PyGT Documentation

Release 2020

Thomas D. Swinburne, Deepti Kannan

Mar 26, 2023

Contents:

Graph transformation is designed for the analysis of highly metastable (ill-conditioned) Markov chains, where linear algebra methods fail. [Wales09]

`PyGT` **produces stable coarse-grained models with exact branching probabilities and mean first passage times, in discrete or continuous time.** [Swinburne20a]

Note: You can install `PyGT` using the `pip` package manager (preferably in a virtual environment):

```
pip install PyGT
```

Simplest possible usage in continuous time with transition rates k_{ij}:

- Vector `tau` of mean state waiting times $\tau_j = 1/\left(\sum_i k_{ij}\right)$
- Sparse or dense matrix `B` of branching probabilities $B_{ij} = k_{ij}\tau_j$
- Boolean vector `rm_vec` selecting nodes to remove

```
import PyGT
# Removes nodes in blocks of <=50 whilst retaining numerical stability
gt_B, gt_tau = PyGT.GT.blockGT(rm_vec,B,tau,block=50,screen=True)
```

```
GT: 100% |████████████████████| 4811/4811 [00:01<00:00, 2661.69it/s]

1-by-1 GT subloop for stability: 100% |████| 50/50 [00:00<00:00, 132.38it/s]
GT removed 4811 nodes in 1.8 seconds with 1 floating point corrections
```

Note: Tutorials (see menu) can be run online with binder:

The notebooks can also be cloned from the `PyGT` github repo:

```
# clone entire source code and examples
git clone https://github.com/tomswinburne/PyGT.git
# go to examples folder
cd PyGT/examples
# run notebook
jupyter-notebook basic-functions.ipynb
```

Graph transformation [Wales09] is a deterministic dimensionality reduction algorithm that iteratively removes nodes from a Markov Chain while preserving the mean first passage time (MFPT) and branching probabilites between the retained nodes. The original Markov chain does not need to satisfy detailed balance.

This package provides an efficient implementation of the graph transformation algorithm, accelerated via partial block matrix inversions [Swinburne20a] for abitrary discrete-time or continuous-time Markov chains [Kannan20a]. Code is also provided for the calculation of first passage time statistics and phenomenological rate constants between endpoint macrostates [Wales09] [Swinburne20b].

We also include code for two different approaches to the dimensionality reduction of Markov chains using the graph transformation algorithm. In the first approach, we consider the problem of estimating a reduced Markov chain given a partitioning of the original Markov chain into communities of microstates (nodes) [Kannan20a]. Various implementations of the inter-community rates are provided, including the simplest expression given by the local equilibrium approximation, as well as the optimal rates originally derived by Hummer and Szabo. In the second approach, which

we call partial graph transformation [Swinburne20a] [Kannan20b], select nodes that contribute the least to global dynamics are renormalized away with graph transformation. The result is a smaller-dimensional Markov chain that is better-conditioned for further numerical analysis.

All methods are discussed in detail in the following manuscripts, which should also be cited when using this software:

References

CHAPTER 1

PyGT.io

This module reads in the following input files:

1.1 Files defining sets of nodes

min.A: single-column[int], (N_A + 1,) First line contains the number of nodes in community A. Subsequent lines contain the node ID (1-indexed) of the nodes belonging to A. Example:

```
10 # number of nodes in A
1
2
...
```

min.B: single-column[int], (N_B + 1,) First line contains the number of nodes in community B. Subsequent lines contain the node ID (1-indexed) of the nodes belonging to B.

communities.dat [single-column[int] (nnnodes,)] Each line contains community ID (0-indexed) of the node specified by the line number in the file

Example:

```
0
2
1
0
...
```

1.2 Files describing minimia and saddle points of energy landscape

The following files are designed to describe a Markov chain in which nodes correspond to potential or free energy minima, and edges correspond to the transition states that connect them. These files are also used as input to the PATHSAMPLE program implemented in the Fortran language:

min.data: multi-column, (nnodes, 6) Line numbers indicate node-IDs. Each line contains energy of local minimum [float], log product of positive Hessian eigenvalues $\sum_i \log|m\omega_i^2|$ [float], isometry [int], sorted eigenvalues of inertia tensor itx [float], ity [float], itz [float]

Example (LJ13 dataset):

```
-44.3268014195  158.2464487383  120     9.3629926605    9.3629926606    9.
↪3629926607 # node 1
-41.4719798478  153.8092000860  2       8.7543536583    10.6788841871   11.
↪4404999401 # node 2
...
```

ts.data: multi-column, (nts, 8) Each line contains energy of transition state [float], log product of positive Hessian eigenvalues $\sum_i \log|m\omega_i^2|$ [float], isometry [int], ID of first minimum it connects [int], ID of second minimum it connects [int], sorted eigenvalues of inertia tensor itx [float], ity [float], itz [float]

Example (LJ13 dataset):

```
-40.4326640775  148.9095497699  1       2       1       9.0473340846    10.
↪4342879996   10.9389332953
-40.9062828304  150.4182291647  2       3       1       8.6169912461    10.
↪3990395875   11.4674850889
...
```

The *PyGT.io* module then calculates transition rates using unimolecular rate theory.

1.3 Files defining a continuous-time Markov chain

The following files describe the transition rates between nodes and their stationary probabilities in an arbitrary continuous-time Markov chain.

ts_conns.dat [double-column[int], (nedges,)] Edge table where each row contains the IDs (1-indexed) of the nodes connected by each edge.

Example:

```
1   238 #edge connecting node 1 and node 238
2   307
...
```

ts_weights.dat [single-column[float], (2*nedges,)] Each pair of lines are the edge weights, $\ln(k_{i\leftarrow j})$ and $\ln(k_{j\leftarrow i})$ for the $i \leftrightarrow j$ bi-directional edge consistent with *ts_conns.dat*.

Example:

```
-5.6600770665 #ln[k(1<-238)]
-9.3074770665 #ln[k(238<-1)]
-5.8668770665 #ln[k(1<-307)]
-7.3402770665 #ln[k(307<-1)]
...
```

stat_prob.dat [single-column[float], (nnodes,)] Log stationary probabilities of nodes.

`PyGT.io.`**`load_ktn`**(*path*, *beta=1.0*, *Nmax=None*, *Emax=None*, *screen=False*, *discon=False*)
Load in min.data and ts.data files, calculate rates, and find connected components.

Parameters

- **`path`** (*str*) – path to min.data and ts.data files

- **beta** (*float, optional*) – value for $1/(k_BT)$. Default = 1.0
- **Nmax** (*int, optional*) – maximum number of minima to include in KTN. Default = None (i.e. infinity)
- **Emax** (*float, optional*) – maximum potential energy of minima/TS to include in KTN. Default = None (i.e. infinity)
- **screen** (*bool, optional*) – whether to print progress. Default = False
- **discon** (*bool, optional*) – Output data for disconnectivity graph construction (undocumented) Default = False

Returns

- **B** (*(N,N) matrix*) – sparse matrix of branching probabilities
- **K** (*(N,N) csr matrix*) – sparse matrix where off-diagonal elements K_{ij} contain $i \leftarrow j$ transition rates. Diagonal elements are 0.
- **tau** (*(N,) array_like*) – vector of waiting times such that total escape rate in state i is $1/\tau_i$ Full rate matrix is then given by $K_{ij} - \delta_{ij}/\tau_i$
- **N** (*int*) – number of nodes in the largest connected component of the Markov chain
- **u** (*(N,) array_like*) – energies of the N nodes in the Markov chain
- **s** (*(N,) array_like*) – entropies of the N nodes in the Markov chain
- **Emin** (*float*) – energy of global minimum (energies in *u* are rescaled so that Emin=0)
- **retained** (*np.ndarray[bool] (nnodes,)*) – Boolean array selecting out largest connected component (retained.sum() = N).

PyGT.io.**load_ktn_AB**(*data_path*, *retained=None*)

Read in *A_states* and *B_states* from min.A and min.B files, only keeping the states that are part of the largest connected set, as specified by *retained*.

Parameters

- **data_path** (*str*) – path to location of min.A, min.B files
- **retained** (*array-like[bool] (nnodes,)*) – selects out indices of the maximum connected set

Returns

- **A_states** (*array-like[bool] (retained.size,)*) – boolean array that selects out the A states
- **B_states** (*array-like[bool] (retained.size,)*) – boolean array that selects out the B states

PyGT.io.**read_communities**(*file*, *retained*, *screen=False*)

Read in a single column file called communities.dat where each line is the community ID (zero-indexed) of the nodes given by the line number. Produces boolean arrays, one per community, selecting out the nodes that belong to each community.

Parameters

- **file** (*str*) – single-column file containing community IDs of each minimum
- **retained** (*(N,) boolean array*) – selects out the largest connected component of the network

Returns **communities** – mapping from community ID (0-indexed) to a boolean array which selects out the states in that community.

Return type dictionary

PyGT.io.**read_ktn_info**(*path*, *suffix=''*)

Read input files stat_prob.dat, ts_weights.dat, and ts_conns.dat and return a rate matrix and vector of stationary probabilities.

Parameters

- **path** (*str*) – path to directory containing stat_prob.dat, ts_weights.dat, and ts_conns.dat files.
- **suffix** (*str*) – Suffix for file names, i.e. 'ts_weights{suffix}.dat'. Defaults to ''.

Returns

- **pi** (*array-like (nnodes,)*) – vector of stationary probabilities.
- **K** (*array-like (nnodes,nnodes)*) – CTMC rate matrix in dense format.

CHAPTER 2

PyGT.GT

2.1 Iteratively remove nodes from a Markov chain with graph transformation

This module implements the graph transformation algorithm to eliminate nodes from a discrete- or continuous-time Markov chain. When the removed nodes are chosen judiciously, the resulting network is less sparse, of lower dimensionality, and is generally better-conditioned. See *PyGT.tools* for various tools which help select nodes to eliminate, namely, ranking nodes based on their mean waiting times and equilibrium occupation probabilities. [Kannan20b]

The graph transformation algorithm requires a branching probability matrix **B** with elements $B_{ij} = k_{ij}\tau_j$ where k_{ij} is the $i \leftarrow j$ inter-microstate transition rate and τ_j is the mean waiting time of node j. In a discrete-time Markov chain, **B** is replaced with the discrete-time transition probability matrix $\mathbf{T}(\tau)$ and the waiting times of all nodes are uniform, equal to the lag time τ.

In each iteration of GT, a single node x is removed, and the branching probabilities and waiting times of the neighboring nodes are updated according to

to

$$B'_{ij} \leftarrow B_{ij} + \frac{B_{ix}B_{xj}}{1 - B_{xx}}$$
$$\tau'_j \leftarrow \tau_j + \frac{B_{xj}\tau_j}{1 - B_{xx}} \quad (2.1)$$

$$\leftarrow \tau_j + \frac{B_{xj}\tau_j}{1 - B_{xx}}$$

A matrix version of the above equations permits the removal of blocks of nodes simulatenously. The code monitors for stability in the inversion algorithm, to allow for for one-by-one node removal instead if a block is ill-conditioned. [Swinburne20a]

PyGT.GT.**blockGT**(*rm_vec*, *B*, *tau*, *block=20*, *order=None*, *rates=False*, *Ndense=50*, *screen=False*, *cond_thresh=10000000000000.0*)

Main function for GT code, production of a reduced matrix by graph transformation.

Parameters

- **rm_vec** (*(N,) array-like, bool*) – Boolean array of which nodes to remove
- **B** (*(N,N) dense or sparse matrix*) – Matrix of branching probabilities (CTMC) or transition probabilities (DTMC)
- **tau** (*(N,) array-like*) – Array of waiting times (CTMC) or lag times (DTMC)
- **block** (*int, optional*) – Number of node to attempt to remove simultaneously. Default = 20
- **order** (*(N,) array-like, optional*) – Order in which to remove nodes. Default ranks on node connectivity. Modify with caution: large effect on performance
- **rates** (*bool, optional*) – Whether to return the GT-reduced rate matrix in addition to B and tau. Only vaid for CTMC case. Default = False
- **Ndense** (*int, optional*) – Force switch to dense representation if N<Ndense. Default = 50
- **screen** (*bool, optional*) – Whether to print progress of GT. Default = False
- **cond_thresh** (*float, optional*) – Threshold condition number below which block matrix inversion is attempted. If block condition number is too high, conventional GT is used to remove nodes, which is less efficient but more stable. Default= `1e13`

Returns

- **B** (*(N',N') dense or sparse matrix*) – Matrix of N'<N renormalized branching probabilities. Will be returned as same type (sparse/dense) as input
- **tau** (*(N',) array-like*) – Array of N'<N renormalized waiting times
- **K** (*(N',N') dense or sparse matrix (same type as B)*) – Matrix of N'<N renormalized transition rates. Only if `rates=True`

PyGT.GT.**singleGT**(*rm_vec*, *B*, *tau*, *cond_thresh=10000000000000.0*)

Single iteration of GT algorithm used by main GT function. Either removes a single node with float precision correction [Wales09] or attemps node removal via matrix inversion [Swinburne20a]. In the latter case, if an error is raised by `np.linalg.inv` this is communicated through `success`

Parameters

- **rm_vec** (*(N,) array-like, bool*) – Boolean array of which nodes to remove
- **B** (*(N,N) dense or sparse matrix*) – Matrix of branching probabilities
- **tau** (*(N,) array-like*) – Array of waiting times
- **cond_thresh** (*float, optional*) – Threshold condition number below which matrix inversion is attempted. If condition number is higher than `cond_thresh`, `singleGT()` returns `success=False`. Default= `1.0e13`

Returns

- **B** (*(N',N') dense or sparse matrix*) – Matrix of N'<N renormalized branching probabilities
- **tau** (*(N',) array-like*) – Array of N'<N renormalized waiting times
- **success** (*bool*) – False if estimated condition number is too high or `LinAlgError` raised by `np.linalg.inv`

CHAPTER 3

PyGT.stats

3.1 Calculate first passage statistics between macrostates

Tools to calculate the first passage time distribution and phenomenological rate constants between endpoint macrostates $\mathcal{A}$ and $\mathcal{B}$.

Note: Install the *pathos* package to parallelize MFPT computations.

`PyGT.stats.`**`compute_passage_stats`**(*A_sel*, *B_sel*, *pi*, *K*, *dopdf=True*, *rt=None*)

Compute the A->B and B->A first passage time distribution, first moment, and second moment using eigendecomposition of a CTMC rate matrix.

Parameters

- **`A_sel`** (*`(N,) array-like`*) – boolean array that selects out the A nodes
- **`B_sel`** (*`(N,) array-like`*) – boolean array that selects out the B nodes
- **`pi`** (*`(N,) array-like`*) – stationary distribution
- **`K`** (*`(N, N) array-like`*) – CTMC rate matrix
- **`dopdf`** (*`bool, optional`*) – Do we calculate full fpt distribution or just the moments. Defaults=True.
- **`rt`** (*`array, optional`*) – Vector of times to evaluate first passage time distribution in multiples of $\langle t \rangle$ for A->B and B->A. If `None`, defaults to a logscale array from $0.001\ \langle t \rangle$ to $1000\ \langle t \rangle$ in 400 steps, i.e. `np.logspace(-3,3,400)`. Only relevant if `dopdf=True`

Returns

- **tau** (*(4,) array-like*) – First and second moments of first passage time distribution for A->B and B->A [$\mathcal{T}_{\mathcal{BA}}$, $\mathcal{V}_{\mathcal{BA}}$, $\mathcal{T}_{\mathcal{AB}}$, $\mathcal{V}_{\mathcal{AB}}$]
- **pt** (*(len(rt),4) array-like*) – time and first passage time distribution p(t) for A->B and B->A

PyGT.stats.**compute_escape_stats**(*B_sel*, *pi*, *K*, *tau_escape=None*, *dopdf=True*, *rt=None*)

Compute escape time distribution and first and second moment from the basin specified by *B_sel* using eigendecomposition.

Parameters

- **B_sel** (*(N,) array-like*) – boolean array that selects out the nodes in the active basin
- **pi** (*(N,) array-like*) – stationary distribution for CTMC
- **K** (*(N, N) array-like*) – CTMC rate matrix
- **tau_escape** (*float*) – mean time to escape from B. Used to calculate the escape time distribution in multiple of tau_escape (p(t/tau_escape). If None, uses the first moment in network defined by K.
- **dopdf** (*bool*) – whether to calculate full escape time distribution, defaults to True
- **rt** (*array, optional*) – Vector of times to evaluate first passage time distribution in multiples of $\langle t \rangle$ for A->B and B->A. If None, defaults to a logscale array from $0.001\ \langle t \rangle$ to $1000\ \langle t \rangle$ in 400 steps, i.e. `np.logspace(-3,3,400)`. Only relevant if `dopdf=True`

Returns

- **tau** (*(2,) array-like*) – First and second moments of escape time distribution, $[\langle t \rangle_{\mathcal{B}}, \langle t^2 \rangle_{\mathcal{B}}]$
- **pt** (*(2, len(rt)) array-like*) – time and escape time distribution $p(t)\ \langle t \rangle$

PyGT.stats.**compute_rates**(*A_sel*, *B_sel*, *B*, *tau*, *pi*, *initA=None*, *initB=None*, *MFPTonly=True*, *fullGT=False*, *pool_size=None*, *block=1*, *screen=False*, ***kwargs*)

In a total state space partitioned into three sets A,B and I, calculate various approximate A<->B transition rates [Wales09] and the MFPT from a rate matrix K. K can be the matrix of an original Markov chain, or a partially graph-transformed Markov chain. [Swinburne20a]

Rate definitions (all assume A,B in local equilibrium) see [Swinburne20a] for details

kSS : assumes I in steady state, A,B local equilibrium

kNSS : Relaxes counts for non-steady state in I. Tends to exact rate if reactant metastable

k^F : Boltzmann weighted sum of inverse of exact MFPT from each reactant state, i.e. $\sum_j \pi_j / \mathcal{T}_j$

k^* : Inverse of exact MFPT from reactant, i.e. $1/\sum_j \pi_j \mathcal{T}_j$

`compute_rates()` differs from `compute_passage_stats()` in that this function removes all intervening states using GT before computing FPT stats and rates on the fully reduced network with state space $(\mathcal{A} \cup \mathcal{B})$. This implementation also does not rely on a full eigendecomposition of the non-absorbing matrix; it instead performs a matrix inversion, or if *fullGT* is specified, all nodes in the set $(\mathcal{A} \cup b)^c$ are removed using GT for each $b \in \mathcal{B}$ so that the MFPT is given by an average:

$$\mathcal{T}_{\mathcal{AB}} = \frac{1}{\sum_{b \in \mathcal{B}} p_b(0)} \sum_{b \in \mathcal{B}} \frac{p_b(0) \tau'_b}{1 - P'_{bb}}$$

If the MFPT is less than 10^{20}, *fullGT* should not be needed since the inversion of the non-absorbing matrix should be numerically stable. However, a condition number check is performed regardless, which forces a full GT if the MFPT problem is considered numerically unstable.

Parameters

- **A_sel** (*array-like (N,)*) – selects the N_A nodes in the $\mathcal{A}$ set.
- **B_sel** (*array-like (N,)*) – selects the N_B nodes in the $\mathcal{B}$ set.
- **B** (*sparse or dense matrix (N,N)*) – branching probability matrix for CTMC
- **tau** (*array-like (N,)*) – vector of waiting times from each node.
- **pi** (*array-like (N,)*) – stationary distribution of CTMC
- **initA** (*array-like (N_A,), optional*) – normalized initial occupation probabilities in $\mathcal{A}$ set. Default= local Boltzmann distribution
- **initB** (*array-like (N_B,), optional*) – normalized initial occupation probabilities in $\mathcal{B}$ set. Default= local Boltzmann distribution
- **MFPTonly** (*bool*) – If True, only MFPTs are calculated (rate calculations ignored).
- **fullGT** (*bool*) – If True, all source nodes are isolated with GT to obtain the average MFPT.
- **pool_size** (*int*) – Number of cores over which to parallelize fullGT computation.

Returns **results** – dictionary of results, with keys 'MFPTAB', 'kSSAB', 'kNSSAB', 'kQSDAB', 'k*AB', 'kFAB' for A<-B and 'MFPTBA', 'kSSBA', 'kNSSBA', 'kQSDBA', 'k*BA', 'kFBA' for B<-A such that results['MFPTAB'] = mean first passage time for A<-B

Return type dictionary

CHAPTER 4

PyGT.mfpt

4.1 Calculate matricies of mean first passage times with graph transformation

Note: Install the *pathos* package to parallelize MFPT computations, with e.g.

```
pip install pathos
```

PyGT.mfpt.**full_MFPT_matrix**(*B*, *tau*, *pool_size=1*, *screen=False*, ***kwargs*)

Compute full matrix of inter-microstate MFPTs with GT.

Parameters

- **B** (*sparse or dense matrix (N,N)*) – branching probability matrix.
- **tau** (*array-like (N,)*) – vector of waiting times from each node.
- **pool_size** (*int, optional*) – Number of cores over which to parallelize computation. only attempted if `pool_size>1` and `pathos` package is installed. Default=1.
- **screen** (*bool, optional*) – Show progress bar. Default=False

Returns **mfpt** – matrix of inter-microstate MFPTs between all pairs of nodes

Return type np.ndarray[float64] (N,N)

PyGT.mfpt.**community_MFPT_matrix**(*communities*, *B*, *tau*, *pi*, *MS_approx=False*, *diagzero=True*, *pool_size=1*, *screen=False*, ***kwargs*)

Compute matrix of effective inter-macrostate MFPTs with GT, defined as

$$[\mathcal{T}_{\mathrm{C}}]_{IJ} = \frac{1}{\Pi_I \Pi_J} \sum_{i \in I} \sum_{j \in J} \pi_i \mathcal{T}_{ij} \pi_j - \frac{1}{\Pi_I \Pi_I} \sum_{i \in I} \sum_{i' \in I} \pi_{i'} \mathcal{T}_{i'i} \pi_i,$$

This matrix has diagonal elements equal to zero and can be shown to satisfy the Kemeny constant constraint, and thus is suitable for producing a coarse grained rate matrix. If *diagzero* is set to False, an alternative matrix is computed where the second term of the above equation is ignored so that the diagonal elements of the matrix will no longer be zero. This alternative matrix also satisfies the Kemeny constant constraint. [Kannan20a]

Parameters

- **communities** (*dict*) – mapping from community ID (0-indexed) to a boolean array of shape (N,) which selects out the states in that community. Communities must be disjoint. Communities must be disjoint.
- **B** (*sparse or dense matrix (N,N)*) – branching probability matrix.
- **tau** (*array-like (N,), float*) – vector of waiting times from each node.
- **pi** (*array-like (N,), float*) – stationary probability distribution of microstates
- **MS_approx** (*bool, optional*) – If True, assume all communities are sufficiently metastable to use only one microstate pair per community pair, a much more efficient but approximate computation. [Kannan20a]
- **diagzero** (*bool*) – Choose the inter-community MFPTs such that the diagonal elements are zero. Defaults to True.
- **pool_size** (*int, optional*) – Number of cores over which to parallelize computation. Default=1 Only active if MS_approx = False
- **screen** (*bool, optional*) – Print progress. Default = False

Returns

- **Pi** (*array-like (N_comm,)*) – Macroscopic stationary distribution. Indexed in ascending order
- **tau_AB** (*dense matrix (N_comm,N_comm)*) – matrix of effective inter-macrostate MFPTs

PyGT.mfpt.**compute_MFPT**(*i*, *j*, *B*, *tau*, *block=1*, ***kwargs*)

Compute the inter-microstate $i \leftrightarrow j$ MFPT using GT. Called by `full_MFPT_matrix()`. Unlike `compute_rates()` function, which assumes there is at least 2 microstates in the absorbing macrostate, this function does not require knowledge of equilibrium occupation probabilities since $\mathcal{T}_{ij} = \tau'_j / B'_{ij}$ when there are only two nodes remaining after GT

Parameters

- **i** (*int*) – node-ID (0-indexed) of first microstate.
- **j** (*int*) – node-ID (0-indexed) of second microstate.
- **B** (*sparse or dense matrix (N,N)*) – branching probability matrix.
- **tau** (*array-like (N,)*) – vector of waiting times from each node.
- **block** (*int, optional*) – block size for matrix generalization GT procedure. Reverts to slower but guaranteed stable one-by-one GT (block=1) when `numpy` matrix inversion routine raises errors. Default=10

Returns

- **MFPTij** (*float*) – mean first passage time $i \leftarrow j$

- **MFPTji** (*float*) – mean first passage time $j \leftarrow i$

CHAPTER 5

PyGT.spectral

5.1 Spectral analysis for community-based dimensionality reduction

Produce projection matricies in order to reduce a CTMC rate matrix from a given community structure, using the local equilibrium approximation (LEA) or spectral clustering method as investigated in [Swinburne20b].

The LEA method assumes each community is locally metastable, meaning the distribution of states in each community will be approximately proportional to the local Boltzmann distribution π_j.

The LEA thus takes a single left and right vector pair per community *J*, namely

$$[\mathbf{1}_J]_j = \delta(j \in J), \quad [\hat{\pi}_J]_j = \delta(j \in J)\frac{\pi_j}{\sum_{j' \in J}\pi_{j'}},$$

to produce the reduced rate matrix, which corresponds to the local equilibrum distribution projected onto the community.

The spectral clustering method generalizes this approach, performing a local eigendecomposition and projection to generate a set of left and right vector pairs, from which a subset is used to produce the reduced rate matrix. In particular, the set is projected such that the slowest eigenvector pair becomes the LEA pair $(\mathbf{1}_J, \hat{\pi}_J)$, allowing interpolation between the LEA and the exact solution

The vector pairs are chosen until the first `nmoments` moments of the community escape time is reprodued to a relative error of `tol`. Further details can be found in [Swinburne20b].

`PyGT.spectral.project()` returns left and right projections matricies $\mathbf{Y}, \mathbf{X}$ such that the reduction operation is given by

$$\mathbf{Q} \in \mathbb{R}^{N\times N} \to \mathbf{YQX} \in \mathbb{R}^{N'\times N'}, \quad N' \leq N$$

With an error tolerance `tol=0` we recover the exact solution, i.e. $N' \to N, \mathbf{Y} \to \mathbb{I}_N, \mathbf{X} \to \mathbb{I}_N$

`PyGT.spectral.reduce()` uses these matricies to produce the reduced rate matrix and reduced left and right stationary vectors and initial distribution ρ given by

$$\hat{\pi} \in \mathbb{R}^N \to \mathbf{Y}\hat{\pi} \in \mathbb{R}^{N'}, \quad \mathbf{1} \in \mathbb{R}^N \to \mathbf{1}\mathbf{X} \in \mathbb{R}^{N'}, \quad \rho \in \mathbb{R}^N \to \mathbf{Y}\rho \in \mathbb{R}^{N'}$$

`PyGT.spectral.`**`reduce`**(*communities*, *pi*, *Q*, *initial_dist=None*, *style='specBP'*, *nmoments=2*, *tol=0.01*)

Returns a reduced rate matrix and initial distribution (optional) using `PyGT.spectral.project()`

Parameters

- **`communities`** (*`dict`*) – mapping from community ID (0-indexed) to a boolean array of shape (N,) which selects out the states in that community. Communities must be disjoint.
- **`pi`** (*`(N,) array-like`*) – stationary distribution
- **`Q`** (*`(N, N) array-like`*) – rate matrix
- **`initial_dist`** (*`(N,) array-like, optional`*) – Initial probability distribution for first passage time problems, for example the local Boltzmann distribtion of a community Default=None
- **`style`** (*`string, optional`*) – Reduction method. Must be one of
  - `LEA` : Apply local equilibrum approximation.
  - `specBP` : Perform spectral reduction with mode basis modified such that slowest mode becomes the LEA mode, with all other modes orthogonal to this mode but not mutually orthonormal. (Default)
  - `specBPO` : Perform spectral reduction with mode basis *rotated* such that slowest mode becomes the LEA mode, with all other modes mutually orthonormal. Gives similar results to `specBP`.
- **`nmoments`** (*`int, optional`*) – Number of escape time moments to monitor for accuracy. Ignored if `style=LEA`. Higher number implies less reduction in matrix rank. Default=2.
- **`tol`** (*`float, optional`*) – Relative error tolerance for moments. Ignored if `style=LEA`. Default=0.01
- **`communities`** – mapping from community ID (0-indexed) to a boolean array of shape (N,) which selects out the states in that community. Communities must be disjoint.
- **`pi`** – stationary Boltzmann distribution for entire system
- **`Q`** – rate matrix

Returns

- **Q** (*(N', N') array-like*) – Reduced rate matrix

- **pi** (*(N',2) array-like*) – Reduced left and right stationary distribution vectors for calculation of first passage distributions. With no projection pi[0] = vector of ones, pi[1] = Boltzmann distribution.
- **rho** (*(N') array-like*) – Reduced initial distribution (only if `initial_dist` is provided)

`PyGT.spectral.`**`project`**(*communities*, *pi*, *Q*, *style='specBP'*, *nmoments=2*, *tol=0.01*)

Produce a projection matricies in order to reduce a CTMC rate matrix from a given community structure.

Parameters

- **`communities`** (*`dict`*) – mapping from community ID (0-indexed) to a boolean array of shape (N,) which selects out the states in that community. Communities must be disjoint.
- **`pi`** (*`(N,) array-like`*) – stationary distribution
- **`Q`** (*`(N, N) array-like`*) – rate matrix
- **`style`** (*`string, optional`*) – Reduction method. Must be one of
  - `LEA` : Apply local equilibrum approximation.
  - `specBP` : Perform spectral reduction with mode basis modified such that slowest mode becomes the LEA mode, with all other modes orthogonal to this mode but not mutually orthonormal. (Default)
  - `specBPO` : Perform spectral reduction with mode basis *rotated* such that slowest mode becomes the LEA mode, with all other modes mutually orthonormal. Gives similar results to `specBP`.
- **`nmoments`** (*`int, optional`*) – Number of escape time moments to monitor for accuracy. Ignored if `style=LEA`. Higher number implies less reduction in matrix rank. Default=2.
- **`tol`** (*`float, optional`*) – Relative error tolerance for moments. Ignored if `style=LEA`. Default=0.01

Returns

- **Y** (*(N', N) array-like*) – left projection matrix
- **X** (*(N, N') array-like*) – right projection matrix

CHAPTER 6

PyGT.tools

6.1 Optimal Markovian coarse-graining for a given community structure

Various functions to analyze Markov chains, including estimating the optimal coarse-grained CTMC for a given community structure.

PyGT.tools.**choose_nodes_to_remove**(*rm_region*, *pi*, *tau*, *style='free_energy'*, *percent_retained=50*)

Given a branching probability matrix, stationary distribution and waiting times of a CTMC, return a Boolean array selecting nodes from a given subset to remove by graph transformation according to some simple criteria. [Kannan20b]

Parameters

- **rm_region** (*(N,) array*) – boolean array that specifies region in which nodes can be removed i.e. for A<->B dynamics, all nodes in A,B should be retained.
- **pi** (*(N,) array*) – stationary distribution
- **tau** (*(N,) array*) – vector of waiting times
- **B** (*sparse matrix, optional*) – sparse matrix of branching (CTMC) or transition (DTMC) probabilities, used for *style='node_degree'*
- **style** (*str, optional*) – ranking used to remove nodes, from high to low. Highest percentile is removed 'escape_time' : (=*tau*)ascending,

  'free_energy' : (=-*log|pi|*), descending,

  'hybrid' : remove nodes in highest percentile in *escape_time* and *free_energy*

  'combined' : *tau* * *pi*, ascending,

  'node_degree': descending (requires *B*),

  Default = *free_energy*

- **percent_retained** (*float, optional*) – percent of nodes to keep in reduced network. Default = 50.0

Returns **rm_vec** – boolean array that specifies nodes to remove, which will always be a subset of the nodes selected by *rm_region*

Return type (N,) array

PyGT.tools.**check_detailed_balance**(*pi*, *K*)

Check if Markov chain satisfies detailed balance condition, $k_{ij}\pi_j = k_{ji}\pi_i$ for all i, j.

Parameters

- **pi** (*array-like (N,)*) – stationary probabilities of full or reduced system
- **K** (*sparse or dense matrix (N, N)*) – transition rate matrix of full or reduced system

Returns Self-explanatory

Return type success, bool

PyGT.tools.**make_fastest_path**(*G*, *i*, *f*, *depth=1*, *limit=None*)

Wrapper for *scipy.sparse.csgraph.shortest_path* which returns node indicies on as-determined shortest i->f path and those *depth* connections away. Used to determine which nodes to remove by graph transformation during sensitivity analysis applied to kinetic transition networks [Swinburne20a].

Parameters

- **B** (*(N,N) sparse matrix*) – Matrix that will be interpreted as weighted graph for path calculation
- **i** (*int*) – Initial node index
- **f** (*int*) – Initial node index
- **depth** (*int, (default=1)*) – Size of near-path region

Returns

- **path** (*array_like*) – indices of path nodes
- **path_region** (*array_like*) – indices of near-path nodes

PyGT.tools.**check_kemeny**(*pi*, *tauM*)

Check that Markov chain satisfies the Kemeny constant relation, $\sum_i \pi_i, \mathcal{T}_{ij} = \xi$ for all j, where ξ is a constant and $\mathcal{T}_{ij}$ is the j->i mean first passage time.

Parameters

- **pi** (*array-like (N,)*) – stationary probabilities of full or reduced system
- **tauM**(*sparse or dense matrix (N, N)*) – mean first passage time matrix of full or reduced system

Returns

- *kemeny_constant, float* – Average kemeny constant $mean(xi)$ across nodes
- *success, bool* – Self-explanatory. False if $std(xi)/mean(xi) > 1e-9$

PyGT.tools.**load_CTMC**(*K*)

Setup a GT calculation for a transition rate matrix representing a continuous-time Markov chain.

Parameters **K** (*array-like (nnodes, nnodes)*) – Rate matrix with elements K_{ij} corresponding to the $i \leftarrow j$ transition rate and diagonal elements $K_{ii} = \sum_\gamma K_{\gamma i}$ such that the columns of **K** sum to zero.

Returns

- **B** (*np.ndarray[float64] (nnodes, nnodes)*) – Branching probability matrix in dense format, used as input to GT
- **tau** (*np.ndarray[float64] (nnodes,)*) – Vector of waiting times of nodes, used as input to GT

PyGT.tools.**load_DTMC**(*T*, *tau_lag*)

Setup a GT calculation for a transition probability matrix representing a discrete-time Markov chain.

Parameters

- **T** (*array-like (nnodes, nnodes)*) – Discrete-time, column-stochastic transition probability matrix.
- **tau_lag** (*float*) – Lag time at which *T* was estimated.

Returns

- **B** (*np.ndarray[float64] (nnodes, nnodes)*) – Branching probability matrix in dense format, used as input to GT
- **tau** (*np.ndarray[float64] (nnodes,)*) – Vector of waiting times of nodes, used as input to GT

PyGT.tools.**eig_wrapper**(*M*)

Wrapper of `scipy.linalg.eig` that returns real eigenvalues and orthonormal left and right eigenvector pairs

Parameters **M** (*(N,N) dense matrix*) –

Returns

- **nu** (*(N,) array-like*) – Real component of eigenvalues
- **v** (*(N,N) array-like*) – Matrix of left eigenvectors
- **w** (*(N,N) array-like*) – Matrix of right eigenvectors

class PyGT.tools.**Analyze_KTN**(*path*, *K=None*, *pi=None*, *commpi=None*, *communities=None*, *commdata=None*)

Estimate a coarse-grained continuous-time Markov chain given a partiioning $\mathcal{C} = \{I, J, ...\}$ of the V nodes into $N < V$ communities. Various formulations for the inter-community rates are implemented, including the local equilibrium approximation, Hummer-Szabo relation, and other routes to obtain the optimal coarse-grained Markov chain for a given community structure. [Kannan20a]

path

path to directory with all relevant files

Type str or Path object

K

Rate matrix with elements K_{ij} corresponding to the *i leftarrow j* transition rate, and diagonal elements $K_{ii} = -\sum_\gamma K_{\gamma i}$ such that the columns sum to zero.

Type array-like (nnodes, nnodes)

pi

Stationary distribution of nodes, π_i, i.e. vector of equilibrium occupation probabilities.

Type array-like (nnodes,)

commpi

Stationary distribution of communities, $\Pi_J = \sum_{j \in J} \pi_j$.

Type array-like (ncomms,)

communities
dictionary mapping community IDs (1-indexed) to node IDs (1-indexed).

Type dict

commdata
Filename, located in directory specified by *path*, of a single-column file where each line contains the community ID (0-indexed) of the node specified by the line number in the file.

Type str

Note: Either *communities* or *commdata* must be specified.

---

construct_coarse_rate_matrix_LEA()
Calculate the coarse-grained rate matrix obtained using the local equilibrium approximation (LEA).

construct_coarse_rate_matrix_Hummer_Szabo()
Calculate the optimal coarse-grained rate matrix using the Hummer-Szabo relation, aka Eqn. (12) in Hummer & Szabo *J. Phys. Chem. B.* (2015).

construct_coarse_rate_matrix_KKRA(*mfpt=None*, *GT=False*, ***kwargs*)
Calculate optimal coarse-grained rate matrix using Eqn. (79) of Kells et al. *J. Chem. Phys.* (2020), aka the KKRA expression in Eqn. (10) of [Kannan20a].

Parameters

- **mfpt** (*(nnodes, nnodes)*) – Matrix of inter-microstate MFPTs between all pairs of nodes. Defaults to None.
- **GT** (*bool*) – If True, matrix of inter-microstate MFPTs is computed with GT. Kwargs can then be specified for GT (such as the pool_size for parallelization). Defaults to False.

get_intermicrostate_mfpts_linear_solve()
Calculate the matrix of inter-microstate MFPTs between all pairs of nodes by solving a system of linear equations given by Eq.(8) of [Kannan20a].

get_intermicrostate_mfpts_fundamental_matrix()
Calculate the matrix of inter-microstate MFPTs between all pairs of nodes using Eq. (6) of [Kannan20a].

get_intercommunity_MFPTs_linear_solve()
Calculate the true MFPTs between communities by inverting the non-absorbing rate matrix. Equivalent to Eqn. (14) in Swinbourne & Wales *JCTC* (2020).

get_intercommunity_weighted_MFPTs(*mfpt*, *diagzero=True*)
Comppute the matrix $\widetilde{\mathcal{T}}_{\mathrm{C}}$ of appropriately weighted inter-community MFPTs, as defined in Eq. (18) in [Kannan20a].

Parameters

- **mfpt** (*array-like (N,N)*) – matrix of intermicrostate MFPTs.
- **diagzero** (*bool*) – Whether to define the inter-community weighted MFPTs such as the diagonal elements are zero. Defaults to True.

get_timescale_error(*m*, *K*, *R*)
Calculate the ith timescale error for i in {1,2,...m} of a coarse-grained rate matrix R compared to the full matrix K.

Parameters

- **m** (*int*) – Number of dominant eigenvalues (m < N)

- **K** (*np.ndarray[float] (V, V)*) – Rate matrix for full network
- **R** (*np.ndarray[float] (N, N)*) – Coarse-grained rate matrix

Returns **timescale_errors** – Errors for m-1 slowest timescales

Return type np.ndarray[float] (m-1,)

get_eigenfunction_error(*m*, *K*, *R*)

Calculate the i^{th} eigenvector approximation error for $i \in 1, 2, ...m$ of a coarse-grained rate matrix *R* by comparing its eigenvector to the correspcorresponding eigenvector of the full matrix.

get_comm_stat_probs(*pi*, *log=False*)

Calculate the community stationary probabilities by summing over the stationary probabilities of the nodes in each community.

Parameters **pi** (*list (nnodes,)*) – stationary probabilities of node in original Markov chain

Returns **commpi** – stationary probabilities of communities in coarse coarse_network

Return type list (ncomms,)

read_communities(*commdat*)

Read in a single column file called communities.dat where each line is the community ID (zero-indexed) of the minima given by the line number.

Parameters **commdat** (*dat file*) – single-column file containing community IDs of each minimum

Returns **communities** – mapping from community ID (1-indexed) to minima ID (1-indexed)

Return type dict

CHAPTER 7

Basic Tutorial

[1]:
```
# Uncomment if PyGT has not been installed via pip
# import sys; sys.path.insert(0,"../")

import numpy as np
import matplotlib.pyplot as plt
# for colorbar placement
from mpl_toolkits.axes_grid1 import make_axes_locatable

# To construct rate matrix
from scipy.sparse import issparse, diags

import PyGT
```

7.1 Load in matrix and vectors selecting $\mathcal{A}, \mathcal{B}$ regions using the KTN format

KTN (Kinetic Transition Network) format requires two files:

- `min.data` columns : E, $2S/\mathrm{k_B}$, D, I_x, I_y, I_z
- `ts.data` columns : E, $2S/\mathrm{k_B}$, D, f, i, I_x, I_y, I_z

where

- E = Energy
- S = Entropy
- D = Degeneracy
- I_x = x-moment of intertia
- f, i = final,initial state indicies

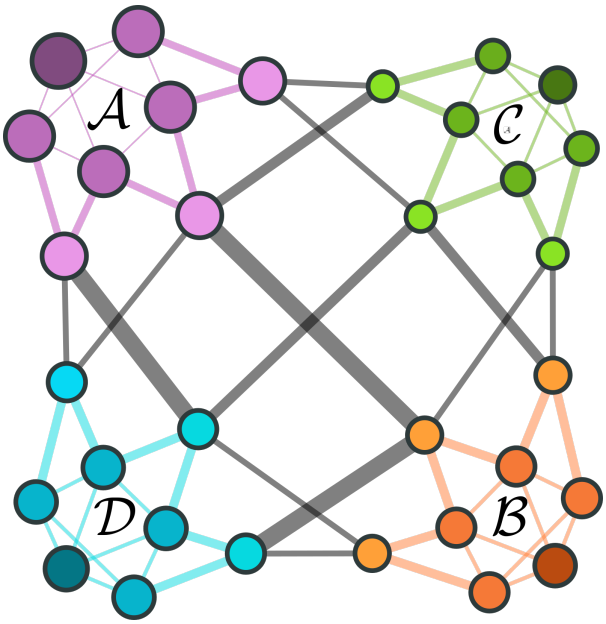
Some technical details: - `load_ktn()` function looks for files `[min,ts].data` to build KTN, pruning isolated nodes, giving a new node indexing

- `load_ktn_AB()` function looks for files `min.[A,B]` which use the same indicies as the data file
- `retained` is a vector that maps from the unpruned to pruned indexing convention, allowing `min.[A,B]` to be read
- Note that `A_vec, B_vec` can clearly be determined without using `load_ktn_AB()` or `retained`

For this example we are loading in a 32 state network:

[1]:
```
from IPython.display import Image
Image(filename = "32state.png", width = 300)
```

[1]:



[2]:
```
data_path = "KTN_data/32state"
temp = 1.
beta = 1./temp

B, K, tau, N, u, s, Emin, retained = PyGT.io.load_ktn(path=data_path,beta=beta,
↪screen=True)

F = u - s/beta # free energy

pi = np.exp(-beta * F) / np.exp(-beta * F).sum() # stationary distribution

# K has no diagonal entries
if issparse(K):
    Q = K - diags(1.0/tau)
else:
    Q = K - np.diag(1.0/tau)

A_vec, B_vec = PyGT.io.load_ktn_AB(data_path,retained)
I_vec = ~(A_vec+B_vec)
print(f'States in A,I,B: {A_vec.sum(),I_vec.sum(),B_vec.sum()}')

communities = PyGT.io.read_communities(data_path+"/communities.dat",retained,
↪screen=True)
```

(continues on next page)

(continued from previous page)

```
print('\nCommunities file identifies %d macrostates' % len(communities.keys()))
```

```
        Connected Clusters: 1, of which 95% have <= 32 states
        Retaining largest cluster with 32 nodes

States in A,I,B: (8, 16, 8)
Community 0: 8
Community 1: 8
Community 2: 8
Community 3: 8

Communities file identifies 4 macrostates
```

7.2 Remove a set of nodes in $\mathcal{I}$ using graph transformation

- We remove all nodes in $\mathcal{I} = (\mathcal{A} \cup \mathcal{B})^c$ above the 10th percentile in free energy
- See documentation of `PyGT.tools.choose_nodes_to_remove()` for other options

[3]:
```
rm_vec = PyGT.tools.choose_nodes_to_remove(rm_region=I_vec,
                                           pi=pi,
                                           tau=tau,
                                           style="free_energy",
                                           percent_retained=10
                                          )


gt_B, gt_tau, gt_K = PyGT.GT.blockGT(rm_vec,B,tau,block=10,rates=True,screen=True)
```

```
HBox(children=(FloatProgress(value=0.0, description='GT', max=11.0,␣
↪style=ProgressStyle(description_width='ini...
```

```
GT BECAME DENSE AT N=32, density=0.136719
GT done in 0.038 seconds with 0 floating point corrections
```

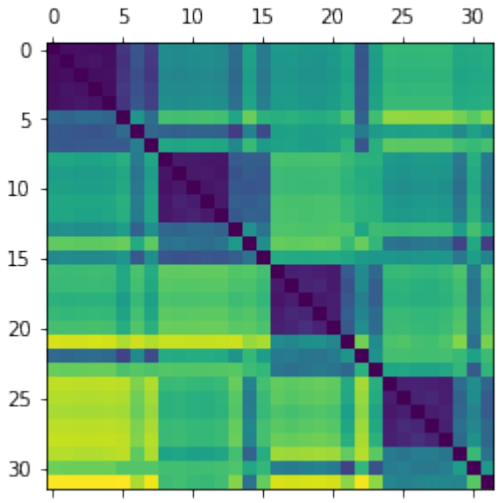
7.3 Find full MFPT matrix

Calculate the 32×32 matrix of inter-microstate mean first passage times using GT.

[4]:
```
tauM = PyGT.mfpt.full_MFPT_matrix(B,tau,screen=True)
plt.matshow(tauM)
```

```
HBox(children=(FloatProgress(value=0.0, description='MFPT matrix computation',␣
↪max=496.0, style=ProgressStyle(...
```

[4]:
```
<matplotlib.image.AxesImage at 0x7fe991f72190>
```



7.4 Find communitity MFPT matrix via full MFPT calculation or metastability approx

Reduced Boltzmann is the same in both cases

[5]:
```
#exact weighted-MFPT matrix
c_pi, c_tauM = PyGT.mfpt.community_MFPT_matrix(communities,B,tau,pi,MS_approx=False,
↪screen=True)
#approximate weighted-MFPT matrix
c_pi, c_tauM_approx = PyGT.mfpt.community_MFPT_matrix(communities,B,tau,pi,MS_
↪approx=True,screen=True)
#stationary distribution of macrostates
print(c_pi)
print(c_tauM)
#alternatively, we can compute the weighted-MFPTs from a pre-specified full inter-
↪microstate MFPT matrix
ktn = PyGT.tools.Analyze_KTN(data_path, K=Q.todense(), pi=pi, commdata='communities.
↪dat')
c_tauM_ktn = ktn.get_intercommunity_weighted_MFPTs(tauM)
print(c_tauM_ktn)
```

```
HBox(children=(FloatProgress(value=0.0, description='MFPT matrix computation',␣
↪max=496.0, style=ProgressStyle(...
```

```
HBox(children=(FloatProgress(value=0.0, description='MFPT matrix computation (MS␣
↪approx)', max=10.0, style=Pro...
```

```
[0.31641201 0.19191359 0.25727061 0.23440379]
[[  0.         463.4908166  524.60802635 707.2325798 ]
 [511.6322427    0.         675.54122209 435.79646546]
 [575.5277695  678.31953914   0.         547.48055353]
 [861.79173354 542.2141931  651.11996412   0.        ]]
[[  0.         463.4908166  524.60802635 707.2325798 ]
 [511.6322427    0.         675.54122209 435.79646546]
```

(continues on next page)

(continued from previous page)

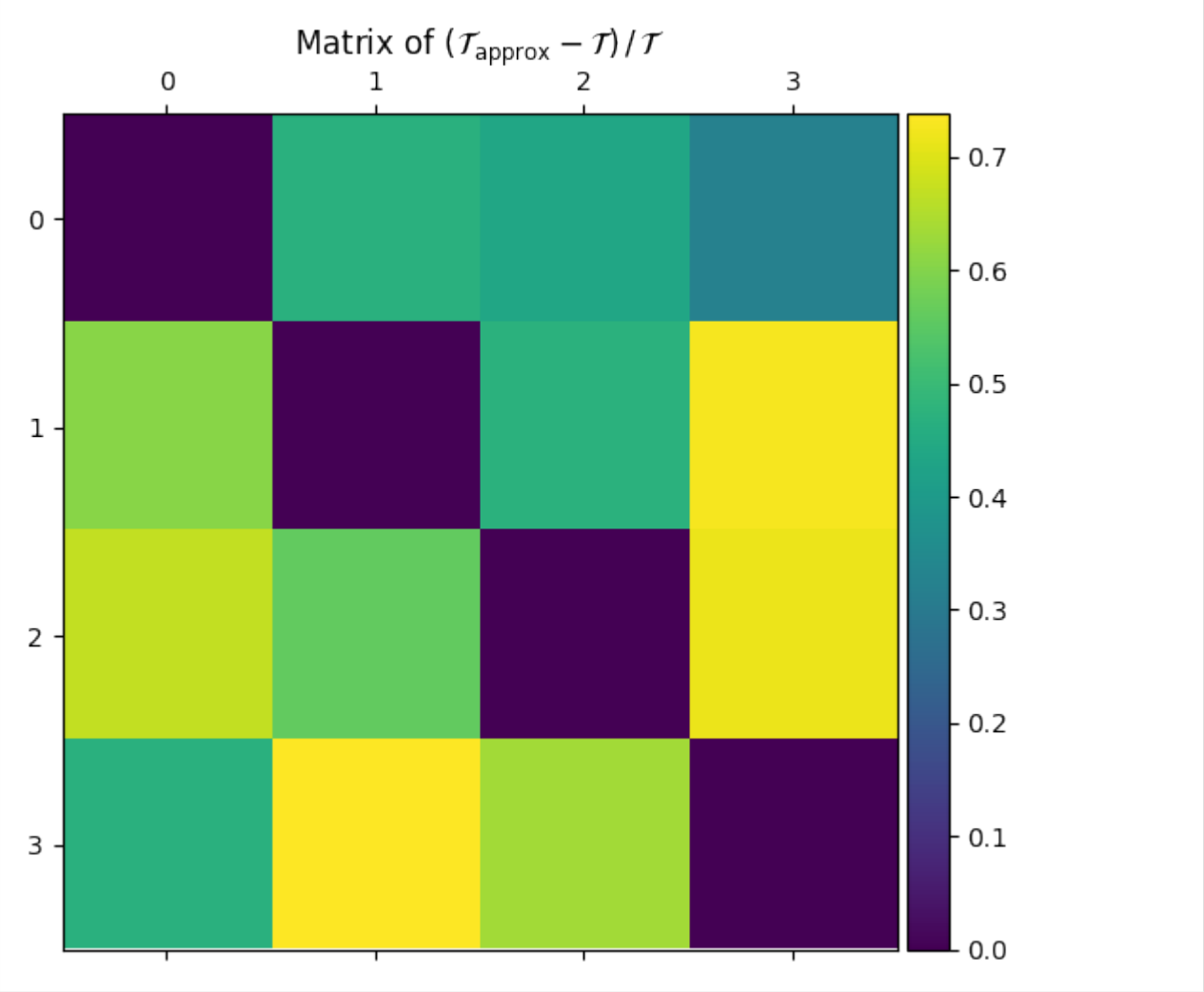
```
 [575.5277695  678.31953914   0.         547.48055353]
 [861.79173354 542.2141931  651.11996412   0.        ]]
```

7.5 Plot ratio of exact to approximate MFPT matrix

Diagonal entries will be set to zero in application, but here we set diagonal terms to unity to avoid errors when taking ratio.

[6]:
```
c_tauM += np.eye(c_pi.size) - np.diag(c_tauM) # i.e. remove diagonal and replace with␣
↪1
c_tauM_approx += np.eye(c_pi.size) - np.diag(c_tauM_approx) # i.e. remove diagonal␣
↪and replace with 1

plt.figure(figsize=(6,6),dpi=100)
plt.title(r"Matrix of $(\mathcal{T}_{\rm approx}-\mathcal{T})\,/\,\mathcal{T}$")
ax = plt.gca()
im = ax.matshow((c_tauM_approx-c_tauM)/c_tauM)
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(im, cax=cax)
```

[6]:
```
<matplotlib.colorbar.Colorbar at 0x7fe991872b20>
```



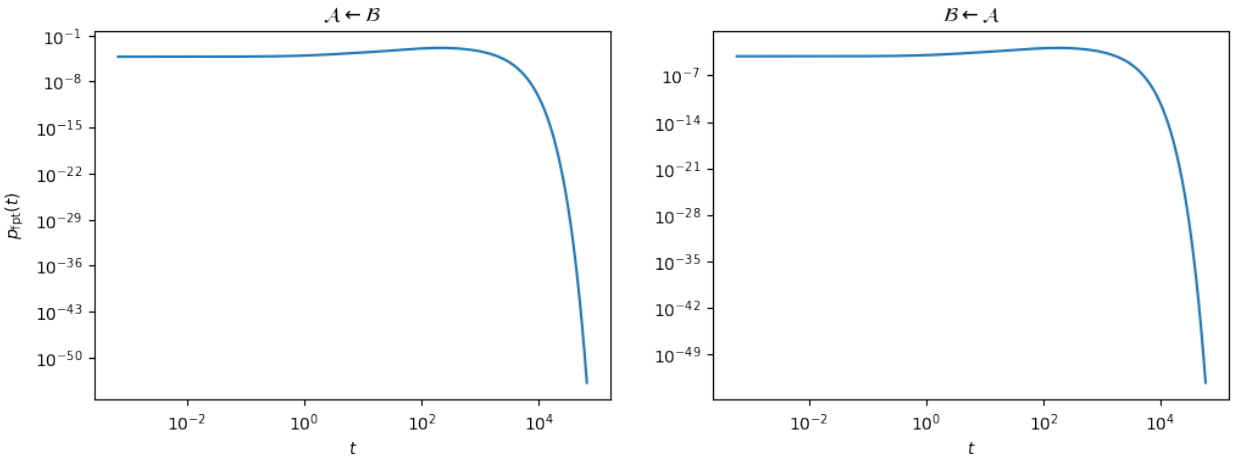
7.6 First passage time distribution between $\mathcal{A}$ and $\mathcal{B}$

[7]:
```
moments, pt = PyGT.stats.compute_passage_stats(A_vec,B_vec,pi,Q,dopdf=True,rt=np.
↪logspace(-6,2,400))

fig,axs = plt.subplots(1,2,figsize=(12,4),dpi=100)

axs[0].loglog(pt[:,0],pt[:,1]/moments[0])
axs[0].set_title(r"$\mathcal{A}\leftarrow\mathcal{B}$")
axs[0].set_ylabel(r"$p_{\rm fpt}(t)$")
axs[0].set_xlabel(r"$t$")
axs[1].loglog(pt[:,2],pt[:,3]/moments[2])
axs[1].set_title(r"$\mathcal{B}\leftarrow\mathcal{A}$")
axs[1].set_xlabel(r"$t$")

print("MFPT A<-B : ",moments[0],np.sqrt(moments[1]))
print("MFPT B<-A : ",moments[2],np.sqrt(moments[3]))
```

(continues on next page)

(continued from previous page)

```
MFPT A<-B :  664.5826837564705 878.2798938434962
MFPT B<-A :  585.7934067017948 777.2852968888889
```



7.7 MFPTs and Phenomenological Rate Constants

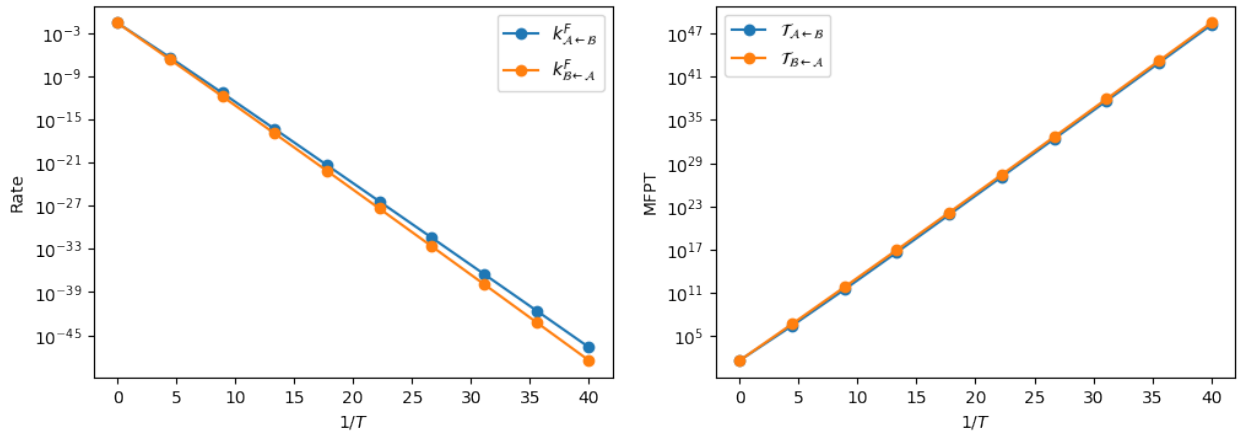
Compute the mean first passage time and rates between $\mathcal{A}$ and $\mathcal{B}$.

[8]:
```
invtemps = np.linspace(0.01, 40, 10)
data = np.zeros((4, len(invtemps)))
for i, beta in enumerate(invtemps):
    B, K, tau, N, u, s, Emin, retained = PyGT.io.load_ktn(path=data_path,beta=beta,
↪screen=False)
    F = u - s/beta # free energy
    pi = np.exp(-beta * F) / np.exp(-beta * F).sum() # stationary distribution
    rates = PyGT.stats.compute_rates(A_vec, B_vec, B, tau, pi, fullGT=False,␣
↪MFPTonly=False, screen=False)
    data[0,i] = rates['kFAB']
    data[1,i] = rates['kFBA']
    data[2,i] = rates['MFPTAB']
    data[3,i] = rates['MFPTBA']

fig, (ax, ax1) = plt.subplots(1,2, figsize=(12,4), dpi=100)
ax.plot(invtemps, data[0,:], '-o', label='$k^F_{\mathcal{A} \leftarrow \mathcal{B}}$')
ax.plot(invtemps, data[1,:], '-o', label='$k^F_{\mathcal{B} \leftarrow \mathcal{A}}$')
ax1.plot(invtemps, data[2,:], '-o', label='$\mathcal{T}_{\mathcal{A} \leftarrow␣
↪\mathcal{B}}$')
ax1.plot(invtemps,  data[3,:], '-o', label='$\mathcal{T}_{\mathcal{B} \leftarrow␣
↪\mathcal{A}}$')
ax.set_xlabel('$1/T$')
ax1.set_xlabel('$1/T$')
ax.set_ylabel('Rate')
ax1.set_ylabel('MFPT')
ax.legend()
ax1.legend()
```

(continues on next page)

(continued from previous page)

```
ax.set_yscale('log')
ax1.set_yscale('log')
```



CHAPTER 8

Coarse-graining Tutorial

We illustrate the tools avalable in the `PyGT` package for the dimensionality reduction of Markov chains using a model 32-state network. The network can be divided into 4 competing macrostates. We will compute the optimal 4×4 coarse-grained rate matrix with various numerical methods and compare the reduced dynamics to the original model.

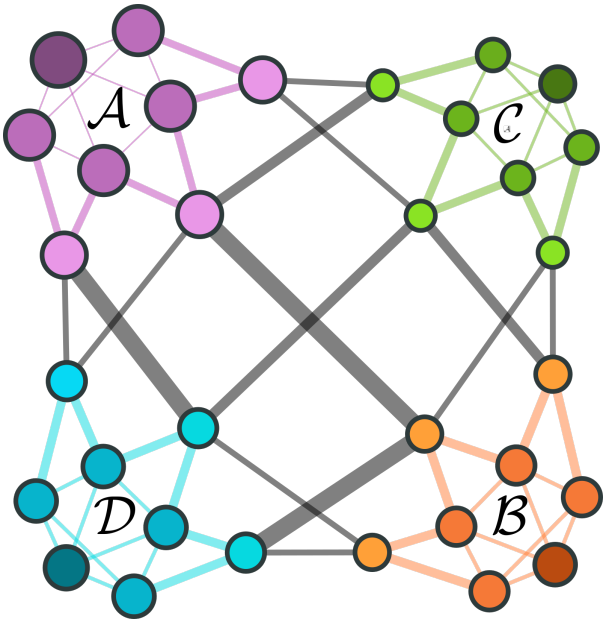
[2]:
```
#uncomment if PyGT not installed via pip
import sys; sys.path.insert(0,"../")
import PyGT
#other modules
import numpy as np
import scipy.linalg as spla
from scipy.sparse import issparse, diags
from pathlib import Path
import pandas as pd
from matplotlib import pyplot as plt
# optional
try:
    import seaborn as sns
    sns.set()
    has_seaborn=True
except:
    has_seaborn=False
```

8.1 Model 32-state network

Each community has 8 nodes, including 1 attractor node, 4 internal nodes, and 3 boundary nodes. Nodes are colored by the community to which they belong. Darker, larger nodes have higher equilbrium occupation probabilities, and thicker edges indicate slower transitions.

[3]:
```
from IPython.display import Image
Image(filename = "32state.png", width = 300)
```

[3]:



8.2 GT setup

Let's load in the Markov chain as well as its community structure. Community assignments are specified in a single-column file where each line contains the community ID of the node corresponding to the line number.

[4]:
```
data_path = Path('KTN_data/32state')
temp = 10.0
beta = 1./temp
#GT setup
B, K, tau, N, u, s, Emin, retained = PyGT.io.load_ktn(path=data_path,beta=beta)
#rate matrix with columns that sum to zero
# K has no diagonal entries
if issparse(K):
    Q = K - diags(1.0/tau)
else:
    Q = K - np.diag(1.0/tau)

BF = beta*u-s
BF -= BF.min()
#stationary distribution
pi = np.exp(-BF)
pi /= pi.sum()
#A and B sets
AS,BS = PyGT.io.load_ktn_AB(data_path,retained)
#Read in community structure
comms = PyGT.io.read_communities(data_path/'communities.dat', retained, screen=True)
for comm in comms:
    if np.all(comms[comm] == AS):
        print(f'Community A: {comm}')
    if np.all(comms[comm] == BS):
        print(f'Community B: {comm}')
```

```
Community 0: 8
Community 1: 8
Community 2: 8
Community 3: 8
Community A: 0
Community B: 3
```

8.3 Matrix of inter-microstate MFPTs with GT vs. linear algebra methods

The 32×32 matrix of inter-microstate MFPTs between all pairs of nodes can be used to obtain the optimal reduced coarse-grained Markov chain for a given community structure. Let's compute this matrix with GT and with two alternative linear algebra methods: inversion to obtain the fundamental matrix and solving a linear equation.

[5]:
```
#compute matrix of inter-microstate MFPTs with GT
mfpt_gt = PyGT.mfpt.full_MFPT_matrix(B, tau)

#check that the Kemeny constant is indeed constant
kemeny, success = PyGT.tools.check_kemeny(pi, mfpt_gt)
if success:
    print("Kemeny constant from mfpts with GT: ", kemeny)

#compute matrix of inter-microstate MFPTs with fundamental matrix
ktn = PyGT.tools.Analyze_KTN(data_path, K=Q.todense(), pi=pi, commdata='communities.
↪dat')
mfpt_fund = ktn.get_intermicrostate_mfpts_fundamental_matrix()
kemeny_fund, success = PyGT.tools.check_kemeny(pi, mfpt_fund)
if success:
    print("Kemeny constant from mfpts with fundamental matrix: ", kemeny_fund)

#compute matrix of inter-microstate MFPTs by solving a linear equation
mfpt_lin = ktn.get_intermicrostate_mfpts_linear_solve()
kemeny_lin, success = PyGT.tools.check_kemeny(pi, mfpt_lin)
if success:
    print("Kemeny constant from mfpts with linear solve: ", kemeny_lin)
```

```
Kemeny constant from mfpts with GT:  97.53002029348983
Kemeny constant from mfpts with fundamental matrix:  97.53002029348986
Kemeny constant from mfpts with linear solve:  97.53002029348977
```

8.4 Compute inter-community weighted-MFPTs

[6]:
```
#compute weighted-MFPT between communities
commpi = ktn.get_comm_stat_probs(np.log(pi), log=False)
ktn.commpi = commpi
ncomms = len(commpi)
pt = ktn.get_intercommunity_weighted_MFPTs(mfpt_gt)
#Kemeny constant of reduced Markov chain
print("Weighted-MFPT matrix:")
print(pt)
c_kemeny, success = PyGT.tools.check_kemeny(commpi, pt)
```

(continues on next page)

(continued from previous page)

```
if success:
    print('\nKemeny constant of coarse-grained Markov chain: ', c_kemeny)
```

```
Weighted-MFPT matrix:
[[ 0.         53.8206528  54.95852132 69.11554233]
 [53.00851035  0.         66.57240778 52.25954611]
 [54.71235842 67.13838734  0.         54.16914917]
 [69.93796797 53.8941142  55.2377377   0.        ]]

Kemeny constant of coarse-grained Markov chain:  44.06002077112634
```

8.5 Different routes to obtain the optimal coarse-grained CTMC

In Kannan et al. *J. Chem. Phys.* (2020), we discuss three different expression for the optimal coarse-grained rate matrix given a partitioning of the V nodes in the original Markov chain into N communities: the HS relation, the KKRA relation, and an expression obtained from inverting the matrix of weighted-MFPTs. We illustrate the computation of all 3 methods below:

[7]:
```
""" Three different version of the optimal reduced CTMC."""
#1) the original HS relation
K_hs = ktn.construct_coarse_rate_matrix_Hummer_Szabo()
#2) the KKRA relation involving inversion of matrix of inter-microstate mfpts
K_kkra = ktn.construct_coarse_rate_matrix_KKRA(GT=True)
#3) based on inversion of weighted-MFPTs
K_invert = spla.inv(pt)@(np.diag(1./commpi) - np.ones((ncomms,ncomms)))

print('Optimal reduced CTMC from Hummer-Szabo relation:')
print(K_hs)
print('Optimal reduced CTMC from KKRA relation:')
print(K_kkra)
print('Optimal reduced CTMC from inversion of weighted-MFPT matrix:')
print(K_invert)

#check that detailed balance is satisfied
if not PyGT.tools.check_detailed_balance(commpi, K_invert):
    print('Detailed balance not satisfied for K_C.')
if not PyGT.tools.check_detailed_balance(pi, Q):
    print('Detailed balance not satisfied for K')
```

```
Optimal reduced CTMC from Hummer-Szabo relation:
[[-0.05239043  0.02620963  0.02440272  0.0036629 ]
 [ 0.02493137 -0.05761829  0.00478585  0.02595583]
 [ 0.02390442  0.00492849 -0.05366994  0.0247118 ]
 [ 0.00355464  0.02648017  0.02448137 -0.05433052]]
Optimal reduced CTMC from KKRA relation:
[[-0.05239043  0.02620963  0.02440272  0.0036629 ]
 [ 0.02493137 -0.05761829  0.00478585  0.02595583]
 [ 0.02390442  0.00492849 -0.05366994  0.0247118 ]
 [ 0.00355464  0.02648017  0.02448137 -0.05433052]]
Optimal reduced CTMC from inversion of weighted-MFPT matrix:
[[-0.05239043  0.02620963  0.02440272  0.0036629 ]
 [ 0.02493137 -0.05761829  0.00478585  0.02595583]
 [ 0.02390442  0.00492849 -0.05366994  0.0247118 ]
 [ 0.00355464  0.02648017  0.02448137 -0.05433052]]
```

8.6 Numerical comparison of coarse-grained Markov chains

To compare the numerical stability of these various routes to obtain the optimal reduced CTMC, let's compute the mean first passage times $\mathcal{A} \leftrightarrow \mathcal{B}$ on the original network and compare it to the corresponding observables on the various reduced networks.

[8]:
```
def compare_HS_LEA(temps, data_path):
    """ Calculate coarse-grained rate matrices using the 3 versions of the optimal
    reudced Markov chain and the local equilibrium approximation (LEA).
    Compute MFPTAB/BA on the full and coarse-grained networks. """

    dfs = []
    for temp in temps:
        df = pd.DataFrame()
        df['T'] = [temp]
        #KTN input
        beta = 1./temp
        B, K, tau, N, u, s, Emin, retained = PyGT.io.load_ktn(path=data_path,
↪beta=beta)
        Q = K - diags(1.0/tau)
        BF = beta*u-s
        BF -= BF.min()
        #stationary distribution
        pi = np.exp(-BF)
        pi /= pi.sum()
        #A and B sets
        AS,BS = PyGT.io.load_ktn_AB(data_path,retained)
        #ktn setup
        ktn = PyGT.tools.Analyze_KTN(data_path, K=Q, pi=pi, commdata='communities.dat
↪')
        commpi = ktn.commpi
        ncomms = len(ktn.commpi)
        #MFPT calculations on full network
        full_df = PyGT.stats.compute_rates(AS, BS, B, tau, pi, fullGT=True, block=1)
        df['MFPTAB'] = full_df['MFPTAB']
        df['MFPTBA'] = full_df['MFPTBA']

        #compute coarse-grained networks
        mfpt = PyGT.mfpt.full_MFPT_matrix(B, tau)
        pt = ktn.get_intercommunity_weighted_MFPTs(mfpt)
        labels = []
        matrices = []
        try:
            Rhs = ktn.construct_coarse_rate_matrix_Hummer_Szabo()
            matrices.append(Rhs)
            labels.append('HS')
        except Exception as e:
            print(f'HS had the following error: {e}')
        try:
            Rhs_kkra = ktn.construct_coarse_rate_matrix_KKRA(mfpt=mfpt)
            matrices.append(Rhs_kkra)
            labels.append('KKRA')
        except Exception as e:
            print(f'KKRA had the following error: {e}')
        try:
            Rhs_invert = spla.inv(pt)@(np.diag(1./commpi) - np.ones((ncomms,ncomms)))
            matrices.append(Rhs_invert)
```

(continues on next page)

(continued from previous page)

```
            labels.append('PTinvert_GT')
        except Exception as e:
            print(f'Inversion of weighted-MFPTs from GT had the following error: {e}')
        try:
            Rlea = ktn.construct_coarse_rate_matrix_LEA()
            matrices.append(Rlea)
            labels.append('LEA')
        except Exception as e:
            print(f'LEA had the following error: {e}')

        if len(matrices)==0:
            continue

        for i, R in enumerate(matrices):
            """ get A->B and B->A mfpt on coarse network"""
            rK = R - np.diag(np.diag(R))
            escape_rates = -1*np.diag(R)
            B = rK@np.diag(1./escape_rates)
            tau = 1./escape_rates
            #B, tau = PyGT.tools.load_CTMC(R)
            Acomm = 0
            Bcomm = 3
            MFPTAB, MFPTBA = PyGT.mfpt.compute_MFPT(Acomm, Bcomm, B, tau, block=1)
            df[f'AB_{labels[i]}'] = [MFPTAB]
            df[f'BA_{labels[i]}'] = [MFPTBA]
        dfs.append(df)
    bigdf = pd.concat(dfs, ignore_index=True, sort=False)
    return bigdf
```

8.7 Plot KKRA, H-S against exact, LEA and GT systems at high temperature

[9]:
```
#some mid temperature calculations
invtemps = np.linspace(0.1, 4, 6)
midtemp_df = compare_HS_LEA(1./invtemps, data_path)
```

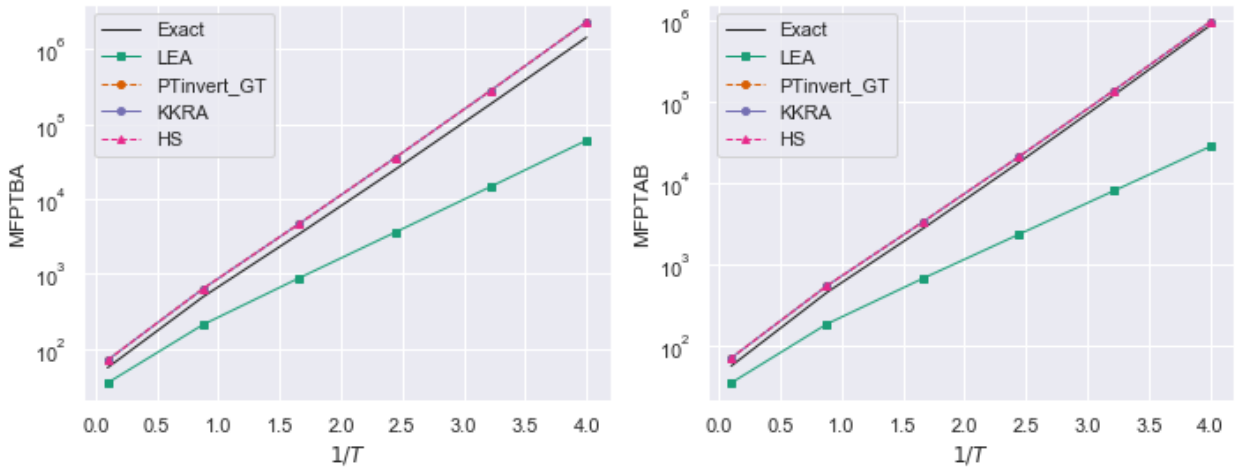
[10]:
```
def plot_mfpts_32state(df):
    """Plot MFPTs computed on coarse-grained networks against true MFPT from full␣
↪network."""
    if has_seaborn:
        colors = sns.color_palette("Dark2", 4)
    else:
        colors = ['C0','C1','C2','C3']
    df.replace([np.inf, -np.inf], np.nan)
    df2= df.sort_values('T')
    symbols = ['-s', '--o', '-o', '--^']
    rates = ['LEA', 'PTinvert_GT', 'KKRA', 'HS']
    labels = rates
    denom = 'MFPT'
    #first plot A<-B direction
    fig, (ax, ax2) = plt.subplots(1, 2, figsize=[10, 4])
    ax.plot(1./df2['T'], df2['MFPTBA'], '-', color='k', label='Exact', lw=1,␣
↪markersize=4)
```

(continues on next page)

(continued from previous page)

```
    for j, CG in enumerate(rates):
        #then only plot HSK for temperatures that are not NaN
        df2CG = df2[-df2[f'BA_{CG}'].isna()]
        ax.plot(1./df2CG['T'], df2CG[f'BA_{CG}'],
                symbols[j], label=labels[j], color=colors[j], linewidth=1,
                markersize=4)
    ax.set_xlabel(r'$1/T$')
    ax.set_yscale('log')
    ax.set_ylabel('MFPTBA')
    ax.legend(frameon=True)
    ax2.plot(1./df2['T'], df2['MFPTAB'], '-', color='k', label='Exact', lw=1,␣
↪markersize=4)
    for j, CG in enumerate(rates):
        #then only plot HSK for temperatures that are not NaN
        df2CG = df2[-df2[f'AB_{CG}'].isna()]
        ax2.plot(1./df2CG['T'], df2CG[f'AB_{CG}'],
                symbols[j], label=labels[j], color=colors[j], linewidth=1,
                markersize=4)
    ax2.set_xlabel(r'$1/T$')
    ax2.set_ylabel('MFPTAB')
    ax2.set_yscale('log')
    ax2.legend(frameon=True)
    fig.tight_layout()
```

[11]:
```
plot_mfpts_32state(midtemp_df)
```

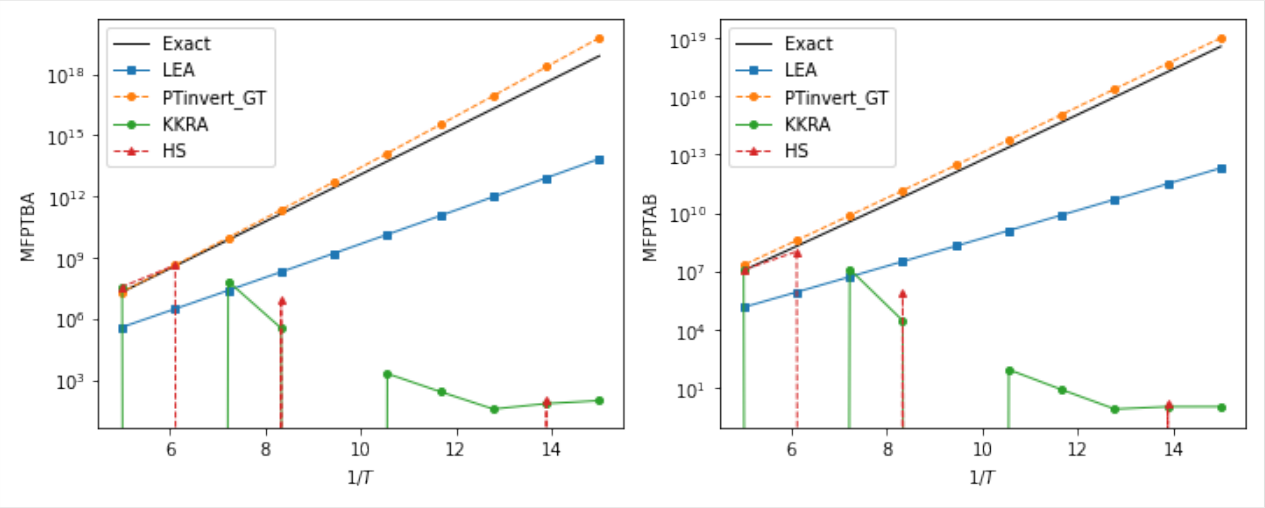


8.8 Plot KKRA, H-S against exact, LEA and GT systems at slightly lower temperature

KKRA, H-S fails

[14]:
```
invtemps = np.linspace(5, 15, 10)
lowtemp_df = compare_HS_LEA(1./invtemps, data_path)
```

[16]:
```
plot_mfpts_32state(lowtemp_df)
```


Exact
LEA
PTinvert_GT
KKRA
HS
MFPTBA
1/T
MFPTAB
1/T

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[Wales09] D.J. Wales, *Calculating rate constants and committor probabilities for transition networks by graph transformation*, J. Chemical Physics (2009), https://doi.org/10.1063/1.3133782

[Swinburne20a] T.D. Swinburne and D.J. Wales, *Defining, Calculating, and Converging Observables of a Kinetic Transition Network*, J. Chemical Theory and Computation (2020), https://doi.org/10.1021/acs.jctc.9b01211

[Swinburne20b] T.D. Swinburne, D. Kannan, D.J. Sharpe and D.J. Wales, *Rare Events and First Passage Time Statistics From the Energy Landscape*, Submitted to J. Chemical Physics (2020)

[Kannan20a] D. Kannan, D.J. Sharpe, T.D. Swinburne and D.J. Wales, *Dimensionality reduction of Markov chains using mean first passage times with graph transformation*, In Prep. (2020)

[Kannan20b] D. Kannan, D.J. Sharpe, T.D. Swinburne and D.J. Wales, *Dimensionality reduction of complex networks with graph transformation*, In Prep. (2020)

Python Module Index

p

Index

S