msmtools Documentation

Release 1.2.4+0.g54dc76d.dirty

CMB group

Nov 12, 2018

Contents

MSMTools is a Python library for the estimation, validation and analysis Markov state models.

It supports the following main features:

- Markov state model (MSM) estimation and validation.
- Computing Metastable states and structures with Perron-cluster cluster analysis (PCCA).
- Systematic coarse-graining of MSMs to transition models with few states.
- Extensive analysis options for MSMs, e.g. calculation of committors, mean first passage times, transition rates, experimental expectation values and time-correlation functions, etc.
- Transition Path Theory (TPT).

For a high-level interface to these functionalities, we encourage you to use PyEMMA.

Technical features:

- Code is implemented in Python (supports 2.7, 3.3/3.4) and C.
- Runs on Linux (64 bit), Windows (32 or 64 bit) or MacOS (64 bit).

CHAPTER 1

Documentation

1.1 Installation

To install the msmtools Python package, you need a few Python package dependencies. If these dependencies are not available in their required versions, the installation will fail. We recommend one particular way for the installation that is relatively safe, but you are welcome to try another approaches if you know what you are doing.

1.1.1 Anaconda install (Recommended)

We strongly recommend to use the Anaconda scientific python distribution in order to install python-based software, including msmtools. Python-based software is not trivial to distribute and this approach saves you many headaches and problems that frequently arise in other installation methods. You are free to use a different approach (see below) if you know how to sort out problems, but play at your own risk.

If you already have a conda installation, directly go to step 3:

1. Download and install miniconda for Python 2 or 3, 32 or 64 bit depending on your system: http://conda.pydata.org/miniconda.html

   For Windows users, who do not know what to choose for 32 or 64 bit, it is strongly recommended to read the second question of this FAQ first: http://windows.microsoft.com/en-us/windows/32-bit-and-64-bit-windows

   Run the installer and select **yes** to add conda to the **PATH** variable.

2. If you have installed from a Linux shell, either open a new shell to have an updated PATH, or update your PATH variable by `source ~/.bashrc` (or .tcsh, .csh - whichever shell you are using).

3. Add the omnia-md software channel, and install (or update) msmtools:

   ```
   conda config --add channels omnia
   conda install msmtools
   ```

   if the command conda is unknown, the PATH variable is probably not set correctly (see 1. and 2.)

4. Check installation:

```
conda list
```

shows you the installed python packages. You should find a msmtools 1.1 (or later) and ipython, ipython-notebook 3.1 (or later). If ipython is not up to date, you can still use msmtools, but you won't be able to load our example notebooks. In that case, update it by

```
conda install ipython-notebook
```

1.1.2 Python Package Index (PyPI)

If you do not like Anaconda for some reason you should use the Python package manager **pip** to install. This is not recommended, because in the past, various problems have arisen with pip in compiling the packages that msmtools depends upon.

1. If you do not have pip, please read the install guide: install guide.
2. Make sure pip is enabled to install so called wheel packages:

   ```
   pip install wheel
   ```

   Now you are able to install binaries if you use MacOSX or Windows. At the moment of writing PyPI does not support Linux binaries at all, so Linux users have to compile by themselves.
3. Install msmtools using

   ```
   pip install msmtools
   ```

4. Check your installation

   ```
   python
   >>> import msmtools
   >>> msmtools.__version__
   ```

   should print 1.1 or later

   ```
   >>> import IPython
   >>> IPython.__version__
   ```

   should print 3.1 or later. If ipython is not up to date, update it by `pip install ipython`

1.1.3 Building from Source

If you refuse to use Anaconda, you will build msmtools from the source. In this approach, all msmtools dependencies will be built from the source too. Building these from source is sometimes (if not usually) tricky, takes a long time and is error prone - though it is **not** recommended nor supported by us. If unsure, use the anaconda installation.

1. Ensure that you fulfill the following prerequisites. You can install either with pip or conda, as long as you met the version requirements.
   - C/C++ compiler
   - setuptools > 18
   - cython >= 0.22
   - numpy >= 1.6

- scipy >= 0.11

If you do not fulfill these requirements, try to upgrade all packages:

```
pip install --upgrade setuptools
pip install --upgrade cython
pip install --upgrade numpy
pip install --upgrade scipy
```

Note that if pip finds a newer version, it will trigger an update which will most likely involve compilation. Especially NumPy and SciPy are hard to build. You might want to take a look at this guide here: http://www.scipy.org/scipylib/building/

2. The build and install process is in one step, as all dependencies are dragged in via the provided *setup.py* script. So you only need to get the source of Emma and run it to build Emma itself and all of its dependencies (if not already supplied) from source.

```
pip install msmtools
```

1.1.4 For Developers

If you are a developer, clone the code repository from GitHub and install it as follows

1. Ensure the prerequisites (point 1) described for "Building from Source" above.
2. Make a suitable directory, and inside clone the repository via

```
git clone https://github.com/markovmodel/msmtools.git
```

3. install msmtools via

```
python setup.py develop [--user]
```

The develop install has the advantage that if only python scripts are being changed e.g. via an pull or a local edit, you do not have to re-install anything, because the setup command simply created a link to your working copy. Repeating point 3 is only necessary if any of msmtools C-files change and need to be rebuilt.

1.2 Documentation

msmtools provides these packages to perform estimation and further analysis of Markov models.

1.2.1 estimation - MSM estimation from data (`msmtools.estimation`)

Countmatrix

count_matrix(dtraj, lag[, sliding, ...])	Generate a count matrix from given microstate trajectory.
cmatrix(dtraj, lag[, sliding, ...])	Generate a count matrix from given microstate trajectory.

msmtools.estimation.count_matrix

msmtools.estimation.**count_matrix**(*dtraj*, *lag*, *sliding=True*, *sparse_return=True*, *nstates=None*)

Generate a count matrix from given microstate trajectory.

Parameters

- **dtraj** (*array_like or list of array_like*) – Discretized trajectory or list of discretized trajectories
- **lag** (*int*) – Lagtime in trajectory steps
- **sliding** (*bool, optional*) – If true the sliding window approach is used for transition counting.
- **sparse_return** (*bool (optional)*) – Whether to return a dense or a sparse matrix.
- **nstates** (*int, optional*) – Enforce a count-matrix with shape=(nstates, nstates)

Returns **C** – The count matrix at given lag in coordinate list format.

Return type scipy.sparse.coo_matrix

Notes

Transition counts can be obtained from microstate trajectory using two methods. Couning at lag and sliding-window counting.

Lag

This approach will skip all points in the trajectory that are seperated form the last point by less than the given lagtime τ.

Transition counts $c_{ij}(\tau)$ are generated according to

$$c_{ij}(\tau) = \sum_{k=0}^{\lfloor \frac{N}{\tau} \rfloor - 2} \chi_i(X_{k\tau}) \chi_j(X_{(k+1)\tau}).$$

$\chi_i(x)$ is the indicator function of i, i.e $\chi_i(x) = 1$ for $x = i$ and $\chi_i(x) = 0$ for $x \neq i$.

Sliding

The sliding approach slides along the trajectory and counts all transitions sperated by the lagtime τ.

Transition counts $c_{ij}(\tau)$ are generated according to

$$c_{ij}(\tau) = \sum_{k=0}^{N-\tau-1} \chi_i(X_k) \chi_j(X_{k+\tau}).$$

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import count_matrix
```

```
>>> dtraj = np.array([0, 0, 1, 0, 1, 1, 0])
>>> tau = 2
```

Use the sliding approach first

```
>>> C_sliding = count_matrix(dtraj, tau)
```

The generated matrix is a sparse matrix in CSR-format. For convenient printing we convert it to a dense ndarray.

```
>>> C_sliding.toarray()
array([[ 1.,  2.],
       [ 1.,  1.]])
```

Let us compare to the count-matrix we obtain using the lag approach

```
>>> C_lag = count_matrix(dtraj, tau, sliding=False)
>>> C_lag.toarray()
array([[ 0.,  1.],
       [ 1.,  1.]])
```

msmtools.estimation.cmatrix

msmtools.estimation.**cmatrix**(*dtraj*, *lag*, *sliding=True*, *sparse_return=True*, *nstates=None*)

Generate a count matrix from given microstate trajectory.

Parameters

- **dtraj** (*array_like or list of array_like*) – Discretized trajectory or list of discretized trajectories
- **lag** (*int*) – Lagtime in trajectory steps
- **sliding** (*bool, optional*) – If true the sliding window approach is used for transition counting.
- **sparse_return** (*bool (optional)*) – Whether to return a dense or a sparse matrix.
- **nstates** (*int, optional*) – Enforce a count-matrix with shape=(nstates, nstates)

Returns **C** – The count matrix at given lag in coordinate list format.

Return type scipy.sparse.coo_matrix

Notes

Transition counts can be obtained from microstate trajectory using two methods. Couning at lag and sliding-window counting.

Lag

This approach will skip all points in the trajectory that are seperated form the last point by less than the given lagtime τ.

Transition counts $c_{ij}(\tau)$ are generated according to

$$c_{ij}(\tau) = \sum_{k=0}^{\lfloor \frac{N}{\tau} \rfloor - 2} \chi_i(X_{k\tau}) \chi_j(X_{(k+1)\tau}).$$

$\chi_i(x)$ is the indicator function of i, i.e $\chi_i(x) = 1$ for $x = i$ and $\chi_i(x) = 0$ for $x \neq i$.

Sliding

The sliding approach slides along the trajectory and counts all transitions sperated by the lagtime τ.

Transition counts $c_{ij}(\tau)$ are generated according to

$$c_{ij}(\tau) = \sum_{k=0}^{N-\tau-1} \chi_i(X_k)\chi_j(X_{k+\tau}).$$

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import count_matrix
```

```
>>> dtraj = np.array([0, 0, 1, 0, 1, 1, 0])
>>> tau = 2
```

Use the sliding approach first

```
>>> C_sliding = count_matrix(dtraj, tau)
```

The generated matrix is a sparse matrix in CSR-format. For convenient printing we convert it to a dense ndarray.

```
>>> C_sliding.toarray()
array([[ 1.,  2.],
       [ 1.,  1.]])
```

Let us compare to the count-matrix we obtain using the lag approach

```
>>> C_lag = count_matrix(dtraj, tau, sliding=False)
>>> C_lag.toarray()
array([[ 0.,  1.],
       [ 1.,  1.]])
```

Connectivity

connected_sets(C[, directed])	Compute connected sets of microstates.
largest_connected_set(C[, directed])	Largest connected component for a directed graph with edge-weights given by the count matrix.
largest_connected_submatrix(C[, directed, lcc])	Compute the count matrix on the largest connected set.
connected_cmatrix(C[, directed, lcc])	Compute the count matrix on the largest connected set.
is_connected(C[, directed])	Check connectivity of the given matrix.

msmtools.estimation.connected_sets

msmtools.estimation.**connected_sets**(*C*, *directed=True*)

Compute connected sets of microstates.

Connected components for a directed graph with edge-weights given by the count matrix.

Parameters

- **C** (*scipy.sparse matrix*) – Count matrix specifying edge weights.
- **directed** (*bool, optional*) – Whether to compute connected components for a directed or undirected graph. Default is True.

Returns **cc** – Each entry is an array containing all vertices (states) in the corresponding connected component. The list is sorted according to the size of the individual components. The largest connected set is the first entry in the list, lcc=cc[0].

Return type list of arrays of integers

Notes

Viewing the count matrix as the adjacency matrix of a (directed) graph the connected components are given by the connected components of that graph. Connected components of a graph can be efficiently computed using Tarjan's algorithm.

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import connected_sets
```

```
>>> C = np.array([[10, 1, 0], [2, 0, 3], [0, 0, 4]])
>>> cc_directed = connected_sets(C)
>>> cc_directed
[array([0, 1]), array([2])]
```

```
>>> cc_undirected = connected_sets(C, directed=False)
>>> cc_undirected
[array([0, 1, 2])]
```

msmtools.estimation.largest_connected_set

msmtools.estimation.**largest_connected_set**(*C*, *directed=True*)

Largest connected component for a directed graph with edge-weights given by the count matrix.

Parameters

- **C** (*scipy.sparse matrix*) – Count matrix specifying edge weights.
- **directed** (*bool, optional*) – Whether to compute connected components for a directed or undirected graph. Default is True.

Returns **lcc** – The largest connected component of the directed graph.

Return type array of integers

See also:

connected_sets()

Notes

Viewing the count matrix as the adjacency matrix of a (directed) graph the largest connected set is the largest connected set of nodes of the corresponding graph. The largest connected set of a graph can be efficiently computed using Tarjan's algorithm.

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import largest_connected_set
```

```
>>> C =  np.array([[10, 1, 0], [2, 0, 3], [0, 0, 4]])
>>> lcc_directed = largest_connected_set(C)
>>> lcc_directed
array([0, 1])
```

```
>>> lcc_undirected = largest_connected_set(C, directed=False)
>>> lcc_undirected
array([0, 1, 2])
```

msmtools.estimation.largest_connected_submatrix

msmtools.estimation.**largest_connected_submatrix**(*C*, *directed=True*, *lcc=None*)

Compute the count matrix on the largest connected set.

Parameters

- **C** (*scipy.sparse matrix*) – Count matrix specifying edge weights.
- **directed** (*bool, optional*) – Whether to compute connected components for a directed or undirected graph. Default is True
- **lcc** (*(M,) ndarray, optional*) – The largest connected set

Returns **C_cc** – Count matrix of largest completely connected set of vertices (states)

Return type scipy.sparse matrix

See also:

largest_connected_set()

Notes

Viewing the count matrix as the adjacency matrix of a (directed) graph the larest connected submatrix is the adjacency matrix of the largest connected set of the corresponding graph. The largest connected submatrix can be efficiently computed using Tarjan's algorithm.

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import largest_connected_submatrix
```

```
>>> C = np.array([[10, 1, 0], [2, 0, 3], [0, 0, 4]])
```

```
>>> C_cc_directed = largest_connected_submatrix(C)
>>> C_cc_directed
array([[10,  1],
       [ 2,  0]]...)
```

```
>>> C_cc_undirected = largest_connected_submatrix(C, directed=False)
>>> C_cc_undirected
array([[10,  1,  0],
       [ 2,  0,  3],
       [ 0,  0,  4]]...)
```

msmtools.estimation.connected_cmatrix

msmtools.estimation.**connected_cmatrix**(*C*, *directed=True*, *lcc=None*)

Compute the count matrix on the largest connected set.

Parameters

- **C** (*scipy.sparse matrix*) – Count matrix specifying edge weights.
- **directed** (*bool, optional*) – Whether to compute connected components for a directed or undirected graph. Default is True
- **lcc** (*(M,) ndarray, optional*) – The largest connected set

Returns **C_cc** – Count matrix of largest completely connected set of vertices (states)

Return type scipy.sparse matrix

See also:

largest_connected_set()

Notes

Viewing the count matrix as the adjacency matrix of a (directed) graph the larest connected submatrix is the adjacency matrix of the largest connected set of the corresponding graph. The largest connected submatrix can be efficiently computed using Tarjan's algorithm.

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import largest_connected_submatrix
```

```
>>> C = np.array([[10, 1, 0], [2, 0, 3], [0, 0, 4]])
```

```
>>> C_cc_directed = largest_connected_submatrix(C)
>>> C_cc_directed
array([[10,  1],
       [ 2,  0]]...)
```

```
>>> C_cc_undirected = largest_connected_submatrix(C, directed=False)
>>> C_cc_undirected
array([[10,  1,  0],
       [ 2,  0,  3],
       [ 0,  0,  4]]...)
```

msmtools.estimation.is_connected

msmtools.estimation.**is_connected**(*C*, *directed=True*)

Check connectivity of the given matrix.

Parameters

- **C** (*scipy.sparse matrix*) – Count matrix specifying edge weights.
- **directed** (*bool, optional*) – Whether to compute connected components for a directed or undirected graph. Default is True.

Returns **is_connected** – True if C is connected, False otherwise.

Return type bool

See also:

largest_connected_submatrix()

Notes

A count matrix is connected if the graph having the count matrix as adjacency matrix has a single connected component. Connectivity of a graph can be efficiently checked using Tarjan's algorithm.

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import is_connected
```

```
>>> C = np.array([[10, 1, 0], [2, 0, 3], [0, 0, 4]])
>>> is_connected(C)
False
```

```
>>> is_connected(C, directed=False)
True
```

Estimation

transition_matrix(C[, reversible, mu, method])	Estimate the transition matrix from the given countmatrix.
tmatrix(C[, reversible, mu, method])	Estimate the transition matrix from the given countmatrix.
rate_matrix(C[, dt, method, sparsity, ...])	Estimate a reversible rate matrix from a count matrix.
log_likelihood(C, T)	Log-likelihood of the count matrix given a transition matrix.
tmatrix_cov(C[, k])	Covariance tensor for non-reversible transition matrix posterior.
error_perturbation(C, S)	Error perturbation for given sensitivity matrix.

msmtools.estimation.transition_matrix

msmtools.estimation.**transition_matrix**(*C*, *reversible=False*, *mu=None*, *method='auto'*, ***kwargs*)

Estimate the transition matrix from the given countmatrix.

Parameters

- **C** (*numpy ndarray or scipy.sparse matrix*) – Count matrix
- **reversible** (*bool (optional)*) – If True restrict the ensemble of transition matrices to those having a detailed balance symmetry otherwise the likelihood optimization is carried out over the whole space of stochastic matrices.
- **mu** (*array_like*) – The stationary distribution of the MLE transition matrix.
- **method** (*str*) – Select which implementation to use for the estimation. One of 'auto', 'dense' and 'sparse', optional, default='auto'. 'dense' always selects the dense implementation, 'sparse' always selects the sparse one. 'auto' selects the most efficient implementation according to the sparsity structure of the matrix: if the occupation of the C matrix is less then one third, select sparse. Else select dense. The type of the T matrix returned always matches the type of the C matrix, irrespective of the method that was used to compute it.
- ****kwargs** (*Optional algorithm-specific parameters. See below for special cases*) –
- **Xinit** (*(M, M) ndarray*) – Optional parameter with reversible = True. initial value for the matrix of absolute transition probabilities. Unless set otherwise, will use X = diag(pi) t, where T is a nonreversible transition matrix estimated from C, i.e. T_ij = c_ij / sum_k c_ik, and pi is its stationary distribution.
- **maxiter** (*1000000 : int*) – Optional parameter with reversible = True. maximum number of iterations before the method exits

- **maxerr** (*1e-8 : float*) – Optional parameter with reversible = True. convergence tolerance for transition matrix estimation. This specifies the maximum change of the Euclidean norm of relative stationary probabilities ($x_i = \sum_k x_{ik}$). The relative stationary probability changes $e_i = (x_i^{(1)} - x_i^{(2)})/(x_i^{(1)} + x_i^{(2)})$ are used in order to track changes in small probabilities. The Euclidean norm of the change vector, $|e_i|_2$, is compared to maxerr.
- **rev_pisym** (*bool, default=False*) – Fast computation of reversible transition matrix by normalizing $x_{ij} = \pi_i p_{ij} + \pi_j p_{ji}$. p_{ij} is the direct (nonreversible) estimate and pi_i is its stationary distribution. This estimator is asympotically unbiased but not maximum likelihood.
- **return_statdist** (*bool, default=False*) – Optional parameter with reversible = True. If set to true, the stationary distribution is also returned
- **return_conv** (*bool, default=False*) – Optional parameter with reversible = True. If set to true, the likelihood history and the pi_change history is returned.
- **warn_not_converged** (*bool, default=True*) – Prints a warning if not converged.
- **sparse_newton** (*bool, default=False*) – If True, use the experimental primal-dual interior-point solver for sparse input/computation method.

Returns

- **P** (*(M, M) ndarray or scipy.sparse matrix*) – The MLE transition matrix. P has the same data type (dense or sparse) as the input matrix C.
- *The reversible estimator returns by default only P, but may also return*
- *(P,pi) or (P,lhist,pi_changes) or (P,pi,lhist,pi_changes) depending on the return settings*
- **P** (*ndarray (n,n)*) – transition matrix. This is the only return for return_statdist = False, return_conv = False
- **(pi)** (*ndarray (n)*) – stationary distribution. Only returned if return_statdist = True
- **(lhist)** (*ndarray (k)*) – likelihood history. Has the length of the number of iterations needed. Only returned if return_conv = True
- **(pi_changes)** (*ndarray (k)*) – history of likelihood history. Has the length of the number of iterations needed. Only returned if return_conv = True

Notes

The transition matrix is a maximum likelihood estimate (MLE) of the probability distribution of transition matrices with parameters given by the count matrix.

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import transition_matrix
```

```
>>> C = np.array([[10, 1, 1], [2, 0, 3], [0, 1, 4]])
```

Non-reversible estimate

```
>>> T_nrev = transition_matrix(C)
>>> T_nrev
array([[ 0.83333333,  0.08333333,  0.08333333],
       [ 0.4       ,  0.        ,  0.6       ],
       [ 0.        ,  0.2       ,  0.8       ]])
```

Reversible estimate

```
>>> T_rev = transition_matrix(C, reversible=True)
>>> T_rev
array([[ 0.83333333,  0.10385551,  0.06281115],
       [ 0.35074677,  0.        ,  0.64925323],
       [ 0.04925323,  0.15074677,  0.8       ]])
```

Reversible estimate with given stationary vector

```
>>> mu = np.array([0.7, 0.01, 0.29])
>>> T_mu = transition_matrix(C, reversible=True, mu=mu)
>>> T_mu
array([[ 0.94771371,  0.00612645,  0.04615984],
       [ 0.42885157,  0.        ,  0.57114843],
       [ 0.11142031,  0.01969477,  0.86888491]])
```

msmtools.estimation.tmatrix

msmtools.estimation.**tmatrix**(*C*, *reversible=False*, *mu=None*, *method='auto'*, ***kwargs*)

Estimate the transition matrix from the given countmatrix.

Parameters

- **C** (*numpy ndarray or scipy.sparse matrix*) – Count matrix
- **reversible** (*bool (optional)*) – If True restrict the ensemble of transition matrices to those having a detailed balance symmetry otherwise the likelihood optimization is carried out over the whole space of stochastic matrices.
- **mu** (*array_like*) – The stationary distribution of the MLE transition matrix.
- **method** (*str*) – Select which implementation to use for the estimation. One of 'auto', 'dense' and 'sparse', optional, default='auto'. 'dense' always selects the dense implementation, 'sparse' always selects the sparse one. 'auto' selects the most efficient implementation according to the sparsity structure of the matrix: if the occupation of the C matrix is less then one third, select sparse. Else select dense. The type of the T matrix returned always matches the type of the C matrix, irrespective of the method that was used to compute it.
- ****kwargs** (*Optional algorithm-specific parameters. See below for special cases*) –
- **Xinit** (*(M, M) ndarray*) – Optional parameter with reversible = True. initial value for the matrix of absolute transition probabilities. Unless set otherwise, will use X = diag(pi) t, where T is a nonreversible transition matrix estimated from C, i.e. T_ij = c_ij / sum_k c_ik, and pi is its stationary distribution.
- **maxiter** (*1000000 : int*) – Optional parameter with reversible = True. maximum number of iterations before the method exits
- **maxerr** (*1e-8 : float*) – Optional parameter with reversible = True. convergence tolerance for transition matrix estimation. This specifies the maximum change of the Eu-

clidean norm of relative stationary probabilities ($x_i = \sum_k x_{ik}$). The relative stationary probability changes $e_i = (x_i^{(1)} - x_i^{(2)})/(x_i^{(1)} + x_i^{(2)})$ are used in order to track changes in small probabilities. The Euclidean norm of the change vector, $|e_i|_2$, is compared to maxerr.

- **rev_pisym** (*bool, default=False*) – Fast computation of reversible transition matrix by normalizing $x_{ij} = \pi_i p_{ij} + \pi_j p_{ji}$. p_{ij} is the direct (nonreversible) estimate and pi_i is its stationary distribution. This estimator is asympotically unbiased but not maximum likelihood.
- **return_statdist** (*bool, default=False*) – Optional parameter with reversible = True. If set to true, the stationary distribution is also returned
- **return_conv** (*bool, default=False*) – Optional parameter with reversible = True. If set to true, the likelihood history and the pi_change history is returned.
- **warn_not_converged** (*bool, default=True*) – Prints a warning if not converged.
- **sparse_newton** (*bool, default=False*) – If True, use the experimental primal-dual interior-point solver for sparse input/computation method.

Returns

- **P** (*(M, M) ndarray or scipy.sparse matrix*) – The MLE transition matrix. P has the same data type (dense or sparse) as the input matrix C.
- *The reversible estimator returns by default only P, but may also return*
- *(P,pi) or (P,lhist,pi_changes) or (P,pi,lhist,pi_changes) depending on the return settings*
- **P** (*ndarray (n,n)*) – transition matrix. This is the only return for return_statdist = False, return_conv = False
- **(pi)** (*ndarray (n)*) – stationary distribution. Only returned if return_statdist = True
- **(lhist)** (*ndarray (k)*) – likelihood history. Has the length of the number of iterations needed. Only returned if return_conv = True
- **(pi_changes)** (*ndarray (k)*) – history of likelihood history. Has the length of the number of iterations needed. Only returned if return_conv = True

Notes

The transition matrix is a maximum likelihood estimate (MLE) of the probability distribution of transition matrices with parameters given by the count matrix.

References

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import transition_matrix
```

```
>>> C = np.array([[10, 1, 1], [2, 0, 3], [0, 1, 4]])
```

Non-reversible estimate

```
>>> T_nrev = transition_matrix(C)
>>> T_nrev
array([[ 0.83333333,  0.08333333,  0.08333333],
       [ 0.4       ,  0.        ,  0.6       ],
       [ 0.        ,  0.2       ,  0.8       ]])
```

Reversible estimate

```
>>> T_rev = transition_matrix(C, reversible=True)
>>> T_rev
array([[ 0.83333333,  0.10385551,  0.06281115],
       [ 0.35074677,  0.        ,  0.64925323],
       [ 0.04925323,  0.15074677,  0.8       ]])
```

Reversible estimate with given stationary vector

```
>>> mu = np.array([0.7, 0.01, 0.29])
>>> T_mu = transition_matrix(C, reversible=True, mu=mu)
>>> T_mu
array([[ 0.94771371,  0.00612645,  0.04615984],
       [ 0.42885157,  0.        ,  0.57114843],
       [ 0.11142031,  0.01969477,  0.86888491]])
```

msmtools.estimation.rate_matrix

msmtools.estimation.**rate_matrix**(*C*, *dt=1.0*, *method='KL'*, *sparsity=None*, *t_agg=None*, *pi=None*, *tol=10000000.0*, *K0=None*, *maxiter=100000*, *on_error='raise'*)

Estimate a reversible rate matrix from a count matrix.

Parameters

- **C** (*(N,N) ndarray*) – count matrix at a lag time dt
- **dt** (*float, optional, default=1.0*) – lag time that was used to estimate C
- **method** (*str, one of {'KL', 'CVE', 'pseudo', 'truncated_log'}*) – Method to use for estimation of the rate matrix.
  - 'pseudo' selects the pseudo-generator. A reversible transition matrix T is estimated and $(T - Id)/d$ is returned as the rate matrix.
  - 'truncated_log' selects the truncated logarithm[3]. A reversible transition matrix T is estimated and $max(logm(T * T)/(2dt), 0)$ is returned as the rate matrix. logm is the matrix logarithm and the maximum is taken element-wise.
  - 'CVE' selects the algorithm of Crommelin and Vanden-Eijnden[1]. It consists of minimizing the following objective function:

$$f(K) = \sum_{ij} \left(\sum_{kl} U_{ik}^{-1} K_{kl} U_{lj} - L_{ij} \right)^2 |\Lambda_i \Lambda_j|$$

    where Λ_i are the eigenvalues of T and U is the matrix of its (right) eigenvectors; $L_{ij} = \delta_{ij} \frac{1}{\tau} \log |\Lambda_i|$. T is computed from C using the reversible maximum likelihood estimator.

[3] E. B. Davies. Embeddable Markov Matrices. Electron. J. Probab. 15:1474, 2010.

[1] D. Crommelin and E. Vanden-Eijnden. Data-based inference of generators for markov jump processes using convex optimization. Multiscale. Model. Sim., 7(4):1751-1778, 2009.

– 'KL' selects the algorihtm of Kalbfleisch and Lawless[2]. It consists of maximizing the following log-likelihood:

$$f(K) = \log L = \sum_{ij} C_{ij} \log(e^{K\Delta t})_{ij}$$

where C_{ij} are the transition counts at a lag-time Δt. Here e is the matrix exponential and the logarithm is taken element-wise.

- **sparsity** (*(N,N) ndarray or None, optional, default=None*) – If sparsity is None, a fully occupied rate matrix will be estimated. Alternatively, with the methods 'CVE' and 'KL' a ndarray of the same shape as C can be supplied. If sparsity[i,j]=0 and sparsity[j,i]=0 the rate matrix elements K_{ij} and K_{ji} will be constrained to zero.
- **t_agg** (*float, optional*) – the aggregated simulation time; by default this is the total number of transition counts times the lag time (no sliding window counting). This value is used to compute the lower bound on the transition rate (that are not zero). If sparsity is None, this value is ignored.
- **pi** – the stationary vector of the desired rate matrix K. If no pi is given, the function takes the stationary vector of the MLE reversible T matrix that is computed from C.
- **tol** (*float, optional, default = 1.0E7*) – Tolerance of the quasi-Newton algorithm that is used to minimize the objective function. This is passed as the *factr* parameter to `scipy.optimize.fmin_l_bfgs_b`. Typical values for factr are: 1e12 for low accuracy; 1e7 for moderate accuracy; 10.0 for extremely high accuracy.
- **maxiter** (*int, optional, default = 100000*) – Minimization of the objective function will do at most this number of steps.
- **on_error** (*string, optional, default = 'raise'*) – What to do then an error happend. When 'raise' is given, raise an exception. When 'warn' is given, produce a (Python) warning.

Returns **K** – the optimal rate matrix

Return type (N,N) ndarray

Notes

In this implementation the algorithm of Crommelin and Vanden-Eijnden (CVE) is initialized with the pseudo-generator estimate. The algorithm of Kalbfleisch and Lawless (KL) is initialized using the CVE result.

Example

```
>>> import numpy as np
>>> from msmtools.estimation import rate_matrix
>>> C = np.array([[100,1],[50,50]])
>>> rate_matrix(C)
array([[-0.01384753,  0.01384753],
       [ 0.69930032, -0.69930032]])
```

[2] J. D. Kalbfleisch and J. F. Lawless. The analysis of panel data under a markov assumption. J. Am. Stat. Assoc., 80(392):863-871, 1985.

References

msmtools.estimation.log_likelihood

msmtools.estimation.**log_likelihood**(*C*, *T*)

Log-likelihood of the count matrix given a transition matrix.

Parameters

- **C** (*(M, M) ndarray or scipy.sparse matrix*) – Count matrix
- **T** (*(M, M) ndarray orscipy.sparse matrix*) – Transition matrix

Returns **logL** – Log-likelihood of the count matrix

Return type float

Notes

The likelihood of a set of observed transition counts $C = (c_{ij})$ for a given matrix of transition counts $T = (t_{ij})$ is given by

$$L(C|P) = \prod_{i=1}^{M} \left(\prod_{j=1}^{M} p_{ij}^{c_{ij}} \right)$$

The log-likelihood is given by

$$l(C|P) = \sum_{i,j=1}^{M} c_{ij} \log p_{ij}.$$

The likelihood describes the probability of making an observation C for a given model P.

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import log_likelihood
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
```

```
>>> C = np.array([[58, 7, 0], [6, 0, 4], [0, 3, 21]])
>>> logL = log_likelihood(C, T)
>>> logL
-38.2808034725...
```

```
>>> C = np.array([[58, 20, 0], [6, 0, 4], [0, 3, 21]])
>>> logL = log_likelihood(C, T)
>>> logL
-68.2144096814...
```

References

msmtools.estimation.tmatrix_cov

msmtools.estimation.**tmatrix_cov**(*C*, *k=None*)

Covariance tensor for non-reversible transition matrix posterior.

Parameters

- **C** (*(M, M) ndarray or scipy.sparse matrix*) – Count matrix
- **k** (*int (optional)*) – Return only covariance matrix for entires in the k-th row of the transition matrix

Returns **cov** – Covariance tensor for transition matrix posterior

Return type (M, M, M) ndarray

Notes

The posterior of non-reversible transition matrices is

$$\mathbb{P}(T|C) \propto \prod_{i=1}^{M} \left(\prod_{j=1}^{M} p_{ij}^{c_{ij}} \right)$$

Each row in the transition matrix is distributed according to a Dirichlet distribution with parameters given by the observed transition counts c_{ij}.

The covariance tensor $\mathrm{cov}[p_{ij}, p_{kl}] = \Sigma_{i,j,k,l}$ is zero whenever $i \neq k$ so that only $\Sigma_{i,j,i,l}$ is returned.

msmtools.estimation.error_perturbation

msmtools.estimation.**error_perturbation**(*C*, *S*)

Error perturbation for given sensitivity matrix.

Parameters

- **C** (*(M, M) ndarray*) – Count matrix
- **S** (*(M, M) ndarray or (K, M, M) ndarray*) – Sensitivity matrix (for scalar observable) or sensitivity tensor for vector observable

Returns **X** – error-perturbation (for scalar observables) or covariance matrix (for vector-valued observable)

Return type float or (K, K) ndarray

Notes

Scalar observable

The sensitivity matrix $S = (s_{ij})$ of a scalar observable $f(T)$ is defined as

$$S = \left(\left. \frac{\partial f(T)}{\partial t_{ij}} \right|_{T_0} \right)$$

evaluated at a suitable transition matrix T_0.

The sensitivity is the variance of the observable

$$\mathbb{V}(f) = \sum_{i,j,k,l} s_{ij} \mathrm{cov}[t_{ij}, t_{kl}] s_{kl}$$

Vector valued observable

The sensitivity tensor $S = (s_{ijk})$ for a vector valued observable $(f_1(T), \ldots, f_K(T))$ is defined as

$$S = \left(\frac{\partial f_i(T)}{\partial t_{jk}} \Big|_{T_0} \right)$$

evaluated at a suitable transition matrix T_0.

The sensitivity is the covariance matrix for the observable

$$\mathrm{cov}[f_\alpha(T), f_\beta(T)] = \sum_{i,j,k,l} s_{\alpha ij} \mathrm{cov}[t_{ij}, t_{kl}] s_{\beta kl}$$

Sampling

tmatrix_sampler(C[, reversible, mu, T0, ...])	Generate transition matrix sampler object.

msmtools.estimation.tmatrix_sampler

msmtools.estimation.**tmatrix_sampler**(*C*, *reversible=False*, *mu=None*, *T0=None*, *nsteps=None*, *prior='sparse'*)

Generate transition matrix sampler object.

Parameters

- **C** (*(M, M) ndarray or scipy.sparse matrix*) – Count matrix
- **reversible** (*bool*) – If true sample from the ensemble of transition matrices restricted to those obeying a detailed balance condition, else draw from the whole ensemble of stochastic matrices.
- **mu** (*array_like*) – A fixed stationary distribution. Transition matrices with that stationary distribution will be sampled
- **T0** (*ndarray, shape=(n, n) or scipy.sparse matrix*) – Starting point of the MC chain of the sampling algorithm. Has to obey the required constraints.
- **nstep** (*int, default=None*) – number of full Gibbs sampling sweeps per sample. This option is meant to ensure approximately uncorrelated samples for every call to sample(). If None, the number of steps will be automatically determined based on the other options and the matrix size. nstep>1 will only be used for reversible sampling, because nonreversible sampling generates statistically independent transition matrices every step.

Returns **sampler**

Return type A :py:class:dense.tmatrix_sampler.TransitionMatrixSampler object that can be used to generate samples.

Notes

The transition matrix sampler generates transition matrices from the posterior distribution. The posterior distribution is given as a product of Dirichlet distributions

$$\mathbb{P}(T|C) \propto \prod_{i=1}^{M} \left(\prod_{j=1}^{M} p_{ij}^{c_{ij}} \right)$$

The method can generate samples from the posterior under the following constraints

Reversible sampling

Using a MCMC sampler outlined in .. [1] it is ensured that samples from the posterior are reversible, i.e. there is a probability vector (μ_i) such that $\mu_i t_{ij} = \mu_j t_{ji}$ holds for all i, j.

Reversible sampling with fixed stationary vector

Using a MCMC sampler outlined in .. [2] it is ensured that samples from the posterior fulfill detailed balance with respect to a given probability vector (μ_i).

References

Bootstrap

bootstrap_counts(dtrajs, lagtime[, corrlength])	Generates a randomly resampled count matrix given the input coordinates.
bootstrap_trajectories(trajs, correlation_length)	Generates a randomly resampled trajectory segments.

msmtools.estimation.bootstrap_counts

msmtools.estimation.**bootstrap_counts**(*dtrajs*, *lagtime*, *corrlength=None*)

Generates a randomly resampled count matrix given the input coordinates.

Parameters

- **dtrajs** (*array-like or array-like of array-like*) – single or multiple discrete trajectories. Every trajectory is assumed to be a statistically independent realization. Note that this is often not true and is a weakness with the present bootstrapping approach.
- **lagtime** (*int*) – the lag time at which the count matrix will be evaluated
- **corrlength** (*int, optional, default=None*) – the correlation length of the discrete trajectory. N / corrlength counts will be generated, where N is the total number of frames. If set to None (default), corrlength = lagtime will be used.

Notes

This function can be called multiple times in order to generate randomly resampled realizations of count matrices. For each of these realizations you can estimate a transition matrix, and from each of them computing the observables of your interest. The standard deviation of such a sample of the observable is a model for the standard error.

The bootstrap will be generated by sampling N/corrlength counts at time tuples (t, t+lagtime), where t is uniformly sampled over all trajectory time frames in [0,n_i-lagtime]. Here, n_i is the length of trajectory i and N = sum_i n_i is the total number of frames.

See also:

bootstrap_trajectories()

msmtools.estimation.bootstrap_trajectories

msmtools.estimation.**bootstrap_trajectories**(*trajs*, *correlation_length*)

Generates a randomly resampled trajectory segments.

Parameters

- **trajs** (*array-like or array-like of array-like*) – single or multiple trajectories. Every trajectory is assumed to be a statistically independent realization. Note that this is often not true and is a weakness with the present bootstrapping approach.
- **correlation_length** (*int*) – Correlation length (also known as the or statistical inefficiency) of the data. If set to < 1 or > L, where L is the longest trajectory length, the bootstrapping will sample full trajectories. We suggest to select the largest implied timescale or relaxation timescale as a conservative estimate of the correlation length. If this timescale is unknown, it's suggested to use full trajectories (set timescale to < 1) or come up with a rough estimate. For computing the error on specific observables, one may use shorter timescales, because the relevant correlation length is the integral of the autocorrelation function of the observables of interest [3]. The slowest implied timescale is an upper bound for that correlation length, and therefore a conservative estimate [4].

Notes

This function can be called multiple times in order to generate randomly resampled trajectory data. In order to compute error bars on your observable of interest, call this function to generate resampled trajectories, and put them into your estimator. The standard deviation of such a sample of the observable is a model for the standard error.

Implements a moving block bootstrapping procedure [1] for generation of randomly resampled count matrixes from discrete trajectories. The corrlation length determines the size of trajectory blocks that will remain contiguous. For a single trajectory N with correlation length t_corr < N, we will sample floor(N/t_corr) subtrajectories of length t_corr using starting time t. t is a uniform random number in [0, N-t_corr-1]. When multiple trajectories are available, N is the total number of timesteps over all trajectories, the algorithm will generate resampled data with a total number of N (or slightly larger) time steps. Each trajectory of length n_i has a probability of n_i to be selected. Trajectories of length n_i <= t_corr are returned completely. For longer trajectories, segments of length t_corr are randomly generated.

Note that like all error models for correlated time series data, Bootstrapping just gives you a model for the error given a number of assumptions [2]. The most critical decisions are: (1) is this approach meaningful at all (only if the trajectories are statistically independent realizations), and (2) select an appropriate timescale of the correlation length (see below). Note that transition matrix sampling from the Dirichlet distribution is a much better option from a theoretical point of view, but may also be computationally more demanding.

References

Priors

prior_neighbor(C[, alpha])	Neighbor prior for the given count matrix.
prior_const(C[, alpha])	Constant prior for given count matrix.
prior_rev(C[, alpha])	Prior counts for sampling of reversible transition matrices.

msmtools.estimation.prior_neighbor

msmtools.estimation.**prior_neighbor**(*C*, *alpha=0.001*)

Neighbor prior for the given count matrix.

Parameters

- **C** (*(M, M) ndarray or scipy.sparse matrix*) – Count matrix
- **alpha** (*float (optional)*) – Value of prior counts

Returns **B** – Prior count matrix

Return type (M, M) ndarray or scipy.sparse matrix

Notes

The neighbor prior b_{ij} is defined as

$$b_{ij} = \begin{cases} \alpha & c_{ij} + c_{ji} > 0 \\ 0 & \text{else} \end{cases}$$

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import prior_neighbor
```

```
>>> C = np.array([[10, 1, 0], [2, 0, 3], [0, 1, 4]])
>>> B = prior_neighbor(C)
>>> B
array([[ 0.001,  0.001,  0.   ],
       [ 0.001,  0.   ,  0.001],
       [ 0.   ,  0.001,  0.001]])
```

msmtools.estimation.prior_const

msmtools.estimation.**prior_const**(*C*, *alpha=0.001*)

Constant prior for given count matrix.

Parameters

- **C** (*(M, M) ndarray or scipy.sparse matrix*) – Count matrix
- **alpha** (*float (optional)*) – Value of prior counts

Returns **B** – Prior count matrix

Return type (M, M) ndarray

Notes

The prior is defined as

$$b_{ij} = \alpha \quad \forall i, j$$

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import prior_const
```

```
>>> C = np.array([[10, 1, 0], [2, 0, 3], [0, 1, 4]])
>>> B = prior_const(C)
>>> B
array([[ 0.001,  0.001,  0.001],
       [ 0.001,  0.001,  0.001],
       [ 0.001,  0.001,  0.001]])
```

msmtools.estimation.prior_rev

msmtools.estimation.**prior_rev**(*C*, *alpha=-1.0*)

Prior counts for sampling of reversible transition matrices.

Prior is defined as

b_ij= alpha if i<=j b_ij=0 else

Parameters

- **C** (*(M, M) ndarray or scipy.sparse matrix*) – Count matrix
- **alpha** (*float (optional)*) – Value of prior counts

Returns **B** – Matrix of prior counts

Return type (M, M) ndarray

Notes

The reversible prior is a matrix with -1 on the upper triangle. Adding this prior respects the fact that for a reversible transition matrix the degrees of freedom correspond essentially to the upper triangular part of the matrix.

The prior is defined as

$$b_{ij} = \begin{cases} \alpha & i \leq j \\ 0 & \text{elsewhere} \end{cases}$$

Examples

```
>>> import numpy as np
>>> from msmtools.estimation import prior_rev
```

```
>>> C = np.array([[10, 1, 0], [2, 0, 3], [0, 1, 4]])
>>> B = prior_rev(C)
>>> B
array([[-1., -1., -1.],
       [ 0., -1., -1.],
       [ 0.,  0., -1.]])
```

1.2.2 analysis - MSM analysis functions (msmtools.analysis)

This module contains functions to analyze a created Markov model, which is specified with a transition matrix T.

Validation

is_transition_matrix(T[, tol])	Check if the given matrix is a transition matrix.
is_tmatrix(T[, tol])	Check if the given matrix is a transition matrix.
is_rate_matrix(K[, tol])	Check if the given matrix is a rate matrix.
is_connected(T[, directed])	Check connectivity of the given matrix.
is_reversible(T[, mu, tol])	Check reversibility of the given transition matrix.

msmtools.analysis.is_transition_matrix

msmtools.analysis.**is_transition_matrix**(*T*, *tol=1e-12*)

Check if the given matrix is a transition matrix.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Matrix to check
- **tol** (*float (optional)*) – Floating point tolerance to check with

Returns **is_transition_matrix** – True, if T is a valid transition matrix, False otherwise

Return type bool

Notes

A valid transition matrix $P = (p_{ij})$ has non-negative elements, $p_{ij} \geq 0$, and elements of each row sum up to one, $\sum_j p_{ij} = 1$. Matrices wit this property are also called stochastic matrices.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import is_transition_matrix
```

```
>>> A = np.array([[0.4, 0.5, 0.3], [0.2, 0.4, 0.4], [-1, 1, 1]])
>>> is_transition_matrix(A)
False
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> is_transition_matrix(T)
True
```

msmtools.analysis.is_tmatrix

msmtools.analysis.**is_tmatrix**(*T*, *tol=1e-12*)

Check if the given matrix is a transition matrix.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Matrix to check
- **tol** (*float (optional)*) – Floating point tolerance to check with

Returns **is_transition_matrix** – True, if T is a valid transition matrix, False otherwise

Return type bool

Notes

A valid transition matrix $P = (p_{ij})$ has non-negative elements, $p_{ij} \geq 0$, and elements of each row sum up to one, $\sum_j p_{ij} = 1$. Matrices wit this property are also called stochastic matrices.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import is_transition_matrix
```

```
>>> A = np.array([[0.4, 0.5, 0.3], [0.2, 0.4, 0.4], [-1, 1, 1]])
>>> is_transition_matrix(A)
False
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> is_transition_matrix(T)
True
```

msmtools.analysis.is_rate_matrix

msmtools.analysis.**is_rate_matrix**(*K*, *tol=1e-12*)

Check if the given matrix is a rate matrix.

Parameters

- **K** (*(M, M) ndarray or scipy.sparse matrix*) – Matrix to check
- **tol** (*float (optional)*) – Floating point tolerance to check with

Returns **is_rate_matrix** – True, if K is a valid rate matrix, False otherwise

Return type bool

Notes

A valid rate matrix $K = (k_{ij})$ has non-negative off diagonal elements, $k_{ij} \leq 0$, for $i \neq j$, and elements of each row sum up to zero, $\sum_j k_{ij} = 0$.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import is_rate_matrix
```

```
>>> A = np.array([[0.5, -0.5, -0.2], [-0.3, 0.6, -0.3], [-0.2, 0.2, 0.0]])
>>> is_rate_matrix(A)
False
```

```
>>> K = np.array([[-0.3, 0.2, 0.1], [0.5, -0.5, 0.0], [0.1, 0.1, -0.2]])
>>> is_rate_matrix(K)
True
```

msmtools.analysis.is_connected

msmtools.analysis.**is_connected**(*T*, *directed=True*)

Check connectivity of the given matrix.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Matrix to check
- **directed** (*bool (optional)*) – If True respect direction of transitions, if False do not distinguish between forward and backward transitions

Returns **is_connected** – True, if T is connected, False otherwise

Return type bool

Notes

A transition matrix $T = (t_{ij})$ is connected if for any pair of states (i, j) one can reach state j from state i in a finite number of steps.

In more precise terms: For any pair of states (i, j) there exists a number $N = N(i, j)$, so that the probability of going from state i to state j in N steps is positive, $\mathbb{P}(X_N = j | X_0 = i) > 0$.

A transition matrix with this property is also called irreducible.

Viewing the transition matrix as the adjency matrix of a (directed) graph the transition matrix is irreducible if and only if the corresponding graph has a single connected component. Connectivity of a graph can be efficiently checked using Tarjan's algorithm.

References

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import is_connected
```

```
>>> A = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.0, 1.0]])
>>> is_connected(A)
False
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> is_connected(T)
True
```

msmtools.analysis.is_reversible

msmtools.analysis.**is_reversible**(*T*, *mu=None*, *tol=1e-12*)

Check reversibility of the given transition matrix.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix
- **mu** (*(M,) ndarray (optional)*) – Test reversibility with respect to this vector
- **tol** (*float (optional)*) – Floating point tolerance to check with

Returns **is_reversible** – True, if T is reversible, False otherwise

Return type bool

Notes

A transition matrix $T = (t_{ij})$ is reversible with respect to a probability vector $\mu = (\mu_i)$ if the follwing holds,

$$\mu_i \, t_{ij} = \mu_j \, t_{ji}.$$

In this case μ is the stationary vector for T, so that $\mu^T T = \mu^T$.

If the stationary vector is unknown it is computed from T before reversibility is checked.

A reversible transition matrix has purely real eigenvalues. The left eigenvectors (l_i) can be computed from right eigenvectors (r_i) via $l_i = \mu_i r_i$.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import is_reversible
```

```
>>> P = np.array([[0.8, 0.1, 0.1], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> is_reversible(P)
False
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> is_reversible(T)
True
```

Decomposition

Decomposition routines use the scipy LAPACK bindings for dense numpy-arrays and the ARPACK bindings for scipy sparse matrices.

stationary_distribution(T)	Compute stationary distribution of stochastic matrix T.
statdist(T)	Compute stationary distribution of stochastic matrix T.
eigenvalues(T[, k, ncv, reversible, mu])	Find eigenvalues of the transition matrix.
eigenvectors(T[, k, right, ncv, reversible, mu])	Compute eigenvectors of given transition matrix.
rdl_decomposition(T[, k, norm, ncv, ...])	Compute the decomposition into eigenvalues, left and right eigenvectors.
timescales(T[, tau, k, ncv, reversible, mu])	Compute implied time scales of given transition matrix.

msmtools.analysis.stationary_distribution

msmtools.analysis.**stationary_distribution**(*T*)

Compute stationary distribution of stochastic matrix T.

Parameters **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix

Returns **mu** – Vector of stationary probabilities.

Return type (M,) ndarray

Notes

The stationary distribution μ is the left eigenvector corresponding to the non-degenerate eigenvalue $\lambda = 1$,

$$\mu^T T = \mu^T.$$

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import stationary_distribution
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.4, 0.2, 0.4], [0.0, 0.1, 0.9]])
>>> mu = stationary_distribution(T)
>>> mu
array([ 0.44444444, 0.11111111, 0.44444444])
```

msmtools.analysis.statdist

msmtools.analysis.**statdist**(*T*)

Compute stationary distribution of stochastic matrix T.

Parameters **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix

Returns **mu** – Vector of stationary probabilities.

Return type (M,) ndarray

Notes

The stationary distribution μ is the left eigenvector corresponding to the non-degenerate eigenvalue $\lambda = 1$,

$$\mu^T T = \mu^T.$$

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import stationary_distribution
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.4, 0.2, 0.4], [0.0, 0.1, 0.9]])
>>> mu = stationary_distribution(T)
>>> mu
array([ 0.44444444, 0.11111111, 0.44444444])
```

msmtools.analysis.eigenvalues

msmtools.analysis.**eigenvalues**(*T*, *k=None*, *ncv=None*, *reversible=False*, *mu=None*)

Find eigenvalues of the transition matrix.

Parameters

- **T** (*(M, M) ndarray or sparse matrix*) – Transition matrix
- **k** (*int (optional)*) – Compute the first *k* eigenvalues of *T*
- **ncv** (*int (optional)*) – The number of Lanczos vectors generated, *ncv* must be greater than k; it is recommended that ncv > 2*k
- **reversible** (*bool, optional*) – Indicate that transition matrix is reversible
- **mu** (*(M,) ndarray, optional*) – Stationary distribution of T

Returns **w** – Eigenvalues of *T*. If *k* is specified, *w* has shape (k,)

Return type (M,) ndarray

Notes

Eigenvalues are returned in order of decreasing magnitude.

If reversible=True the the eigenvalues of the similar symmetric matrix *sqrt(mu_i / mu_j) p_{ij}* will be computed.

The precomputed stationary distribution will only be used if reversible=True.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import eigenvalues
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> w = eigenvalues(T)
>>> w
array([ 1.0+0.j, 0.9+0.j, -0.1+0.j])
```

msmtools.analysis.eigenvectors

msmtools.analysis.**eigenvectors**(*T*, *k=None*, *right=True*, *ncv=None*, *reversible=False*, *mu=None*)

Compute eigenvectors of given transition matrix.

Parameters

- **T** (*numpy.ndarray, shape(d,d) or scipy.sparse matrix*) – Transition matrix (stochastic matrix)
- **k** (*int (optional)*) – Compute the first k eigenvectors
- **ncv** (*int (optional)*) – The number of Lanczos vectors generated, *ncv* must be greater than *k*; it is recommended that *ncv* > 2*k
- **right** (*bool, optional*) – If right=True compute right eigenvectors, left eigenvectors otherwise
- **reversible** (*bool, optional*) – Indicate that transition matrix is reversible
- **mu** (*(M,) ndarray, optional*) – Stationary distribution of T

Returns **eigvec** – The eigenvectors of T ordered with decreasing absolute value of the corresponding eigenvalue. If k is None then n=d, if k is int then n=k.

Return type numpy.ndarray, shape=(d, n)

See also:

rdl_decomposition()

Notes

Eigenvectors are computed using the scipy interface to the corresponding LAPACK/ARPACK routines.

If reversible=False, the returned eigenvectors v_i are normalized such that

$$\langle v_i, v_i \rangle = 1$$

This is the case for right eigenvectors r_i as well as for left eigenvectors l_i.

If you desire orthonormal left and right eigenvectors please use the rdl_decomposition method.

If reversible=True the the eigenvectors of the similar symmetric matrix *sqrt(mu_i / mu_j) p_{ij}* will be used to compute the eigenvectors of T.

The precomputed stationary distribution will only be used if reversible=True.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import eigenvectors
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> R = eigenvectors(T)
```

Matrix with right eigenvectors as columns

```
>>> R
array([[  5.77350269e-01,   7.07106781e-01,   9.90147543e-02], ...
```

msmtools.analysis.rdl_decomposition

msmtools.analysis.**rdl_decomposition**(*T*, *k=None*, *norm='auto'*, *ncv=None*, *reversible=False*, *mu=None*)

Compute the decomposition into eigenvalues, left and right eigenvectors.

Parameters

- **T** (*(M, M) ndarray or sparse matrix*) – Transition matrix
- **k** (*int (optional)*) – Number of eigenvector/eigenvalue pairs
- **norm** (*{'standard', 'reversible', 'auto'}, optional*) – which normalization convention to use

norm	
'standard'	LR = Id, is a probabilitydistribution, the stationary distributionof *T*. Right eigenvectors *R*have a 2-norm of 1
'reversible'	*R* and *L* are related via `L[0, :]*R`
'auto'	reversible if T is reversible, else standard.

- **ncv** (*int (optional)*) – The number of Lanczos vectors generated, *ncv* must be greater than k; it is recommended that ncv > 2*k
- **reversible** (*bool, optional*) – Indicate that transition matrix is reversible
- **mu** (*(M,) ndarray, optional*) – Stationary distribution of T

Returns

- **R** (*(M, M) ndarray*) – The normalized ("unit length") right eigenvectors, such that the column `R[:,i]` is the right eigenvector corresponding to the eigenvalue `w[i]`, `dot(T, R[:,i])``=``w[i]*R[:,i]`
- **D** (*(M, M) ndarray*) – A diagonal matrix containing the eigenvalues, each repeated according to its multiplicity
- **L** (*(M, M) ndarray*) – The normalized (with respect to *R*) left eigenvectors, such that the row `L[i, :]` is the left eigenvector corresponding to the eigenvalue `w[i]`, `dot(L[i, :], T)``=``w[i]*L[i, :]`

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import rdl_decomposition
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> R, D, L = rdl_decomposition(T)
```

Matrix with right eigenvectors as columns

```
>>> R
array([[  1.00000000e+00,   1.04880885e+00,   3.16227766e-01], ...
```

Diagonal matrix with eigenvalues

```
>>> D
array([[ 1.0+0.j,  0.0+0.j,  0.0+0.j],
       [ 0.0+0.j,  0.9+0.j,  0.0+0.j],
       [ 0.0+0.j,  0.0+0.j, -0.1+0.j]])
```

Matrix with left eigenvectors as rows

```
>>> L # +doctest: +ELLIPSIS
array([[  4.54545455e-01,   9.09090909e-02,   4.54545455e-01], ...
```

msmtools.analysis.timescales

msmtools.analysis.**timescales**(*T*, *tau=1*, *k=None*, *ncv=None*, *reversible=False*, *mu=None*)

Compute implied time scales of given transition matrix.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix
- **tau** (*int (optional)*) – The time-lag (in elementary time steps of the microstate trajectory) at which the given transition matrix was constructed.
- **k** (*int (optional)*) – Compute the first *k* implied time scales.
- **ncv** (*int (optional, for sparse T only)*) – The number of Lanczos vectors generated, *ncv* must be greater than k; it is recommended that ncv > 2*k
- **reversible** (*bool, optional*) – Indicate that transition matrix is reversible
- **mu** (*(M,) ndarray, optional*) – Stationary distribution of T

Returns **ts** – The implied time scales of the transition matrix. If *k* is not None then the shape of *ts* is (k,).

Return type (M,) ndarray

Notes

The implied time scale t_i is defined as

$$t_i = -\frac{\tau}{\log|\lambda_i|}$$

If reversible=True the the eigenvalues of the similar symmetric matrix *sqrt(mu_i / mu_j) p_{ij}* will be computed.

The precomputed stationary distribution will only be used if reversible=True.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import timescales
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> ts = timescales(T)
>>> ts
array([        inf,  9.49122158,  0.43429448])
```

Expected counts

expected_counts(T, p0, N)	Compute expected transition counts for Markov chain with n steps.
expected_counts_stationary(T, N[, mu])	Expected transition counts for Markov chain in equilibrium.

msmtools.analysis.expected_counts

msmtools.analysis.**expected_counts**(*T*, *p0*, *N*)

Compute expected transition counts for Markov chain with n steps.

Parameters

- **T** (*(M, M) ndarray or sparse matrix*) – Transition matrix
- **p0** (*(M,) ndarray*) – Initial (probability) vector
- **N** (*int*) – Number of steps to take

Returns **EC** – Expected value for transition counts after N steps

Return type (M, M) ndarray or sparse matrix

Notes

Expected counts can be computed via the following expression

$$\mathbb{E}[C^{(N)}] = \sum_{k=0}^{N-1} \mathrm{diag}(p^T T^k) T$$

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import expected_counts
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> p0 = np.array([1.0, 0.0, 0.0])
>>> N = 100
>>> EC = expected_counts(T, p0, N)
```

```
>>> EC
array([[ 45.44616147,   5.0495735 ,   0.        ],
       [  4.50413223,   0.        ,   4.50413223],
       [  0.        ,   4.04960006,  36.44640052]])
```

msmtools.analysis.expected_counts_stationary

msmtools.analysis.**expected_counts_stationary**(*T*, *N*, *mu=None*)

Expected transition counts for Markov chain in equilibrium.

Parameters

- **T** (*(M, M) ndarray or sparse matrix*) – Transition matrix.
- **N** (*int*) – Number of steps for chain.
- **mu** (*(M,) ndarray (optional)*) – Stationary distribution for T. If mu is not specified it will be computed from T.

Returns **EC** – Expected value for transition counts after N steps.

Return type (M, M) ndarray or sparse matrix

Notes

Since μ is stationary for T we have

$$\mathbb{E}[C^{(N)}] = N D_\mu T.$$

D_μ is a diagonal matrix. Elements on the diagonal are given by the stationary vector μ

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import expected_counts_stationary
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> N = 100
>>> EC = expected_counts_stationary(T, N)
```

```
>>> EC
array([[ 40.90909091,   4.54545455,   0.        ],
       [  4.54545455,   0.        ,   4.54545455],
       [  0.        ,   4.54545455,  40.90909091]])
```

Passage times

mfpt(T, target[, origin, tau, mu])	Mean first passage times (from a set of starting states - optional) to a set of target states.

msmtools.analysis.mfpt

msmtools.analysis.**mfpt**(*T*, *target*, *origin=None*, *tau=1*, *mu=None*)

Mean first passage times (from a set of starting states - optional) to a set of target states.

Parameters

- **T** (*ndarray or scipy.sparse matrix, shape=(n,n)*) – Transition matrix.
- **target** (*int or list of int*) – Target states for mfpt calculation.
- **origin** (*int or list of int (optional)*) – Set of starting states.
- **tau** (*int (optional)*) – The time-lag (in elementary time steps of the microstate trajectory) at which the given transition matrix was constructed.
- **mu** (*(n,) ndarray (optional)*) – The stationary distribution of the transition matrix T.

Returns **m_t** – Mean first passage time or vector of mean first passage times.

Return type ndarray, shape=(n,) or shape(1,)

Notes

The mean first passage time $\mathbf{E}_x[T_Y]$ is the expected hitting time of one state y in Y when starting in state x.

For a fixed target state y it is given by

$$\mathbb{E}_x[T_y] = \begin{cases} 0 & x = y \\ 1 + \sum_z T_{x,z}\mathbb{E}_z[T_y] & x \neq y \end{cases}$$

For a set of target states Y it is given by

$$\mathbb{E}_x[T_Y] = \begin{cases} 0 & x \in Y \\ 1 + \sum_z T_{x,z}\mathbb{E}_z[T_Y] & x \notin Y \end{cases}$$

The mean first passage time between sets, $\mathbf{E}_X[T_Y]$, is given by

$$\mathbb{E}_X[T_Y] = \sum_{x \in X} \frac{\mu_x \mathbb{E}_x[T_Y]}{\sum_{z \in X} \mu_z}$$

References

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import mfpt
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> m_t = mfpt(T, 0)
>>> m_t
array([  0.,  12.,  22.])
```

Committors and PCCA

committor(T, A, B[, forward, mu])	Compute the committor between sets of microstates.
pcca(T, m)	Compute meta-stable sets using PCCA++ _[1] and return the membership of all states to these sets.

msmtools.analysis.committor

msmtools.analysis.**committor**(*T*, *A*, *B*, *forward=True*, *mu=None*)

Compute the committor between sets of microstates.

The committor assigns to each microstate a probability that being at this state, the set B will be hit next, rather than set A (forward committor), or that the set A has been hit previously rather than set B (backward committor). See [1] for a detailed mathematical description. The present implementation uses the equations given in [2].

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix
- **A** (*array_like*) – List of integer state labels for set A
- **B** (*array_like*) – List of integer state labels for set B
- **forward** (*bool*) – If True compute the forward committor, else compute the backward committor.

Returns **q** – Vector of comittor probabilities.

Return type (M,) ndarray

Notes

Committor functions are used to characterize microstates in terms of their probability to being visited during a reaction/transition between two disjoint regions of state space A, B.

Forward committor

The forward committor $q_i^{(+)}$ is defined as the probability that the process starting in *i* will reach *B* first, rather than *A*.

Using the first hitting time of a set S,

$$T_S = \inf\{t \geq 0 | X_t \in S\}$$

the forward committor $q_i^{(+)}$ can be fromally defined as

$$q_i^{(+)} = \mathbb{P}_i(T_A < T_B).$$

The forward committor solves to the following boundary value problem

$$\begin{array}{rl} \sum_j L_{ij} q_j^{(+)} = 0 & i \in X \setminus (A \cup B) \\ q_i^{(+)} = 0 & i \in A \\ q_i^{(+)} = 1 & i \in B \end{array}$$

$L = T - I$ denotes the generator matrix.

Backward committor

The backward committor is defined as the probability that the process starting in x came from A rather than from B.

Using the last exit time of a set S,

$$t_S = \sup\{t \geq 0 | X_t \notin S\}$$

the backward committor can be formally defined as

$$q_i^{(-)} = \mathbb{P}_i(t_A < t_B).$$

The backward comittor solves another boundary value problem

$$\begin{array}{rl} \sum_j K_{ij} q_j^{(-)} = 0 & i \in X \setminus (A \cup B) \\ q_i^{(-)} = 1 & i \in A \\ q_i^{(-)} = 0 & i \in B \end{array}$$

$K = (D_\pi L)^T$ denotes the adjoint generator matrix.

References

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import committor
>>> T = np.array([[0.89, 0.1, 0.01], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> A = [0]
>>> B = [2]
```

```
>>> u_plus = committor(T, A, B)
>>> u_plus
array([ 0. ,  0.5,  1. ])
```

```
>>> u_minus = committor(T, A, B, forward=False)
>>> u_minus
array([ 1.        ,  0.45454545,  0.        ])
```

msmtools.analysis.pcca

msmtools.analysis.**pcca**(*T*, *m*)

Compute meta-stable sets using PCCA++ _[1] and return the membership of all states to these sets.

Parameters

- **T** (*(n, n) ndarray or scipy.sparse matrix*) – Transition matrix
- **m** (*int*) – Number of metastable sets

Returns clusters – Membership vectors. clusters[i, j] contains the membership of state i to metastable state j

Return type (n, m) ndarray

Notes

Perron cluster center analysis assigns each microstate a vector of membership probabilities. This assignement is performed using the right eigenvectors of the transition matrix. Membership probabilities are computed via numerical optimization of the entries of a membership matrix.

References

Fingerprints

fingerprint_correlation(T, obs1[, obs2, ...])	Dynamical fingerprint for equilibrium correlation experiment.
fingerprint_relaxation(T, p0, obs[, tau, k, ncv])	Dynamical fingerprint for relaxation experiment.
expectation(T, a[, mu])	Equilibrium expectation value of a given observable.
correlation(T, obs1[, obs2, times, maxtime, ...])	Time-correlation for equilibrium experiment.
relaxation(T, p0, obs[, times, k, ncv])	Relaxation experiment.

msmtools.analysis.fingerprint_correlation

msmtools.analysis.**fingerprint_correlation**(*T*, *obs1*, *obs2=None*, *tau=1*, *k=None*, *ncv=None*)

Dynamical fingerprint for equilibrium correlation experiment.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix
- **obs1** (*(M,) ndarray*) – Observable, represented as vector on state space
- **obs2** (*(M,) ndarray (optional)*) – Second observable, for cross-correlations
- **k** (*int (optional)*) – Number of time-scales and amplitudes to compute
- **tau** (*int (optional)*) – Lag time of given transition matrix, for correct time-scales
- **ncv** (*int (optional)*) – The number of Lanczos vectors generated, *ncv* must be greater than k; it is recommended that ncv > 2*k

Returns

- **timescales** (*(N,) ndarray*) – Time-scales of the transition matrix
- **amplitudes** (*(N,) ndarray*) – Amplitudes for the correlation experiment

See also:

correlation(), *fingerprint_relaxation()*

References

Notes

Fingerprints are a combination of time-scale and amplitude spectrum for a equilibrium correlation or a non-equilibrium relaxation experiment.

Auto-correlation

The auto-correlation of an observable $a(x)$ for a system in equilibrium is

$$\mathbb{E}_\mu[a(x,0)a(x,t)] = \sum_x \mu(x)a(x,0)a(x,t)$$

$a(x,0) = a(x)$ is the observable at time $t = 0$. It can be propagated forward in time using the t-step transition matrix $p^t(x,y)$.

The propagated observable at time t is $a(x,t) = \sum_y p^t(x,y)a(y,0)$.

Using the eigenvlaues and eigenvectors of the transition matrix the autocorrelation can be written as

$$\mathbb{E}_\mu[a(x,0)a(x,t)] = \sum_i \lambda_i^t \langle a, r_i\rangle_\mu \langle l_i, a\rangle.$$

The fingerprint amplitudes γ_i are given by

$$\gamma_i = \langle a, r_i\rangle_\mu \langle l_i, a\rangle.$$

And the fingerprint time scales t_i are given by

$$t_i = -\frac{\tau}{\log|\lambda_i|}.$$

Cross-correlation

The cross-correlation of two observables $a(x)$, $b(x)$ is similarly given

$$\mathbb{E}_\mu[a(x,0)b(x,t)] = \sum_x \mu(x)a(x,0)b(x,t)$$

The fingerprint amplitudes γ_i are similarly given in terms of the eigenvectors

$$\gamma_i = \langle a, r_i\rangle_\mu \langle l_i, b\rangle.$$

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import fingerprint_correlation
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> a = np.array([1.0, 0.0, 0.0])
>>> ts, amp = fingerprint_correlation(T, a)
```

```
>>> ts
array([        inf,  9.49122158,  0.43429448])
```

```
>>> amp
array([ 0.20661157,  0.22727273,  0.02066116])
```

msmtools.analysis.fingerprint_relaxation

msmtools.analysis.**fingerprint_relaxation**(*T*, *p0*, *obs*, *tau=1*, *k=None*, *ncv=None*)

Dynamical fingerprint for relaxation experiment.

The dynamical fingerprint is given by the implied time-scale spectrum together with the corresponding amplitudes.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix
- **obs1** (*(M,) ndarray*) – Observable, represented as vector on state space
- **obs2** (*(M,) ndarray (optional)*) – Second observable, for cross-correlations
- **k** (*int (optional)*) – Number of time-scales and amplitudes to compute
- **tau** (*int (optional)*) – Lag time of given transition matrix, for correct time-scales
- **ncv** (*int (optional)*) – The number of Lanczos vectors generated, *ncv* must be greater than k; it is recommended that ncv > 2*k

Returns

- **timescales** (*(N,) ndarray*) – Time-scales of the transition matrix
- **amplitudes** (*(N,) ndarray*) – Amplitudes for the relaxation experiment

See also:

relaxation(), *fingerprint_correlation()*

References

Notes

Fingerprints are a combination of time-scale and amplitude spectrum for a equilibrium correlation or a non-equilibrium relaxation experiment.

Relaxation

A relaxation experiment looks at the time dependent expectation value of an observable for a system out of equilibrium

$$\mathbb{E}_{w_0}[a(x,t)] = \sum_x w_0(x) a(x,t) = \sum_x w_0(x) \sum_y p^t(x,y) a(y).$$

The fingerprint amplitudes γ_i are given by

$$\gamma_i = \langle w_0, r_i \rangle \langle l_i, a \rangle.$$

And the fingerprint time scales t_i are given by

$$t_i = -\frac{\tau}{\log|\lambda_i|}.$$

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import fingerprint_relaxation
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> p0 = np.array([1.0, 0.0, 0.0])
>>> a = np.array([1.0, 0.0, 0.0])
>>> ts, amp = fingerprint_relaxation(T, p0, a)
```

```
>>> ts
array([        inf,  9.49122158,  0.43429448])
```

```
>>> amp
array([ 0.45454545,  0.5       ,  0.04545455])
```

msmtools.analysis.expectation

msmtools.analysis.**expectation**(*T*, *a*, *mu=None*)

Equilibrium expectation value of a given observable.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix
- **a** (*(M,) ndarray*) – Observable vector
- **mu** (*(M,) ndarray (optional)*) – The stationary distribution of T. If given, the stationary distribution will not be recalculated (saving lots of time)

Returns **val** – Equilibrium expectation value fo the given observable

Return type float

Notes

The equilibrium expectation value of an observable a is defined as follows

$$\mathbb{E}_\mu[a] = \sum_i \mu_i a_i$$

$\mu = (\mu_i)$ is the stationary vector of the transition matrix T.

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import expectation
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> a = np.array([1.0, 0.0, 1.0])
>>> m_a = expectation(T, a)
>>> m_a
0.909090909...
```

msmtools.analysis.correlation

msmtools.analysis.**correlation**(*T*, *obs1*, *obs2=None*, *times=1*, *maxtime=None*, *k=None*, *ncv=None*, *return_times=False*)

Time-correlation for equilibrium experiment.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix

- **obs1** (*(M,) ndarray*) – Observable, represented as vector on state space
- **obs2** (*(M,) ndarray (optional)*) – Second observable, for cross-correlations
- **times** (*array-like of int (optional), default=(1)*) – List of times (in tau) at which to compute correlation
- **maxtime** (*int, optional, default=None*) – Maximum time step to use. Equivalent to . Alternative to times.
- **k** (*int (optional)*) – Number of eigenvalues and eigenvectors to use for computation
- **ncv** (*int (optional)*) – The number of Lanczos vectors generated, *ncv* must be greater than k; it is recommended that ncv > 2*k

Returns

- **correlations** (*ndarray*) – Correlation values at given times
- **times** (*ndarray, optional*) – time points at which the correlation was computed (if return_times=True)

References

Notes

Auto-correlation

The auto-correlation of an observable $a(x)$ for a system in equilibrium is

$$\mathbb{E}_\mu[a(x,0)a(x,t)] = \sum_x \mu(x)a(x,0)a(x,t)$$

$a(x,0) = a(x)$ is the observable at time $t = 0$. It can be propagated forward in time using the t-step transition matrix $p^t(x,y)$.

The propagated observable at time t is $a(x,t) = \sum_y p^t(x,y)a(y,0)$.

Using the eigenvlaues and eigenvectors of the transition matrix the autocorrelation can be written as

$$\mathbb{E}_\mu[a(x,0)a(x,t)] = \sum_i \lambda_i^t \langle a, r_i\rangle_\mu \langle l_i, a\rangle.$$

Cross-correlation

The cross-correlation of two observables $a(x)$, $b(x)$ is similarly given

$$\mathbb{E}_\mu[a(x,0)b(x,t)] = \sum_x \mu(x)a(x,0)b(x,t)$$

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import correlation
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> a = np.array([1.0, 0.0, 0.0])
>>> times = np.array([1, 5, 10, 20])
```

```
>>> corr = correlation(T, a, times=times)
>>> corr
array([ 0.40909091,  0.34081364,  0.28585667,  0.23424263])
```

msmtools.analysis.relaxation

msmtools.analysis.**relaxation**(*T*, *p0*, *obs*, *times=1*, *k=None*, *ncv=None*)

Relaxation experiment.

The relaxation experiment describes the time-evolution of an expectation value starting in a non-equilibrium situation.

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix
- **p0** (*(M,) ndarray (optional)*) – Initial distribution for a relaxation experiment
- **obs** (*(M,) ndarray*) – Observable, represented as vector on state space
- **times** (*list of int (optional)*) – List of times at which to compute expectation
- **k** (*int (optional)*) – Number of eigenvalues and eigenvectors to use for computation
- **ncv** (*int (optional)*) – The number of Lanczos vectors generated, *ncv* must be greater than k; it is recommended that ncv > 2*k

Returns **res** – Array of expectation value at given times

Return type ndarray

References

Notes

Relaxation

A relaxation experiment looks at the time dependent expectation value of an observable for a system out of equilibrium

$$\mathbb{E}_{w_0}[a(x,t)] = \sum_x w_0(x) a(x,t) = \sum_x w_0(x) \sum_y p^t(x,y) a(y).$$

Examples

```
>>> import numpy as np
>>> from msmtools.analysis import relaxation
```

```
>>> T = np.array([[0.9, 0.1, 0.0], [0.5, 0.0, 0.5], [0.0, 0.1, 0.9]])
>>> p0 = np.array([1.0, 0.0, 0.0])
>>> a = np.array([1.0, 1.0, 0.0])
>>> times = np.array([1, 5, 10, 20])
```

```
>>> rel = relaxation(T, p0, a, times=times)
>>> rel
array([ 1.        ,  0.8407    ,  0.71979377,  0.60624287])
```

Sensitivity analysis

stationary_distribution_sensitivity(T, j)	Sensitivity matrix of a stationary distribution element.
eigenvalue_sensitivity(T, k)	Sensitivity matrix of a specified eigenvalue.
timescale_sensitivity(T, k)	Sensitivity matrix of a specified time-scale.
eigenvector_sensitivity(T, k, j[, right])	Sensitivity matrix of a selected eigenvector element.
mfpt_sensitivity(T, target, i)	Sensitivity matrix of the mean first-passage time from specified state.
committor_sensitivity(T, A, B, i[, forward])	Sensitivity matrix of a specified committor entry.
expectation_sensitivity(T, a)	Sensitivity of expectation value of observable A=(a_i).

msmtools.analysis.stationary_distribution_sensitivity

msmtools.analysis.**stationary_distribution_sensitivity**(*T*, *j*)

Sensitivity matrix of a stationary distribution element.

Parameters

- **T** (*(M, M) ndarray*) – Transition matrix (stochastic matrix).
- **j** (*int*) – Index of stationary distribution element for which sensitivity matrix is computed.

Returns **S** – Sensitivity matrix for the specified element of the stationary distribution.

Return type (M, M) ndarray

msmtools.analysis.eigenvalue_sensitivity

msmtools.analysis.**eigenvalue_sensitivity**(*T*, *k*)

Sensitivity matrix of a specified eigenvalue.

Parameters

- **T** (*(M, M) ndarray*) – Transition matrix
- **k** (*int*) – Compute sensitivity matrix for k-th eigenvalue

Returns **S** – Sensitivity matrix for k-th eigenvalue.

Return type (M, M) ndarray

msmtools.analysis.timescale_sensitivity

msmtools.analysis.**timescale_sensitivity**(*T*, *k*)

Sensitivity matrix of a specified time-scale.

Parameters

- **T** (*(M, M) ndarray*) – Transition matrix

- **k** (*int*) – Compute sensitivity matrix for the k-th time-scale.

Returns **S** – Sensitivity matrix for the k-th time-scale.

Return type (M, M) ndarray

msmtools.analysis.eigenvector_sensitivity

msmtools.analysis.**eigenvector_sensitivity**(*T*, *k*, *j*, *right=True*)

Sensitivity matrix of a selected eigenvector element.

Parameters

- **T** (*(M, M) ndarray*) – Transition matrix (stochastic matrix).
- **k** (*int*) – Eigenvector index
- **j** (*int*) – Element index
- **right** (*bool*) – If True compute for right eigenvector, otherwise compute for left eigenvector.

Returns **S** – Sensitivity matrix for the j-th element of the k-th eigenvector.

Return type (M, M) ndarray

msmtools.analysis.mfpt_sensitivity

msmtools.analysis.**mfpt_sensitivity**(*T*, *target*, *i*)

Sensitivity matrix of the mean first-passage time from specified state.

Parameters

- **T** (*(M, M) ndarray*) – Transition matrix
- **target** (*int or list*) – Target state or set for mfpt computation
- **i** (*int*) – Compute the sensitivity for state *i*

Returns **S** – Sensitivity matrix for specified state

Return type (M, M) ndarray

msmtools.analysis.committor_sensitivity

msmtools.analysis.**committor_sensitivity**(*T*, *A*, *B*, *i*, *forward=True*)

Sensitivity matrix of a specified committor entry.

Parameters

- **T** (*(M, M) ndarray*) – Transition matrix
- **A** (*array_like*) – List of integer state labels for set A
- **B** (*array_like*) – List of integer state labels for set B
- **i** (*int*) – Compute the sensitivity for committor entry *i*
- **forward** (*bool (optional)*) – Compute the forward committor. If forward is False compute the backward committor.

Returns **S** – Sensitivity matrix of the specified committor entry.

Return type (M, M) ndarray

msmtools.analysis.expectation_sensitivity

msmtools.analysis.**expectation_sensitivity**(*T*, *a*)
Sensitivity of expectation value of observable A=(a_i).

Parameters

- **T** (*(M, M) ndarray*) – Transition matrix
- **a** (*(M,) ndarray*) – Observable, a[i] is the value of the observable at state i.

Returns **S** – Sensitivity matrix of the expectation value.

Return type (M, M) ndarray

1.2.3 flux - Reactive flux an transition pathways (msmtools.flux)

This module contains functions to compute reactive flux networks and find dominant reaction pathways in such networks.

TPT-object

tpt(T, A, B[, mu, qminus, qplus, rate_matrix])	Computes the A->B reactive flux using transition path theory (TPT)
ReactiveFlux(A, B, flux[, mu, qminus, ...])	A->B reactive flux from transition path theory (TPT)

msmtools.flux.tpt

msmtools.flux.**tpt**(*T*, *A*, *B*, *mu=None*, *qminus=None*, *qplus=None*, *rate_matrix=False*)
Computes the A->B reactive flux using transition path theory (TPT)

Parameters

- **T** (*(M, M) ndarray or scipy.sparse matrix*) – Transition matrix (default) or Rate matrix (if rate_matrix=True)
- **A** (*array_like*) – List of integer state labels for set A
- **B** (*array_like*) – List of integer state labels for set B
- **mu** (*(M,) ndarray (optional)*) – Stationary vector
- **qminus** (*(M,) ndarray (optional)*) – Backward committor for A->B reaction
- **qplus** (*(M,) ndarray (optional)*) – Forward committor for A-> B reaction
- **= False** (*rate_matrix*) – By default (False), T is a transition matrix. If set to True, T is a rate matrix.

Returns **tpt** – A python object containing the reactive A->B flux network and several additional quantities, such as stationary probability, committors and set definitions.

Return type msmtools.flux.ReactiveFlux object

Notes

The central object used in transition path theory is the forward and backward committor function.

TPT (originally introduced in [1]) for continous systems has a discrete version outlined in [2]. Here, we use the transition matrix formulation described in [3].

See also:

msmtools.analysis.committor(), *ReactiveFlux()*

References

msmtools.flux.ReactiveFlux

class msmtools.flux.**ReactiveFlux**(*A*, *B*, *flux*, *mu=None*, *qminus=None*, *qplus=None*, *gross_flux=None*)

A->B reactive flux from transition path theory (TPT)

This object describes a reactive flux, i.e. a network of fluxes from a set of source states A, to a set of sink states B, via a set of intermediate nodes. Every node has three properties: the stationary probability mu, the forward committor qplus and the backward committor qminus. Every pair of edges has the following properties: a flux, generally a net flux that has no unnecessary back-fluxes, and optionally a gross flux.

Flux objects can be used to compute transition pathways (and their weights) from A to B, the total flux, the total transition rate or mean first passage time, and they can be coarse-grained onto a set discretization of the node set.

Fluxes can be computed in EMMA using transition path theory - see `msmtools.tpt()`

Parameters

- **A** (*array_like*) – List of integer state labels for set A
- **B** (*array_like*) – List of integer state labels for set B
- **flux** (*(n,n) ndarray or scipy sparse matrix*) – effective or net flux of A->B pathways
- **mu** (*(n,) ndarray (optional)*) – Stationary vector
- **qminus** (*(n,) ndarray (optional)*) – Backward committor for A->B reaction
- **qplus** (*(n,) ndarray (optional)*) – Forward committor for A-> B reaction
- **gross_flux** (*(n,n) ndarray or scipy sparse matrix*) – gross flux of A->B pathways, if available

Notes

Reactive flux contains a flux network from educt states (A) to product states (B).

See also:

`msmtools.tpt`

__init__(*A*, *B*, *flux*, *mu=None*, *qminus=None*, *qplus=None*, *gross_flux=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

__init__(A, B, flux[, mu, qminus, qplus, ...])	x.__init__(...) initializes x; see help(type(x)) for signature
coarse_grain(user_sets)	Coarse-grains the flux onto user-defined sets.
major_flux([fraction])	Returns the main pathway part of the net flux comprising at most the requested fraction of the full flux.
pathways([fraction, maxiter])	Decompose flux network into dominant reaction paths.

Attributes

A	Returns the set of reactant (source) states.
B	Returns the set of product (target) states
I	Returns the set of intermediate states
backward_committor	Returns the backward committor probability
committor	Returns the forward committor probability
flux	Returns the effective or net flux
forward_committor	Returns the forward committor probability
gross_flux	Returns the gross A–>B flux
mfpt	Returns the rate (inverse mfpt) of A–>B transitions
net_flux	Returns the effective or net flux
nstates	Returns the number of states.
rate	Returns the rate (inverse mfpt) of A–>B transitions
stationary_distribution	Returns the stationary distribution
total_flux	Returns the total flux

A
Returns the set of reactant (source) states.

B
Returns the set of product (target) states

I
Returns the set of intermediate states

backward_committor
Returns the backward committor probability

coarse_grain(*user_sets*)
Coarse-grains the flux onto user-defined sets.

Parameters **user_sets** (*list of int-iterables*) – sets of states that shall be distinguished in the coarse-grained flux.

Returns (sets, tpt) – sets contains the sets tpt is computed on. The tpt states of the new tpt object correspond to these sets of states in this order. Sets might be identical, if the user has already provided a complete partition that respects the boundary between A, B and the intermediates. If not, Sets will have more members than provided by the user, containing the "remainder" states and reflecting the splitting at the A and B boundaries. tpt contains a new tpt object for the coarse-grained flux. All its quantities (gross_flux, net_flux, A, B, committor, backward_committor) are coarse-grained to sets.

Return type (list of int-iterables, tpt-object)

Notes

All user-specified sets will be split (if necessary) to preserve the boundary between A, B and the intermediate states.

committor
Returns the forward committor probability

flux
Returns the effective or net flux

forward_committor
Returns the forward committor probability

gross_flux
Returns the gross A–>B flux

major_flux (*fraction=0.9*)
Returns the main pathway part of the net flux comprising at most the requested fraction of the full flux.

mfpt
Returns the rate (inverse mfpt) of A–>B transitions

net_flux
Returns the effective or net flux

nstates
Returns the number of states.

pathways (*fraction=1.0*, *maxiter=1000*)
Decompose flux network into dominant reaction paths.

Parameters

- **fraction** (*float, optional*) – Fraction of total flux to assemble in pathway decomposition
- **maxiter** (*int, optional*) – Maximum number of pathways for decomposition

Returns

- **paths** (*list*) – List of dominant reaction pathways
- **capacities** (*list*) – List of capacities corresponding to each reactions pathway in paths

References

rate
Returns the rate (inverse mfpt) of A–>B transitions

stationary_distribution
Returns the stationary distribution

total_flux
Returns the total flux

Reactive flux

flux_matrix(T, pi, qminus, qplus[, netflux])	Compute the TPT flux network for the reaction A–>B.
to_netflux(flux)	Compute the netflux from the gross flux.
flux_production(F)	Returns the net flux production for all states
flux_producers(F[, rtol, atol])	Return indexes of states that are net flux producers.
flux_consumers(F[, rtol, atol])	Return indexes of states that are net flux producers.
coarsegrain(F, sets)	Coarse-grains the flux to the given sets.

msmtools.flux.flux_matrix

msmtools.flux.**flux_matrix**(*T*, *pi*, *qminus*, *qplus*, *netflux=True*)

Compute the TPT flux network for the reaction A–>B.

Parameters

- **T** (*(M, M) ndarray*) – transition matrix
- **pi** (*(M,) ndarray*) – Stationary distribution corresponding to T
- **qminus** (*(M,) ndarray*) – Backward comittor
- **qplus** (*(M,) ndarray*) – Forward committor
- **netflux** (*boolean*) – True: net flux matrix will be computed False: gross flux matrix will be computed

Returns **flux** – Matrix of flux values between pairs of states.

Return type (M, M) ndarray

Notes

Computation of the flux network relies on transition path theory (TPT) [1]. Here we use discrete transition path theory [2] in the transition matrix formulation [3].

See also:

committor.forward_committor(), committor.backward_committor()

Notes

Computation of the flux network relies on transition path theory (TPT). The central object used in transition path theory is the forward and backward comittor function.

The TPT (gross) flux is defined as

$$f_{ij} = \begin{cases} \pi_i q_i^{(-)} p_{ij} q_j^{(+)} & i \neq j \\ 0 & i = j \end{cases}$$

The TPT net flux is then defined as

$$f_{ij} = \max\{f_{ij} - f_{ji}, 0\} \quad \forall i, j.$$

References

msmtools.flux.to_netflux

msmtools.flux.**to_netflux**(*flux*)

Compute the netflux from the gross flux.

Parameters **flux** (*(M, M) ndarray*) – Matrix of flux values between pairs of states.

Returns **netflux** – Matrix of netflux values between pairs of states.

Return type (M, M) ndarray

Notes

The netflux or effective current is defined as

$$f_{ij}^{+} = \max\{f_{ij} - f_{ji}, 0\}$$

f_{ij} is the flux for the transition from A to B.

References

msmtools.flux.flux_production

msmtools.flux.**flux_production**(*F*)

Returns the net flux production for all states

Parameters **F** (*(M, M) ndarray*) – Matrix of flux values between pairs of states.

Returns **prod** – Array containing flux production (positive) or consumption (negative) at each state

Return type (M,) ndarray

msmtools.flux.flux_producers

msmtools.flux.**flux_producers**(*F*, *rtol=1e-05*, *atol=1e-12*)

Return indexes of states that are net flux producers.

Parameters

- **F** (*(M, M) ndarray*) – Matrix of flux values between pairs of states.
- **rtol** (*float*) – relative tolerance. fulfilled if max(outflux-influx, 0) / max(outflux,influx) < rtol
- **atol** (*float*) – absolute tolerance. fulfilled if max(outflux-influx, 0) < atol

Returns **producers** – indexes of states that are net flux producers. May include “dirty” producers, i.e. states that have influx but still produce more outflux and thereby violate flux conservation.

Return type (M,) ndarray of int

msmtools.flux.flux_consumers

msmtools.flux.**flux_consumers**(*F*, *rtol=1e-05*, *atol=1e-12*)

Return indexes of states that are net flux producers.

Parameters

- **F** (*(M, M) ndarray*) – Matrix of flux values between pairs of states.
- **rtol** (*float*) – relative tolerance. fulfilled if max(outflux-influx, 0) / max(outflux,influx) < rtol
- **atol** (*float*) – absolute tolerance. fulfilled if max(outflux-influx, 0) < atol

Returns **producers** – indexes of states that are net flux producers. May include "dirty" producers, i.e. states that have influx but still produce more outflux and thereby violate flux conservation.

Return type (M,) ndarray of int

msmtools.flux.coarsegrain

msmtools.flux.**coarsegrain**(*F*, *sets*)

Coarse-grains the flux to the given sets.

Parameters

- **F** (*(n, n) ndarray or scipy.sparse matrix*) – Matrix of flux values between pairs of states.
- **sets** (*list of array-like of ints*) – The sets of states onto which the flux is coarse-grained.

Notes

The coarse grained flux is defined as

$$fc_{I,J} = \sum_{i \in I, j \in J} f_{i,j}$$

Note that if you coarse-grain a net flux, it does n ot necessarily have a net flux property anymore. If want to make sure you get a netflux, use to_netflux(coarsegrain(F,sets)).

References

Reaction rates and fluxes

total_flux(F[, A])	Compute the total flux, or turnover flux, that is produced by
rate(totflux, pi, qminus)	Transition rate for reaction A to B.
mfpt(totflux, pi, qminus)	Mean first passage time for reaction A to B.

msmtools.flux.total_flux

msmtools.flux.**total_flux**(*F*, *A=None*)

Compute the total flux, or turnover flux, that is produced by the flux sources and consumed by the flux sinks.

Parameters

- **F** (*(M, M) ndarray*) – Matrix of flux values between pairs of states.
- **A** (*array_like (optional)*) – List of integer state labels for set A (reactant)

Returns **F** – The total flux, or turnover flux, that is produced by the flux sources and consumed by the flux sinks

Return type float

References

msmtools.flux.rate

msmtools.flux.**rate**(*totflux*, *pi*, *qminus*)

Transition rate for reaction A to B.

Parameters

- **totflux** (*float*) – The total flux between reactant and product
- **pi** (*(M,) ndarray*) – Stationary distribution
- **qminus** (*(M,) ndarray*) – Backward comittor

Returns **kAB** – The reaction rate (per time step of the Markov chain)

Return type float

See also:

committor(), *total_flux()*, *flux_matrix()*

Notes

Computation of the rate relies on discrete transition path theory (TPT). The transition rate, i.e. the total number of reaction events per time step, is given in [1] as:

$$k_{AB} = \frac{1}{F} \sum_i \pi_i q_i^{(-)}$$

F is the total flux for the transition from A to B.

References

msmtools.flux.mfpt

msmtools.flux.**mfpt**(*totflux*, *pi*, *qminus*)

Mean first passage time for reaction A to B.

Parameters

- **totflux** (*float*) – The total flux between reactant and product

- **pi** (*(M,) ndarray*) – Stationary distribution
- **qminus** (*(M,) ndarray*) – Backward comittor

Returns **tAB** – The mean first-passage time for the A to B reaction

Return type float

See also:

rate()

Notes

Equal to the inverse rate, see [1].

References

Pathway decomposition

pathways(F, A, B[, fraction, maxiter])	Decompose flux network into dominant reaction paths.

msmtools.flux.pathways

msmtools.flux.**pathways**(*F*, *A*, *B*, *fraction=1.0*, *maxiter=1000*)

Decompose flux network into dominant reaction paths.

Parameters

- **F** (*(M, M) scipy.sparse matrix*) – The flux network (matrix of netflux values)
- **A** (*array_like*) – The set of starting states
- **B** (*array_like*) – The set of end states
- **fraction** (*float, optional*) – Fraction of total flux to assemble in pathway decomposition
- **maxiter** (*int, optional*) – Maximum number of pathways for decomposition

Returns

- **paths** (*list*) – List of dominant reaction pathways
- **capacities** (*list*) – List of capacities corresponding to each reactions pathway in paths

Notes

The default value for fraction is 1.0, i.e. all dominant reaction pathways for the flux network are computed. For large netorks the number of possible reaction paths can increase rapidly so that it becomes prohibitevely expensive to compute all possible reaction paths. To prevent this from happening maxiter sets the maximum number of reaction pathways that will be computed.

For large flux networks it might be necessary to decrease fraction or to increase maxiter. It is advisable to begin with a small value for fraction and monitor the number of pathways returned when increasing the value of fraction.

References

1.2.4 dtraj - Discrete trajectories functions (`msmtools.dtraj`)

Discrete trajectory io

read_discrete_trajectory(filename)	Read discrete trajectory from ascii file.
read_dtraj(filename)	Read discrete trajectory from ascii file.
write_discrete_trajectory(filename, dtraj)	Write discrete trajectory to ascii file.
write_dtraj(filename, dtraj)	Write discrete trajectory to ascii file.
load_discrete_trajectory(filename)	Read discrete trajectory form binary file.
load_dtraj(filename)	Read discrete trajectory form binary file.
save_discrete_trajectory(filename, dtraj)	Write discrete trajectory to binary file.
save_dtraj(filename, dtraj)	Write discrete trajectory to binary file.

msmtools.dtraj.read_discrete_trajectory

msmtools.dtraj.**read_discrete_trajectory**(*filename*)

Read discrete trajectory from ascii file.

Parameters **filename** (*str*) – The filename of the discretized trajectory file. The filename can either contain the full or the relative path to the file.

Returns **dtraj** – Discrete state trajectory.

Return type (M,) ndarray of int

See also:

write_discrete_trajectory()

Notes

The discrete trajectory file contains a single column with integer entries.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import write_discrete_trajectory, read_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp(suffix=".dtraj")
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as ascii file)

```
>>> write_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = read_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

msmtools.dtraj.read_dtraj

msmtools.dtraj.**read_dtraj**(*filename*)

Read discrete trajectory from ascii file.

Parameters **filename** (*str*) – The filename of the discretized trajectory file. The filename can either contain the full or the relative path to the file.

Returns **dtraj** – Discrete state trajectory.

Return type (M,) ndarray of int

See also:

write_discrete_trajectory()

Notes

The discrete trajectory file contains a single column with integer entries.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import write_discrete_trajectory, read_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp(suffix=".dtraj")
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as ascii file)

```
>>> write_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = read_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

msmtools.dtraj.write_discrete_trajectory

msmtools.dtraj.**write_discrete_trajectory**(*filename*, *dtraj*)

Write discrete trajectory to ascii file.

Parameters

- **filename** (*str*) – The filename of the discrete state trajectory file. The filename can either contain the full or the relative path to the file.
- **dtraj** (*array-like of int*) – Discrete state trajectory

See also:

read_discrete_trajectory()

Notes

The discrete trajectory is written to a single column ascii file with integer entries.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import write_discrete_trajectory, read_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp()
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as ascii file)

```
>>> write_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = read_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

msmtools.dtraj.write_dtraj

msmtools.dtraj.**write_dtraj**(*filename*, *dtraj*)

Write discrete trajectory to ascii file.

Parameters

- **filename** (*str*) – The filename of the discrete state trajectory file. The filename can either contain the full or the relative path to the file.

- **dtraj** (*array-like of int*) – Discrete state trajectory

See also:

read_discrete_trajectory()

Notes

The discrete trajectory is written to a single column ascii file with integer entries.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import write_discrete_trajectory, read_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp()
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as ascii file)

```
>>> write_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = read_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

msmtools.dtraj.load_discrete_trajectory

msmtools.dtraj.**load_discrete_trajectory**(*filename*)

Read discrete trajectory form binary file.

Parameters **filename** (*str*) – The filename of the discrete state trajectory file. The filename can either contain the full or the relative path to the file.

Returns **dtraj** – Discrete state trajectory

Return type (M,) ndarray of int

See also:

save_discrete_trajectory()

Notes

The binary file is a one dimensional numpy array of integers stored in numpy .npy format.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import load_discrete_trajectory, save_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp(suffix='.npy')
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as npy file)

```
>>> save_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = load_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

msmtools.dtraj.load_dtraj

msmtools.dtraj.**load_dtraj**(*filename*)

Read discrete trajectory form binary file.

Parameters **filename** (*str*) – The filename of the discrete state trajectory file. The filename can either contain the full or the relative path to the file.

Returns **dtraj** – Discrete state trajectory

Return type (M,) ndarray of int

See also:

save_discrete_trajectory()

Notes

The binary file is a one dimensional numpy array of integers stored in numpy .npy format.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import load_discrete_trajectory, save_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp(suffix='.npy')
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as npy file)

```
>>> save_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = load_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

msmtools.dtraj.save_discrete_trajectory

msmtools.dtraj.**save_discrete_trajectory**(*filename*, *dtraj*)

Write discrete trajectory to binary file.

Parameters

- **filename** (*str*) – The filename of the discrete state trajectory file. The filename can either contain the full or the relative path to the file.
- **dtraj** (*array-like of int*) – Discrete state trajectory

See also:

load_discrete_trajectory()

Notes

The discrete trajectory is stored as ndarray of integers in numpy .npy format.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import load_discrete_trajectory, save_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp(suffix='.npy')
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as npy file)

```
>>> save_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = load_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

```
>>> os.unlink(tmpfile)
```

msmtools.dtraj.save_dtraj

msmtools.dtraj.**save_dtraj**(*filename*, *dtraj*)

Write discrete trajectory to binary file.

Parameters

- **filename** (*str*) – The filename of the discrete state trajectory file. The filename can either contain the full or the relative path to the file.
- **dtraj** (*array-like of int*) – Discrete state trajectory

See also:

load_discrete_trajectory()

Notes

The discrete trajectory is stored as ndarray of integers in numpy .npy format.

Examples

```
>>> import numpy as np
>>> import os
>>> from tempfile import mktemp
>>> from msmtools.dtraj import load_discrete_trajectory, save_discrete_trajectory
```

Use temporary file

```
>>> tmpfile = mktemp(suffix='.npy')
```

Discrete trajectory

```
>>> dtraj = np.array([0, 1, 0, 0, 1, 1, 0])
```

Write to disk (as npy file)

```
>>> save_discrete_trajectory(tmpfile, dtraj)
```

Read from disk

```
>>> X = load_discrete_trajectory(tmpfile)
>>> X
array([0, 1, 0, 0, 1, 1, 0])
```

```
>>> os.unlink(tmpfile)
```

Simple statistics

count_states(dtrajs[, ignore_negative])	returns a histogram count
visited_set(dtrajs)	returns the set of states that have at least one count
number_of_states(dtrajs[, only_used])	returns the number of states in the given trajectories.
index_states(dtrajs[, subset])	Generates a trajectory/time indexes for the given list of states

msmtools.dtraj.count_states

msmtools.dtraj.**count_states**(*dtrajs*, *ignore_negative=False*)

returns a histogram count

Parameters

- **dtrajs** (*array_like or list of array_like*) – Discretized trajectory or list of discretized trajectories
- **bool, default=False** (*ignore_negative,*) – Ignore negative elements. By default, a negative element will cause an exception

Returns **count** – the number of occurrances of each state. n=max+1 where max is the largest state index found.

Return type ndarray((n), dtype=int)

msmtools.dtraj.visited_set

msmtools.dtraj.**visited_set**(*dtrajs*)

returns the set of states that have at least one count

Parameters **dtraj** (*array_like or list of array_like*) – Discretized trajectory or list of discretized trajectories

Returns **vis** – the set of states that have at least one count.

Return type ndarray((n), dtype=int)

msmtools.dtraj.number_of_states

msmtools.dtraj.**number_of_states**(*dtrajs*, *only_used=False*)

returns the number of states in the given trajectories.

Parameters

- **dtraj** (*array_like or list of array_like*) – Discretized trajectory or list of discretized trajectories
- **= False** (*only_used*) – If False, will return max+1, where max is the largest index used. If True, will return the number of states that occur at least once.

msmtools.dtraj.index_states

msmtools.dtraj.**index_states**(*dtrajs*, *subset=None*)

Generates a trajectory/time indexes for the given list of states

Parameters

- **dtraj** (*array_like or list of array_like*) – Discretized trajectory or list of discretized trajectories
- **subset** (*ndarray((n)), optional, default = None*) – array of states to be indexed. By default all states in dtrajs will be used

Returns **indexes** – For each state, all trajectory and time indexes where this state occurs. Each matrix has a number of rows equal to the number of occurances of the corresponding state, with rows consisting of a tuple (i, t), where i is the index of the trajectory and t is the time index within the trajectory.

Return type list of ndarray((N_i, 2))

Sampling trajectory indexes

sample_indexes_by_distribution(indexes, ...)	Samples trajectory/time indexes according to the given probability distributions
sample_indexes_by_state(indexes, nsample[, ...])	Samples trajectory/time indexes according to the given sequence of states
sample_indexes_by_sequence(indexes, sequence)	Samples trajectory/time indexes according to the given sequence of states

msmtools.dtraj.sample_indexes_by_distribution

msmtools.dtraj.**sample_indexes_by_distribution**(*indexes*, *distributions*, *nsample*)

Samples trajectory/time indexes according to the given probability distributions

Parameters

- **indexes** (*list of ndarray((N_i, 2))*) – For each state, all trajectory and time indexes where this state occurs. Each matrix has a number of rows equal to the number of occurrences of the corresponding state, with rows consisting of a tuple (i, t), where i is the index of the trajectory and t is the time index within the trajectory.
- **distributions** (*list or array of ndarray ((n))*) – m distributions over states. Each distribution must be of length n and must sum up to 1.0
- **nsample** (*int*) – Number of samples per distribution. If replace = False, the number of returned samples per state could be smaller if less than nsample indexes are available for a state.

Returns **indexes** – List of the sampled indices by distribution. Each element is an index array with a number of rows equal to nsample, with rows consisting of a tuple (i, t), where i is the index of the trajectory and t is the time index within the trajectory.

Return type length m list of ndarray((nsample, 2))

msmtools.dtraj.sample_indexes_by_state

msmtools.dtraj.**sample_indexes_by_state**(*indexes*, *nsample*, *subset=None*, *replace=True*)
Samples trajectory/time indexes according to the given sequence of states

Parameters

- **indexes** (*list of ndarray((N_i, 2))*) – For each state, all trajectory and time indexes where this state occurs. Each matrix has a number of rows equal to the number of occurrences of the corresponding state, with rows consisting of a tuple (i, t), where i is the index of the trajectory and t is the time index within the trajectory.
- **nsample** (*int*) – Number of samples per state. If replace = False, the number of returned samples per state could be smaller if less than nsample indexes are available for a state.
- **subset** (*ndarray((n)), optional, default = None*) – array of states to be indexed. By default all states in dtrajs will be used
- **replace** (*boolean, optional*) – Whether the sample is with or without replacement

Returns indexes – List of the sampled indices by state. Each element is an index array with a number of rows equal to N=len(sequence), with rows consisting of a tuple (i, t), where i is the index of the trajectory and t is the time index within the trajectory.

Return type list of ndarray((N, 2))

msmtools.dtraj.sample_indexes_by_sequence

msmtools.dtraj.**sample_indexes_by_sequence**(*indexes*, *sequence*)
Samples trajectory/time indexes according to the given sequence of states

Parameters

- **indexes** (*list of ndarray((N_i, 2))*) – For each state, all trajectory and time indexes where this state occurs. Each matrix has a number of rows equal to the number of occurrences of the corresponding state, with rows consisting of a tuple (i, t), where i is the index of the trajectory and t is the time index within the trajectory.
- **sequence** (*array of integers*) – A sequence of discrete states. For each state, a trajectory/time index will be sampled at which dtrajs have an occurrences of this state

Returns indexes – The sampled index sequence. Index array with a number of rows equal to N=len(sequence), with rows consisting of a tuple (i, t), where i is the index of the trajectory and t is the time index within the trajectory.

Return type ndarray((N, 2))

1.2.5 generation - MSM generation tools (msmtools.generation)

This module contains function to generate simple MSMs.

Metropolis-Hastings chain

transition_matrix_metropolis_1d(E[, d])	Transition matrix describing the Metropolis chain jumping between neighbors in a discrete 1D energy landscape.
generate_traj(P, N[, start, stop, dt])	Generates a realization of the Markov chain with transition matrix P.

msmtools.generation.transition_matrix_metropolis_1d

msmtools.generation.**transition_matrix_metropolis_1d**(*E*, *d=1.0*)

Transition matrix describing the Metropolis chain jumping between neighbors in a discrete 1D energy landscape.

Parameters

- **E** (*(M,) ndarray*) – Energies in units of kT
- **d** (*float (optional)*) – Diffusivity of the chain, d in (0, 1]

Returns **P** – Transition matrix of the Markov chain

Return type (M, M) ndarray

Notes

Transition probabilities are computed as .. math:

```
p_{i,i-1} &=& 0.5 d \min \left{ 1.0, \mathrm{e}^{-(E_{i-1} - E_i)} \right}, \\
p_{i,i+1} &=& 0.5 d \min \left{ 1.0, \mathrm{e}^{-(E_{i+1} - E_i)} \right}, \\
p_{i,i}   &=& 1.0 - p_{i,i-1} - p_{i,i+1}.
```

msmtools.generation.generate_traj

msmtools.generation.**generate_traj**(*P*, *N*, *start=None*, *stop=None*, *dt=1*)

Generates a realization of the Markov chain with transition matrix P.

Parameters

- **P** (*(n, n) ndarray*) – transition matrix
- **N** (*int*) – trajectory length
- **start** (*int, optional, default = None*) – starting state. If not given, will sample from the stationary distribution of P
- **stop** (*int or int-array-like, optional, default = None*) – stopping set. If given, the trajectory will be stopped before N steps once a state of the stop set is reached
- **dt** (*int*) – trajectory will be saved every dt time steps. Internally, the dt'th power of P is taken to ensure a more efficient simulation.

Returns **traj_sliced** – A discrete trajectory with length N/dt

Return type (N/dt,) ndarray

CHAPTER 2

Development

2.1 Changelog

2.1.1 1.2.4 (11-12-18)

Fixes: - Effective count matrix parallel evaluation fixes. #116, #117

2.1.2 1.2.3 (7-27-18)

New features

- Effective count matrix can be evaluated in parallel. #112

2.1.3 1.2.2 (6-25-18)

New features

- Added new transition matrix estimator which uses a primal-dual interior-point iteration scheme. #82

Fixes:

- Fixed a corner case, in which the pathway decomposition could fail (no more paths left). #107

2.1.4 1.2.1 (5-16-17)

New features

- Added fast reversible transition matrix estimation. #94

Fixes:

- Fixed some minor issues in rate matrix estimation. #97 #98

2.1.5 1.2 (10-24-16)

New features:

- Continous MSM (rate matrix) estimation

2.1.6 1.1.4 (9-23-16)

Fixes:

- Fixed sparsity pattern check in transition matrix sampler. #91, thanks @fabian-paul

2.1.7 1.1.3 (8-10-16)

New features:

- added documentation

2.2 Developer's Guide

2.2.1 Contributing

Basic Idea

We use the "devel" branch to develop pyEMMA. When "devel" has reached a mature state in terms of current functionality and stability, we merge "devel" into the master branch. This happens at the time when a release is made.

In order to develop certain features you should not work on "devel" directly either, but rather branch it to a new personal feature branch (here you can do whatever you want). Then, after testing your feature, you can offer the changes to be merged on to the devel branch by doing a *pull request* (see below). If accepted, that branch will be merged into devel, and unless overridden by other changes your feature will make it eventually to master and the next release.

Why

- Always have a tested and stable master branch.
- Avoid interfering with other developers until changes are merged.

How

One of the package maintainers merges the development branch(es) periodically. All you need to do is to make your changes in the feature branch (see below for details), and then offer a pull request. When doing so, a bunch of automatic code tests will be run to test for direct or indirect bugs that have been introduced by the change. This is done by a continuous integration (CI) software like Jenkins http://jenkins-ci.org or Travis-CI http://travis-ci.org, the first one is open source and the second one is free for open source projects only. Again, you do not have to do anything here, as this happens automatically after a pull request. You will see the output of these tests in the pull request page on github.

Commit messages

Use commit messages in the style "[$package]: change" whenever the changes belong to one package or module. You can suppress "pyemma" (that is trivial) and "api" (which doesn't show up in the import).

E.g.:

```
[msm.analysis]: implemented sparse pcca
```

That way other developers and the package managers immediately know which modules have been changed and can watch out for possible cross-effects. Also this makes commits look uniform.

If you have a complex commit affecting several modules or packages, break it down into little pieces with easily understandable commit messages. This allows us to go back to intermediate stages of your work if something fails at the end.

Testing

We use Pythons unittest module to write test cases for all algorithm.

To run all tests invoke:

```
python setup.py test
```

or directly invoke nosetests in pyemma working copy:

```
nosetests $PYEMMA_DIR
```

It is encouraged to run all tests (if you are changing core features), you can also run individual tests by directly invoking them with the python interpreter.

Documentation

Every function, class, and module that you write must be documented. Please check out

> https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

how to do this. In short, after every function (class, module) header, there should be a docstring enclosed by """ ... """, containing a short and long description what the function does, a clear description of the input parameters, and the return value, and any relevant cross-references or citations. You can include Latex-style math in the docstring.

For a deeper understanding and reference please have a look at the Sphinx documentation

> http://sphinx-doc.org/

In particular, the API functions (those publicly visible to the external user) should be well documented.

To build the documentation you need the dependencies from the file requirements-build-docs.txt which you can install via pip:

```
pip install -r requirements-build-docs.txt
```

afterwards you are ready to build the documentation:

```
cd doc
make html
```

The HTML docs can then be found in the doc/build/html directory.

2.2.2 Workflow

A developer creates a feature branch "feature" and commits his or her work to this branch. When he or she is done with his work (have written at least a working test case for it), he or she pushes this feature branch to his or her fork and creates a pull request. The pull request can then be reviewed and merged upstream.

0. Get up to date - pull the latest changes from devel

```
# first get the latest changes
git pull
```

1. Compile extension modules (also works with conda distributions)

```
python setup.py develop
```

In contrast to *install*, which copies the development version into your package directory, the develop flag results in simply putting a link from your package directory into your development directory. That way local changes to python files are immediately active when you import the package. You only need to re-execute the above command, when a C extension was changed.

2. Create a new feature branch by copying from the devel branch and switch to it:

```
# switch to development branch
git checkout devel
# create new branch and switch to it
git checkout -b feature
```

3. Work on your feature branch. Here you can roam freely.

4. Write unit test and TEST IT (see above)! :-)

```
touch fancy_feat_test.py
# test the unit
python fancy_feat_test.py
# run the whole test-suite
# (to ensure that your newfeature has no side-effects)
cd $PYEMMA_DIR
python setup.py test
```

5. Commit your changes

```
git commit fancy_feat.py fancy_feat_test.py \
    -m "Implementation and unit test for fancy feature"
```

repeat 3.-5. as often as necessary to accomplish your task. Remember to split your changes into small commits.

6. Make changes available by pushing your commits to the server and creating a pull request

```
# push your branch to your fork on github
git push myfork feature
```

On github create a pull request from myfork/feature to origin/devel, see https://help.github.com/articles/using-pull-requests

Conclusions

- Feature branches allow you to work without interfering with others.

- The devel branch contains all tested implemented features.
- The devel branch is used to test for cross-effects between features.
- Work with pull request to ensure your changes are being tested automatically and can be reviewed.
- The master branch contains all tested features and represents the set of features that are suitable for public usage.

2.2.3 Publish a new release

1. Merge current devel branch into master

```
git checkout master; git merge devel
```

2. Make a new tag 'vmajor.minor.patch'. major means major release (major new functionalities), minor means minor changes and new functionalities, patch means no new functionality but just bugfixes or improvement to the docs.

```
git tag -m "release description" v1.1
```

3. IMPORTANT: first push, then push –tags

```
git push; git push --tags
```

4. Update conda recipes and perform binstar pushing (partially automatized)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

E

F

G

I

L

M

N

P

R

S

T

V

W