Report for exercise 1 from group K

Tasks addressed:	5
Authors:	SONJA KRAFFT (03681252) LUDWIG-FERDINAND STUMPP (03736583) TIMUR WOHLLEBER (03742932)
Last compiled:	2021–11–03
Source code:	https://gitlab.lrz.de/00000000014A9334/mlcs-ex1-modeling-crowds

The work on tasks was divided in the following way:

SONJA KRAFFT (03681252)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
LUDWIG-FERDINAND STUMPP (03736583)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
TIMUR WOHLLEBER (03742932)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%

1 Introduction

Goal of this exercise

The subsequent sections take a closer look at modelling crowds using a self-developed simulation environment. Since real crowds and their interactions with each other are complex dynamic systems, some simplifications have to be made to reduce the complexity. A commonly used approach for this is the use of a cellular automaton, which divides the simulated world into spatially and temporally discretized steps. In the following report, we are going to discuss what the pipelines of such a prototype might look like and whether they are sufficient to approximate realistic scenarios. In particular, the proof of reality is done by performing and evaluating standardized tests according to the Guidelines for Microscopic Escape Analysis (RiMEA) [2].

Setting up the Python environment

Based on our own experience with Python and the large selection of high-performance third-party libraries, we chose Python as our programming language. Table 1 shows an overview about the required packages. In order to be able to replicate the results, one can install any version of these libraries that is compatible with the here listed version numbers. In addition to the here listed packages, we are using tkinter[1] to provide a graphical user interface (GUI). Some operating systems have tkinter not installed by default which requires to install it manually.

python	3.9.5
pandas	1.3.4
numpy	1.21.3
matplotlib	3.4.3
autopep8	1.5.7
flake8	4.0.1
mypy	0.910
jupyterlab	3.2.1
VSCode	1.61.2

Table 1: Python, libraries and code editor.

General work approach

For this work, we choose a combination of Jupyter[2] notebooks and standard Python scripts to combine the strengths of both worlds. In the early stages of the project, notebooks are used primarily for initial prototyping applications. As the project progresses, the code snippets are more and more bundled into proper functions and moved into Python modules that one can later import. Until the end, these notebooks remain useful to document the work for each subtask.

[1] `https://docs.python.org/3/library/tkinter.html`
[2] `https://jupyter.org/`

2 Individual Tasks

Report on task 1/5: Setting up the modelling environment

Objective

In the first task, we set up our implementation and simulation environment with following requirements:

1. Basic visualization.
2. Adding pedestrians in cells.
3. Adding targets in cells.
4. Adding obstacles by making certain cells inaccessible.
5. Movement of pedestrians during simulation.
6. Change the scenario without changing the code.

General approach

We decide to split the functionalities into two separate classes, one class to implement the basic functionalities of a cellular automaton, the `CellularAutomaton` , and another class to offer a graphical user interface, the `GUI` . The `GUI` makes use of the interface provided by the `CellularAutomaton` class and represents the functionalities in a visually pleasing way for the user.

Theory on cellular automata

A cellular automata tries to simplify the continuous simulation problem by discritizing both in time and space. It is based on a two dimensional grid consisting of cells, each with one state $X_i := \{E, P, O, T\}$.

- E: Empty cell
- P: Cell occupied by one pedestrian
- O: Cell occupied by one obstacle
- T: Target cell

During the simulation process, a cellular automata makes use of a simple, discrete-time update scheme with constant time shifts, in order to move the pedestrians to the target cell. (See [3] [4])

CellularAutomaton class

The `CellularAutomaton` class provides the low level interface for creating, simulating and visualizing a cellular automaton. It is divided into the following steps:

1. Create a cellular grid consisting of cells with state {empty, target, pedestrian, obstacle}, either one-by-one or by reading from a `scenario.csv` file.
   $>>>$ # *manual scenario creation*
   $>>> myCellularAutomaton = CellularAutomaton(grid\_size = (1, 9))$
   $>>> myCellularAutomaton.add\_obstacle(pos\_idx = (0, 8))$
   $>>> myCellularAutomaton.add\_pedestrian(pos\_idx = (0, 0),\ speed\_desired = 1.0)$
   $>>> myCellularAutomaton.add\_target(pos\_idx = (0, 7))$

```
>>> # create from scenario file
>>> myCellularAutomaton = fill_from_scenario_file("scenario.csv")
```

2. Run the simulation which propagates the positions of the pedestrian forward until they reach a target cell. One can either choose to simulate an explicit number of iterations, or to simulate until no further change in the simulated pedestrian positions is going to happen.
```
>>> # simulate n iterations
>>> myCellularAutomaton.simulate_next_n(n = 1)
>>> # simulate until no change
>>> myCellularAutomaton.simulate_until_no_change()
```

3. Visualize the state of the cellular grid at either the current iteration or show all past simulated iterations. One can also visualize the utility grid that is used to steer the pedestrian towards its target.
```
>>> # visualize current grid
>>> myCellularAutomaton.visualize_state_grid()
>>> # visualize all iterations
>>> myCellularAutomaton.visualize_simulation()
>>> # visualize utility grid
>>> myCellularAutomaton.print_utilities()
```

4. Analyze the past simulation by inspecting the history of all pedestrians, divided into pedestrians that are still on the grid and pedestrians that have reached a target.
```
>>> # not finished pedestrians
>>> myCellularAutomaton.pedestrians_history
>>> # finished pedestrians
>>> myCellularAutomaton.finished_pedestrians_history
```

The interested reader can take a look at the Jupyter notebook `task_1.ipynb`[3] which fully describes the functionality of the `CellularAutomaton`class with hands-on examples.

Definition of a scenario

For the first task it is necessary to consider which parameters are essential for modeling a real crowd. For this purpose, the term "*scenario*" is explained in the following section. A scenario is a set of parameters sufficient to define the interaction of a pedestrian with obstacles or other pedestrians until a target cell is reached. Already from these simple requirements, it can be deduced that each cell in the locally discretized world can be described by a tuple of cell states.

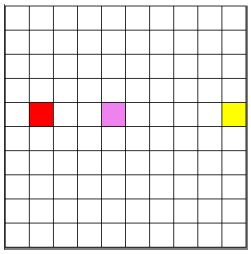


Figure 1: This figure illustrates an example of a 10x10 grid with a pedestrian marked as a red cell at the postion (4, 1) who tries to arrive at the target cell marked in yellow at (4, 9). Between the pedestrian and target is an obstacle cell highlighted in purple at position (4, 4).

[3] `https://gitlab.lrz.de/00000000014A9334/mlcs-ex1-modeling-crowds/-/blob/main/task_1.ipynb`

The association of all cells and, likewise, the total size of the simulated world is to be defined in the following as *grid size*. This is a parameter that determines the number of cells in X and Y direction. Figure 1 shows an example of a scenario in which a grid is set up starting from the upper left corner. The position of pedestrians, obstacles and target cells can be defined by a coordinate tuple in the coordinate system of the discretized grid. For the interaction of the pedestrians in the simulated world, it is also necessary to define a movement speed and to assign an identifier for evaluation purposes.

Reading in a scenario from a configuration file

Creating a simulation scenario is the start of every simulation. This essentially poses the two challenges that one wants to quickly simulate a large number of different scenarios and that one can quickly make adjustments in each of these scenarios. In order to avoid the need to add pedestrians, obstacles or target cells manually using the `CellularAutomaton` class, the outsourcing of the scenario definition in a separate CSV file seemed to be the most resource-saving and convenient approach. Figure 2 shows how such a scenario definition could look like. We have supplemented the import of the prepared scenario files with modules with which it is still possible to manipulate after the import, such as adding further objects. This is especially important for the debugging process.

grid_size	initial_position_obstacles	position_target_zone	pedestrian_id	initial_position_pedestrian	avg_velocity_pedestrian
(5, 8)	(1, 3)	(2, 6)	1	(2, 1)	1.0
	(1, 4)				
	(2, 4)				
	(3, 4)				
	(3, 3)				

Figure 2: This figure illustrates an example of a scenario CSV file.The first columns defines the grid size from the upper left corner. The following columns define five obstacle cells, one target zone and one pedestrian with the average velocity of one cell travelling velocity per iteration.

Basic visualisation

The visualisation of the simulation results presents different challenges during the development of the simulation tool. The first challenge is to be able to evaluate the simulation modules quickly and with little effort. For this purpose, a command line interface (CLI), as provided by the `CellularAutomaton` class, seems to make the most sense. Such an output is illustrated in Figure 3, for example.

```
[                    ]
[       O  O         ]
[  P      O     T    ]
[       O  O         ]
[                    ]
```

Figure 3: This figure shows an example of a command line interface displaying the scenario as specified as in the CSV File of Figure 2. Empty cells are not displayed. Cells containing a pedestrian are marked with an "P", obstacles with an "O" and target cells with an "T" here.

Graphical user interface (GUI)

Initially, we were convinced by the simplicity of this approach, but the user interface quickly reaches its limits with a large grid size or many iterations and became confusing. At this point in the development process, it seems to make sense to invest time and effort in building a real graphical user interface (GUI). In an incremental process, we gather all the necessary requirements and develop the following components:

- Import of the scenario file
- Colour representation of the scenario
- General overview of the scenario properties
- Function field: simulation commands
- Function field: simulation options

Based on the list of requirements and the limited time frame of the project, we decide to develop the GUI in the same programming language as the simulation calculating modules, as this allows us to reduce the interface effort considerably. In terms of rapid prototyping, we therefore choose to use the Python GUI toolkit *Tkinter*.

The basic structure consists of embedding a canvas element in the root frame and iteratively mapping rectangular cell bodies onto it, starting from the upper left corner with a previously defined cell width and cell height. The cell state read out from the scenarios or in the later course of the simulation is then used to highlight the corresponding cells. The scenario details are mapped via label elements and can be static or dynamic. Dynamic elements are fed with new information in temporal iterations through update functions. In addition, buttons and check boxes enable the triggering of functionalities which are explained in more detail in the following section using Figure 4.

In this figure subarea one (marked with a blue rectangle and the number one) offers the user the possibility to read in a scenario CSV-file via the standard file explorer and,thus,solves the first requirement to change the scenario without the need to carry out any other modifications within the code. The second area visualises the imported scenario in color in the grid and enables a more precise analysis through the overall view of the scenario details. Sub-area three allows you to run an entire simulation. In order to achieve a simpler evaluation of the simulation results, the simple start function was supported by the functionalities "Previous Step", "Next Step" and "Reset Simulation". These steps allow to click step by step through the iterations of the simulation or to reset it to the initial state. Subsection four of the GUI is used to change other simulation-relevant parameters such as obstacle avoidance and target absorption via the two check boxes.

In the sense of a modular structure of this report, the function behind the checkbox "Obstacle Avoidance" is considered in more detail in section of Task 4 and that of the checkbox "Target Absorption" in Task 3.

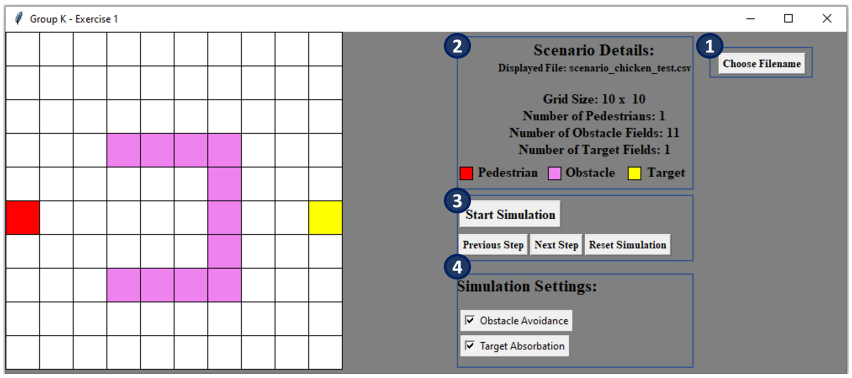


Figure 4: Example of a graphical user interface displaying the "chicken test scenario" in a 10 x 10 grid. Not shown but included in the simulation is that all cells that are part of the pedestrians path are highlighted in red. The numbered blue rectangles have been added during post-processing and are not part of the actual GUI, they are for illustration and reference purposes only.

Report on task 2/5: First steps of a single pedestrian

Objective

The main objective of task 2 is to model and simulate the first steps of a single pedestrian with the developed cellular automaton until reaching a statically defined goal. The fact that we are dealing with a single pedestrian on the entire grid makes it possible to neglect the interaction between several pedestrians. Furthermore, the absence of obstacles solves the issue of not being able to access certain cells. These simplifications allows us to focus on the development of a utility function that initially only has to ensure that the pedestrian walks in the direction of the destination.

Draft of a utility function

Using the simplifications defined in the upper section, a simple utility function based on the euclidean distance to the closest target is defined as a first draft (see equation 1). Iterating over all cells, one can assign values based on the euclidean distance, so that a minimum value results in the target and maximum values for the cells with the greatest distance to the target. Using these values, it is then possible to determine the neighboring cell with the lowest distance to the target at each discrete timestamp t during the simulation. This guides the pedestrian to its goal. Note however, that this air-line distance does not take into account the effect of obstacles on its way.

$$d(c_{ij}, c_{kl}) = \sqrt{(i-k)^2 + (j-l)^2} \tag{1}$$

Scenario details

The scenario consists of a 50x50 grid with a single pedestrian in the location (5, 25) and a target cell at (25, 25). The pedestrian should walk to the target and wait there. For this purpose we define a static scenario CSV-file as specified in Figure 5.

grid_size	initial_position_obstacles	position_target_zone	pedestrian_id	initial_position_pedestrian	avg_velocity_pedestrian
(50, 50)		(25, 25)	1	(5, 25)	1.0

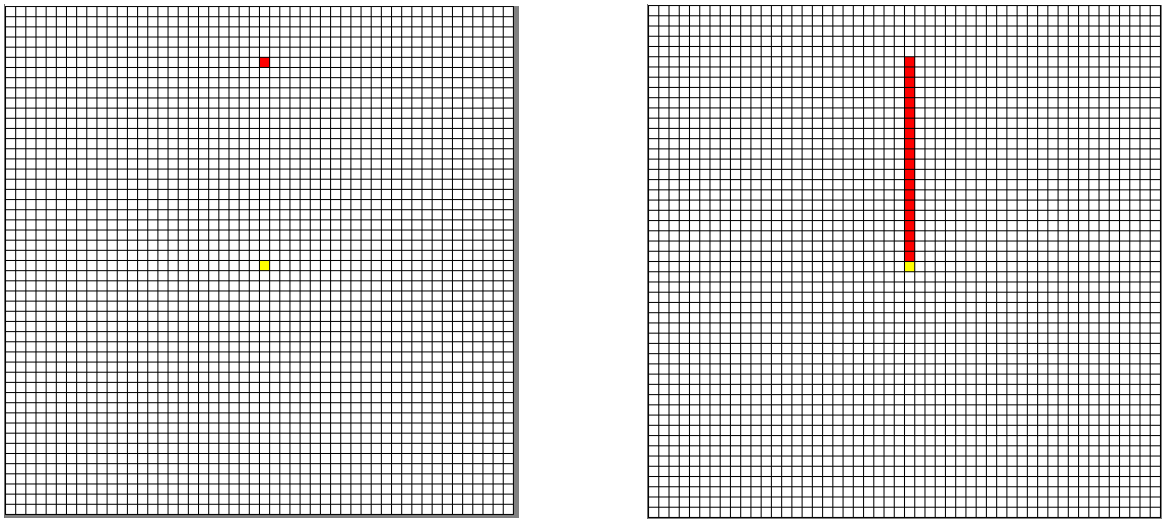
Figure 5: Scenario CSV file for task 2.

Results

The results of the simulation are illustrated in the figures below whereas the left figure Figure 6a shows the grid before the simulation and Figure 6b after running the 25 time steps. One can see that the pedestrian marked as a red rectangle finds his yellow target at the middle within the shortest distance. The red line displays the path travelled by the pedestrian. Note: the index shift of the start and target position is due to the counting of the origin at (0, 0).

Evaluation

The pedestrian successfully finds the shortest and direct path to the target on a straight line. We therefore conclude the first real check of our implementation as successful.



(a) Grid: Pre-Simulation

(b) Grid: Post-Simulation

Figure 6: Implementation of line scenario in cellular grid.

Report on task 3/5: Interaction of pedestrians

Objective

This task aims to investigate how our cellular automaton deals with multiple pedestrians at once, as well as to check whether the pedestrians have approximately equal speed when running diagonally.

Scenario details

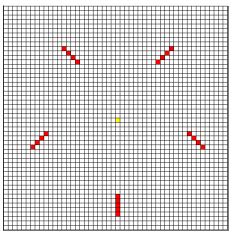
To test and simulate the agents' competitive situations, the grid from task 2 is used and five pedestrians are arranged in a circle around a single target cell. The exact positions of the pedestrians are calculated based on the trigonometric functions of points on a circle to be then be discretized on the grid.

grid_size	initial_position_obstacles	position_target_zone	pedestrian_id	initial_position_pedestrian	avg_velocity_pedestrian
(50, 50)		(25, 25)	1	(46, 25)	1.0
			2	(31,6)	1.0
			3	(9,13)	1.0
			4	(9,37)	1.0
			5	(31,44)	1.0

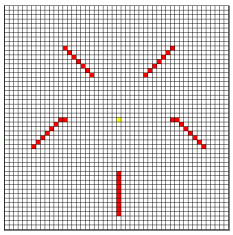
Figure 7: CSV-file configuration for scenario task 3 with five pedestrians around a single target cell.

Simulation results

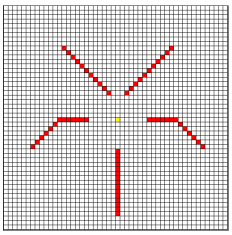
Figure 8 shows the state of the simulation at four different iterations. One can see the pedestrians approach the target all at the same time while finally coming together in the closest neighborhood of the target cell.



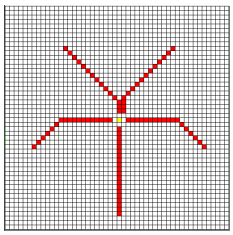
(a) Simulation at time step t=5.



(b) Simulation at time step t=10.



(c) Simulation at time step t=15.



(d) Simulation at time step t=20.

Figure 8: Simulation of the star scenario.

Interaction of pedestrians

In order to avoid collisions between pedestrians, cells occupied by other pedestrians are ignored during the simulation process. Instead of handling the avoidance of pedestrians explicitly, one could also think of achieving a similar but more configurable behaviour by applying a pedestrian-based utility function on top of an already existing obstacle based utility grid. While this is certainly useful for future iterations of this project, we decide to stick to the explicit pedestrian check. Additionally, it is important to note that the position of other pedestrians is not considered during planning as this would dramatically increase the complexity of our simulation algorithm.

Target absorption

A necessary simplification in the simulation is the *"target absorption"* which describes the process that the pedestrians who reach a target cell fall through this target and are thus no longer part of the next iteration. The necessity of this simplification is best explained by the detail breakout Z in the figure below (Figure 9).

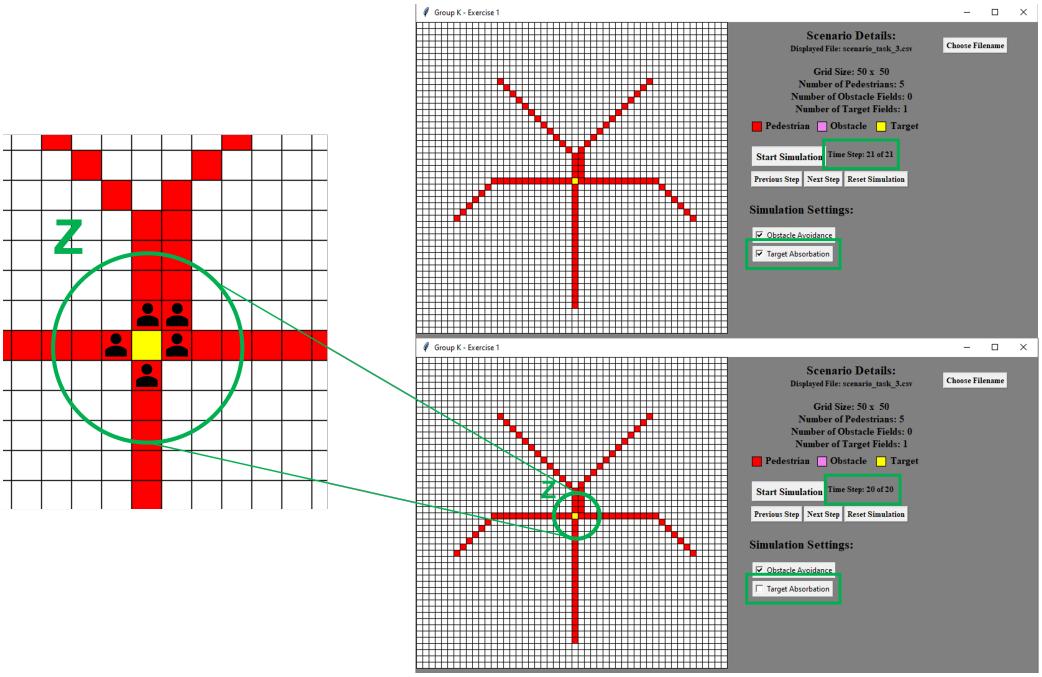


Figure 9: Five pedestrians distributed on a circular line at a distance of 72 degrees so that they all have the same distance to the single yellow target cell in the middle. The upper screenshot shows the GUI with target absorption activated and the lower screenshot with it deactivated. The detail breakout Z is to illustrate the position of the pedestrians around the target. Note: all green markings and the person symbols have been added to a post-processing step and are not part of the actual GUI.

Detail breakout Z represents the final position of the pedestrians with *"Obstacle Avoidance"* deactivated. It quickly becomes apparent that all five pedestrians have positioned themselves around the target. At this point, the problem arises that no more pedestrians can reach the target, unless they are removed after reaching the target. There is no visual difference between the two cases of activated and deactivated *"Target absorption"*, since reaching the target is no longer displayed visually, but it becomes clear in the additional iteration that is necessary when this property is activated in the checkbox.

Achieving equal velocities

Letting pedestrians move diagonally, allows them to move faster than an axes-parallel movement, given that there is no corrective intervention activated. This effect is illustrated in Figure 10.

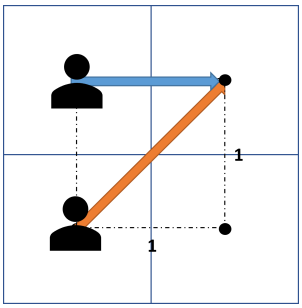


Figure 10: Schematic sketch of two pedestrians in a 2x2 grid world. The upper pedestrian follows the blue path horizontally exactly one unit of length. In order to reach the same destination, the lower pedestrian has to cover the orange path, which is longer by a factor $\sqrt{2}$.

Equal velocity is achieved by first assigning a desired average speed to each pedestrian as well as keeping track of the so far travelled distance on the grid. During the simulation step of our cellular automaton, one needs to optimize the error in average speed over the amount of steps one pedestrian can move per iteration. Following this behaviour one can then observe situations where the pedestrian decides to skip its move or to travel more than one cell at once in order to reduce its error in average speed. A concrete implementation of this speed correction is implemented inside the simulation step of the `CellularAutomaton`.

Evaluation

All five pedestrians reach the target in the middle at approximately the same iteration step. The pedestrians are able to avoid collisions with each other.

Report on task 4/5: Obstacle avoidance

Objective

This tasks aims to let pedestrians choose a path to a target and to implement a rudimentary, as well as an intelligent, obstacle avoidance. The latter is then tested and evaluated by two test scenarios, namely the *Chicken Test* and the *Bottleneck Test*.

Computation of an utility grid with the Dijkstra algorithm

In order to let pedestrians know in which direction they are able to find a target, a utility for each cell is computed. The corresponding method is:

```
get_dijkstra_utility_grid(self, state_grid: np.ndarray, smart_obstacle_avoidance: bool)
-> np.ndarray
```

The resulting array visualizes the minimal distance to the next target and is calculated by using the Dijkstra Algorithm. Each cell corresponds to a node in a complete graph and we are iterating over all targets to individually compute the minimal distance of one target to each cell. We choose this approach because Dijkstra is a single-source-shortest-path algorithm and we can easily achieve the final utility grid by storing the minimal value for each cell while computing the distance grids.

Algorithm 1 Dijkstra-Algorithm

Input: t - target cell
Output: dist - array which stores minimal distances for each cell
Intermediate Variables: visited - array indicating whether a cell has already been processed
$visited = \emptyset$
$\forall u : dist[u] = \infty$
$dist[t] = 0$
for number n of cells in grid G **do**
  $u = argmin_{\forall v \notin visited} dist(v)$
  $visited = visited \cup u$
  for v: v is neighbor of u **do**
    $dist(v) = \min dist(v), dist(u) + d(u, v)$
  end for
end for

Algorithm 1 visualizes pseudo code of the Dijkstra Algorithm [1]. We start with a single target cell t and aim to compute minimal distances that are stored in an array *dist*. In order to process cells only once, we additionally store if a cell has already been visited in a corresponding array. Instead of initializing *visited* as an empty set, we used a `np.array` of booleans with the same shape as the utility grid. Furthermore, we set all distance values to infinity (`np.inf`) except the one of the target. In n iterations, with n being the number of cells in the grid, we start by choosing the cell u with the smallest *dist* value that has not been visited. This cell u is included in the set of visited cells by setting its value to `True`. Additionally, all the distance values of its neighbor cells are updated by taking the minimum of the previous value and the sum of the distance of t to u and the euclidean distance of u to its neighbor Equation (2).

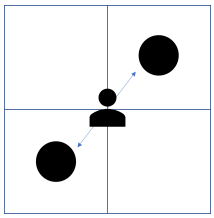
$$d(u, v) = \|v - u\|_2 = \sqrt{(v_1 - u_1)^2 + (v_2 - u_2)^2} \tag{2}$$

Obstacle avoidance

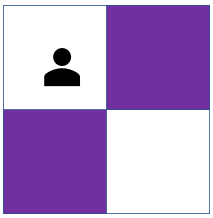
Using the Dijkstra Algorithm allows to implement different levels of obstacle avoidance. A rudimentary variant is implemented by setting the values of obstacle cells to `np.inf` after the computation of the utility grid. This way, pedestrians will never choose to step on these cells. Without any kind of obstacle avoidance, pedestrians will step on obstacles in order to choose the direct path to a target. In some scenarios like the *Chicken Test*, a more advanced implementation of obstacle avoidance is required. Pedestrians need to be navigated around obstacles by assigning cells behind obstacles higher values. Using the Dijkstra Algorithm on graphs, we see that we need to eliminate edges between obstacles and other cells. In our implementation, we simply do not update the distance values for obstacles when we iterate over neighbor cells as if obstacle cells are not accessible from other cells. This smarter variant of obstacle avoidance can be used by calling the function and setting `smart_obstacle_avoidance = True`.

Diagonal glitch

Which cells represent a real obstacle in a spatial discretized simulation world is essentially a matter of definition. This becomes particularly clear in the scenarios illustrated in the figures below.



(a) Shows a pedestrian moving diagonally between two obstacles, as the obstacles do not occupy an entire cell.



(b) Shows a pedestrian whose diagonal walking direction is blocked by two obstacles filling entire cells.

Figure 11: Comparison of the impact of obstacle dimensions on the walking behaviour of the pedestrians in a 2x2 grid world.

The pedestrian walks through the diagonal constellation of obstacles. We decide to allow this because we cannot know what these obstacles are. They might just be plant pots with a radius of 20cm. With a cell size of 50cm, a pedestrian will be able to walk diagonal without touching any obstacles.

Chicken test

Without a smart obstacle avoidance, pedestrians will not step on obstacles but might not reach their goal in some scenarios. Figure 12 illustrates the *Chicken Test* that consists of a pedestrian, a target and a U-shaped obstacle. This scenario clearly shows how different utility computations lead to different results and why an intelligent obstacle avoidance is necessary.

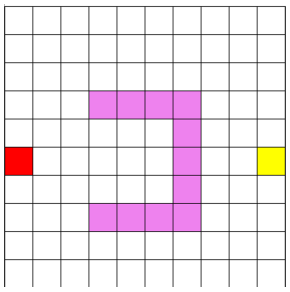
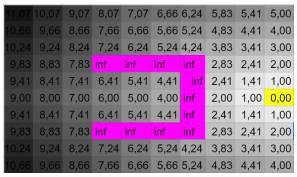


Figure 12: Chicken Test: an U-shaped obstacle is placed between a pedestrian and a target.

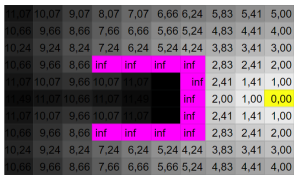
Comparing the utility grids of the same scenario with a rudimentary (see Figure 13a) and a intelligent (see

Figure 13b) obstacle avoidance, one is able to see that utilities are higher in the U-shape in the latter case. The illustration presents a color scale which associates higher utilities with darker color. In our simulation, pedestrians tend to walk into the direction of the smallest utility. This way, we already can make the assumption that pedestrians will walk towards the lighter cells and, therefore, will get stuck in the U-shape if there is no smart obstacle avoidance. Opposed to that, they will avoid the dark spot in the U-shape in the more advanced implementation.

Running the simulation confirms the previous assumption. Figure 14a shows that the pedestrian gets stuck and does not reach their target, but succeeds and navigates around the obstacles if intelligent obstacle avoidance is activated (see Figure 14b).

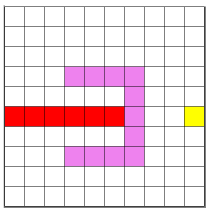


(a) Utility Grid of the Chicken Test without an intelligent obstacle avoidance: darker spots represent higher utilities, the U-shape is filled with lighter spots.

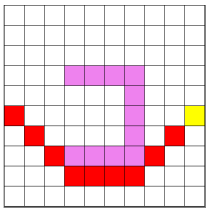


(b) Utility Grid of the Chicken Test with an intelligent obstacle avoidance: darker spots represent higher utilities, the U-shape is filled with dark spots.

Figure 13: Utility grid of chicken test scenario.



(a) Simulation of the Chicken Test without an intelligent obstacle avoidance: the pedestrian gets stuck in U-shaped obstacle.

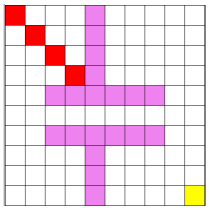


(b) Simulation of the Chicken Test with an intelligent obstacle avoidance: the pedestrian reaches their target.

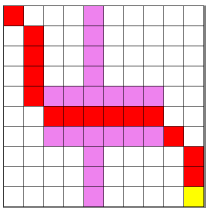
Figure 14: Chicken test scenario.

Bottleneck scenario

The same effect can be observed for the bottleneck scenario where obstacles are placed in a way that only leaves a small opening like a door or a tunnel. With a rudimentary version of obstacle avoidance, pedestrians might get stuck in a corner (see Figure 15a), but succeed to navigate through the tunnel with a more intelligent computation of utilities (see Figure 15b).



(a) Simulation of the Bottleneck Scenario without an intelligent obstacle avoidance: the pedestrian gets stuck in a corner.



(b) Simulation of the Bottleneck Scenario with an intelligent obstacle avoidance: the pedestrian successfully navigates through the tunnel and reaches their target.

Figure 15: Bottleneck scenario.

Report on task 5/5: Tests

Overview

This section covers the testing of the simulation based on the RiMEA guidelines [2]. Four scenarios are examined in order to verify and validate features of the simulation and to disclose features that are not implemented.

Test 1: RiMEA scenario 1 (straight line)

Objective

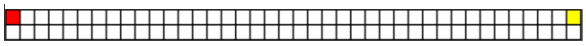
The following is a quote of the official scenario description from RiMEA test 1:

> "It is to be proven that a person in a 2 m wide and 40 m long corridor with a defined walking speed will cover the distance in a given time period. If 40 cm (body dimension), 1 s (premovement time) and 5 % (walking speed) are set as imprecise values, then the following requirement results with a typical pedestrian speed of 1.33 m/s: the speed should be set at a value between 4.5 and 5.1 km/h. The travel time should lie in the range of 26 to 34 seconds when 1.33 m/s is set as the speed."

The objective of this RiMEA scenario is to test if the average velocity of the pedestrian in the simulation matches the expectations of a real pedestrian.

Implementation

The simulation scenario is created in alignment to the official scenario description from RiMEA and is illustrated in Figure 16a. It consists of one pedestrian on the very left and a target on the very right and no obstacles in between. The grid is specified to be 40 cells long and 2 cells wide using a cell unit of 1 m. It is further assumed that the pedestrian covers the entire cell. One iteration is equal to 1 s.



(a) RiMEA scenario #1: one pedestrian and one target on a straight line.

(b) RiMEA scenario #1: the pedestrian has reached the target.

Figure 16: RiMEA test 1.

Results

The results are visualized in Figure 16b and Table 2. The pedestrian succeeds to reach the target and the required limit values are observed with a travel time of $29 \in [26, 34]$ s and a speed of 1.34 m/s = 4.82 km/h $\in [4.5, 5.1]$ km/h.

Travel time:	29 s
Distance travelled:	39.0 m
Speed desired	1.33 m/s
Speed achieved	1.34 m/s

Table 2: Measured data for RiMEA test 1.

Discussion

The pedestrian in the simulation is able to keep the speed according to the given ranges from RiMEA. We therefore conclude this first test as succesful. Note, that the premovement time specified as 1 s can be neglected for our cellular automaton with discrete time steps. Our space- and time-discrete simulation does not differentiate between "just about to move" and "currently moving". The error of neglecting the premovement time can be considered relatively small compared to the introduced discretization error of our cellular automaton. Taking a close look at Figure 16b one can also see cells that are not visited during the simulation. This is due to our speed correction algorithm that here lets the pedestrian walk two cells during one single iteration in order to keep the desired velocity. As the visualization only shows the state of the grid after the full simulation step, these cells look as if they have not been visited at all.

Test 2: RiMEA scenario 4 (fundamental diagram)

Objective

Figure 17 visualizes the setup of a corridor that is to be filled with different densities ($\rho \in \{0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0\}$ P/m^2) of persons with a walking speed as equal as possible (for example 1.2 – 1.4 m/s). At the measuring points, the average speed of a person over a period of 60 seconds is to be determined for the specified density. The first 10 seconds can be ignored as a "transient response". From the results (speed at the specified density), the corresponding fundamental diagrams can be created, with the calculation of the flow f as the product of the velocity v and the density ρ:

$$f = v \times \rho \tag{3}$$

In order to ensure that the fundamental diagram is also reproduced by the program in case of a *line movement*, the corridor should be reduced in width to such a degree that the persons can only move one behind the other and overtaking is not possible.

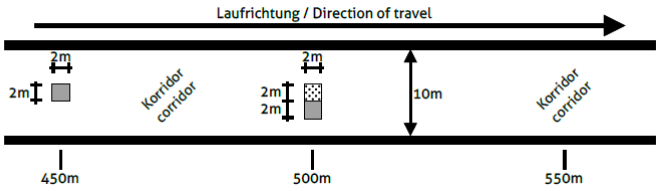


Figure 17: RiMEA test 4: Measurement in a corridor (1000 m long and 10 m wide) with three 2 m x 2 m measuring points, a main one (dotted) and two control points (grey) [2].

Implementation

For our simulation of the scenario we make the assumption that the corridor is 20 m long and 5 m wide. Only one person fits in one cell which has a width and height of 0.5 m. The main measuring point also takes one cell and we measure the last 10 iterations whenever one pedestrian steps on the cell. One iteration takes 0.5 s and the first 10 iterations are ignored as a *"transient response"*. The simulation is evaluated for the densities $\rho \in \{0.5, 1, 2, 3, 4\}$ P/m^2. Pedestrians are distributed randomly on the grid according to ρ and targets are placed at the end of the corridor. For the simulation it is also necessary to randomly select the movement priority of pedestrians within one movement step to not have the blocking behaviour only once at the very beginning of the simulation, but constantly throughout the entire process.

Results

Results are presented in Figure 18. The pedestrians' velocity decreases when the density is increased while flow and density are almost proportional up to a density $\rho = 3.0$ P/m^2. For higher values, differences between

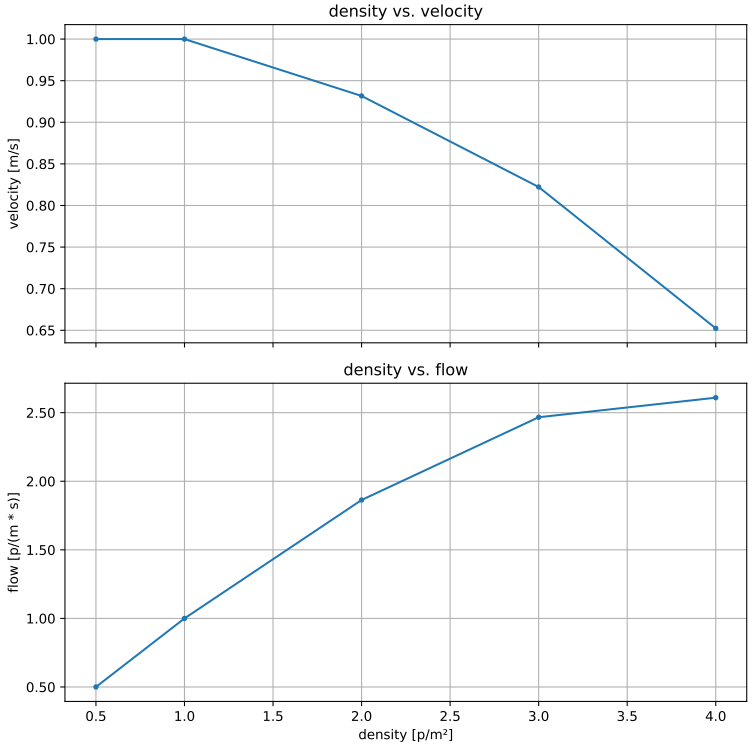


Figure 18: RiMEA test 4: Diagrams of density vs. velocity and density vs. flow: velocity decreases and flow increases with increasing density.

densities seem to have a smaller influence on changes of the flow.

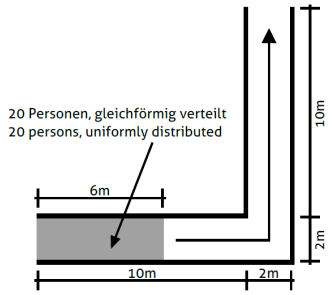
Discussion

Both results of Figure 18 seem reasonable. For one, the average velocity of pedestrians is decreasing with an increasing density. This is due to each pedestrian having less space to freely walk and an increasing likelihood of blocking moments. This blocking effect is even increased in our space- and time-discrete cellular automaton where we do not implicitly consider the next destination cell of another neighboring pedestrian when planning the move for the current one. The approaching upper-limit of the pedestrian flow with increasing density also seems plausible due to the maximum density of every cell being occupied by one single pedestrian (this is the case for a density of $\rho = 4.0\,\mathrm{P/m^2}$). It needs to be noted that our scenario does not perfectly match the one described in RiMEA test 4: Computational boundaries require keeping the number of cells in the cellular grid rather low. Also the cell unit is set to be equal to 0.5 m. This results in a fully occupied grid for a density $\rho = 4.0\,\mathrm{P/m^2}$, which is why the density values above this limit are not tested. Nonetheless we believe that our simulation results would have been the same in essence if we were to increase the total number of cells.

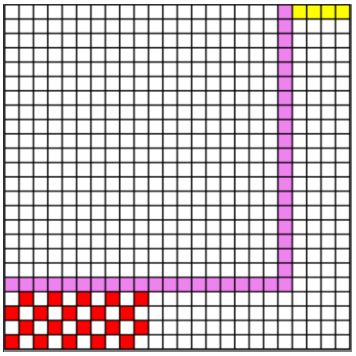
Test 3: RiMEA scenario 6 (movement around a corner)

Objective

The RiMEA test 6 analyzes the performance of the simulation when a group of pedestrians is moving around a corner. The dimensions of the scenario are specified in Figure 19a.



(a) RiMEA test 6: Official scenario description.



(b) RiMEA test 6: Reconstructed implementation in the cellular grid.

Figure 19: RiMEA test 6: Scenario specification.

Implementation

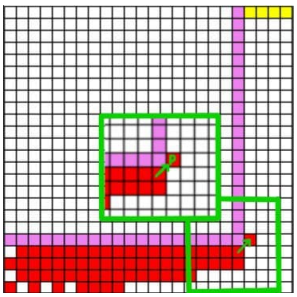
Figure 19b shows the implementation of the given scenario. In order to achieve an approximate uniform distribution of the 20 pedestrian, the cell dimensions are specified as 0.5 m per cell. One pedestrian fully occupies one cell.

Results

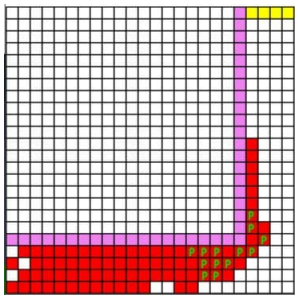
Figure 20 show the resulting simulation of our cellular automaton at two different timesteps in the simulation. All 20 pedestrians successfully reach one of the four target cells. The boundaries set up by the obstacles are being kept in check.

Discussion

Figure 20a reveals that pedestrians technically do not move correctly around corners because of the effect of the *diagonal glitch*. However, this implementation seems to be sufficient and close to reality when we consider that the body dimension does not fill a complete cell (see the sub-chapter on diagonal glitch for a full discussion). The behaviour of the crowd can be observed in Figure 20b. The density increases in the area around the corner because pedestrians try to take the shortest path that goes right past the corner. After the corner most pedestrians walk next to the wall and reach the target on the left. This behavior seems to be close to reality and, finally, all pedestrians get to a target after about 50 iterations without getting stuck or walking through an obstacle.



(a) RiMEA test 6: simulation at time step t = 12. Breakout in green visualizes a diagonal glitch move.



(b) RiMEA test 6: simulation at time step t = 20.

Figure 20: RiMEA test 6: simulation.

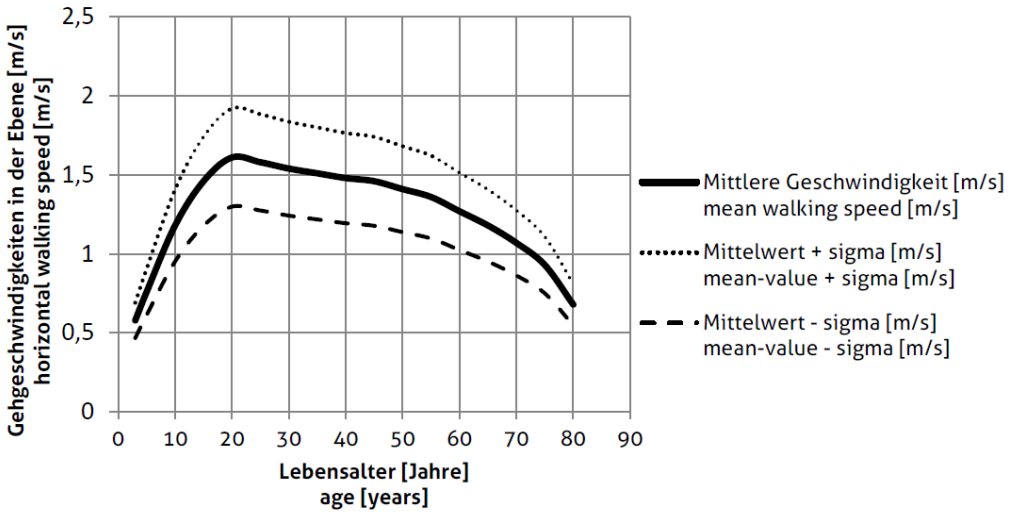


Figure 21: Walking speed in the plane as a function of age based on Weidmann [2].

Test 4: RiMEA scenario 7 (demographic parameters)

Objective

Scenario 7 tests the allocation of demographic parameters by choosing a distribution of 50 pedestrians according to Figure 21 and letting them walk in a specified scenario of choice. It is to be shown that the resulting distribution of walking speeds as observed during the simulation matches the original distribution.

Implementation

The implementation of the scenario consists of 50 pedestrians set up vertically on the left side and 50 targets on the right side. The grid size is specified as 40 cells long and 50 cells wide. One cell unit corresponds to one meter and one iteration step to 1 second.

In order to replicate the given population as seen in Figure 21, velocity values at defined support points of age were read from the graph (Table 3). Then, the population of 50 pedestrians inside the simulation where assigned to age values equally distributed over the interval $[10; 80]$. Using linear interpolation and the beforehand defined support points, one could assign a velocity value to all pedestrians. After these velocity values have been assigned to all pedestrians, the simulation has been started and the resulting average velocity calculated afterwards.

age [years]	velocity [m/s]
10	1.20
20	1.60
30	1.53
40	1.50
50	1.40
60	1.25
70	1.10
80	0.70

Table 3: RiMEA test 7: Support points of velocities per age.

Results

The upper plot in Figure 22 compares the interpolated velocity values read from Figure 21 (orange) to the measured values after the simulation (blue). The lower plot in Figure 22 visualizes a histogram of the velocity error and a normal distribution fit for visualizing the average offset and the spread of the simulation error. In average over our tested population of 50 pedestrians, the velocity error is μ =-0.005 m/s with a variance of $\sigma^2 = 0.01$ m/s.

Discussion

Our results show that the measured distribution is very similar to the desired distribution. One can clearly see the error resulting from space- and time-discrete cellular automaton as stepwise horizontal velocity values. For an increasing number of simulated grid cells, these errors are expected to converge towards zero. However, this improvement in simulation accuracy would also lead to higher computational requirements for the simulation of the grid.

Instead of only performing this experiment once, one could think of a statistical test to prove the significance of the obtained results, much like in a fashion as described on page 11 of [2] (chapter "Statistical evaluation of repeated simulation runs"). One might then think of a process like:

1. Uniformly sample 50 age values and corresponding velocity values from a range of $[10, 80]$.
2. Run the simulation and calculate the average velocity error
3. Repeat step 1 + 2 for a sufficient number of times to obtain a distribution of errors over the number of experiments.
4. Analyze this distribution and obtain statistical values such as the min, max, mean, standard deviation or p-values of a null-hypothesis.

However, the focus in the RiMEA test 7 is to analyze the errors within one age distribution of one single simulation run, as depicted in Figure 22.

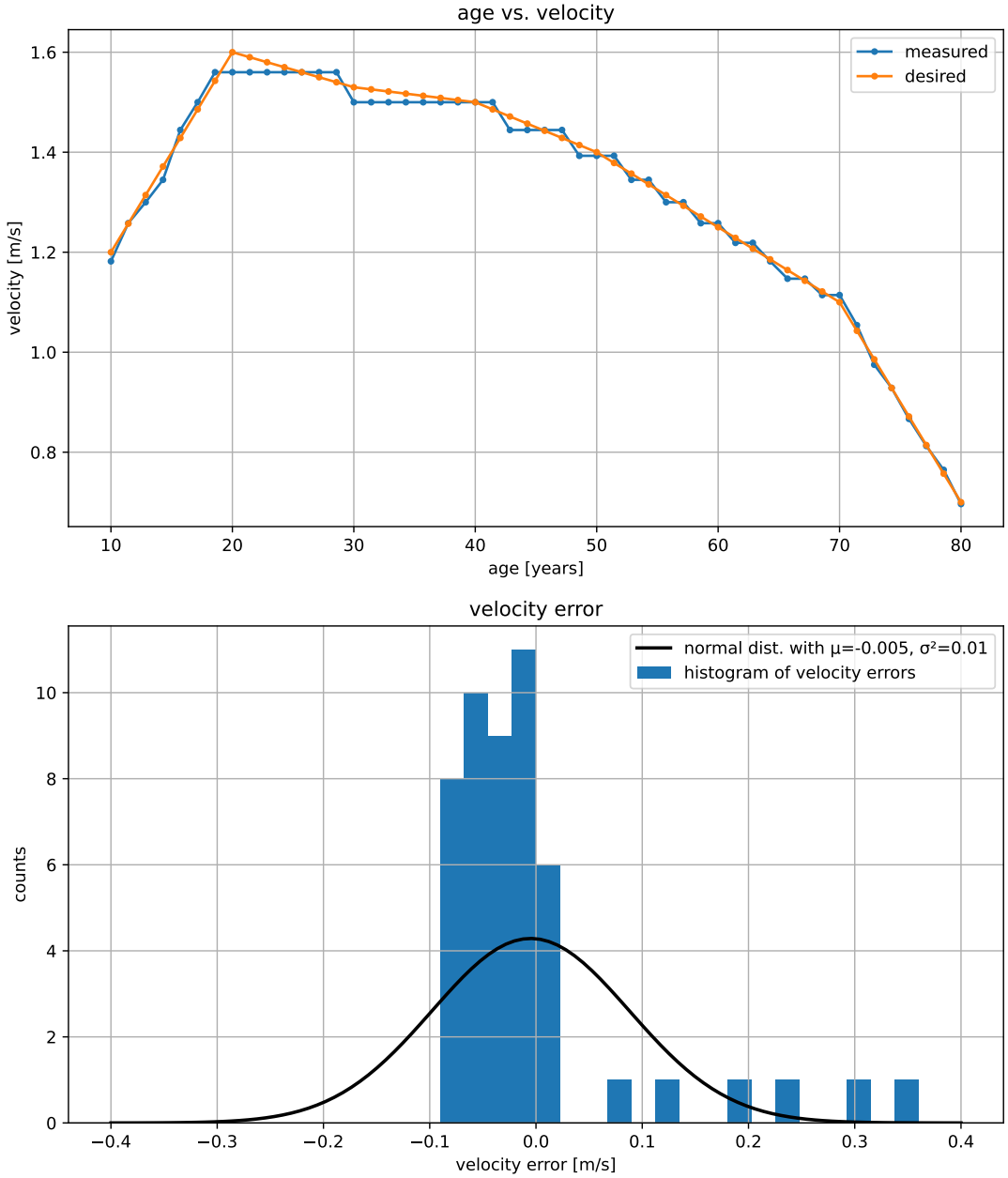


Figure 22: RiMEA test 7: Diagram that compares the desired and measured distribution of people's velocities and ages.

3 Review

This section will now review the work we have done in a broader context and give an outlook on what the next steps could look like:

The goal of this work, given the very limited time frame and resources, was to develop a prototype that can be used to model and simulate crowds. For this purpose, two extensive and modular pipelines were implemented with which it is possible to simulate both static and changing scenarios over time. Due to the time constraints of the project, it was possible to develop a cellular automaton that can independently cope with the main challenges of a crowd, such as finding a target area, multi-agent interaction and dealing with obstacles. It should be emphasised that standard problems of motion planning such as the "chicken test" or the "bottleneck scenario" as described in Section 2 could be successfully completed. Furthermore, the two scenarios show impressively that dealing with obstacles in a simulation is not a trivial activity, but, as described in more detail in Section 2 it is always a question of definition.

Another challenge that arises in a multi-agent environment is that pedestrians can become obstacles themselves at certain points in time. The necessary simplification that pedestrians who have reached their destination disappear from the simulation environment clearly shows the limits of the simulation tool. That this is not just a plain simplification becomes clear when one takes a closer look at scenarios from reality. For example, the guidelines for microscopic escape analysis were developed to elaborate better fire protection and evacuation plans with a focus at precisely this problem. A possible extension of our tool could be, for example, to implement an outflow rate through the target cell, this would come closer to the process of a crowd flowing through a constriction like a door.

However, the results presented in Section 2 for the execution of the different RiMEA tests invalidate the fact that this could question the realism of the simulation as a whole. For example, the average travel time of 29 iterations calculated in test 1 lies in the middle of the interval I of $[26, 34]$ and thus indicates only a small deviation. Furthermore, our prototype also takes into account the change in travel speed that is dependent on demographic factors like age.

The limited resources highlighted at the beginning of this section became particularly apparent in RiMEA test 2, as this could become very computationally intensive with its 1000m long and 10m wide corridor, depending on the scaling of the grid. A possible starting point for improvements would therefore be to check the code for efficiency gaps, especially in the calculation of the utility grid. Another possibility would be to use a programming language that improves the runtime, like C++.

All in all, this simulation tool with its diagonal glitch and slow runtime for complex scenarios still has numerous starting points for improvements. Nevertheless, the results in Section 2 show that all the observed RiMEA tests could be completed successfully. In summary, these results indicate that the prototype can already solve many challenges and thus a successful approximation of reality has been achieved.

References

[1] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959.

[2] V Rimea. Richtlinie für Mikroskopische Entfluchtungsanalysen. 2016.

[3] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of modern physics*, 1983.

[4] Stephen Wolfram. Cellular automata as models of complexity. *Nature*, 1984.