*) – transition matrix of the hidden states
- **B** (*numpy.array of floating numbers and shape (N,M)*) – symbol probability matrix for each hidden state
- **ob** (*numpy.array of integers and shape (T)*) – observation sequence of integer between 0 and M, used as indices in B

Returns **beta** (*np.array of floating numbers and shape (T,N)*) – beta[t,i] is the ith backward coefficient of time t

See also:

backward : Compute backward coefficients using given scaling factors.

`kernel.python.`**`draw_state`**(*distr*)

helper function for random_sequence to get the state to a given probability

Parameters **distr** (*array with probabilities where state are the indices*)

Returns **state** (*integer*) – which randomly chosen with given distribution

`kernel.python.`**`forward`**(*A*, *B*, *pi*, *ob*, *dtype=<type 'numpy.float32'>*)

Compute P(ob|A,B,pi) and all forward coefficients. With scaling!

Parameters

- **A** (*numpy.array of floating numbers and shape (N,N)*) – transition matrix of the hidden states
- **B** (*numpy.array of floating numbers and shape (N,M)*) – symbol probability matrix for each hidden state

- **pi** (*numpy.array of floating numbers and shape (N)*) – initial distribution
- **ob** (*numpy.array of integers and shape (T)*) – observation sequence of integer between 0 and M, used as indices in B

Returns

- **prob** (*floating number*) – The probability to observe the sequence *ob* with the model given by *A*, *B* and *pi*.
- **alpha** (*np.array of floating numbers and shape (T,N)*) – alpha[t,i] is the ith forward coefficient of time t. These can be used in many different algorithms related to HMMs.
- **scaling** (*np.array of floating numbers and shape (T)*) – scaling factors for each step in the calculation. can be used to rescale backward coefficients.

See also:

forward_no_scaling : Compute forward coefficients without scaling

`kernel.python.`**`forward_no_scaling`**(*A*, *B*, *pi*, *ob*, *dtype=<type 'numpy.float32'>*)

Compute P(ob|A,B,pi) and all forward coefficients. No scaling done.

Parameters

- **A** (*numpy.array of floating numbers and shape (N,N)*) – transition matrix of the hidden states
- **B** (*numpy.array of floating numbers and shape (N,M)*) – symbol probability matrix for each hidden state
- **pi** (*numpy.array of floating numbers and shape (N)*) – initial distribution
- **ob** (*numpy.array of integers and shape (T)*) – observation sequence of integer between 0 and M, used as indices in B

Returns

- **prob** (*floating number*) – The probability to observe the sequence *ob* with the model given by *A*, *B* and *pi*.
- **alpha** (*numpy.array of floating numbers and shape (T,N)*) – alpha[t,i] is the ith forward coefficient of time t. These can be used in many different algorithms related to HMMs.

See also:

forward : Compute forward coefficients and scaling factors

`kernel.python.`**`random_sequence`**(*A*, *B*, *pi*, *T*)

Generate an observation sequence of length T from the model A, B, pi.

Parameters

- **A** (*numpy.array shape (N,N)*) – transition matrix of the model
- **B** (*numpy.array shape (N,M)*) – symbol probability matrix of the model
- **pi** (*numpy.array shape (N)*) – starting probability vector of the model
- **T** (*integer*) – length of generated observation sequence

Returns **obs** (*numpy.array shape (T)*) – observation sequence containing only symbols, i.e. ints in [0,M)

Notes

This function relies on the function draw_state(distr).

See also:

draw_state [draw the index of the state, obeying the probability] distribution vector distr

`kernel.python.`**`state_counts`**`(`*gamma*, *T*, *dtype=<type 'numpy.float32'>*`)`
Sum the probabilities of being in state i to time t

Parameters

- **gamma** (*numpy.array shape (T,N)*) – gamma[t,i] is the probabilty at time t to be in state i !
- **T** (*number of observationsymbols*)
- **dtype** (*item datatype [optional]*)

Returns **count** (*numpy.array shape (N)*) – count[i] is the summed probabilty to be in state i !

Notes

This function is independ of alpha and beta being scaled, as long as their scaling is independ in i.

See also:

forward, forward_no_scaling : to calculate *alpha* backward, backward_no_scaling : to calculate *beta*

`kernel.python.`**`state_probabilities`**`(`*alpha*, *beta*, *dtype=<type 'numpy.float32'>*`)`
Calculate the (T,N)-probabilty matrix for being in state i at time t.

Parameters

- **alpha** (*numpy.array shape (T,N)*) – forward coefficients
- **beta** (*numpy.array shape (T,N)*) – backward coefficients
- **dtype** (*item datatype [optional]*)

Returns **gamma** (*numpy.array shape (T,N)*) – gamma[t,i] is the probabilty at time t to be in state i !

Notes

This function is independ of alpha and beta being scaled, as long as their scaling is independ in i.

See also:

forward, forward_no_scaling : to calculate *alpha* backward, backward_no_scaling : to calculate *beta*

`kernel.python.`**`symbol_counts`**`(`*gamma*, *ob*, *M*, *dtype=<type 'numpy.float32'>*`)`
Sum the observed probabilities to see symbol k in state i.

Parameters

- **gamma** (*numpy.array shape (T,N)*) – gamma[t,i] is the probabilty at time t to be in state i !
- **ob** (*numpy.array shape (T)*)
- **M** (*integer. number of possible observationsymbols*)
- **dtype** (*item datatype, optional*)

Returns **counts** (*numpy.array shape (N,M)*)

Notes

This function is independ of alpha and beta being scaled, as long as their scaling is independ in i.

See also:

forward, forward_no_scaling : to calculate *alpha* backward, backward_no_scaling : to calculate *beta*

`kernel.python.`**`transition_counts`**(*alpha*, *beta*, *A*, *B*, *ob*, *dtype=<type 'numpy.float32'>*)

Sum for all t the probability to transition from state i to state j.

Parameters

- **alpha** (*numpy.array shape (T,N)*) – forward coefficients
- **beta** (*numpy.array shape (T,N)*) – backward coefficients
- **A** (*numpy.array shape (N,N)*) – transition matrix of the model
- **B** (*numpy.array shape (N,M)*) – symbol probabilty matrix of the model
- **ob** (*numpy.array shape (T)*) – observation sequence containing only symbols, i.e. ints in [0,M)
- **dtype** (*item datatype [optional]*)

Returns **counts** (*numpy.array shape (N, N)*) – counts[i, j] is the summed probability to transition from i to j int time [0,T)

Notes

It does not matter if alpha or beta scaled or not, as long as there scaling does not depend on the second variable.

See also:

transition_probabilities : return the matrix of transition probabilities forward : calculate forward coefficients *alpha* backward : calculate backward coefficients *beta*

`kernel.python.`**`transition_probabilities`**(*alpha*, *beta*, *A*, *B*, *ob*, *dtype=<type 'numpy.float32'>*)

Compute for each t the probability to transition from state i to state j.

Parameters

- **alpha** (*numpy.array shape (T,N)*) – forward coefficients
- **beta** (*numpy.array shape (T,N)*) – backward coefficients
- **A** (*numpy.array shape (N,N)*) – transition matrix of the model
- **B** (*numpy.array shape (N,M)*) – symbol probabilty matrix of the model
- **ob** (*numpy.array shape (T)*) – observation sequence containing only symbols, i.e. ints in [0,M)
- **dtype** (*item datatype [optional]*)

Returns **xi** (*numpy.array shape (T-1, N, N)*) – xi[t, i, j] is the probability to transition from i to j at time t.

Notes

It does not matter if alpha or beta scaled or not, as long as there scaling does not depend on the second variable.

See also:

state_counts : calculate the probability to be in state i at time t forward : calculate forward coefficients *alpha* backward : calculate backward coefficients *beta*

`kernel.python.`**`update`**(*gamma*, *xi*, *ob*, *M*, *dtype=<type 'numpy.float32'>*)

Return an updated model for given state and transition counts.

Parameters

- **gamma** (*numpy.array shape (T,N)*) – state probabilities for each t
- **xi** (*numpy.array shape (T,N,N)*) – transition probabilities for each t
- **ob** (*numpy.array shape (T)*) – observation sequence containing only symbols, i.e. ints in [0,M)

Returns

- **A** (*numpy.array (N,N)*) – new transition matrix
- **B** (*numpy.array (N,M)*) – new symbol probabilities
- **pi** (*numpy.array (N)*) – new initial distribution
- **dtype** (*{ nupmy.float64, numpy.float32 }, optional*)

Notes

This function is part of the Baum-Welch algorithm for a single observation.

See also:

state_probabilities : to calculate *gamma* transition_probabilities : to calculate *xi*

CHAPTER

FOUR

THE AMUSE ALGORITHM

The implementation of the Amuse algorithm consists of the following modules:

4.1 Tica_Amuse

class `Tica_Amuse.`**`TicaAmuse`**(*i_inFileName=None*, *i_outFileName='../testdata/tica_independentComp.npy'*, *i_addEps=1e-09*, *i_timeLag=1*, *i_useDampingAdapt=True*)

Implementation of AMUSE-Algorithm, which basically uses functionality of `TicaPrinComp`. A `CommonDataImporter` object to be used for importing the numerical data.

Parameters

- **i_inFileName** (*string*) – A filename of the binary file which loads the data
- **i_outFileName** (*string*) – A filename of the binary file which stored the results.
- **i_addEps** (*float*) – A damping parameter to avoid dividing by zero in the normalization part of the amuse algorithm.
- **i_timeLag** (*int*) – In this setting the data has time-dependencies where i_timeLag is some lag constant.
- **i_useDampingAdapt** – A Boolean flag to use a method for adapting the damping parameter *i_addEps*.

Default setting is *True* :type bool

`performAmuse`(*i_numDomComp=1*)

Runs the AMUSE-Algorithm and stores the results in a binary file with the stated filename(see above *i_outFileName*). Performs the hole algorithm only for one given time-lag. If needed you can run the method again for another time-lag by setting the time-lag with the function `setTimeLag()`.

Parameters **i_numDomComp** (*int*) – Number of independent components which are needed.

`setTimeLag`(*i_timeLag*)

Set the time-lag. :param i_timeLag: New time-lag. :type i_timeLag: int

4.2 Tica_PrincipleComp

class `Tica_PrincipleComp.`**`TicaPrinComp`**(*i_inFileName=None*, *i_outFileName='../testdata/tica_tempOutput.npy'*, *i_addEpsilon=1e-09*, *i_timeLag=1*, *i_useDampingAdapt=True*)

Implementation of computation of principle components highly adapted for the amuse algorithm. The class

`TicaPrinComp` contains the subclass `TicaPrinCompTimeLagged`. A `CommonDataImporter` object to be used for importing the numerical data.

Parameters

- **i_inFileName** (*string*) – A filename of the binary file which loads the data
- **i_outFileName** (*string*) – A filename of the binary file in which the results are stored
- **i_addEpsilon** – A damping parameter to avoid dividing by zero in the normalization part of the amuse algorithm.
- **i_timeLag** (*int*) – In this setting the data has time-dependencies where i_timeLag is some lag constant.
- **i_useDampingAdapt** – A Boolean flag to use a method for adapting the damping parameter *i_addEps*.

Default setting is *True* :type bool

class **`TicaPrinCompTimeLagged`**(*i_ticaPrinComp*)

A subclass contained in `TicaPrinComp`. This class contains the time-lagged relevant implementations of the AMUSE-Algorithm. Especially it is implemented the computation of the time-lagged covariance matrix C^{τ}. The class `TicaPrinCompTimeLagged` also performs the transformations, which are leads to the independent components Z.

Parameters **i_ticaPrinComp** (`TicaPrinComp`) – A `TicaPrinComp` object as input parameter to indicate the dependencies.

`computeCovariance`()

Computes the time-lagged covariance matrix C^{τ} with $c_{ij}^{\tau} = \frac{1}{N-\tau-1}\sum_{t=1}^{N-\tau} x_{it}x_{jt+\tau}$

`computeICs`(*i_numDomComp=1*)

Main method in the class `TicaPrinCompTimeLagged`. Computes the independent components on the basis of normalized principle components supplied by `TicaPrinComp`.

Parameters **i_numDumComp** – Number of needed independent components.

`performTransformation`(*i_domComp*)

Computes the independent components and saves needed components in a output numpy binary file. :param i_domComp: Needed components of TICA :type i_domComp: int

`setTimeLag`(*i_timeLag*)

Set a new time-lag value. :param i_timeLag: :type: int

`symmetrizeCovariance`()

Symmetrizes the time-lagged covariance matrix C^{τ} by $C_{sym}^{\tau} = \frac{1}{2}\left[C^{\tau} + \left(C^{\tau}\right)^{T}\right]$

`TicaPrinComp.`**`calcNumbOfDomComps`**(*i_amountOfTotalVariance*)

Not used actually! This function returns, by a given amount of the total variance, the number of relevant principle components.

Parameters **i_amountOfTotalVariance** (*float*) – Amount of total variance.

:return : Return the number of dominant principle components. :rtype: int

`TicaPrinComp.`**`computeChunkSize`**()

This function computes in a naive way the chunk size, which is used to load a data chunk from the memory map of `CommonDataImporter`. If the file size of the input file is greater than the threshold a bisection of the number of rows of the input file will be performed.

`TicaPrinComp.`**`computeColMeans`**()

Computes mean per column of the input data.

TicaPrinComp.**computeCovariance**()

This function computes chunk by chunk the instantaneous covariance matrix C. If a c-extension is available, then it will used by the flag *use_extension*

TicaPrinComp.**computePCs**(*i_dataChunk*, *i_domComp*)

This function computes the principle components of the input data X. Let Γ be the matrix which contains the ordered eigenvectors of the instantaneous covariance matrix C. The principle components Y are computed by $Y = \Gamma^T(X - mean(X))$.

Parameters

- **i_dataChunk** (*numpy.array*) – A data chunk as a subset of hole data.
- **i_domComp** (*int*) – Number of dominated components which are considered.

Return o_pc The principle components Y.

Return type numpy.array

TicaPrinComp.**naiveDampingParamAdapt**()

This function adapts possible singular eigenvalues of the covariance matrix. This is done in a naive way by adding small constants to the effect that small negative eigenvalues become positive and eigenvalues which are *nan* will be set on a small not negative number. :return:

TicaPrinComp.**normalizePCs**(*i_pcsChunk*)

This function computes the normalizes the principle components Y of the input data X. Let Λ be a diagonal matrix with the ordered eigenvalues of the instantaneous #covariance matrix C. The normalized principle components $\tilde{Y}$ are computed by $\tilde{Y} = \Lambda^{-\frac{1}{2}}Y)$.

Parameters **i_pcsChunk** (*numpy.array*) – A chunk of principle components.

Return o_pcNorm The normalized principle components $\tilde{Y}$.

Return type numpy.array

4.3 Tica_EigenDecomp

class Tica_EigenDecomp.**TicaEigenDecomp**(*i_matrix*)

A class that post processes the results of the eigen decomposition of `numpy.linalg.eig`. Performs a reordering of the eigen values and the corresponding eigen vectors. Here are considered only real eigen values.

Parameters **i_matrix** (*numpy.array[]*) – A matrix from which the eigen decomposition will realized.

computeEigenDecomp(*i_matrix*)

Performs the eigen decomposition of `numpy.linalg` and rearrange(see `reorderingEigenV()`) the eigenvalues if they are real. If the eigenvalues have a imaginary part a warning will logged, the imaginary parts will be set to zero and the real parts will be rearranged.

reorderingEigenV()

Reordering of eigenvalues and corresponding eigenvectors by the largest eigenvalue. Ordering only for real eigenvalues.

4.4 Tica_DataImport

class Tica_DataImport.**TicaDataImport**(*i_fileName*)

Simple class which import csv data with the `numpy` function `numpy.loadtxt()`.

getData()

Return the loaded csv data.

Returns Return a numpy matrix.

Return type numpy.matrix

readTable(*i_fileName*)

Read a csv file.

Parameters **i_fileName** (*string*) – A filename of the csv file.

CHAPTER

FIVE

HOW TO BUILD THE DOCUMENTATION

1. Merge your code branch into the dev branch.

   ```
   $ git checkout dev
   $ git merge <YOUR BRANCH>
   ```

2. Edit the rst files in docs/ to include your documentation in the code. Python, NumPy and Google style documentation markup is supported.

3. In the docs/ folder, run the compilation:

   ```
   $ make html
   ```

4. Commit the documentation changes. Be careful to not include any other changes you may have done, as we will cherry-pick this commit into the GitHub Pages branch.

   ```
   $ git add docs/
   $ git commit -m "updated documentation."
   ```

5. Switch to the gh-pages branch and cherry-pick the commit into this branch.

   ```
   $ git checkout gh-pages
   $ git cherry-pick <COMMIT ID>
   ```

6. Push!

   ```
   $ git push
   ```

CHAPTER SIX

INDICES AND TABLES

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

PYTHON MODULE INDEX

INDEX