Report for exercise 1 from group C

Tasks addressed:	5
Authors:	Alejandro Hernandez Artiles (03785345)
	Pavel Sindelar (03785154)
	Haoxiang Yang (03767758)
	Jianfeng Yue (03765255)
	Leonhard Chen (03711258)
Last compiled:	2024–03–02
Source code:	https://github.com/alejandrohdez00/Exercises-MLCMS-Group-C/tree/main/Exercise-1

The work on tasks was divided in the following way:

Alejandro Hernandez Artiles (03785345) **(Project Lead)**	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Pavel Sindelar (03785154)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Haoxiang Yang (03767758)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Jianfeng Yue (03765255)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Leonhard Chen (03711258)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

Introduction In this project we introduce ourselves to the field of human crowd modelling using a simple cellular automaton, a useful approach for modelling complex and dynamical systems. The automaton works by having cells follow certain rules to determine their next state. Despite the simplicity of these rules, their iteration in different cells produces a complex and interesting behaviour.

Each cell of our cellular automaton can be in four different states: "Empty", "Pedestrian", "Obstacle" and "Target". In the system, the pedestrians will try to reach a target cell, avoiding the obstacles by means of update scheme, which basically tries to reduce the Euclidean distance of each pedestrian to one of the target cells by updating the position of the pedestrian to the nearest neighbouring cell to a target. This update scheme could be changed during the project.

Report on task 1, Setting up the modeling environment

In this first task we want to setup the basic visualization for running the simulations. The initial project provides a simple GUI for the simulation of the cellular automaton. Each cell draws pedestrians, targets, obstacles as a colored rectangle and by clicking the step button a single time-step is simulated. We use the following colors for each object:

- White for empty cells
- Blue for target cells
- Magenta for obstacle cells
- Red for cells with pedestrians
- Light Grey for cells traversed by a pedestrian (New)

While the example project does provide a rough framework, we added several major changes to allow flexibility for adding new functionality and ease of use for the following tasks. We will first explain the implementation decisions and then at the end of this report section we will provide instructions on how to use the program. To improve the initial setup the main goals were:

1. to define a file format for simulation scenarios,
2. to implement loading scenarios from external files and
3. to implement more controls for performing simulations.

1. Define a file format for simulation scenarios After observing how a simple scenario was hard-coded inside the `start_gui(...)` function in `gui.py`, we first defined the format of the scenario files. All simulation scenarios are stored in `.json`-files and their format is as follows:

```
{
    "iterations": n,
    "cell_size": [width, height],
    "targets": [
        [tar_x, tar_y]
    ],
    "obstacles": [
        [obs_x, obs_y]
    ],
    "pedestrians": [
        { "position": [pos_x, pos_y], "speed": v }
    ]
}
```

- `"iterations"` Defines the integer n as the number of simulation time-steps that are performed automatically after starting the simulation
- `"cell_size"` Defines the integers `width` and `height` for the cells of the cellular automaton. Note that GUI will display all ratios between width and height in a square canvas, hence an uneven ratio will lead to a stretched display of cells.
- `"targets"` Defines a list of target coordinates stored as an integer list.
- `"obstacles"` Defines a list of obstacle coordinates stored as an integer list.
- `"pedestrians"` Defines a list of pedestrians, each containing integer starting positions and a float velocity. Note that multiple pedestrians can be placed in the same cell.

2. Implement loading scenarios from external files To load these scenario files, we implemented both loading from CLI and GUI. Loading from CLI requires us to modify `main.py`, so we can parse one additional optional arguments when starting the program. The arguments allow us to load a specific scenario file, choose a specific algorithm and whether we want to pedestrians to avoid each other. By default the scenario `scenario_task1.json` is loaded with the fast marching algorithm and pedestrian avoidance enabled. All of these arguments are passed to the GUI class.

Inside the GUI object the simulation of the scenario is setup in 2 steps:

1. The `start_gui(...)` function initially loads the contents of the scenario file inside the class attribute `config=None`, which is later on needed for resetting and loading a scenario. (Similarly many other variables created in `start_gui(...)` are added as class attributes to shorten the signature of functions inside this class.)
2. Then `self.load_scenario_from_config(...)` is called to actually setup the simulation environment with the given parameters from the scenario file.

To load from the GUI we perform these same 2 steps inside the `load_scenario(...)` function.

3. Implement more controls for performing simulations For debugging purposes the simulation was changed to run automatically after pressing the start button. Additionally we added a reset button to reload the starting configuration of the scenario file. On the top right of the program the remaining iterations are displayed. Once the simulation has ended the program will display "Simulation finished". Additionally on the bottom left the change background button visualizes the the distance values with a blue hue.

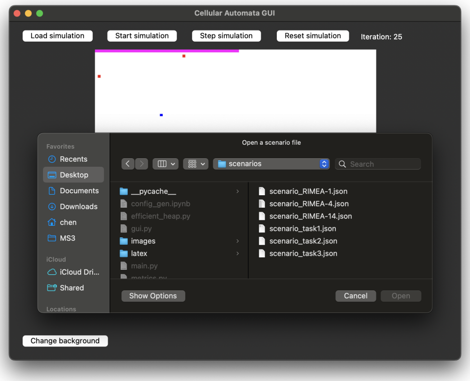
Instructions To define a scenario file follow the file format defined in the previous section. The program can be started from CLI and optionally a scenario file, algorithm choice and pedestrian avoidance flag can be passed to initially load it into the simulation. Possible algorithm arguments are "F" for fast marching (default), "D" for Dijkstra and "S" for the default update rule. To disable pedestrian avoidance adding the flag *–ignorePedestrians* suffices. The default starting command for task 1 is as follows:

```
python main.py --scenario scenario_task1.json --algorithm S --ignorePedestrians
```

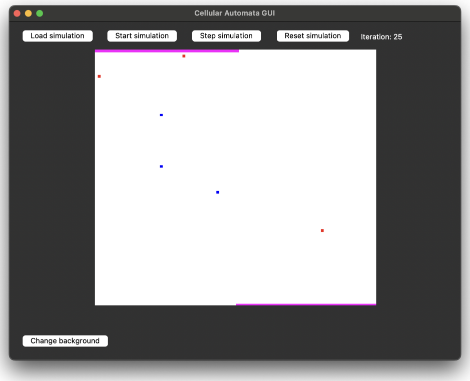
After the simulation has loaded there are 5 buttons available.

- `Load simulation`: Opens a dialogue window. Selecting a .json file loads a scenario into the program.
- `Start simulation`: Starts the simulation and stops once the pedestrians arrive at their target or the iteration counter on the top right reaches 0.
- `Step simulation`: Simulates a single time-step of the pedestrians and pauses the current running simulation.
- `Reset simulation`: Resets the simulation to the initial state of the scenario.
- `Change background`: Changes the background to display the distance to targets. It takes into account the distances computed by the algorithm in use.

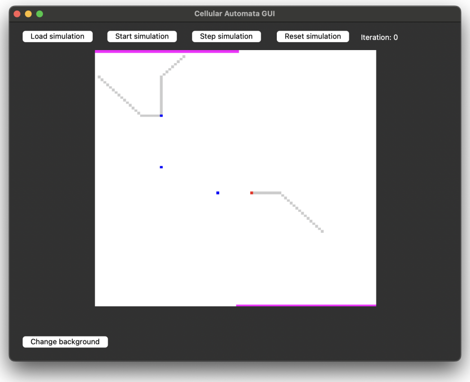
The following figures provide an example of using the simulation controls with `scenario_task1.json`, which is the scenario provided in the example project. Adding obstacles is possible, but will be demonstrated in task 2. Figure 1a shows the dialogue window when loading a new simulation. Figure 1b shows an example scenario with 3 pedestrians, 3 targets and objects. In figure 1c we see the result while running the simulation. In figure 1d the pedestrians have reached their targets by either manually performing simulation steps or running the simulation automatically. In Figure 1e the visualization of distances to the targets has been enabled.



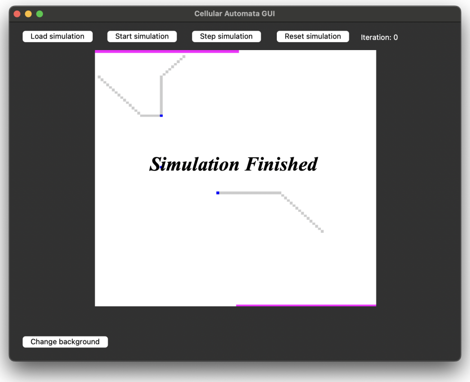
(a) Load simulation



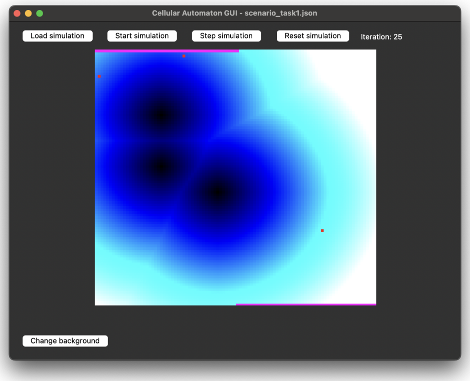
(b) Start simulation



(c) Simulation Finished



(d) Target reached



(e) Visualizing target distances

Figure 1: Use of the GUI

Report on task 2, First step of a single pedestrian

In the second task we want to test the basic functionalities of the simulations in a simple scenario. Here we created a new scenario file that satisfies the following specifications:

- Iterations: 25
- Cell size: 50x50
- Target position: (25, 25)
- Pedestrian position: (5, 25)

The scenario can be opened via the load simulation button and selecting "scenario_task2.json" in the dialogue window. To run this test from CLI run the following command:

```
python main.py --scenario scenario_task2.json --algorithm S
```

Afterwards, the simulation finishes with the pedestrian reaching the target and waiting there.

(a) Initial state (b) Final state

Figure 2: Simulation of task 2

In the Figure 3 we make certain cells inaccessible by adding a vertical line of obstacles to block the pedestrian. Here we still use the standard movement algorithm with Euclidean distance. We treat all 8 connected neighboring cells as the "next step" to calculate the optimal step. As the pedestrian reached the cell neighbouring to the obstacles, the pedestrian get stuck at that point. Use the following command to reproduce this example.

```
python main.py --scenario scenario_task2a.json --algorithm S
```

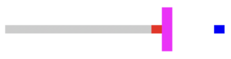
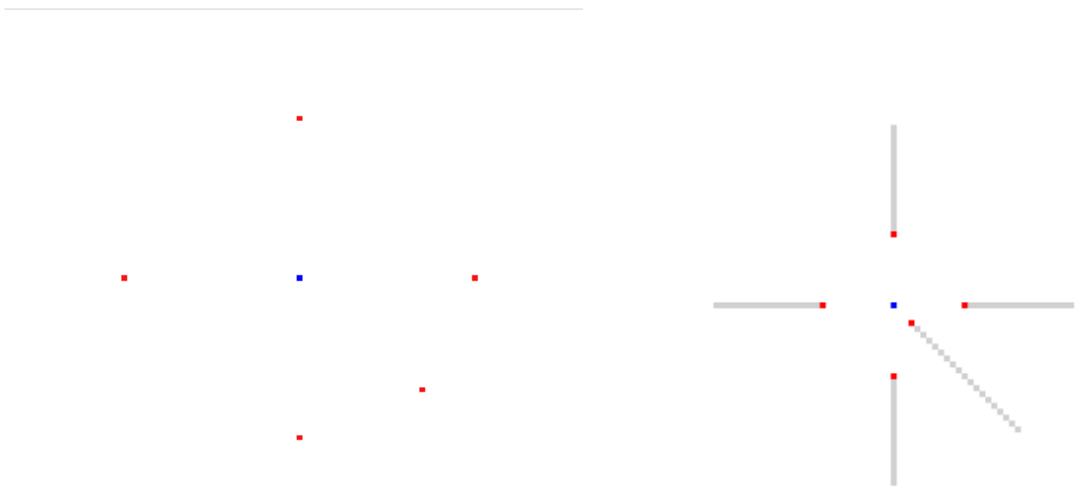


Figure 3: Pedestrian blocked by obstacle

Report on task 3, Interaction of pedestrians

The third task serves to manage the speeds of pedestrians. To do this, we created a scenario where 5 pedestrians are at approximately the same distance from a target, forming a circle. In particular, as we want to check the difference in speed between horizontal, vertical and diagonal movements. In the scenario there are four pedestrians forming a cross with respect to the centre, providing the horizontal and vertical movements, and we have placed a pedestrian at the same distance on a diagonal.



(a) Initial state (b) Final state

Figure 4: Initial simulation of task 3

As we can observe in Figure 4b, the diagonal path to the target is faster than the horizontal and vertical ones. That is because our system is using a discrete grid, but the distance from a pedestrian to the target is computed using the Euclidean distance. Therefore, if a pedestrian moves horizontally or vertically, it is advancing a distance of 1, but if it moves diagonally it is advancing a distance of $\sqrt{2} = 1.41...$, thus travelling more distance in the same time.

We tried to solve the problem by putting a penalty on the diagonal movement. We calculated the Euclidean distance travelled by a pedestrian and when a series of time steps were fulfilled we calculated its velocity as the radius of the distance and the number of those time steps. If the pedestrian's speed was greater than the desired speed, entered as a parameter in the scenario's .json file, it was forced to stop at the next time step.

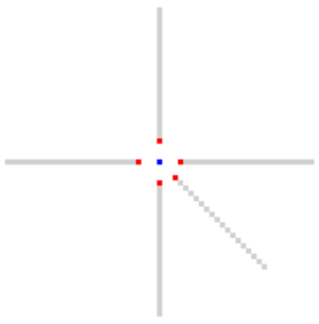
As we can observe in Figure 5a, this method achieved that all pedestrians reached the target at the same time step, reducing the difference in their speeds. However, it is not very realistic for a crowd to be constrained in diagonal motion only, so we generalised this concept and made the motion of pedestrians stochastic, still trying to penalise more the diagonal movements. We achieve this using formula (1):

$$\frac{\text{desired\_speed}}{\text{dist}(\text{position}, \text{min\_cost\_pos})} < \epsilon \tag{1}$$

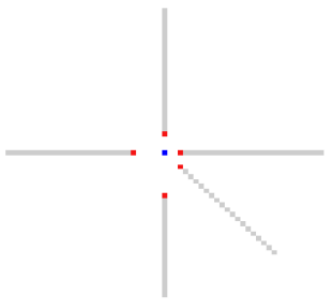
where ϵ is a random number $\epsilon \in [0, 1]$. This formula is checked inside the `update_step` function in the Pedestrian class, and if the condition is fulfilled, the pedestrian does not advance a position in that time step. Basically, the pedestrian's speed becomes a probability distribution, since we are comparing the ratio between their desired speed and the Euclidean distance to a random number.

If the Euclidean distance from the pedestrian's current position and its next position is greater (it is diagonal) it is more likely to stop than making a horizontal or vertical movement. On the other hand, as the divisor is the desired velocity, if this is very large, it will be less likely to stop as well. Therefore, in this simple way a velocity distribution can be modelled for the pedestrians with the diagonal motion constraints.

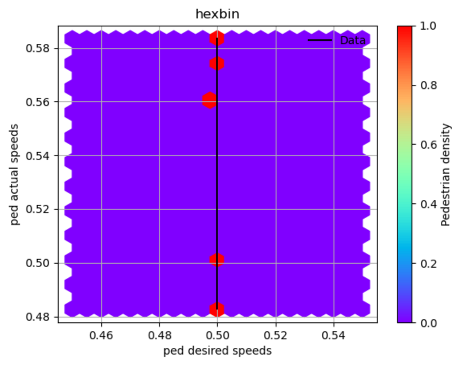
In Figure 5b we can observe a non-uniform speed distribution allowing horizontal and vertical paths to be equally fast as diagonal ones. In Figure 5c is shown the comparison between the desired speed and the actual speed of each of the pedestrians. The stochastic distribution of speeds can be noticed. For example the 3 fastest pedestrians in Figure 5b have a higher current speed and are at the top of the graph, and the last two pedestrians have a current speed of approximately 0.1 less.



(a) First implementation



(b) Stochastic implementation



(c) Actual speed vs desired speed distribution

Figure 5: Implementations of task 3

Pedestrian avoidance was not needed for this exercise, however it is still a feature of pedestrian interaction. Pedestrian avoidance is explained in the next section. Use the following command to run the experiment:

```
python main.py --scenario scenario_task3.json --algorithm S
```

Report on task 4, Obstacle avoidance

The starting point was to create a heap with decrease key data structure. The decrease function was implemented through integer IDs. After every add an ID was returned (simply the number of times add was called on the particular heap beforehand), this seams like a cleaner implementation than using the python id method or some hashing function as it is language oblivious, doesn't necessitate keeping the original object around and allows for the same values to occur multiple times with different keys if necessary. The first implementation was a simple list heap with O(N) time complexity. This was later changed into a binary tree in list heap.

The Dijkstra's algorithm implementation was originally a path finding one because the author had miss-read the problem definition. The author was of the belief that it was necessary to calculate the distance from each position to the targets by starting the Dijkstra's algorithm at the position while it is of course elementary to instead start at the target and work one's way backward. In hopes of somehow decreasing the substantial runtime this caused the author tried to make the Heap more efficient by using numpy arrays instead of lists of tuples. Turning the algorithm into an A* by adding a euclidian distance heuristic was also tried. In the end it turned out this was not enough and the author soon discovered his error and that these additions were not necessary, so they were removed from the final code. The implementation was made modular by making the distance and neighborhood functions it's parameters of type typing.Callable.

For implementing the Fast Marching Method algorithm all that was necessary was to realize that having implemented Dijkstra's algorithm all that was required to implement the Fast Marching Method was to add a dictionary with distances to closed keys as a parameter to the distance function. All that was needed then was to pass a Fast Marching Method distance update function as a distance function to the Dijkstra's algorithm implementation.

The Fast Marching Method distance update is based on the simple idea that if the derivative of the distance between neighboring nodes is to be constant the identity

$$max(D_{0,0} - D_{1,0}, D_{0,0} - D_{-1,0})^2 + max(D_{0,0} - D_{0,1}, D_{0,0} - D_{0,-1})^2 - C = 0$$

must hold. Where $D_{0,0}$ is the distance to the updated node at position a,b and $D_{x,y}$ is simply the node at position (a+x,b+y). C being the square of the derivative of the distance function which in our case where we use the Fast Marching Method only for distance calculation is an arbitrary positive number, though 1 seems to work the best as at larger numbers the method's inaccuracy increases. All we need to get the update is to get the greater of the roots of this polynomial. In case there is no such root we simply use the approximate expression

$$\frac{-b}{2a}$$

where b is the first power multiplier in the polynomial and a the second power one.

In cases where there is another value in the direction of the maximum distance we can use the second order Fast Marching Method, this has been implemented in the project but describing the theory behind it seems out of scope of this work.

The user interface function `path_lengths` was for the user's ease of use made to only take a neighborhood function and `is_obstacle` function, with a simple enum used to choose between different implementations.

As stated before using this implementation in the application was as simple as calling the function for every target with the neighborhood function set to either adjacent for the Fast Marching Method implementation or adjacent+diagonal for the Dijkstra's algorithm implementation, the obstacle function simply checking whether the grid contains and obstacle at given position and recording the minimum resultant distance for each cell.

Pedestrian avoidance As instructed we only take into consideration obstacles and ignore pedestrians while calculating distances. This significantly reduces computational complexity but forces any pedestrian avoidance to have a purely greedy character. Of course in most ways this is exactly the behaviour that makes most sense as humans are generaly very poor at predicting the behaviour of other humans in a crowd over any longer period of time. We implement this through a simple cost function. To calculate this for cost for a target position the algorithm takes the count of pedestrians at a given surrounding position and multiplies them with weights we assign based on the offset from the target position. These weights are a dictionary of weights decreasing exponentially with euclidian distance of the key from the center of the matrix.

$$f(d) = \begin{cases} exp(\frac{p}{d^2 - d_{max}^2}) \text{ if } d < d_{max} \\ 0 \text{ otherwise} \end{cases}$$

p being the parameter determining how fast f decreases.

To allow for easier manipulation these values are then normalized.

Setting p too low can cause clustering into lines or points since the small distance positions occur less frequently than the high distance ones. It would therefore seem reasonable that at least half the distribution be at the position (0,0).

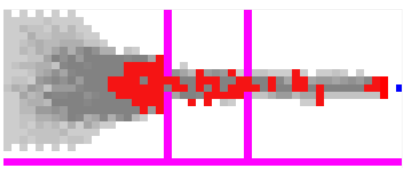
In general because the complexity of the calculation increases with the square of d_{max} with the python implementation values greater than 10 decrease the speed of the simulation. Then again multipliers of less than 0.01 (which such a distribution would have even were it uniform) hardly contribute much. These are especially preelent if we want at least half the distribution be near the center. Therefore if we explicitly ignore such small multipliers most reasonable distributions of any size are computationally reasonable for given pedestrian counts. This needs pre-computing the values and caching them instead of calculating the coefficient during every cost calculation as it was originally, this doesn't have much impact on the complexity neither positive or negative (apart from the aforementioned ability to ignore small coefficients). It also allows for the code to be more modular as we can theoretically pass any coefficient dictionary to the function.

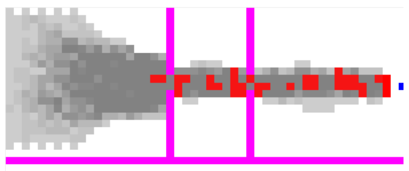
Experiment description In this experiment we perform the RiMEA-12 test according to [1], where pedestrians have to pass a tight hallway. This experiment should show that pedestrians will form a congestion only at the first bottleneck. Notably here are the different CLI prompts we can use to simulate different pedestrian avoidance behaviour. By default we run the following command:

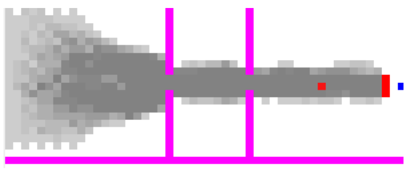
```
python main.py --scenario scenario-RIMEA-12.json
```

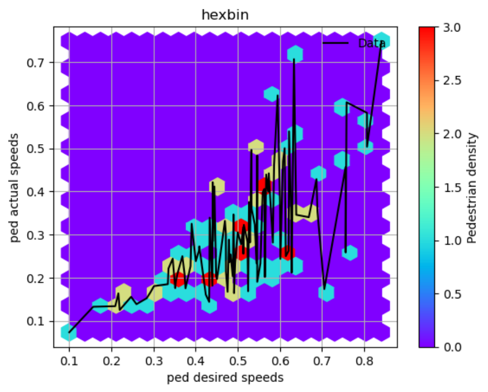
Use the flag `--multiplePedestriansInOneCell` to change the behaviour of pedestrian avoidance. If the flag `--ignorePedestrians` is set before pedestrian avoidance will be completely disabled, hence setting the flag `--multiplePedestriansInOneCell` does nothing.

If the simulation with no flags is run we observe the following figures 6a, 6b and 6c to show the behaviour in the bottleneck scenario. When presented with the bottleneck the pedestrians are forced to wait as the cost of being forced through the bottleneck all at once is too high. The pedestrians also immediately disperse again after leaving the bottleneck to minimize their cost.


(a) Congested pedestrians


(b) Pedestrians flowing through after bottleneck ended

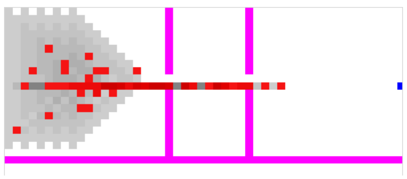

(c) Bottleneck end

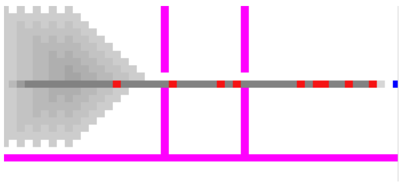


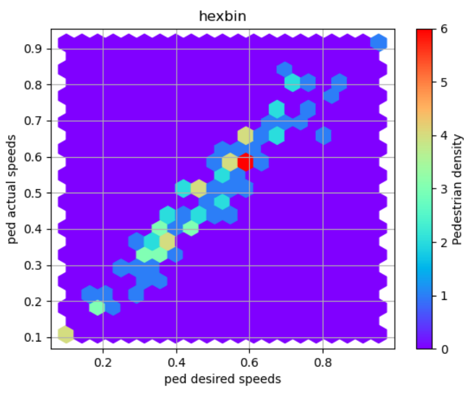
(d) Bottleneck speeds

Figure 6: RiMEA-12 simulation

It seems that the blockage causes the actual speeds to not correlate as much with the desired speeds. We can observe this better if we don't use pedestrian avoidance in the same scenario. In Figure 7a and 7b we can see that now there is no bottleneck and pedestrians follows a straightforward path to the target, with no need for waiting the other pedestrians. This saved time is exhibited in Figure 7c, where the speeds are more uniformly and linearly distributed, showing that pedestrians have made the way according to their desired speed, instead of reducing the actual speed due to the bottleneck. Also, analyzing the graph, we can observe that, naturally, without bottleneck there is more density of pedestrians with equivalent desired_speed/actual_speed ratio, as pedestrians can move freely. The maximum density in Figure 6d is 3 and in 7c is 6.


(a) No bottleneck


(b) More advanced step in simulation



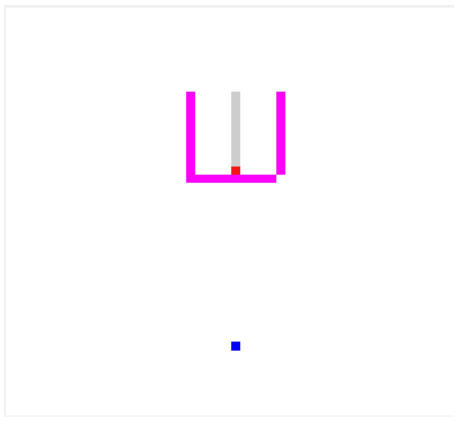
(c) No bottleneck speeds

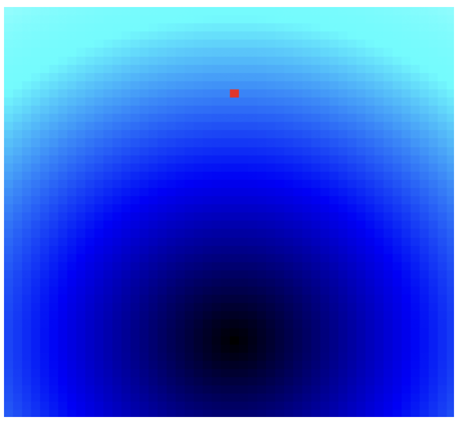
Figure 7: RiMEA-12 simulation without pedestrian avoidance

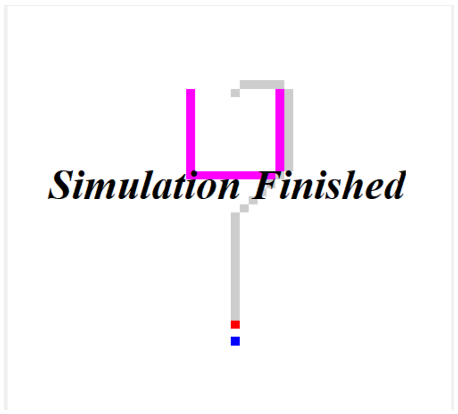
Then, in the chicken test if we simply use distance to target as path to target cost, thus ignoring obstacles, the pedestrian gets stuck at the first intersection of a line to the target and the obstacle (Figure 8a). However, if we calculate the distance to the target using Fast Marching, the pedestrian can find an optimal path (Figure 8c). Going deeper, by changing the background to the computed distances by the algorithm in use, we can analyse the cause of the problem, as it can be seen in Figure 8b and 8d. The pedestrian has a more informed distance measurement using the Fast Marching Method. The CLI prompts for 8a and and 8c are respectively:

```
python main.py --scenario chicken_test.json --algorithm S

python main.py --scenario chicken_test.json --algorithm F
```


(a) Chicken test no path finding


(b) Standard algorithm visual test



(c) Chicken test with path finding

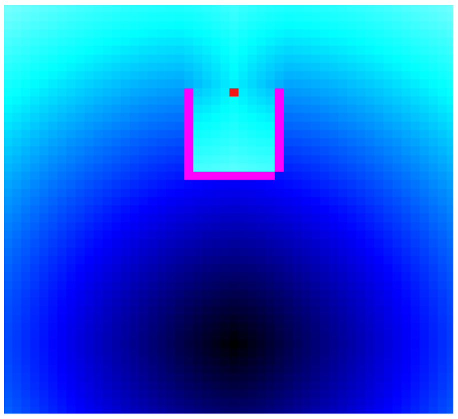

(d) FMM visual test

Figure 8: Chicken test simulation

Report on task 5, Tests

In the following sections we want to perform multiple tests to see the capabilites of our simulation program. All referred tests are extensively explained in the RiMEA guieline [1]. All of the path finding in all the scenarios is done using the Fast Marching Method, which means obstacle avoidance is enabled by default. For an explicit visual test showing the distances to target, you can change the background like Figure 8d by clicking the change background button.

Pedestrian features The only parameter implemented was free walking speed. Pre-movement time is not necessary as stated in the task description and only complicates testing and evaluation. Free walking speed on stairs is unnecessary as there are no stairs in any of the scenarios. Age distribution is only necessary when determining free walking speed distributions.

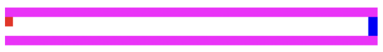
RiMEA scenario 1 In this experiment we want to confirm that the pedestrians are moving at the correct speed. The scenario is a 40 meter long hallway which has to be traversed in a specific time frame. In this simulation 1 cell represents 1 meter. In our simulation, one iteration is equal to 0.5 seconds because max number of cells traveled by a pedestrian per iteration is 1 and we wish to represent the required speed 1.33 m/s. Figure 9 shows the setup and result of this experiment. A sample of the measured escape times in simulation iterations are the following:

```
[52, 57, 65, 50, 55, 45, 56, 54, 51, 61]
```

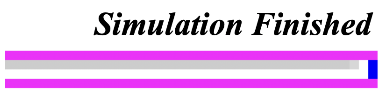
Because one simulation iteration is equivalent to 0.5 seconds we can convert that to the following escape times in seconds.

```
[26, 28.5, 32.5, 25, 27.5, 22.5, 28, 27, 25.5, 30.3]
```

This is within the 26 to 34 range specified in the RiMEA guideline.



(a) RiMEA 1 setup



(b) RiMEA 1 finished

Figure 9: Scenario RiMEA-1

To reproduce the samples and plots from the previous part run the following iPythonNotebook `scenario_1.ipynb`. The command line interface prompt for running the simulation only is as follows:

```
python main.py --scenario scenario_RIMEA-1.json
```

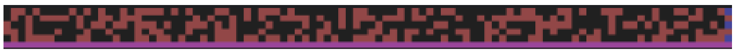
RiMEA scenario 4 In this test we want to observe the pedestrian speeds depending on crowd density over a time frame of 60 seconds. Notably we want to show the relation between pedestrian flow and density, where pedestrian $flow = speed * density$. For this scenario the RiMEA-4 guideline specifies a corridor of width 10 meters and length 1000 meters. In this scenario we needed to decrease the corridor length to be able to handle such numbers of pedestrians. In order to decrease the length of the corridor we need to decrease the pedestrian speeds so they do not run out of the corridor too fast and therefore also decrease the corridor width. A factor of 2 for the corridor width and factor of 10 for the corridor length should be enough. Therefore we need to decrease the speed by a factor of 2 to 0.66 m per second or 0.33 cells per iteration.

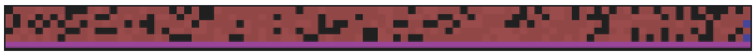
For this test we used the following densities:

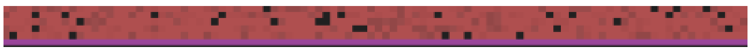
0.5 1 2 3 4 5 6

In order to allow up to 6 pedestrians to occupy the same cell we must increase the limit of pedestrians in a cell and select correct pedestrian avoidance coefficients. In Figure we see the starting point for the RiMEA-4 scenarios for different pedestrian densities.

Flow is defined as average speed at the measuring points multiplied by the overall density at start according to the RiMEA guideline. The density vs flow fundamental diagram 10 seems to have the shape of the such diagrams the flow increasing with density until saturation is reached and then decreasing again as congestion occurs. It is strange that the relationship appears linear but this is most likely due to chance and the small amount of samples. To reproduce visualizations of different densities run the following iPythonNotebook `scenario_4.ipynb`.


(a) The starting point for Rimea Scenario 4 with density 0.5


(b) The starting point for Rimea Scenario 4 with density 1


(c) The starting point for Rimea Scenario 4 with density 4

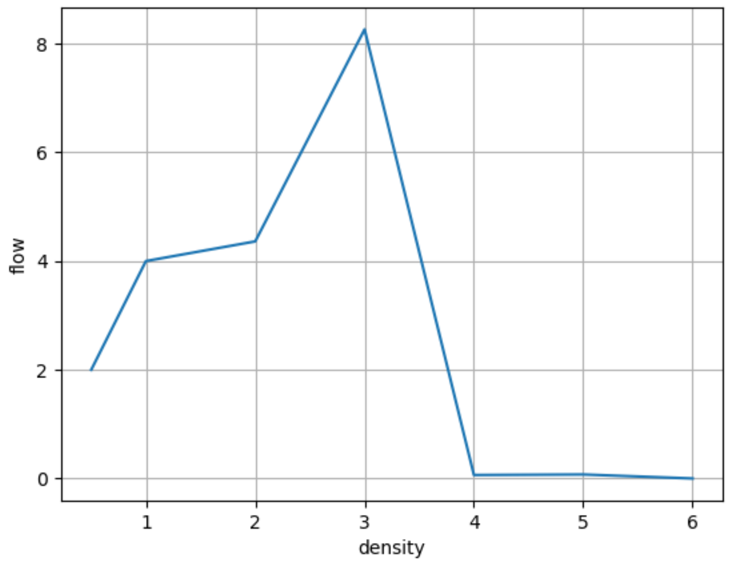
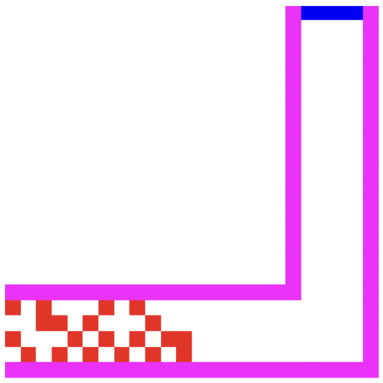
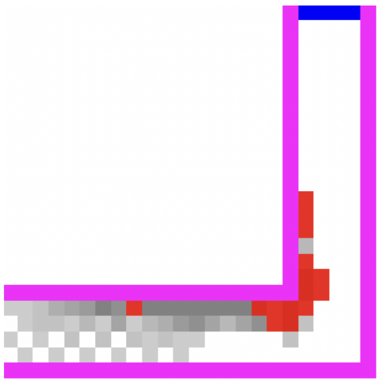


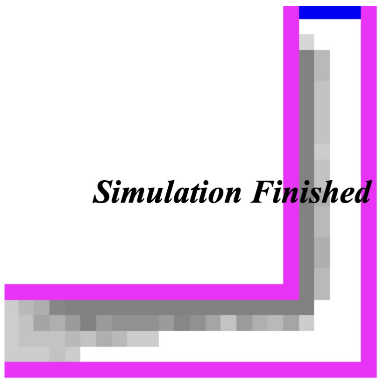
Figure 10: Density vs flow fundamental diagram

RiMEA scenario 6 In this scenario twenty pedestrians are placed uniformly distributed in a hallway of 6 by 2 meters with a turn. It is noteworthy that in this scenario file 2 cells are equivalent to 1 meter. The scenario was setup like in Figure 11a. While rounding the corner the pedestrians form a congestion at the turn as seen in Figure 11b. After most pedestrians round the corner the movement continues like before the turn and finishes in Figure 11c. We observer that in our simulation no pedestrian walks through the walls. Furthermore pedestrians are slowed down from the congestion and pedestrian avoidance and the speeds are visible in Figure 11d. The simulation can be reproduced with the following command:

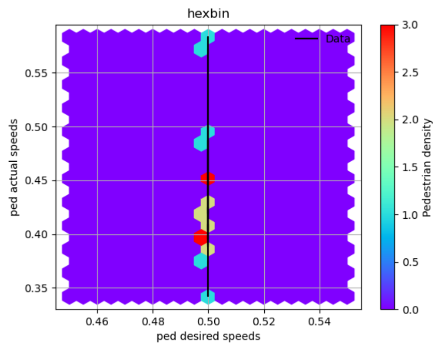
```
python main.py --scenario scenario_RIMEA-6.json
```


(a) RiMEA 6 setup


(b) RiMEA 6 congestion at corner



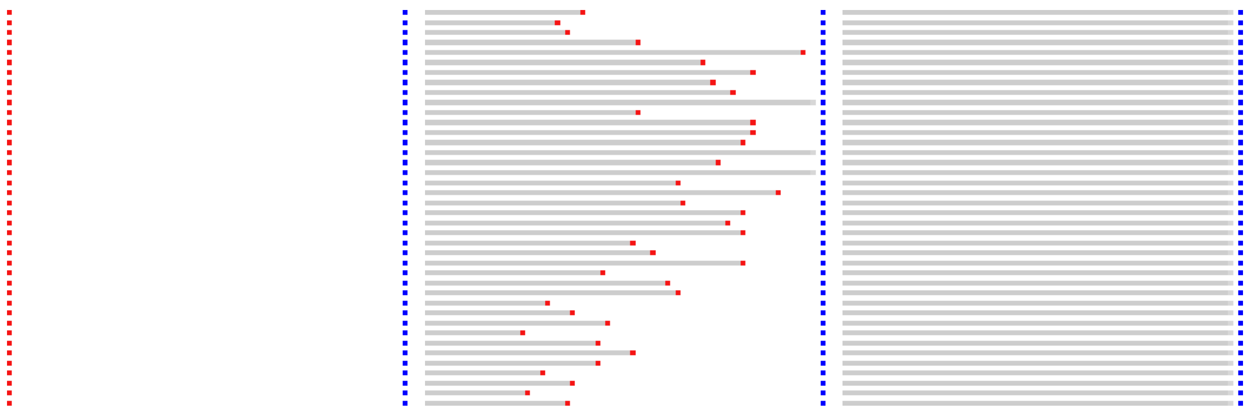
(c) RiMEA 6 finshed



(d) RiMEA 6 speeds

Figure 11: RiMEA 6 simulation

RiMEA scenario 7 In this test we want to sample 50 pedestrians of different ages. Each age group has a different average movement speed and variance of it. In our simulation scenario we choose a 80 square meter area where pedestrians traverse this area from the left to the right. The setup is shown in Figure 12a. Figure 12b shows the different speeds at which pedestrians of different age groups are moving. The result after finishing the simulation is visible in 12c.



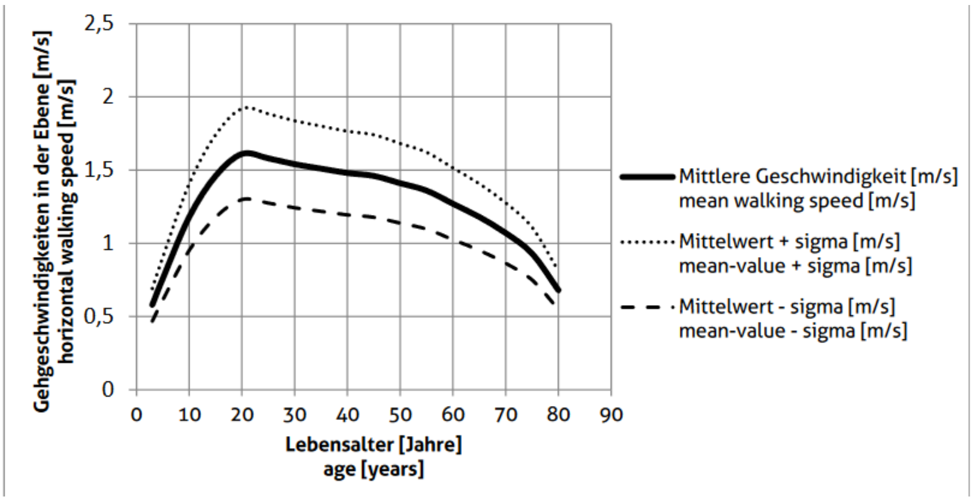
(a) Start of the simulation (b) Middle simulation (c) Finishing simulation

Figure 12: RiMEA 7 simulation

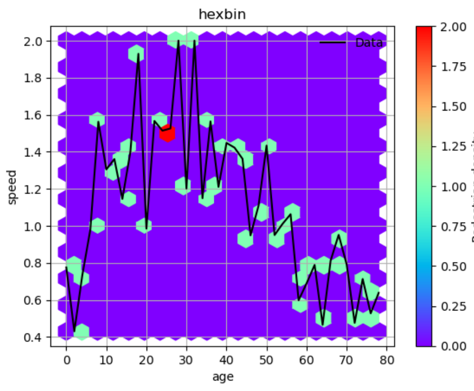
In this test the simulation speeds are halved as each simulation iteration is half a second. For ease of display the pedestrian's starting position's y coordinate is representative of their age. That is 0 to 80 with for the youngest to the oldest age groups. As seen in figure 12b the running simulation forms a speed distribution.

For this test the population speed is specifically modeled using a parameterized normal distribution. It was made to approximate the RiMEA guideline's distribution seen in figure 13a. In the age distribution graph provided in RiMEA-7 we see a mean movement speed with a sigma variance around it. In our scenario the simulation only considers a sample size of 50, hence we can not reproduce the exact distribution, but it is reasonably close. As seen in figure 13b the data points are roughly within the sigma variance of corresponding to the ages as shown in the RiMEA-7 guideline figure. Once again comparing our running simulation in Figure 12b to the histogram Figure 13b we can see the similarity of the speed distribution. In Figure 13c we want to additionally confirm that actual speeds of the simulation actually conform with the desired speeds set for the simulation. Note that figure 13c uses the internal metric for speed, while in Figure 13b the values on the speed axis are converted to meter per second. For a more extensive visualization of the process during simulation run the following iPythonNotebook `scenario_7.ipynb`, which also created the following plots. Note that the last plot from the notebook is the handcrafted reference to the RiMEA age speed distribution. The simulation can be reproduced with the following command:

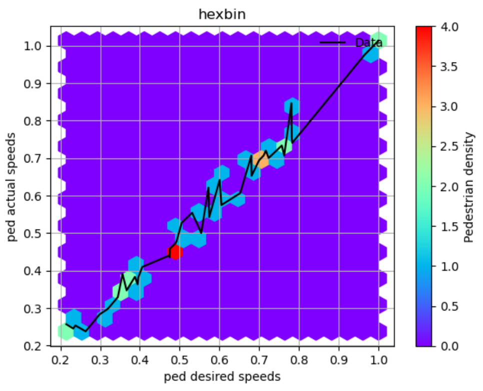
```
python main.py --scenario scenario_RIMEA-7.json --ignorePedestrians
```



(a) RiMEA 7 guideline figure 2 distribution



(b) RiMEA 7 population speed vs age



(c) RiMEA 7 actual speed vs desired speed

Figure 13: RiMEA 7 statistical test

Statistical test In this final section we want to perform a statistical test. The only fast way to get approximately accurate reference samples from the distribution seemed to be generating samples for age groups in multiples of ten. The values were manually measured from the image using `https://eleif.net/photomeasure`. We compare them with the measured actual speeds using the Welch's test from `scipy.stats.ttest_ind` with `equal_var` set to false. We have to multiply our measured speeds by two as the time in our simulation passes half as fast. In our hypothesis test the p value was as is standard practice set to 0.05. The resulting statistic from the function was 0.23, but since we are testing for the opposite of the null hypothesis with a p value of 0.05 we would need a statistic of 0.95 or more. Therefore, it seems our crude approximation is not accurate enough. Though 0.25 seems to indicate that it should not be extremely off. For example when we forget to multiply by two we get a statistic of 0.02 which corresponds to 99.98 percent chance that the result was too extreme for them to have the same mean.

The results of the test can be found at the bottom of the following iPythonNotebook `scenario_7.ipynb` before the last plot.

References

[1] RiMEA. *Guideline for Microscopic Evacuation Analysis.* RiMEA e.V, 3.0.0 edition edition, 2016.