Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

# Evacuation Bottleneck

Daniel Zünd & Simon Schmid

Zürich
May 2010

## Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Gruppenarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen-Hilsmittel verwenden habe, und alle Stellen, die wörtlich oder sinngemäss aus veröffentlichen Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Gruppenarbeit nicht, auch nicht auszugsweise, bereits für andere Prüfung ausgefertigt wurde.

Daniel Zünd Simon Schmid

## Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Daniel Zünd Simon Schmid

# Contents

# 1 Individual contributions

For the implementation, Daniel Zünd did most of the continuous model and Simon Schmid implemented most of the best response dynamics. The report was a teamwork of both, both wrote, reviewed each other and corrected it.

## 2 Introduction and Motivations

People tend to form a crowd in states of emergency. The typical flight behaviour is moving away from the source of danger. In open space people would diffuse in all directions but if there are boundaries like walls or a street the only way out is an exit. Usually, exits are small in comparison to the crowd so the flow of people through the exit will be larger than the exit's capacity. The result manifests itself as a bottleneck. The typical appearance of a bottleneck is a semi-circular crowd around the exit. The main objective of every evacuation plan is a sufficient amount of exits which are well distributed so that the crowd splits up in smaller crowds. The interesting point here is that the crowd will not spread evenly because of the individual and collective behaviour of human beings. People tend to head towards known and visible exits which aren't crowded. This preference for a specific exit may change depending on the circumstances. Our simulation is focused on how people chose an exit and how this decision affects the collective behaviour.

# 3 Description of the Model

## 3.1 Model Overview

This section is a brief description of the model we implemented. We have chosen a continous model for our simulation. The benefit of a continuous simulation is that infinitesimal movements are possible and we think, it shows the movement of people in a more natural way.

The room is a two dimensional space, which includes three different types of agents. The first are the people, which need to be evacuated, and the second are the wall elements. The third kind are the door agents, which define a door. The big difference between the three kinds, is that the door and wall agents can not move. The agents on the other side, need to move, so that they can get out of the room. They move according to potentials, in whose radius they are. Since we are working with potential fields, the agents want to go into the direction of the negative gradient of the sum of all fields. This is mathematically described as:

$$m\frac{\delta^2 x_p}{\delta t^2} = -\sum_{q=1,q\neq p}^{N} \nabla_{x_p} V_{agent}(|x_p - x_q|) - \nabla_{x_p} V_{door}(|x_p - x_q|) - \sum_{q=1}^{W} \nabla_{x_p} V_{wall}(|x_p - x_q|)$$

where

- $V_{agent}$ ... potential-field of other people.
- $V_{wall}$ ... potential-field of wall elements.
- $V_{door}$ ... potential-field of the door, an agent is heading for.

Since the wall and door elements do not move, they build a static field together. The dynamic part of the field comes from the moving people. Each person in the room induces a field, that repels the other agents. So this one has a strong influence on how the people move in the room. In other words, the doors and walls introduce a static field on the whole room, and the people a dynamic field. This allows us to simulate realistic escape dynamics.

The model is chosen according to a homework from the lecture *Simulations using Particles* by Prof. Petros Koumoutsakos.

It is known, that people try to follow each others, as long as there is a constant flow. Once the flow stagnates, it is a matter of patience before people start to panic. In this situation people push each other towards the exit, trying to get out of the room. Instead of moving on faster this behaviour will cause clogging. If this happens people on the margin of the crowd will perhaps reconsider their decision an move away from the crowded exit to an uncrowded one, even if the door was familiar to them. In conclusion this means people will follow only moving people.

In our model, the people choose their door according to some game theoretical approach (6). The agents will calculate the opportunity costs of each exit by weighting the queue in front of the door, the distance to the door and individual preference factors like familiarity and visibility of the door. The individual preference factors and velocities are distributed randomly on initialization.

### 3.2 Wall Potentials

The simulation takes the natural behaviour of avoiding to walk close to walls into account by using repulsive wall potentials inversely proportional to the distance from the walls. Actually the walls are formed by a row of fixed agents.

$$V_{wall}(r) = k_W \frac{1}{r}$$

The range of the wall effect is restricted up to the distance $D_{max}$ from the walls. This prevents taking a wall into account which is on the other side of the room. $k_W$ is a constant, which describes, how strong the repulsive force of the wall is.

### 3.3 Door Potentials

The door potentials behave almost like the wall potential, the big difference here is, that they are attracting. This means that they are proportional to the square of the distance an agent is away from it.

$$V_{door}(r) = k_D (r + s)^2$$

$k_D$ is another constant describing the strength of the attracting force caused by the door. The shifting $s$ factor is needed because the potential mentioned above would have a zero gradient if the radius is zero. The door is formed by a row of door agents which are uniformly distributed on the door's width.

### 3.4 People Potentials

The potentials of the people is pretty much like the potential of the walls. It also repels people, which are close.

$$V_{agent}(r) = k_A \frac{1}{r}$$

What we used in our simulation is that the agents have an other constant $k_A$ in front of the $\frac{1}{r}$.

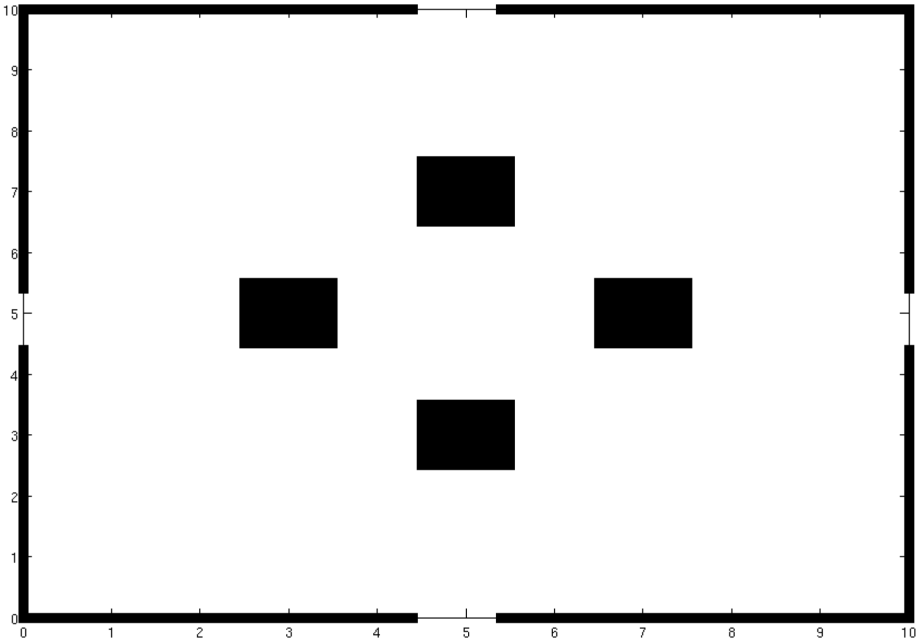

Figure 1: An example of an empty room

### 3.5 Potential field

All the various potentials result in a single force which acts on the agent. The agent reacts according to this field and moves, as mentiond above, along the negative gradient of the sum of all potentials. The following pictures show how the static part of a room may look like. The figure 1 illustrates an empty room. The static field of this figure looks like the plots shown in figure 2 and 3. For these plots, the field was calculated, as if an agent was heading for the door which is on the west side of the room [1].

If we also want to take the dynamic potentials into account, the room looks like in figure 4 and 5. On those plots, the room has been filled with ten agents at random positions.

---

[1] The plots are limited to a maximum value of 2500. Otherwise the values could go up to infinity, if we hit a wall element exactly. In that case, we would not see anything of the rest, just a plain area.

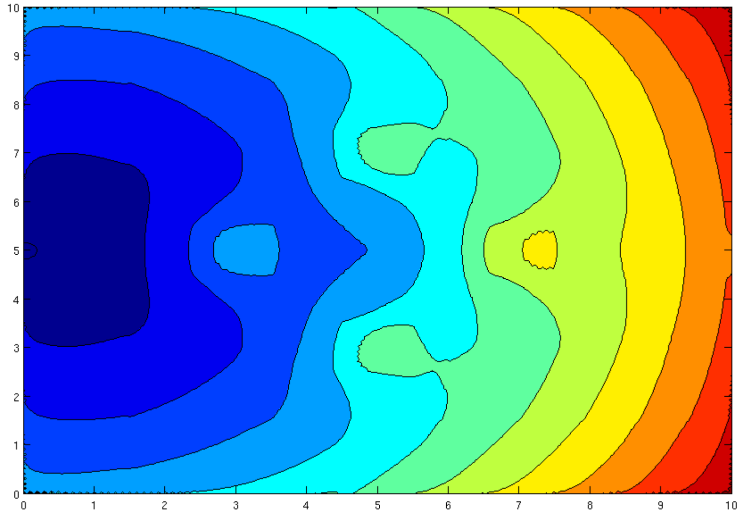

Figure 2: Contour plot of room in figure 1

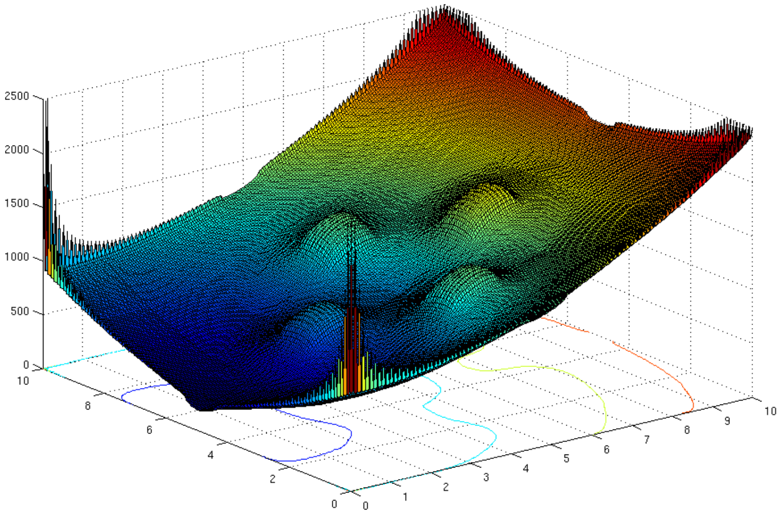

Figure 3: 3D of static potential field of the room in figure 1

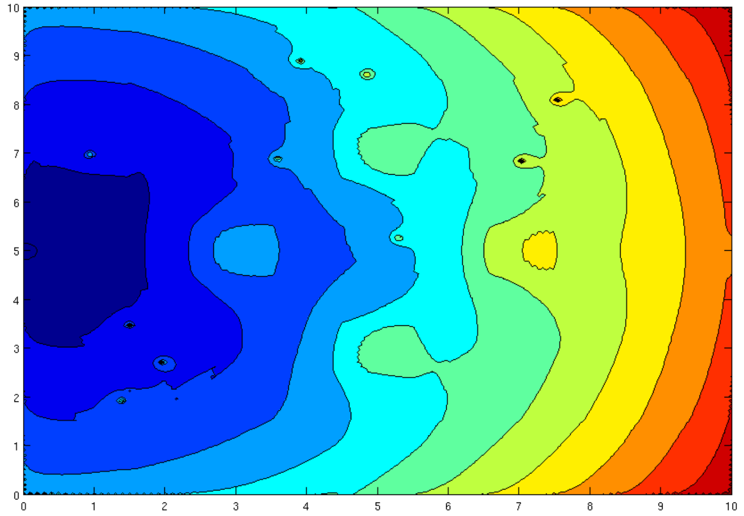

Figure 4: Contour plot of room in figure 1 with 10 agents randomly positioned

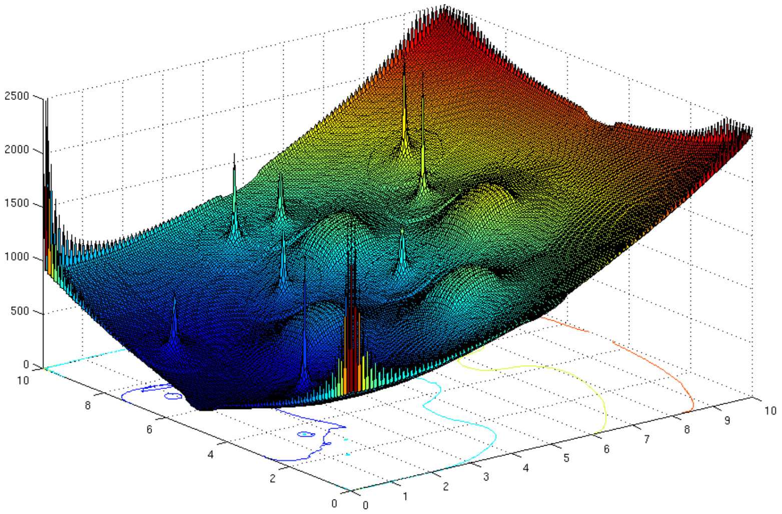

Figure 5: 3D of static potential field of the room in figure 1 with 10 agents randomly positioned

# 4 Exit Selection

In emergency evcuation, the selection of the exit route is one of the most important decisions. We take this into account in our simulation by the implementation of the paper "Exit Selection with Best Response Dynamics" (6). The paper describes an algorithm about how people choose an appropriate exit based on the game theoretic concept of best response dynamics. In the model the agents are the player and the strategies are the possible target exits.

We assume that agents will select the fastest evacuation route. Despite of the time related factor we include two other factors: familiarity and visibility of the exits. The estimated evacuation time of an agent is the sum of the estimated moving time and the estimated queueing time. The estimated moving time is estimated simply by dividing the distance to the exit by the velocity of the agent. The estimated queuing time depends on the exit's capacity and on the number of the other agents that are heading towards the exit and are closer to it than the agent itself. The estimated queuing time binds the decision of a single agent to the decision of other agents. In conclusion, this means the fastest exit route for a specific agent may change during the evacuation.

The familiarity and visibility factor constrain the set of possible exits. These factors can be seen as binary flags and the number of possible combinations form the preference groups. Every door will be divided into a preference group. Agents will select an exit from the nonempty group that has the best preference. The doors in other preference groups are not of any interest.

## 4.1 Mathematical Formulation of the Model

The agents are refered with indices $i$ and $j$, where $i, j \in \mathcal{N} = \{1, 2, 3, ..., N\}$. Exits can be seen as strategies, exits are denoted by $e_k, k \in \mathcal{K} = \{1, 2, ..., K\}$. Strategies are denoted by $s_i \in \{e_1, ..., e_K\} = S_i, i \in \mathcal{N}$ where $S_i$ is a strategy set.

The agent's strategies are concluded by

$$s := (s_1, ..., s_N) \in S_1 \times \cdots \times S_N = S$$

The strategies of all other agents but agent $i$ is defined by

$$s_{-i} := (s_i, ..., s_{i-1}, s_{i+1}, ..., s_N) \in S_{-i}$$

The estimated moving time depends on the agent $i$'s position $\mathbf{r}_i$ and the exit $e_k$'s position $\mathbf{b}_k$. The positions of the agents are in the set $\mathbf{r} := (\mathbf{r}_1, ..., \mathbf{r}_N)$. So the distance between agent $i$ and the exit $e_k$ is

$$d(e_k; \mathbf{r}_i) = ||\mathbf{r}_i - \mathbf{b}_k||$$

The estimated moving time is the division of the distance $d(e_k;\mathbf{r}_i)$ by agent $i$'s velocity $v_i^0$

$$\tau_i(e_k;\mathbf{r}_i) = \frac{1}{v_i^0} d(e_k;\mathbf{r}_i)$$

The estimated queueing time is defined by the sum of all agents but agent $i$ heading towards exit $e_k$ and are closer to exit $e_k$ divided by the exit $e_k$'s capacity $\beta_k$.

The subset of all agents $j \neq i$ who are closer to $e_k$ than agent $i$ is given by

$$\Lambda_i(e_k, s_{-i};\mathbf{r}) = \{j \neq i | s_j = e_k, d(e_k;\mathbf{r}_j) \leq d(e_k;\mathbf{r}_i)\}$$

The number of elements in the subset $\Lambda_i(e_k, s_{-i};\mathbf{r})$ is denoted by

$$\lambda_i(e_k, s_{-1};\mathbf{r}) = |\Lambda_i(e_k, s_{-i};\mathbf{r})|$$

The exit $e_k$'s capacity $\beta_k$ is a scalar value telling us how many agents can pass the exit $e_k$ at once.

So the estimated queueing time is

$$\frac{1}{\beta_k}\lambda_i(e_k, s_{-1};\mathbf{r}) = |\Lambda_i(e_k, s_{-i};\mathbf{r})|$$

The sum of the estimated moving time and estimated queueing time gives us the estimated evacuation time for agent $i$ through the exit $e_k$

$$T_i(s_i, s_{-i};\mathbf{r}) = \frac{1}{\beta_k}\lambda_i(e_k, s_{-1};\mathbf{r}) + \tau_i(e_k;\mathbf{r}_i)$$

As a result of the game theoretic principle, the strategy of agent $i$ is the best response to the other agents' strategies. This means every agent will choose the exit which has the lowest evacuation time.

$$s_i = BR_i(s_{-i};\mathbf{r}) = \arg\min_{s_i' \in S_i} T_i(s_i', s_{-i};\mathbf{r})$$

As we have mentioned before the effects of familiarity and visibility of exits can constrain the group of possible exits for agent $i$, these conditions are taken into account by defining two binary flags

$$fam_i(e_k),\ vis(e_k;\mathbf{r_i}), \quad \forall\ i \in \mathcal{N}, k \in K$$

The binary flags give certain information about agent $i$:

$$fam_i(e_k) \ = \ \begin{cases} 1 & \text{if exit } e_k \text{ is familiar to agent } i \\ 0 & \text{if exit } e_k \text{ is not familiar to agent } i \end{cases}$$

$$vis(e_k; \mathbf{r}_i) = \begin{cases} 1 & \text{if exit } e_k \text{ is visible to agent } i \\ 0 & \text{if exit } e_k \text{ is not visible to agent } i \end{cases}$$

These factors are the criterias for dividing the exits in to groups with preference numbers. There are four possible combinations which means there are four groups of exits with preference numbers from one to four. The smaller the preference number is, the more preferable the exit. The familiarity of an exit has a bigger influence about how preferable an exit is. Studies have shown that evacuees prefere familiar routes even if there is a shorter route (6). The visibility flag is important for the calculation of the estimated queueing time beacause an agent is only able to estimate the queue in front of a door if he can see the door.

According to the previous definition the doors will be grouped as shown in the table below.

| Preference number | Exit group | $vis(e_k; \mathbf{r}_i)$ | $fam_i(e_k)$ |
|---|---|---|---|
| 1 | $E_i(1)$ | 1 | 1 |
| 2 | $E_i(2)$ | 0 | 1 |
| 3 | $E_i(3)$ | 1 | 0 |
| 4 | No Preference | 0 | 0 |

**Table 1** The preference groups in which the exits will be divdided into. The smaller the preference number, the more preferable the exit. The fourth preference group describes people in panic which are not familiar with the exits and can not see any either. (6)

Mathematically the selection of the door is defined as

$$s_i = BR_i(s_{-i}; \mathbf{r}) = \arg \min_{s_i' \in S_i} T_i(s_i', s_{-i}; \mathbf{r})$$

$$s_i' \in E_i(\overline{z})$$

The specific agent $i$ chooses an exit from the non-empty Group $E_i(\overline{z})$ which has the best preference number $\overline{z}$ for him.

In addition to the paper we added an extra patience factor. The patience factor is a simple comparison between the evacuation time of the preferable new exit and the previously chosen exit. This is needed because it may happen that an exit in a better preference group gets in sight. Despite the fact that the exit is in a better preference group the evacuation time could take much longer. So the agent will not redecide if the evacuation time of the new preferable exit is greater than the evacuation time of the agent's previous decision. This could be omitted if the number of exits is significant higher than the number of possible preference groups.

# 5 Implementation

The simulation is split into several function files. The main file, where the whole simulation is running, is the *simulation.m*. This file needs some information of the room, the walls and doors, the agents and so on to run. What it exactly needs, can be looked up in the comment of the file. To run some different kinds of simulation, we provide with the code some *initX.m* ($X \in 1 \ldots 5$) which construct different examples of rooms and place the people at random positions. For an example of a running matlab script please have a look at the first element in appendix A.

## 5.1 Time Integration

For the time integration we do an simple explicit euler. This means that we integrate according to the following scheme:

$$v_{i+1} = v_i + \delta t \cdot a_i$$

$$x_{i+1} = x_i + \delta t \cdot v_{i+1}$$

The $a$ is calculated as it was shown in the introduction of this report:

$$a = \frac{\delta^2 x_p}{\delta t^2} = \frac{1}{m}\left( - \sum_{q=1, q \neq p}^{N} \nabla_{x_p} V_{agent}(|x_p - x_q|) - \nabla_{x_p} V_{door}(|x_p - x_q|) - \sum_{q=1}^{W} \nabla_{x_p} V_{wall}(|x_p - x_q|) \right)$$

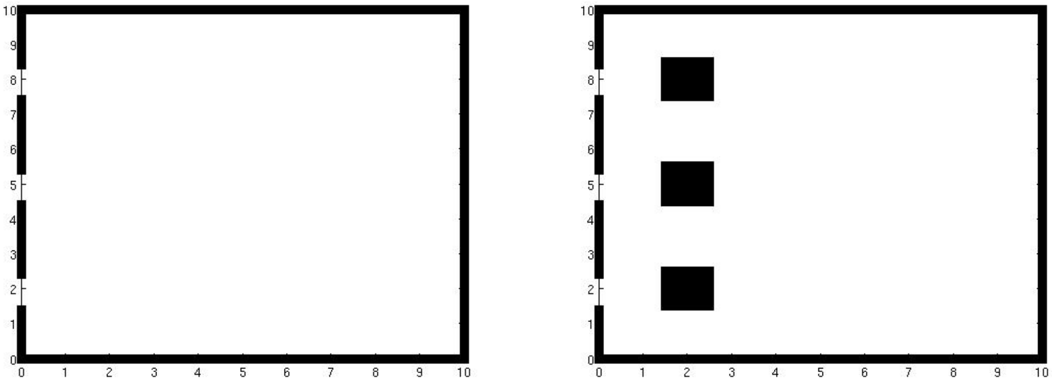

Figure 6: The rooms with and without piles used in the simulation.

# 6 Simulation Results and Discussion

The basic configuration of the simulation consists of a square room with a side length of ten units. There are three evenly spread exits, located on the west side. The exits are all of the same width and a capacity of one agent per timestep. The simulation has two scenarios, the first one is an empty room without any obstacles and the second scenario uses the same room geometry but there is a pile in the front of every door. The piles are modelled as square blocks with a sidelength of one unit. They use the same repulsive force as the wall does. (see figure 6)

There are five cases with 100, 200, 300, 400 and 500 agents. Every test case consists of twelve runs. The average of these twelve runs will be used in the analysis.

## 6.1 Exit Time Comparison

We see some differences between the two room configurations. In the configuration with the piles, it takes longer until the people start to leave the room. We think the reason may be that everybody has a direct way to the doors and the doors are visible to all if there are no piles. This means if the door is in sight, the people can estimate the queueing time so they are able to choose the door with best response in the first place. By having piles, the people only know the route to doors they are familiar to. If people get behind the piles all doors are in the line of sight. This means the possibilites of choosing an exit expands rapidly and the frequency of redicisions increases. An other explanation for the slower evacuation in the pile scenario may be that the pressure on the doors is lower. This causes a smaller force acting on the

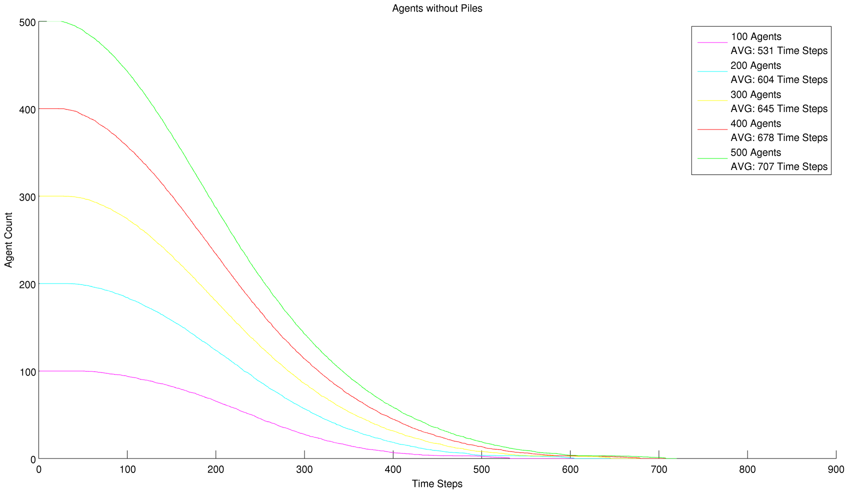

Figure 7: Exit times for different numbers of people in the room without piles

people which results in a slower evacuation. In figure 7 and 8 one can see the number of people in the room versus the time.

## 6.2 Decisions

We also have some plots were one can see, how many people changed their mind per timestep. Here we can see a big difference between the two room configurations. When we have piles, the number of people changing the door is much smaller then without the piles but it goes much longer until we have a small number of redecisions (figure 9 and 10). We think this makes perfectly sense, since due to the exit selection we implemented, a person which does not know something about a door and does not see it, would not go to that door even if it was nearest. We think this is how people would act in reality too.

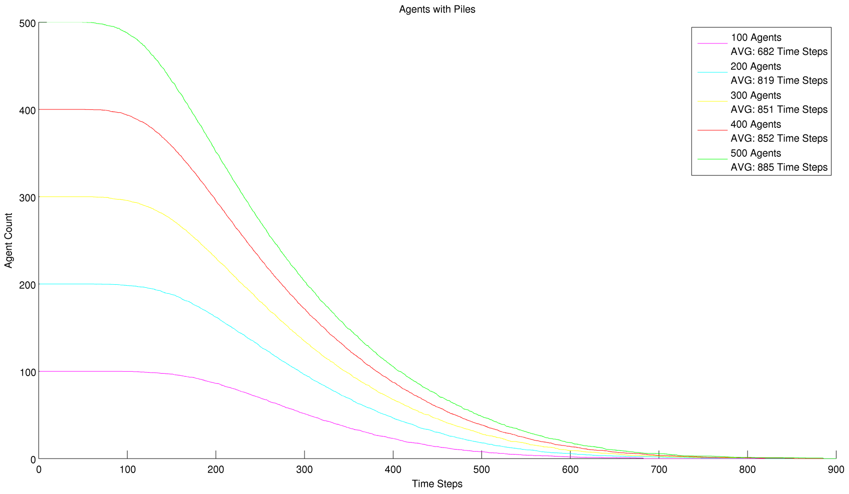

Figure 8: Exit times for different numbers of people in the room with piles

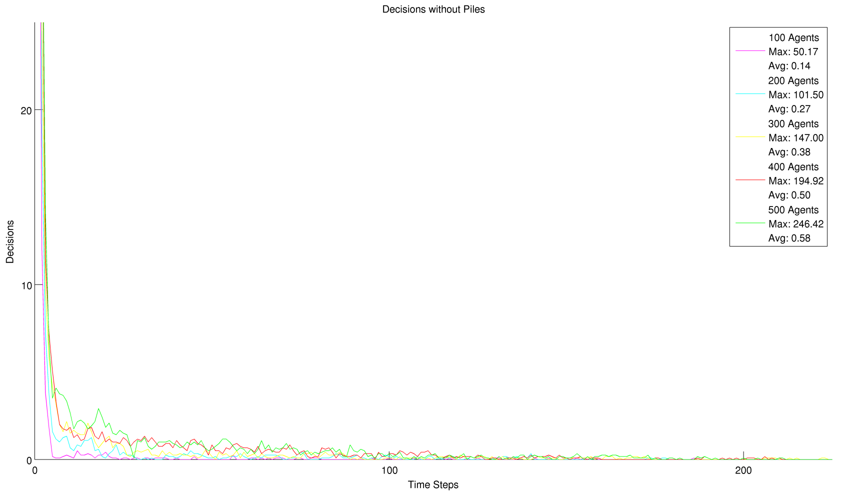

Figure 9: Number of redecisions of persons in the room without piles

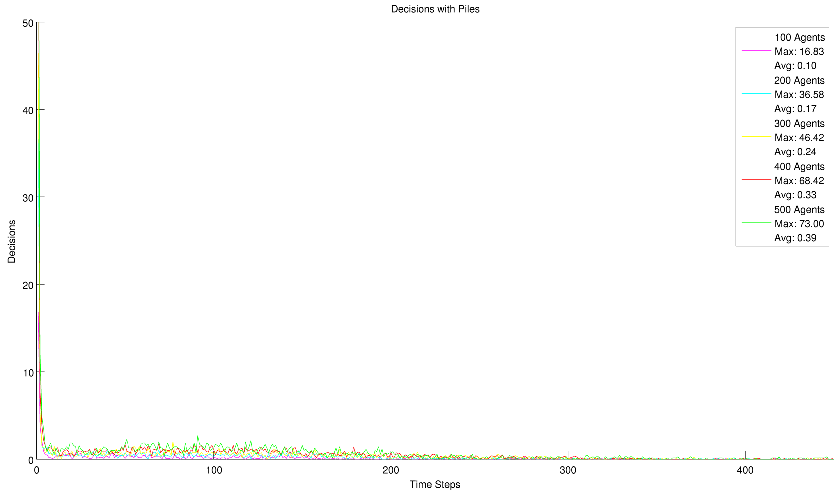

Figure 10: Number of redecisions of persons in the room with piles

## 7 Summary and Outlook

A continuous model for evacuation scenarios was implemented. By running the software, we get some charactersitics of the crowd, which also happen in reality. Additionally, the choosing of the door was done by best response dynamics. Which is a game theoretical approach. The implemented model shows crowd characteristics, such as the circular form of the crowd in front of a door, the redecition of the preferred door of people in the crowd and more.

For further work, one could possibly implement the model with different potential fields instead of the ones used. Also one could extend the static fields in such a way, that the geometry can be more complex then it is in our cases.

As a comparison, one could take the results from social experiments (9) for choosing the door and look if they give the same result. One example of such an experiment gave the following evacuation strategies:

1. I escaped according to the signs and instructions, and also broadcast or guide by shop-girls (46.7%).
2. I chose the opposite direction to the smoking area to escape from the fire as soon as possible (26.3%).
3. I used the door because it was the nearest one (16.7%).
4. I just followed the other persons (3.0%).

5. I avoided the direction where many other persons go (3.0%).
6. There was a big window near the door and you could see outside. It was the most "bright" door, so I used it (2.3%).
7. I chose the door which I am used to (1.7%).

## References

[1] K. Nishinari, A. Kirchner, A. Namazi and A. Schadschneider. 'Extended floor field CA model for evacuation dynamics.' IEICE Trans Inf Syst (Inst Electron Inf Commun Eng) VOL.E87-D;NO.3 (2006) pp(726-732).

[2] A. Kirchner, K. Nishinari and A. Schadschneider. 'Friction effects and clogging in a cellular automaton model for pedestrian dynamics.' PHYSICAL REVIEW E67 056122(2003)

[3] D. Helbing, A. Johansson, J. Mathiesen, M. H. Jensen, and A. Hansen. 'Analytical Approach to Continuous and Intermittent Bottleneck Flows.' PHYSICAL REVIEW LETTERS PRL 97 168001 (2006)

[4] W. J. Yu, R. Chen, L. Y. Dong, and S. Q. Dai. 'Centrifugal force model for pedestrian dynamics'. PHYSICAL REVIEW E72 026112 (2005)

[5] Takashi Nagatani 'Dynamical transition and scaling in a mean-field model of pedestrian flow at a bottleneck' Physica A 300 (2001) pp(558-566)

[6] Harri Ehtamo , Simo Heliövaara1 , Simo Hostikka , Timo Korhonen. 'Modeling Evacuees Exit Selection with Best Response Dynamics'. Fire Safety Journal, Volume 41, Issue 5, July 2006, Pages 364-369

[7] Hai-Jun Huang, Ren-Yong Guo. 'Static Floor Field and exit choice for pedestrian evacuation in rooms with internal obstacles and multiple exits'. Phys. Rev. E 78, 021131 (2008)

[8] Andreas Schadschneider , Ansgar Kirchner , Katsuhiro Nishinari. 'CA Approach to Collective Phenomena in Pedestrian Dynamics'. Lecture Notes In Computer Science; Vol. 2493 (2002) pp(239-248)

[9] Liana Manukyan, 'Evacuation Bottleneck', Modelling and Simulating Social Systems with MATLAB, ETH Zurich, December 2009

# Appendix A: Matlab Code

```matlab
%%% Matlab Socio %%%
% This is the main file, where the simulations should be started from.

doorW = [0.5,0.4];
cornerDist = [1,2];
pileDist = [0.5,0.5];
pileNr = [5,4];
nrP = 500;
xmax = 10;
ymax = 10;
patience = 0;

% initialization
[agentCoord, doorCoord, wallCoord, pileCoord, prefDoor, doorFam, v, rad, doorW,...
xmax, ymax] = init5(xmax, ymax, nrP, doorW, cornerDist, pileNr, pileDist);

% simulation
simulation(agentCoord, doorCoord, wallCoord, pileCoord, prefDoor,...
                     doorFam, v, rad, doorW, xmax, ymax, patience, false, '')
```

```matlab
%%% Matlab Socio %%%
% This is the debug file for logging

doorW = [0.5,0.4];
cornerDist = [1,2];
pileDist = [0.5,0.5];
pileNr = [5,4];
nrP = 500;
xmax = 10;
ymax = 10;
patience = 0;

cases = [100,200,300,400,500]; % people count
evals = 12; % 12 runs

logfile = fopen('logfile.log', 'w');



for i=1:size(cases,2)

    ppCnt = cases(1,i);
    disp(strcat('Case Nr. ', num2str(i), ' - ', num2str(ppCnt), '\n'));

    % -100,[peopleCount]   // -100 defines a case
```

```
    fprintf(logfile, strcat('-100,',num2str(ppCnt),'\n'));

    for j=1:evals
        disp(strcat('---> Run Nr. ', num2str(j), '\n'));

        % -200,[runNr] // -200 defines a run
        fprintf(logfile, strcat('-200,',num2str(j),'\n'));

        % init
        [agentCoord, doorCoord, wallCoord, pileCoord, prefDoor, doorFam, v, rad, doorW,...
        xmax, ymax] = init5(xmax, ymax, ppCnt, doorW, cornerDist, pileNr, pileDist);

        % simulate
        simulation(agentCoord, doorCoord, wallCoord, pileCoord, prefDoor,...
                                     doorFam, v, rad, doorW, xmax, ymax, patience, true, logfile);

    end

end

fclose(logfile);
```

```
function [i] = simulation( agentCoord, doorCoord, wallCoord, pileCoord, ...
    prefDoor, doorFam, v, rad, doorW, xmax, ymax, patience, debug, logf)
% The function simulation is the main file, where the simulation runs.
%
% INPUT:
% The *Coord Matrices should all be N x 2, where the N is the number of
% elements and 2 is the corresponding x and y coordinate.
% agentCoord  ... The coordinates of the people.
% doorCoord   ... The coordinates of the doors (i.e. the middle of the door)
% wallCoord   ... The coordinates of the wall-"people". These are particles,
%                 which don't move, thus represent wall-elements.
% prefDoor    ... This gives the currently prefered door of the people, it's
%                 a vector with one entry for each person in agentCoord. The
%                 index of the value corresponds to the person with the same
%                 index in the matrix agentCoord
% v           ... These should be the initial velocities of the people. It
%                 should have the same size as agentCoord.
% rad         ... This gives how big persons are.
% doorW       ... For each Door, we need to know its size.
% xmax, ymax  ... The dimensions of the room.
% patience    ... This is a parameter, which describes how patience the
%                 people are with their door.
% debug       ... Defines if we shall log anything
% logf        ... Handle to logfile
%
% OUPUT:
```

```
% The return value indicates how long it took until all persons left the
% room.


colors = ['m', 'c', 'y', 'r', 'g', 'b'];

%% Parameters
% maximal running time
Time = 10;

% step size of the time integration
dt = 10^-2;

% maximal velocity an agent can have
vmax = [10,10];

% how much one takes the old velocity into account
oldPartV = 0.5;

% the probability of reevaluate the doors to choose
probDoorUpdate = 1;

%% Statistics initialization
%initially door chosen
chosenDoor = [];
exitThrough = [];

for k=1:size(doorW,2)
    chosenDoor(1,k) = length(prefDoor(prefDoor == k));
end
    exitThrough = zeros(numel(doorW));


%% Time integration
% the time integration is done by a simple explicit euler time stepping
for i = 0:dt:Time
%      i %#ok<NOPRT>

    decisionChanges = 0;
    activeAgents    = 0;


    % in which order the agents are updated
    whichOne = randperm(size(agentCoord,1));

    % update all the agents for this timestep
    for j = 1:size(agentCoord,1)
       currAgent = whichOne(j);

       % coordinates of the current agent
```

```
        currx = agentCoord(currAgent,1);
        curry = agentCoord(currAgent,2);

        % if the current agent has already left the room, continue.
        if (currx > xmax || curry > ymax || currx < 0 || curry < 0)
            continue;
        end

        % reconsider the preferred door
        oldPrefDoor = prefDoor(currAgent);

        if (rand(1) ≤ probDoorUpdate)
            [prefDoor(currAgent), doorFam] = ...
                 basic2(currAgent, agentCoord, v, prefDoor, doorCoord, ...
                 doorW, patience, wallCoord, pileCoord, doorFam, rad);
        end

        if oldPrefDoor ≠ prefDoor(currAgent)
            decisionChanges = decisionChanges + 1;
        end

        % calculate the current acceleration
        dv = − force(currAgent, agentCoord, wallCoord, doorCoord, rad, ...
             prefDoor(currAgent), doorW, xmax, ymax);

        % update the velocity and ensure, it is not faster then the max
        % velocity
        v(currAgent, :) = 0.5 * max(min(oldPartV * v(currAgent,:) +  dt * dv,...
             vmax), −vmax);

        % update the coordinates
        agentCoord(currAgent, :) = agentCoord(currAgent, :) + dt ...
             .* v(currAgent,:);

        % test if we have left the room after this step
        currx = agentCoord(currAgent,1);
        curry = agentCoord(currAgent,2);
        if (currx > xmax || curry > ymax || currx < 0 || curry < 0)
            agentCoord(currAgent,:) = [−100. −100];
            v(currAgent,:) = [0,0];
            exitThrough(prefDoor(currAgent)) = ...
                 exitThrough(prefDoor(currAgent)) + 1;
            prefDoor(currAgent) = −1;
        end

     end



      % plot everything if not in debug mode
```

```
     if debug == false
         figure(1);

         plot(wallCoord(:,1), wallCoord(:,2), 's', 'MarkerEdgeColor', 'k', ...
                'MarkerFaceColor','k', 'MarkerSize', 7);
         hold on;



         for k=1:size(doorW,2)
             plot(agentCoord(prefDoor == k,1), agentCoord(prefDoor == k,2),...
                'o', 'MarkerEdgeColor', colors(1,k), 'MarkerFaceColor',...
                colors(1,k), 'MarkerSize', 7);
         end

         plot(agentCoord(prefDoor == 0,1), agentCoord(prefDoor == 0,2),...
                    'o', 'MarkerEdgeColor', 'k', 'MarkerFaceColor','k', ...
                    'MarkerSize', 7);

         plot(wallCoord(:,1), wallCoord(:,2), 's', 'MarkerEdgeColor', 'k', ...
                'MarkerFaceColor','k', 'MarkerSize', 7);
         axis([-0.01, xmax+0.01, -0.01, ymax+0.01]);
         daspect([1,1,1]);
         set(gca,'XTickLabel','');
         set(gca,'YTickLabel','');

        % there has to be a folder "../bilder" that the pictures can be saved
        % comment the next three lines if you don't want to save every step
        %nameStr = sprintf('../bilder/2sociSim_patience%03.1f_%05.2f.png',...
        %    patience, i);
        %saveas(1,nameStr,'png');
         hold off;
     end

    for k=1:size(doorW,2)
        chosenDoor(k) = length(prefDoor(prefDoor == k)) + exitThrough(k);
        exitThrough(k) = 0;
    end

    activeAgents = length(prefDoor(prefDoor > -1));

    if debug == true
        % log
        fprintf(logf, strcat(num2str(activeAgents),',',num2str(decisionChanges),'\n'));
    end

    % exit integration if no one is in the room left
    if (isempty(prefDoor(prefDoor > -1)))
        break;
    end
```

```


end

%% Statistic plots
%figure(2);
%plot(chosenDoor(1:numel(chosenDoor(1,:)),:) * 100);
%xlabel('step number');
%ylabel('%');
%axis([0, index, 0, 100]);
%legend('upper door', 'lower door');
%title([num2str(exitThrough(1)),' / ', num2str(exitThrough(2))])


end
```

```
function [f] = force(agentNr, agentCoord, wallCoord, doorCoord, rad,...
    prefDoor, doorW, xmax, ymax)
% calculates the force acting on the agent
% with the number agentNr
%
% INPUT:
% agentNr       ... the number of the agent, we want to
%                   forces for.
% agentCoord    ... the coordinates of all agents.
% wallCoord     ... the coordinates of the wall—elements.
% coorCoord     ... the coordinates of the doors.
% rad           ... the size of the agents in agentCoord.
% prefDoor      ... the number of the prefered door of agent with agentNr.
%
% OUTPUT:
% The forces acting on agent with agentNr as a two dimensinal vector.

% parameter for the wall
wallR = 1.5;

% initialize the forces
f = [0,0];
potA = zeros(2,1);
potD = potA;
potW = potA;

% first calculate forces from agents
for i = 1:size(agentCoord,1)

    % we don't have a force coming
    % form ourselves.
    if (i == agentNr)
```

```
        continue;
    end;

    acor = agentCoord(agentNr,:);
    bcor = agentCoord(i,:);
    dist = norm(acor - bcor);
    % only calculate the force, if we are in
    % the others radius
    if (rad(i) > dist)
        potA = potAgent(acor, bcor);
        f = f + potA(:)';
    end
end

% then the wall-forces
for i = 1:size(wallCoord,1);
   dist = norm(agentCoord(agentNr, :) - wallCoord(i,:));
   % only calculate the force, if we are
   % within the radius of a wall element.
   if (dist < wallR)
       potW = potWall(agentCoord(agentNr, :), wallCoord(i,:));
       f = f + potW(:)';
   end
end

% and finally door-force

% if he has no door preference, let him move around randomly
if prefDoor > 0

    potD = potDoor(agentCoord(agentNr,:), doorCoord(prefDoor,:),...
        doorW(prefDoor), xmax, ymax);

    f = f + potD(:)';
end
end
```

```
function [prefDoorID, door_fams]  = basic2(aid, agent_coords, ...
    agent_speeds, agent_prefs, door_coords, door_caps, patience,...
    wall_coords, pile_coords, door_fams, peopleRad)
% This function calculates the door we prefere at our current
% position and velocity.
%
% aid            = Agent ID
% agents         = Vector of all Agents
% agent_coords   = Agent Positions
% agent_speeds   = Agent Speeds
% agent_prefs    = Agent's Preferred Doors
```

```
% doors          = Vector of all Doors
% door_coords    = Door Positions
% door_caps      = Door Capacitivities
% patience       = how much better an other door needs to be to be chosen

    % init
    agent_pos       = agent_coords(aid,:)';
    agent_vel       = agent_speeds(aid,:)';
    door_caps       = door_caps';

    d_weights       = [];
    d_vis           = [];

    prefDoorID              = 0;


    old_door        = agent_prefs(aid);

    % get weigthing for doors

    for i=1:size(door_coords,1)

        d_vis(i) = is_vis(aid, i, agent_coords, door_coords, wall_coords,...
            pile_coords);

        if is_fam(aid, i, door_fams) == 1 && d_vis(i) == 1
            % door is visible and familiar
            d_weights(i) = 1;
        elseif is_fam(i, i, door_fams) == 1 && d_vis(i) == 0
             % door is familiar but not visible
            d_weights(i) = 2;
        elseif is_fam(aid, i, door_fams) == 0 && d_vis(i) == 1
            % door is visible but not familiar
            d_weights(i) = 3;
        else
            % door is invisible and not familiar
            d_weights(i) = 4;
        end

    end

    % select the group with the best (lowest) preference numbers

    bPrefNr     = min(d_weights);

    % worst case, person doesn't know any doors and can't see any
    if bPrefNr   == 4
        % he goes panic!!!!
        prefDoorID = 0;
    end
```

```
    if bPrefNr < 4

        % get best group of door indices
        bDoorInd    = find(d_weights == bPrefNr)';
        d_time      = zeros(size(bDoorInd,1), 1);
        d_time_raw  = zeros(size(bDoorInd,1), 1);

        % loop through these doors and find the one with the
        % best waiting time

        for i=1:size(bDoorInd,1)

            % door capacity (people per time step it can take
            bk  = 1/(door_caps(bDoorInd(i))*10);

            % estimated moving time:
            est_mtime   = distance_time(norm(agent_pos -...
                door_coords(bDoorInd(i),:)'), agent_vel);

            % estimated queueing time
            est_qtime   = bk * get_queue_count(bDoorInd(i), aid,...
                agent_coords, agent_prefs, door_coords);


            % we cannot calculate the queue time if the door is not visible!
            d_time_raw(i) = est_mtime + est_qtime;
            est_qtime   = d_vis(bDoorInd(i))*est_qtime;

            d_time(i)   = est_mtime + est_qtime;

        end

        % get the best one!

        prefDoorID     = bDoorInd(find(d_time == min(d_time), 1, 'first'));
    end

    % calculate time of old door

    % door capacity (people per time step it can take
    bk  = 1/(door_caps(old_door)*10);

    % estimated moving time:
    est_mtime   = distance_time(norm(agent_pos -...
        door_coords(old_door,:)'), agent_vel);

    % estimated queueing time
    est_qtime   = bk * get_queue_count(old_door, aid, agent_coords,...
        agent_prefs, door_coords);
```

```
    % we cannot calculate the queue time if the door is not visible!
    est_qtime   = d_vis(old_door)*est_qtime;

    old_time = est_mtime + est_qtime;

    % compare new preferable door and the old one, only take the new one
    % if it is better!
    if old_time ≤ d_time_raw(find(d_time == min(d_time), 1, 'first'))
        prefDoorID = old_door;
    end


end
```

```
function [pot] = potAgent(xp, xq)
% potential between agents with positions
% xp and xq

pot = zeros(2,1);

div = (xp(1)^2 - 2*xp(1)*xq(1) + xq(1)^2 + ...
       xp(2)^2 - 2*xp(2)*xq(2) + xq(2)^2 )^(3/2);

pot(1) = - 15.84893192 * (xp(1) - xq(1))/ div;
pot(2) = - 15.84893192 * (xp(2) - xq(2))/ div;

end
```

```
function [pot] = potWall(xp, xq)
% potential between agent xp and wall-element at xq.
pot = zeros(2,1);

div = (xp(1)^2 - 2*xp(1)*xq(1) + xq(1)^2 + ...
       xp(2)^2 - 2*xp(2)*xq(2) + xq(2)^2 )^(3/2);

pot(1) = - 5 * (xp(1) - xq(1))/ div;
pot(2) = - 5 * (xp(2) - xq(2))/ div;
```

```
function [pot] = potDoor(xp, xq, width, xmax, ymax)
% Potential between an agent  and doors
% The doors are not just a point source, they are
% stretched, so that the the field is computed
% from multiple points,
```

```
%
% INPUT:
% xp    ... position of an agent.
% xq    ... position of a door middle.
% width ... width of the door xq.
% xmax  ... roomwidth in x direction.
% ymax  ... roomwidth in y direction.

% initial potential from the door
pot = zeros(2,1);

% describes the how far the points in the stretched
% potential are from each other.
eps = 0.01;

% make the potential field not only from a point.
if (xq(1) ≥ xmax || xq(1) ≤ 0)
    yCoords = (0:eps:width)' + xq(2) - width/2;
    iter = [xq(1) * ones(size(yCoords)), yCoords];
else
    xCoords = (0:eps:width)' + xq(1) - width/2;
    iter = [xCoords, xq(2) * ones(size(xCoords))];
end

% iterate over all created points from above
iterSize = size(iter,1);
for i = 1:iterSize
    div = norm(xp - iter(i,:));
    pot(1) = pot(1) + 60 * (div + 4) * (xp(1) - iter(i,1)) / (div *iterSize);
    pot(2) = pot(2) + 60 * (div + 4) * (xp(2) - iter(i,2)) / (div *iterSize);
end
```

```
function [ time ] = distance_time(dist, speed)
% Calculate Travelling Time if we can hold our speed
    time = dist / sqrt(speed(1)^2 + speed(2)^2);

end
```

```
function [queue] = get_queue_count(did, aid, agent_coords, agent_prefs,...
    door_coords)
% This function computes, how many people are in front of agent did
% and are heading for the same door
%
% did            = Door ID
% aid            = Agent ID
% agents         = Vector of all Agents
% agent_coords   = Agent Coordinates
```

```
% agent_prefs   = Agent's preferred Door
% doors         = Vector of all Doors
% door_coords   = Door Coordinates


% Returns queue count of agents heading in direction of Door did


    agent_dist      = norm(agent_coords(aid,:)' - door_coords(did,:)');
    queue           = 0;

    for i=1:size(agent_coords, 1)

        c_did      = agent_prefs(i);

        % exclude our agent and agents heading for a different door %
        if(i == aid || c_did ≠ did)
           continue
        end


        c_dist      = norm(agent_coords(i,:)' - door_coords(c_did,:)');

        if(c_dist ≤ agent_dist)
            queue = queue + 1;
        end
    end

end
```

```
function [vis] = is_vis(aid, did, agent_coords, door_coords,...
    wall_coords, pile_coords)

    % input:
    %   aid:    agent id
    %   did:    door id
    %   agent_coords: coordinate matrix of all agents
    %   door_coords: coordinate matrix of all doors
    %   wall_coords: coordinate matrix of all walls
    %   pile_coords: coordinate matrix of all piles

    % output:
    %   returns 1 if door is visible to agent
    %   returns 0 if door is invisible for agent


    % is door "did" visible to agent "aid" Default: true
    vis  = 1;
```

```

    % door doesnt exist
    if did == 0
        % not visible
        vis = 0;
        return;
    end

    % accuracy (resolution) same as walls/piles
    Weps = 0.1;

    % get agent's position
    agentCX  = agent_coords(aid, 1);
    agentCY  = agent_coords(aid, 2);

    % get the door's position
    doorCX   = door_coords(did, 1);
    doorCY   = door_coords(did, 2);

    % gradient of the line between agent and the middle of the door
    lineGrad = (doorCY - agentCY)/(doorCX - agentCX);


    % rectangle between agent and door (interval)
    rectLeft    = doorCX;
    rectRight   = agentCX;
    rectTop     = agentCY;
    rectBottom  = doorCY;

    % swap boundaries of rectangle if necessary
    if rectLeft > rectRight
        tmpLeft = rectLeft;
        rectLeft = rectRight;
        rectRight = tmpLeft;
    end

    if rectBottom > rectTop
        tmpBottom = rectBottom;
        rectBottom = rectTop;
        rectTop    = tmpBottom;
    end


    % loop through all piles
    for i=1:size(pile_coords,1)

        % pile coordinates
        pileX   = pile_coords(i, 1);
        pileY   = pile_coords(i, 2);

```

```
        % check if pile is out of the rectangle
        if pileX < rectLeft || pileX > rectRight...
            || pileY < rectBottom || pileY > rectTop
            % if yes, the pile is not of any interest, skip
            continue;
        end

        % check if pile is on the sight-line!
        tmpY    = round((lineGrad * (pileX - agentCX) ...
            + agentCY)*(1/Weps))/(1/Weps);

        if pileY == tmpY
            %hold on;
            %plot([agentCX, doorCX], [agentCY, doorCY]);

            % the pile is in the agent's sightline to the door
            % the door is not visible to the agent
            vis = 0;
            return;
        end

    end


end
```

```

function [fam] = is_fam(aid, did, famDoors)
    % input:
    %   aid: agent id
    %   did: door id
    %   famDoors: a matrix with a row for each agent and one column for
    %   ...each door with a binary flag (known/unknown)

    % output:
    % returns 0 if door (did) is not familiar to agent (aid)
    % returns 1 if door (did) is familiar to agent (aid)
    fam = 0;

    if famDoors(aid, did) ≠ 0
        fam = 1;
    end


end
```

```
function [agentCoord, doorCoord, wallCoord, pileCoord, prefDoor, doorFam, ...
```

```
    v, rad, doorW, xmax, ymax] = init1(xmax, ymax, nrPeople, doorW)
% This function creates a world, where we have four doors, which are
% located in the middle of all the walls. With:
% — the first door in the north
% — the second door in the south
% — the third door in the east
% — the fourth door in the west
%
% INPUT:
% xmax, ymax    ... the dimensions of the room
% nrPeople      ... how many people it will have in the room
% doorw         ... the widths of the doors, Must contain four
%                   values. If a value is smaller or equal to
%                   zero, the door will not be place.
%
% OUTPUT:
% agentCoord ... The coordinates of the people.
% doorCoord  ... The coordinates of the doors (i.e. the middle of the door)
% wallCoord  ... The coordinates of the wall—"people". These are particles,
%                which don't move, thus represent wall—elements.
% prefDoor   ... This gives the currently prefered door of the people, it's
%                a vector with one entry for each person in agentCoord. The
%                index of the value corresponds to the person with the same
%                index in the matrix agentCoord
% v          ... These should be the initial velocities of the people. It
%                should have the same size as agentCoord.
% rad        ... This gives how big persons are.
% doorW      ... For each Door, we need to know its size.
% xmax, ymax ... The dimensions of the room.
% patience   ... This is a parameter, which describes how patience the
%                people are with their door.

%% Parameters
Deps = 0;
Weps = 0.1;
peopleRad = 0.75;

%% The room
wallCoord = [];

middlex = xmax/2;
middley = ymax/2;

% test if doorwidths are smaller or equal to the maximum size
% of the wall, else shrink it to that size
doorW(1) = min(doorW(1), xmax);
doorW(2) = min(doorW(2), xmax);
doorW(3) = min(doorW(3), ymax);
doorW(4) = min(doorW(4), ymax);

```

```
% construct the north wall
leftN = (0:Weps:(middlex - doorW(1)/2))';
rightN = (middlex + doorW(1)/2:Weps:xmax)';
northWall = [ leftN, ymax * ones(length(leftN), 1)];
northWall = [northWall; [rightN, ymax * ones(length(rightN), 1)]];

% construct the south wall
leftS = (0:Weps:(middlex - doorW(2)/2))';
rightS = (middlex + doorW(2)/2:Weps:xmax)';
southWall = [ leftS, zeros(length(leftS), 1)];
southWall = [southWall; [rightS, zeros(length(rightS), 1)]];

% construct the east wall
lowerE = (0:Weps:middley - doorW(3)/2)';
upperE = (middley + doorW(3)/2:Weps:ymax)';
eastWall = [xmax * ones(length(lowerE), 1), lowerE];
eastWall = [eastWall; [xmax * ones(length(upperE), 1), upperE]];

% construct the west wall
lowerW = (0:Weps:middley - doorW(4)/2)';
upperW = (middley + doorW(4)/2:Weps:ymax)';
westWall = [zeros(length(lowerW), 1), lowerW];
westWall = [westWall; [zeros(length(upperW), 1), upperW]];

% put all the walls into one matrix
wallCoord = [wallCoord; northWall; southWall; westWall; eastWall];

pileCoord = [];
doorFam = ones(nrPeople, numel(doorW(doorW ≠ 0)));
%% Doors
doorCoord = [];
fak = 2;

% set the doors
% if the width of a door is smaller or equal to zero, it will
% not be placed
if (doorW(1) > 0)
    doorCoord = [doorCoord; [middlex, ymax+Deps * doorW(1)/fak]];
end

if (doorW(2) > 0)
    doorCoord = [doorCoord; [middlex, -Deps * doorW(2)/fak]];
end

if (doorW(3) > 0)
    doorCoord =[doorCoord; [xmax+Deps * doorW(3)/fak, middley]];
end

if (doorW(4) > 0)
    doorCoord =[doorCoord;[-Deps * doorW(4)/fak, middley]];
```

```
end
doorW = doorW(doorW > 0);


%% People
% place the people
agentCoord = rand(nrPeople,2) .* repmat([xmax, ymax],nrPeople, 1);
prefDoor = ceil(rand(nrPeople,1) .* size(doorCoord,1));
rad = peopleRad * ones(nrPeople,1);
v = zeros(nrPeople, 2);

% test if the people have chosen a valid door
for i = 1:nrPeople
    while (doorW(prefDoor(i)) == 0)
        prefDoor(i) = ceil(rand(1) * size(doorCoord,1));
    end
end

% set value and direction of the initial velocities
% of the people
for i = 1:nrPeople
    dir = doorCoord(prefDoor(i),:) - agentCoord(i,:);
    v(i,:) = (dir./norm([xmax,ymax])) * norm([15,15]);
end

end
```

```
function [agentCoord, doorCoord, wallCoord, pileCoord, prefDoor, doorFam,...
    v, rad, doorW, xmax, ymax] = init2(xmax, ymax, nrPeople, doorW, doorDist)
% This function gives a room back, which has two doors at one wall,
% the west wall
%
% INPUT:
% xmax, ymax     ... the dimensions of the room.
% nrPeople       ... how many people it will have in the room.
% doorW          ... the width of the doors.
% doorDist       ... the distance of between the two doors.
%
% OUTPUT:
% agentCoord ... The coordinates of the people.
% doorCoord  ... The coordinates of the doors (i.e. the middle of the door)
% wallCoord  ... The coordinates of the wall-"people". These are particles,
%                which don't move, thus represent wall-elements.
% prefDoor   ... This gives the currently prefered door of the people, it's
%                a vector with one entry for each person in agentCoord. The
%                index of the value corresponds to the person with the same
%                index in the matrix agentCoord
% v          ... These should be the initial velocities of the people. It
```

```matlab
%                should have the same size as agentCoord.
% rad        ... This gives how big persons are.
% doorW      ... For each Door, we need to know its size.
% xmax, ymax ... The dimensions of the room.
% patience   ... This is a parameter, which describes how patience the
%                people are with their door.

%% Parameters
% some parameters for the doors
Deps = 0;
fak = 2;

% the distance between two wall elements
Weps = 0.1;

% the size of the people
peopleRad = 0.75;

%% the room
% we will have here only two doors. which will be next to each other.
pileCoord = [];
doorFam = ones(nrPeople, 2);

% the full walls
northWall = 0:Weps:xmax;
northWall = northWall(:);
northWall = [northWall, ymax * ones(size(northWall))];

southWall = 0:Weps:xmax;
southWall = southWall(:);
southWall = [southWall, zeros(size(southWall))];

eastWall = 0:Weps:ymax;
eastWall = eastWall(:);
eastWall = [xmax * ones(size(eastWall)), eastWall];

% constuction of the wall, which contains the doors.
doorDist = min(ymax/2, doorDist);
doorW(1) = min(doorW(1), (ymax - doorDist)/2);
doorW(2) = min(doorW(2), (ymax - doorDist)/2);

lower = 0:Weps: ymax/2 - doorW(2) - doorDist/2;
middle = (0:Weps:doorDist) + ymax/2 - doorDist/2;
upper = ymax/2 + doorDist/2 + doorW(1):Weps:ymax;
lower = lower(:); middle = middle(:); upper = upper(:);

westWall = [ zeros(size(lower)), lower; zeros(size(middle)), middle; ...
    zeros(size(upper)), upper];

% put all the walls into one matrix
```

```
wallCoord = [northWall; southWall; westWall; eastWall];

%% Doors
doorCoord = [-Deps * doorW(1)/fak, ymax/2 + doorDist/2 + doorW(1)/2; ...
    -Deps * doorW(2)/fak, ymax/2 - doorDist/2 - doorW(2)/2];


%% People
% place the people
agentCoord = rand(nrPeople,2) .* repmat([xmax, ymax],nrPeople, 1);
prefDoor = ceil(rand(nrPeople,1) .* size(doorCoord,1));
rad = peopleRad * ones(nrPeople,1);
v = zeros(nrPeople, 2);

% test if the people have chosen a valid door
for i = 1:nrPeople
    while (doorW(prefDoor(i)) == 0)
        prefDoor(i) = ceil(rand(1) * size(doorCoord,1));
    end
end

% set value and direction of the initial velocities
% of the people
for i = 1:nrPeople
    dir = doorCoord(prefDoor(i),:) - agentCoord(i,:);
    v(i,:) = (dir./norm([xmax,ymax])) * norm([15,15]);
end
```

```
function [agentCoord, doorCoord, wallCoord, pileCoord, prefDoor, doorFam,...
    v, rad, doorW, xmax, ymax] = init3(xmax, ymax, nrPeople, doorW,...
    distToCorner)
% This function creates a world, where the two doors are at one corner
% The first door lies in the west wall, the second in the south wall
%
% INPUT:
% xmax, ymax    ... the dimensions of the room
% nrPeople      ... how many people it will have in the room
% doorW         ... the width of the doors (doorW(1), west
%                   door; doorW(2), southDoor)
% distToCorner  ... the distance of the doors form the corner
%                   in south-west
%
% OUTPUT:
% agentCoord ... The coordinates of the people.
% doorCoord  ... The coordinates of the doors (i.e. the middle of the door)
% wallCoord  ... The coordinates of the wall-"people". These are particles,
%                which don't move, thus represent wall-elements.
% prefDoor   ... This gives the currently prefered door of the people, it's
```

```
%                 a vector with one entry for each person in agentCoord. The
%                 index of the value corresponds to the person with the same
%                 index in the matrix agentCoord
% v           ... These should be the initial velocities of the people. It
%                 should have the same size as agentCoord.
% rad         ... This gives how big persons are.
% doorW       ... For each Door, we need to know its size.
% xmax, ymax ... The dimensions of the room.
% patience    ... This is a parameter, which describes how patience the
%                 people are with their door.

%% Parameters
% some parameters for the doors
Deps = 0;
fak = 2;

% the distance between two wall elements
Weps = 0.1;

% the size of the people
peopleRad = 0.75;

%% the room
% boarder walls
pileCoord = [];
doorFam = ones(nrPeople, 2);

% the full walls
northWall = 0:Weps:xmax;
northWall = northWall(:);
northWall = [northWall, ymax * ones(size(northWall))];

eastWall = 0:Weps:ymax;
eastWall = eastWall(:);
eastWall = [xmax * ones(size(eastWall)), eastWall];

% correct the parameters if they are to big.
distToCorner(1) = min(ymax, distToCorner(1));
distToCorner(2) = min(xmax, distToCorner(2));

doorW(1) = min(doorW(1), ymax - distToCorner(1));
doorW(2) = min(doorW(2), xmax - distToCorner(2));

% the construction of the south wall, which includes
% one door
southLeft = 0:Weps:distToCorner(2);
southLeft = southLeft(:);
southRight = distToCorner(2) + doorW(2):Weps:xmax;
southRight = southRight(:);
southWall = [southLeft, zeros(size(southLeft));...
```

```
    southRight, zeros(size(southRight))];

% the construction of the west wall, which includes
% one door
westLower = 0:Weps:distToCorner(1);
westLower = westLower(:);
westUpper = distToCorner(1) + doorW(1):Weps:ymax;
westUpper = westUpper(:);
westWall = [ zeros(size(westLower)), westLower;...
    zeros(size(westUpper)), westUpper];

% put all the walls into one matrix
wallCoord = [northWall; southWall; westWall; eastWall];

% set the doors
doorCoord = [-Deps * doorW(1)/fak, distToCorner(1) + doorW(1)/2; ...
    distToCorner(2) + doorW(2)/2, -Deps * doorW(2)/fak];
doorW = doorW(1:2);

%% People
% place the people
agentCoord = rand(nrPeople,2) .* repmat([xmax, ymax],nrPeople, 1);
prefDoor = ceil(rand(nrPeople,1) .* size(doorCoord,1));



rad = peopleRad * ones(nrPeople,1);
v = zeros(nrPeople, 2);

% test if the people have chosen a valid door
for i = 1:nrPeople
    while (doorW(prefDoor(i)) == 0)
        prefDoor(i) = ceil(rand(1) * length(doorW));
    end
end

% set value and direction of the initial velocities
% of the people
for i = 1:nrPeople
    dir = doorCoord(prefDoor(i),:) - agentCoord(i,:);
    v(i,:) = (dir./norm([xmax,ymax])) * norm([15,15]);
end
```

```
function [agentCoord, doorCoord, wallCoord, pileCoord, prefDoor, doorFam,...
    v, rad, doorW, xmax, ymax] = init4(xmax, ymax, nrPeople, ...
    doorW, distToCorner, pileNr, pileDist)
% This function creates a world, where the two doors are at one corner
% The first door lies in the west wall, the second in the south wall
```

```
% additionally, the doors have piles in front of it.
%
% INPUT:
% xmax, ymax    ... the dimensions of the room
% nrPeople      ... how many people it will have in the room
% doorW         ... the width of the doors (doorW(1), west
%                   door; doorW(2), southDoor)
% distToCorner  ... the distance of the doors form the corner
%                   in south—west
% pileNr        ... for each door the number of piles in front
% pileDist      ... the distance of the piles from the door (2dim vector)
%
% OUTPUT:
% agentCoord ... The coordinates of the people.
% doorCoord  ... The coordinates of the doors (i.e. the middle of the door)
% wallCoord  ... The coordinates of the wall—"people". These are particles,
%                which don't move, thus represent wall—elements.
% prefDoor   ... This gives the currently prefered door of the people, it's
%                a vector with one entry for each person in agentCoord. The
%                index of the value corresponds to the person with the same
%                index in the matrix agentCoord
% v          ... These should be the initial velocities of the people. It
%                should have the same size as agentCoord.
% rad        ... This gives how big persons are.
% doorW      ... For each Door, we need to know its size.
% xmax, ymax ... The dimensions of the room.

%% Parameters
% some parameters for the doors
Deps = 0;
fak = 2;

% the distance between two wall elements
Weps = 0.1;
Peps = 0.5;

% the size of the people
peopleRad = 0.75;

%% the room
% boarder walls
wallCoord = [];
pileCoord = [];

% the full walls
northWall = 0:Weps:xmax;
northWall = northWall(:);
northWall = [northWall, ymax * ones(size(northWall))];

eastWall = 0:Weps:ymax;
```

```
eastWall = eastWall(:);
eastWall = [xmax * ones(size(eastWall)), eastWall];

% correct the parameters if they are to big.
distToCorner(1) = min(ymax, distToCorner(1));
distToCorner(2) = min(xmax, distToCorner(2));

doorW(1) = min(doorW(1), ymax - distToCorner(1));
doorW(2) = min(doorW(2), xmax - distToCorner(2));

% the construction of the south wall, which includes
% one door
southLeft = 0:Weps:distToCorner(2);
southLeft = southLeft(:);
southRight = distToCorner(2) + doorW(2):Weps:xmax;
southRight = southRight(:);
southWall = [southLeft, zeros(size(southLeft));...
    southRight, zeros(size(southRight))];

% the construction of the west wall, which includes
% one door
westLower = 0:Weps:distToCorner(1);
westLower = westLower(:);
westUpper = distToCorner(1) + doorW(1):Weps:ymax;
westUpper = westUpper(:);
westWall = [ zeros(size(westLower)), westLower;...
    zeros(size(westUpper)), westUpper];

% add the piles
if pileNr(1) > 0
    if pileNr(1) == 1
        westPiles = [pileDist(1), ...
            (doorW(1)/2 + distToCorner(1))'];
    else
        westPiles = [ones(pileNr(1),1) * pileDist(1),...
            ((0:Peps:Peps*(pileNr(1)-1)) + distToCorner(1) + ...
            doorW(1)/2  - Peps*(pileNr(1)-1)/2)'];
    end
    wallCoord = [wallCoord; westPiles];
end

if pileNr(2) > 0
    if pileNr(2) == 1
        westPiles = [(doorW(2)/2 + distToCorner(2))',...
            pileDist(2)];
    else
        westPiles = [((0:Peps:Peps*(pileNr(2)-1)) + distToCorner(2) + ...
            doorW(2)/2  - Peps*(pileNr(2)-1)/2)', ...
            ones(pileNr(2),1) * pileDist(2)];
    end
```

```
    wallCoord = [wallCoord; westPiles];
end

% put all the walls into one matrix
wallCoord = [wallCoord; northWall; southWall; westWall; eastWall];

% set the doors
doorCoord = [-Deps * doorW(1)/fak, distToCorner(1) + doorW(1)/2; ...
    distToCorner(2) + doorW(2)/2, -Deps * doorW(2)/fak];
doorW = doorW(1:2);
doorFam = ones(nrPeople, 2);

%% People
% place the people
agentCoord = rand(nrPeople,2) .* repmat([xmax, ymax],nrPeople, 1);
prefDoor = ceil(rand(nrPeople,1) .* 2);
rad = peopleRad * ones(nrPeople,1);
v = zeros(nrPeople, 2);

% test if the people have chosen a valid door
% for i = 1:nrPeople
%     while (doorW(prefDoor(i)) == 0)
%         prefDoor(i) = ceil(rand(1) * length(doorW));
%     end
% end

% set value and direction of the initial velocities
% of the people
for i = 1:nrPeople
    dir = doorCoord(prefDoor(i),:) - agentCoord(i,:);
    v(i,:) = (dir./norm([xmax,ymax])) * norm([5,5]);
end
```

```
function [agentCoord, doorCoord, wallCoord, pileCoord, prefDoor, doorFam,...
    v, rad, doorW, xmax, ymax] = init5(xmax, ymax, nrPeople, doorW,...
    distToCorner, pileNr, pileDist)
% This function creates a room with doors and piles
% The doors are specified in a CSV file called "doors.csv"
% The piles are specified in a CSV file called "piles.csv"
%
% INPUT:
% xmax, ymax    ... the dimensions of the room
% nrPeople      ... how many people it will have in the room
% doorW         ... has no further use anymore
% distToCorner  ... has no further use anymore
% pileNr        ... has no further use anymore
% pileDist      ... has no further use anymore
%
```

```
% OUTPUT:
% agentCoord ... The coordinates of the people.
% doorCoord  ... The coordinates of the doors (i.e. the middle of the door)
% wallCoord  ... The coordinates of the wall—"people". These are particles,
%                which don't move, thus represent wall—elements.
%                This matrix also contains the coordinates of the piles in
%                the first column
% pileCoord  ... The explicit coordinates of the piles (middle of the pile)
% prefDoor   ... This gives the currently prefered door of the people, it's
%                a vector with one entry for each person in agentCoord. The
%                index of the value corresponds to the person with the same
%                index in the matrix agentCoord
% doorFam    ... Stores information about every agent. Tells us which doors
%                an agent is familiar to.
% v          ... These should be the initial velocities of the people. It
%                should have the same size as agentCoord.
% rad        ... This gives how big persons are.
% doorW      ... For each Door, we need to know its size.
% xmax, ymax ... The dimensions of the room.

%% Parameters
% some parameters for the doors
Deps = 0;
fak = 2;

% the distance between two wall elements
Weps = 0.1;

% the size of the people
peopleRad = 0.75;

%% the room
% boarder walls
piles = [];

% get coordinates from CSV file
doors       = csvread('doors.csv');
%piles       = csvread('piles.csv');


% the full walls

% the construction of the north wall
northWall = 0:Weps:xmax;
northWall = northWall(:);
northWall = [northWall, ymax * ones(size(northWall))];

% the construction of the east wall
eastWall = 0:Weps:ymax;
eastWall = eastWall(:);
```

```
eastWall = [xmax * ones(size(eastWall)), eastWall];


% the construction of the south wall
southWall = 0:Weps:xmax;
southWall = southWall(:);
southWall = [southWall, 0 * ones(size(southWall)) ];

% the construction of the west wall
westWall = 0:Weps:ymax;
westWall = westWall(:);
westWall = [0 * ones(size(westWall)), westWall];


% place doors into wall

% hold door widths (capacities)
doorW = [];
% hold door coordinates
doorCoord = [];

% loop through all doors
for i=1:size(doors, 1)

    % position
    cDoorX  = doors(i, 1);
    cDoorY  = doors(i, 2);

    % capacity
    cDoorW  = doors(i, 3);

    if cDoorX == 0
        % west wall
        startY  = (cDoorY - (cDoorW / 2));
        endY    = (cDoorY + (cDoorW / 2));

        % cut the door out of the wall
        westWall = [westWall(1:(startY/Weps),:);...
        westWall((endY/Weps):size(westWall),:)];
    end

    if cDoorX == xmax
        % east wall
        startY  = (cDoorY - (cDoorW / 2));
        endY    = (cDoorY + (cDoorW / 2));

        % cut the door out of the wall
        eastWall = [eastWall(1:(startY/Weps),:);...
        eastWall((endY/Weps):size(eastWall),:)];
    end
```

```

    if cDoorY == 0
        % south wall
        startX  = (cDoorX - (cDoorW / 2));
        endX    = (cDoorX + (cDoorW / 2));

        % cut the door out of the wall
        southWall = [southWall(1:(startX/Weps),:);...
        southWall((endX/Weps):size(southWall),:)];
    end

    if cDoorY == ymax
        % north wall
        startX  = (cDoorX - (cDoorW / 2));
        endX    = (cDoorX + (cDoorW / 2));

        % cut the door out of the wall
        northWall = [northWall(1:(startX/Weps),:);...
        northWall((endX/Weps):size(northWall),:)];
    end

    % add door to the door coordinates container
    doorCoord(i,1) = cDoorX;
    doorCoord(i,2) = cDoorY;
    doorW(i) = cDoorW;

end

% init pile coordinates
pileCoord = [];

% loop through all piles
for i=1:size(piles, 1)

    % coordinates
    cPileX  = piles(i, 1);
    cPileY  = piles(i, 2);

    % pile width (default 1)
    cPileW  = 1;

    startX  = (cPileX - (cPileW / 2));
    endX    = (cPileX + (cPileW / 2));

    startY  = cPileY - (cPileW / 2);
    endY    = cPileY + (cPileW / 2);

    % x and y coordinates of the pile
    pileCoordX = [];
    pileCoordY = [];
```

```

    % cut pile into small piles (Weps)
    for k=startY:Weps:endY

        % store coordinates of current pile
        pileCoordX = [startX:Weps:endX];
        pileCoordX = pileCoordX(:);

        % calculate Y coordinates
        pileCoordY = k * ones(size(pileCoordX));

        % append to other piles
        pileCoord  = [pileCoord;[pileCoordX, pileCoordY]];
    end


end

% put the walls and piles together
wallCoord = [pileCoord;northWall; southWall; westWall; eastWall];

%% People
% place the people
%agentCoord = rand(nrPeople,2) .* repmat([xmax, ymax],nrPeople, 1);

% ensure no agent will be placed inside of a pile
agentCoord = [];
i = 1;

while i ≤ nrPeople

    % random coordinates
    agentCX = rand() * xmax;
    agentCY = rand() * ymax;

    % position is ok by default
    coordOk = true;

    % loop through walls and piles
    for k=1:size(wallCoord,1)

        if abs(wallCoord(k,1)−agentCX) ≤ peopleRad &&...
            abs(wallCoord(k,2)−agentCY) ≤ peopleRad
            % to close to a wall or pile, retry
            coordOk = false;
            break;
        end

    end

```

```
    if coordOk == false
        % to close, retry
        continue;
    else
        % coordinates ok, store
        agentCoord(i,1) = agentCX;
        agentCoord(i,2) = agentCY;
        i = i + 1;
    end
end

% set random door preferences
prefDoor = ceil(rand(nrPeople,1) .* size(doorCoord,1));


% setup random door acknowledges
doorFam = [];

for i=1:nrPeople
    for j=1:size(doorCoord,1)
        doorFam(i,j) = round(rand());
    end
end

% test if the people have chosen a valid door
for i = 1:nrPeople
    while (doorW(prefDoor(i)) == 0)
        prefDoor(i) = ceil(rand(1) * length(doorW));
    end
end


% set value and direction of the initial velocities
% of the people

rad = peopleRad * ones(nrPeople,1);
v = zeros(nrPeople, 2);

for i = 1:nrPeople
    dir = doorCoord(prefDoor(i),:) - agentCoord(i,:);
    v(i,:) = (dir./norm([xmax,ymax])) * norm([15,15]);
end
```

```
function [] = plotField(agentCoord, wallCoord, doorCoord, doorW, xmax, ymax)
% function that evaluates the field and gives then a
% contour plot and a 3d-plot of the field.
% the field is only calculated with the door which is the
% first one in the doorCoord input.
```

```
%
% INPUT:
% agentCoord    ... the coordinates of the agents
% wallCoord     ... the coordinates of the wall-agents
% doorCoord     ... the coordinates of the doors-middle
% doorW         ... the width of the doors
% xmax, ymax    ... the size of room


% the number of points to be evaluated per dimension.
nrEvals = 200;

% some parameters
wallR = 1.5;
agentR = 0.75;

% initialization
sol = zeros(nrEvals,nrEvals);
evalx = linspace(0,xmax,nrEvals);
evaly = linspace(0,ymax,nrEvals);

% parellelized loop for the evaluation
% if you want multiple processes running
% you need to write the following into the
% command window: matlabpool open
parfor i = 1:length(evalx);
    i %#ok<PFPRT>
    for j = 1:length(evaly);
        tsol = sol(i,:);


        %% potential we got from the agents
        for k = 1:size(agentCoord,1)
            r = norm([evalx(i), evaly(j)] - agentCoord(k,:));
            if (r ≤ agentR)
                tsol(j) = tsol(j) + 10^1.2 * 1/r;
            end
        end

        %% potential we get from the walls
        for k = 1:size(wallCoord,1)
            r = norm([evalx(i), evaly(j)] -  wallCoord(k,:));
            if (r < wallR)
                tsol(j) = tsol(j) + 1 * 1/r;
            end
        end

        %% potential we get from the Door 1
        r = norm([evalx(i), evaly(j)] - doorCoord(1,:));
        tsol(j) = tsol(j) + 10 * (r+4)^2;
```

```

        % since the values can go to infinity
        % this corrects those, that we still can
        % see something in the plot
        tsol(j) = min(tsol(j), 2500);
        sol(i,:) = tsol;
    end
end

% plot the 3d plot
figure(99);
[x,y] = meshgrid(evalx, evaly);
daspect([1,1,1]);
surfc(x,y,sol);

% plot the contour plot
figure(98);
daspect([1,1,1000]);
contourf(evalx,evaly, sol);
```

```
function[] = plotStats(logfile, plottitle)

% plots statistics for result CSV file logfile
% input:
%   logfile: path to csv logfile
%   plottitle: title for plot (ex. with piles / without piles)

% output:
%   nothing — draws a plot!

% get raw data
raw_data        = csvread(logfile);

% containers
agent_count     = [];
door_changes    = [];

evac_times      = [];

cases           = [];

case_count      = 0;

% colors for plot
colors = ['m', 'c', 'y', 'r', 'g', 'b'];

run_rows        = [];
run_counts      = [];
```

```

c_rows          = 0;

% collecting data
for i=1:length(raw_data)

    % -100 indicates a new case
    if raw_data(i,1) == -100

        % output
        disp(strcat(num2str(raw_data(i,1)), ' - ', num2str(raw_data(i,2))));

        % increase case
        case_count = case_count+1;

        % store count of people
        cases(case_count) = raw_data(i,2);

        % reset values
        run_counts(case_count) = 0;
        run_rows(case_count) = 0;
        c_rows               = 0;

        agent_count(1, case_count) = 0;
        door_changes(1, case_count) = 0;

        continue;

    end

    % -200 indicates a run within a case
    if raw_data(i,1) == -200
        % output
        disp(strcat('--> ',  num2str(raw_data(i,1)), ' - ', num2str(raw_data(i,2))));

        % increase run count
        run_counts(case_count) = run_counts(case_count) + 1;
        % reset rows
        c_rows = 0;

        continue;
    end

    % this is a data set

    % increase rows for this run
    run_rows(case_count) = run_rows(case_count) + 1;
    c_rows = c_rows + 1;

    % reserve space for stats
```

```
    if size(agent_count, 1) < c_rows
        agent_count(c_rows, case_count) = 0;
    end

    % append agent count
    agent_count(c_rows, case_count) = ...
        agent_count(c_rows, case_count) + raw_data(i,1);

    % reserve space for stats
    if size(door_changes, 1) < c_rows
        door_changes(c_rows, case_count) = 0;
    end

    % append door changes
    door_changes(c_rows, case_count) = ...
        door_changes(c_rows,case_count) + raw_data(i,2);

end


% analyze data (calculating averages)
for i=1:case_count
    % loop through all cases

    % average timesteps
    evac_times(i) = 0;
    evac_times(i) = round(run_rows(1,i) / run_counts(1,i));


    % calculate average agent count
    for j=1:size(agent_count, 1)
        agent_count(j,i)  = agent_count(j,i) / run_counts(1,i);

        if j > evac_times(i)
            agent_count(j,i) = 0;
        end

    end

    % calculate average door changes
    for k=1:size(door_changes,1)
        door_changes(k,i) = door_changes(k,i) / run_counts(1,i);

        if k > evac_times(i)
            door_changes(k,i) = 0;
        end
    end

end

```

```
% setup plots

% first plot (agent count)
figure(98);
set(gca, 'XTick', 0:100:900);
set(gca, 'YTick', 0:100:max(cases));

axis([0 900 0 500]);

title(strcat({'Agents '},plottitle));

xlabel('Time Steps');
ylabel('Agent Count');

% second plot (decision count)
figure(99);

set(gca, 'XTick', 0:100:900);
set(gca, 'YTick', 0:10:300);

axis([0 900 0 100]);

title(strcat({'Decisions '}, plottitle));

xlabel('Time Steps');
ylabel('Decisions');

legend1 = cell(1, case_count);
legend2 = cell(1, case_count);


% loop through all cases an generate plot using average values
for i=1:case_count

    disp(strcat('Evac Time of Case ', num2str(i), ': ', num2str(evac_times(i))));

    % create legend
    legend1{i} = sprintf('%d Agents\nAVG: %d Time Steps', cases(i), evac_times(i));
    legend2{i} = sprintf('%d Agents\nMax: %.2f\nAvg: %.2f', cases(i), ...
        max(door_changes(:,i)), mean(door_changes(1:evac_times(i),i)));



    figure(98);
    hold on;
    plot(agent_count(:,i), colors(i));


    figure(99);
```

```
    hold on;
    plot(door_changes(:,i), colors(i));


end

% set legend

figure(98);
legend(legend1);
figure(99);
legend(legend2);


end
```