User Guide to Matlab Toolbox
"Optimal Transport Networks in Spatial Equilibrium"

Pablo Fajgelbaum   Dorian Henricot   Cristiano Mantovani
Edouard Schaal

October 15, 2019
version 1.0.4

Contents

1 Introduction **3**

2 Problem formulation **3**
2.1 Immobile labor, no cross-good congestion . . . 3
2.2 Immobile labor, cross-good congestion . . . 4
2.3 Mobile labor, no cross-good congestion . . . 4
2.4 Mobile labor, cross-good congestion . . . 5
2.5 Partial mobility case . . . 5

3 Installation **5**
3.1 Toolbox . . . 5
3.2 IPOPT . . . 5
3.3 ADiGator (optional) . . . 6

4 Examples **6**
4.1 Running the examples . . . 6
4.2 Step-by-step example . . . 6

5 Data structures **9**
5.1 Model parameter structure `param` . . . 9
5.2 Graph structure `graph` . . . 9
5.3 Results structure `res` . . . 9
5.4 Geography structure `geography` . . . 9

6 Functions **11**
6.1 Function `init_parameters()` . . . 11
6.2 Function `create_graph()` . . . 12
6.3 Function `optimal_network()` . . . 14
6.4 Function `annealing()` . . . 16
6.5 Function `solve_allocation()` . . . 18

6.6 Function call_adigator() 19
6.7 Function plot_graph() 19
6.8 Function add_node() 21
6.9 Function remove_node() 22
6.10 Function find_node() 22
6.11 Function apply_geography() 23

7 Comments on the resolution method **23**
7.1 Overview of the algorithm 23
7.2 The six model cases 25
7.3 Custom model 26

8 Known bugs and issues **27**

9 Release Notes **28**

1 Introduction

In this document, we explain the code implementation of the model described in Fajgelbaum and Schaal [2019]. Please refer to the paper for additional details on the model.

For a given graph (set of nodes and edges), with a given endowment (productivity, non-tradable good, and population per node in the immobile case), and a given set of parameters, the code returns the optimal network, flow, and allocation, with a graphical representation of it.

The document is organized as follows. In Section 2, we remind the reader about the model and problem formulation that the toolbox is designed to solve. Section 3 provides installation steps. In Section 4, we explain how to get quickly started and run the examples. In Section 5, we explain the main data structures used in the code. In Section 6, we provide a reference on how to use the various key functions. In Section 7, we provide a few additional information on the resolution method and instructions to solve a custom model. Section 8 concludes with some known issues and bugs.

2 Problem formulation

2.1 Immobile labor, no cross-good congestion

The social planning problem in the case of no labor mobility and no cross-good congestion is

$$\max_{c_j,h_j,C_j^n,L_j^n,Q_{jk}^n,I_{jk}} \sum_j \omega_j L_j U\left(c_j,h_j\right) \tag{1}$$

with $U\left(c,h\right) = \left(c^\alpha h^{1-\alpha}\right)^{1-\rho} / \left(1-\rho\right), \alpha \in \left[0,1\right], \rho > 1$, subject to the availability of traded goods

$$c_j L_j \leqslant \left[\sum_{n=1}^{N} \left(C_j^n\right)^{\frac{\sigma-1}{\sigma}}\right]^{\frac{\sigma}{\sigma-1}}, \sigma \geqslant 1, \text{ for all } j; \tag{2}$$

the availability of non-traded goods

$$h_j L_j \leqslant H_j \text{ for all } j; \tag{3}$$

the balanced-flows constraint

$$C_j^n + \sum_{k\in\mathcal{N}(j)} \left(1+\tau_{jk}\left(Q_{jk}^n, I_{jk}\right)\right) Q_{jk}^n \leqslant Z_j^n \left(L_j^n\right)^a + \sum_{k\in\mathcal{N}(j)} Q_{kj}^n \text{ for all j,n} \tag{4}$$

with per unit shipping cost $\tau_{jk}\left(Q_{jk}^n, I_{jk}\right) = \delta_{jk}^\tau \left(Q_{jk}^n\right)^\beta I_{jk}^{-\gamma}$; the network-building constraint

$$\sum_j \sum_{k\in\mathcal{N}(j)} \delta_{jk}^I I_{jk} \leqslant K; \tag{5}$$

exogenous bounds on network

$$0 \leqslant \underline{I}_{jk} \leqslant I_{jk} \leqslant \overline{I}_{jk} \text{ for all } j,k \in \mathcal{N}\left(j\right); \tag{6}$$

network symmetry

$$I_{jk} = I_{kj} \text{ for all } j, k \in \mathcal{N}(j); \tag{7}$$

local labor market clearing

$$\sum_{n=1}^{N} L_j^n \leqslant L_j \text{ for all } j; \tag{8}$$

and the non-negativity of c_j, C_j^n, L_j^n, I_{jk} and Q_{jk}^n.

2.2 Immobile labor, cross-good congestion

In the case of cross-good congestion, paid in term of the tradeable good bundle in each location, the problem is as in the previous formulation except that now:

- equation (2) becomes

$$c_j L_j + \sum_{k \in \mathcal{N}(j)} \tau_{jk}(Q_{jk}, I_{jk}) Q_{jk} \leqslant \left[\sum_{n=1}^{N} \left(D_j^n\right)^{\frac{\sigma-1}{\sigma}} \right]^{\frac{\sigma}{\sigma-1}}, \nu \geqslant 1, \tag{2'}$$

  where $\tau_{jk}(Q_{jk}, I_{jk}) \doteq \delta_{jk}^{\tau} (Q_{jk})^{\beta} I_{jk}^{-\gamma}$, and where we define

$$Q_{jk} = \left(\sum_{n=1}^{N} m_n \left(Q_{jk}^n\right)^{\nu} \right)^{\frac{1}{\nu}}, \nu \geqslant 1,$$

  where $m = (m_1, \ldots, m_N)'$ is a vector of positive weights; and

- the balanced flow constraint (4) in terms of each good becomes

$$D_j^n + \sum_{k \in \mathcal{N}(j)} Q_{jk}^n \leqslant Z_j^n \left(L_j^n\right)^a + \sum_{k \in \mathcal{N}(j)} Q_{kj}^n \text{ for all j,n.} \tag{4'}$$

2.3 Mobile labor, no cross-good congestion

The problem with labor mobility becomes

$$\max_{u, c_j, h_j, C_j^n, L_j^n, L_j, Q_{jk}^n, I_{jk}} u \tag{9}$$

subject to the free-labor mobility constraint

$$L_j u \leqslant L_j U(c_j, h_j) \text{ for all } j; \tag{10}$$

and the aggregate labor market clearing constraint

$$\sum_j L_j \leqslant 1 \tag{11}$$

in addition to constraints (2)-(8).

2.4 Mobile labor, cross-good congestion

As in the case with immobile labor, the problem is the one defined in subsection 2.3 but substituting equations (2) with (2)' and (4) with (4)'.

2.5 Partial mobility case

We also provide an intermediate model of labor mobility where agents can move freely within regions but not across regions. There are $1, \ldots, R$ regions. Each location j is associated with a given region $r(j) \in \{1, \ldots, R\}$. The total labor endowment for region r is L_r.

The objective function is

$$\max_{u_r, c_j, h_j, C_j^n, L_j^n, L_j, Q_{jk}^n, I_{jk}} \sum_{r=1}^{R} \omega_r L_r u_r \tag{12}$$

subject to the region-specific labor mobility constraint for each location

$$L_j u_{r(j)} \leqslant L_j U(c_j, h_j) \text{ for all } j; \tag{13}$$

and the region-specific labor market clearing constraint

$$\sum_{j|r(j)=r} L_j \leqslant L_r \text{ for all } r \in \{1, ..., R\}. \tag{14}$$

The rest of the constraints are identical to the previous cases whether we are in the case with cross-good congestion or without.

3 Installation

The toolbox is written for Matlab and uses two additional open source packages: IPOPT for optimization and ADiGator for autodifferentiation. We have tested the toolbox with IPOPT 3.11.8, ADiGator 1.4 and Matlab R2018a on Linux Ubuntu 18.04 and Windows 10.

3.1 Toolbox

Unpack the content of the zip file in your working folder, or unzip it in a folder of your choice and add the full path of the `Code/` subfolder into Matlab's search path (command addpath or 'Set Path' dialog box of the Matlab workspace).

3.2 IPOPT

IPOPT is a software designed to solve large-scale nonlinear optimization problems using an interior-point algorithm. It is required to use our toolbox and can be found at `https://projects.coin-or.org/Ipopt`. You will also have to install the Matlab interface as described here. In practice, installing IPOPT and compiling the mex files for Matlab can be annoying, so we have been using the precompiled mex files available at `https://www.coin-or.org/download/binary/Ipopt/`. Simply download the most recent version, for instance

Ipopt-3.11.8-linux64mac64win32win64-matlabmexfiles.zip,
unzip it in your working folder or in any folder of your choice and add it to Matlab's search path.[1]

3.3 ADiGator (optional)

ADiGator is optional and is only used to compute the gradient, jacobian and hessian of the Lagrangian for cases that we have not coded by hand. It can be found here: https://sourceforge.net/projects/adigator/. Follow the installation procedure detailed in ADiGator User Guide under the \doc folder of the ADiGator package. Make sure to run the script startupadigator.m before using.

4 Examples

4.1 Running the examples

We describe below the procedure to run the various examples that we provide (make sure to follow the previous steps to install IPOPT and ADiGator before following these steps!):

Step 1: Go to the Code/ directory.

Step 2: Open any of the scripts 'example0x.m' or 'paper_example0x.m'.

Step 3 (optional): Set the parameters of the model by defining the arguments of the function 'init_parameters' within the script. This includes defining whether there is cross-good congestion, whether there is mobility of labor, whether there is convexity or not (through the relative values of beta and gamma), or possibly the number of goods, etc.

Step 4: Execute script 'example0x.m' or 'paper_example0x.m' in Matlab.

4.2 Step-by-step example

Before getting started, make sure Matlab's search path includes your IPOPT folder as well as ADiGator's (don't forget to run the script startupadigator() that comes with the ADiGator package!). Once this is done, either get into the toolbox's folder or add its path to Matlab's search path as well.

Step 1. The first important step in using the toolbox is to create the param structure that contains the model's parameters by using the function init_parameters() described in section 6.1. For default parameters (no labor mobility, no cross good congestion, convex case), simply write:

```
param = init_parameters();
```

Non-default parameter are set by specifying them as arguments in the function init_parameters(). For instance, to consider a non-convex case with labor mobility and cross-good congestion, set instead:

[1] Another option is to use the precompiled mex files from mexIPOPT by Enrico Bertolazzi available on Matlab Central at here. Simply copy the files ipopt.mexa64 (Linux) or ipopt.mexmaci64 (OSX) under precompiled_mex/ in your working folder as well as lib/ipopt.m and lib/ipopt_auxdata.m, or simply add these folders to your Matlab path.

```
param = init_parameters('LaborMobility','on','CrossGoodCongestion','on','gamma',2,'beta',1);
```

This case is non-convex because gamma>beta. The arguments in init_parameters can be introduced in any order.

Step 2. We then want to create the underlying graph on which trade will take place. To do so, we use the function create_graph() described in section 6.2. For a default 'map' network, simply call:

```
[param,graph] = create_graph(param, 11, 11); % create an 11x11 map network
```

For a 5 x 5 triangular network, call instead

```
[param,graph] = create_graph(param, 5, 5, 'Type', 'triangle'); % 5x5 triangular network
```

Note that the `param` structure is also modified by the function `create_graph()` which enriches it with additional entries that can only be set when the number of location is known, such as J, L_j, H_j, Z_j^n. By default, all these endowments are uniformly distributed.

Step 3. Unless there is some asymmetry across locations, there will be no trade. We thus want to customize the endowment/productivity distribution and introduce some asymmetry across locations. For instance, we set the productivity of the central location to be twice as much as in other locations ($Z_j^n = 1$ everywhere else by default). Since we have created an 11x11 network, it will be the (6,6) node:

```
i = find_node(graph,6,6); % get the index of the central node located in (6,6)
param.Zjn(i) = 2; % make the central node more productive
```

Step 4. We are now ready to compute the optimal network. To do so, we call the function `optimal_network()` described in section 6.3.

```
res = optimal_network(param,graph);
```

The result structure res will contain all the optimal infrastructure investments in `res.Ijk`, but the full allocation as well in `res.Cj`, `res.Lj`, etc.

Step 5. To conclude, let us plot the resulting distribution of optimal infrastructure investments using the function `plot_graph()` described in section 6.7. Figure 1 shows the result.

```
plot_graph(param,graph,res.Ijk);
```

Full code:

```
1  % First, initialize the param structure containing the model parameters
2  param = init_parameters();
3
4  % Second, initialize the underlying graph
5  [param,graph] = create_graph(param, 11, 11, 'Type', 'map'); % create an 11x11 map network
6
7  % Third, customize the distribution of endowment
8  i = find_node(graph,6,6); % get the index of the central node
9  param.Zjn(i) = 2; % make the central node more productive (others are set to 1 by default)
10
11 % Fourth, compute the optimal network
12 res = optimal_network(param,graph);
13
14 % Fifth, plot the result
15 plot_graph(param,graph,res.Ijk);
```

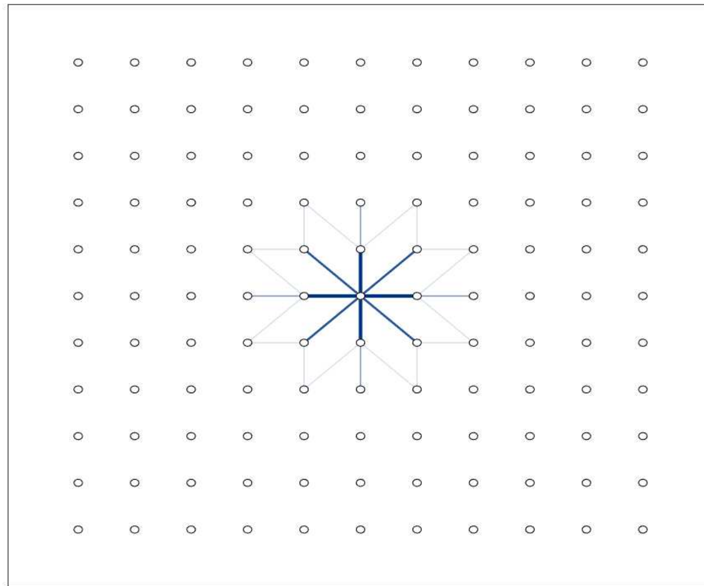


Figure 1: Simple example from section 4.2

5 Data structures

5.1 Model parameter structure **param**

The param structure is created by the function init_parameters() detailed in section 6. It contains most of the model parameters as described in Table 1.

5.2 Graph structure **graph**

The graph structure is created by the function create_graph() detailed in section 6. It contains the graph on which trade is taking place. Table 2 below describes its entries.

5.3 Results structure **res**

The content of the results structure res is the output of function optimal_network() described in Section 6. The various entries are explained in Table 3.

Variable	**Dimension**	**Description**
welfare	real	Total utilitarian welfare
uj	[J,1]	Utility in each location
cj	[J,1]	Per capita final consumption per location
Cj	[J,1]	Total tradeable consumption per location
Dj	[J,1]	Total availability of tradeable good pre-transport cost per location (cross-good congestion only)
Cjn	[J,N]	Total consumption per good (post-transport costs if cross-good congestion)
Djn	[J,N]	Total availability of each good pre-transport cost per location (cross-good congestion only)
hj	[J,1]	Non-tradable good per capita
Qjkn	[J,J,N]	Flows from j to k of good n
Qin	[ndeg,N]	Flows along all edges (linearly indexed, for computation purposes)
Lj	[J,1]	Population
Ljn	[J,N]	Working population per good
Pjn	[J,N]	Prices of traded goods per location
PCj	[J,N]	Aggregate prices of tradeable good bundle per location
Yjn	[J,N]	Production per good per location
Ijk	[J,J]	Optimal infrastructure per edge (symmetric)

Table 3: Results structure res

5.4 Geography structure **geography**

The geography structure is only used to encode additional information about the terrain such as elevation and obstacles to be used when plotting the terrain in 3D using plot_graph(). It is also used by the function apply_geography() which provides a way to encode such geographical

Variable	Dimension	Description
Optimization options		
`mobility`	0, 1 or 0.5	Determines whether labor is mobile (1), immobile (0) or partially mobile (0.5).
`cong`	boolean	Determines whether is cross good congestion or not. Yes if true.
`custom`	boolean	Determines whether to maximize the custom objective function objective_custom() subject to the constraints constraints_custom().
`adigator`	boolean	Determines whether to use ADiGator or force using hand calculated jacobiones. Yes if true.
`annealing`	boolean	Determines whether to automatically run the simulated annealing algorithm after optimization in the nonconvex case. Yes if true.
`verbose`	boolean	Determines whether to display the steps and results of optimization. Yes if true.
Model parameters		
`J`	$\mathbb{N}$	Number of locations J
`N`	$\mathbb{N}$	Number of traded goods N
`sigma`	real>1	Elasticity of substitution σ between goods in CES $C_j = \left(\sum_n (C_j^n)^{\frac{\sigma-1}{\sigma}}\right)^{\frac{\sigma}{\sigma-1}}$
`omegaj`	[J,1]	Vector of Pareto weights ω_j (only relevant in the immobile labor case)
`omegar`	[nregions,1]	Vector of Pareto weights ω_r per region (partial mobility case only)
`Lj`	[J,1]	Vector of population L_r per region (partial mobility case only)
`Lr`	[`nregions`,1]	Vector of population L_j per location (immobile labor case only)
`Hj`	[J,1]	Endowment of nontradeable good H_j (housing) per location
`hj`	[J,1]	Nontradeable good per person h_j in each location (only relevant in the immobile labor case)
`Zjn`	[J,N]	Matrix of productivity Z_j^n of each location j in good n
`K`	real>0	Total endowment of concrete/asphalt
Preferences and technology		
`a`	real$\in(0,1]$	Returns to scale to labor in production function $Z_j^n \left(L_j^n\right)^a$
`rho`	real$\geqslant 1$	Inequality aversion ρ in utility $U(c,h) = \left((\frac{c}{\alpha})^\alpha (\frac{h}{1-\alpha})^{1-\alpha}\right)^{1-\rho} / (1-\rho)$
`alpha`	real$\in(0,1)$	Spending share α on traded goods in utility $U(c,h) = \left((\frac{c}{\alpha})^\alpha (\frac{h}{1-\alpha})^{1-\alpha}\right)^{1-\rho} / (1-\rho)$
Transport cost parameters		
`beta`	real>0	Parameter β governing intensity of congestion in transport
`gamma`	real>0	Parameter γ governing returns to scale in infrastructure investment
`nu`	real>1	Parameter ν governing degree of cross-good congestion (irrelevant if no cross-good congestion)
`m`	Nx1	Good-specific weight parameters in the case with cross-good congestion

Table 1: `param` structure

Variable	Dimension	Description
`J`	$\mathbb{N}$	Number of locations J
`x`	[J,1]	Vector of x coordinates for each location j
`y`	[J,1]	Vector of y coordinates for each location j
`nodes`	[J,1] array of struct	Each nodes is a structure that contains the entry neighbors, which is a list of indices of neighboring nodes. Ex.: `graph.nodes{i}.neighbors` is a row vector containing the indices of node `i`'s neighbors.
`adjacency`	[J,J]	Adjacency matrix of underlying undirected graph
`delta_i`	[J,J]	Matrix of edge specific investment costs δ^i_{jk}
`delta_tau`	[J,J]	Matrix of edge specific transport costs δ^τ_{jk}
`ndeg`	$\mathbb{N}$	Number of degrees of freedom of undirected graph, also equal to # of edges (e.g.,=sum(reshape(tril(adjacency),[J^2,1])))
`region`	[J,1]	Vector indicating in which region each location belongs. The region id should be between 1 and *nregions* (partial mobility only)

Table 2: `graph` structure

Variable	Dimension	Description
`z`	`[J,1]`	Vector of z-coordinates (elevation) for each node
`obstacles`	`[nb,2]`	Matrix of pairs of node indices indicating where obstacles like rivers, etc. are placed, where `nb` is an arbitrary number of obstacles. Each index must be an integer between 1 and `graph.J`. E.g., to place obstacles on links 1-2 and 2-3, define: `obstacles = [1,2; 2,3];` Each pair of nodes must be neighbors in the underlying graph.

Table 4: `geography` structure

details in the cost matrices of the `graph` structure. Each `geography` structure must be created from scratch as we do not provide any function to automatically generate it. See example04.m for a working example. Table 4 below explains its various entries.

6 Functions

This section describes the main functions from the Toolbox that a regular user may want to use.

6.1 Function `init_parameters()`

The function `[param]=init_parameters(varargin)` is the first function from the toolbox that should be called to initialize the model parameters.

Example usage:

```
% Simplest call, default parameters
param = init_parameters();
```

```
% Some custom parameters, others set to default
param = init_parameters('LaborMobility','on','N',2,'beta',1,'gamma',2);
```

The list of customizable parameters are described in Table 5. See Table 1 for additional description on the corresponding parameters. Note that by default all unspecified parameters are set to their default values, unless a pre-existing param object is provided, in which case it will solely override the specified parameters and check consistency with the others.

6.2 Function `create_graph()`

Function `[param,graph] = create_graph(param,w,h,varargin)` creates the basic underlying graph in which trade will take place.

Arguments:

- 'param': param structure created by the function `init_parameters()` that contains the model parameters;
- 'w': width of the graph (max # of nodes along horizontal axis);
- 'h': height of the graph (# of nodes along vertical axis);
- varargin: additional options (see below).

The additional options that can be put in varargin are listed in Table 6.

The different types of preprogrammed networks acceptable by the function are:

- 'square': regular network of w x h nodes placed in a rectangular graph with coordinates (i, j) for $i = 1..w$ and $j = 1..h$. Each interior node has 4 neighbors: 2 along the vertical and 2 along the horizontal axes.
- 'map': same as the square network except that each interior node has 8 neighbors along the vertical, horizontal and diagonal axes.
- 'triangle': triangular network with h rows such that nodes are placed at coordinates (i, j) for $i = 1..w$, j odd and $i = -0.5 + 1..w - 1$, j even between 1 and h. Each interior node has 6 neighbors: 2 along the horizontal and 4 along the two diagonals. Note that we require h to be odd for triangular networks.
- 'custom': custom network provided by user with a given adjacency matrix, X and Y coordinates (in that case, w and h are unused and can be set to []).

Figure 2 displays these different types of networks, the way nodes are indexed by default and their corresponding (x, y) coordinates. These default networks can be customized using the function `add_node()` and `remove_node()`. Their location in space can be modified directly by changing the coordinate vectors `graph.x` and `graph.y`. By default, the cost matrices `delta_tau` and `delta_i` are set equal to the cartesian distance between nodes. Graphs do not have to be created by the function `create_graph()` and can be customized at will. You should make sure, however, that each entry detailed in Table 2 are correctly set to be compatible with the other functions in the toolbox.

Output:

Options	Value	Default	Description
Miscellaneous options			
`'LaborMobility'`	'on', 'off', 'partial'	'off'	Turn on/off labor mobility or allow for partial mobility (labor is mobile only within prespecified regions).
`'CrossGoodCongestion'`	'on'/'off'	'off'	Turn on/off cross-good congestion.
`'Custom'`	'on'/'off'	'off'	Turn on/off custom objective/constraint usage.
`'ADiGator'`	'on'/'off'	'off'	Use of ADiGator even in hand-calculated cases.
`'Annealing'`	'on'/'off'	'on'	Turn on/off simulated annealing in nonconvex case.
`'Verbose'`	'on'/'off'	'on'	Turn on/off display of info about optimization.
'param'	`param` structure	none	Provide an already existing `param` structure to be used as default
Preferences and technology			
`'a'`	real$\in (0,1]$	0.8	Returns to scale to labor in production function
`'rho'`	real$\geqslant 1$	2	Inequality aversion ρ in utility
`'alpha'`	real$\in (0,1)$	0.5	Spending share α on traded goods in utility
`'N'`	$\mathbb{N}$	1	Number of traded goods N
`'sigma'`	real>1	5	Elasticity of substitution σ
`'K'`	real>0	1	Total endowment of concrete/asphalt
Transport cost parameters			
`'beta'`	real>0	1	Transport congestion parameter β
`'gamma'`	real>0	1	Returns to infrastructure investment γ
`'nu'`	real>1	1	Elasticity of substitution ν in cross-good congestion
`'m'`	Nx1	ones(N,1)	Good-specific weight in cross-good congestion

Table 5: Options of `init_parameters`

Options	Value	Default	Description
'Type'	'map', 'square', 'triangle' or 'custom'	'map'	Set the type of the network (see below)
'ParetoWeights'	[J,1]	ones(J,1)	Vector of Pareto weights
'Adjacency'	[J,J]	[]	Adjacency matrix of the network (for custom type only)
'X'	[J,1]	[]	Vector of x coordinates of locations (for custom type only)
'Y'	[J,1]	[]	Vector of y coordinates of locations (for custom type only)
'NRegions'	$\mathbb{N}$	1	Number of regions (partial mobility only)
'Region'	[J,1]	ones(J,1)	Vector indicating to which region each location belongs. The region index should be within 1 and `nregions` (partial mobility only)

Table 6: Options for `varargin` in `create_graph`

- 'param': an updated param structure (augmented with values for J, L_j, H_j, h_j and Z_j^n once the number of locations is defined). By default, productivity, endowments and population are uniform across locations.
- 'graph': the graph structure containing the nodes, their location, its adjacency matrix, etc.

Example usage:

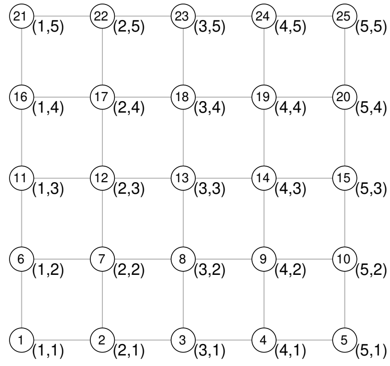
```
% Initialize model parameters
param=init_parameters();

% Create an 11x11 triangular network
[param,graph] = create_graph(param,11,11,'Type','triangle');
```

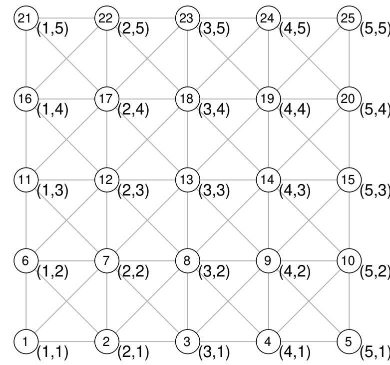
6.3 Function `optimal_network()`

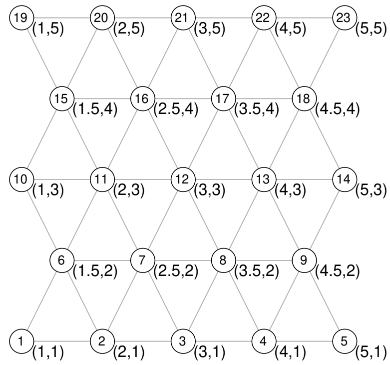
Function `res = optimal_network(param,graph,I0,Il,Iu,verbose)` solves for the optimal transport network of the economy described in variables `param` and `graph`.

Arguments:

- 'param': `param` structure created by the function `init_parameters()` that contains the model parameters;
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'I0': (optional) $J \times J$ symmetric matrix containing the seed for the matrix of infrastructure investment provided to the solver, set to `[]` or nothing for default seed (non-connected links should be set to 0);
- 'Il': (optional) $J \times J$ matrix containing lower bounds for the matrix of infrastructure investment $\underline{I}_{jk}$ as in equation (6), set to `[]` or nothing for none;







(c) Triangle network

Figure 2: Various types of preprogrammed networks for $w = 5$ and $h = 5$

- 'Iu': (optional) $J \times J$ matrix containing upper bounds for the matrix of infrastructure investment $\overline{I}_{jk}$ as in equation (6), set to `[]` or nothing for none.
- '`verbose`': (optional) boolean that instructs IPOPT to display each iteration.

Output:

- 'res': result structure containing the optimal transport infrastructure and economic allocation.

Example usage:

```
% Initialize default parameters and graph
param=init_parameters();
[param,graph] = create_graph(param,11,11);

% Solve for the optimal transport network
res = optimal_network(param,graph);
```

6.4 Function `annealing()`

Function `res = annealing(param,graph,I0,varargin)` applies the simulated annealing algorithm to approximate for the globally optimal transport network for the economy described in variables `param` and `graph`. This function is only useful in the nonconvex case when $\gamma > \beta$.

Arguments:

- 'param': `param` structure created by the function `init_parameters()` that contains the model parameters;
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'I0': (optional) $J \times J$ symmetric matrix containing the seed for the matrix of infrastructure investment provided to the algorithm (usually the outcome of a first optimization from optimal_network()), set to `[]` or nothing for default seed (non-connected links should be set to 0);
- varargin: additional options (see below).

The additional options that can be put in varargin are listed in Table 7.

Different methods are proposed to perturb the network in each step of the simulated annealing method (option '`PerturbationMethod`'):

- '`random`' adds `NbRandomPerturbations` links at random in the network, making sure the asphalt resource constraint is met;
- '`shake`' modifies the network as if it had been "shaked" in some random direction before applying some Gaussian smoothing;
- '`rebranching`' implements the rebranching algorithm described in the paper whereby each location is reconnected to its best connected neighbor;
- '`random rebranching`' is identical to rebranching except that only #NbRandomPerturbations nodes are picked at random for rebranching;
- '`hybrid alder`' implements the Hybrid Alder-FS algorithm described in the paper which attempts to replicate the spirit of the algorithm described in Alder (2018).

Output:

- 'res': result structure containing the optimal transport infrastructure and economic allocation.

Options	Value	Default	Description
`'PerturbationMethod'`	'shake', 'random', 'rebranching', 'random rebranching', 'hybrid alder'	'rebranching'	See description
`'PreserveCentralSymmetry'`	'on'/'off'	'off'	(shake only) Make sure perturbations preserve central symmetry
`'PreserveVerticalSymmetry'`	'on'/'off'	'off'	(shake only) Make sure perturbations preserve vertical symmetry
`'PreserveHorizontalSymmetry'`	'on'/'off'	'off'	(shake only) Make sure perturbations preserve horizontal symmetry
`'SmoothingRadius'`	real>0	0.25	(shake only) Gaussian smoothing radius
`'MuPerturbation'`	real>0	log(0.3)	(shake only) Average radius of translation
`'SigmaPerturbation'`	real>0	0.05	(shake only) Stdev of translation
`'Display'`	'on'/'off'	'off'	Display network after each perturbation
`'TStart'`	real>0	100	Initial temperature of annealing
`'TEnd'`	real>0	1	Final temperature of annealing
`'TStep'`	0<real<1	0.9	Cooling step (e.g., temperature at t is $TStart \times TStep^t$)
`'NbDeepening'`	$\mathbb{N}^*$	4	# of FOC iterations between perturbations
`'NbRandomPerturbations'`	$\mathbb{N}^*$	1	# of links randomly changed in '`random`' and '`random rebranching`' methods
`'Funcs'`	funcs structure	[]	`funcs` structure provided by ADiGator to skip rederiving jacobian and hessian
`'Il'`	[J,J]	[]	matrix of lower bounds $\underline{I}_{jk}$
`'Iu'`	[J,J]	[]	matrix of upper bounds $\overline{I}_{jk}$

Table 7: Options for `varargin` in `annealing`

Example usage:

```
% Init in nonconvex case gamma>beta
param = init_parameters('beta',1,'gamma',2);
[param,graph] = create_graph(param,11,11);

% Solve for the optimal transport network (convex solver)
res = optimal_network(param,graph);

% Improve result using simulated annealing algorithm
res2 = annealing(param,graph,res.I);
```

6.5 Function `solve_allocation()`

Function `[res,flag,x] = solve_allocation(param,graph,I,verbose,x0,funcs)` computes the equilibrium allocation for the economy for a given network of infrastructure investment I.

Arguments:

- 'param': `param` structure created by the function `init_parameters()` that contains the model parameters;
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'I': $J \times J$ symmetric matrix containing the level of infrastructure in each link of the network (non-connected links should be set to 0);
- 'verbose': (optional, false by default) switch telling IPOPT whether to display iterations or not;
- 'x0': (optional) you may provide the seed given to IPOPT (the size varies depending on the resolution method, see section 7). May be set to [] for none;
- 'funcs': (optional) to avoid calling ADiGator everytime, you may provide the `funcs` structure containing the various derivatives yourself (use `call_adigator()` to obtain these derivatives).

Output:

- 'res': result structure containing the optimal transport infrastructure and economic allocation;
- 'flag': flag variable returned by IPOPT. 0 (solved) or 1 (solved to acceptable level) indicate success. See IPOPT reference for other error codes;
- 'x': optimal control variable as returned by IPOPT (may be used to warm-start the optimization).

Example usage:

```
% Init in nonconvex case gamma>beta
param = init_parameters('beta',1,'gamma',2);
[param,graph] = create_graph(param,11,11);

% Create a uniform infrastructure grid with one for each link (i.e. the adjacency matrix)
I = graph.adjacency;

% Solve for the economic allocation
res = solve_allocation(param,graph,I);
```

6.6 Function `call_adigator()`

Function `funcs=call_adigator(param,graph,I,verbose)` calls ADiGator in order to compute the gradient, jacobian and hessian of the social planning problem for a given network of infrastructure investment I.

Arguments:

- 'param': `param` structure created by the function `init_parameters()` that contains the model parameters;
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'I': $J \times J$ symmetric matrix containing the level of infrastructure in each link of the network (non-connected links should be set to 0);
- 'verbose': (optional, false by default) whether or not ADiGator should display results on the Matlab screen;

Output:

- 'funcs': structure `funcs` with derivative functions used as input by IPOPT
-

Example usage:

```
% Init in nonconvex case gamma>beta
param = init_parameters('beta',1,'gamma',2);
[param,graph] = create_graph(param,11,11);

% Create a uniform infrastructure grid with one for each link (i.e. the adjacency matrix)
I = graph.adjacency;

% Precalculate the derivatives
funcs = call_adigator(param,graph,I,true);

% Solve for the economic allocation, skipping the call to ADiGator
res = solve_allocation(param,graph,I,false,[],funcs);
```

6.7 Function `plot_graph()`

Function `plot_graph(param,graph,edges,varargin)` plots the underlying graph with link intensity defined by `edges`.

Arguments:

- 'param': `param` structure created by the function `init_parameters()` that contains the model parameters;
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'edges': $J \times J$ matrix containing the capacity/investment/flow for each link of the network. If non-symmetric, the max of `edges(j,k)` or `edges(k,j)` is plotted. Should be equal to 0 for unconnected nodes.
- varargin: additional options (see below).

The additional options that can be put in varargin are listed in Table 9.

Options	Value	Default	Description
'Mesh'	'on'/'off'	'off'	Plot the underlying mesh
'Edges'	'on'/'off'	'on'	Plot the edge intensity given in edges
'Arrows'	'on'/'off'	'off'	Plot arrows indicating direction of flow
'Nodes'	'on'/'off'	'on'	Plot nodes for each location
'Geography'	'on'/'off'	'off'	Whether or not a full 3D geography is to be plotted (a geography structure must be specified)
'Obstacles'	'on'/'off'	'off'	(geography only) plot obstacles (rivers, forest, etc) specified in geography.obstacles
'Transparency'	'on'/'off'	'on'	Whether to use transparency when plotting (nicer but slower)
'EdgeScaling'	'on'/'off'	'on'	Whether to rescale edge thickness and intensity between min and max or use absolute value specified in edges
'MinEdge'	real>0	0	Minimum edge intensity under which links are not plotted
'MaxEdgeThickness'	real>0	2	Maximal thickness of edge to be plotted
'MinEdgeThickness'	real>0	0.1	Minimal thickness of edge to be plotted
'Shades'	[J,1]	zeros(J,1)	Color of each node is given by shades(i)*NodeBgColor +(1-shades(i))*NodeFgColor
'Sizes'	[J,1]	ones(J,1)	Size of each node
'NodeFgColor'	[1,3]	[1,0,0] (red)	Node foreground color
'NodeBgColor'	[1,3]	[1,1,1] (white)	Node background color
'NodeOuterColor'	[1,3]	[0,0,0] (black)	Node outer color
'NodeColorMap'	[x,3]	[]	Colormap to be used for nodes (overrides NodeBgColor and NodeFgColor)
'EdgeColor'	[1,3]	[0,0.2,0.5] (dark blue)	Node outer color
'MeshColor'	[1,3]	[0.9,0.9,0.9] (light gray)	Underlying mesh color
'MeshStyle'	'-' / ':' / '.-' / '--'	'-'	Line style used to draw underlying mesh
'MeshTransparency'	0<real<1	1	alpha transparency for underlying mesh
'ObstacleColor'	[1,3]	[0.4,0.7,1] (light blue)	Obstacle color (if Obstacles is on)
'CMax'	[1,3]	[0.9,0.95,1] (bluish white)	(geography only) maximal color for heat map (e.g., mountain peak color)
'CMin'	[1,3]	[0.4,0.65,0.6] (funky green)	(geography only) minimal color for heat map (e.g., green plains)

'Margin'	real>0	0.1	(non-geography only) default margin around graph as a percentage
'ArrowScale'	real>0	1	(arrows only) scaling factor for arrow size
'ArrowStyle'	'long'/'thin'	'long'	(arrows only) arrow style
'GeographyStruct'	geography structure	[]	(geography only) geography structure that defines elevation of terrain and position of obstacles
'View'	[1,2]	[30,45]	(geography only) [AZ,EL] that defines the view angle in the 3D view

Table 9: Options for `varargin` in `plot_graph`

Output:

- None.

Example usage:

```
% ..after initialization

% Solve for the optimal transport network
res = optimal_network(param,graph);

% Plot network
plot_graph(param,graph,res.Ijk);
```

6.8 Function `add_node()`

Function `[param,graph]=add_node(param,graph,x,y,neighbors)` adds a node to the underlying network located in (x, y) with neighbors listed in `neighbors`.

Arguments:

- 'param': `param` structure created by the function `init_parameters()` that contains the model parameters;
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'x': x coordinate of the node to be created;
- 'y': y coordinate of the node to be created;
- 'neighbors': $1 \times nb$ list of node indices to which the new node is to be connected, where `nb` is an arbitrary number of neighbors. Indices must integers comprised between 1 and J.

Output:

- 'param': updated param structure with default population, endowments and productivity;
- 'graph': new graph augmented with additional node (indexed by $J + 1$)

Example usage:

```
% Initialize default parameters and graph
param=init_parameters();
[param,graph] = create_graph(param,10,10,'Type','square');

% Add a node in the middle, connected to its 4 closest neighbors
[param,graph] = add_node(param,graph,5.5,5.5,[45,46,55,56]);
```

6.9 Function `remove_node()`

Function `[param,graph]=remove_node(param,graph,i)` removes the node indexed by `i`.

Arguments:

- 'param': `param` structure created by the function `init_parameters()` that contains the model parameters;
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'i': index of the node to be removed, comprised between 1 and J (see Figure (2) to understand how nodes are indexed or use `find_node()`).

Output:

- 'param': updated param structure with default population, endowments and productivity;
- 'graph': new graph with node i removed (remaining nodes are re-indexed between 1 and $J-1$)

Example usage:

```
% Initialize default parameters and graph
param=init_parameters();
[param,graph] = create_graph(param,11,11);

% Remove central node
i=find_node(graph,6,6); % get the index of the central node
[param,graph]=remove_node(param,graph,i); % remove it
```

6.10 Function `find_node()`

Function `i=find_node(graph,x,y)` returns the index i of the node in graph closest to location (x, y).

Arguments:

- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'x': x coordinate of the node to be found;
- 'y': y coordinate of the node to be found.

Output:

- 'i': index of the node closest to location.

Example usage:

```
% Initialize default parameters and graph
param=init_parameters();
[param,graph] = create_graph(param,11,11);

% Get index of node located at (4,8)
i=find_node(graph,4,8);
```

6.11 Function `apply_geography()`

Function `graph = apply_geography(graph, geography, varargin)` takes a geography structure to encode it in the cost matrices `delta_i` and `delta_tau` of the corresponding graph structure. More specifically, it parametrizes the building cost of each segment as

$$\hat{\delta}^l_{jk} = \text{Euclidian distance}_{jk} \times \left(1 + \alpha_{l,\text{up}} \max\left(z_k - z_l, 0\right)^{\beta_{l,\text{up}}} + +\alpha_{l,\text{down}} \max\left(z_l - z_k, 0\right)^{\beta_{l,\text{down}}}\right) \text{ for } l = i, \tau$$

where z_j is the elevation of node j as specified in the `z` entry of the geography structure. As for obstacles, the function sets

$$\delta^l_{jk} = \hat{\delta}^l_{jk} \times \begin{cases} \delta_{l,\text{along}} & \text{if the link is where an obstacle is located} \\ \delta_{l,\text{across}} & \text{if the link crosses an obstacle} \\ 1 & \text{otherwise.} \end{cases} \quad \text{for } l = i, \tau$$

Arguments:
- 'graph': `graph` structure created by the function `create_graph()` that contains the underlying network;
- 'geography': geography structure as defined in
- 'varargin': additional parameters to the cost parametrization (see Table 10);

Output:
- 'graph': `graph` structure with updated cost matrices.

Example usage: see example04.m

7 Comments on the resolution method

7.1 Overview of the algorithm

The example scripts `example0x.m` initialize parameters and builds the graph, then calls the function `optimal_network()`. The function `optimal_network()` runs the optimization, with the convex solver, allows to define the initial value of the optimization variables, as well as the objective and the constraints functions within the files of prefix `objective_xxx.m` and `constraints_xxx.m`. Using the autodifferentiation package ADiGator, it calculates the gradient of the objective, jacobian of the constraints, and hessian of the Lagrangian. It calls the functions with prefix `solve_allocation_xxx.m` which solves the inner problem (optimal flows and allocation for a given infrastructure) and iterates over the FOC of the infrastructure network to solve the outer problem.

Options	**Value**	**Default**	**Description**
`'AlphaUp_i'`	real>0	0	Parameter $\alpha_{i,\text{up}}$ that governs the building cost going up
`'BetaUp_i'`	real>0	1	Parameter $\beta_{i,\text{up}}$ that governs the building cost going up
`'AlphaUp_tau'`	real>0	0	Parameter $\alpha_{\tau,\text{up}}$ that governs the transport cost going up
`'BetaUp_tau'`	real>0	1	Parameter $\beta_{\tau,\text{up}}$ that governs the transport cost going up
`'AlphaDown_i'`	real>0	0	Parameter $\alpha_{i,\text{down}}$ that governs the building cost going down
`'BetaDown_i'`	real>0	1	Parameter $\beta_{i,\text{down}}$ that governs the building cost going down
`'AlphaDown_tau'`	real>0	0	Parameter $\alpha_{\tau,\text{down}}$ that governs the transport cost going down
`'BetaDown_tau'`	real>0	1	Parameter $\beta_{\tau,\text{down}}$ that governs the transport cost going down
`'AlongObstacleDelta_i'`	real>0	1	Rescaling parameter for building cost along an obstacle
`'AlongObstacleDelta_tau'`	real>0	1	Rescaling parameter for transport cost along an obstacle
`'AcrossObstacleDelta_i'`	real>0	1	Rescaling parameter for building cost across an obstacle
`'AcrossObstacleDelta_tau'`	real>0	1	Rescaling parameter for transport cost across an obstacle

Table 10: Options for `varargin` in `apply_geography`

7.2 The six model cases

There six main different model cases which have to be solved with a specific code. These cases are: primal mobile with and without cross good congestion (cgc), primal immobile with and without cross good congestion, dual immobile without cross good congestion and small beta (fastest but only applicable when the dual is twice-differentiable, that is $\beta \leqslant 1$).These cases are summarized in a table below.

In the cases with cross good congestion, the transport cost is expressed in final good which requires c_j as an additional optimization variables, in addition to the D_j^n. The mobile cases require J more optimization variables, which are the total population in each node L_j. The dual can only be solved in the immobile case (otherwise the problem is quasi-convex and strong duality does not hold) without cross-good congestion (because we were lazy...). The dual does not require a constraints function by definition. In the case with $a = 1$ (constant returns to scale of the production function) and multiple goods produced in each location, the optimal labor L_j^n cannot be expressed as a function of prices directly and we turn to the primal. The custom case default settings are presented in Table 11 below, but can be modified as described in section 7.3.

Note that none of these cases depend on the relative values of β and γ. This is because we solve the model in two steps: an always convex inner problem, while we simply iterate on the FOCs of the outer problem. As a result, in the non-convex case $\gamma > \beta$, we have the same six cases described above. The differences will be that i) the algorithm will converge to a local optimum ii) there is the option of refining this local optimum using simulated annealing (either run function `annealing()` or set 'Annealing' to 'on' in `init_parameters()`).

Case	Parameters	Optimization variables
Dual immobile	cong = off mobility = off $\beta \leqslant 1$ $a < 1$	P_j^n (multiplier on flow constraint) W_j (multiplier on local labor constraint)
Primal immobile	cong = off mobility = off $\beta > 1$ or $a = 1$	Q_{jk}^n (flows along each edge) L_j^n (labor per good) D_j^n (consumption per good)
Primal mobile	cong = off mobility = on	$Q_{jk}^{n,d}$ (flows along each edge in the edge direction) $Q_{jk}^{n,ind}$ (flows along each edge in the edge opposite direction) L_j^n (labor per good) D_j^n (consumption per good) L_j (population)
Primal immobile with cgc	cong = on mobility = off	$Q_{jk}^{n,d}$ (flows along each edge in the edge direction) $Q_{jk}^{n,ind}$ (flows along each edge in the edge opposite direction) L_j^n (labor per good) D_j^n (consumption per good) c_j (aggregate consumption, since transport cost in final good)
Primal mobile with cgc	cong = on mobility = on	$Q_{jk}^{n,d}$ (flows along each edge in the edge direction) $Q_{jk}^{n,ind}$ (flows along each edge in the edge opposite direction) L_j^n (labor per good) D_j^n (consumption per good) c_j (aggregate consumption, since transport cost in final good) L_j (population)
Custom (default description)	cong = off mobility = off	Q_{jk}^n (flows along each edge) L_j^n (labor per good) D_j^n (consumption per good)

Table 11: The six different cases

7.3 Custom model

One might want to customize the model in a number of ways. We allow the user to provide its own objective and constraint functions solved by the social planner in the inner problem. We provide the files `objective_custom`, `constraints_custom`, and `solve_allocation_custom_ADiGator` for that purpose that are pre-coded according to the primal immobile without cross good congestion case described in the previous section. We explain below the steps to follow in order to adjust the code to your own use. Additional suggestions are also provided in the files. Note that only in the cases preserving the concavity of the utility function and the convexity of the constraints will the code provide a global optimum (otherwise it should converge to a local optimum improved by simulated annealing).

The user might want in particular to modify the following: utility function, production function, or transport cost.

Here is a list of the steps to follow:

Step 1: Modify objective function `objective_custom.m` and constraint functions `constraints_custom.m` according to new model specification desired.

Step 2: In `optimal_network.m`, provide the initial seed of the optimization with the right number of variables (section CUSTOMIZATION 1 in the code).

Step 3: In `optimal_network.m`, modify the expression of the optimal infrastructure level as a function of prices (section CUSTOMIZATION 2 in the code).

Step 4: (If non convex) In `annealing.m`, make the same modifications as in `optimal_network()` in sections CUSTOMIZATION 1, 2 and 2' in the code.

Step 5: In `solve_allocation_custom_ADiGator.m`, modify the 'return allocation' section and the `recover_allocation()` function according to your needs.

8 Known bugs and issues

- *Important*: It is very important to scale the problem properly for IPOPT to find the maximum. The IPOPT documentation recommends to keep numerical values between 1e-3 and 1e3. If you encounter such problems with IPOPT, rescaling variables and parameters so that you remain in this region may be a solution. This rescaling includes setting the average level of productivity, population, endowments of nontradeables, trade costs so that the resulting endogenous variables fall broadly within these bounds. Special care must be taken for derivatives. For instance, we recommend to avoid setting population, productivity or H in a location equal to 0 (even though the theoretical problem is well defined), as it can easily lead to infinite derivatives in the problem. Setting these endowment values to something small like 1e-2 or 1e-3 is often enough to fix IPOPT problems.
- When solving the mobile labor case, ADiGator returns the following warning "`Warning: derivative of discontinuous sign function - making derivatives zero`". This is caused by the fact that the objective function or constraints may include absolute values, indicator functions or sign functions which have a discontinuity at 0. These warnings can be ignored as ADiGator sets the derivative to 0 at the discontinuity. To the extent that the result of the optimization is usually away from 0 (especially in the convex case), this discontinuity is irrelevant for our purposes.
- ADiGator has difficulty dealing with large systems (e.g., 200+ variables). Hence, we have coded some jacobians/hessians by hand in various cases. Unfortunately, we haven't coded every possible case. Cases we have (with or without cross-good congestion):
  - no labor mobility: convex case ($\beta \leqslant 1$, $\gamma \leqslant \beta$, $a < 1$), solved by duality
  - no labor mobility, $\beta \geqslant 1$, any γ, any $a \leqslant 1$ but at most one good is produced in each location (that is, there is at most one good with a strictly positive Z_j^n in each location), solved by primal

- labor mobility, any β, any γ, any $a \leqslant 1$ but at most one good is produced in each location, solved by primal
- partial mobility case, any β, any γ, any $a \leqslant 1$ but at most one good is produced in each location, $\rho = 1$, solved by primal

- Sometimes IPOPT crashes with Matlab altogether. These come from problems catching errors between Matlab and IPOPT. They often come from numerical or tiny mistakes in the computation of the objective, constraints and their derivatives. Recompiling the mex file or using a newer IPOPT-Matlab interface may help.

- Feel free to contact us if you find additional bugs. We may not be able to respond or correct the code. We only provide the code to help with research projects, but we do not provide any guarantee that the code will work under any circumstances.

9 Release Notes

Version	Date	Description
1.0.0	09/14/2018	Initial release
1.0.2	02/13/2019	Added option 'Adjacency' in `create_graph()`; added 'hybrid alder' perturbation method in `annealing()` and a clear distinction between 'rebranching' and 'random rebranching' with parameter 'NbRandomPerturbations'; corrected issues in 'random' perturbation method in `annealing()`
1.0.4	06/27/2019	- corrected an important bug in `solve_allocation_mobility_cgc()` - added support for partial mobility - corrected bugs in `annealing()` - changed `solve_allocation_mobility()` (slower but more stable) - added illustrative examples from the paper

References

Pablo D. Fajgelbaum and Edouard Schaal. Optimal transport networks in spatial equilibrium. *Working paper*, 2019. 1