PyGenStability

Alexis Arnaudon, Dominik Schindler

CONTENTS

This `python` package is designed for multiscale community detection with Markov Stability (MS) analysis [1, 2] and allows researchers to identify robust network partitions at different resolutions. It implements several variants of the MS cost functions that are based on graph diffusion processes to explore the network (see illustration below). Whilst primarily built for MS, the internal architecture of *PyGenStability* has been designed to solve for a wide range of clustering cost functions since it is based on optimising the so-called generalized Markov Stability function [3]. To maximize the generalized Markov Stability cost function, *PyGenStability* provides a convenient `python` interface for C++ implementations of Louvain [4] and Leiden [5] algorithms. We further provide specific analysis tools to process and analyse the results from multiscale community detection, and to facilitate the automatic selection of robust partitions [6]. *PyGenStability* is accompanied by a software paper that further details the implementation, result analysis, benchmarks and applications [7].
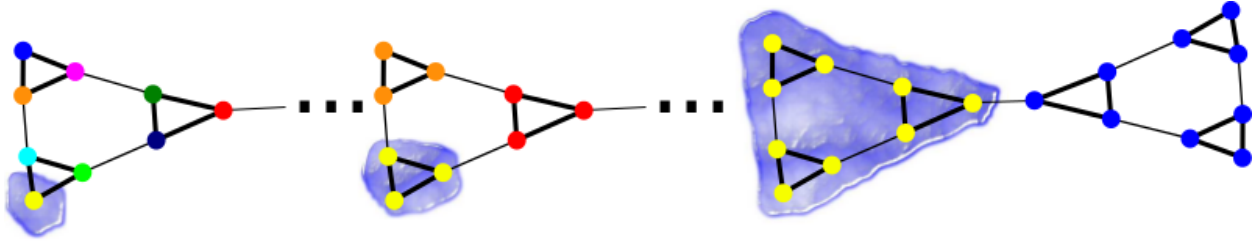
INSTALLATION

The wrapper uses Pybind11 https://github.com/pybind/pybind11 and the package can simply be installed by first cloning this repo with

```
git clone --recurse-submodules https://github.com/ImperialCollegeLondon/PyGenStability.
→git
```

(if the `--recurse-submodules` has not been used, just do `git submodule update --init --recursive` to fetch the submodule with M. Schaub's code).

Then, to install the package, simply run

```
pip install .
```

using a fresh `virtualenv` in python3 may be recommended to avoid conflict of python packages.

To use plotly for interactive plots in the browser, install this package with

```
pip install .[plotly]
```

To use a contrib module, with additional tools, run

```
pip install .[contrib]
```

To install all dependencies, run

```
pip install .[all]
```

USING THE CODE

The code is simple to run with the default settings. We can input our graph (of type scipy.csgraph), run a scan in scales with a chosen Markov Stability constructor and plot the results in a summary figure presenting different partition quality measures across scales (values of MS cost function, number of communities, etc.) with an indication of optimal scales.

```python
import pygenstability as pgs
results = pgs.run(graph)
pgs.plot_scan(results)
```

Although it is enforced in the code, it is advised to set environment variables

```
export OPENBLAS_NUM_THREADS=1
export OMP_NUM_THREADS=1
export NUMEXPR_MAX_THREADS=1
```

to ensure numpy does not use multi-threadings, which may clash with the parallelisation and slow down the computation.

There are a variety of further choices that users can make that will impact the partitioning, including:

- Constructor: Generalized Markov Stability requires the user to input a quality matrix and associated null models. We provide an object-oriented module to write user-defined constructors for these objects, with several already implemented (see `pygenstability/constructors.py` for some classic examples).

- Generalized Markov Stability maximizers: To maximize the NP-hard optimal generalized Markov Stability we interface with two algorithms: (i) Louvain and (ii) Leiden.

While Louvain is defined as the default due to its familiarity within the research community, Leiden is known to produce better partitions and can be used by specifying the run function.

```python
results = pgs.run(graph, method="leiden")
```

There are also additional post-processing and analysis functions, including:

- Plotting via matplotlib and plotly (interactive).

- Automated optimal scale selection.

Optimal scale selection [6] is performed by default with the run function but can be repeated with different parameters if needed, see `pygenstability/optimal_scales.py`. To reduce noise, e.g., one can increase the parameter values for `block_size` and `window_size`. The optimal network partitions can then be plotted given a NetworkX nx_graph.

```python
results = pgs.identify_optimal_scales(results, block_size=10, window_size=5)
pgs.plot_optimal_partitions(nx_graph, results)
```

CONSTRUCTORS

We provide an object-oriented module for constructing quality matrices and null models in `pygenstability/constructors.py`. Various constructors are implemented for different types of graphs:

- `linearized` based on linearized MS for large undirected weighted graphs [2]

- `continuous_combinatorial` based on combinatorial Laplacian for undirected weighted graphs [2]

- `continuous_normalized` based on random-walk normalized Laplacians for undirected weighted graphs [2]

- `signed_modularity` based on signed modularity for large signed graphs [8]

- `signed_combinatorial` based on signed combinatorial Laplacian for signed graphs [3]

- `directed` based on random-walk Laplacian with teleportation for directed weighted graphs [2]

- `linearized_directed` based on random-walk Laplacian with teleportation for large directed weighted graphs

For the computationally efficient analysis of **large** graphs, we recommend using the `linearized`, `linearized_directed` or `signed_modularity` constructors instead of `continuous_combinatorial`, `continuous_normalized`, `directed` or `signed_combinatorial` that rely on the computation of matrix exponentials.

For those of you that wish to implement their own constructor, you will need to design a function with the following properties:

- take a scipy.csgraph `graph` and a float `time` as argument

- return a `quality_matrix` (sparse scipy matrix) and a `null_model` (multiples of two, in a numpy array)

CONTRIBUTORS

- Alexis Arnaudon, GitHub: `arnaudon` <`https://github.com/arnaudon`>

- Robert Peach, GitHub: `peach-lucien` <`https://github.com/peach-lucien`>

- Dominik Schindler, GitHub: `d-schindler` <`https://github.com/d-schindler`>

We always look out for individuals that are interested in contributing to this open-source project. Even if you are just using *PyGenStability* and made some minor updates, we would be interested in your input.

CITE

Please cite our paper if you use this code in your own work:

```
@article{pygenstability,
  author = {Arnaudon, Alexis and Schindler, Dominik J. and Peach, Robert L. and
→Gosztolai, Adam and Hodges, Maxwell and Schaub, Michael T. and Barahona, Mauricio},
  title = {PyGenStability: Multiscale community detection with generalized Markov
→Stability},
  publisher = {arXiv},
  year = {2023},
  doi = {10.48550/ARXIV.2303.05385},
  url = {https://arxiv.org/abs/2303.05385}
}
```

The original paper for Markov Stability can also be cited as:

```
@article{delvenne2010stability,
  title={Stability of graph communities across time scales},
  author={Delvenne, J-C and Yaliraki, Sophia N and Barahona, Mauricio},
  journal={Proceedings of the National Academy of Sciences},
  volume={107},
  number={29},
  pages={12755--12760},
  year={2010},
  publisher={National Acad Sciences}
}
```

RUN EXAMPLE

In the `example` folder, a demo script with a stochastic block model can be tried with

```
python simple_example.py
```

or using the click app:

```
./run_simple_example.sh
```

Other examples can be found as jupyter-notebooks in the `examples/` directory, including:

- Example 1: Undirected SBM
- Example 2: Multiscale SBM
- Example 3: Directed networks
- Example 4: Custom constructors
- Example 5: Hypergraphs
- Example 6: Signed networks

Finally, we provide applications to real-world networks in the `examples/real_examples/` directory, including:

- Power grid network
- Protein structures

OUR OTHER AVAILABLE PACKAGES

If you are interested in trying our other packages, see the below list:

- GDR : Graph diffusion reclassification. A methodology for node classification using graph semi-supervised learning.

- hcga : Highly comparative graph analysis. A graph analysis toolbox that performs massive feature extraction from a set of graphs, and applies supervised classification methods.

- MSC : MultiScale Centrality: A scale-dependent metric of node centrality.

- DynGDim : Dynamic Graph Dimension: Computing the relative, local and global dimension of complex networks.

- RMST : Relaxed Minimum Spanning Tree: Computing the relaxed minimum spanning tree to sparsify networks whilst retaining dynamic structure.

- StEP : Spatial-temporal Epidemiological Proximity: Characterising contact in disease outbreaks via a network model of spatial-temporal proximity.

REFERENCES

[1] J.-C. Delvenne, S. N. Yaliraki, and M. Barahona, 'Stability of graph communities across time scales', *Proceedings of the National Academy of Sciences*, vol. 107, no. 29, pp. 12755–12760, Jul. 2010, doi: 10.1073/pnas.0903215107.

[2] R. Lambiotte, J.-C. Delvenne, and M. Barahona, 'Random Walks, Markov Processes and the Multiscale Modular Organization of Complex Networks', *IEEE Trans. Netw. Sci. Eng.*, vol. 1, no. 2, pp. 76–90, Jul. 2014, doi: 10.1109/TNSE.2015.2391998.

[3] M. T. Schaub, J.-C. Delvenne, R. Lambiotte, and M. Barahona, 'Multiscale dynamical embeddings of complex networks', *Phys. Rev. E*, vol. 99, no. 6, Jun. 2019, doi: 10.1103/PhysRevE.99.062308.

[4] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, 'Fast unfolding of communities in large networks', *J. Stat. Mech.*, vol. 2008, no. 10, Oct. 2008, doi: 10.1088/1742-5468/2008/10/p10008.

[5] V. A. Traag, L. Waltman, and N. J. van Eck, 'From Louvain to Leiden: guaranteeing well-connected communities', *Sci Rep*, vol. 9, no. 1, p. 5233, Mar. 2019, doi: 10.1038/s41598-019-41695-z.

[6] D. J. Schindler, J. Clarke, and M. Barahona, 'Multiscale Mobility Patterns and the Restriction of Human Movement', *Royal Society Open Science*, vol. 10, no. 10, p. 230405, Oct. 2023, doi: 10.1098/rsos.230405.

[7] A. Arnaudon, D. J. Schindler, R. L. Peach, A. Gosztolai, M. Hodges, M. T. Schaub, and M. Barahona, 'PyGenStability: Multiscale community detection with generalized Markov Stability', *arXiv pre-print*, Mar. 2023, doi: 10.48550/arXiv.2303.05385.

[8] S. Gómez, P. Jensen, and A. Arenas, 'Analysis of community structure in networks of correlated data'. *Physical Review E*, vol. 80, no. 1, p. 016114, Jul. 2009, doi: 10.1103/PhysRevE.80.016114.

LICENCE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

9.1 API documentation

Documentation of the API of *PyGenStability*.

9.1.1 PyGenStability module

PyGenStability code to solve generalized Markov Stability including Markov stability.

The generalized Markov Stability is of the form

$$Q_{gen}(t, H) = \text{Tr}\left[H^T\left(F(t) - \sum_{k=1}^{m} v_{2k-1}v_{2k}^T\right)H\right]$$

where $F(t)$ is the quality matrix and v_k are null model vectors. The choice of the quality matrix and null model vectors are arbitrary in the generalized Markov Stability setting, and can be parametrised via built-in constructors, or specified by the user via the constructor module.

pygenstability.pygenstability.**run**(*graph=None*, *constructor='linearized'*, *min_scale=-2.0*, *max_scale=0.5*, *n_scale=20*, *log_scale=True*, *scales=None*, *n_tries=100*, *with_NVI=True*, *n_NVI=20*, *with_postprocessing=True*, *with_ttprime=True*, *with_spectral_gap=False*, *exp_comp_mode='spectral'*, *result_file='results.pkl'*, *n_workers=4*, *tqdm_disable=False*, *with_optimal_scales=True*, *optimal_scales_kwargs=None*, *method='louvain'*, *constructor_kwargs=None*)

This is the main function to compute graph clustering across scales with Markov Stability.

This function needs a graph object as an adjacency matrix encoded with scipy.csgraph. The default settings will provide a fast and generic run with linearized Markov Stability, which corresponds to modularity with a scale parameter. Other built-in constructors are available to perform Markov Stability with matrix exponential

computations. Custom constructors can be added via the constructor module. Additional parameters can be used to set the range and number of scales, number of trials for generalized Markov Stability optimisation, with Louvain or Leiden algorithm.

> **Parameters**
>
> - **graph** (`scipy.csgraph`) – graph to cluster, if None, the constructor cannot be a str
> - **constructor** (`str/function`) – name of the generalized Markov Stability constructor, or custom constructor function. It must have two arguments, graph and scale.
> - **min_scale** (`float`) – minimum Markov scale
> - **max_scale** (`float`) – maximum Markov scale
> - **n_scale** (`int`) – number of scale steps
> - **log_scale** (`bool`) – use linear or log scales for scales
> - **scales** (`array`) – custom scale vector, if provided, it will override the other scale arguments
> - **n_tries** (`int`) – number of generalized Markov Stability optimisation evaluations
> - **with_NVI** (`bool`) – compute NVI(t) between generalized Markov Stability optimisations at each scale t
> - **n_NVI** (`int`) – number of randomly chosen generalized Markov Stability optimisations to estimate NVI
> - **with_postprocessing** (`bool`) – apply the final postprocessing step
> - **with_ttprime** (`bool`) – compute the NVI(t,tprime) matrix to compare scales t and tprime
> - **with_spectral_gap** (`bool`) – normalise scale by spectral gap
> - **exp_comp_mode** (`str`) – mode to compute matrix exponential, can be expm or spectral
> - **result_file** (`str`) – path to the result file
> - **n_workers** (`int`) – number of workers for multiprocessing
> - **tqdm_disable** (`bool`) – disable progress bars
> - **with_optimal_scales** (`bool`) – apply optimal scale selection algorithm
> - **optimal_scales_kwargs** (`dict`) – kwargs to pass to optimal scale selection, see optimal_scale module.
> - **method** (`str`) – optimiation method, louvain or leiden
> - **constructor_kwargs** (`dict`) – additional kwargs to pass to constructor prepare method
>
> **Returns**
>
> > **Results dict with the following entries**
> >
> > - 'run_params': dict with parameters used to run the code
> > - 'scales': scales of the scan
> > - 'number_of_communities': number of communities at each scale
> > - 'community_id': community node labels at each scale
> > - 'NVI': NVI at each scale
> > - 'ttprime': ttprime matrix

pygenstability.pygenstability.**evaluate_NVI**(*index_pair*, *partitions*)

Evaluations of Normalized Variation of Information (NVI).

NVI is defined for two partitions $p1$ and $p2$ as:

$$NVI = \frac{E(p1) + E(p2) - 2MI(p1, p2)}{JE(p1, p2)}$$

where E is the entropy, JE the joint entropy and MI the mutual information.

> **Parameters**
>
> - **index_pair** (*list*) – list of two indices to select pairs of partitions
>
> - **partitions** (*list*) – list of partitions
>
> **Returns**
> float, Normalized Variation Information

9.1.2 Constructors module

Module to create constructors of quality matrix and null models.

The generalized Markov Stability is given as

$$Q_{gen}(t, H) = \text{Tr}\left[H^T\left(F(t) - \sum_{k=1}^{m} v_{2k-1}v_{2k}^T\right)H\right]$$

where $F(t)$ is the quality matrix and v_k are null model vectors.

In the following we denote by A the adjacency matrix of a graph with N nodes and M edges. The out-degree of the graph is given by $d = A\mathbf{1}$, where $\mathbf{1}$ is the vector of ones, and we denote the diagonal degree matrix by $D = \text{diag}(d)$.

pygenstability.constructors.**load_constructor**(*constructor*, *graph*, ***kwargs*)

Load a constructor from its name, or as a custom Constructor class.

class pygenstability.constructors.**Constructor**(*graph*, *with_spectral_gap=False*, *exp_comp_mode='spectral'*, ***kwargs*)

Parent class for generalized Markov Stability constructor.

This class encodes generalized Markov Stability through the quality matrix and null models. Use the method prepare to load and compute scale independent quantities, and the method get_data to return quality matrix, null model, and possible global shift.

The constructor calls the prepare method upon initialisation.

> **Parameters**
>
> - **graph** (*csgraph*) – graph for which to run clustering
>
> - **with_spectral_gap** (*bool*) – set to True to use spectral gap to rescale
>
> - **kwargs** (*dict*) – any other properties to pass to the constructor.
>
> - **exp_comp_mode** (*str*) – mode to compute matrix exponential, can be expm or spectral

prepare(***kwargs*)

Prepare the constructor with non-scale dependent computations.

> **get_data**(*scale*)
>
>> Return quality and null model at given scale as well as global shift (or None).
>>
>> User has to define the _get_data so we can enure numpy does not use multiple threads

class pygenstability.constructors.**constructor_linearized**(*graph*, *with_spectral_gap=False*,
exp_comp_mode='spectral', ***kwargs*)

> Constructor for continuous linearized Markov Stability.
>
> The quality matrix is:
>
> $$F(t) = t\frac{A}{2M},$$
>
> and the associated null model is $v_1 = v_2 = \frac{d}{2M}$.
>
> The constructor calls the prepare method upon initialisation.
>
>> **Parameters**
>>
>>> - **graph** (*csgraph*) – graph for which to run clustering
>>>
>>> - **with_spectral_gap** (*bool*) – set to True to use spectral gap to rescale
>>>
>>> - **kwargs** (*dict*) – any other properties to pass to the constructor.
>>>
>>> - **exp_comp_mode** (*str*) – mode to compute matrix exponential, can be expm or spectral
>
>> **prepare**(***kwargs*)
>>
>>> Prepare the constructor with non-scale dependent computations.

class pygenstability.constructors.**constructor_continuous_combinatorial**(*graph*,
with_spectral_gap=False,
exp_comp_mode='spectral',
**kwargs)

> Constructor for continuous combinatorial Markov Stability.
>
> The quality matrix is:
>
> $$F(t) = \Pi \exp(-Lt)$$
>
> where $L = D - A$ is the combinatorial Laplacian and $\Pi = \mathrm{diag}(\pi)$, with null model $v_1 = v_2 = \pi = \frac{1}{N}$.
>
> The constructor calls the prepare method upon initialisation.
>
>> **Parameters**
>>
>>> - **graph** (*csgraph*) – graph for which to run clustering
>>>
>>> - **with_spectral_gap** (*bool*) – set to True to use spectral gap to rescale
>>>
>>> - **kwargs** (*dict*) – any other properties to pass to the constructor.
>>>
>>> - **exp_comp_mode** (*str*) – mode to compute matrix exponential, can be expm or spectral
>
>> **prepare**(***kwargs*)
>>
>>> Prepare the constructor with non-scale dependent computations.

class pygenstability.constructors.**constructor_continuous_normalized**(*graph*,
with_spectral_gap=False,
exp_comp_mode='spectral',
**kwargs)

Constructor for continuous normalized Markov Stability.

The quality matrix is:

$$F(t) = \Pi \exp(-Lt)$$

where $L = D^{-1}(D - A)$ is the random-walk normalized Laplacian and $\Pi = \mathrm{diag}(\pi)$ with null model $v_1 = v_2 = \pi = \frac{d}{2M}$.

The constructor calls the prepare method upon initialisation.

> **Parameters**
>
> - **graph** (*csgraph*) – graph for which to run clustering
> - **with_spectral_gap** (*bool*) – set to True to use spectral gap to rescale
> - **kwargs** (*dict*) – any other properties to pass to the constructor.
> - **exp_comp_mode** (*str*) – mode to compute matrix exponential, can be expm or spectral

prepare(***kwargs*)

> Prepare the constructor with non-scale dependent computations.

class pygenstability.constructors.**constructor_signed_modularity**(*graph*, *with_spectral_gap=False*, *exp_comp_mode='spectral'*, ***kwargs*)

Constructor of signed modularity.

This implementation is equation (18) of[1], where quality is the adjacency matrix and the null model is the difference between the standard modularity null models based on positive and negative degree vectors.

References

The constructor calls the prepare method upon initialisation.

> **Parameters**
>
> - **graph** (*csgraph*) – graph for which to run clustering
> - **with_spectral_gap** (*bool*) – set to True to use spectral gap to rescale
> - **kwargs** (*dict*) – any other properties to pass to the constructor.
> - **exp_comp_mode** (*str*) – mode to compute matrix exponential, can be expm or spectral

prepare(***kwargs*)

> Prepare the constructor with non-scale dependent computations.

class pygenstability.constructors.**constructor_signed_combinatorial**(*graph*, *with_spectral_gap=False*, *exp_comp_mode='spectral'*, ***kwargs*)

Constructor for continuous signed combinatorial Markov Stability.

The quality matrix is:

$$F(t) = \exp(-Lt)^T \exp(-Lt)$$

---

[1] Gómez, S., Jensen, P., & Arenas, A. (2009). Analysis of community structure in networks of correlated data. Physical Review E, 80(1), 016114.

where $L = D_{\text{abs}} - A$ is the signed combinatorial Laplacian, $D_{\text{abs}} = \text{diag}(d_{\text{abs}})$ the diagonal matrix of absolute node strengths d_{abs}, and the associated null model is given by $v_1 = v_2 = \mathbf{0}$, where $\mathbf{0}$ is the vector of zeros.

The constructor calls the prepare method upon initialisation.

> **Parameters**
>> - **graph** (`csgraph`) – graph for which to run clustering
>> - **with_spectral_gap** (`bool`) – set to True to use spectral gap to rescale
>> - **kwargs** (`dict`) – any other properties to pass to the constructor.
>> - **exp_comp_mode** (`str`) – mode to compute matrix exponential, can be expm or spectral

prepare(***kwargs*)

> Prepare the constructor with non-scale dependent computations.

get_data(*scale*)

> Return quality and null model at given scale.

class pygenstability.constructors.**constructor_directed**(*graph*, *with_spectral_gap=False*, *exp_comp_mode='spectral'*, ***kwargs*)

Constructor for directed Markov stability.

The quality matrix is:

$$F(t) = \Pi \exp\left(t\left(M(\alpha) - I\right)\right)$$

where I denotes the identity matrix, $M(\alpha)$ is the transition matrix of a random walk with teleportation and damping factor $0 \le \alpha < 1$, and $\Pi = \text{diag}(\pi)$ for the associated null model $v_1 = v_2 = \pi$ given by the eigenvector solving $\pi M(\alpha) = \pi$, which is related to PageRank.

The transition matrix $M(\alpha)$ is given by

$$M(\alpha) = \alpha D^{-1} A + \left((1-\alpha)I + \alpha \text{diag}(a)\right) \frac{\mathbf{1}\mathbf{1}^T}{N},$$

where D denotes the diagonal matrix of out-degrees with $D_{ii} = 1$ if the out-degree $d_i = 0$ and a denotes the vector of dangling nodes, i.e. $a_i = 1$ if the out-degree $d_i = 0$ and $a_i = 0$ otherwise.

The constructor calls the prepare method upon initialisation.

> **Parameters**
>> - **graph** (`csgraph`) – graph for which to run clustering
>> - **with_spectral_gap** (`bool`) – set to True to use spectral gap to rescale
>> - **kwargs** (`dict`) – any other properties to pass to the constructor.
>> - **exp_comp_mode** (`str`) – mode to compute matrix exponential, can be expm or spectral

prepare(***kwargs*)

> Prepare the constructor with non-scale dependent computations.

class pygenstability.constructors.**constructor_linearized_directed**(*graph*, *with_spectral_gap=False*, *exp_comp_mode='spectral'*, ***kwargs*)

Constructor for linearized directed Markov stability.

The quality matrix is:

$$F(t) = \Pi t M(\alpha)$$

where $M(\alpha)$ is the transition matrix of a random walk with teleportation and damping factor $0 \leq \alpha < 1$, and $\Pi = \mathrm{diag}(\pi)$ for the associated null model $v_1 = v_2 = \pi$ given by the eigenvector solving $\pi M(\alpha) = \pi$, which is related to PageRank.

The transition matrix $M(\alpha)$ is given by

$$M(\alpha) = \alpha D^{-1} A + ((1-\alpha)I + \alpha \mathrm{diag}(a)) \frac{\mathbf{1}\mathbf{1}^T}{N},$$

where I denotes the identity matrix, D denotes the diagonal matrix of out-degrees with $D_{ii} = 1$ if the out-degree $d_i = 0$ and a denotes the vector of dangling nodes, i.e. $a_i = 1$ if the out-degree $d_i = 0$ and $a_i = 0$ otherwise.

The constructor calls the prepare method upon initialisation.

> **Parameters**
>
> - **graph** (`csgraph`) – graph for which to run clustering
>
> - **with_spectral_gap** (`bool`) – set to True to use spectral gap to rescale
>
> - **kwargs** (`dict`) – any other properties to pass to the constructor.
>
> - **exp_comp_mode** (`str`) – mode to compute matrix exponential, can be expm or spectral

prepare(***kwargs*)

> Prepare the constructor with non-scale dependent computations.

9.1.3 The pygenstability cli

Command line interface.

cli

App initialisation.

```
cli [OPTIONS] COMMAND [ARGS]...
```

plot_communities

Plot communities on networkx graph.

```
cli plot_communities [OPTIONS] GRAPH_FILE RESULTS_FILE
```

Arguments

`GRAPH_FILE`
> Required argument

`RESULTS_FILE`
> Required argument

plot_scan

Plot results in scan plot.

```
cli plot_scan [OPTIONS] RESULTS_FILE
```

Arguments

`RESULTS_FILE`
> Required argument

run

Run pygenstability.

graph_file: path to either a .pkl with adjacency matrix in sparse format, or a text file with three columns encoding node indices and edge weight. The columns need a header, or the first line will be dropped. Notice that doubled edges with opposite orientations are needed for symetric graph.

See https://barahona-research-group.github.io/PyGenStability/ for more information.

```
cli run [OPTIONS] GRAPH_FILE
```

Options

`--constructor` <constructor>
> Name of the quality constructor.
>
> > **Default**
> > > linearized

`--min-scale` <min_scale>
> Minimum scale.
>
> > **Default**
> > > -2.0

`--max-scale` <max_scale>
> Maximum scale.
>
> > **Default**
> > > 0.5

--n-scale <n_scale>

> Number of scale steps.

> > **Default**
> > 20

--log-scale <log_scale>

> Use linear or log scales.

> > **Default**
> > True

--n-tries <n_tries>

> Number of Louvain evaluations.

> > **Default**
> > 100

--NVI, --no-NVI

> Compute the normalized variation of information between runs.

> > **Default**
> > True

--n-NVI <n_nvi>

> Number of randomly chosen runs to estimate the NVI.

> > **Default**
> > 20

--postprocessing, --no-postprocessing

> Apply the final postprocessing step.

> > **Default**
> > True

--ttprime, --no-ttprime

> Compute the ttprime matrix.

> > **Default**
> > True

--spectral-gap, --no-spectral-gap

> Normalize scale by spectral gap.

> > **Default**
> > True

--result-file <result_file>

> Path to the result file.

> > **Default**
> > results.pkl

--n-workers <n_workers>

> Number of workers for multiprocessing.

> > **Default**
> > 4

`--tqdm-disable` `<tqdm_disable>`

>   disable progress bars

>>   **Default**
>>>   False

`--method` `<method>`

>   Method to solve modularity, either Louvain or Leiden

>>   **Default**
>>>   louvain

`--with-optimal-scales`, `--no-with-optimal-scales`

>   Search for optimal scales

>>   **Default**
>>>   True

`--exp-comp-mode` `<exp_comp_mode>`

>   Method to compute matrix exponential, can be 'spectral' or 'expm'

>>   **Default**
>>>   spectral

Arguments

`GRAPH_FILE`

>   Required argument

9.1.4 Plotting module

Plotting functions.

pygenstability.plotting.**plot_scan**(*all_results*, *scale_axis=True*, *figure_name='scan_results.pdf'*, *use_plotly=False*, *live=True*, *plotly_filename='scan_results.html'*)

>   Plot results of pygenstability with matplotlib or plotly.

>   **Parameters**

>>   • **all_results** (*dict*) – results of pygenstability scan

>>   • **scale_axis** (*bool*) – display scale of scale index on scale axis

>>   • **figure_name** (*str*) – name of matplotlib figure

>>   • **use_plotly** (*bool*) – use matplotlib or plotly backend

>>   • **live** (*bool*) – for plotly backend, open browser with pot

>>   • **plotly_filename** (*str*) – filename of .html figure from plotly

pygenstability.plotting.**plot_scan_plotly**(*all_results*, *live=False*, *filename='clusters.html'*)

>   Plot results of pygenstability with plotly.

pygenstability.plotting.**plot_single_partition**(*graph*, *all_results*, *scale_id*, *edge_color='0.5'*, *edge_width=0.5*, *node_size=100*)

>   Plot the community structures for a given scale.

>   **Parameters**

- **graph** (`networkx.Graph`) – graph to plot
- **all_results** (`dict`) – results of pygenstability scan
- **scale_id** (`int`) – index of scale to plot
- **folder** (`str`) – folder to save figures
- **edge_color** (`str`) – color of edges
- **edge_width** (`float`) – width of edges
- **node_size** (`float`) – size of nodes
- **ext** (`str`) – extension of figures files

pygenstability.plotting.**plot_optimal_partitions**(*graph*, *all_results*, *edge_color='0.5'*, *edge_width=0.5*, *folder='optimal_partitions'*, *ext='.pdf'*, *show=False*)

Plot the community structures at each optimal scale.

Parameters

- **graph** (`networkx.Graph`) – graph to plot
- **all_results** (`dict`) – results of pygenstability scan
- **edge_color** (`str`) – color of edges
- **edge_width** (`float`) – width of edgs
- **folder** (`str`) – folder to save figures
- **ext** (`str`) – extension of figures files
- **show** (`bool`) – show each plot with plt.show() or not

pygenstability.plotting.**plot_communities**(*graph*, *all_results*, *folder='communities'*, *edge_color='0.5'*, *edge_width=0.5*, *ext='.pdf'*)

Plot the community structures at each scale in a folder.

Parameters

- **graph** (`networkx.Graph`) – graph to plot
- **all_results** (`dict`) – results of pygenstability scan
- **folder** (`str`) – folder to save figures
- **edge_color** (`str`) – color of edges
- **edge_width** (`float`) – width of edgs
- **ext** (`str`) – extension of figures files

pygenstability.plotting.**plot_communities_matrix**(*graph*, *all_results*, *folder='communities_matrix'*, *ext='.pdf'*)

Plot communities at all scales in matrix form.

Parameters

- **graph** (`array`) – as a numpy matrix
- **all_results** (`dict`) – clustring results
- **folder** (`str`) – folder to save figures
- **ext** (`str`) – figure file format

pygenstability.plotting.**plot_scan_plt**(*all_results*, *scale_axis=True*, *figure_name='scan_results.svg'*)

 Plot results of pygenstability with matplotlib.

pygenstability.plotting.**plot_clustered_adjacency**(*adjacency*, *all_results*, *scale*, *labels=None*,
                 figsize=(12, 10), *cmap='Blues'*,
                 figure_name='clustered_adjacency.pdf')

 Plot the clustered adjacency matrix of the graph at a given scale.

> **Parameters**
>
> - **adjacency** (`ndarray`) – adjacency matrix to plot
> - **all_results** (`dict`) – results of PyGenStability
> - **scale** (`int`) – scale index for clustering
> - **labels** (`list`) – node labels, or None
> - **figsize** (`tubple`) – figure size
> - **cmap** (`str`) – colormap for matrix elements
> - **figure_name** (`str`) – filename of the figure with extension

9.1.5 Optimal scales module

Detect optimal scales from a scale scan.

pygenstability.optimal_scales.**identify_optimal_scales**(*results*, *kernel_size=3*, *window_size=3*,
                     max_nvi=1, *basin_radius=1*)

 Identifies optimal scales in Markov Stability[1].

 Robust scales are found in a sequential way. We first search for large diagonal blocks of low values in the NVI(t, t') matrix that are located at local minima of its pooled diagonal, called block detection curve, and we obtain basins of fixed radius around these local minima. We then determine the minima of the NVI(t) curve for each basin, and these minima correspond to the robust partitions of the network.

> **Parameters**
>
> - **results** (`dict`) – the results from a Markov Stability calculation
> - **kernel_size** (`int`) – size of kernel for average-pooling of the NVI(t,t') matrix
> - **window_size** (`int`) – size of window for moving mean, to smooth the pooled diagonal
> - **max_nvi** (`float`) – threshold for local minima of the pooled diagonal
> - **basin_radius** (`int`) – radius of basin around local minima of the pooled diagonal
>
> **Returns**
> 'selected_partitions' and 'block_detection_curve'
>
> **Return type**
> result dictionary with two new keys

---

[1] D. J. Schindler, J. Clarke, and M. Barahona, 'Multiscale Mobility Patterns and the Restriction of Human Movement', *arXiv:2201.06323*, 2023

PyGenStability

References

9.1.6 I/O module

I/O functions.

pygenstability.io.**save_results**(*all_results, filename='results.pkl'*)

> Save results in a pickle.

pygenstability.io.**load_results**(*filename='results.pkl'*)

> Load results from a pickle.

9.1.7 Example: Markov Stability with PyGenStability

This example illustrates how to use PyGenStability for multiscale community detection with Markov Stability analysis.

```
[1]:  import matplotlib.pyplot as plt
      import networkx as nx
      import scipy.sparse as sp

      import pygenstability as pgs
      from pygenstability import plotting
      from pygenstability.pygenstability import evaluate_NVI

      from multiscale_example import create_graph
```

We first create a stochastic block model graph with some planted partitions at different scales.

```
[2]:  A, coarse_scale_id, middle_scale_id, fine_scale_id = create_graph()

      # Create nx graph
      G = nx.from_numpy_array(A)

      # Compute spring layout
      pos_G = nx.layout.spring_layout(G, seed=1)


      # plot matrix
      plt.figure()
      plt.imshow(A, interpolation="nearest")
```

```
[2]:  <matplotlib.image.AxesImage at 0x125897010>
```

9.1. API documentation **31**
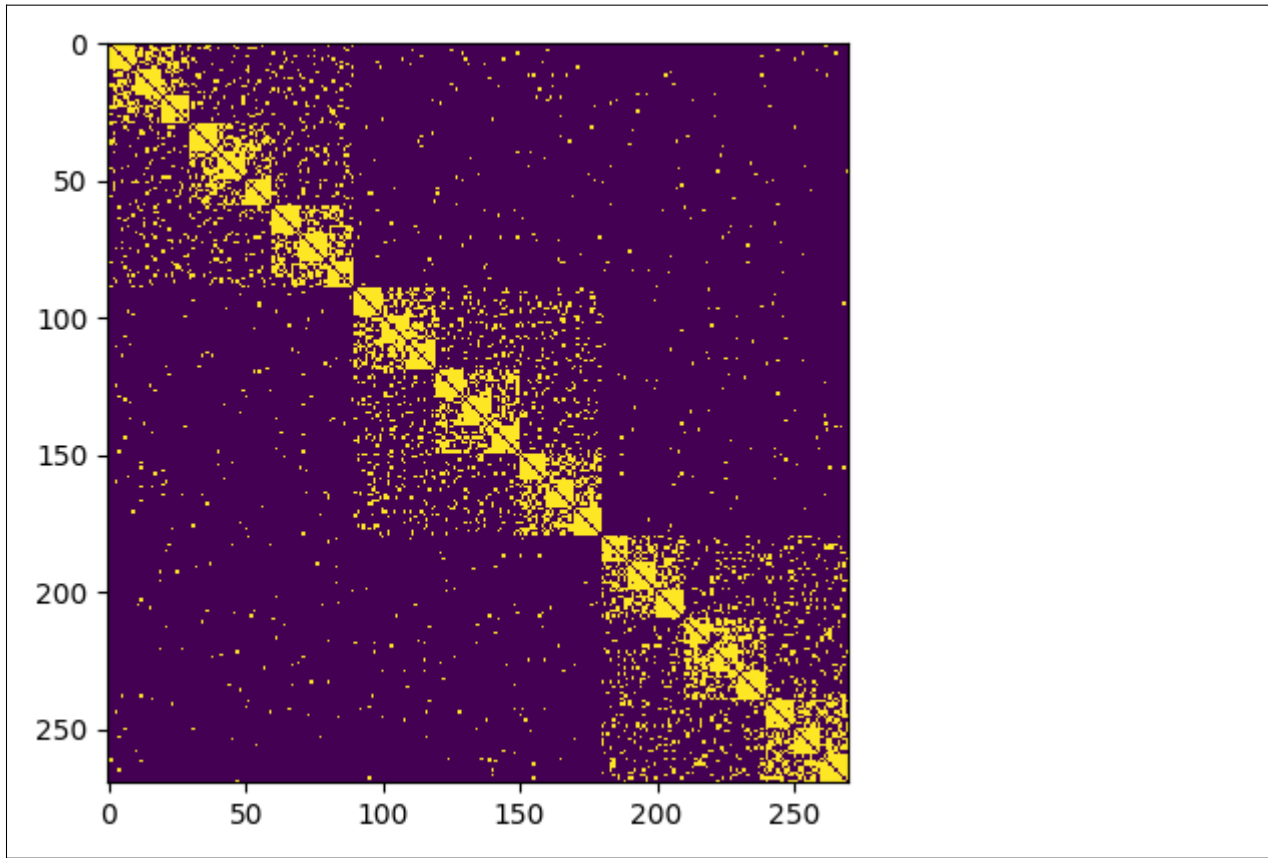
We then run pygenstability with the continuous_combinatoral constructor, which corresponds to using the combinatorial Laplacian matrix in the Markov Stability. The number and range of markov times, or scales can be specified with max_scale, min_scale and n_scales. They are in log scale by default. The number of Louvain evaluations is specified with n_tries argument.

Other options are available, see the documentation: https://barahona-research-group.github.io/PyGenStability/

```
[3]: # run markov stability and identify optimal scales
results = pgs.run(
    sp.csgraph.csgraph_from_dense(A),
    min_scale=-1.25,
    max_scale=0.75,
    n_scale=50,
    n_tries=20,
    constructor="continuous_combinatorial",
    n_workers=4
)
```

```
INFO:pygenstability.pygenstability:Precompute constructors...
INFO:pygenstability.pygenstability:Optimise stability...
INFO:pygenstability.pygenstability:Apply postprocessing...
INFO:pygenstability.pygenstability:Compute ttprimes...
INFO:pygenstability.pygenstability:Identify optimal scales...
```

The standard plot to analyse multiscale clustering results in plot_scan, which shows various informations, such as the number of cluster, stability, normalized variation of information (NVI) between Louvain evalutions, and accros scales (NVI(t, t')). Finally, if computed a scale selection algorithm highlights most robust scales.

```
[4]:  # plots results
      _ = plotting.plot_scan(results, figure_name=None)
```
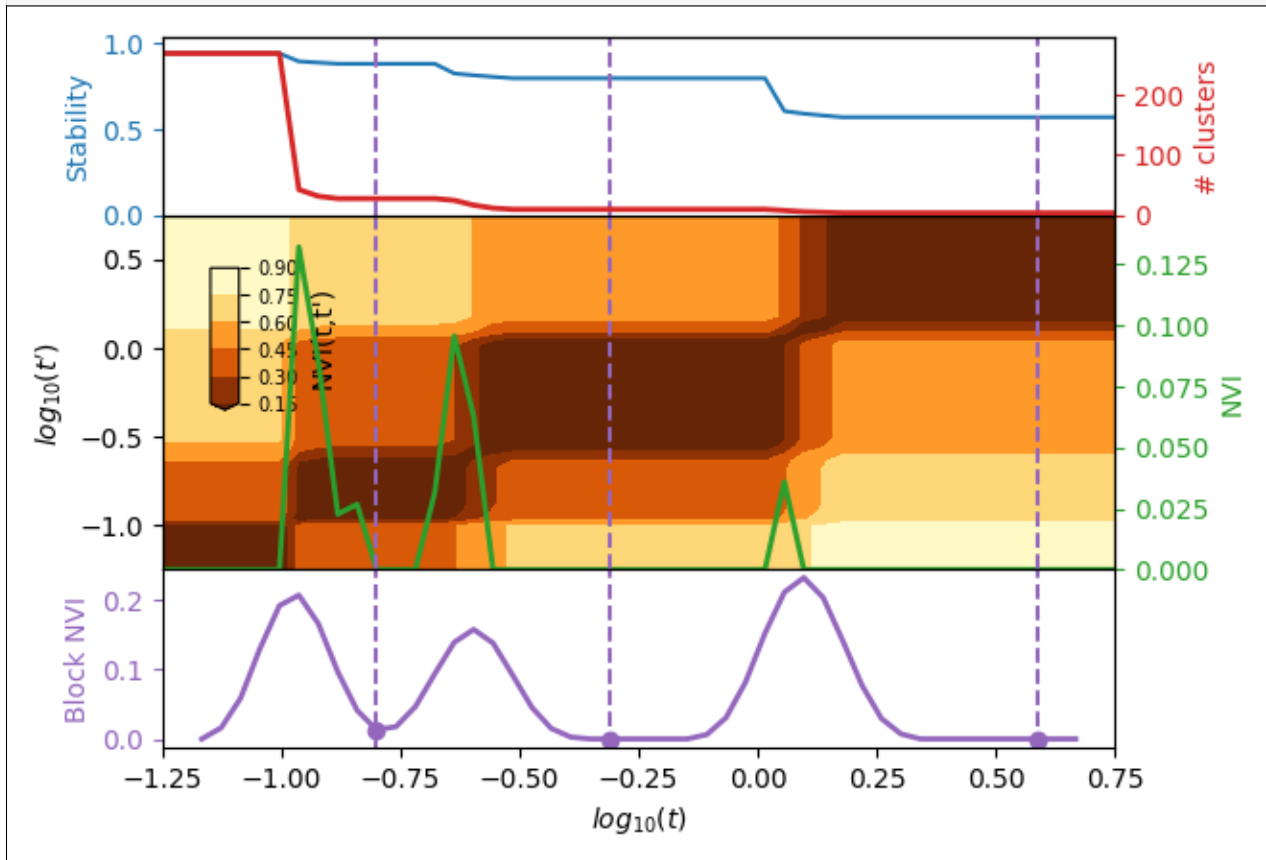

We can then plot the optimall partitions determined with the scale selection algorithm.

```
[7]:  # plot optimal partitions
      plotting.plot_optimal_partitions(G, results, show=True)
```

$log_{10}(scale) = -0.8$, with 27 communities
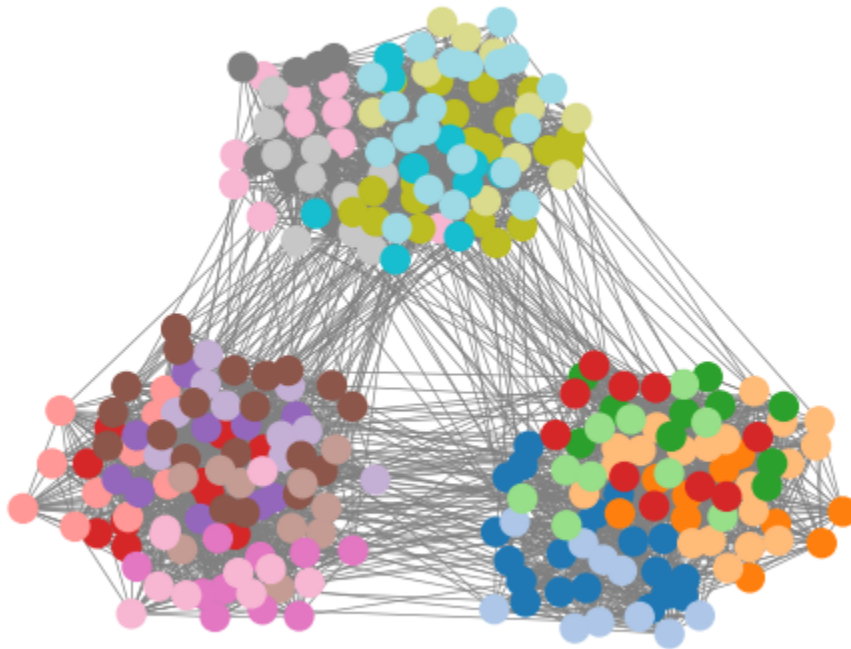
$log_{10}(scale) = -0.31$, with 9 communities
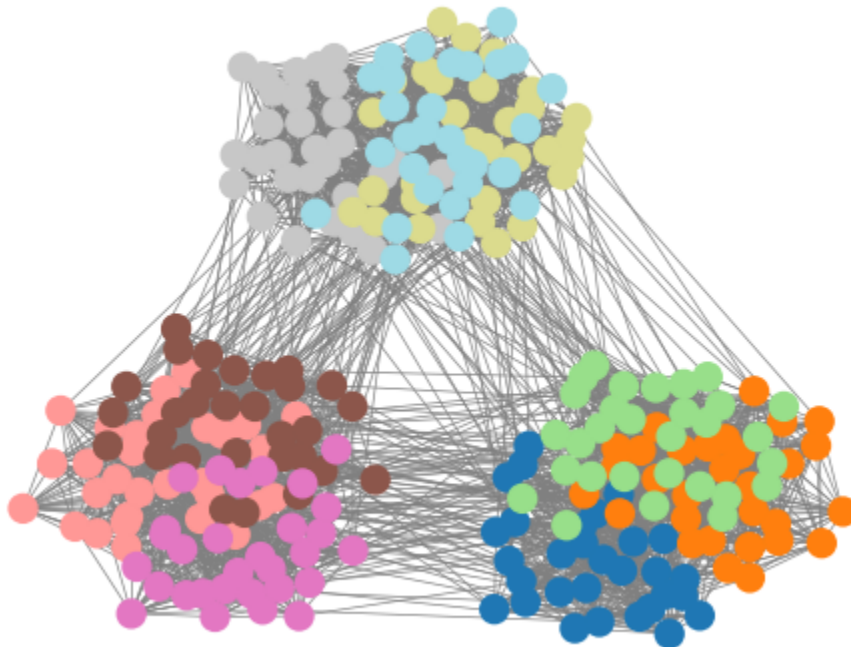
$log_{10}(scale) = 0.59$, with 3 communities
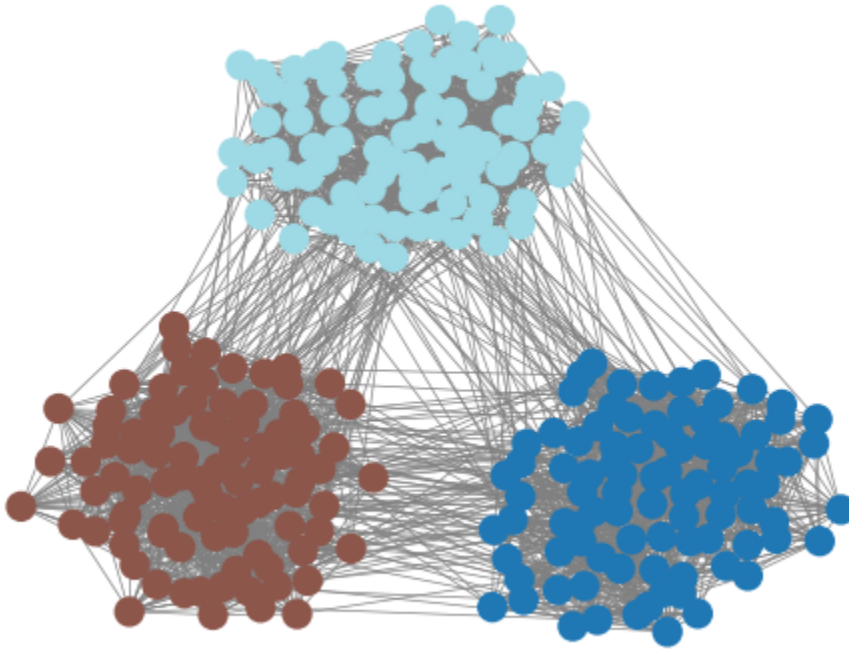
Finally, we compare the selected partitions with the ground-truth planted partitions using the Normalised Variation of Information (NVI) and observe that MS analysis with optimal scale selection recovers the planted multiscale structure of the network.

```python
[8]: # compare MS partitions to ground truth with NVI
     def _get_NVI(ref_ids):
         return [
             evaluate_NVI([0, i + 1], [ref_ids] + results["community_id"])
             for i in range(len(results["scales"]))
         ]

     NVI_scores_fine = _get_NVI(fine_scale_id)
     NVI_scores_middle = _get_NVI(middle_scale_id)
     NVI_scores_coarse = _get_NVI(coarse_scale_id)
     scales = results["scales"]

     # plot NVI scores
     fig, ax = plt.subplots(1, figsize=(7, 4))
     ax.plot(scales, NVI_scores_fine, label="Fine")
     ax.plot(scales, NVI_scores_middle, label="Middle")
     ax.plot(scales, NVI_scores_coarse, label="Coarse")

     # plot selected partitions
     selected_partitions = results["selected_partitions"]
     ax.axvline(
         x=results["scales"][selected_partitions[0]],
```

```
        ls="--",
        color="red",
        label="Selected Markov scales",
    )
for i in selected_partitions[1:]:
    ax.axvline(x=results["scales"][i], ls="--", color="red")

ax.set(xlabel=r"$log_{10}(t)$", ylabel="NVI")
plt.axhline(0, c="k", ls="--")
ax.legend(loc=3)
plt.xscale("log")
```
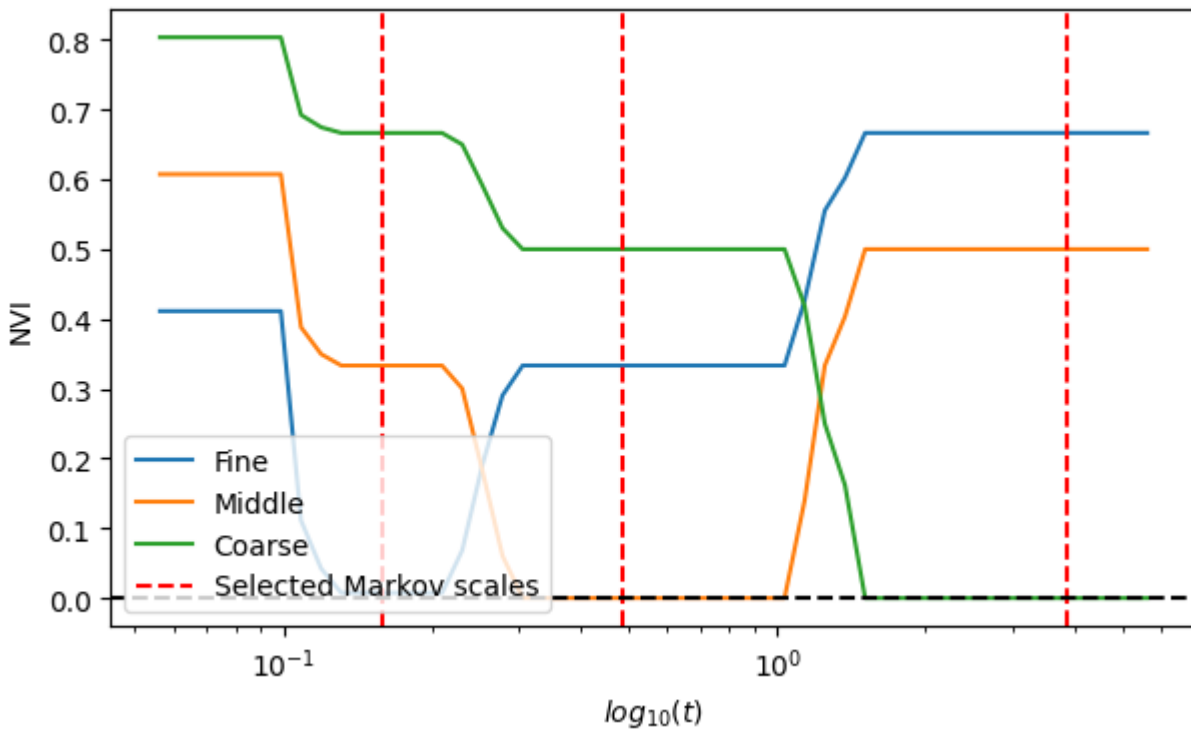

9.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

p