ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Project Report

Simulation and optimization of pedestrian dynamics at an intersection

Luca Forni & Fabian Jenelten

Zürich
Spring semester 2014

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SIMULATION AND OPTIMIZATION OF PEDESTRIAN DYNAMICS AT AN INTERSECTION

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
LUCA	FORNI
FABIAN	JENELTEN

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Agreement for free-download

We hereby agree to make our source code of 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.

Luca Forni   Fabian Jenelten

Table of content

1. Abstract

In our simulation we want to test and analyze the dynamics of a crowded intersection and determine if the flow of people through this intersection can be maximized by applying rules that already exist in traffic regulations for vehicles on the road.
We managed to apply some ground rules to each pedestrian so that it can look around to judge the environmental situation dictated by other pedestrians around him and take action , i.e. not colliding with other people on his way to his specific target finding a way through the crowd to his destination point.
It can be observed that this individually based behavior can become problematic if the broadness of the road shrinks to a critical point or the density of the pedestrians arises to a point where jams are formed and groups of people get stuck decreasing furthermore the fluidity of future passages by other pedestrians through that jam. We think that applying some non-individualistic ground rules can significantly improve the dynamics of the crowd and the speed at which a person can cross the intersection.

2. Individual contributions

Fabian Jenelten: Discretization of the model
Synthesizing of ground functions and principles
Implementation model and functions with Matlab

Luca Forni: Discretization of the model
Plotting result graphics and optimization of cases with Matlab
Writing of the Paper

3. Introduction and motivations

3.1 Motivation

Everybody who has been at a large scale event taking place on a big areal with lots of people knows the struggle of reaching a destination walking through a crowd practically moving in every direction. Every time we walk at an intersection with lots of people we tend to deviate from our originally intended path due to indirect interaction with others, trying to constantly adjust our walking direction to overtake slower people in front of us, not come to contact with surrounding people and avoid oncoming pedestrians without knowing ex ante where they are headed. When everybody decides for itself which path he wants to follow the overall picture gets lost and we unintentionally form jams, conglomerate on one side of the road with plenty of room on the other and take non-optimal decisions regarding our reaching of the target. Our projects' objective is to optimize this process and find a solution that everybody can take advantage of.

Fig 1: One of the world's most heavily used pedestrian scrambles, at Hachikō Square in Shibuya, Tokyo. Notice the density distribution and how local jams begin to form.

3.2 Fundamental questions

Where else can we see individuals trying to reach their own target and have no choice that passing by one another. The first example that comes to mind is vehicle traffic in a city. We can observe that cars are much bigger than people, have much less controllability, their size in relation to the broadness of the street is a lot bigger than that of people related to a walking path, they move at greater speed and have fewer streets they can run on. Soon the question arises, why is this traffic much more organized and fluent compared to pedestrian dynamics? What applies to vehicles that does not to people. The trivial question is, ground rules that apply to each member of the traffic equally and everyone has to follow at the same manner in the same situation. So which rules can be captured from city traffic and translated to pedestrian behavior at an intersection. We have extracted three major methods to organize an apparently random behavior of pedestrians at an intersection, of course combination of these methods are also possible.

- Roundabouts
- Lights
- Traffic lanes

3.3 Expected results

We expect to see an improvement in fluidity in the crowd due to universal laws. Everybody will follow the ground rule first, only after that the individual behavior comes into play.

4. Description of the Model

4.1 General

For our simulation we choose to adopt an agent based model. An agent, in our case a pedestrian, should be able to walk freely in direction of his target until another agent, boundary or obstacle appears in his viewing field, in which case they should be able to avoid the considered obstacle. The main objective of each agent is to get to the other side of the intersection as quick as possible without crashing into other things. As the global situation, i.e. the distribution of other agents across the intersection, changes with each step our agent has to iterate its surroundings each step to evaluate the situation and find the optimal way for that precise step to get across the intersection safely. Due to the step by step iteration the absolute quickest path through the intersection may not be found because for such a result the agent has to know future paths of other agents to avoid future collisions efficiently. Our model though gives a realistic and hopefully optimal solution by deciding the best path of each pedestrian for each step so that it can get as close as possible to the real situation.

4.2 Moving of an agent through space

Our simulated pedestrian moves through space by forces applied to the agent in direction of his target. Since every pedestrian has a own mass the force results in a velocity leading to the final destination of the chosen agent. Multiple targets are chosen to avoid forces to point a path that leads for example through a wall or against an object or agent. The main target consists of different secondary targets which are updated and changed depending on the actual situation. The agent tries to reach his secondary target, if he is successful the secondary target is updated to another secondary target down the path, which he tries to reach next. If the path if obstructed, the target is moved to a more optimal position.

4.3 Checking of an agents' surroundings

Every pedestrian is located in a square cell and recognizes objects and other agents in his and the surrounding 8 cells. Every step the model iterates through each cell confining the agents' cell to detect objects and optimally set a target. The size of one cell can be seen a variable. But we have to consider that thus a greater cell size gives us a more complete comprehension of the surroundings the update time for each step dramatically increases due to the detection of more objects in this field and their relative evaluation.

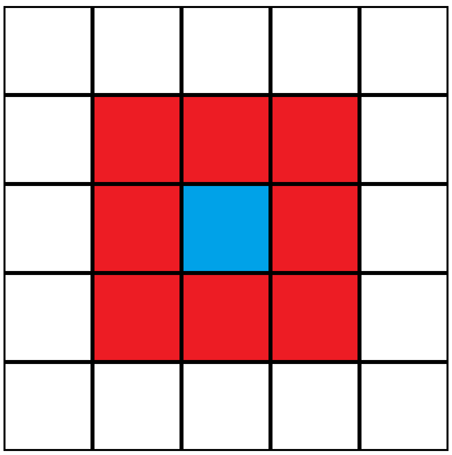

Fig 2: First order 3x3 Moore neighborhood

4.4 Avoidance of objects

In this chapter the main idea surrounding the "social force" model comes into play. In reality we tend to conserve our nearest personal space to ourselves. We don't feel comfortable sharing this space with other people, except the conditions of our environment absolutely forces us to. It is possible to implement this idea by applying forces to each agent-to-agent interaction. If the distance between agents is large the force will have little or no impact on the behavior of singles. According to that the more two agents get close to one another the more this force increases and tends the two agents to increase the distance between each other. In this way we can simultaneously evaluate the behavior of the individual and the avoidance of objects, i.e. agents, on its path.

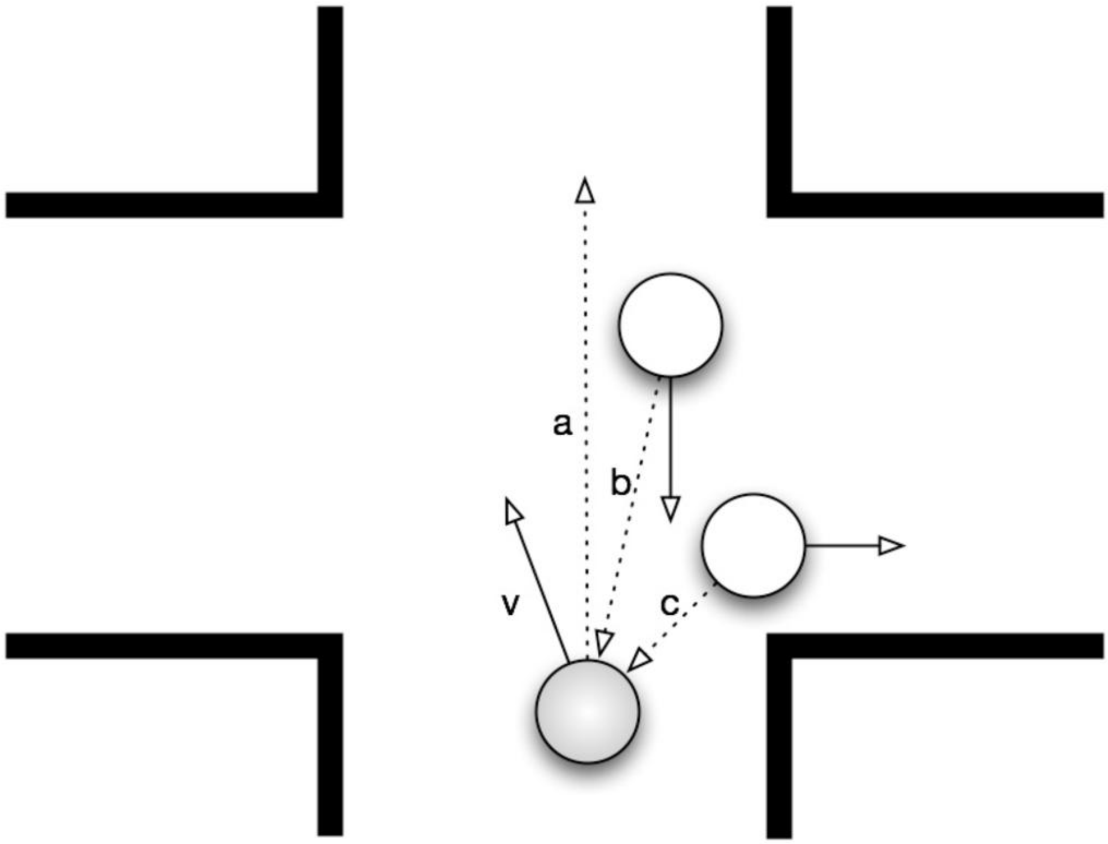

Fig 3: A diagram exemplifying an analytical model for pedestrian movement: the gray pedestrian, in the intersection, has an overall velocity *v* that is the result of an aggregation of the contributions related to the effects of attraction by its own reference point (**a**), and the repulsion by other pedestrians (**b**) and (**c**).

Other than avoiding collisions with other agents we have to take in consideration the collision with the model boundaries, in our case walls and objects like roundabouts. This is also computed by a force. Different than forces exercised by agents wall forces have a fix action radius which is chosen to be relatively small so that it has no impact if the agent isn't directly in contact with the wall. When there is a contact with a boundary the value of the forces' amplitude has to be chosen so great that even if the observed pedestrian is surrounded by many other agents, each exercising a force that pushes him in direction of the boundary, they are not able to push him through the wall.
Summarizing, for each step each agents' force has to be evaluated based on the force that draws the pedestrian towards his destination and the forces that push him away from boundaries and other agents, as it can be seen in the following equation.

$$F_{agent} = F_{target} + \sum F_{surrounding-agent} + \sum F_{boundary}$$

5. Implementation

5.1 General

Before we started implementing our model we defined a few ground rules which would help us with the evaluation of the result and simplify the overview of the code. Here are the most important:

- All variables had to be easily accessible and exchangeable. This becomes important especially when the same model has to be analyzed under different conditions. All our main variables can be changed in the *main.m* folder.

- Instead of one long code solving many different tasks we broke it down into multiple functions, where each one does his specific job. This helps maintaining a clear overview and changes of a specific task becomes much easier to find and modify.

- Features such as graphs, videos and any other additional functions should be easily turned off if a simple observation is required. This dramatically improves update time.

5.2 Cases

For our simulation we implemented different cases and situations. For the standard intersection the following cases can be applied:

- Roundabouts
- 4 roundabouts

Every one of these cases can be optimized with the following modifications:

- Intersection with cut off edges
- Separation lanes
- Separation lanes with lights

It is of course possible to also evaluate combinations of the above listed modifications. But only realistic situations can be evaluated. For example we can’t apply traffic lights without having a lane separation at the middle of the road.

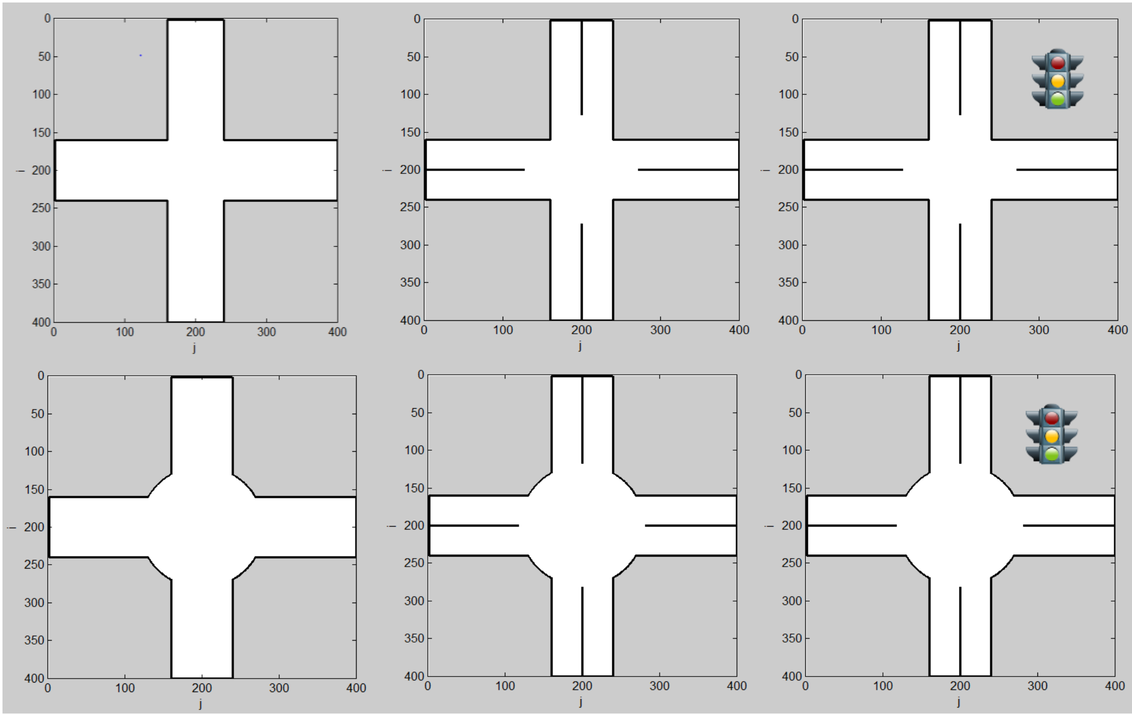

Fig 4: Every possible modification applied to the case of a simple intersection.

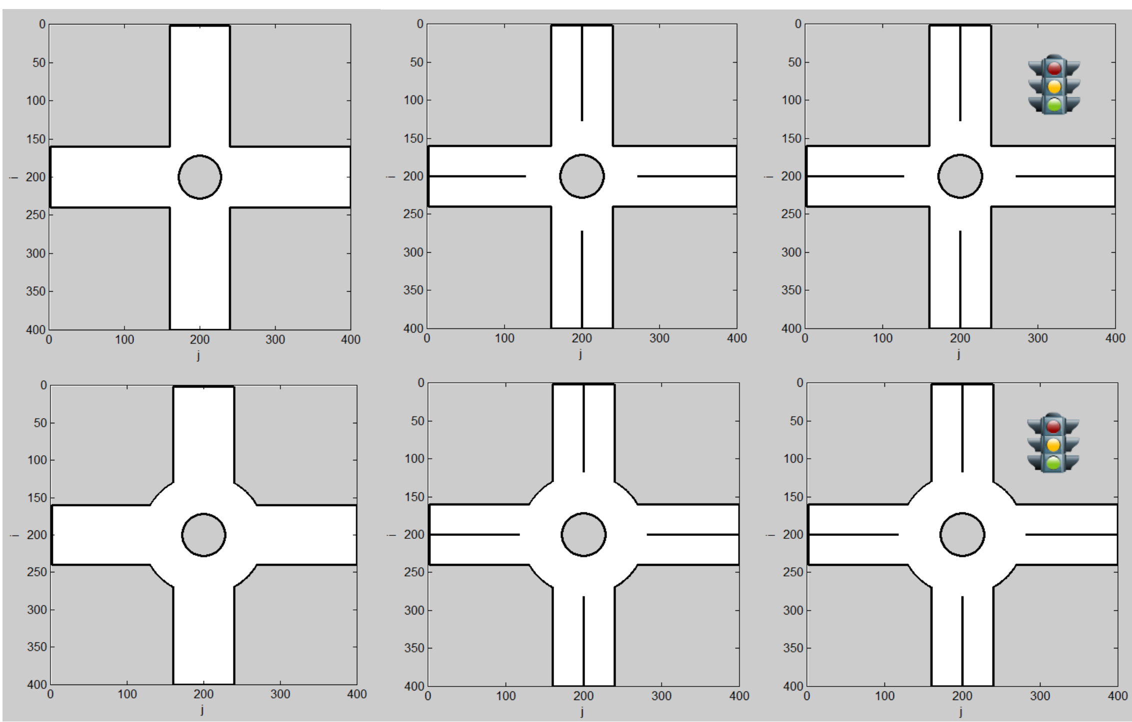

Fig 5: Every possible modification applied to the case of an intersection with roundabouts.

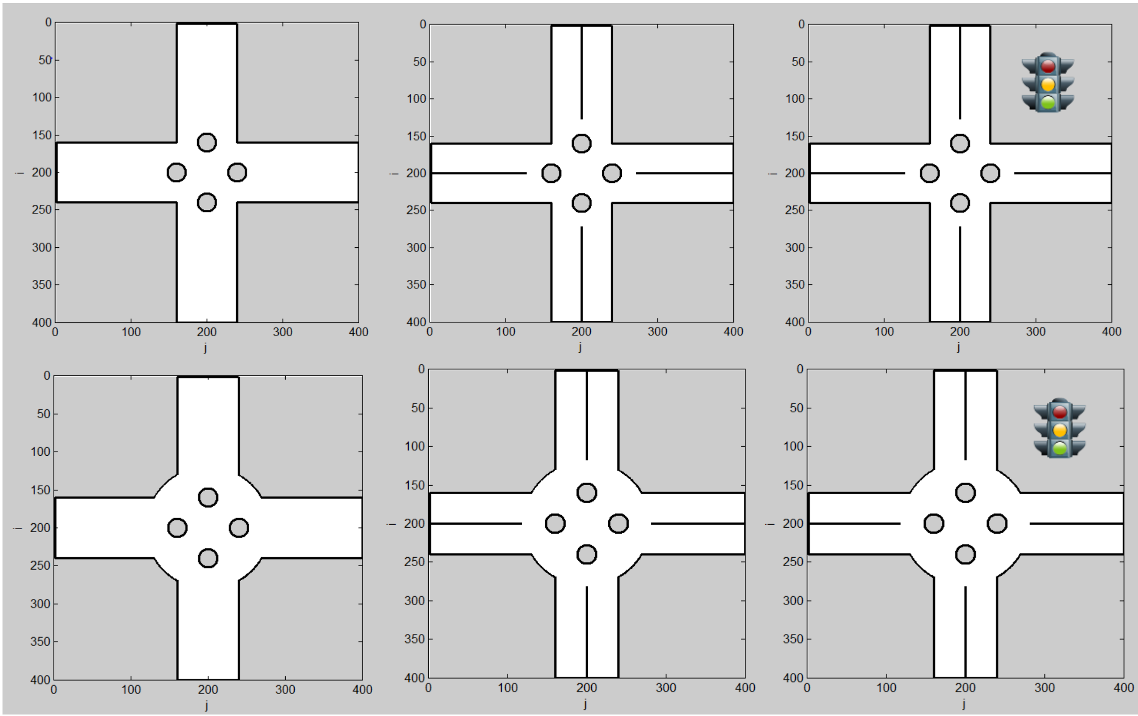

Fig 6: Every possible modification applied to the case of an intersection with four roundabouts.

Although it is possible to evaluate every single case we excluded some in the beginning. Due to high computational times we limited ourselves to the optimal cases. For example it can be easily seen that a roundabout with an intersection with cut off edges allows a more dynamic pedestrian flow compared to a roundabout at a normal intersection.

5.3 Variables

Variable	Description
Nx, Ny	Grid resolution, distance between two grid points in x and y
W_line	Width of lines
update_type	Available update types (Euler or Runge-Kutta)
dt	Integration step
t_end	Simulation time
with_graphic	If true graphic output is displayed
with_video	If true video of simulation is generated
r	Radius of each pedestrian
w	Probability that pedestrian is generated at

	boundaries
interaction_distance	Size of square cell around pedestrian
target_radius	Maximum distance to next secondary target
color_set	Color of pedestrians in the graphic output can be chosen to identify speed or destination.
p_max	Maximum number of pedestrians allowed on the grid
intersection_type	Can be chosen as a narrow hallway or a 4 branched intersection
roundabout_radius	Dimension of the roundabout
roundabout_4_radius	Dimension of the four roundabouts
super_radius	Radius on which the four roundabouts are placed
intersection_radius	Radius at which the intersection angles are cut
traffic_light_time	Time after which the traffic lights get changed
traffic_light_time_closed	Amount of time where all traffic lights are closed
number_of_traffic_lights_red	Amount of traffic lights which are red at the same time

5.4 Functions

Function	**Description**
cell_interaction.m	
draw_obstacle.m	Generates on the grid obstacles such as roundabouts, middle lanes etc.
draw_pedestrian.m	Displays pedestrians on the grid based on coordinates, velocity and location where the pedestrian is generated
evade_boundary.m	Corrects velocity and position of a pedestrian if collision with boundary is expected
evade_pedestrian.m	Corrects velocity and position of a pedestrian if collision with other pedestrians is expected
evade_traffic_lights.m	Stops the pedestrian if traffic light is red
final_target.m	Defines a randomized target branch
find_min_distance.m	find the vector with minimum distance pointing from a point located on AB to the point C

force_collision_boundary.m	Avoids that pedestrians block generation points
force_collision_pedestrian.m	Produces an impulse that acts only in some updating steps but has huge amplitude. It tries to correct the current direction and prevent of a total-blocking situation
force_comfortable_zone.m	Avoids if possible other pedestrians entering personal space (separates pedestrians)
force_reach_target.m	Evaluates the needed force to reach the next target point
generate_grid.m	Generates a NxM grid with intersection boundaries
generate_pedestrian.m	Generates a pedestrian at boundaries
grid2cell.m	Uses a cell storage method of Grid to reduce the numerical effort when computing interaction between several pedestrians
is_available.m	Verifies whether the updating position is free or occupied
is_available2.m	Checks if updating position is available or not
is_collision.m	Verifies whether collision is expected or not
is_visible.m	checks whether the neighbor is visible for the current pedestrian or not
main.m	Setting of all variables
pedestrian_boundary_distance.m	Generates vector pointing from nearest boundary to center of current pedestrian
simulate.m	Organization of the simulation
traffic_lights_regulator.m	Helper function for regulating the traffic light dynamics
update.m	Update of grid and indices Calculation of all average measurements Elimination pedestrians who left the grid
update_force.m	Evaluation of the forces for each pedestrian
update_pedestrian.m	Elimination of pedestrians who left the grid Generation of new pedestrians at boundaries
update_position.m	Updates position of each pedestrian Updates velocity of each pedestrian

6. Simulation results and Discussion

6.1 Resulting graphs

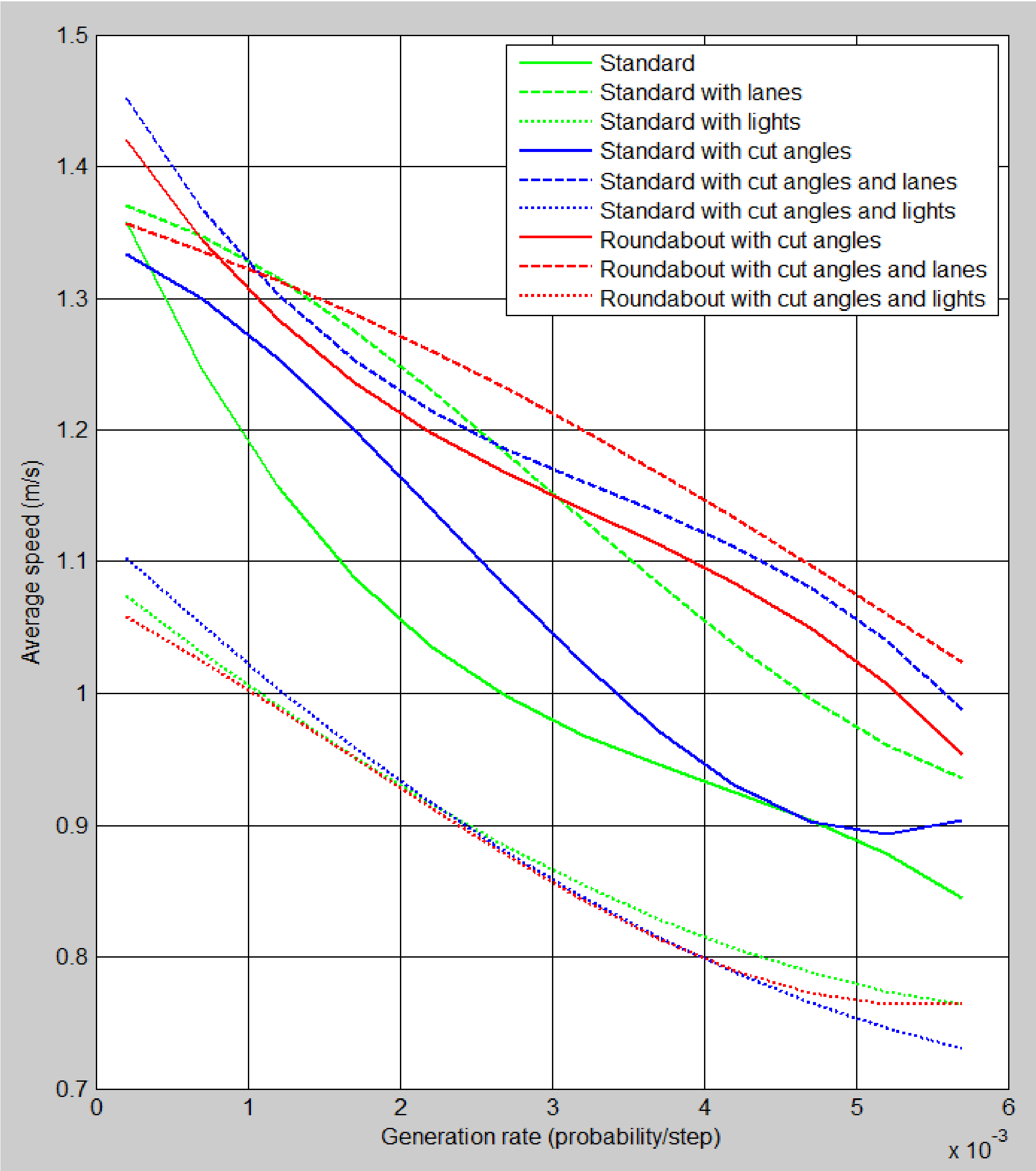

Fig 7: Average speed behaviour with constantly increasing geeration rate.

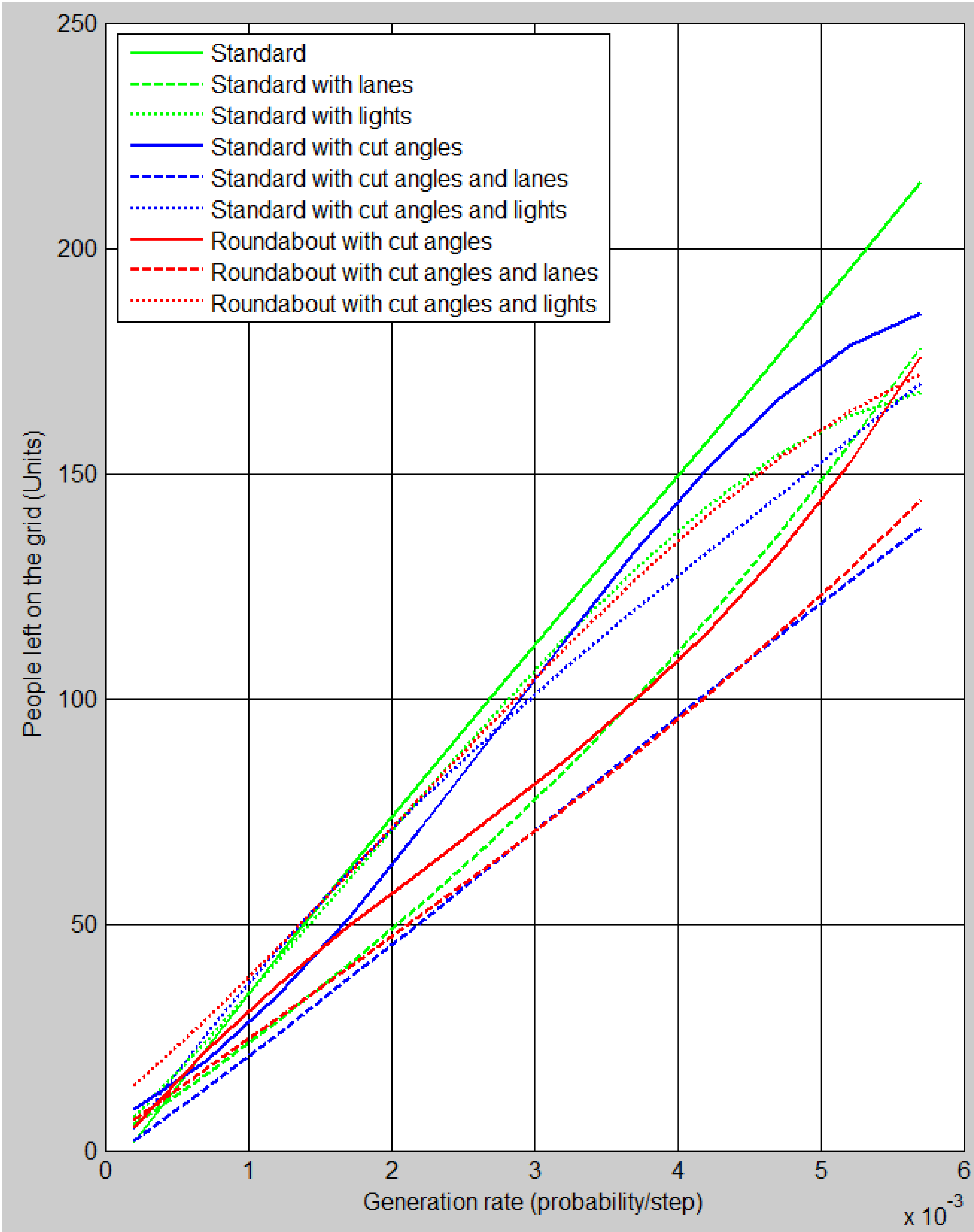

Fig 8: Number of pedestrians left on the grid at the end of the simulation with constantly increasing generation rate. Note the slope of the lines.

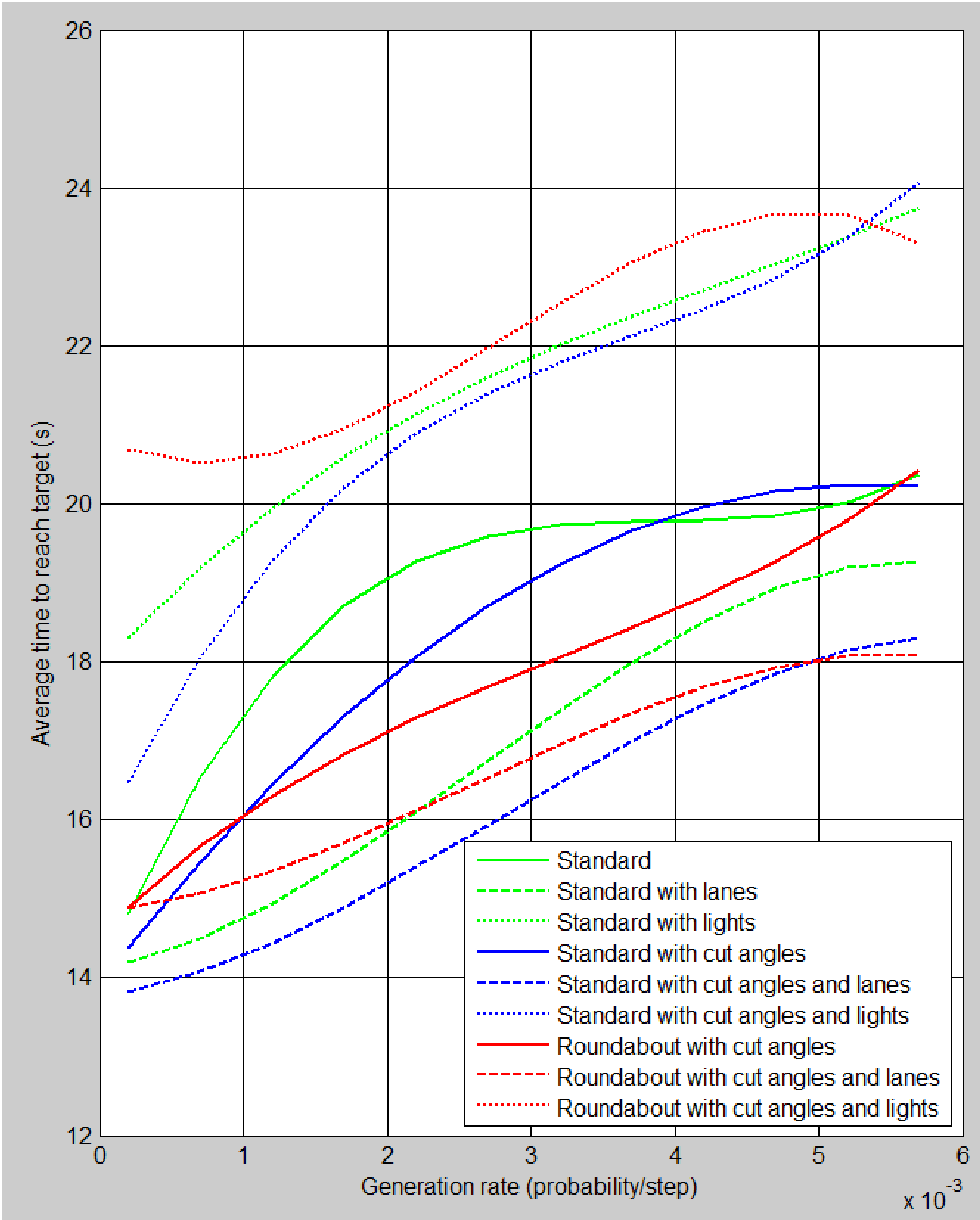

Fig 9: Average time to reach target with constantly increasing generation rate. Note the saturation points of the curves.

6.2 Evaluation of cases

6.2.1 Exclusion of the case with four roundabouts

Our first thought when we decided to introduce the case with four roundabouts was, that they should help channel the pedestrians towards their target. This case would then be applied to the big roundabout hoping this would increase the order in which the pedestrians moved across the grid.
Simulating our model we found out that the results of this case were far from ideal. It was not nearly half as good as the standard intersection without modifications. The pedestrians saw the small roundabouts more as obstacles than a help to get through the crowd. Jams started to form earlier, pedestrians couldn't find their ideal path to their target and got carried away by all people moving in other directions.
The principle of our model is to apply simple ground rules for the crowd to follow, apparently the rules that were to follow for the four roundabouts where not simplistic enough for them to have a positive impact on the dynamics of the pedestrians. We abandoned further tests of this case and choose to exclude it from the evaluation of the graphs so that we could concentrate our simulation on the cases we thought were more relevant and produced cleaner overall results.

6.2.2 Average speed evaluation

It can be seen that the best solutions regarding overall speed are the ones with traffic lanes in the branches. Especially the case "roundabout with cut off angles and lanes" shows the best results when it comes to high generation rates. Also the dynamics of this case are pretty constant (slope of the curve has no fast change). A fast change in the slope of the curve signals a complete obstruction of a branch. This can easily be seen in the case "roundabout with cot off angles", where a fast decrease of the average speed is seen up to a generation rate of 0.002 meaning a fast jamming in a branch. At this point this branch is completely shut and the slope becomes less steep because the flow throughout the other three branches does not change by much. Until at a generation rate of 0.004 another branch starts jamming and the slope of the curve drops again. An absence of this behavior signals a solid distribution of the pedestrian density throughout the intersection and, if overcrowded, the blocking of all four branches occurs simultaneously. Note that even if the cases where lights have been applied appear to have a slower average velocity, this is due to the fact that the time pedestrians spend waiting for green light decreases dramatically the average speed, have the most constant slopes of all cases. This behavior is almost identical independently from cases, such as roundabouts etc.
The fact that even with a total jam in the system the average speed does not fall to zero is due to newly generated pedestrians still walking into the jam with relatively high speeds.

6.2.3 Pedestrians left on the grid evaluation

We can see that the standard intersection has the steepest slope, whether it is evaluated normally, with lanes or with lights, compared to the roundabout and the standard intersection with cut off angles. This is because jams are very likely to begin forming themselves in angles, which leads to a bigger amount of people stuck in the intersection relative to the increasing generation rate. To minimize this value lights may not be the preferred case to apply, due to the fact that when thy turn green swarms of people access the intersection and are much more likely to interfere with one another than if pedestrians get into the intersection one at the time.
Out of this graph it can be seen that the best behavior yields the traffic lanes with cut off intersection angles.

6.2.4 Average time to reach target evaluation

The standard intersection has a very early saturation point for this value (at a generation rate of about 0.003). Because only people reaching their final destination contribute to the average time effort, people stuck in the intersection are not taken into consideration. This means that saturation points of lines are synonymous for a complete blockage of the intersection. This means that the worst possible scenario is in fact the standard intersection.
Again the high values attributed to the cases where lights are applied are a consequence of the waiting time at the intersection and due not commute to an overall unwanted behavior.
The best result is the one obtained with a roundabout with traffic lanes. Even if at low generation rates it is beaten by the other two cases with traffic lanes, this behavior is correlated with the fact that pedestrians have to walk around an object in the middle of the intersection, it displays a less steep slope which means it has a more constant dynamic behavior.

7. Summary and Outlooks

7.1 Summary

We saw in the last chapter the dynamic behavior of different cases applied to varying generation rates and their effect on pedestrian crowds. In the table below are listed the cases with their positive and negative effects on the system.

Case/Modification	Pros	Cons
Cut off angles	-More space to avoid collisions -Jams in angles are retarded	
Roundabouts	-Better/more constant flow -Pedestrians circle in the same direction (no frontal collisions) -Easy access to every branch equally	-Increased time to reach target due to circling
Traffic lanes	-Division between out coming and incoming traffic -Easier to channel pedestrians into roundabout -Reduced time to reach target at high generation rates	-At very high density people can get drawn into a not desired branch and can't escape -At low density time to reach target can increase
Traffic lights	-Amount of people entering the intersection can be regulated -Average speed is more constant -Dynamics of the system increases	-Increased time to reach target -Lower average speed -People are send in the intersection in groups

Furthermore we can now say that the best overall improvement in the systems' dynamics is obtained with a **roundabout with cut off angles and traffic lanes**.

7.2 Conclusion and final words

We are very proud of the good results our model has shown. Even prouder of the fact that we did not apply any existing model, instead we came up by ourselves with all the model properties, laws by which it works and functions that regulate its dynamics. The only thing we took as outside help was the general idea of crowd behavior based on social forces. Everything else is the result of our work and deductions.

We are aware of the fact that our model only applies to social forces. Although a realistic crowd behaves with a lot of different properties not included in our model, such as individualistic decisions, preferences, direct interactions and many more we have a pretty solid approximation of the ground principles with which a big crowd moves. The best part of our model is that it can be amplified and extended with all those principles and can get even closer to reality. Individualistic decisions have to be modeled and can be included in our existing code. At this point the only limit is the capacity of the update time of the simulations. With bigger processors and stronger and more powerful computers the application of this model are merely infinite.

8. References

[1] Ramin Mehran, Alexis Oyama, Mubarak Shah (2009), *Abnormal crowd behavior detection using social force model.*

9. Matlab source code

9.1 draw_obstacle.m

```
function [] = draw_obstacle()
%=========================================================================
% This function draws the whole grid including all obstacles.
%=========================================================================

global with_roundabout Nx Ny with_middle_line W_line ...
    with_inner_radius ay by with_4_roundabouts N M

% draw grid
colormap([0.8 0.8 0.8])
clf                 % clear figure
imagesc(zeros(N,M)) % display grid
axis equal % sets the aspect ratio so that the data units are the same
% in every direction. The aspect ratio of the x-, y-, and z-axis is
% adjusted automatically according to the range of data units
axis([0 M,0,N]) % set axis limits
xlabel('j')
ylabel('i')


load('variables', 'bounds') % load struct profile

% draw profile
profile = bounds.profile;
patch(profile(1,:), profile(2,:), [1 1 1], 'LineWidth', 2)

% draw inner circle
if with_inner_radius
    inner_circle = bounds.inner_circle;
    R = inner_circle(1)/Nx;       % radius
    Mid = inner_circle(2:3);      % center point (indices)

    ky = (by-ay)/2;
    kx = sqrt((R)^2-ky^2);
    for i=0:3
        angles = pi/2*i+atan(ky/kx) : 0.01 : pi/2*i+atan(kx/ky);
        dR = R;
        x = dR*cos(angles);
        y = dR*sin(angles);
        patch(Mid(1)+x, Mid(2)+y, [1 1 1], 'LineWidth', 2);
    end
    angles = 0:0.01:(2*pi);
    dR = 0.99*R;
    x = dR*cos(angles);
    y = dR*sin(angles);
    patch(Mid(1)+x, Mid(2)+y, [1 1 1], 'EdgeColor', 'None');
end


% draw roundabouts
if with_roundabout || with_4_roundabouts
    outer_circle = bounds.outer_circle;
    [ns, ms] = size(outer_circle);
```

```
    for k=1:ms
        circle = outer_circle(:,k);
        R = circle(1)/Nx;       % radius
        Mid = circle(2:3);      % center point (indices)
        angles = 0:0.01:(2*pi);
        dR = R;
        x = dR*cos(angles);
        y = dR*sin(angles);
        patch(Mid(1)+x, Mid(2)+y, [0.8 0.8 0.8], 'LineWidth', 2)
    end
end

% draw lines
if with_middle_line
    straight_lines = bounds.straight_lines;
    [nl, ml] = size(straight_lines);
    for k = 1:2:ml-1
        line = straight_lines(:,k:k+1);
        A = line(:,1);
        B = line(:,2);
        patch([A(1), B(1)],[A(2), B(2)], 'w', 'LineWidth', W_line/Ny)
    end
end


end
```

9.2 draw_pedestrian.m

```
function [] = draw_pedestrian(i,j,velocity, gen_location)
%==========================================================================
% This function draws all pedestrian as a circle with radius r at the
% position where the pedestrian is located. Two color schemes are available
%   >> the color depends on the velocity: red if v=0, green if v=vmax
%   >> the color depends on the position where the pedestrians where
%   generated.
%--------------------------------------------------------------------------
% Input:    > coordinate of the pedestrian (i,j)
%           > its velocity vector
%           > and the location where the pedestrian is generated
%==========================================================================

global Nx r color_set v_min v_max
v_min = 0;

hold on         % Hold the graphics

% Define the coordinates for the pedestrian
angles = 0:0.1:(2*pi);
dR = r/Nx;
x = dR*cos(angles);
y = dR*sin(angles);

if color_set == 1 % make the color depending on the generating location
    switch gen_location
        case 1
            color = [1 0 0];
        case 2
            color = [0 1 0];
```

```
        case 3
            color = [0 0 1];
        case 4
            color = [0.5 0.5 0.5];
    end

else % make the collor dependend of the velocity
    v = norm(velocity);
    scale = min(1, max(0, (v-v_min)/(v_max-v_min)));
    red     = max(0, 1 - 2*scale);       % slow
    blue    = 2*(0.5 - abs(0.5-scale)); % medium
    green   = max(2*scale - 1, 0);       % fast
    color   = [red green blue];
end

% Draw pedestrian!
patch(j+x, i+y, color, 'EdgeColor','none')
end
```

9.3 evade_boundary.m

```
function [x, v] = evade_boundary(dist_critical, distance, x_old, x_new, v)
%==========================================================================
% evade_boundary() checks wheter the updating position is available or if
% there is a boundary point that prevents from update. If the update is not
% possible, new position and velocity vectores will be created based on
% the filter-principle ("set the component to zero that is orthogonal to
% the boundary line").
%--------------------------------------------------------------------------
% > Input:   > dist_critical: cell array with dist_crit (vector pointing
%            from current center pedestrian to nearest boundary point)
%            and boundary type (string with type of the boundary point)
%            > distance: vector pointing from current pedestrian to
%            preferable updating position
%            > x_old: current position
%            > x_new: preferable updating position
%            > v: preferable velocity vector
% > Output:  > corrected position x and corrected velocity v
%==========================================================================

global r dt W_line

% extract infomrations
boundary_type = dist_critical{1};
dist_crit = dist_critical{2};

x=x_new;        % assumption
security = 1.3; % security factor

% exclude all cases, where no touching of boundary points expecte
cos_alpha = dist_crit'*distance/(norm(dist_crit)*norm(distance));
if (strcmp(boundary_type,'profile') || strcmp(boundary_type,'outer_circle')) &&
(cos_alpha*norm(distance) < norm(dist_crit)-security*r || cos_alpha<=0)
    return % leave this function
elseif strcmp(boundary_type,'corner') && norm(dist_crit)>0.9*r
    % do not use any security factor here. This will ensure that
    % pedestrians no not remain on corners when they are blocked.
    return
```

```
elseif strcmp(boundary_type,'straight_lines') && (cos_alpha*norm(distance) <
norm(dist_crit)-(security*r+W_line/2) || cos_alpha<=0)
    % the lines have a width and have therfore a 3dim charater!
    return
end

% perform a coordinate transformation (dist_critical -> new x-axis):
cos_gamma = [1 0]*dist_crit/(norm(dist_crit)); % angle between dist_crit and x-
axis
sin_gamma = sqrt(1-cos_gamma^2);

if dist_crit(2)>0
    M = [cos_gamma sin_gamma; -sin_gamma cos_gamma]; % rotation in positive
direction
else
    M = [cos_gamma -sin_gamma; sin_gamma cos_gamma]; % rotation in negative
direction
end

% transformation of some date
V = M*v;
DIST_critical = M*dist_crit;

if DIST_critical(2)>10^-3 % check if transformation was successful
    disp('(evade_boundary) Attention: Coordinate transformation faild')
end

if strcmp(boundary_type,'profile') || strcmp(boundary_type,'corner') ||
strcmp(boundary_type,'outer_circle') || strcmp(boundary_type,'straight_lines')
    V(1) = 0;     % filter: let only pass the y-component of the velocity
    v = M\V;      % inverse transformation
    x=x_old+dt*v;% Calculate new position (Euler)
end


end
```

9.4 evade_pedestrian.m

```
function [x, v] = evade_boundary(dist_critical, distance, x_old, x_new, v)
%==========================================================================
% evade_boundary() checks wheter the updating position is available or if
% there is a boundary point that prevents from update. If the update is not
% possible, new position and velocity vectores will be created based on
% the filter-principle ("set the component to zero that is orthogonal to
% the boundary line").
%--------------------------------------------------------------------------
% > Input:   > dist_critical: cell array with dist_crit (vector pointing
%            from current center pedestrian to nearest boundary point)
%            and boundary type (string with type of the boundary point)
%            > distance: vector pointing from current pedestrian to
%            preferable updating position
%            > x_old: current position
%            > x_new: preferable updating position
%            > v: preferable velocity vector
% > Output:  > corrected position x and corrected velocity v
%==========================================================================
```

```
global r dt W_line

% extract infomrations
boundary_type = dist_critical{1};
dist_crit = dist_critical{2};

x=x_new;         % assumption
security = 1.3; % security factor

% exclude all cases, where no touching of boundary points expecte
cos_alpha = dist_crit'*distance/(norm(dist_crit)*norm(distance));
if (strcmp(boundary_type,'profile') || strcmp(boundary_type,'outer_circle')) &&
(cos_alpha*norm(distance) < norm(dist_crit)-security*r || cos_alpha<=0)
    return % leave this function
elseif strcmp(boundary_type,'corner') && norm(dist_crit)>0.9*r
    % do not use any security factor here. This will ensure that
    % pedestrians no not remain on corners when they are blocked.
    return
elseif strcmp(boundary_type,'straight_lines') && (cos_alpha*norm(distance) <
norm(dist_crit)-(security*r+W_line/2) || cos_alpha<=0)
    % the lines have a width and have therfore a 3dim charater!
    return
end

% perform a coordinate transformation (dist_critical -> new x-axis):
cos_gamma = [1 0]*dist_crit/(norm(dist_crit)); % angle between dist_crit and x-
axis
sin_gamma = sqrt(1-cos_gamma^2);

if dist_crit(2)>0
    M = [cos_gamma sin_gamma; -sin_gamma cos_gamma]; % rotation in positive
direction
else
    M = [cos_gamma -sin_gamma; sin_gamma cos_gamma]; % rotation in negative
direction
end

% transformation of some date
V = M*v;
DIST_critical = M*dist_crit;

if DIST_critical(2)>10^-3 % check if transformation was successful
    disp('(evade_boundary) Attention: Coordinate transformation faild')
end

if strcmp(boundary_type,'profile') || strcmp(boundary_type,'corner') ||
strcmp(boundary_type,'outer_circle') || strcmp(boundary_type,'straight_lines')
    V(1) = 0;     % filter: let only pass the y-component of the velocity
    v = M\V;      % inverse transformation
    x=x_old+dt*v;% Calculate new position (Euler)
end


end
```

9.5 evade_traffic_lights.m

```
function [x, v] = evade_traffic_lights(traffic_lights, distance, x_old, x_new,
v)
%==========================================================================
% evade_traffic_lights() checks wheter the updating position is available
% or if there is an other traffic light that prevents from update. If the
% update is not possible, the velocity is set to zero.
%--------------------------------------------------------------------------
% > Input:   > traffic_lights: location of the traffic lights
%            > distance: distance from pedestrian to traffic light
%            > x_old (current position), x_new (updating poistion), v
%            (current velocity)
% > Output:  > corrected position x and corrected velocity v
%==========================================================================

global r
security = 1.3;
x = x_new;
dist_crit = traffic_lights;

% exclude all cases, where no touching of boundary points expecte
if ~traffic_lights
    return % leave this function
end

cos_alpha = dist_crit'*distance/(norm(dist_crit)*norm(distance));

if cos_alpha*norm(distance) < norm(dist_crit)-security*r || cos_alpha<=0
    % mos_alpha might be NaN, but this does not matter
    return
end

% total deacceleration it traffic light is arrived
x = x_old;
v = [0;0];


end
```

9.6 final_target.m

```
function [target] = final_target(gen_location)
%==========================================================================
% All pedestrian follow a manual generated (preferred) trajectory.
% This trajectroy consits of several mid_points and exactely one final
% point. The the target points should be chosen such that the resulting
% trajectory minimizes the way the pedestrian have to walk on. Notice that
% is is not necessary to implements several points to obtain a smooth. A
% great number of target points causes predictable peestrian movements and
% lead to a loss of dynamic.
% The definition of the trajectory depends on the application that are
% chosen:
%   >> no applications: the pedestrians approach the center and skip as far
%   as possible to the final target.
%   >> with_roundabout=true: all pedestrians walk on counter clock wise
%   around the roundabout. The try to walk as near as posible to the
%   roundabout.
%   >> with_inner_radius=true: no infuence on the preferred trajectory.
%   >> with_middle_line=true: preffered trajectory is restricted inside the
%   branches.
```

```
%   >> with_4_roundabouts=true: If with_roundabout=false, than the
%   the pedestrian walk on trajectories like in the case where no
%   applications are chosen. The only difference is that the roundabouts
%   have a separating impact on the whole dynamik. If with_roundabout=true
%   the trajectory is the same as if only the roundabout is used. The four
%   roundabouts act like a "pre-separater".
%--------------------------------------------------------------------------
%Input:     > gen_location: number of the branch where pedestrian is
%           generated on
%Output:    > indices of the final target
%==========================================================================

global N M Nx Ny intersetction_type ay by ax bx roundabout_radius ...
    with_roundabout with_middle_line with_inner_radius W_street ...
    intersection_radius with_4_roundabouts super_radius with_traffic_lights

% uniform distribution
uniform=@(a, b)(a + (b-a)*rand);

%--------------------------------------------------------------------------
if intersetction_type==1
   final_target = mod(gen_location,2)+1;

   switch(final_target)
        case 1 % left branch
            final_point = [N/2; -M*0.5]; % overshooting!
        case 2 % right branch
            final_point = [N/2; M*1.5];
   end
   mid_points = [];

%--------------------------------------------------------------------------
elseif intersetction_type==2 && ~with_roundabout && ~with_4_roundabouts

    % Introduce additional space for mid_points (do not walk into
    % boundary points!!). To make sure that not all pedestrian tries to reach
    % the same mid points, let this factor be varying
    security_x = round(uniform(0.7, W_street*0.6)/Nx);
    security_y = round(uniform(0.7, W_street*0.6)/Ny);


    % Randomize the final target
    final_target=gen_location;
    while final_target==gen_location % do not walk back to initial point
        final_target = round(uniform(1,4)); % random number in the range (1,4)
    end

    switch(final_target)
        case 1 % left branch
            final_point = [N/2; -M*0.5];
            switch(gen_location)
                case 2
                    mid_points = [N/2;M/2]; % this point is necessary if the
pedestrian is pushed aside
                case 3
                    mid_points = [ay;ax] + [security_y; security_x];
                case 4
                    mid_points = [by; ax] + [-security_y; security_x];
            end
        case 2 % right branch
            final_point = [N/2; M*1.5];
            switch(gen_location)
                case 1
```

```
                    mid_points = [N/2;M/2];
                case 3
                    mid_points = [ay; bx] + [security_y; -security_x];
                case 4
                    mid_points = [by; bx] + [-security_y; -security_x];
            end
        case 3 % upper branch
            final_point = [-N*0.5; M/2];
            switch(gen_location)
                case 1
                    mid_points = [ay; ax] + [security_y; security_x];
                case 2
                    mid_points = [ay; bx] + [security_y; -security_x];
                case 4
                    mid_points = [N/2;M/2];
            end
        case 4 % lower branch
            final_point = [N*1.5; M/2];
            switch(gen_location)
                case 1
                    mid_points = [by; ax] + [-security_y; security_x];
                case 2
                    mid_points = [by; bx] + [-security_y; -security_x];
                case 3
                    mid_points = [N/2;M/2];
            end
    end

%--------------------------------------------------------------------------
elseif intersetction_type==2 && with_roundabout

    final_target=gen_location;
    while final_target==gen_location
        final_target = round(uniform(1,4));
    end

    % radius the pedestrian try to wolk on
    if with_inner_radius
        R = uniform(roundabout_radius*1.1,intersection_radius*0.9);
    else
        R = uniform(roundabout_radius*1.1, sqrt((Nx*(bx-ax)/2)^2+(Ny*(by-
ay)/2)^2)*0.9 );
    end

    % two middle points include an angle of pi/8 (-> 16 middle points)
    varphi = 0:pi/8:2*pi;
    % here are the x- and y-components of the middle points
    mid_x = R*cos(varphi);
    mid_y = R*sin(varphi);
    mid_points = [mid_x; mid_y]; % non-indicies notation

    if with_inner_radius
        r_max = intersection_radius; % take greates radius that is possible
    else
        r_max = R/0.9;
    end

    switch(final_target)
        case 1 % left branch
            final_point = [N/2; -M*0.5];
            switch(gen_location)
                case 2
                    mid_points = mid_points(:,1:9);
```

```
                case 3
                    mid_points = mid_points(:,5:9)/R*r_max*0.9;
                case 4
                    mid_points = [mid_points(:,13:16), mid_points(:,1:9)];
            end
        case 2 % right branch
            final_point = [N/2; M*1.5];
            switch(gen_location)
                case 1
                    mid_points = [mid_points(:,9:16), mid_points(:,1)];
                case 3
                    mid_points = [mid_points(:,5:16), mid_points(:,1)];
                case 4
                    mid_points = [mid_points(:,13:16),
mid_points(:,1)]/R*r_max*0.9;
            end
        case 3 % upper branch
            final_point = [-N*0.5; M/2];
            switch(gen_location)
                case 1
                    mid_points = [mid_points(:,9:16), mid_points(:,1:5)];
                case 2
                    mid_points = mid_points(:,1:5)/R*r_max*0.9;
                case 4
                    mid_points = [mid_points(:,13:16), mid_points(:,1:5)];
            end
        case 4 % lower branch
            final_point = [N*1.5; M/2];
            switch(gen_location)
                case 1
                    mid_points = mid_points(:,9:13)/R*r_max*0.9;
                case 2
                    mid_points = mid_points(:,1:13);
                case 3
                    mid_points = mid_points(:,5:13);
            end
    end
    % transform mid_points to indices notation
    mid_points = round([-mid_points(2,:)/Ny+N/2; mid_points(1,:)/Nx+M/2]);
%---------------------------------------------------------------------------
elseif intersetction_type==2 && ~with_roundabout && with_4_roundabouts

    final_target=gen_location;
    while final_target==gen_location
        final_target = round(uniform(1,4));
    end

    kx = round((bx-ax)/4);
    ky = (round(by-ay)/4);
    px = round((super_radius+roundabout_radius)*uniform(1.15,1.6)/Nx);
    py = round((super_radius+roundabout_radius)*uniform(1.15,1.6)/Ny);

    switch(final_target)
        case 1 % left branch
            final_point = [N/2; -M*0.5];
            switch(gen_location)
                case 2
                    mid_points = [[N/2-ky;M/2+px],[N/2-ky;M/2],[N/2-ky;M/2-
px]];
                case 3
                    mid_points = [[N/2-py;M/2-kx],[ay+ky;ax+kx],[N/2-ky;M/2-
px]];
                case 4
```

```
                    mid_points = [[N/2+py;M/2-kx],[by-ky;ax+kx],[N/2+ky;M/2-
px]];
            end
        case 2 % right branch
            final_point = [N/2; M*1.5];
            switch(gen_location)
                case 1
                    mid_points = [[N/2+ky;M/2-
px],[N/2+ky;M/2],[N/2+ky;M/2+px]];
                case 3
                    mid_points = [[N/2-py;M/2+kx],[ay+ky;bx-kx],[N/2-
ky;M/2+px]];
                case 4
                    mid_points = [[N/2+py;M/2+kx],[by-ky;bx-
kx],[N/2+ky;M/2+px]];
            end
        case 3 % upper branch
            final_point = [-N*0.5; M/2];
            switch(gen_location)
                case 1
                    mid_points = [[N/2-ky;M/2-py],[ay+ky;ay+kx],[N/2-py;M/2-
kx]];
                case 2
                    mid_points = [[N/2-ky;M/2+py],[ay+ky;ay-kx],[N/2-
py;M/2+kx]];
                case 4
                    mid_points = [[N/2+py;M/2+kx],[N/2;M/2+kx],[N/2-
py;M/2+kx]];
            end
        case 4 % lower branch
            final_point = [N*1.5; M/2];
            switch(gen_location)
                case 1
                    mid_points = [[N/2+ky;M/2-px],[by-ky;ax+kx],[N/2+py;M/2-
kx]];
                case 2
                    mid_points = [[N/2+ky;M/2+px],[by-ky;bx-
kx],[N/2+py;M/2+kx]];
                case 3
                    mid_points = [[N/2-
py;M/2+kx],[N/2;M/2+kx],[N/2+py;M/2+kx]];
            end
    end

%--------------------------------------------------------------------------
end
%--------------------------------------------------------------------------

% replace last element mid_point and adjust the last target point
if with_middle_line || with_4_roundabouts
    dj = round((bx-ax)/4);
    di = round((by-ay)/4);

    if with_inner_radius
       Ay = N/2-intersection_radius/Ny;
       By = N/2+intersection_radius/Ny;
       Ax = M/2-intersection_radius/Nx;
       Bx = M/2+intersection_radius/Nx;
    else
        Ay = ay;
        By = by;
        Ax = ax;
        Bx = bx;
```

```
    end

    switch(final_target)
        case 3
            last_mid_point = [Ay; M/2];
            Delta = [0;dj];
        case 4
            last_mid_point = [By; M/2];
            Delta = [0;-dj];
        case 1
            last_mid_point = [N/2; Ax];
            Delta = [-di;0];
        case 2
            last_mid_point = [N/2; Bx];
            Delta = [di;0];
    end

    if isempty(mid_points) || length(mid_points)==1
        mid_points = last_mid_point;
    else
        if with_roundabout
            mid_points(:,end) = [];
        else
            last_mid_point = last_mid_point + Delta;
            mid_points(:,end) = last_mid_point;
        end
    end
    final_target = final_target + Delta;

end

% if traffic light are used distribute the pedestrians in front of the
% traffic light over the entire space that is available. THe following code
% will ensure this purpose:
if with_traffic_lights
    dj = round((bx-ax)/4);
    di = round((by-ay)/4);

    if with_inner_radius
       Ay = N/2-intersection_radius/Ny;
       By = N/2+intersection_radius/Ny;
       Ax = M/2-intersection_radius/Nx;
       Bx = M/2+intersection_radius/Nx;
    else
        Ay = ay;
        By = by;
        Ax = ax;
        Bx = bx;
    end

    switch(gen_location)
        case 1
            first_mid_point = [N/2; Ax];
            Delta = [di;0];
        case 2
            first_mid_point = [N/2; Bx];
            Delta = [-di;0];
        case 3
            first_mid_point = [Ay;M/2];
            Delta = [0;-dj];
        case 4
            first_mid_point = [By;M/2];
            Delta = [0;dj];
```

```
    end
    first_mid_point = first_mid_point+Delta;
    mid_points = [first_mid_point, mid_points];
end


if isempty(mid_points) % no mid points
    target = final_point;
else
    target = [mid_points, final_point];
end

end
```

9.7 find_min_distance.m

```
function [distance] = find_min_distance(A,B,C,distance,with_transformation)
%===========================================================================
% This function find the vector with minumum distance pointing from
% a point located on AB to the point C. This function works with indices
% notation only!!
%---------------------------------------------------------------------------
% Input:    > Two points A,B that define the line
%           > point C
%           > with_transfomration: true if transformation is necessary.
%           That is if the line AB should be unpassable for both sides.
%           Notice that the orientation of AB does matter!!
%           > distance: starting vector (for minimization)
% Output:   > distance (vector)
%===========================================================================

global N M

expected_intersection = true;

a = norm(B-C);
b = norm(A-C);
c = norm(B-A);

cos_beta = (C-B)'*(A-B)/(a*c);
cos_alpha = (C-A)'*(B-A)/(b*c);
sin_alpha = sqrt(1-cos_alpha^2);

if cos_beta<0 || cos_alpha<0 || ((A(1)==1 && B(1)==1) || (A(1)==N && B(1)==N)
|| (A(2)==1 && B(2)==1) || (A(2)==M && B(2)==M))
    expected_intersection = false;
end

dist = sin_alpha*b;

if dist < norm(distance) && expected_intersection
    e_AB = (B-A)/norm(B-A);
    distance = dist;

    if with_transformation

        % perform a coordinate transformation (AB -> new j-axis):
```

```
        cos_gamma = [0 1]*e_AB; % angle between x-axis and v1
        sin_gamma = (1-cos_gamma^2)^(1/2);

        if e_AB(1)>0
            T = [cos_gamma sin_gamma; -sin_gamma cos_gamma]; % rotation in
positive direction
        elseif e_AB(1)<0
            T = [cos_gamma -sin_gamma; sin_gamma cos_gamma]; % rotation in
negative direction
        else % e_AB(1)=0
            T = eye(2,2); % no rotation;
        end

        % Adjust the sign of the axis
        E_AB = T*e_AB;
        if E_AB(2)<0
            T = -T;
        end

        % check if transformation was successful
        E_AB = T*e_AB;
        if E_AB(1)>10^-5 || E_AB(2)<0
            warning('Attention: Coordinate transformation faild')
        end

        % Derive an expression for the direction
        check = T*C-T*A;
        if check(1)<0
            e = [-e_AB(2); e_AB(1)]; % rotation with -pi/2
        else
            e = [e_AB(2); -e_AB(1)]; % rotation with +pi/2
        end
        distance = e*distance; % vector pointing from boundary to pedestrian

    else
        e = [-e_AB(2); e_AB(1)]; % rotation with +pi/2
        distance = e*distance; % vector pointing from boundary to pedestrian
    end

end

end
```

9.8 force_collision_boundary.m

```
function [distance] = find_min_distance(A,B,C,distance,with_transformation)
%==========================================================================
% This function find the vector with minumum distance pointing from
% a point located on AB to the point C. This function works with indices
% notation only!!
%--------------------------------------------------------------------------
% Input:    > Two points A,B that define the line
%           > point C
%           > with_transfomration: true if transformation is necessary.
%           That is if the line AB should be unpassable for both sides.
%           Notice that the orientation of AB does matter!!
%           > distance: starting vector (for minimization)
% Output:   > distance (vector)
%==========================================================================
```

```
global N M

expected_intersection = true;

a = norm(B-C);
b = norm(A-C);
c = norm(B-A);

cos_beta = (C-B)'*(A-B)/(a*c);
cos_alpha = (C-A)'*(B-A)/(b*c);
sin_alpha = sqrt(1-cos_alpha^2);

if cos_beta<0 || cos_alpha<0 || ((A(1)==1 && B(1)==1) || (A(1)==N && B(1)==N)
|| (A(2)==1 && B(2)==1) || (A(2)==M && B(2)==M))
    expected_intersection = false;
end

dist = sin_alpha*b;

if dist < norm(distance) && expected_intersection
    e_AB = (B-A)/norm(B-A);
    distance = dist;

    if with_transformation

        % perform a coordinate transformation (AB -> new j-axis):
        cos_gamma = [0 1]*e_AB; % angle between x-axis and v1
        sin_gamma = (1-cos_gamma^2)^(1/2);

        if e_AB(1)>0
            T = [cos_gamma sin_gamma; -sin_gamma cos_gamma]; % rotation in
positive direction
        elseif e_AB(1)<0
            T = [cos_gamma -sin_gamma; sin_gamma cos_gamma]; % rotation in
negative direction
        else % e_AB(1)=0
            T = eye(2,2); % no rotation;
        end

        % Adjust the sign of the axis
        E_AB = T*e_AB;
        if E_AB(2)<0
            T = -T;
        end

        % check if transformation was successful
        E_AB = T*e_AB;
        if E_AB(1)>10^-5 || E_AB(2)<0
            warning('Attention: Coordinate transformation faild')
        end

        % Derive an expression for the direction
        check = T*C-T*A;
        if check(1)<0
            e = [-e_AB(2); e_AB(1)]; % rotation with -pi/2
        else
            e = [e_AB(2); -e_AB(1)]; % rotation with +pi/2
        end
        distance = e*distance; % vector pointing from boundary to pedestrian

    else
        e = [-e_AB(2); e_AB(1)]; % rotation with +pi/2
```

```
        distance = e*distance; % vector pointing from boundary to pedestrian
    end

end

end
```

9.9 force_collision_pedestrian.m

```
function [f] = force_collision_pedestrian(x1, x2, v1, v2, density)
%===========================================================================
% This force produces an impulse that acts only in some updating
% steps but has a hugh amplitude. It tries to correct the current
% direction and prevent of a total-blocking situation. This force has
% no effect if two pedestrians "overlap".
% such that collisions with other pedestrians can be avoided.
%----------------------------------------------------------------------------
% Input:    > position and velocity of pedestrian 1 and pedestrian 2
%           > density
% Output:   > force f
%===========================================================================

f = [0;0];  % assume no interaction

% do not compute the force if evading does not help. That is if
%   >> density is too great. Evading will produce an unatural float
%   behavior
%   >> the neighbour remains on its current position (v2<0.05). The
%   vanishing neighbour velocity indicates high density. Notice that the
%   parameter "density" can fail becouse we use the efficient cell storage
%   method. An other reason for the failur of "density" might be a small
%   cut off (which depends on the density on the last iterations)
if density>2 || norm(v2)<0.05
    return
end

r12 = x2-x1;

% First of all, Pedestrian 1 has to regognize in which situation he is
% > he walks behind pedestrian 2 (overtaking might be a possibility)
% > he walks in front of pedestrian 2 (evading might be a possiblility)
cos_beta = v1'*v2/(norm(v1)*norm(v2)); % angle between velocities
if  cos_beta>=0 % pedestrians walk in the same direction (-> overtaking?)
    % expected dinstance until collision occurs
    distance0 = norm(r12)*(norm(v1)+norm(v2))/(norm(v1)-norm(v2));
else % pedestrians walk against each other (-> evading?)
    distance0 = norm(r12);
end


% perform a coordinate transformation (v1 -> new x-axis):
cos_gamma = [1 0]*v1/(norm(v1)); % angle between x-axis and v1
sin_gamma = (1-cos_gamma^2)^(1/2);

if v1(2)>0
    M = [cos_gamma sin_gamma; -sin_gamma cos_gamma]; % rotation in positive
direction
else
```

```
    M = [cos_gamma -sin_gamma; sin_gamma cos_gamma]; % rotation in negative
direction
end

% transformation of same date
V1 = M*v1;
V2 = M*v2;
X1 = M*x1;
X2 = M*x2;
R12= M*r12;

if V1(2)>10^-5 % check if transformation was successful
    warning('Coordinate transformation failed')
end

% Decide in which direction the pedestrian want to evade/ overtake
cos_alpha = [1,0]*r12/(norm(r12)); % angle between x-axis and r12
if abs(cos_alpha) > 0.9 % special case if x-axis is approximalty parallel with
r12.
    % alpha depends on the coordinate system and has therefore no physical
    % meaning. However, this special case solves the problem of
    % "oscilating pedestirans", that is, if y-component of the neighboour's
    % velocity vector changes its sign and the current pedestrian reacts
    % with changing the evading/overtaking direction (sometomes this
    % yields in abstruse situatios)
    e = ([r12(2); -r12(1)])/norm(r12); % evade/ overtake to the righ
elseif R12(2)>=0 % make somthing for this case (random implemented)
    if V2(2)<=0
       e = ([r12(2); -r12(1)])/norm(r12); % evade/ overtake to the right
    else
        e = ([-r12(2); r12(1)])/norm(r12); %evade/ overtake to the left
    end
else % do exactly the oppposite
    if V2(2)<=0
        e = ([r12(2); -r12(1)])/norm(r12); % evade/ overtake to the right
    else
        e = ([-r12(2); r12(1)])/norm(r12); %evade/ overtake to the left
    end
end
% Notice: This algorithm works properly for evading and overtaking
% siuations such that the pedestrians do not make the same decision (
% evade/ overtake in the same direction). A better approach, for instante,
% might be based on reaction-decision-making: One pedestrian decide to
% evade/overtake and the other pedestrians occupies the situation and reacts
% on the neighbour's movement. However, This model is succiciently exact
% for high pedestrian flow systems and provides even in a 2-pedestrian-
% interaction-system good results.

% With the coordinate transformation is is also possible to detect if a
% collision can be excluded:
if X2(1)<X1(1) % neighbour is behind me (I can not see him!!)
    distance0 = -1; % set distance0 to a any invalid value
elseif (X2(2)>0 && V2(2)>0) || (X2(2)<0 && V2(2)<0) % neighbour moves away
    distance0 = -1;
elseif (X2(1)>0 && V2(2)>0) || (X2(1)<0 && V2(2)<0) % neighbour moves away
end

[Theta, Delta_t] = is_collision(x1, x2, v1, v2, distance0);

% Now derive a force that describes the interaction behavior
% > the bigger the expected time until collision uccures is, the bigger the
%   influence becomes
A1 = 6000/(density^2+1);
```

```
if Theta == 1
    % maximum Amplitude depends linearly on number of force influences
    % the force potential is assumed to be a function of Delta_t with a
    % weakly exponentially decreasing potential and without cut-off.
    f = A1*exp(-0.001*Delta_t.^2) .*e;
elseif norm(v1) == 0 % special case if two pedestrians are blocked
    f = A1.*e;
else
    f = [0;0];
end

if isnan(norm(f))
    warning('force is NaN')
    f=[0;0] ;
end


end
```

9.10 force_comfortable_zone.m

```
function [f] = force_comfortable_zone(v1, v2, r12, fac)
%==========================================================================
% This force tries to separate pedestrians from each other ("clear" the
% comfortable zone from neighbours). The force acts continuoulsly on all
% pedestrians, even during "overlapping".
% Note that this force should be depending on the pedestrian density (this
% correlation is considered in the compute_force function)
%--------------------------------------------------------------------------
% Input:    > velocities v1, v2
%           > distance between pedestrian 1 and 2
%           > weighting factor fac
%==========================================================================
f = [0;0];
cos_beta = v1'*v2/(norm(v1)*norm(v2)); % angle between velocities

% calculate the angle between v1 and r12
if norm(v1)~=0
    cos_varphi = r12'*v1/(norm(r12)*norm(v1));
else
    cos_varphi=0;
end

cos_alpha = v1'*r12/(norm(v1)*norm(r12));
if cos_alpha <-0.5
    return % sharp cut-off
end

% compute the force: distinguish overtaking and evading situation
lambda = 100;
% pedestrians are not influenced by pedestians walking behind them
f_varphi = (lambda+(1-lambda)*(1-cos_varphi)/2)/lambda;
f_Deltav = norm(v1-v2);        % velocities have an influence on the force
e = (-r12)/(norm(r12));        % direction: force separets pedestrian
r12 = norm(r12);
if  cos_beta>=0          % pedestrians walk in the same direction
    cut_off = 1;       % no influence at r>cut_off
    B1 = 500;            % amplitude at r=0
```

```
    f_r12 = exp(-0.7*r12.^2) .*heaviside(-r12+cut_off); % force with smooth
cut_off
else % pedestrians walk against each other
    cut_off = 1;      % no influence at r>cut_off
    B1 = 800;           % amplitude at r=0
    f_r12 = exp(-0.7*r12.^2) .*heaviside(-r12+cut_off); % force with smooth
cut_off
end

f = fac*B1*f_varphi*f_r12*f_Deltav.*e;
if isnan(norm(f))
    warning('force is NaN')
    f=[0;0] ;
end


end
```

9.11 force_reach_target.m

```
function [f, location] = force_reach_target(p, location, Grid)
%===========================================================================
% First of all, the pedestrian should understand the situation in which he
% is, i.e he has to to do the following things (in order of importante)
%   (1) Check whether the next target point is visible. If yes, go to this
%   point.
%   (2) Check whether the current target is visible (the pedestrian might be
%   walkt backwards). If yes, go back to the previous target point
% pedestrian tries to reach his target with a preferable velocity v0
% (which is restricted with vmax and vmin). The force that
% correct the velocity direction points from the pedestrian to the target.
%---------------------------------------------------------------------------
%Input:     > number of pedestrian p
%           > location: counter in the vector that saves the target points
%           > Grid
% Output:   > firc t
%           > location
%===========================================================================

global target_radius with_middle_line with_roundabout ax bx Nx Ny r ...
    with_traffic_lights

% time constant (reaction time) for the "approach-target"-force
% This parameter can be also interpreted as an agressivity indicator
tau = 0.005;

vmin = p.v_boundaries(1);
vmax = p.v_boundaries(2);

if norm(p.target_position(:,location)-p.target_position(:,end)) ~= 0
    % check if target point is visible
    if Grid(p.target_indices(1,location), p.target_indices(2,location)) == inf
        error('target point inside boundary')
    end

    % Decide how much the pedestrian should approach the target point
    % the following values are estimated using graphical interpretation
```

```
    if with_middle_line && norm(p.target_position(:,location) - 
p.target_position(:,end-1))==0
        approach_target_distance = (bx-ax)*Nx/4; % approach second last target 
as near as possible
    elseif with_roundabout && location~=1 && norm(p.target_position(:,location)
- p.target_position(:,end-1))~=0 && p.density>0
        % do not walk to the center of the circle. This will cause a total 
blocking situation.
        approach_target_distance = 1;
    elseif with_traffic_lights && location==1
        approach_target_distance = (bx-ax)*Nx/4;
    else
        approach_target_distance = target_radius;
    end
else
    approach_target_distance = target_radius;
end


% take care of multiple updating of the target position
if norm(p.target_position(:,location)-p.target_position(:,end))~=0 && 
location~=1
    loc_current = norm(p.position - p.target_position(:,location));     % 
distance to current target
    visibility = is_visible(p.indices(1), p.indices(2),p.target_indices(1, 
location+1), p.target_indices(2,location+1), Grid);
    ready_to_update = (loc_current<approach_target_distance) && visibility;
elseif location==1 && norm(p.target_position(:,location)-
p.target_position(:,end))~=0
    loc_current = norm(p.position - p.target_position(:,location));     % 
distance to current target
    visibility = is_visible(p.indices(1), p.indices(2),p.target_indices(1, 
location+1), p.target_indices(2,location+1), Grid);
    ready_to_update = loc_current<approach_target_distance && visibility;
else % location = last target
    ready_to_update = false; % no more targets
end

% Check if the pedestrian can focus the the next target point
if with_roundabout && location~=1 && norm(p.target_position(:,location)-
p.target_position(:,end))~=0 ...
        && norm(p.target_position(:,location)-p.target_position(:,end-1))~=0 
...
        && norm(p.target_position(:,location)-p.target_position(:,end-2))~=0
    loc_next = norm(p.position - p.target_position(:,location+1));      % 
distance to next target
    ready_to_update = ready_to_update && (loc_next<2*approach_target_distance);
end

% approach the current target as near possible. If the target visible, than
% update to the next target.
if ready_to_update
    location = location+1;
end

% Sometimes the pedestrian walks backwards and the current target becomes
% suddenly unvisible. In that case, switch the previous target.
% Include also the critical cases, where the center of the pedestrian can
% see the target but some border-points of the pedestrian do not (this
% special case casues blocking pedestrian on edge points)
if location>1
    if is_visible(p.indices(1), p.indices(2), p.target_indices(1, location), 
p.target_indices(2,location), Grid)==0
```

```
        location = max(location-1,1); % pedestrian can not see current target
    elseif norm(p.velocity)<0.1 % now check if pedestrian is blocked
        visibility = true; % first estimation
        % 4 iterations around the center point (be a little bit
        % conservative and add the factor 1.4)
        for i= round([-1:2:1].*r/Ny*1.4)
            for j= round([-1:2:2].*r/Nx*1.4)
                if is_visible(p.indices(1)+i, p.indices(2)+j,
p.target_indices(1, location), p.target_indices(2,location), Grid)==0
                    visibility = false; % corrected estimation
                end
            end
        end
        if ~visibility % pedestrian can see target but something prevent him
from moving
            location = max(location-1,1); % go back and try again
        end
    end
end

% get current target
r_target = p.target_position(:,location);

% If no other forces act on the pedestrian, he will walk with its
% preferable velocity v0. He tries to keep his velocity constant at this
% value.
v0 = p.initial_velocity;
v_opt = v0*(r_target - p.position)/norm(r_target - p.position);

% get sure that the pedestrian is not borred (restriction v_opt > vmin)
% and not overstrained (restriction v_opt < vmax)
v_opt = v_opt/norm(v_opt) * max(min(norm(v_opt), vmax), vmin);

f = (v_opt - p.velocity)/tau;

if isnan(norm(f)) && norm(p.velocity)==0
    if isnan(norm(f))
        warning('force is NaN')
    end
    f=[0;0] ;
end

end
```

9.12 generate_grid.m

```
function [Grid] = generate_grid(type)
%==========================================================================
% The grid is represented by a matrix Grid. To acces the matrix use
% Grid(i,j). In this simulation, two kind of position representation are
% used: (1) indices-notation {i,j} with coordinatesystem {i,j} and
% (2) position-notation {j*Nx,i*Ny} with coordinate system {x,y}. The
% two coordinat systems are connetced as follows: x->j and y->-i.
% Notice that some applications requieres a symmetric grid (M=N)!
% If a pedestrian should be able to see an obstacle, this obstacle should
% be implemented in the grid represented as inf-numbers.
%--------------------------------------------------------------------------
% Input:    > type (type of the intersection):
```

```
% output:   > Grid: two dimensional NxM grid with intersection boundaries
%===========================================================================

global ax ay bx by N M L_grid W_grid Nx Ny roundabout_radius ...
    intersection_radius with_roundabout with_inner_radius ...
    with_middle_line W_line with_traffic_lights lx ly ...
    number_of_traffic_lights_red traffic_light_time traffic_light__time_closed
...
    with_4_roundabouts super_radius roundabout_4_radius W_street

Grid = zeros(N,M); % all grid points assumed to be empty (0)

% stuct with caracteristic boundary points
%   > profile: contains border points of the profile (counter cklockwise)
%   > streigh_line: contains borer lines of objects
%   > outer_circle: roundabounds
%   > corner: single corner points; often part of the profile
%   > straight_lines: like profile lines but not linked with other lines
%   > traffic_lights: like straight_lines but with additional restrictions
profile = [];
outer_circle = [];
inner_circle = [];
corner = [];
straight_lines = [];
traffic_lights = [];
bounds = struct('profile', profile, 'outer_circle', outer_circle,
'inner_circle', ...
    inner_circle, 'corner', corner, 'straight_lines', straight_lines,...
    'traffic_lights', traffic_lights);

save('variables.mat', 'bounds')

% Check if all parameters are given. The follosing parameters are not
% neccessary for the simulation. However, sometomes an error will arise if
% they are not defined.
if isempty(roundabout_radius) || isempty(intersection_radius)
    warning('roundabout_radius or intersection_radius are not defined yet')
elseif isempty(traffic_light_time) ||isempty(traffic_light__time_closed) ||
isempty(number_of_traffic_lights_red)
    warning('traffic_light_time, traffic_light__time_closed or
number_of_traffic_lights_red is not defined yet')
elseif with_traffic_lights && ~with_middle_line
    warning('with_middle_line is set to false. Pedestrian will be able to walk
around the traffic lieghts')
end




switch type
    %-------------------------------------------------------------------------
    case 1 % straight path
        % size of the grid
        L_grid = 10;        % [m]
        W_grid = 4;        % [m]
        % grid dimesions
        N = W_grid/Nx;
        M = L_grid/Ny;
        % Width of the streets
        W_street = 3; %[m]
        % characteristic points of the intersection
        ay = floor(0.5*N - W_street/(2*Ny));
```

```
        by = floor(0.5*N + W_street/(2*Ny));

        % allocate grid
        Grid(1:ay,1:M) = inf;
        Grid(by:N,1:M) = inf;

        % Save characterisitc points
        profile = [[by; 1],[by; M],[ay; M],[ay; 1]];
    %--------------------------------------------------------------------------
    case 2 % 4 brach intersection
        %------------------------------
        % GENERAL IMPLEMENTATION
        %------------------------------
        % size of the grid
        L_grid = 20;        % [m]
        W_grid = 20;        % [m]
        % grid dimesions
        N = W_grid/Nx;
        M = L_grid/Ny;
        % Width of the streets
        W_street = 4; %[m]
        % characteristic points of the intersection
        ay = floor(0.5*N - W_street/(2*Ny));
        by = floor(0.5*N + W_street/(2*Ny));
        ax = floor(0.5*M - W_street/(2*Nx));
        bx = floor(0.5*M + W_street/(2*Nx));

        % allocate grid
        Grid(1:ay,1:ax) = inf;
        Grid(by:N,1:ax) = inf;
        Grid(1:ay,bx:M) = inf;
        Grid(by:N,bx:M) = inf;

        if intersection_radius<=roundabout_radius
            error('roundabout <= intersection_radius')
        end

        % Save characterisitc points
        profile = [[by; 1],[by; ax],[N; ax],[N; bx],[by; bx],[by; M],...
            [ay; M],[ay; bx],[1; bx],[1; ax],[ay; ax],[ay; 1] ];
        corner = [[by; ax],[by;bx],[ay;bx],[ay;ax]];

        %------------------------------
        % INVERSE ROUNDABOUT
        %------------------------------
        if with_inner_radius
            kx = round(sqrt((intersection_radius/Nx)^2 - ((by-ay)/2)^2));
            ky = round(sqrt((intersection_radius/Ny)^2 - ((bx-ax)/2)^2));
            k1 = [by;M/2-kx];
            k2 = [N/2+ky;ax];
            k3 = [N/2+ky;bx];
            k4 = [by;M/2+kx];
            k5 = [ay;M/2+kx];
            k6 = [N/2-ky;bx];
            k7 = [N/2-ky;ax];
            k8 = [ay;M/2-kx];

            profile = [[by; 1],k1,k2,[N; ax],[N; bx],k3,k4,[by; M],[ay;
M],k5,...
                k6,[1; bx],[1; ax],k7,k8,[ay; 1] ];
            corner = [k1, k2, k3, k4, k5, k6, k7, k8];
```

```
            for i = -round(intersection_radius/Ny) :
round(intersection_radius/Ny)
                for j = -round(intersection_radius/Nx) :
round(intersection_radius/Nx)
                    % inner circle
                    if (i*Ny)^2+(j*Nx)^2 <= intersection_radius^2 &&
(i*Ny)^2+(j*Nx)^2 >= roundabout_radius^2
                        Grid(N/2+i, M/2+j) = 0;
                    end
                end
            end
            inner_circle = [intersection_radius; N/2; M/2];
        end


        %------------------------------
        % ROUNDABOUT
        %------------------------------
        if with_roundabout
            for i = -round(roundabout_radius/Ny) : round(roundabout_radius/Ny)
                for j = -round(roundabout_radius/Nx) :
round(roundabout_radius/Nx)
                    % outer circle (roundabout)
                    if (i*Ny)^2+(j*Nx)^2 < roundabout_radius^2
                        if Grid(N/2+i, M/2+j) == inf
                            error('roundabout radius is too big')
                        end
                        Grid(N/2+i, M/2+j) = inf;
                    end
                end
            end
            outer_circle = [roundabout_radius; N/2; M/2];
        end

        %------------------------------
        % 4 ROUNDABOUTS
        %------------------------------
        if with_4_roundabouts
            kappa = super_radius/Nx;

            for mid_i = N/2-kappa : kappa : N/2+kappa
                for mid_j = M/2-kappa : kappa : M/2+kappa
                    if mid_i==mid_j || (mid_i==N/2-kappa && mid_j==M/2+kappa)
|| (mid_i==N/2+kappa && mid_j==M/2-kappa)
                        continue
                    end
                    for i = -round(roundabout_4_radius/Ny) :
round(roundabout_4_radius/Ny)
                        for j = -round(roundabout_4_radius/Nx) :
round(roundabout_4_radius/Nx)
                            % outer circle (roundabout)
                            if (i*Ny)^2+(j*Nx)^2 < roundabout_4_radius^2
                                if Grid(mid_i+i, mid_j+j) == inf
                                    error('roundabout radius is too big')
                                end
                                Grid(mid_i+i, mid_j+j) = inf;
                            end
                        end
                    end
                end
            end

            circle1 = [roundabout_4_radius; N/2; N/2+kappa];
```

```
            circle2 = [roundabout_4_radius; N/2; N/2-kappa];
            circle3 = [roundabout_4_radius; N/2+kappa; N/2];
            circle4 = [roundabout_4_radius; N/2-kappa; N/2];
            outer_circle = [outer_circle, circle1, circle2, circle3, circle4];
        end



        %-------------------------------
        % MIDDLE LINES
        %-------------------------------
        if with_middle_line
            wx = W_line/Nx;
            wy = W_line/Ny;
            if with_inner_radius % length of a line
                lx = round((ax-kx+(bx-ax)/2)*0.9);
                ly = round((ay-ky+(by-ay)/2)*0.9);
            else
                lx = round(ax*0.8);
                ly = round(ay*0.8);
            end

            Grid(N/2-wx/2:N/2+wx/2,1:lx) = inf;
            Grid(N/2-wx/2:N/2+wx/2,M-lx:M) = inf;
            Grid(1:ly,M/2-wx/2:M/2+wx/2) = inf;
            Grid(N-ly:N,M/2-wx/2:M/2+wx/2) = inf;

            % lines pointing from center of the intersection to the outside
            line_1 = [[N/2;lx], [N/2;1]];
            line_2 = [[N-ly;M/2],[N;M/2]];
            line_3 = [[N/2;M-lx],[N/2;M]];
            line_4 = [[ly;M/2],[1;M/2]];

            straight_lines = zeros(2,8);
            straight_lines(:,1:2) = line_1;
            straight_lines(:,3:4) = line_2;
            straight_lines(:,5:6) = line_3;
            straight_lines(:,7:8) = line_4;

            % interprete the beginning of each line as a corner boundary
            corner = [corner,
[line_1(:,1),line_1(:,1),line_1(:,1),line_1(:,1)]];
        end

        %-------------------------------
        % TRAFFIC LIGHTS
        %-------------------------------
        % A traffic light is in this simulater represented as a line
        % interpreted as both, traffic light and profile. It forces
        % the pedestrian to set v=0 or to prevent the pedestrians to walk
trough.
        % During the simulation traffic_lights will be changed.
        if with_traffic_lights
            traffic_light_1 = [[by;lx],[N/2;lx]];       % brach 1
            traffic_light_4 = [[N-ly;bx],[N-ly; M/2]];  % brach 4
            traffic_light_2 = [[ay;M-lx],[N/2;M-lx]];   % brach 2
            traffic_light_3 = [[ly;ax],[ly; M/2]];      % brach 3
            lights = [traffic_light_1, traffic_light_2, traffic_light_3,
traffic_light_4];
            save('variables.mat', 'lights', '-append')

            switch(number_of_traffic_lights_red)
                case 3
```

```
                    traffic_lights = [traffic_light_1, traffic_light_2,
traffic_light_3];
                    branch_number = [1, 2, 3]; % traffic light appears at
branch 1 2 and 3
                case 2
                    traffic_lights = [traffic_light_1, traffic_light_2];
                    branch_number = [1, 2]; % traffic light appears at branch 1
2
                case 1
                    traffic_lights = [traffic_light_1];
                    branch_number = [1]; % traffic light appears at branch 1
                otherwise
                    error('number_of_traffic_lights_red is greater than 3')
            end
            save('variables.mat', 'branch_number', '-append')

            time_where_branch_is_closed = 0; % helper
            save('variables.mat', 'time_where_branch_is_closed', '-append')

        end


    %-----------------------------------------------------------------------
end

bounds.profile = profile;
bounds.corner  = corner;
bounds.inner_circle = inner_circle;
bounds.outer_circle = outer_circle;
bounds.straight_lines = straight_lines;
bounds.traffic_lights = traffic_lights;
save('variables.mat', 'bounds', '-append') % save struct for later access

end




% Notice that there a 2 possibilities to detect a boundary point:
%   1) numerically: Iterate over Grid in a small range around the current
%   pedestroan and locate all grid points. This use of this method is not
%   recommanded, since ...
%       - the numercial effort increases due to aditional iterations
%       - the acuracy depend in the parameter Nx and Nx and those values
%       can not be decreased more then 0.05 (numerical effort)
%   2) analytically: Save all characterstic points in a struct and
%   derive for each pedestrian the nearest grid point. This method ...
%       + is very efficient
%       - but requieres a lot of additional implementations.
% We decided for the second method after we failed with the first one
% due to not sufficiently high accuracy.
```

9.13 generate_pedestrian.m

```
function [pedestrian] = generate_pedestrian()
%=======================================================================
% Each pedestrian is represented as a circle with radius R. The pedestrian
```

```
% tries to approach the final target as follows (in order of imporatance)
%   (1) reach current target point: That is a point on the preferred
%   trajectory that a pedestrian would like to walk on. All target point
%   are set manually. The pedestrian can choice his final point, but
%   the preferred trajectoriy is fixed (minimization of the distance) for
%   all times after he has chosen (he can not recognize wheter this
%   trajectory lead to a minimized time effort). However, the pedestrian
%   does not have to walk on this trajectory.
%   (2) evade other pedestrians: Pedestrian can see each ohter. They can
%   communicate with each ohter to find an optimal solution that does not
%   lead to a collision.
%   (3) evade boundary points: pedestrians can receive all obstacle. They
%   do not try to evade them actively (this is the task of how setting the
%   target points on the correct position) but the also can not pass
%   trhough them.  A objecle that is crashed acts like a filter that
%   cancles the velocity component perpendicular to the tangent. Traffic
%   lights are interpreted as boundary points.
%   (4) reach final target: The final target is the beginning of a branch.
%   Each pedestrian can decide where this final target is (randomization)
%-------------------------------------------------------------------------
% Output:   struct that contains
%           > direc: current direction
%           > v: current velocity (vector)
%           > indices: indices i,j  on the grid
%           > position: x,y components of the position
%           > mass
%           > v_boundaries: contains maximum and minimum velocity
%           > cell_dim: contains dimensions of the reaction cell
%           the reaction cell is a square-cell around the pedestrian; its
%           "interaction area" contains the eight neighbour cell around the
%           that cell
%           > target_indices: array that contains the intices (i,j) of the
%           coodinates of target
%           > target_position: array that contains the position (x,y) of
%           the coodinates of target
%           > location: actual target that the pedestrian has focused
%           > gen_location: location where the pedestrian is generated
%           > force: motivation to change the current direction
%           > is_updated: 1 pedestrian is already updated, 0 if not
%           > density: indicatator for density
%           > dist_critical, traffic_lights: see pedestrian_boundary_distance
%=========================================================================


global v_min v_max M N Ncell Mcell Nx Ny interaction_distance

v_min = 0;          % [m/s]
v_max = 4;          % [m/s]

% initialize random behavior (uniform distribution)
uniform=@(a, b) a + (b-a)*rand;

v0 = uniform(1.2,1.6); % Wikipedia: humans tend to walk at about 1.4 m/s
vmax = v0*2.5;
vmin = v0*0.3;

% subdivide N in and M in cells with zize dist*dist m^2
Ncell = ceil(N/(interaction_distance/Ny));
Mcell = ceil(M/(interaction_distance/Nx));


% initialize pedestrian
v               = [0;0];
```

```
v_boundaries    = [vmin vmax];
indices         = zeros(2,1);
x               = zeros(2,1);
mass            = uniform(50,90); % max varies between 50kg and 90kg
cell_dim        = [Ncell Mcell];  % assumed to be equal for all pedestrians
target_indices  = 0;              % will be a matrix after initialization
target_position = 0;              % will be a matrix after initialization
location        = 1;
gen_location    = 0;
force           = [0;0];
density         = 0;
dist_critical   = [];
traffic_lights  = [];
generation_time   = 0;

% create pedestrian
pedestrian = struct('velocity', v, 'initial_velocity', v0, 'mass', mass, ...
    'indices', indices, 'position', x, 'v_boundaries', v_boundaries, ...
    'cell_dim', cell_dim, 'target_indices', target_indices, 'targe_position',
...
    target_position , 'location', location, 'gen_location', gen_location,
'force', force,...
    'density', density, 'dist_critical', dist_critical, ...
    'traffic_lights', traffic_lights, 'generation_time', generation_time);

end
```

9.14 grid2cell.m

```
function [Cell_Grid] = grid2cell(pedestrian_saver, Ncell, Mcell)
%============================================================================
% This function uses a cell storage method of Grid to reduce the numerical
% effort when computing interaction between several pedestrians. The Grid
% is subdevided into cells and all pedestrian are assigned to one cell.
%----------------------------------------------------------------------------
% Input:    > pedestrian_saver
%           > dimensions of the cell Ncell and Mcell
% Output:   > Cell_Grid: Matrix that subdivides the grid into cells
%============================================================================

global N M

Cell_Grid = cell(Ncell, Mcell);
p = length(pedestrian_saver);

for id=1:p
   current =  pedestrian_saver{id};
   % determine indices of the cell
   celli = floor(Ncell*(current.indices(1)-0.9)/N)+1;
   cellj = floor(Mcell*(current.indices(2)-0.9)/M)+1;
   % save current pedestrian into Cell_Grid
   Cell_Grid{celli, cellj} = [Cell_Grid{celli, cellj} id]; % "push back"
end


end
```

9.15 is_available.m

```
function [is_available] = is_available(Grid, r, i_new, j_new)
%==========================================================================
% This function verifies wether the updating position is free or occupied
% (with a pedestrian of a grid point)
% ATTENTION: This function strongly deaccelerates the updating speed! If
% possible use is_available2().
%--------------------------------------------------------------------------
% Input:    > Grid
%           > radius of the pedestrian r
%           >updating indices (i_new, j_new)
% Output:   > is_available (1: position is available, 0: position is
%           not available)
%==========================================================================

global N M Nx Ny

is_available = 1;
security = 2;

% iterate over a square with width 4*r and middle point (i_new, j_new)
for i=i_new-2*r*security/Ny : i_new+2*r*security/Ny
    for j=j_new-2*r*security/Nx : j_new+2*r*security/Nx
        if i>0 && j>0 && i<N && j<M % get sure that position exists
            % take care of neighbour pedestria
            if (Grid(i, j) > 0) && Grid(i,j)~=inf
                % calculate vector pointing from updating position to
                % neighbour position
                vec = [j*Nx; i*Ny] - [j_new*Nx; i_new*Ny];
                dist = norm(vec);
                if dist<(2*r)*security
                    is_available = 0;
                    %disp('warning: update was not possible -- pedestrian
crossing')
                    return
                end
            % take care of boundary points
            elseif i>i_new-r/Ny && i<i_new+r/Ny && j>j_new-r/Nx && j<j_new+r/Nx
&& Grid(i, j)==inf
                is_available = 0;
                %disp('warning: update was not possible -- boundary crossing')
                return
            end
        end
    end
end



end
```

9.16 is_available2.m

```
function [availability, position_velocity] =
is_available2(potential_blocking_indices, i_new, j_new, v)
%==========================================================================
% This function check whether the updating position is available or or not.
%--------------------------------------------------------------------------
% Input:     > potential_blocking_indices: indices of neighbour pedestrians
%            > i_new, j_new: indices of updating position
%            > velocity v
% Output:    > availability (true or false)
%==========================================================================

global Nx Ny r

security = 1.1;
local_availability = true;
availability = true;
position_velocity = [];

[n, m] = size(potential_blocking_indices);
if m==0 || v==0
    return
end

for i=1:m
    % Extract indicex and velocity of neighbour pedestrian
    i_blocking = potential_blocking_indices(1, i);
    j_blocking = potential_blocking_indices(2, i);
    v_blocking = potential_blocking_indices(3:4,i);

    % Check if the updating position is around the neighbour location
    r12 = norm([(j_blocking-j_new)*Nx; (i_blocking-i_new)*Ny]);
    local_availability = r12 > 2*r*security;
    % Exlude boundary points from this procedure: Sometimes pedestrian
    % move backward after its generating and an other pedestrian will
    % generated on the current one.
    if r12-2*r<0
        continue
    elseif local_availability==false
        availability = false;   % remember that normal uptading is not possible
        % save position and velocity of "critical" neithbour
        position_velocity = [position_velocity, [j_blocking*Nx; i_blocking*Ny;
v_blocking] ];
        local_availability = true; % make ready for next iteration
    end
end


end
```

9.17 is_collision.m

```
function [is_collision, Delta_t] = is_collision(x1, x2, v1, v2, distance0)
%==========================================================================
% is_collision() predicts a collision (0 or 1) between two pedestrians
%based on the current positions and velocities.
%--------------------------------------------------------------------------
% Input:     > position and velocity of pedestrian 1 and pedestrian 2
%            > distance0: distance between 1 and 2
```

```
% Output:   > is_collision: 1 if collision is expected, 0 if collision is
%           excluded
%           > Delta_t: time to the expected collision
%==========================================================================
global r

dt = 0.1; %[s]
Delta_t = 0;
is_collision=0;
distance = 0;
k=0;

if distance0 <= 2*r || norm(v1) == 0 || distance0>12 % collision occured
    return % leave the function without executing something
end

% start "mini"-simulation
while distance < distance0
    k=k+1;
    %update position
    x1 = x1+dt*v1;
    x2 = x2+dt*v2;

    distance = distance + dt*norm(v1) + dt*norm(v2);

    % check if collision occured
    if norm(x1-x2)<2*r
        is_collision = 1;
        Delta_t = dt*k;
        break
    end
end
% Of course, the numerical effort for this simulation is much more bigger
% than compute a potential collision point. Since the colission time
% Delta_t have a big influence on the force in our model and the exact
% tedermination of this time is complicated (many special cases have to be
% considered), we decided for this conclusive solution


end
```

9.18 is_visible.m

```
function [is_visible] = is_visible(i, j , i_neighbour, j_neighbour, Grid)
%==========================================================================
% This function checks wehter the neighbour (or any other point) is visible
% for the current pedestrian or not.
% ATTENTION: This function slightly deaccelerates the updating speed!
%--------------------------------------------------------------------------
% Output:   > is_visible (true: neighbour is visible, false: neighbour is
%           not visible)
%==========================================================================

global N M
is_visible = true; % assume neighbour is visib


% calculate unit vector pointing from current to neighbour
```

```
e = [j_neighbour; i_neighbour] - [j; i];
% iterate from current pedestrian along vec until neighbour is reached and
% check wheter boundary points were crossed or not
if (e(1)>= 0 && e(2) >= 0) || (e(1)>= 0 && e(2) < 0) % x>=0
    cos_alpha = [1; 0]'*e/norm(e); % compute angle between x-axes and vec
    for y = 0 : abs(i_neighbour-(i)) % iterate over y-axis
        if y==0
            y = sign(i_neighbour-(i)); % skip fist iteration
        end
        % l = y/tan(alpha) = y/sin(alpha)*cos(alpha)
        l = abs(y) / sqrt(1-cos_alpha^2) * cos_alpha; % iterate along x-axis
        x = round(l);
        % Before using the vector y, adjust its sign
        y = sign(i_neighbour-(i))*y;
        if i+y<1 || i+y>N || j+x<1 || j+x>M || isnan(x)
            continue % outside of the grid; no boundary point was found
        elseif i+y==i_neighbour || j+x==j_neighbour
            return % neighbour is reached; no bloundary point was found
        elseif Grid(i+y, j+x) == inf
            is_visible = false; % boundary point was found
            return           % abort the function
        end
        % make y ready for the next itearation
        y = abs(y);
    end

elseif (e(1)< 0 && e(2) >= 0) || (e(1)< 0 && e(2) < 0) % x<0
    cos_alpha = [-1; 0]'*e/norm(e);
    for y = 0 : abs(i_neighbour-(i));
        if y==0
            y = sign(i_neighbour-(i)); % skip fist iteration
        end
        l = abs(y) / sqrt(1-cos_alpha^2) * cos_alpha;
        x = -round(l);
        y = sign(i_neighbour-(i))*y;
        if i+y==i_neighbour || j+x==j_neighbour
            return
        elseif i+y<1 || i+y>N || j+x<1 || j+x>M || isnan(x)
            continue
        elseif Grid(i+y, j+x) == inf
            is_visible = false;
            return
        end
        y = abs(y);
    end

elseif e(2)==0 % special case
    for x = 0: abs(j_neighbour-(j))
        if x==0
            x = sign(j_neighbour-(j));
        end
        x = sign(j_neighbour-(j))*x;
        if j+x==j_neighbour
            return
        elseif i<1 || i>N || j+x<1 || j+x>M
            continue
        elseif Grid(i+y, j+iter_j+x) == inf
            is_visible = false;
            return
        end
        x = abs(x);
    end
end
```

```
end
```

9.19 main.m

```
clear all
close all
clc
format compact


%==========================================================================
% Parameters
%==========================================================================

global p_max interaction_distance dt t_end update_type intersetction_type ...
    w target_radius Nx Ny r color_set roundabout_radius intersection_radius ...
    with_roundabout with_inner_radius with_middle_line W_line
traffic_light_time ...
    with_traffic_lights traffic_light__time_closed number_of_traffic_lights_red
...
    with_4_roundabouts super_radius roundabout_4_radius


% INTERSECTION
%--------------------------------------------------------------------------
% grid resolution (distance between two grid points)
Nx = 0.05;                  % [m] in x-direction
Ny = 0.05;                  % [m] in y-direction

% Width of lines
W_line = 0.1;               % [m]


% SIMULATION
%--------------------------------------------------------------------------
% Two updating distinct types are available:
%   1: euler forward method: efficient
%   2: low-storage runga-kutta method: more exactly
update_type = 1;

dt          = 0.05;         % [s] integration step
t_end       = 50;          % [s] simulation time

with_graphic = true;        % if true a graphical output is obtained
with_video   = false;        % if true a video called
'pedestrian_simulation.avi is generated


% PEDESTRIANS
%--------------------------------------------------------------------------
r = 0.2;                    % "radius" of each pedestrian
w = 0.003;                  % possibility that pedestrian is generatated

% Each pedestrian is located in a cell with size interaction_distance^2.
% All pedestrian recognize neighbour in this cell and in 8 neighbour cells.
% Increase this value if just a few pedestrians are located on the grid,
```

```
% decrease this value if the pedestrian densitiy is high (improve the
% updating speed).
interaction_distance = 4;    % [m]

% pedestrians approch the target until target_radius and then they follow
% the next target. If target_radius is chosen to small, the trajectories
% become edgy and the dynamic might be lost. If target_radius is chosen to
% big the update speed decreases pereceptible.
target_radius = 3;          %[m]

% Two color sets are available:
%   1: color depends on generating location: Use this for a clear overall view
%   2: color depends on speed: Use this to determine "blocking-sources"
color_set = 2;


% COMMON CONTROLL PARAMETERS
%--------------------------------------------------------------------------
% maximum number of pedestrians that are allowed on the grid
p_max = 10000;

% intersection type
%   >> 1: straight path (long and narrow): Use this to adjust the
%   parameters of the forces and to derive expressions for cu-off values
%   >> 2: intersection with 4 braches: Use this for simulation
intersetction_type = 2;

% applications:
%   >> with_roundabout: use a roundabout in the middle of the intersection
%   >> with_inner_radius: extend the intersection wih an outer circle
%   ("inverse roundabout")
%   >> with_middle_line: draw lines in the center of each brack to force
%   the pedestrian to walk in predfined sectors
%   >> with_traffic_lights: Use traffic lights. One traffic light of four
%   is green. Use number_of_traffic_lights_red to define how many traffic
%   lights are red at the same time (number between 1 and 3).
with_roundabout     = true;
with_4_roundabouts  = false;
with_inner_radius   = true;
with_middle_line    = true;
with_traffic_lights = true;
roundabout_radius   = 1.4;            % [m] radius of the single reoundabout
roundabout_4_radius = 0.2;           % [m] radius of the four roundabouts
super_radius        = 2.5;            % [m] used to adjust the position of the
4 roundabouts
intersection_radius = 4;             % [m]
traffic_light_time  = 7;            % [s] change traffic lights after this time
traffic_light__time_closed = 3;      % [s] time in which all brached are closed
number_of_traffic_lights_red = 2;    % [s] number of red traffic light (1,2,3)
% Important notes:
%   (1) all features are only applicable if intersection_type is chosen
%    to be 2
%   (2) Set roundabout_radius and intersection_radius to suitable values
%   even if they are not used! Set traffic_light_time even if
%   with_traffic_lights is set to false.
%   (3) If with_traffic_lights=true make sure that with_middle_line=true is
%   set as well. Ohterwise, the pedestrians will bypass the traffic lights.
%   (4) do not use with_roundabouts and with_4_roundabouts at the same
%   time.



%==========================================================================
```

```
% Simulation
%==========================================================================
video_name = 'simulation_video_1.avi';
[measurements, velocity_distribution] = simulate(with_graphic, with_video,
video_name);



%==========================================================================
% Output, Results
%==========================================================================
fprintf('\n')
fprintf('\n')
disp('measurements:')
fprintf('\n')

% the following result are obtained:

% (1) avarage velocity
v_avarage = measurements(1)

% (2) avarage of preferred velocity
v0_avarage = measurements(2)

% (3) averaged number of pedestrians walking slower than 0.2 m/s. This value
% indicates the loss of dynamic.
low_speed_index = measurements(3)

% (4) averaged velocity of all pedestrian with preferred velocity >1.5. This
% value compared with v_avarage answers the question whether "stressing"
% does help or not.
high_speed_index = measurements(4)

% (5) avarage time effort to reach the final target.
avarage_time_effort = measurements(5)

% (6) number of pedestrians that have not left the grid after the end of
% the simulation
number_of_pedestrians_left = measurements(6)

% (7) velocity_distribution is a NxM grid that contains in
% velocity_distribution(i,j) the avarage velocity at the indices {i,j}
velocity_distribution;
figure
imagesc(velocity_distribution)
```

9.20 pedestrian_boundary_distance.m

```
function [dist_min, dist_critical, traffic_lights] =
pedestrian_boundary_distance(x)
%--------------------------------------------------------------------------
% Input:    > pedestrian and Grid
% Output:   > dist_critical: cell array with dist_crit (vector
%           pointing from center of current pedestrian to nearest boundary
%           point)and boundary type (string with type of boundary point)
%           > distance: vector pointing from nearest boundary point to
%           pedestrian (center)
%--------------------------------------------------------------------------
```

```
global r Nx Ny N M with_roundabout with_middle_line with_traffic_lights ...
    with_4_roundabouts
distance_saver = zeros(3, 4);
load('variables', 'bounds') % load struct profile

dist_critical = cell(1,3);

C = [round(x(2)/Ny); round(x(1)/Nx)]; % indices of current pedestrian

%---------------------------------------------------------------------------
% (1) CONSIDER PROFILE BOUNDS
profile = bounds.profile;
[np, mp] = size(profile);
distance = [N; M];

for k=1:mp-1
    A = profile(:,k);
    B = profile(:,k+1);
    distance = find_min_distance(A,B,C,distance,false);
end

if with_traffic_lights
    traffic_lights = bounds.traffic_lights;
    [nt, mt] = size(traffic_lights);
    for k = 1:2:mt-1
        B = traffic_lights(:,k);
        A = traffic_lights(:,k+1);
        [distance] = find_min_distance(A,B,C,distance,false);
    end
end

distance_saver(:,1) = [norm(distance); distance];


%---------------------------------------------------------------------------
% (2) CONSIDER OUTER CIRCLE BOUNDS
if with_roundabout || with_4_roundabouts
    distance = [N; M];
    outer_circle = bounds.outer_circle;
    [ns, ms] = size(outer_circle);
    for k=1:ms
        circle = outer_circle(:,k);
        R = circle(1)/Nx;       % radius
        Mid = circle(2:3);      % center point (indices)
        dist = (norm(C-Mid)-R)*(C-Mid)/norm(C-Mid); % vector pointing from
boundary to pedestrian
        if norm(dist)<norm(distance)
            distance = dist;
        end
    end
    distance_saver(:,2) = [norm(distance); distance];
else
    distance_saver(:,2) = [norm([N; M]); [N; M]];
end

%---------------------------------------------------------------------------
% (4) CONSIDER INNER CIRCLE
% This step is not neccessary since the profile desribes the circle well
% (for big sufficiently big radius)

%---------------------------------------------------------------------------
% (3) CONSIDER CORNER BOUNDS
```

```
corner  = bounds.corner;
[nc, mc] = size(corner);
distance = [N; M];
for k=1:mc
    dist = C - corner(:,k); % vector pointing from boundary point to pedestrian
    % find local minimum
    if norm(dist) < norm(distance)
        distance = dist;
    end
end
distance_saver(:,3) = [norm(distance); distance];

%------------------------------------------------------------------------
% (4) CONSIDER STRAIGHT LINES
if with_middle_line
    straight_lines = bounds.straight_lines;
    [nl, ml] = size(straight_lines);
    distance = [N; M];
    for k = 1:2:ml-1
        line = straight_lines(:,k:k+1);
        A = line(:,1);
        B = line(:,2);
        [distance] = find_min_distance(A,B,C,distance,true);
    end
    distance_saver(:,4) = [norm(distance); distance];
end

%------------------------------------------------------------------------
% (5) CONSIDER TRAFFIC LIGHTS
if with_traffic_lights
    distance = [N; M];
    for k = 1:2:mt-1
        A = traffic_lights(:,k);
        B = traffic_lights(:,k+1);
        [distance] = find_min_distance(A,B,C,distance,false);
    end

    if norm(distance) == norm([N; M]) % nothing found
        traffic_lights = false;
    else
        traffic_lights = -[distance(2)*Nx; distance(1)*Ny];
    end

else
    traffic_lights = false;
end



%------------------------------------------------------------------------
% FIND GLOBAL MINIMUM
dist_min = [N; M];
for k=1:4
    distance = distance_saver(1,k);
    if distance>0 && distance<norm(dist_min);
        dist_min = distance_saver(2:3,k);
        p = k;
    end
end

if isempty(p)
    dist_type = 'empty';
    dist_critical{1} = 'empty';
```

```
elseif p==1
    dist_type = 'profile';
elseif p==2
    dist_type = 'outer_circle';
elseif p==3
    dist_type = 'corner';
elseif p==4
    dist_type = 'straight_lines';
end

% vector pointing from boundary point to pedestrian
critical = false;
dist_min = [dist_min(2)*Nx; dist_min(1)*Ny];
if norm(dist_min)<=3*r
    critical = true;
end

if critical
    dist_critical{2} = -dist_min; % vector pointing from pedestrian to boundary
point
    dist_critical{1} = dist_type;
else
    dist_critical{1} = 'empty';
end



end
```

9.21 simulate.m

```
function [measurements, velocity_distribution] = simulate(with_graphic,
with_video, video_name)
%==========================================================================
% simulate() calls all function for each iteration step.
% Input:    > with_graphic (1, 0)
%           > with_video (1, 0)
%           > video_name (1, 0)
% Output:   > measurements
%           > velocity_distribution
%==========================================================================

global dt t_end intersetction_type N M

iter = t_end/dt;        % number of iterations
Grid = generate_grid(intersetction_type); % prepare grid
p=0;                    % number of created pedestrians
pedestrian_saver = [];
measurements = zeros(iter, 5);
velocity_distribution = zeros(N,M);

if with_video
    hFig = figure;
    set(hFig, 'Position', [500 60 750 750])
else
    figure
end
```

```
disp('========================================')
disp('Start simulation')
disp('----------------------------------------')

fprintf('\n')
disp(['progress: ',num2str(0), 's of ', num2str(t_end), 's' ])


for it=1:iter
    time = it*dt;
    save('variables.mat', 'time','-append')
    [Grid, pedestrian_saver, deleted_pedestrians, m1, m2, m3, time_effort] =
update(Grid, pedestrian_saver, p);
    % update measurements
    measurements(it,1:3) = m1;
    measurements(it,4) = m2;
    measurements(it,5) = time_effort;
    velocity_distribution = velocity_distribution + m3;

    if with_graphic || with_video % with graphical output

        draw_obstacle()

        % check if pedestrians left the grid and allocate new pedestrians
        [pedestrian_saver, Grid] = update_pedestrians(pedestrian_saver,
deleted_pedestrians, Grid);
        p = length(pedestrian_saver);

        for k=1:p % draw all pedestrians
            current =  pedestrian_saver{k};
            draw_pedestrian(current.indices(1), current.indices(2),
current.velocity, current.gen_location)
        end

        if with_video
            % store video frames (may rise a warning in MATLAB 2011a)
            A(it)=getframe(gcf);
        end

        pause(dt)

    else % without graphical output
        [pedestrian_saver, Grid] = update_pedestrians(pedestrian_saver,
deleted_pedestrians, Grid);
        p = length(pedestrian_saver);
    end

    % Show progress after each 100 timesteps
    if mod(it, 100) == 0
       disp(['progress: ',num2str(time), 's of ', num2str(t_end), 's' ])
    end

end

if with_video
    fprintf('\n')
    disp('Save vidoe frame to file. This may take some minutes...')
    movie2avi(A, video_name,'compression','None', 'fps',7, 'quality', 100);
    fprintf('\n')
    disp('Video sucessfully generated.')
end
```

```
% take the avarge of all measurements
fprintf('\n')
disp('Compute measurements')

% measuremetns 2
m2 = measurements(:,4);
m2_new = [];
for i=1:length(m2)
    if m2(i)~=inf % find non marked values
        m2_new = [m2_new m2(i)];
    end
end
m2 = sum(m2_new)/length(m2_new); % avarage the values

% time effort
time_effort = measurements(:,5);
number_of_zeros = sum(time_effort==0); % exclude zero-marked values
time_effort = sum(time_effort)/(iter-number_of_zeros);

% measurements 1
m1 = measurements(:,1:3);
m1_new = [];
for i=1:size(measurements,1)
    if m1(i,1)~=inf % find non marked values
        m1_new = [m1_new; m1(i,:)];
    end
end
m1 = sum(m1_new)/size(m1_new,1); % avarage the values

% save all measurements
measurements = [m1'; m2; time_effort; p]; % reallocation

% velocity distribution
velocity_distribution = velocity_distribution/iter; % avarage distribution


fprintf('\n')
disp('Measurements succesfully stored')

fprintf('\n')
disp('----------------------------------------')
disp('Simulation is done')
disp('========================================')


end
```

9.22 traffic_lights_regulator.m

```
function [] = traffic_lights_regulator()
%==========================================================================
% Helper function for ragulating the trafic light dynamic:
% After each traffic_light_time all traffic lieght change to red. Aftre
% additional traffic_light__time_closed some other traffic light are
% changing to green.
%==========================================================================
```

```
global  traffic_light_time traffic_light__time_closed
number_of_traffic_lights_red

load('variables', 'time', 'time_where_branch_is_closed')


% close all branches
if mod(time,traffic_light_time+traffic_light__time_closed)==0
    time_where_branch_is_closed = time;
    save('variables.mat', 'time_where_branch_is_closed', '-append')
    load('variables', 'lights', 'bounds')
    bounds.traffic_lights = lights;
    save('variables.mat', 'bounds','-append')


% change traffic light
elseif time == time_where_branch_is_closed + traffic_light__time_closed
    load('variables', 'lights', 'branch_number', 'bounds')

    % empty the saver
    bounds.traffic_lights=[];

    % update branch numbers
    branch_number = branch_number+(4-number_of_traffic_lights_red);
    for i=1:length(branch_number)
        if branch_number(i)>4
            branch_number(i)= branch_number(i)-4;
        end
        % reallocation
        bounds.traffic_lights = [bounds.traffic_lights,
[lights(:,2*branch_number(i)-1:2*branch_number(i)) ]];
    end
     save('variables.mat', 'bounds','-append')
     save('variables.mat', 'branch_number','-append')
end


end
```

9.23 update.m

```
function [Grid, pedestrian_saver, deleted_pedestrians, measurements_1,
measurements_2, measurements_3, time_effort] ...
    = update(Grid, pedestrian_saver, p)
%===========================================================================
% This fnction updates the position if the velocity is given.
%---------------------------------------------------------------------------
% Input:    > Grid
%           > pedestrian_saver
%           > p: number of pedestrians
% Output:   > updated Grid, updated pedestrian_saver
%           > deleted_pedestrians: array that contains indices of
%           pedestrians which have left the grid
%           > avaraged measurements (1, 2 and 3)
%===========================================================================

global N M Nx Ny with_traffic_lights
load('variables', 'time');
```

```
deleted_pedestrians = [];
measurements_1 = zeros(p,3);
measurements_2 = [];
measurements_3 = zeros(N,M);
time_effort = [];

% Note: Since some pedestrian can react on decisions of other pedestrians
% (the react on the forces), this function can not be involved in the loop
% below (However, this would would be much more efficient)

[pedestrian_saver, potential_blocking_indices] = update_force(pedestrian_saver,
Grid);

if with_traffic_lights
    traffic_lights_regulator()
end


for k=1:p
    current = pedestrian_saver{k}; % extract current pedestrian
    i = current.indices(1);        % extract its position
    j = current.indices(2);
    % compute the force that acts on the current pedestrian
    f = current.force;
    [x, v] = update_position(f, current.position, current.velocity,
current.mass); % update position
    distance = x-current.position; % vector pointing from current position to
updating position
    if ~strcmp(current.dist_critical{1},'empty')
        % check if current postion is critical (any boundary points around?)
        [x1, v1] = evade_boundary(current.dist_critical, distance,
current.position,x, v);
    else
        x1 = x;
        v1 = v;
    end
    % Take care of traffic lights
    if with_traffic_lights
        [x, v] = evade_traffic_lights(current.traffic_lights, distance,
current.position, x1, v1);
    else
        x = x1;
        v = v1;
    end

    % check if updating position is available
    [availability_pedestrian, neighbour] =
is_available2(potential_blocking_indices{k}, floor(x(2)/Ny), floor(x(1)/Nx),
norm(v));
    if ~availability_pedestrian
        [x, v] = evade_pedestrian(x,current.position, neighbour(1:2,:), v1,
neighbour(3:4,:));
    end

    % new grid coordinates
    i_new = floor(x(2)/Ny);
    j_new = floor(x(1)/Nx);

    % is the pedestrian located on the grid?
    if (i_new <= N && j_new <= M) && (i_new > 0 && j_new >0 )
        % save id on the grid
        Grid(i_new,j_new) = k;
        % save new state
```

```
        pedestrian_saver{k}.indices = [i_new; j_new];
        pedestrian_saver{k}.position = x;
        pedestrian_saver{k}.velocity = v;
        % release old position
        Grid(i,j) = 0;

        % take some "measurements" and save them
        measurements_1(k,1) = norm(v);
        measurements_1(k,2) = norm(current.initial_velocity);
        measurements_1(k,3) = norm(v)<0.2;
        if current.initial_velocity>=1.5 % fast pedestrians
            measurements_2 = [measurements_2; norm(v)];
        end
        measurements_3(i_new, j_new) = norm(v);
    % Special case for pedestrians that have reached its target
    elseif ((i_new > N) || (j_new > M) || (i_new <= 0) || (j_new <= 0))
        deleted_pedestrians = [deleted_pedestrians k];
        Grid(i,j) = 0;
        % take measuremetns here. Get also sure that the pedestrian do not
        % walk  backwards after beeing generated (this distorts the
        % measurements). Exclude those cases.
        if (time-current.generation_time>4)
            time_effort = [time_effort, time-current.generation_time];
        end
    else
        warning('updating position is NaN') % tell me we have problems here
    end
    % If grid point is occupied then do not update. This, of course, sould
    % influence the velocity. Since this situation rarely occures,
    % the effect of deacceleration is not considered here.
end


if p~= 0
    % avarage_velocity(1): avarage velocity
    % avarage_velocity(2): avarage of initial velocity (preferable velocity)
    measurements_1 = sum(measurements_1)/p;
else
    measurements_1 = [inf;inf;inf];   % mark this case with infinity values
end

if ~isempty(measurements_2)
    measurements_2 = sum(measurements_2)/length(measurements_2);
else
    measurements_2 = inf;       % mark this case with infintiy value
end

if ~isempty(time_effort)
    time_effort = sum(time_effort)/length(time_effort);
else
    time_effort=0;              % mark this case with zero value
end
```

9.24 update_force.m

```
function [pedestrian_saver, potential_blocking_indices] =
update_force(pedestrian_saver, Grid)
%==========================================================================
```

```
% This function computes the force for each pedestrian
%-----------------------------------------------------------------------------
% Input:     > pedestrian_saver
%            > Grid
% Output:    > updated pedestrian_saver
%            > potential_blocking_indices: indices of pedestrian in a small
%            range around an other pedestrian
%=============================================================================
global N M Ncell Mcell interaction_distance r

fmin=0;
fmax = 500;

potential_blocking_indices = cell(1,length(pedestrian_saver));

% subdivide the grid into cells
Cell_Grid = grid2cell(pedestrian_saver, Ncell, Mcell);

for id=1:length(pedestrian_saver)
    indices_saver = [];

    density = 0;% is a kind of density-indicator
    current = pedestrian_saver{id};
    v = current.velocity;
    x = current.position;
    i = current.indices(1);
    j = current.indices(2);

    f = zeros(2,1);

    % compute indices of the cell in which the current pedestrian is located
    celli = floor(Ncell*(current.indices(1)-0.9)/N)+1;
    cellj = floor(Mcell*(current.indices(2)-0.9)/M)+1;

    % Calulate indices of "relevant" neighbour cells (note that pedestrians
    % are assumend to be blind for everything that happens behind them)
    accurancy = 1;      % round without decimal accuracy
    if abs(round(v(1)/accurancy)*accurancy) >
abs(round(v(2)/accurancy)*accurancy)
        % "main direction": [1; 0];
        i_stop  = 1;
        i_start = -1;
        j_stop  = max(sign(v(1)), 0);
        j_start = j_stop-1;
    elseif abs(round(v(1)/accurancy)*accurancy) <
abs(round(v(2)/accurancy)*accurancy)
        % "main direction": [0; 1];
        i_stop  = max(sign(v(2)), 0);
        i_start = i_stop-1;
        j_stop  = 1;
        j_start = -1;
    else % iterate over 3 neighbour cells
        % " main direction" [1; 1]/sqrt(2)
        i_stop  = max(sign(v(2)), 0);
        i_start = i_stop-1;
        j_stop  = max(sign(v(1)), 0);
        j_start = j_stop-1;
    end

    % iterate over 9 neighbour cells (exlude all cells where no cell
    % interaction is expected)
    for di = i_start:i_stop
        for dj = j_start:j_stop
```

```
            % compute neighbour indices (no enfocing of periodic boundary
            % condition -> this would not make much sense when trying to model
            % in intersection!)
            if celli+di==0 || celli+di==Ncell+1 || cellj+dj==0 ||
cellj+dj==Mcell+1
                % the neighbour cell does not exist; assume an empty cell
                % instead (non periodic boundary conditions)
                neighbour_cell = [];
            else
                celli_neighbour = celli+di;
                cellj_neighbour = cellj+dj;
                neighbour_cell = Cell_Grid{celli_neighbour, cellj_neighbour};
            end

            % Influence of the density (density measures the number of
            % forces actung on the current pedestrian. It is therefore an
            % indicator for future density)
            fac1 = min(max(1-current.density/5, 0), 1);

            for k=1:length(neighbour_cell)
                neighbour_id = neighbour_cell(k);
                neighbour = pedestrian_saver{neighbour_id};
                i_neighbour = neighbour.indices(1);
                j_neighbour = neighbour.indices(2);
                distance = norm(current.position-neighbour.position);

                % Influence of the relative differnce velocity
                % ( 1-(v-v0)/v0 is an indicator for the actual density )
                fac2 = min(max(2-
norm(current.velocity)/current.initial_velocity, 0), 1);

                % Intoduce a sharp cut-off
                %   > to have a better acceleration behavior for high
                %   density crowd flows
                %   > to improve the updating speed
                cut_off = max(interaction_distance*abs(fac1*fac2), r+0.3);
                % avoid interaction  with "myself" and apply cut-off
                if norm([i j])-norm([i_neighbour,j_neighbour])~=0 && distance
<cut_off

                    x_neighbour = neighbour.position;
                    v_neighbour = neighbour.velocity;
                    % check if neighbour is visible (notice that the following
                    % computations holds for "boundary-cells" as well

                    % avoid collision with pedestrians (iterate over each
                    % current-neighbour-pair only once)
                    f3 = force_collision_pedestrian(x, x_neighbour, v,
v_neighbour, current.density);
                    f=f+f3;

                    % ensure comfortable zone (redurce the force if many
neighbours are around)
                    f4 = force_comfortable_zone(v, v_neighbour, x_neighbour-x,
min(1/(fac1*fac2^2+0.1),10));
                    f = f+f4;

                    % Update density value
                    density = density+1;

                    % Finally, save the indices of the neighbour pedestrian
                    % (also save its velocity)
```

```
                    indices_saver = [indices_saver [i_neighbour; j_neighbour;
v_neighbour]];
                end
            end
        end
    end

    % approach target
    [f1, location] = force_reach_target(current, current.location, Grid);
    f = f+f1;

    % avoid collision with boundary
    [f2, dist_critical, traffic_lights] = force_collision_boundary(v, x);
    f = f + f2;

    % restrict the force
    if norm(f)>0
        f = f/norm(f) * max(min(norm(f), fmax), fmin);
    end
    % update force of the current pedestrian
    pedestrian_saver{id}.force=f;
    pedestrian_saver{id}.location = location;
    pedestrian_saver{id}.density = density;
    pedestrian_saver{id}.dist_critical = dist_critical;
    pedestrian_saver{id}.traffic_lights = traffic_lights;
    potential_blocking_indices{id} = indices_saver;

end

% Notice that we did not implement any following or friction forces. Those
% effects are generated automatically!

end
```

9.25 update_pedestrian.m

```
function [pedestrian_saver, Grid] = update_pedestrians(pedestrian_saver,
deleted_pedestrians, Grid)
%==========================================================================
% This function delets pedestrians that have left the grid and generates
% new pedestrians on the boundary points.
%--------------------------------------------------------------------------
% Input:    > pedestrian_saver: cell array that contains all pedestrians
%           > deleted_pedestrians: array that contains indices of
%           pedestrians which have left the grid
%           > Grid
% Output:   > updated pedestrian_saver
%==========================================================================

global intersetction_type p_max ay by ax bx w Nx Ny r N M
load('variables', 'time');

%--------------------------------------------------------------------------
% delete all pedestrians that left the grid
%--------------------------------------------------------------------------
if ~isempty(deleted_pedestrians) % check if any pedestrians left the grid
    % make sure that the order of deleting elements is correct
    % (delete great numbers first)
```

```
    deleted_pedestrians = sort(deleted_pedestrians,'descend');
    for k=1:length(deleted_pedestrians)
        pedestrian_saver(deleted_pedestrians(k)) = [];
        % notice that the id is changed here -> requiers a new update of id
    end
end
p_new = length(pedestrian_saver);

%-------------------------------------------------------------------------
% add new pedestrians which enter into the grid
%-------------------------------------------------------------------------
p = 0;    % number of created pedestrians
switch(intersetction_type)
    case {1,2} % left brach (1)
    for i = ay+floor((by-ay)/2)+1 : by-floor(r/Ny)-1
        if  (w>rand) && length(pedestrian_saver)<p_max
            % note that this function requires a lot of iterations and
            % should therfore only be used if necessairy
            if is_available(Grid, r, i, 1)==1
            p = p+1;
            pedestrian = generate_pedestrian();
            % allocate pedestrian:
            pedestrian.gen_location = 1; % number of the branch where
pedestrian is generated on
            pedestrian.velocity = pedestrian.initial_velocity*[1;0];
            pedestrian.indices = [i;1];
            pedestrian.position = [1*Nx; i*Ny];
            pedestrian.generation_time = time;
            % target indices
            [target] = final_target(1);
            pedestrian.target_indices = [target(1,:); target(2,:)];
            pedestrian.target_position = [target(2,:).*Nx; target(1,:).*Ny];
            pedestrian_saver{p_new + p} = pedestrian; % save pedestrian
behavior
            Grid(i,1) = p_new+p; % save id
            % each pedestrian has an unique id for each time step.
            end
        end
    end
end

switch(intersetction_type)
    case {1, 2} % right branch (2)
    for i = ay+floor(r/Nx)+1 : by-floor((by-ay)/2)-1

        if  (w>rand) && length(pedestrian_saver)<p_max
            if is_available(Grid, r, i, M)==1
            p = p+1;
            pedestrian = generate_pedestrian();
            pedestrian.gen_location = 2;
            pedestrian.velocity = pedestrian.initial_velocity*[-1;0];
            pedestrian.indices = [i;M];
            pedestrian.position = [M*Nx; i*Ny];
            pedestrian.generation_time = time;
            [target] = final_target(2);
            pedestrian.target_indices = [target(1,:); target(2,:)];
            pedestrian.target_position = [target(2,:).*Nx; target(1,:).*Ny];
            pedestrian_saver{p_new + p} = pedestrian;
            Grid(i,1) = p_new+p;
            end
        end
    end
end
```

```
switch(intersetction_type)
    case{2}  % upper branch (3)
    for j = ax+floor(r/Ny)+1 : bx-floor((bx-ax)/2)-1
        if  (w>rand) && length(pedestrian_saver)<p_max
            if is_available(Grid, r, 1, j)==1
            p = p+1;
            pedestrian = generate_pedestrian();
            pedestrian.gen_location = 3;
            pedestrian.velocity = pedestrian.initial_velocity*[-1;0];
            pedestrian.indices = [1;j];
            pedestrian.position = [j*Nx; Ny];
            pedestrian.generation_time = time;
            [target] = final_target(3);
            pedestrian.target_indices = [target(1,:); target(2,:)];
            pedestrian.target_position = [target(2,:).*Nx; target(1,:).*Ny];
            pedestrian_saver{p_new + p} = pedestrian;
            Grid(1,j) = p_new+p;
            end
        end
    end
end

switch(intersetction_type)
    case{2} % lower branch (4)
    for j = ax+floor((by-ax)/2)+1 : bx-floor(r/Ny)-1
        if  (w>rand) && length(pedestrian_saver)<p_max
            if is_available(Grid, r, N, j)==1
            p = p+1;
            pedestrian = generate_pedestrian();
            pedestrian.gen_location = 4;
            pedestrian.velocity = pedestrian.initial_velocity*[-1;0];
            pedestrian.indices = [N; j];
            pedestrian.position = [j*Nx; N*Ny];
            pedestrian.generation_time = time;
            [target] = final_target(4);
            pedestrian.target_indices = [target(1,:); target(2,:)];
            pedestrian.target_position = [target(2,:).*Nx; target(1,:).*Ny];
            pedestrian_saver{p_new + p} = pedestrian;
            Grid(N,j) = p_new+p;
            end
        end
    end
end


end
```

9.26 update_position.m

```
function [x, v] = update_position(f, x, v, m)
%==========================================================================
% This function computs the position and velocity using one of two updating
% types
%--------------------------------------------------------------------------
% Input:    > force f
%           > x (position), v (velocity), m (mass)
% Output:   > updated position x and new velocity v
```

```
%==========================================================================
global dt update_type

if update_type ==1 % euler forward
    a = f/m;        % compute acceleration
    x = x+dt*v;     % update position
    v = v+dt*a;     % update velocity

elseif update_type == 2 % low-storage runga-kutta
    a=f/m;
    c = [0 -17/32 -32/27]';
    b = [1/4 8/9 3/4]';
    q1 = zeros(2,1);
    q2 = zeros(2,1);

    for i=1:2
        q1 = c(i)*q1+dt*v;
        q2 = c(i)*q2+dt*a;
        x = x + b(i)*q1;
        v = v + b(i)*q2;
    end

end

end
```