Akademie věd České republiky
Ústav teorie informace a automatizace, v.v.i.

Czech Academy of Sciences
Institute of Information Theory and Automation

RESEARCH REPORT

Pavel Hrabák, Ondřej Ticháček

Prediction of Pedestrian Movement During The Egress Situation

Application of Recursive Estimation of High-Order Markov Chains Using the Approximation by Finite Mixtures

No. 2346 January 27, 2015

GAČR 13-13502S

ÚTIA AV ČR, P.O.Box 18, 182 08 Prague, Czech Republic
Tel: (+420)266052422, Fax: (+420)286890378, Url: http://www.utia.cas.cz, E-mail: utia@utia.cas.cz

This report presents a draft of a manuscript, which is intended to be submitted for publication. Any opinions and conclusions expressed in this report are those of the authors and do not necessarily represent the views of the involved institutions.

Abstract
The report summarizes the up-to-now progress in the application of the recursive estimation on the prediction of the pedestrian movement during the egress or evacuation situation. For these purposes a simple decision-making model has been introduced taking into account only the forward and sideways movement of pedestrians. Based on this model, a test simulation has been developed in order to test the applicability of the estimation tool to the stated decision-making model. Two main approaches of the decision process incorporated in the simulation are discussed and a modified version of the original model is presented.

The report contains a manual to the used MATLAB scripts and functions. The codes of needed m-files are incorporated as well.

Keywords:
Recursive estimation, mixture of Markov chains, pedestrian movement, egress simulation.

Acknowledgement:
The work is supported by the Czech Science Foundation, grant number 13-13502S.

Contents

1 Introduction

Modelling and predicting the behaviour of pedestrians in a crowded area or during the evacuation is nowadays very popular and important topic. In order to reproduce the behaviour of pedestrian flow by computer simulations, a variety of microscopic models has been developed [8] ranging from physical force-based models [3, 4] to stochastic cellular models [6, 7]. Very important task is the calibration and validation of the model parametres, which is an important topic of the pedestrian flow community, e.g., within the international conference Pedestrian and Evacuation Dynamics.

The long-term aim of the research, a part of which is presented in this report, is to use existing theory and estimation tool [5] to bring a sufficient method extracting the parameters directly from the microscopic data represented by the data from pedestrian experiments [1, 2].

This report introduces the supposed underlying Markov chain based decision-making model and the progress in the estimation of this process by means of the data obtained by the test simulation. A manual to the created MATLAB scripts and functions together with the source codes are available in the appendix of the report.

1.1 Considered Scenario

The main question to be answered is: "Wheather and how does the information from the pedestrian's neighbourhood affect his behaviour (decision-making process) within the crowd in an egress situation." Two different tactics can be distinguished for pedestrians during the formation of a crowd in front of a bottleneck (an exit):

1. Walking straight on towards the exit and wait for empty space in this direction (this leads to the spontaneous line formation).

2. Trying to walk in tangential direction around the crowd looking for a more advantageous position.

The aim is to study and predict the decision process, which lies beneath the decision, which tactics to choose, based on the information from the pedestrian neighbourhood. Let us consider following scenario. A rectangular room with one exit and possible obstacles is occupied by pedestrians which are motivated to leave the room as fast as possible. This room can be considered as an element of more complex evacuation network, nevertheless, it serves as a starting point for model calibration.

1.2 Description of the Estimation Tool

The below described model of pedestrian decision process has been developed in order to apply the Finite-Mixture-Approximation method [5] to estimate the supposed below-laying high-order Markov chain.

Let us describe the scope of the method. Scalar-valued observations $\Delta_t \in \boldsymbol{\Delta} = \{1, \ldots, |\boldsymbol{\Delta}|\}$ are made at discrete time moments $t \in \mathbf{t} = \{1, ..., |\mathbf{t}|\}$. The observation depends on the regression vector $\psi_t \in \boldsymbol{\psi} = \{1, \ldots, |\boldsymbol{\psi}|\}$. The dependence is assumed to be in the form of high-order Markov chain

$$\mathsf{f}(\Delta_t \mid \Xi, \psi_t) = \prod_{\psi \in \boldsymbol{\psi}} \prod_{\Delta \in \boldsymbol{\Delta}} \Xi_{\Delta|\psi}^{\delta(\Delta,\Delta_t)\delta(\psi,\psi_t)} \,. \tag{1}$$

The regression vector ψ_t is supposed to consist of discrete-valued entries $\psi_{t;i} \in \boldsymbol{\psi}_i = \{1, \ldots, |\boldsymbol{\psi}_i|\}$, $i \in \{1, \ldots, \ell_\psi\}$, where the number of "regressors" ℓ_ψ can be relatively high and the number of regressor values $|\boldsymbol{\psi}_i|$ is supposed to be relatively small.

In order to estimate the dependence (1) it is necessary to estimate $|\boldsymbol{\Delta}| \prod_i |\boldsymbol{\psi}_i|$ values. The described estimation tool reduces the number of estimated values by approximating the dependence (1) by the probabilistic mixture model

$$\mathsf{f}(\Delta_t \mid \Theta, \psi_t) = \sum_i \alpha_i \Theta_{\Delta_t|\psi_{t;i}} \,, \tag{2}$$

which reduces the number of estimated values to $\ell_\psi + \sum_i |\boldsymbol{\psi}_i|$.

Further discussion on the estimation technique is not the subject of this technical report. For more details we refer the reader to Miroslav Kárný (Institute of Information Theory and Automation, CAS, `http://www.utia.cas.cz/people/karny`), the author of the estimation technique described in [5].

2 Model of the Decision-Making

Let us consider a simple egress (or evacuation) situation. A room is occupied by certain number of pedestrians, which are at certain time instructed to evacuate the room through one available escape exit, which is visible to all pedestrians. There may be obstacles in the room, which are to be avoided, e.g. a column, table.

Based on the observation of conducted experiments and personal experience the motion of pedestrians can be described by following simple rules:

- The main intention is to walk towards the exit in the most straightforward direction.
- Deviations from the straight direction are mainly initiated by the inability to walk straight. This is caused by another pedestrian, wall, or obstacle.
- The interaction between pedestrians is more complex then the interaction between a pedestrian and a wall or an obstacle. The decision, whether to deviate from the straight direction, is influenced by the previous motion of obstructing pedestrian as well.
- Information recognized by the pedestrian during the decision process is the state of his neighbourhood, i.e., the occupation of close area, movement of surrounding pedestrians, distance to the wall, obstacle, etc.

From these observations a simple model of the decision process can be abstracted. Let us assume that the motion of a pedestrian consists of steps forward or sideways, i.e., a pedestrian makes either one step (of variable size) forward or one step to the right or to the left. The direction forward, left, or right is not absolute, but relative to the forward direction towards the exit – see figure 1.

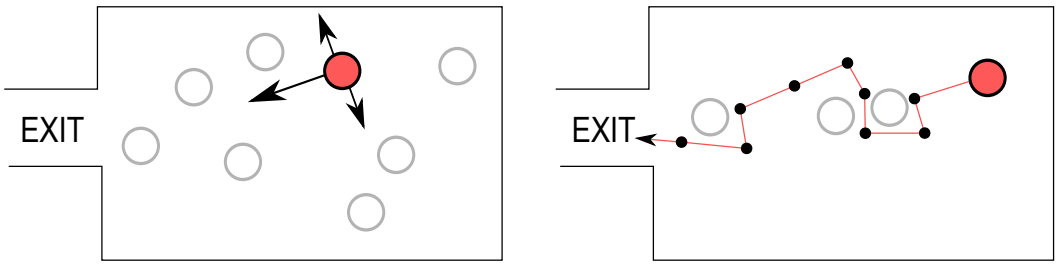

Figure 1: Pedestrians are supposed to move in either forward direction towards the exit or in sideways direction to the left or to the right.

The influence of the neighbourhood state on the decision process is inspired by the concept of cellular models of pedestrian flow [7]. The neighbourhood $\mathcal{N}$ is represented by a rectangular lattice, which consists of $|\mathcal{N}|$ cells. A cell is said to be occupied by a pedestrian if the pedestrian's centre of mass falls into the cell. The position and orientation of the lattice are again not absolute, but relative with respect to the direction towards the exit. Therefore, the lattice does not discretize the motion, it serves to the decision process only. An example of such neighbourhood lattice of 25 cells is depicted in Figure 2 together with the indexation used in further text.

The introduced decision-making model supposes that pedestrians can make four different decisions Δ_t:

1. staying in their current position ($\bullet$),
2. walking one step forward in the exit direction ($\leftarrow$),
3. walking sideways to the right ($\uparrow$),
4. walking sideways to the left ($\downarrow$).

Such decision-making model is inspired by the stepping discretization principle [9].

To meat the notation introduced in Section 1.2, we write $\Delta_t \in \mathbf{\Delta} = \{\bullet, \leftarrow, \uparrow, \downarrow\}$, where the elements of the set $\mathbf{\Delta} = \{1, 2, 3, 4\}$ are changed to $\{\bullet, \leftarrow, \uparrow, \downarrow\}$ for readability reasons only.

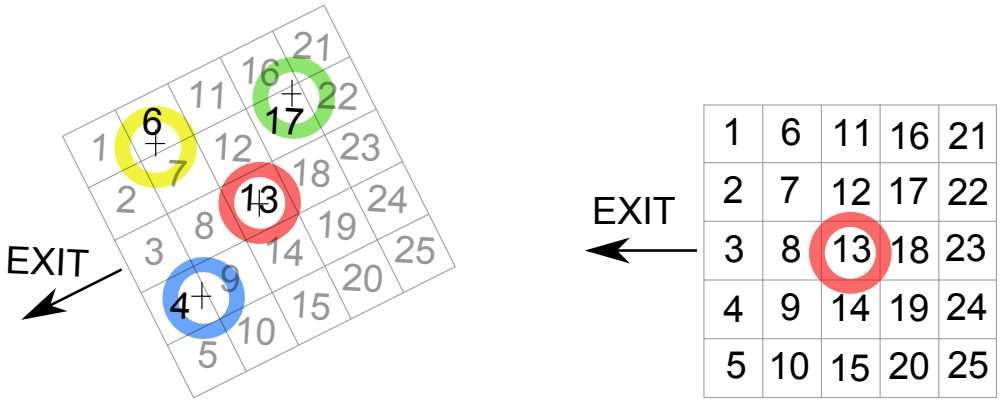

Figure 2: The lattice is rotated parallel to the exit direction. The agent, who is making the decision stands always in the middle cell.

The regression vector ψ_t consists of 25 entries $\psi_{t;i}$, $i \in \{1, \ldots, 25\}$, where the entry $\psi_{t;i}$ represents the state of the cell with index i. The state $\psi_{t;i}$ notes whether the cell is occupied by another pedestrian, what was the last decision of the pedestrian, whether the cell contains a wall or an obstacle. The set $\boldsymbol{\psi}_i$ for $i \in \{1, \ldots, 12, 14, \ldots, 25\}$ consists of 7 different local states:

1. cell is occupied by a pedestrian, which last decision was to stay ($\bullet$),
2. cell is occupied by a pedestrian, which last decision was to walk forward ($\leftarrow$),
3. cell is occupied by a pedestrian, which last decision was to walk to the right ($\uparrow$),
4. cell is occupied by a pedestrian, which last decision was to walk to the left ($\downarrow$),
5. cell is empty (0),
6. cell contains wall, i.e., the cell is outside the "playground" (w),
7. cell contains an obstacle/barrier (b).

Therefore, we can write for readability $\boldsymbol{\psi}_i = \{\bullet, \leftarrow, \uparrow, \downarrow, 0, w, b\}$. The cell with index 13 is always occupied by the pedestrian, who is making the decision. Therefore the corresponding regressor $\psi_{t;13} \in \boldsymbol{\psi}_{13} = \{\bullet, \leftarrow, \uparrow, \downarrow\}$ stores the information of the previous decision of this pedestrian.

To complete the construction of the regression vector ψ_t let us specify the exact meaning of the expression about the cell occupancy and movement direction.

- The cell is occupied by a pedestrian if it contains the center of mass of the pedestrian.
- The cell contains an obstacle if the cell is not occupied by any pedestrian and at least one of the corners of the cell lies in the area of the obstacle.
- The cell contains a wall if it is not occupied by any pedestrian and at least one of the corners of the cell lies outside the area of the room.

An example of such construction is shown in Figure 3.

3 Test Simulation

Although the goal of the research is to apply the estimation method on the video records from experiments performed at FNSPE reference, the first step is to test the method on a simulation record, which is covered by this report. The test simulation is designed according to the above described decision process.

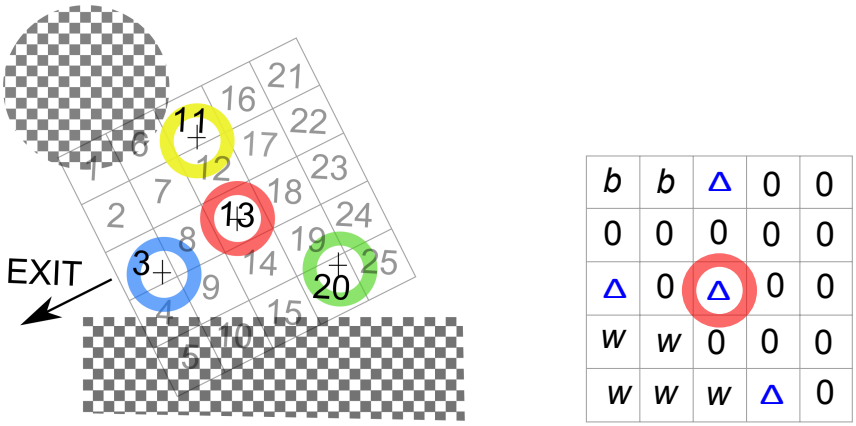

Figure 3: An example of a situation from the point of view of the red agent. Delta stands for the previous decision of the occupying agent.

3.1 Description of the Simulation Principle

As common in the field of microscopic pedestrian modelling, the entities representing pedestrians are referred to as agents. Agents are represented by circles with unit diameter. The scaling to the metric values is possible but not necessary for the test of the method. Agents are distinguishable identified by natural numbers. Nevertheless, the decision process is the same for all agents. The length of the cell edges from the decision lattice is set to $1/\sqrt{2}$ in order to disable two agents to occupy the same cell.

The agent decision and movement are realized in following steps

1. The information of the state of the agent neighbourhood is stored to the regression vector ψ_t.
2. The next movement direction Δ_t is randomly chosen according to the decision process $\mathsf{f}(\Delta_t \mid \Xi, \psi_t)$ with given time-invariant probabilities Ξ.
3. The agent moves in chosen direction, i.e., the agent moves in the direction until the maximal step length is reached or until the motion is blocked by another agent, wall, or obstacle.

The model allows certain compressibility of agent during the step. The agent can "sneak" between two agents, between agent and wall, etc., if the overlap during the motion is under certain threshold and the overlap at the end is under another threshold (usually equal to zero), as depicted in Figure 4.

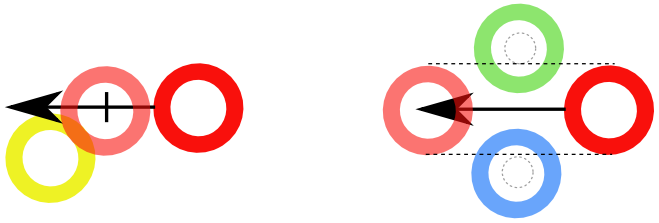

Figure 4: Illustration to the compressibility. An agent can press a little the preceding agent (left) or sneak between one or two agents (right).

An example of one run of the evacuation simulation is shown in Figure 5

3.2 Estimation of the Decision Making Process

The simulation experiment providing data for the estimation was conducted with following parameters. The size of the room was set to 20×10, number of agents to 50. An obstacle (column) was placed in front of the exit. The simulation Was set to run until all agents leave the room and the short adjacent

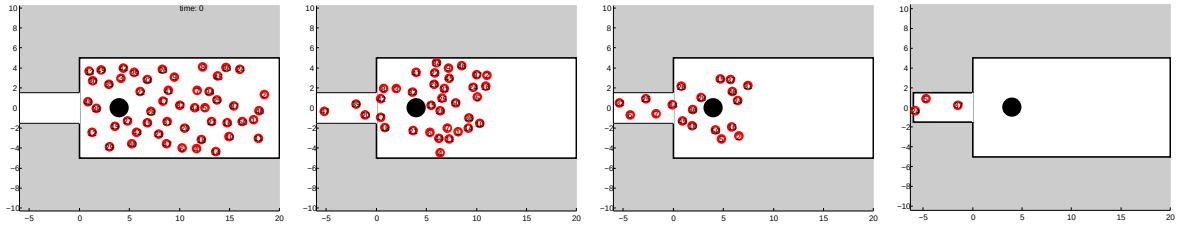

Figure 5: A visualisation of one run of the simulation experiment in time steps $t \in \{0, 20, 40, 60\}$.

hallway. Important statistic of the simulation is the ratio of all decisions made by agents, in this case $P(\bullet, \leftarrow, \uparrow, \downarrow) = (0.046, 0.461, 0.245, 0.248)$.

Crucial feature of the simulation model is the decision process $\mathsf{f}(\Delta \mid \Xi, \psi)$ determining the conditional probability of the decision Δ given that the state of the neighbourhood is ψ. Due to the large number of possible configurations ψ it is a non-trivial task to create in detail the dependence Ξ. Therefore, in the test scenarios a simplified algorithm for the probability computation has been used, where only the occupation of the cell plays role in the decision-making process. Two main approaches were studied: neighbourhood- and component-based decision process. The decision process can be implemented by changing the function `getAgentDecision`. In the appendix, two versions of the decision process are provided within this function.

3.2.1 Neighbourhood-based decision

This approach is referred to as version nine `'v9'`. The decision is build algorithmically based on possible configurations of the neighbourhood, where cells are considered to be either occupied $(\bullet, \leftarrow, \uparrow, \downarrow, w, b)$ or empty (0). 14 cells are considered for the decision, namely cells $\{2, 3, 4, 6, \ldots, 15, 16, 17, 19, 20\}$. The states of these cells are as well used as regression entries for the estimation.

The ratio of exact predictions generated by the estimation tool is, in this case, 23.45 %. Focusing on the divergence or non-divergence from the forward direction (i.e. merge the decisions 'right' and 'left' in the evaluation), the success rate of the prediction increases to 48.59 %. Furthermore, the vector of the estimated influence of the regressors to the decision $\hat{\alpha}$ shows that the main influence on the decision have the cells number 3, 7, and 9 (i.e., regressors with indexes 2, 5, and 7 in the MATLAB code), since

$$\hat{\alpha}_2 = 0.2833, \quad \hat{\alpha}_5 = 0.3615, \quad \hat{\alpha}_7 = 0.3285, \quad \hat{\alpha}_j < 0.01 \quad j \notin \{2, 5, 7\}\,. \tag{3}$$

The estimation of the contribution of these regressors to the decision is given in Table 1

Table 1: Estimation $\hat{\Theta}$ of the contribution of chosen regessors to the decision

	cell 3 (# 2)		cell 7 (# 5)		cell 9 (# 7)	
	occup.	empty	occup.	empty	occup.	empty
$\bullet$	0.1913	0.0073	0.4509	0.0025	0.3716	0.0045
$\leftarrow$	0.0085	**0.9847**	0.0091	0.4670	0.0040	0.2761
$\uparrow$	0.1485	0.0040	0.0023	**0.5287**	**0.6224**	0.0026
$\downarrow$	**0.6518**	0.0040	**0.5378**	0.0019	0.0021	**0.7168**

3.2.2 Component-based decision

This approach is referred to as version ten `'v10'`. The decision is build algorithmically based on the number of occupied cells in three components: front cells $\{2, 3, 4, 7, 8, 9\}$, right cells $\{6, 7, 11, 12, 17\}$, left cells $\{9, 10, 14, 15, 19\}$. Similarly is transformed the estimation tool, where only three regressors are considered correspondingly to the components. The possible entries are therefore $\psi_t \in \{0, 1, \ldots, 6\} \times \{0, 1, \ldots, 5\} \times \{0, 1, \ldots, 5\}$, where each entry denotes number of occupied cells in given component.

The ratio of exact predictions generated by the estimation tool is, in this case, 48.01 %. Focusing on the divergence or non-divergence from the forward direction (i.e. merge the decisions 'right' and 'left'

in the evaluation), the success rate of the prediction increases to 69.81 %. The main influence has the regressor 1, corresponding to the occupancy of front cells $\{2, 3, 4, 7, 8, 9\}$ as seen from the estimation

$$\hat{\alpha} = (\mathbf{0.9014}, 0.0487, 0.0499)\,. \tag{4}$$

The contribution of this component to the decision is shown in Table 2.

Table 2: Estimation $\hat{\Theta}$ of the contribution of regessor 1 (front cells) to the decision

ψ_1	0	1	2	3	4	5	6
•	0.0116	0.0237	0.0167	0.0078	0.0177	0.0230	0.0864
←	**0.8501**	**0.8469**	**0.3258**	0.2666	0.1517	0.0751	0.0118
↑	0.0785	0.0633	**0.3153**	**0.3511**	**0.4568**	0.3164	**0.4345**
↓	0.0598	0.0660	**0.3421**	**0.3745**	0.3737	**0.5855**	**0.4673**

4 Conclusions and Future Plans

The recursive estimation of high-order Markov chains using the approximation by finite mixtures seems to be an appropriate tool for the prediction of the pedestrian decisions within the crowd. Nevertheless, the quality of the prediction is highly influenced by the incorporated decision process included in the simulation model.

This opens two new tasks to be fulfilled before the application of the tool on the experimental data:

1. To study the quality of the estimation with respect to chosen decision process of the simulation model in order to discuss the robustness of the estimation.

2. To apply the estimation on the simulation data obtained by a regressor structure that is different from the structure used in decision process of the simulation (the purpose is to optimize the estimation inputs regardless to the underlying decision process).

Hopefully, the result of such study will be a prediction tool, which can be applicable to the records from pedestrian experiments. Furthermore, such estimation tool can help to calibrate model parameters from the microscopic information provided by this information rather than from the macroscopic observations and aggregated values as so far common.

A Manual to the m-files – How to Use the Program

The program consists of two main scripts:a simulation script `evacuationSimulation.m` and a test script for prediction using the Markov chain mixtures `testMarkovHigh.m`. Both scripts and auxiliary functions are located in the appendix. Appendix B contains the main scripts. Appendix C is composed of functions needed for the simulation, i.e., called by the script `evacuationSimulation.m`. These functions are authored by O. Ticháček. Appendix D consists of functions called by the script `testMarkovHigh.m`. These functions realize the estimation tool described theoretically in [5] and are authored by M. Kárný.

A.1 Readme for `evacuationSimulation.m`

To run the simulation, run the script `evacuationSimulation.m` in MATLAB. The program will run with default settings. The output should be similar to the following snippet[1].

```
>> evacuationSimulation
time 1:      50 active agents
time 2:      50 active agents
...
time 121:    2 active agents
time 122:    1 active agents
stepRatio =
4.5999   46.1289   24.5222   24.7489
```

The variable `stepRatio` stores the ratios [%] of performed observations $\Delta_t \in \{1, \ldots, 4\} \equiv \{\bullet, \leftarrow, \uparrow, \downarrow\}$.

A.1.1 Settings

The simulation setting as variable values, simulation type, visualisation, etc. are globally set within the function `settingsInit.m`. To-be-set variables are listed in Table 3, all variables of this kind are global.

Important settings are:

- `cutOutAfterExit`: Set the variable to
  - `1` (`true`) to exclude the agent from the simulation after he leaves the room through the hallway.
  - `0` (`false`) to maintain the agents in the simulation until the final simulation time step, i.e, all steps of the agent including the forward steps in the corridor behind the exit are taken into account during the estimation.
- `simulationEndMethod`: Set the variable to
  - `'simulate_until_all_exit'` to perform simulation until all agents exit the room.
  - `'for_1_to_Time_of_simulation'` to perform simulation in a for loop for all time steps $\leq$ `TimeOfSimulation`.
- `DRAW.agentPath`: Set the variable to
  - `1` (`true`) to draw also the path of agents' previous movement.
  - `0` (`false`) not to draw the path of agents' previous movement.
- `DRAW.redraw`: Set the variable to
  - `'all'` to redraw all steps of all agents. Warning: this option can be very slow, especially in older versions of MATLAB.
  - `'abs'` to redraw system at times specified in `DRAW.redrawTime`; eg. `DRAW.redrawTime = 1:TimeOfSimulation`
  - `'step'` to redraw system at each time, that is divisible by `DRAW.redrawTimeStep`; eg. `DRAW.redrawTimeStep = 5`

[1] If the lines showing number of active agents in current time step do not appear, check if the variable `Verbose` is set to 1 (true).

Table 3: List of setting variables

variable name	variable type	description
`ROOM_SIZE`	1×2 double	size of the room with agents
`EXIT_WIDTH`	double	width of the exit
`HALLWAY_SIZE`	1×2	size of the hallway
`EXIT_POSITIONS`	cell array of 1×2 double	positions of exits
`EXIT_ORIENTATION`	cell array of char	orientations of exits
`BARRIER_DIAMETER`	double	diameter of barriers
`BARRIER_POSITIONS`	cell array of 1×2 double	positions of barriers
`AGENT_DIAMETER`	double	diameter of agent
`NumberOfAgents`	integer	number of agents in the system
`MovementOverlayAgent`	double	allowed owerlay of two agents while moving (during update)
`StationaryOverlayAgent`	double	allowed owerlay of two agents while stationary (after update)
`MovementOverlayWall`	double	allowed overlay of agent and wall while moving (during update)
`StationaryOverlayWall`	double	allowed owerlay of agent and wall while stationary (during update)
`TimeOfSimulation`	integer	time of simulation
`DRAW`	structure	structure with draw parameters
`UPDATE`	string	update method
`Debug`	logical	debug switcher (0/1)
`Verbose`	logical	verbose switcher (0,1)
`cutOutAfterExit`	logical	cut out agent recording it leaves hallway (0/1)
`simulationEndMethod`	string	simulation stop condition

- `ROOM_SIZE`: adjust room size to `room_length, room_width`
- `EXIT_WIDTH`: adjust the width of the exit to a single value
- `NumberOfAgents`: adjust the number of agents
- `TimeOfSimulation`: adjust the number of simulation time steps

A.1.2 Output Data

The data needed to the further process of the decision estimation are stored in the cell array `data`, containing matrices for all agents `i=1:NumberOfAgents` in the format

$$\texttt{data\{i\}} = \begin{pmatrix} \texttt{observations} \\ \texttt{regressor\_1} \\ \texttt{regressor\_2} \\ \cdots \\ \texttt{regressor\_k} \end{pmatrix} = \begin{pmatrix} \Delta_1^{(i)} & \Delta_2^{(i)} & \cdots & \Delta_{T_i}^{(i)} \\ \psi_{1;1}^{(i)} & \psi_{2;1}^{(i)} & \cdots & \psi_{T_i;1}^{(i)} \\ \psi_{1;2}^{(i)} & \psi_{2;2}^{(i)} & \cdots & \psi_{T_i;2}^{(i)} \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{1;k}^{(i)} & \psi_{2;k}^{(i)} & \cdots & \psi_{T_i;k}^{(i)} \end{pmatrix}.$$

Here the upper index $^{(i)}$ denotes that the element refers to the decision or regression vector corresponding to the decision of agent i. Final ordering of the data to the structure required by the estimator is discussed below. The letter k stands for the number of considered regressors, in our case, the number of cells, from which the influence on the decision is estimated. T_i is the number of steps the agent i spent in the simulation. The cell array is saved into file `dataFF.mat` for further treatment.

A.2 Readme for `testMarkovHigh.m`

The script `testMarkovHigh.m` uses the simulation data in the format of the array `data` to the prediction using the Markov chain mixtures program. Of course, the structure `data` can be filled by different method using different regressors then the above mentioned simulation. The output of the script is the same or similar to the following snippet[2].

```
>> testMarkovHigh
*** settings summary ***
NumberOfAgents: 50
joinData: 1
useDataDifferences: 0
dataLags: []
VERBOSE: 1
DEBUG: 1
forgettype: 3
LRPredictionNoError: 1

factor = ' front_cells '    ' right_cells '    ' left_cells'
hatalpha =     0.9014             0.0487             0.0499

component = front_cells
influence_of_regressor_to_decision =
0.0116    0.0237    0.0167    0.0078    0.0177    0.0230    0.0864
0.8501    0.8469    0.3258    0.2666    0.1517    0.0751    0.0118
0.0785    0.0633    0.3153    0.3511    0.4568    0.3164    0.4345
0.0598    0.0660    0.3421    0.3745    0.3737    0.5855    0.4673
component = right_cells
influence_of_regressor_to_decision =
0.4018    0.5059    0.1200    0.0662    0.1587    0.1548
0.3978    0.0541    0.6131    0.4757    0.4720    0.1969
0.1236    0.4051    0.1626    0.0656    0.2399    0.0618
0.0767    0.0349    0.1043    0.3924    0.1294    0.5865
component = left_cells
influence_of_regressor_to_decision =
0.4415    0.1430    0.0579    0.1082    0.0756    0.2048
0.4690    0.7728    0.7759    0.5244    0.1391    0.0470
0.0350    0.0334    0.0812    0.1564    0.6505    0.7055
0.0545    0.0507    0.0851    0.2111    0.1348    0.0427

count_of_exact_predictions_forgettype_frg =
1482           3           1
ndat_count_toc_sec =
1.0e+03 *
3.0870    0.0788

countOfExactPredicitons: 1482
ratioOfExactPredicitons: 0.4801
countOfExactPredicitons_LRcorrection: 2155
ratioOfExactPredicitons_LRcorrection: 0.6981

ratioOfDataValues =
0.0460    0.4613    0.2452    0.2475
```

[2] If the output shows less information check if `VERBOSE` is set to 1 (true).

A.2.1 How To Read the Results

To evaluate the goodness of the prediction, it is important to discuss the ratio of correctly predicted decisions. The value `dispResult.ratioOfExactPredicitons` denotes the ratio of situation, where the estimation tool predicted exactly the next decision performed by the simulation model. Since we are interested whether the pedestrian under given conditions maintains the forward direction or rather makes a step aside, regardless to the direction, it is usable to consider the approach in which the decision to the right and to the left are considered as one (in the evaluation only). Ratio of such correct decisions is stored in value `dispResult.ratioOfExactPredicitons_LRcorrection`.

The estimated influence of given regressor is stored in variable `hatalpha`. The estimation of the decision probabilities $\hat{\Theta}$ is displayed as `influence_of_regressor_to_decisions`. Each row corresponds to one of the four the decisions $\Delta \in \{1,\ldots,4\} \equiv \{\bullet, \leftarrow, \uparrow, \downarrow\}$; each column to possible value of the regressor. The information for regressor k is stored in the variable `Mix(k).V_del_psi` – the values need to be normalized by

```
>> Mix(k).V_del_psi/diag(sum(Mix(k).V_del_psi))
```

Supplementary information is provided by five graphs: processed data (scatter), predicted values (histogram), prediction and filtering errors (scatter), prediction errors (histogram) and forgetting (scatter).

A.2.2 Settings

Settings are located at the beginning of the script `testMarkovHigh.m`. Options to be set are listed in Table 4. The commonly changed variables are described in more detail below.

Table 4: List of settings of the estimation

variable name	variable type	description
`VERBOSE`	logical	verbose output 0/1 = NO/YES
`DEBUG`	logical	debugging 0/1 = NO/YES
`forgettype`	integer	switch between forgettings [1,2,3] = [fixed, global, partial]
`lam`	double	fixed forgetting factor
`laml`	double	lower bound on forgetting factor
`countupper`	integer	upper bound on the number of iterations
`eps`	double	precision in numerics
`joinData`	logical	join data for agents
`useDataDifferences`	logical	use time changes of data instead of absolute values
`dataLags`	vector of integer	use data lags of specified order
`LRPredictionNoError`	logical	do not count as error if prediction is Left/Right and the predicted value is Right/Left

- `joinData`: Set the variable to
  - `1` (`true`) to join data for all agents. The matrices `data{i}` matrices for all agents will be grouped together to form one data matrix.

    $$\texttt{DATA} = \big(\texttt{data\{1\}} \quad \texttt{data\{2\}} \quad \ldots \quad \texttt{data\{end\}}\big)$$

    This means that decisions of all agents are understood as one realization of the stochastic process.
  - `0` (`false`) to evaluate each agent separately.
- `useDataDifferences`: Set the variable to
  - `1` (`true`) to predict changes rather than absolute values. Let $\{a_1, a_2, ..., a_n\}$ be the possible values of the categorical data. The difference (change from a_i to a_j) should not be equal to any other difference (change from a_k to a_l) for every $i \neq k$ and $j \neq l$, $i \neq j$, $k \neq l$. Furthermore, the difference (change from a_i to a_i) should be same for all categorical values, i.e. a *zero*. Therefore, the set $\{a_1, a_2, ..., a_n\}$ is mapped to $\{b_1, b_2, ..., b_n\} = \{2^1, ..., 2^n\}$ and differences are

obtained in simple algebraic manner, i.e. $\mathrm{diff}(a_i, a_j) = b_i - b_j$. If the variable item `joinData` is set to `1` (`true`), the differences are computed first (for each agent separately) and joined afterwards.

  - `0` (`false`) to predict absolute values of the decisions.

- `dataLags`: Set the variable to
  - `[a, b, c]` to use lagged data of orders `a`, `b` and `c`. The data set will be shortened by the maximal value of the vector or lag orders; the lagged data will be added as regressors (as last rows of the data matrix).
  - `[]` not to use lagged data.

- `LRPredictionNoError` Set the variable to
  - `1` (`true`) to not count as error if prediction is Left/Right and the predicted value is Right/Left; this may only be used if `useDataDifferences` is set to `0` (`false`);
  - `0` (`false`) to count Left/Right prediction error as an error

B Matlab m-files – Main Scripts

B.1 evacuationSimulation

```
%% evacuationSimulation
%% Description
% evacuationSimulation simulates interaction between agents in room
    evacuation
%% Update history
%% Code starts here
clear all
close all
%% Inicialization of random number generator
if verLessThan('matlab', '8.4')
     rand('twister');
else
     rng('shuffle');
end
%% Initial settings of simulation parameters
global ROOM_SIZE EXIT_WIDTH HALLWAY_SIZE EXIT_POSITIONS EXIT_ORIENTATION
global AGENT_DIAMETER NumberOfAgents
global MovementOverlayAgent StationaryOverlayAgent
global MovementOverlayWall StationaryOverlayWall
global TimeOfSimulation
global DRAW UPDATE Debug Verbose
global cutOutAfterExit simulationEndMethod NamesOfRegressors
settingsInit();
[agents, positions] = agentsCon(NumberOfAgents, 'uniform');

%% Inicialization of figure
figure;
drawRoom(ROOM_SIZE, HALLWAY_SIZE, EXIT_WIDTH);
hold on
for i = 1:NumberOfAgents
    agents(i).handles = drawAgent(agents(i),i,1);
end
handle_time_text = text(10,10,sprintf('time: %d',0));
   % show current time step in the figure
```

```
pause(eps)
%% simulation
% initialize agents permutation
agentsPermutation = randperm(NumberOfAgents);
% initialize index of previously updated agent
previous_index = [];
% initialize number of agents to be updated
NumberOfActiveAgents = length(agentsPermutation);
% the simulation will continue while continueSimulation = 1
continueSimulation = 1;
% initialize time step value
t = 0;
while continueSimulation
    % increment time step
    t = t + 1;
    if Verbose
        % print some simulation progress statistics
        fprintf('time %d: \t %d active agents\n', t,
           NumberOfActiveAgents)
    end
    % update current time step shown in the figure
    handle_time_text.String = sprintf('time: %d',t);
    % update all agents in given order -- given by UPDATE
    switch UPDATE
        case 'shuffle'
            NumberOfActiveAgents = length(agentsPermutation);
            agentsPermutation = agentsPermutation(randperm(
               NumberOfActiveAgents));
        case 'frozen_shuffle'
            agentsPermutation = agentsPermutation;
    end
    % update all agents in the order given by agentsPermutation
    for i = agentsPermutation
        agents(i).activeTime = t;
        if Debug
            % print out some info
            fprintf('\t updating agent %d \n', i);
        end
        % compute desired movement direction
        agents(i).direction{t} = movementDirection(agents(i).position{t
           },...
                                   EXIT_POSITIONS{agents(i).exit});
        if verLessThan('matlab', '8.4')
            % get the neighborhood of agent i
            [neighborhood,positionGrid,wallGrid,barrierGrid,
               positionOfOtherAgents] ...
                 = neighborhoodOfAgent(agents, i, t, positions);
            % save neighborhood, positions of other agents in grid,
               positions of walls
            % and barriers in grid and relative positions of other
               agents
            agents(i).neighborhood{t} = neighborhood;
            agents(i).positionGrid{t} = positionGrid;
            agents(i).wallGrid{t} = wallGrid;
            agents(i).barrierGrid{t} = barrierGrid;
            agents(i).positionOfOtherAgents{t} = positionOfOtherAgents;
```

```
            % get the decision of agent based on neighborhood
            [step, directionDecision, regressors] = ...
               getAgentDecision(agents(i).neighborhood{t}, agents(i).
                  direction{t});
            % save decided step (stand/forward/right/left) and the
               direction
            agents(i).step{t} = step;
            agents(i).directionDecision{t} = directionDecision;
            agents(i).regressors{t} = regressors;
        else
            [ agents(i).neighborhood{t}, ...
              agents(i).positionGrid{t}, ...
              agents(i).wallGrid{t}, ...
              agents(i).barrierGrid{t}, ...
              agents(i).positionOfOtherAgents{t} ]  ...
                 = neighborhoodOfAgent(agents, i, t, positions);
            [ agents(i).step{t}, ...
              agents(i).directionDecision{t}, ...
              agents(i).regressors{t} ]...
            = getAgentDecision(agents(i).neighborhood{t}, agents(i).
               direction{t});
        end
        if Debug
            % pause simulation until a button is pressed
            w = waitforbuttonpress;
        end
        % move agent (update position)
        agents(i).position{t+1} = moveAgent(agents, i, t, positions);
        positions(i,:) = agents(i).position{t+1};
        if cutOutAfterExit
            if agents(i).position{t+1}(1) < - HALLWAY_SIZE(1) % agent
               left the hallway
                % remove from updating
                agentsPermutation(agentsPermutation == i) = [];
                % set position to inf
                agents(i).position{t+1} = [Inf, Inf];
                positions(i,:) = agents(i).position{t+1};
            end
        end
        % redraw system image representation after one agent's step
        if isa(DRAW.redraw, 'char')
            if strcmp(DRAW.redraw,'all')
                if verLessThan('matlab', '8.4')
                    hold off
                    drawSystem( agents, [], [] ); % redraw
                else
                    drawSystem( [], agents(i), agents(previous_index) );
                       % move last agent
                end
                pause(eps)
                previous_index = i;
            end
        end
    end
    % redraw system image representation after one time step
    if isa(DRAW.redraw, 'char')
```

```
        drawNow = 0;
        % decide whether to update waitbar, redraw image or do nothing
        if strcmp(DRAW.redraw,'abs')
            if DRAW.redrawTime == TimeOfSimulation
                % update wait bar
                handle_waitbar  = mywaitbar(t/TimeOfSimulation);
            end
            if find(DRAW.redrawTime == t)
                drawNow = 1;
            end
        elseif strcmp(DRAW.redraw,'step')
            if mod(t,DRAW.redrawTimeStep) == 0
                drawNow = 1;
            end
        end
        % redraw image
        if drawNow
            hold off
            if verLessThan('matlab', '8.4')
                figure;
            end
            drawSystem( agents, [], [] ); % redraw
            if ~verLessThan('matlab', '8.4')
                snapnow;
            end
            pause(eps)
        end
    end
    % decide whether to continue simulation
    if strcmp(simulationEndMethod, 'simulate_until_all_exit')
        continueSimulation = NumberOfActiveAgents > 0;
    elseif strcmp(simulationEndMethod, 'for_1_to_Time_of_simulation')
        continueSimulation = t < TimeOfSimulation &&
           NumberOfActiveAgents > 0;
    end
end
if exist('handle_waitbar', 'var')
    % close wait bar
    close(handle_waitbar)
end
%% gather simulation results
% gather simulation data
[data, miscellaneous] = getSimulationData(agents);
% save data in format for markovHigh
save('dataFF.mat','data','miscellaneous');
% display the steps counts and ratios
allSteps = [];
for i = 1:NumberOfAgents
    allSteps = [ allSteps ,cell2mat(agents(i).step)];
end
figure
if verLessThan('matlab', '8.4')
    hist(allSteps)
else
    histogram(allSteps)
end
```

```
c = zeros(1,max(allSteps));
for j = 1:length(allSteps)
    c(allSteps(j)) =  c(allSteps(j)) +1;
end
stepRatio = c./sum(c)*100
```

B.2 testMarkovHigh

```
%% testMarkovHigh tests learning Markov chain mixture describing room
   evacuation
%% Category: learning
%% Description
%% Theory
%  sip/publications/2015/markovmixtures2.tex
%% Update history
%% Code starts here
clear all
close all
global DATA TIME DEBUG
%% Tailoring of experiments
VERBOSE = 1;   % verbose output 0/1 = NO/YES
DEBUG = 1;     % debugging 0/1 = NO/YES
forgettype = 3; % switch between forgettings
                % [1,2,3] = [fixed, global, partial] = [worse,
                    comparable, best]
lam   = 0.99;          % fixed forgetting factor
laml  = 0.75;          % lower bound on forgetting factor
countupper = 100;      % upper bound on the number of iterations
eps   =  1e-5;         % precision in numerics
% DATA
%       decision:                  levels 1,...,4
%       neghborhood regressors:    levels 1,...,n
%       decision lags:             levels 1,...,4
joinData = 1;      % join data for agents
% joinData = 0;
% useDataDifferences = 1;   % use time changes of data instead of
   absolute values
useDataDifferences = 0;
LRPredictionNoError = 1;  % do not count as error if prediciton is Left/
   Right
% and the predicted value is Right/Left; only if useDataDifferences = 0;
% LRPredictionNoError = 0;
% dataLags = 1;    % use lagged data of order 1
% dataLags = [1, 3, 5];  % use lagged data of order 1, 3 and 5
dataLags = [];      % do not use lagged data
% Initialization
load('dataFF')
NumberOfAgents = length(data);
if joinData             % join data for agents
    DATA = [];
    for i = 1:NumberOfAgents
        if useDataDifferences    % use time changes of data instead of
           absolute values
            data_tmp = data{i};
            data_tmp = dataDifferences(data_tmp);
        else
```

```
            data_tmp = data{i};
        end
        DATA = [ DATA,  data_tmp ];
    end
    dataToEvaluate = 1;
else
    dataToEvaluate = 1:NumberOfAgents;
end
%% display settings summary
fprintf('*** settings summary ***\n\n')
s.NumberOfAgents = NumberOfAgents;
s.joinData = joinData;
s.useDataDifferences = useDataDifferences;
s.dataLags = dataLags;
s.VERBOSE = VERBOSE;
s.DEBUG = DEBUG;
s.forgettype = forgettype;
s.LRPredictionNoError = LRPredictionNoError;
disp(s)
for indexOfAgent = dataToEvaluate
    if ~joinData
        DATA = data{indexOfAgent};
        if useDataDifferences % use time changes of data instead of
           absolute values
            DATA = dataDifferences(DATA);
        end
    end
    DATA = incrementalDataValues(DATA); % shift data to incremental
       values 1,...,n
    ndat = size(DATA,2);
    nreg = size(DATA,1) - 1;
    % lagged data
    if ~isempty(dataLags)
        laggedDATA = [];
        maxLag = max(dataLags);
        for lag = [dataLags]
            dataRange = (maxLag - lag) + 1 : ndat - lag;
            laggedDATA = [laggedDATA; DATA(1,dataRange)];
        end
        dataRange = (maxLag - 0) + 1 : ndat - 0;
        DATA = [ DATA(:,dataRange); laggedDATA ];
    ndat = ndat - max(dataLags);
    end
    %% Construction of the mixture
    lenpsi    = 0;  % length of the regression vector
    Mix       = [];  % mixture
    % predicted decisions
    delchn    = 1;   % channel with predicted variable
    delcar    = max(DATA(1,:));  % cardinality of the values of the
       predicted variable
    % gradual construction of the mixture
    for i = 2 : nreg+1
        psichn    = i;   % channel with the regression vector entry
        psicar    = max(DATA(i,:)); % the number of the values of the
           regression vector entry
        psidelay = 0;    % the delay of the regression vector entry
```

```
        mfac     = mfaccon(delchn,delcar,psichn,psicar,psidelay); %
           construct factor
        Mix      = [Mix,mfac];  % extend the mixture
        lenpsi   = lenpsi+1;    % increase regression vector length
    end
    % lagged data
    if ~isempty(dataLags)
        for lag = dataLags
            i = i + 1;
            nreg = nreg + 1;      % increase the number of regressors
            psichn   = i;         % channel with the regression vector
               entry
            psicar   = max(DATA(i,:));
                   % the number of the values of the regression vector
                      entry
            psidelay = 0;     % the delay of the regression vector entry
            mfac     = mfaccon(delchn,delcar,psichn,psicar,psidelay);
            % construct factor
            Mix      = [Mix,mfac];     % extend the mixture
            lenpsi   = lenpsi+1;    % increase regression vector length
        end
    end
    % Construct the weights
    mwei = mweicon(lenpsi);   % weights
    %% Arrays for graphical outputs
    fig = 0;       % figure counter
    predfilterrors = zeros(4,ndat);  % prediction and filtering errors
    if forgettype > 2
       forgetting_factor = lam*ones(lenpsi+1,ndat);   % forgetting
          factor
    else
       forgetting_factor = lam*ones(1,ndat);
    end
% Simulation start
    tic
    for TIME = 1:ndat
        handle_waitbar = mywaitbar( TIME/ndat );
        psit = mgetpsi(Mix);    % get current regression vector
        delt = DATA(1,TIME);    % gets the predicted variable
        [rhskap,rhsV] = mrhs(Mix,mwei,delt,psit);
        % preparation of right-hand sides used in updating and
           projection
           mweiold = mwei;   % store the current values serving to
              forgetting
           Mixold  = Mix;   % store the current components for
              forgetting
         mwei.kappa = moptn(mwei.kappa',rhskap)'; % update and project
            weights kappa
        for k =1:lenpsi
            Mix(k).V_del_psi(:,psit(k)) =...
            moptn(Mixold(k).V_del_psi(:,psit(k)),rhsV(:,k)); % update
               and project Vs
        end
%% Predictions
        mpreddel    = mpred(Mix,mwei,psit);
        mpreddelold = mpred(Mixold,mweiold,psit);
```

```
        predold = round([1:delcar]*mpreddelold);
        pred = round([1:delcar]*mpreddel);
        predfilterrors(1,TIME) = delt - predold;      % prediction error
        predfilterrors(2,TIME) = delt - pred;         % filtering error
        predfilterrors(3,TIME) = predfilterrors(1,TIME);
        % prediction error not-diferentiating left and right movement
        predfilterrors(4,TIME) = predfilterrors(2,TIME);
        % filtering error not-diferentiating left and right movement
        if useDataDifferences == 0 && LRPredictionNoError == 1
            % correction of error value for L-R or R-L error
            if delt == 3 || delt == 4
                if predold == 3 || predold == 4
                    predfilterrors(3,TIME) = 0;
                end
                if pred == 3 || pred == 4
                    predfilterrors(4,TIME) = 0;
                end
            end
        end
%% Forgetting
        if forgettype == 1   % fixed forgetting
            mwei.kappa = lam*mwei.kappa +(1-lam)*mweiold.kappa;
            for k = 1:lenpsi
                Mix(k).V_del_psi(:,psit(k))=lam*Mix(k).V_del_psi(:,psit(
                   k))+...
                   (1-lam)*Mixold(k).V_del_psi(:,psit(k));
            end
        else
            if forgettype == 2  % global data-dependent forgetting
                [Mix,mwei,lam] = mforgetglobal(Mix,mwei,Mixold,mweiold,
                   ...
                   delt,psit,countupper,eps,laml);
            else
                [mwei.kappa,lam] = mforget(mwei.kappa,mweiold.kappa,...
                   countupper,eps,laml); % partial data-dependent
                      forgetting factors
                for k = 1:lenpsi
                    [Mix(k).V_del_psi(:,psit(k)),forgetting_factor(k+1,
                       TIME)]=...
                    mforget(Mix(k).V_del_psi(:,psit(k)),...
                    Mixold(k).V_del_psi(:,psit(k)),countupper,eps,laml);
                end
            end
        end
        forgetting_factor(1,TIME) = lam;
        % time course of the forgetting factor on kappa (and common)
    end     % simulation end
end
%% Graphical outputs
    close(handle_waitbar);
    close all

    figure
    plot(DATA','.')
    title('processed data')
    xlabel('time')
```

```
    ylabel('data')

    figure
    hist(DATA(1,:))
    title('predicted values')
    xlabel('values')
    ylabel('occurrences')

    figure
    plot(predfilterrors','.')
    title('prediction and filtering errors')
    xlabel('time')
    ylabel('errors')

    figure
    hist(predfilterrors(1,:))
    title('prediction errors')
    xlabel('values')
    ylabel('occurrences')

    figure
    if forgettype<3
       plot([1:ndat],forgetting_factor,'b.')
    else
        hold on
        for i = 1:nreg
            scatter([1:ndat],forgetting_factor(i,:));
        end
        hold off
    end

    title('forgetting')
    xlabel('time')
    ylabel('forgetting factor')

%% Numerical outputs
    % regressor names loaded from dataFF.mat
    factor =  miscellaneous.NamesOfRegressors;
    % update factor if data lags were used in estimation
    if ~isempty(dataLags)
        for lag = dataLags
            factor = [ factor, sprintf('time-%d',lag) ];
        end
    end
    hatalpha = mwei.kappa/sum(mwei.kappa);   % estimated component
       weights
    if VERBOSE
        factor
        hatalpha
        % [ factor ; arrayfun(@num2str, hatalpha, 'unif', 0) ]
    end
    for k = 1:lenpsi
       component = factor{k};
       influence_of_regressor_to_decision =...
           Mix(k).V_del_psi/diag(sum(Mix(k).V_del_psi));    % estimated
              components
```

```
        if VERBOSE
            component
            influence_of_regressor_to_decision
        end
    end
    count = sum( predfilterrors(1,:) == 0 );
          % counting of the number of exact predictions
    countLR = sum( predfilterrors(3,:) == 0 );
          % counting of the number of exact predictions with LR
             correction
    count_of_exact_predictions_forgettype_frg =[count,forgettype,lam];
    ndat_count_toc_sec = [ndat,toc];
          % the number of observations and elapsed time
    if VERBOSE
        count_of_exact_predictions_forgettype_frg
        ndat_count_toc_sec
    end
    c(indexOfAgent) = count;
    n(indexOfAgent) = ndat;
    if ~joinData
        dispResult.indexOfAgent = indexOfAgent;
    end
    dispResult.countOfExactPredicitons = count;
    dispResult.ratioOfExactPredicitons = count/ndat;
    dispResult.countOfExactPredicitons_LRcorrection = countLR;
    dispResult.ratioOfExactPredicitons_LRcorrection = countLR/ndat;
    fprintf('\n');
    disp(dispResult)
if ~joinData
    meanRatioOfExactPredictions = sum(c)/sum(n)
end
for j = 1:delcar
    d(j) =  sum(DATA(1,:) == j);
end
ratioOfDataValues = d./sum(d)
```

C Matlab m-files – Test Simulation

C.1 settingsInit

```
function [ ] = settingsInit( )
%% Description: Inicializes global variables used in the simulation
%% Code starts here
%% agent parameters
global AGENT_DIAMETER NumberOfAgents
AGENT_DIAMETER = 1;
NumberOfAgents = 50;
global MovementOverlayAgent StationaryOverlayAgent
global MovementOverlayWall StationaryOverlayWall
global MovementOverlayBarrier StationaryOverlayBarrier
MovementOverlayAgent = 0.5*AGENT_DIAMETER;
StationaryOverlayAgent = 0;
MovementOverlayWall = 0.25*AGENT_DIAMETER;
StationaryOverlayWall = 0;
MovementOverlayBarrier = 0.5*AGENT_DIAMETER;
StationaryOverlayBarrier = 0;
```

```
global maxStepSize
maxStepSize = AGENT_DIAMETER * pi/2;
%% room parameters
global ROOM_SIZE EXIT_WIDTH HALLWAY_SIZE EXIT_POSITIONS EXIT_ORIENTATION
global BARRIER_DIAMETER BARRIER_POSITIONS
ROOM_SIZE = [20, 10];
HALLWAY_SIZE = [6, 3];
EXIT_WIDTH = 3;
EXIT_POSITIONS = {[0, 0]};
EXIT_ORIENTATION = {'y'};
BARRIER_POSITIONS = {[4*AGENT_DIAMETER, 0]};
BARRIER_DIAMETER = 2*AGENT_DIAMETER;
%% simulation parameters
global TimeOfSimulation
TimeOfSimulation = 50;
global DRAW
%% visualization settings
% DRAW.agentPath =
%   1     : draw also the history of agents' movement
%   0     : do NOT draw the history of agents' movement
% DRAW.redraw =
%   'all'  : redraw all steps of all agents
%   'abs'  : redraw system at times specified in DRAW.redrawTime
%   'step' : redraw system at each time, that is divisible by DRAW.
   redrawTimeStep
DRAW.agentPath = 1;
DRAW.redraw = 'all';
DRAW.redrawTimeStep = 20;
global UPDATE
UPDATE = 'shuffle';
% UPDATE = 'frozen_shuffle';
global Debug Verbose
Debug = 0; % turn debuging on or off (1/0)
Verbose = 0; % turn verbose output on or off (1/0)
global cutOutAfterExit simulationEndMethod
cutOutAfterExit = 1; % cut out agent recording after it leaves hallway
% cutOutAfterExit = 0;
% simulationEndMethod = 'simulate_until_all_exit';
   % perform simulation until all agents exit the room
simulationEndMethod = 'for_1_to_Time_of_simulation';
   % perform simulation in a for loop for all time steps <=
      TimeOfSimulation
end
```

C.2 getSimulationData

```
function [ data, miscellaneous ] = getSimulationData( agents )
%% Description
% Gathers some simulation data in single matrix for markovHigh decision
   estimation
%%% Inputs
%       agents          = structure with all agents data
%%% Outputs
%       data            = matrix with regressors/neigborhood and
   decisions
%       miscellaneous   = miscellaneous data
```

```
%% Code starts here
%% inicialization
global NumberOfAgents TimeOfSimulation
global NamesOfRegressors
global simulationDataMethod RelevantNeighborhoodFields
switch simulationDataMethod
    case 'neighborhood'
        sizeOfNeighborhood = numel(agents(1).neighborhood{end});
    case 'regressors'
        sizeOfRegressors = numel(agents(1).regressors{end});
end
%% get relevant data in two for loops
for i = 1:NumberOfAgents
    decision = NaN(1,agents(i).activeTime);
    switch simulationDataMethod
        case 'neighborhood'
            neighborhood = NaN(sizeOfNeighborhood, agents(i).activeTime)
                ;
        case 'regressors'
            regressors = NaN(sizeOfRegressors, agents(i).activeTime);
    end
    for t = 1:agents(i).activeTime
        decision(1,t) = agents(i).step{t};
        switch simulationDataMethod
            case 'neighborhood'
                ind = 1 : sizeOfNeighborhood;
                neighborhood(ind,t) = reshape(agents(i).neighborhood{t},
                    ...
                                            [sizeOfNeighborhood, 1]);
        %         neighborhood(ind,t) = reshape(logical(agents(i).
            positionGrid{t}),...
                                            [sizeOfNeighborhood, 1]);
                if t > 1
                    neighborhood(13,t) = agents(i).step{t-1};
                else
                    neighborhood(13,t) = 1; % standing in t = 0
                end
             case 'regressors'
                jnd = 1 : sizeOfRegressors;
                regressors(jnd,t) = reshape(agents(i).regressors{t}, [
                    sizeOfRegressors, 1]);

        end
    end
    switch simulationDataMethod
        case 'neighborhood'
%             map neighborhood to {1,...,6} instead of {-1,...,4}
%             neighborhood(neighborhood == 0) = 5;
%             neighborhood(neighborhood == -1) = 6;

%             map the neighborhood to {1,2}
            neighborhood(neighborhood ~= 0 & ~isnan(neighborhood)) = 1;
            neighborhood(neighborhood == 0) = 2;
%             only relevant points of the neighborhood
            data{i} = [ decision; ...
                        neighborhood(RelevantNeighborhoodFields,:)];
```

```
%             set names of regressors (for testMarkovHigh.m)
            for ind = 1:length(RelevantNeighborhoodFields)
                miscellaneous.NamesOfRegressors{ind} =...
                            sprintf('n(%d)',RelevantNeighborhoodFields(
                               ind));
            end
        case 'regressors'
            data{i} = [ decision; ...
                        regressors ];

%             set names of regressors (for testMarkovHigh.m)
            miscellaneous.NamesOfRegressors = NamesOfRegressors;

    end
end
end
```

C.3 agentsCon

```
function [ agents, positions ] = agentsCon( NumberOfAgents, distribution
    )
%% Description: Constructor of agents performing initial distribution in
    space
% Inputs
%       NumberOfAgents  = number of agents (integer)
%       distribution    = setting of space distribution of agents (
   string)
% Outputs
%       agents          = structure with data about all agents
%       positions       = current positions of all agents (matrix)
%% Code starts here
global ROOM_SIZE EXIT_WIDTH AGENT_DIAMETER TimeOfSimulation
global EXIT_POSITIONS BARRIER_POSITIONS BARRIER_DIAMETER
%% initialization of the structure with agents
agents = struct('position',{},...
                'step',{},...
                'direction',{},...
                'directionDecision',{},...
                'handles',{},...
                'neighborhood',{},...
                'positionGrid',{},...
                'wallGrid',{},...
                'barrierGrid',{},...
                'positionOfOtherAgents',{},...
                'regressors',{},...
                'activeTime',{},...
                'exit',{});
% initialize the size of the structure
agents(NumberOfAgents).position = [];
%% settings based on desired distribution of agents
[xroom_range, yroom_range] = roomRange( ROOM_SIZE, AGENT_DIAMETER/2 );
switch distribution
    case 'uniform' % uniform distribution in the room
        X_range = xroom_range;
        Y_range = yroom_range;
        randomDistribution = 1;
```

```
    case 'uniform-back' % uniform distribution in the back third of the
        room
         X_range = [xroom_range(1) + ROOM_SIZE(1)*(2/3), xroom_range(2)];
         Y_range = yroom_range;
         randomDistribution = 1;
end
%% distribute agents in the room
% initialize exits
numberOfExits = length(EXIT_POSITIONS);
exitProbabilities = ones(1,numberOfExits) ./ numberOfExits;
% initialize positions matrix
positions = zeros(NumberOfAgents, 2);
if randomDistribution == 1
    for i = 1:NumberOfAgents % iterate through agents
        agents(i).exit = randIntProb( exitProbabilities );
           % asign exit number to agent
        agent_placed = 0; % agent is not placed yet
        time = 1;
        agents(i).activeTime = time;
        agents(i).step{time} = 1; % 1 ... agent was standing still
        while ~agent_placed % repeat until the agent is placed
            % try to place the agent
            agents(i).position{time}(1) = RANDX( [1,1], X_range );
            agents(i).position{time}(2) = RANDX( [1,1], Y_range );
            positions(i,:) = agents(i).position{time};
            % first check distance form barriers
            distance_from_barrier_too_low = 0;
            for b = 1:length( BARRIER_POSITIONS )
                if norm( agents(i).position{time} - BARRIER_POSITIONS{b}
                    ) ...
                       < ( BARRIER_DIAMETER + AGENT_DIAMETER )/2
                    distance_from_barrier_too_low = 1;
                end
            end
            % if the distance from bariers is ok
            if ~distance_from_barrier_too_low
                % get the distace from other placed agents
                distance = arrayfun(@(a) norm( a.position{time}...
                             - agents(i).position{time} ), agents(1:i-1)
                                ...
                                );
                % if no distance less than AGENT_DIAMETER => correct
                   placement
                if ~any(distance < AGENT_DIAMETER)
                    agent_placed = 1;
                end
            end
        end
    end
end
end
```

C.4 roomRange

```
function [ xroom_range, yroom_range ] = roomRange( room_size, border )
%% Description: computes room borders - an area where agents can be
   located
```

```
%%% Inputs
%       room_size    = size of the room (2x1 double)
%       border       = padding (double)
%%% Outputs
%       xroom_range = room range in x axis (2x1 double)
%       yroom_range = room range in y axis (2x1 double)
%% Code starts here
if nargin == 1
    border = 0;
end
xroom_range = [ border , room_size(1) - border ];
yroom_range = [ border - room_size(2)/2 , room_size(2)/2 - border ];
end
```

C.5 movementDirection

```
function [ direction, target ] = movementDirection( position,
   exitPosition )
%% Description: Computes the desired direction based on agent's position
   .
%     The room is divided into three parts, for each there is a
   different target
%%% Inputs:
%       position    = position of agent (2x1 matrix)
%%% Outputs:
%       direction   = desired direction (2x1 matrix)
%       target      = target of agent (2x1 matrix)
%% Code starts here
global EXIT_WIDTH AGENT_DIAMETER ROOM_SIZE
    target = exitPosition;
    % differ between exit positions
    if (exitPosition(2) == ROOM_SIZE(2)/2) || (exitPosition(2) == -
       ROOM_SIZE(2)/2)
        direction = [0, sign(exitPosition(2))];
        target_right = [ exitPosition(1) + EXIT_WIDTH/2 - AGENT_DIAMETER
           /2, ...
                         exitPosition(2) - sign(exitPosition(2)) *
                            AGENT_DIAMETER/2 ];
        target_left  = [ exitPosition(1) - EXIT_WIDTH/2 + AGENT_DIAMETER
           /2, ...
                         exitPosition(2) - sign(exitPosition(2)) *
                            AGENT_DIAMETER/2 ];
        if position(end,1) > target_right(1)
            direction = target_right - position;
            target = target_right;
        elseif position(end,1) < target_left(1)
            direction = target_left - position;
            target = target_left;
        end
    else
        if exitPosition(1) == 0
            direction = [-1,0];
            target_top    = [ exitPosition(1) + AGENT_DIAMETER/2 ,...
                              exitPosition(2) + EXIT_WIDTH/2 -
                                 AGENT_DIAMETER/2 ];
            target_bottom = [ exitPosition(1) + AGENT_DIAMETER/2 ,...
```

```
                                exitPosition(2) - EXIT_WIDTH/2 +
                                   AGENT_DIAMETER/2 ];
        elseif exitPosition(1) == ROOM_SIZE(1)
            direction = [1,0];
            target_top    = [ exitPosition(2) + EXIT_WIDTH/2 -
               AGENT_DIAMETER/2, ...
                                exitPosition(1) - AGENT_DIAMETER/2 ];
            target_bottom = [ exitPosition(2) - EXIT_WIDTH/2 +
               AGENT_DIAMETER/2, ...
                                exitPosition(1) - AGENT_DIAMETER/2 ];
        end
        if position(end,2) > target_top(2)
            direction = target_top - position;
            target = target_top;
        elseif position(end,2) < target_bottom(2)
            direction = target_bottom - position;
            target = target_bottom;
        end
    end
    direction = direction / norm(direction);
end
```

C.6 neighborhoodOfAgent

```
function [ neighborhood, positionGrid, wallGrids, barrierGrid,
   positionOfOtherAgents ]...
    = neighborhoodOfAgent( agents, index, t, positions)
%% Description: constructs the neighborhood of the agent.
%%% Inputs:
%       agents      = structure with all agents data
%       index       = index of current agent (int)
%       t           = current time step (int)
%       positions   = current positions of all agents in matrix form
%
%%% Output:
%       neighborhood            = neighborhood of the agent (matrix)
%       positionGrid            = neighborhood with index of agents at
   their positions
%       wallGrids               = neighborhood with walls
%       positionOfOtherAgents   = relative positions of other agents
%% Code starts here
global NumberOfAgents AGENT_DIAMETER ROOM_SIZE EXIT_WIDTH
   BARRIER_DIAMETER BARRIER_POSITIONS
%% specify SS coordinate system
minusDirection = - agents(index).direction{t};
% theta in radians
[theta,rho] = cart2pol(minusDirection(1),minusDirection(2));
% counterclockwise rotation of axis by angle theta
Rtheta = rotationMatrix2D(theta,'axis');
Rminustheta = rotationMatrix2D(-theta,'axis');
%% neighborhood-grid parameters
gridSize = [5,5];
centerCellMargin = (gridSize - 1)/2;
% cell size
cellSize = AGENT_DIAMETER / sqrt(2);
boundsLR = (- centerCellMargin(1) : centerCellMargin(1))*cellSize;
```

```
boundsTB = (centerCellMargin(2) :-1: -centerCellMargin(2))*cellSize;
boundsTop    = boundsTB + cellSize/2;
boundsBottom = boundsTB - cellSize/2;
boundsRight  = boundsLR + cellSize/2;
boundsLeft   = boundsLR - cellSize/2;
%% get index and position of other agents
% all other agents
indexOfOtherAgents = [1:index-1, index+1:NumberOfAgents];
% compute the distance between agents{index} and other agents
distance = sqrt(sum( ...
                    (positions(indexOfOtherAgents,:) - repmat(positions(
                       index,:),length(indexOfOtherAgents),1)).^2 ...
                    ,2))';
% regardless of the shape of the neigborhood,
% all its agents are also inside of a eucleidan-distance neighborhood of
    the radius
neigborhoodRadius = max(gridSize)*cellSize*sqrt(2)/2;
% account only for near agents
indexOfOtherAgents = indexOfOtherAgents(distance < neigborhoodRadius);
% compute the position of other agents in the SS coordinate system
if ~isempty(indexOfOtherAgents)
    positionOfOtherAgents = Rtheta * (positions(indexOfOtherAgents,:) -
       repmat(agents(index).position{t},length(indexOfOtherAgents),1))';
else
    positionOfOtherAgents = [];
end
positionGrid = zeros(gridSize);
for j = 1:length(indexOfOtherAgents)
    ind = indexOfOtherAgents(j);
    xpos = positionOfOtherAgents(1,j);
    ypos = positionOfOtherAgents(2,j);
    xind = find(( xpos > boundsLeft ) & ( xpos <= boundsRight ));
    yind = find(( ypos > boundsBottom ) & ( ypos <= boundsTop ));
    if ~isempty(xind) && ~isempty(yind) % else, agent is too far
        if ~positionGrid(yind,xind)
            positionGrid(yind,xind) = ind;
        else
            agentApos = positionOfOtherAgents(:,j)
            agentBpos = positionOfOtherAgents(:,find(indexOfOtherAgents
               ==...
                        positionGrid(yind,xind)))
            dist = norm(agentApos - agentBpos)
            agentApos = positions(ind,:)
            agentBpos = positions(positionGrid(yind,xind),:)
            dist = norm(agentApos - agentBpos)
            errormsg = sprintf('\nAttempted to place agent %d to
               position [%d, %d],...
            where was already agent %d. \nThis is likely caused by wrong
                cell size....
            \n', ind, xind, yind, positionGrid(yind,xind));
            error('neighborhoodOfAgent:positionGrid',['Two agents in one
                cell. ', errormsg]);
        end
    end
end
%% wall & barrier grid
```

```
wallGrids = zeros([gridSize,4]);
barrierGrid = zeros(gridSize);
K = ROOM_SIZE(2)/2;
L = ROOM_SIZE(1);
% bounds
% top: y = K
% bottom: y = -K
% left: x = 0
% right: x = L
Q = zeros(4,2);
for i = 1 : length(boundsLR)
    for j = 1 : length(boundsTB)
        Q = SStoStransformation2( [ ...
                                    [boundsLeft(i); boundsBottom(j)],
                                       ...
                                    [boundsLeft(i); boundsTop(j)], ...
                                    [boundsRight(i); boundsBottom(j)],
                                       ...
                                    [boundsRight(i); boundsTop(j)]
                                    ]);
        Q = Q';
        % intersections
        % top: y = K
        wallGrids(j,i,1) = ( wallGrids(j,i,1) || (...
            ( Q(1,2) > K || ...
              Q(2,2) > K ) || ...
            ( Q(3,2) > K || ...
              Q(4,2) > K ))) ...
              * 1;
        % bottom: y = -K
        wallGrids(j,i,2) = ( wallGrids(j,i,2) || (...
            ( Q(1,2) < -K || ...
              Q(2,2) < -K ) || ...
            ( Q(3,2) < -K || ...
              Q(4,2) < -K ))) ...
              * 2;
        % left: x = 0
        wallGrids(j,i,3) = (( wallGrids(j,i,3) || (...
            ( Q(1,1) < 0 || ...
              Q(2,1) < 0 ) || ...
            ( Q(3,1) < 0 || ...
              Q(4,1) < 0 ))) && (...
            ( ( Q(1,2) > EXIT_WIDTH/2 || Q(1,2) < -EXIT_WIDTH/2 ) || ...
              ( Q(2,2) > EXIT_WIDTH/2 || Q(2,2) < -EXIT_WIDTH/2 ) ) ||
                 ...
            ( ( Q(3,2) > EXIT_WIDTH/2 || Q(3,2) < -EXIT_WIDTH/2 ) || ...
              ( Q(4,2) > EXIT_WIDTH/2 || Q(4,2) < -EXIT_WIDTH/2 ) )) ...
              )* 3;
        % right: x = L
        wallGrids(j,i,4) = ( wallGrids(j,i,4) || (...
            ( Q(1,1) > L || ...
              Q(2,1) > L ) || ...
            ( Q(3,1) > L || ...
              Q(4,1) > L ))) ...
              * 4;
        % barriers
```

```
        for b = 1:length( BARRIER_POSITIONS )
            barrierGrid(j,i) = ( barrierGrid(j,i) || (...
                ( norm(Q(1,:) - BARRIER_POSITIONS{b}) < BARRIER_DIAMETER
                   /2 || ...
                  norm(Q(2,:) - BARRIER_POSITIONS{b}) < BARRIER_DIAMETER
                     /2 ) || ...
                ( norm(Q(3,:) - BARRIER_POSITIONS{b}) < BARRIER_DIAMETER
                   /2 || ...
                  norm(Q(4,:) - BARRIER_POSITIONS{b}) < BARRIER_DIAMETER
                     /2 ))) ...
                  * 9;
        end
    end
end
wallGrid = (wallGrids(:,:,1) | wallGrids(:,:,2)) | ...
           (wallGrids(:,:,4) | wallGrids(:,:,3));
%% neighborhood
centralPositon = 13;
neighborhood = -wallGrid;
for i = 1:gridSize(1)
    for j = 1:gridSize(2)
        if positionGrid(i,j)
            neighborhood(i,j) = agents(positionGrid(i,j)).step{end};
        elseif barrierGrid(i,j)
            neighborhood(i,j) = barrierGrid(i,j);
        end
    end
end
if t > 1
    neighborhood(centralPositon) = agents(index).step{t-1};
else
    neighborhood(centralPositon) = 1;
end
%% nested functions
    function pos = getAgentPosition(a)
        % position in default coordinates
        pos = a.position{end};
        % position with respect to agent i
        pos = pos - agents(index).position{t};
        % position with respect to agent i with rotated coordinate
           system
        pos = Rtheta * pos(:);
    end
    function pos_SS = StoSStransformation( pos_S )
        p =  pos_S(:) - agents(index).position{t}(:);
        pos = Rtheta * p;
        pos_SS = pos(:)';
    end
    function pos_S = SStoStransformation( pos_SS )
        pos = Rminustheta * pos_SS(:);
        p =  pos(:) + agents(index).position{t}(:);
        pos_S = p(:)';
    end
    function pos_S = SStoStransformation2( pos_SS )
        pos = Rminustheta * pos_SS;
```

```
        pos_S = pos + repmat(agents(index).position{t}(:),1,size(pos_SS
            ,2));
    end
end
```

C.7 getAgentDecision

```
function [ step, direction, regressors ] = getAgentDecision(
    neighborhood, optimalDirection )
%% Description
% Get decision of agent based on his neighborhood. Returns also movement
    direction.
%%% Inputs:
%       neighborhood        = neigborhood of agent (matrix)
%       optimalDirection    = desired direction of movement (2x1 matrix)
%%% Outputs:
%       step        = decision (discrete) stand/forward/right/left (int)
%       direction   = decided direction (2x1 matrix)
%% Code starts here
global Debug
global simulationDataMethod NamesOfRegressors RelevantNeighborhoodFields
decisionVersion = 'v9';
if  strcmp(decisionVersion ,'v10')
%% v10
simulationDataMethod = 'regressors';
%% linear indexing
%    1   6  11  16  21
%    2   7  12  17  22
%    3   8  13  18  23
%    4   9  14  19  24
%    5  10  15  20  25
%% relevant fields
%    *   6  11   *   *
%    2   7  12  17   *
%    3   8   *   *   *
%    4   9  14  19   *
%    *  10  15   *   *
%% groups
%  *   *   *   *   *       *   6  11   *   *       *   *   *   *   *
%  2   7   *   *   *       *   7  12  17   *       *   *   *   *   *
%  3   8   *   *   *       *   *   *   *   *       *   *   *   *   *
%  4   9   *   *   *       *   *   *   *   *       *   9  14  19   *
%  *   *   *   *   *       *   *   *   *   *       *  10  15   *   *
%%
frontGroup = sum(logical( neighborhood( [2:4,7:9] ) ));
rightGroup = sum(logical( neighborhood( [6:7,11:12,17] ) ));
leftGroup = sum(logical( neighborhood( [9:10,14:15,19] ) ));
%%
frontGroupProbReduction = [0, 0.2, 0.25, 0.30, 0.35, 0.40, 0.45];
                % maximally 6 particles in the front group
rightGroupProbReduction = [0, 0.1, 0.15, 0.18, 0.19, 0.20];
                % maximally 5 particles in the right group
leftGroupProbReduction  = [0, 0.1, 0.15, 0.18, 0.19, 0.20];
                % maximally 5 particles in the left group
defaultDecisionProbability = [ 0.025, 0.45, 0.05, 0.05 ];
probability = defaultDecisionProbability;
```

```
if frontGroup > 1
    probability = probability + [ 0, ...
                - frontGroupProbReduction( frontGroup +1 ), ...
                leftGroupProbReduction( leftGroup +1 ), ...
                rightGroupProbReduction( rightGroup +1 )];
                          % +1 because of the null count has index 1
end
%% probability of movement directions & stochastic decision
% firstOrderNeighborhood
% neighborhood
probability = probability ./ sum(probability);
step = randIntProb( probability, 1:4 );
if Debug
    neighborhood
    probability * 100
    step
end
regressors = [frontGroup, rightGroup, leftGroup];
%% computation of direction based on decision
switch step
    case 1
        direction = [0,0];
    case 2
        direction = optimalDirection;
    case 3
        direction = rotationMatrix2D(-pi/2,'object')*optimalDirection(:)
           ;
        direction = direction';
    case 4
        direction = rotationMatrix2D(pi/2,'object')*optimalDirection(:);
        direction = direction';
end
%% set names of regressors (for testMarkovHigh.m)
if ~exist('NamesOfRegressors', 'var') || isempty(NamesOfRegressors)
    NamesOfRegressors = { ...
                          ' front_cells ',...
                          ' right_cells ',...
                          ' left_cells' };
end
elseif  strcmp(decisionVersion , 'v9')
% v9
simulationDataMethod = 'neighborhood';
%% linear indexing
%    1   6  11  16  21
%    2   7  12  17  22
%    3   8  13  18  23
%    4   9  14  19  24
%    5  10  15  20  25

%% relevant fields
%    *   6  11   *   *
%    2   7  12  17   *
%    3   8   *   *   *
%    4   9  14  19   *
%    *  10  15   *   *
%% set relevant neighborhood fields (for testMarkovHigh.m)
```

```
if ~exist('RelevantNeighborhoodFields', 'var') || isempty(
    RelevantNeighborhoodFields)
     RelevantNeighborhoodFields = [2:4,6:10,11,12,14,15,17,19];
end
%%
N = neighborhood;
QQQ = sum(logical(neighborhood(7:9) > 0));
PPP = sum(logical(neighborhood(2:4) > 0));
firstOrderNeighborhood = logical([PPP | QQQ, neighborhood(14),
    neighborhood(12)]);
if all(firstOrderNeighborhood == [0 0 0])
     probability = [ 0, 1, 0, 0 ];
elseif all(firstOrderNeighborhood == [0 0 1])
     probability = [ 0, 1, 0, 0 ];
elseif all(firstOrderNeighborhood == [0 1 0])
     probability = [ 0, 1, 0, 0 ];
elseif all(firstOrderNeighborhood == [1 1 1])
     probability = [ 1, 0, 0, 0 ];
elseif all(firstOrderNeighborhood == [1 0 1])
% relevant fields
%     *    *    *    *    *
%     *    *    X    *    *
%     *    X    *    *    *
%     *    9    0   19    *
%     *   10   15    *    *
     if all( [ N(9), N(15) ] )
         probability = [ 0.9, 0, 0, 0.1];
     elseif all( [ N(9), ~N(15) ] )
         probability = [ 0.9, 0, 0, 0.1];
     elseif all( [ ~N(9), N(15), ~N(19) ] )
         probability = [ 0.8, 0, 0, 0.2];
     elseif all( [ ~N(9), N(10), ~N(15), ~N(19) ] )
         probability = [ 0.5, 0, 0, 0.5];
     elseif all( [ ~N(9), ~N(10), ~N(15), ~N(19) ] )
         probability = [ 0.3, 0, 0, 0.7];
     else
         probability = [ 0.3, 0, 0, 0.7];
     end
elseif all(firstOrderNeighborhood == [1 1 0])
% relevant fields
%     *    6   11    *    *
%     *    7    0   17    *
%     *    X    *    *    *
%     *    *    X    *    *
%     *    *    *    *    *
     if all( [ N(7), N(11) ] )
         probability = [ 0.9, 0, 0.1, 0];
     elseif all( [ N(7), ~N(11) ] )
         probability = [ 0.9, 0, 0.1, 0];
     elseif all( [ ~N(7), N(11), ~N(17) ] )
         probability = [ 0.8, 0, 0.2, 0];
     elseif all( [ ~N(7), N(6), ~N(11), ~N(17) ] )
         probability = [ 0.5, 0, 0.5, 0];
     elseif all( [ ~N(7), ~N(6), ~N(11), ~N(17) ] )
         probability = [ 0.3, 0, 0.7, 0];
     else
```

```
        probability = [ 0.3, 0, 0.7, 0];
    end
elseif all(firstOrderNeighborhood == [0 1 1])
% relevant fields
%    *   *   *   *   *
%    2   7   X   *   *
%    3   0   *   *   *
%    4   9   X   *   *
%    *   *   *   *   *
    if all( [ ~N(3), ~N(7), ~N(9) ] )
        probability = [ 0, 1, 0, 0];
    elseif all( [ N(3), ~N(7), ~N(9) ] )
        probability = [ 0, 1, 0, 0];
    elseif all( [ N(7), N(9) ] )
        probability = [ 1, 0, 0, 0];
    elseif all( [ ~N(7), N(9) ] )
        probability = [ 0.5, 0.5, 0, 0];
    elseif all( [ N(7), ~N(9) ] )
        probability = [ 0.5, 0.5, 0, 0];
    end
elseif all(firstOrderNeighborhood == [1 0 0])
% relevant fields
%    *   6  11   *   *
%    *   7   0   *   *
%    *   X   *   *   *
%    *   9   0   *   *
%    *  10  15   *   *
    if  all( [ ~N(6), ~N(7), ~N(9), N(10), ~N(11), ~N(15) ] )
        probability = [ 0, 0, 0.7, 0.3];
    elseif all( [ N(6), ~N(7), ~N(9), ~N(10), ~N(11), ~N(15) ] )
        probability = [ 0, 0, 0.3, 0.7];
    elseif all( [ N(6), ~N(7), ~N(9), N(10), ~N(11), ~N(15) ] )
        probability = [ 0, 0, 0.3, 0.7];
    elseif all( [ ~N(7), N(9) ] )
        probability = [ 0, 0, 1, 0];
    elseif all( [ N(7), ~N(9) ] )
        probability = [ 0, 0, 0, 1];
    elseif all( [ ~N(7), ~N(9), ~N(11), N(15) ] )
        probability = [ 0, 0, 1, 0];
    elseif all( [ ~N(7), ~N(9), N(11), ~N(15) ] )
        probability = [ 0, 0, 0, 1];
    elseif all( [ N(7), N(9) ] )
        probability = [ 0.5, 0, 0.25, 0.25];
    else
        probability = [ 0, 0, 0.5, 0.5];
    end
end
%% probability of movement directions & stochastic decision
% firstOrderNeighborhood
% neighborhood
step = randIntProb( probability, 1:4 );
if Debug
    neighborhood
    firstOrderNeighborhood
    probability
    step
```

```
end
%% computation of direction based on decision
switch step
    case 1
        direction = [0,0];
    case 2
        direction = optimalDirection;
    case 3
        direction = rotationMatrix2D(-pi/2,'object')*optimalDirection(:)
           ;
        direction = direction';
    case 4
        direction = rotationMatrix2D(pi/2,'object')*optimalDirection(:);
        direction = direction';
end
regressors = [];
end
end
```

C.8 moveAgent

```
function [ newPosition ] = moveAgent( agents, index, time, positions )
%% Description:
% The functuion moves specified agent in a specified direction.
%%% Inputs:
%       agents      = 'structure of all agents with all parameters'
%       index       = 'index of the moving agent'
%       time        = 'discrete point in time'
%%% Outputs:
%       newPosition = 'position of the agent in sequent time step'
%% Code starts here
%% Inicialization
global NumberOfAgents AGENT_DIAMETER ROOM_SIZE EXIT_WIDTH
global MovementOverlayAgent StationaryOverlayAgent
global MovementOverlayWall StationaryOverlayWall
global MovementOverlayBarrier StationaryOverlayBarrier
global maxStepSize
global BARRIER_POSITIONS BARRIER_DIAMETER
% by default, the new position equals current position
newPosition     = agents(index).position{time};
direction       = agents(index).directionDecision{time};
step            = agents(index).step{time};
positionGrid    = agents(index).positionGrid{time};
indexOfOtherAgents = [1:index-1, index+1:NumberOfAgents];
% only agents in the neighborhood are relevant
% check the decision for movement
if step ~= 1 % 1 = stay on place
    trySmallStep = true;
else
    trySmallStep = false;
end
tmpPosition = agents(index).position{time};
maxNumberOfSmallSteps = 10;
smallStepSize = maxStepSize/maxNumberOfSmallSteps;
numberOfSmallSteps = 0;
K = ROOM_SIZE(2)/2;
```

```
L = ROOM_SIZE(1);
%% iterative movement
while trySmallStep && (numberOfSmallSteps < maxNumberOfSmallSteps)
    % try small step in specified direction
    tmpPosition = tmpPosition + direction * smallStepSize;
    %% check movement and stationary condition for agent-wall distance
    % top: y = K
    oobStationaryTop = tmpPosition(2) > K - (AGENT_DIAMETER/2 -
       StationaryOverlayWall);
    oobMovementTop   = tmpPosition(2) > K - (AGENT_DIAMETER/2 -
       MovementOverlayWall);
    % bottom: y = -K
    oobStationaryBottom = tmpPosition(2) < -K + (AGENT_DIAMETER/2 -
       StationaryOverlayWall);
    oobMovementBottom   = tmpPosition(2) < -K + (AGENT_DIAMETER/2 -
       MovementOverlayWall);
    % left: x = 0
    oobStationaryLeft = (tmpPosition(1) < 0 + (AGENT_DIAMETER/2 -
       StationaryOverlayWall)) && ...
        ( tmpPosition(2) > EXIT_WIDTH/2 - (AGENT_DIAMETER/2 -
           StationaryOverlayWall) || ...
          tmpPosition(2) < -EXIT_WIDTH/2 + (AGENT_DIAMETER/2 -
             StationaryOverlayWall) );
    oobMovementLeft = (tmpPosition(1) < 0 + (AGENT_DIAMETER/2 -
       MovementOverlayWall)) && ...
        ( tmpPosition(2) > EXIT_WIDTH/2 - (AGENT_DIAMETER/2 -
           MovementOverlayWall) || ...
          tmpPosition(2) < -EXIT_WIDTH/2 + (AGENT_DIAMETER/2 -
             MovementOverlayWall) );

    % right: x = L
    oobStationaryRight = tmpPosition(1) > L - (AGENT_DIAMETER/2 -
       StationaryOverlayWall);
    oobMovementRight = tmpPosition(1) > L - (AGENT_DIAMETER/2 -
       MovementOverlayWall);
    % if the agent is not out of bounds in the sense movement overlay
       condition
    if ~any([oobMovementTop oobMovementBottom oobMovementLeft
       oobMovementRight])
        % first check distance form barriers
        lowBarrierDistanceMovement = 0;
        lowBarrierDistanceStationary = 0;
        for b = 1:length( BARRIER_POSITIONS )
            if norm( tmpPosition - BARRIER_POSITIONS{b} ) <...
                    ( BARRIER_DIAMETER + AGENT_DIAMETER )/2 -
                       MovementOverlayBarrier
                lowBarrierDistanceMovement = 1;
                trySmallStep = false;
            end
            if norm( tmpPosition - BARRIER_POSITIONS{b} ) < ...
                  ( BARRIER_DIAMETER + AGENT_DIAMETER )/2 -
                     StationaryOverlayBarrier
                lowBarrierDistanceStationary = 1;
            end
        end
        %% check for distance from other agents
```

```
        distance = sqrt(sum((positions(indexOfOtherAgents,:) -...
                    repmat(tmpPosition,length(indexOfOtherAgents),1))
                        .^2, 2))';
        % if the distance from other agents is acceptable
        if ~any(distance < AGENT_DIAMETER - MovementOverlayAgent)
            numberOfSmallSteps = numberOfSmallSteps + 1;
            % if the stationary condition for walls, barriers and agents
                is fulfilled
            if ( ~any(distance < AGENT_DIAMETER - StationaryOverlayAgent
               ) && ...
                 ~any([oobStationaryTop oobStationaryBottom...
                  oobStationaryLeft oobStationaryRight]) ) && ...
                 ~lowBarrierDistanceStationary
                % save this position as possible new position
                newPosition = tmpPosition;
            end
        else
           trySmallStep = false;
        end
    else
           trySmallStep = false;
    end
end
end
```

C.9 drawSystem

```
function [ handles, h_room, h_hallway, h_exit ] = drawSystem( agents,
    agent, previousagent )
%% Description.:
% Draws the state of the entire system.
%%% Inputs:
%       agents          = structure with data about all agents
%       agent           = structure with data about signle agent
%       previousagent   = structure with data about agent updated in
    previous step

%%% Outputs:
%       handles         = handles to the graphical objects representing
    agent
%       h_room          = handle to the graphical representation of the
    room
%       h_hallway       = handle to the graphical representation of the
    hallway
%       h_exit          = handle to the graphical representation of the
    exit
%% Code starts here
global ROOM_SIZE HALLWAY_SIZE EXIT_WIDTH NumberOfAgents AGENT_DIAMETER
    DRAW
if ~isempty(agents)
    hold on
    [h_room, h_hallway, h_exit] = drawRoom(ROOM_SIZE, HALLWAY_SIZE,
        EXIT_WIDTH);
    for i = 1:NumberOfAgents
        if agents(i).position{end}(1) < Inf
            handles{i} = drawAgent(agents(i),i, 'end');
```

```
        end
    end
elseif ~isempty(agent)
    if DRAW.agentPath
        tmp = [agent.position{end-1}; agent.position{end}];
        h = line(tmp(:,1), tmp(:,2));
        set(h, 'Marker', '+', 'Color', 'b');
    end
    if agent.position{end}(1) < Inf
        % update agent position
        agent.handles.circle.Position = [ agent.position{end} -...
            AGENT_DIAMETER/2, AGENT_DIAMETER , AGENT_DIAMETER ];
        agent.handles.circle_inner.Position =[ agent.position{end} -...
            AGENT_DIAMETER/4, AGENT_DIAMETER/2 , AGENT_DIAMETER/2 ];
        agent.handles.number.Position = [ agent.position{end} ];

    % distinguish currently updated agent
    agent.handles.circle.FaceColor = [0.5 0 1];
    else
        agent.handles.circle.Visible = 'off';
        agent.handles.circle_inner.Visible = 'off';
        agent.handles.number.Visible = 'off';
    end
    % set previously updated agent to default color
    if ~isempty(previousagent)
        previousagent.handles.circle.FaceColor = 'r';
    end
end
```

C.10 drawRoom

```
function [h_room,h_hallway,h_exit] = drawRoom(room_size, hallway_size,
    exit_width)
%% Description:
% Draws room in the current figure.
%%% Inputs:
%       room_size       = size of the room (1x2 matrix)
%       hallway_size    = size of the hallway (1x2 matrix)
%       exit_width      = width of the exit (double)
%%% Outputs:
%       h_room          = handle to the graphical representation of the
    room
%       h_hallway       = handle to the graphical representation of the
    hallway
%       h_exit          = handle to the graphical representation of the
    exit
%% Code starts here
global EXIT_POSITIONS EXIT_ORIENTATION BARRIER_DIAMETER
    BARRIER_POSITIONS
set(gca, 'color', [0.8 0.8 0.8]);
set(gcf, 'color', [1 1 1]);
%% room
h_room = rectangle('Position',[ 0 , -room_size(2)/2 , room_size(1) ,
    room_size(2) ],...
          'Curvature',[0,0],...
          'LineWidth',2,...
```

```
            'LineStyle','-',...
            'EdgeColor','black',...
            'FaceColor','white');

%% hallway
h_hallway = rectangle('Position',[ -hallway_size(1), - hallway_size(2)/2
    ,...
             hallway_size(1) , hallway_size(2) ],...
            'Curvature',[0,0],...
            'LineWidth',2,...
            'LineStyle','-',...
            'EdgeColor','black',...
            'FaceColor','white');

%% exit
for i = 1:length(EXIT_POSITIONS)
    switch EXIT_ORIENTATION{i}
        case 'y'
            X = [EXIT_POSITIONS{i}(1), EXIT_POSITIONS{i}(1)];
            Y = [EXIT_POSITIONS{i}(2), EXIT_POSITIONS{i}(2)] +...
                [-exit_width/2, exit_width/2];

        case 'x'
            X = [EXIT_POSITIONS{i}(1), EXIT_POSITIONS{i}(1)] +...
                [-exit_width/2, exit_width/2];
            Y = [EXIT_POSITIONS{i}(2), EXIT_POSITIONS{i}(2)];
    end
    % creates a line from [X(1),Y(1)] to [X(2),Y(2)]
    h_exit{i} = line(X, Y,...
            'LineWidth',2,...
            'Color','white');
end
%% barrier
for b = 1:length( BARRIER_POSITIONS )
    h_barrier{b} = rectangle(...
        'Position',[ BARRIER_POSITIONS{b} - BARRIER_DIAMETER/2 ,
           BARRIER_DIAMETER ,...
         BARRIER_DIAMETER ],...
        'Curvature',[1,1],...
        'LineWidth',0.5,...
        'LineStyle','-',...
        'EdgeColor','none',...
        'FaceColor','k');
end
axis equal;
end
```

C.11 drawAgent

```
function [ handles ] = drawAgent( agent, index, time )
%% Description
% Draws single agent in current figure. The agent's position is drawn at
    time = time.
% The agent's image is composed of inner and outer circles, agent's
   number and
% (if enabled) of the agent's previous step path.
```

```
%%% Inputs:
%       agent    = structure with data about the agent (struct)
%       index    = index of the agent (int)
%       time     = time step (int)
%%% Outputs:
%       handles = handles to the graphical objects representing agent
%% Code starts here
global AGENT_DIAMETER DRAW
% handle time = 'end'
if isa(time,'char')
    switch time
        case 'end'
            time = numel(agent.position);
    end
end
%% path of agent
if DRAW.agentPath
    if time > 1
        tmp = vertcat(agent.position{1:time});
        h = plot(tmp(:,1), tmp(:,2));
        set(h, 'Marker', '+', 'Color', 'b');
    end
end
%% outer circle
handles.circle = rectangle(...
    'Position',[ agent.position{time} - AGENT_DIAMETER/2 ,
       AGENT_DIAMETER ,...
     AGENT_DIAMETER ],...
    'Curvature',[1,1],...
    'LineWidth',0.5,...
    'LineStyle','-',...
    'EdgeColor','none',...
    'FaceColor','r');
%% inner circle
handles.circle_inner = rectangle(...
    'Position',[ agent.position{time} - AGENT_DIAMETER/4 ,
       AGENT_DIAMETER/2 ,...
     AGENT_DIAMETER/2 ],...
    'Curvature',[1,1],...
    'LineWidth',0.5,...
    'LineStyle','-',...
    'EdgeColor','none',...
    'FaceColor',[0.9 0.9 0.9]);
%% agent number
handles.number = text(agent.position{time}(1), agent.position{time}(2),
   ...
                num2str(index),'HorizontalAlignment','center');
end
```

C.12 rotationMatrix2D

```
function [ R ] = rotationMatrix2D( theta, type )
%% Description
% Constructs the matrix that rotates a given vector by a
   counterclockwise angle theta, or the matrix that rotates the
   coordinate system through a counterclockwise angle theta
```

```
%%% Inputs
%       theta    = angle of rotation of axis in radians (double)
%       type     = object or axix (string)
%%% Outputs
%       R        = rotation matrix (2x2 double)
if nargin == 1
    type = 'object';
end
switch type
    case 'object'
        R = [ cos(theta), -sin(theta); ...
              sin(theta), cos(theta) ];
    case 'axis'
        R = [ cos(theta), sin(theta); ...
              -sin(theta), cos(theta) ];
end
end
```

C.13 RANDX

```
function [ matrix ] = RANDX( dimension, range )
%% Description:
% Generates matrix of pseudorandom numbers in a specified range
%%% Inputs:
%       dimension    = vector of integers specifing dimension of the
   matrix
%       range        = range of returned values
%%% Outputs:
%       matrix       = matrix of generated values
%% Code starts here
matrix = range(1) + ( range(2) - range(1) ) * rand( dimension );
end
```

C.14 mywaitbar

```
function [ h ] = mywaitbar( t, h )
%MYWAITBAR creates a wait bar that does not slow computation too much
%% Code starts here
persistent tt hh
if nargin < 2
    h = [];
end
if ~isempty(t)
    if isempty(tt) || tt + 0.01 <= t % update waitbar only after 1%
       increase
        tt = t;
        if isempty(hh)
            hh = waitbar(tt);
        else
            waitbar(tt, hh);
        end
    end
end
if ~isempty(hh)
    h = hh;
end
end
```

D Matlab m-files – Estimation

D.1 dataDifferences

```
function [ DATA ] = dataDifferences( DATA )
%% Description
% modifies the data to diferences (changes in time); time is in second
   dimension (column)
%%% Inputs:
%       DATA    = data matrix to be modified
%%% Outputs:
%       DATA    = modified data matrix
%% Code starts here
DATA_copy = DATA;                                          % backup data
exponent = 0;
for i = min(min((DATA))) : max(max((DATA)))                % for each
   possible data value
    exponent = exponent + 1;                               % increment
       exponent
    DATA(DATA_copy == i) = 2^exponent;                     % set the new
       data value
end
DATA = DATA(:,2:end)-DATA(:,1:end-1);                      % time
   increments
% DATA = DATA -min(DATA')'*ones(1,length(DATA))+1;         % shift to
   positive values
end
```

D.2 deltaf

```
function val = deltaf(c,cc)
%% Category: auxiliary
%% Description
%  val  = column vector of the length cc having 1 on the position c and
   zeros otherwise
%     c = the position
%    cc = length
if c<1 | c>cc, error('improper position in deltaf'),end
val =zeros(cc,1);
val(c,1) = 1;
end
```

D.3 dnoise

```
function e=dnoise(pr)
%% Category: simulation
%% Description: simulates discrete-valued noise according to given
   probabilities
% e  = generated, integer-valued noise
% pr = row vector of probabilities pr(i) = Probability( e = i), i=1,...,
   length of Pr
%% CODE STARTS HERE
if any(pr<0), error('Probabilities must be non-negative'), end
% Convert probabilites into distribution function
le = length(pr);  % the number of modelled levels
pr = cumsum(pr);  % distribution function
pr = pr/pr(le);   % normalisation of the distribution function
```

```
% generate noise sample
ru = rand;          % uniformly generate random number from (0,1)
for i = 1:le
    if ru<pr(i), e = i; break; end
end
end
```

D.4 incrementalDataValues

```
%% Description
% shifts categoric data to incremental values 1,...,n in each data row
%%% Inputs:
%       DATA    = data matrix to be modified
%%% Outputs:
%       DATA    = modified data matrix
%% Code starts here
DATA_copy = DATA;                                   % backup data
for j = 1:size(DATA,1)                              % go through rows of
    data matrix
    ind = 0;                                        % reset value
       counter
    for i = min(DATA(j,:)) : max(DATA(j,:))         % for each data
       value in the row
        if find(DATA_copy(j,:) == i)                % if the value
           exitst
            DATA_row = DATA(j,:);                   % copy the data row
            ind = ind + 1;                          % increment new
               value counter
            DATA_row(DATA_copy(j,:) == i) = ind;    % set new values
            DATA(j,:) = DATA_row;                   % update data row
        end
    end
end
end
```

D.5 mfaccon

```
function mfac = mfaccon(delchn,delcar,psichn,psicar,psidelay)
%% Category: estimation
%% Description
% construct Markov factor used in simulation and learning Markov chain
   mixture
% mfac     = structure completely describing Markov factor
% delchn   = channel number of simulated/predicted variable
% delcar   = cardinality of the set of discrete values of del
% psichn   = channel number of the scalar variable psi in condition
% psicar   = cardinality of the set of discrete values of psi
% psidelay = delay of the variable psi
%% Code starts here
if any([delchn,delcar,psichn,psicar]<=0), error('channel numbers and
   cardinalities in mfaccon must be positive'),end
if psidelay<0,error,('delay in mfaccon must be non-negative'),end
mfac = struct('delchn',delchn,'delcar',delcar,'psichn',psichn, 'psicar',
   psicar, 'psidelay',psidelay, 'theta_del_psi',ones(delcar,psicar)/
   delcar, 'V_del_psi',1/sqrt(delcar)*ones(delcar,psicar));
end
```

D.6 mforget

```
function [pro,lam] = mforget(pro,proold,countupper,eps,laml)
%% Category: estimation
%% Description: evaluates and applies forgetting
% pro          = forgotten vector of occurences
% lam          = forgetting factor
% pro          = vector of occurences to be forgotten
% pro          = vector of occurences before updating
% countupper   = upper bound on the number of iterations
% eps          = precision
%% Code starts here
if nargin < 5, laml= 0.5; end
if nargin < 4, eps = 1e-6;end
if nargin < 3, countupper = 100;end
if nargin < 2, error('mforget needs at least 2 input parameters'),end
lenpsi    = length(pro); % length of the probability
lamaux    = 1;  % initial guess of the forgetting factor
lam       = lamaux; % forgetting factor
diff      = 1;  % difference from zero
count     = 0;  % counter of iterations
elcon     =  sum(gammaln(pro)-gammaln(proold)); % fixed part of e(lambda)
            -gammaln(sum(pro))+gammaln(sum(proold));
while abs(diff)>eps & count <=countupper
    count = count + 1;   % increase the iteration counter
    el    =  elcon;     % value of e(lambda)
    prolam = lam*pro + (1-lam)*proold;      % prolambda
    delpro =  proold-pro;    % difference of pro's
          el = el + sum(delpro.*(psi(prolam)-psi(sum(prolam))));
                    % value of e(lambda)
       derel = -sum(delpro.^2.*psi(1,prolam))+ sum(delpro)^2*psi(1,sum(
           prolam));
                    % derivative of e(lambda)
    if abs(derel)<eps
      diff = 0;
    else
     rat   = el/derel;        % correction
    lamaux = max(laml,min(1,lam - rat));           %%%%%
    diff   = max([abs(rat),abs(lamaux-lam),abs(el)]);
    lam   =lamaux;
    end
end
% apply the forgetting operation for the final lam
pro   = lam*pro + (1-lam)*proold;
end
```

D.7 mgencon

```
function mgen = mgencon(Mix,mwei)
%% Category: simulation
%% Description
% construct generator of external discrete-valued channels
% mgen = structure  describing all generators of external discrete-
   valued channels
% Mix  = structure containing description of the simulate and estimated
   Markov chain mixture
% mwei = structure describing component weights
```

```
%% Code starts here
lenpsi     = length(Mix);  % length of the regression vector
chnum      = 1;            % the number of channels
delm       = 0;            % the maximum delay
for i = 1: lenpsi
  chnum    = max(chnum,Mix(i).psichn);      % the number of channels
  delm     = max(delm,Mix(i).psidelay);     % the maximum delay
end
%%% Prepare generators of external channels
genw    = cell(1,chnum);
for chn =2:chnum
    for i =1:lenpsi
    if chn == Mix(i).psichn
       gen   = mweicon(size(Mix(i).theta_del_psi,2));      % component
          weights
       gen   = rand(size(gen.alpha));    % random filling of component
          weights
       genw{chn} = struct('exterchn',chn,'alphachn',gen/sum(gen));
    end
    end
end
mgen = struct('chnum',chnum,'delm',delm,'genw',genw);
        % creation of the structure describing the generator
end
```

D.8 mgenext

```
function val = mgenext(mgen)
%% Category: simulation
%% Description
%  Generate external discrete-valued channels and store them into DATA
% val  = current realisations of external discrete-valued channels
% mgen = structure containing description of the external discrete-
   valued data
%  WARNING: the external data are put to 2nd and higher channels
%% Code starts here
global DATA TIME
val = zeros(1,mgen(1).chnum);
for chn = 2:mgen(1).chnum
       if ~isempty(mgen(chn).genw)
           val(chn)=dnoise(mgen(chn).genw.alphachn);
                   % generate the output of the external generator
          DATA(mgen(chn).genw.exterchn,TIME)=val(chn);
                   % store the generated data into the global array
                      DATA
       end
   end
end
```

D.9 mgetpsi

```
function psi = mgetpsi(Mix)
%% Category: simulation
%% Description
% construct the current regression vector psi
% psi      = regression vector at given time
```

```
% Mix      = structure containing description of the simulate and
   estimated Markov chain mixture
%% Code starts here
global DATA TIME
% check of input parameters
lenpsi = length(Mix);
psi    = zeros(1,lenpsi); % prepare regression vector
for i = 1:lenpsi
    psi(1,i) = DATA(Mix(i).psichn,TIME-Mix(i).psidelay);
            % fill the regression vector from the global array DATA
end
end
```

D.10 moptn

```
function [v,count] = moptn(v0,rhs,countupper,eps)
%% Category: estimation - auxiliary
%% Description
% searches occurrence vector meeting psi(v_{i})-psi(sum(v_{i})) = rhs_{i
   }
%  v     = the resulting vector of occurrences
%  count = the number of iterations
%     v0 = an initial guess of the vector of occurrences
%    rhs = the right-hand side of the solved equation
% countupper = upper bound on the number of iterations
%    eps = upper bound on precision
% defaults
if nargin < 4, eps = 1e-5; end        % built-in precisions
if nargin < 3, countupper = 100; end      % upper bound on the number
   iterations
if nargin <2, error('moptn needs at least two input arguments'), end
% initial values
 v   = v0;            % initial guess of v
dif = ones(size(v));
   % difference of right-hadn and left-hand side of the solved equation
count = 0;% counter of the number of iterations
flag  = 1;% non-convergence flag  1/0 non-converged/converged
% iterative search
while(flag)
    count =count+1;                  % count the number of iterations
    sv   = sum(v);                   % sum of th oprimised argument
    dif = rhs-psi(v)+psi(sv);        % difference of the righ-hand and
       lef- hand sides
    if max(abs(dif))<eps | count>countupper     % standard stopping rule
       flag = 0;                                % stop
    else
       q    = 1./psi(1,v);                      % 1/trigama(v)
       D    = diag(q);                          % diagonal of 1/trigamma
          (v)
       P    = (D + psi(1,sv)*q*q'/(-psi(1,sv)*sum(q)+1))*dif;  %
          correction vector
       v    = max(v+P,eps);                     % restriction to
          positive v's
       sv   = sum(v);                           % sum of the updated v
       if    abs(psi(1,sv))< eps
```

```
                % stopping due to the expected numerical troubles (
                    probably superluous
            flag = 0;                                   % stop
        end
    end
end
end
```

D.11 mpred

```
function mpreddel = mpred(Mix,mwei,psit)
%% Category: estimation
%% Description
%evaluates predictive pd for given statistics and regression vector
% mpreddel = predictive pd of delta for given regression vector
% Mix      = structure describing the components of the estimated
   mixture
% mvei     = structure describing component weights
% psit     = current value of the regression vector psi
%% Code starts here
lenpsi    = length(Mix);         % length of the regression vector
cardel    = size(Mix(1).V_del_psi,1);         % cardinality of delta
hatalpha  = mwei.kappa/sum(mwei.kappa);       % \hat{\alpha}_{\kappa}
mpreddel  = zeros(cardel,1);                  % predicitve pd
for k = 1: lenpsi
    mpreddel = mpreddel +hatalpha(k)*Mix(k).V_del_psi(:,psit(k))...
                          /sum(Mix(k).V_del_psi(:,psit(k)));
end
end
```

D.12 mpredlam

```
function mpreddel = mpredlam(Mix,mwei,Mixold,mweiold,delt,psit,step)
%% Category: estimation
%% Description
% evaluates predictive pd for the convex combinations of given
   statistics,
% observation and regression vector
% mpreddel = predictive pd as a function of forgetting factor
% Mix      = structure describing the components of the estimated
   mixture
% mvei     = structure describing component weights
% Mixold   = structure describing the old components of the estimated
   mixture
% mveiold  = structure describing old component weights
% delt     = current value of the observation
% psit     = current value of the regression vector psi
%% Code starts here
if nargin < 7, step = 0.01; end
lenpsi    = length(Mix);              % length of the regression vector
cardel    = size(Mix(1).V_del_psi,1); % cardinality of delta
mpreddel = [];
for lam=step:step:1
    hatalpha = lam*mwei.kappa+(1-lam)*mweiold.kappa;
    hatalpha = hatalpha/sum(hatalpha);
    hattheta = zeros(lenpsi,1);
    for k = 1: lenpsi
```

```
        hatheta(k,1)   = (lam *Mix(k).V_del_psi(delt,psit(k))+...
                          (1-lam)*Mixold(k).V_del_psi(delt,psit(k)))/ ...
                          (sum(lam *Mix(k).V_del_psi(:,psit(k))+...
                          (1-lam)*Mixold(k).V_del_psi(:,psit(k))));
    end
   mpreddel =[mpreddel;hatalpha*hatheta];
end
```

D.13 mrhs

```
function [rhskap,rhsV] = mrhs(Mix,mwei,delt,psit)
%% Category: estimation
%% Description
%  evaluates right-hand sides of equations for construction of updated
   and
%  projected posterior pd
% rhskap   = right-hand side for updating kappa's
% rhsV     = righ-hand side for updating V
% Mix      = structure describing the components of the estimated
   mixture
% mvei     = structure describing component weights
% delt     = current measured predicted variable Delta
% psit     = current value of the regression vector psi
%% Code starts here
lenpsi   = length(Mix);          % length of the regression vector
skappa   = sum(mwei.kappa);      % 1'kappa_{t-1;k}
hatalpha = mwei.kappa/skappa;    % \hat{\alpha}_{t-1}
hattheta = zeros(1,lenpsi);      % \hat{\Theta}_{\Delta_{t}|\psi_{t;k}}
sV       = zeros(1,lenpsi);      % 1'V_{t-1|\Delta|\psi_{t;k}}
bartheta = 0;                    % \bar{\Theta}_{t-1;\Delta_{t}|\psi_{t}}
for k = 1:lenpsi
         sV(k)  = sum(Mix(k).V_del_psi(:,psit(k)));
              % 1'V_{t-1|\Delta|\psi_{t;k}}
    hattheta(k) = Mix(k).V_del_psi(delt,psit(k))/sV(k);
              % \hat{\Theta}_{\Delta_{t}|\psi_{t;k}}
    bartheta    = bartheta +hatalpha(k)* hattheta(k);
end
rhskap   = psi(mwei.kappa')-psi(skappa);   % right-hand side for kappa
rhsV     = zeros(Mix(1).delcar,lenpsi);    % right-hand side for V
for k = 1:lenpsi
 rhskap(k)   =  rhskap(k) +(hattheta(k)-bartheta)/bartheta/skappa;
   rhsV(:,k) = psi(Mix(k).V_del_psi(:,psit(k)))- psi(sV(k)) + hatalpha(k
      )*(deltaf(delt,Mix(1).delcar)- hattheta(k))/bartheta/sV(k);
end
```

D.14 mweicon

```
function mwei = mweicon(lenpsi)
%% Category: estimation
%% Description
% construct weights of factors for simulation and learning Markov chain
   mixture
% mwei     = structure describing weights in Markov chains mixture
% lenpsi   = length of the regression vector
%% Code starts here
% check of input parameters
if lenpsi<=0,error,('lenpsi in mweicon must be positive'),end
```

```
mwei = struct('lenpsi',lenpsi,'alpha',ones(1,lenpsi)/...
              lenpsi,'kappa',ones(1,lenpsi)/sqrt(lenpsi));
end
```

D.15 psi

```
function r=psi(k,x)
%. Y = PSI(X) evaluates the psi function for each element of X.
%. X must be real and nonnegative.  The psi function, also know as the
%. digamma function, is the logarithmic derivative of the gamma function
   :
%.
%. psi(x) = digamma(x) = d(log(gamma(x)))/dx = (d(gamma(x))/dx)/gamma(x)
   .
%.
%. Y = PSI(K,X) evaluates the K-derivative of psi at the elements of X.
%. PSI(0,X) is the digamma function, PSI(1,X) is the trigamma function,
%. PSI(2,X) and others are not implemented in mixtools
if(nargin<2)
    x=k;
    k=0;
end;
r=x;
for(i=1:size(r,1))
    for(j=1:size(r,2))
        switch(k)
            case 0
                r(i,j)=digamma(r(i,j));
            case 1
                r(i,j)=trigamma(r(i,j));

        end;
    end;
end;
% function psi = digamma(z)
% %. evaluates digamma function, i.e. 1st derivative of gammaln
% %. z   = positive argument
% %. psi = value of the function
% %.
% %. @author M. Karny
% %. @date 2000
% %. See also : kldistc, kldistcom
% %. Source   : Abramowitz-Stegun
% % patched
% if z<0.0001, z=0.0001; end
% z1=z;
% % shift of small arguments to larger values to get
% % precision of the order Matlab eps
% psi=0;
% while(z1<100)
%   psi=psi-1/z1;
%   z1=z1+1;
% end
% psi = psi+log(z1)-0.5/z1 - 1/(12*z1*z1) + 1/(120*z1^4) - 1/(252*z1^6);
function [ value, ifault ] = digamma ( x )
```

```
%
    ********************************************************************************80

%% DIGAMMA calculates DIGAMMA ( X ) = d ( LOG ( GAMMA ( X ) ) ) / dX
%  Licensing:
%    This code is distributed under the GNU LGPL license.
%  Modified:
%    03 June 2013
%  Author:
%    Original FORTRAN77 version by Jose Bernardo.
%    MATLAB version by John Burkardt.
%  Reference:
%    Jose Bernardo,
%    Algorithm AS 103:
%    Psi ( Digamma ) Function,
%    Applied Statistics,
%    Volume 25, Number 3, 1976, pages 315-317.
%  Parameters:
%    Input, real X, the argument of the digamma function.
%    0 < X.
%    Output, real DIGAMMA, the value of the digamma function at X.
%    Output, integer IFAULT, error flag.
%    0, no error.
%    1, X <= 0.
%  Check the input.
  if ( x <= 0.0 )
    value = 0.0;
    ifault = 1;
    return
  end
%  Initialize.
  ifault = 0;
  value = 0.0;
%  Use approximation for small argument.
  if ( x <= 0.00001 )
    euler_mascheroni = 0.57721566490153286060;
    value = - euler_mascheroni - 1.0 / x;
    return
  end
%  Reduce to DIGAMA(X + N).
  while ( x < 8.5 )
    value = value - 1.0 / x;
    x = x + 1.0;
  end
%  Use Stirling's (actually de Moivre's) expansion.
  r = 1.0 / x;
  value = value + log ( x ) - 0.5 * r;
  r = r * r;
  value = value ...
    - r * ( 1.0 / 12.0 ...
    - r * ( 1.0 / 120.0 ...
    - r *   1.0 / 252.0 ) );

  return
function  result=trigamma(x)
%. evaluates trigamma function, i.e. 2nd derivative of gammaln
```

```
%. x    = positive argument
%. result = value of the function
%.
%. @author P. Nenutil
%. @date 2004
%. Reference:
%.
%. B Schneider,
%. Trigamma Function,
%. Algorithm AS 121,
%. Applied Statistics,
%. Volume 27, Number 1, page 97-99, 1978.
%.
%. From www.psc.edu/~burkardt/src/dirichlet/dirichlet.f
%. (with modification for negative arguments and extra precision)
small = 1e-4;
large = 8;
c = 1.6449340668482264365; % pi^2/6 = Zeta(2)
c1 = -2.4041138063191885707994763;  % -2 Zeta(3)
b2 =  1./6;
b4 = -1./30;
b6 =  1./42;
b8 = -1./30;
b10 = 5./66;
% Use Taylor series if argument <= small
if(x <= small)
    result= 1/(x*x) + c + c1*x;
    return;
end;
result = 0;
% Reduce to trigamma(x+n) where ( X + N ) >= B
while(x < large)
    result =result+ 1/(x*x);
    x=x+1;
end;
% Apply asymptotic formula when X >= B
% This expansion can be computed in Maple via asympt(Psi(1,x),x)
if(x >= large)
    r = 1/(x*x);
    result =result+ 0.5*r + (1 + r*(b2 + r*(b4 + r*(b6 + r*(b8 + r*b10))
       )))/x;
end;
```

References

[1] Marek Bukáček, Pavel Hrabák, and Milan Krbálek. Experimental study of phase transition in pedestrian flow. Transportation Research Procedia. Elsevier Science B.V., 2014.

[2] Marek Bukáček, Pavel Hrabák, and Milan Krbálek. Experimental analysis of two-dimensional pedestrian flow in front of the bottleneck. In M. Chraibi, M. Boltes, A. Schadschneider, and A. Seyfried, editors, *Traffic and Granular Flow 13*, pages 93–101. Springer International Publishing, 2015.

[3] D. Helbing, I. Farkas, and T. Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, 2000.

[4] Dirk Helbing and Péter Molnár. Social force model for pedestrian dynamics. *Phys. Rev. E*, 51:4282–4286, 1995.

[5] M. Kárný. Recursive estimation of high-order markov chains: Approximation by finite mixtures. *Information Sciences*, 2015. conditionally accepted.

[6] Ansgar Kirchner and Andreas Schadschneider. Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. *Physica A: Statistical Mechanics and its Applications*, 312(1-2):260–276, 2002.

[7] Tobias Kretz. *Pedestrian Traffic, Simulation and Experiments.* PhD thesis, Universität Duisburg-Essen, Germany, 2007.

[8] A. Schadschneider, D. Chowdhury, and K. Nishinari. *Stochastic Transport in Complex Systems: From Molecules to Vehicles.* Elsevier Science B. V., Amsterdam, 2010.

[9] Michael J. Seitz and Gerta Köster. Natural discretization of pedestrian movement in continuous space. *Physical Review E*, 86:046108, 2012.