Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Project Report

Modelling Situations of Evacuation in a Multi-level Building

Hans Hardmeier, Andrin Jenal, Beat Küng & Felix Thaler

Zurich
April 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.

Hans Hardmeier	Andrin Jenal
Beat Küng	Felix Thaler

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of Originality

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

I hereby declare that the written work I have submitted entitled

Modelling Situations of Evacuation in a Multi-level Building

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

Author(s)

Last name	First name
Hardmeier	Hans
Jenal	Andrin
Küng	Beat
Thaler	Felix

Supervising lecturer

Last name	First name
Balietti	Stefano
Donnay	Karsten

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.

Zürich, 21.Mai 2012
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

If you are an ETH-Student, you know that at lunch time it is almost impossible to go out of the building through the main entrance to the polyterasse, because of the number of students trying to leave at the same time. What would happen, if in addition to that, an evacuation was involved? Have you ever imagined, how an evacuation at the ETH Main building, ETH CAB-Building or even at your own home would look like? How many people would be able to leave simultaneously? What is the best strategy for people to leave? How would the perfect evacuation plan for your school or enterprise building look like?

In this work, we decided to elaborate a program that is flexible enough to calculate the fastest exit for every given building structure. Using a "2D Range Tree Datastructure" to solve this complex problem and to improve the access to the forces over each agent, the program is capable of simulating the evacuation efficiently.

Different from all other similar projects, we also focused on an efficient and realistic implementation of the program. For the preprocessing, the "*Fast Sweeping* Algorithm Method" enabled us to have only some seconds of initialization time (instead of some minutes with "*Fast Marching* Algorithm" (See 5). Using our own "2D Range Tree" data structure to handle the numerous amount of agents enables us to simulate large buildings with many agents and with a realistic force model.

2 Individual contributions

We all worked together in this project and used the individual strengths of each of us to achieve the best possible result. Detailed information about the individual contribution can be obtained from the git history.

Carl Hans Peter Hardmeier Samame, alias Hans Hardmeier, was responsible for some parts of the documentation, verification of the *MATLAB*-code and calculating efficiency using different operating systems (i.e. Mac OS 10.6).

Contributing to some core functionalities of the social force model, especially repulsive effects, Andrin Jenal focused on the modeling of the buildings layout used in the simulation. Additionally, he elaborated some sections of the documentation.

Beat Küng did most of the plotting functionality and the definition & implementation of the configuration and building image file. He also did several sections of the documentation.

Felix Thaler implemented the algorithms written in *C*, using *MATLAB*'s *MEX*-interface. These are efficient *Fast Sweeping*, a 2D *Range Tree* and fast bilinear interpolation. Further he developed the basic simulation framework and added the social force model as well as some custom algorithms, for instance one to minimize wall penetration of agents. Some sections of the documentation are written by him, too.

3 Introduction and Motivations

3.1 Introduction

Simulating the evacuation scenario of a single-level building is well known but is not general enough. Though we want to introduce a more sophisticated simulation within a multi-level building. E.g.: What would happen, if a multi-level building has to be evacuated? Which escape routes would be mostly used? Which effects would the pressure of other persons have to the situation? Since tower buildings are getting more common in large cities, engineers have to care more about the behavior in situations of emergency, namely evacuations. Apart of the mathematical model and implementation for solving this problem, we also wanted to increase the utility of this program for all readers by giving the possibility of calculate different scenarios in different buildings given by the user. In this work, we will mostly work with the map of the ETH building, however, there are more configurations in the data folder and one can replace any map with others that meet the properties of the Figure 1 in section 5.1.

3.2 Motivation

Intuitive expectations and mathematical model results can stay sometimes in contradiction. Our intuition is full of little concepts that are hard to realize in a conscient way. Using the language of mathematics, we want to describe formaly all the elements that contribute to the complex result of the human behavior. However, a common point of motivation within the group is the connection between the 'real' World and the mathematical formalism.

On the other side, we, as ETH-Students, are not sure if the evacuation potential (the property of a buiding of being evacuated efficiently) of our ETH-buildings are enough for the amount of students during a normal week day. We also want to help the people around the world that have a similar question to find an answer. This two points gave us the ideas and motivation to create a flexible tool that simulates a real-world scenario for any given building or even structure (i.e. planes or boats).

3.3 Fundamental Questions

In the first place, we all are wondering *how exaclty an implementation of a social force model looks like*, based on already published papers coping with this problem

[2] [6]. Of special interest is also *how this model can be implemented efficiently for a multilevel building and how it can be extended to add some dynamic features.* Moreover we are keen to understand *how the simulation behaves applying the model to a real building like the ETH main building.* Generally speaking, *how realistic is the behavior of the agents used in our model?.*

4 Description of the Model

4.1 General Model

For our model, we will create a relatively general framework for behavioral simulation in evacuation scenarios based on the social force model introduced by Dirk Helbing [6]. A core investigation is a simulation, beginning in a general everyday situation, where people are spread randomly in their office or somewhere in the hallway and ending in an evacuation scenario. At the end we should be able to make strong statements anwsering our fundamental questions. Based on the forces explained in the following paragraphs, we tried to figure out how agents overcome major obstacles like tiny passages, stairs or pillars.

4.2 Forces

We implemented the model's forces as described in [6], a brief overview is given here.

The reactions of the agents will mainly be influenced by different forces, modelparameters and the building structure itself. Describing this problem in mathematical terms, leads to an equation, where the mass, the desired direction f_D, the repulsive forces f_{ij} and the wall force f_{iW} of each individual agent contribute to the change of velocity in time.

Helbing stated this problem as the following:"We assume a mixture of socio-psychological and physical forces influencing the behavior in a crowd: each of N pedestrians i of mass m_i likes to move with a certain desired speed v_i^0 in a certain direction $\mathbf{e}_i^0$, and therefore tends to correspondingly adapt his or her actual velocity $\mathbf{v}_i$ with a certain characteristic time τ_i. Simulataneously, he or she tries to keep a velocity-dependent distance from other pedestrians j and walls W."

$$m_i \frac{\mathsf{d}\mathbf{v}_i}{\mathsf{d}t} = m_i f_D + \sum_{j(\neq i)} f_{ij} + \sum_W f_{iW} \tag{1}$$

That means the change of position $\mathbf{r}$ is given by the velocity: $\mathbf{v}_i = \frac{\mathsf{d}\mathbf{r}_i}{\mathsf{d}t}$
To understand the different parties, they will be explained briefly.

4.2.1 Desired Direction Force

An agent always has a desired direction $e(t)$ in which he wants to walk currently with a desired speed of $v(t)$. The reaction time τ affect the speed of the directional change.

$$f_D = \frac{v_i^0(t)\mathbf{e}_i^0(t) - \mathbf{v}_i(t)}{\tau_i} \tag{2}$$

4.2.2 Repulsive Interaction Force

As people don't like to get too close to each other, the main component of this formula is a rather psychological aspect and plays an important role to keep agents apart from each other. However if it should happen that agents get too close, two additional forces, the 'body force' counteracting body compression and 'sliding friction force' contribute to the repulsive force to ensure a certain distance. [6]

$$f_{ij} = \{A_i exp[(r_{ij} - d_{ij})/B_i] + kg(r_{ij} - d_{ij})\}\, \mathbf{n}_{ij} + \kappa g(r_{ij} - d_{ij}) \triangle v_{ji}^t \mathbf{t}_{ij} \tag{3}$$

According to Helbing the repulsive interaction force $A_i exp[(r_{ij} - d_{ij})/B_i]\mathbf{n}_{ij}$ describes the psychological tendency of two pedestrians i and j to stay away from each other, where A_i and B_i are constants. "$d_{ij} = \|r_i - r_j\|$ denotes the distance between the pedestrians' centres of mass, and $\mathbf{n}_{ij} = (n_{ij}^1, n_{ij}^2) = \mathbf{r_i} - \mathbf{r_j}/d_{ij}$ is the normalized vector pointing from pedestrian j to i. The pedestrians touch each other if their distance d_{ij} is smaller than the sum $r_{ij} = (r_i + r_j)$ of their radii r_i and r_j."[6] If this is the case two addtional forces are considered: $k(r_{ij} - d_{ij})\mathbf{n}_{ij}$ and $\kappa(r_{ij} - d_{ij})\triangle v_{ij}^t \mathbf{t}_{ij}$. "Here $\mathbf{t}_{ij} = (-n_{ij}^2, n_{ij}^1)$ means the tangential direction and $\triangle v_{ij}^t = (\mathbf{v}_j - \mathbf{v}_i) \cdot \mathbf{t}_{ij}$ and the function $g(x)$ is zero if the pedestrians do not touch each other ($d_{ij} > r_{ij}$), and otherwise equal to the argument x."[6]

4.2.3 Wall Force

Similar to the repulsive interaction force people don't want to get too close to walls neither. To prevent this, the wall force is introduced. Helbing states it the following:"d_{iW} means the distance to wall W, $\mathbf{n}_{iW}$ denotes the direction perpendicular to it, and $\mathbf{n}_{iW}$ the direction tangential to it, the corresponding interaction force with the wall is given by:" [2]

$$f_{iW} = \{A_i exp[(r_i - d_{iW})/B_i] + kg(r_i - d_{iW})\}\, \mathbf{n}_{iW} - \kappa g(r_i - d_{iW})(\mathbf{v}_i \cdot \mathbf{t}_{iW})\mathbf{t}_{iW} \tag{4}$$

5 Implementation

Our goal was a fast implementation of the model. So we decided to use the *Fast Sweeping* algorithm instead of the *Fast Marching* alorithm to calculate the fastest way out of a building. We knew that *MATLAB* code is not very fast and *MATLAB* provides an interface for other programming languages. So we used the *C* programming language to implement the *Fast Sweeping* method.

Later in the development process we discovered another bottleneck in our code: The social force between the agents which runs in $O(n^2)$. Felix Thaler then introduced a *Range Tree* for this problem, which he also implemented in *C*.

5.1 *MATLAB* Code

We wanted to create a flexibel model which can be used to simulate many possible building structures. Each scenario is described in a configuration file. This includes for example how many agents are placed on each floor or the timestep of the simulation (the exact definition can be found in the file `data/config_file_structure`). Each config file also references one or more building floor images. Figure 1 describes how a building floor image must look like. The agents are placed randomly within the agents spawning area(s).

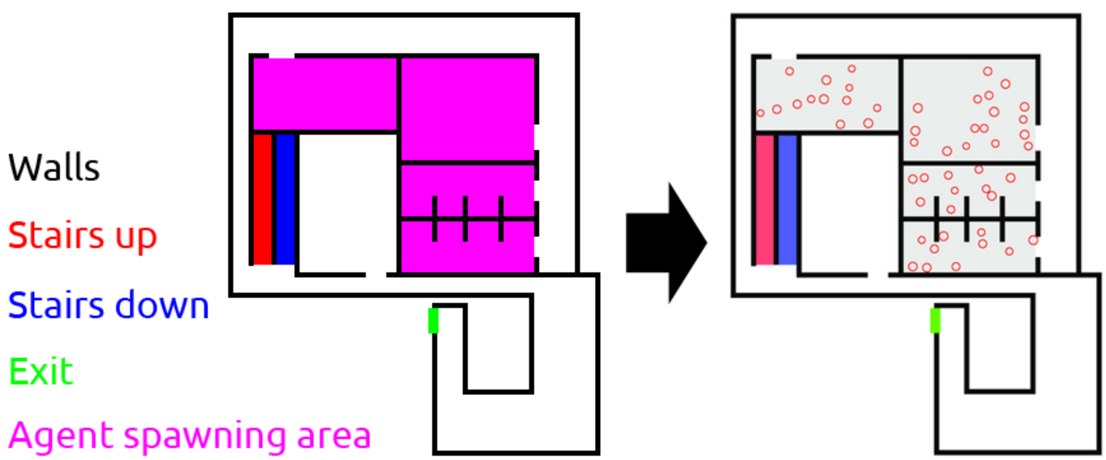

Figure 1: Left the building floor image and right how it looks when the simulation is running

Since *MATLAB* is not really object-oriented, we used a big data structure (called `data`) that includes all internal data that we use (eg. the floors and agents). It is passed as an argument for every function that needs it.

5.2 Pathfinding

As described in [2], the agents always try to reach their desired destination using the shortest possible path. As we use raster graphics to encode simulation data, we decided to use the same discrete grid to compute the nearest path to an exit for every point approximately. Mathematically, this can be expressed as an partial differential equation, the 2D Eikonal equation (equation 5).

$$\|\nabla d(\mathbf{x})\| = 1 \quad d : \mathbb{R}^2 \to \mathbb{R}, \mathbf{x} \in \mathbb{R}^2 \tag{5}$$

The solution $d(\mathbf{x})$ now gives the distance to the nearest exit point, its negative gradient $-\nabla d(\mathbf{x})$ therefor always points in the direction of the shortest path towards the desired exit. There are mainly two competitive agorithms for solving this equation efficiently. First there is the *Fast Marching* method, a specialized version of Dijkstra's well known algorithm [8]. The alternative is *Fast Sweeping*, which leads to a much simpler implementation, faster calculation and better algorithmic complexity of $O(n)$ instead of $O(n \log n)$ with respect to the discretization size. We therefor implemented an efficient *Fast Sweeping* method, closly following [1] as a basis of our pathfinding and repulsive wall forces. The algorithm uses an upwind finite difference scheme where our mesh is defined by the input images. The implementation was done *C* and not directly in *MATLAB*, using optimized boundary condition handling for our pruposes. Our benchmarks showed a high speed increase compared to other Eikonal solvers, e.g. the "Accurate Fast Marching" implementation as found at *MATLAB CENTRAL* [9].

5.3 Profiling & Optimization

Using the *Profiler*-function of *MATLAB*, we were able to discuss the efficiency of our implementation. The image 2 on the next page shows the time consumption for the calculation of the evacuation of 2 floors (ETH CAB-Building Floors E and F) within 4087 seconds.

The first version of our simulation showed a different profiling behavior: The functions `interp2` and `addAgentRepulsiveForce` were at the top and thus needed most of the calculation time. Now, after our improvements, they are further down in the profiling ranking, meaning they are more efficient. These improvements decreased the time for simulation significantly.

5.3.1 Parallelization

We tried to optimize the code using the parallel for-loop `parfor` in *MATLAB*. In the function `applyForcesAndMove` we replaced the for-loop over the agents with

Profile Summary

Generated 16-May-2012 19:44:08 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
isprop	5097390	1085.031 s	1085.031 s	
hggetbehavior	5091386	2203.116 s	594.429 s	
hggetbehavior>localPeek	5091386	1608.687 s	525.234 s	
graphics/private/fireprintbehavior	3002	2503.876 s	290.834 s	
addAgentRepulsiveForce	1501	265.991 s	245.443 s	
graphics/private/prepare	1501	1776.422 s	235.135 s	
graphics/private/restorehg	1501	1429.437 s	165.774 s	
graphics/private/prepareui	1501	168.010 s	160.320 s	
hardcopy	1501	118.852 s	118.676 s	
simulate	1	4087.238 s	112.301 s	
findobjhelper	103573	110.231 s	110.231 s	
plotAgentsPerFloor	3002	79.856 s	63.389 s	
addWallForce	1501	58.936 s	51.482 s	
addDesiredForce	1501	57.739 s	49.697 s	
applyForcesAndMove	1501	35.014 s	32.543 s	
repmat	316711	31.042 s	31.042 s	
plotExitedAgents	1501	30.846 s	29.090 s	
lerp2 (MEX-file)	1801452	17.968 s	17.968 s	
savtoner	3002	63.683 s	17.458 s	
...(a)repmat(a.p,length(ang),1)+a.r*rmul	300200	43.596 s	14.016 s	
allchild	12008	10.848 s	10.848 s	

Figure 2: Profile of the calculation for the CAB-Building with 300 agents over 3 different floors

`parfor`. We measured the time using an input with 200 agents on a 4 core machine using 5 *MATLAB* workers. The result was that the parfor version was even slightly slower than the serial version. The reason for this is how *MATLAB* implements `parfor`: all the memory that is used by a worker must be sent to this worker. Each agent must access the building floor image randomly and this creates a large amount of memory that must be transferred to each worker, which leads to a decreasing performance. This is why we decided not to use `parfor` or any other parallelization in our implementation.

5.3.2 Replacing *MATLAB* standard functions

Through our first messures, we realised that the function `interp2` consumed a lot of resources and time. We decided to create our own operator called `lerp2` (written in *C*) that interpolates between data points. It finds values of a two-dimensional function interpolating the data at intermediate points bilinearly. Here the different given data points are the current values of the frame $i-1$ for the frame i.

5.3.3 Range Tree

The most time consuming part in our program is the calculation of the interaction forces between the agents, at least if their count is high enough. To reduce the natural complexity of order $O(n^2)$ of this inter-agent interaction, we can clamp forces with little effect, e.g. in our implementation all forces smaller than 10^{-4}N. As they are exponentially decreasing, the distance in which the forces need to be adressed is only several meters, so a big part of them can just be ignored without introducing a significant error.

To get all agents influenced by another agent by a force bigger than a threshold, we need to be able to search neighbours of any agent within a given distance. To efficiently query these agents, we implemented a 2D *Range Tree* in *C*, which allows query times of order $O(\log^2 n+k)$, where n is the total number of agents and k is the number of queried agents [10]. In larger simulations this reduces simulation times by a significant factor.

6 Simulation Results and Discussion

6.1 Expected Results

We expect the stairs and the main building exit to be the bottlenecks. The amount of people in lower levels is increasing with time until a certain point, when most of the people have exited the building. Also we think that if the velocity of the people is higher, jams at the exit will increase.

6.2 Simulation Results

We tested our algorithm mainly by simulating an evacuation of ETH's CAB building. The floor plans were downloaded from [11] and converted to our coded bitmap representation. The results of the simulation are, as expected, comparable to those of Helbing as described in [2] and [6]. Our implementation realisticly shows some bottlenecks which get heavily crowded soon, especially the regions before stairs and exits. The speed of the agents decreases heavily due to the crowd and the time they need to reach the exits therefor increases rapidly. As we used real world units in our model, it was relatively simple to get the right scale for the simulation, nonetheless we found that some agents got stuck at the doors, as they block each other. This is of course not a very realistic behavior but due to the simplicity of our agents demand to reach the exit with shortest Euklidean distance. It would be interesting to use a more intelligent and natural search algorithm to determine the best path to escape, but this is out of the scope of this project, see also section 6.3 about other possible improvements.

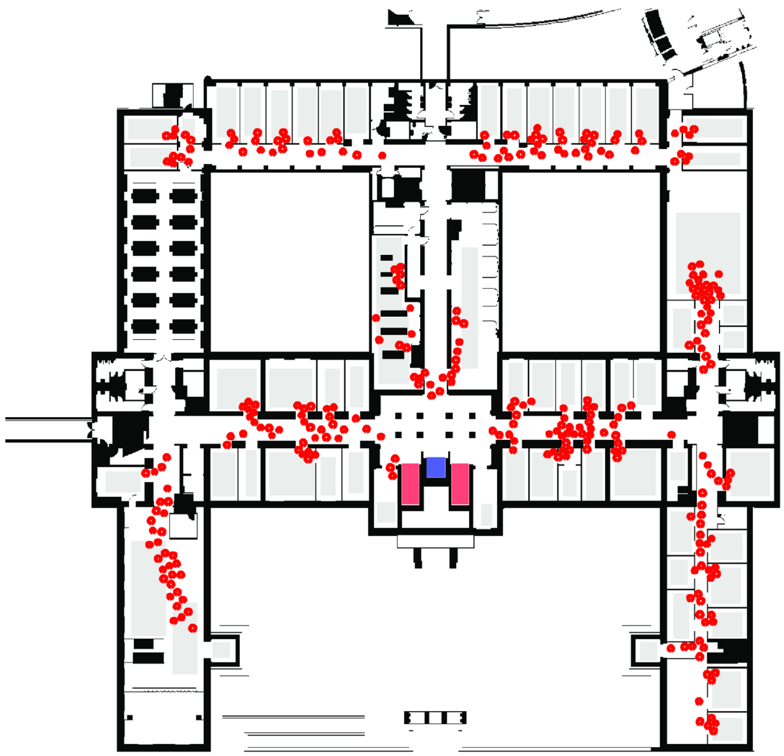

Figure 3: Visualisation of evacuation in the CAB Building E Floor

6.2.1 Comparison to previous projects

As there were already other projects in this course which studied agent based evacuation simulations, we decided to compare our results to those of an older project, namely the project "Evacuation Bottleneck: Simulation and analysis of an evacuation of a high-school building with *MATLAB*" by Alexander Jöhl and Tomáš Nyitray [12].

They use a very simplicistic model, incorporating just two forces: one pointing towards the nearest exit and one inter-agent force. The former is comparable to the one used in our model, but missing some sophisticated interpolation on the discretized grid, while the latter is following a much simpler approach and is just a force centered around the agents with inverse linear fall-off. As their inter-agent forces are so weak, the stability timestep restriction is not that tight and therefor the simulation times are very low. In our implementation using the forces as defined in [6] one the other hand, there is a much stricter timestep restriction, leading to higher simulation times to guarantee stability, but in this case still lower than 5 minutes. But a linear force can't simulate realistic interaction between the agents and they can easily penetrate each other. Therefor it is impossible to simulate crowding and evacuation bottlenecks.

Further, their floorplan is significantly less detailed than our implementation allows and no proper scaling seems to be mentioned or implemented, this made it difficult to get similar parameters, especially because in our model the collision detection works accurately using the agents radii, which means they get stuck at narrow doors.

To recreate the speed of their agents, a time lapse simulation seemed to be most appropriate and indeed, we could get some similar results. Nonetheless our model does not show the weaknesses described above, there are no overlapping agents and they walk along straight lines.

6.3 Improvements

There are several things that could be done to further improve our social force model and to reduce the calculation time used to simulate.

choice of exit Our simulation of the CAB building shows that most of the agents take the side exit on the lowest floor, and only a few exit through the main door, which is right beside the side exit. This is not very realistic and could be improved by adding intelligence to the agents. For example each agent could decide which exit he/she wants to take, and also consider jams at the visible exits.

panic factor We did not implement the panic factor mentioned in [6]. In reality, people in panic behave less organized, they probably don't always run towards the nearest exit.

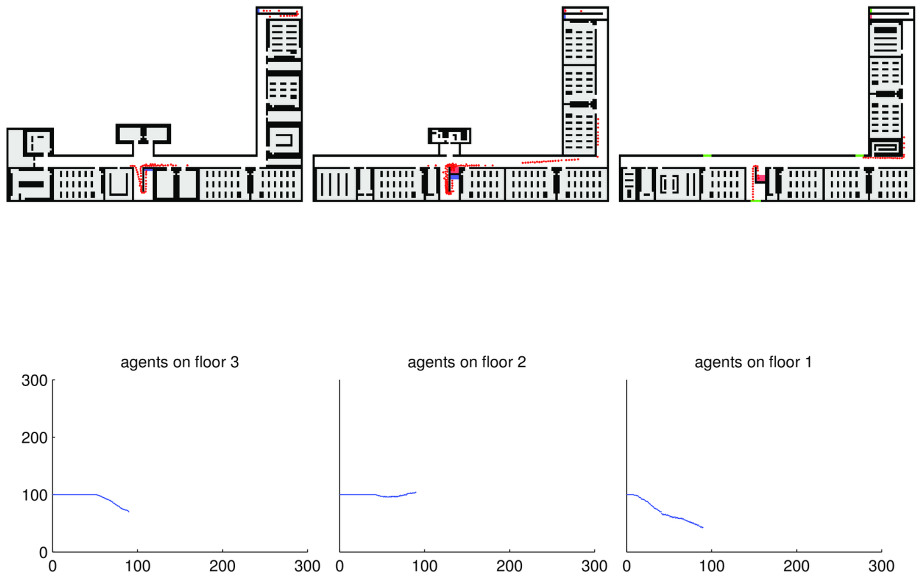

Figure 4: One frame of our simulation of the building given in [12].

timestep If the timestep in the config file is too big, the simulation becomes unstable and behaves like a billiards game. This sometimes also happens when a lot of agents are crowded and push in one direction. Adaptive timestepping or an implicit integration scheme could be introduced to solve this issue.

agent velocity All agents are currently equally fast, which is of course different from reality.

doors In our simulations, the doors of the CAB building were sometimes too narrow for the agents to exit. But this issue can easily be resolved by changing the building map or the agent radii in the config file.

calculation time For no obvious reason, the time used to calculate one simulation-step (one frame) increases linearly over time. We identified the plotting routine as the culprit, but we were not able to figure out whether this is a *MATLAB* problem or our plotting calls are to blame for this. Certainly, a big performace advantage would be to write the whole simulation in *C/C++*.

7 Summary

Summarizing the results of the simulation for the purpose of answering the fundamental questions of this project, we can claim that the implementation of different force models has not been a major problem. Nonetheless the implementation of an *realistic* evacuation model is far more complicated than expected. Details like individual-reactions or non-expected events have been ignored for the sake of simplicity, although they enjoy of a special importance for this kind of social force models.

MATLAB was powerful enough to simulate and it also provides a wide spectrum of useful features and functions. Unfortunately, the efficience of them cannot be compared to their functional equal features in other programming languages like *C or C++*. Nevertheless, *MATLAB* is a good tool for projects, which have not the intention of simulating a social force model in real time.

7.1 Thanks

In this section we would like to thank the MSSSM-Group from the ETH in Zurich, Switzerland directed by Karsten Donnay and Stefano Balietti for their engagement in the lecture "Modeling and Simulating Social Systems with *MATLAB*" during the Spring Semester 2012.

8 References

References

[1] Zhao, Hongkai (2004): A Fast Sweeping Method for Eikonal Equations.

[2] Helbing, Dirk; Molnar, Peter (1995): Social Force Model for Pedestrians Dynamics.

[3] Helbing, Dirk; Johnson, Anders (2006): Analytical Approach to Continuous and Intermittent Bottleneck Flows.

[4] Schadschneider, Andreas et al. (2002): CA Approach to Collective Phenomena in Pedestrian Dynamics.

[5] Helbing, Dirk; Johansson, Anders (2007): Dynamics of crowd disasters: An empirical Study.

[6] Helbing, Dirk et al. (2000): Simulating dynamical features of escape panic.

[7] Helbing, Dirk et al. (2005): Self-Organized Pedestrian Crowd Dynamics: Experiments, Simulations, and Design Solutions.

[8] Dijkstra, Edsger W. (1959): A Note on Two Problems in Connexion with Graphs.

[9] Kroon, Dirk-Jan (2011): http://www.mathworks.com/matlabcentral/fileexchange/24531-accurate-fast-marching

[10] Ottmann, Thomas; Widmayer, Peter (2002): Algorithmen und Datenstrukturen, 4. Auflage.

[11] ETH Zrich, Version 2011.1 prod (prod red2): http://www.rauminfo.ethz.ch

[12] Jhl, Alexander; Nyitray Tomáš: Evacuation Bottleneck: Simulation and analysis of an evacuation of a high-school building with *MATLAB*, https://github.com/jjoolloo/Swiss-Slovak-misunderstanding

9 Appendix

9.1 Code

9.1.1 *MATLAB* code

```
function data = addAgentRepulsiveForce(data)
%ADDAGENTREPULSIVEFORCE Summary of this function goes here
%   Detailed explanation goes here

% Obstruction effects in case of physical interaction

% get maximum agent distance for which we calculate force
r_max = data.r_influence;
tree = 0;

for fi = 1:data.floor_count
    pos = [arrayfun(@(a) a.p(1), data.floor(fi).agents);
           arrayfun(@(a) a.p(2), data.floor(fi).agents)];

    % update range tree of lower floor
    tree_lower = tree;

    agents_on_floor = length(data.floor(fi).agents);

    % init range tree of current floor
    if agents_on_floor > 0
        tree = createRangeTree(pos);
    end

    for ai = 1:agents_on_floor
        pi = data.floor(fi).agents(ai).p;
        vi = data.floor(fi).agents(ai).v;
        ri = data.floor(fi).agents(ai).r;
```

```
        % use range tree to get the indices of all agents near agent ai
        idx = rangeQuery(tree, pi(1) - r_max, pi(1) + r_max, ...
                            pi(2) - r_max, pi(2) + r_max)';

        % loop over agents near agent ai
        for aj = idx

            % if force has not been calculated yet...
            if aj > ai
                pj = data.floor(fi).agents(aj).p;
                vj = data.floor(fi).agents(aj).v;
                rj = data.floor(fi).agents(aj).r;

                % vector pointing from j to i
                nij = (pi - pj) * data.meter_per_pixel;

                % distance of agents
                d = norm(nij);

                % normalized vector pointing from j to i
                nij = nij / d;
                % tangential direction
                tij = [-nij(2), nij(1)];

                % sum of radii
                rij = (ri + rj);

                % repulsive interaction forces
                if d < rij
                   T1 = data.k*(rij - d);
                   T2 = data.kappa*(rij - d)*dot((vj - vi),tij)*tij;
                else
                   T1 = 0;
                   T2 = 0;
                end

                F =  (data.A * exp((rij - d)/data.B) + T1)*nij + T2;

                data.floor(fi).agents(ai).f = ...
                    data.floor(fi).agents(ai).f + F;
                data.floor(fi).agents(aj).f = ...
                    data.floor(fi).agents(aj).f - F;
            end
        end

        % include agents on stairs!
        if fi > 1
            % use range tree to get the indices of all agents near agent ai
            if ~isempty(data.floor(fi-1).agents)
```

```
                idx = rangeQuery(tree_lower, pi(1) - r_max, ...
                        pi(1) + r_max, pi(2) - r_max, pi(2) + r_max)';

                % if there are any agents...
                if ~isempty(idx)
                    for aj = idx
                        pj = data.floor(fi-1).agents(aj).p;
                        if data.floor(fi-1).img_stairs_up(round(pj(1)),
                            round(pj(2)))

                            vj = data.floor(fi-1).agents(aj).v;
                            rj = data.floor(fi-1).agents(aj).r;

                            % vector pointing from j to i
                            nij = (pi - pj) * data.meter_per_pixel;

                            % distance of agents
                            d = norm(nij);

                            % normalized vector pointing from j to i
                            nij = nij / d;
                            % tangential direction
                            tij = [-nij(2), nij(1)];

                            % sum of radii
                            rij = (ri + rj);

                            % repulsive interaction forces
                            if d < rij
                               T1 = data.k*(rij - d);
                               T2 = data.kappa*(rij - d)*dot((vj -
                                   vi),tij)*tij;
                            else
                               T1 = 0;
                               T2 = 0;
                            end

                            F = (data.A * exp((rij - d)/data.B) + T1)*nij
                                + T2;

                            data.floor(fi).agents(ai).f = ...
                                data.floor(fi).agents(ai).f + F;
                            data.floor(fi-1).agents(aj).f = ...
                                data.floor(fi-1).agents(aj).f - F;
                        end
                    end
                end
            end
        end
    end
```

```
end
```

Listing 1: addAgentRepulsiveForce.m

```
function data = addDesiredForce(data)
%ADDDESIREDFORCE add 'desired' force contribution (towards nearest exit or
%staircase)

for fi = 1:data.floor_count

    for ai=1:length(data.floor(fi).agents)

        % get agent's data
        p = data.floor(fi).agents(ai).p;
        m = data.floor(fi).agents(ai).m;
        v0 = data.floor(fi).agents(ai).v0;
        v = data.floor(fi).agents(ai).v;


        % get direction towards nearest exit
        ex = lerp2(data.floor(fi).img_dir_x, p(1), p(2));
        ey = lerp2(data.floor(fi).img_dir_y, p(1), p(2));
        e = [ex ey];

        % get force
        Fi = m * (v0*e - v)/data.tau;

        % add force
        data.floor(fi).agents(ai).f = data.floor(fi).agents(ai).f + Fi;
    end
end
```

Listing 2: addDesiredForce.m

```
function data = addWallForce(data)
%ADDWALLFORCE adds wall's force contribution to each agent

for fi = 1:data.floor_count

    for ai=1:length(data.floor(fi).agents)
        % get agents data
        p = data.floor(fi).agents(ai).p;
        ri = data.floor(fi).agents(ai).r;
        vi = data.floor(fi).agents(ai).v;

        % get direction from nearest wall to agent
        nx = lerp2(data.floor(fi).img_wall_dist_grad_x, p(1), p(2));
        ny = lerp2(data.floor(fi).img_wall_dist_grad_y, p(1), p(2));

        % get distance to nearest wall
```

```
        diW = lerp2(data.floor(fi).img_wall_dist , p(1), p(2));

        % get perpendicular and tangential unit vectors
        niW = [ nx ny];
        tiW = [-ny nx];


        % calculate force
        if diW < ri
            T1 = data.k * (ri - diW);
            T2 = data.kappa * (ri - diW) * dot(vi, tiW) * tiW;
        else
            T1 = 0;
            T2 = 0;
        end
        Fi = (data.A * exp((ri-diW)/data.B) + T1)*niW - T2;

        % add force to agent's current force
        data.floor(fi).agents(ai).f = data.floor(fi).agents(ai).f + Fi;
    end
end
```

Listing 3: addWallForce.m

```
function data = applyForcesAndMove(data)
%APPLYFORCESANDMOVE apply current forces to agents and move them using
%the timestep and current velocity

n_velocity_clamps = 0;

% loop over all floors
for fi = 1:data.floor_count

    % init logical arrays to indicate agents that change the floor or exit
    % the simulation
    floorchange = false(length(data.floor(fi).agents),1);
    exited = false(length(data.floor(fi).agents),1);

    % loop over all agents
    for ai=1:length(data.floor(fi).agents)
        % add current force contributions to velocity
        v = data.floor(fi).agents(ai).v + data.dt * ...
            data.floor(fi).agents(ai).f  / data.floor(fi).agents(ai).m;

        % clamp velocity
        if norm(v) > data.v_max
            v = v / norm(v) * data.v_max;
            n_velocity_clamps = n_velocity_clamps + 1;
        end

```

```
        % get agent's new position
        newp = data.floor(fi).agents(ai).p + ...
               v * data.dt / data.meter_per_pixel;

        % if the new position is inside a wall, remove perpendicular
        % component of the agent's velocity
        if lerp2(data.floor(fi).img_wall_dist, newp(1), newp(2)) < ...
                 data.floor(fi).agents(ai).r

            % get agent's position
            p = data.floor(fi).agents(ai).p;

            % get wall distance gradient (which is off course perpendicular
            % to the nearest wall)
            nx = lerp2(data.floor(fi).img_wall_dist_grad_x, p(1), p(2));
            ny = lerp2(data.floor(fi).img_wall_dist_grad_y, p(1), p(2));
            n = [nx ny];

            % project out perpendicular component of velocity vector
            v = v - dot(n,v)/dot(n,n)*n;

            % get agent's new position
            newp = data.floor(fi).agents(ai).p + ...
                   v * data.dt / data.meter_per_pixel;
        end

        if data.floor(fi).img_wall(round(newp(1)), round(newp(2)))
            newp = data.floor(fi).agents(ai).p;
            v = [0 0];
        end

        % update agent's velocity and position
        data.floor(fi).agents(ai).v = v;
        data.floor(fi).agents(ai).p = newp;

        % reset forces for next timestep
        data.floor(fi).agents(ai).f = [0 0];

        % check if agent reached a staircase and indicate floor change
        if data.floor(fi).img_stairs_down(round(newp(1)), round(newp(2)))
            floorchange(ai) = 1;
        end

        % check if agent reached an exit
        if data.floor(fi).img_exit(round(newp(1)), round(newp(2)))
            exited(ai) = 1;
            data.agents_exited = data.agents_exited +1;
        end
    end

```

```
    % add appropriate agents to next lower floor
    if fi > 1
        data.floor(fi-1).agents = [data.floor(fi-1).agents ...
                                   data.floor(fi).agents(floorchange)];
    end

    % delete these and exited agents
    data.floor(fi).agents = data.floor(fi).agents(~(floorchange|exited));
end

if n_velocity_clamps > 0
    fprintf(['WARNING: clamped velocity of %d agents, ' ...
            'possible simulation instability.\n'], n_velocity_clamps);
end
```

Listing 4: applyForcesAndMove.m

```
function val = checkForIntersection(data, floor_idx, agent_idx)
% check an agent for an intersection with another agent or a wall
% the check is kept as simple as possible
%
%  arguments:
%   data             global data structure
%   floor_idx        which floor to check
%   agent_idx        which agent on that floor
%   agent_new_pos    vector: [x,y], desired agent position to check
%
%  return:
%   0                for no intersection
%   1                has an intersection with wall
%   2                                    with another agent

val = 0;

p = data.floor(floor_idx).agents(agent_idx).p;
r = data.floor(floor_idx).agents(agent_idx).r;

% check for agent intersection
for i=1:length(data.floor(floor_idx).agents)
    if i~=agent_idx
        if norm(data.floor(floor_idx).agents(i).p-p)*data.meter_per_pixel
            ...
                <= r +  data.floor(floor_idx).agents(i).r
            val=2;
            return;
        end
    end
end

```

```
% check for wall intersection
if lerp2(data.floor(floor_idx).img_wall_dist, p(1), p(2)) < r
    val = 1;
end
```

Listing 5: checkForIntersection.m

```
mex 'fastSweeping.c'
mex 'getNormalizedGradient.c'
mex 'lerp2.c'
mex 'createRangeTree.c'
mex 'rangeQuery.c'
```

Listing 6: compileC.m

```
function data = initAgents(data)

% place agents randomly in desired spots, without overlapping



function radius = getAgentRadius()
    %radius of an agent in meters
    radius = data.r_min + (data.r_max-data.r_min)*rand();
end

data.agents_exited = 0; %how many agents have reached the exit
data.total_agent_count = 0;

floors_with_agents = 0;
agent_count = data.agents_per_floor;
for i=1:data.floor_count
    data.floor(i).agents = [];
    [y,x] = find(data.floor(i).img_spawn);

    if ~isempty(x)
        floors_with_agents = floors_with_agents + 1;
        for j=1:agent_count
            cur_agent = length(data.floor(i).agents) + 1;

            % init agent
            data.floor(i).agents(cur_agent).r = getAgentRadius();
            data.floor(i).agents(cur_agent).v = [0, 0];
            data.floor(i).agents(cur_agent).f = [0, 0];
            data.floor(i).agents(cur_agent).m = data.m;
            data.floor(i).agents(cur_agent).v0 = data.v0;

            tries = 10;
            while tries > 0
                % randomly pick a spot and check if it's free
```

```
                idx = randi(length(x));
                data.floor(i).agents(cur_agent).p = [y(idx), x(idx)];
                if checkForIntersection(data, i, cur_agent) == 0
                    tries = -1; % leave the loop
                end
                tries = tries - 1;
            end
            if tries > -1
                %remove the last agent
                data.floor(i).agents = data.floor(i).agents(1:end-1);
            end
        end
        data.total_agent_count = data.total_agent_count +
            length(data.floor(i).agents);

        if length(data.floor(i).agents) ~= agent_count
            fprintf(['WARNING: could only place %d agents on floor %d ' ...
                'instead of the desired %d.\n'], ...
                length(data.floor(i).agents), i, agent_count);
        end
    end
end
if floors_with_agents==0
    error('no spots to place agents!');
end

end
```

Listing 7: initAgents.m

```
function data = initEscapeRoutes(data)
%INITESCAPEROUTES Summary of this function goes here
%   Detailed explanation goes here

for i=1:data.floor_count

    boundary_data = zeros(size(data.floor(i).img_wall));
    boundary_data(data.floor(i).img_wall) =  1;
    %if (i == 1)
        boundary_data(data.floor(i).img_exit) = -1;
    %else
        boundary_data(data.floor(i).img_stairs_down) = -1;
    %end

    exit_dist = fastSweeping(boundary_data) * data.meter_per_pixel;
    [data.floor(i).img_dir_x, data.floor(i).img_dir_y] = ...
        getNormalizedGradient(boundary_data, -exit_dist);
end
```

Listing 8: initEscapeRoutes.m

```matlab
function data = initialize(config)
% initialize the internal data from the config data
%
%  arguments:
%   config      data structure from loadConfig()
%
%  return:
%   data        data structure: all internal data used for the main loop
%
%               all internal data is stored in pixels NOT in meters


data = config;

%for convenience
data.pixel_per_meter = 1/data.meter_per_pixel;

fprintf('Init escape routes...\n');
data = initEscapeRoutes(data);
fprintf('Init wall forces...\n');
data = initWallForces(data);
fprintf('Init agents...\n');
data = initAgents(data);

% maximum influence of agents on each other

data.r_influence = data.pixel_per_meter * ...
    fzero(@(r) data.A * exp((2*data.r_max-r)/data.B) - 1e-4, data.r_max);

fprintf('Init plots...\n');
%init the plots
%exit plot
data.figure_exit=figure;
hold on;
axis([0 data.duration 0 data.total_agent_count]);
title(sprintf('agents that reached the exit (total agents: %i)',
   data.total_agent_count));

%floors plot
data.figure_floors=figure;
data.figure_floors_subplots_w = data.floor_count;
data.figure_floors_subplots_h = 2;
for i=1:config.floor_count
    data.floor(i).agents_on_floor_plot =
        subplot(data.figure_floors_subplots_h,
        data.figure_floors_subplots_w ...
    , data.floor_count - i+1 + data.figure_floors_subplots_w);
    data.floor(i).building_plot = subplot(data.figure_floors_subplots_h,
        data.figure_floors_subplots_w ...
```

```
    , data.floor_count - i+1);
end
```

Listing 9: initialize.m

```
function data = initWallForces(data)
%INITWALLFORCES init wall distance maps and gradient maps for each floor

for i=1:data.floor_count

    % init boundary data for fast sweeping method
    boundary_data = zeros(size(data.floor(i).img_wall));
    boundary_data(data.floor(i).img_wall) = -1;

    % get wall distance
    wall_dist = fastSweeping(boundary_data) * data.meter_per_pixel;
    data.floor(i).img_wall_dist = wall_dist;

    % get normalized wall distance gradient
    [data.floor(i).img_wall_dist_grad_x, ...
     data.floor(i).img_wall_dist_grad_y] = ...
     getNormalizedGradient(boundary_data, wall_dist-data.meter_per_pixel);
end
```

Listing 10: initWallForces.m

```
function config = loadConfig(config_file)
% load the configuration file
%
%  arguments:
%   config_file     string, which configuration file to load
%


% get the path from the config file -> to read the images
config_path = fileparts(config_file);
if strcmp(config_path, '') == 1
    config_path = '.';
end

fid = fopen(config_file);
input = textscan(fid, '%s=%s');
fclose(fid);

keynames = input{1};
values = input{2};

%convert numerical values from string to double
v = str2double(values);
idx = ~isnan(v);
```

```
values(idx) = num2cell(v(idx));

config = cell2struct(values, keynames);


% read the images
for i=1:config.floor_count

    %building structure
    file = config.(sprintf('floor_%d_build', i));
    file_name = [config_path '/' file];
    img_build = imread(file_name);

    % decode images
    config.floor(i).img_wall = (img_build(:, :, 1) ==   0 ...
                              & img_build(:, :, 2) ==   0 ...
                              & img_build(:, :, 3) ==   0);

    config.floor(i).img_spawn = (img_build(:, :, 1) == 255 ...
                               & img_build(:, :, 2) ==   0 ...
                               & img_build(:, :, 3) == 255);

    config.floor(i).img_exit = (img_build(:, :, 1) ==   0 ...
                              & img_build(:, :, 2) == 255 ...
                              & img_build(:, :, 3) ==   0);

    config.floor(i).img_stairs_up = (img_build(:, :, 1) == 255 ...
                                   & img_build(:, :, 2) ==   0 ...
                                   & img_build(:, :, 3) ==   0);

    config.floor(i).img_stairs_down = (img_build(:, :, 1) ==   0 ...
                                     & img_build(:, :, 2) ==   0 ...
                                     & img_build(:, :, 3) == 255);

    %init the plot image here, because this won't change
    config.floor(i).img_plot = 5*config.floor(i).img_wall ...
        + 4*config.floor(i).img_stairs_up ...
        + 3*config.floor(i).img_stairs_down ...
        + 2*config.floor(i).img_exit ...
        + 1*config.floor(i).img_spawn;
    config.color_map = [1 1 1; 0.9 0.9 0.9; 0 1 0; 0.4 0.4 1; 1 0.4 0.4; 0
        0 0];
end
```

Listing 11: loadConfig.m

```
function plotAgentsPerFloor(data, floor_idx)
%plot time vs agents on floor

h = subplot(data.floor(floor_idx).agents_on_floor_plot);
```

```
set(h, 'position',[0.05+(data.floor_count -
    floor_idx)/(data.figure_floors_subplots_w+0.2), ...
    0.05, 1/(data.figure_floors_subplots_w*1.2), 0.3-0.05 ]);

if floor_idx~=data.floor_count
    set(h,'ytick',[]) %hide y-axis label
end

axis([0 data.duration 0 data.total_agent_count]);

hold on;
plot(data.time, length(data.floor(floor_idx).agents), 'b-');
hold off;

title(sprintf('agents on floor %i', floor_idx));
```

Listing 12: plotAgentsPerFloor.m

```
function plotExitedAgents(data)
%plot time vs exited agents

hold on;
plot(data.time, data.agents_exited, 'r-');
hold off;
```

Listing 13: plotExitedAgents.m

```
function plotFloor(data, floor_idx)
%PLOTFLOOR plot floor

h=subplot(data.floor(floor_idx).building_plot);

set(h, 'position',[(data.floor_count -
    floor_idx)/data.figure_floors_subplots_w, ...
    0.4, 1/(data.figure_floors_subplots_w+0.1), 0.6 ]);

hold off;
% the building image
imagesc(data.floor(floor_idx).img_plot);
hold on;

%plot options
colormap(data.color_map);
axis equal;
axis manual; %do not change axis on window resize

set(h, 'Visible', 'off')
%title(sprintf('floor %i', floor_idx));
```

```
% plot agents
if ~isempty(data.floor(floor_idx).agents)
    ang = [linspace(0,2*pi, 10) nan]';
    rmul = [cos(ang) sin(ang)] * data.pixel_per_meter;
    draw = cell2mat(arrayfun(@(a) repmat(a.p,length(ang),1) + a.r*rmul, ...
           data.floor(floor_idx).agents, 'UniformOutput', false)');
    line(draw(:,2), draw(:,1), 'Color', 'r');
end

% old drawing code...
% ang = linspace(0,2*pi, 10);
% cosang = cos(ang);
% sinang = sin(ang);
% for i=1:length(data.floor(floor_idx).agents)
%     p = data.floor(floor_idx).agents(i).p;
%     r = data.floor(floor_idx).agents(i).r * data.pixel_per_meter;
%     %r = norm(agent.v);
%     xp = r * cosang;
%     yp = r * sinang;
%     plot(p(2) + yp, p(1) + xp, 'Color', 'r');
%     %text(agent.pos(2),agent.pos(1),int2str(i));
% end

hold off;
end
```

Listing 14: plotFloor.m

```
function simulate(config_file)
% run this to start the simulation

if nargin==0
    config_file='../data/config1.conf';
end

fprintf('Load config file...\n');
config = loadConfig(config_file);

data = initialize(config);


data.time = 0;
frame = 0;
fprintf('Start simulation...\n');

while (data.time < data.duration)
    tstart=tic;
    data = addDesiredForce(data);
    data = addWallForce(data);
    data = addAgentRepulsiveForce(data);
```

```
    data = applyForcesAndMove(data);

    % do the plotting
    set(0,'CurrentFigure',data.figure_floors);
    for floor=1:data.floor_count
        plotAgentsPerFloor(data, floor);
        plotFloor(data, floor);
    end
    if data.save_frames==1
        print('-depsc2',sprintf('frames/%s_%04i.eps', ...
            data.frame_basename,frame), data.figure_floors);
    end

    set(0,'CurrentFigure',data.figure_exit);
    plotExitedAgents(data);


    % print mean/median velocity of agents on each floor
%     for fi = 1:data.floor_count
%         avgv = arrayfun(@(agent) norm(agent.v), data.floor(fi).agents);
%         fprintf('Mean/median velocity on floor %i: %g/%g m/s\n', fi,
    mean(avgv), median(avgv));
%     end


    if (data.time + data.dt > data.duration)
        data.dt = data.duration - data.time;
        data.time = data.duration;
    else
        data.time = data.time + data.dt;
    end

    if data.agents_exited == data.total_agent_count
        fprintf('All agents are now saved (or are they?). Time: %.2f
            sec\n', data.time);
        fprintf('Total Agents: %i\n', data.total_agent_count);

        print('-depsc2',sprintf('frames/exited_agents_%s.eps', ...
            data.frame_basename), data.figure_floors);
        break;
    end


    telapsed = toc(tstart);
    pause(max(data.dt - telapsed, 0.01));
    fprintf('Frame %i done (took %.3fs; %.3fs out of %.3gs simulated).\n',
       frame, telapsed, data.time, data.duration);
    frame = frame + 1;
end
```

```
fprintf('Simulation done.\n');
```

Listing 15: simulate.m

9.1.2 *C* code

```

#include <mex.h>
#include <string.h>

#include "tree_build.c"
#include "tree_query.c"
#include "tree_free.c"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
    *prhs[])
{
    point_t *points;
    tree_t *tree;
    int m, n;
    uchar *data;
    int *root_index;

    if (nlhs < 1)
        return;

    points = (point_t*) mxGetPr(prhs[0]);
    m = mxGetM(prhs[0]);
    n = mxGetN(prhs[0]);

    if (m != 2)
        mexErrMsgTxt("...");

    tree = build_tree(points, n);

    plhs[0] = mxCreateNumericMatrix(tree->first_free + sizeof(int), 1,
        mxUINT8_CLASS, mxREAL);
    data = (uchar*) mxGetPr(plhs[0]);

    root_index = (int*) data;
    *root_index = tree->root_index;
    memcpy(data + sizeof(int), tree->data, tree->first_free);

    free_tree(tree);
}
```

Listing 16: createRangeTree.c

```
#include "mex.h"

#include <math.h>

#if defined __GNUC__ && defined __FAST_MATH__ && !defined __STRICT_ANSI__
#define MIN(i, j) fmin(i, j)
#define MAX(i, j) fmax(i, j)
#define ABS(i)    fabs(i)
#else
#define MIN(i, j) ((i) < (j) ? (i) : (j))
#define MAX(i, j) ((i) > (j) ? (i) : (j))
#define ABS(i)    ((i) < 0.0 ? -(i) : (i))
#endif


#define SOLVE_AND_UPDATE    udiff = uxmin - uymin; \
                            if (ABS(udiff) >= 1.0) \
                            { \
                                up = MIN(uxmin, uymin) + 1.0; \
                            } \
                            else \
                            { \
                                up = (uxmin + uymin + sqrt(2.0 - udiff *
                                    udiff)) / 2.0; \
                                up = MIN(uij, up); \
                            } \
                            err_loc = MAX(ABS(uij - up), err_loc); \
                            u[ij] = up;

#define I_STEP(_uxmin, _uymin, _st) if (boundary[ij] == 0.0) \
                                    { \
                                        uij = un; \
                                        un = u[ij + _st]; \
                                        uxmin = _uxmin; \
                                        uymin = _uymin; \
                                        SOLVE_AND_UPDATE \
                                        ij += _st; \
                                    } \
                                    else \
                                    { \
                                        up = un; \
                                        un = u[ij + _st]; \
                                        ij += _st; \
                                    }



#define I_STEP_UP(_uxmin, _uymin)   I_STEP(_uxmin, _uymin, 1)
```

```c
#define I_STEP_DOWN(_uxmin, _uymin) I_STEP(_uxmin, _uymin,-1)

#define UX_NEXT un
#define UX_PREV up
#define UX_BOTH MIN(UX_PREV, UX_NEXT)

#define UY_RIGHT u[ij + m]
#define UY_LEFT  u[ij - m]
#define UY_BOTH  MIN(UY_LEFT, UY_RIGHT)



static void iteration(double *u, double *boundary, int m, int n, double
    *err)
{
    int i, j, ij;
    int m2, n2;
    double up, un, uij, uxmin, uymin, udiff, err_loc;

    m2 = m - 2;
    n2 = n - 2;

    *err = 0.0;
    err_loc = 0.0;

    /* first sweep */
    /* i = 0, j = 0 */
    ij = 0;
    un = u[ij];
    I_STEP_UP(UX_NEXT, UY_RIGHT)

    /* i = 1->m2, j = 0 */
    for (i = 1; i <= m2; ++i)
        I_STEP_UP(UX_BOTH, UY_RIGHT)

    /* i = m-1, j = 0 */
    I_STEP_UP(UX_PREV, UY_RIGHT)

    /* i = 0->m-1, j = 1->n2 */
    for (j = 1; j <= n2; ++j)
    {
        I_STEP_UP(UX_NEXT, UY_BOTH)

        for (i = 1; i <= m2; ++i)
            I_STEP_UP(UX_BOTH, UY_BOTH)

        I_STEP_UP(UX_PREV, UY_BOTH)
    }

    /* i = 0, j = n-1 */
```

```
    I_STEP_UP(UX_NEXT, UY_LEFT)

    /* i = 1->m2, j = n-1 */
    for (i = 1; i <= m2; ++i)
        I_STEP_UP(UX_BOTH, UY_LEFT)

    /* i = m-1, j = n-1 */
    I_STEP_UP(UX_PREV, UY_LEFT)


    /* sweep 2 */
    /* i = 0, j = n-1 */
    ij = (n-1)*m;
    un = u[ij];
    I_STEP_UP(UX_NEXT, UY_LEFT)

    /* i = 1->m2, j = n-1 */
    for (i = 1; i <= m2; ++i)
        I_STEP_UP(UX_BOTH, UY_LEFT)

    /* i = m-1, j = n-1 */
    I_STEP_UP(UX_PREV, UY_LEFT)

    /* i = 0->m-1, j = n2->1 */
    for (j = n2; j >= 1; --j)
    {
        ij = j*m;
        un = u[ij];
        I_STEP_UP(UX_NEXT, UY_BOTH)

        for (i = 1; i <= m2; ++i)
            I_STEP_UP(UX_BOTH, UY_BOTH)

        I_STEP_UP(UX_PREV, UY_BOTH)
    }

    /* i = 0, j = 0 */
    ij = 0;
    un = u[ij];
    I_STEP_UP(UX_NEXT, UY_RIGHT)

    /* i = 1->m2, j = 0 */
    for (i = 1; i <= m2; ++i)
        I_STEP_UP(UX_BOTH, UY_RIGHT)

    /* i = m-1, j = 0 */
    I_STEP_UP(UX_PREV, UY_RIGHT)

    /* sweep 3 */
    /* i = m-1, j = n-1 */
```

```
    ij = m*n - 1;
    un = u[ij];
    I_STEP_DOWN(UX_NEXT, UY_LEFT)

    /* i = m2->1, j = n-1 */
    for (i = m2; i >= 1; --i)
        I_STEP_DOWN(UX_BOTH, UY_LEFT)

    /* i = 0, j = n-1 */
    I_STEP_DOWN(UX_PREV, UY_LEFT)

    /* i = m-1->0, j = n2->1 */
    for (j = n2; j >= 1; --j)
    {
        I_STEP_DOWN(UX_NEXT, UY_BOTH)

        for (i = m2; i >= 1; --i)
            I_STEP_DOWN(UX_BOTH, UY_BOTH)

        I_STEP_DOWN(UX_PREV, UY_BOTH)
    }

    /* i = m-1, j = 0 */
    I_STEP_DOWN(UX_NEXT, UY_RIGHT)

    /* i = m2->1, j = 0 */
    for (i = m2; i >= 1; --i)
        I_STEP_DOWN(UX_BOTH, UY_RIGHT)

    /* i = 0, j = 0 */
    I_STEP_DOWN(UX_PREV, UY_RIGHT)

    /* sweep 4 */
    /* i = m-1, j = 0 */
    ij = m - 1;
    un = u[ij];
    I_STEP_DOWN(UX_NEXT, UY_RIGHT)

    /* i = m2->1, j = 0 */
    for (i = m2; i >= 1; --i)
        I_STEP_DOWN(UX_BOTH, UY_RIGHT)

    /* i = 0, j = 0 */
    I_STEP_DOWN(UX_PREV, UY_RIGHT)

    /* i = m-1->0, j = 1->n2 */
    for (j = 1; j <= n2; ++j)
    {
        ij = m - 1 + j*m;
        un = u[ij];
```

```
        I_STEP_DOWN(UX_NEXT, UY_BOTH)

        for (i = m2; i >= 1; --i)
            I_STEP_DOWN(UX_BOTH, UY_BOTH)

        I_STEP_DOWN(UX_PREV, UY_BOTH)
    }

    /* i = m-1, j = n-1 */
    ij = m*n - 1;
    un = u[ij];
    I_STEP_DOWN(UX_NEXT, UY_LEFT)

    /* i = m2->1, j = n-1 */
    for (i = m2; i >= 1; --i)
        I_STEP_DOWN(UX_BOTH, UY_LEFT)

    /* i = 0, j = n-1 */
    I_STEP_DOWN(UX_PREV, UY_LEFT)

    *err = MAX(*err, err_loc);
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
   *prhs[])
{
    double *u, *boundary;
    double tol, err;
    int m, n, entries, max_iter, i;

    /* Check number of outputs */
    if (nlhs < 1)
        return;
    else if (nlhs > 1)
        mexErrMsgTxt("At most 1 output argument needed.");

    /* Get inputs */
    if (nrhs < 1)
        mexErrMsgTxt("At least 1 input argument needed.");
    else if (nrhs > 3)
        mexErrMsgTxt("At most 3 input arguments used.");



    /* Get boundary */
    if (!mxIsDouble(prhs[0]) || mxIsClass(prhs[0], "sparse"))
        mexErrMsgTxt("Boundary field needs to be a full double precision
           matrix.");

    boundary = mxGetPr(prhs[0]);
```

```c
    m = mxGetM(prhs[0]);
    n = mxGetN(prhs[0]);
    entries = m * n;

    /* Get max iterations  */
    if (nrhs >= 2)
    {
        if (!mxIsDouble(prhs[1]) || mxGetM(prhs[1]) != 1 ||
            mxGetN(prhs[1]) != 1)
            mexErrMsgTxt("Maximum iteration needs to be positive
                integer.");
        max_iter = (int) *mxGetPr(prhs[1]);
        if (max_iter <= 0)
            mexErrMsgTxt("Maximum iteration needs to be positive
                integer.");
    }
    else
        max_iter = 20;

    /* Get tolerance */
    if (nrhs >= 3)
    {
        if (!mxIsDouble(prhs[2]) || mxGetM(prhs[2]) != 1 ||
            mxGetN(prhs[2]) != 1)
            mexErrMsgTxt("Tolerance needs to be a positive real number.");
        tol = *mxGetPr(prhs[2]);
        if (tol < 0)
            mexErrMsgTxt("Tolerance needs to be a positive real number.");
    }
    else
        tol = 1e-12;


    /* create and init output (distance) matrix */
    plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
    u = mxGetPr(plhs[0]);

    for (i = 0; i < entries; ++i)
        u[i] = boundary[i] < 0.0 ? 0.0 : 1.0e10;

    err = 0.0;
    i = 0;
    do
    {
        iteration(u, boundary, m, n, &err);
        ++i;
    } while (err > tol && i < max_iter);
}
```

Listing 17: fastSweeping.c

```c
#include "mex.h"

#include <math.h>

#define INTERIOR(i, j)  (boundary[(i) + m*(j)] == 0)

#define DIST(i, j)  dist[(i) + m*(j)]
#define XGRAD(i, j) xgrad[(i) + m*(j)]
#define YGRAD(i, j) ygrad[(i) + m*(j)]

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
    *prhs[])
{
    double *xgrad, *ygrad, *boundary, *dist;
    double dxp, dxm, dyp, dym, xns, yns, nrm;
    int m, n, i, j, nn;

    /* Check number of outputs */
    if (nlhs < 2)
        mexErrMsgTxt("At least 2 output argument needed.");
    else if (nlhs > 2)
        mexErrMsgTxt("At most 2 output argument needed.");

    /* Get inputs */
    if (nrhs < 2)
        mexErrMsgTxt("At least 2 input argument needed.");
    else if (nrhs > 2)
        mexErrMsgTxt("At most 2 input argument used.");



    /* Get boundary */
    if (!mxIsDouble(prhs[0]) || mxIsClass(prhs[0], "sparse"))
        mexErrMsgTxt("Boundary field needs to be a full double precision
            matrix.");

    boundary = mxGetPr(prhs[0]);
    m = mxGetM(prhs[0]);
    n = mxGetN(prhs[0]);

    /* Get distance field */
    if (!mxIsDouble(prhs[1]) || mxIsClass(prhs[1], "sparse") ||
        mxGetM(prhs[1]) != m || mxGetN(prhs[1]) != n)
        mexErrMsgTxt("Distance field needs to be a full double precision
            matrix with same dimension as the boundary.");

    dist = mxGetPr(prhs[1]);
    m = mxGetM(prhs[1]);
    n = mxGetN(prhs[1]);
```

```c
    /* create and init output (gradient) matrices */
    plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
    plhs[1] = mxCreateDoubleMatrix(m, n, mxREAL);
    xgrad = mxGetPr(plhs[0]);
    ygrad = mxGetPr(plhs[1]);



    for (j = 0; j < n; ++j)
        for (i = 0; i < m; ++i)
            if (INTERIOR(i,j))
            {
                if (i > 0)
                    dxm = INTERIOR(i-1,j) ? DIST(i-1,j) : DIST(i,j);
                else
                    dxm = DIST(i,j);

                if (i < m-1)
                    dxp = INTERIOR(i+1,j) ? DIST(i+1,j) : DIST(i,j);
                else
                    dxp = DIST(i,j);

                if (j > 0)
                    dym = INTERIOR(i,j-1) ? DIST(i,j-1) : DIST(i,j);
                else
                    dym = DIST(i,j);

                if (j < n-1)
                    dyp = INTERIOR(i,j+1) ? DIST(i,j+1) : DIST(i,j);
                else
                    dyp = DIST(i,j);

                XGRAD(i, j) = (dxp - dxm) / 2.0;
                YGRAD(i, j) = (dyp - dym) / 2.0;
                nrm = sqrt(XGRAD(i, j)*XGRAD(i, j) + YGRAD(i, j)*YGRAD(i,
                    j));
                if (nrm > 1e-12)
                {
                    XGRAD(i, j) /= nrm;
                    YGRAD(i, j) /= nrm;
                }
            }
            else
            {
                XGRAD(i, j) = 0.0;
                YGRAD(i, j) = 0.0;
            }

    for (j = 0; j < n; ++j)
```

```
        for (i = 0; i < m; ++i)
            if (!INTERIOR(i, j))
            {
                xns = 0.0;
                yns = 0.0;
                nn = 0;
                if (i > 0 && INTERIOR(i-1,j))
                {
                    xns += XGRAD(i-1,j);
                    yns += YGRAD(i-1,j);
                    ++nn;
                }
                if (i < m-1 && INTERIOR(i+1,j))
                {
                    xns += XGRAD(i+1,j);
                    yns += YGRAD(i+1,j);
                    ++nn;
                }
                if (j > 0 && INTERIOR(i,j-1))
                {
                    xns += XGRAD(i,j-1);
                    yns += YGRAD(i,j-1);
                    ++nn;
                }
                if (j < n-1 && INTERIOR(i,j+1))
                {
                    xns += XGRAD(i,j+1);
                    yns += YGRAD(i,j+1);
                    ++nn;
                }

                if (nn > 0)
                {
                    XGRAD(i, j) = xns / nn;
                    YGRAD(i, j) = yns / nn;
                }
            }
}
```

Listing 18: getNormalizedGradient.c

```

#include <mex.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
    *prhs[])
{
    int m, n, i0, i1, j0, j1, idx00;
    double *data, *out, x, y, wx0, wy0, wx1, wy1;
    double d00, d01, d10, d11;
```

```c
    if (nlhs < 1)
        return;
    else if (nlhs > 1)
        mexErrMsgTxt("Exactly one output argument needed.");

    if (nrhs != 3)
        mexErrMsgTxt("Exactly three input arguments needed.");

    m = mxGetM(prhs[0]);
    n = mxGetN(prhs[0]);
    data = mxGetPr(prhs[0]);
    x = *mxGetPr(prhs[1]) - 1;
    y = *mxGetPr(prhs[2]) - 1;

    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    out = mxGetPr(plhs[0]);

    x = x < 0 ? 0 : x > m - 1 ? m - 1 : x;
    y = y < 0 ? 0 : y > n - 1 ? n - 1 : y;
    i0 = (int) x;
    j0 = (int) y;
    i1 = i0 + 1;
    i1 = i1 > m - 1 ? m - 1 : i1;
    j1 = j0 + 1;
    j1 = j1 > n - 1 ? n - 1 : j1;

    idx00 = i0 + m * j0;
    d00 = data[idx00];
    d01 = data[idx00 + m];
    d10 = data[idx00 + 1];
    d11 = data[idx00 + m + 1];

    wx1 = x - i0;
    wy1 = y - j0;
    wx0 = 1.0 - wx1;
    wy0 = 1.0 - wy1;

    *out = wx0 * (wy0 * d00 + wy1 * d01) + wx1 * (wy0 * d10 + wy1 * d11);
}
```

Listing 19: lerp2.c

```c
#include <mex.h>
#include <string.h>

#include "tree_build.c"
#include "tree_query.c"
#include "tree_free.c"
```

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
    *prhs[])
{
    tree_t *tree;
    int n, i;
    int *point_idx, *root_idx;
    range_t *range;
    uchar *data;

    if (nlhs != 1)
        mexErrMsgTxt("...");

    if (nrhs < 5)
        mexErrMsgTxt("...");
    else if (nrhs > 5)
        mexErrMsgTxt("...");

    data = (uchar*) mxGetPr(prhs[0]);

    tree = (tree_t*) malloc(sizeof(tree_t));
    tree->first_free = mxGetM(prhs[0]) - sizeof(int);
    tree->total_size = tree->first_free;
    root_idx = (int*) data;
    tree->root_index = *root_idx;
    tree->data = data + sizeof(int);

    n = mxGetN(prhs[0]);
    if (n != 1)
        mexErrMsgTxt("...");

    range = range_query(tree, *mxGetPr(prhs[1]), *mxGetPr(prhs[2]),
        *mxGetPr(prhs[3]), *mxGetPr(prhs[4]));

    plhs[0] = mxCreateNumericMatrix(range->n, 1, mxUINT32_CLASS, mxREAL);
    point_idx = (int*) mxGetPr(plhs[0]);

    for (i = 0; i < range->n; ++i)
        point_idx[i] = range->point_idx[i] + 1;

    free_range(range);
    free(tree);
}
```

Listing 20: rangeQuery.c

```
#ifndef TREE_H
#define TREE_H

#include "tree_types.h"
```

```c
/* build a 2D range tree using the given points */
tree_t* build_tree(point_t *points, int n);

/* query a range tree */
range_t* range_query(tree_t *tree, double x_min, double x_max, double
    y_min, double y_max);

/* free memory of a tree */
void free_tree(tree_t *tree);

/* free memory of a range */
void free_range(range_t *range);

#endif
```

Listing 21: tree.h

```c
#ifndef TREE_BUILD_H
#define TREE_BUILD_H

#include "tree.h"

/* recursively build a subtree */
int build_subtree(tree_t *tree, double *x_vals, const int nx, point_t
    *points, int *point_idx, const int np);

/* double comparison for qsort */
int compare_double(const void *a, const void *b);

/* index array sorting functions, sort point index array by point y
    coordinates */
void index_sort_y(const point_t *points, int *point_idx, const int n);
void index_quicksort_y(const point_t *points, int *point_idx, int l, int
    r);
int index_partition_y(const point_t *points, int *point_idx, int l, int r);

#endif
```

Listing 22: tree_build.h

```c

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "tree_build.h"

tree_t* build_tree(point_t *points, int n)
```

```c
{
    int nx, i, j, *point_idx;
    double *x_vals;
    tree_t *tree;

    /* get x coordinate values of all points */
    x_vals = (double*) malloc(n * sizeof(double));
    for (i = 0; i < n; ++i)
        x_vals[i] = points[i].x;

    /* sort x values */
    qsort(x_vals, n, sizeof(double), compare_double);

    /* count number of unique x values */
    nx = 1;
    for (i = 1; i < n; ++i)
        if (x_vals[i] != x_vals[i - 1])
            ++nx;

    /* remove duplicates */
    j = 0;
    for (i = 0; i < nx; ++i)
    {
        x_vals[i] = x_vals[j];
        while (x_vals[i] == x_vals[j])
            ++j;
    }

    /* create an index array */
    point_idx = (int*) malloc(n * sizeof(int));
    for (i = 0; i < n; ++i)
        point_idx[i] = i;

    /* sort index array by y coordinates of associated points */
    index_sort_y(points, point_idx, n);

    /* init tree */
    tree = (tree_t*) malloc(sizeof(tree_t));
    tree->total_size = n * sizeof(point_t);
    tree->data = (uchar*) malloc(tree->total_size);

    /* copy point coordinates to tree data */
    memcpy(tree->data, points, n * sizeof(point_t));

    /* set first free byte and root index of the tree */
    tree->first_free = n * sizeof(point_t);
    tree->root_index = tree->first_free;

    /* recursively build tree */
    build_subtree(tree, x_vals, nx, points, point_idx, n);
```

```c
    /* free temporaries */
    free(x_vals);
    return tree;
}


int build_subtree(tree_t *tree, double *x_vals, const int nx, point_t
   *points, int *point_idx, const int np)
{
    int i, j, k, nx_left, np_left, node_size, right_idx;
    node_t *node;
    int *node_point_idx, *point_idx_left, *point_idx_right, node_idx;
    uchar *new_data;

    assert(nx > 0);
    assert(np > 0);

    /* allocate memory in the tree data structure */
    node_size = sizeof(node_t) + np * sizeof(int);
    while (tree->first_free + node_size > tree->total_size)
    {
        tree->total_size <<= 1;
        new_data = (uchar*) malloc(tree->total_size * sizeof(uchar));
        for (i = 0; i < tree->first_free; ++i)
            new_data[i] = tree->data[i];
        free(tree->data);
        tree->data = new_data;
    }
    node_idx = tree->first_free;
    node = (node_t*) &tree->data[node_idx];
    tree->first_free += node_size;

    /* set number of stored points */
    node->np = np;
    node_point_idx = (int*) (node + 1);

    /* copy point indices to node */
    memcpy(node_point_idx, point_idx, np * sizeof(int));

    /* create child node if there is only one x value left, otherwise
       create interior node */
    if (nx == 1)
    {
        node->right_idx = -1;
        node->x_val = x_vals[0];
    }
    else
    {
        /* get median of x values */
```

```
        nx_left = nx >> 1;
        node->x_val = x_vals[nx_left - 1];

        /* count points belonging to the left child */
        np_left = 0;
        for (i = 0; i < np; ++i)
        {
            if (points[point_idx[i]].x <= node->x_val)
                ++np_left;
        }

        /* allocate memory for children's index arrays */
        point_idx_left = (int*) malloc(np_left * sizeof(int));
        point_idx_right = (int*) malloc((np - np_left) * sizeof(int));

        /* fill index arrays */
        j = 0;
        k = 0;
        for (i = 0; i < np; ++i)
        {
            if (points[point_idx[i]].x <= node->x_val)
                point_idx_left[j++] = point_idx[i];
            else
                point_idx_right[k++] = point_idx[i];
        }

        /* free current node's temporary index array */
        free(point_idx);

        /* build left subtree */
        build_subtree(tree, x_vals, nx_left, points, point_idx_left,
            np_left);

        /* build right subtree and get its root node index */
        right_idx = build_subtree(tree, x_vals + nx_left, nx - nx_left,
            points, point_idx_right, np - np_left);
        /* update node pointer (could have changed during build_subtree,
            because of data allocation) */
        node = (node_t*) &tree->data[node_idx];
        /* update node's right child index */
        node->right_idx = right_idx;
    }

    /* return node index to parent */
    return node_idx;
}

int compare_double(const void *a, const void *b)
{
    double ad, bd;
```

```
    ad = *((double*) a);
    bd = *((double*) b);
    return (ad < bd) ? -1 : (ad > bd) ? 1 : 0;
}

void index_sort_y(const point_t *points, int *point_idx, const int n)
{
    index_quicksort_y(points, point_idx, 0, n - 1);
}

void index_quicksort_y(const point_t *points, int *point_idx, int l, int r)
{
    int p;

    /* quicksort point indices by point y coordinates, don't touch point
       array itself */
    while (l < r)
    {
        p = index_partition_y(points, point_idx, l, r);
        if (r - p > p - l)
        {
            index_quicksort_y(points, point_idx, l, p - 1);
            l = p + 1;
        }
        else
        {
            index_quicksort_y(points, point_idx, p + 1, r);
            r = p - 1;
        }
    }
}

int index_partition_y(const point_t *points, int *point_idx, int l, int r)
{
    int i, j, tmp;
    double pivot;

    /* rightmost element is pivot */
    i = l;
    j = r - 1;
    pivot = points[point_idx[r]].y;

    /* quicksort partition */
    do
    {
        while (points[point_idx[i]].y <= pivot && i < r)
            ++i;

        while (points[point_idx[j]].y >= pivot && j > l)
            --j;
```

```c
        if (i < j)
        {
            tmp = point_idx[i];
            point_idx[i] = point_idx[j];
            point_idx[j] = tmp;
        }
    } while (i < j);

    if (points[point_idx[i]].y > pivot)
    {
        tmp = point_idx[i];
        point_idx[i] = point_idx[r];
        point_idx[r] = tmp;
    }

    return i;
}
```

Listing 23: tree_build.c

```c

#include <stdlib.h>

#include "tree.h"

void free_tree(tree_t *tree)
{
    free(tree->data);
}

void free_range(range_t *range)
{
    free(range->point_idx);
}
```

Listing 24: tree_free.c

```c
#ifndef TREE_QUERY_H
#define TREE_QUERY_H

#include "tree_types.h"

/* appends a point-index to a range, icnreases range capacity if needed */
void range_append(range_t *range, int idx);

/* finds the split node of a given query */
int find_split_node(tree_t *tree, int node_idx, range_t *range);

/* query the points of a node by a given range by y-coordinate */
```

```c
void range_query_y(tree_t *tree, int node_idx, range_t *range);

#endif
```

Listing 25: tree_query.h

```c

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "tree_query.h"

#define LEFT_CHILD_IDX(node_idx, node)  (node_idx) + sizeof(node_t) +
    (node)->np * sizeof(int)
#define RIGHT_CHILD_IDX(node_idx, node) (node)->right_idx
#define NODE_FROM_IDX(tree, node_idx)   (node_t*) &(tree)->data[node_idx];

range_t* range_query(tree_t *tree, double x_min, double x_max, double
    y_min, double y_max)
{
    int split_node_idx, node_idx;
    node_t *split_node, *node;
    range_t *range;

    /* init range */
    range = (range_t*) malloc(sizeof(range_t));
    range->min.x = x_min;
    range->max.x = x_max;
    range->min.y = y_min;
    range->max.y = y_max;
    range->n = 0;
    range->total_size = 16;
    range->point_idx = (int*) malloc(range->total_size * sizeof(int));

    /* find split node */
    split_node_idx = find_split_node(tree, tree->root_index, range);
    split_node = NODE_FROM_IDX(tree, split_node_idx);

    /* if split node is a child */
    if (split_node->right_idx == -1)
    {
        range_query_y(tree, split_node_idx, range);
        return range;
    }

    /* follow left path of the split node */
    node_idx = LEFT_CHILD_IDX(split_node_idx, split_node);
    node = NODE_FROM_IDX(tree, node_idx);
    while (node->right_idx != -1)
```

```
    {
        if (range->min.x <= node->x_val)
        {
            range_query_y(tree, RIGHT_CHILD_IDX(node_idx, node), range);
            node_idx = LEFT_CHILD_IDX(node_idx, node);
        }
        else
            node_idx = RIGHT_CHILD_IDX(node_idx, node);
        node = NODE_FROM_IDX(tree, node_idx);
    }
    range_query_y(tree, node_idx, range);

    /* follow right path of the split node */
    node_idx = split_node->right_idx;
    node = NODE_FROM_IDX(tree, node_idx);
    while (node->right_idx != -1)
    {
        if (range->max.x > node->x_val)
        {
            range_query_y(tree, LEFT_CHILD_IDX(node_idx, node), range);
            node_idx = RIGHT_CHILD_IDX(node_idx, node);
        }
        else
            node_idx = LEFT_CHILD_IDX(node_idx, node);
        node = NODE_FROM_IDX(tree, node_idx);
    }
    range_query_y(tree, node_idx, range);

    return range;
}

void range_append(range_t *range, int idx)
{
    int *new_point_idx;
    int new_size, i;

    /* just append if there is enough place, otherwise double capacity and
       append */
    if (range->n < range->total_size)
        range->point_idx[range->n++] = idx;
    else
    {
        new_size = range->total_size << 1;
        new_point_idx = (int*) malloc(new_size * sizeof(int));
        for (i = 0; i < range->n; ++i)
            new_point_idx[i] = range->point_idx[i];
        new_point_idx[range->n++] = idx;
        free(range->point_idx);
        range->point_idx = new_point_idx;
        range->total_size = new_size;
```

```
    }
}

int find_split_node(tree_t *tree, int node_idx, range_t *range)
{
    node_t *node;

    node = (node_t*) &tree->data[node_idx];
    /* check if this node is the split node */
    if (range->min.x <= node->x_val && range->max.x > node->x_val)
        return node_idx;

    /* ...or if it is a child (and therefor the split node) */
    if (node->right_idx == -1)
        return node_idx;

    /* otherwise search the split node at the left or right of the current
       node */
    if (range->max.x <= node->x_val)
        return find_split_node(tree, LEFT_CHILD_IDX(node_idx, node),
            range);
    else
        return find_split_node(tree, RIGHT_CHILD_IDX(node_idx, node),
            range);
}

void range_query_y(tree_t *tree, int node_idx, range_t *range)
{
    point_t *points;
    double y;
    int i, j, k, m, start, end;
    int *point_idx;
    node_t *node;

    node = (node_t*) &tree->data[node_idx];
    points = (point_t*) tree->data;
    point_idx = (int*) (node + 1);

    /* return if all points are outside the range */
    if (points[point_idx[0]].y > range->max.y || points[point_idx[node->np
       - 1]].y < range->min.y)
        return;

    /* binary search for lower end of the range */
    y = range->min.y;
    j = 0;
    k = node->np - 1;
    while (j != k)
    {
        m = (j + k) / 2;
```

```
        if (points[point_idx[m]].y >= y)
            k = m;
        else
            j = m + 1;
    }
    start = j;

    /* binary search for higher end of the range */
    y = range->max.y;
    j = 0;
    k = node->np - 1;
    while (j != k)
    {
        m = (j + k + 1) / 2;
        if (points[point_idx[m]].y > y)
            k = m - 1;
        else
            j = m;
    }
    end = j;

    /* append found points to the range */
    for (i = start; i <= end; ++i)
        if (points[point_idx[i]].x <= range->max.x)
            range_append(range, point_idx[i]);

}
```

Listing 26: tree_query.c

```
#ifndef TREE_TYPES_H
#define TREE_TYPES_H

typedef unsigned char uchar;

/* 2D point */
typedef struct
{
    double x;
    double y;
} point_t;

/* tree */
typedef struct
{
    /* byte data array with points and nodes */
    uchar *data;

    /* index of first unused byte */
    int first_free;
```

```c
    /* total number of allocated bytes */
    int total_size;

    /* index of the root node in the data array*/
    int root_index;
} tree_t;

/* node */
typedef struct
{
    /* index of the right child node (left child follows directly after
       current) */
    int right_idx;

    /* number of associated points */
    int np;

    /* associated x-coordinate value */
    double x_val;
} node_t;

/* range */
typedef struct
{
    /* point index list */
    int *point_idx;

    /* number of saved indices */
    int n;

    /* total number of allocated indices */
    int total_size;

    /* minimum range point */
    point_t min;

    /* maximum range point */
    point_t max;
} range_t;

#endif
```

Listing 27: tree_types.h