



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Project Report

<h2>Crowd Simulation</h2>

Samuel Oberholzer & Philipp Lütolf

Zürich
11.05.2014



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Crowd Simulation

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Oberholzer

Lütolf

First name(s):

Samuel

Philipp

With my signature I confirm that

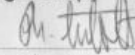
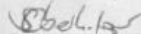
- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 13.05.2014

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Agreement for free-download

We hereby agree to make our source code of 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.

Samuel Oberholzer

Philipp Lütolf

Table of Content

1	Abstract	5
2	Individual contributions	5
3	Introduction and Motivations.....	6
4	Description of the Model	7
4.1	Social force model.....	7
4.2	Acceleration force	7
4.3	Pedestrian Interactions	8
4.4	Boundary Interactions	8
5	Implementation	10
5.1	Initialization and Simulation overview (the testModel.m file)	10
5.2	Initialization	10
5.2.1	Walls and Waypoints	11
5.2.2	Agents	11
5.2.3	The “cross”-situation	11
5.2.4	The “curve”-situation.....	12
5.3	Simulation	12
5.3.1	Setting the next destination.....	13
5.3.2	The acceleration force.....	13
5.3.3	The wall force	13
5.3.4	The pedestrian force.....	14
5.3.5	Position update and saving data.....	14
5.4	Plotting the data and making a video (simulate.m).....	14
5.5	Overview of the result (maxPeopleOnSquare.m)	14
5.6	Model constants.....	15
6	Simulation Results and Discussion.....	16
6.1	Simulation without obstacles	16
6.2	Simulation with obstacles.....	18
7	Summary and Outlook	19
8	References.....	20
9	Source code.....	20
9.1	Test model	20
9.2	Simulation	26
9.3	Max People per square	27

1 Abstract

The goal of this project is to model big crowds in different locations and identify the dangerous spots. Our model is programmed in Matlab, is agent-based and in continuous space. The physics are based on the ‘social force model’ from Helbing [1]. To identify the critical spots, the density of the people is computed and illustrated in the simulation. In this report two different locations are considered, a crossroad and a ninety-degree curve which are simulated with and without obstacles to compare the differences. The results are then discussed and compared to real life experiences.

2 Individual contributions

The work on this project was shared evenly between both authors because we always worked together.

3 Introduction and Motivations

Big events with a lot of people bear a great risk of mass panic. There are numerous events from the past where people got injured or died because of the forces acting on pedestrians in dense crowds. The most recent crowd disaster was the love parade in Duisburg 2010, where 21 people died and over 500 people were injured.

One critical factor of such tragedies is the location. Obstacles, corners, intersections and other bottlenecks where the crowd is compressed are the most critical spots. The main questions which we are going to answer are the following:

Where are the most critical spots for people in a crowd in a given environment?

What are the possibilities to reduce the risk to individual people by rearranging the environment?

To answer these questions we decided to use the “social force model” by D. Helbing [1]. This model is able to reproduce many real life situations and is explained in section 4. We have decided to implement this model in Matlab as an agent-based model in continuous space. To model different arrangements we chose two different situations, each one with and without obstacles. One situation is a crossroad and the other situation is a ninety-degree curve. Explanations concerning the implementation are given in section 5.

Our results are then discussed and compared to real life experiences (section 6).

4 Description of the Model

4.1 Social force model

The model is based on the ‘social force model’ developed by D. Helbing and P. Molnar [1] [2]. It describes the “social forces” acting on a pedestrian in a crowd. Each pedestrian α in a crowd can be represented with a point \vec{r}_α in space, the speed of the pedestrians can be described by the equation

$$\frac{d\vec{r}_\alpha(t)}{dt} = \vec{v}_\alpha(t)$$

and the acceleration by

$$\frac{d\vec{v}_\alpha}{dt} = \vec{f}_\alpha(t).$$

D. Helbing and P. Molnar added a fluctuation term which include random variations of the behaviour.

$$\frac{d\vec{v}_\alpha}{dt} = \vec{f}_\alpha(t) + \text{fluctuations}$$

One term of \vec{f}_α describes the driving force which accelerates the pedestrian towards the desired velocity $\vec{f}_\alpha^0(\vec{v}_\alpha)$, another term consists of the repulsion force from other pedestrians $\sum_{\beta(\neq\alpha)} \vec{f}_{\alpha\beta}(\vec{r}_\alpha, \vec{v}_\alpha, \vec{r}_\beta, \vec{v}_\beta)$ and obstacles $\vec{f}_{\alpha B}(\vec{r}_\alpha)$ and the last force outlines attractive effects $\sum_i \vec{f}_{\alpha i}(\vec{r}_\alpha, \vec{r}_i, t)$.

$$\vec{f}_\alpha(t) = \vec{f}_\alpha^0(\vec{v}_\alpha) + \vec{f}_{\alpha B}(\vec{r}_\alpha) + \sum_{\beta(\neq\alpha)} \vec{f}_{\alpha\beta}(\vec{r}_\alpha, \vec{v}_\alpha, \vec{r}_\beta, \vec{v}_\beta) + \sum_i \vec{f}_{\alpha i}(\vec{r}_\alpha, \vec{r}_i, t)$$

4.2 Acceleration force

The driving force $\vec{f}_\alpha^0(\vec{v}_\alpha)$ accelerates the pedestrian towards the destination \vec{p} which results in the desired direction

$$\vec{e}_\alpha(t) = \frac{\vec{p} - \vec{r}_\alpha}{\|\vec{p} - \vec{r}_\alpha\|}$$

Deviations of the actual velocity \vec{v}_α from the desired velocity $\vec{v}_\alpha^0 = v_\alpha^0 \vec{e}_\alpha$ occurring from obstacles or other pedestrians are corrected within the “relaxation time” τ_α .

$$\vec{f}_\alpha^0 = \frac{1}{\tau} (\vec{v}_\alpha^0(t) - \vec{v}_\alpha(t)) = \frac{1}{\tau_\alpha} (v_\alpha^0(t) \vec{e}_\alpha(t) - \vec{v}_\alpha(t))$$

The desired speed $v_\alpha^0(t)$ is defined as

$$v_\alpha^0(t) = (1 - n_\alpha(t)) v_\alpha^0(0) + n_\alpha(t) v_\alpha^{\max}$$

Where $v_\alpha^0(0)$ is the initial velocity and v_α^{\max} the maximum desired velocity.

The parameter

$$n_\alpha(t) = 1 - \frac{\overline{v}_\alpha(t)}{v_\alpha^0(t)}$$

characterizes the nervousness $n_\alpha(t)$ of the pedestrian to reach their destination and $\overline{v}_\alpha(t)$ describes the average speed of the pedestrian.

4.3 Pedestrian Interactions

The term $\overrightarrow{f_{\alpha\beta}}(\overrightarrow{r_\alpha}, \overrightarrow{v_\alpha}, \overrightarrow{r_\beta}, \overrightarrow{v_\beta})$ describes the repulsive force from another pedestrian β . This force is defined as

$$\overrightarrow{f_{\alpha\beta}}(t) = A_\alpha^1 \exp\left(\frac{(r_{\alpha\beta} - d_{\alpha\beta})}{B_\alpha^1}\right) \overrightarrow{n_{\alpha\beta}} F_{\alpha\beta} + A_\alpha^2 \exp\left(\frac{r_{\alpha\beta} - d_{\alpha\beta}}{B_\alpha^2}\right) \overrightarrow{n_{\alpha\beta}}$$

Where A_α denotes the respective interaction strength and B_α the range of the repulsive interaction. The parameter $r_{\alpha\beta}$ is the sum of the radii of both pedestrians, $d_{\alpha\beta}$ is the distance between the centres of mass and $\overrightarrow{n_{\alpha\beta}}$ is the normalised vector pointing from β to α .

$$\overrightarrow{n_{\alpha\beta}} = \frac{(\overrightarrow{x_\alpha}(t) - \overrightarrow{x_\beta}(t))}{d_{\alpha\beta}(t)}$$

The last parameter $F_{\alpha\beta}$ accounts for the directionally dependent behaviour of pedestrians. In the context of crowds the factor $F_{\alpha\beta}$ gives the pedestrians within sight greater influence than those out of sight.

$$F_{\alpha\beta} = \lambda_\alpha + \frac{(1 - \lambda_\alpha)(1 + \cos(\varphi_{\alpha\beta}))}{2}$$

The parameter λ_α characterizes the directionally dependent behaviour with

$$\cos(\varphi_{\alpha\beta}) = -\overrightarrow{n_{\alpha\beta}}(t) \overrightarrow{e_\alpha}(t) \quad \text{and} \quad \overrightarrow{e_\alpha}(t) = \frac{\overrightarrow{v_\alpha}(t)}{\|\overrightarrow{v_\alpha}(t)\|}$$

4.4 Boundary Interactions

The force acting on pedestrians from boundaries is almost identical to the repulsion force from other pedestrians. The only difference is that the directionally dependent behaviour can be neglected ($A_\alpha^1 = 0$).

$$\overrightarrow{f_{\alpha B}}(\overrightarrow{r_\alpha}) = A_{\alpha B} \exp\left(\frac{r_\alpha - d_{\alpha B}}{B_{\alpha B}}\right) \overrightarrow{n_{\alpha B}}$$

Where $\overrightarrow{n_{\alpha\beta}}$ is the normal vector pointing from the boundary to the pedestrian and $d_{\alpha B}$ the distance between the boundary and the pedestrian.

5 Implementation

The social force model is implemented in Matlab as an agent-based model in continuous space. The implementation consists of three parts. First, the testModel.m file where all the computation is done and the data is saved. Secondly, a simulate function, that plots the saved data and makes a video out of it. And third, the maxPeopleOnSquare function, which visualizes the simulation data for analyzing.

5.1 Initialization and Simulation overview (the testModel.m file)

The implementation of the core file (testModel.m) roughly follows the Pseudocode shown below:

—Initialization—

Set walls and waypoints

Set start positions of agents

—Simulation—

FOR each frame of the simulation

 FOR every pedestrian in the system do

 Set up next destination and calculate desired direction

 Calculate acceleration force and influence on velocity

 FOR each wall element do

 Check distance to element

 Calculate wall force of closest element and influence on velocity

 END

 FOR every other agent of the matrix do

 Calculate pedestrian force and influence on velocity

 END

 Update position according to calculated velocity

 save agent in a updated matrix

 END

 set matrix to updated matrix

 save data for plotting

END

The initialization and the simulation are separately clarified below.

5.2 Initialization

The implementation allows to specify and run the simulation on different maps, only depending on the initialization of the walls, waypoints and agents. Two situations were specified to investigate our research questions.

5.2.1 Walls and Waypoints

Walls are specified by a start- and endpoint. So every wall needs to store x and y position for two points, which means four values per wall are stored. Obstacles can be placed into the situation as well by placing them as walls.

Because it's unlikely that all agents head to a single point in space, a new concept of waypoints is introduced. Here, waypoints are more like „waylines“ and stored in the same format as walls. It's a line that sets the next destination of an agent. The shortest path to this line is used to calculate the desired direction of an agent. If an agent reaches the last waypoint, he respawns at his starting area.

5.2.2 Agents

There is one matrix containing all the agents of the simulation. Every agent consists of eight different values. These are the agent's position and its current velocity, its desired velocity which is Gaussian-distributed at $1.3 \pm 0.1 \text{ m/s}$ and its type. The type determines to which group of agents the agent belongs and therefore where its starting area and destination is. Further values are the waypoint the agent is aiming for and its average speed. The positions and waypoints for the initialization depend on the situation. In the project, the following two situations were specified:

agent
- x position
- y position
- x velocity
- y velocity
- desired velocity
- type
- current waypoint
- average speed

Figure 1: Agent vector

5.2.3 The “cross”-situation

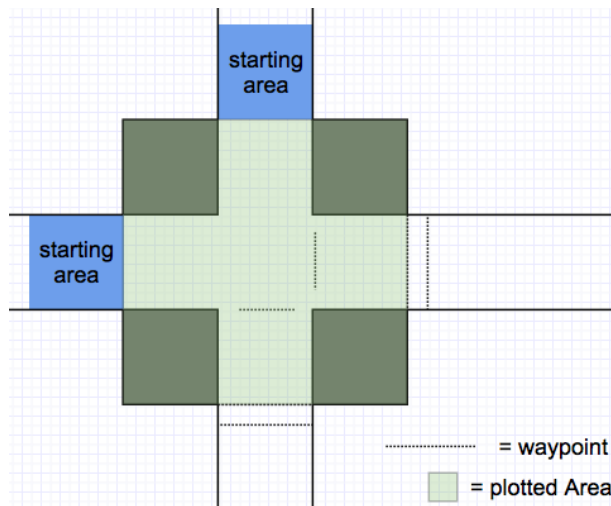


Figure 2: Sketch of the “cross” situation

The walls are defined as seen in the picture above. Agents are initialized randomly distributed in a 5x5 area on the left side and on the top (outside the plotted area). Their

goal is to reach the other side of the cross-way. To achieve this, the way points were set as seen in the picture.

The crowd flow in this situation is investigated with and without obstacles and the outcome is compared.

5.2.4 The “curve”-situation

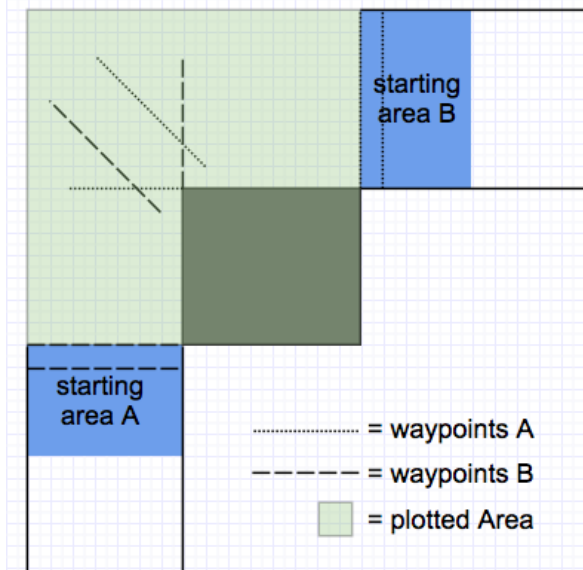


Figure 3: Sketch of the “curve” situation

In this situation, the goal of the agents is to follow the street and make it to the other end. One half starts at the bottom-left end and the other half at the top-right. Again, obstacles were added to compare the crowd flow with obstacles to the one without. The waypoints are set as seen in the picture to overcome the obstacles.

5.3 Simulation

In order to realistically simulate movement of agents in continuous space, they need to be updated a lot. Time between these updates was chosen at 0.05s, which corresponds to 20 frames per second. If the time between the updates is chosen too high, the steps of the agents are larger and important factors influencing their movement could be skipped. For example if an agent moves towards a wall and the update occurs shortly before the agent is pushed back by the wall force, the agent could be past the wall in the next update.

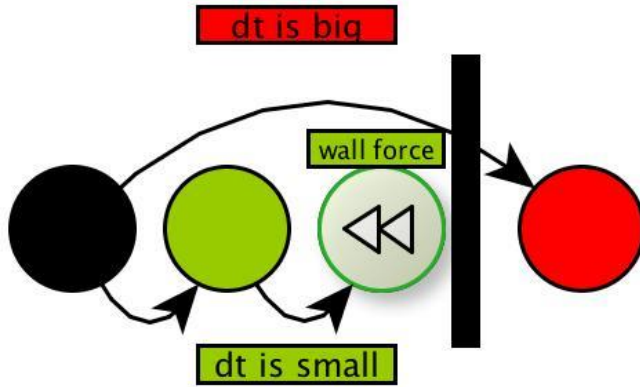


Figure 4: Time discretization bug if dt is too big

In every step, all the agents have to be updated. These updates depend on the current waypoint they are aiming for, the walls, and the positions of other agents. The following things are executed for each agent.

5.3.1 Setting the next destination

The distance from the agent to the next waypoint is computed with *vectorFromWall*. If the distance is small enough, the next waypoint is set as new destination.

If the agent passes the last waypoint it has reached its destination and is re-initialized at the starting area. Like this, the flow of incoming agents can be maintained without having to create new ones and therefore performance is tuned. The desired direction is calculated again using *vectorFromWall* so the acceleration force can be computed.

The function *vectorFromWall* takes a waypoint and the agent's position and gives back the normalized vector pointing to this waypoint and the distance to it.

5.3.2 The acceleration force

The acceleration force is computed using *accelerationF*. Then the force is added to the current velocity of the agent.

The function *accelerationF* takes the desired direction, current velocity, average speed and the desired velocity and returns the calculated acceleration force according to section 4.2.

5.3.3 The wall force

Next, the nearest wall is sought after using *vectorFromWall*, then the wall force is computed using *wallF*. The wall force is added to the velocity afterwards.

The function *wallF* takes the distance to a wall and the normalized vector pointing to it. According to section 4.4 the wall force is computed.

5.3.4 The pedestrian force

Every other pedestrian has an influence on the agent. Therefore, the pedestrian force resulting from each agent is computed using *pedestrianF* and added to the velocity of the agent. The loop over all agents uses the matrix from the last step, so updates on all agents are made simultaneously.

The function *pedestrianF* computes the pedestrian force using the positions of two agents as arguments according to section 4.3.

5.3.5 Position update and saving data

When all the forces contributing to the new velocity are computed, the position of the agent is updated. Additionally, the matrix containing information about how many people are in a square meter is updated.

After the agents have been updated, the whole matrix is updated simultaneously. The positions of the agents are stored separately so that the simulation run can be plotted later on.

5.4 Plotting the data and making a video (simulate.m)

function simulate(filename,mode,savevideo)

In this function, the saved data from the file ,filename' is visualized. The x positions of all the agents are plotted against their y position. Combined with the plot of the walls this results in the visual simulation of the situation. If ,mode' is equal to one, the whole plot is provided with a background picture. This background picture shows how many people are standing in each square meter. The whole plot is saved to a video file called ,savevideo'.

5.5 Overview of the result (maxPeopleOnSquare.m)

function maxPeopleOnSquare(inputfile)

This function creates a figure with two subplots to summarize the data of the plot. Input data is loaded from the file ,inputfile'. The first plot visualizes what the maximum number of people standing in a square meter was at each frame of the video. The second plot shows for each square meter of the map, what the maximum number of people on it was throughout the whole video.

5.6 Model constants

Name	Parameter	Section	Value
Initial desired velocity	v_{α}^0	5.2	1.3 m/s
Relaxation time	τ_{α}	5.2	1 s
Maximum speed	v_{α}^{max}	5.2	$1.3 v_{\alpha}^0$
Territorial sphere pedestrian interaction strength	A_{α}^1	5.3	0
Territorial sphere pedestrian interaction range	B_{α}^1	5.3	0.3 m
Anisotropic character	λ_{α}	5.3	0.75
Physical pedestrian interaction strength	A_{α}^2	5.3	3 m/s^2
Physical pedestrian interaction range	B_{α}^2	5.3	0.2 m
Boundary interaction strength	$A_{\alpha\beta}$	5.4	5 m/s^2
Boundary interaction range	$B_{\alpha\beta}$	5.4	0.1 m
Radius of pedestrian	r_{α}	5.4	0.3 m

The model constants were chosen as proposed by Helbing [2].

6 Simulation Results and Discussion

As stated before, two different situations were considered. The situation with a ninety degree corner (section 5.2.4) and the “cross” situation (section 5.2.3). The results from our simulations without obstacles are presented in section 6.1, where the main goal was to identify the critical spots. Further simulations included bottlenecks and obstacles which are discussed in section 6.2. In these simulations we examined what influence the obstacles and bottlenecks had and if they reduced or increased the safety of the pedestrians.

For our simulations the following parameters had to be set:

- Number of pedestrians
- Time frame
- Obstacles
- Situation

6.1 Simulation without obstacles

To determine the critical spots in a given situation, a measurement for risk is needed. We decided to measure risk with the crowd density which is defined as people per square meter. The risk increases significantly with higher crowd density. According to G. Keith Still [3] crowd densities above 4 people per square are considered dangerous. In our simulations we plotted the crowd densities to identify the dangerous spots.

For the simulation without obstacles we chose the number of pedestrians to be 200 and 300 pedestrians. The time frame was set to 30 seconds.

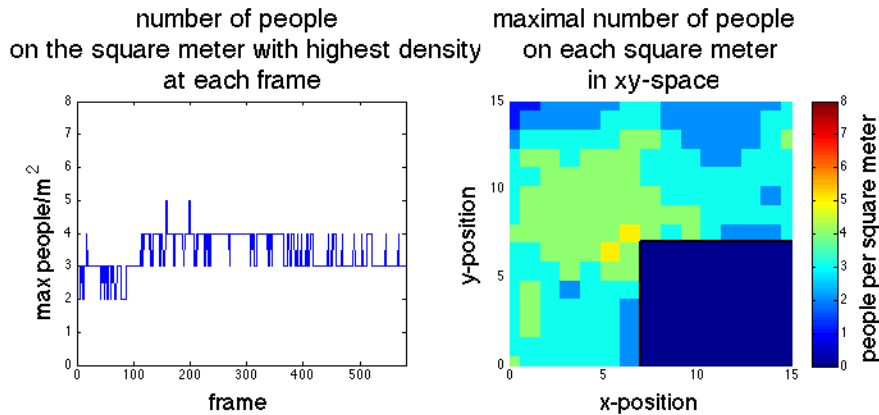


Figure 5: Simulation of the “curve” situation with 200 people and no obstacles

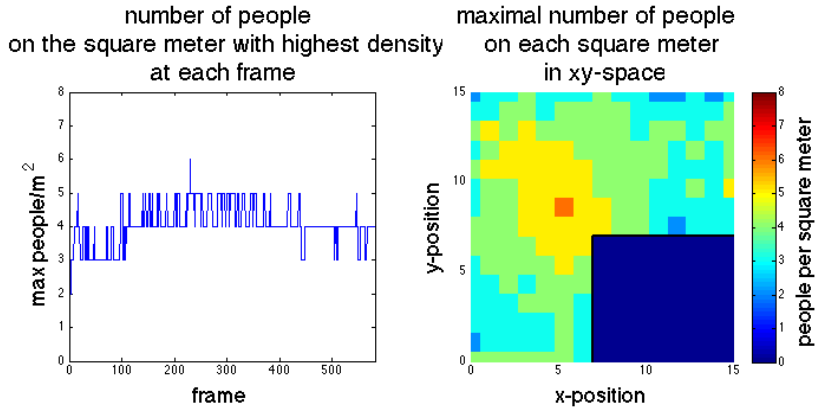


Figure 6: Simulation of the “curve” situation with 300 people and no obstacles

It can be seen that the overall crowd density is higher in the simulation with 300 pedestrians. The critical spots appear to be sharp corners in both plots which confirms our assumptions.

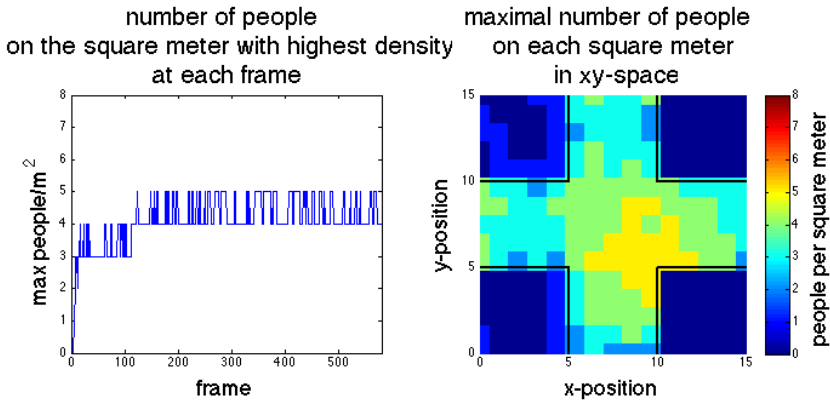


Figure 7: Simulation of the “cross” situation with 200 people and no obstacles

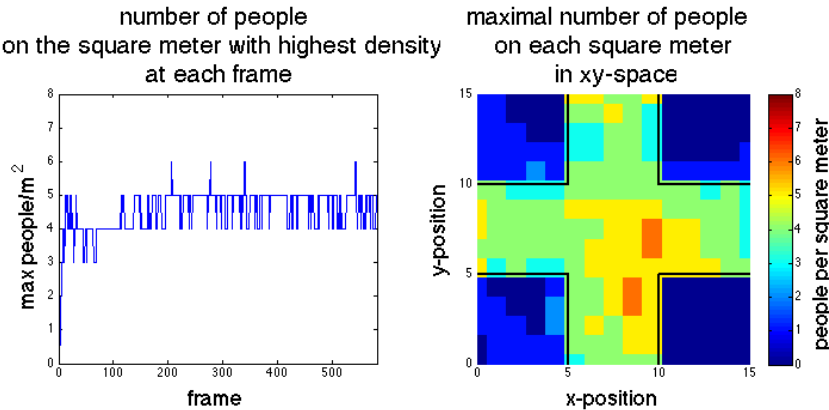


Figure 8: Simulation of the “cross” situation with 300 people and no obstacles

The resulting plots of the “cross” situation are similar to those of the “curve” situation. Dangerous spots are corners. It can be seen that the pedestrians are pushed down to the bottom right corner because of the perpendicular crowd flows.

Another phenomenon which can be observed is lane formation. Lane formations consist of pedestrians with a uniform walking direction as described by Helbing [1].

Videos of the simulation runs can be found in the videos folder of the Github repository.

6.2 Simulation with obstacles

In these simulations we placed obstacles in the path of the agents to identify their influence on the safety of the pedestrians. Further, we wanted to find obstacles which reduce the crowd densities, hence the safety of pedestrians. We therefore chose the same situations to compare the results.

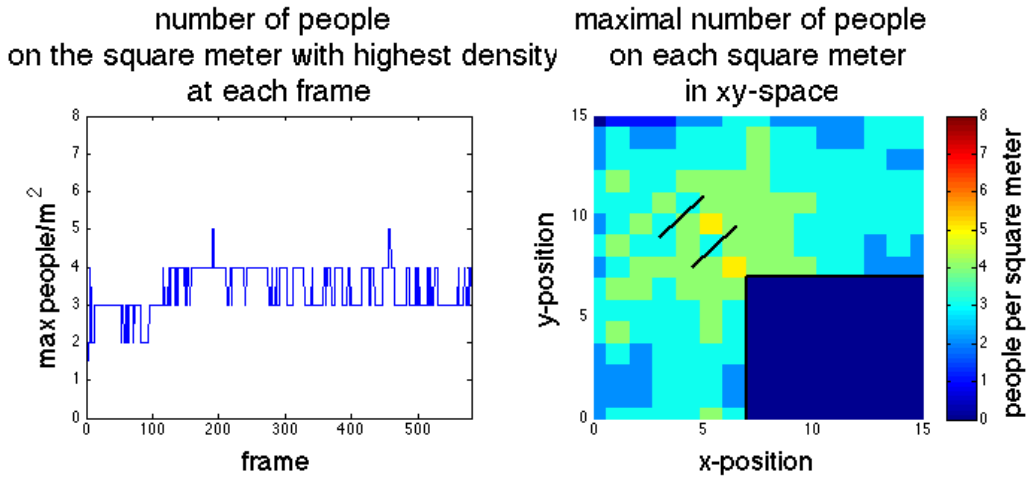


Figure 9: Simulation of the “curve” situation with 200 people and obstacles

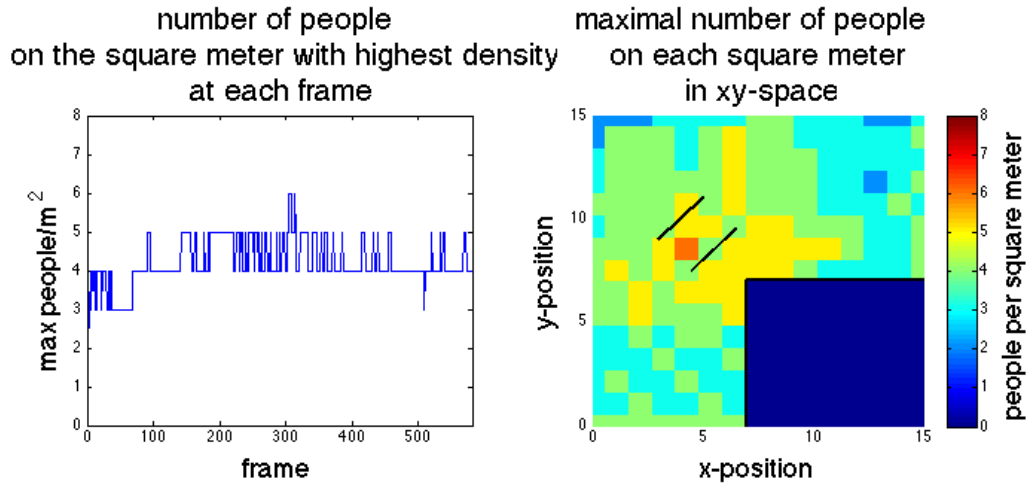


Figure 10: Simulation of the “curve” situation with 300 people and obstacles

We placed the obstacle as seen in figure 9 and 10. We thought it might guide the crowd flow and make it less chaotic.

However, it can be observed that the risk for the pedestrians isn’t reduced. The critical spots are displaced to bottlenecks. The resulting plot might be misleading because of

repulsive pedestrian forces acting through walls (see section 7). Again lane formations occur in the simulation.

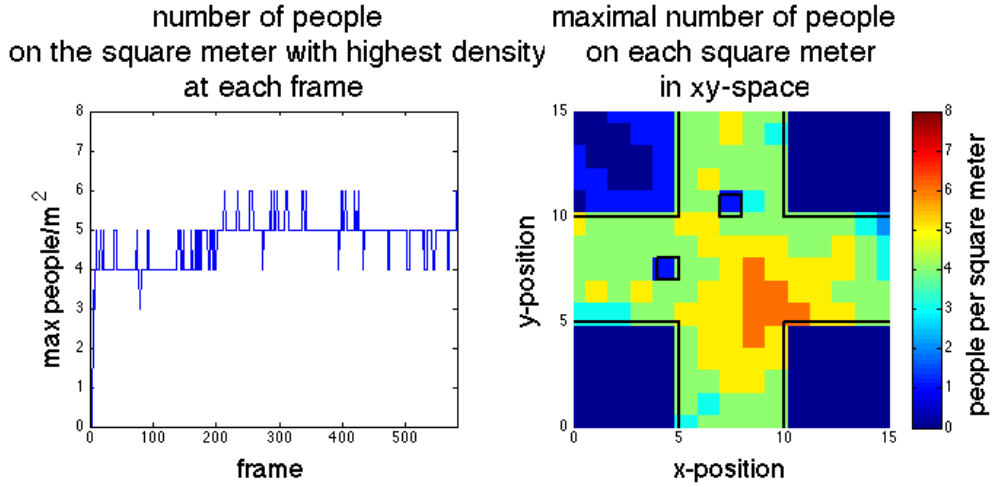


Figure 11: Simulation of the “cross” situation with 300 people and obstacles

To reduce the risk for the agents we placed two obstacles in the lanes. The idea was to buffer the incoming pedestrians and distribute the density. As seen in figure 11 that is not the case. The pressure at the corner is even higher than before.

7 Summary and Outlook

Our model is sufficient for identifying critical spots in different locations with dense crowds. To find arrangements which reduce the maximum crowd density, further situations would have to be investigated.

The chosen constants make the model work best in dense crowds. One could adjust the computation of the acceleration force so that the simulation becomes more realistic for situations with fewer people. One could also implement attraction effects as mentioned by Helbing [2].

Further improvements of our implementation could be made. For better performance, it would be best if pedestrian force was only computed amongst agents that are near to each other. For this purpose one could store agents in a quadtree, this would enhance the performance further.

More important though is the fact, that agents still influence each other even though they are separated by a wall. This has an influence on the quality of the simulation. Therefore, pedestrian force computation should consider walls between agents. For analysing the density of the crowd, a different method could be applied. Instead of discretizing the space into square meter “buckets” that count people on them, one could take a finer mesh grid and calculate for every point the number of people in the square meter around that point.

All in all, there are no significant new insights gained. However, the implementation comes up with similar results as Helbing in [2] and backs up personal experiences gained in a curve like situation. Further, a reusable implementation is presented to which a range of different situations could be applied.

The project was an interesting extension to our study routine. We gained insight into the topic of simulating social systems, how to program with Matlab and we practiced our scientific writing skills.

8 References

- [1] P. M. D. Helbing, "Social force model for pedestrian dynamics," *Physical Review E*, vol. 51, no. 5, 1995.
- [2] L. B. A. J. T. W. D. Helbing, "Self-Organized Pedestrian Crowd Dynamics: Experiments, Simulations, and Design Solutions," *Transportation Science*, vol. 39, no. 1, pp. 1-24, 2005.
- [3] G. K. Still, "Crowd Risk Analysis and Crowd Safety," [Online]. Available: <http://www.gkstill.com/Support/crowd-density/CrowdDensity-1.html>. [Accessed 13 May 2014].

9 Source code

9.1 Test model

```
function testModel(steps,filename, situation, noAgents)
%filename has to be a string (e.g. 'test.mat')
%situation is either 0 --> cross situation
%                               1 --> curve situation

%save the seed on the Random Stream Generator so the results can be
%reproduced
stream = RandStream.getGlobalStream;

%Global variables
NOAGENTS = noAgents;
SPEED_MEAN = 1.3;
SPEED_DISTR = 0.1;
dt = 0.05;
framesNo = (steps-1)/dt+1;

%agent = [x-position, y-position, x-speed, y-speed, desired Velocity,
type (which start side), waypointNO, average speed]
%agents = [agent1; agent2; agent3;...]
agents = zeros(NOAGENTS,8);
agentsUpdated = zeros(NOAGENTS,8);
positionDataX = zeros(NOAGENTS,framesNo);
positionDataY = zeros(NOAGENTS,framesNo);
pplSqData = zeros(15,15,framesNo);

if(situation == 0)
    %Init waypoints to reach destination
```

```

waypointsA = [10,6,10,9;
              15,5,15,10;
              16,5,16,10];
waypointsB = [6,5,9,5;
              5,0,10,0;
              5,-1,10,-1];

%walls form a cross
walls = [-20,5,5,5;
        -20,10,5,10;
        10,5,35,5;
        10,10,35,10;
        5,-10,5,5;
        5,10,5,35;
        10,-10,10,5;
        10,10,10,35];

%Obstacles are added here:
walls = [walls; makeObstacleRect(4,7,5,8);
makeObstacleRect(7,10,8,11)];
%walls = [walls; makeObstacleTriangle(7,7,8,8,6,9)];
%walls = [walls;2,10,4,8; 8,11,10,13];

end

if(situation == 1)
    %Init waypoints to reach destination
    waypointsA = [2,7,7,7;
                  3,13,8,8;
                  15,7,15,15;
                  16,7,16,15];
    waypointsB = [7,7,7,13;
                  1,11,6,6;
                  0,0,7,0;
                  0,-1,7,-1];

    %walls form a curve
    walls = [0,-10,0,15;
            0,15,25,15;
            7,-10,7,7;
            7,7,25,7];

    %Obstacles are added here:
    %walls = [walls;5,7,7,9;4,8,6,10;3,9,5,11];
    %walls = [walls;4.5,7.5,6.5,9.5;3,9,5,11];

end

%Initialize agents according to situation
for a = 1:NOAGENTS

    % Sets the desired speed of the agent in a normaldistribution with
    % SPEED_MEAN +- SPEED_DISRTR
    speed = SPEED_MEAN + sqrt(SPEED_DISTR)*randn;

    if(situation == 0)

```

```

    %agents on left side
    if(a<=NOAGENTS/2)
        posx = -5 + 4*rand;
        posy = 5 + (10-5)*rand;
        agent = [posx,posy,speed,0,speed,0,1,speed];
    %agents on top
    else
        posx = 5 + (10-5)*rand;
        posy = -5 + 4*rand;
        agent = [posx,15-posy,0,-speed,speed,1,1,speed];
    end
end

if(situation == 1)
    %agents at the bottom
    if(a<=NOAGENTS/2)
        posx = 0 + (7-(0))*rand;
        posy = -5 + (0-(-5))*rand;
        agent = [posx,posy,0,speed,speed,0,1,speed];
    %agents at the right side
    else
        posx = 15 + (20-15)*rand;
        posy = 0 + (8-(0))*rand;
        agent = [posx,15-posy,-speed,0,speed,1,1,speed];
    end
end

agents(a,:) = agent;

end

counter=1;
%-----START OF SIMULATION-----
----
% time loop
for time = 1:dt:steps
    pplSqUnit = zeros(15,15);
    %-----FOR EACH AGENT DO CALCULATIONS-----
    ----
    % agent loop
    for a = 1:NOAGENTS
        agent = agents(a,:);

        %-----Set up desired destination for the agent-----
        ---
        if(agent(6)==0)
            [ex,ey,d] =
vectorFromWall(waypointsA(agent(7,:),:),agent(1),agent(2));
        else
            [ex,ey,d] =
vectorFromWall(waypointsB(agent(7,:),:),agent(1),agent(2));
        end
        % waypoint is reached
        if(d<0.5)
            [s,dontcare] = size(waypointsA);
            %agent not at en-destination yet -> set next waypoint
            if (agent(7) < s)

```

```

        agent(7) = agent(7)+1;
        %agent at end-destination -> set agent back into start area
    else
        speed = SPEED_MEAN + sqrt(SPEED_DISTR)*randn;

        if(situation == 0)
            if(agent(6)==0)
                posx = -5 + (2-(-2))*rand;
                posy = 5 + (10-5)*rand;
                agent = [posx,posy,speed,0,speed,0,1,speed];
            else
                posx = 5 + (10-5)*rand;
                posy = -5 + (2-(-2))*rand;
                agent = [posx,15-posy,0,-
speed,speed,1,1,speed];
            end
        end

        if(situation == 1)
            if(agent(6)==0)
                posx = 0 + (7-(0))*rand;
                posy = -5 + (0-(-5))*rand;
                agent = [posx,posy,0,speed,speed,0,1,speed];
            else
                posx = 15 + (20-15)*rand;
                posy = 0 + (8-(0))*rand;
                agent = [posx,15-posy,-
speed,0,speed,1,1,speed];
            end
        end

        if(agent(6)==0)
            [ex,ey,d] =
vectorFromWall(waypointsA(agent(7,:),:),agent(1),agent(2));
        else
            [ex,ey,d] =
vectorFromWall(waypointsB(agent(7,:),:),agent(1),agent(2));
        end
    end

    %-----Acceleration force of every agent is calculated-----
    ----
    [accFx,accFy] = accelerationF(-ex, -ey, agent(3), agent(4),
agent(8), agent(5));
    %Add acceleration force to velocity
    agent(3) = agent(3) + dt*accFx;
    agent(4) = agent(4) + dt*accFy;

    %-----Wall forces for every agent-----
    ----
    mind = 10000;
    [noWalls,dontcare] = size(walls);
    %calculate nearest wall
    for w = 1:noWalls
        wall = walls(w,:);
        [vx,vy,d] = vectorFromWall(wall,agent(1),agent(2));
    end

```

```

        if (d<mind)
            mind = d;
            minn = [vx,vy];
        end
    end
    %Influence of nearest wall on agent
    [wallFx, wallFy] = wallF(mind, minn);
    agent(3) = agent(3) + dt*wallFx;
    agent(4) = agent(4) + dt*wallFy;

    %-----Force from other agents-----
    ----

    for i = 1:NOAGENTS
        if (a~=i)
            %Do calc for all other agents except oneself
            otheragent = agents(i,:);
            [pedFx,pedFy] =
pedestrianF(agent(1),agent(2),otheragent(1),otheragent(2));
            %Add pedestrian force to velocity
            agent(3) = agent(3) + dt*pedFx;
            agent(4) = agent(4) + dt*pedFy;
        end
    end

    %-----Update position according to new velocity-----
    ----

    agent(1) = agent(1) + dt*agent(3);
    agent(2) = agent(2) + dt*agent(4);

    %-----Save in which square meter the agent is in-----
    ----

    if(ceil(agent(2)) >0 & ceil(agent(1)) >0 & ceil(agent(2))<=15 &
ceil(agent(1))<=15);
        pplSqUnit(ceil(agent(2)),ceil(agent(1))) =
pplSqUnit(ceil(agent(2)),ceil(agent(1)))+1;
    end

    %Update average speed
    agent(8) = averageSpeed(agent(8),sqrt(agent(3)^2 + agent(4)^2),
time);

    %We use a new matrix agentsUpdated so all agents are updated in
one

    %step and for updates depending on the others positions the old
%positions of already updated agents are used
    agentsUpdated(a,:) = agent;

end

agents = agentsUpdated;
%save data into matrices for plotting later
positionDataX(:,counter) = agents(:,1);
positionDataY(:,counter) = agents(:,2);

```



```

    pplSqData (:,:,counter) = pplSqUnit;
    counter = counter+1;

end

save(filename);

end

%Average speed of the agent, considering current step in time
function avgSpeed = averageSpeed(old_avg, newSpeed, time)
avgSpeed = (old_avg * (time-1) + newSpeed) / time;
end

% nervousness, initial desired velocity is given by desSpeed
function nerv = nervousness(avgSpeed, desSpeed)
nerv= 1-avgSpeed/desSpeed;
end

% acceleration force from own desired direction
function [accFx, accFy] = accelerationF(desX, desY, vx, vy, vavg,
desSpeed)
n = nervousness(vavg, desSpeed);
desVel = (1-n)*desSpeed + n*(desSpeed*1.3);
accFx = desVel*desX-vx;
accFy = desVel*desY-vy;
end

%force from walls
function [wallFx, wallFy] = wallF(mind, n)

wallFx = 5*exp((0.3-mind)/0.1) * n(1);
wallFy = 5*exp((0.3-mind)/0.1) * n(2);

end

%force from one other pedestrian
function [pedFx, pedFy] = pedestrianF(x, y, otherx, othery)

d = [x,y]-[otherx,othery];
pedFx = 3*exp((0.6-norm(d))/0.2) * d(1)/norm(d);
pedFy = 3*exp((0.6-norm(d))/0.2) * d(2)/norm(d);

end

%the most direct vector from a wall to a position(x,y) and its length
function [vx,vy,d] = vectorFromWall(wall,x,y)

vectorA = [wall(1),wall(2)] - [wall(3),wall(4)];
vectorB = [x,y] - [wall(3),wall(4)];
l2 = norm(vectorA)^2;
t = dot(vectorB,vectorA)/l2;
if(t<0)

```

```

        n = vectorB;
        d = norm(n);
        n = n/norm(n);
    else if (t>1)
        n = [x,y] - [wall(1),wall(2)];
        d = norm(n);
        n = n/norm(n);
    else
        n = [-vectorA(2),vectorA(1)];
        d = abs(det([vectorA;vectorB])) / norm(vectorA);
        n = n/norm(n)*sign(dot(n,vectorB));
    end
end
vx = n(1);
vy = n(2);

end

%creates a rectangular obstacle out of walls
function obstacle = makeObstacleRect(x,y,x2,y2)

obstacle = [x,y,x,y2;
            x,y2,x2,y2;
            x2,y2,x2,y;
            x2,y,x,y];

end

%creates a triangular obstacle out of walls
function obstacle = makeObstacleTriangle(x,y,x2,y2,x3,y3)

obstacle = [x,y,x2,y2;
            x2,y2,x3,y3;
            x3,y3,x,y];

end

```

9.2 Simulation

```

function simulate(filename,mode,savevideo)
%filename has to be a string (e.g. 'test.mat')
%mode is either 0 (no background color with people per square) or 1
%savevideo has to be a string (e.g. 'video.avi')

load(filename);
h = figure();

vidObj = VideoWriter(savevideo);
vidObj.FrameRate = 20;
open(vidObj);

for time = 1:framesNo
    %plot with people per square or without
    if (mode==1)

```

```

        hold on
        %plot people per square
        imagesc(0:15,0:15,pplSqData(:,:,time),[0,8]);
        colormap jet;
        cb = colorbar('vert');
        zlab = get(cb,'ylabel');
        set(zlab,'String','people per square meter','fontsize',14);
    end
    %plot agents in black
    plot(positionDataX(:,time),positionDataY(:,time), 'Marker',
'o','LineStyle','none','MarkerSize', 10, 'MarkerEdgeColor','k')
    tlhand = get(gca,'title');
    set(tlhand,'string',sprintf('Agents in xy space over
time'),'fontsize',16);
    xlhand = get(gca,'xlabel');
    set(xlhand,'string','x-position','fontsize',14);
    ylhand = get(gca,'ylabel');
    set(ylhand,'string','y-position','fontsize',14);
    set (gca, 'YLimMode', 'Manual', 'YLim', [0 15], 'XLim', [0 15]);
    [m,n] = size(walls);
    for w = 1 : m

line([walls(w,1);walls(w,3)], [walls(w,2);walls(w,4)], 'Color','k', 'LineW
idth',2)
    end
    if(mode==1)
        hold off
    end

    currFrame = getframe(h);
    writeVideo(vidObj,currFrame);

end

close(vidObj);

end

```

9.3 Max People per square

```

function maxPeopleOnSquare(inputfile)

load(inputfile);
ppsVsTime = zeros(framesNo);
maxPerSquare = zeros(15,15);

for time = 1:framesNo
    ppsVsTime(time) = max(max(pplSqData(:,:,time)));
end

for i = 1:15
    for j = 1:15
        maxPerSquare(i,j) = max(pplSqData(i,j,:));
    end
end

```

```

end
hFig = figure;
set(gcf, 'PaperPositionMode', 'auto')
set(hFig, 'Position', [0 0 1070 405])
subplot(1,2,1)
plot(ppsVsTime)
axis([0 framesNo 0 8])
tlhand = get(gca, 'title');
set(tlhand, 'string', sprintf('number of people \n on the square meter
with highest density \n at each frame'), 'fontsize', 20);
xlhand = get(gca, 'xlabel');
set(xlhand, 'string', 'frame', 'fontsize', 18);
ylhand = get(gca, 'ylabel');
set(ylhand, 'string', 'max people/m^2', 'fontsize', 18);
subplot(1,2,2)
hold on
imagesc(0:15, 0:15, maxPerSquare, [0, 8]);
[m,n] = size(walls);
for w = 1 : m

line([walls(w,1);walls(w,3)], [walls(w,2);walls(w,4)], 'Color', 'k', 'LineW
idth', 2)
end
axis([0 15 0 15])
tlhand = get(gca, 'title');
set(tlhand, 'string', sprintf('maximal number of people \n on each square
meter \n in xy-space'), 'fontsize', 20);
colormap jet;
cb = colorbar('vert');
zlab = get(cb, 'ylabel');
set(zlab, 'String', 'people per square meter', 'FontSize', 18);
xlhand = get(gca, 'xlabel');
set(xlhand, 'string', 'x-position', 'fontsize', 18);
ylhand = get(gca, 'ylabel');
set(ylhand, 'string', 'y-position', 'fontsize', 18);
%print -depsc2 test.eps;
hold off

```