pyGPCCA

Release main

Bernhard Reuter

Oct 31, 2022

OVERVIEW

Generalized Perron Cluster Cluster Analysis program to coarse-grain reversible and non-reversible Markov state models.

Markov state models (MSM) enable the identification and analysis of metastable states and related kinetics in a very instructive manner. They are widely used, e.g., to model molecular or cellular kinetics. Common state-of-the-art Markov state modeling methods and tools are very well suited to model reversible processes in closed equilibrium systems. However, most are not well suited to deal with non-reversible or even non-autonomous processes of non-equilibrium systems. To overcome this limitation, the Generalized Robust Perron Cluster Cluster Analysis (GPCCA or G-PCCA) was developed. The GPCCA method implemented in the *pyGPCCA* program readily handles equilibrium as well as non-equilibrium data by utilizing real Schur vectors instead of eigenvectors. *pyGPCCA* enables the semiautomatic coarse-graining of transition matrices representing the dynamics of the system under study. Utilizing *pyGPCCA*, metastable states as well as cyclic kinetics can be identified and modeled.

Key Contributors

pyGPCCA graph | = maintainer

- Bernhard Reuter: lead developer
- Michal Klein: developer, diverse contributions
- Marius Lange: developer, diverse contributions

CHAPTER

ONE

INSTALLATION

pyGPCCA requires Python >= 3.6 to run. If any problems arise, please consult the *Troubleshooting* section.

1.1 Methods

1.1.1 Conda

pyGPCCA is available as a conda package and can be installed as:

```
conda install -c conda-forge pygpcca
```

This is the recommended way of installing, since this package also includes PETSc/SLEPc libraries. We use PETSc/SLEPc internally to speed up the computation of the leading Schur vectors. These are optional dependencies - if they're not present, we compute a full Schur decomposition instead and sort it using the method introduced by Brandts (2002). Note that this scales cubically in sample number, making it essential to use PETSc/SLEPc for large sample numbers. PETSc/SLEPc implement iterative methods to only compute the leading Schur vectors, which is computationally much less expensive.

1.1.2 PyPI

In order to install *pyGPCCA* from The Python Package Index, run:

```
pip install pygpcca
# or with libraries utilizing PETSc/SLEPc
pip install pygpcca[slepc]
```

1.1.3 Development version

If you want to use the development version of *pyGPCCA* from GitHub, run:

```
pip install git+https://github.com/msmdev/pygpcca
```

1.2 Troubleshooting

During the installation of `petsc`, `petsc4py`, `slepc`, and `slepc4py`, the following error(s) might appear:

```
ERROR: Failed building wheel for <package name>
```

However, this should be fine if in the end, it also outputs:

```
Successfully installed <package name>
```

To quickly verify that the packages have been installed, you can run:

```
python3 -c "import petsc4py; import slepc4py; print(petsc4py.__version__, slepc4py.__
↪version__)"
```

1.2.1 Debian-based systems

Below are an alternative steps for installing PETSc/SLEPc, in case any problems arise, especially when installing from *PyPI*:

```
# install dependencies
sudo apt-get update -y
sudo apt-get install gcc gfortran libopenmpi-dev libblas-dev liblapack-dev petsc-dev␣
↪slepc-dev -y

# install a message passing interface for Python
pip install --user mpi4py

# install petsc and and petsc4py
pip install --user petsc
pip install --user petsc4py

# install slepc and slepc4py
pip install --user slepc
pip install --user slepc4py
```

1.2.2 macOS

The most robust way is to follow the PETSc installation guide and the SLEPc installation guide or to take a look at our continuous integration steps for macOS.

The installation steps can be roughly outlined as:

```
# install dependencies
brew install gcc open-mpi openblas lapack arpack

# follow the PETSc installation steps
# follow the SLEPc installation steps

# install petsc4py
pip install --user petsc4py
```

(continues on next page)

(continued from previous page)

```
# install slepc4py
pip install --user petsc4py
```

CHAPTER

TWO

API

pyGPCCA can be imported as:

```
import pygpcca as gp
```

2.1 Functions

pygpcca.stationary_distribution()	Compute stationary distribution of stochastic matrix P.
pygpcca.gpcca_coarsegrain(P, m[, eta, z, method])	Coarse-grain the transition matrix P into m sets using G-PCCA [Reuter18], [Reuter19].

2.1.1 pygpcca.stationary_distribution

pygpcca.**stationary_distribution**(P)
pygpcca.**stationary_distribution**(P)
pygpcca.**stationary_distribution**(P)

Compute stationary distribution of stochastic matrix P.

Parameters **P** (Union[ndarray, spmatrix]) – The transition matrix (row-stochastic).

Return type ndarray

Returns Vector of stationary probabilities.

Notes

The stationary distribution π is the left eigenvector corresponding to the non-degenerate eigenvalue $\lambda = 1$ of a reversible transition matrix P,

$$\pi^T P = \pi^T .$$

References

The code and docstring of this function origins (with some adjustments) from MSMTools, Copyright (c) 2015, 2014 Computational Molecular Biology Group, Freie Universitaet Berlin (GER).

2.1.2 pygpcca.gpcca_coarsegrain

pygpcca.**gpcca_coarsegrain**(*P*, *m*, *eta=None*, *z='LM'*, *method='brandts'*)

Coarse-grain the transition matrix *P* into *m* sets using G-PCCA [Reuter18], [Reuter19].

Performs optimized spectral clustering via G-PCCA and coarse-grains *P* such that the dominant Perron eigenvalues are preserved using:

$$P_c = (\chi^T D \chi)^{-1} (\chi^T D P \chi)$$

with D being a diagonal matrix with *eta* on its diagonal [Reuter18], [Reuter19].

Parameters

- **P** (Union[ndarray, spmatrix]) – The transition matrix (row-stochastic).
- **m** (Union[int, Tuple[int, int], List[int], Dict[str, int]]) – The number of clusters or a range where a search for potentially optimal cluster numbers is performed. Valid options are:
  - int: number of clusters to group into.
  - tuple: minimal and maximal number of clusters.
  - dict: minimal and maximal number of clusters given as `{'m_min': int, 'm_max': int}`.
- **eta** (Optional[ndarray]) – The input probability distribution of the (micro)states. In theory *eta* can be an arbitrary distribution as long as it is a valid probability distribution (i.e., sums up to 1). A neutral and valid choice would be the uniform distribution (default).

  In case of a reversible transition matrix, the stationary distribution can (but don't has to) be used here. In case of a non-reversible *P*, some initial or average distribution of the states might be chosen instead of the uniform distribution.

  Vector of shape *(n,)* which sums to 1. If *None* (default), uniform distribution is used.
- **z** (str) – Specifies which portion of the eigenvalue spectrum of *P* is to be sought. The returned invariant subspace of *P* will be associated with this part of the spectrum. Valid options are:
  - 'LM': largest magnitude (default).
  - 'LR': largest real part.
- **method** (str) – Which method to use to determine the invariant subspace. Valid options are:
  - 'brandts': perform a full Schur decomposition of *P* utilizing scipy.linalg.schur() (without the intrinsic sorting option, since it is flawed) and sort the returned Schur form *R* and Schur vector matrix *Q* afterwards using a routine published by Brandts [Brandts02]. This is well tested and thus the default method, although it is also the slowest choice.
  - 'krylov': calculate an orthonormal basis of the subspace associated with the *m* dominant eigenvalues of *P* using the Krylov-Schur method as implemented in SLEPc. This is the fastest choice and especially suitable for very large *P*, but it is still experimental.

See the installation instructions for more information.

Return type ndarray

Returns The coarse-grained row-stochastic transition matrix.

References

If you use this code or parts of it, please cite [Reuter19].

2.2 Classes

pygpcca.GPCCA(P[, eta, z, method])	G-PCCA spectral clustering method with optimized memberships [Reuter18], [Reuter19].

2.2.1 pygpcca.GPCCA

class pygpcca.**GPCCA**(*P*, *eta=None*, *z='LM'*, *method='brandts'*)

G-PCCA spectral clustering method with optimized memberships [Reuter18], [Reuter19].

Clusters the dominant *m* Schur vectors of a transition matrix.

This algorithm generates a fuzzy clustering such that the resulting membership functions are as crisp (characteristic) as possible.

Parameters

- **P** (Union[ndarray, spmatrix]) – The transition matrix (row-stochastic).
- **eta** (Optional[ndarray]) – The input probability distribution of the (micro)states. In theory *eta* can be an arbitrary distribution as long as it is a valid probability distribution (i.e., sums up to 1). A neutral and valid choice would be the uniform distribution (default).

  In case of a reversible transition matrix, the stationary distribution can (but don't has to) be used here. In case of a non-reversible *P*, some initial or average distribution of the states might be chosen instead of the uniform distribution.

  Vector of shape *(n,)* which sums to 1. If *None*, uniform distribution is used.
- **z** (str) – Specifies which portion of the eigenvalue spectrum of *P* is to be sought. The returned invariant subspace of *P* will be associated with this part of the spectrum. Valid options are:
  - 'LM': largest magnitude (default).
  - 'LR': largest real part.
- **method** (str) – Which method to use to determine the invariant subspace. Valid options are:
  - 'brandts': perform a full Schur decomposition of *P* utilizing scipy.linalg.schur() (without the intrinsic sorting option, since it is flawed) and sort the returned Schur form *R* and Schur vector matrix *Q* afterwards using a routine published by Brandts [Brandts02]. This is well tested and thus the default method, although it is also the slowest choice.

– 'krylov': calculate an orthonormal basis of the subspace associated with the *m* dominant eigenvalues of *P* using the Krylov-Schur method as implemented in `SLEPc`. This is the fastest choice and especially suitable for very large *P*, but it is still experimental.

See the installation instructions for more information.

References

If you use this code or parts of it, please cite [Reuter19].

Methods

minChi(m_min, m_max)	Calculate the minChi indicator (see [Reuter18]) for every $m \in [m_{min}, m_{max}]$.
optimize(m)	Full G-PCCA spectral clustering method with optimized memberships [Reuter18], [Reuter19].

pygpcca.GPCCA.minChi

GPCCA.**minChi**(*m_min*, *m_max*)

Calculate the minChi indicator (see [Reuter18]) for every $m \in [m_{min}, m_{max}]$.

The minChi indicator can be used to determine an interval $I \subset [m_{min}, m_{max}]$ of good (potentially optimal) numbers of clusters.

Afterwards either one $m \in I$) or the whole interval I is chosen as input to *optimize()* for further optimization.

Parameters

- **m_min** (int) – Minimal number of clusters to group into.
- **m_max** (int) – Maximal number of clusters to group into.

Return type List[float]

Returns List of minChi indicators for cluster numbers $m \in [m_{min}, m_{max}]$, see [Roeblitz13], [Reuter18].

pygpcca.GPCCA.optimize

GPCCA.**optimize**(*m*)

Full G-PCCA spectral clustering method with optimized memberships [Reuter18], [Reuter19].

It also has the option to optimize the number of clusters (macrostates) *m* as well.

If a single integer *m* is given, the method clusters the dominant *m* Schur vectors of the *transition_matrix*. The algorithm generates a fuzzy clustering such that the resulting membership functions *chi* are as crisp (characteristic) as possible, given *m*.

Instead of a single number of clusters *m*, a tuple or a dict `{'m_min': int, 'm_max': int}` containing a minimum and a maximum number of clusters can be given. This results in repeated execution of the G-PCCA core algorithm for $m \in [m_{min}, m_{max}]$. Among the resulting clusterings, the sharpest/crispest one (with maximal *crispness*) will be selected.

Parameters **m** (Union[int, Tuple[int, int], List[int], Dict[str, int]]) – The number of clusters or a range where a search for potentially optimal cluster numbers is performed. Valid options are:

- int: number of clusters to group into.
- tuple: minimal and maximal number of clusters.
- dict: minimal and maximal number of clusters given as {'m_min': int, 'm_max': int}.

See *minChi()* for selection of good (potentially optimal) number of clusters.

Return type *GPCCA*

Returns

Returns self and updates the following attributes:

- *coarse_grained_input_distribution*
- coarse_grained_stationary_distribution
- *coarse_grained_transition_matrix*
- *crispness_values*
- *dominant_eigenvalues*
- *input_distribution*
- *macrostate_assignment*
- *macrostate_sets*
- *memberships*
- *n_m*
- *optimal_crispness*
- *rotation_matrix*
- *schur_matrix*
- *schur_vectors*
- *stationary_probability*
- *top_eigenvalues*
- *transition_matrix*

Attributes

coarse_grained_input_distribution	Coarse grained input distribution of shape *(n_m,)*.
coarse_grained_stationary_probability	Coarse grained stationary distribution of shape *(n_m,)*.
coarse_grained_transition_matrix	Coarse grained transition matrix of shape *(n_m, n_m)*.
crispness_values	Vector of crispness values for clustering into the requested cluster numbers.
dominant_eigenvalues	Dominant *n_m* eigenvalues of *P*.

continues on next page

Table 4 – continued from previous page

input_distribution	Input probability distribution of the (micro)states.
macrostate_assignment	Crisp clustering using G-PCCA.
macrostate_sets	Crisp clustering using G-PCCA.
memberships	Array of shape *(n, n_m)* containing the membership χ_{ij} (or probability) of each microstate i (to be assigned) to each macrostate or cluster j.
n_m	Optimal number of clusters or macrostates to group the *n* microstates into.
optimal_crispness	Crispness for clustering into *n_m* clusters.
rotation_matrix	Optimized rotation matrix A.
schur_matrix	Ordered top left part of shape *(n_m, n_m)* of the real Schur matrix of P.
schur_vectors	Array Q of shape *(n, n_m)* with *n_m* sorted Schur vectors in the columns.
stationary_probability	Stationary probability distribution π of the microstates.
top_eigenvalues	Top *m* respective *m_max* eigenvalues of *P*.
transition_matrix	Row-stochastic transition matrix *P*.

pygpcca.GPCCA.coarse_grained_input_distribution

property GPCCA.**coarse_grained_input_distribution: Optional[numpy.ndarray]**

Coarse grained input distribution of shape *(n_m,)*.

$$\eta_c = \chi^T \eta$$

Return type Optional[ndarray]

pygpcca.GPCCA.coarse_grained_stationary_probability

property GPCCA.**coarse_grained_stationary_probability: Optional[numpy.ndarray]**

Coarse grained stationary distribution of shape *(n_m,)*.

$$\pi_c = \chi^T \pi$$

Return type Optional[ndarray]

pygpcca.GPCCA.coarse_grained_transition_matrix

property GPCCA.**coarse_grained_transition_matrix: Optional[numpy.ndarray]**

Coarse grained transition matrix of shape *(n_m, n_m)*.

$$P_c = (\chi^T D \chi)^{-1} (\chi^T D P \chi)$$

with D being a diagonal matrix with η on its diagonal.

Return type Optional[ndarray]

pygpcca.GPCCA.crispness_values

property GPCCA.**crispness_values: Optional[numpy.ndarray]**

Vector of crispness values for clustering into the requested cluster numbers.

The crispness $\xi \in [0, 1]$ quantifies the optimality of the solution (higher is better). It characterizes how crisp (sharp) the decomposition of the state space into *m* clusters is. It is given via (Eq. 17 from [Roeblitz13]):

$$\xi = (m - f_{opt})/m = \texttt{trace}(S)/m = \texttt{trace}(\tilde{D}\chi^T D\chi)/m$$

with D being a diagonal matrix with η on its diagonal.

Return type Optional[ndarray]

pygpcca.GPCCA.dominant_eigenvalues

property GPCCA.**dominant_eigenvalues: Optional[numpy.ndarray]**

Dominant *n_m* eigenvalues of *P*.

Vector of shape *(n_m,)* containing the *n_m* dominant eigenvalues of *P*.

Return type Optional[ndarray]

pygpcca.GPCCA.input_distribution

property GPCCA.**input_distribution: numpy.ndarray**

Input probability distribution of the (micro)states.

In theory η can be an arbitrary distribution as long as it is a valid probability distribution (i.e., sums up to 1). A neutral and valid choice would be the uniform distribution (default).

In case of a reversible transition matrix, the stationary distribution π can (but don't has to) be used here. In case of a non-reversible *P*, some initial or average distribution of the states might be chosen instead of the uniform distribution.

Vector of shape *(n,)* which sums to 1.

Return type ndarray

pygpcca.GPCCA.macrostate_assignment

property GPCCA.**macrostate_assignment: Optional[numpy.ndarray]**

Crisp clustering using G-PCCA.

This is recommended only for visualization purposes. You *cannot* compute any actual quantity of the coarse-grained kinetics without employing the fuzzy memberships!

Return type Optional[ndarray]

Returns Integer vector of shape *(n,)* containing the macrostate each microstate is located in.

References

The code and docstring of this property origins (with some adjustments) from MSMTools, Copyright (c) 2015, 2014 Computational Molecular Biology Group, Freie Universitaet Berlin (GER).

pygpcca.GPCCA.macrostate_sets

property GPCCA.**macrostate_sets: Optional[List[numpy.ndarray]]**

Crisp clustering using G-PCCA.

This is recommended only for visualization purposes. You *cannot* compute any actual quantity of the coarse-grained kinetics without employing the fuzzy memberships!

Return type Optional[List[ndarray]]

Returns

A list of length equal to *n_m*.

Each element is an array with microstate indexes contained in it.

References

The code and docstring of this property origins (with some adjustments) from MSMTools, Copyright (c) 2015, 2014 Computational Molecular Biology Group, Freie Universitaet Berlin (GER).

pygpcca.GPCCA.memberships

property GPCCA.**memberships: Optional[numpy.ndarray]**

Array of shape *(n, n_m)* containing the membership χ_{ij} (or probability) of each microstate i (to be assigned) to each macrostate or cluster j.

The rows sum to 1.

Return type Optional[ndarray]

pygpcca.GPCCA.n_m

property GPCCA.**n_m: Optional[int]**

Optimal number of clusters or macrostates to group the *n* microstates into.

Return type Optional[int]

pygpcca.GPCCA.optimal_crispness

property GPCCA.**optimal_crispness: Optional[float]**

Crispness for clustering into *n_m* clusters.

The crispness $\xi \in [0, 1]$ quantifies the optimality of the solution (higher is better). It characterizes how crisp (sharp) the decomposition of the state space into *m* clusters is. It is given via (Eq. 17 from [Roeblitz13]):

$$\xi = (m - f_{opt})/m = \texttt{trace}(S)/m = \texttt{trace}(\tilde{D}\chi^T D\chi)/m$$

with D being a diagonal matrix with η on its diagonal.

Return type Optional[float]

pygpcca.GPCCA.rotation_matrix

property GPCCA.**rotation_matrix: Optional[numpy.ndarray]**
Optimized rotation matrix A.

Array of shape *(n_m, n_m)* which rotates the dominant Schur vectors to yield the G-PCCA *memberships*, i.e. $\chi = XA$.

Return type Optional[ndarray]

pygpcca.GPCCA.schur_matrix

property GPCCA.**schur_matrix: Optional[numpy.ndarray]**
Ordered top left part of shape *(n_m, n_m)* of the real Schur matrix of P.

The ordered real partial Schur matrix R of P fulfills

$$PQ = QR$$

with the ordered matrix of dominant Schur vectors Q.

Return type Optional[ndarray]

pygpcca.GPCCA.schur_vectors

property GPCCA.**schur_vectors: Optional[numpy.ndarray]**
Array Q of shape *(n, n_m)* with *n_m* sorted Schur vectors in the columns.

The constant Schur vector is in the first column.

Return type Optional[ndarray]

pygpcca.GPCCA.stationary_probability

property GPCCA.**stationary_probability**
Stationary probability distribution π of the microstates.

Vector of shape *(n,)* which sums to 1.

pygpcca.GPCCA.top_eigenvalues

property GPCCA.**top_eigenvalues: Optional[numpy.ndarray]**
Top *m* respective *m_max* eigenvalues of *P*.

If a single integer *m* was given, the upper *m* eigenvalues are returned.

If a `tuple` or `dict` containing a minimum *m_min* and maximum number *m_max* of clusters was given, the upper *m_max* eigenvalues are returned.

Return type Optional[ndarray]

pygpcca.GPCCA.transition_matrix

property GPCCA.**transition_matrix: Union[numpy.ndarray, scipy.sparse._base.spmatrix]**

Row-stochastic transition matrix P.

Return type Union[ndarray, spmatrix]

CHAPTER

THREE

COARSE-GRAIN A SIMPLE TRANSITION MATRIX

To illustrate the application of *pyGPCCA* we will coarse-grain a simple irreducible transition matrix P as a toy example.

Firstly, we will import needed packages like `numpy`, `matplotlib` and of course `pygpcca`:

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pygpcca as gp
```

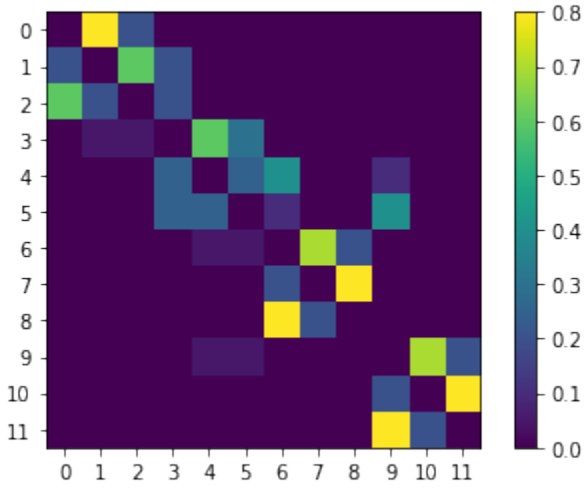
Next, we define a simple 12×12 row-stochastic (meaning that the rows of P each sum up to one) transition matrix P and plot it:

[24]:
```
P = np.array(
    [
    # 0.   1.   2.   3.   4.   5.   6.   7.   8.   9.   10.  11.
    [0.0, 0.8, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], #0
    [0.2, 0.0, 0.6, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], #1
    [0.6, 0.2, 0.0, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], #2

    [0.0, 0.05, 0.05, 0.0, 0.6, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], #3
    [0.0, 0.0, 0.0, 0.25, 0.0, 0.25, 0.4, 0.0, 0.0, 0.1, 0.0, 0.0], #4
    [0.0, 0.0, 0.0, 0.25, 0.25, 0.0, 0.1, 0.0, 0.0, 0.4, 0.0, 0.0], #5

    [0.0, 0.0, 0.0, 0.0, 0.05, 0.05, 0.0, 0.7, 0.2, 0.0, 0.0, 0.0], #6
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.2, 0.0, 0.8, 0.0, 0.0, 0.0], #7
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.8, 0.2, 0.0, 0.0, 0.0, 0.0], #8

    [0.0, 0.0, 0.0, 0.0, 0.05, 0.05, 0.0, 0.0, 0.0, 0.0, 0.7, 0.2], #9
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.2, 0.0, 0.8], #10
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.8, 0.2, 0.0], #11
    ],
    dtype=np.float64,
)

# plot the matrix P:
fig, ax = plt.subplots()
c = ax.imshow(P)
plt.xticks(np.arange(P.shape[1]))
plt.yticks(np.arange(P.shape[0]))
plt.ylim(-0.5, P.shape[0]-0.5)
ax.set_ylim(ax.get_ylim()[::-1])
```

(continues on next page)

(continued from previous page)

```
fig.colorbar(c)
plt.show()
```



Following this, we initialize a *GPCCA* object from the transition matrix P:

[3]:
```
gpcca = gp.GPCCA(P, z='LM', method='brandts')
```

GPCCA is a spectral clustering method with optimized memberships. It clusters the dominant m Schur vectors of a transition matrix. This algorithm generates a fuzzy clustering such that the resulting membership functions are as crisp (characteristic) as possible.

The parameter `z` specifies which portion of the eigenvalue spectrum of P is to be sought. The returned invariant subspace of P will be associated with this part of the spectrum.

In case of `z='LM'`, the eigenvalues with the largest magnitude will be sorted to the top in descending order. This way the dominant (top) eigenvalues will be located near the unit circle in the complex plane, unraveling not only stable or metastable states, but cyclic states that are associated with eigenvalues near the roots of unity as well.

In case of `z='LR'`, the eigenvalues with the largest real part will be sorted to the top in descending order. Thus stable and metastable states near the Perron root 1 are selected.

The parameter `method` specifies which method will be used to determine the invariant subspace.

If `method='brandts'`, a full Schur decomposition of P utilizing `scipy.linalg.schur` is performed and afterwards the returned Schur form R and Schur vector matrix Q are partially sorted to gain an orthonormal basis of the subspace associated with the m dominant eigenvalues of P. This is well tested and thus the default method, although it is also the slowest choice.

If `method='krylov'`, an orthonormal basis of the subspace associated with the m dominant eigenvalues of P is calculated using the Krylov-Schur method as implemented in SLEPc. This is the fastest choice and especially suitable for very large P, but it is still experimental.

Afterwards, we can get a list of $minChi$ values for numbers of macrostates m in an interval $[2, 12]$ of possible m ($m = 1$ is illegal here, since there is no point in clustering 12 microstates into one single macrostate). The $minChi$ values help us to determine an interval $[m_{min}, m_{max}]$ of nearly optimal numbers of macrostates for clustering:

[4]:
```
gpcca.minChi(2, 12)
```

[4]:
```
[-1.6653345369377348e-16,
 -2.255418698866725e-16,
```

(continues on next page)

(continued from previous page)

```
 -0.9923508699724691,
 -0.9972757370249341,
 -0.926802576904497,
 -0.2705117206956666,
 -0.3360447945215935,
 -0.2973036186306221,
 -0.29104047575515346,
 -0.42902208201892694,
 -3.5809019888001215e-16]
```

The $minChi$ criterion states that cluster numbers m (i.e. clustering into m clusters) with a $minChi$ value close to zero will potentially result in a optimal (meaning especially *crisp* or sharp) clustering. Obviously, only $m = 3$ qualifies as non-trivially potentially optimal, since for $m = 2$ and $m = 12$ always $minChi \approx 0$ trivially holds.

Now, we would optimize the clustering for numbers of macrostates m in a selected interval $[m_{min}, m_{max}]$ of potentially optimal macrostate numbers to find the optimal number of macrostates n_m resulting in the crispest clustering in the given interval.

Here, this interval would contain only $m = 2, 3$, since there is no benefit in clustering $n = 12$ data points into $m = n$ clusters.

Having said that, we here choose the whole interval $[2, 12]$ of legal cluster numbers to get a better impression of the spectrum associated with P later, when looking at the eigenvalues of P:

[5]:
```
gpcca.optimize({'m_min':2, 'm_max':12})
```

```
/home/breuter/g-pcca/pyGPCCA/pygpcca/_gpcca.py:999: UserWarning: Clustering into 4␣
↪clusters will split complex conjugate eigenvalues. Skipping clustering into 4 clusters.
  f"Clustering into {m} clusters will split complex conjugate eigenvalues. "
/home/breuter/g-pcca/pyGPCCA/pygpcca/_gpcca.py:999: UserWarning: Clustering into 6␣
↪clusters will split complex conjugate eigenvalues. Skipping clustering into 6 clusters.
  f"Clustering into {m} clusters will split complex conjugate eigenvalues. "
/home/breuter/g-pcca/pyGPCCA/pygpcca/_gpcca.py:999: UserWarning: Clustering into 9␣
↪clusters will split complex conjugate eigenvalues. Skipping clustering into 9 clusters.
  f"Clustering into {m} clusters will split complex conjugate eigenvalues. "
/home/breuter/g-pcca/pyGPCCA/pygpcca/_gpcca.py:1033: UserWarning: Clustering 12 data␣
↪points into 12 clusters is always perfectly crisp. Thus m=12 won't be included in the␣
↪search for the optimal cluster number.
  f"Clustering {n} data points into {max(m_list)} clusters is always perfectly crisp. "
```

[5]:
```
<pygpcca._gpcca.GPCCA at 0x7fe440c5f090>
```

The optimized *GPCCA* object is returned above and we can now access different properties of it.

Note: *pyGPCCA* warns that clustering P into 4, 6, and 9 clusters would split complex conjugate eigenvalues and thus skips the optimization for those cluster numbers. Further, *pyGPCCA* warns that Clustering 12 data points into 12 clusters is always perfectly crisp, i.e. $\xi \approx 1$. Thus m=12 won't be included in the search for the optimal cluster number, since it will always be selected to be optimal despite there is no benefit from clustering $n = 12$ data points into $m = n$ clusters.

The crispness values $\xi \in [0, 1]$ for numbers of macrostates m in the selected interval $[m_{min}, m_{max}]$ can be accessed via:

[6]:
```
gpcca.crispness_values
```

[6]:
```
array([0.74151472, 0.81512805, 0.        , 0.38587154, 0.        ,
       0.41628049, 0.41788963, 0.        , 0.55513151, 0.53758366,
       1.        ])
```

The crispness $\xi \in [0, 1]$ quantifies the optimality of a clustering (higher is better). It characterizes how crisp (sharp) the decomposition of the state space into m clusters is. Since $m = 4, 6, 9$ were skipped, the associated crispness values are assigned to zero, because no clustering was performed.

The optimal crispness for the optimal number of macrostates n_m in the selected interval $[m_{min}, m_{max}]$ can be accessed via:

[7]:
```
gpcca.optimal_crispness
```

[7]:
```
0.8151280474517894
```

The optimal number of macrostates n_m can be accessed via:

[8]:
```
gpcca.n_m
```

[8]:
```
3
```

The optimal number of clusters or macrostates n_m is the cluster number m with the maximum crispness.

A vector containing the top m_{max} eigenvalues of P can be accessed via (here we get the full sorted spectrum of P, since we chose $m_{max} = n$):

[10]:
```
gpcca.top_eigenvalues
```

[10]:
```
array([ 1.        +0.j        ,  0.96554293+0.j        ,
        0.88404279+0.j        , -0.48277146+0.48908574j,
       -0.48277146-0.48908574j, -0.49366905+0.47392788j,
       -0.49366905-0.47392788j,  0.58853656+0.j        ,
       -0.43198962+0.39030468j, -0.43198962-0.39030468j,
       -0.37126202+0.j        , -0.25      +0.j        ])
```

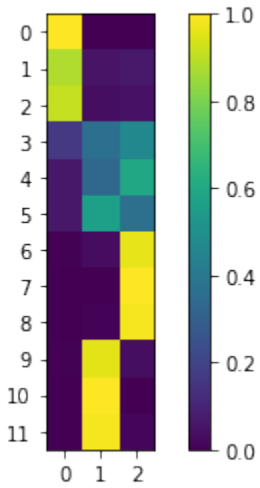
A vector containing the dominant n_m eigenvalues of P can be accessed via:

[11]:
```
gpcca.dominant_eigenvalues
```

[11]:
```
array([1.        +0.j, 0.96554293+0.j, 0.88404279+0.j])
```

An array χ containing the membership χ_{ij} (or probability) of each microstate i (to be assigned) to each cluster or macrostate j is available via:

[25]:
```
chi = gpcca.memberships
# plot chi:
fig, ax = plt.subplots()
c = ax.imshow(chi)
plt.xticks(np.arange(chi.shape[1]))
plt.yticks(np.arange(chi.shape[0]))
plt.ylim(-0.5, chi.shape[0]-0.5)
ax.set_ylim(ax.get_ylim()[::-1])
fig.colorbar(c)
plt.show()
# show chi matrix:
chi
```



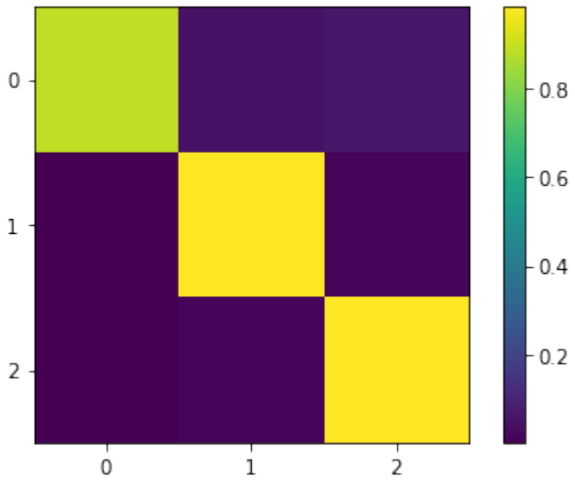
[25]:
```
array([[1.00000000e+00, 0.00000000e+00, 8.44548736e-18],
       [8.80009662e-01, 5.26338400e-02, 6.73564979e-02],
       [9.12351139e-01, 3.84758290e-02, 4.91730323e-02],
       [1.64953404e-01, 3.67067094e-01, 4.67979502e-01],
       [6.51588802e-02, 3.35927186e-01, 5.98913934e-01],
       [6.51588802e-02, 5.64708356e-01, 3.70132763e-01],
       [5.02066367e-03, 3.27724659e-02, 9.62206870e-01],
       [1.27462550e-16, 1.83492669e-16, 1.00000000e+00],
       [1.78878958e-03, 1.11025375e-02, 9.87108673e-01],
       [5.02066367e-03, 9.59750248e-01, 3.52290885e-02],
       [0.00000000e+00, 1.00000000e+00, 2.42354345e-17],
       [1.78878958e-03, 9.86286754e-01, 1.19244563e-02]])
```

The optimal coarse-grained transition matrix

$P_c = (\chi^T D \chi)^{-1} (\chi^T D P \chi)$

can be accessed via:

[26]:
```
P_c = gpcca.coarse_grained_transition_matrix
# plot P_c:
fig, ax = plt.subplots()
c = ax.imshow(P_c)
plt.xticks(np.arange(P_c.shape[1]))
plt.yticks(np.arange(P_c.shape[0]))
plt.ylim(-0.5, P_c.shape[0]-0.5)
ax.set_ylim(ax.get_ylim()[::-1])
fig.colorbar(c)
plt.show()
# show P_c matrix:
P_c
```



[26]:
```
array([[0.88647796, 0.04980224, 0.0637198 ],
       [0.00243516, 0.98097945, 0.01658538],
       [0.00243516, 0.01543652, 0.98212831]])
```

There are many more properties that can be accessed as you can see in the API documentation here, e.g.:

The input probability distribution of the microstates:

[14]:
```
gpcca.input_distribution
```

[14]:
```
array([0.08333333, 0.08333333, 0.08333333, 0.08333333, 0.08333333,
       0.08333333, 0.08333333, 0.08333333, 0.08333333, 0.08333333,
       0.08333333, 0.08333333])
```

The coarse grained input distribution of the macrostates:

[15]:
```
gpcca.coarse_grained_input_distribution
```

[15]:
```
array([0.25843757, 0.36239369, 0.37916873])
```

An integer vector containing the macrostate each microstate is located in (This employs an absolute crisp clustering, e.g. $\chi_{ij} \in \{0, 1\}$, and is recommended only for visualization purposes. You *cannot* compute any actual quantity of the coarse-grained kinetics without employing the fuzzy memberships, e.g. $\chi_{ij} \in [0, 1]$.):

[16]:
```
gpcca.macrostate_assignment
```

[16]:
```
array([0, 0, 0, 2, 2, 1, 2, 2, 2, 1, 1, 1])
```

A list were each element is an array that contains the indices of microstates assigned to the respective (first, second, third, ...) macrostate (This employs an absolute crisp clustering, e.g. $\chi_{ij} \in \{0, 1\}$, and is recommended only for visualization purposes. You *cannot* compute any actual quantity of the coarse-grained kinetics without employing the fuzzy memberships, e.g. $\chi_{ij} \in [0, 1]$.):

[17]:
```
gpcca.macrostate_sets
```

[17]:
```
[array([0, 1, 2]), array([ 5,  9, 10, 11]), array([3, 4, 6, 7, 8])]
```

The optimized rotation matrix, which rotates the dominant Schur vectors to yield the GPCCA memberships, i.e. $\chi = XA$:

[18]:
```
gpcca.rotation_matrix
```

[18]:
```
array([[ 0.25843757,  0.36239369,  0.37916873],
       [ 0.03366022, -0.36112468,  0.32746445],
       [-0.39024389,  0.164552  ,  0.22569189]])
```

Ordered top left part of the real Schur matrix R of P. The ordered partial real Schur matrix R of P fulfills $PQ = QR$ with the ordered matrix of dominant Schur vectors Q:

[20]:
```
gpcca.schur_matrix
```

[20]:
```
array([[ 1.00000000e+00,  6.38159449e-04, -7.04970784e-02],
       [ 0.00000000e+00,  9.65542930e-01,  7.02973961e-03],
       [ 0.00000000e+00,  0.00000000e+00,  8.84042793e-01]])
```

Array Q with the sorted Schur vectors in the columns (The constant Schur vector is in the first column):

[21]:
```
gpcca.schur_vectors
```

[21]:
```
array([[ 1.        ,  0.14326506, -1.88789655],
       [ 1.        ,  0.13739019, -1.58092803],
       [ 1.        ,  0.13889124, -1.66367359],
       [ 1.        ,  0.10015109,  0.24819166],
       [ 1.        ,  0.31120067,  0.52211907],
       [ 1.        , -0.34824151,  0.46523933],
       [ 1.        ,  1.25811096,  0.75789837],
       [ 1.        ,  1.35867697,  0.77943807],
       [ 1.        ,  1.32450074,  0.77190645],
       [ 1.        , -1.41382332,  0.52743247],
       [ 1.        , -1.52373762,  0.53081733],
       [ 1.        , -1.48638447,  0.52945543]])
```

Stationary probability distribution π of the microstates (a vector that sums to 1):

[22]:
```
gpcca.stationary_probability
```

[22]:
```
array([0.00356002, 0.00462803, 0.00439069, 0.01803744, 0.03463189,
       0.03030291, 0.16883048, 0.14873162, 0.15275139, 0.15584352,
       0.13729072, 0.14100128])
```

Coarse grained stationary distribution $\pi_c = \chi^T \pi$:

[23]:
```
gpcca.coarse_grained_stationary_probability
```

[23]:
```
array([0.02100054, 0.46893777, 0.51006168])
```

CHAPTER

FOUR

HOW TO CITE PYGPCCA

If you use *pyGPCCA* or parts of it to model molecular dynamics, e.g. to coarse-grain protein conformational dynamics, cite [Reuter18] as:

```
@article{Reuter18,
    author = {Reuter, Bernhard and Weber, Marcus and Fackeldey, Konstantin and Röblitz,␣
↪Susanna and Garcia, Martin E.},
    title = {Generalized Markov State Modeling Method for Nonequilibrium Biomolecular␣
↪Dynamics:
    Exemplified on Amyloid  Conformational Dynamics Driven by an Oscillating Electric␣
↪Field},
    journal = {Journal of Chemical Theory and Computation},
    volume = {14},
    number = {7},
    pages = {3579-3594},
    year = {2018},
    doi = {10.1021/acs.jctc.8b00079},
    note  = {PMID: 29812922},
}
```

If you use *pyGPCCA* or parts of it in a more general context, e.g. to model cellular dynamics, cite [Reuter19] as:

```
@article{Reuter19,
    author = {Reuter,Bernhard and Fackeldey,Konstantin and Weber,Marcus },
    title = {Generalized Markov modeling of nonreversible molecular kinetics},
    journal = {The Journal of Chemical Physics},
    volume = {150},
    number = {17},
    pages = {174103},
    year = {2019},
    doi = {10.1063/1.5064530},
}
```

Please also consider to cite the appropriate version of the *pyGPCCA* package as deposited on Zenodo [Reuter22].

CHAPTER

FIVE

ACKNOWLEDGMENTS

We thank Marcus Weber and the Computational Molecular Design (CMD) group at the Zuse Institute Berlin (ZIB) for the longstanding and productive collaboration in the field of Markov modeling of non-reversible molecular dynamics. M. Weber, together with Susanna Röblitz and K. Fackeldey, had the original idea to employ Schur vectors instead of eigenvectors in the coarse-graining of non-reversible transition matrices. Further, we would like to thank Fabian Paul for valuable discussions regarding the sorting of Schur vectors and his effort to translate the original sorting routine for real Schur forms, SRSchur published by Jan Brandts, from MATLAB into Python code, M. Weber and Alexander Sikorski for pointing us to SLEPc for sorted partial Schur decompositions, and A. Sikorski for supplying us with an code example and guidance how to interface SLEPc in Python. The development of *pyGPCCA* started - based on the original GPCCA program written in MATLAB - at the beginning of 2020 in a fork of MSMTools, since it was planned to integrate GPCCA into MSMTools at this time. Due to this, some similarities in structure and code (indicated were evident) can be found. Further, utility functions found in pygpcca/utils/_utils.py originate from MSMTools.

CHAPTER

SIX

RELEASE NOTES

6.1 Version 1.0

6.1.1 1.0.4 2022-10-31

Fixes

- Fix 'Operation done in wrong order' error when calling SLEPc #42.
- Minor pre-commit/linting fixes #39, #40, #41, #44, #45, #46.
- Fix intersphinx numpy/scipy #37.

Improvements

- Update and improve documentation and README #47.

6.1.2 1.0.3 2022-02-13

Fixes

- Fix CI, unpin some requirements, pin docs, enable doc linting #25, #26.
- Patch release preparation #35.

Improvements

- Print deviations, if a test is failing since a threshold is exceeded #29.
- Adjust too tight thresholds in some tests #30, #34.

6.1.3 1.0.2 2021-03-26

Bugfixes

- Fix not catching `ArpackError` when computing stationary distribution and mypy-related linting issues #21.

Improvements

- Use PETSc/SLEPc, if installed, to speed up the computation of the stationary distribution #22.

6.1.4 1.0.1 2021-02-01

General

- Minor improvements/fixes in README and acknowledgments.

6.1.5 1.0.0 2021-01-29

Initial release.

CHAPTER

SEVEN

REFERENCES

BIBLIOGRAPHY

[Reuter19] Bernhard Reuter, Konstantin Fackeldey, and Marcus Weber, *Generalized Markov modeling of nonreversible molecular kinetics*, The Journal of Chemical Physics, 150(17):174103, 2019. doi:10.1063/1.5064530.

[Reuter18] Bernhard Reuter, Marcus Weber, Konstantin Fackeldey, Susanna Röblitz, and Martin E. Garcia, *Generalized Markov State Modeling Method for Nonequilibrium Biomolecular Dynamics: Exemplified on Amyloid Conformational Dynamics Driven by an Oscillating Electric Field.*, Journal of Chemical Theory and Computation, 14(7):3579–3594, 2018. doi:10.1021/acs.jctc.8b00079.

[Brandts02] Jan H. Brandts, *Matlab code for sorting real Schur forms*, Numerical Linear Algebra with Applications, 9(3):249-261, 2002. doi:10.1002/nla.274.

[Roeblitz13] Susanna Röblitz and Marcus Weber, *Fuzzy spectral clustering by PCCA+: application to Markov state models and data classification.*, Advances in Data Analysis and Classification, 7:147-179, 2013. doi:10.1007/s11634-013-0134-6.

INDEX