Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Project Report

Pedestrian Dynamics

Robert Gantner & Patrick Wyss

Zurich
May 2010

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Gruppenarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellenhilfsmittel verwendt 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 Prüfungen ausgefertigt wurde.

Patrick Wyss   Robert Gantner

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.

Patrick Wyss   Robert Gantner

Contents

1 Individual contributions

The entire project was concepted, programmed and debugged by both authors simultaneously. Therefore, it is not possible to seperate the work into individual contributions.

2 Introduction and Motivations

The general security awareness of our society is continuously increasing, especially with respect to creating emergency exit plans. The design and layout of a room or building contributes directly to the efficiency of such plans. Because it is not possible to conduct empirical experiments in real-life panic situations, one must rely on simulation models which accurately describe these situations.
In our project, we were interested in examining the differences between two building configurations. We wanted to find out if a simple model could yield basic results describing the effect of such layouts.

3 Description of the Model

In this project, we first recreated an agent-based model found in [3] and described in detail in [1]. Here we will give a basic overview of this model, along with our additions to it.

3.1 Room Representation and Floor Field

The model is based on a square matrix representing the room in which the agents "live". Depending on the value of each field one can designate it as a wall, door, agent, or a free field. Once the basic structure of this matrix is defined (meaning the walls and doors), a so-called static force field can be computed. This force field can be interpreted as a potential depending on the distance to the exits (see figure 1).

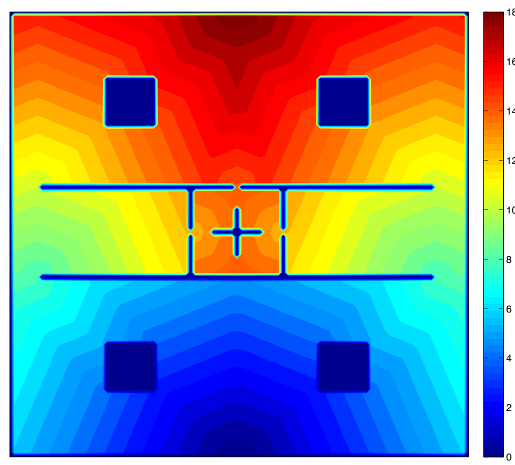

Fig.1: Static floor field for the ground level.

Additionally, a dynamic floor field is used during the simulation to represent the naïve following of other pedestrians. Like the name says, it is updated during the simulation. This is done by simply increasing the entries in the dynamic field corresponding to each occupied room entry. [1]
In order to avoid a situation in which the dynamic floor field dominates the transition probability (and thus creating an

oscillatory situation), we decided to normalize the dynamic floor field after each timestep.
In this paper we regard agents as obstacles – this means that if an agent occupies a cell, no other agent can move there in the current timestep. The reason for this is that if the agent currently on the destination cell does not move or is moved back by the conflict resolution algorithm, another conflict unnecessarily occurs. (See section 3.3)

3.2 Timestep

In each timestep, we update the room matrix by moving around the pedestrians with certain probabilities. These probabilities are calculated according to the two floor fields. The following formula is used for this:

$$P_{ij} = N\xi_{ij}\exp(k_D D_{ij} - k_S S_{ij})$$

where P is a 3×3 matrix containing the probabilities of moving in each direction as well as staying in the current cell, $N^{-1} = \sum_{i,j} P_{ij}$ is a normalization constant, $\xi_{ij} = 1$ if the cell in direction ij is free and 0 otherwise, k_S is the coefficient of the static floor field, k_D is the coefficient of the dynamic floor field, and S and D are the respective floor fields. [3]

3.3 Conflicts

If more than one agent tries to move to the same cell, a conflict is encountered. We resolve these conflicts according to the theory in [3]. With a probability of μ, every agent involved in the conflict is moved back to his original cell. With probability $1 - \mu$, one randomly chosen agent is allowed to stay.

3.4 Update Procedure

1. **Calculate Probabilities**
   The probabilities are computed for each agent according to the formula explained above.

2. **Calculate New Positions**
   According to these probabilities, each agent moves to a new cell. The origin and destination cells are stored in order to be able to detect conflicts.

3. **Detect and Resolve Conflicts**
   Each new position is compared to the other destination positions in order to determine new conflicts. After this is done for each agent, the conflicts are resolved as described above.

4. **Update Room**
   Now that there are no conflicts, the room can be updated. If an agent moves to a cell marked as a door, it is removed. If the agent is not on the first floor, it is moved to the same position on the next lowest floor. In the case that this position is already occupied, it is moved back to its original position. [1]

4 Implementation

4.1 Overview

To better explain our implementation, we first provide an overview of each function. Then, an in-depth explanation of the simulation is provided.

- **simulation.m**
  This file contains the main loops. It calls the other functions and manages the dynamic floor field.

- **createRoom.m**
  This file creates the initial room matrix and populates it with agents. The following values are used: 1: free floor, 2: wall, 3: agent, 4: door. See also [2].

- **staticFF.m**
  The staticFF function calculates the static floor field for a given room matrix using the Dijkstra shortest-path algorithm.

- **agentloop.m**
  This function implements the agent loop. It is responsible for computing probabilities, storing the movement of agents and detecting conflicts.

- **updateroom.m**
  In this function, all conflicts are resolved. Subsequently, the agents are moved to their new positions. Outgoing agents are removed.

- **drawRoom.m**
  This function is responsible for drawing the room.

- **drawFloor.m**
  This function draws multiple floors.

4.2 Initialization

First, a number of matrices are initialized. Since multiple floors must be created, the createRoom function is called for each floor. This is accomplished by storing the floors in a cell array named *room*. The i-th floor would then be $room\{i\}$.
For each floor, the static floor field must be computed. This is accomplished by storing the static floor field in a cell array named $S\{i\}$. Since the parameter that is really used is $S \cdot k_S$, which does not change during the simulation, this multiplication can be computed once at the beginning. The result is stored in the cell array $SkS\{i\}$ which is used for further computation.
Since we also use a dynamic floor field, it must be initialized at the beginning. We name it $D\{i\}$, also indexed according to the room it represents. At the beginning of the simulation it contains matrices composed only of zeros.

4.3 Agent Loop

The agent loop is the first procedure executed within each timestep. Its arguments are a room, the static floor field multiplied with the parameter k_S, the dynamic floor field, and the parameter k_D.
The agent loop, as the name implies, loops over the number of agents. For each agent, it computes the probabilities of moving to a different cell. This is de-

scribed in 3.2.
After the probabilities are calculated, a destination cell can be selected. The selected destination, along with the origin (current position of the agent), is stored in two vectors, *inew* and *jnew*. The format for the k-th agent is: $inew(k, :) = [destination, origin]$. The reason both origin and destination are stored is to allow the conflict resolution procedure to move an agent back to his original position if he is not chosen to proceed.
Now that the new positions are known, conflicts can be detected. Once a collision is detected, it is stored in the $conflict$ vector.
This concludes the agentloop procedure. The vectors *inew*, *jnew* and *conflict* are returned to the main program for further processing.

4.4 Updating a Room

The first operation in *updateroom* is the resolution of conflicts. This is done with a probability of $1 - \mu$, where μ is the so-called friction parameter. The higher it is, the lower the probability of resolving a conflict becomes.
If a conflict shall be resolved, one agent is chosen at random to move to the destination. All others are moved back to their original position, which is why the old positions are also stored in *inew* and *jnew*.
Next, all outgoing agents are detected. This is done by comparing each door's coordinates to the agents' positions. The outgoing agents are stored in a vector named *outgoing* and returned, in order to move agents around between floors.

4.5 Move Agents between Floors

In our multi-floor model, the agents are moved between floors after one timestep over all the floors is finished. The outgoing agents from each floor are placed in the floor below, provided the corresponding cell is free. If this is not the case, the agent is moved back up to his original position. The agents exiting from the lowest floor are just removed since they have completely escaped the building.

4.6 Finishing the Time Loop

At the end of the time loop, the current number of agents per floor is stored for further analysis. The floors are then drawn and if everyone has left the building the program is stopped.

5 Simulation Results and Discussion

In our tests we analyzed various scenarios using different room geometries. First, everything was simulated with empty rooms. Then, more realistic situations were modeled with a room configuration resembling that of an office building. The ground floor (see figure 2) contains an exit of size 3; the upper floors have offices and corridors. Two possible stairway configurations were analyzed: stairways in the corners and in the center (as in figure 3). The total number of stairways in each case are the same.

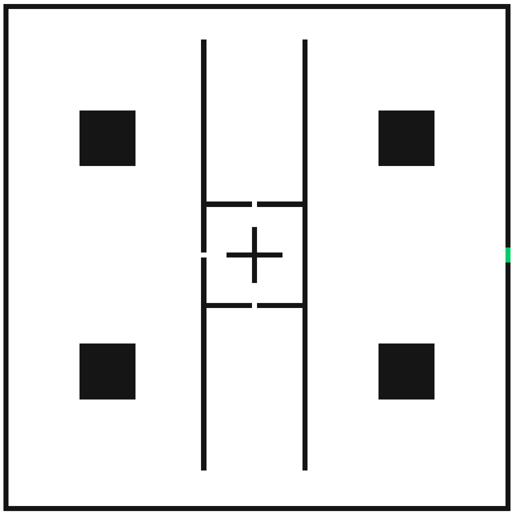

Fig.2: Ground Floor

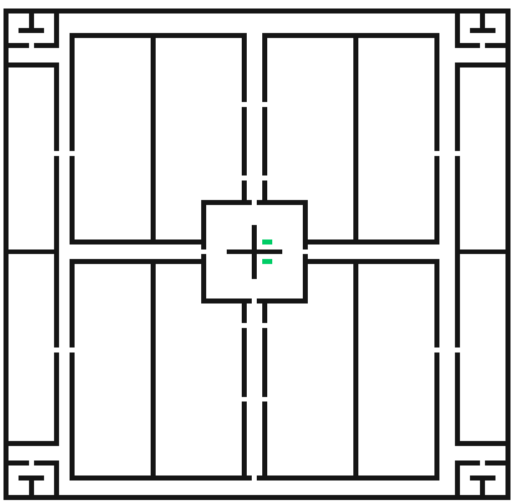

Fig.3: Higher Floors (Stairway in Center)

If nothing is stated, the following parameters were used: gridsize: 100, numagents: 90 (per floor), $k_S = 1.5$, $k_D = 3$, $\mu = 0.25$.

5.1 Evacuation Time vs. Number of Agents

In an empty room, the evacuation time depends linearly on the number of agents once a certain minimum density has been reached.
If the density is less than this minimum, the total evacuation time is more or less constant because there are barely any conflicts. The only factor contributing to the total evacuation time is the time the agent with the longest trajectory needs to reach the exit. The expectation value of this trajectory can be regarded as constant because the probability of having at least one agent in a certain area barely changes and approaches 1.
In our simulation, this minimum density was reached at around 180 agents per 10000 grid cells (see figure 4).

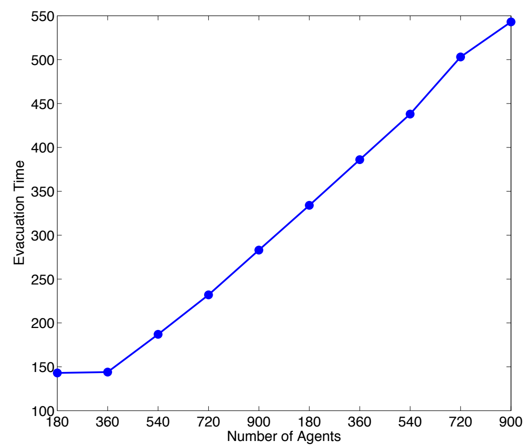

Fig.4: Evacuation Time vs. Number of Agents (One Floor)

5.2 Evacuation Time vs. Number of Floors

The total evacuation time in the office building model was found to depend linearly on the number of floors for the tested geometries.

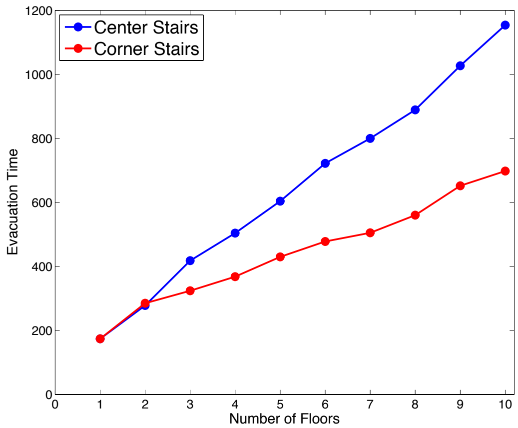

Fig.5: Evacuation Time vs. Number of Floors

As is evident from figure 5, our office model shows that placing stairs in the corners results in shorter evacuation times (on average) than placing them in the center. This is due to the higher number of conflicts arising when stairways are beside each other.

5.3 Evacuation Time vs. Friction Parameter μ

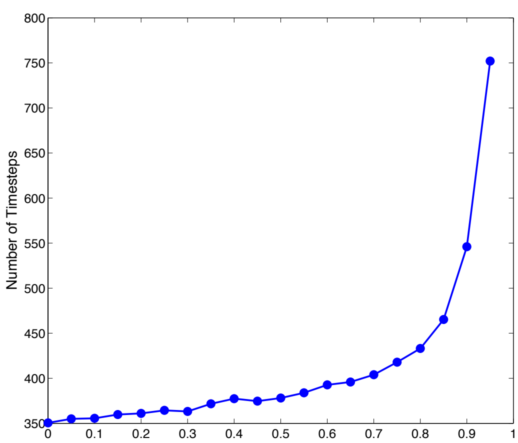

Fig.6: Time vs. μ

The total evacuation time was measured for different μ values. Since a higher μ value leads to less conflict resolution, we expected that the evacuation time increases with increasing μ. As can be seen in figure 6 this was confirmed by our simulations.

5.4 Evacuation Time vs. k_S and k_D

The dependence on k_S and k_D was determined by simulating both versions of our office building model (stairs in center, stairs in corners) with four floors for different (k_S, k_D) pairs. The simulation was repeated 10 times and the resulting times averaged to gain a more balanced measurement. This task required quite some time, in total approximately 6.25 days of serial computation. Luckily, we had access to a Brutus account capable of running the matlab code, reducing the time to about 5 hours. Further optimizations were achieved by parallelizing over the k_D loop and loading a pre-computed static floor field instead of recomputing the same one every time.
This parallelization was achieved by calling a script that runs simulations for different k_S values (10 times each) with different parameters for k_D on distributed nodes. The details are, however, beyond the scope of this project.
The resulting data summarized in figures 7-9. (Note that each simulation was allowed a maximum of 2000 timesteps.)

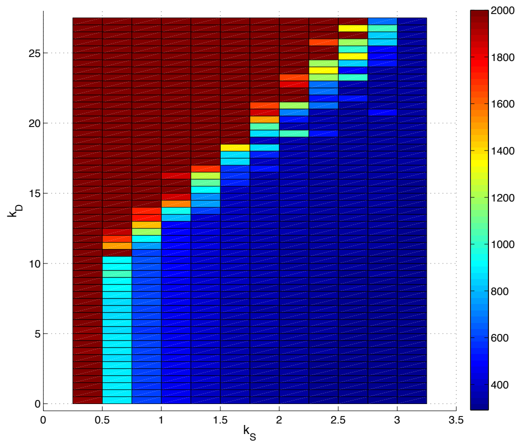

Fig.7: Number of timesteps for different (k_S, k_D) pairs. (Stairs in corners)

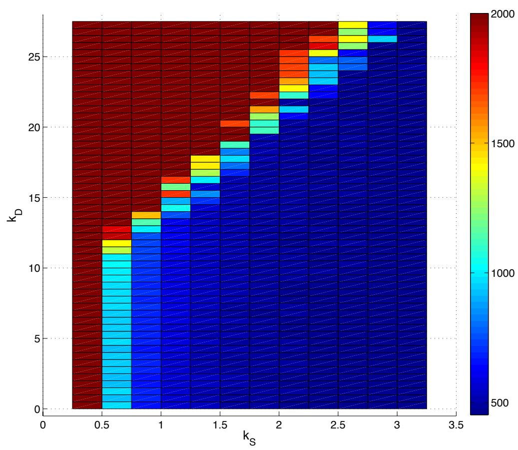

Fig.8: Number of timesteps for different (k_S, k_D) pairs. (Central stairs)

As can be seen, the number of timesteps for a complete evacuation of the building changes extremely quickly depending on the parameters k_D and k_S. The value of k_D either causes the dynamic floor field to not influence the probability at all, or it is so large that the probability is essentially only influenced by the dynamic floor field, removing the effect of the static floor field on the probabilities. The clumping of the agents caused by large k_D values also results in more conflicts, causing a longer evacuation time.

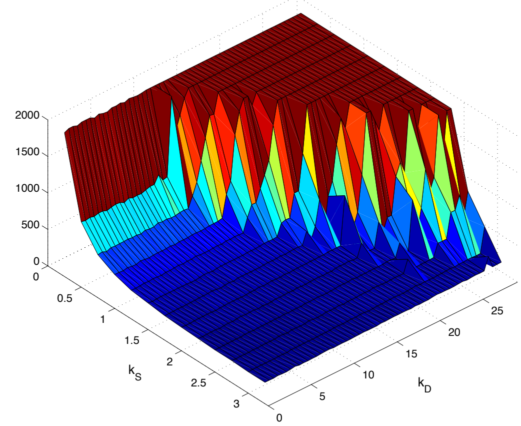

Fig.9: Number of timesteps for different (k_S, k_D) pairs. (Stairs in corners)

As can be seen in figure 10, once a certain k_S value has been reached, the evacuation time can no longer be shortened by increasing k_S. This is of course because if the potential is very large, every agent will move in the direction of the shortest path and hardly take any detours. Increasing the potential even more has no additional effect on the probabilities. This can be observed for both corner and central stairs.

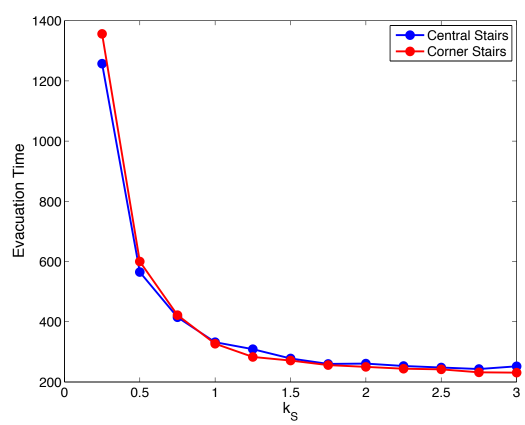

Fig.10: Time vs. k_S (4 Floors)

5.5 Heterogeneous Agent Distributions

After analyzing the effects of the above mentioned parameters, we changed the distribution of the agents in the building. Each odd floor has 120 agents, each even floor 60. The number of agents per floor was plotted against the time (see figures 11 and 12). These results were obtained by simulating a total of 50 times and averaging the results.

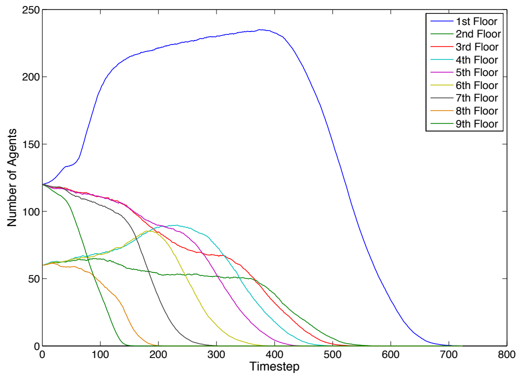

Fig.11: Number of Agents vs. Timesteps (Stairs in corners)

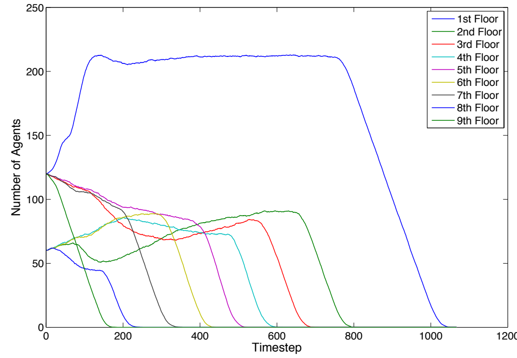

Fig.12: Number of Agents vs. Timesteps (Central Stairs)

As can be clearly seen, the two floor geometries yield very different results. In figure 11, the configuration with stairs in the corners has conjestion on the fourth floor. With central stairs, this conjestion is mainly present in floors two and three (see figure 12).

In figure 13, the results can be seen for a homogeneous agent distribution.

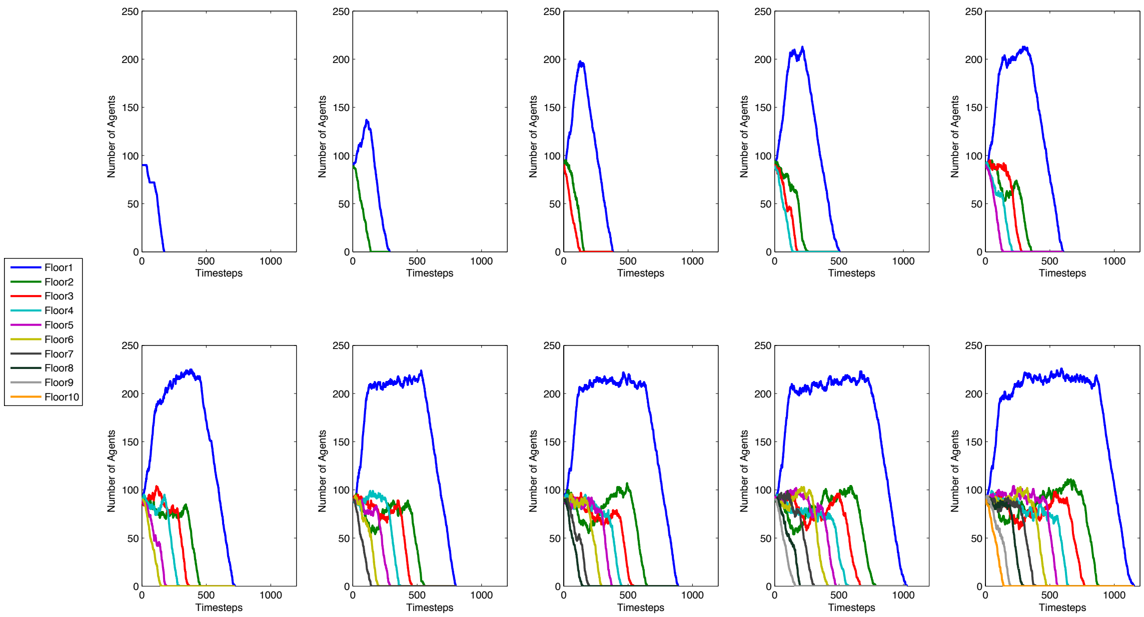

Fig.13: Number of agents per floor. From upper left to lower right: increasing number of floors.

6 Summary and Outlook

Our implementation of the floor field model provides a basis for simulating multiple floors concurrently. The office building model we used to test different staircase configurations allows for a coarse analysis of the evacuation time. Of course, the results presented herein have not been verified by empirical observations and should be understood as purely theoretical conclusions; any direct application to real-world scenarios would require an empirically confirmed model.

7 References

[1] C. Burstedde, K. Klauck, A. Schadschneider, and J. Zittartz. Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295(3-4):507 – 525, 2001.

[2] Lea Müller and David Hasenfratz. Pedestrian dynamics. *Lecture with Computer Exercises: Modelling and Simulating Social Systems with MATLAB*, page 48, May 2009.

[3] Daichi Yanagisawa, Ayako Kimura, Akiyasu Tomoeda, Ryosuke Nishi, Yushi Suma, Kazumichi Ohtsuka, and Katsuhiro Nishinari. Introduction of frictional and turning function for pedestrian outflow with an obstacle. *Phys. Rev. E*, 80(3):036110, Sep 2009.

A Code Listing

A.1 simulation.m

```
% simulation.m
% input:
% gridSize: dimension of NxN room
% numAgent: number of agents
% kS: static floor field coupling parameter
% kD: dynamic floor field coupling parameter
% mu: probability that conflict is not resolved
% tmax: number of timesteps
% p: time of pause between timesteps
% numfloors: number of floors to simulate, floor 1 is lowest floor
% stairposition: 0 -> corner stairs, 1 -> central stairs
%
% output:
% t: time
% n: number of agents in the room
function [tt,nn] = simulation(gridSize,numAgent,kS,kD,mu,tmax,p,...
                              numfloors,stairposition)

close all

% define default values - use if function called with too few arguments
if nargin < 1, gridSize = 100; end
if nargin < 2, numAgent=gridSize; end
if nargin < 3, kS = 1.5; end
if nargin < 4, kD = 0.01; end
if nargin < 5, mu = 0.25; end
if nargin < 6, tmax = 100000; end
if nargin < 7, p = 0.01; end
if nargin < 8, numfloors = 3; end
if nargin < 9, stairposition = 0; end

% initialization
N = gridSize;
numfloors2 = numfloors;
nn = []; % vector of n's to return

% create the room
room{1} = createRoom(gridSize, numAgent);
for i=2:numfloors
    room{i} = createRoom2(gridSize, numAgent);
    if mod(i,2) == 0
        if stairposition == 0 % stairs in corners
            %add stairs for even floors
            room{i}(98,96) = 4; room{i}(3,96) = 4;
```

```
            room{i}(3,5) = 4;   room{i}(98,5) = 4;
        else %central stairs
            room{i}(48,52:53) = 4; room{i}(52,52:53) = 4;
        end
    else
        if stairposition == 0 % stairs in corners
            %add stairs for odd floors
            room{i}(98,94) = 4; room{i}(3,94) = 4;
            room{i}(3,7) = 4;   room{i}(98,7) = 4;
        else %central stairs
            room{i}(48,47:48) = 4; room{i}(52,47:48) = 4;
        end
    end
end

% loop over floors to calculate static field and initialize dynamic field
for i=1:numfloors
    % calculate static field
    S{i} = staticFF(room{i});
    SkS{i} = S{i}*kS;
    % initialize dynamic floor field
    D{i} = zeros(N);
end

% time loop
for t=1:tmax
    % loop over floors
    for f=1:numfloors2
        % agent loop (returns new room matrix
        %f
        [inew, jnew, conflict] = agentloop(room{f},SkS{f},D{f},kD);
        % update the room
        [room{f},outgoing{f}] = updateroom(room{f},conflict,inew,jnew,mu);
        % update dynamic field
        D{f} = updatedynamicff(D{f}, room{f}, outgoing{f});
    end

    % insert outgoing agents of higher floors into lower floors
    for i=2:numfloors2 % start at 2 because those exiting floor 1 are gone
        if ¬isempty(outgoing{i})
            for u=1:size(outgoing{i},1)
                if room{i−1}(outgoing{i}(u,1),outgoing{i} (u,2)) == 1
                    room{i−1}(outgoing{i}(u,1),outgoing{i} (u,2)) = 3;
                else
                    room{i}(outgoing{i}(u,3), outgoing{i}(u,4)) = 3;
                end
            end
        end
    end
    % number of agents
```

```
    n = [];
    for f=1:numfloors2
        agent = find(room{f} == 3);
        n = [n ; length(agent)];
    end
    n = [n ; zeros(numfloors-numfloors2,1)];
    % reduce number of floors that are looked at (speed)
    if n(numfloors2) == 0
        numfloors2 = numfloors2 - 1;
    end
    % remember how many total agents per timestep
    nn = [nn,n];
    % draw
    drawFloor(room, numfloors);
    pause(p);
    % stop if everyone is evacuated
    if sum(n) == 0
        tt = 1:t;
        return
    end
end
tt = 1:t;
end


function [ D ] = updatedynamicff( D , room , outgoing)
% add 1 to fields where a person is
i = find(room==3);
for j=1:length(i)
    k = i(j); % i is the room vector
    D(k) = D(k)+1;
end
% add 1 to fields where a person exited
for j=1:size(outgoing,1)
    D(outgoing(j,1),outgoing(j,2)) = D(outgoing(j,1),outgoing(j,2)) + 1;
end
%D = D/norm(D);
end
```

A.2 createRoom.m

```
% createRoom.m
% This function creates a quadratic room of size gridSize.
% numAgent defines the number of agents placed in the room.
% i: floor
% The following integer coding is choosen for the room matrix:
% 1 free floor  2 wall
```

```
% 3 agent       4 open door
function [room, posAgent] = createRoom(gridSize, numAgent, i)

% create room matrix
room = ones(gridSize);

% create the walls
room(:,1) = 2; room(:,gridSize) = 2;
room(1,:) = 2; room(gridSize,:) = 2;

% create obstacles
% --------------------------------------
room(22:32,16:26) = 2; room(68:78,16:26) = 2;
room(22:32,75:85) = 2; room(68:78,75:85) = 2;
room(8:92,40) = 2; room(8:92,60) = 2;

%stair room in the middle
room(40,40:60) = 2; room(60,40:60) = 2;
room(40:60,40) = 2; room(40:60,60) = 2;
room(45:55,50) = 2; room(50,45:55) = 2;
room(40,50) = 1; room(60,50) = 1;
room(50,40) = 1;

% add door
room(round(gridSize/2)-1, gridSize) = 4;
room(round(gridSize/2), gridSize) = 4;
room(round(gridSize/2)+1, gridSize) = 4;

% Add the agents in the middle (range 1/3 to 2/3 of the gridsize).
n=0;
while n≠numAgent %while used because of retrying in case of occupied field
    % Calculate X-position.
    xPos = round(rand*(gridSize/3)+gridSize/3);
    % Calculate Y-position.
    yPos = round(rand*(gridSize/3)+gridSize/3);
    % Take position if it is free.
    if room(xPos, yPos) == 1
        n=n+1;
        room(xPos, yPos) = 3;
        % Save agent's position.
        posAgent(n,1) = xPos;
        posAgent(n,2) = yPos;

    end
end
end
```

A.3 createRoom2.m

```
% createRoom2.m
% This function creates a quadratic room of size gridSize.
% numAgent defines the number of agents placed in the room.
% i: floor
% The following integer coding is choosen for the room matrix:
% 1 free floor  2 wall
% 3 agent       4 open door
function [room, posAgent] = createRoom2(gridSize, numAgent)

% create room matrix
room = ones(gridSize);

% create the walls
room(:,1) = 2; room(:,gridSize) = 2;
room(1,:) = 2; room(gridSize,:) = 2;

% create obstacles/room design, optimized for gridSize 100
% ---------------------------------------
room(:,11) = 2; room(:,14) = 2;
room(:,30) = 2; room(:,70) = 2;
room(:,86) = 2; room(:,90) = 2;

room(9:11,11) = 1; room(90:92,11) = 1;
room(9:11,90) = 1; room(90:92,90) = 1;

room(8,1:10) = 2;
room(12,1:10) = 2; room(50,1:10) = 2;
room(89,1:10) = 2; room(93,1:10) = 2;

room(8,90:100) = 2;
room(12,90:100) = 2; room(50,90:100) = 2;
room(89,90:100) = 2; room(93,90:100) = 2;

room(2:5,14) = 1; room(2:5,30) = 1;
room(2:5,70) = 1; room(2:5,86) = 1;

room(96:99,14) = 1; room(96:99,30) = 1;
room(96:99,70) = 1; room(96:99,86) = 1;

room(6,14:86) = 2; room(48,14:86) = 2;
room(52,14:86) = 2; room(96,14:86) = 2;

%stair room in the middle
room(41:59,41:59) = 1;
room(40,40:60) = 2; room(60,40:60) = 2;
room(40:60,40) = 2; room(40:60,60) = 2;
```

```
room(45:55,50) = 2; room(50,45:55) = 2;

room(40,50) = 1; room(60,50) = 1;
room(50,40) = 1; room(50,60) = 1;

%corridors
room(49:51,14) = 1; room(49:51,30) = 1;
room(49:51,70) = 1; room(49:51,86) = 1;

room(6,49:51) = 1;  room(96,49:51) = 1;
room(6:40,48) = 2;  room(6:40,52) = 2;
room(60:95,48) = 2; room(60:95,52) = 2;

room(20,48) = 1; room(35,48) = 1;
room(65,48) = 1; room(80,48) = 1;
room(20,52) = 1; room(35,52) = 1;
room(65,52) = 1; room(80,52) = 1;

%doors for these rooms
room(8,6) = 1;  room(8,95) = 1;
room(93,6) = 1; room(93,95) = 1;

room(30,11) = 1; room(70,11) = 1;
room(70,90) = 1; room(30,90) = 1;
room(30,14) = 1; room(70,14) = 1;
room(30,86) = 1; room(70,86) = 1;

room(6,50) = 1; room(96,50) = 1;

%exits/stairs design
%stairs are added in simulation
room(2:5,6) = 2;
room(2:5,95) = 2;
room(96:99,6) = 2;
room(96:99,95) = 2;

room(5,4:8) = 2;
room(5,93:97) = 2;
room(96,4:8) = 2;
room(96,93:97) = 2;

% Add the agents in the middle (range 1/3 to 2/3 of the gridsize).
n=0;
while n≠numAgent %while used because of retrying in case of occupied field
    % Calculate X-position.
    xPos = ceil(rand*100);
    % Calculate Y-position.
    yPos = ceil(rand*100);

```

```
    % Take position if it is free.
    if room(xPos, yPos) == 1
        n=n+1;
        room(xPos, yPos) = 3;
        % Save agent's position.
        posAgent(n,1) = xPos;
        posAgent(n,2) = yPos;

    end
end
end
```

A.4 staticFF.m

```
% staticFF.m
function [ S ] = staticFF( room )
% calculate static floor field
% 2010 Robert Gantner Patrick Wyss

%start at exit, calculate distance to all neighbors (max. 8).
% room size
N = size(room,1);
% find exits
e = find(room == 4);
assert(¬isempty(e),'no doors found');
% coordinates of exits
i = mod(e,N);
ind = find(i == 0);
i(ind) = N;
j = (e−i)./N+1;

% initialize S
S = inf(N,N,length(e));
% loop over all exits
for q = 1:length(e)
    % initialize cut
    cut(1,1) = i(q); % x−component of current exit
    cut(1,2) = j(q); % y−component of current exit
    % initialize oldcut
    oldcut = [];
    % initialize S
    S(i(q),j(q),q) = 0;
    % as long as there are still fields to look at
    while ¬isempty(cut)
        [cut, S(:,:,q)] = updatecut(room,cut,oldcut,N,S(:,:,q));
    end
end
```

```
% take minimal value over third dimension (exits)
S = min(S,[],3);
end

function [newcut, S] = updatecut(room,cut,oldcut,N,S)
newcut = [];
for i=1:size(cut,1) % iterate over cut
    n = neighbors(cut(i,:),N,room);
    for j=1:size(n,1) % over all neighbors
        % if the current value stored at S(neighbor) is larger than the
        % distance to the cut value plus the distance to the neighbor, it
        % should be overwritten.
        if S(cut(i,1),cut(i,2))+n(j,3) < S(n(j,1),n(j,2))
            % save new minimal distance to field
            S(n(j,1),n(j,2)) = S(cut(i,1),cut(i,2))+n(j,3);
            % add neighbor to cut
            newcut = addtocut(newcut, oldcut, n(j,1), n(j,2));
        end
        % if this is not the case, the neighbor has already been reached—
        % no need to add to cut.
    end

end
end

function [neigh] = neighbors(x,N,room)
w = sqrt(2);
neigh = [[x 0]+[0 1 1];
[x 0]+[0 −1 1]; [x 0]+[1 0 1];
[x 0]+[−1 0 1]; [x 0]+[1 1 w];
[x 0]+[1 −1 w]; [x 0]+[−1 1 w];
[x 0]+[−1 −1 w] ];
% i component in limits
i = find(neigh(:,1) ≤ 0);
j = find(neigh(:,1) > N);
% j component in limits
k = find(neigh(:,2) ≤ 0);
l = find(neigh(:,2) > N);
rem = [i;j;k;l];
if ¬isempty(rem), neigh(rem,:) = []; end
% test if neighbor is an obstacle or another door
rem = [];
for i=1:length(neigh);
    if room(neigh(i,1),neigh(i,2)) == 2 || room(neigh(i,1),neigh(i,2)) == 4
        rem = [rem,i];
    end
end
neigh(rem,:) = [];
end

```

```
function newcut = addtocut(cut, oldcut, ipos, jpos)
if isempty(cut)
    newcut = [ipos, jpos];
else
    % find all matching first requirement
    ind = cut(:,1) == ipos;
    % find all matching second requirement while looking only at those
    % fulfilling the first requirement.
    res = cut(ind,2) == jpos;
    % same for oldcut (if not empty)
    res2 = 0;
    if ¬isempty(oldcut)
        ind2 = oldcut(:,1) == ipos;
        res2 = oldcut(ind2,2) == jpos;
    end
    if ¬any(res) && ¬any(res2)
        %not found in cut AND not found in oldcut −> add
        newcut = [cut ; ipos, jpos];
    else
        newcut = cut;
    end
end
end
```

A.5 agentloop.m

```
% agentloop.m
function [ inew,jnew,conflict ] = agentloop( room,SkS,D,kD )
% loops over all agents

% norm D.
if norm(D) ≠ 0, D = D/norm(D); end

N = size(room,1);
agent = find(room == 3);
n = length(agent);

i = mod(agent,N);
j = (agent−i)./N+1;
agent = [i j];

conflict = [];
inew=[]; jnew=[];

% neighbor matrix determining movement direction
nstep = [ kron([−1;0;1],ones(3,1)) , kron(ones(3,1),[−1;0;1]) ];
for a = 1:n % a... current agent
```

```
    % p(i,j) is probability for agent to go to cell i,j
    % (only neighboring cells)
    p = zeros(3);
    for i=1:3
        for j=1:3
            % start at left upper edge
            if room(agent(a,1)+i-2,agent(a,2)+j-2) == 2 || ...
                    room(agent(a,1)+i-2,agent(a,2)+j-2) == 3
                %continue
                p(j,i) = 0;
                % p = zeros at beginning, so this element is 0.
            else
                p(j,i) = exp(+kD*D(agent(a,1)+i-2,agent(a,2)+j-2)-...
                    SkS(agent(a,1)+i-2,agent(a,2)+j-2));
            end
        end
    end
    % if all neighbor cells are taken, agent has to stay
    if sum(sum(p)) == 0, p(2,2) = 1; end
    p = p./sum(sum(p));
    % make vector with partial sums of probabilities
    v(1) = p(1);
    for k=2:length(p)^2
        v(k) = v(k-1)+p(k);
    end
    % determine index of where to go
    l = find(rand<=v, 1 ); %,1 means first instance (minimal)
    newindi = agent(a,1)+nstep(l,1);
    newindj = agent(a,2)+nstep(l,2);
    % move a person
    %       new position old position
    inew(a,:) = [newindi agent(a,1)];
    jnew(a,:) = [newindj agent(a,2)];

    if exists(conflict,newindi,newindj) == 0
        % look through inew,jnew if they have the same destination.
        for t=1:size(inew,1)-1
            if inew(t,1) == newindi
                if jnew(t,1) == newindj
                    % now add to conflicts
                    %conflict = addifnexists(conflict, newindi, newindj);
                    conflict = [conflict; newindi, newindj];
                    break;
                end
            end
        end
    end
end

end
```

```

function found = exists(dest,i,j)
found = 0;
for t=1:size(dest,1)
    if dest(t,1) == i
        if dest(t,2) == j
            found = 1;
        end
    end
end
end
```

A.6 updateroom.m

```
% updateroom.m
function [ room , outgoing] = updateroom( room, conflict, inew, jnew, mu )
% resolve conflicts
% overview: conflict resolved with probability (1-mu)
% if conflict is resolved, chose an agent at random
% outgoing is vector containing agents leaving the room
outgoing = [];

% 1. resolve conflicts
for c = 1:size(conflict,1)
    cindex = [];
    % find indexes in inew vector of agents involved in conflict
    for t=1:size(inew,1)
        if inew(t,1) == conflict(c,1)
            if jnew(t,1) == conflict(c,2)
                cindex = [cindex , t];
            end
        end
    end
    if rand ≤ mu
        % everyone goes back to where they were. delete from inew,jnew
        inew(cindex,:) = [];
        jnew(cindex,:) = [];
    else
        % chose one at random to stay
        nconf = length(cindex); % number of agents in conflict
        nth = ceil(rand*nconf); % select the nth of the nconf
        cindex(nth) = []; % don't delete the one that stays
        inew(cindex,:) = [];
        jnew(cindex,:) = [];
    end
end

```

```
% 2. detect outgoing agents
% first find all doors:
e = find(room == 4);
N = length(room);
% coordinates of exits (vectors)
i = mod(e,N); ind = find(i == 0);
i(ind) = N;    j = (e−i)./N+1;

for q=1:length(e) % for all doors
    delete = [];
    for t=1:size(inew,1)
        if inew(t,1) == i(q) % test if someone wants to move to current door
            if jnew(t,1) == j(q)
                outgoing =[outgoing;inew(t,1) jnew(t,1) inew(t,2) jnew(t,2)];
                delete = [delete; t];
            end
        end
    end
    for tt=1:length(delete)
        del = delete(tt);
        % delete agent
        room(inew(del,2),jnew(del,2)) = 1;
    end
    % don't update agent
    inew(delete,:) = []; jnew(delete,:) = [];
end

% 3. update room matrix (conflicts have been resolved)
for i=1:size(inew,1)
    % remove from origin
    room(inew(i,2),jnew(i,2)) = 1;
    % insert at destination
    room(inew(i,1),jnew(i,1)) = 3;
end
end
```

A.7 drawFloor.m

```
% drawFloor.m
function drawFloor( room, numfloors )
c = sqrt(numfloors);
a = ceil(c);
if a*(a−1) ≥ numfloors, b = a−1;
else b = a;
end

for f=1:numfloors
```

```
    subplot(a,b,f);
    % define colors for each value in the matrix
    % floor color white (empty cells)
    myColorMap(1,:) = [1 1 1];
    % wall color grey
    myColorMap(2,:) = [21/255 21/255 21/255];
    % agent color blue (agent)
    myColorMap(3,:) = [54/255   100/255 139/255];
    % open door color green
    myColorMap(4,:) = [0/255 205/255 102/255];

    % draws the 2D matrix as image using defined color map
    colormap(myColorMap);
    imagesc(room{f});
    Title(strcat('Floor ',num2str(f)))
    axis off;
end
end
```