Automated Routing in Pedestrian Dynamics

Fachhochschule Aachen, Campus Jülich
Medizintechnik und Technomathematik,
Technomathematik

Masterthesis from Arne Graf

2015-10-06

Automated Routing in Pedestrian Dynamics

Fachhochschule Aachen, Campus Jülich
Medizintechnik und Technomathematik,
Technomathematik

Masterthesis from Arne Graf

2015-10-06

Abstract

In this thesis, the effect of an alternate floor-field was analyzed, by using it in a newly composed test-model for pedestrian dynamics. In the simulation of pedestrian (crowd) movement, the routing of agents[1] is an integral part. Routing can be seen as the composition of two aspects: the global pathfinding through a geometry and the avoidance of static or dynamic obstacles[2] (like walls or other agents) in a local[3] situation.

The history of pedestrian simulation shows various models with different answers to the question of navigation. Many of which make use of manually added elements[4] to solve the global pathfinding, which enable the user to simulate crowd movement in that very geometry. Other models use an automated algorithm, that will supply a navigation direction calculated from the agent's current position, the goal (area) and the geometry data. The model described by Dietrich is one of the later. It uses the solution of the Eikonal Equation (see chapter 3.1), which describes a 2-D wave-propagation. The wave starts in the target region and propagates through the geometry. To navigate agents, they are directed in the opposite direction of the gradient of said solution of the Eikonal Equation. It is to be noted, that the solution of the Eikonal Equation can be calculated beforehand and does not contribute to the runtime of any given simulation scenario. The Routing using the plain floor-field will yield non-smooth pathways as described later. This could pose a problem for some models. Dietrich shows the existance and uniqueness of his problem-formulation by using the theorem of Picard-Lindelöf. [5] To apply this theorem, Lipschitz-continuous first derivatives of the input-functions must be given. Dietrich solves that problem by the use of a mollifier, which basically takes a locally integrable function and returns a smooth approximation.

In this thesis, a different floor-field is described, which solves above issue (non-smoothness) as a welcome side-effect. The research-group in Madrid (add ref) is working on the safe navigation of robots. They are not to follow paths, which cut corners (which come close to any obstacles). A so-called distance-field is created and used as we see later. The welcome side-effect is in *smooth*

[1] An agent is the representation of a pedestrian in the simulation. Depending on the used model, an agent incorporates some kind of artificial intelligence or basic agent-attributes only (like size, speed attributes, etc.). In the latter case the model takes over the task of navigating agents.

[2] collision detection

[3] local in time and/or in space

[4] like some sort of domain-decomposition, e.g. through helplines

[5] Picard-Lindelöf theorem: Consider the initial value problem

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0, \quad t \in [t_0 - \epsilon, t_0 + \epsilon].$$

Suppose f is uniformly Lipschitz continuous in y and continuous in t. Then, for some value $\epsilon > 0$, there exists a unique solution $y(t)$ to the initial value problem on the interval $[t_0 - \epsilon, t_0 + \epsilon]$.

pathways through the avoidance of walls and corners. The researchers take that approach even further, by reducing any geometry into a graph of edges and knots and thus having the domain in which the 2-D wave propagates reduced dramatically. Their intent is to re-calculate the floor-field in real-time using it for the reduced view-field of the robot's sensors. Our interest in this sleight of hand is different. We welcome the smoothness of the resulting pathways and take special interest in the behavior of agents close to obstacles. The floor-field itself shows pathways, which show a wall-repulsive character in the vector-field. This phenomenon enables us to formulate a new model, one that uses an altered floor-field. Thanks to the rich experience of Chraibi in creation and testing of pedestrian dynamics models, we follow his intuition to use that altered floor-field in a new model. The results seen in the simulations show remarkably good behavior. The model is easy to use, fast and shows superior characteristics in routing through complex geometries. The extent to which we alter the floor-field is subject to our analysis.

Contents

1 Pedestrian Dynamics: Introduction

Pedestrian Dynamics defines a field of research trying to understand the kinematic and mechanic of pedestrian crowd movement. Understanding, how crowds will react in different geometries under various circumstances, will lead to the ability to design our environment to best fit the needs of civil and security engineering. It is applied to safely conduct large events, to create architecture (traffic infrastructure), through which large crowds can safely be moved and to optimize evacuation time in case of an emergency. To simulate crowd behavior, many models exist with different characteristics. The history of the analysis reaches back to the 1970's to Predtetschenski and Milinkskii, as Kemloh states in his dissertation. Since then, new models have been described throughout the decades. To maintain orientation, these models can be grouped into classes in the following manner, common in Pedestrian Dynamics (Chraibi, 2012):

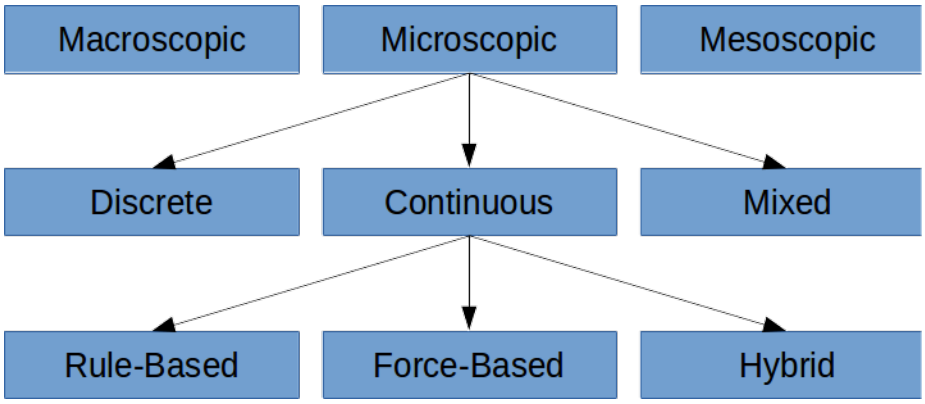


Figure 1.1: A possible hierarchical classification of models in pedestrian dynamics by Chraibi, 2012

Macroscopic models describe crowd behavior without the need to characterize individuals, which make up the crowd. Aggregated values, like density or flow for instance, are used to describe the dynamic within the system. Imagine a model, which describes the change of the density over time throughout a geometry. Such a model can be mathematically captured by an PDE. The action of a single agent is neglected and it is assumed, that aggregated values are sufficient to describe the crowd behavior. Thus a crowd is seen as a continuous fluid, which can be modeled by the aggregated observables (density, speed, flow) only. No inter-particle relations are explicitly considered. Larger roadmap- and city-traffic-simulation are fields, where macroscopic models are widely spread and can supply travel times and point out bottlenecks. (include ref to commercial software)

Microscopic models consist of mathematical formulations describing the state and/or the interactions of every agent. Each agent has a position in the domain

and interacts with his/its(?) environment. It is assumed that the behavior of any crowd is only based on the sum of these individual actions. Within the model, these individual actions obviously must be far different from the attempt to model the complete, complex system of a person's psychology, shaping its motivation of movement inside a crowd. It is desirable to have few and simple equations to model the agent's movement. Equations, that do scale as good as possible to achieve real-time capability for simulations. If one can not achieve real-time capability, at least reasonable computing time is a must-have-criteria for the model to be successful. A popular starting point origins in the modeling of the behavior of electrical charges in an electro-magnetic field. Charges of the same sign act on each other with a repelling force. This effect is used in the modeling of the natural collision-avoidance of a person to other persons, walls and obstacles in pedestrian dynamics. Agents can be modeled to react equally to charges in the electro-magnetic field of the surrounding environment. Let us add a driving force, that acts on the agent, forcing him towards its goal and we end up with microscopic model called "social force model".[6]

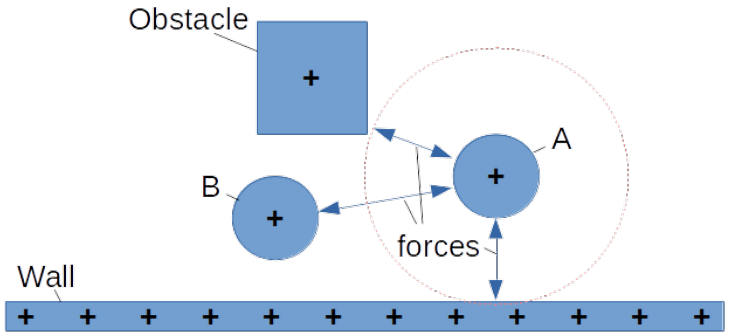


Figure 1.2: Forces acting on agent A from: wall, obstacle and agent B

We will leave the overview and set the stage for the ODE based models, for in this category our new model is to be found.

2 ODE based, microscopic models

We now focus on one group, ODE-based microscopic models, which are very successful in producing system phenomena like congestions in front of bottlenecks

[6]This simplified describtion shall be enough for this introduction. For further reference, please see corresponding literature. A comprehensive insight in SFMs is given by Chraibi's "Validated force-based modeling of pedestrian dynamics".

and showing good accordance with experimentally determined fundamental diagrams [7] [Dietrich, 2014]. The most popular models depend on ODEs of first and/or second order. The order of the ODE is connected to certain characteristics which will be observable in the simulations. To understand the connection, we will take a look at an example. Newton mechanics teaches us, that force is equal to acceleration times mass. Pedestrian models, that derive from this rule, most certainly lead to second order ODEs, as the location of an agent, derived twice, yields the acceleration. Acceleration driven models show oscillations in their trajectories (if the system is not over-damped). Agents seem to sway left

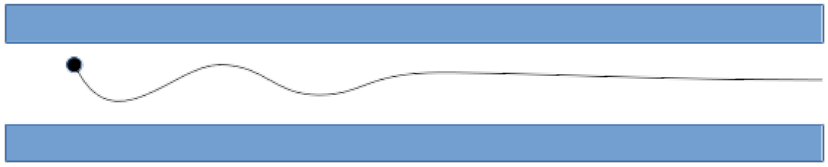


Figure 2.1: Trajectory in a force-based model, when starting close to a repulsive wall. (qualitative)

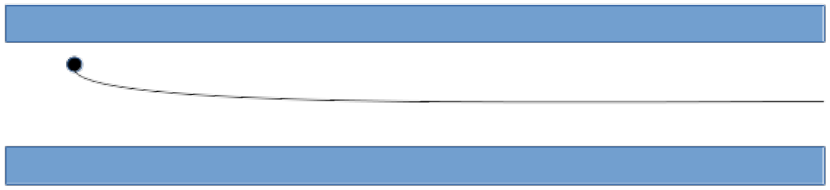


Figure 2.2: Trajectory in a force-based model, when starting close to a wall and parameters yield overdamped system. (qualitative)

and right and do not steer as wanted. Various researcher/authors try to find new models with enhanced characteristics in terms of directing agents [Pan et al, 2007; Moussaïd et al., 2011; Chraibi et al., 2011].

The difficult calibration of a model is another important issue. This type of models find their limit, if one is to find one constant set of parameters for various situations. Best results are achieved with a special calibrated set of parameters for each situation. The change of parameter sets would be problematic, if we want to make extensive use of computational(?) mathematical (parallel) solvers. In terms of ergonomics, a constant set would be more user-friendly.

[7] Fundamental diagrams plot the (pedestrian) flux/flow over the (pedestrian) density. (include figure?)

Velocity based models, which often lead to first order ODEs, make up another important group. These models change the agent's velocity directly and thus show much better trajectories in terms of oscillation. The TEST[8]-model of this thesis (see below), is partly derived from a velocity-based model, Dietrich's *Gradient Navigation Model*. Dietrich motivates the creation of his model. He intends to overcome short-comings of both groups, the SFM and the Optimal Step Models, yet have the positive characteristics remain. Namely he wants to achieve the following features:

- no model induced oscillations
- fast solver methods through one-dimensional acceleration, smooth functions and small gradients
- realistic timing through bulk updates
- mathematical treatment is easily possible and used to calibrate the model[9]

A major step towards this goal is the use of a navigation field, the solution to the Eikonal Equation. This approach, introduced by Hartmann[2010], provides routing and navigation information. In the Gradient Navigation Model Dietrich divides the navigation into two components. A static and a dynamic navigation vector is described. The static navigation field comprehends the geometry, the dynamic navigation field integrates pedestrians and mobile obstacles. It is clear, that the dynamic navigation field must be computed for every time-step throughout the simulation.

Besides oscillation and calibration, there is a third issue, on which we will feast in the next chapter: Overlapping. It describes a situation, where an agent's position is invalid either because the agent overlapps with another agent or because his simulated presence overlapps with a wall or even an obstacle. Once an agent is fully clipped through a wall-surface, faulty trajectories are most certain.

These issues are to sketch some prominent motives to further develop pedestrian models and search for yet another model, which might overcome some of the shortcomings and can produce as good results as existing models already provide.

3 Modelling

In the latter, a new model is described,

- aiming for the avoidance of faulty interaction of pedestrians and walls

[8] substitute name of model
[9] features copied from his paper, but they kind of break the context...

- while maintaining the positive characteristics of row-formation, stop-and-go waves and such - like seen in pedestrian crowd behavior/experiments/reality. (split up into more sentences).

In many of the existing models (using mathematical formulations in the continuous space/domain), agents breach wall-surfaces and get stuck inside of walls. This undesired phenomenon shows the challenge in calibrating forces and parameters of existing models, so that agents show valid natural behavior while not getting overlapping in extreme situations. Especially in situations of high crowd density, e.g. when facing bottlenecks, overlapping can occur. The model or the data-post-processing needs to find a special treatment of such artifacts in the data. It leads to problems in counting, flow-calculation, simulation-stop-criterion and such. There are three mechanics used in the model to avoid

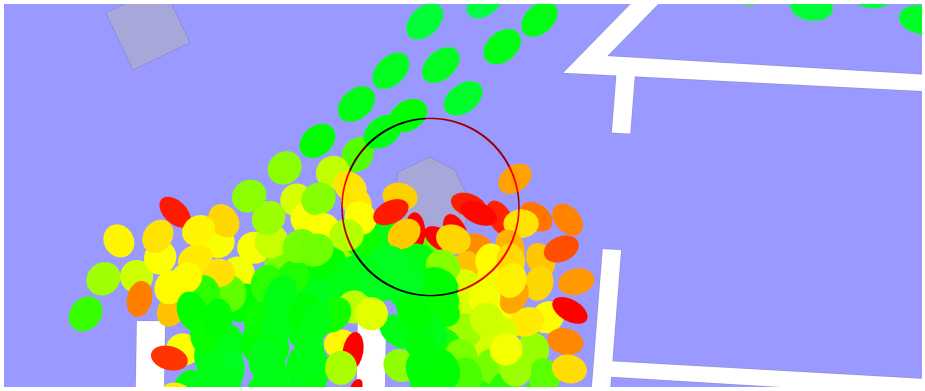


Figure 3.1: Agents got pushed into obstacles by the large amount of other agents. (Simulated with SFM, wall-forces reduced)

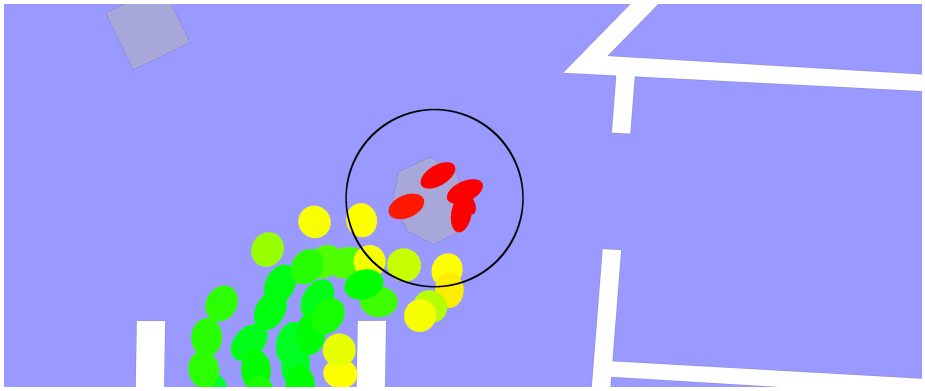


Figure 3.2: Agents remain inside obstacle. (Simulated with SFM, wall-forces reduced)

"overlapping/clipping" in the vicinity of walls (include a figure for each):

1. The routing of pedestrians makes use of the eikonal-equation, computed with an inhomogeneous speed-function/time-cost-function, $s(x)$, whose resulting floor-field[10] favors keeping a distance to obstacles, walls and corners.

2. The angle between an agent's moving-direction and the wall-surface-perpendicular affects the moving speed if and only if the agent's moving vector includes a component geared towards the wall.

3. If an agent's distance to a wall drops below a fixed parameter, he is redirected to move parallel to the wall if and only if the agent's moving vector includes a component geared towords the wall.

In order to keep the model simple, repulsive wall forces as seen in Social Force Models are omitted. An analogy to repulsive pedestrian forces though is used to keep agents from colliding with each other. The model differs from SFMs, as in SFMs, other agents repulsive forces are transformed into acceleration vectors and from there into a velocity component, which is part of the agent's velocity. In this model though, repulsive forces are not treated as Newton mechanics teaches us, but are only used to factor the repulsive pedestrian effect into a direction component. The magnitude on the other hand is effected by the other

[10] see chapter **Eikonal Equation, Safe Navigation using the Floor-field**

agents only to a certain degree (as discussed below).

Definitions:

$$
\begin{array}{llllllll}
d & : \Omega & \ni \vec{x} & \longrightarrow & d(\vec{x}) & \in \mathbb{R} & := & \text{distance to the closest wall} \\
P & : \mathbb{R}^2 \times \Omega & \ni (\vec{v}, \vec{x}) & \longrightarrow & P(\vec{v}, \vec{x}) & \in \mathbb{R}^2 & := & \text{orth. proj. of } \vec{v} \text{ onto closest wall of } \vec{x} \\
v_{ff} & : \Omega & \ni \vec{x} & \longrightarrow & v_{ff}(\vec{x}) = \vec{v}_{ff} & \in \mathbb{R}^2 & := & \text{floor-field at position } \vec{x} \\
g & : \mathbb{R}^2 & \ni \vec{v} & \longrightarrow & g(\vec{v}) & \in \mathbb{S}^2 & := & \text{proj. onto the unit-sphere in } \mathbb{R}^2
\end{array}
$$

TEST Model:

$$
\begin{aligned}
\Delta\vec{x}_n &= \Delta t \cdot \vec{v}_{n,res} \\
\vec{v}_{n,res} &= \begin{cases}
\left(1 - \frac{1}{2}\left[\langle \vec{v}_n, -\nabla\hat{d}\rangle + |\langle \vec{v}_n, -\nabla\hat{d}\rangle|\right]\right) & P(\vec{v}_n) \quad : \quad d(\vec{x}) < 0.1 \\
\left(1 - \frac{1}{2}\left[\langle \vec{v}_n, -\nabla\hat{d}\rangle + |\langle \vec{v}_n, -\nabla\hat{d}\rangle|\right]\right) & \vec{v}_n \quad : \quad 0.1 < d(\vec{x}) < 0.2 \\
& \vec{v}_n \quad : \quad d(\vec{x}) > 0.2
\end{cases} \\
\vec{v}_n &= 0.8 \cdot \vec{v}_{n-1,res} + 0.2 \cdot g\left(g(\vec{v}_{ff}) + g(\sum \vec{v}_{repP,i})\right)
\end{aligned}
$$

What might seem curious at first, is the fact, that both, the navigation field $\vec{v}_{ff}$ and the *sum* of pedestrian forces $\sum \vec{v}_{repP,i}$, are restricted to the length of 1 unit by g. Then their sum in turn is restricted again. This obviously breaks the principle of superposition of forces. We cannot talk about a force-based model here and loose the analogy to Newton's second law the first time applying g on the sum of forces $\sum \vec{v}_{repP,i}$. The resulting vector indicates a new orientation and a slow-down mechanic, as the vector can be of length ≤ 1 unit.

The sum of the navigation field (static) and the accumulated pedestrian forces (dynamic) get restricted by g and then weighted by 20%, the speed vector of the last time-step gets weighted by 80%. This is done to reduce flickering[11] of agents and must be kept in mind, as this approach could result in a tendency to oscillate. If oscillation should occur in any case, the weight should be shifted away from the last time-step.

In the next step, we process $\vec{v}_n$. It is checked, how distant the next wall is and if the agent is close to any wall, we build the scalar product $\langle \vec{v}_n, -\nabla\hat{d}\rangle$ to evaluate,

[11] We want to shortly address a second alternative approach. If one is willing to accept flickering agents with fast changing orientations, one could ommit the speed-vector of the time-step $n-1$ and postprocess the trajectories. As we only get positions at discrete time-steps, one could easily create a smooth trajectory by using *B-splines*.

if there is a orientation towards the closest wall. If there is, the agent is slowed down. If the agent is already very close to the wall, the velocity-component towards the wall gets neglected. Thus, the agent gets directed to move parallel to the wall.

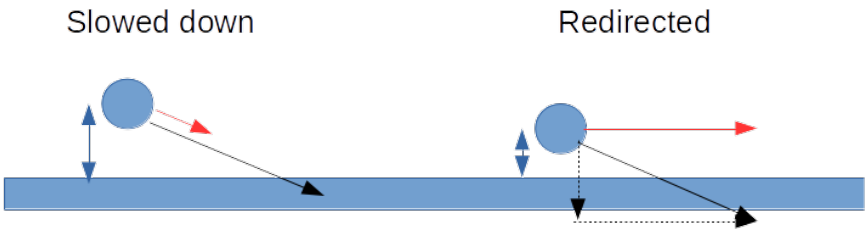


Figure 3.3: Alteration of the speed-vector to avoid clipping: Slow down, Redirect

3.1 Eikonal Equation

The "Eikonal Equation" in a domain Ω, subset of $\mathbb{R}^n$,

$$\begin{aligned} |\nabla u(\vec{x})| &= F(\vec{x}), \quad \vec{x} \in \Omega, \\ \text{s.t.} \quad u|_{\partial\Omega} &= 0 \end{aligned}$$

yields "first-arrival-times" $u(\vec{x})$ in a spacial domain, provided a target region within the domain as input as well as a slowness-field $F(\vec{x})$. A valid interpretation of "first-arrival-times"-iso-lines is to picture a wavefront at a given time t, originating in the target region ($t = 0$) and propagating throughout the spacial domain Ω with the given speed $v = \frac{1}{F(\vec{x})}$ while flowing around any obstacles (see figure 3.4).

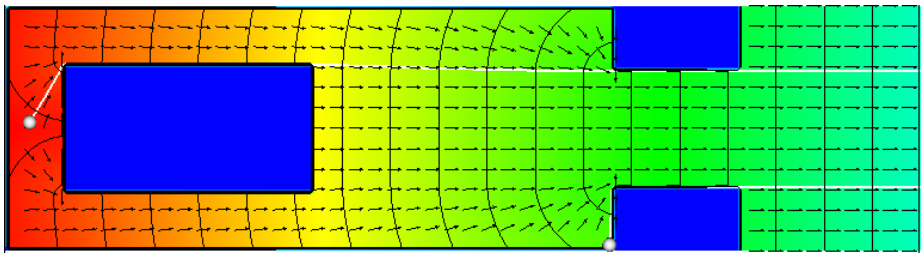


Figure 3.4: Isolines of "time-cost".

Given a discretization of the domain Ω and the target region $\partial\Omega$, the solution to the Eikonal Equation can be approached (angenähert) by using the

Fast-Marching Algorithm. The algorithm provides a first order approximation, yet sufficient for our cause (pedestrian navigation). Computing-time of Fast-Marching is independent[12] of the complexity of obstacles and walls.

The negative gradient $-\nabla u$ of the "time-cost" will be a useful tool in the routing of pedestrians/agents to the target region used as part of the algorithm's input. The Fast-Marching algorithm is described in the appendix in detail for further reference.

We will refer to the result of the Fast-Marching Algorithm as "floor-field". To successfully use these floor-fields, we will discuss and analyze a modification, which gives us a smooth floor-field, as proposed in roboticslab.uc3m.es.

3.2 Safe Navigation using the Floorfield

When using the plain approximation to the Eikonal Solution, agents anticipate a non-smooth pathway that leads very close to walls (see white trajectories in figure 3.4). In most of the models for pedestrian dynamics, pedestrians, which are very close to walls or obstacles, could overlap with them in rare occasions. Agents might leave the valid domain and find themselves captured inside walls or obstacles. In the model described in this paper, we aim to fix that problem. In reality, we can observe, pedestrians avoiding walls and obstacles through keeping a certain distance.

Therefore, it is desirable to define a modified quality of an optimal route, which accounts for a minimal arrival time and a safe pathway. Safe in respect to avoiding the vicinity of walls and obstacles, if and only if possible. If a space is very crowded (high density), then agents should make use of the given space even if that means getting close to walls.

This crowd behavior, described above, is commonly achieved with adding a repulsive characteristic to walls.

In the "social force model", the walls will have a repulsive force pointing perpendicular to the wall-surface, aiming to keep agents away from the wall. These forces need to be calibrated to work as intended.

Smaller forces might not be strong enough to avoid overlapping with the wall if an agent is in between a wall on one side and many other agents on the other side. The agents on the other side affect that one agent, forcing him towards the wall, while the wall itself acts on the agent in the opposite direction.

If the repulsive wall forces are too strong, pedestrians will not use the space close to a wall, even if the domain is very crowded.

It is a difficult task, to find a set of parameters, that work as desired in a broad set of situations and geometries.

[12] Fast-Marching completion-time depends mainly on the length of the wavefronts. If the geometry leads to small lengths, as in geometries with large amounts of narrow corridors, completion time decreases.

Instead of modeling the repulsive character of walls (seen as the avoidance of walls by pedestrians) via repulsive wall forces (social force model), we modify the floor-field in a way, that pathways avoid the vicinity of walls to a certain, adjustable degree, thus integrating this repulsive character into the navigation/routing.

How can an agent "avoid" the close vicinity of any wall or obstacle?

3.3 Distances-Field

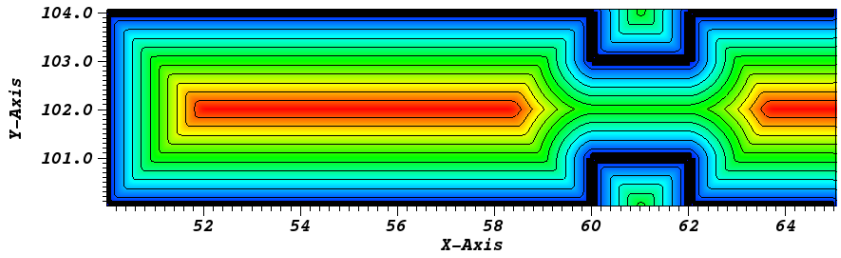


Figure 3.5: Distances Field of the bottleneck geometry

Having above question in mind, we first need to introduce and understand the *Distances-Field*, a function d living on the spacial domain Ω, and d holds information, how distant the closest wall is. This function will prove useful when altering the one floor-field, we will use for routing. To avoid confusion, let it be emphasized, that this distances-field will be used in two different parts of the model. It is used to:

1. create a Direction-to-Wall-Field (vector-field) and
2. to create a slowness-field (scalar-field) to initialize the Fast-Marching algorithm of the Navigation-Field[13] (vector-field).

[14]

ad 1: The Direction-to-Wall-Field is a normalized vector-field. Each vector has unit length. The orientation is gained by the negative gradient to the Distances-Field. This way, every given vector at $\vec{x}$ directs to the closest wall of that given grid-point $\vec{x}$.

[13] ask MC, about denotation of floor-field (vector-field or field of scalars)
[14] change "target" to "origin" in figure? easier to understand?

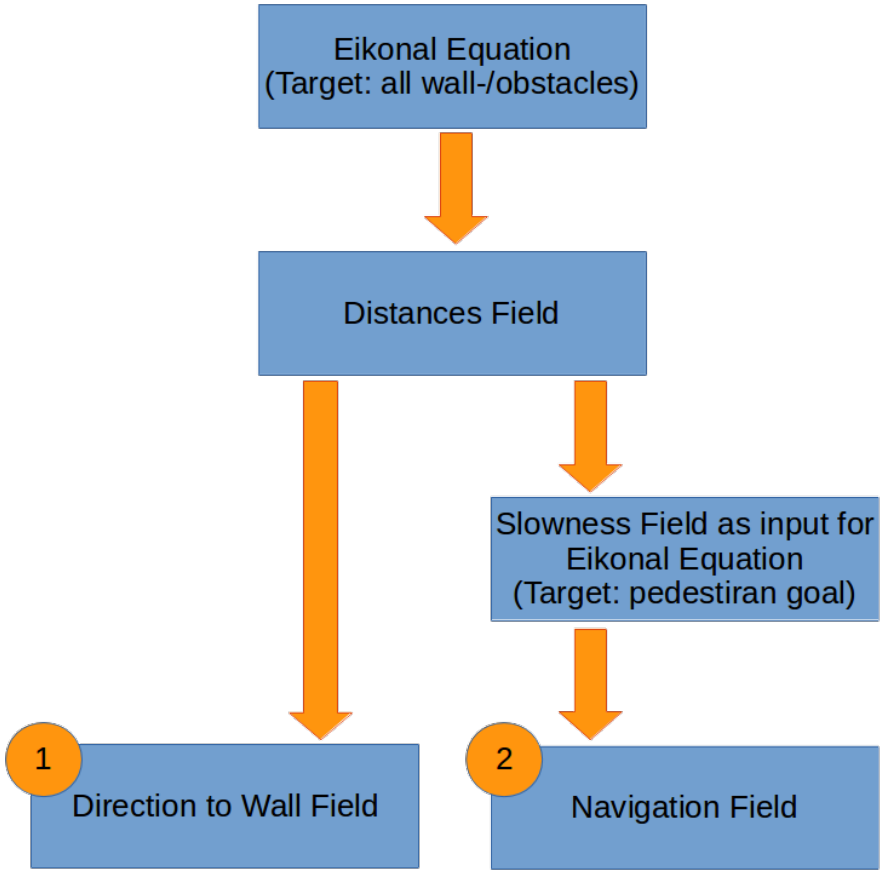


Figure 3.6: Usage of the Distances-Field in the vector-fields

ad 2: The function value $d(\vec{x})$ will be used to create a slowness-field on the discrete grid. This slowness-field holds the value, which determines, how slow the 2-D wave will propagate over the grid-point in the final *navigation*-field. Points, that are relatively close to a wall, will have a correspondingly[15] low value $d(\vec{x})$. Therefore, the 2-D wavefront will slow-down over these points. Routes/Pathways passing these points will take more time and be less optimal in a sense, that combines distances and wall-avoidance. We will refer to the value $u(\vec{x})$ as the *time-cost-value* (*cost* in short).

« explain the pre-step incl. the threshold "cut-off"; have some nice pictures »

[15] outch!

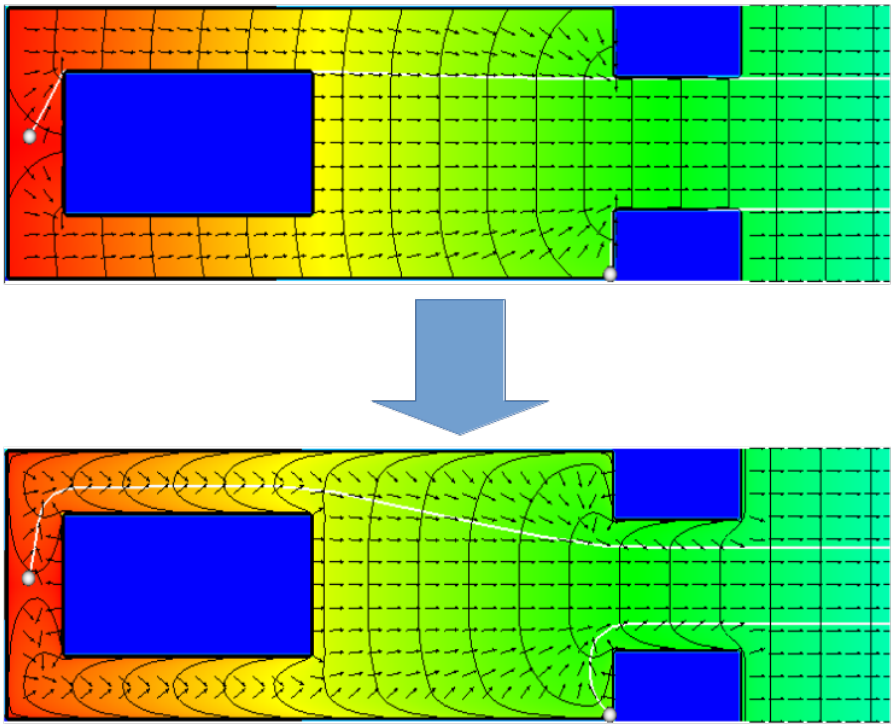


Figure 3.7: Isolines of a floor-field (above) compared to isolines of the enhanced floor-field (below). Sample trajectories in white.

3.3.1 Cost of a "full" preprocessing step

Close to all time of the needed computation spent on the enhanced floorfield, the prohibition of overlapping, is spent in a preprocessing step before the actual simulation starts and therefore does not effect the real-time factor. A factor, which is a prominent metric, when comparing different models. In this TEST-model, the preprocessing is increased compared to Dietrich's *Gradient Navigation Model*. Where he uses the Eikonal solution once (mollifiers not considered), the TEST-model uses two runs through the Fast-Marching algorithm. In this chapter we want to elaborate on the doubled effort we spend.

The granularity of the rectangular grid governs the cost of the FM[16]. For pedestrian navigation, we chose a point-to-point distance of neighboring grid-points of $0.0625m$. A geometry spanning $100m \times 100m$ can be processed in (enter profiling results). Most of the computing time is spend on the output.

The Fast-Marching algorithm is that fast, that it basically can be neglected

[16] Fast-Marching

when rating the performance. A prequel Fast-Marching run does not change that verdict. The usage of the Direction-to-Wall Field in the model shows no more overlapping and seems easily worth the cost. During runtime, using the floor-field or the Distance-to-Wall field means reading a vector and performing up to 5 scalar products[17].

To show the performance of the TEST model, we started a simulation in a complex geometry with more than 3000 agents. Compared to other models available in *JuPedSim*, we could not see any significant performance difference. (ADD PROFILING TIMES, NOT YET EXISTING) The trajectories improved and look much more natural. This was achieved without the need to manually adding decomposing help-lines or intermediate navigation goals.

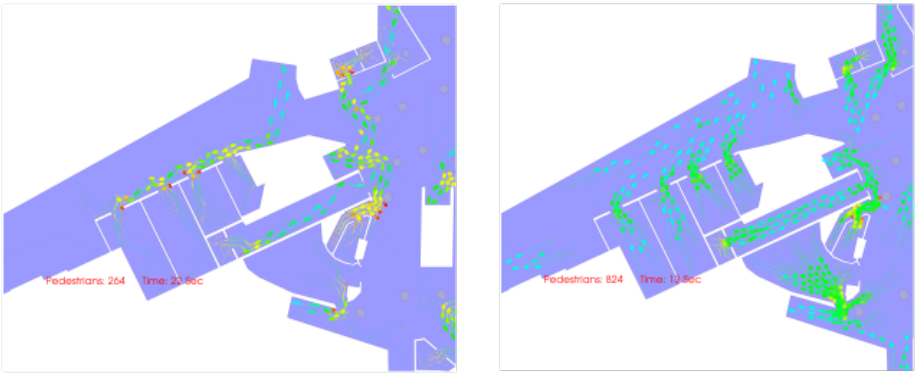


Figure 3.8: Comparison between trajectories: Routing with helplines (left) to routing with navigation field (right).

4 Validation

4.1 Basic Tests

The validation of a model is a complex task and makes up a separate research field. Researchers in the field of Civil Engineering are working on various approaches on how to validate a model. The research group, CST - Pedestrian Dynamics and Traffic Simulation, is developing a set of test-cases any serious model should aim to pass. Tests include the behavior of a single moving agent passing static objects like a dummy agent or an obstacle. (see figure 4.2)

The TEST-model passed these tests and it was shown, that the basic mechanics of the routing are working as specified.

[17] to be verified

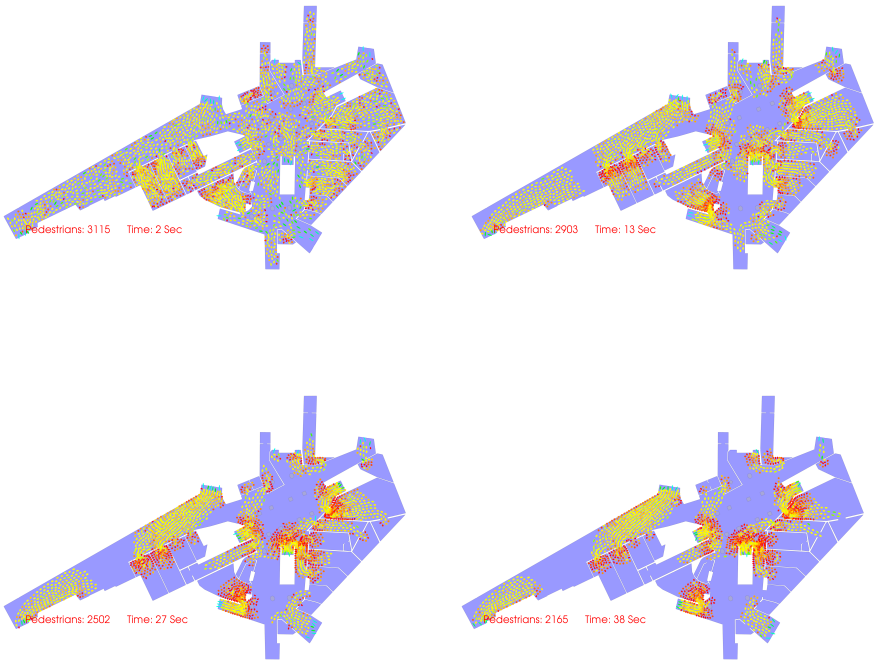


Figure 3.9: Simulation of a complex geometry with multiple exits and app. 3000 agents

4.2 Variation of the Parameter

In this thesis, we take special interest in the following testcase, as it validates the model to yield a fundamental diagram like seen in real world experiments: The tests demands to simulate a bottleneck experiment several times. This is to be repeated for various bottleneck-widths. The calculated flow through the bottleneck shall match the empirical data. We have conducted this test for various *wall-avoid-distance* parameter and want to analyze the change in the flow. It can be seen, that the more the agents will gear to avoid obstacles and walls, the lesser the flow will become. This result is expectable. It was also shown, that without the enhancement of the floor-field, few agents did not pass the bottleneck, but got caught inside of walls.

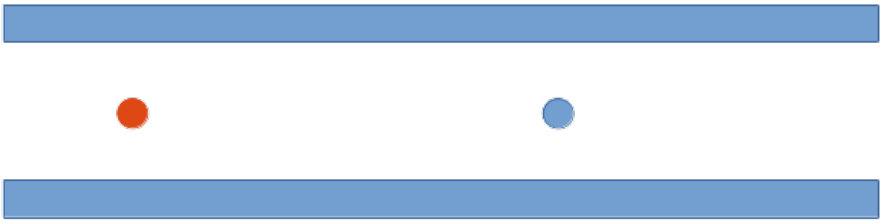


Figure 4.1: Testcase: Orange agent should pass the (static) blue agent.

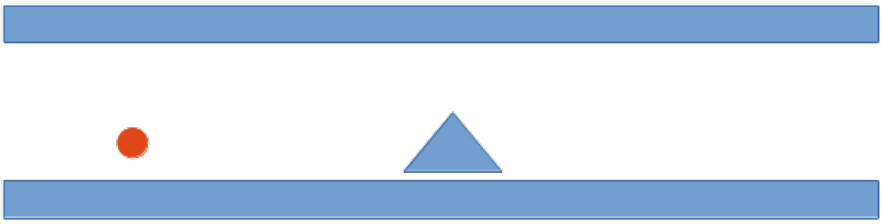


Figure 4.2: Testcase: Orange agent should pass the (static) blue obstacle.

5 Outlook

In this thesis, an altered usage of an enhanced floor-field was shown, integrated in a suitable new TEST-model. The *wall-avoid-distance* parameter was analyzed and shown to be able to show good convergence to empirical data. During the course of the work process, it became clear how versatile and powerful floor-fields can be. Not only in the current state, they can provide valuable assistance to pedestrian models, but they can be further developed to fit into much more contexts. Any of the outlooks provided in this chapter easily catch my interest and I am already looking forward pursuing the development and seeing how they will improve future simulation suits.

5.1 Floor-field

5.1.1 Multiple Goals

The floor-field is a useful tool in routing of pedestrians through any geometry. To unfold its full power, one can imagine to calculate a floor-field for each of many goals. The combination of many floor-fields, each corresponding to a goal, can easily be managed by selecting the direction vector of the one floor-field, that provides the minimal time-cost of an active set of floor-fields evaluated at the grid-point of the current agent's position. Dynamic in-world events in

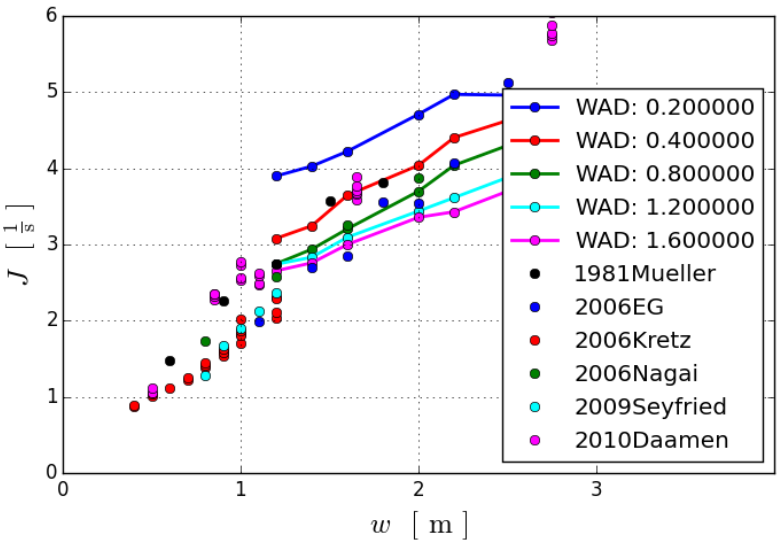


Figure 4.3: Fundamental diagram with empiric data and simulation results

a simulation could alter the set of active floor-fields. This way, models can implement navigation in a dynamically changing world. It can be also used, to be a tool in a simulation suit, which has agents change their destination during runtime. This would be realized by simply changing the active set to the floor-fields corresponding to the new destination.

« Verteilerebene(?) oder etwas kleinere Geometrie mit mehreren Ausgängen: Nebeneinander 2-4 Vectorfelder nebeneinander »

5.1.2 Multiple Floors

In the current state, the floor-field provides time-cost on a discrete grid, a rectangular grid with equidistant spacing in each dimension. The grid-points are stored in a one-dimensional array by the row-major order. In a arrangement like this, it is easy to formulate 4-neighboring[18] relations. These values are easily accessed, if you are provided the stride value, namely how much grid-points make up the length of both dimensions in a 2-D world. The Fast-Marching algorithm needs the time-cost values of the 4-neighborhood. This will change, if you need to simulate in a building with multiple floors, which are connected via

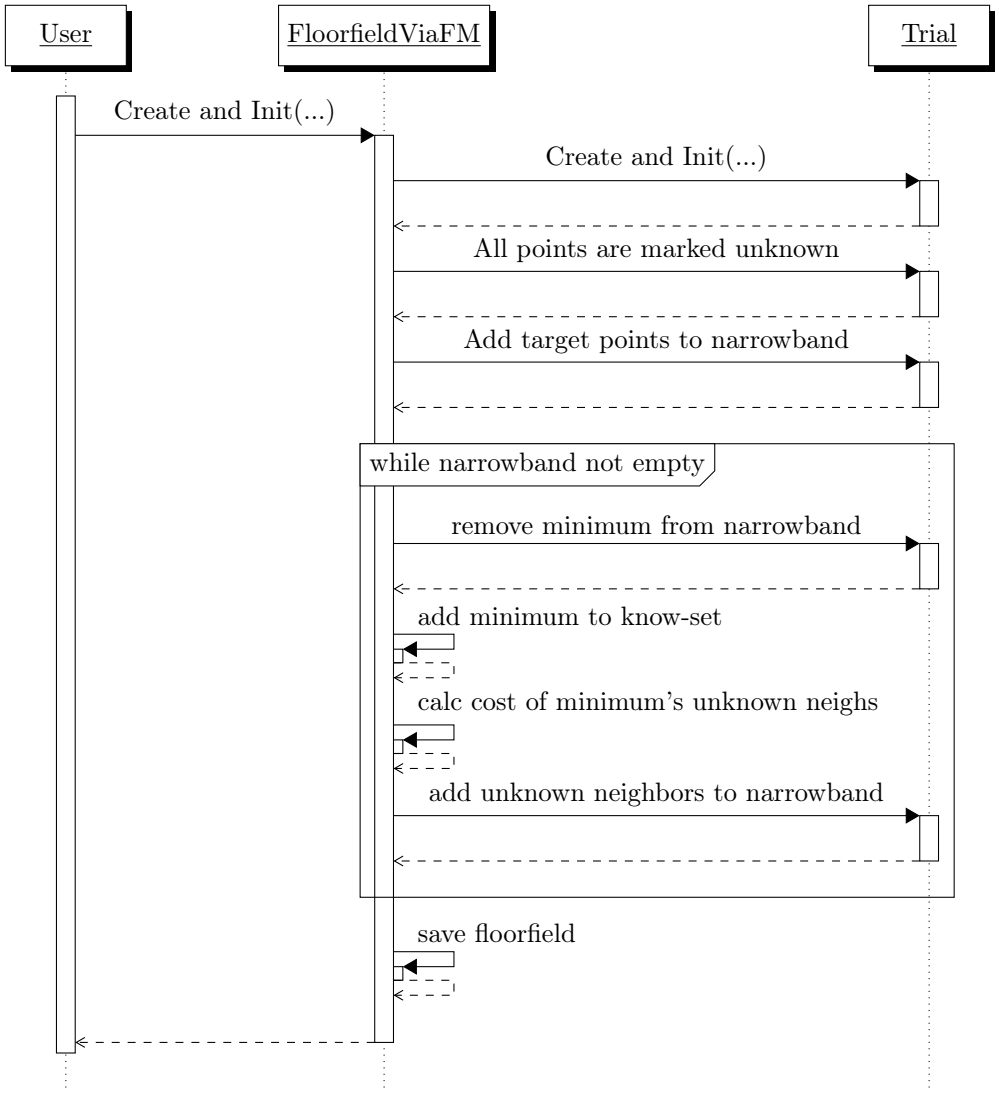
[18] Grid-points north, south, east and west to the current are called 4-neighbors.

stairs. We need to introduce a third dimension, which can be treated equally handy inside each floor. Any position, projected onto the x-y-plane, may not be unique anymore. On the other hand, it would be a waste of memory, if a third dimension would be introduced[19] introduced, to represent the hull-cube circumscribing a building. One is to find a solution, which describes the geometry of the rooms in a memory-efficient way and yet be able to comfortably access the 4-neighbors' time-cost value.

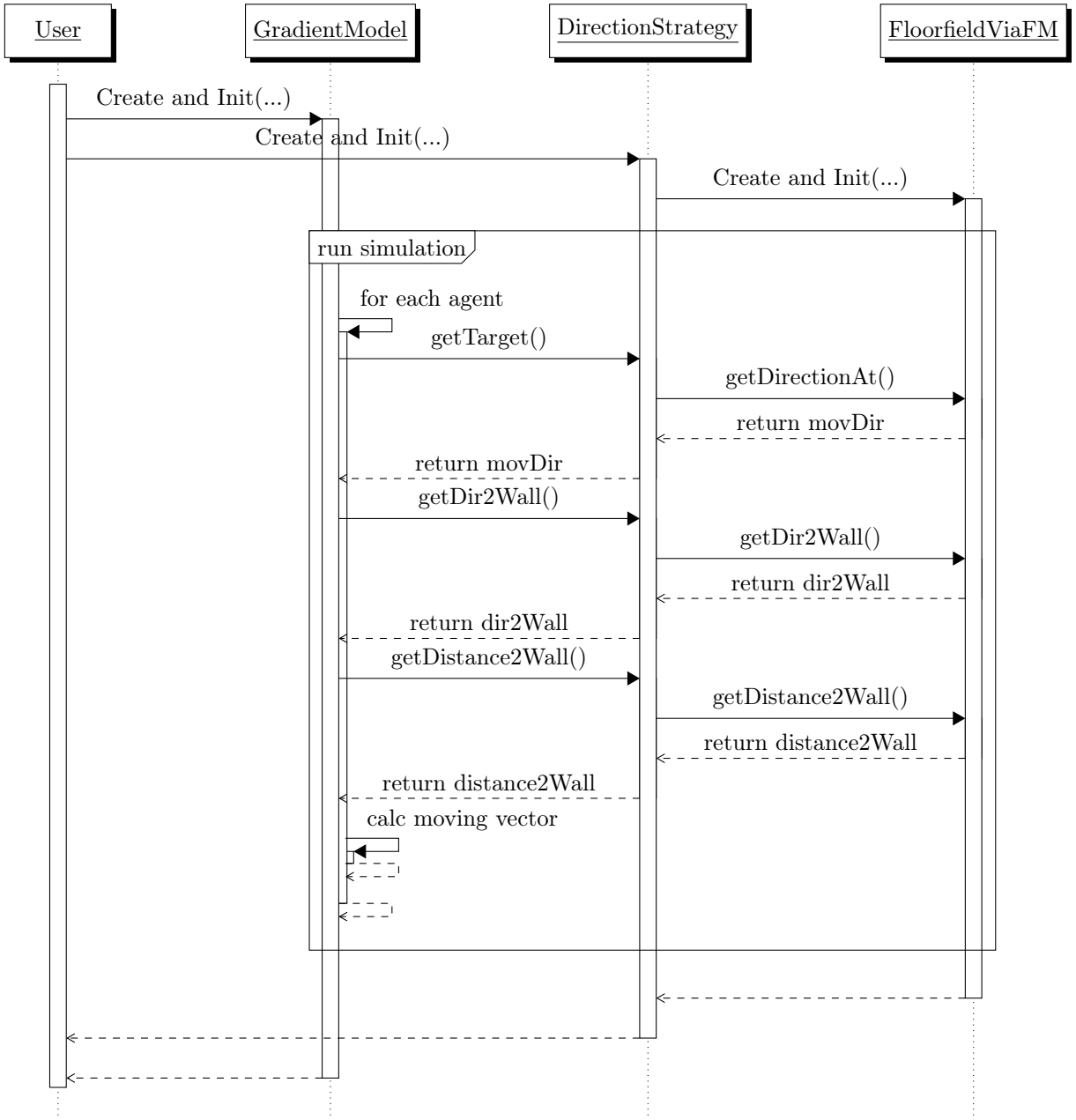
[19] As we are interested in points representing the floor of a room, all the volume (air) above would be not used.

6 Appendices

6.1 Fast-Marching Algorithm



6.2 Gradient Model using a Floorfield



6.3 Code Snippets

Listing 1: FloorfieldViaFM Class Header

```
/**
 * \file        FloorfieldViaFM.h
 * \date        Mar 05, 2015
 * \version     N/A (v0.6)
 * \copyright   <2009-2014> Forschungszentrum JÃŒlich GmbH. All
     rights reserved.
 *
 * \section License
 * This file is part of JuPedSim.
 *
 * JuPedSim is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as
     published by
 * the Free Software Foundation, either version 3 of the License, or
 * any later version.
 *
 * JuPedSim is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
     License
 * along with JuPedSim. If not, see <http://www.gnu.org/licenses/>.
 *
 * \section Description
 * Implementation of classes for ...
 *
 *
 **/

 //remark:
 //refac the code to use enums instead of integer values where
     integer values code sth
 //was considered, but enum classes do not implicitly cast to int
 //rather use makros/masks like in plain C? or just makros (defines)
     ?
 //this would make it easier to read

#ifndef FLOORFIELDVIAFM_H
#define FLOORFIELDVIAFM_H

#include <vector>
#include <cmath>
#include "mesh/RectGrid.h"
#include "../geometry/Wall.h"
#include "../geometry/Point.h"
#include "../geometry/Building.h"
#include "../geometry/SubRoom.h" //check: should Room.h include
    SubRoom.h??
#include "../routing/mesh/Trial.h"

//maybe put following in macros.h
#define LOWSPEED 0.001

class FloorfieldViaFM
```

```
{
    public:
        FloorfieldViaFM();
        FloorfieldViaFM(const std::string&);
        FloorfieldViaFM(const Building* const buildingArg, const
            double hxArg, const double hyArg, const double
            wallAvoidDistance, const bool useDistancefield, const
            std::string&);
        virtual ~FloorfieldViaFM();
        FloorfieldViaFM(const FloorfieldViaFM& other);
        //FloorfieldViaFM& operator=(const FloorfieldViaFM& other);

        void getDirectionAt(const Point& position, Point& direction)
            ;
        void getDir2WallAt(const Point& position, Point& direction);
        double getDistance2WallAt(const Point& position);

        void parseBuilding(const Building* const buildingArg, const
            double stepSizeX, const double stepSizeY);
        void resetGoalAndCosts(const Goal* const goalArg);
        void resetGoalAndCosts(std::vector<Wall>& wallArg, int
            numOfExits);
        void lineScan(std::vector<Wall>& wallArg, double* const
            target, const double outside, const double inside);
        void drawLinesOnGrid(std::vector<Wall>& wallArg, double*
            const target, const double outside);
        void calculateFloorfield(bool useDistance2Wall);     //make
            private
        void calculateDistanceField(const double thresholdArg); //
            make private

        void update(const long int key, double* target, double*
            speedlocal);
        void checkNeighborsAndAddToNarrowband(Trial* &smallest,
            Trial* &biggest, const long int key, std::function<void
            (const long int)> checkNeighborsAndCalc);

        void checkNeighborsAndCalcDist2Wall(const long int key);
        void checkNeighborsAndCalcFloorfield(const long int key);
        //void (*checkNeighborsAndCalc)(const long int key);

        inline double onesidedCalc(double xy, double hDivF);
        inline double twosidedCalc(double x, double y, double hDivF)
            ;

        void testoutput(const char*, const char*, const double*);
        void writeFF(const std::string&);

#ifdef TESTING
        void setGrid(RectGrid* gridArg) {grid = gridArg;}
        Trial* getTrial() {return trialfield;}
#endif //TESTING

    protected:
    private:
        RectGrid* grid;
        std::vector<Wall> wall;
```

```
        int numOfExits;


        //stuff to handle wrapper grid (unused, cause RectGrid
            handles offset)
        double offsetX;
        double offsetY;

        //GridPoint Data in independant arrays (shared primary key)
        int* flag;                  //flag:( 0 = unknown, 1 = singel
            , 2 = double, 3 = final, 4 = added to trial but not
            calculated, -7 = outside)
        double* dist2Wall;
        double* speedInitial;
        double* cost;
        long int* secKey;  //secondary key to address ... not used
            yet
        Point* neggrad; //gradients
        Point* dirToWall;
        Trial* trialfield;

        double threshold;
};

#endif // FLOORFIELDVIAFM_H
```

Listing 2: RectGrid Class Header

```
/**
 * \file        RectGrid.h
 * \date        Mar 05, 2014
 * \version     v0.5
 * \copyright   <2009-2014> Forschungszentrum JÃŒlich GmbH. All
     rights reserved.
 *
 * \section License
 * This file is part of JuPedSim.
 *
 * JuPedSim is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as
     published by
 * the Free Software Foundation, either version 3 of the License, or
 * any later version.
 *
 * JuPedSim is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
     License
 * along with JuPedSim. If not, see <http://www.gnu.org/licenses/>.
 *
 * \section Description
 * Header of class for rectangular grids.
 *
```

```
 *
 **/

#ifndef RECTGRID_H
#define RECTGRID_H

#include "../../geometry/Point.h"

// geometric interpretation of index in "directNeighbor"
//          x (1)               ^ y
//          |                   |
// (2) x----0----x (0)          o---> x
//          |
//          x (3)

typedef struct directNeighbor_t {
    long int key[4];
} directNeighbor;


class RectGrid
{
    public:
        RectGrid() {
            this->isInitialized = false;
        }

        virtual ~RectGrid() {
        }

        RectGrid(const RectGrid& other) {
            nPoints = other.nPoints;
            xMin = other.xMin;
            yMin = other.yMin;
            xMax = other.xMax;
            yMax = other.yMax;
            hx = other.hx;
            hy = other.hy;
            iMax = other.iMax;
            jMax = other.jMax;
            isInitialized = other.isInitialized;
        }

        RectGrid(   long int nPointsArg,
                    double xMinArg,
                    double yMinArg,
                    double xMaxArg,
                    double yMaxArg,
                    double hxArg,
                    double hyArg,
                    long int iMaxArg, // indices must be smaller
                        than iMax
                    long int jMaxArg, // indices must be smaller
                        than jMax
                    bool isInitializedArg) {
            xMin = xMinArg;
            yMin = yMinArg;
```

```
    xMax = xMaxArg;
    yMax = yMaxArg;
    hx   = hxArg;
    hy   = hyArg;
    iMax = iMaxArg;
    jMax = jMaxArg;
    isInitialized = isInitializedArg;
}

long int GetnPoints() const { return nPoints; }
//void SetnPoints(long int val) { nPoints = val; }
double GetxMin() const { return xMin; }
void SetxMin(double val) { if (!isInitialized) xMin = val; }
double GetyMin() const { return yMin; }
void SetyMin(double val) { if (!isInitialized) yMin = val; }
double GetxMax() const { return xMax; }
void SetxMax(double val) { if (!isInitialized) xMax = val; }
double GetyMax() const { return yMax; }
void SetyMax(double val) { if (!isInitialized) yMax = val; }
long int GetiMax() const { return iMax; }
//void SetiMax(long int val) { iMax = val; }
long int GetjMax() const { return jMax; }
//void SetjMax(long int val) { jMax = val; }
double Gethx() const { return hx; }
//void Sethx(double val) { hx = val; }
double Gethy() const { return hy; }
//void Sethy(double val) { hy = val; }

double get_x_fromKey (long int key) const { return (key%iMax
    )*hx+xMin; }
double get_y_fromKey (long int key) const { return (key/iMax
    )*hy+yMin; }
double get_i_fromKey (long int key) const { return (key%iMax
    ); }
double get_j_fromKey (long int key) const { return (key/iMax
    ); }

long int getKeyAtXY(const double x, const double y) const
    {//key = index in (extern managed) array
    //Point nearest = getNearestGridPoint(Point(x,y));
    long int i = (long int)(((x-xMin)/hx)+.5);
    long int j = (long int)(((y-yMin)/hy)+.5);
    if ((i < iMax) && (j < jMax))
        return (j*iMax+i); // 0-based; index of (closest
            gridpoint)
    return -1; // invalid indices
}

long int getKeyAtPoint(const Point p) const {
    long int i = (long int) (((p.GetX()-xMin)/hx)+.5);
    long int j = (long int) (((p.GetY()-yMin)/hy)+.5);
    if ((i < iMax) && (j < jMax))
        return (j*iMax+i); // 0-based; index of (closest
            gridpoint)
    return -1; // invalid indices
}
```

```
void setBoundaries(const double xMinA, const double yMinA,
                   const double xMaxA, const double yMaxA) {
    if (!isInitialized) {
        xMin = xMinA;
        xMax = xMaxA;
        yMin = yMinA;
        yMax = yMaxA;
    }
}

void setBoundaries(const Point xy_min, const Point xy_max) {
    if (!isInitialized) {
        xMin = xy_min.GetX();
        xMax = xy_max.GetX();
        yMin = xy_min.GetY();
        yMax = xy_max.GetY();
    }
}

void setSpacing(const double h_x, const double h_y) {
    if (!isInitialized) {
        hy = h_y;
        hx = h_x;
    }
}

void createGrid(){ // @todo ar.graf : what if cast chops off
     float, if any changes: get_x_fromKey still correct?
    if (!isInitialized) {
        iMax = (long int)((xMax-xMin)/hx) + 2;  //check plus
             2 (one for ceil, one for starting point)
        jMax = (long int)((yMax-yMin)/hy) + 2;
        nPoints = iMax * jMax;
        //@todo: see if necessary to align xMax/yMax
        xMax = xMin + iMax*hx;
        yMax = yMin + jMax*hy;
        isInitialized = true;
    }
}

Point getNearestGridPoint(const Point& currPoint) const {
    if ((currPoint.GetX() > xMax) || (currPoint.GetY() >
        yMax))
        return Point(-7, -7); // @todo: ar.graf : find good
            false indicator
    long int i = (long int)(((currPoint.GetX()-xMin)/hx)+.5)
        ;
    long int j = (long int)(((currPoint.GetY()-yMin)/hy)+.5)
        ;
    return Point(i*hx+xMin, j*hy+yMin);
}

Point getPointFromKey(const long int key) const {
    long int i = key%iMax;
    long int j = key/iMax; //integer division

    return Point(i*hx+xMin, j*hy+yMin);
```

```
        }

        directNeighbor getNeighbors(const long int key) const {
            directNeighbor neighbors = {{-1, -1, -1, -1}}; //
                curleybrackets for struct, then for int[4]
            long int i = get_i_fromKey(key);
            long int j = get_j_fromKey(key);

            //right                         //-2 marks invalid
                neighbor
            neighbors.key[0] = (i == (iMax-1)) ? -2 : (j*iMax+i+1);
            //upper
            neighbors.key[1] = (j == (jMax-1)) ? -2 : ((j+1)*iMax+i)
                ;
            //left
            neighbors.key[2] = (i == 0) ? -2 : (j*iMax+i-1);
            //lower
            neighbors.key[3] = (j == 0) ? -2 : ((j-1)*iMax+i);

            return neighbors;
        }

    protected:
    private:
        long int nPoints;
        double xMin;
        double yMin;
        double xMax;
        double yMax;
        double hx;
        double hy;
        long int iMax; // indices must be smaller than iMax
        long int jMax; // indices must be smaller than jMax
        bool isInitialized;
};

#endif // RECTGRID_H
```

7 Bibliography

Mohcine's diss

Ulrich's diss

Dietrich's arbeit (introduction)

micro-/macroscopic arbeiten (desktop/READ)

fast-marching arbeiten: madrid code

outlook: triangulated version (master/pdfs/fmarcher)

Helbing and Molnár 1995, Hartmann 2010, Hartmann et al. 2012, (+ Felix 4.2)

history: helbing, ... (einleitungen von mc, uk, fd)