



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



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
Hardmeier
Jenal
Küng
Thaler

First name
Häns
Andrin
Beat
Felix

Supervising lecturer

Last name
Baliotti
Donnay

First name
Stefano
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.

Contents

1	Abstract	4
2	Individual contributions	4
3	Introduction and Motivations	5
3.1	Introduction	5
3.2	Motivation	5
3.3	Fundamental Questions	6
4	Description of the Model	6
4.1	General Model	6
4.2	Forces	6
4.2.1	Desired Direction Force	7
4.2.2	Repulsive Interaction Force	7
4.2.3	Wall Force	7
5	Implementation	8
5.1	<i>MATLAB</i> Code	8
5.2	Pathfinding	8
5.3	Profiling & Optimization	9
5.3.1	Parallelization	10
5.3.2	Replacing <i>MATLAB</i> standard functions	11
5.3.3	Range Tree	11
6	Simulation Results and Discussion	11
6.1	Expected Results	11
6.2	Simulation Results	12
6.2.1	Comparison to previous projects	12
6.3	Improvements	13
7	Summary	15
7.1	Thanks	15
8	References	15
9	Appendix	16
9.1	Code	16
9.1.1	<i>MATLAB</i> code	16
9.1.2	<i>C</i> code	31

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 formally 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 building 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 exactly 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 Helbling [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 answering 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. Whereas, the shape and complexity of rooms and building is a key factor, contributing to the behavior of the agents as well.

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, model parameters 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.

Helbling 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 . Simultaneously, he or she tries to keep a velocity-dependent distance from other pedestrians j and walls W ."

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

That means the change of position $\mathbf{r}$ is given by the velocity: $\mathbf{v}_i = \frac{d\mathbf{r}_i}{dt}$. 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.

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

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]

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

According to Helbling 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 additional forces are considered: $k(r_{ij} - d_{ij}) \mathbf{n}_{ij}$ and $\kappa(r_{ij} - d_{ij}) \Delta v_{ij}^t \mathbf{t}_{ij}$. "Here $\mathbf{t}_{ij} = (-n_{ij}^2, n_{ij}^1)$ means the tangential direction and $\Delta v_{ij}^t = (\mathbf{v}_j - \mathbf{v}_i) \cdot \mathbf{t}_{ij}$ 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

Helbling states it the following: " d_{iW} means the distance to wall W , $\mathbf{n}_{iW}$ denotes the direction perpendicular to it, and $\mathbf{t}_{iW}$ the direction tangential to it, the corresponding interaction force with the wall is given by:" [2]

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

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* algorithm 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 flexible 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 on the following page describes how a building floor image must look like. The agents are placed randomly within the agents spawning area(s).

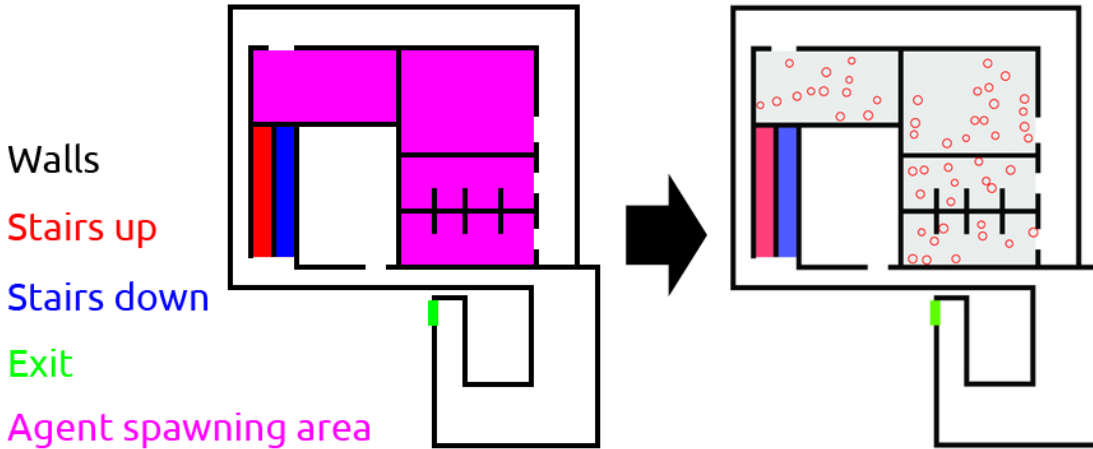


Figure 1: Left the building floor image and right how it looks when the simulation is running

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).

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

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 algorithms 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, closely 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 purposes. 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 measures, 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 addressed 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 Helbling as described in [2] and [6]. Our implementation realistically 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.

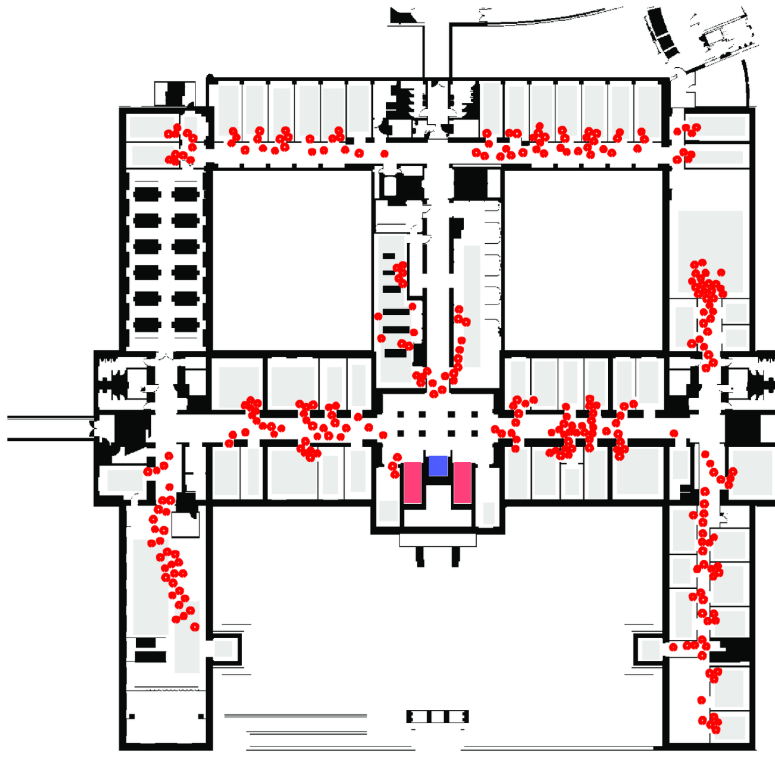


Figure 3: Visualisation of evacuation in the CAB Building E Floor

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 simplistic 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 therefore the simulation times are very low. In our implementation using the forces as defined in [6] on 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. Therefore 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.

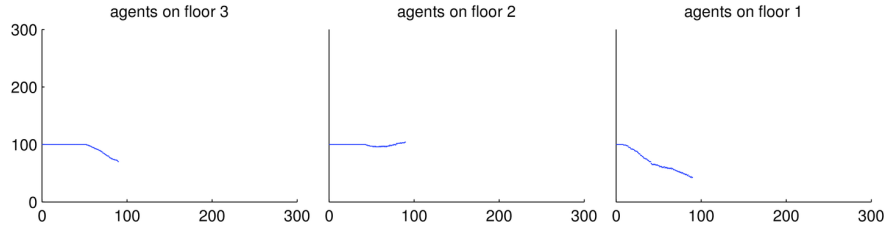
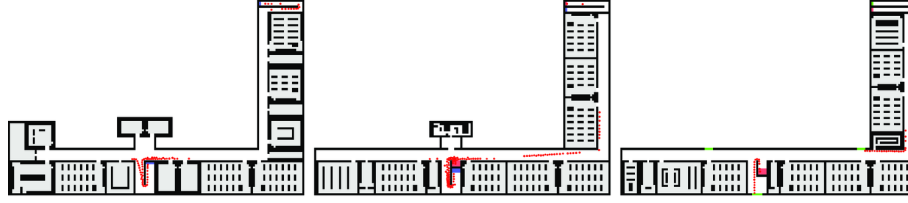


Figure 4: One frame of our simulation of the building given in [12].

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 performance 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 efficiency 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] Helbling, Dirk; Molnar, Peter (1995): Social Force Model for Pedestrians Dynamics.
- [3] Helbling, 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] Helbling, Dirk; Johansson, Anders (2007): Dynamics of crowd disasters: An empirical Study.
- [6] Helbling, Dirk et al. (2000): Simulating dynamical features of escape panic.
- [7] Helbling, 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 Zurich, 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

```

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

5  % Obstruction effects in case of physical interaction

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

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

15  % update range tree of lower floor
    tree_lower = tree;

17

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

19

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

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

```

```

29      % use range tree to get the indices of all agents near agent ai
30      idx = rangeQuery(tree, pi(1) - r_max, pi(1) + r_max, ...
31                          pi(2) - r_max, pi(2) + r_max)';
32
33      % loop over agents near agent ai
34      for aj = idx
35
36          % if force has not been calculated yet...
37          if aj > ai
38              pj = data.floor(fi).agents(aj).p;
39              vj = data.floor(fi).agents(aj).v;
40              rj = data.floor(fi).agents(aj).r;
41
42              % vector pointing from j to i
43              nij = (pi - pj) * data.meter_per_pixel;
44
45              % distance of agents
46              d = norm(nij);
47
48              % normalized vector pointing from j to i
49              nij = nij / d;
50              % tangential direction
51              tij = [-nij(2), nij(1)];
52
53              % sum of radii
54              rij = (ri + rj);
55
56              % repulsive interaction forces
57              if d < rij
58                  T1 = data.k*(rij - d);
59                  T2 = data.kappa*(rij - d)*dot((vj - vi),tij)*tij;
60              else
61                  T1 = 0;
62                  T2 = 0;
63              end
64
65              F = (data.A * exp((rij - d)/data.B) + T1)*nij + T2;
66
67              data.floor(fi).agents(ai).f = ...
68                  data.floor(fi).agents(ai).f + F;
69              data.floor(fi).agents(aj).f = ...
70                  data.floor(fi).agents(aj).f - F;
71          end
72      end
73
74      % include agents on stairs!
75      if fi > 1
76          % use range tree to get the indices of all agents near agent ai
77          if ~isempty(data.floor(fi-1).agents)

```

```

79         idx = rangeQuery(tree_lower, pi(1) - r_max, ...
80                             pi(1) + r_max, pi(2) - r_max, pi(2) + r_max)';
81
82         % if there are any agents...
83         if ~isempty(idx)
84             for aj = idx
85                 pj = data.floor(fi-1).agents(aj).p;
86                 if data.floor(fi-1).img_stairs_up(round(pj(1)),
87                                                         round(pj(2)))
88
89                     vj = data.floor(fi-1).agents(aj).v;
90                     rj = data.floor(fi-1).agents(aj).r;
91
92                     % vector pointing from j to i
93                     nij = (pi - pj) * data.meter_per_pixel;
94
95                     % distance of agents
96                     d = norm(nij);
97
98                     % normalized vector pointing from j to i
99                     nij = nij / d;
100                    % tangential direction
101                    tij = [-nij(2), nij(1)];
102
103                    % sum of radii
104                    rij = (ri + rj);
105
106                    % repulsive interaction forces
107                    if d < rij
108                        T1 = data.k*(rij - d);
109                        T2 = data.kappa*(rij - d)*dot((vj -
110                                                         vi),tij)*tij;
111                    else
112                        T1 = 0;
113                        T2 = 0;
114                    end
115
116                    F = (data.A * exp((rij - d)/data.B) + T1)*nij
117                        + T2;
118
119                    data.floor(fi).agents(ai).f = ...
120                        data.floor(fi).agents(ai).f + F;
121                    data.floor(fi-1).agents(aj).f = ...
122                        data.floor(fi-1).agents(aj).f - F;
123                end
124            end
125        end
end

```

```
end
```

Listing 1: addAgentRepulsiveForce.m

```
1 function data = addDesiredForce(data)
2 %ADDDESIREDFORCE add 'desired' force contribution (towards nearest exit or
3 %staircase)
4
5 for fi = 1:data.floor_count
6
7     for ai=1:length(data.floor(fi).agents)
8
9         % get agent's data
10        p = data.floor(fi).agents(ai).p;
11        m = data.floor(fi).agents(ai).m;
12        v0 = data.floor(fi).agents(ai).v0;
13        v = data.floor(fi).agents(ai).v;
14
15
16        % get direction towards nearest exit
17        ex = lerp2(data.floor(fi).img_dir_x, p(1), p(2));
18        ey = lerp2(data.floor(fi).img_dir_y, p(1), p(2));
19        e = [ex ey];
20
21        % get force
22        Fi = m * (v0*e - v)/data.tau;
23
24        % add force
25        data.floor(fi).agents(ai).f = data.floor(fi).agents(ai).f + Fi;
26    end
27 end
```

Listing 2: addDesiredForce.m

```
function data = addWallForce(data)
2 %ADDWALLFORCE adds wall's force contribution to each agent
3
4 for fi = 1:data.floor_count
5
6     for ai=1:length(data.floor(fi).agents)
7         % get agents data
8         p = data.floor(fi).agents(ai).p;
9         ri = data.floor(fi).agents(ai).r;
10        vi = data.floor(fi).agents(ai).v;
11
12        % get direction from nearest wall to agent
13        nx = lerp2(data.floor(fi).img_wall_dist_grad_x, p(1), p(2));
14        ny = lerp2(data.floor(fi).img_wall_dist_grad_y, p(1), p(2));
15
16        % get distance to nearest wall
```

```

18     diW = lerp2(data.floor(fi).img_wall_dist, p(1), p(2));
20     % get perpendicular and tangential unit vectors
21     niW = [ nx ny];
22     tiW = [-ny nx];
24     % calculate force
25     if diW < ri
26         T1 = data.k * (ri - diW);
27         T2 = data.kappa * (ri - diW) * dot(vi, tiW) * tiW;
28     else
29         T1 = 0;
30         T2 = 0;
31     end
32     Fi = (data.A * exp((ri-diW)/data.B) + T1)*niW - T2;
34     % add force to agent's current force
35     data.floor(fi).agents(ai).f = data.floor(fi).agents(ai).f + Fi;
36 end
end

```

Listing 3: addWallForce.m

```

function data = applyForcesAndMove(data)
2 %APPLYFORCESANDMOVE apply current forces to agents and move them using
  %the timestep and current velocity
4
  n_velocity_clamps = 0;
6
  % loop over all floors
8 for fi = 1:data.floor_count

10     % init logical arrays to indicate agents that change the floor or exit
11     % the simulation
12     floorchange = false(length(data.floor(fi).agents),1);
13     exited = false(length(data.floor(fi).agents),1);
14
15     % loop over all agents
16     for ai=1:length(data.floor(fi).agents)
17         % add current force contributions to velocity
18         v = data.floor(fi).agents(ai).v + data.dt * ...
19             data.floor(fi).agents(ai).f / data.floor(fi).agents(ai).m;
20
21         % clamp velocity
22         if norm(v) > data.v_max
23             v = v / norm(v) * data.v_max;
24             n_velocity_clamps = n_velocity_clamps + 1;
25         end
26

```

```

28     % get agent's new position
    newp = data.floor(fi).agents(ai).p + ...
        v * data.dt / data.meter_per_pixel;
30
31     % if the new position is inside a wall, remove perpendicular
32     % component of the agent's velocity
    if lerp2(data.floor(fi).img_wall_dist, newp(1), newp(2)) < ...
34         data.floor(fi).agents(ai).r
35
36         % get agent's position
        p = data.floor(fi).agents(ai).p;
38
39         % get wall distance gradient (which is off course perpendicular
40         % to the nearest wall)
        nx = lerp2(data.floor(fi).img_wall_dist_grad_x, p(1), p(2));
42        ny = lerp2(data.floor(fi).img_wall_dist_grad_y, p(1), p(2));
        n = [nx ny];
44
45        % project out perpendicular component of velocity vector
46        v = v - dot(n,v)/dot(n,n)*n;
47
48        % get agent's new position
        newp = data.floor(fi).agents(ai).p + ...
50            v * data.dt / data.meter_per_pixel;
    end
52
53    if data.floor(fi).img_wall(round(newp(1)), round(newp(2)))
54        newp = data.floor(fi).agents(ai).p;
55        v = [0 0];
56    end
57
58    % update agent's velocity and position
    data.floor(fi).agents(ai).v = v;
60    data.floor(fi).agents(ai).p = newp;
61
62    % reset forces for next timestep
    data.floor(fi).agents(ai).f = [0 0];
64
65    % check if agent reached a staircase and indicate floor change
66    if data.floor(fi).img_stairs_down(round(newp(1)), round(newp(2)))
        floorchange(ai) = 1;
68    end
69
70    % check if agent reached an exit
    if data.floor(fi).img_exit(round(newp(1)), round(newp(2)))
72        exited(ai) = 1;
        data.agents_exited = data.agents_exited + 1;
74    end
    end
76

```

```

78     % add appropriate agents to next lower floor
    if fi > 1
        data.floor(fi-1).agents = [data.floor(fi-1).agents ...
80                                data.floor(fi).agents(floorchange)];
    end
82
    % delete these and exited agents
    data.floor(fi).agents = data.floor(fi).agents(~(floorchange|exited));
84 end
86
    if n_velocity_clamps > 0
88         fprintf(['WARNING: clamped velocity of %d agents, ' ...
                'possible simulation instability.\n'], n_velocity_clamps);
90 end

```

Listing 4: applyForcesAndMove.m

```

function val = checkForIntersection(data, floor_idx, agent_idx)
2 % check an agent for an intersection with another agent or a wall
% the check is kept as simple as possible
4 %
% arguments:
6 % data            global data structure
% floor_idx        which floor to check
8 % agent_idx       which agent on that floor
% agent_new_pos    vector: [x,y], desired agent position to check
10 %
% return:
12 % 0              for no intersection
% 1              has an intersection with wall
14 % 2              with another agent

16 val = 0;

18 p = data.floor(floor_idx).agents(agent_idx).p;
r = data.floor(floor_idx).agents(agent_idx).r;
20
% check for agent intersection
22 for i=1:length(data.floor(floor_idx).agents)
    if i~=agent_idx
24         if norm(data.floor(floor_idx).agents(i).p-p)*data.meter_per_pixel
            ...
                <= r + data.floor(floor_idx).agents(i).r
26             val=2;
            return;
28         end
    end
30 end
32

```

```

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

```

Listing 5: checkForIntersection.m

```

1 mex 'fastSweeping.c'
  mex 'getNormalizedGradient.c'
3 mex 'lerp2.c'
  mex 'createRangeTree.c'
5 mex 'rangeQuery.c'

```

Listing 6: compileC.m

```

1 function data = initAgents(data)

3 % place agents randomly in desired spots, without overlapping

5

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

11

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

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

21     if ~isempty(x)
        floors_with_agents = floors_with_agents + 1;
23         for j=1:agent_count
            cur_agent = length(data.floor(i).agents) + 1;
25
                % init agent
27             data.floor(i).agents(cur_agent).r = getAgentRadius();
                data.floor(i).agents(cur_agent).v = [0, 0];
29             data.floor(i).agents(cur_agent).f = [0, 0];
                data.floor(i).agents(cur_agent).m = data.m;
31             data.floor(i).agents(cur_agent).v0 = data.v0;

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

```

```

37         idx = randi(length(x));
38         data.floor(i).agents(cur_agent).p = [y(idx), x(idx)];
39         if checkForIntersection(data, i, cur_agent) == 0
40             tries = -1; % leave the loop
41         end
42         tries = tries - 1;
43     end
44     if tries > -1
45         %remove the last agent
46         data.floor(i).agents = data.floor(i).agents(1:end-1);
47     end
48     data.total_agent_count = data.total_agent_count +
49         length(data.floor(i).agents);
50
51     if length(data.floor(i).agents) ~= agent_count
52         fprintf(['WARNING: could only place %d agents on floor %d ' ...
53             'instead of the desired %d.\n'], ...
54             length(data.floor(i).agents), i, agent_count);
55     end
56 end
57 if floors_with_agents==0
58     error('no spots to place agents!');
59 end
60
61 end

```

Listing 7: initAgents.m

```

1 function data = initEscapeRoutes(data)
2 %INITESCAPEROUTES Summary of this function goes here
3 % Detailed explanation goes here
4
5 for i=1:data.floor_count
6
7     boundary_data = zeros(size(data.floor(i).img_wall));
8     boundary_data(data.floor(i).img_wall) = 1;
9     %if (i == 1)
10         boundary_data(data.floor(i).img_exit) = -1;
11     %else
12         boundary_data(data.floor(i).img_stairs_down) = -1;
13     %end
14
15     exit_dist = fastSweeping(boundary_data) * data.meter_per_pixel;
16     [data.floor(i).img_dir_x, data.floor(i).img_dir_y] = ...
17         getNormalizedGradient(boundary_data, -exit_dist);
18 end

```

Listing 8: initEscapeRoutes.m

```

1 function data = initialize(config)
  % initialize the internal data from the config data
3 %
  % arguments:
5 %   config      data structure from loadConfig()
  %
7 % return:
  %   data        data structure: all internal data used for the main loop
9 %
  %               all internal data is stored in pixels NOT in meters
11
13 data = config;

15 %for convenience
  data.pixel_per_meter = 1/data.meter_per_pixel;
17
  fprintf('Init escape routes...\n');
19 data = initEscapeRoutes(data);
  fprintf('Init wall forces...\n');
21 data = initWallForces(data);
  fprintf('Init agents...\n');
23 data = initAgents(data);

25 % maximum influence of agents on each other

27 data.r_influence = data.pixel_per_meter * ...
    fzero(@(r) data.A * exp((2*data.r_max-r)/data.B) - 1e-4, data.r_max);
29
  fprintf('Init plots...\n');
31 %init the plots
  %exit plot
33 data.figure_exit=figure;
  hold on;
35 axis([0 data.duration 0 data.total_agent_count]);
  title(sprintf('agents that reached the exit (total agents: %i)',
    data.total_agent_count));
37
  %floors plot
39 data.figure_floors=figure;
  data.figure_floors_subplots_w = data.floor_count;
41 data.figure_floors_subplots_h = 2;
  for i=1:config.floor_count
43     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);
45     data.floor(i).building_plot = subplot(data.figure_floors_subplots_h,
        data.figure_floors_subplots_w ...

```

```

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

```

Listing 9: initialize.m

```

1 function data = initWallForces(data)
  %INITWALLFORCES init wall distance maps and gradient maps for each floor
3
  for i=1:data.floor_count
5
      % init boundary data for fast sweeping method
7      boundary_data = zeros(size(data.floor(i).img_wall));
      boundary_data(data.floor(i).img_wall) = -1;
9
      % get wall distance
11     wall_dist = fastSweeping(boundary_data) * data.meter_per_pixel;
      data.floor(i).img_wall_dist = wall_dist;
13
      % get normalized wall distance gradient
15     [data.floor(i).img_wall_dist_grad_x, ...
      data.floor(i).img_wall_dist_grad_y] = ...
17     getNormalizedGradient(boundary_data, wall_dist-data.meter_per_pixel);
  end

```

Listing 10: initWallForces.m

```

1 function config = loadConfig(config_file)
  % load the configuration file
3
  % arguments:
5  %   config_file      string, which configuration file to load
  %
7
9  % get the path from the config file -> to read the images
  config_path = fileparts(config_file);
11 if strcmp(config_path, '') == 1
      config_path = '.';
13 end
15 fid = fopen(config_file);
  input = textscan(fid, '%s=%s');
17 fclose(fid);
19 keynames = input{1};
  values = input{2};
21
  %convert numerical values from string to double
23 v = str2double(values);
  idx = ~isnan(v);

```

```

25 values(idx) = num2cell(v(idx));
27 config = cell2struct(values, keynames);
29
30 % read the images
31 for i=1:config.floor_count
32
33     %building structure
34     file = config.(sprintf('floor_%d_build', i));
35     file_name = [config_path '/' file];
36     img_build = imread(file_name);
37
38     % decode images
39     config.floor(i).img_wall = (img_build(:, :, 1) == 0 ...
40                                & img_build(:, :, 2) == 0 ...
41                                & img_build(:, :, 3) == 0);
42
43     config.floor(i).img_spawn = (img_build(:, :, 1) == 255 ...
44                                  & img_build(:, :, 2) == 0 ...
45                                  & img_build(:, :, 3) == 255);
46
47     config.floor(i).img_exit = (img_build(:, :, 1) == 0 ...
48                                 & img_build(:, :, 2) == 255 ...
49                                 & img_build(:, :, 3) == 0);
50
51     config.floor(i).img_stairs_up = (img_build(:, :, 1) == 255 ...
52                                      & img_build(:, :, 2) == 0 ...
53                                      & img_build(:, :, 3) == 0);
54
55     config.floor(i).img_stairs_down = (img_build(:, :, 1) == 0 ...
56                                         & img_build(:, :, 2) == 0 ...
57                                         & img_build(:, :, 3) == 255);
58
59     %init the plot image here, because this won't change
60     config.floor(i).img_plot = 5*config.floor(i).img_wall ...
61     + 4*config.floor(i).img_stairs_up ...
62     + 3*config.floor(i).img_stairs_down ...
63     + 2*config.floor(i).img_exit ...
64     + 1*config.floor(i).img_spawn;
65     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
66                         0 0];
67 end

```

Listing 11: loadConfig.m

```

1 function plotAgentsPerFloor(data, floor_idx)
2 %plot time vs agents on floor
3
4 h = subplot(data.floor(floor_idx).agents_on_floor_plot);

```

```

5  set(h, 'position',[0.05+(data.floor_count -
    floor_idx)/(data.figure_floors_subplots_w+0.2), ...
7    0.05, 1/(data.figure_floors_subplots_w*1.2), 0.3-0.05 ]);

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

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

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

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

```

Listing 12: plotAgentsPerFloor.m

```

function plotExitedAgents(data)
2 %plot time vs exited agents

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

```

Listing 13: plotExitedAgents.m

```

function plotFloor(data, floor_idx)
2 %PLOT FLOOR plot floor

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

6 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 ]);

8 hold off;

10 % the building image
    imagesc(data.floor(floor_idx).img_plot);
12 hold on;

14 %plot options
    colormap(data.color_map);
16 axis equal;
    axis manual; %do not change axis on window resize

18 set(h, 'Visible', 'off')
20 %title(sprintf('floor %i', floor_idx));

```

```

22 % plot agents
23 if ~isempty(data.floor(floor_idx).agents)
24     ang = [linspace(0,2*pi, 10) nan]';
25     rmul = [cos(ang) sin(ang)] * data.pixel_per_meter;
26     draw = cell2mat(arrayfun(@(a) repmat(a.p,length(ang),1) + a.r*rmul, ...
27         data.floor(floor_idx).agents, 'UniformOutput', false)'));
28     line(draw(:,2), draw(:,1), 'Color', 'r');
29 end
30
31 % old drawing code...
32 % ang = linspace(0,2*pi, 10);
33 % cosang = cos(ang);
34 % sinang = sin(ang);
35 % for i=1:length(data.floor(floor_idx).agents)
36 %     p = data.floor(floor_idx).agents(i).p;
37 %     r = data.floor(floor_idx).agents(i).r * data.pixel_per_meter;
38 %     %r = norm(agent.v);
39 %     xp = r * cosang;
40 %     yp = r * sinang;
41 %     plot(p(2) + yp, p(1) + xp, 'Color', 'r');
42 %     %text(agent.pos(2),agent.pos(1),int2str(i));
43 % end
44
45 hold off;
46 end

```

Listing 14: plotFloor.m

```

function simulate(config_file)
2 % run this to start the simulation
3
4 if nargin==0
5     config_file='../data/config1.conf';
6 end
7
8 fprintf('Load config file...\n');
9 config = loadConfig(config_file);
10
11 data = initialize(config);
12
13
14 data.time = 0;
15 frame = 0;
16 fprintf('Start simulation...\n');
17
18 while (data.time < data.duration)
19     tstart=tic;
20     data = addDesiredForce(data);
21     data = addWallForce(data);
22     data = addAgentRepulsiveForce(data);

```

```

24     data = applyForcesAndMove(data);
26     % do the plotting
27     set(0, 'CurrentFigure', data.figure_floors);
28     for floor=1:data.floor_count
29         plotAgentsPerFloor(data, floor);
30         plotFloor(data, floor);
31     end
32     if data.save_frames==1
33         print('-depsc2', sprintf('frames/%s_%04i.eps', ...
34             data.frame_basename, frame), data.figure_floors);
35     end
36     set(0, 'CurrentFigure', data.figure_exit);
37     plotExitedAgents(data);
38
40     % print mean/median velocity of agents on each floor
41     % for fi = 1:data.floor_count
42     %     avgv = arrayfun(@(agent) norm(agent.v), data.floor(fi).agents);
43     %     fprintf('Mean/median velocity on floor %i: %g/%g m/s\n', fi,
44     %         mean(avgv), median(avgv));
45     % end
46
47     if (data.time + data.dt > data.duration)
48         data.dt = data.duration - data.time;
49         data.time = data.duration;
50     else
51         data.time = data.time + data.dt;
52     end
53
54     if data.agents_exited == data.total_agent_count
55         fprintf('All agents are now saved (or are they?). Time: %.2f
56             sec\n', data.time);
57         fprintf('Total Agents: %i\n', data.total_agent_count);
58
59         print('-depsc2', sprintf('frames/exited_agents_%s.eps', ...
60             data.frame_basename), data.figure_floors);
61         break;
62     end
63
64     telapsed = toc(tstart);
65     pause(max(data.dt - telapsed, 0.01));
66     fprintf('Frame %i done (took %.3fs; %.3fs out of %.3gs simulated).\n',
67         frame, telapsed, data.time, data.duration);
68     frame = frame + 1;
69 end

```

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

Listing 15: simulate.m

9.1.2 C code

```
1
#include <mex.h>
3 include <string.h>

5 include "tree_build.c"
include "tree_query.c"
7 include "tree_free.c"

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

17     if (nlhs < 1)
        return;

19

    points = (point_t*) mxGetPr(prhs[0]);
21     m = mxGetM(prhs[0]);
    n = mxGetN(prhs[0]);
23

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

27     tree = build_tree(points, n);

29     plhs[0] = mxCreateNumericMatrix(tree->first_free + sizeof(int), 1,
        mxUINT8_CLASS, mxREAL);
    data = (uchar*) mxGetPr(plhs[0]);
31

    root_index = (int*) data;
33     *root_index = tree->root_index;
    memcpy(data + sizeof(int), tree->data, tree->first_free);
35

    free_tree(tree);
37 }
```

Listing 16: createRangeTree.c

```

1  #include "mex.h"

3  #include <math.h>

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

15
16 #define SOLVE_AND_UPDATE    udiff = uxmin - uymin; \
17                             if (ABS(udiff) >= 1.0) \
18                             { \
19                                 up = MIN(uxmin, uymin) + 1.0; \
20                             } \
21                             else \
22                             { \
23                                 up = (uxmin + uymin + sqrt(2.0 - udiff *
                                   udiff)) / 2.0; \
24                                 up = MIN(uij, up); \
25                             } \
26                             err_loc = MAX(ABS(uij - up), err_loc); \
27                             u[ij] = up;

29 #define I_STEP(_uxmin, _uymin, _st) if (boundary[ij] == 0.0) \
30                                     { \
31                                         uij = un; \
32                                         un = u[ij + _st]; \
33                                         uxmin = _uxmin; \
34                                         uymin = _uymin; \
35                                         SOLVE_AND_UPDATE \
36                                         ij += _st; \
37                                     } \
38                                     else \
39                                     { \
40                                         up = un; \
41                                         un = u[ij + _st]; \
42                                         ij += _st; \
43                                     }

45
47 #define I_STEP_UP(_uxmin, _uymin)    I_STEP(_uxmin, _uymin, 1)

```

```

49 #define I_STEP_DOWN(_uxmin, _uymin) I_STEP(_uxmin, _uymin, -1)

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

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

59

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

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

69     *err = 0.0;
   err_loc = 0.0;

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

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

83     /* i = m-1, j = 0 */
   I_STEP_UP(UX_PREV, UY_RIGHT)
85

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

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

93         I_STEP_UP(UX_PREV, UY_BOTH)
95     }

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

```

```

I_STEP_UP(UX_NEXT, UY_LEFT)
99
/* i = 1->m2, j = n-1 */
101 for (i = 1; i <= m2; ++i)
    I_STEP_UP(UX_BOTH, UY_LEFT)
103
/* i = m-1, j = n-1 */
105 I_STEP_UP(UX_PREV, UY_LEFT)
107
/* sweep 2 */
109 /* i = 0, j = n-1 */
    ij = (n-1)*m;
111    un = u[ij];
    I_STEP_UP(UX_NEXT, UY_LEFT)
113
/* i = 1->m2, j = n-1 */
115 for (i = 1; i <= m2; ++i)
    I_STEP_UP(UX_BOTH, UY_LEFT)
117
/* i = m-1, j = n-1 */
119 I_STEP_UP(UX_PREV, UY_LEFT)
121
/* i = 0->m-1, j = n2->1 */
123 for (j = n2; j >= 1; --j)
{
    ij = j*m;
125    un = u[ij];
    I_STEP_UP(UX_NEXT, UY_BOTH)
127
    for (i = 1; i <= m2; ++i)
129        I_STEP_UP(UX_BOTH, UY_BOTH)
131
    I_STEP_UP(UX_PREV, UY_BOTH)
}
133
/* i = 0, j = 0 */
135 ij = 0;
    un = u[ij];
137 I_STEP_UP(UX_NEXT, UY_RIGHT)
139
/* i = 1->m2, j = 0 */
141 for (i = 1; i <= m2; ++i)
    I_STEP_UP(UX_BOTH, UY_RIGHT)
143
/* i = m-1, j = 0 */
    I_STEP_UP(UX_PREV, UY_RIGHT)
145
/* sweep 3 */
147 /* i = m-1, j = n-1 */

```

```

149     ij = m*n - 1;
150     un = u[ij];
151     I_STEP_DOWN(UX_NEXT, UY_LEFT)
152
153     /* i = m2->1, j = n-1 */
154     for (i = m2; i >= 1; --i)
155         I_STEP_DOWN(UX_BOTH, UY_LEFT)
156
157     /* i = 0, j = n-1 */
158     I_STEP_DOWN(UX_PREV, UY_LEFT)
159
160     /* i = m-1->0, j = n2->1 */
161     for (j = n2; j >= 1; --j)
162     {
163         I_STEP_DOWN(UX_NEXT, UY_BOTH)
164
165         for (i = m2; i >= 1; --i)
166             I_STEP_DOWN(UX_BOTH, UY_BOTH)
167
168         I_STEP_DOWN(UX_PREV, UY_BOTH)
169     }
170
171     /* i = m-1, j = 0 */
172     I_STEP_DOWN(UX_NEXT, UY_RIGHT)
173
174     /* i = m2->1, j = 0 */
175     for (i = m2; i >= 1; --i)
176         I_STEP_DOWN(UX_BOTH, UY_RIGHT)
177
178     /* i = 0, j = 0 */
179     I_STEP_DOWN(UX_PREV, UY_RIGHT)
180
181     /* sweep 4 */
182     /* i = m-1, j = 0 */
183     ij = m - 1;
184     un = u[ij];
185     I_STEP_DOWN(UX_NEXT, UY_RIGHT)
186
187     /* i = m2->1, j = 0 */
188     for (i = m2; i >= 1; --i)
189         I_STEP_DOWN(UX_BOTH, UY_RIGHT)
190
191     /* i = 0, j = 0 */
192     I_STEP_DOWN(UX_PREV, UY_RIGHT)
193
194     /* i = m-1->0, j = 1->n2 */
195     for (j = 1; j <= n2; ++j)
196     {
197         ij = m - 1 + j*m;
198         un = u[ij];

```

```

199         I_STEP_DOWN(UX_NEXT, UY_BOTH)
201         for (i = m2; i >= 1; --i)
202             I_STEP_DOWN(UX_BOTH, UY_BOTH)
203         I_STEP_DOWN(UX_PREV, UY_BOTH)
204     }
205
206     /* i = m-1, j = n-1 */
207     ij = m*n - 1;
208     un = u[ij];
209     I_STEP_DOWN(UX_NEXT, UY_LEFT)
210
211     /* i = m2->1, j = n-1 */
212     for (i = m2; i >= 1; --i)
213         I_STEP_DOWN(UX_BOTH, UY_LEFT)
214
215     /* i = 0, j = n-1 */
216     I_STEP_DOWN(UX_PREV, UY_LEFT)
217
218     *err = MAX(*err, err_loc);
219 }
220
221 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
222                  *prhs[])
223 {
224     double *u, *boundary;
225     double tol, err;
226     int m, n, entries, max_iter, i;
227
228     /* Check number of outputs */
229     if (nlhs < 1)
230         return;
231     else if (nlhs > 1)
232         mexErrMsgTxt("At most 1 output argument needed.");
233
234     /* Get inputs */
235     if (nrhs < 1)
236         mexErrMsgTxt("At least 1 input argument needed.");
237     else if (nrhs > 3)
238         mexErrMsgTxt("At most 3 input arguments used.");
239
240
241     /* Get boundary */
242     if (!mxIsDouble(prhs[0]) || mxIsClass(prhs[0], "sparse"))
243         mexErrMsgTxt("Boundary field needs to be a full double precision
244                        matrix.");
245
246     boundary = mxGetPr(prhs[0]);

```

```

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

249
    /* Get max iterations */
251 if (nrhs >= 2)
    {
253     if (!mxIsDouble(prhs[1]) || mxGetM(prhs[1]) != 1 ||
        mxGetN(prhs[1]) != 1)
        mexErrMsgTxt("Maximum iteration needs to be positive
            integer.");
255     max_iter = (int) *mxGetPr(prhs[1]);
        if (max_iter <= 0)
257         mexErrMsgTxt("Maximum iteration needs to be positive
            integer.");
    }
259 else
    max_iter = 20;

261
    /* Get tolerance */
263 if (nrhs >= 3)
    {
265     if (!mxIsDouble(prhs[2]) || mxGetM(prhs[2]) != 1 ||
        mxGetN(prhs[2]) != 1)
        mexErrMsgTxt("Tolerance needs to be a positive real number.");
267     tol = *mxGetPr(prhs[2]);
        if (tol < 0)
269         mexErrMsgTxt("Tolerance needs to be a positive real number.");
    }
271 else
    tol = 1e-12;

273
    /* create and init output (distance) matrix */
275 plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
277 u = mxGetPr(plhs[0]);

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

281
    err = 0.0;
283 i = 0;
    do
285     {
        iteration(u, boundary, m, n, &err);
287         ++i;
    } while (err > tol && i < max_iter);
289 }

```

Listing 17: fastSweeping.c

```

1  #include "mex.h"

3  #include <math.h>

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

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

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

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

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

29

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

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

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

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

```

```

47  /* create and init output (gradient) matrices */
48  plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
49  plhs[1] = mxCreateDoubleMatrix(m, n, mxREAL);
50  xgrad = mxGetPr(plhs[0]);
51  ygrad = mxGetPr(plhs[1]);
52
53
54
55  for (j = 0; j < n; ++j)
56      for (i = 0; i < m; ++i)
57          if (INTERIOR(i,j))
58              {
59                  if (i > 0)
60                      dxm = INTERIOR(i-1,j) ? DIST(i-1,j) : DIST(i,j);
61                  else
62                      dxm = DIST(i,j);
63
64                  if (i < m-1)
65                      dxp = INTERIOR(i+1,j) ? DIST(i+1,j) : DIST(i,j);
66                  else
67                      dxp = DIST(i,j);
68
69                  if (j > 0)
70                      dym = INTERIOR(i,j-1) ? DIST(i,j-1) : DIST(i,j);
71                  else
72                      dym = DIST(i,j);
73
74                  if (j < n-1)
75                      dyp = INTERIOR(i,j+1) ? DIST(i,j+1) : DIST(i,j);
76                  else
77                      dyp = DIST(i,j);
78
79                  XGRAD(i, j) = (dxp - dxm) / 2.0;
80                  YGRAD(i, j) = (dyp - dym) / 2.0;
81                  nrm = sqrt(XGRAD(i, j)*XGRAD(i, j) + YGRAD(i, j)*YGRAD(i,
82                      j));
83                  if (nrm > 1e-12)
84                      {
85                          XGRAD(i, j) /= nrm;
86                          YGRAD(i, j) /= nrm;
87                      }
88                  else
89                      {
90                          XGRAD(i, j) = 0.0;
91                          YGRAD(i, j) = 0.0;
92                      }
93
94  for (j = 0; j < n; ++j)

```

```

95     for (i = 0; i < m; ++i)
96         if (!INTERIOR(i, j))
97             {
98                 xns = 0.0;
99                 yns = 0.0;
100                 nn = 0;
101                 if (i > 0 && INTERIOR(i-1,j))
102                     {
103                         xns += XGRAD(i-1,j);
104                         yns += YGRAD(i-1,j);
105                         ++nn;
106                     }
107                 if (i < m-1 && INTERIOR(i+1,j))
108                     {
109                         xns += XGRAD(i+1,j);
110                         yns += YGRAD(i+1,j);
111                         ++nn;
112                     }
113                 if (j > 0 && INTERIOR(i,j-1))
114                     {
115                         xns += XGRAD(i,j-1);
116                         yns += YGRAD(i,j-1);
117                         ++nn;
118                     }
119                 if (j < n-1 && INTERIOR(i,j+1))
120                     {
121                         xns += XGRAD(i,j+1);
122                         yns += YGRAD(i,j+1);
123                         ++nn;
124                     }
125
126                 if (nn > 0)
127                     {
128                         XGRAD(i, j) = xns / nn;
129                         YGRAD(i, j) = yns / nn;
130                     }
131             }
132 }

```

Listing 18: getNormalizedGradient.c

```

2  #include <mex.h>

4  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
    *prhs[])
5  {
6      int m, n, i0, i1, j0, j1, idx00;
7      double *data, *out, x, y, wx0, wy0, wx1, wy1;
8      double d00, d01, d10, d11;

```

```

10     if (nlhs < 1)
11         return;
12     else if (nlhs > 1)
13         mexErrMsgTxt("Exactly one output argument needed.");
14
15     if (nrhs != 3)
16         mexErrMsgTxt("Exactly three input arguments needed.");
17
18     m = mxGetM(prhs[0]);
19     n = mxGetN(prhs[0]);
20     data = mxGetPr(prhs[0]);
21     x = *mxGetPr(prhs[1]) - 1;
22     y = *mxGetPr(prhs[2]) - 1;
23
24     plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
25     out = mxGetPr(plhs[0]);
26
27     x = x < 0 ? 0 : x > m - 1 ? m - 1 : x;
28     y = y < 0 ? 0 : y > n - 1 ? n - 1 : y;
29     i0 = (int) x;
30     j0 = (int) y;
31     i1 = i0 + 1;
32     i1 = i1 > m - 1 ? m - 1 : i1;
33     j1 = j0 + 1;
34     j1 = j1 > n - 1 ? n - 1 : j1;
35
36     idx00 = i0 + m * j0;
37     d00 = data[idx00];
38     d01 = data[idx00 + m];
39     d10 = data[idx00 + 1];
40     d11 = data[idx00 + m + 1];
41
42     wx1 = x - i0;
43     wy1 = y - j0;
44     wx0 = 1.0 - wx1;
45     wy0 = 1.0 - wy1;
46
47     *out = wx0 * (wy0 * d00 + wy1 * d01) + wx1 * (wy0 * d10 + wy1 * d11);
48 }

```

Listing 19: lerp2.c

```

2 #include <mex.h>
3 #include <string.h>
4
5 #include "tree_build.c"
6 #include "tree_query.c"
7 #include "tree_free.c"

```

```

8 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
    *prhs[])
10 {
    tree_t *tree;
12     int n, i;
    int *point_idx, *root_idx;
14     range_t *range;
    uchar *data;
16
    if (nlhs != 1)
18         mexErrMsgTxt("...");

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

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

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

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

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

40     plhs[0] = mxCreateNumericMatrix(range->n, 1, mxUINT32_CLASS, mxREAL);
    point_idx = (int*) mxGetPr(plhs[0]);
42
    for (i = 0; i < range->n; ++i)
44         point_idx[i] = range->point_idx[i] + 1;

46     free_range(range);
    free(tree);
48 }

```

Listing 20: rangeQuery.c

```

1 #ifndef TREE_H
2 #define TREE_H
4 #include "tree_types.h"

```

```

6  /* build a 2D range tree using the given points */
   tree_t* build_tree(point_t *points, int n);
8
   /* query a range tree */
10 range_t* range_query(tree_t *tree, double x_min, double x_max, double
    y_min, double y_max);
12
   /* free memory of a tree */
   void free_tree(tree_t *tree);
14
   /* free memory of a range */
16 void free_range(range_t *range);
18 #endif

```

Listing 21: tree.h

```

   #ifndef TREE_BUILD_H
2  #define TREE_BUILD_H

4  #include "tree.h"

6  /* 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);
8
   /* double comparison for qsort */
10 int compare_double(const void *a, const void *b);

12 /* 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);
14 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);
16
   #endif

```

Listing 22: tree_build.h

```

1
   #include <assert.h>
3  #include <stdio.h>
   #include <stdlib.h>
5  #include <string.h>

7  #include "tree_build.h"

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

```

```

11 {
12     int nx, i, j, *point_idx;
13     double *x_vals;
14     tree_t *tree;
15
16     /* get x coordinate values of all points */
17     x_vals = (double*) malloc(n * sizeof(double));
18     for (i = 0; i < n; ++i)
19         x_vals[i] = points[i].x;
20
21     /* sort x values */
22     qsort(x_vals, n, sizeof(double), compare_double);
23
24     /* count number of unique x values */
25     nx = 1;
26     for (i = 1; i < n; ++i)
27         if (x_vals[i] != x_vals[i - 1])
28             ++nx;
29
30     /* remove duplicates */
31     j = 0;
32     for (i = 0; i < nx; ++i)
33     {
34         x_vals[i] = x_vals[j];
35         while (x_vals[i] == x_vals[j])
36             ++j;
37     }
38
39     /* create an index array */
40     point_idx = (int*) malloc(n * sizeof(int));
41     for (i = 0; i < n; ++i)
42         point_idx[i] = i;
43
44     /* sort index array by y coordinates of associated points */
45     index_sort_y(points, point_idx, n);
46
47     /* init tree */
48     tree = (tree_t*) malloc(sizeof(tree_t));
49     tree->total_size = n * sizeof(point_t);
50     tree->data = (uchar*) malloc(tree->total_size);
51
52     /* copy point coordinates to tree data */
53     memcpy(tree->data, points, n * sizeof(point_t));
54
55     /* set first free byte and root index of the tree */
56     tree->first_free = n * sizeof(point_t);
57     tree->root_index = tree->first_free;
58
59     /* recursively build tree */
60     build_subtree(tree, x_vals, nx, points, point_idx, n);

```

```

61     /* free temporaries */
    free(x_vals);
63     return tree;
}
65

67 int build_subtree(tree_t *tree, double *x_vals, const int nx, point_t
    *points, int *point_idx, const int np)
{
69     int i, j, k, nx_left, np_left, node_size, right_idx;
    node_t *node;
71     int *node_point_idx, *point_idx_left, *point_idx_right, node_idx;
    uchar *new_data;
73
    assert(nx > 0);
75     assert(np > 0);

77     /* allocate memory in the tree data structure */
    node_size = sizeof(node_t) + np * sizeof(int);
79     while (tree->first_free + node_size > tree->total_size)
    {
81         tree->total_size <= 1;
        new_data = (uchar*) malloc(tree->total_size * sizeof(uchar));
83         for (i = 0; i < tree->first_free; ++i)
            new_data[i] = tree->data[i];
85         free(tree->data);
        tree->data = new_data;
87     }
    node_idx = tree->first_free;
89     node = (node_t*) &tree->data[node_idx];
    tree->first_free += node_size;
91

    /* set number of stored points */
93     node->np = np;
    node_point_idx = (int*) (node + 1);
95

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

99     /* create child node if there is only one x value left, otherwise
        create interior node */
    if (nx == 1)
101     {
        node->right_idx = -1;
103         node->x_val = x_vals[0];
    }
105     else
    {
107         /* get median of x values */

```

```

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

111     /* count points belonging to the left child */
    np_left = 0;
113     for (i = 0; i < np; ++i)
    {
115         if (points[point_idx[i]].x <= node->x_val)
            ++np_left;
117     }

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

123     /* fill index arrays */
    j = 0;
125     k = 0;
    for (i = 0; i < np; ++i)
    {
127         if (points[point_idx[i]].x <= node->x_val)
            point_idx_left[j++] = point_idx[i];
129         else
            point_idx_right[k++] = point_idx[i];
131     }
133
135     /* free current node's temporary index array */
    free(point_idx);

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

141     /* 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) */
143     node = (node_t*) &tree->data[node_idx];
    /* update node's right child index */
145     node->right_idx = right_idx;
    }

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

151 int compare_double(const void *a, const void *b)
153 {
    double ad, bd;

```

```

155     ad = *((double*) a);
156     bd = *((double*) b);
157     return (ad < bd) ? -1 : (ad > bd) ? 1 : 0;
158 }
159
160 void index_sort_y(const point_t *points, int *point_idx, const int n)
161 {
162     index_quicksort_y(points, point_idx, 0, n - 1);
163 }
164
165 void index_quicksort_y(const point_t *points, int *point_idx, int l, int r)
166 {
167     int p;
168
169     /* quicksort point indices by point y coordinates, don't touch point
170        array itself */
171     while (l < r)
172     {
173         p = index_partition_y(points, point_idx, l, r);
174         if (r - p > p - l)
175         {
176             index_quicksort_y(points, point_idx, l, p - 1);
177             l = p + 1;
178         }
179         else
180         {
181             index_quicksort_y(points, point_idx, p + 1, r);
182             r = p - 1;
183         }
184     }
185 }
186
187 int index_partition_y(const point_t *points, int *point_idx, int l, int r)
188 {
189     int i, j, tmp;
190     double pivot;
191
192     /* rightmost element is pivot */
193     i = l;
194     j = r - 1;
195     pivot = points[point_idx[r]].y;
196
197     /* quicksort partition */
198     do
199     {
200         while (points[point_idx[i]].y <= pivot && i < r)
201             ++i;
202
203         while (points[point_idx[j]].y >= pivot && j > l)

```

```

205     if (i < j)
206     {
207         tmp = point_idx[i];
208         point_idx[i] = point_idx[j];
209         point_idx[j] = tmp;
210     }
211 } while (i < j);

213 if (points[point_idx[i]].y > pivot)
214 {
215     tmp = point_idx[i];
216     point_idx[i] = point_idx[r];
217     point_idx[r] = tmp;
218 }

219 return i;
221 }

```

Listing 23: tree_build.c

```

1  #include <stdlib.h>
3
5  #include "tree.h"
7
9  void free_tree(tree_t *tree)
10 {
11     free(tree->data);
12 }
13
14 void free_range(range_t *range)
15 {
16     free(range->point_idx);
17 }

```

Listing 24: tree_free.c

```

1  #ifndef TREE_QUERY_H
2  #define TREE_QUERY_H

4  #include "tree_types.h"

6  /* appends a point-index to a range, icnreases range capacity if needed */
7  void range_append(range_t *range, int idx);
8
9  /* finds the split node of a given query */
10 int find_split_node(tree_t *tree, int node_idx, range_t *range);
11
12 /* query the points of a node by a given range by y-coordinate */

```

```

void range_query_y(tree_t *tree, int node_idx, range_t *range);
14
#endif

```

Listing 25: tree_query.h

```

1
#include <assert.h>
3
#include <stdio.h>
#include <stdlib.h>
5
#include "tree_query.h"
7
#define LEFT_CHILD_IDX(node_idx, node) (node_idx) + sizeof(node_t) +
    (node)->np * sizeof(int)
9
#define RIGHT_CHILD_IDX(node_idx, node) (node)->right_idx
#define NODE_FROM_IDX(tree, node_idx) (node_t*) &(tree)->data[node_idx];
11
range_t* range_query(tree_t *tree, double x_min, double x_max, double
    y_min, double y_max)
13 {
    int split_node_idx, node_idx;
15
    node_t *split_node, *node;
    range_t *range;
17

    /* init range */
19
    range = (range_t*) malloc(sizeof(range_t));
    range->min.x = x_min;
21
    range->max.x = x_max;
    range->min.y = y_min;
23
    range->max.y = y_max;
    range->n = 0;
25
    range->total_size = 16;
    range->point_idx = (int*) malloc(range->total_size * sizeof(int));
27

    /* find split node */
29
    split_node_idx = find_split_node(tree, tree->root_index, range);
    split_node = NODE_FROM_IDX(tree, split_node_idx);
31

    /* if split node is a child */
33
    if (split_node->right_idx == -1)
    {
35
        range_query_y(tree, split_node_idx, range);
        return range;
37
    }

39
    /* follow left path of the split node */
    node_idx = LEFT_CHILD_IDX(split_node_idx, split_node);
41
    node = NODE_FROM_IDX(tree, node_idx);
    while (node->right_idx != -1)

```

```

43     {
44         if (range->min.x <= node->x_val)
45         {
46             range_query_y(tree, RIGHT_CHILD_IDX(node_idx, node), range);
47             node_idx = LEFT_CHILD_IDX(node_idx, node);
48         }
49         else
50             node_idx = RIGHT_CHILD_IDX(node_idx, node);
51         node = NODE_FROM_IDX(tree, node_idx);
52     }
53     range_query_y(tree, node_idx, range);
54
55     /* follow right path of the split node */
56     node_idx = split_node->right_idx;
57     node = NODE_FROM_IDX(tree, node_idx);
58     while (node->right_idx != -1)
59     {
60         if (range->max.x > node->x_val)
61         {
62             range_query_y(tree, LEFT_CHILD_IDX(node_idx, node), range);
63             node_idx = RIGHT_CHILD_IDX(node_idx, node);
64         }
65         else
66             node_idx = LEFT_CHILD_IDX(node_idx, node);
67         node = NODE_FROM_IDX(tree, node_idx);
68     }
69     range_query_y(tree, node_idx, range);
70
71     return range;
72 }
73
74 void range_append(range_t *range, int idx)
75 {
76     int *new_point_idx;
77     int new_size, i;
78
79     /* just append if there is enough place, otherwise double capacity and
80      append */
81     if (range->n < range->total_size)
82         range->point_idx[range->n++] = idx;
83     else
84     {
85         new_size = range->total_size << 1;
86         new_point_idx = (int*) malloc(new_size * sizeof(int));
87         for (i = 0; i < range->n; ++i)
88             new_point_idx[i] = range->point_idx[i];
89         new_point_idx[range->n++] = idx;
90         free(range->point_idx);
91         range->point_idx = new_point_idx;
92         range->total_size = new_size;

```

```

    }
93 }

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

99     node = (node_t*) &tree->data[node_idx];
    /* check if this node is the split node */
101     if (range->min.x <= node->x_val && range->max.x > node->x_val)
        return node_idx;
103
    /* ...or if it is a child (and therefor the split node) */
105     if (node->right_idx == -1)
        return node_idx;
107
    /* otherwise search the split node at the left or right of the current
       node */
109     if (range->max.x <= node->x_val)
        return find_split_node(tree, LEFT_CHILD_IDX(node_idx, node),
                               range);
111     else
        return find_split_node(tree, RIGHT_CHILD_IDX(node_idx, node),
                               range);
113 }

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

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

127     /* 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)
129         return;

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

```

```

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

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

159     /* append found points to the range */
    for (i = start; i <= end; ++i)
161         if (points[point_idx[i]].x <= range->max.x)
            range_append(range, point_idx[i]);
163 }

```

Listing 26: tree_query.c

```

1 #ifndef TREE_TYPES_H
2 #define TREE_TYPES_H

4 typedef unsigned char uchar;

6 /* 2D point */
7 typedef struct
8 {
9     double x;
10    double y;
11 } point_t;

12 /* tree */
13 typedef struct
14 {
15     /* byte data array with points and nodes */
16     uchar *data;

18     /* index of first unused byte */
19     int first_free;
20 }

```

```

22     /* total number of allocated bytes */
23     int total_size;
24
25     /* index of the root node in the data array*/
26     int root_index;
27 } tree_t;
28
29 /* node */
30 typedef struct
31 {
32     /* index of the right child node (left child follows directly after
33        current) */
34     int right_idx;
35
36     /* number of associated points */
37     int np;
38
39     /* associated x-coordinate value */
40     double x_val;
41 } node_t;
42
43 /* range */
44 typedef struct
45 {
46     /* point index list */
47     int *point_idx;
48
49     /* number of saved indices */
50     int n;
51
52     /* total number of allocated indices */
53     int total_size;
54
55     /* minimum range point */
56     point_t min;
57
58     /* maximum range point */
59     point_t max;
60 } range_t;
61
62 #endif

```

Listing 27: tree.types.h