ETH
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
Train Platform Simulation

Dominic Hänni, Patrick Manser & Stefan Zoller

Zurich
May 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.

Dominic Hänni Patrick Manser Stefan Zoller

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

_____

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

Author(s)

Last name _____ First name _____

Supervising lecturer

Last name _____ First name _____

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

The idea of our project is to model and simulate train deboarding situations of Swiss commuter railway systems during rush hours. Apparently during peak times, train platforms are overcrowded. This everyday experience inspired us to examine pedestrian dynamics.
We wanted to create a model with MATLAB with freely customizable parameters. This model was used to analyse different stairs configurations and existing platform designs.
For the implementation we used an agent based social force model. For the evaluation we used different characteristics like walking times or walked distances of pedestrians. The first result of our simulations is an optimal width of stair according to our model. Secondly, we simulated the existing platform designs of Winterthur and Zurich Altstetten. By analysing the characteristics, it turned that the number of stairs does have quite a small influence on the mean walking time.

2 Individual Contributions

In accordance with the following table, we roughly split the work into the following parts. But nevertheless, many problems were solved together and we often helped out each other.

Table 1: division of labour

Task	Assigned person
Creating maps of different train boarding platforms	Patrick Manser
Generating platform matrices out of maps	Patrick Manser
Generating target leading vector fields	Patrick Manser
Writing algorithm that allows pedestrians to change stairs	Patrick Manser
Initializing anisotropic social force between pedestrians	Stefan Zoller
Initializing wall force between wall elements and pedestrians	Stefan Zoller
Writing function that spawns pedestrians onto the platform	Stefan Zoller
Time integration to calculate velocities and positions	Dominic Hänni
Graphical illustration of the calculated flow	Dominic Hänni
Analysis of the simulation results	Dominic Hänni

3 Introduction and Motivations

Figure 1: Crowded train platform in Winterthur, Source: [1]

In 2011, the SBB train company registered a total of 356.6 million drives of customers, which is 2.7 percent more compared to 2010. [3]
This annually increasing number of passengers, together with the unhurried enlargement of train infrastructure, leads to increasing pedestrian bottleneck on train platforms every year. This is a fact that two of us experience every day during the travel from Winterthur to Zurich. According to this daily experience, we realized that it would be interesting for us to model and simulate deboarding pedestrians on train platforms and to study the influence of several parameters on bottlenecks and the times for customers to leave platforms.

3.1 Fundamental Questions

Is the social force model described in endless scientific papers sufficient for this situation or are there any adjustments we have to implement to get proper results?
What are suitable parameters for the analysis of bottleneck situations on train plat-

forms?
Is our implemented model compatible with the simulation of train platforms and useful for the analysis of platform evacuation?
What is the optimal width of stairs on the train platforms?
Is it possible to simulate real train platforms with our model and getting proper results?

3.2 Expected Results

Because we assume that simulations and experiments of experts are considered when train platforms are built in Switzerland, we do not expect results that differ too much from situations observed in reality.
This means that a width of the stairs of about half of the width of the platform will probably provide good results. In contrast to this, regarding the number of stairs on a platform we expect a pretty big number to be the most efficient. But of course this is only true if you simply look at the deboarding times. In reality you would have to reach a compromise between deboarding times and building cost.

3.3 Research Methods

Continuous modelling by implementing the agent based social force model.
We decided to create a model with a continuous treatment of motion, because we basically just have a systematic motion on a train platform and the unknown influences are relatively small. Systematic motion means that people are very likely confronted with standard situations and react automatically. For example, a pedestrian does not walk into a wall. This can be modelled very easily by implementing the social force model. [8]

4 Description of the Model

Our main goal was to have a model of a train station which is freely customizable. This makes it possible to simulate any design of train platforms and therefore to compare different configurations of stairs.
Therefore, a map containing walls, slow areas and spawn points can be imported into MATLAB. To define the targets, layers can be imported. The number of layers must be equal to the number of targets. Each special area has its own colour, by which they can be recognized by MATLAB.
MATLAB generates the shortest path to each target and calculates the minimal distance. Those informations lead every pedestrian to his preferred target.
The simulation is then performed with an implementation of the social force model described in [6].
The model consists of several forces and random decisions that influence the pedestrians route.

4.1 Target Force

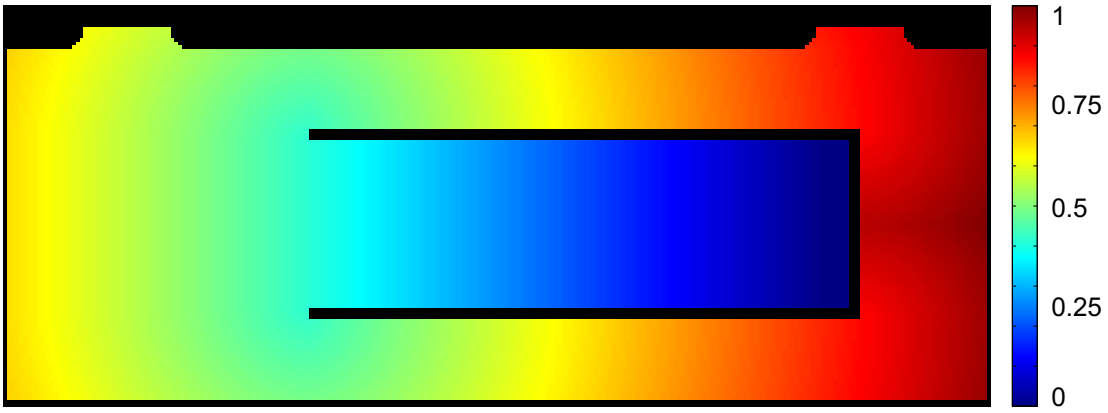


Figure 2: Relative distance to the end of the stairs

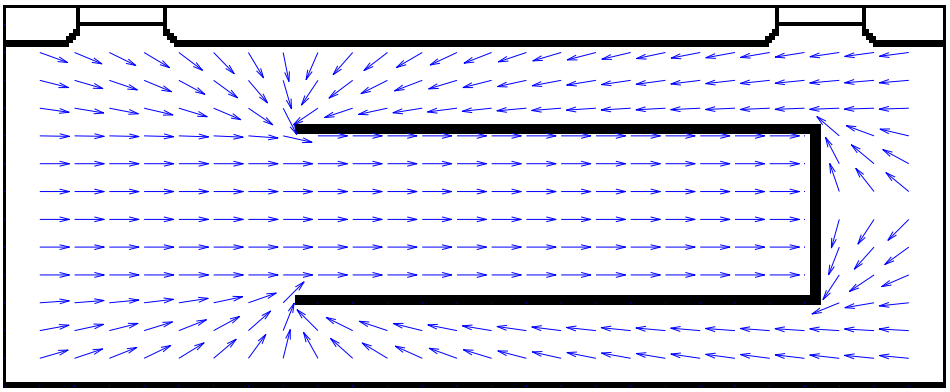


Figure 3: Target leading vectorfield

We used vector fields to make sure that every pedestrian finds his goal. These vector fields can be generated using the fast marching toolbox (see `Section 5.1.2`). An example can be seen in `Figure 3`. The vector arrows are pointing to the goal, the pedestrians basically follow the direction of them. By using vector fields we assume that pedestrians dynamics on train platforms are similar to fluid dynamics. This means that movements of pedestrians are not random.
Generating vector fields is a powerful and easily implemented aid to calculate the target force. Nevertheless the vector field has its not ignorable disadvantages. The vector field is based on the assumption that every pedestrian wants to minimize his walking distance and not his walking time. For example this can be disadvantageous, when a pedestrian stays behind the stairs a little bit above the center. In this case he will take the upper way around the stairs no matter how many people are blocking his way. It would be better for him to take the other way around the stairs, but he can not differ between a way free of pedestrians and a heavily used way. Anyway this assumption is acceptable except for quite a few circumstances.

`Figure 2` shows us the relative distance between each position on the map and the target. Red areas are far away from the target, dark blue areas in contrast are equal to a relative distance of zero. We need the distance later on in order to let the pedestrians decide between two focused stairs. The fastest way can be found in dependence of the distance and the amount of pedestrians heading to each target. The relative distance is sufficient, because the absolute amount is not important for a decision between two stairs.

4.2 Interaction Forces

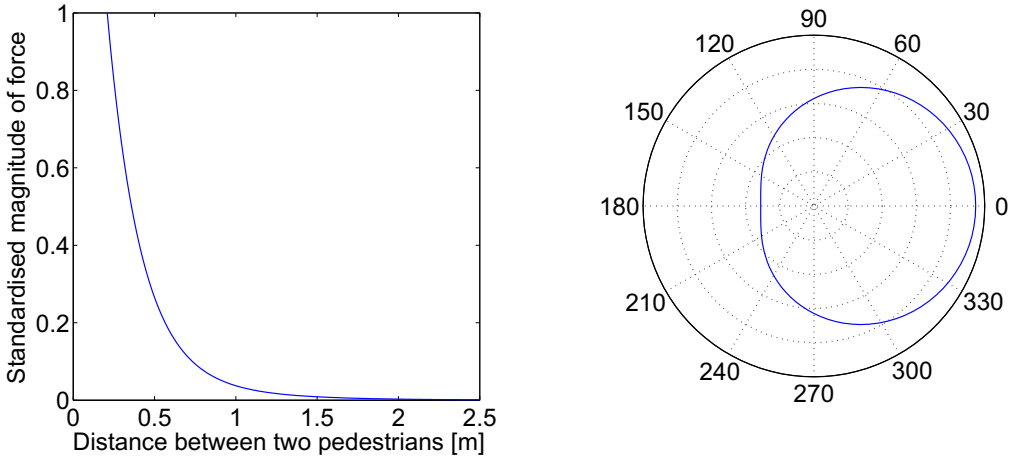
There are two interaction forces that are calculated separately for every pedestrian in every time step. The first one is the so called pedestrian interaction force, which is initialized as a repulsive force and therefore allows each pedestrian to keep a certain distance to all the other pedestrians.
The second one is the wall force, initialised as a repulsive force as well. The wall force prevents pedestrians from walking into or even through walls.

4.2.1 Pedestrian Interaction Force

In our simulation, the repulsive pedestrian interaction force has been specified according to the formula:

$$\vec{f}_{\alpha\beta}(t) = A^1_\alpha e^{\frac{r_{\alpha\beta}-d_{\alpha\beta}}{B^1_\alpha}} \vec{n}_{\alpha\beta} \cdot \left(\lambda_\alpha + (1-\lambda_\alpha)\frac{1+\cos\phi_{\alpha\beta}}{2}\right) + A^2_\alpha e^{\frac{r_{\alpha\beta}-d_{\alpha\beta}}{B^2_\alpha}} \vec{n}_{\alpha\beta} \qquad (1)$$

,where $d_{\alpha\beta}(t) = \|\vec{x}_\alpha(t) - \vec{x}_\beta(t)\|$ is the distance between the two pedestrians α and β, $r_{\alpha\beta} = (r_\alpha + r_\beta)$ is the sum of their radii and $\vec{n}_{\alpha\beta}(t) = [\vec{x}_\alpha(t) - \vec{x}_\beta(t)]/d_{\alpha\beta}$ is the normalized vector from β to α. $\phi_{\alpha\beta}(t)$ is the angle between $\vec{n}_{\alpha\beta}(t)$ and the direction of movement $\vec{e}_\alpha(t) = \vec{v}_\alpha(t)/\|\vec{v}_\alpha(t)\|$ of the pedestrian α.
This force is an addition of two different parts. The second part is a standard exponential function with the parameters A^2_α and B^2_α set in a way that it is nearly zero for most of the possible distances between two pedestrians, but only for small distances it increases really fast and rapidly exceeds all the other forces. This means that this part of the formula assures that two pedestrian do not collide or even overlap themselves.
In contrast to that, the first part of the pedestrian interaction force, which is also an exponential function of the distance, is isotropic. This means that it decreases with increasing angle between the direction of movement of pedestrian α and the vector from pedestrians α to β. Here, the chosen parameters A^1_α and B^1_α cause that this part of the force has a bigger interaction range, but it does not increase that much with decreasing distance. Therefore, pedestrians avoid to walk too close next to each other if there is enough space.
The following two MATLAB plots illustrate these characteristics of the pedestrian interaction force:



(a) Standardised magnitude of the pedestrian interaction force as a function of the distance $d_{\alpha\beta}$

(b) Relative magnitude of the pedestrian interaction force as a function of the angle $\phi_{\alpha\beta}$

Figure 4: Characteristics of the pedestrian interaction force

4.2.2 Wall Force

The wall force is implemented as an exponential force too, but it is not isotropic, as its only duty is to prevent pedestrians from walking into walls. It is calculated according to the formula:

$$A_\alpha^\omega e^{\frac{r_\alpha - d_{\alpha\omega}}{B_\alpha^w}} \vec{n}_{\alpha\omega} \tag{2}$$

,where $d_{\alpha\omega}(t) = \|\vec{x}_\alpha(t) - \vec{x}_\omega\|$ is the distance between the pedestrian α and the wall element ω, r_α is the radius of pedestrian α and $\vec{n}_{\alpha\omega}(t) = [\vec{x}_\alpha(t) - \vec{x}_\omega]/d_{\alpha\omega}(t)$ is the normalized vector from α to ω.
For every person in every timestep, only one wall force is calculated, namely the one of the nearest wall element. To find this wall element, we search a quadratic force range with a side length of 1.5 meters, as you can see in the following graphic:

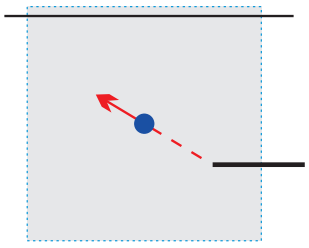


Figure 5: Wall force range

4.3 Stairs Decision Model

One important modification of the social force model for our requirements is that it should be possible for pedestrians to change the focused stairs when they leave the platform. This helps to get more realistic results, because pedestrians will not head to the stairs with a crowd of people in front of, assumed that the distance to the another stairs is almost equal.
For the calculation of the probability to change the stairs, we consider three possible situations. First of all the pedestrian can be on the left side of all the stairs. In this case, he will take the closest stairs to leave the platform. When the pedestrian moves between two stairs, he can only decide between those two. And the third situation is the opposite of the first. When he is on the right side of all the stairs, he can not decide too and he will take the closest target.
Furthermore, it is only possible to change during the first ten time steps and if a minimum value of pedestrians is already heading to the same target. Above all, the relative distance between the closest and the second closest stairs has to exceed a fixed value of 0.35. By trying out, we found out that these conditions are required for proper simulation results.
If all these conditions are fulfilled, the total probability is a product of the two different probability functions shown in `Figure 6`.

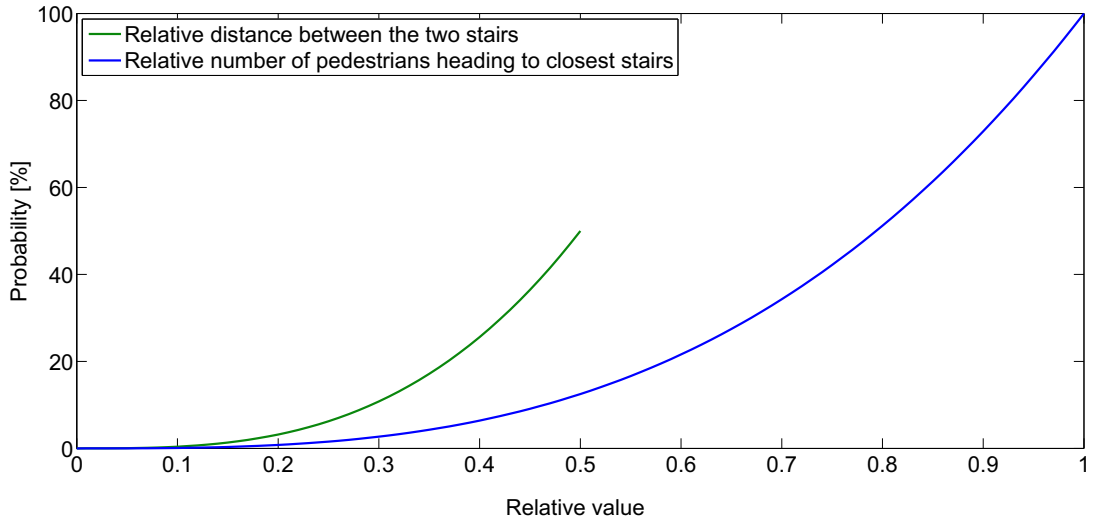


Figure 6: Probability to change to another stairs as a function of the relative distance and as a function of the relative amount of pedestrians

The total probability is then calculated according to the following formula:

$$Probability = (4 \cdot d_{rel}^3) \cdot n_{rel}^3 \tag{3}$$

,where $d_{rel} = d_{min}/(d_{min} + d_{min,sec})$ is a relative distance between 0 and 0.5, with d_{min} being the distance between the pedestrian and the closest stairs and $d_{min,sec}$ being the distance between the pedestrian and the second closest stairs.
$n_{rel} = n_{cls}/(n_{cls} + n_{cls,sec})$ is a relative number between 0 and 1, with n_{cls} being the number of pedestrians heading to the closest stairs and $n_{cls,sec}$ being the number of pedestrians heading to the second closest stairs. In the next plot, you can see the total probability as a function of this two parameters. As you can see, if one parameter is set to its maximum value, the shape of the plot is consistent with the previous plots.

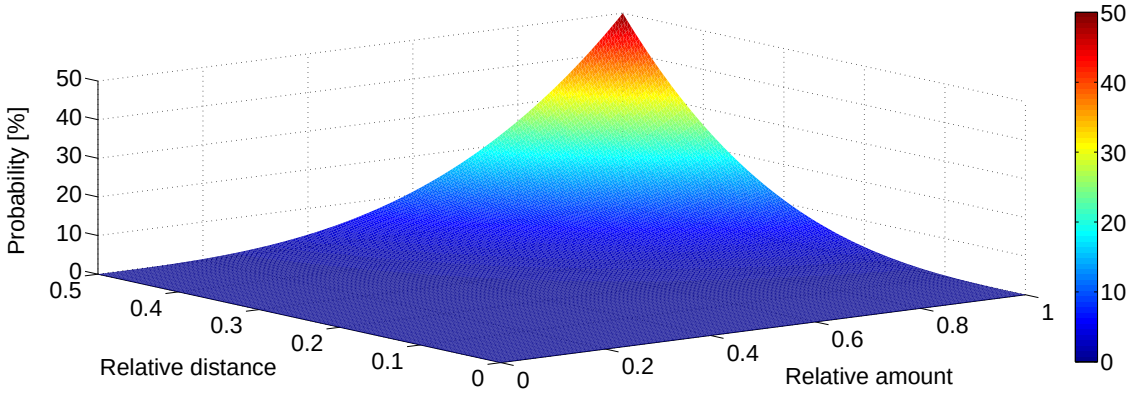


Figure 7: Probability to change to the second closest stairs

5 Implementation

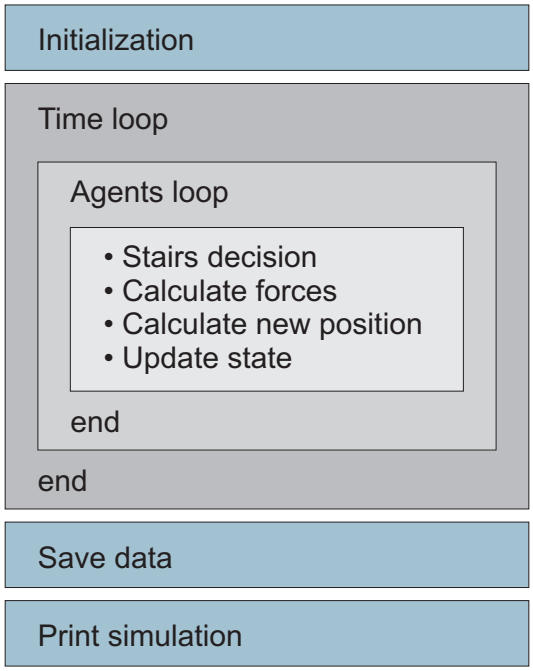


Figure 8: Program chart, inspired by [4]

The basis of the whole program is the main.m file, which must be executed to start a simulation. The main.m file is basically responsible for everything shown in `Figure 8`. Many of these processes are not calculated in the main.m file itself, but it calls the other m-Files and functions. Therefore, it contains the initialization of all the variables, the time loop, the pedestrian loop and the storage of the results to plot the simulation later on.

5.1 Initialization

The init.m file defines all variables such as the pedestrian matrix and the parameters needed to calculate the forces. In addition, it executes getMap.m and generateVectorfields.m.

5.1.1 Generating Platform Matrices out of Maps

The function getMap.m is the first out of three basic elements of the main function. The input is a *.bmp file containing walls, slow areas and spawn areas. All the areas

are defined by specific colors as shown in `Table 2` in order to recognise the different regions. In `Figure 9` an illustration map can be seen.

Table 2: Meaning of the colors in the input map and layers

Color	Hex	Description
White	FFFFFF	Free space, the pedestrians can move freely
Black	000000	Wall element, impassable for the pedestrians
Green	0000FF	Spawn area, pedestrians are spawned there
Yellow	FFFF00	Slow area, pedestrians walk slower there
Red	FF0000	Target area

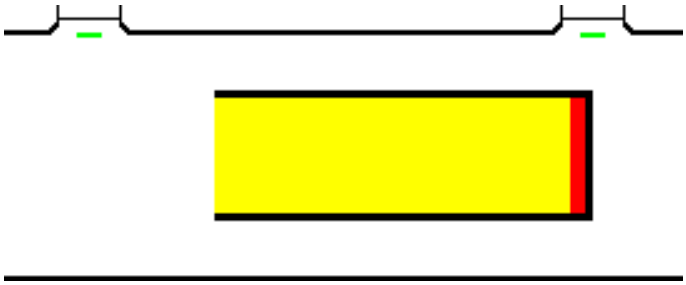


Figure 9: Illustrian map containing white, black, green and red areas. Targets (red) must be indicated in separate *.bmp files

Furthermore, the map targets must be indicated. This happens with different layers as it is done in [2]. The layers must have exactly the same size as the input map, but they only contain the target area. One layer is needed for each target. The output of getMap.m is the two dimensional matrix Map and the three dimensional matrix Layer. Both matrices together illustrate the platform with the following numbers:

Table 3: Meaning of the numbers in the output matrices Map and Layer

Number	Description
0	Wall element
1	Free space
2	Spawn area
3	Slow area
inf	Target area

5.1.2 Generating Target leading Vector Fields

The different layers help us to generate a vectorfield for each target (see `Figure 3`). The first function is generateVectorfields.m, which needs the outputs of getMap.m as inputs. It basically just overlays the map with each layer. The produced matrix is the input of the function computeVF.m, which is responsible for generating the actual vector fields. Just like many recent groups we use a toolbox for this task. It is called fast marching algorithm and can be found at the following link:
`http://www.mathworks.com/matlabcentral/fileexchange/6110`
This toolbox is the basis of the target force. It commits the direction the pedestrians have to follow as well as the distance to the target. The results of the toolbox can be seen in `Figure 2` and in `Figure 3`.
For the calculation of the final gradient field we copied the function gradientField.m from [2]. The same function can also be seen in [5]. The function gradientField.m commits the two vector matrices, one for the x-direction and one for the y-direction. The vectors are normalized to a constant length of one. All the vectors and the distance matrix are saved in a $n \times 1$ cell, n is the number of layers.

5.2 Time Loop

The time loop is the second essential element of the main.m file. Within this loop, pedestrians might be spawn and the total force as well as the new position of each pedestrian is calculated. In addition, pedestrians have the possibility to change the focused stairs if one is too crowded. Last but not least all the named results are saved in a cell.

5.2.1 Spawn System

At the beginning of each time step new pedestrians can be spawned, if the maximal amount of pedestrians is not yet reached. The spawning process is quite simple. First, the function searchStartingpoints.m locates all the possible starting points. Then the function spawnPed.m searches a square with side length of 1.2 meters for other pedestrians. This happens quite similar to the search for wall elements implemented to calculate the wall force, which is illustrated in `Figure 5`. If no pedestrian can be found in that square, the function spawns a new pedestrian onto the center of it. The new pedestrian has a certain weight, a radius and a maximal velocity. The weight and the radius have the same amount for every pedestrian, but the maximal velocity is individually distributed. It might be more realistic to have individual weights and radii as well, but the influence on the effects we are trying to have a look at is quite small.
The program spawns a new pedestrian wherever a starting point is unoccupied. Therefore, there is no fixed number of pedestrians trying to get out of the train at each door, because this number is calculated dynamically. This can lead to undesirable situations. For example if one door is blocked by many pedestrians waiting in front of it. In this situation, the pedestrians simply get spawned onto another starting point located at another door. This is unrealistic and causes falsified results. It could be resolved by spawning a fixed number of pedestrians at each door, but as we assume this situation to be quite rare, we neglect this extension in our model.

5.2.2 Agents Loop

The agents loop is the third crucial part of the main.m file. First of all pedestrians can change their target. This happens in the function Stairsdecision.m. For a detailed description of what this function does see `Section 4.3`.
Besides, the agent loop applies the formulas seen in `Section 4.2.1` and `Section 4.2.2` to the pedestrians in order to calculate the social force and the wall force as well as the target force, which is given by the calculated vector field. (`Section 4.1`) The total force is the result of the summation of these three forces with in certain weighting.
For the calculation of the new position of every pedestrian we use Newton's law of motion. This law sets the acceleration in relation to the total force on a pedestrian and his mass and is described by the following formula:

$$Acceleration\ \vec{a} = \frac{\sum\limits_{i} \vec{F_i}}{Mass} \tag{4}$$

,where the acceleration is equal to the first time derivative of the velocity and equal to the second time derivative of the position:

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{d^2\vec{r}}{dt^2} \tag{5}$$

For a numerical solution of this equation, we use the Explicit Forward Euler Method twice to calculate the new velocity as well as the new position of the pedestrian:

$$\vec{v}_{t+dt} = \vec{v}_t + dtcdot\vec{a} \qquad \vec{r}_{t+dt} = \vec{r}_t + dt \cdot \vec{v} \tag{6}$$

After calculating the new velocity we assure that $|v| \leq |v_{max}|$ and check if the pedestrian is on one of the stairs. If he is on a so called slow area, his maximum velocity is reduced by multiplying with a constant factor $c_{velo,stairs} = 0.5$.
After calculating the new position, we have to round the result because the position of a pedestrian can only be an integer given that the simulation is pixel based.

6 Simulation Results and Discussion

We made two completely different simulation runs. First we tried to find out how the width of stair influences the average time the pedestrians have to leave the platform and whether an optimum exists. Secondly we applied this optimum width of stair to two larger simulations with different stairs configurations and analysed and compared the results.

6.1 Optimal Width of Stair

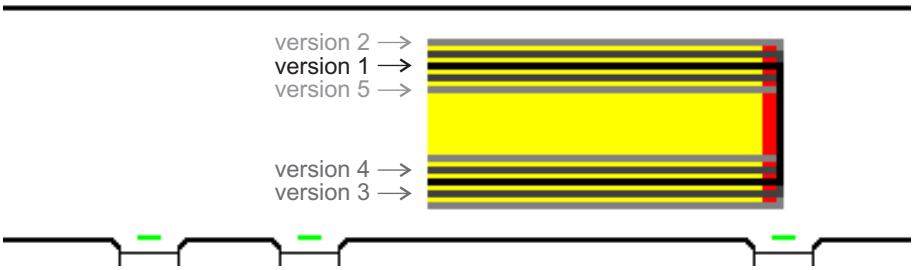


Figure 10: Different widths of stair with the corresponding version numbers

For this simulation we built a map containing a door situation which is typical for train platforms. Due to the computation time the map is quite small and contains only one stair. In `Figure 10` the door situation and the five analysed widths of stair can be seen.
Two of the doors are located in front and only one is behind the stairs. This is a realistic positioning because in real situations, stairs are often located near to the end of platforms, which means that most of the train passengers reach the stairs from the front. The different widths of stair are between two assumed extremes. With version 2, only one pedestrian from behind can pass between the stairs and the train. In contrast to that, with version 5 the stairs are so small that not enough pedestrians can walk side by side to achieve small evacuation times. Each simulation was run with 200 pedestrians.

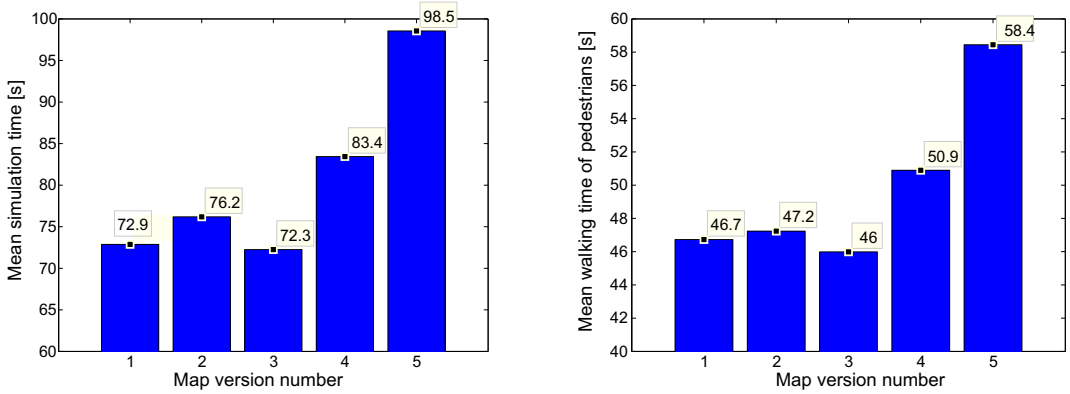


Figure 11: Left side: Mean time until every pedestrian has abandoned the platform. Right side: Mean walking time of the pedestrians.

Due to the probability functions and randomness of some pedestrian parameters in our model, we decided to run ten simulations for each stairs configuration. With the mean value out of the ten simulations we compared the performance of the different stairs. In `Figure 11` you see a plot of the mean simulation and walking times for the tested stairs. The first interesting result is that the simulation time is nearly equal for the map versions 1 to 3. The time difference between those fastest maps and map version 4 is about ten seconds. This means that it takes almost 15% longer until the pedestrians have left the platform. Map version 5 is obviously the slowest map. The same result we see in the right plot of `Figure 11`, illustrating the mean walking time. In version 5, the mean walking time is about 12 seconds more than in version 3.
The following figures demonstrate the simulation runs in action. The current run time is always 20 seconds:

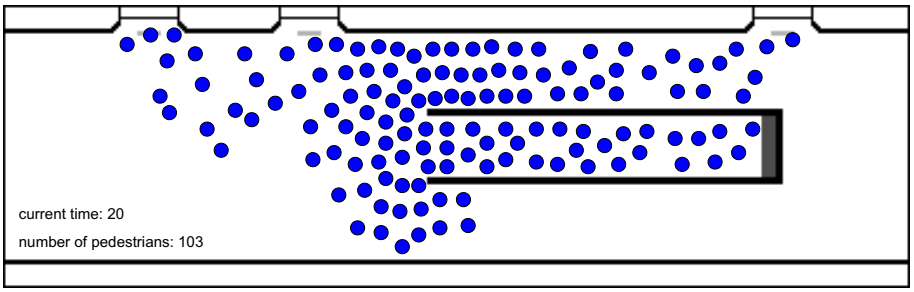


Figure 12: Map version 5

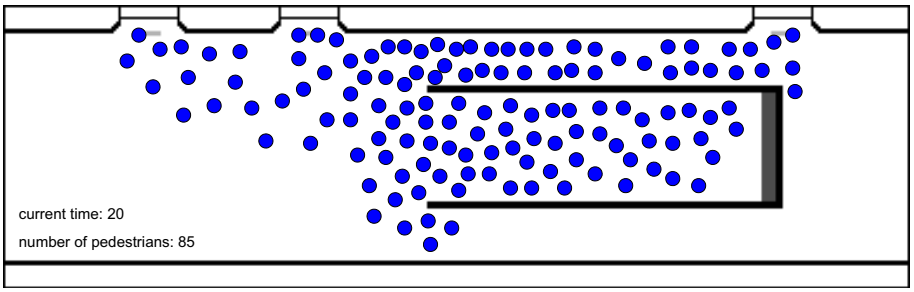


Figure 13: Map version 1

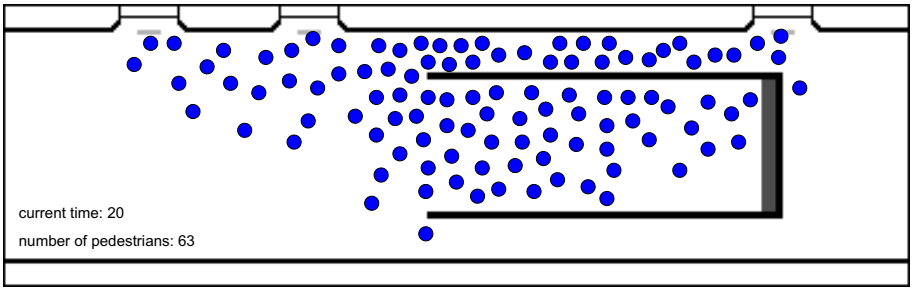


Figure 14: Map version 3

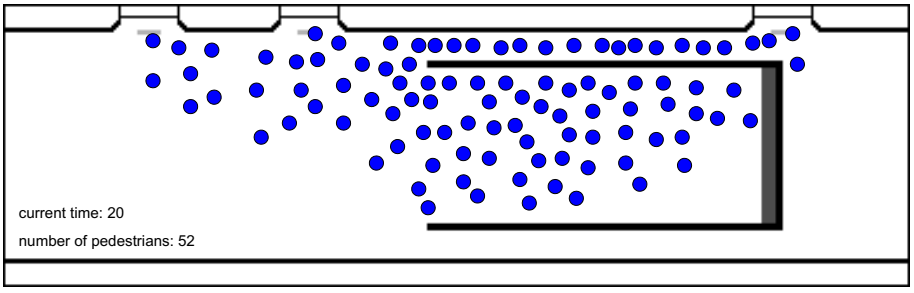


Figure 15: Map version 2

As expected, map version 5 is totally useless. In `Figure 12` the massive bottleneck in front of the stairs can be seen. Map version 4 is better, but still not optimal. The width of stair is just a way too small to prevent bottlenecks. The plot looks quite similar to the one of version 5, therefore we did not include it.
Most interesting is the comparison of the first three situations. The mean walking time is almost the same for all of them, but when we have a look at the differences of `Figure 13` and `Figure 15`, this can not be expected.

How is it possible that the difference of the mean walking time between map version 1 and map version 2 is not as significant as it is between map version 1 and map version 5? The answer to this question has to do with the spawn system. As mentioned in `Section 5.2.1`, our spawn system has one big disadvantage. There is no fixed number of spawns at each door. In `Figure 15` you can see this problem. The door on the right side is blocked, which means that much more pedestrians will appear in the two doors on the left. `Figure 16` confirms the problem too. The mean walking distance in map version 1 is significantly higher than in map version 2. This means that less pedestrians get out of the train on the right door, which is not realistic. So in the end we get the same walking time, but map version 1 is more realistic than version 3.
There is still map version 3, which seems to be better than version 1 considering the statistical analysis. But on train platforms, security criteria have a big impact as well. There should be enough space for a minimal of three persons to walk next to each other, otherwise it will not comply with security criteria.
Considering all this motivations, map version 1 should be applied to train platforms according to our model.

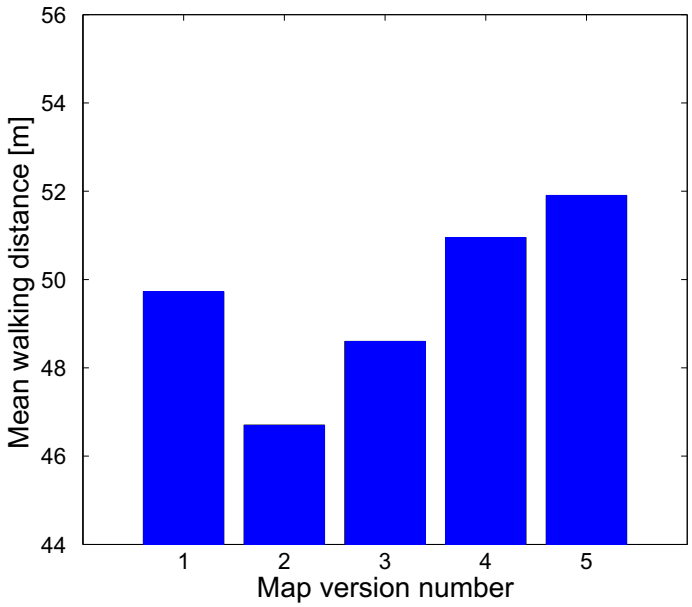


Figure 16: Left side: Mean walking distance for the pedestrians in the different maps. Right side: Standard deviation of the walking distance.

6.2 Application of our Model to realistic Situations

6.2.1 Introduction

With the optimal width of stair, which we found in the first part of our simulation, we tested two platform configurations. Because we wanted to compare the situation on real existing platforms, we draw a map very similar to platform 4/5 in Winterthur as well as a map similar to platform 2/3 in Altstetten. The scale of the maps is not exactly true and therefore they can not represent reality entirely. Nevertheless the platforms will be called Winterthur and Alstetten hereafter. In `Figure 17` and `Figure 18` the two different platforms with the current stairs arrangements can be seen.

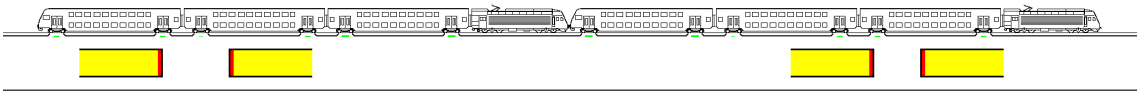


Figure 17: Platform in Winterthur with 4 stairs. Source of the train picture: [7]

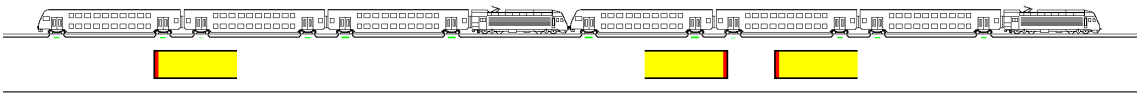


Figure 18: Platform in Zürich Altstetten with only 3 stairs.

We are most interested in the influence of this two situations on the pedestrian runtime. These are both highly frequented platforms in rush hours and a fast emptying is central. The simulations were run with 600 pedestrians. 600 pedestrians produced realistic results and the computing time was around four hours, which is a good limit. In reality much more pedestrians leave the train platforms during rush hours, but the trains are even longer than in our simulation, which means that the pedestrians are better distributed too. Altogether our simulation comes quite close to reality.

6.2.2 Statistical Effects

To estimate the statistical influence of the model on the results, we simulated the situation in Winterthur 9 times. As shown in `Figure 19` the mean runtime of the pedestrians varied in nine simulations only about 1 second, as well as the standard deviation. The standard deviation being nearly 15 seconds shows that the runtime

varies a lot for different pedestrians. A pedestrian who deboards early can be twice as fast as others who get off the train later and stuck in traffic in front of the stairs. The same observation we made in `Figure 20`, where you can see the mean walking distance in the simulations.
This means that in big simulations with 600 pedestrians the statistical influence of the model is rather insignificant. The reason is that the maximal velocity is the only thing besides the stairs decision model which is statistically distributed. Thus the different simulation runs can not differ to much. This is beneficial for the analysis of simulation results, but not realistic. In reality, much more elements can vary such as the amount of people or the weight and radius of each person. We do not include that in our model for simplification.

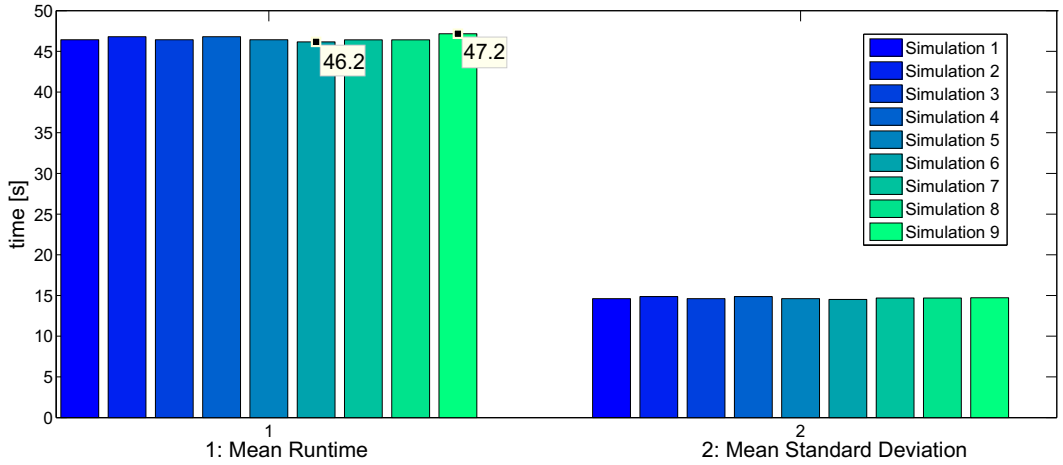


Figure 19: Mean pedestrian runtime and standard deviation of 9 simulations on platform Winterthur.

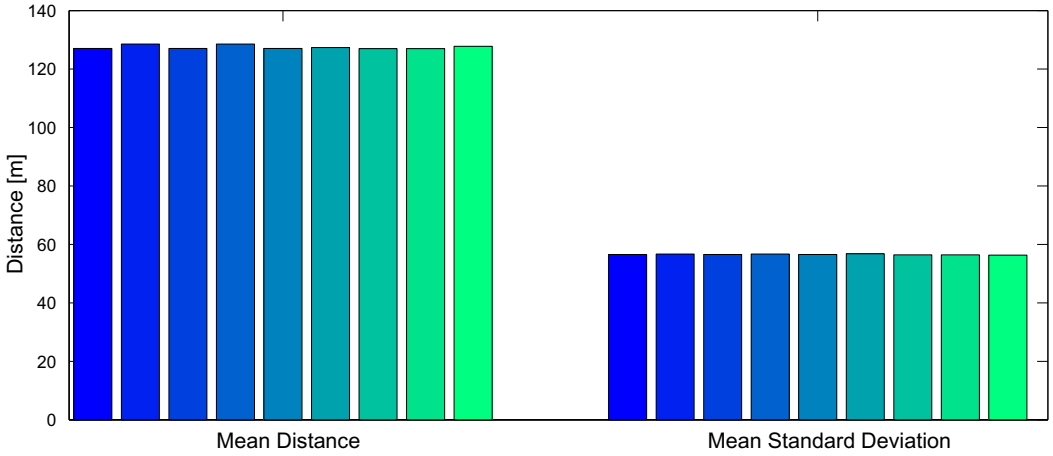


Figure 20: Mean walking distance for the pedestrians and standard deviation of 9 simulations on platform Winterthur.

6.2.3 Comparison between Winterthur and Altstetten

When we compare the two platforms, we notice that in Winterhur the average runtime is about 3 seconds faster than in Altstetten as well as the average walked distance by a pedestrian is a bit shorter. `Figure 21` shows the Plot for the different runtimes and `Figure 22` illustrates the compared distance.

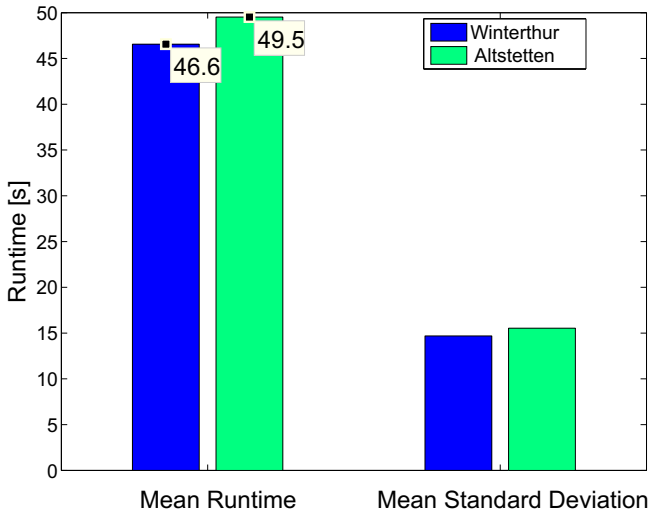


Figure 21: Different mean runtime for a pedestrian in Winterthur and Altstetten and the associated standard deviation.

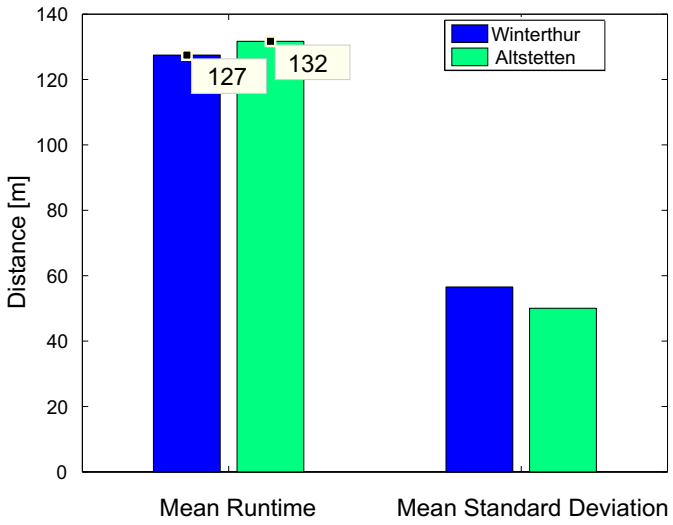


Figure 22: Different mean walking distance in Winterthur and Altstetten and the associated standard deviation.

So far these observations do not imply a big difference between the two simula-

tions. For a more detailed analysis of the performance we need to take a look at some other key parameters. One evidence for smooth traffic is the average velocity. If a crowd is building up in front of stairs, all pedestrians have to slow down. The pedestrians stuck in traffic are a well suited indication for the performance too. `Figure 23` illustrate these two parameters.
In both situations the number of pedestrians stuck in traffic increases linearly for the first 30 seconds. At this time the biggest crowd exist in front of the stairs. After this peak the value decreases in case of Winterthur again linearly and reaches zero after 60 seconds. In Altstetten we see that the number of pedestrian stuck in traffic stays high for about 10 seconds and decreases afterwards linearly to zero. This is because in Altstetten we only have three stairs, which means that less pedestrians can leave the platform at the same time. The same is true for the relative velocity. Due to the faster breaking up of the crowd, the pedestrians in Winterthur reach the maximum walking speed earlier.

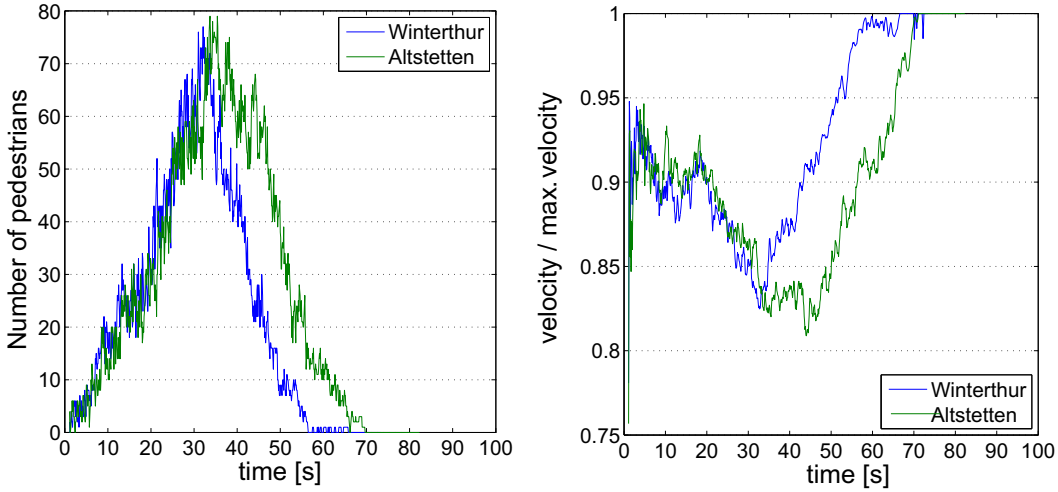


Figure 23: Left: The variation of pedestrians who are slower than a certain velocity and therefore stuck in traffic. Right: The variation of the relative velocity of the pedestrians.

In the following plots the situations for Winterthur and Altstetten after 30 seconds and after 60 seconds can be seen:

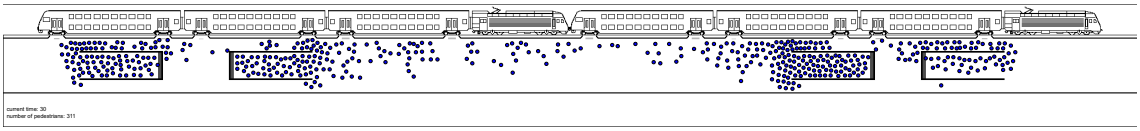


Figure 24: Platform Winterthur 30 seconds after simulation begin

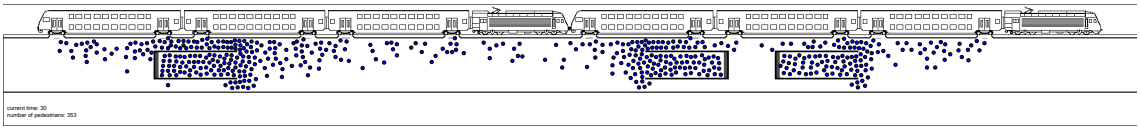


Figure 25: Platform Altstetten 30 seconds after simulation begin

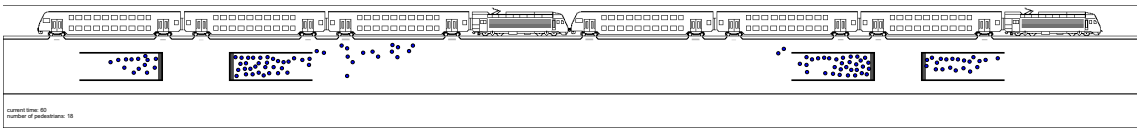


Figure 26: Platform Winterthur 60 seconds after simulation begin

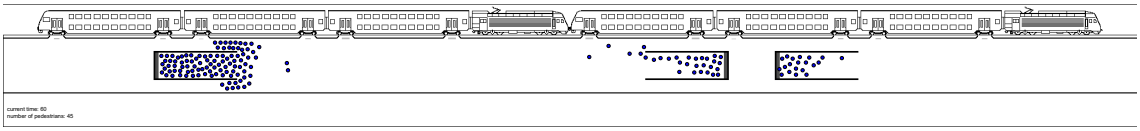


Figure 27: Platform Altstetten 60 seconds after simulation begin

It is conspicuous that the platform in Altstetten seems to be much more crowded after 30 seconds. Even after 60 seconds simulation time the difference on the left side of the platform is apparent. But in the end, the difference between the two simulation times is only 3 seconds. So even if a pedestrians way is blocked by many others, he does not loose that much time as expected.

6.3 Discussion

In the first part of the simulations we found an optimal width of stair for our model. As we assumed in `Section 3.2` the optimal width of stair is approximately half of the platform size. In our testing scenario this means that minimal two pedestrians can walk side by side between the stairs and the train. As mentioned before we need to keep in mind that the model does not consider things like security space which is normally labelled with a white stripe on the platform.

In the second part we compared two different train platforms illustrating the railway stations Winterthur and Altstetten. Even with one stair less in Altstetten, the observed mean walking time does not differ that much. The difference, illustrated in `Figure 21`, is only 3 seconds, which is about 6%. This is unexpected and could result from different influences.
We could imagine that the crowd does not have such a big effect on the mean running time. In `Figure 28` the two different fragmentations of the running times for the simulations are illustrated. In Winterthur we see that there are about four different levels of travel times. In Altstetten we have a more continuous distribution. Our interpretation of this effect is that a bottleneck in front of stairs results on the one hand in slower movements, but on the other hand it can result in a more consistent flow.

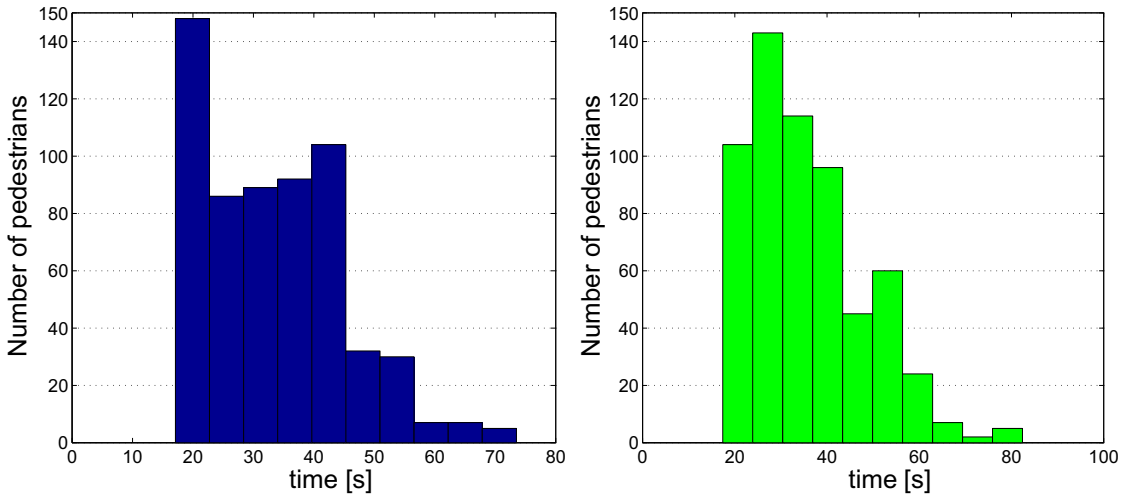


Figure 28: Histogram plots comparing the travel times in Winterthur and Altstetten.

7 Summary and Outlook

7.1 Summary

The main aim of our project was to implement a model that is able to simulate any kind of train platform, especially with different arrangements of stairs. The social force model found in scientific papers with certain adjustments like the stairs decision model was able to produce proper results.
These results were influenced by some unrealistic side effects. For example, it is only possible to change stairs during the first ten time steps, but we had to implement this to avoid situations where two pedestrians block each others path.
Nevertheless, we were able to analyse different sizes of stairs and to find an appropriate width of stair for our model. In addition, it was possible to compare existing situations by applying our model to the platform designs of Winterthur and Zurich Altstetten. For the analysis characteristics like walking time of pedestrians or total evacuation time turned out to be significant.

7.2 Outlook

During the implementation of the model, we realized that there would be plenty of ways to improve the adapted social force model in general as well as our own adjustments. The following suggestions add up to an incomplete list:

- Substitution of the target force implemented as a vector field with a force depending on the surrounding of a pedestrian.
- Initialization of the pedestrian interaction force as a function of the speed of movement. This makes sense because the faster a pedestrian walks, the bigger he tries to keep the distance to surrounding obstacles, as the time he needs to stop is also a function of his speed.
- Allow pedestrians to get into the train and therefore walk in the opposite direction than the platform leaving pedestrians. This modification would have a big influence on the behavior of the pedestrians, especially on stairs and in bottleneck areas. But it would be an extensive enlargement of our model and we did not have time to implement this too.
- Simulation of the stress of pedestrians by reducing social forces acting on them after they had to wait in crowded areas.
- Making the simulation more stochastic by implementing more randomly distributed parameters.

Nevertheless, the task was to implement a model and not to illustrate reality. This is because for many problems, simplifying reality in a model is essential. Without this simplifications, simulations would not be possible at all.

8 References

[1] URL: http://www.nzz.ch.

[2] Katja Briner, Marcel Marti, and Thomas Meier. *train jamming*. HS 2011.

[3] *Die SBB in Zahlen und Fakten - 2011*. URL: http://www.sbb.ch.

[4] Karsten Donnay and Stefano Balietti. *Lecture Notes*. SOMS, 2012. URL: http://www.soms.ethz.ch.

[5] Philipp Heer and Lukas Bühler. *Pedestrian Dynamics Airplane Evacuation Simulation*. FS 2011.

[6] Dirk Helbling et al. *Self-Organized Pedestrian Crowd Dynamics: Experiments, Simulations, and Design Solutions*. Germany, 2005.

[7] Erwin Schaerer, Kaspar Andreas Streiff, and Bruno Studer. "Doppelstock-Pendelzuege". In: *Schweizer Ingenieur und Architekt* (1991).

[8] M. Schreckenberg and S.D. Sharma. *Pedestrian and Evacuation Dynamics*. Springer, 2001.

A Matlab code

Listing 1: main.m

```
% Modeling and Simulating Social Systems with MATLAB
% BlueMen – Pedestrian Dynamics
% Dominic Hänni, Patrick Manser, Stefan Zoller

% This is the main file. Therefore, the iterations over time and over all
% of the pedestrians is done in here, which leads to the calculation of the
% forces, as well as the integrations to calculate the velocities and
% positions.

tic

run init

t = 1 + dt;

%%%%%%%%%%%%%%%
%% time loop %%
%%%%%%%%%%%%%%%
while t < T

    % if the maximum amount of pedestrians is not yet achieved, the
    % function spawnPed will spawn pedestrians if there is no pedestrian in
    % the way.
    if nrped_mom < nrped_end
        [pedM, nrped_mom] = spawnPed(s_area, pedM, nrped_mom, nrped_end,...
        pos, velo, vmax, weight, radius, tstart, tend, status, mtopix, t);
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% loop over pedestrians %%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    for i = 1 : nrped_mom
        if (pedM(status, i) == 1 || pedM(status, i) == 2)

            % implementation of the stairsdecision model. conditions to
            % call out the function: the pedestrian has not yet reached
            % his target, the pedestrian has not yet made his decision
            % bevor he entered the platfrom and the pedestrian has just 10
            % timestep the change the stairs, afterwards he wont change.
            if pedM(status,i) ~= 4 && pedM(statstairs,i) == 0 && ...
                    (pedM(tstart,i) + 10*dt) > t
                [pedM(layer, i), stairselected, pedM(statstairs, i)] = ...
                    StairsDecision(pedM(pos(1),i), pedM(pos(2), i),...
                    Vectorfields, pedM(layer, i), stairselected, Layer);
            end

            % loop over every other pedestrian
            for j = 1 : nrped_mom
                if (i ~= j && pedM(status,j) ~= 4 && pedM(status,j) ~= 0)

                    %% calculation of the pedestrian interaction force
                    % distance between pedestrians i and j
                    d_ped = norm(pedM(pos, i) – pedM(pos, j));
```

```
                % radii of the two pedestrians, plus a constant to make
                % sure that the pedestrians dont overlap
                r_ped = (pedM(radius, i) + pedM(radius, j)) + c_soc;
                % normalized vector from pedestrians j to i
                n_ped = (pedM(pos, i) - pedM(pos, j)) / d_ped;
                % normalized velocity vector of i
                e_ped = pedM(velo, i) / norm(pedM(velo, i));
                % angle between n_ij and e_i
                phi_ped = acos(dot(-n_ped, e_ped));

                % actual force
                pedM(fped, i) = pedM(fped, i) + (A_soc * exp((r_ped...
                    - d_ped) / B_soc) * n_ped * (lambda + (1 -...
                    lambda) * ((1 + cos(phi_ped)) / 2)) + A_phys *...
                    exp((r_ped - d_ped) / B_phys) * n_ped);
            end
        end
        % weight of the social force
        pedM(fped,i) = weight_fsoc * pedM(fped,i);

        %% calculation of the wall force
        % radius of pedestrian i minus a constant to make sure that
        % there is not too much free space between wall and pedestrian
        r_wall = pedM(radius, i) - c_wall;
        % set the minimal wall distance to infinity for every timeloop
        dist_wall = inf;

        % iteration over x-axis from position of pedestrian with
        % range of radius intRad
        for j = (pedM(pos(1),i)-intRad_pix):(pedM(pos(1),i)+intRad_pix)
            % iteration over y-axis from position of pedestrian with
            % range of radius intRad (This means that we are scanning
            % the surrounding square with length 2*intRad of the
            % pedestrian for wall elements.)
            for k = (pedM(pos(2),i)-intRad_pix):(pedM(pos(2),i)+...
                    intRad_pix)
                % assure that position to analyze is within our matrix
                % and that there is a wall element
                if (k >= 1 && k <= length(M(:, 1)) && j >= 1 ...
                    && j <= length(M(1, :)) && M(k, j) == 0)

                    % assure that wall element is within the wall force
                    % range and the distance of the new wall force
                    % element is smaller than the last one
                    if (norm(pedM(pos, i) - [j; k]) <= intRad_pix &&...
                            norm(pedM(pos, i) - [j; k]) <= dist_wall)

                        % vector between the closest wall element and
                        % the pedestrian i
                        vect_wall = (pedM(pos, i) - [j; k]);
                        % distance between the closest wall element
                        % and the pedestrian i
                        dist_wall = norm(vect_wall);
                        % normalized vector between the closest wall
                        % element and the pedestrian i
                        n_wall = vect_wall / dist_wall;

                    end
```

```
            end
        end
    end


    % assure that there is a wall element in the wall element area
    % before the calculation of the wall force
    if (dist_wall <= intRad_pix)
        % actual wall force
        pedM(fwall,i)=A_wall*exp((r_wall-dist_wall)/B_wall)*n_wall;
    end
    % weight of the wall force
    pedM(fwall,i) = weight_fwall * pedM(fwall,i);

    % VFX and VFY: Vectorfields in x and y-direction
    [VFX] = Vectorfields{pedM(layer, i),1}(:,:,1);
    [VFY] = Vectorfields{pedM(layer, i),1}(:,:,2);

    %% calculation of the target force
    % CAUTION: reversed coordinates because of matrix indices
    pedM(ftarg, i) = [VFX(pedM(pos(2), i), pedM(pos(1), i)) ...
        VFY(pedM(pos(2), i), pedM(pos(1), i))]';
    % weight of the target force
    pedM(ftarg,i) = weight_ftarg * pedM(ftarg,i);

    %% calculation of the total force
    pedM(ftot, i) = pedM(fped, i) + pedM(fwall, i) + ...
        pedM(ftarg, i);


    %% calculation of the acceleration, new velocity & position
    % acceleration according to Newton's law of motion
    pedM(acc, i) = pedM(ftot, i) / pedM(weight, i);
    % intergration according to Euler
    pedM(velo, i) = pedM(velo, i) + dt * pedM(acc, i);

    % if the pedestrian is on the stairs, he is going to slow down
    if M(pedM(pos(2),i),pedM(pos(1),i)) == 3
        % set status to 'on stairs'
        pedM(status,i) = 2;
        if ((norm(pedM(velo, i))) > c_onstairs * pedM(vmax, i))
            % reduce velocity to the maximal value
            pedM(velo, i) = (pedM(velo, i) / norm(pedM(velo, i))) * ...
                c_onstairs * pedM(vmax, i);
        end
    else
        % assure that calculated speed does not excess the speed maximum
        if ((norm(pedM(velo, i))) > pedM(vmax, i))
            % reduce velocity to the maximal value
            pedM(velo, i) = (pedM(velo, i) / norm(pedM(velo, i))) * ...
                pedM(vmax, i);
        end
    end


    %% calculate new position of pedestrian i (Forward Euler)
    pedM(pos, i) = round(pedM(pos, i) + dt * pedM(velo, i));

```

```
            % check if pedestrian has already reached his final target
            if Layer(pedM(pos(2),i),pedM(pos(1),i),pedM(layer,i)) == inf && ...
                    pedM(status,i) ~= 4
                % set the status of pedestrian i to 'finished'
                pedM(status, i) = 4;
                pedM(tend, i) = t;
                stairselected(pedM(layer,i),1) = ...
                    stairselected(pedM(layer,i),1) - 1;

            end
        end
    end
    % save the amount of pedestrians heading to each target for each
    % timestep
    stair_cell{1, round(((t - 1) / dt))} = stairselected;

    % save results of this particular time in the cell, where all the
    % pedestrian matrizes for all times are saved
    pedM_cell{1, round(((t - 1) / dt))} = pedM;

    % all forces for pedestrian i set to zero because of the timeloop
    % (pedM is only a auxiliary variable)
    pedM(ftarg,:) = zeros(2,nrped_end);
    pedM(fwall,:) = zeros(2,nrped_end);
    pedM(fped,:) = zeros(2,nrped_end);
    pedM(ftot,:) = zeros(2,nrped_end);

    % stop criterion: as soon as every pedestrian has reached his target
    % the while loop will stop
    endloop = find(pedM(status,:) == 4);
    if size(endloop,2) == nrped_mom
        break
    else
        t = t + dt;
    end
end

elapsedTime = toc;

% save all variables which are needed to generate a proper print. The while
% loop is to not overwrite existing files
save_iter = 1;
while save_iter < 20
    save_ped = 'pedestrian_cell_v1-';
    version = num2str(save_iter);
    data_type = '.mat';
    if exist([save_ped version data_type]) == 0
        save([save_ped version])
        break
    end
    save_iter = save_iter + 1;
end

% clear variables for pedestrians after all of the loops
clear d_ped r_ped n_ped e_ped phi_ped
```

Listing 2: init.m

```
% Modeling and Simulating Social Systems with MATLAB
% BlueMen – Pedestrian Dynamics
% Dominic Hänni, Patrick Manser, Stefan Zoller

% This is the initialization file. In here, the used matrizes and variables
% are initialized

clear all
clc

% conversion meter to pixel
mtopix = 10; % one meter in reality are 10 pixels in our matrizes

% variables for loop over pedestrians
nrped_end = 50;
nrped_mom = 0;

% pedestrian matrix
pedM = zeros(22, nrped_end);

% aid to access to entries of pedestrian matrix
% CAUTION: Because of the indices of a matrix (for M(i,j), i is the
% y-coordinate and j is the x-coordinate), we change here the order of
% the indices to address coordinates in a vector
pos = [2 1]';
velo = [4 3]';
acc = [6 5]';
ftarg = [8 7]';
fwall = [10 9]';
fped = [12 11]';
ftot = [14 13]';
vmax = 15;
layer = 16;
weight = 17;
radius = 18;
tstart = 19;
tend = 20;
status = 21;
% statstairs is 0 or 1, 1 meaning the pedestrian already made his
% decision and 0 meaning he can still change the target
statstairs = 22;

% variables for time iteration
dt = 0.1;
T = 400;

% variables for pedestrian interaction force
c_soc = 0.75 * mtopix;
A_soc = 100 * mtopix;
B_soc = 5;
lambda = 0.3;
A_phys = 30 * mtopix;
B_phys = 2;
d_ped = zeros(2,1);
r_ped = 0;
n_ped = zeros(2,1);
e_ped = zeros(2,1);
```

```
phi_ped = 0;

% constant to reduce speed on stairs
c_onstairs = 0.5;

% variables for wall force
c_wall = 0.12 * mtopix;
intRad_m = 1.5;
intRad_pix = intRad_m * mtopix; % CAUTION: This has to be a whole number!
vect_wall = zeros(2,1);
dist_wall = inf;
n_wall = zeros(2,1);
A_wall = 15 * mtopix;
B_wall = 0.2 * mtopix;

% variables used to calculate the total force
weight_fsoc = 1.1;    % weight of the social force
weight_fwall = 250;  % weight of the wall force
weight_ftarg = 3650; % weight of the target force

% calculate target forces with fast marching algorithm
[M, Layer, Train] = getMap();
[Vectorfields] = generateVectorfields(M, Layer);

% stairselected is the number of pedestrians heading to each target at the
% moment
[stairselected] = zeros(size(Vectorfields,1),1);

% find possible starting points
s_area = searchStartingpoints(M);

% cells to save pedM for each person and each timestep. Is the base for
% the video.
pedM_cell = cell(1, T/dt);
stair_cell = cell(1, T/dt);
for i=1:T/dt
    pedM_cell{1,i} = zeros(22, nrped_end);
    stair_cell{1,i} = zeros(size(Vectorfields,1),1);
end
```

Listing 3: getMap.m

```
function [Map, Layers, Train] = getMap()
%getMap: Conversion of a Map.bmp and i Layer.bmp into i+1
%       two-dimensional matrices containing zeros, ones, twos and inf
%
%   Only the following colors with their interpretations are allowed:
%
%   Color               Hex         Decsription
%   White               FFFFFF      Free space. Passengers can free walk in
%                                   white areas.
%   Black               000000      Wall.
%   Red                 FF0000      Target for a Traveller.
%   Green               0000FF      Start for a Traveller.
%   Yellow              FFFF00      Slow areas (stairs etc. ).
%
%   The output map - matrix contains:
%   0 => Wall, 1 => Free space, 2 => starting point, 3 => slow area
```

```
%
%   The output layer - matrices contain:
%   1 => Free space, Inf => ending point

% exit is a value to find out how many input layers are given
exit = 1;
% S contains the path to each selected file
S = {};

% Instructions to get the programm working
uiwait(msgbox(['Hey there, ',...
        'please select any map.bmp you like first, ',...
        'then select as many matching layer_i.bmp as ',...
        'targets your map has. When you have selected all ',...
        'your layers, simply press cancel and the programm ',...
        'will start running. ',...
        'Have fun.']));

while exit ~= 0
    % x is the number of layers at the end
    x = exit - 2;
    % first, a map must be selected (see instruction)
    if exit == 1
        % get the path to the file
        [FileName, PathName] = uigetfile('*.bmp', 'Select your Map.bmp');
        % make sure that a map is selected
        if FileName == 0
            exit = 0;
            uiwait(msgbox('Select a Map'));
        else
            % fill S with the path to the file and raise exit by one.
            S{exit,1} = strcat(PathName, FileName);
            exit = exit + 1;
        end

        %% generate a train matrix if there is one
        if exist(strcat(PathName, 'train.bmp')) ~= 0
            pathtrain = strcat(PathName, 'train.bmp');

            % imread generates a matrix containing the RGB value of each
            % pixel
            rawTrain = imread(pathtrain);
            % find the different colors and locate them in the matrix
            walls = findColor(rawTrain, 0, 0, 0);
            space = findColor(rawTrain, 255, 255, 255);

            [lines, columns, depth] = size(rawTrain);

            % fill the matrix Map with numbers representing the special
            % areas.
            Train = zeros(lines, columns);
            Train(walls) = 0;
            Train(space) = 1;
        else
            Train = 0;
        end

    % after selecting a map, as many layers as needed can be selected
```

```
    else
        % generate a nice window saying Select Layer'i
        sel = 'Select Layer';
        num = int2str(exit-1);
        [FileName, PathName] = uigetfile('*.bmp', strcat(sel, num));
        % make sure that a layer is selected
        if FileName == 0;
            exit = 0;
        else
            % fill S with the path to the file and raise exit by one.
            S{exit,1} = strcat(PathName, FileName);
            exit = exit + 1;
        end
    end
end

% save number of layers and files in total
nLayers = x;
nFiles = length(S);

%% generate Map

% imread generates a matrix containing the RGB value of each pixel
rawMap = imread(S{1,1});
% find the different colors and locate them in the matrix
walls = findColor(rawMap, 0, 0, 0);
space = findColor(rawMap, 255, 255, 255);
slow = findColor(rawMap, 255, 255, 0);
starts  = findColor(rawMap, 0, 255, 0);

[lines, columns, depth] = size(rawMap);
% make sure that just the needed colors are used
if (length(walls) + length(space) + length(slow) + length(starts))...
        ~= lines*columns
    error('Invalid input Map.');
end

% fill the matrix Map with numbers representing the special areas.
Map = zeros(lines, columns);
Map(walls) = 0;
Map(space) = 1;
Map(slow) = 3;
Map(starts) = 2;

% Add a wall around the map to make sure nobody runs out of the map.
Map(:, 1)       = 0;
Map(1, :)       = 0;
Map(:, columns) = 0;
Map(lines, :)   = 0;


%% generate Layers

% make sure that there is a target
if nLayers == 0
    Layers = [];
    uiwait(msgbox('Select a Map'));
else
```

```
    % loop from 2 to number of files
    for i=2:nFiles
        % imread generates a matrix containing the RGB value of each pixel
        rawLayer = imread(S{i,1});
        % find the different colors and locate them in the matrix rawLayer
        ends = findColor(rawLayer, 255, 0, 0);
        space = findColor(rawLayer, 255, 255, 255);

        % make sure that just the needed colors are used
        if (length(ends) + length(space)) ~= lines*columns
            error('Invalid input Layer.');
        end

        Layer = zeros(lines, columns);
        % fill the matrix Map with numbers representing the special areas.
        Layer(space) = 1;
        Layer(ends) = inf;

        Layers(:,:,i-1) = Layer;
    end
end

end


function [Entries] = findColor(Image, R, G, B)
%findColor: Finds all entries in an image of the specified RGB color.

[m,n,t] = size(Image);
if R > 255 | R < 0 | G > 255 | G < 0 | B > 255 | B < 0 | t ~= 3,
    error('Input error in function findColor');
end

search_px = [R;G;B];

Entries = [];
for i = 1:m,
    for j = 1:n,
        px = [Image(i,j,1); Image(i,j,2); Image(i,j,3)];
        if px == search_px,
            Entries = [Entries; i + j*m - m];
        end
    end
end
end
```

Listing 4: generateVectorfields.m

```
function [Vectorfields] = generateVectorfields(M, Layers)
%generateVectorfields: Computes the corresponding vectorfields on to a map
%   and x layers. Plots the results if needed.
%
% Input: Map-Matrix containing walls, slow areas, starts. Layer-Matrix
% containing information about target.
%
% Output: The output is a four dimensional cell array containing the
% vectorfields for each layer.
```

```
% k ist the number of layer which is synonymous with the number of targets.
[m, n, k] = size(Layers);
% find the walls, starts, slow areas in the map to write them in each layer
% matrix
Walls = find(M == 0);
Starts = find(M == 2);
Slow = find(M == 3);
% initialize Vectorfields as a cell with as many lines as existing targets.
% Lateron the lines will be filled up with the vectorfields and the
% distance matrix.
Vectorfields = cell(k,1);

% loop over all layers
for i=1:k
    Layer = Layers(:,:,i);
    Ends = find(Layer == Inf);
    % write the walls into the layer matrix
    Layer(Walls) = 0;
    % write the slow areas into the layer matrix
    Layer(Slow) = 1;
    % write the starts into the layer matrix
    Layer(Starts) = 2;
    % write the ends into the layer matrix
    Layer(Ends) = Inf;
    % generate the vectorfields and the distance matrix with the function
    % computeVF
    [VFX, VFY, I] = computeVF(Layer);
    % initialize the help matrix VF which contains the informations we got
    % with computeVF
    VF = zeros(m,n,3);
    VF(:,:,1) = VFX;
    VF(:,:,2) = VFY;
    VF(:,:,3) = I;
    % fill the i'th line of Vectorfields with VF
    Vectorfields{i,1} = VF;
end

end
```

Listing 5: computeVF.m

```
function [VFX, VFY, I] = computeVF(M)
%computeVF: Computes one vectorfield of a Map for each Layer
%   (each target)
%
%   The input is the m*n — Map containing information about walls, spaces,
%   ends positions.
%   The output are two matrices which contain the x and y components of the
%   vectors of a vectorfield in every point of the input matrix
%   representing the direction a passenger has to walk to follow the
%   shortest path to the nearest target. The additional matrix I contains
%   information about the length of way the traveller has to walk to the
%   target.
%
%   As defined in the loadSituation.m file, the codes which we need are:
%   Wall = 0, Space = 1, Exit = inf, Start = 2

% Implement the fast marching toolbox
```

```
path(path, 'fast_marching/');
path(path, 'fast_marching/toolbox/');
path(path, 'fast_marching/data/');

[m, n] = size(M);

% Find walls in the input matrix.
F = ones(m , n);
Walls = find(M == 0);
F(Walls) = 0;

% Find exits. Exit are set to infinity in the layer matrix
[ExitRows, ExitCols, V] = find( M == Inf );

% Generate exit vector. Must be done to use the fast marching toolbox
nExits = length(V);
Exits = zeros(2, nExits);
Exits(1, :) = ExitRows;
Exits(2, :) = ExitCols;

% Apply fast marching and gradient. The fast marching toolbox is the base
% of the field force and handles over the direction
options.nb_iter_max = Inf;
[D, S] = perform_fast_marching(F, Exits, options);
% generate the vectorfields VFX and VFY using the function gradientField
[VFX, VFY] = gradientField(D);

% I is a matrix containing the information about the distance to go to the
% chosen target.
[A, I] = convert_distance_color(D,M);

%% Uncomment following lines if colormap-plot needed

% fig = figure('visible', 'on');
% hold on
% imageplot(A);
% h= colorbar;
% set(h,'fontsize',15);
% hold off


%% Uncomment following lines if vectorfield-plot needed
%
% fig = figure('visible', 'on');
% hold on
% targ = find(M == Inf);
% star = find(M == 2);
% Plo = M;
% Plo(targ) = 1;
% Plo(star) = 1;
%
% imageplot(Plo);
%
% x = 1:10:n;
% y = 1:8:m;
% quiver(x, y, VFX(1:8:m,1:10:n), VFY(1:8:m, 1:10:n),0.7,'b');
% hold off
```

```
76
end
```

Listing 6: gradientField.m

```
function [FFX, FFY] = gradientField(M)
%gradient:  Calculate the gradient on a matrix M and ignore entries which
%           are infinitely large.
%
%   Input is a matrix M with a potential field and entries set to 'Inf'.
%   The infinite entries represent walls. This function generates a vector
%   field representing the field of M, ignoring all infinite entries.
%
%   This function is based on the gradient_special function used in the
%   base project.
[m, n] = size(M);

FX = zeros(m, n);
FY = zeros(m, n);
FFX = zeros(m ,n);
FFY = zeros(m, n);

%   Check every element of the input matrix.
for i = 1:m
    for j = 1:n
        %   Is M(i, j) part of the wall?
        if M(i, j) ~= Inf,
            %   Is M(i, j) on the border of the map?
            if j > 1 && j < n,
                %   M(i, j) is not on the border.
                %   Check the following cases:

                %   1. WřW
                if M(i, j - 1) == Inf && M(i, j + 1) == Inf,
                    FX(i, j) = 0;
                %   2. Wřř
                elseif M(i, j - 1) == Inf,
                    FX(i, j) = M(i, j) - M(i, j + 1);
                %   3. řřW
                elseif M(i, j + 1) == Inf,
                    FX(i, j) = M(i, j - 1) - M(i, j);
                %   4. řřř
                else
                    FX(i, j) = 0.5*(M(i, j - 1) - M(i, j + 1));
                end

            elseif j > 1,
                %   M(i, j) is on the right border.
                %   Check the following cases:

                %   1. Wř
                if M(i, j - 1) == Inf,
                    F(i, j) = 0;
                %   2. řř
                else
                    F(i, j) = M(i, j - 1) - M(i, j);
                end

```

```
            else
                %   M(i, j) is on the left border.
                %   Check the following cases:

                %   1. řW
                if M(i, j + 1) == Inf,
                    FX(i, j) = 0;
                %   2. řř
                else
                    FX(i, j) = M(i , j) - M(i, j + 1);
                end
            end

            %   is M(i, j) on the border of the map?
            if i > 1 && i < n,
                %   M(i, j) is not on the border.
                %   Check the following cases:

                %   1.  W
                %       ř
                %       W
                if M(i - 1, j) == Inf && M(i + 1, j) == Inf,
                    FY(i, j) = 0;
                %   2.  W
                %       ř
                %       ř
                elseif M(i - 1, j) == Inf,
                    FY(i, j) = M(i, j) - M(i + 1, j);
                %   3.  ř
                %       ř
                %       W
                elseif M(i + 1, j) == Inf,
                    FY(i, j) = M(i - 1, j) - M(i, j);
                %   4.  ř
                %       ř
                %       ř
                else
                    FY(i, j) = 0.5*(M(i - 1, j) - M(i + 1, j));
                end
            elseif i > 1,
                %   M(i, j) is on the bottom border.
                %   Check the following cases:

                %   1.  W
                %       ř
                if M(i - 1, j) == Inf,
                    FY(i, j) = 0;
                %   2.  ř
                %       ř
                else
                    FY(i, j) = M(i - 1, j) - M(i, j);
                end
            else
                %   M(i, j) is on the top border.
                %   Check the following cases:

                %   1.  ř
                %       W
```

```
                if M(i + 1, j) == Inf,
                    FY(i, j) = 0;
                %   2.  ř
                %       ř
                else
                    FY(i, j) = M(i, j) - M(i + 1, j);
                end
            end
        else
            %   M(i, j) is part of the wall.
            FX(i, j) = 0;
            FY(i, j) = 0;
        end

        %   Normalize vector
        if FX(i, j) ~= 0 && FY(i, j) ~= 0,
            FFX(i, j) = FX(i, j)/(sqrt( FX(i, j)^2 + FY(i, j)^2 ));
            FFY(i, j) = FY(i, j)/(sqrt( FX(i, j)^2 + FY(i, j)^2 ));
        elseif FX(i, j) ~= 0,
            FFX(i, j) = FX(i, j)/abs( FX(i, j) );
            FFY(i, j) = 0;
        elseif FY(i, j) ~= 0,
            FFX(i, j) = 0;
            FFY(i, j) = FY(i, j)/abs( FY(i, j) );
        end
    end
end
end
```

Listing 7: spawnPed.m

```
function  [pedM, nrped_mom] = spawnPed(s_area, pedM, nrped_mom, nrped_end,...
    pos, velo, vmax, weight, radius, tstart, tend, status, mtopix, t)
% spawnSystem: spawns a new pedestrian whenever there is enough space

n = length(s_area(1,:));

 % loop over all starting points
 for i=1:n
        % isfree is a variable either 1 (no pedestrian in the examined
        % square) and 0 (there is someone near the spawn point)
        isfree = 1;

        % define a square around each starting points to search for
        % pedestrians in it.
        for y = s_area(2,i) - 6 : s_area(2,i) + 6
            for x = s_area(1,i) - 6 : s_area(1,i) + 6
                if (nrped_mom > 0)
                    % iteration over all pedestrians
                    for k = 1 : nrped_mom
                        % set isfree to 0 if someone is in the square
                        if (pedM(pos, k) == [x; y])
                            isfree = 0;
                        end
                    end
                end
            end
        end
```

```
        % pedestrians can be spawned if isfree is 1 and the maximum number
        % of pedestrians is not yet achieved
        if (isfree == 1 && nrped_mom < nrped_end)
            % raise the number of pedestrians by one
            nrped_mom = nrped_mom + 1;
            % the pedestrian will spawn at the examined point
            pedM(pos, nrped_mom) = [s_area(1,i) s_area(2,i)]';
            % each has pedestrians has a very little velocity at the
            % beginning
            pedM(velo, nrped_mom) = 1e-5 * ones(2,1);
            % calculate vmax using a distribution with the mean 1.5 *
            % mtopix and the deviation 1.5
            pedM(vmax, nrped_mom) = randnvect(1,1.5,1.5 * mtopix);
            % set the weight of each pedestrian
            pedM(weight, nrped_mom) = 80;
            % set the width of each pedestrian
            pedM(radius, nrped_mom) = 0.3 * mtopix;
            % define the starting time
            pedM(tstart, nrped_mom) = t;
            pedM(tend, nrped_mom) = inf;
            % [0: not started; 1: on the road; 2: on the stairs;
            % 4: finished]
            pedM(status, nrped_mom) = 1;

        end
end

end
```

Listing 8: searchStartingpoints.m

```
function s_area = searchStartingpoints(M)
%searchStartingpoints returns a matrix s_area with starting positions and
%nbrs_starea with the numbers of starting points.
%   The function takes a matrix M and returns an matrix s_area
%   with all the positions where an entry in M is equal to value.


% search matrix M for starting areas and stores the the values in an row-
% and collumvector
[row,col,v] = find(M == 2);

% creates an matrix s_area with the positions [x,y] as entries
for i=1:length(row)
    s_area(:,i)=[col(i);row(i)];
end

end
```

Listing 9: StairsDecision.m

```
function [layerNr, stairselected, statstairs] = StairsDecision(xpos,...
    ypos, Vectorfields, layer, stairselected, Layer)
% StairsDecision: hands over the chosen stairs for each pedestrian.
%
% Input: X-Position, Y-Position and number of other pedestrians, who are
```

```
%   going to the each target.
%
% Output: The chosen target and the updated vector stairselected.

% initialize statstairs as 0 (condition to run the function)
statstairs = 0;

m = size(Vectorfields,1);
% distance: vector containg the distance to each target
% set distance to inf first
distance = inf(m,1);

% distance: vector containing distance to each target
for i=1:m
    distance(i,1) = Vectorfields{i,1}(ypos,xpos,3);
end

% locate distance to clostest target, c is the value of the layer with the
% minimal distance
min_distance = min(distance);
V = find(distance == min_distance);
if size(V,1) > 1
    c = V(1);
else
    c = V;
end

% choose closest target if there is only one target.
if m == 1
    if layer ~= c
        % raise stairselected by one
        stairselected(c) = stairselected(c) + 1;
        layerNr = c;
        % pedestrian made his decision, he will not able to choose the
        % target anymore.
        statstairs = 1;
    else
        layerNr = c;
    end


elseif m > 1

    % find positions of the targets
    [row,col] = find(Layer(:,:,c) == inf);

    % creates an matrix end_x with the positions [x,y] as entries
    endpoint(:,1)=[col(1);row(1)];

    % if condition in order to find out where the pedestrian stays.
    % Afterwards, he can decide between the stairs on the lefthand side and
    % the stairs on the righthand side
    if xpos <= endpoint(1,1) && c == 1
        d = c;
        sec_distance = distance(d);
    elseif xpos <= endpoint(1,1) && c ~= 1
        d = c - 1;
        sec_distance = distance(d);
```

```
64      elseif xpos > endpoint(1,1) && c ~= m
            d = c + 1;
66          sec_distance = distance(d);
        elseif xpos > endpoint(1,1) && c == m
68          d = c;
            sec_distance = distance(d);
70      end

72      % calculate the relative distance between closest and second closest
        % distance.
74      reldiff_dist = min_distance/(sec_distance + min_distance);

76      % calculate the relative amount of pedestrians going to the closest
        % and second closest target.
78      reldiff_ped = stairselected(c) / (stairselected(c) +...
            stairselected(d));
80


82      %% StairsElection-Model
        % each pedestrian can choose between closest and second closest target.
84
        % take closest stairs if pedestrian just got spawned
86      if layer == 0

88          % the pedestrian takes the closest target when he enters the
            % platform.
90          stairselected(c) = stairselected(c) + 1;
            layerNr = c;
92
        % the pedestian is already on its way
94      else

96          % the decision depends on the absolute amount of pedestrians, who
            % head to each target
98          p_abs = 0.02 * (stairselected(c) - 35);

100         % the pedestrian is not going to change stairs if p_abs is near 0
            if p_abs > rand() && reldiff_dist > 0.35
102
                % calculate the total probability to change stairs depending
104             % on the weight of the to probabilities Inf_reldiff_dist
                % and inf_reldiff_ped.
106             totProb = 4 * (reldiff_dist)^3 * (reldiff_ped)^3;

108             % pedestrian takes second closest target if the total
                % probability is high
110             if totProb > rand()
                    layerNr = d;
112                 stairselected(c) = stairselected(c) - 1;
                    stairselected(d) = stairselected(d) + 1;
114                 statstairs = 1;
                else
116                 layerNr = c;
                end
118         else
                layerNr = c;
120         end
        end
```

```
122 end

124 end
```

Listing 10: randnvect.m

```
function [y] = randnvect(number, deviation, center)
%RANDVECT Vector with random values around center
%   number: quantity of returned values in vector (number,1)
%   deviation: deviation of the random values (not mathematically correct!)
%   center: middlepoint of the random values

y = randn(number,1)*deviation + center;

% sets a limit for the maximum and the minimum so that no negative and not
% too high numbers are generated
for i = 1 : number
    if y(i) < center/2
        y(i) = center/2;
    elseif y(i) > center * (3/2)
        y(i) = center * (3/2);
    end
end

end
```

Listing 11: Print.m

```
% Modeling and Simulating Social Systems with MATLAB
% BlueMen – Pedestrian Dynamics
% Dominic Hänni, Patrick Manser, Stefan Zoller

% this file – as the name says – is for printing the situation. Necessary
% to make the print working is the input matrix M (M), the matrix Layer
% (containg all targets) and the input cell pedM_cell. Output will be a
% nice simulation and a .avi file containing the video.

% get size of the map to intialize a figure with that size
[m,n] = size(M);
h=figure('Position',[ 0,180,2*n,2*m]);

% Implement the fast marching toolbox
path(path, 'fast_marching/');
path(path, 'fast_marching/toolbox/');
path(path, 'fast_marching/data/');

hold on

% generate a movie of the iteration. Make sure that you are not moving the
% window because videowriter actually makes a screenshot of each frame. It
% wont work if you move the window. The while loop is to not overwrite
% existing files.
save_vid = 1;
while save_vid < 20
    vid = 'video_V1-';
    version = num2str(save_vid);
    data_type = '.avi';
```

```
    if exist([vid version data_type]) == 0
        vidObj = VideoWriter([vid version data_type]);
        break
    end
    save_vid = save_vid + 1;
end
vidObj.FrameRate =  8;
open(vidObj);

% get the number of layers
m = size(Layer,3);

% ends is a help-cell in which vectors with the locations of all targets
% are saved.
ends = {1,m};

% Plo is a help-matrix containing the colors of all the elements appearing
% in the final plot
Plo = M;
% locate the slow-areas
slowarea = find(M == 3);
% locate the starts
starts = find(M == 2);
% locate the ends
for i=1:m
    ends{1,i} = find(Layer(:,:,i) == inf);
end
% slowareas are white
Plo(slowarea) = 1;
% starts are a 70% grey
Plo(starts) = 0.7;
% ends are a 30% grey
for i=1:m
    Plo(ends{1,i})=0.3;
end

% plot the image into the figure
imageplot(Plo);

% if a trainmap exists, it will appear in the map
trainiter = size(Train,2);
if trainiter > 1
    trainloop = 0;
    k = 0;
    while trainloop <= trainiter
        % clear the last iteration step
        clf
        hold on

        Plo(165:219,1:trainloop) = Train(165:219,(trainiter+1-trainloop):...
            trainiter);

        % plot the map
        imageplot(Plo);

        % save the current frame and write it into the video file
        currFrame = getframe(h);
        writeVideo(vidObj, currFrame);
```

```
88
        % function to brake the train
90      breakfunc = -(16/trainiter)*trainloop + 20;
        k = floor(breakfunc);
92      trainloop = trainloop + k;

94      % the smaller the pause the faster the print
        pause(0.03)
96  end

98  % small break after the train arrives
    for i=1:20
100     currFrame = getframe(h);
        writeVideo(vidObj, currFrame);
102
        pause(0.02)
104 end
end
106
% stop criterion for the video: the video keeps going until the last
108 % pedestrian reached his target
StopIter = 1;
110 while StopIter < T/dt
    % pedM_cell has matrices in it, which are zero when every pedestrian
112 % has disappeared
    if pedM_cell{1,StopIter}(1,1) == 0
114     break
    else
116     StopIter = StopIter + 1;
    end
118 end

120 % iteration in which the pedestrians are moving with every step. The
% informations about velocities, positions and so on are saved in the
122 % pedM_cell
for i = 1 : StopIter
124 % clear the last iteration step
    clf
126 hold on

128 % plot the map
    imageplot(Plo);
130
    % text frame containing the current time
132 mom_time = num2str(i*dt);
    time = 'current␣time:␣';
134 text(5, 38, [time mom_time]);

136 % text frame containing the current number of passengers
    mom_ped = num2str(length(find(pedM_cell{1,i}(status,:)==1)));
138 numb_ped = 'number␣of␣pedestrians:␣';
    text(5, 18, [numb_ped mom_ped]);
140
    % plot every pedestrian who is moving. pedestrians who reached
142 % the target wont be plotted
    for j=1:nrped_mom
144     if pedM_cell{1,i}(status,j) ~= 4
            drawPedestrian(pedM_cell{1,i}(pos,j),pedM_cell{1,i}(radius,j))
```

```
        end
    end

    % save the current frame and write it into the video file
    currFrame = getframe(h);
    writeVideo(vidObj, currFrame);

    % the smaller the pause the faster the print
    pause(0.03)

end

% video is done and saved in the same folder as the Print.m file.
close(vidObj);
```

Listing 12: drawPedestrian.m

```
function drawPedestrian(position, r)
% This function is for drawing a Pedestrian
% INPUT:
%   position: [x,y] vector with central point of the Pedestrian
%   r: Radius of the Pedestrian

% source: lecture notes (week 4), FS 2012, Karsten Donnay and Stefano
% Balietti

hold on

% Define the coordinates for the Pedestrian
angles = 0:0.1:(2*pi);
Pedestrian_x = r*cos(angles);
Pedestrian_y = r*sin(angles);

% Draw the Pedestrian
patch(position(1)+Pedestrian_x, position(2)+Pedestrian_y, 'b')

end
```