Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Project Report

Pedestrian dynamics in long, narrow hallways

Moser Manuel, Suter Yannick & Theiler Raffael

Zurich
December 2012

Agreement for free-download

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

Moser Manuel Suter Yannick Theiler Raffael

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of Originality

This sheet must be signed and enclosed with every piece of written work submitted at ETH.

I hereby declare that the written work I have submitted entitled

Pedestrian dynamics in long, narrow hallways

is original work which I alone have authored and which is written in my own words.*

Author(s)

Last name	First name
Moser	Manuel
Suter	Yannick
Theiler	Raffael

Supervising lecturer

Last name	First name
Donnay	Karsten
Balietti	Stefano

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (http://www.ethz.ch/students/exams/plagiarism_s_en.pdf). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date

Signature

*Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Print form

Contents

1 Abstract

Authors: Manuel Moser, Yannick Suter, Raffael Theiler
Title: Pedestrian dynamics in long, narrow hallways

In this project, we want to have a closer look at pedestrians in narrow hallways, motivated by a situation at Zurich main station. To do this, we simulate pedestrians by agents with Matlab who walk according to some rules. We managed our agents to pass each other by, to look ahead a few meters and to decide where to walk next. At first, we wanted to have a closer look at the pedestrian flux when the number of persons per minute entering is increased, when there are clearly more people moving in the same direction against a few in the other direction. Next, we wanted to analyze the influence of aggressive fast people in a rush, slowly moving obstacles and the influence of drunkard (randomized walking) on the pedestrian flux.
But soon, our attention turned more to building/creating everything on our own and less about a fast simulation of different situations.
The main outcome of our simulations was that pedestrians tend to get stuck or create jams as soon as there are lots of people trying to pass the same hallway. The exact same hallway can work smoothly if there are not too many people, but jams can arise quickly. And once a jam starts, it often spreads out because people have to stop and walk slower.
Finally, we arrived at the following conclusions: To improve the pedestrian flux, broadening a hallway is a superb solution. Therefore, at our situation scene at Zurich main station, it would be best if the hall users would try to leave more space for the pedestrians, and if the food shops Imagine and Nordsee wouldn't have chairs and tables outside.

2 Individual contributions

Manuel:	coordination, report, revising & graphics/plots
Yannick:	logical functions, revising, debugging & matlab overview
Raffael:	graphical functions, visualization & agents

Listed above are only the main tasks everyone of us took care of, but we shared most of the work. The github commit report is not always mirroring the work behind those uploads because we often worked together on the code, debugged, commented and worked on the documentation while swapping laptops.

3 Introduction and Motivation

3.1 Motivation

As many commuters too, we all got annoyed by people rushing through the small corridor left in the Zurich main station hall (the path between burger king and groups meeting point) during the Oktoberfest, market days, concerts and other occasions when going from the platform, passing the meeting points in direction towards the Central and further on. So when thinking about a simulation project, this quickly crossed our minds.
In comparison to other pedestrian simulations, our simulation is special in different ways: this situation of a narrow but long corridor implies that there's not much space to avoid other pedestrians to the side, especially if there are lots of other people, whereas there is no need of path optimisation as the ultimate goal is to cross the hallway without crashing into someone else.
Being sometimes in such a situation inside that small corridor, we observed very special dynamics evolving between people rushing by, people just strolling around and people who want to talk to each other and so on.

3.2 Fundamental Questions

The fundamental questions of our simulation project we started with were:

- We try to simulate the pedestrian flux of agents in a hallway in the following different situations: Rush hour (danger of jamming), much more agents in one direction than in the other, with an static obstacle (if possible), with aggressive/very fast or slow agents, random path agent (drunkard). Will the pedestrian flux run smoothly or will they block each other and be stuck?
- Will the implementation of a rudimentary kind of thinking/looking ahead help to avoid blockages? If possible, we may determine the limits for which the goal of passing is achieved with and without this implementation and compare them.
- Will there be group dynamics or similar behaviours of agents even if they're only programmed to walk to the other side, each on his own?

As soon as the programming phase started, we realized there was a major point of importance about this work we all were aware of, but had forgot to think about beforehand. We did not want to start with a simulation already created, but build something "new" on our own. So we started off by creating our logic functions that would allow the agents to avoid crashing into other agents, then turned to the graphical part. As a consequence, new fundamental questions arose:

- Starting with the idea of looking ahead, how can we turn this into a rudimentary kind of intelligence?
- How can we have a look at the main station situation but keep our agents as random as possible?
- Using our own kind of rudimentary intelligence, will a form of collective behaviour follow? Will we be able to remodel the dynamics that appear when people start to rush forwards while other are considerable slower?

3.3 Expected Results

We thought that there would be lots of changes in the direction of walking to the left and right while trying to avoid other agents. With rising amount of agents we expected more jams, although this seems obvious. We thought that in our simulation we would have to deal with massive jams because the agents were not communicating with each other in any way. Our implementation of "looking ahead" as a form of intelligence would probably improve the people flux but only to a limited range. The formation of columns of people as a way to avoid the zig-zag-pattern of the agents' ways might appear.

4 Description of the Model

4.1 General considerations

As we wanted to describe a situation with people, we chose a model based on discrete agents. They should be able to walk freely along their path until they hit an obstacle, in which case they should be able to determine a way to avoid the obstacle. Obstacles could be walls, objects placed in the path or simply other agents. The main goal of any agent is to get to the other side of the path as quickly as possible, if possible without crashing into other things. As the global situation changes with each "step" an agents does and also with the appearance of new agents, a step-by-step iteration was chosen to propagate the situation in time in which the the optimal direction is calculated all the time. The other approach of calculating a certain path from start to finish was rejected as it probably cannot be done for the uncertainties mentioned above, namely the random appearance of new agents which could block the calculated path. Also, this is not a main point for our situation because there's not a specific destination an agent walks to, but a intended zone to go to. In a hallway without exits on the sides, the goal only is to traverse it.

Any agent's goal is to get to the other side as quickly as possible, although our model cannot accommodate the requirement of the quickest possible path. Because of the step-by-step iteration, any situation is repeatedly analyzed and (hopefully) the best way to proceed is chosen. As this is only a short time period and we only consider an agent's situation in a local environment, it cannot guarantee to give the best outcome overall. To make a long story short, we used a *local search algorithm* combined with a greedy algorithm for each agent.

4.2 Walls and other static obstacles

If a narrow hallway should be a narrow hallway, it needs two walls. Although this statement is obvious, it can be implemented in various ways. We chose to use static agents with a small radius to act as a wall, as it allows the creation of many different hallways. They can also be used as static objects representing obstacles like chairs and tables which could stand in the path an agent has to follow to get to the other side. For our simulation, this was a very flexible way which also allows a quick adaption to other situations if necessary.

4.3 Agents

The basis of the simulation is the agent. An agent should represent a person in real life. We assumed that the hallway was not a place to linger about, therefore they should try to get to the other side as quickly as possible. In order to do so, they need to be aware of all the things around them that they might bump into. This was the origin of the thought that any agents only looks forward as the obstacles will be in the path before them. In our model, they also need some intelligence to get around an obstacle and avoid running into other agents. To do so, the agent should consider all things within a circle of defined radius in front of him while everything else doesn't bother him. Again, this tries to get a local solution to our problem in the hope that the overall solution is still a good one. As one usually doesn't walk backwards, our agents only walk forwards. This might not be true for all situations but for most of the situations a pedestrian walks into, and it simplifies the model considerably. We also think that, if a pedestrian does only walk more or less forwards and won't be involved in a jam or collision, his passing time will be quite good compared a perfect chosen path, if a such path even exists.

In comparison to previous models which used a force field to guide the agents, we thought that this approach is more realistic as the force field approach given that the force field already defines the optimal solution one is looking for. It should also prove to be more adaptable within a reasonable time frame to other problems.

4.4 Intelligence

4.4.1 General considerations regarding the necessary intelligence

For the model to work properly, a sensible solution to deal with the problem of finding the right path must be found. We do not claim to have found the best solution but believe our model to be a fairly good solution.
The presence of every other agent, be it agent or wall, inside a semi-circle in front of the currently considered agent clearly must have an influence in order not to bump into it. The fundamental idea is to get a function for every agent inside this agent's semi-circle which reproduces the effect it has on the path the considered agent would take without them being in his semi-circle. All of these functions will then be added and the maximum value would be taken as the best way to go forward. An additional function is added which should represent the tendency to go straight forward.

All functions were calculated as functions of x. This x is connected to an angle φ given by $arctan(x)$. A value of $\varphi = -\pi/2$ would correspond to walking sideways to the left, $\varphi = 0$ to walk straight forward and $\varphi = \pi/2$ to walking sideways to the right, always viewed in the direction the agent wants to go.
A function calculated in x will then be superimposed on φ which corresponds to a mapping of the function onto a polar grid. A possible disadvantage of this procedure

is that φ does not range from $-\pi/2$ to $\pi/2$ and therefore doesn't reach the full semi-circle as x is bounded and doesn't reach infinity. We consider this to be negligible as for reasonably high values of x, φ gets close to $\pm\pi/2$ and it is rarely in the interest of an agent to walk purely sideways.
We chose this way is it seemed easier and more intuitive to handle x-values for many different situations. It allowed an easier and (hopefully) more understandable way to the functions used to model various influences.
For an agent coming from the other side, we used a function of the type

$$y(x) = \frac{1}{(|x - x_{\text{Agent}}|)^n} \tag{1}$$

and set the y values corresponding to x values on a collision course to zero. The exact implementation will be treated in chapter *Implementation* where equation (1) is modified to accommodate for various different parameters. If an agent has another agent in front of him which is walking in the same direction but slower, the function given in (1) is also applied to overtake the slower agent.
Should another agent walk in the same direction, but with higher speed, we thought that it would be logical to follow him. This was done using a Gaussian curve centered on the direction of the faster agent.
Agents moving in the same direction with the same speed have no influence on each other in our model. Two agents standing still on the other hand will also be handled using the function given in (1) to avoid standoffs.

4.4.2 Necessary parameters and variables

We reckon that the important parameters necessary to assess the influence of one agent onto the other agents are

- α_X, the angle between the centers of the two agents with respect to the y-axis
- Δv, the difference in velocity, set to be negative for this situation
- β_{Links} and β_{Rechts}, two angles describing which directions of the path to be chosen would lead to a collision.
- $r_S = r_1 + r_2$ being the sum of the two involved agent's radii
- d, the distance between the two involved agents

In figure 4 (given in the implementation chapter) a graphical depiction of α_X, β_{Links} and β_{Rechts} is given alongside the way to calculate them.
The sign of Δv was chosen to be positive in the case of two agents walking in the same direction and negative in the case of crossing agents.

4.4.3 Collision detection and the its connection to the iteration speed

Collision detection has to be carried out in order to keep the simulation physically possible. We used an approach in which the vector describing the path the agent walks on is subdivided into several small steps. The agent then goes on as far as he can. To avoid the agents sticking too close together and therefore not being able to move anymore, the number of subdivisions should be rather small.
Given that the length of the vector is determined by the iteration time intervall Δt, it is also important for Δt to be not too small. This has a phyiscal representation as in reality one goes on in discrete steps, not infinitesimally small steps that one would get in the case of $\Delta t \to 0$.

4.5 Discretization

As it is a numerical calculation, at some point there has to be a numerical discretization. Since we wanted the agents to be able to move around freely, we abandoned the idea of using a grid on which the agents could walk around in favour of a continuous space simulation. The discretization comes about in the form of the functions and angles to be calculated.

4.6 Spawning of new agents

A hallway usually has at its two endings a more open space than the hallway where the flow of people arriving from the hallway will disperse. In order for things not to get too messy, we decided that agents would simply cease to exist if they reach a line (called dstruction line) which would signify the beginning of this region of dispersion. Newly spawned agents would be created on a line (called spawn line) below the destruction line and pass a short distance without oncoming agents so that they could already orientate themselfes according to the flow of agents walking the other way. This could be important in cases where the agents would form columns and newly spawned agents could adapt to that by not interrupting the column.

5 Implementation

5.1 General considerations

5.1.1 Introduction

We had to build our implementation around several different desires:

- The code has to be expandable with additional features.
- If there is a class model, it has to be simple because the project is not that big.
- Constants should be easy to find and adjust.

According to those points, the simulation code grew to be a mixture between classes and cascaded functions. At the top is a file defining all the global constants called *defineConstants.m*. This approach is maybe not very correct in terms of a good programming style, but it makes it very simple to adjust smaller details and to keep an overview over all the constants necessary to make the model work and to specify the field the agents walk in. It also provides an easy way to store and therefore document each run by simply saving the file containing all constants to a text file. All constants and variables are considered to be in SI-Units, if not mentioned otherwise. This means that all position specifications are given in meters, the velocity in meters per second and so on. Of course, this is only an approximation of reality, but it makes comparisons possible and easy.

There are 3 classes implemented: *simulation.m*, *agent.m* and *drawing.m*. A further description about these classes can be found in the table beneath or directly in the header of the class files. The rest of the code is split up into different logical functions.

Class	Description	Parent
simulation.m	This class offers all the functions for simulations. It can be executed with different parameters depending on the users preferences.	Matlab "handle" class
agent.m	This class is a container for all the agent data such as its radius, velocity and position.	Matlab "handle" class
drawing.m	This class can draw a field containing agents.	Matlab "handle" class

The class diagram given in figure 1 is a summary of the most important functions and properties in this project.

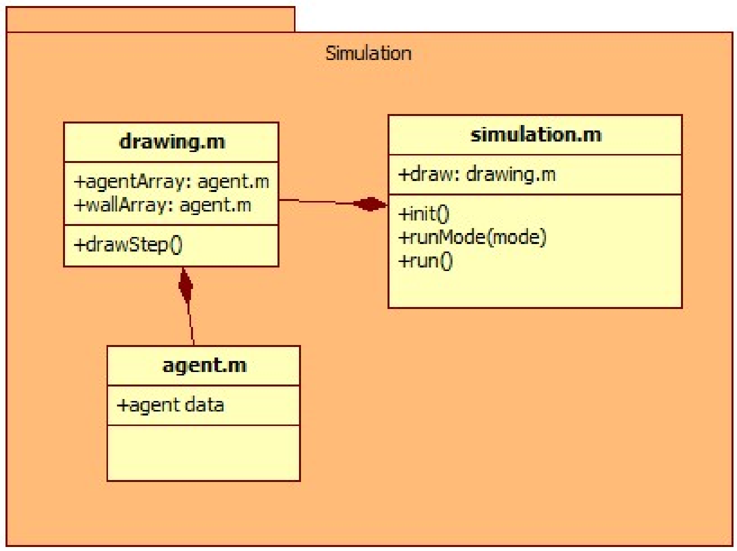

Figure 1: Class diagram of our model. From bottom to top we implemented a class *agent.m*, *drawing.m* and *simulation.m*.

5.1.2 Important classes to run the simulation

The simulation class *simulation.m* describes an object that wraps all the different possibilities of our simulation program. It's the starting point for a new simulation, runs this simulation and collects all the data requested from the agents and the simulated environment. The drawing class *drawing.m* handles all graphical aspects of our simulation. The agent class *agent.m* has a subchapter of its own because it fulfills a very different role than the two classes mentioned previously.
The main flow of a run cycle is described in the activity diagram given in figure 2.

Important properties of the simulation class *simulation.m* are:

- DRAW: The *drawing.m* implementation.
- RESULT: A matrix containing simulation results.

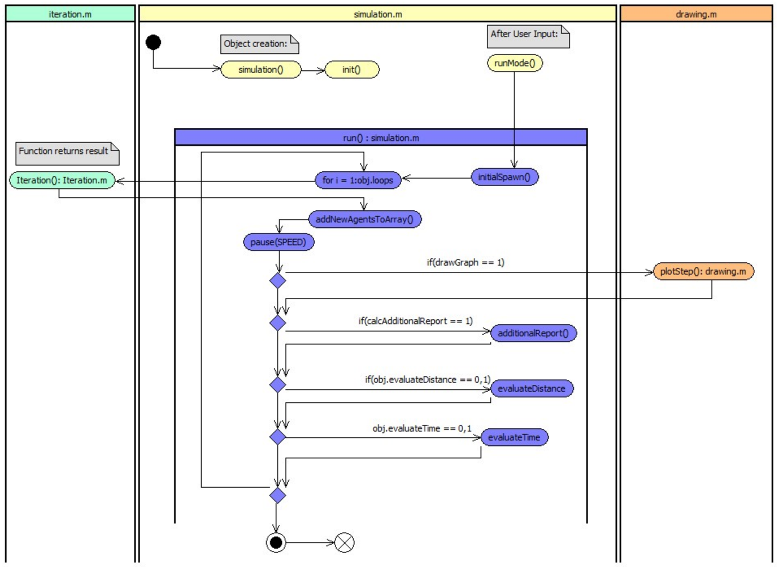

Figure 2: Activity diagram of our model.

- additionalResult: A matrix containing further simulation results.
- evaluateTime: A vector used to evaluate the time an agent has spent in the simulation during its existence.
- evaluateDistance: A vector used to evaluate the distance an agent has covered in the simulation during its existence.

All methods of the simulation class are listed here:

- simulation(): Constructor, sets up the object.
- init(): Initializes the existing object: Fills up arrays with empty objects, sets up vectors, etc.
- runMode(): Pass console like parameters to this function to run a simulation and choose between various runmodes.
- additionalReport(): Fills the additionalResult matrix with data.
- calcPossibleAgents(): Calculates the maximum possible number of agents for the set area.

- randPrefix(): Randomly generates -1 or 1 to set the spawnpoint (top or bottom), if wanted to. It was replaced by spawning probabilities for top and bottom.
- initialSpawn(): Fills up the agent array with new *agent.m* objects. They have priority 0 which corresponds to non-existing agents.
- addNewAgentsToArray(): Used to add new agents to the simulation while running.
- balanceProbability(): Balances the probabilities to spawn agents, if wanted to. It was replaced by spawning probabilities for top and bottom with given constant probability.
- run(): Main function to run a simulation after everything is set up and ready. This function should not be executed directly by the user. New simulations should be started with the runMode() function.

See also the activity diagram (figure 2) for a better overview on how these method functions are used in the simulation.

The drawing class *drawing.m* basically adapts simple Matlab drawing functions and converts them into useful functions for this project. As a result it's very easy to draw the new situation after a simulation step by simply calling the method plotStep().

5.1.3 Properties

Important properties are:

- particleDensity: Resolution for wall *agent.m* objects in $\frac{\text{agents}}{\text{meter}}$.
- width: The field width.
- length: The field length.
- wallArray: All the agent.m objects for the wall.
- agentArray: All the agent.m objects for the simulation.

5.1.4 Methods

All methods are listed here:

- drawing(): Constructor, sets up the object.

- createWall(): Creates wall agents according to the settings.
- plotStep(): Main function, plots all the agents on a field with the walls and the starting lines.
- drawWallSquares: Draws the walls on the side.
- circlePlot: Draws circles for the agents.
- drawLine: Draws a line with coordinates. Used for the direction indicators etc.

How the field is created is the subject of subchapter 5.4.

5.1.5 Simplifications

Some simplifications and constraints on the model had to be introduced during the implementation to keep the whole simulation manageable. At first, we decided that the agents should walk either up or down and used the sign of an agent's velocity as an indication in which direction the agent goes. In the logic functions, a discretization had to be introduced for the numerical evaluations of functions. To inter-convert values between different vectors, the function *closest.m* was used.
As a consequence of the model used for the logic functions, the agents will only be able to look forwards. Because of the actual way *logicFunction.m* was implemented (see subchapter 5.5), they would not use the full $\pm 90°$ in front and on the side of them, but almost. We thought this not to be too critical as the human visual field is also smaller than $180°$, if one doesn't move the head.
If one walks through the main station, it is visible that many people are not on their own. Groups of people tend to walk together as they want to chat. This can be seen very clearly in couples which try to walk side-by-side. To incorporate this in our model, we would have to introduce some kind of coupling between agents, for example that they would always be below a certain distance from each other. As we constructed our model from scratch, we thought it should be challenging enough to make it work with completely independent agents and decided to make every agents independent of all others.
The hallway in the main station in Zurich is approximately 60 meters long. We shortened the way to 30 meters as we reckon that the same effects should be visible over that length. This saves some time for the simulations as the way until the agents from the opposite directions meet is considerable shortened. Also, it is of practical use as one can observe more details in a less stretched picture.

5.2 How to run a simulation

To start a simulation, simply change to the "/code" directory in your local matlab installation. Then type "run" into the command window. This will set up a new simulation environment. This environment can be accessed trough the "sim" variable.
The run script will ask for a "runmode" and whether you'd like to save the data. Supported runmodes are:

normal	normal mode with many wall agents
fasttest	faster mode with less wall agents

Supported parameters are:

-nograph	simulation draws no graph, can be used if graphical output is not necessary.
-report	additional report will be recorded
-nodirections	removes the direction markers on top of the agents.

A string could be for example: "normal -nograph -report". For a more detailed example please view the header of the run.m file.
If one wants to save the data, type "Y" when asked and type in the name of the location/filename where the data shall be stored to. As a default, they are saved into the sim folder. Then the *defineConstants.m* file containing all the constants is saved as a .txt file with the runmode appended. After the simulation is finished, the workspace variables containing the "sim" variable are saved as a .mat file. The situation after the last iteration is drawn and saved as a .png file, also when the parameter -nograph was called. This allows a first visual check on how a run went.

5.3 Agents

As the model is agent-based, we wanted to implement them as such. Therefore we created a class called agent.m. Every agent has the following properties which are critical for the simulation:

- Radius of the agent called RADIUS
- The x coordinate of its position called CORDX
- The y coordinate of its position called CORDY
- A maximal velocity called MAXSPEED

- An actual velocity called ACTSPEED
- A priority called PRIORITY

Two more properties were introduced later for the analysis of the model. They are called DISTANCE and TIME and are used to monitor the time and pathlength an agent needs for crossing the field. The property ANGLE stores the angle an agent wants to walk. This was used to graphically show the way they want to walk to during the simulation.

A circle is the mathematically easiest shape to consider, especially for collision detections which have to be done later all the time. In addition, we consider the circle to be a good approximation as one also needs some space to move as the legs cover some space in front and behind the body. Also, a circle is very practical as it only needs one parameter entirely for the shape which is the radius. Using this approach, every agent can have a different radius.

The coordinates of the agent with respect to a cartesian grid centered at the lower left of the whole filed are vital for all calculations. They are also needed to define the circle representing the agent in the plane. After each iteration, they will be adjusted to the new situation.

Every agent has a maximum velocity. The actual velocity of an agent gives its actual speed. If there is no obstacle, it will be the maximum velocity. In the initialization of an agent, the first actual velocity is set to be equal to the maximum velocity. As velocity is in principle a vectorial quantity, we used the sign of the maximum velocity to determine the way an agent walks which is always well-defined as we don't let agents walk backwards. By our own convention, a positive sign means to go upwards (increasing y coordinate) and a negative sign means to go downwards (decreasing y coordinate).

Every agent has a priority and usually this is a random number between 0 and 1. It is used to determine the order in which the agents move through one iteration step. A high priority means that the agent will walk first. A high priority value can be attributed to an agent which will always try to push his way through.
The priority is also used to mark inactive agents. As all arrays are stored in an array of class agent, a priority of 0 denotes an empty position and will not be drawn. Any new agent is placed at the first position in the array of agents with a priority of 0. If an agent reaches its goal, it can be deleted by setting the priority to zero. This is the only time the priority value is changed after the creation of an agent.

The class *agent.m* was implemented using *handle* in the class definition. This has the same effect as *call by reference* in C++ as there is after the creation only one instance of the agent. This is important in the case where it is passed down to functions. Thereby a change of a property inside the function will also be effective outside the function which saves the step of giving the whole array of agents back after every call of a function which has to change some properties.

5.4 Drawing of the field

To draw the field, the matlab API is used in a very basic way. Matlab supports the drawing of different shapes like circles, squares and lines. They can be combined in a single plot to create more complex graphic objects. The field is repainted after every simulation step by different functions. There is a separate function for the wall painting and to paint the spawn-lines on both sides. The agents are painted in two steps. First it paints a round "rectangle" to create a cyclic shape and then, if the user desires, direction indicators on top of it. All the drawing functions are processed in a procedural manner. Every agent is handled individually.

The field is defined by its wall agents. They could be in any shape. For example, if one wants to extend the field with an obstacle somewhere between the spawn lines it can be defined by adding wall-agents to the specific coordinates. The easiest way to do so is to modify the createWall() function (which currently creates wall-agents on the left and on the right side to simulate a hallway). Keep in mind that the coordinate system's zero point is on the lower left side. All wall agents are currently invisible.

The de-spawn-line indicates where agents coming from the other side will disappear. We had to separate spawn and de-spawn areas because of possible positioning conflicts between arriving and new created agents.

We adapted the model we used for our simulations to the given facts inside the main station concourse "Zürich HB" which are given in figure 3. According to the measured distances we could calculate the average width of the passage (l for length, w for width):

$$\frac{l_{Nordsee} * w_{Nordsee} + l_{Imagine} * w_{Imagine} + l_{BurgerKing} * w_{BurgerKing}}{l_{Nordsee} + l_{Imagine} + l_{BurgerKing}} =$$

$$\frac{19 * 2.75 + 25 * 2.5 + 16 * 3.4}{19 + 25 + 16} = 2.819$$

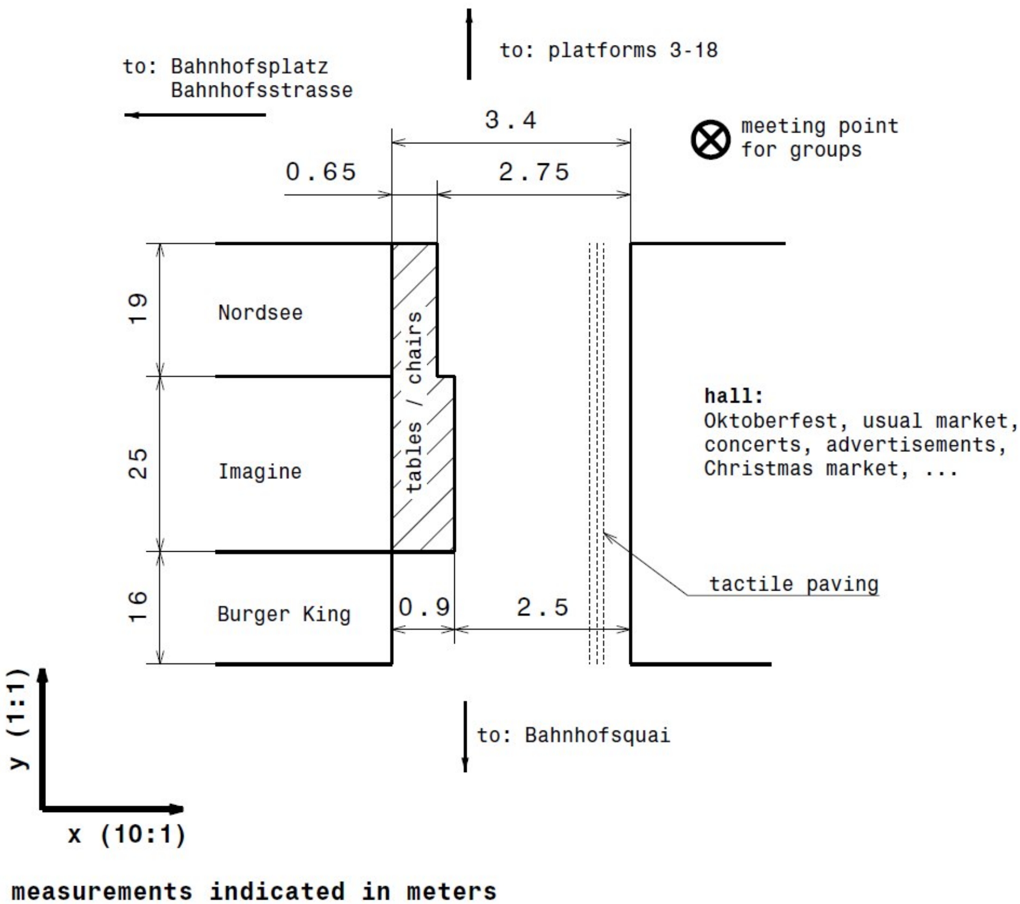

Figure 3: Scheme of the considered space in the main station of Zurich. Values are given in meters.

If one does extend the field with other obstacles or different shaped walls and if it is not possible to frame the new situation with a simple geometric structure, the wall agents have to be drawn again. This is currently disabled because it slows down the simulation speed, and because the straight walls can be easily substituted with a long small rectangle. Drawing wall-agents is very simple due to the fact that they are exactly the same object type like a normal, moving agent. One can combine the two arrays (wall-agents and the moving agents) and let it draw by the iteration loop.

Currently, walls are coloured black, agents moving from top to bottom are painted in blue and those from bottom to top are red. The agent's direction indicators are painted black. All the distances are in meters.

The proportions of a long narrow passage does not fit very well onto an usual computer monitor because the width-height ratios used for the passage in our simulations were up to 21:1. To prevent the images from being small and unclear we introduced a scaling factor variable called xStretchFactor with a default value of 5. This has some side effects one has to consider: Agents displayed as an oval are circles in the non transformed "real" environment. To obtain a better general overview about the situation one can set the xStretchFactor to 1 which resets the image to a compensated x/y ratio.

5.5 Logical functions

5.5.1 General considerations

The heart of the simulation is the function *logicFunction.m*, which determines the path any agent will choose to get to the other side of the hallway. To be more precise, it determines only the next step an agent will take and not the whole path. It relies heavily on the two functions *xValuesLogic.m* to deal with other agents and *xWallLogic.m* to deal with agents representing the wall or static obstacles. At first, the functioning of *xValuesLogic.m* will be explained and afterwards the functioning of *xWallLogic*.

5.5.2 How to get $\beta_{\textbf{Links}}$ and $\beta_{\textbf{Rechts}}$?

For our model, it is crucial to determine where an agent shouldn't go. The function *getBeta.m* returns the angles which describe the interval of angles leading to a collision. A graphical depiction of the situation is given in figure 4. The equations (2) to (4) were used to get β_{Links} and β_{Rechts}. They had to be converted into the angles given with respect to φ, $\beta_{\varphi,\text{ left}}$ and $\beta_{\varphi,\text{ right}}$ as shown in equations (5) to (6).

$$\gamma = arccos\left(\frac{r_S}{d}\right),\ \alpha = arctan\left(\frac{\Delta y}{\Delta x}\right) \tag{2}$$

$$\beta_{\text{Links}} = \gamma + \alpha - \frac{\pi}{2} \tag{3}$$

$$\beta_{\text{Rechts}} = \alpha + \frac{\pi}{2} - \gamma \tag{4}$$

$$\beta_{\varphi,\text{ left}} = \frac{\pi}{2} - \beta_{\text{Links}} = \pi - (\gamma + \alpha) \tag{5}$$

$$\beta_{\varphi,\text{ right}} = \frac{\pi}{2} - \beta_{\text{Rechts}} = \gamma - \alpha \tag{6}$$

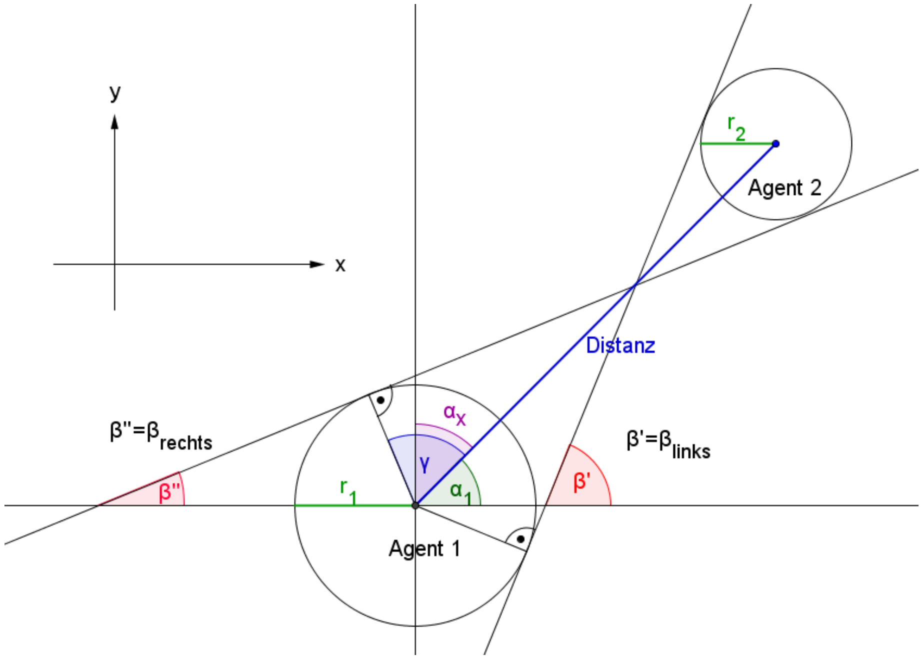

Figure 4: The graph shows the angles and variables used to get β_{Links} and β_{Rechts}. α_X is the angle between the two agents with respect to the y-axis. This depiction was engineered to work also for agents walking the other way.

This works between agents as well as between agents and the wall agents. Care was taken to engineer a calculation that allows for it to be used for agents walking in both directions.

5.5.3 Calculation of the interaction with another dynamic agents

xValuesLogic.m distinguishes three different cases.

- For two crossing agents or if the agent in front of the agent in question is slower, we used equation (7) to get x'_{out}. It was also used for two not moving agents, setting Δv equal to an arbitrary value given in `STANDOFF`. This was a quick way to resolve standoffs, although this would eventually turn out to be in its actual form an Achilles heel of the model.

$$x'_{\text{out}} = \frac{1}{(|x - \alpha_X|)^{\left(\frac{-\Delta v}{a}\right)}} = (|x - \alpha_X|)^{\left(\frac{\Delta v}{a}\right)}, \ \Delta v < 0 \tag{7}$$

All values which correspond to a collision course in x'_{out} are set to zero. This also deals with the singularity of equation (7) as it is set to zero. This is done using the β-angles shown before. Afterwards, x'_{out} is normalized and modificated further using equation (8).

$$x_{\text{out}} = x'_{\text{out}} \cdot \frac{b}{max(x'_{\text{out}})} \cdot \left(\frac{r_S}{d}\right)^c \tag{8}$$

The variables a (called `SLOPEFACTOR`), b (`HEIGHT`) and c (`REPULSIONAGENT`) have to chosen in a way that the simulation runs smoothly. The term $\frac{b}{max(x'_{\text{out}})}$ normalized the function to a maximum value b while the term $\left(\frac{r_S}{d}\right)^c$ controls that the repulsive influence gets stronger, the closer the two agents get. c is usually chosen to be larger than 1.
For two agents walking the in the same direction, the function given in equation 8 is additionally multiplied with the difference in speed $|\delta v|$ in a try to make them avoid standing agents more resolute as it that case $|\delta v|$ would be rather big.

If the x_{out} given in equation (8) would be returned, the agent in question would aim to miss the other agent exactly. We thought that this would be too close as in reality, one also leaves a bit of space if possible between each other. Therefore we introduced an offset given as `WALLANGLEOFFSET` which gives the angle additionally to the β angles for which an agent should aim to. To account for this, x_{out} is modified with an linear interpolation between the values at β+`WALLANGLEOFFSET` and β (which was set to zero before).

- If the agent in front of the agent in question is faster, a gaussian curve was used with the mean α_X and standard deviation rS/d. It is then modified further with Δv and `HEIGHT` to make it a weak influence.
- For two agents moving with the same speed, the influence is set to zero by returning a vector of zeros.

5.5.4 Calculation of the interaction with a wall agents

To avoid hitting the wall, we used a very simple approach. Every angle corresponding to a collision course is set to a negative value accoring to equation (9).

$$x_{\text{Out}} = x \cdot \frac{a}{d - rS} \tag{9}$$

As before for agents, an offset is introduced so the agent in question doesn't just try to avoid the wall-agent but also to leave some buffer space. The offset is also given in `WALLANGLEOFFSET`, a can be accessed with the constant variable `WALLFACTOR`. a has to be set negative as otherwise the wall would have an attracting force. To set a good value for this factor a is quite delicate because if it is too low, agents will be stuck in the wall while if it too high, they will never approach the wall even slightly.

5.5.5 Calculating functions in x and transforming them into a polar axis in φ

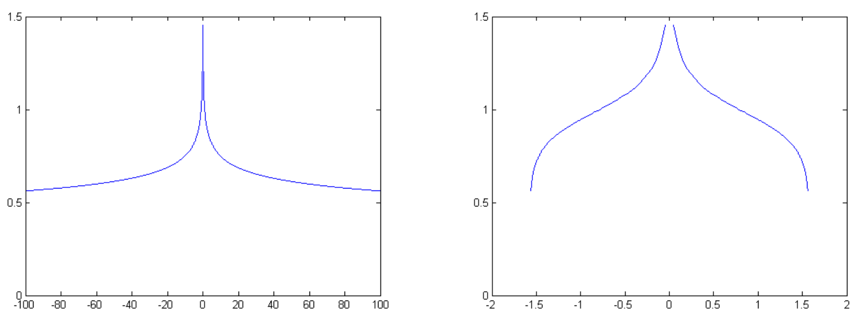

Figure 5: Graphical depiction of the function given in equation (7). In the graphic on the left, the horizontal axis is given in x while in the right graphic the horizontal axis is given in φ.

As the functions given above are given in x but the direction to go on is determined in polar values, it needs to be transformed into a φ axis. This is done using a vector for x ranging $\pm$ `XSCOPE` with a step of `XRES`. Applying the arcustangent on it yields the axis in angular values (φ). The transforming is done simply by using the φ axis instead of the x axis. This is shown in the two figures 5 and 6. As those functions are only chosen to show the principle, they were not normalized in height according to the equations mentioned above.
In figure 5, the function (7) is shown graphically in the x as well as φ axis. Figure 6 shows the x_{out} the function *xValuesLogic.m* returns.

5.5.6 Graphical example

This subchapter shall give a visual example of how the logic functions work. Let's consider the situation given in figure 7.
The blue agent moving up in figure 7 only sees the influence of the wall. What the blue agent "sees" is given in figure 8. The influence of the wall causes the overall function to decrease for all φ corresponding to a collision course. The offset causes

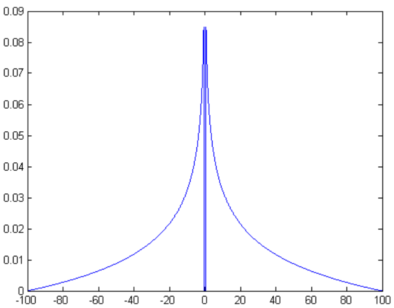 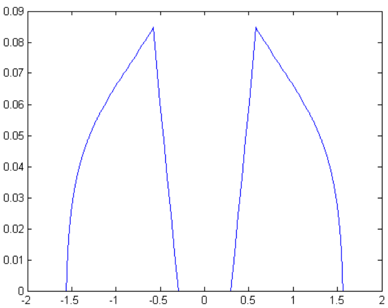

Figure 6: Graphical depiction of the return values of the function x*ValuesLogic.m*. In the graphic on the left, the horizontal axis is given in x while in the right graphic the horizontal axis is given in φ. α_X was set to 0 corresponding to an other agent directly ahead, β was set to ± 0.3 with an offset of 0.25.

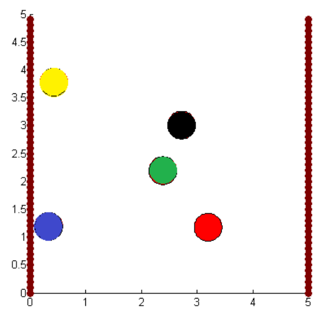

Figure 7: Exemplary case used to demonstrate the working principle of the logic functions.

the overall function to have its maximum α at a positive φ. This causes the agent to walk in the direction of α.

The green agent moving up sees only the influence of the black agent who is moving down. The effect of that it given in figure 9. The underlying gaussian function can be seen as well as the addition of a modified version of figure 6, left. The agent will move slightly to the left to avoid hitting the black agent.

The black agent moving down sees the oncoming green and red agent going up. The effect of them is given in figure 10. The superposition of two functions onto the underlying gaussion can be seen by the discontinities. The effect of the green agent is stronger as it is nearer to the black agent than the red agent which causes the

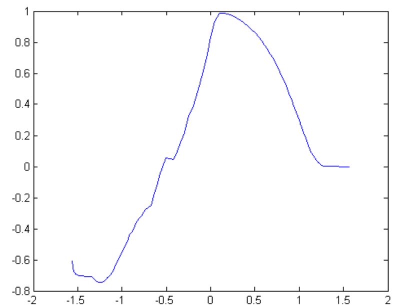

Figure 8: Output of all logic functions combined for the blue agent in figure 7. Visible is the effect of wall agents on the blue agent as negative values on the left side.

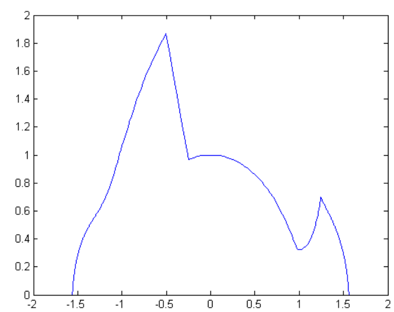

Figure 9: Output of all logic functions combined for the green agent in figure 7. Visible is the effect of the oncoming black agent as a superposition on the underlying gaussian curve.

black agent to go left (looking top-down) in order to avoid hitting the green agent. This shows the dependency of the strength of an agent of the distance.

5.6 Iteration

5.6.1 General considerations

One iteration step is carried out using the function *Iteration.m*. For it to work properly, the array of all agents and all wall agents has to be passed to it. All other inputs as well as all output variables are introduced for evalutation of the model and are per se not necessary for the iteration. It deals with three main tasks, the propagation of the simulation in time, the collision detection and the destruction of an agent after reaching its goal.
The spawning of new agents is treated in the function *Spawn.m*. Both functions are

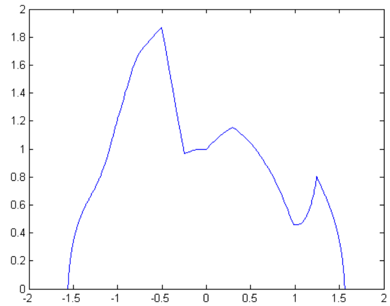

Figure 10: Output of all logic functions combined for the black agent in figure 7. Visible is the effect of the oncoming green and red agents as superpositions on the underlying gaussian curve. The black agent will move left (from his point of view) to avoid hitting the closer green agent.

called from the *simulation.m* which controls the simulation and stores all the data.

5.6.2 Iteration step and collision detection

This function takes four input array, although only two are critical for the success of the iteration. Those two are the arrays for all dynamic and static agents (equals wall agents). The output consists of the number of agents which disapeared during the iteration step, the distance covered by the disapeared agents as well as the time the disapeared agents had spent in the simulations.

At first, an index array is calculated with the indices of the dynamic agents in the given array of agents sorted according to their priority. This is done using the functions *getPriorityArray.m* and *getSortedPriorityArray.m*. Then a loop over the index array is carried out.

For every agent, the desired direction is calculated using the function *logicFunction.m* as explained above. Using the angle obtained by the call of *logicFunction.m* und the speed given as the agent property, the new x- and y-coordiantes-to-be are determined. The path to it is then split in several substeps with the number of divisions given in the constant `PRECISIONCOLLISION`. In addition, the place where the agent stands is also included in case the agent cannot move at all.

For each of these positions, the distance to all other agents minus the radii is calculated yielding a distance matrix. This is done over all agents, dynamic as well as static. The last position for each other agent is left -1 as a sentinel. The distance matrix is then sorted using the matlab command *sort* which leaves the most negative value for each column in the first row. The position of the first negative minimal

distance indicates a collision. This position minus 1 will then be the distance the agent in question walks.
In very rare cases, all positions in the first row of the sorted distance matrix are negative. This has the very nasty significance of an agent that could not even stand at its actual position. We reckon this has to do with some small numerical errors, as it occurs very rarely. To leave the agent at its actual position will now only result in a complete freeze of two agents. It is certainly not a nice solution, but in these cases we simply delete the faulty agent by setting its priority to 0 and reset its distance and time properties to enable the simulation to keep on running.

5.6.3 Destruction of agents

If an agent reaches his goal, namely the other side, it is automatically deleted inside *Iteration.m*. This is simply done by setting the priority of the agent to 0. The attributes time and distance are read out before resetting the agent and handed back to the calling function for later evaluation. After a deletion, the priority array giving all the indices of active agents is recalculated in order to avoid any influence of the deleted agent on other agents as the iteration step proceeds through the residual agents in the loop over all agents.

5.6.4 Spawning new agents

The spawning of one new agents is done by the function *spawn.m*. Every call of *spawn.m* spawns a new agent, whether the function is called at all is handled an instance higher in the simulation class. After each iteration step, namely by calling *Iteration.m*, the class simulation determines whether a new agent is spawned or not. There are two constants determining the amount of agents spawned over a long period called `DENSITYUP` and `DENSITYDOWN`. They correspond to a density of people or, in other words, the number of people per second that should appear. As the names suggest, one is used for the number of agents spawned in the upper spawn zone while the other is used for the number of agents in the lower spawn zone. This information is then passed down to the function *spawn.m* in the variable *position* which can only take the values 1 or -1. Using our implementation, maximally one agent is spawned per iteration per side according to the following equation with the according density ρ and the time step length Δt.

$$p(\text{spawn}) = \begin{cases} \Delta t \cdot \rho & \Delta t \cdot \rho \leq 1 \\ 1 & \Delta t \cdot \rho > 1 \end{cases} \tag{10}$$

For sufficiently small values of Δt and ρ, it can be in good approximation assumed that only one agent is spawned per time step. The probability of spawning two

agents would go with p^2 which is small for the values we have chosen. Otherwise one would have to implement an additional condition which would determine at first how many agents are to be spawned with the probability for spawning n agents going in principle with p^n.
To spawn an agent, the first agent position in the array of agents with a priority of 0 is taken. The radius is generated using a normal distribution (*randn* in matlab) with a mean of `MEANRADIUS` and a standard deviation of `STDRADIUS`. The maximal deviation possible was set to be three times the standard deviation. Should the generated radius be out of bounds, the radius generating procedure was simply repeated. A starting position in x-direction is then generated with an uniform random distribution over all possible starting values. A collision detection is then carried out to see whether the chosen position is possible without spawning onto another agent. If it fails, this procedure is repeated `REP` times, a constant defined at the beginning. The y-position is given by the dimension of the field while the velocity is determined using a normal distribution in the same way as before for the radius with the mean given by `MEANSPEED` and the standard deviation given by `STDSPEED`.
It is possible to implement other spawn sequences than the one used for this model. If one wishes to get specific agents with certain properties, they could also be spawned directly from the simulation.

5.7 Readout of informations of after a simulation

To evaluate how the model has worked, both a graphical and a mathematical output is given. As a graphical check one can take a look at the last situation which is saved as a png. This gives immediate information about whether the agents got caught in a jam or not.
For further and more precise analysis, several variables listed below are stored which can be used to monitor several aspects like the overall efficiency of the model. In the list below, the variables are stated in the way they can be called after a simulation.

- **sim.loops**: Gives back the number of iterated loops.
- **sim.evaluateDistance**: ($1\times$(# of arrived agents))-matrix containing the distances of all agents which have finished their way across the hallway.
- **sim.evaluateTime**: ($1\times$(# of arrived agents))-matrix containing the spent time in the simulation of all agents which successfully crossed the hallway.
- **sim.spawned**: ($2\times$loops)-Matrix containing the number of spawned agents at the top (column 1) and bottom (column 2) of the hallway.

- **sim.result**: (2×loops)-Matrix containing the number of deleted agents at the top (column 1) and bottom (column 2) of the hallway.
- **sim.additionalresult**: (2×loops)-Matrix containing the total distance walked by all agents during each iteration step (column 1) and the total number of agents in the system (column 2).

5.8 Defining all constants

The file *defineConstants.m* contains all the constants that are used somewhere in the simulation. They can be grouped into several categories which are listed below, only the most important and most frequently changed constants are listed explicitly.

- The first section containing `XSCOPE` and `XRES` define the extent of the numerical approximations. An acceptable compromise between numerical resolution and runtime has to be chosen.
- In the second section, the model parameters can be set. They were explained in the subchapters about the logical functions and the spawning.
- The third section is used to define the field in which the agents will walk.
- In the fourth and last section, general parameters concerning the simulation can be set. They include the time increment `DELTAT` and the number of loops `LOOPS` for the iteration process. The seed for the random number generator can be set with `SEED`.

The seed for the random number generator is important to get random numbers but reproducible results.
Please note that early simulations may have a slightly different ordering of the constant variables in their logfiles.

6 Performed simulations

Listed below are the carried out simulations with their most important parameters. For each series of simulations, a brief explanation is given to state the questions which will be looked at with the actual simulation series.
All parameters can be found in the corresponding logfiles. Usually only one parameter was varied while all others were kept constant. We chose a mean radius for the agents of 0.25 meters with a standard deviation of 0.03 meters. A mean velocity of 1.5 meters per second was chosen with a rather large standard deviation of 0.25 meters per second.
It should be noted that we usually used high people flux densities since we wanted to test the model under stress conditions. Therefore we expected a considerable amount of failure in the examined situations.

6.1 Influence of different pedestrian flux densities

To check the influence of different densities on our model, we ran the simulation with the density combinations 0.4/0.4, 0.4/0.6, 0.4/0.8, 0.4/1.0, 0.6/0.6, 0.6/0.8, 0.6/1.0, 0.8/0.8, 0.8/1.0 and 1.0/1.0. The first number represents the value chosen for `DENSITYDOWN`, the second for `DENSITYUP`. We didn't run the inverted combinations due to the situation's symmetry. The simulations were repeated with three different seeds each, 51, 71 and 91. A high value for `DISPERSIONFACTOR` of 1.0 was chosen which corresponds to people having a strong tendency to try to overtake slow agents. All simulations were run for 120 seconds.

6.2 Influence of overtaking or lane formation on the success of the model

It soon became clear to us that the parameter `DISPERSIONFACTOR` would be absolutely crucial if one wants to force the model to succeed. A negative value encourages the agents to form lanes while a positive value encourages them to try finding their own way. In order to investigate this property, specially with the dilemma of personal vs. group success in mind, we ran a simulation series where we incremented the `DISPERSIONFACTOR` from -0.2 to 1 each time by 0.1. A high density flux of 1 person per second on both sides was used to test the model in a stress situation. This was done for three different seeds each, 51, 151 and 351. All simulations were run for 120 seconds.

6.3 Influence of the radius of sight of an agent

The constant variable `INFLUENCESPHERE` determines the radius of the semi-circle in which the agent considers other agents around him. With flux densities of 1.0 each and a `DISPERSIONFACTOR` of 0.7, the `INFLUENCESPHERE` was tested using the values 1.5, 2.0, 2.5 and 3.0 (in meters). This was done for three seeds each, 51, 77 and 151. All simulations were run for 120 seconds.

6.4 Influence of the hallway width on the success of the simulation

To account for the influence of the width of the hallway on the success of the simulation, we did a simulation series with different widths. The tested widths were 2.2, 2.5, 2.8, 3 and 3.5 meters. A high density flux of 1 person per second on both sides was used with a `DISPERSIONFACTOR` of 0.75 corresponding to a high number of overtaking attempts. The simulations were repeated with the seeds 51, 77 and 151 each. All simulations were run for 100 seconds.

6.5 Simulating measurements of the main station Zurich

Saturday, Nov 17th, we did some quick measurements right at Zurich main station to have some data we could try to compare. Two measurements were taken, only some minutes lay between these, that was when we measured the length and breadth of our corridor. The measurements were:

1. The "boring" measurement: During 2 minutes, 14 pedestrians headed towards tracks 3-18, and 20 pedestrians directed towards tram station "Bahnhofsquai". No problems at all, very fluently.

2. The "crowded" measurement: During 2 minutes, 41 pedestrians headed towards tracks 3-18, and 33 pedestrians directed towards tram station "Bahnhofsquai". People got stuck, ran into each other, and had to walk stop-and-go-like for some moments.

In order to simulate this, we used values of 0.12 (up) and 0.17 (down) as flux densities for the first measurement, the second measurement was simulated with flux densities of 0.34 (up) and 0.275 (down). In all cases, a `DISPERSIONFACTOR` of 0.7 was chosen. These values represent the measured flux densities.
To get also something like a rush hour, we simulated a situation with approximately double flux densities than in the "crowded" measurement, namely with 0.6 (up) and 0.5 (down). All simulations were run for 180 seconds.

6.6 Simulation of a big inequality in the flux densities

This series was designed to test how a small number of people walking up would react to a big number of walking down. To test this, the flux density walking down was kept constant at 1.0 while the flux density of people walking up was varied with the values 0.2, 0.3 and 0.4. `DISPERSIONFACTOR` was again set to 0.7 with an iteration time of 180 seconds. The used seeds were 51, 91 and 113.

The simulations were carried out with MATLAB R2011a on a HP EliteBook 8400p (Intel Core i7 CPU M620 @ 2.67 GHz) with a windows 7 professional operating system.

7 Simulation Results and Discussion

7.1 Goals

First, let's have a look at what our goals were. We planned to have a look at the pedestrian flux, how it can be improved and jammings be avoided. We furthermore wanted to have a closer look to what happens during rush-hours and in a situation when much more people are moving in one direction than in the other.
On the agent-based side of our model, we wanted to analyze the influence of aggressive fast people in a rush, slowly moving obstacles (eg. mothers with baby buggies) and the influence of drunkard (more or less randomized walking) on the pedestrian flux.
If everything went well, we also wanted to implement a static obstacle and see what happens. As a reminder before the discussion of the results, our fundamental research questions were:

- How does the simulation behave in the following situations: rush hour, with obstacle, with very slow/fast agents, random path agent (drunkard)? Does it run smoothly or will ther be jams?
- How will our implementation of a rudimentary kind of "thinking ahead" affect the simulation? Will it work good or bad? Can we compare it to other implementations?
- Are there any group dynamics evolving as lane or group formation?

7.2 General achievements

As soon as we started programming we realized there was a major point of importance about this work we all were aware of, but had forgot to put it in the project proposal. We all did not want to start with an already known program or existing algorithms, but build something "new" on our own. So we started off creating our logic function that would allow the agents to avoid crashing into other agents and not working with repulsive forces as for example Helbing in [1] did.
Quite proudly, we can now say we managed to do this. Our idea of the agents "thinking ahead" by consulting where other agents are and not just being pushed around by repulsive forces worked.
We now are able to play with lots of input variables, the most important being number of agents entering the corridor per time and the agents' characteristics as size, speed and lots more.
A nice thing we built but did not originally plan to is that we planned to and did research on the situation as explained earlier in the long, narrow corridor in Zurich

main station. But in our simulation, one can also change dimensions as length and shape of the walls easily as well as inserting obstacles.

We therefore decided first of all to make sure that the model works and what its operating parameters are. This meant that we had to drop a lot of our former goals because we did not want to carry on with a faulty model. Therefore we have included some results that were not included in our first questions we set out to answer in the beginning.
On the downside of this, we dropped the investigation into the behaviour of the pedestrian flux when exposed to aggressive, slow or random people. Even though these situations were not simulated, the functionality to introduce them without much work was implemented into the model as they were considered when we built our model.

7.3 Results from the simulation series

7.3.1 Influence of different pedestrian flux densities

For this experiment we monitored the total agents count as an indicator for jams. This number was then scaled by the combined flux densities to give comparable results which can be found in figure 11.

With the highest flux densities used in the simulations, the jam formation was very fast. It is interesting that 0.6/0.6, 0.8/0.4 and 0.8/0.8 also caused jams within the observed time frame, especially since 1/0.4 didn't jam.
Our model seems not to be able to cope well with that many people. There should be more simulations with different seeds to get a better statistic.

Given in figure 12 are the mean covered distance and mean time spent during the simulation for all agents who reached their destruction line compared with the combined flux densities used. As would be expected, the agents had to cover a longer distance and spend more time in the simulation as they had to avoid collisions with other agents.
These variables were monitored to analyze whether they give a good description of the model and to ensure that we got results that run within the same expectations as in reality. But as they only take the values of those agents that have left the simulation, they may have a decreased significance when jams occur and are not at all able to tell whether a jam has occured or not.

To sum up, we could observe what we expected to see: the more people, the more

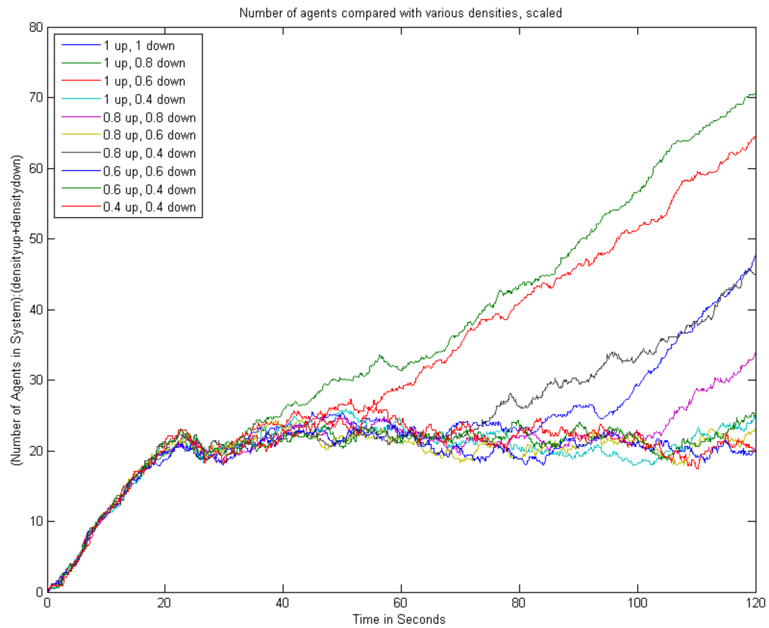

Figure 11: The scaled total number of agents in the system for various combinations of flux densities with respect to time. The used flux densities are given in the graph legend. As soon as the total number of agents runs away, a jam has formed. The mean over three simulations with different seeds was taken for each case. As expected high flux densities caused massive jams.

probably a jam pops up. Also, when the densities are increased, the agents have to walk longer ways, need a bit more time and thus have a smaller average velocity.

7.3.2 Influence of overtaking or lane formation on the success of the model

Some examples of how our simulation did look like after a simulation time of 120 second are given in figure 13.

In figure 14 all collected data is consensed into one graph which was visualized in two ways. Another representation of the same data is given in the appendix in figure 24 (page 65). They correlate the average distance covered per agent per iteration step with the simulation time. There were two expectations: As soon as a jam starts to form, this variable should decrease quite fast. Any kind of lane formation should be detectable as most agents will not be able to walk with their maximum speed so

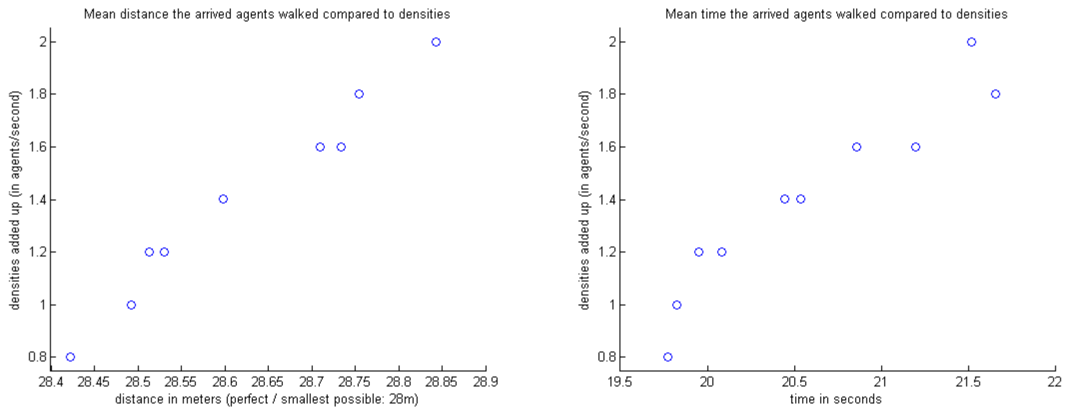

Figure 12: Graphs of the mean covered distance and mean time spent during the simulation for all agents who reached their destruction line compared with the combined flux densities used. As expected, for high flux densities the agents have to cover longer distances and spend more time as they have to avoid colliding with other agents. Note that even though this data is interestingly distributed, its statistical relevance is rather small, because the standard deviation is rather high due to the agent's speeds being Gaussian distributed.

the average distance per agent per timestep should be significantly below the mean value given by $\bar{d} = \Delta t \cdot \bar{v}$, the product of `DELTAT` with `MEANSPEED`.

Given the two graphs, it is striking that the majority of the simulations represented by reddish lines representing high values of `DISPERSIONFACTOR` fail during the observed time frame. At rare occasion though the simulation was successful even with a relatively high `DISPERSIONFACTOR`.
On the other side, there was always lane formation for a `DISPERSIONFACTOR` below 0.4, represented by the more blueish lines which ultimately resulted mostly in successful simulations without jamming.
The lane formation can be seen quite clearly in the graphs given in figure 14. The reddish lines try to keep the mean distance at almost all cost, this is visible in the initial height of the reddish lines. In contrast, the blueish lines quickly fall down by about 0.02 meters per agent per iteration step which is a precise indication of lane formation. But in the long run, the best reddish line performed about equally well as most blueish lanes, indicating that in crowded situations like this, cooperation between agents in the form of line formation is not worse in performance as the best egoistic approach, but succeeds way more often.

Summing up, it seems that the forced lane formation was a successful way to resolve

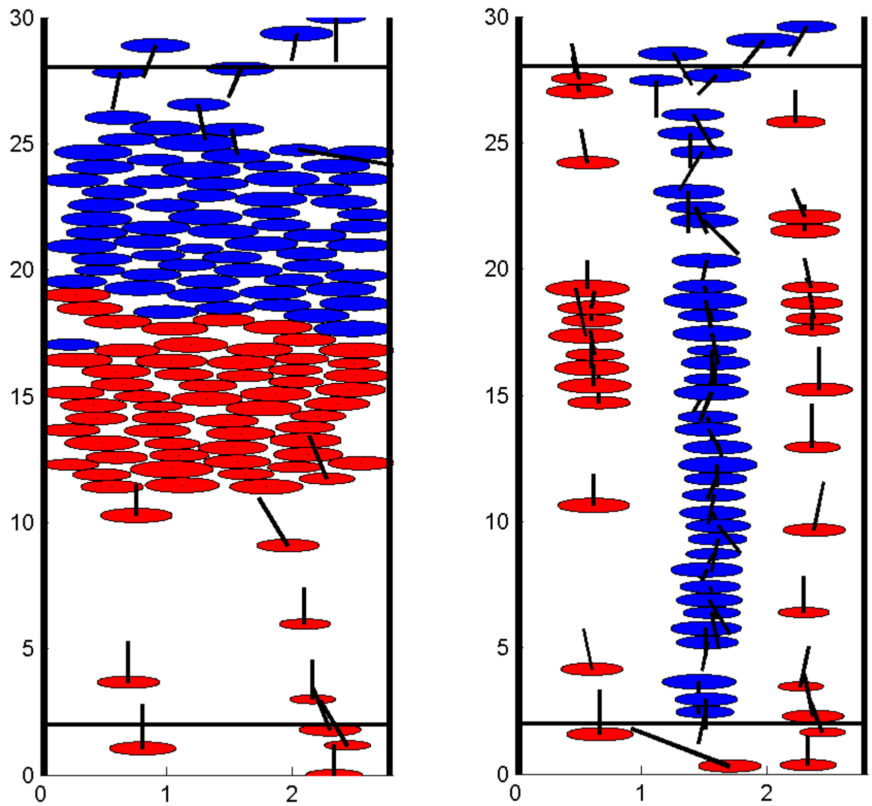

Figure 13: Exemplary pictures of our simulation after a simulation time of 120 seconds. The left one was run with a `DISPERSIONFACTOR` of 1 while the right picture has one of 0.1. The jaming to the left and the lane formation to the right can be seen.

the problem of jamming. It also highlights the importance of cooperation and the sensitivity of our model towards this parameter.

7.3.3 Influence of the radius of sight of an agent

In this test series, the constant variable INFLUENCESPHERE, which determines the radius of the semi-circle in which the agent considers other agents around him, was varied. First of all, the dependency between the influence sphere's radius and the total distance walked by all agents had to be evaluated.

As shown in figure 15, neither dependencies nor tendencies can be seen clearly, as for every tested radius of influence, some seeds worked well where other simulations for the same seed jammed.

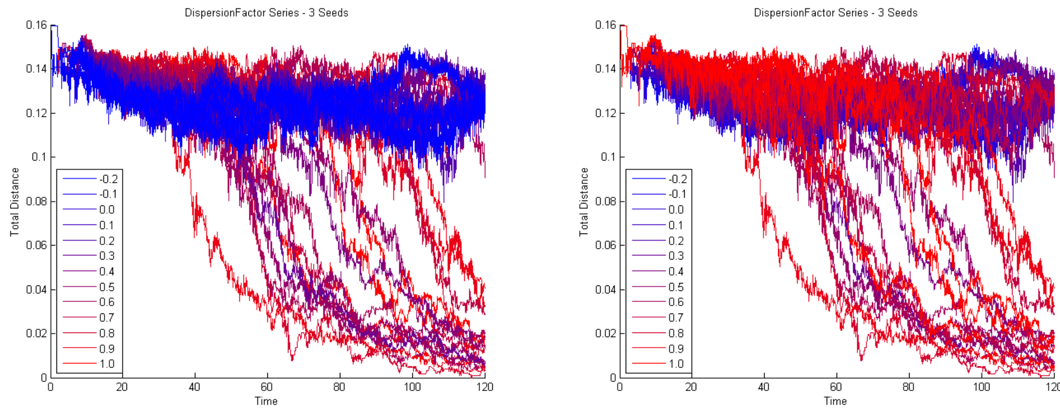

Figure 14: Graph of the average covered distance per agent present in the simulation per step as a function of time. The more blueish the color is, the stronger was the agents tendency to form lanes while the more reddish the color is, the stronger was the agents tendency to try to overtake slow agents. In the left graph the more blue lines are highlighted while in the right graph the more red lines are highlighted. Although the red lines representing "greedy" agents cope well at the very start, the usually lead to a jam very quickly.

Next, there maybe is a dependence between the success of a simulation and the seed which would explain why no clear tendencies for the radii was observed in figure 15. So, in figure 16, the same seeds were coloured equally, which showed much more of a tendency: A seed seems to have the property that it will most likely jam or not: Seed 51 caused jams for every radius, whereas seed 77 and seed 151 produced a similar result except for outliers.
To investigate this further, the plot was now split up into 3 subplots with one for each seed as given in figure 17. Again, this seems to show some dependency as assumed in figure 16, but this could also be coincidence.
Here, no dependency can be seen because with seed 51, every single simulation will cause jams. Seed 151 shows what one could expect to see: For a small influence sphere radius, a jam pops up. But seed 77 shows the exact opposite of this as here, the larger influence radii caused jams. So neither dependence nor tendency between influence sphere radius and total distance walked can be observed because of the opposing trends.

This leads us to think that coincidence played a main role here and shows us another thing: In our formulas weighing the influence of other agents on a specific agents, the weights are too big for very close agents in comparison to agents in a larger

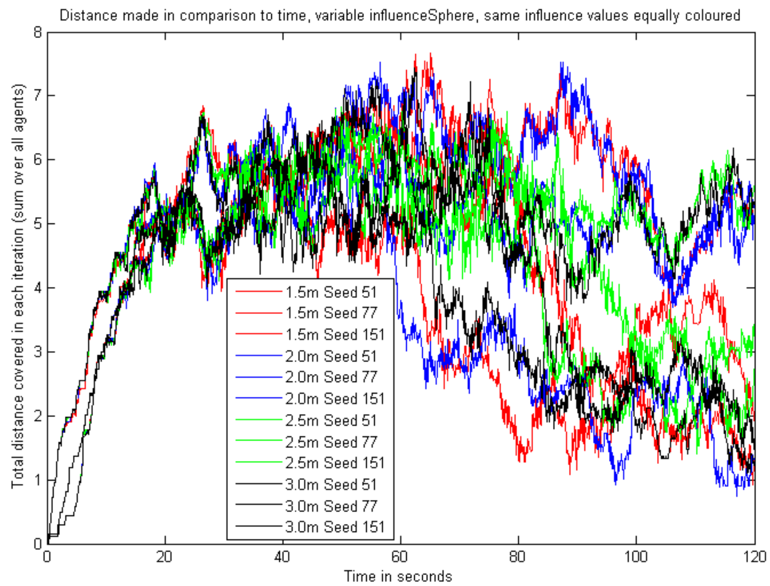

Figure 15: This plot shows the total distance covered by all agents in the system for different influence sphere radii and seeds in dependence of the simulation time. This distance decreases rapidly when a jam starts. To show the total distance's dependence on the influence sphere radius, each seed to a certain radius was coloured equally. One can see here that there is no clear dependence of the radii to the total distance as for every radius except 1.5 m there are seeds that work and seeds that don't, which means they had jams.

distance. This explains why larger radii didn't show the expected results: the additional influence of those agents was too little to matter. The assumption that our weighing function is not perfect can also be observed in any simulation: The agents walk straight forwards to each other for a long time and avoid each other only when they are very close to each other and not from a few meters ahead.
But thinking about the implementation, these results are not really surprising as we weighted short distance interactions quadratically, therefore the inclusion of more influences with a big distance will in comparison not have a big effect. This could change if the weighting would not be quadratic but maybe linear, but then the agents should bump into each other much more frequently.

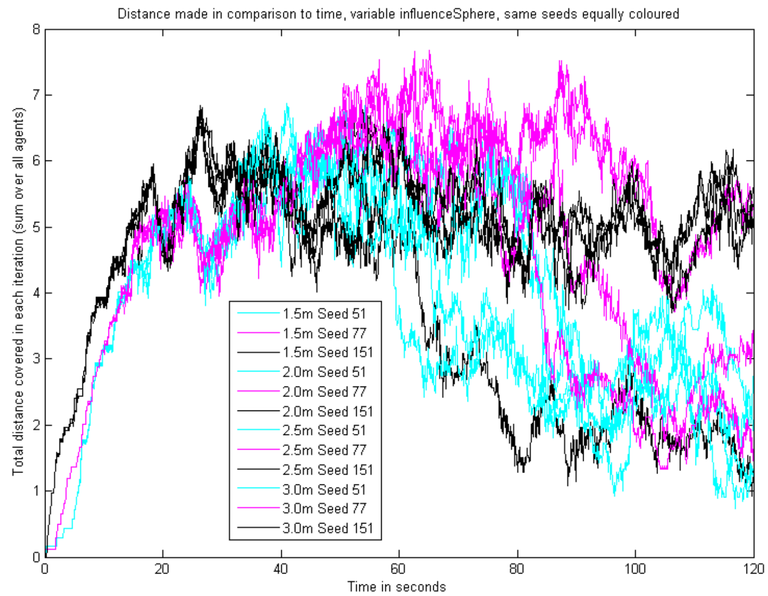

Figure 16: This plot shows the total distance covered by all agents in the system for different influence sphere radii and seeds in dependence of the simulation time. This distance decreases rapidly when a jam starts. To show the total distance's dependence on the seed, each radius to a certain seed was coloured equally. One can see here that whether a simulation jams or not seems to depend on the seed, as the seed 51 simulations all jammed independent of the radius, and for the other two seeds, there was only one radius that was different.

7.3.4 Influence of the hallway width on the success of the simulation

As a fast visual analysis, one can see that for 2.2 and 2.5 meters width, every simulation got stuck. Two 2.8 meters width simulations also had a jam started at the end of the simulation. In addition to those, also one of the three simulations of 3.0 and 3.5 meters width each got stuck too.

To detect jams, looking at the total number of agents in the system was a suitable way as this number starts to increase strongly when agents start to get stuck. Figure 18 shows the total agent number in all 15 simulations that were carried out evolving in time. What one can clearly see is that the results are not perfect and that random chance plays a big role as for example also one simulation of a 3.5 meters wide hallway started jammed even though there would have been plenty of space left at the formation of the jam.

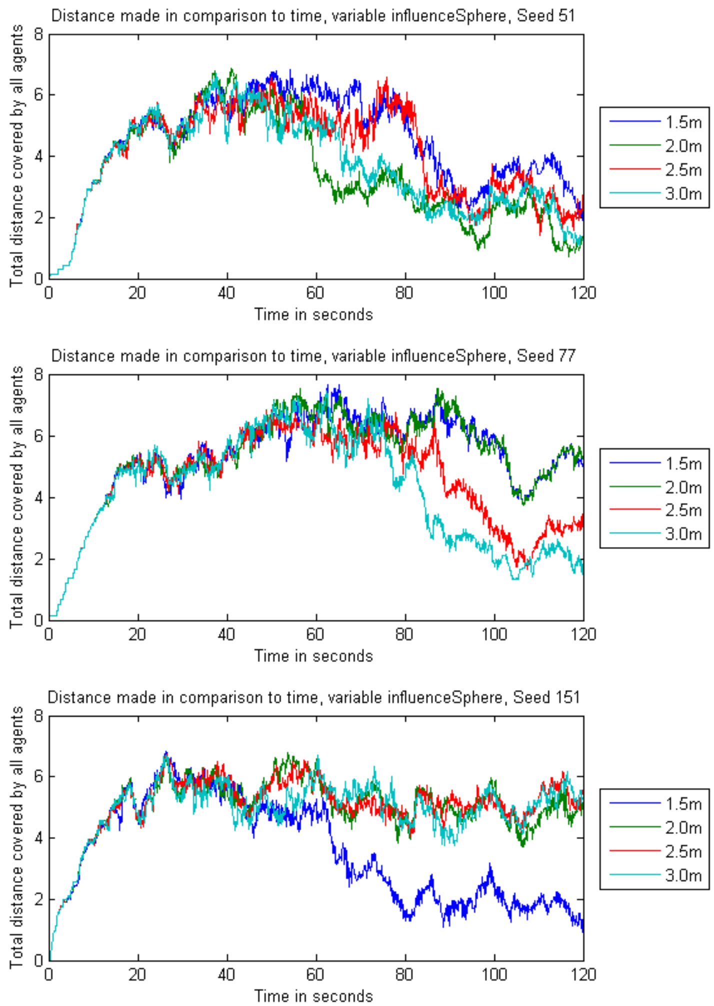

Figure 17: This plot shows the total distance covered by all agents in the system for different influence sphere radii and seeds in dependence of the simulation time. This distance decreases rapidly when a jam starts. To investigate the radius' dependence on the total distance, the total plot was split up into three parts, each containing all simulations for one seed. One can see here that in seed 51, every simulation jammed independently of the radius. In seed 77, the larger radii jammed, on the other hand only the small radius crashed in seed 151.

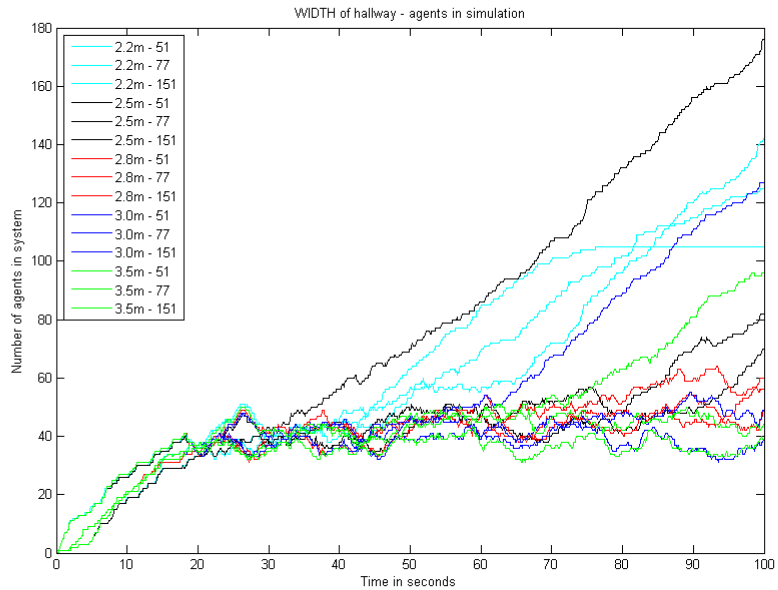

Figure 18: The plot shows the total number of agents in the system for different widths and seeds in dependence of the simulation time. This number exceeds an equilibrium value (around 40-50 agents) when a jam started. To show the total agent number's dependence on the hallway width, each seed to a certain width was coloured equally. Hallway widths used were 2.2, 2.5, 2.8, 3.0 and 3.5 m. One can see here that the narrower a hallway is, the more probable jams pop up, but from 2.8 meters on it mostly worked well.

But there are tendencies which are nevertheless visible: The most obvious thing is that there really is a equilibrium of people in the simulations when the simulation runs smoothly. This is where most of the graphs are. When a jam pops up, the number of agents starts to increase quite linearly because agents are spawned but can't reach the end of the hallway. The horizontal line is a clear sign that both spawn zones are clogged up.

In figure 19, the data sets for each seeds were averaged for better visibility. One can clearly see the tendency for narrower hallways to get more jams, although the others rise up too due to the average taken over equilibrium seeds and jam seeds.

The observed tendency was in accordance with what we expected, although we hoped for a bigger difference in the success rate.

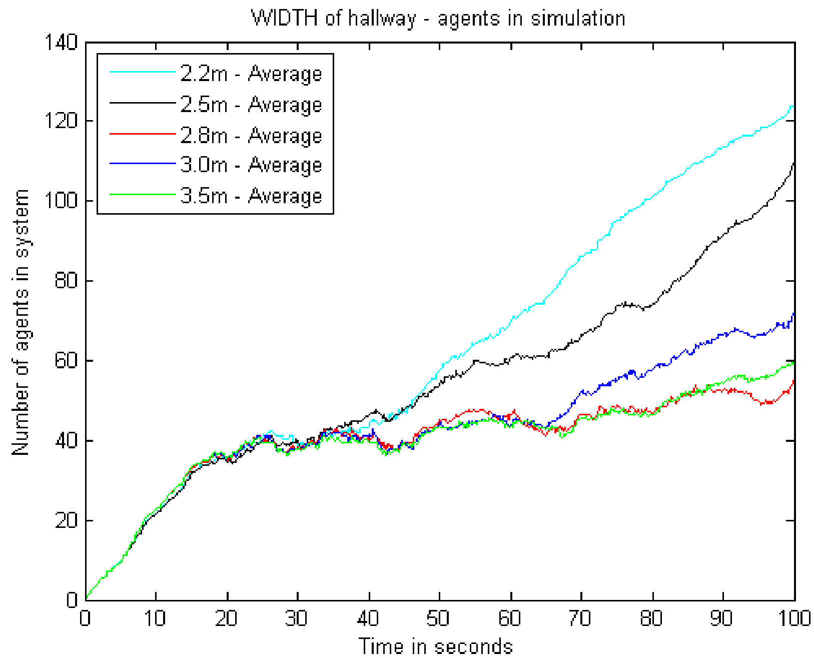

Figure 19: This plot shows the number of agents in the system for different widths and seeds in dependence of the simulation time. The total agent numbers of three seeds were averaged to one vector for better visibility. This number exceeds equilibrium (around 40-50 agents) when a jam started. Hallway widths used were 2.2, 2.5, 2.8, 3.0 and 3.5 m. The total agent number's dependence on the hallway width can be seen here: The narrower a hallway is, the more probable jams pop up, but from 2.8 meters on it mostly worked well. The 3.0 m graph can be looked at as an exception because one of its seeds exploded quite badly so the average looks high too.

To sum up, the tendencies observed in this simulation series are quite obvious: The more narrow a hallway gets, the higher the probability is to get stuck, which can be seen as all the 2.2 and 2.5 meter wide simulations ended in jams. Also, two of the 2.8 meter hallway simulations ended in jams, but for wider hallways, only coincidence made jams possible.
One can derive from this, that there is a certain width that will mostly work and is between 2.8 and 3.0 meters. Anything higher won't be much of an improvement for the chosen densities, because the flux is already smooth but won't be faster. But as the hallway is narrowed, jams are inevitable.

7.3.5 Simulating measurements of the main station Zurich

The experiments were only analyzed visually by judging how well our model could cope with the given task and how it compared to the real observations. In figure 20, one final situation for each run is given as an example.

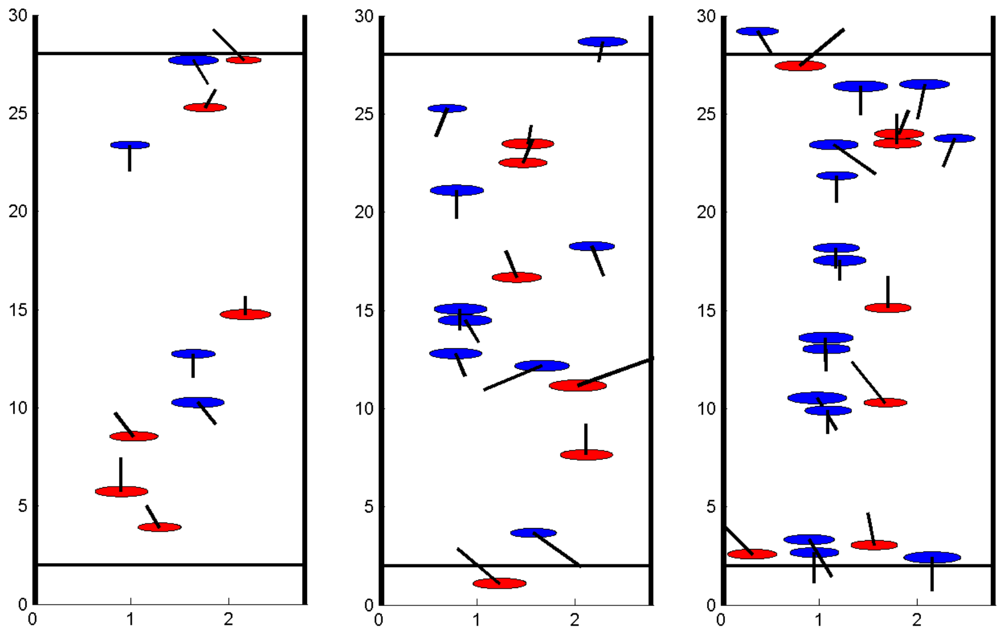

Figure 20: Exemplary pictures for the simulation of the long narrow hallway in the Zurich main station. Red agents are walking up and blue down, the black line denotes the actual velocity vector with its angle and length. The left was done with flux densities of 0.12/0.17, the middle with 0.34/0.275 and the right with 0.6/0.5 after a simulation time of 180 seconds. In all investigated situations, the agents managed to cross the hallway without significant hindrance from other agents.

For the first simulation series with a low people flux, the agents had no problems and could avoid collisions/walking into each other easily. This is in accordance with the observations.

For the second simulation series with a medium people flux, the agents had little problems crossing the hallway. The stop-and-go of the observation was only rarely seen, suggesting that the logic behind our model is actually quite good.

In the third simulation series with a high people flux, the agents had to stop sometimes while getting on the other side. But overall they could cope quite well with the task and the frequency of agents bumping into each other was quite slow and definitely in the same range as in real situations. Sometimes a small lane formation could be observed.

We can say that our simulation worked well on the measured quantities. Of course, in reality, when jams start, there will be more side effects as people pushing, turning around, or trying to walk another way, but for a straightforward walk, our simulations mirrors the reality nicely.

7.3.6 Simulation of a big inequality in the flux densities

The experiments were only analyzed visually by judging how well our model could cope with the given task. Some exemplary pictures are given in figure 21.

For the densities 0.2/1 used in the first series, the overall success was satisfying. The agents walking up (red, minority) were wandering about quite strong due to all the oncoming blue agents. But even though the blue outnumbered the red agents 1:4, the red agents stopped very rarely.

In the case of the densities being 0.3/1, the results were quite similar with those obtained by 0.2/1. But the tendency for the red agents to walk in lanes or small groups increased clearly, probably because there are more red agents which are shuffled together by the big number of blue agents trying to stomp their way through.

For the last considered case, the simulation failed in one case after approximatly 140 seconds (seed 51). The situation after 180 seconds is given in figure 22 as a classical example of how a jam looks like in our model. Before that and in the other two simulations, the red agents formed lanes and little groups very frequently as a way to not always get tossed over the whole width.
But the failure suggests that we reached about the limit which can be modelled with our simulation.

To sum up, we can say that what we expected could be observed. Anyone who ever went against the tide, for example when people debark a train or bus, knows that advancement is hard to reach. This effect of slowing down and flocking together with other people who try to walk the same way popped up nicely.

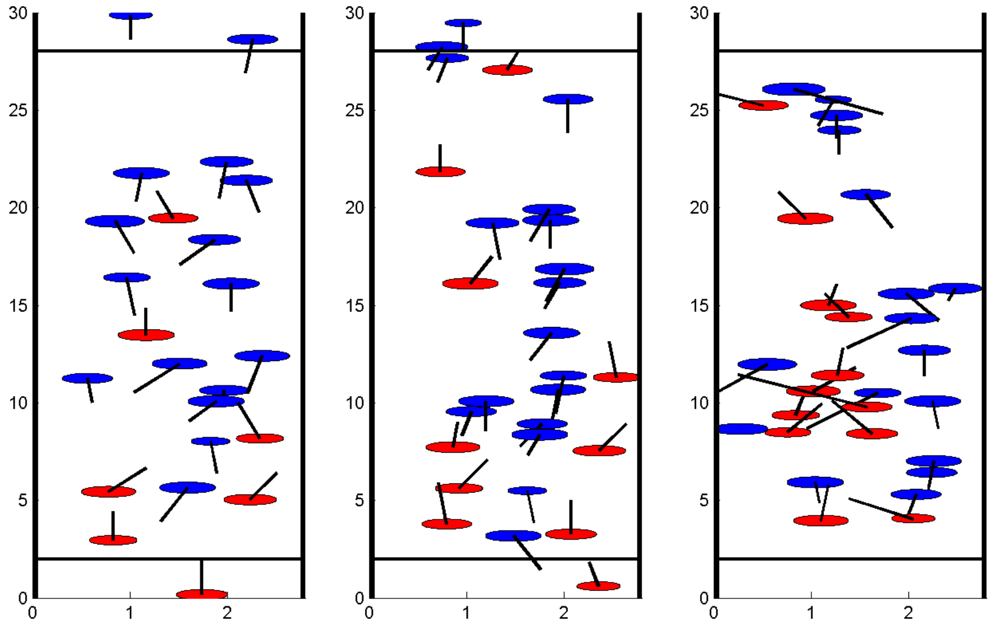

Figure 21: Exemplary pictures for the simulation of the long narrow hallway in the Zurich main station with highly unbalanced flux densities. Red agents are walking up and blue down, the black line denotes the actual velocity vector with its angle and length. The one on the left was done with flux densities of 1.0/0.2, the one in the middle with 1.0/0.3 and the one on the right with 1.0/0.4 after a simulation time of 180 seconds. The red agents (minority) were wandering about quite strong and usually formed small lanes.

7.4 Discussion

7.4.1 Simulations

Overall, we were quite pleased with the results we got from our simulations as they mostly fulfilled our expectations. We could underscore the importance of the choice of a good set of parameters for our model to succeed. It is also possible, as in the case of `DISPFACTOR`, that parameter changes can change the result drastically.
To address the question whether our model is a good description of the reality even if it cannot decide on itself whether it should start something like a cooperative mode including lane formation or adapt an egoistic approach, we would like to state that the model is only as intelligent as the one who uses it. We leave it to the user of the simulation to set reasonable (and therefore also realistic) values for the global parameters which should match the situation one would like to research.

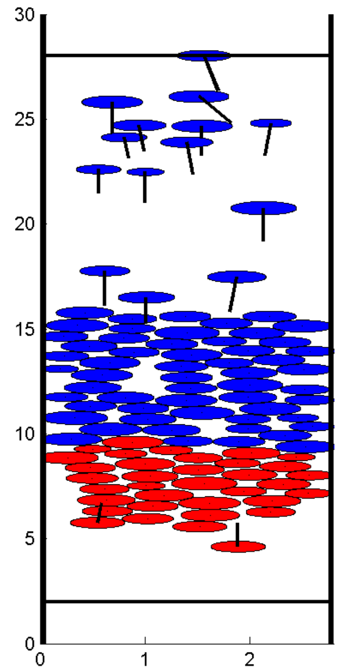

Figure 22: Exemplary pictures for a massive jam in our model. Red agents are trying walking up and blue down, the black line denotes the actual velocity vector with its angle and length. Parameters were flux densities 0.4/1 with a seed of 51. Our model has no way to resolve a jam like that.

The main simulations concerning the modelling of an actual situation was very satisfying as it performed at least as good as reality. A criticism onto our implementation of the main station in Zurich could be that the flux of people arriving is always kept constant. People familiar with the main station in Zurich would know that there is a pedestrian traffic light just in front of the Burger King, therefore the simulation should probably be adapted to allowing intermittent people spawning with different people flux densities just after the red or green light at the traffic light.

It should be highlighted that we saw a big overall performance improvement once we introduced the preference for lane formation. This can be interpreted as implementing a social norm which states that one should back down a little bit from the egoistic main goal of crossing as fast as possible but work together in order to get an acceptable result for everyone. Also, it is similar to the phenomenon sometimes observed that people have the tendency to walk on the right hand side of their walking direction, used to this by traffic.

7.4.2 Discussion on various implementational issues

As we created and implemented our model from scratch, there are obviously some undealt issues that would need refinement if one wants an even better performance of the simulation as such.

Probably the most important issue of all would be the need of implementing a smarter way to resolve standoffs or detect them earlier. We reckon that a good implementation of this should prove to be rather difficult as one has to distinguish between various cases with various ways to resolve them. This is visualized in figure 23 in the right graph. As soon as two (or more) agents totally immobilize each other, they are prone to form a jam.
One would probably have to introduce walking backwards to resolve these situations. The effect of not being able to walk backwards is given in figure 23 in the left graph on the top in a very instructive way. Three red agents were able to totally freeze the simulation at the top, something that would never happen in reality.
Another advantage of a good implementation would be that the runtime on the machine would not explode as it does with the current implementation as soon as a jam has formed. This is rooted in the consideration that all agents within a certain radius shall be considered. Then the logical routine would do calculations over a lot of other agents, even though the agent will not be able to move anyway as he is stuck in a jam.

We used two different axis, a x- and a φ-axis as we thought this would make it significantly easier to model various aspects without the trouble of doing the transformation on paper and only use the φ-axis all the way. Also all angular values had to be discretized to their closest values the φ-axis has, which was done with the function *closest.m*. This works fine as long as the simulation runs smoothly, but as soon as a jam was formed, the number of calls of *closest.m* exploded. In one case, it was called over 2 million times for 1200 iteration steps in a simulation that had a matlab runtime of about 2 hours (with 4 Matlab instances running parallel).

It rarely happens during a simulation that an agent is in an impossible position. This is determined in the collision detection because the first of all calculated points is the position the agent stands on before moving. This was easier to implement and probably not significantly worse in system runtime than excluding it. In rare occasions, the collision detection determines that the agent could not stand at its

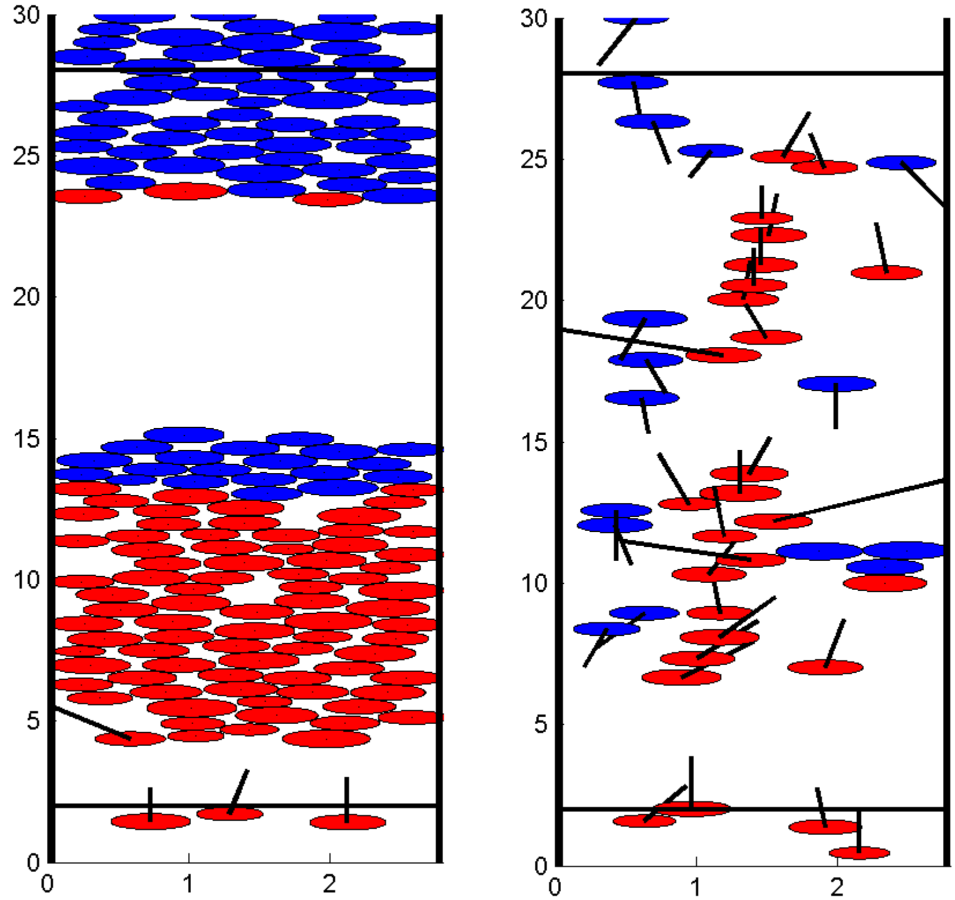

Figure 23: Graphical examples of instructive failures. Graphs were taken from the simulation series to test `DISPERSIONFACTOR`. In the left graph, we have in the top region of the hallway the unrealistic situation where three agents block the whole hallway. On the right, the earliest stage of a jam is visible with the three blue and one red agent standing in the middle at the right wall. This situation won't resolve and so, a jam will start because agents walking into them from behind will get stuck too.

actual position. We think that this happens due to some small roundoff errors or other computational mistakes.
If the agent is at an impossible position, its position would freeze and cause everything around him not to work properly anymore. We could not think of an effective algorithm to resolve these conflicts because they would probably involve setting back the simulation by some iteration steps, so we decided to simply delete such an agent.

We knew that it was not an elegant solution but could settle on the argument that the simulation would otherwise crash. And as the frequency of this occurrence is really small, we also thought that the number of disappearing agents could be neglected. For long simulations, this may not hold true any longer.

It is worth mentioning that an optimization of all the global parameters is quite difficult as they are anything but independent of each other. Therefore we settled on parameters that seemed to work quite well without an extensive testing.

8 Summary and Outlook

First of all, we can say that our simulation series showed that our own model works well. It may have its limitations we're well aware of, but it runs properly what it's supposed to do. A very nice point about our simulation is that it can be applied onto other problems quickly as for example by adding walls or specifying agents.

As for our fundamental research questions, we could investigate most of what was our interest. We showed in subchapter 7.3.4 that wider hallways lead to better flux, if there are lots of jams. This is in good accordance with reality as for the Christmas market, Nordsee and Imagine put away their tables and chairs which leads to a good passenger flux even if there are lots of people.
We did not do lots of our originally planned simulations on specified agents like big/small, fast/slow agents because it turned out to be far more interesting to work with Gaussian distributed agents while investigating the dependency between the passenger flux and other influences as the hallway's width, the flux density and so on. This came also with the need of optimizing the whole set of parameters necessary for the simulation.
Our simulations highlighted the importance of social norms like lane formation as a way to scale down egoistic interests for the benefits of the community (subchapter 7.3.2).

Our own kind of intelligence, the agent being able to look ahead, worked out quite nicely. If there are not too many other agents, they'll find a way to transit the hallway without crashing, which is mostly what a commuter's thoughts are like.
To sum up, we mostly found what we were looking for: A wider hallway leads to better pedestrian density, and egoists trying to overtake everybody can cause jams. In reality, we can't do much about all this, but it was nice having our thoughts confirmed.

If one would want to go further with this project, there are some points clear enough where to start, but hard to do: It's to un-simplify our model. For example, insted of only walking forwards with an almost 180° vision could be changed to 360° of possibilities, but this would also require much more sophisticated path finding algorithms and couldn't be done that fast.
One thing we did not bear in mind until we were in a very final state of the coding process was the performance of our simulation. First tests showed that looping trough arrays, even if they are only a few 1000 entries large, is very slow. Following this, we could speed up the drawing part of our simulation by almost 200%. If one wants to simulate bigger environments with a huge number of agents and wall points,

a faster approach of the iteration implementation should be taken into consideration.

Another point one might be interested in would be to embed groups into the model as in reality, groups of people flocking together are a common view. This would imply some kind of sticking-together algorithm such that the groups wouldn't be torn apart.
Another challenge would be to try to improve our weighing function. As seen in subchapter 7.3.3, our weighing function is a bit too focussed on agents very close to other agents and thinks less about agents in some distance.
In comparison to those improvements, adjusting the parameters to work better as an improvement almost seems easy. But also there, just the question "what is better and why?" can often not be answered clearly.

9 References

[1] D. Helbing, P. Molnár, *Social force model for pedestrian dynamics*, Phys. Review E, Volume 51-5, 4282-4286, 1995

[2] I. Steinacher*Main Station Situation Sketch*, 27.11.2012

[3] K. Briner, M. Marti, T. Meier, *Train jamming*, Zürich, 16.12.2012 https://github.com/msssm/Train_Jammin (13.12.2012)

10 List of figures

List of Figures

Appendix

A MATLAB HS2012 - Research Plan

Version info: the submitted and approved version, 2012-10-24 17h

- **Group Name**: Mayara
- **Group participants names**: Moser Manuel (Mathematics BSc, 3rd Sem), Suter Yannick (Chemistry BSc, 5th Sem), Theiler Raffael (Informatics BSc, 3rd Sem)
- **Project Title**: Pedestrian dynamics in long, narrow hallways

General Introduction

Annoyed by people rushing through the small corridor left in Zurich main station hall (the path between burger king and groups meeting point) during the Oktoberfest, market days, concerts and other occasions, we decided to have a look at pedestrian dynamics in hallways which are mostly crowded and narrow (3-4 meters in breadth) compared to normal days when the hall is empty, and where people walk through in opposite directions all the time. We want to have a look at how the pedestrian flux can be improved and how the walk-through time behaves during rush-hours, but also in the case of much more persons moving in one direction than in the other. Also, we want to have a look at the influences of aggressive, fast people in a rush, slowly moving obstacles like mothers with baby buggies and some drunkards, and try to figure out how to avoid jammings. Maybe we'll also implement a static obstacle to observe what happens. The simulation of problems like this will also help understand the phenomena of group dynamics which usually control and resolve such problems in real life.

The Model

We want to do an agent-based simulation of people moving through a long corridor (dimensions will be proportional to those encountered in our object, the Zurich main station). The people will primary want to move forwards at different speeds but also be able to move diagonally or even sideways if needed. A nice thing will be to try implementing agents being able to see some fields/meters ahead whether their path (assumed straight as long as possible) is free or not, and if they're about to crash into someone, try to avoid them. Independent variables in our model are the amount of people per time arriving, the corridor and its obstacles and the characteristics of the agents like walking speed and aggressiveness. Dependent variables will be the amount of people leaving, which should in the end determine whether the people will be stuck or if they can get through. Should the amount of people leaving be smaller than those arriving (per time unit), one can expect a blockage. As a reference, we will use a simulation of a corridor without any obstacles and only few people. Then the collective success or failure of an other situation can be compared to this.

Fundamental Questions

- We try to simulate the pedestrian flux in the following different situations: Rush hour (danger of jamming), with an static obstacle, with aggressive/very fast or slow agents, random path agent (drunkard). Will the pedestrian flux run smoothly or will they block each other and be stuck?
- Will the implementation of a rudimentary kind of thinking/looking ahead help to avoid blockages? If possible, we may determine the limits for which the goal of passing is achieved with and without this implementation and compare them.
- Will there be group dynamics or similar behaviors of agents if they're only programmed to walk to the other side, each on his own?

Expected Results

We think that there will be lots of walking around left/right while trying to avoid other agents, and with rising amount of agents there will be more jams, this seems obvious. We think that in our simulation we'll have to deal with massive jams because the agents are not communicating with each other in any way. Implementation of "looking ahead" will probably improve the people flux but only to a limited range. Obstacles will also lead to more jams, whilst the drunkard simulation will for the amusement of our group.

References

Just some ideas where to get inspiration from:

- Project Suggestions - 16 - Pedestrian Dynamics - 5 papers - http://www.soms.ethz.ch/teaching/MatlabFall2012/projects/16-Pedestrian_Dynamics.zip - (01.10.2012)
- Mehdi Moussaid Publications - http://mehdimoussaid.com/publications.html (24.10.2012)

- Crowd-Flow-Optimization - FS2012 - https://github.com/nfloery/crowd-flow-optimization (01.10.2012)
- Train Jamming - FS2011 - https://github.com/msssm/Train_Jammin (01.10.2012)
- Airplane Evacuation / FS2011 https://github.com/msssm/Airplane_Evacuation_2011_FS (01.10.2012)

Research Methods

For our project, an agent-based model is the most satisfying because there we can really implement different speeds and directions. A disadvantage will be the complicated collision handling.

Other

For the measurements of the corridor, we'll go to the main station and measure it one afternoon when it's not fully crowded. We also could count the rate of incoming and leaving people during a rather relaxed afternoon and a crowded rush-hour.

B Additional figures

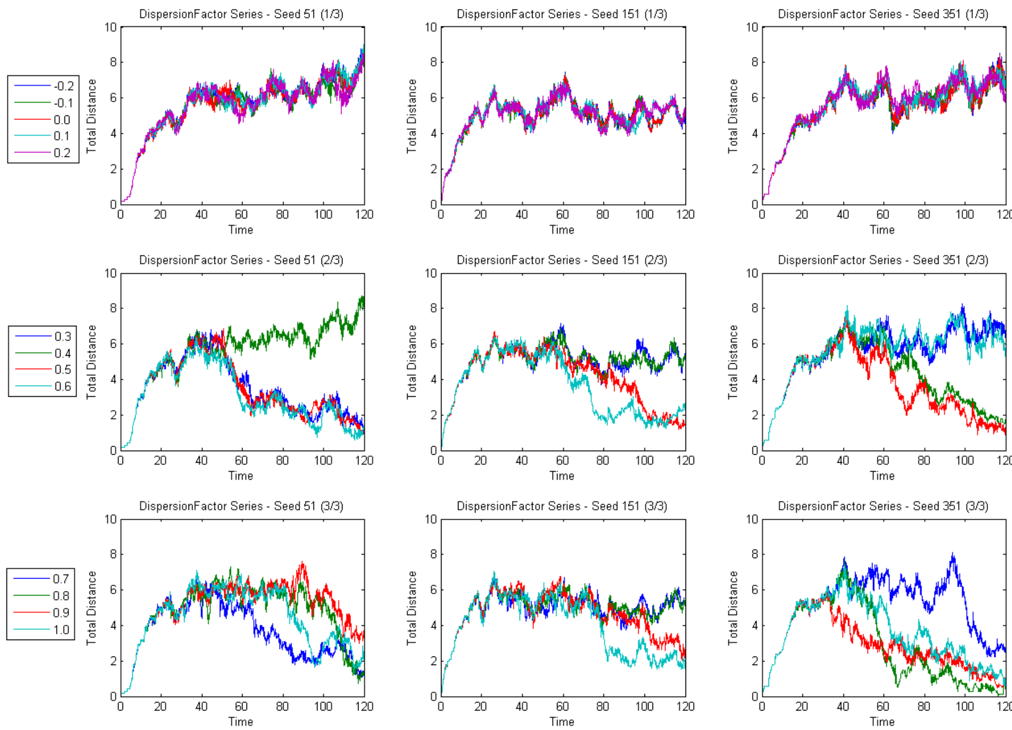

Figure 24: Another representation of figure 14, split up into a 3×3 composition for a better view on the single lines.

C Matlab source code

```

%Run.m is a runscript for our project. It cleans up the workspace first
%and then reinitialises a new copy of the simulation.m environment.
%This copy can be accessed trough the sim variable. See the documentation
%about the simulation.m to find out which modes are supported.

%Example run:

%run

%Which runmode?
%Input: normal / fasttest ... [optional parameters]
%Save output [Y/N]? Y
%Write data to:
%Input: [file]

%The programm should respond after this with some details about the
%simulation:

%Running normal mode:

%And some time about the Simulation time:

%Actual time on system: 15 h 39 min 4.424000e+00 sec
%Time elapsed: 12 Seconds

%Note: The time elapsed can vary with the perfomance of the host system!

%After the simulation a "profile view" opens the profiler data to measure
%the perfomance of the code, and the measurement results are available
%trough different matrices inside of the sim object.
%They can be used with a simle sim.[matrice] command.

profile off
profile on
clear
clc
defineConstants();
sim = simulation();
sim.init()

mode = input('Which runmode? ','s');
ifSave = input('Save output [Y/N]? ','s');
if ifSave == 'Y'
    outName = input('Write data to: ', 's');
    copyfile('defineConstants.m',['sim/' outName '.txt']);
    fileID = fopen(['sim/' outName '.txt'],'a+');
    fprintf(fileID,['\nMode utilized: ' mode]);
    fclose(fileID);
end

sim.runMode(mode);

if ifSave == 'Y'
    save(['sim/' outName '.mat']);
```

```
    sim.draw.plotStep();
    print('-dpng',['sim/' outName '.png']);
    %load(<pfad>)
end
```

```
function [] = defineConstants()
%   defineConstants defines all global constants that will be used later
%
%   Change the constants with caution as the simulation can be very
%   sensitive to some changes. After the variable definition, a reliable
%   value is given with a short description of the influence of the
%   constant variable on the simulation. Consult the documentation for a
%   detailed description of the functioning of these constant variables.
%   ntbc* stands for not to be changed. Changing them anyway will probably
%   lead to a complete breakdown of the simulation.
%
%   WARNING: The standard deviation for speed and radius has to be at least 3 times smaller than
%   the corresponding mean. Otherwise unlogical results in the form of
%   negative numbers will appear. This is not checked during the simulation.

    global ANGLE GAUSSANGLE XSCOPE XRES XVALUES DELTAT...
        HEIGHT SLOPEFACTOR WALLFACTOR AGENTANGLEOFFSET...
        WALLANGLEOFFSET INFLUENCESPHERE PRECISIONCOLLISION...
        SPEED MEANRADIUS STDRADIUS MEANSPEED STDSPEED WIDTH...
        YSPB1 YSPB2 YSPT1 YSPT2 REP DENSITYUP DENSITYDOWN...
        REPULSIONAGENT STANDOFF SEED DISPERSIONFACTOR LOOPS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Name & Value......................value.Description.......................
    XSCOPE = 120;                     % 120      How far one looks to the sides
    XRES = 0.04;                      % 0.04      x-Resolution, possible angles for logical functions
    XVALUES = -XSCOPE:XRES:XSCOPE;    % ntbc*     Generate all possible x-values
    breite = 1;                       % 1         Standard deviation of Gaussian function determines the tendency to walk straightforwards
    GAUSSANGLE = gaussmf(XVALUES,[breite 0]); % ntbc* Generate Gaussian function for direction
    ANGLE = atan(XVALUES);            % ntbc* Calculate angular values for Gaussian function

    HEIGHT = 4;                       % 4     Normalization factor in logical calculations for agents
    SLOPEFACTOR = 8;                  % 8     Parameter for other agents' influence
    REPULSIONAGENT = 2;               % 2     Parameter for the repulsion between two agents. The higher it is, the higher HEIGHT has to be
    WALLFACTOR = -0.05;               % -0.05 Parameter for walls' influence
    AGENTANGLEOFFSET = pi/14;         % pi/14 Offset angle in angle calculations for agents
    WALLANGLEOFFSET = pi/20;          % pi/20 Offset angle in angle calculations for walls
    INFLUENCESPHERE = 2.5;            % 2.5   How far an agents "looks", in meters
    PRECISIONCOLLISION = 4;           % 4     Numerical precision for collision detection
    STANDOFF = -0.5;                  % -0.5  "Strength" to resolve standoffs, must be negative
    DISPERSIONFACTOR = 0.7;           % This constant variable determines the willingness of the agents to follow an agent in front of him walking in the same direction
                                      % -0.2  A negative value will cause the agents to form lanes
                                      % 0     No influence of the agent in front
                                      % 0.3 or 0.7    The agents will try to overtake a slower agent in front of them. Prone to
```

```
                                     the formation of jams for high values
    DENSITYUP =0.4;                   % 0.4   Flux density of people spawned on the
        upper line (in people/sec)
    DENSITYDOWN = 0.4;                % 0.4   Flux density of people spawned on the
        lower line (in people/sec)

    MEANRADIUS = 0.25;                % 0.25  Mean radius of the agents
    STDRADIUS = 0.03;                 % 0.03  Standard deviation of the distribution
        of radii
    MEANSPEED = 1.5;                  % 1.5   Mean speed of the agents in meters/
        second
    STDSPEED = 0.25;                  % 0.25  Standard deviation of the distribution
        of speeds

    WIDTH = 2.8;                      % 2.8   Width of the field
    YSPT2 = 30;                       % 30    Spawn line of the upper agents
    %YSPT2-----------
    %YSPT1-----------
    YSPT1 = 28;                       % 28    Arrival line for agents walking up
    %
    %Field here
    %
    YSPB2 = 2;                        % 2     Arrival line for agents walking down
    %YSPB2-----------
    %YSPB1-----------
    YSPB1 = 0;                        % 0     Spawn line of the lower agents

    DELTAT = 0.1;                     % 0.1   Iteration steps in seconds
    REP = 10;                         % 10    Number of tries to spawn a new agent
    LOOPS = 1800;                     % 500   Number of iteration steps
    SEED = 113;                       %       Seed for random number generator
    SPEED = 0.001;                    % 0.001 Time delay between two iterations in
        seconds
end
```

```matlab
classdef simulation < handle
    %This Class contains all the objects and data needed and generated for
    %a simulation. its generated by the run script.

    %simulation allows different modes
    %Mode: normal - This mode does a slower calculation with many wall
    %points to calculate precise agent positioning
    %Mode fasttest - This mode uses less wall points for the calculatin as
    %a result it is faster but agents can shake a little bit

    %Parameters:
    % -nodirections: Disables the direction line of the agents for a better
    % general overview

    % -nograph: Does what it says.. A non graphical mode to save simulation
    % time

    % -report: This parameter lets the simulation generate more detailed
    % output code


     properties
         %Iteration properties
         loops;


         %Objects
         draw;                %The drawing object
         agentSize = 200;     %Defaultwert 200 kann mit Funktion berechnet werden

         %Booleans
         drawGraph = 1;               %Draw graph decision boolean
         calcAdditionalReport = 0;    %Additional report boolean
         %Results

         spawned;                     %tracks how many agents are spawned
         result;                      %the most important Data from iteration.m
         additionalresult;            %more details aout the agents,
                                      %calculated in simulation.m
         evaluateDistance;            %distance evaluation of the iteration.m
         evaluateTime;                %Time evaluation of the iteration.m

         %Properties
         agentMinRadius = 0.25;       %Minimaler Radius von einem Agent
         agentMaxSpeed;               %Used to initialize agents max speed


    end

    methods

         %Function: Constructor, constructs the sim object and reads
         %the loops from the global declarations.
         function obj = simulation()
             global LOOPS
             obj.loops = LOOPS;
             obj.spawned = zeros(2,obj.loops);
         end
```

```
    end

%----------------------------------------------------------------
%Public:

    methods(Access = public)

        %Function: Initialize the simulation object.
        %This Function allocates memory for some arrays and initializes the
        %Agents
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        function obj = init(obj)
            defineConstants(); %defne all global variables
            global SEED DENSITYUP DENSITYDOWN DELTAT
            rng(SEED);   % Set seed for random number generator
            obj.draw = drawing.empty(1,0);
            obj.draw = drawing();   %create the drawing object
            calcPossibleAgents(obj);
            obj.evaluateDistance = zeros(1,ceil((DENSITYUP+DENSITYDOWN) * DELTAT * obj.loops + 30));
            obj.evaluateTime = zeros(1,ceil((DENSITYUP+DENSITYDOWN) * DELTAT * obj.loops + 30));

            obj.draw.agentArray = agent.empty(ceil((DENSITYUP+DENSITYDOWN) * DELTAT * obj.loops + 30) ...
                ,0);


%               obj.draw.agentArray = agent.empty(obj.agentSize ...
%                   ,0);
        end


        %Function: This Function converts the input parameters to some
        %project specific booleans to enable or disable some functions or
        %to modify the speed of the simulation
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        %   mode    - the mode string to parse
        function obj = runMode(obj,mode)

               if(isempty(strfind(mode, '-nodirections')) == 0)
                   disp('MODE: keine Richtungsanzeige');
                   obj.draw.drawP = 0;
               end

               if(isempty(strfind(mode, '-nograph')) == 0)
                   disp('MODE: kein Graph');
                   obj.drawGraph = 0;
               end


               if(isempty(strfind(mode, '-report')) == 0)
                   disp('MODE: zusätzlicher report');
                   obj.calcAdditionalReport = 1;
               end

```

```
                if (isempty(strfind(mode, 'normal')) == 0)
                    obj.draw.particleDensity = 4;
                    obj.draw.createWall();
                  disp('Running normal mode:');
                  run(obj);
                elseif (isempty(strfind(mode, 'fasttest')) == 0)
                  disp('Running fast mode with reduced wall points:');
                  obj.draw.particleDensity = 1;
                  obj.draw.createWall();
                  run(obj);
                else
                  disp('unknown mode');
                end
        end


        %Function: This function calculates more details about the agents
        %Enable it with the parameter -report
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        %   step    - the current ime step
        %Results: (saved in additionalResults)
        %row1: total distance traveled during this step
        %row2: sum of agents in the system
        function obj = additionalReport(obj, step)
            global DELTAT;

            sortedPrioArray = getSortedPriorityArray(getPriorityArray(obj.draw.
                agentArray));
            for i = sortedPrioArray;
                way = abs(obj.draw.agentArray(i).actSpeed)*DELTAT;
                obj.additionalresult(1,step) = obj.additionalresult(1,step)+ way; %
                    Gesammtweg aufsummieren
            end
            obj.additionalresult(2,step) = size(sortedPrioArray,2);
        end

    end
%-----------------------------------------------------------------------
%Private:

    methods(Access = private)

        %Function: This function calculates the maximum of possible agents
        %for the current playfield this is used to predefine the array of
        %agents
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        function obj =  calcPossibleAgents(obj)
            obj.agentSize = floor((obj.draw.width*obj.draw.length)/...
                (3*obj.agentMinRadius^2));
        end

        %Function: This Function returns -1 or +1 randomly (used to
        %randomize the spawn , top or bot
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        %Result:
```

```
        %   randOut -the number(-1 or +1)
        function randOut = randPrefix(obj)
           randOut = randi(2);
           if (randOut == 2)
               randOut = 1;
           else
               randOut = -1;
           end
        end


        %Function: This function fills up the agent array with new agent
        %objects. All agents have priority 0, which means they exist but are not
        %into the actual calculation.
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        function obj = initialSpawn(obj)
            sze = size(obj.draw.agentArray);

            for i = 1:(sze)
                %For testing
                %obj.draw.length
                %obj.draw.width
                %obj.draw.agentArray(i) =  agent(0.25,obj.draw.width*rand()...
                %     ,obj.draw.length*rand(),10000,1)

                obj.draw.agentArray(i) = agent(0.01, 0 ,0  ,...
                    obj.agentMaxSpeed, 0);
            end

        end


        %Spawne einen neuen Agent
%          function obj = addNewAgentsToArray(obj, currentStep)
%              if (balanceProbability(obj) == 1)
%                  pfx = randPrefix(obj);
%                  if(pfx == 1)
%                      %Unten gespawnt
%                      obj.spawned(2,currentStep) = obj.spawned(currentStep) +1;
%                  else
%                      %Oben gespawnt
%                      obj.spawned(1,currentStep) = obj.spawned(currentStep) +1;
%                  end
%                  spawn(obj.draw.agentArray,pfx);
%              end
%          end


        %Function:This function spawns new agents into the simulation. it
        %uses the spawn.m to randomly get a new free position. If and where
        %the agent is spawned depends on the probabilities (See globals)
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        %   currentStep -the currentStep of the simulation, used for the
        %   spawned report function
        function obj = addNewAgentsToArray(obj, currentStep)
            [probOben probUnten] = balanceProbability(obj);
```

```
            if (probOben == 1)
                obj.spawned(1,currentStep) = obj.spawned(currentStep) +1;
                spawn(obj.draw.agentArray,1);
            end
            if (probUnten == 1)
                obj.spawned(2,currentStep) = obj.spawned(currentStep) +1;
                spawn(obj.draw.agentArray,-1);
            end
        end


        %Function: If the threshold from the globals (can be different for
        %top and bottom) is reached it returns true, otherwise false. This
        %results in a new agent trough the addNewAgentsToArray() function.
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        %Result:
        %   probOben - boolean if its necessary to spawn at top
        %   probUnten - boolean if its necessary to spawn at bottom
        function [probOben probUnten] = balanceProbability(obj)
            global DENSITYUP DENSITYDOWN DELTAT
            %Genähert kommen genau im schnitt density agents
            if (rand(1) > 1-DELTAT*DENSITYUP)
                probUnten = 1; % Von unten nach oben
            else
                probUnten = 0;
            end
            if (rand(1) > 1-DELTAT*DENSITYDOWN)
                probOben = 1; % Von oben nach unten
            else
                probOben = 0;
            end
        end


        %Function: Main procedure, this function contains the loop trough
        %the smulation. It calls all the other functions in the project
        %such as draw the map and add new agents or determine the direction
        %and the speed of the existing agents.
        %   Object  - the object of this instance (matlab specific)
        function obj = run(obj)
            global SPEED DELTAT;
            obj.result = zeros(2,obj.loops);
            %Speicher für additionale resultate
            if(obj.calcAdditionalReport == 1)
                obj.additionalresult = zeros(2, obj.loops);
            end

            initialSpawn(obj);

            for i = 1:obj.loops
                if(mod(i,floor(DELTAT * obj.loops)) == 0)
                    c = clock;
                    fprintf('Actual time on system: %d h %d min %d sec\n',c(4),c(5),
                        c(6));
                    fprintf('Time elapsed: %i Seconds \n',(DELTAT*i))
                end
```

```
                    [obj.result(1,i), obj.result(2,i), obj.evaluateDistance, obj.
                        evaluateTime] = ...
                        Iteration(obj.draw.agentArray,...
                        obj.draw.wallArray, obj.evaluateDistance, obj.evaluateTime);


                    addNewAgentsToArray(obj, i);
                    pause(SPEED);

                    if(obj.drawGraph == 1)
                        obj.draw.plotStep();
                    end

                    if(obj.calcAdditionalReport == 1)
                        additionalReport(obj, i);
                    end

                end

                ind = find(obj.evaluateDistance == 0,1);
                if (size(ind,2) ~= 0) || (ind ~= 1)
                    obj.evaluateDistance = obj.evaluateDistance(1:(ind-1));
                end
                ind = find(obj.evaluateTime == 0,1);
                if (size(ind,2) ~= 0) || (ind ~= 1)
                    obj.evaluateTime = obj.evaluateTime(1:(ind-1));
                end


            end



        end

    end
```

```
%This class can draw the current agent positions onto a graph. The agents
%are simplified as cyrcles. Every agent has another line indicator which is
%the speed and the angle of its movement.

%The drawing class contains the reference to the array of agents. It also
%generates the wall agents direct in its contructor.

classdef drawing < handle
    properties(SetAccess = public, GetAccess = public)
        %Details about this plot
        title = 'Plot';              %not implemented
        xAxisTitle = 'xAxis';        %not implemented
        yAxisTitle = 'yAxis';        %not implemented
        xStretchFactor = 5;          %stretches the x-axis to get a better overwiew

        %care: if stretched, the cycles will apear as an oval!


        particleDensity = 10;        %wall agent density
        wallRadius = 0.005;          %diameter of the wall = 2x wallRadius
        drawP = 1;                   %P = Direction of Attraction =)

        %Field definitions:
        width = 2.8;                 %in meters
        length = 30;                 %in meters
        wallDia = 0.05;              %in meters

        %Note difference beween wallDia and wallRadius:
        %Radius is used for the agents, diameter for the drawing of the
        %wall.

        %the spawn-zone:

        %we need this zone because otherwise agents could be spawned direct
        %on top of arriving agents from the other side.
        spawnZoneDistanceTop = 2.5; %in meters
        spawnZoneDistanceBot = 2.5; %in meters

        %Data
        wallArray;               %reference to the wall agents
        agentArray;              %reference to the agents

        %Testdata
        %Used to test the drawing
        testAgents;
        testWall;
        activateTesting = 0;
    end


    methods
        %Constructor
        %Function: Creates a new drawing object. For this task,
        %several global constants are loaded into the object.
        %Also creates random agents if the testing mode is activated.
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        function obj = drawing()
```

```matlab

            %Setup the Playfield
            global YSPT2 YSPT1 YSPB1 YSPB2 WIDTH
            obj.width = WIDTH;
            obj.length = YSPT2;
            obj.spawnZoneDistanceTop = YSPT2 - YSPT1;
            obj.spawnZoneDistanceBot = YSPB2 - YSPB1;

            %Create testagents
            if(obj.activateTesting)
                obj.testAgents = agent.empty(200,0);

               for k=1:200
                   obj.testAgents(k)= agent(0.25,obj.width*rand(),obj.length*rand(),10000,1);

               end
              obj.agentArray = obj.testAgents;

            end
            createWall(obj);
        end

        %Set the agents
        function obj = set.agentArray(obj, value)
            obj.agentArray = value;
        end

        %Set the wall
        function obj = set.wallArray(obj, value)
            obj.wallArray = value;
        end

    end

    methods(Access = public)


        %Function: This Function creates the wall dummy - agents
        %The positioning and the ammount are dynamicaly calculated based on
        %the particle density (constant)
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        function obj = createWall(obj)
            obj.wallArray = agent.empty(2*obj.length*obj.particleDensity,0);
            for k=1:(obj.length*obj.particleDensity)
                %left wall
               obj.wallArray(2*k-1)= agent(obj.wallRadius,0,((k-1)/obj.particleDensity),0,0);
               obj.wallArray(k*2)...
               = agent(obj.wallRadius,obj.width,((k-1)/obj.particleDensity),0,0);
            end%end create wall
        end


        %Function: This function plots one timestep onto the selected graph.
        %To create an animation call this function in a for loop and
```

```
        %implement some agent logic. (for an example implementation see simulation.m ->
        %the run function.
        %Variables:
        %   Object  - the object of this instance (matlab specific)
        function obj= plotStep(obj)

            %Clear the graph window
            clf;


            sizeA = size(obj.agentArray,2); %Determine the size of the agent array
            sizeW = size(obj.wallArray,2);  %Determine the size of the wall agent array
            if obj.activateTesting
                sizeA
                sizeW
            end
            %1 = X coordinates
            %2 = Y coordinates
            %3 = color

            %For wall painting use this:
            %coords = zeros(sizeA+ sizeW,4);
            for i = 1:sizeA
                 if(sign(obj.agentArray(i).maxSpeed) == -1)
                    color = 'blue';
                else
                    color = 'red';
                 end

                %Draw only if the priority of the agent != 0
                %Priority 0 means that this agent is inactive,
                %see agent.m
                if(obj.agentArray(i).priority ~= 0)
                circlePlot(obj, obj.agentArray(i).cordX,...
                   obj.agentArray(i).cordY,...
                   obj.agentArray(i).radius,...
                   color);


              %Draw the direction indicator lines
              %The line is longer if the agent is faster and points to the
              %direction the specific agent wants to move.
               if(obj.drawP == 1)
                   drawLine(obj, obj.agentArray(i).cordX,...
                       obj.agentArray(i).cordY,...
                       obj.agentArray(i).cordX+...
                       sin(obj.agentArray(i).angle)*obj.agentArray(i).actSpeed,...
                       obj.agentArray(i).cordY+...
                       cos(obj.agentArray(i).angle)*obj.agentArray(i).actSpeed);
               end
             end
            end


%             for i = (sizeA+1):(sizeA+sizeW)
%                 coords(i,1)=obj.wallArray(i-sizeA).cordX;
```

```
%                   coords(i,2)=obj.wallArray(i-sizeA).cordY;
%               end
%

            %Setup the graph window: This is matlab specific code and has
            %nothing to do with the implementation.
            xlim([0,obj.width])
            ylim([0,obj.length])
            daspect([1,obj.xStretchFactor,1]) %stretch the window

            %The wall drawing:
            drawWallSquares(obj, obj.width, obj.length, obj.wallDia);

            %Draw start and end lines:
            %Line at the bottom
            drawLine(obj, 0+obj.wallDia...
                , obj.spawnZoneDistanceBot, obj.width- obj.wallDia, ...
               obj.spawnZoneDistanceBot);

           %Line at the top
           drawLine(obj, 0+obj.wallDia...
                , obj.length-obj.spawnZoneDistanceTop, obj.width- obj.wallDia, ...
               obj.length-obj.spawnZoneDistanceTop);
        end

    end

     %Those methods are simple function calls from the matlab api to
     %draw all the geometric shapes:
     methods(Access = private)


        %Function:  Draws side wall squares with the given parameters
        %Variables:
        %   width   - the width of the field
        %   height  - the height of the field
        %   wallDi..- the desired wall diameter
        %   Object  - the object of this instance (matlab specific)
        function obj = drawWallSquares(obj, width, height, wallDiameter)
            rectangle('Position',[0,0,...
                wallDiameter,height],...
               'FaceColor','black');

           rectangle('Position',[width-wallDiameter,0,...
                wallDiameter,height],...
               'FaceColor','black');
        end

        %Function: Draws a circle with the given parameters
        %Variables:
        %   x       - the x coordinate of the center
        %   y       - the y coordinate of the center
        %   radius  - the radius of the circle
        %   color   - fill color of the circle
        %   Object  - the object of this instance (matlab specific)
        function obj = circlePlot(obj,x, y, radius, color)
            rectangle('Position',[x-(radius),y-(radius),...
                radius*2,radius*2],...
```

```
                'Curvature',[1,1],...
                'FaceColor',color);
        end

        %Function:   draws a single line with the given parameters
        %Variables:
        %   x1      -x coordinate of endpoint 1
        %   y1      -y coordinate of endpoint 1
        %   x2      -x coordinate of endpoint 2
        %   y2      -y coordinate of endpoint 2
        %   Object  - the object of this instance (matlab specific)
        function obj = drawLine(obj, x1,y1, x2,y2)
           array = [x1,y1;x2,y2];
           line(array(:,1),array(:,2), [0;0],'Color','black', ...
                'LineWidth', 2);
          % get(l)
        end


    end
end
```

```
classdef agent < handle
%   The class agent represents the agents used in the simulation. An agent
%   is basically a container for all properties an agent needs to interact
%   with other agents.
%
%   A short explanation is given behind each property. See the
%   documentation for a more complete descrition of them.
%
%   The first six properties are crucial for the success of the simulation.
%   The latter three are used for monitoring and analysis.

    properties(GetAccess = public, SetAccess = public)
        radius   %   "Size"(radius) of an agent
        cordX    %   Actual x-coordinate
        cordY    %   Actual y-coordinate
        maxSpeed %   Maximal velocity the agent wants to reach. + mean walking up, -
            walking down.
        actSpeed %   Actual velocity
        priority %   Priority of the agent. Determines the order of evalutation
            during the iteration. 0 for inactive agents

        angle    %   Angle of sight, given by the logic functions
        distance %   Covered distance of an agent during his lifetime
        time     %   Time an agent spends in the simulation
    end



    methods(Access = public)
        function obj = agent(radius, cordX, cordY, maxSpeed, priority) % Construktor
            obj.radius = radius;
            obj.cordX = cordX;
            obj.cordY = cordY;
            obj.maxSpeed = maxSpeed;
            obj.actSpeed = maxSpeed;
            obj.priority = priority;
            obj.angle = 0;
            obj.distance = 0;
            obj.time = 0;
        end

        function obj = reset(obj)
            obj.distance = 0; %reset distance
        end

    end

end
```

```
function [topOut,botOut, outDistArray, outTimeArray] = Iteration( agentsArray, wallArray, distArray , timeArray)
%   Funktion ruft die Funktion logicFunction auf
%   Muss mit einer Prioritätenliste auf alle Agents ausgeweitet werden
%   distArray: Speichere die zurückgelegte Weglänge eines Agents wenn er
%   gelöscht wird

    global INFLUENCESPHERE PRECISIONCOLLISION DELTAT YSPT1 YSPB2

    dist = linspace(0,1,PRECISIONCOLLISION);
    topOut=0;
    botOut=0;
    prioArray = getPriorityArray(agentsArray);
    sortedPrioArray = getSortedPriorityArray(prioArray);
    lenSort = length(sortedPrioArray);
    lenWall = length(wallArray);

    for k = sortedPrioArray
        angleShift = logicFunction(agentsArray, k, INFLUENCESPHERE, prioArray, wallArray);

        %Kollisionstest
        xCordNeu = agentsArray(k).cordX + sin(angleShift) * DELTAT * agentsArray(k).maxSpeed * dist;
        yCordNeu = agentsArray(k).cordY + cos(angleShift) * DELTAT * agentsArray(k).maxSpeed * dist;
        distMat = zeros(lenSort+lenWall,PRECISIONCOLLISION+1);
        for l = sortedPrioArray
            if (((agentsArray(k).cordX - agentsArray(l).cordX)^2 + (agentsArray(k).cordY - agentsArray(l).cordY)^2) > (INFLUENCESPHERE * INFLUENCESPHERE)) || (l == k)
                continue
            end
            distMat(l,:) = [(sqrt((xCordNeu - agentsArray(l).cordX).^2 + (yCordNeu - agentsArray(l).cordY).^2) - agentsArray(k).radius - agentsArray(l).radius),-1]; %-1 als Sentinel
        end
        for l = (lenSort+1):(lenSort+lenWall)
            lneu = l-lenSort;
            if (((agentsArray(k).cordX - wallArray(lneu).cordX)^2 + (agentsArray(k).cordY - wallArray(lneu).cordY)^2) < (INFLUENCESPHERE * INFLUENCESPHERE))
                distMat(l,:) = [(sqrt((xCordNeu - wallArray(lneu).cordX).^2 + (yCordNeu - wallArray(lneu).cordY).^2) - agentsArray(k).radius - wallArray(lneu).radius),-1]; %-1 als Sentinel
            end
        end
        distMat = sort(distMat);

        maxL = find(distMat(1,:) < 0, 1) - 1;

        if maxL == 0 %  Sollte eigentlich nicht passieren, dann ist ein Fehler in der Iteration geschehen. Loesche den Agent, da er sich nicht bewegen kann, da ein anderer Agent auf seiner Position steht
            agentsArray(k).priority = 0;
            agentsArray(k).distance = 0;
            fprintf('Ein Agent ist in einer unmöglichen Position. Agent gelöscht\n')
            continue
```

```
        end

        %neue koordinante setzen
        agentsArray(k).distance = agentsArray(k).distance + sqrt((xCordNeu(maxL) - agentsArray(k).cordX)^2 + (yCordNeu(maxL) - agentsArray(k).cordY)^2);
        agentsArray(k).cordX = xCordNeu(maxL);
        agentsArray(k).cordY = yCordNeu(maxL);
        agentsArray(k).angle = angleShift;
        agentsArray(k).time = agentsArray(k).time + DELTAT;

        if (agentsArray(k).cordY < YSPB2 && agentsArray(k).maxSpeed < 0) %von oben nach unten, grenze erreicht
           agentsArray(k).priority = 0;
           agentsArray(k).actSpeed = 0;
           %evaluateAgent(agent);
            botOut = botOut +1;
            test = find(distArray == 0,1);
            if size(test,2) == 0
                fprintf('Distanzarray ist zu kurz\n')
            else
                distArray(test) = agentsArray(k).distance;
            end
            test = find(timeArray == 0,1);
            if size(test,2) == 0
                fprintf('Distanzarray ist zu kurz\n')
            else
                timeArray(test) = agentsArray(k).time;
            end
            agentsArray(k).time = 0;
            agentsArray(k).distance = 0;
            prioArray = getPriorityArray(agentsArray);
        elseif(agentsArray(k).cordY > YSPT1 && agentsArray(k).maxSpeed > 0) %von unten nach oben, grenze erreicht
           agentsArray(k).priority = 0;
           agentsArray(k).actSpeed = 0;
           %evaluateAgent(agent);
           topOut = topOut + 1;
            test = find(distArray == 0,1);
            if size(test,2) == 0
                fprintf('Distanzarray ist zu kurz\n')
            else
                distArray(test) = agentsArray(k).distance;
            end
            test = find(timeArray == 0,1);
            if size(test,2) == 0
                fprintf('Distanzarray ist zu kurz\n')
            else
                timeArray(test) = agentsArray(k).time;
            end
            agentsArray(k).time = 0;
            agentsArray(k).distance = 0;
            prioArray = getPriorityArray(agentsArray);
        else
            agentsArray(k).actSpeed = agentsArray(k).maxSpeed() * (maxL - 1) / (PRECISIONCOLLISION - 1);
        end
    end
```

```
    outTimeArray = timeArray;
    outDistArray = distArray;
end
```

```
function [angleOut] = logicFunction( agentsArray, agentPosition, influenceSphere, priorityArray, wallArray)
%   Funktion berechnet die Richtung, in die ein Agent gehen wird. Mögliche
%   Werte liegen zwischen -pi/2 bis pi/2, wobei 0 nach rechts bedeutet und
%   180 nach links.
%   agentPosition gibt an, für welchen Agent dass es gemacht werden soll.
%   influenceSphere gibt an, wie gross der betrachtete Halbkreis eines
%   Arrays ist
%   Funktion summiert die einzelnen Einflüsse aller anderen Agents, welche
%   innerhalbt der influenceSphere sind. Danach werden die Einflüsse aller
%   Wandagents innerhalb der influenceSphere addiert. Anschliessend wird
%   das Maximum dieser entstehenden Funktion genommen und der entsprechende
%   Winkel zurückgegeben.

%   This function calculates the direction in which an agent will move.
%   Possible values are between -pi/2 to pi/2 whereby 0 means to the right
%   and 180 to the left.
%   agentPosition indicates for which agent it will be done.
%   influenceSphere indicates, how big the regarded semi cycle of an array
%   is.
%   The function summs up the different influences of every agent that is
%   inside of the influenceSphere. After this step the influences of the
%   wall-agents inside the influenceSphere are added.  In the end, the
%   maximum of these resulting functions is calculated and a resulting
%   angle is returned.
    global ANGLE GAUSSANGLE DISPERSIONFACTOR
    len = length(agentsArray);
    radius = zeros(len,1);

    sumXaxis = GAUSSANGLE;
    if len > 0 %Sollte eigentlich immer erfüllt sein
        for i = 1:len
            radius(i) = agentsArray(i).radius;
        end
        maxRadius = max(radius);
        if influenceSphere < (3*maxRadius)
            influenceSphere = 3*maxRadius;
        end

        for i = 1:len %Durch den Array mit allen Agenten durchgehen
            if i ~= agentPosition && priorityArray(i) ~= 0
                deltaY = agentsArray(i).cordY - agentsArray(agentPosition).cordY;
                deltaX = agentsArray(i).cordX - agentsArray(agentPosition).cordX;
                distance = sqrt(deltaX^2 + deltaY^2);

                if distance < influenceSphere && (sign(agentsArray(agentPosition).maxSpeed) * agentsArray(i).cordY) > (sign(agentsArray(agentPosition).maxSpeed) * agentsArray(agentPosition).cordY)

                    angleXY = atan(deltaX/deltaY);

                    [alpha,indexX] = closest(ANGLE, angleXY);
                    alpha = (pi/2 - alpha); %   Alpha umrechnen in das Alpha, welches für die Berechnung der Betawinkel benötigt wird (siehe Dokumentation)

                    radiusSum = agentsArray(agentPosition).radius + agentsArray(i).radius;
```

```


                    if (sign(agentsArray(i).actSpeed) == 0) %Anderer Agent bleibt stehen
                        diffVelocity = -abs(agentsArray(i).actSpeed);
%                   elseif (sign(agentsArray(i).actSpeed) + sign(agentsArray(agentPosition).actSpeed)) == 0 || ((abs(agentsArray(agentPosition).actSpeed) - abs(agentsArray(i).actSpeed)) > 0) %Agents laufen in unterschiedliche Richtung
                    elseif (sign(agentsArray(i).actSpeed) + sign(agentsArray(agentPosition).actSpeed)) == 0 %Agents laufen in unterschiedliche Richtung
                        diffVelocity = -abs(agentsArray(i).actSpeed - agentsArray(agentPosition).actSpeed);
                    elseif ((abs(agentsArray(agentPosition).actSpeed) - abs(agentsArray(i).actSpeed)) > 0)
                        diffVelocity = -abs(agentsArray(i).actSpeed - agentsArray(agentPosition).actSpeed) * DISPERSIONFACTOR;
                    else %Agents laufen in gleiche Richtung oder einer bleibt stehen
                        diffVelocity = abs(agentsArray(i).actSpeed - agentsArray(agentPosition).actSpeed);
                    end

                    [betaLeft, betaRight] = getBeta(radiusSum, alpha, distance);
                    if sign(agentsArray(agentPosition).actSpeed) == 0
                        moving = 0;
                    elseif abs(sign(agentsArray(agentPosition).actSpeed) + sign(agentsArray(i).actSpeed)) == 2 %Agents laufen in die gleiche Richtung
                        moving = 1;
                    else
                        moving = 2;
                    end

                    sumXaxis = sumXaxis + xValuesLogic(indexX, distance, betaLeft, betaRight, diffVelocity, radiusSum, moving);
                    %angleOut = sumXaxis %Für debugging
                end
            end
        end

    end

    lenWall = length(wallArray);
    if lenWall > 0
        for i = 1:lenWall
            deltaY = wallArray(i).cordY - agentsArray(agentPosition).cordY;
            deltaX = wallArray(i).cordX - agentsArray(agentPosition).cordX;
            distance = sqrt(deltaX^2 + deltaY^2);
            if distance < influenceSphere && (sign(agentsArray(agentPosition).maxSpeed) * wallArray(i).cordY) > (sign(agentsArray(agentPosition).maxSpeed) * agentsArray(agentPosition).cordY)
                    angleXY = atan(deltaX/deltaY);
                    [alpha,~] = closest(ANGLE, angleXY);
                    alpha = (pi/2 - alpha);
                    [betaLeft, betaRight] = getBeta(radius(agentPosition) + wallArray(i).radius, alpha, distance);

```

```
                    sumXaxis = sumXaxis + xWallLogic(distance, betaLeft, betaRight, agentsArray(agentPosition).radius);
            end
            %plot(ANGLE, sumXaxis);
        end
    end
    [~,indexAngle] = max(sumXaxis);
    angleOut = ANGLE(indexAngle);
end
```

```matlab
function [ xOut ] = xValuesLogic( indexX, distance, betaLeft, betaRight, diffVelocity, radiusSum , isMoving )
%   Berechnet die Werte für die Berechnung der Richtung
%  Calculates the values for the calculation of the direction
    global XVALUES HEIGHT SLOPEFACTOR ANGLE AGENTANGLEOFFSET REPULSIONAGENT STANDOFF

    if isMoving == 0 && abs(diffVelocity) < 0.2    %Agents bleiben beide stehen (ungefähr)
        diffVelocity = STANDOFF;
    end

    if diffVelocity < 0
        alphaX = XVALUES(indexX);
        xOut = 1./(abs(XVALUES - alphaX).^(-diffVelocity/SLOPEFACTOR)); %Zentrierung um alphaX

        [~,indLeft] = closest(ANGLE, betaLeft);
        [~,indRight] = closest(ANGLE, betaRight);
        [~,indLeftS] = closest(ANGLE, betaLeft - AGENTANGLEOFFSET);
        [~,indRightS] = closest(ANGLE, betaRight + AGENTANGLEOFFSET);

        xOut = (xOut - min(xOut));
        xOut(indLeft:indRight) = 0;

        if isMoving == 1
            xOut = xOut * HEIGHT / max(xOut) * (-diffVelocity) * (radiusSum / distance)^REPULSIONAGENT;
        else
            xOut = xOut * HEIGHT / max(xOut) * (radiusSum / distance)^REPULSIONAGENT;
        end

        xOut(indLeftS:indLeft) = linspace(xOut(indLeftS),0,(indLeft-indLeftS+1));
        xOut(indRight:indRightS) = linspace(0,xOut(indRightS),(indRightS-indRight+1));

    elseif diffVelocity > 0
        alphaX = XVALUES(indexX);
        xOut = gaussmf(XVALUES, [radiusSum/distance alphaX]) * diffVelocity * HEIGHT /5 * radiusSum/distance;

    else
        xOut = zeros(1,length(XVALUES));    %Agents laufen gleich schnell

    end

end
```

```
function [ xOut ] = xWallLogic(distance, betaLeft, betaRight, radius)
%   xWallLogic calculates and returns the values for the repulsive strength and
    angle of an wall agent to an agent specified by distance, betaLeft und betaRight
    . Radius is the radius of the agent in question. See documentation for a
    visualisation of the setup

    global XVALUES WALLFACTOR ANGLE WALLANGLEOFFSET

    xOut = zeros(1, length(XVALUES));

    [~,indLeft] = closest(ANGLE, betaLeft);
    [~,indRight] = closest(ANGLE, betaRight);
    [~,indLeftS] = closest(ANGLE, betaLeft - WALLANGLEOFFSET);
    [~,indRightS] = closest(ANGLE, betaRight + WALLANGLEOFFSET);

    xOut(indLeft:indRight) = WALLFACTOR / (distance - radius);

    xOut(indLeftS:indLeft) = linspace(0, xOut(indLeft), (indLeft-indLeftS+1));
    xOut(indRight:indRightS) = linspace(xOut(indRight), 0, (indRightS-indRight+1));

end
```

```
function [   ] = spawn( agentArray, position )
%SPAWN function that generates values for new spawned agents
% For given agentArray and information whether to spawn on bottom or top,
% this function calculates the initial values for a new spawned agent.
% position - takes value 1 (spawn at bottom) or -1 (spawn at top)

global MEANRADIUS STDRADIUS MEANSPEED STDSPEED WIDTH YSPB1 YSPT2 REP
% MEANRADIUS, STDRADIUS - for Gaussian for radius
% MEANSPEED, STDSPEED - for Gaussian for speed
% WIDTH is the width of the corridor, YSP(b/t)(1/2) are the y-coordinates
% of the spawning fields, b=the bottom ones, t=the top ones, both numbered
% from bottom to top (which means, YSPB1=0, then +spawning height=YSPB2, then
% +length of corridor=YSPT1, and +spawning height=YSPT2
% Here we only use the lower boundary (YSPB1) and the top boundary (YSPT2)
% REP; #of tries to spawn the agent (if always collision, don't spawn him)

priorityArray = getPriorityArray(agentArray);
sortedPriorityArray = getSortedPriorityArray(priorityArray);
index = find(priorityArray==0,1); %position of empty agent slot in agentArray
    if (size(index,2)~=0) %if agentArray is not full.

    % Generate radius with MEANRADIUS, STDRADIUS and a Gaussian
    % distribution
        radius=4*STDRADIUS;
        while (abs(radius)>(3*STDRADIUS))
            radius=normrnd(0,STDRADIUS);
        end %while
        radius=MEANRADIUS+radius;

        %y-coordinate
        if position==-1
            ycoord=YSPT2;
        else
            ycoord=YSPB1;
        end %if-else-ycoord.

        %x-coordinate
        distMat = zeros(1,length(sortedPriorityArray));
        if size(distMat,2) == 0 %Kein Agent vorhanden
            xcoord=(WIDTH-2*radius)*rand(1)+radius;
            speed=4*STDSPEED;
            while (abs(speed)>(3*STDSPEED))
                speed=normrnd(0,STDSPEED);
            end %while
            speed=position*(MEANSPEED+speed);

            agentArray(index).cordX = xcoord;
            agentArray(index).cordY = ycoord;
            agentArray(index).maxSpeed = speed;
            agentArray(index).actSpeed = speed;
            agentArray(index).radius = radius;
            agentArray(index).priority = rand(1);
            while (agentArray(index).priority == 0) %Damit priority sicher nicht 0
                ist
                agentArray(index).priority = rand(1);
            end
        else

```

```
            for count=1:REP
                xcoord=(WIDTH-2*radius)*rand(1)+radius;
                for k = sortedPriorityArray
                    distMat(k) = sqrt((ycoord - agentArray(k).cordY)^2 + (xcoord - agentArray(k).cordX)^2) - agentArray(k).radius - radius;
                end
                distMat = sort(distMat);
                if (distMat(1) >= 0)
                    break
                elseif (count ~= REP)
                    continue
                else
                    fprintf('No agent spawned\n')
                    return
                end
            end

            if (distMat(1) >= 0)
            %Generate velocity with MEANSPEED, STDSPEED and position (move up
            %or down!) and a Gaussian distribution
                speed=4*STDSPEED;
                while (abs(speed)>(3*STDSPEED))
                    speed=normrnd(0,STDSPEED);
                end %while
                speed=position*(MEANSPEED+speed);

                agentArray(index).cordX = xcoord;
                agentArray(index).cordY = ycoord;
                agentArray(index).maxSpeed = speed;
                agentArray(index).actSpeed = speed;
                agentArray(index).radius = radius;
                agentArray(index).priority = rand(1);
                while (agentArray(index).priority == 0) %Damit priority sicher nicht 0 ist
                    agentArray(index).priority = rand(1);
                end
            end %if kollisionsabfrage
        end %if size
    else
        fprintf('Warning: Agent array is full\n')
    end %if
    agentArray(index);
end %Function
```

```matlab
function [betaLeft, betaRight] = getBeta(radiusSum, alpha, distance)
%getBeta: For two agents: for given radii, angle alpha (direction of other agent wrt. x-Axis) and the distance between, this function calculates the angles outside. See documentation for the derivation and exact meaning of alpha, betaLeft, betaRight and gamma

    gamma = acos(radiusSum / distance);
    betaRight = pi - (gamma + alpha);    %betaRight = pi/2 - (gamma + alpha - pi/2);
    betaLeft = - alpha + gamma;          %betaLeft = pi/2 - (pi/2 + alpha - gamma);

end
```

```matlab
function [ priorityArray ] = getPriorityArray( agentsArray )
    %For given agents, this function reads out their priority values.

    priorityArray = zeros(1,length(agentsArray));
    for i = 1:length(agentsArray)
        priorityArray(i) = agentsArray(i).priority;
    end

end
```

```matlab
function [ outList ] = getSortedPriorityArray(priorityArray)
%For given priorityArray, this function sorts them according to decreasing priority ignoring all zeros and returning the indices of the sorted priorities.

   sortMat = ([priorityArray; 1:length(priorityArray)])'; %prios with index
   sortMat = (sortrows(sortMat, 1))'; %sort them decreasingly wrt priority
   %Hier könnte man alle mit den gleichen Prioritätswerten
   %durcheinandermischen
   prioNotZero = fliplr(find(sortMat(1,:))); %only non-zero values
   outList = sortMat(2,prioNotZero); %return indices

end
```

```matlab
function [ valueClosest, indexClosest ] = closest( searchArray, searchValue )
% This function returns the closest value and index, which is in searchArray, closest to searchValue.

    [~, indexClosest] = min(abs(searchArray - searchValue)); %save index
    valueClosest = searchArray(indexClosest); %return value
end
```