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

Project Report

Modelling Crowd Behaviour in the Polymensa Using the Social Force Model

Moritz Vifian, Matthias Roggo, Michael Aebli
Zurich

December 2011

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

Moritz Vifian	Matthias Roggo	Michael Aebli
________________	________________	________________

Abstract

The aim of this report to simulate crowd behaviour in the ETHZ polymensa. Being able to realistically simulate crowd behaviour could serve as a useful tool in architectural design in general. Thus the core goal of this report is to find the degree of realism with which crowd behaviour may be simulated. In order to answer this question the Social Force Model was implemented in form of a matlab program. This model was extended with queueing heuristics and a real time fast marching algorithm. The results of the simulation show patterns in the crowd dynamics which resemble empirical observations. Inorder to justify these claims emperical data would need to be collected.

Contents

1 Individual contributions

The main idea of the queueing in the *Polymensa* was developed as a group. From the beginning our goal was to use matrices and matrix operation wherever possible. Later on Matthias Roggo was mostly occupied with the task of converting bitmaps into vector fields. It was also him who found the Fast Marching Algorithm to be useful for the path finding of our agents. In the end he added the support for logging and statitics. Michael Aebli was the one who initially started the report and wrote most of the general text. He also tried to calculate the agent forces. In his attempt he used ellipsical potential which were confusing which needed to simplified in the end. Moritz Vifian created the simulation sequence and the basic drawing and contributed a simplified formula for the agent forces.

2 Introduction and Motivation

2.1 General Introduction

It has been noted by many people that the architectural design of the Polymensa is sub-par. The queue formation seems chaotic in nature and not predetermined. For example, people are forced to cross each others paths. The question arises if there is room for improvement or if the current system is already optimal as it is. More broadly speaking, this is a problem concerning pedestrian/crowd dynamics. How do pedestrians chose their paths and how do they interact. Pedestrian dynamics has many fields of application (such as evacuation) and is becoming more important as cities and buildings are more densely populated. There have been many proposed models for pedestrian dynamics. A well known example is the Social Force Model of 1995 (Helbing Dirk et. al).

2.1.1 Fundamental Questions

1. How do pedestrian groups with different destinations interact with one another?

    - How does one group of pedestrians with one destination behave?

- How do two groups of pedestrians with separate destinations interact? (Fig 1, top)
- How do two groups of pedestrians with separate destinations interact when there are obstacles?
- How do two groups of pedestrians with separate destinations interact closely in a room 2D room resembling a birds-eye-view of the Polymensa?
- What happens when there are three groups or more? (Fig 1, middle)
- Is queue formation accurately modelled? (Fig 1, bottom) What changes/additions need to be made if this is not the case?

2. How do pedestrian groups with different destinations interact when all pedestrians return to the same checkout point (cash register) once they have reached their respective destination?

3. How accurately are pedestrian-dynamics models able to depict the empirically collected crowd behaviour of the Mensa queue area?

   - Are queues formed at the same locations?
   - Are similar distributions observed?
   - Are densely occupied areas in the same location?
   - Are certain areas never occupied?
   - What geometrical changes could be made to the Mensa to increase the flow of pedestrians per unit time?

2.2 Variables of Interest

- The time it takes for all pedestrians to reach their destinations – measured by logging simulation
- The number of times pedestrians cannot move because their path is blocked by another pedestrian – measured by logging simulation
- The location of the queues (are they realistic?) – measured by analysing simulation snapshots

2.3 Expected Results

Results which resemble the crowd behaviour observed at the Polymensa are expected. This means queue formation at similar localities and similar flow of pedestrians per unit of time. We expect that the flow of pedestrians per unit of time can be influenced by the number of different destinations. The more destinations the slower the whole process.

3 Description of Model

This section aims to describe the social force model. The social force model aims to reduce pedestrian dynamics to force fields. Force fields act upon agents. Agents are an abstraction of pedestrians. The nomenclature used is consistent with the paper "Social Force model for pedestrian dynamics" by Dirk Helbing et al.

3.1 Agents

Agents are objects with the local parameters **:**

- actual velocity $\vec{v}_\alpha(t)$
- current actual position $\vec{r}_\alpha(t)$
- desired speed $v_\alpha^0(t)$
  - This is a speed which is initiated at the beginning of the simulation and remains constant throughout the simulation. The speeds are randomly Gaussian distributed $\mathscr{N}(v^0, \sqrt{\theta})$.

And the global parameters :

- relaxation time: τ_0
- step size: Δs

3.2 Force Fields

All objects recognizable by an agent have a potential field. Using the potential field of a repulsive or attractive object, the force at a certain location and time may be determined. The final force acting on an agent is the superposition of all forces. These forces are: (1) Destination, (2) other agents, (3) boarders/buildings/walls, (4) objects of attraction. These will be described more closely in the following section.

3.2.1 Destination

The destination attracts an agent. More specifically, the closest point, $\vec{r}_\alpha^k$, of the destination polygon to the agent at position $\vec{r}_\alpha(t)$ attracts an agent. An agent will attempt to take the most direct route to reach his destination. The vector from an agent to the closest destination point forms the desired direction. The force caused by the destination is referred to as acceleration term. It is independent of distance and time.

$$
\begin{aligned}
\vec{e}_\alpha(t) :=& \frac{\vec{r}_\alpha^k - \vec{r}_\alpha(t)}{||\vec{r}_\alpha^k - \vec{r}_\alpha(t)||} \\
\vec{F}_\alpha^0(\vec{v}_\alpha, \vec{v}_\alpha^0, \vec{e}_\alpha(t)) :=& \frac{1}{\tau_\alpha}(v_\alpha^0 \vec{e}_\alpha - \vec{v}_\alpha)
\end{aligned}
$$

3.2.2 Other Pedestrians/Agents

An agent attempts to avoid other agents. Therefore other agents have a repulsive nature. This repulsive nature is described by a monotonically decreasing potential field $V_{\alpha\beta}[b(\vec{r}_{\alpha\beta})]$. This potential field has elliptic equipotential lines. The agent is at the center of the ellipses. The semi major axis is aligned parallel to the desired direction of the agent. The potential is calculated as a function of the semi minor axis of the ellipse which is in turn dependent on the agent to agent distance.

$$
\begin{aligned}
\text{Potential :} V_{\alpha\beta}\left[b\vec{r}_{\alpha\beta}\right] =& V_{\alpha\beta}^0 e^{-b/\sigma} \\
\text{semi Axis: } b =& \frac{1}{2}\sqrt{||\vec{r}_{\alpha\beta}|| + ||\vec{r}_{\alpha\beta} - v_\beta \Delta t e_\beta||^2 - (v_\beta \Delta t)||} \\
\text{Force Field: } \vec{f}_{\alpha\beta}(\vec{r}_{\alpha\beta}) =& -\nabla_{\vec{r}_{\alpha\beta}} V_{\alpha\beta}\left[b(\vec{r}_{\alpha\beta})\right]
\end{aligned}
$$

Lastly one must take into consideration that the field of vision of an agent has an influence on the forces. An other agent which is standing behind agent alpha will have a considerably smaller influence than one which agent alpha is facing.

$$
\begin{aligned}
\vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_\beta) :=& w(\vec{e}_\alpha, -\vec{f}_{\alpha\beta}(\vec{r}_\alpha - \vec{r}_\beta))\vec{f}_{\alpha\beta}(\vec{r}_\alpha - \vec{r}_\beta) \\
w(\vec{e}, \vec{f}) :=& \begin{cases} 1 & \text{if } \vec{e} \cdot \vec{f} \geq ||\vec{f}|| cos\varphi \\ c & \text{otherwise} \end{cases}
\end{aligned}
$$

A simplified approach to this problem is proposed by "Self organized pedestrian crowd dynamics" by Helbing et al. (2005). Here the agent to agent force is calculated as follows:

$$
\begin{aligned}
\vec{F}_{\alpha\beta}(t) =& A_\alpha^1 \exp\left[(r_{\alpha\beta} - d_{\alpha\beta})/B_\alpha^1\right] \vec{n}_{\alpha\beta} \cdot \left(\lambda_\alpha + (1-\lambda_\alpha)\frac{1+\cos(\varphi_{\alpha\beta})}{2}\right) \\
&+ A_\alpha^2 \exp\left[(r_{\alpha\beta} - d_{\alpha\beta})/B_\alpha^2\right] \vec{n}_{\alpha B}
\end{aligned}
$$

In this approach the agent force does not have elliptic equipotential lines as the gradient is equal for all points in equal distance from an agent. The values for the constants $A_\alpha^1, A_\alpha^2, B_\alpha^1$, and B_α^2suggested by by Helbing et al. were used (See 4. Implementation).

3.2.3 Boarders/ Buildings/ Walls

Boarders, buildings and walls repulse agents. They induce monotonically decreasing potential fields.

$$\begin{aligned} \text{boundry Repulsion Force } \vec{F}_{\alpha B} =& -\nabla_{\vec{r_{\alpha B}}} U_{\alpha B}(||\vec{r}_{\alpha B}||) \\ U_{\alpha B}(||\vec{r}_{\alpha B}||) =& U^0{}_{\alpha B} e^{-||\vec{r}_{\alpha B}||/R} \end{aligned}$$

3.2.4 Objects of Attraction

Objects of attraction draw agents towards themselves. They are similar to destination forces. This force is a function of time as it decays over time.

$$\text{attractive Force } \vec{f_{\alpha i}}(||\vec{r}_{\alpha i}||, t) = -\nabla_{\vec{r}_{\alpha i}} \cdot W_{\alpha i}(||\vec{r}_{\alpha i}||, t)$$

As with other agents attractive objects must be weighed according to the field of vision of the agent.

$$\begin{aligned} \vec{F}_{\alpha i}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_\beta) :=& w(\vec{e}_\alpha, -\vec{f}_{\alpha\beta}(\vec{r}_\alpha - \vec{r}_\beta))\vec{f}_{\alpha i}(\vec{r}_\alpha - \vec{r}_\beta) \\ w(\vec{e}, \vec{f}) :=& \begin{cases} 1 & \text{if } \vec{e} \cdot \vec{f} \geq ||\vec{f}|| cos\varphi \\ c & \text{otherwise} \end{cases} \end{aligned}$$

3.2.5 Total Force

The total force is the superposition of the above mentioned forces.

$$\vec{F}_\alpha(t) = \vec{F}^0_\alpha(\vec{v}_\alpha \vec{v}^0_\alpha \vec{e}_\alpha) + \sum_\beta \vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_\beta) + \sum_B \vec{F}_{\alpha B}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}^\alpha_B) + \sum_i \vec{F}_{\alpha i}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_i, t)$$

3.3 Social Force Model

The movement of the agents must now be derived from the total force acting on them at a specific time. This can be written as:

$$\frac{d\vec{w_\alpha}}{dt} := \vec{F}_\alpha(t) + fluctuations$$

The actual velocity must not be greater then the agent's desired velocity v^α_0defined during the initialization of an agent. The velocity is thus given as follows:

$$\text{Actual velocity } \vec{v}_\alpha = \vec{w}_\alpha \cdot g\left(\frac{v_0^\alpha}{||\vec{w}_\alpha||}\right)$$

$$g\left(\frac{v_0^\alpha}{||\vec{w}_\alpha||}\right) = \begin{cases} 1 & \text{, if } ||\vec{w}_\alpha|| < v_0^\alpha \\ \frac{v_o^\alpha}{||\vec{w}_\alpha||} & \text{, otherwise} \end{cases}$$

3.4 Simple Queueing

When many agents have the same destination, the social force model simpy produces a circular shaped crowd of agents all moving in the same direction. These agents do not naturally form queues. Agents need to be taught to form queues. If a person in real life wants to walk towards a specific location, she or her looks around to check whether there are other people headed in the same direction. If so, she or he joins the crowd and this leads to a queue. The thought process in que formation can be summarised as:

1. Look towards the goal. Is there anyone else in front of me headed the same way?
2. If there are more than one: Which is the latest in the queue?
3. Join him or her

This simple queueing model was the fundament of the queuing heuristics explained in the next Section, 4.Implementation.

4 Implementation

The social force model, as described in the previous section, was implemented as described in this section. The simulation input consists of a .png image file and a list of global parameters. On the input image information concerning destinations, starting positions and boundaries are stored. The social force model is then applied to this environment. The output two dimensional animation from a birds eye view simulating the movement of pedestrians according to the social force model.

4.1 Pseudo-code

The following is a pseudo code of the implementation of the social force model. It gives an overview of the chronology of the matlab program.

```
initialize global parameters
load map from image file
    create boundary matrix
    create destination matrix
    create starting matrix
```

```
calculate static potential fields from boundary matrix
    2d convolution of potential function
    and boundry matrix in frequency domain
calculate destination force
    calculate shortest path and convolute
    with potential function in frequency domain
initialize agents
    set agent starting positions in boundary matrix
    set agent initial speeds
    set agent desired speed
    set agent type (which destination?)
begin simulation loop
    for each angent
        calculate boundary force
        calculate destination force
        calculate other agents force
        calculate total force
        calculate new velocity and positions using euler method
    plot image and agents
end simulation loop
```

4.2 Initialization

4.2.1 Global Parameter values

The values of global parameters (directly related to the social force model) which were used are listed in the 1 on the following page. Unless specifically stated otherwise the following parameters were used.

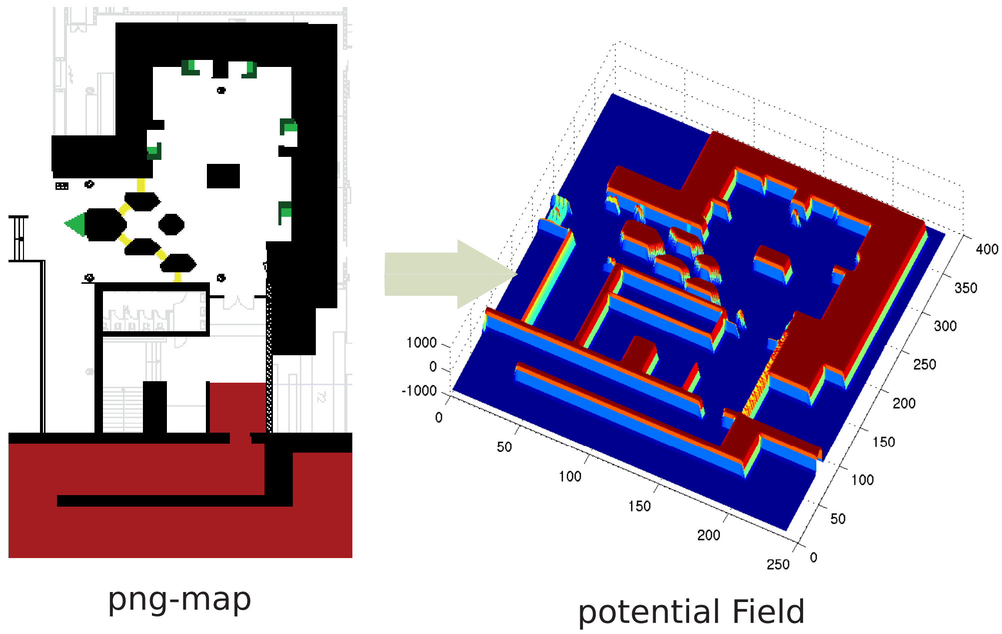
4.2.2 Image input / map of environment

An input image was used to initialize the destination, starting and boundary positions. The image was color coded. Red pixels were defined as starting positions, green pixels as destinations and black pixels as boundaries. "LOAD_MAP.M" is responsible for importing an image and making it usable. The potential field is generated through the convolution of a 2d potential function (see Section 2) and the black areas of the map which represent walls. The convolution was calculated as a multiplication in the frequency domain via matlab's Fast Fourier Transformation.

matlab code	**symbolic**	**used Value [$Units$]**	**recommended**	**Parameter description**
agent_number	-	150 [$Agents$]	-	Number of agents in simulation
duration	-	4000 [dt]	-	Number of frames
dt	dt	.1 [s]	-	time unit step
meter	$meter$	15 [px/m]	-	pixel to meter conversion factor
R	R	0.05 [m]	0.2 [m] *	parameter of boundary potential
U_alphaB_0	$U^0_{\alpha B}$	5 [m^2/s^2]	10 [m^2/s^2] *	Parameter of boundary potential
v0_mean	$< v_0 >$	1.34 [m/s]	1.34 [m/s] *	desired agent velocity
tau_alpha	τ_α	0.3 [s]	0.5 [s] *	relaxation time
sqrt_theta	$\sqrt{\theta}$	0.26 [m/s]	0.26 [m/s] *	standard deviation of v0_mean
sigma	σ	0.3 [m]	0.3 [m] *	parameter of agent potential
A1	A^1_α	0 [m]	0 [m] **	first interaction strength
A2	A^2_α	3 [m/s^2]	3 [m/s^2] **	second interaction strength
B1	B^1_α	0.1 [m]	0.1 [m] **	first interaction range
B2	B^2_α	2 [m]***	0.2 [m] **	second interaction range
sight	-	1.5 [m]	-	fadius within which other agents have influence

* recommend by Helbing et al. in "Social force model for pedestrian dynamics" (1995) Section IV. COMPUTER SIMULATION
** recommended by Helbing et al. in "Self-organized crowd dynamics Helbing" (2005) section 4. (8)
*** This value was increased by factor 10 to make agent to increase agent distances

Table 1: Parameter Table



4.3 Social Force model algorithm

In order to speed up our simulations, we separated the effective forces on our agents in two groups, being either constant on a given place or depending on all other agents positions.

4.3.1 Environmental forces

The forcefields related to the destination ($\vec{F}_\alpha^0$), walls ($\vec{F}_{\alpha B}$) and attractive objects ($\vec{F}_{\alpha i}$) are independent of the agents positions, and can thus be calculated in advance. This is done by LOAD_MAP.M, which generates a force field for the walls' potential and every target area on the map.

Fast marching algorithm To prevent agents from getting stuck on more complex floor plans, we employ the *accurate fast marching algorithm* implemented by Dirk-Jan Kroon[1].

Based on a "speed map", it creates a potential field representing the shortest time needed to get to one or more target points. Its gradient thereby serves as an indicator for the shortest path, our new $\vec{e}_\alpha$ (see figure 2 for an example). The gradient field is then normalized to $\frac{v_\alpha^0}{\tau_\alpha}$ to represent the first term of $\vec{F}_\alpha^0$ (3.2.1 on page 8).

[1] http://www.mathworks.com/matlabcentral/fileexchange/24531

4.3.2 Agent forces

Our method for calculating the agent forces is a mix of cellular automaton and agent based model. Usually one must calculate the forces from one agent to every other and so on. Meaning: The complexity for n Agents is n^2. As the force from one agent decays exponentially with the distance we consider only the agent inside the radius *sight*. For our purposes *sight* is set to 1 meter.

In Matlab it selecting entries hat fulfill a certain condition from a matrix is very simple:

```
close_agents = sqrt(sum(r_alphabeta_matrix.^2))<sight;
agent_others = A(:,close_agents);
```

This lowers the complexity to n as there is a maximum of agents inside a certain area.

The final expression for calculating the forces from the current agent to all others is not totally self explanatory.

```
F_tot = F_tot ...
                + A2*sum(...
                (ones(2,1)...
                *((2*(agent_others(6,:)==1)+1)...
                .*exp((2+(agent_alpha(6)==1))...
                *sigma*ones(1,agent_number_back-1)-sum(
                    r_alphabeta_matrix.^2)/B2))...
                ).*e_beta_matrix,2);
```

The first step is calculating the exponential for all distances from alpha to beta, minus their own radius.

```
exp(((radius_alpha+radius_beta)-distance_alpha_beta))/B2)
```

For calculating the distances we sum up the squares of distance vectors stored in `r_alphabeta_matrix`.

4.4 Etiquette training for ruthless agents (queueing heuristics)

Different attempts were made to improve the agent-flow through the menus. Since the agents don't take the crowd into account for their route-planing , even simple corrections (far from queuing) reduce the time it takes for hungry agents' to pass through the mensa.

Emerging and vanishing walls Instead of only drawing target areas on the map (green), we implemented support for target-specific walls (dark green). These destination sorrounding walls only impact agents targeting them. Agents targeting other destinations or leaving said destination remain unaffected by these additional walls. These additional wall potentials force waiting people into a "waiting-area", which allows served agents to leave more quickly.

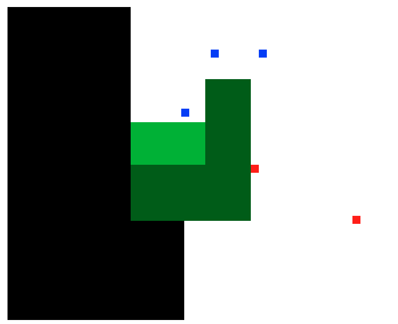


Figure 1: "Subjective" walls

Real-time fast marching So far, the FM (fast marching)-algorithm did not take into account whether an area was crowded or not: The agents are forced (by $\vec{e}_\alpha$) towards "their" check-out, even when another (free) one would be way quicker. This cause a huge crowd waiting in front of the first checkout (first when entering the room). Real-time FM can solve this problem, but since it takes 250 ms to calculate the FM-fields for all the five layers (each one corresponding to a target) we decided to neglect small changes. As a result FM-fields are updated every 20 frames rather than every frame. Furthermore, REFRESH_FIELDS.M draws the agents' location into an additional map and convolutes it with an exponential peak. The resulting force field aids the agents in avoiding large clusters of other agents. See figure 2 to see the difference between the forcefields that are drawn for agents headed for check-outs.

Queue sensing agents The queuing behaviour of an agent is simple. The estimated times until an agent reaches its destination are stored in the eighth entry of matrix A. These estimations are produced by the Fast Marching Algorithm. Agent alpha looks for all agents that are closer to the goal then its current position.

```
closer_agents = agent_others([1 2 8],((agent_others(8,:)<agent_alpha(8)
    )&(agent_others(6,:)==agent_alpha(6))));
```

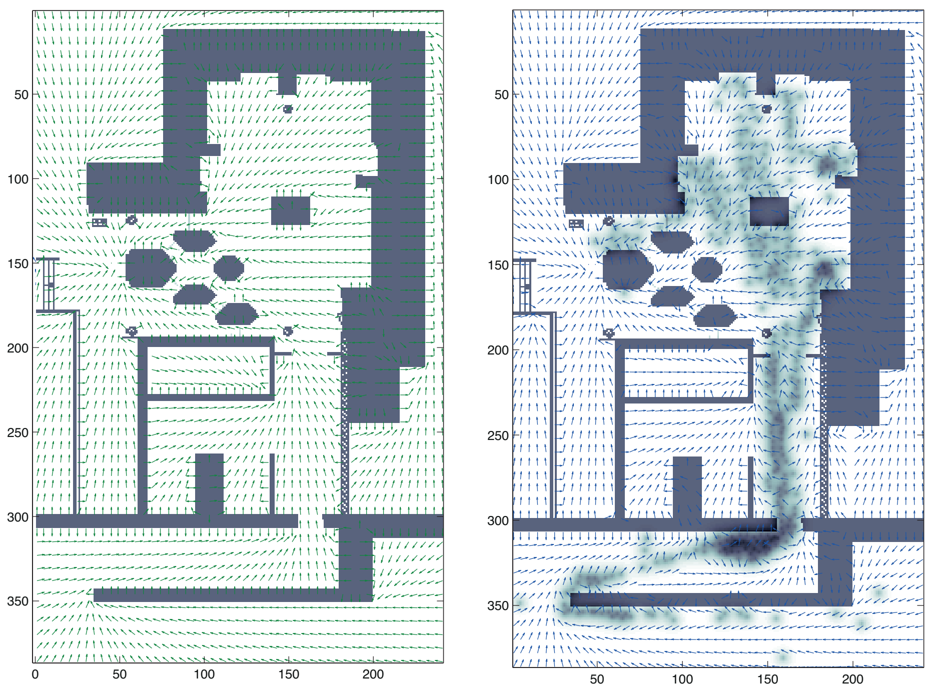


Figure 2: “Naive” vs. crowd-aware FM

Of these closer agents, agent alpha then selects the agent which is closest to its location.

```
[¬,I] = max (closer_agents(3,:));
        closest_agent = closer_agents(1:2,I);
```

The new goal is now the agent next in line.

```
d_direction = (closest_agent-agent_alpha(1:2))/norm(closest_agent-
    agent_alpha(1:2),2);
```

5 Simulation and Discussion

In order to run a simulation the PARAMETERS.M file must be altered. To run a simulation the file SIMULATION_V1.M must be launched. The implemented model was simulated with various parameters in an attempt to answer the questions which were initially posed.

5.1 Primitive Simulation with and without queues

One of the preliminary goals was to analyze crowd behavior for two groups with two goals. This is presented in the following section. The simulations were conducted with 200 agents and a time step of .1 seconds. First the simulation was executed with queue formation and then without it. The difference is very apparent. The queuing heuristics cause the formation of very distinct one man queues. Without the queues the crowds move in packs.

The code for the queuing can be found in the following snippet. Firstly, only the agents near by are looked at. Secondly, of the close by agents, the agent which is closest to the destination is found. Thirdly, the direction is influenced by the direction of the closest agent.

```
closer_agents=agent_others([1,2,8],((agent_others(8,:)<agent_alpha(8))
    ...
    &(agent_others(6,:)==agent_alpha(6))));
if(size(closer_agents,2)>0&&agent_alpha(6)≠1)
    [¬,I] = max (closer_agents(3,:));
    closest_agent = closer_agents(1:2,I);
    d_direction = d_direction*.5+0.5*(closest_agent-agent_alpha(1:2))
        ...
    /norm(closest_agent-agent_alpha(1:2),2);
end
```

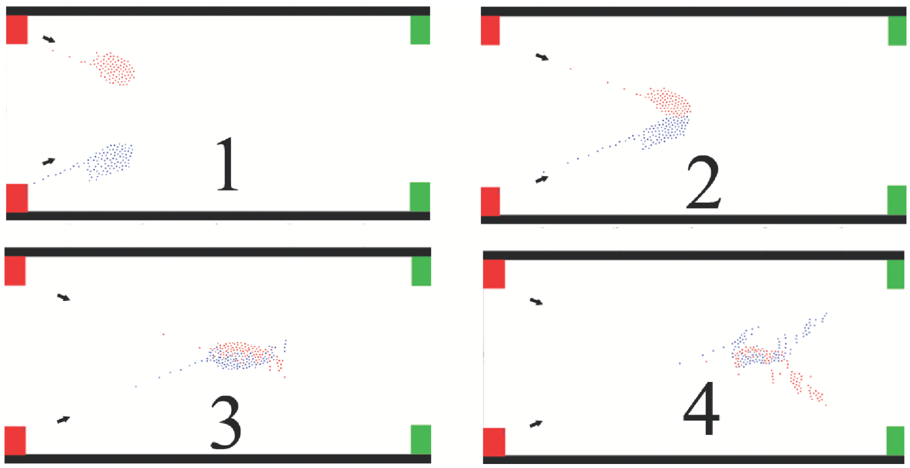


Figure 3: primitive simulation without queues

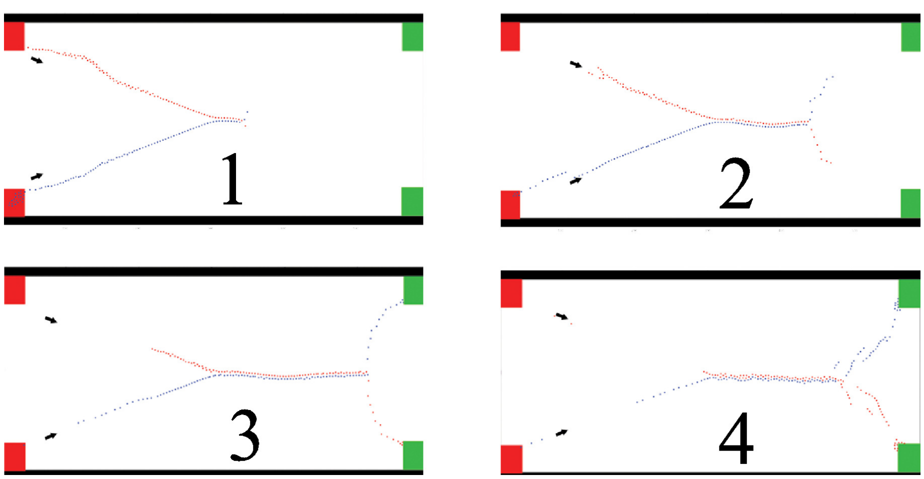


Figure 4: primitive simulation with queues

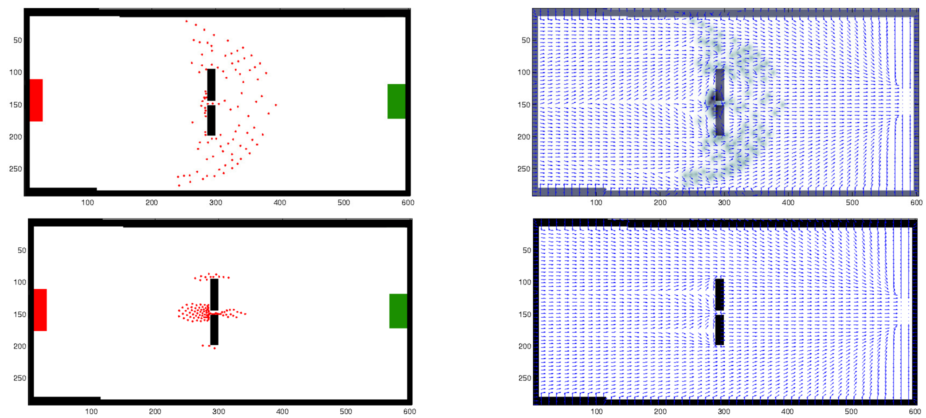


Figure 5: Top is with Fast marching algorithm, bottom image without it.

5.2 Primitive Simulation with and without real time fast marching algorithm

In Figure 5. the difference between agents using the real time fast marching algorithm are demonstrated. The differences are very apperant. Without real time FM agents are realtively closely packe. On the other hand, simulations using FM show agents more spread out taking the mass of the crowd into account.

5.3 Parameterization for the Mensa

To simulate hungry people rushing through the Mensa, a map based on *rauminfo.ethz.ch*[2] was constructed. The distribution of the agents to the different "menus" was done by rough estimation. It was estimate that about 10% of mensa-goers decide to buy the *bio-menü*.

While experimenting with different parameters, the walls had to be thickened several times. Otherwise, agents were pushed through alls due to the high forces.

The following plots are based upon 4000 to 5000 steps. Due to the slow simulation (5000 steps take about 15 minutes), the "parameter-sweeping" was done manually and very time consuming. Two exemplary problems we ran into:

Literally pitfalls With too few or slow checkouts, too many agents were trapped and no "continuos flow of agents" could be achieved. This can be verified in figure 7 which shows (in the upper part) the agents' passing-through-time compared to their arrival time: No upper limit seems to emerge.

High inrush After adapting some parameters (mostly METER), the map needed some tweaking to avoid unrealistically high inflow of agents (figure 8) which ultimately floods the mensa with people.

[2]http://www.rauminfo.ethz.ch/Rauminfo/grundrissplan.gif?region=Z&areal=Z&gebaeude=MM&geschoss=B

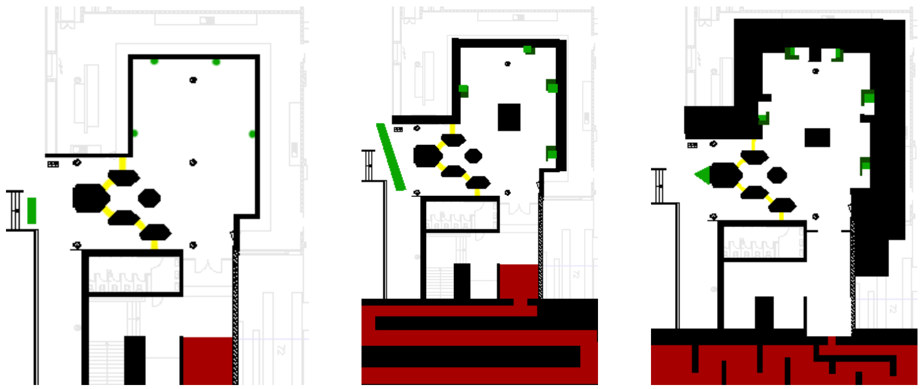


Figure 6: Map evolution process

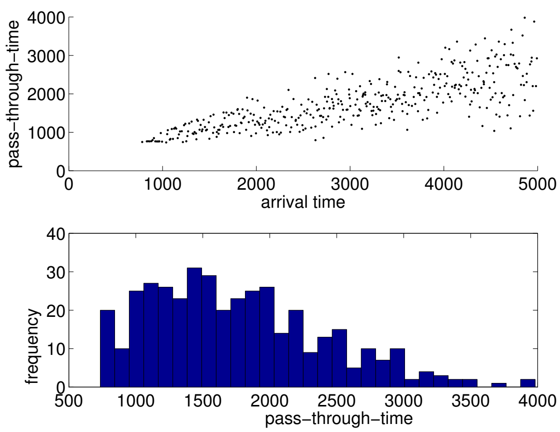


Figure 7: No steady-state

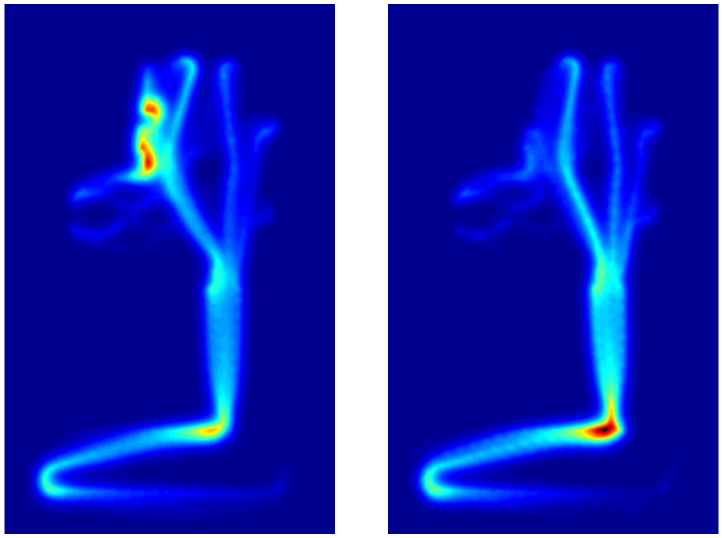


Figure 8: Tweaking the inflow

5.4 Comparing heuristics

With the found parameters (see Appendix on p. 30), a benchmark of our heuristic was done. The first run included just the **bare social-force model**, the second run used our **"simple queueing"** and the third featured **real-time fast marching** as well as our **queueing** extension (Shown from left to right in the following three plots).

Pass-through-time In figure 9, an "upper limit" for the pass-through-time shows up in all three runs: thus, no agents got stuck for very long periods of time. The different heuristics however do not seem to impact the passing-through-time.

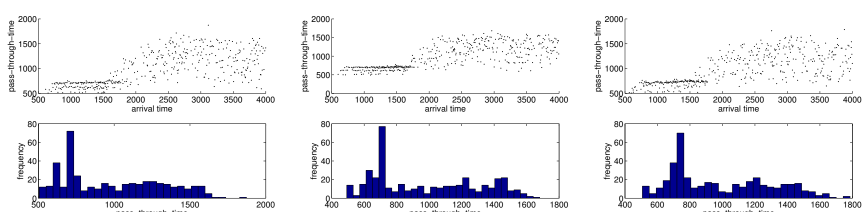


Figure 9: Pass-through-time

Space usage In the third comparison, the effort made finally pays of: By adding queuing heuristics (left vs. center image), it can be seen how the crowd just in front of the upper left service sparses. It can be concluded that queue aware agents rush less.

By recalculating the fast-marching-paths (center vs. right image), agents realise the crow in their way and take longer, but quicker ways to other checkouts.

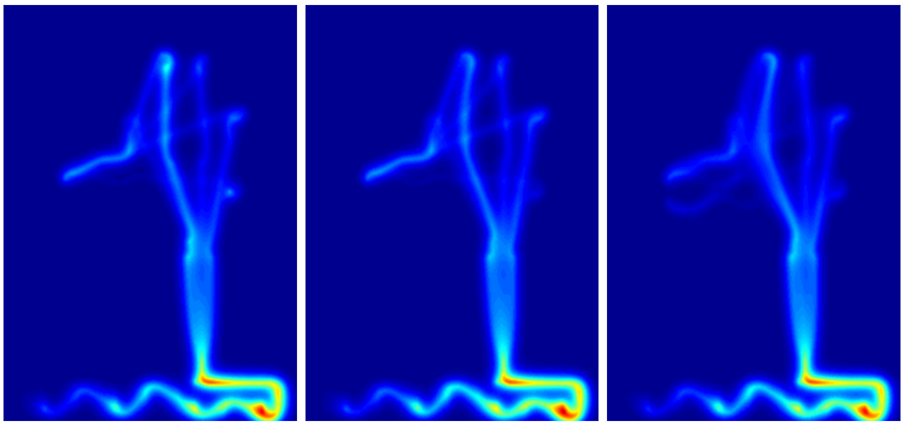


Figure 10: Comparing space usage

In the following screencaptures – and way better in the videos – we can see how the agents on their way to the checkout move more flexible with the real-time-fast-marching.

Figure 11: Simulation after 300 Frames, from left to right. 1. with queues, with fast marching, 2. without queues, with fast marching 3. with queues, without fast marching

Figure 12: Simulation after 1800 Frames from left to right. 1. with queues, with fast marching, 2. without queues, with fast marching 3. with queues, without fast marching

Figure 13: Simulation after 3600 Frames, from left to right. 1. with queues, with fast marching, 2. without queues, with fast marching 3. with queues, without fast marching

Figures 11- 13 show screencaptures of the videos which were made whilst collecting the data presented in Section 5.3. In these screencaptures – which are available as videos on github – it can be see how the agents on their way to the checkout move more flexible with the real-time-fast-marching.

6 Summary and Outlook

Building mainly on the concepts published by the “Social Force Model of Pedestrian dynamics” (1995) and “Self-organized Pedestrian Crowd Dynamics” (2005) which were both published by Dirk Helbing et al. an algorithm to simulate crowd behaviour was implemented. The task was to fabricate a concrete model derived from these general models. The models were to some extent modified and extended to realistically simulate crowd behaviour in the ETHZ Polymensa. Summarized, the added extensions were queue heuristics as well as crowd destination dynamics (first the different meals, then the cash register) which are described in detail in sections 4 and 5.

The simulations showed characteristics which group members have observed to exist (see Section 5.2), but the reproduction of a realistic queue turned out to be much more demanding than expected. Further improvements, such as agents who estimate their neighbours' speed and include this in their path planning, urge to be implemented.

From the current results it is not yet possible to draw a conclusion concerning the degree of realism of the simulation. In order to this, empirical data would have to be collected. This information could then be compared to simulated data. In a further step empirical data could be collected from various mensas which could also be simulated. If these prove to be realistic with only little modification of the model, then the simulation could be used as an aid in the architectural design of mensas. It is only possible through the extensive collection of empirical data to draw conclusions about empirical reality from computer simulated reality which was outside the scope of this report.

6.1 References

- *Social Force Model for Pedestrian dynamics (1995) – Dirk Helbing et al.*
  Describes model we are considering to build on (Social Force Model)

- *Self-organized Pedestrian Crowd Dynamics (2005) – Helbing et al.*
  Uses model we are considering to build on (Social Force Model)

- *Pedestrian Dynamics Airplane Evacuation Simulation – Author(s): P. Heer, L. Bühler*
  Model we are considering to build on (Social Force Model)

- *Response to intrusion into waiting lines. (2010) By Milgram, Stanley et al.*[3]
  A possible extension to queue modeling. This extension would include intruders which are more aggressive and attempt to intrude into waiting lines.

- *Approach to Collective Phenomena in Pedestrian Dynamics (2002) Andreas Schadschneider et al.*
  In case we attempt to use Cellular Automata this paper would be useful.

- *Writing Fast MATLAB Code (2004) – Pascal Getreuer*

[3]http://psycnet.apa.org/index.cfm?fa=buy.optionToBuy&id=1987-04011-001

7 Appendix

7.1 Source Code

All the source code is stored in the directory 'src/'

7.1.1 Agents Forces | agents_force.m

```
function [F_tot, agent_number_back] = agents_force(A,alpha)
    % DISCRIPTION:
    % This function calculates the force caused on the agent in coloumn
        alpha
    % of matrix A.
    %
    % DETAILED DESCRIPTION:
    % The calculation is based on the social force model. The
    % potential function used was V_alphabeta=v0_alphabeta*exp(-b/sigma
      ).
    % The parameter b calculates the smaller semi axis of the ellipse
      which
    % is centered on other agents (beta agents). The longer axis of
      this ellipse
    % is parallel with the directio vector of other agents. This means
      that
    % the exerted force from other agents depends which direction they
      are
    % facing. For more information see "Social forces model for
      pedestrian
    % dynamics"  III. FORMULATION OF SOCIAL FORCE MODEL  Dirk Helbing
      et
    % al
    %
    %
    % PARAMETERS:
    % A                      = 2 by agent_number matrix, Agentmatrix
    % alpha                  = Agent ID for which Force is to be
      calculated
    % b                      = smaller semi axis of ellipse
    % F                      = 2 by 1 vector, force caused by other
      Agents
    % agent_others           = 2 by agent_number-1 matrix containing
      other agent information
    % agent_alpha            = 2 by 1 vector containing agent alpha
      information
    % r_alphabeta_matrix     = 2 by agent_number-1 matrix with all
      distances between alpha/others
    % v_beta_matrix          = 2 by agent_number-1 matrix  all
      velocities of other agents
    % e_beta_matrix          = 2 by agent_number-1 matrix, direction
      vector
    % F_abs                  = 1 by agent_number-1 vector, absolute
      Force
    % e_alpha                = 2 by 1 vector, desired direction of the
      agent
    % Structure of Matrix A:
```

```
%
%                  | Agent 1   | Agent 2 | Agent 3
%-----------------------------------------------
% 1 position x      |
% 2 position y      |
% 3 speed    x      |
% 4 speed    y      |
% 5 desired speed v0|
% 6 goal            |
% 7 last counter
% 8 ETR from FM
% 9 Red Carpet...
%
%
% ==Some hints to understand the calculations below:==
%
% if A=[a b c; d e f] sum(A) or sum(A,1) creates the matrix [a+d,b+
    e,c+f]
% and sum(A,2) creates the matrix [a+b+c;d+e+f]
%
% if B=[3 4 2 5 1], then (B>2) creates the matrix
% [1 1 0 1 0] which one can use select partial matrices
% for example B([1 1 0 1 0]) results in [3 4 5]
%



global sigma tau_alpha agent_number; % global constants defined in
    parameters.m
global A1 A2 B1 B2 e_alpha_x e_alpha_y v0_mean sight tray_factor;
agent_alpha=A(:,alpha);

% calculate all r_alphabeta vectors (distance between) and store in
     matrix
r_alphabeta_matrix=(agent_alpha(1:2,:)*ones(1,agent_number)-A
    (1:2,:));


% Select only Agents inside the radius "sight"
close_agents = sqrt(sum(r_alphabeta_matrix.^2))<sight;

% Exclude agent_alpha from the selection
close_agents(alpha) = 0;

agent_others = A(:,close_agents);
agent_number_back = size(agent_others,2)+1;

% Remove unnecessary distances
r_alphabeta_matrix(:,¬close_agents) = [];



% In case two agents are on top of each other
% Find the entries where x and y are zero
% Totally unlikely, but it would cause a division by zero
null_entries = find(r_alphabeta_matrix(1,:)==0 & r_alphabeta_matrix
    (1,:)==0);
```

```
    % Replace the entries by random numbers
    r_alphabeta_matrix(:,null_entries)=rand(2,size(null_entries,2));

    % Calculating the normal vectors (aka the directions) towards the
        other agents
    e_beta_matrix=r_alphabeta_matrix./(ones(2,1)*sqrt(sum(
        r_alphabeta_matrix.^2)));


    % d_direction: Desired Direction
    % Read the desired direction from the Fast March Gradient
    d_direction = [e_alpha_x(round(agent_alpha(2)),round(agent_alpha(1)
        ),agent_alpha(6));...
        e_alpha_y(round(agent_alpha(2)),round(agent_alpha(1)),
            agent_alpha(6))];

    % Queueing
    % Select all agents around alpha that have smaller time till
        reaching the goal.
    closer_agents = agent_others([1 2 8],((agent_others(8,:)<
        agent_alpha(8))&(agent_others(6,:)==agent_alpha(6))));

    % Exclude the agents towards the first goal (aka cash point)
    % They should not queue as the cassa is not designed like it
    if((agent_alpha(2)<200)&&(size(closer_agents,2)>0)&&agent_alpha(6)≠
        1)
        % Of these take the one that has the longest
        % expected time as the new desired direction.
        [¬,I] = max (closer_agents(3,:));
        closest_agent = closer_agents(1:2,I);
        % Mix the desired direction with the queueing direction
        d_direction = (closest_agent-agent_alpha(1:2))/norm(
            closest_agent-agent_alpha(1:2),2);
    end

    % Relaxion term
    % If the agent already has its desired speed
    % and direction, no force comes from this term
    F_tot = 1/tau_alpha*(...
        v0_mean*((agent_alpha(6)==1)+1)*d_direction/norm(d_direction,2)
            ...
        -agent_alpha(3:4)...
        );

    F_agents =  A2*sum(...
                (ones(2,1)...
                *exp((2+tray_factor*(agent_alpha(6)==1)...and because
                    of the tray the have a double radius
                *sigma*ones(1,agent_number_back-1)-sum(
                    r_alphabeta_matrix.^2)+sigma*tray_factor*(
                    agent_others(6,:)==1))/B2)...
                ).*e_beta_matrix,2);
            F_tot = F_tot + F_agents;

end
```

7.1.2 Count Passes | `count_passes.m`

```
% Find Agents on counter areas

for agentID = 1:size(A,2)
   X = round(A(2, agentID)); Y= round(A(1,agentID));

   a_counter = X_counter(X, Y);

   if (  a_counter ≠ A(7, agentID) ... % agent was not on the area yet
       & a_counter ≠ 0 )                % agent is indeed on an area

       passes( a_counter ) = passes ( a_counter ) + 1; % boah, matlab
           !!!
       A(7, agentID) = a_counter;
   end

end

% das geht auch anders:

% a_pos = round(A(1,:)) + size(A,2) * round(A(2,:)); % liste der pixel
    der a.
%
% a_counter = X_counter( a_pos ); % greife auf elemente zu
%
% a_counter( a_counter == A(7,:) ) = 0; % wirf besuchte weg
%
% for c = 1:n_counters % auszählen: über counter it.
%     passes(c) = passes(c) + sum( a_counter == c );
% end
%
% % overwrite last area-value where a new counter was detectet
% A(7, a_counter ≠ 0) = a_counter( a_counter ≠ 0);
```

7.1.3 Init Agents | init_agents.m

```
function Anew=init_agents(agentID,A)

    % DESCRIPTION:
    % This is a function which initializes the Agentsmatrix. The
        agentmatrix
    % holds all required agent information. The structure of the matrix
         is as
    % follows:
    %                    | Agent 1   | Agent 2 | Agent 3
    %------------------------------------------------
    % 1 position x      |
    % 2 position y      |
    % 3 speed    x      |
    % 4 speed    y      |
    % 5 desired speed v0|
    % 6 type            |
    % 7 last counter    |
    % 9 red-carpet-time | (indicates the frame when the agent actually
    %                      left the red area; thus "entered" the mensa)
    %
    % PARAMETERS:
```

```
% A                  = agent Matrix
% agent_number      = global constant, total # of agents at any
    time
% map               = matrix containing zeros/ones legal/nonlegal
    positions
% legal_pos         = all legal positions ordered
% shuffled_legal_pos= all legal positions shuffled


parameters; % load global parameters

global agent_number v0_mean sqrt_theta map_init n_goals;

% check if agentID is set, if so reinitialize only this agent and
    leave
% the rest of the A matrix alone

a_num=agent_number;
if(nargin≠0) % if inputargument given, only one agent
    a_num=1;
    if(A(6,agentID)≠1)
        A(6,agentID)=1;
        Anew = A;
        return
    end
end

map = map_init';
%map(140:175,100:300)=1;


%map = ones(300,300);
% find legal x and y positions on map
%[row,col,v] = find(X, ...) returns a column or row vector v of the
     nonzero
%entries in X

[row, col, ¬]=find(map);


% all legal positions ordered
legal_pos=[row col];

% random order positions
rand_index=randperm(size(legal_pos,1));
shuffled_legal_pos=legal_pos(rand_index,:);


Anew=shuffled_legal_pos(1:a_num,:)';

% generate gaussian distributed v0, the desired speed and initial
    speed of
% an agent

v0=normrnd(v0_mean,sqrt_theta,1,a_num);

%random unit direction
```

```
    agent_directions=[sin(rand(1,a_num)*2*pi);cos(rand(1,a_num)*2*pi)];

    % adding speeds to A

    agent_speeds = agent_directions.*(ones(2,1)*v0);

    % add v0 and agent speeds to A


    Anew(3:4,:)=agent_speeds;
    Anew(5,:)=v0;
    % Random Goal from 2 until n_goals, the first goal is the cash
        point
    %Anew(6,:)=randi(n_goals-3,1,a_num)+randi(2,1,a_num)+1;
    %1=Bio
    %2=Menu1
    %3=Spezial
    %4=Salat
    %5=Vegi
    Menus = [ 1 2 2 2 2 3 3 4 5 5 ];
    Anew(7,:) = 0;
    for i = 1:a_num
        bla = randi(size(Menus,2));
        Anew(6,i)= Menus(bla)+1;
    end
    Anew(9,:) = -1;

    if(nargin≠0)
        A(:,agentID)=Anew;
        Anew=A;
    end

end
```

7.1.4 Parameters

```
%%%%%%%%%%%%%%%%%%%%
% GLOBAL PARAMETERS
%%%%%%%%%%%%%%%%%%%%

global ...
    dt ...
    meter...
    hue_goal ...
    hue_init ...
    hue_counter ...
    wall_th...
    map_file ...
    R ...
    v0_alphabeta ...
    sigma ...
    agent_number...
    v0_mean...
    tau_alpha...
    U_alphaB_0...
```

```
    sqrt_theta...
    noplot...
    video_on...
    log_on...
    duration...
    A1...
    A2...
    B1...
    B2...
    sight...
    tray_factor;

% General

agent_number = 150;
dt = .1;
noplot = false;
video_on = true;
log_on = true;
duration = 4000; % frames

% Map
meter = 15; % px/m, according to:
map_file = 'grundrissplan4.png'; % Bitmap file for goals and walls

% Map colors  % Value / Tolerance
hue_goal = [0.3 0.1]; % green
hue_init = [0.0 0.1]; % red
hue_counter = [0.1 0.1]; % yellow

wall_th = 0.2; % less than 20% grey is ignored and counts as free space

% Boundary potential
R = 0.05 * meter; % m, according to paper
U_alphaB_0 = 5 * meter^2; % m^2/s^2, max. boundary potential

% Agents
v0_alphabeta = 2.1*meter^2 * 10; % m^2/s^2, max. agent potential
sigma = .3*meter; % m
v0_mean = 1.34 * meter; % this is the mean of the desired speed of an
    agent
              % m/s equals v_alpha_0 in formula (2)
tau_alpha = .3; % s, "relaxation time"
sqrt_theta = 0.26*meter; % m/s standard deviation of gaussian
    distributed v0_mean

A1 = 0*meter;
B1 = 0.2*meter;
A2 = 3*meter; % m/s^2
B2 = 10*0.2*meter; % m

sight = 1.5*meter;
tray_factor = 1;
```

7.1.5 Plot Stat | plot_stat.m

```
% Structure of A_stat:

%              Agent 1 | Agent 2 | ...
%             _________|_________|____
% 1 Velocity |\
% 2 Crowd    |  \
% 3 Goal     |    \
% 4 w-time   |      \
%            |        \ 3rd Dimension: frames

loglen = nnz( sum( sum(A_stat, 2), 1) );

global fetchtimes agent_number;

% to improve: nur agenten, die den roten bereich verlassen haben,
    beachten!
%% speed graph

speeds = permute( A_stat(1, :, 1:loglen), [3,2,1] );

hold on;

plot( speeds, 'k' );
s_avg = plot( sum( speeds, 2)/size(A_stat, 2) , 'r' , 'LineWidth', 2);
legend(s_avg, 'Average speed');

set(gca,'FontSize',16)
xlabel('time [frames]');
ylabel('speed [px/frame]');

% des sagt uebrigens rein gar nichts aus, nicht verwenden!

%% spatial density graph

image( X_traces , 'CDataMapping','scaled' ); axis image;

imwrite(X_traces/20, colormap('jet'), 'dens.png');

%% stuck agents


bl = zeros(duration);
for t = 1:duration
    a_stat = A_stat(:,:,t);
    a_stat( :, a_stat(4,:)==0 ) = [];
    bl(t) = sum( a_stat(1,:) < 5 );
end

plot(bl(1:10:end, 1:10:end));
set(gca,'FontSize',16)
xlabel('time [frames/10]');
ylabel('stuck agents');


%% density graph

% dens = permute( A_stat(2, :, 1:loglen), [3,2,1] );
```

```
%
% hold on;
%
% plot( dens, 'k' );
% d_avg = plot( sum( dens, 2)/size(A_stat, 2) , 'b' , 'LineWidth', 2);
% legend(d_avg, 'Average density');
%
% set(gca,'FontSize',16)
% xlabel('time [frames]');
% ylabel('density [neighbours within 1m]');


avg = zeros(duration);
for t = 1:duration
    a_stat = A_stat(:,:,t);
    a_stat( :, a_stat(4,:)==0 ) = [];
    avg(t) = sum( a_stat(2,:) )/ size(a_stat, 2);
end

plot(avg(1:50:end, 1:50:end));


%% walkingtime-dependend

figure();
hold on;
for id = 1:agent_number
    a = permute(A_stat(:, id, :), [1,3,2]);
    a( 2, a(4,:) == 0 ) = Inf;
    plot( a(4, :), a(2,:), 'Color', rand(3,1)); % dens / wt
end

%% fetching time graph

dft = fetchtimes(2,:)-fetchtimes(1,:)

subplot(2,1,1);
hold on;

set(gca,'FontSize',16)
plot(fetchtimes(2,:), dft, '.k');


ylabel('pass-through-time');
xlabel('arrival time');

subplot(2,1,2);

set(gca,'FontSize',16)
hist(dft, 30)
ylabel('frequency');
xlabel('pass-through-time');
```

7.1.6 Potential Force | potential_force.m

```
function f = potential_force( x, y, goal_layer )
```

```
% This function might not really be necessary, but serves as an example
     how
% to use the results of load_map.m

    global fields_x fields_y;

    f = [fields_x(y,x,goal_layer);
         fields_y(y,x,goal_layer)];

end
```

7.1.7 Refresh Fields | refresh_fields.m

```
% Fast marching algorithm

f = v0_mean / tau_alpha; % see formula (2) in paper

for i = 1
    X_fm(:,:,i) = 0.00001 + X_walls(:,:,i)*0.9;

    X_people = 0*X_walls(:,:,i);
    a_pos = round(A(2,:)) + (map_x) * round(A(1,:)-1); % liste der
        pixel der a.
    X_people( a_pos ) = 1;

    R_p = 5 * R;

    g_people = exp( -sqrt( (k-k_0).^2+(l-l_0).^2 )/R_p );

    X_people_conv = conv2(X_people, g_people, 'same');

    X_fm(:,:,i) = 2*X_fm(:,:,i) - X_people_conv;


    addpath fm/;
    [t_x, t_y] = find(X_goals(:,:,i) == 1); % Create list of target-pxs

    [T(:,:,i), Y] = msfm(X_fm(:,:,i), [t_x t_y]'); % Do the fast
        marching thing

    [e_alpha_x(:,:,i), e_alpha_y(:,:,i)] = gradient(-T(:,:,i));
    r = sqrt( e_alpha_x(:,:,i).^2 + e_alpha_y(:,:,i).^2 );

    r( r==0 ) = inf;

    e_alpha_x(:,:,i) = e_alpha_x(:,:,i)./r;
    e_alpha_y(:,:,i) = e_alpha_y(:,:,i)./r;
    % Now we've got the fields for the desired direction, e_alpha.
%
%      %fields_x(:,:,i) =  field_walls_x(:,:,i);% + f*e_alpha_x(:,:,i);
%      %fields_y(:,:,i) =  field_walls_y(:,:,i);% + f*e_alpha_y(:,:,i);
    % Scale & sum fields
end
```

7.1.8 Remote Worker | remote_worker.m

```
% remote worker

reset_sim;
simulate_v1;


sim_id = ['test' datestr(now)];
save('sim_id');
```

7.1.9 Reset Simulation | reset_sim.m

```
clear all;
clear global;

parameters;
load_map;
A = init_agents();
```

7.1.10 Separate Areas | seperateAreas.m

```
function [areas, n] = seperateAreas( pattern )
%SEPERATEAREAS Summary of this function goes here
%   Detailed explanation goes here

    CC = bwconncomp( pattern );
    areas = [];
    n = CC.NumObjects;

    for i = 1:n % for every found component in the goal-layer
        layer = pattern*0;
        layer(CC.PixelIdxList{i}) = 1;
        areas(:,:,i) = layer; % add a layer with it to X_goals
        %spy(layer, 'MarkerFaceColor', 'r');
    end

end
```

7.1.11 Simulate | simulate_v1.m

```
%% INIT

%clear all;
clear global;

parameters();
load_map();
global dt agent_number statistic agents_f p_gain fetchtimes; %X_goals;
A=init_agents();

if(video_on)
vidObj= VideoWriter(['videos/foodtrail ' datestr(now) '.avi']);
```

```
open(vidObj);
end

%% STATISTICS

n_through = 0;
fetchtimes = [];
cpu_a = 0;
X_traces = X_walls(:,:,1)*0;

if (log_on)
    A_stat = zeros(3, agent_number, duration);
end

%% Simulation Loop
timestep=dt;

%my_figure = figure('Position', [20, 100, 600, 600], 'Name','Simulation
     Plot Window');

for stepnumber=1:duration

%timestep = stepnumber^-.2
% Calculate the Forces
% Calculate the resulting velocities ?
agents_f = zeros(2,agent_number);
agents_p = zeros(2,agent_number);
for agentID = 1:size(A,2)
    A(8,agentID) = T(round(A(2,agentID)),round(A(1,agentID)),A(6,
        agentID));
    agents_f(:,agentID) = agents_force(A,agentID);
    [agents_f(:,agentID), nb] = agents_force(A,agentID);
    agents_p(:,agentID) = potential_force(round(A(1,agentID)),round(A
        (2,agentID)),A(6,agentID));
    A(3:4,agentID) = (agents_p(:,agentID)...
        +1*agents_f(:,agentID)...
        +10 *[(rand(1)-.5);(rand(1)-.5)])...
        *timestep;
    if (log_on) % record density around agent
        A_stat(2, agentID, stepnumber) = nb;
    end
end

% log velocities and goals and walkingtimes
if (log_on)
    A_stat(1, :, stepnumber) = sqrt(A(3,:).^2+A(4,:).^2);
    A_stat(3, :, stepnumber) = A(6,:);
    wtimes = stepnumber - A(9,:);
    wtimes(wtimes == stepnumber+1) = 0; % erase not-yet-started ones
    A_stat(4, :, stepnumber) = wtimes;

    % create "tracemap"


end

%Find Agents that exceed their max velocity
```

```
too_fast=find(sqrt(A(3,:).^2+A(4,:).^2)>A(5,:));
nan = (isnan(A(3,:))|isnan(A(4,:)));
A(3,nan) = 0; A(4,nan) = 0;
num_toofast = size(too_fast,2);
%too_fast_x=find(abs(A(3,:))>A(5,:));
%too_fast_y=find(abs(A(4,:))>A(5,:));

% Throttle them to their desired velocity
A(3:4,too_fast)= A(3:4,too_fast)./([1 1]'*sqrt(A(3,too_fast).^2+A(4,
    too_fast).^2))...
   .*[A(5,too_fast); A(5,too_fast)];

%A(3,too_fast_x) = A(5,too_fast_x).*sign(A(3,too_fast_x));
%A(4,too_fast_y) = A(5,too_fast_y).*sign(A(4,too_fast_y));

% Calculate the new positions (X+V*t)
ΔPos=A(3:4,:)*timestep;
A(1:2,:)=A(1:2,:)+ΔPos;

% Find Agents that exceed the boundries
A(1, A(1,:)<1 ) = 1;
A(1, A(1,:)>map_y ) = map_y;

A(2, A(2,:)<1 ) = 1;
A(2, A(2,:)>map_x ) = map_x;

% Find Agents on target areas
for agentID = 1:size(A,2)
   X = round(A(2, agentID)); Y= round(A(1,agentID));
   %            (X,             Y,            Target layer);
   % Find Agents on target areas
   if ( X_goals(X,Y, A(6, agentID) ) )
       % agent reached his target
       %A(1, agentID) = randi(300,1,1);
       %A(2, agentID) = randi(300,1,1);

       if ( A(6, agentID) == 1) % agent past kassa?
           % pass-specific statistics are done here
           fetchtimes = [fetchtimes [A(9, agentID); stepnumber]];
           n_through = n_through + 1;

           A = init_agents(agentID,A); % restart agent
       else
           A(6, agentID) = 1; % shoo him to the kassa!
       end
   end

   % Find Agents leaving the red init area
   if ( A(9, agentID) == -1 && X_init(X,Y) == 0)
        A(9, agentID) = stepnumber;
   end
end


% Draw the the agents

% Statistics
```

```
count_passes;

fps = 1/(cputime - cpu_a);
cpu_a = cputime;

statistic = {'Durchgaenge:',[passes], '', 'Zu schnell:', num_toofast,
    ...
            'Gefuettert:', n_through, 'fps', fps,...
            'Frame', stepnumber };

% draw

if (noplot)
    [fps stepnumber duration]
else

clf();

if (¬video_on)

subplot(1,2,2);
image( X_fm(:,:,1) , 'CDataMapping','scaled' ); axis image;
hold on;

fieldplot = 1; % e_alpha-field to plot next to the simulation
space_x = floor(linspace(1,map_x, 50));
space_y = floor(linspace(1,map_y, 50));
quiver(space_y, space_x, ...
       e_alpha_x(space_x, space_y, fieldplot), ...
       e_alpha_y(space_x, space_y, fieldplot), 0.5);

subplot(1,2,1);
end
imagesc( map_pretty );        %Hintergrundbild laden
colormap('bone');
hold on;
 draw_field = 1;
plot(A(1,A(6,:)==1),A(2,A(6,:)==1),'o','MarkerSize',sigma/2,'
    MarkerEdgeColor','r','MarkerFaceColor','r'); %Punkte zeichnen
plot(A(1,A(6,:)≠1),A(2,A(6,:)≠1),'o','MarkerSize',sigma/2,'
    MarkerFaceColor','b'); %Punkte zeichnen
axis image;

annotation(figure(1),'textbox',...
    [0 0 0.1 0.5],...
    'String', statistic ,...
    'FitBoxToText','off');

pause(0.01);

if (video_on)
currentFrame=getframe;
writeVideo(vidObj,currentFrame);
end

end %noplot
```

```
if (mod(stepnumber, 10) == 0)
    refresh_fields;
    X_traces = X_traces + X_people_conv;
end

end

if (video_on)
close(vidObj);
end
```

7.1.12 Test Agents Force | test_agents_force.m

```
%% File to test the agent_force.m

parameters; % load global varibales

A=rand(5,100);

for i=1:size(A,2)
    F_agents=agents_force(A,i);
end
```

7.1.13 Test Load Map | test_load_map.m

```
% Which maps shall be drawn?

%draw_these = 1:n_goals;
draw_these = [1];


parameters;
load_map;

global fields_x fields_y n_goals v0_mean tau_alpha map_pretty map_x
    map_y;

n_plots = size(draw_these,2);

space_x = floor(linspace(1,map_x, 400));
space_y = floor(linspace(1,map_y, 400));

f = v0_mean / tau_alpha;

for i = draw_these;

    field_x = fields_x(:,:,i);
    field_y = fields_y(:,:,i);

    subplot(1,n_plots,find(draw_these == i));
    hold on;

    image(map_pretty);
    quiver(space_y, space_x, ...
```

```
            e_alpha_x(space_x, space_y,i), e_alpha_y(space_x, space_y,i)
                ,...
        'Color','g');
    quiver(space_y, space_x, ...
            field_x(space_x, space_y), field_y(space_x, space_y),...
            'Color','b');

    quiver(space_y, space_x, ...
            field_x(space_x, space_y)+f*e_alpha_x(space_x, space_y),...
            field_y(space_x, space_y)+f*e_alpha_y(space_x, space_y),...
            'Color','r');

end
```