GraphHomogenization README

Preston Donovan

Contents

1 Overview

This is a README for the GraphHomogenization MATLAB package, a tool that numerically computes the effective diffusivity matrix D_e of a periodic, directed, weighted graph $(\mathcal{S}, \mathcal{E}, \lambda)$. These notes assume an understanding of Chapter 4 of my thesis. Functions in `thisFont` typically refer to MATLAB functions.

1.1 Technical Background

Under certain assumptions, a periodic, directed, weighted graph $(\mathcal{S}, \mathcal{E}, \lambda)$ induces a continuous-time Markov process $Z(t) \in \mathcal{S}$ with generator

$$Lf(x) = \sum_{(x,y)\in\bar{\mathcal{E}}} \big(f(y) - f(x)\big)\lambda(x, y), \tag{1}$$

where $\bar{\mathcal{E}}$ is the quotient edge set. The quotient node set is $\bar{\mathcal{S}}$. The scaled process $\varepsilon Z(t/\varepsilon^2)$ converges weakly in Skorokhod space to a Brownian motion $B(t)$ where

$$\mathbb{E}[B(t)B(t)^T] = 2D_e t. \tag{2}$$

The effective diffusivity matrix is given by

$$D_e = \frac{1}{2}\sum_{y\in\bar{\mathcal{S}}}\sum_{e\in\bar{\mathcal{E}}_y} \Big(\nu_e\nu_e^T\lambda_e\pi(y) - \nu_e\omega(y)^T\lambda_e - \omega(y)\nu_e^T\lambda_e\Big). \tag{3}$$

Here, ν_e and λ_e denote the jump size and jump rate, respectively, of an edge e. The stationary distribution π satisfies

$$L^T\pi = 0 \tag{4}$$

and ω is the solution to the unit-cell problem

$$L^T\omega = \sigma. \tag{5}$$

1.2 Workflow

To numerically computing D_e, one must:

I Calculate the rate matrix L, the quotient node set $\bar{\mathcal{S}}$, the quotient edge set $\bar{\mathcal{E}}$, the jump rates $\{\lambda_e\}_{e\in\bar{\mathcal{E}}}$, and the jump sizes $\{\nu_e\}_{e\in\bar{\mathcal{E}}}$.

II Solve $L^T\pi = 0$.

III Solve $L^T\omega = \sigma$.

Conceptually, the items in step I are clear and the values of π and ω may be less intuitive. However, computing the former is significantly more involved in terms of lines of code. The linear solves in steps II and III can each be performed in a single line in MATLAB. Thus, much of the code I developed aids in completing step I. The development of this project is motivated by the application of a random walk on a subset of the integer lattice $\mathbb{Z}^d$. Thus, for graphs whose node set is embedded in $h\mathbb{Z}^d$, there are tools in place to aid in completing step I.

Strictly speaking, not all of the items in step I need to be calculated (and some information is technically redundant), but each plays an important role in facilitating the calculation of D_e. Certain functions can probably be improved to alleviate this redundancy.

Remark: this tool assumes $\bar{\mathcal{E}}$ can be identified with $(\Pi \times \Pi)(\mathcal{E})$. This assumption is not satisfied for most graphs satisfying $\bar{\mathcal{S}} \subset \frac{1}{2}\mathbb{Z}^d$. For this case, we have a special driver with hard-coded fixes.

2 The `LatticeGeometry` Class

The `LatticeGeometry` class is a central variable of the tool; it is an object with various fields that characterize a lattice graph. The user must specify a `LatticeGeometry` object if step I is not complete (i.e., if L, $\bar{\mathcal{S}}$, $\bar{\mathcal{E}}$, $\{\lambda_e\}_{e\in\bar{\mathcal{E}}}$, and $\{\nu_e\}_{e\in\bar{\mathcal{E}}}$ are not yet defined). Table 1 lists the fields of `LatticeGeometry`, a brief description, and the possible values. At a minimum, the user must specify `dim`, `m`, `name`, and `obRad`. Anything else that is not specified will be set to a natural default value.

If the jump rate function λ is constant, then setting up a `LatticeGeometry` object is quite simple. The fields `specialSetting`, `driftMult`, `driftDecay`, `obSlowdownFctr`, and `bdyDist` are only potentially necessary if the jump rate function is meant to incorporated a drift of model an interaction between the random walk and the obstruction.

Remark: It is natural to ask why the `LatticeGeometry` class does not simply have a field that stores the rate function as a function handle, rather than the numerous fields currently present. This is a reasonable alternative but comes with a downside: if the function handle depends on temporary workspace variables or a .m file that is changing/updated over time, then reproducing old results can be a headache. By characterizing the rate function via a set of fields and a single consistent .m file (`rate_lattice.m`), reproducing old results is much more reliable.

The function `validate` determines whether or not a `LatticeGeometry` object is valid. For example, this function checks that the dimension is 2 or 3. We now provide in-depth descriptions of the more complicated fields of the `LatticeGeometry` class.

2.1 `diagJumps`

A "diagonal jump" refers to any edge e where $|\nu_e| = (h,h)$ (in $2D$) or $|\nu_e| = (h,h,h)$ (in $3D$). If only jumps along the standard basis vectors are desired, set `diagJumps` $= 0$. Otherwise, set `diagJumps` $= 1$ to incorporate diagonal jumps in a naive and straightforward manner.

Field	Description	Values
dim	Dimension	2 or 3
m	Number of possible nodes in period cell along each dimension	Integer ≥ 3. May be 2 in some cases.
name	Obstruction geometry	'square' or 'circle'
obRad	Radius of obstruction (half side length if square)	$[0, 1)$ if dim $= 2$, $[0, \sqrt{2})$ if dim $= 3$
obCtr	Center of obstruction	$[0, 1]^{\texttt{dim}}$
diagJumps	0 if diagonal jumps are not allowed, 1 if diagonal jumps are allowed, and 2 if the diagonal jumps should have "corrected" jump rates.	0, 1, or 2
specialSetting	Specifies if a special rate function should be used	'none', 'slowdown', 'bdyBonding', 'bdyRepel', 'bdyAttract', 'bdySlow', 'm2_blockOneSite', 'm2_slowOneSite'
driftMult	K_1 in (7)	Real number
driftDecay	K_2 in (7)	Positive real number. Only specify if $K_1 > 0$.
obSlowdownFctr	α in (8), (10), (11), (12)	Positive real number. Only specify if specialSetting = 'slowdown','bdyBonding', 'bdyRepel', 'bdyAttract', 'bdySlow', or 'm2_slowOneSite'
bdyDist	δ in (9) (distance from obstruction at which the bonding, repulsion, attraction, etc. takes place)	$[0, 1]$, only specify if specialSetting = 'bdy*'
h	1/m (the mesh size)	Automatically set
sideLen	2obRad if name = 'square'	Automatically set.
isValid	Specifies if the LatticeGeometry object is a valid object.	Automatically set when validate is called. 0 or 1.

Table 1: Description of fields of LatticeGeometry class. String fields are not case sensitive.

The downside to this setting is that the diagonal jump rates may not be realistic. For example, if the random walker is near the obstruction boundary (its distance to the obstruction is less than h) and it attempts a diagonal jump to a site that is obstructed, the jump would have resulted in a displacement. Setting $\texttt{diagJumps} = 2$ accounts for this but is only implemented for the case when $\texttt{dim} = 2$ and $\texttt{name} =$ 'square'.

2.2 specialSetting

This field determines the functional form of the rate function λ.

Case $\texttt{specialSetting}$ = 'none': In this case,

$$\lambda(x, y) = \frac{D_0}{h^2} + \frac{\mu(x)^T \nu_e}{2h}, \tag{6}$$

where $D_0 = 1$ and $\mu : \bar{\mathcal{S}} \to \mathbb{R}^{\texttt{dim}}$ is the force field,

$$\mu(x) = \frac{K_1}{\exp\big(K_2(||x - \texttt{obCtr}|| - \texttt{obRad})\big)} \cdot \frac{x - \texttt{obCtr}}{||x - \texttt{obCtr}||}. \tag{7}$$

If $K_1 > 0$ (i.e., a drift is present), then one must set $\texttt{name} =$ 'circle' and $\texttt{diagJumps} = 0$. The code can easily be extended to accommodate the case when $\texttt{name} =$ 'square'.

Case $\texttt{specialSetting}$ = 'slowdown': This setting allows modeling a permeable obstruction $\mathcal{O}$, in which the random walker has a different jump rate. This is the only setting wherein the node set is $\mathcal{S} = h\mathbb{Z}^{\texttt{dim}}$ (i.e., nodes in $\mathcal{O}$ are not removed). The jump rate is given by

$$\lambda(x, y) = \begin{cases} 1/h^2 & x \notin \mathcal{O} \\ \alpha/h^2 & x \in \mathcal{O}, \end{cases} \tag{8}$$

where $\alpha = \texttt{obSlowdownFctr}$.

Case $\texttt{specialSetting}$ = 'bdyBonding', 'bdyRepel', or 'bdyAttract': In each of these settings, all jump rates along edges that originate near the obstruction boundary are modified in some way. Intuitively, 'bdyBonding', 'bdyRepel', and 'bdyAttract' model a bonding, repulsion, and attraction effect between the obstruction and random walker.

Define the set of nodes within a distance δ of the obstruction by

$$\mathcal{B}_\delta = \{x \in \bar{\mathcal{S}} \mid d(x, \mathcal{O}) < \delta\} \tag{9}$$

where $\delta \coloneqq \texttt{bdyDist}$ and the distance between a node and the obstruction, $d(x, \mathcal{O})$, is defined in the obvious way.

In the 'bdyBonding' case, the user specifies `bdyDist` $\in [0,1]$ and the jump rate function is given by

$$\lambda(x,y) = \begin{cases} \alpha/h^2 & x \in \mathcal{B}_\delta \\ 1/h^2 & x \notin \mathcal{B}_\delta. \end{cases} \tag{10}$$

This rate function slows the random walker whenever it is near the boundary of an obstruction.

In the 'bdyRepel' case, we impose `bdyDist` $= h$ and define

$$\lambda(x,y) = \begin{cases} 1/(\alpha h^2) & x \in \mathcal{B}_\delta, y \notin \mathcal{B}_\delta \\ \alpha/h^2 & x \notin \mathcal{B}_\delta, y \in \mathcal{B}_\delta \\ 1/h^2 & \text{otherwise}, \end{cases} \tag{11}$$

which causes the random walker to be pushed away from the obstructions.

In the 'bdyAttract' case, we impose `bdyDist` $= h$ and define

$$\lambda(x,y) = \begin{cases} \alpha/h^2 & x \in \mathcal{B}_\delta, y \notin \mathcal{B}_\delta \\ 1/(\alpha h^2) & x \notin \mathcal{B}_\delta, y \in \mathcal{B}_\delta \\ 1/h^2 & \text{otherwise}. \end{cases} \tag{12}$$

This rate function pulls the random walker towards the obstructions.

Case `specialSetting` = 'm2_blockOneSite' or 'm2_slowOneSite': Must use the driver `driver_2x2` in these cases. Internally, most of the code assumes that $\bar{\mathcal{E}}$ can be identified with $(\Pi \times \Pi)(\mathcal{E})$. When `m` $= 2$, this is not the case and thus should be avoided if possible. Currently, there is a hotfix in place when `m` $= 2$ and `dim` $= 2$. In this case, the user *must* set `specialSetting` = 'm2_blockOneSite' or 'm2_slowOneSite'.

When `specialSetting` = 'm2_blockOneSite', all edges originating or ending in the node $(2/3, 2/3)$ are removed. When `specialSetting` = 'm2_slowOneSite', all edges originating or ending in the node $(2/3, 2/3)$ have their rates scaled by `obSlowdownFctr`.

Case `specialSetting` = 'bdySlow': Must use the driver `driver_nodesAtBdy` in this case. This is an experimental setting wherein nodes are placed at the boundary of an obstruction. Rates are doubled along edges that start at the boundary, do not start at a corner, and do not end at the boundary.

2.3 `driftMult` and `driftDecay`

These two fields are only relevant when `specialSetting` = 'none'. `driftMult` and `driftDecay` are equal to K_1 and K_2 in (7), respectively. Conceptually, `driftMult` controls the strength of the drift field and whether it points towards or away from the obstruction center. If `driftMult` > 0, then the drift field will point away from the obstruction. Clearly, the magnitude of the drift decreases as one moves away from the obstruction. As `driftDecay` increases, this rate of decay increases.

2.4 `obSlowdownFctr`

A parameter related to the rate function λ when `specialSetting` = 'slowdown', 'bdyBonding', 'bdyRepel', 'bdyAttract', or 'm2_slowOneSite'. Specifically, $\alpha =$ `obSlowdownFctr` in (8), (10), (11), (12).

2.5 `bdyDist`

A parameter that determines which jump rates are modified when `specialSetting` = 'bdyBonding'. Specifically, $\delta =$ `bdyDist` in (10).

3 Function Descriptions

3.1 Root

A set of drivers that the user can modify and run. Ideally, the user only creates and modifies files in this directory, which should solely consist of drivers that call functions in the subdirectories. A driver should typically do the following (assuming the graph is a lattice graph):

1. Pass the properties of the graph (e.g., d, m, obstruction radius, etc.) to `LatticeGeometry` to create a `LatticeGeometry` object.
2. Pass the `LatticeGeometry` object to `homogInputs_lattice` to create the rate matrix L, the quotient node set $\bar{\mathcal{S}}$, the quotient edge set $\bar{\mathcal{E}}$, the jump rates $\{\lambda_e\}_{e\in\bar{\mathcal{E}}}$, and the jump sizes $\{\nu_e\}_{e\in\bar{\mathcal{E}}}$.
3. Pass L, $\bar{\mathcal{S}}$, $\bar{\mathcal{E}}$, $\{\lambda_e\}_{e\in\bar{\mathcal{E}}}$, $\{\nu_e\}_{e\in\bar{\mathcal{E}}}$, and the `LatticeGeometry` object to `effDiff` to compute the effective diffusivity D_e.
   (a) To estimate D_e via Monte Carlo simulation, also pass the number of trajectories and trajectory starting locations to `effDiff`.
4. (Optional) Plot the results.
5. (Optional) Save the results.

3.2 HomogTools/

Modifying code in this directory may result in changes to the calculation of D_e. The functions in HomogTools/ essentially perform steps II and III. That is, L, $\bar{\mathcal{S}}$, $\bar{\mathcal{E}}$, λ_e, and ν_e must already be computed to use any of these functions. These five items from this step are passed to `effDiff_homog`, which proceeds as follows:

1. Call `LUFull` to compute the LU factorization.
2. Call `statDist` to compute π (4).
3. Call `unitCell` to compute ω (5).
4. Call `buildEffDiff` to compute D_e (3).

There are two other functions in HomogTools/. The function `effDiff_mc` approximates D_e via Monte Carlo simulation and `effDiff` is a simple wrapper for calling `effDiff_homog` and `effDiff_mc`.

- `buildEffDiff.m` Performs that actual computation of the effective diffusivity matrix as in (3) once the stationary distribution and unit-cell solute have been computed.
- `effDiff.m` Wrapper function that calls `effDiff_homog` and `effDiff_mc`.
- `effDiff_homog.m` Computes the effective diffusivity from the rate matrix, graph's nodes, graph's edges, edge weights, and edge jumps. Calls `LUFull`, `statDist`, `unitCell`, and `buildDeff`.
- `effDiff_mc.m` Approximates the effective diffusivity via Monte Carlo simulation.
- `LUFull.m` Computes the full LU factorization (includes permutation matrices and diagonal scaling matrix).
- `statDist.m` Computes the stationary distribution.
- `unitCell.m` Solves the unit-cell problem.

3.3 LatticeTools/

Modifying code in this directory may result in changes to the calculation of D_e. A set of functions for setting up the graph's node set, edge set, and edge weights assuming the graph satisfies certain geometric conditions.

LatticeTools/ contains four functions that aid in setting up the necessary inputs of `effDiff_homog`, `effDiff_mc`, and `effDiff`: L, $\bar{\mathcal{S}}$, $\bar{\mathcal{E}}$, λ_e, and ν_e. However, these functions only apply to a specific graph setting, which we call a *lattice graph*.

For any fixed $h > 0$, a lattice graph is any graph $(\mathcal{S}, \mathcal{E}, \lambda)$ satisfying our standard assumptions in addition to the following:

1. $\mathcal{S} = h\mathbb{Z}^d \backslash \mathcal{O}$ where $d = 2$ or 3 and $\mathcal{O}$ (the "obstructed region") consists of a periodically repeated square or circle (when $d = 2$) or cube or sphere (when $d = 3$),
2. $\mathcal{E}$ consists of all pairs of nodes and their nearest $2d$ neighbors or their nearest $3d$ neighbors (i.e., diagonal jumps are included).

The function `homogInputs_lattice` works in conjunction with `rate_lattice` to generate the five inputs. Various rate functions are allowed.

- `getNodes_lattice.m` Calculates the set of free nodes from a LatticeGeometry object.
- `homogInputs_lattice.m` Sets up the rate matrix, node set, edge set, edge weights, and edge jumps of a `LatticeGeometry` object. Calls `getNodes_lattice`.
- `LatticeGeometry.m` A class that holds the defining features of a lattice geometry. Can also be used to check that a lattice geometry is valid.
- `rate_lattice.m` Computes the rate of an edge given a `LatticeGeometry` object.

3.4 PlottingTools/

This directory contains a set of functions for drawing a periodic cell of a graph, plotting the effective diffusivity coefficients, and drawing the drift field of a rate function (if present).

- `drawCell.m` Draws a periodic cell of the graph.
- `drawCell_lattice.m` Draws a periodic cell of the graph assuming the graph satisfies certain geometric conditions.
- `drawDriftField.m` Draws a vector field based on the drift function.
- `plotObRadVsEffDiff.m` Plots the effective diffusivities computed by homogenization theory and Monte Carlo simulation against obstruction radius.

3.5 MiscTools/

Functions that don't fit elsewhere are stored here. The function `saveResults` generates an appropriate file name and saves the homogenization theory and Monte Carlo calculations. The function `checkDetailedBalance` determines whether a graph satisfies the detailed balance condition.

- `checkDetailedBalance.m` Determines whether a graph satisfies the detailed balance condition.
- `diagnostics.m` A script for ensuring that code changes do not lead to different/inaccurate results.
- `saveResults.m` Saves a results object.

3.6 MiscDrivers/

Some drivers specific to my research.