Report for exercise 1 from group J

Tasks addressed:	5
Authors:	Jia Long Ji Qiu Jiabo Wang Yilun Liu
Last compiled:	2023–02–25
Source code:	https://github.com/jialongjq/mlcms

The work on tasks was divided in the following way:

Jia Long Ji Qiu	Task 1	1/3%
	Task 2	1/3%
	Task 3	1/3%
	Task 4	1/3%
	Task 5	1/3%
Jiabo Wang	Task 1	1/3%
	Task 2	1/3%
	Task 3	1/3%
	Task 4	1/3%
	Task 5	1/3%
Yilun Liu	Task 1	1/3%
	Task 2	1/3%
	Task 3	1/3%
	Task 4	1/3%
	Task 5	1/3%

Report on task 1, Setting up the modeling environment

High level description: Firstly, a graphical visualization environment was set up. It consists in an empty grid and a set buttons on its left side (Figure 1). The project is implemented in Python and the **tkinter** package was used for the graphical aspects.

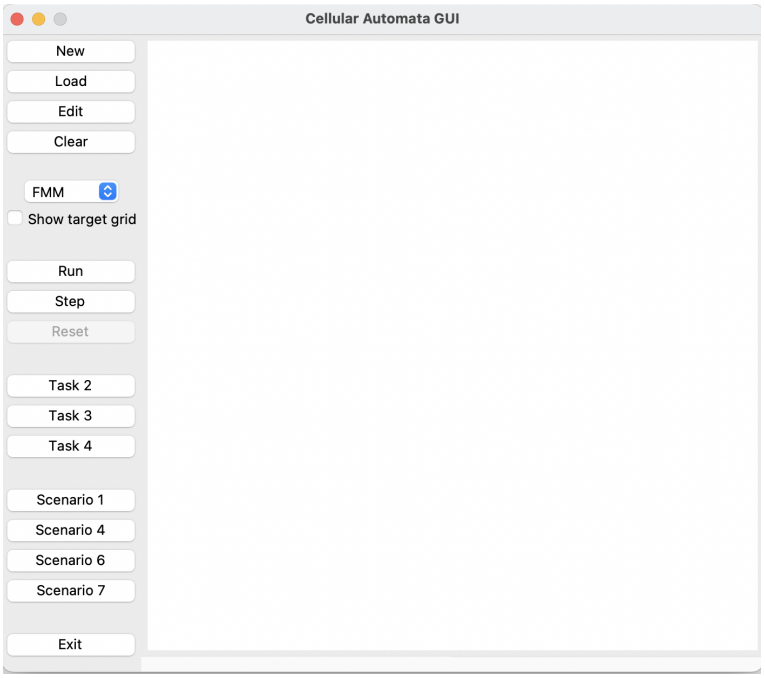


Figure 1: Empty grid with the set of buttons on the left side

The function of the first button **New** is to allow the user to set up the dimensions of the scenario (width and length) as we can see in the figure 2.

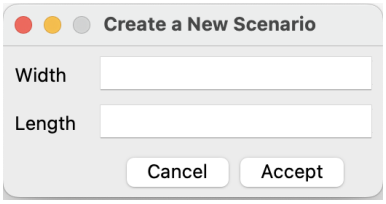


Figure 2: New button window

The next button is **Load** and it allows the user to load a .json file (see an example in figure 3) with personalized scenarios. In this file, it has to be specified the dimensions of the scenario and the positions of the pedestrians, obstacles and targets (obviously inside in the dimensions of the scenario).

```
{
    "width": 50,
    "height": 50,
    "pedestrians": [[[0, 0], 1]],
    "targets": [[49, 0]],
    "obstacles": [[20, 0], [21, 0]]
}
```

Figure 3: Example of a .json file

The button **Edit** allows the user to make an edit of the scenario: it is possible to add or remove pedestrians, targets and obstacles. It has to be specified the position of the objects and in the case of adding pedestrians, it also needs the desired speed that will acquire the pedestrian.

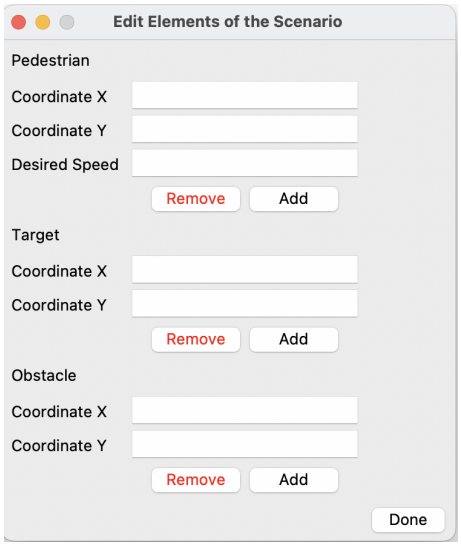


Figure 4: Edit window

In the next position we have the button **Clear** its function is to make ride of all the elements that are in the scenario in that moment: it includes pedestrians, obstacles and targets.

Next there is an **Option Menu** (shown in the figure 5) that allows the user to choose among three different algorithms (FMM, Dijkstra and Basic) that will be used for the computation of distances.

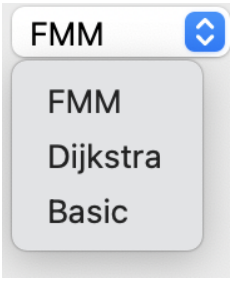


Figure 5: Algorithms Option Menu

Below the option menu, there is a **Check Button** that allows the user to alternate between the normal (figure 6) and target grid distance-based (figure 7) views:

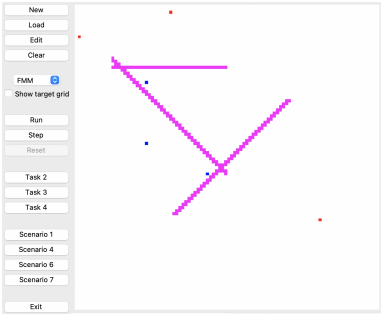


Figure 6: Normal View of the scenario

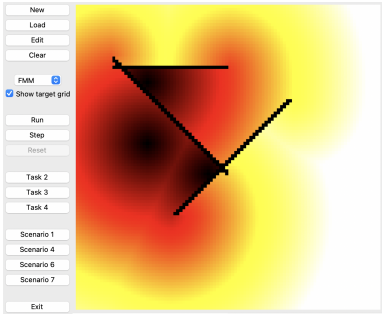


Figure 7: Grid Distance View of the scenario

The next button is **Run** and its function is to run a simulation of the defined scenario, that is the pedestrian

tries to reach the nearest target in the quickest way and avoiding the obstacles.

The button located below is **Step** and its function is similar of the **Run**, but in this case, it only allows the pedestrian to one only movement. A defined can be simulated by one click with **Run** or just clicking the **Step** until all pedestrians reach a target or cannot move.

The following button is the **Restart** button that allows the user to restart the scenario back to the one defined just before running a simulation or making the first step. At the beginning it is disabled and it will just be enable after making a run or a step.

The next buttons **Task 2, Task 3, Task 4, Scenario 1, Scenario 4, Scenario 6, Scenario 7** represents the different scenarios required in the statement. Once clicked the button, it will draw the scenario and by clicking **Run** it will simulate the scenario.

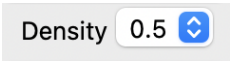


Figure 8: Density Option Menu for the scenario 4

The Option Menu **Density** is to select the different density of pedestrians per square meters for the scenario 4, the options are: 0.5, 1, 2, 3, 4, 5, 6.

Finally, we have the **Exit** button that closes the GUI.

Code and Implementation The project is implemented in **Python** (vers. 3.9.13) and **tkinter** (vers. 8.6). The two main objects are: Scenario and Pedestrian.

The most important attributes of **Scenario** are:

- pedestrians: a list containing all the **Pedestrian** objects
- grid: a numpy array that contains the information of every cell of the scenario, indicating if there is a Pedestrian, a target, an obstacle or it is empty. A pedestrian is represented with the color red, a target is represented with blue and an obstacle is represented as magenta
- target_distance_grid: recomputes the target distances from every cell with the specified algorithm

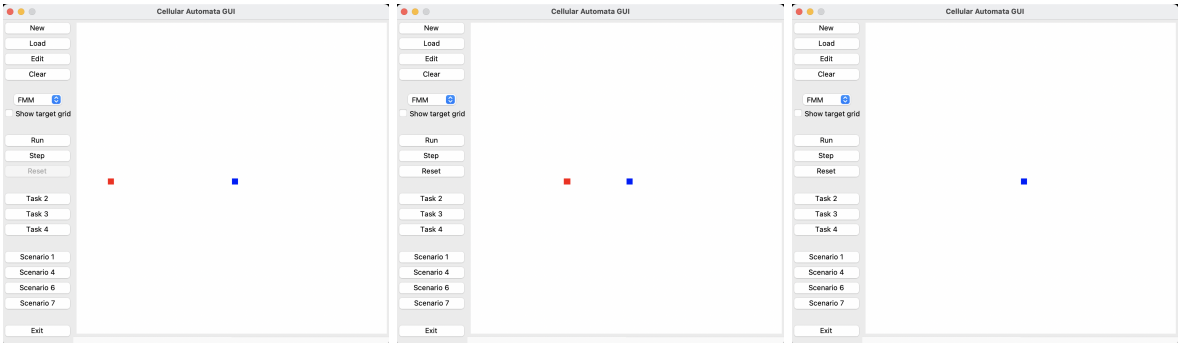
The class **Pedestrian** contains information such as the position and his speed. The main method of the class is:

- update_step: moves to the cell with the lowest distance to the target

The pedestrians are be able to move during a **simulation** because the steps that a pedestrian has to do is based on a basic algorithm that consists of calculating all the distances that each cell of the scenario is to the targets and store the minimum one (that is the nearest target). Then the pedestrian just has to choose among his neighbors cells the one that has the lowest distance to a target and move to it.

Report on task 2, First step of a single pedestrian

Task 2 consists in defining a scenario with 50 by 50 cells (2500 in total), a single pedestrian at position (5, 25) and a target 20 cells away from them at (25, 25).
This scenario can be executed in our cellular automaton step by step or in one fell swoop with the **Run** button. Each cell is 1m x 1m, and the speed of the pedestrian is 1 m/step. After simulating this scenario, the pedestrian reaches the target after 20 steps and waits here in the remaining 5 steps.



(a) Initial setup of the scenario (b) In the middle of the simulation (c) End of the simulation

Figure 9: Screenshoots showing the different stages of the simulation

Report on task 3, Interaction of pedestrians

This task consists in placing five pedestrians equally spaced on a circle in the same distance around a single target in the center of the scenario (Figure 10a), with the objective of showing that they can reach the target at the same time with the same walking speed.

Before this task, the notion of "walking speed" was not introduced yet, so the movement the pedestrians did consisted only in continuously selecting the neighbor cell with the minimum distance to reach the target, without taking into account the speed neither the obstacles. For this reason, the implementation of the walking speed should be explained first.

The walking speed has been defined as "the distance a pedestrian can travel in one tick", where each tick is an execution of a single step. For example, considering the Euclidean distance between cells, if a pedestrian is placed at the position (0, 0) and has a walking speed of 1, it will take 2 ticks to reach the position (0, 2), whereas with a walking speed of 2 it would only take 1 tick.

However, since both the speed and the distance can be positive real numbers, it is not as simple as just moving through the same number of cells in each iteration. Instead, the real distance traveled during one step must be kept in track, so that the remaining distance can be saved for the next one. For example, a pedestrian with a walking speed of 1.5 placed at the position (0, 0), in order to reach the position (0, 3), it should move to (0, 1) in the first tick, save a remaining distance of 0.5 for the next tick, and move from (0, 1) to (0, 3) in the second tick since the available distance to travel will be 2. The same goes for diagonal movements, since two cells placed diagonally have a distance of $\sqrt{2}$, if a pedestrian with a walking speed of 1 placed at the position (0, 0) has to reach the position (2, 2) by only making diagonal movements, it should not move during the first tick, save a distance of 1 for the next tick, move to (1, 1) in the second tick, save $2 - \sqrt{2}$ for the next tick and move to (2, 2) in the third tick, since the accumulated distance $1 + 2 - \sqrt{2}$ will be greater than $\sqrt{2}$. This way, the real walking speed in a simulation will be close to the desired speed in the long run. A pseudo-code of this algorithm is described in Algorithm 1

Algorithm 1 Update Step

```
available distance = movement speed + accumulated distance
accumulated distance = 0
while available distance > 0 do
    next cell distance = distance in current position
    next position = current position
    for each position of the unoccupied neighbors do
        if distance to target in neighbor position < next cell distance then
            update next position and next cell distance to the corresponding neighbor position
        else if distance to target in neighbor position == next cell distance then
            keep the position with shorter walking distance
    end for
    if next position ≠ current position then
        distance to travel = Euclidean distance from current position to next position
        if distance to travel ≤ available distance then
           subtract the distance to travel to the available distance
           move to the next position
        else empty the available distance and save it in accumulated distance
    else break
end while
```

Now, proceeding to the simulation, the scenario dimensions are 50 x 50, so the target is placed in the position (24, 24) and the five pedestrians are placed as follows:

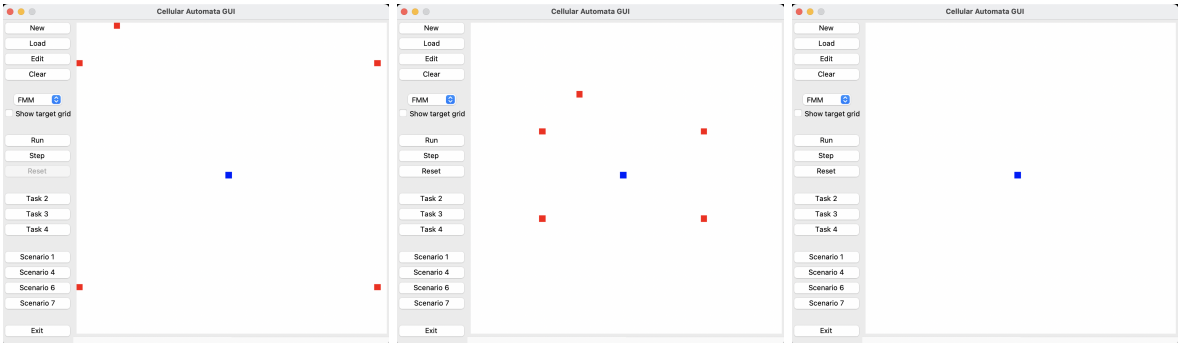
- pedestrian 1: ((0, 6), 1)
- pedestrian 2: ((0, 42), 1)
- pedestrian 3: ((6, 0), 1)
- pedestrian 4: ((48, 6), 1)

- pedestrian 5: ((48, 42), 1)

All pedestrians are located in an Euclidean distance of 30 units from the target and all of them have the same walking speed, so that all of them can reach at the same time the target.

As in the previous task, this task can be simulated step by step or in one fell swoop with the **Run** button. Figure shows the different stages of the simulation. The goal of the task is that all the pedestrians reach the target, so the target have been made to be "absorbing" (Figure 10c).

After simulating the scenario defined previously, all the pedestrians reach the target roughly at the same time because they are at the same distance.



(a) Initial setup of the scenario (b) In the middle of the simulation (c) End of the simulation

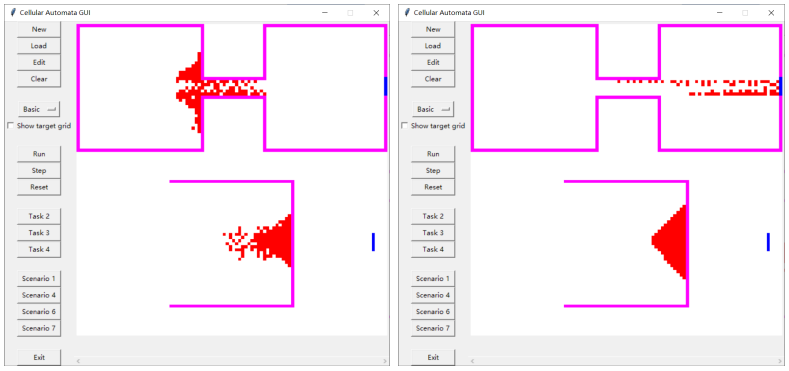
Figure 10: Screenshoots showing the different stages of the simulation

Report on task 4, Obstacle avoidance

Task 4 consists of different implementation of obstacle avoidance functions. To test the performance of these algorithms, we constructed the environment consisting of two different scenarios, the bottleneck and the "chicken test". For both scenarios, 150 pedestrians were distributed randomly and uniformly within the respective areas.

Figure 11: The bottleneck and "chicken test" scenarios

First, for the rudimentary obstacle avoidance for pedestrians (i.e., without obstacle avoidance algorithms), all pedestrians moves with no a priori knowledge of obstacles, but only considering direction towards the nearest target, thus cannot pass the "chicken test" scenarios successfully. However, they managed to successfully pass the bottleneck scenario, although instead of going straight towards the bottleneck for exiting the first room, they rushed to the right and slowly walk along the wall to the outlet. This is because there is still a descending component of the distance gradient that pointing towards the correct direction at the boundaries around the bottleneck, which is similar to the case in the RiMEA scenario 6 of task 5.



(a) In the middle of the simulation (b) Near the end of the simulation

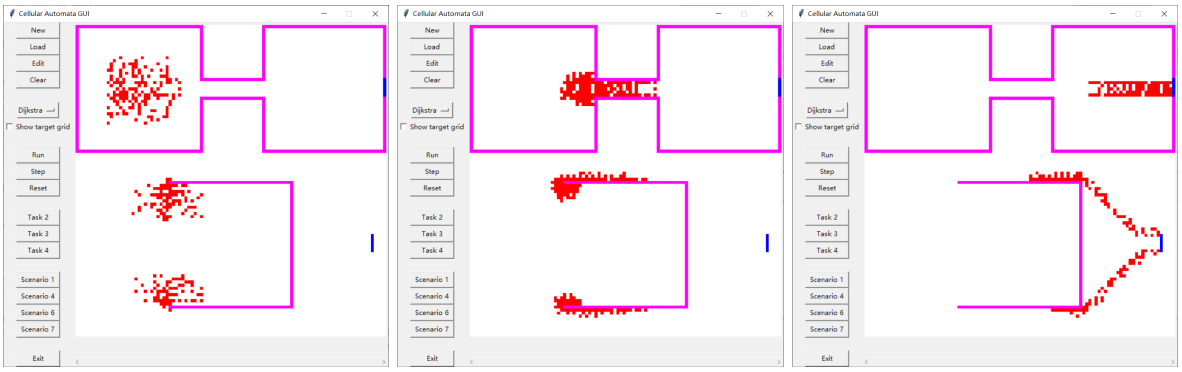
Figure 12: With the basic algorithm, pedestrians failed the "chicken test", but eventually managed to pass the bottleneck scenario

To solve this problem, we firstly implemented the naïve Dijkstra algorithm for flooding cells with distance values from the cells at the target to the entire map.

Algorithm 2 Dijkstra Flooding on Distance Map

```
closed[] = FALSE
closed[target cell positions] = TRUE
distances[] = ∞
distances[target cell positions]
open positions = get open neighbors(closed positions)
step counter = 1
while step counter < step limit AND NOT open positions is empty do
    next open positions = []
    for each open position do
        distances[open position] = the neighbors' smallest distance + distance to that neighbor
        closed[open position] = TRUE
        next open positions += (get open neighbors(closed))
    end for
    open positions = next open positions
    step counter += 1
end while
return distances
```

Based on such algorithm, the distance grid is now flooding from the target to every reachable cells on the map, and the pedestrians can reach the targets in both scenarios. Result of the experiment is shown in Figure 13 on the following page.

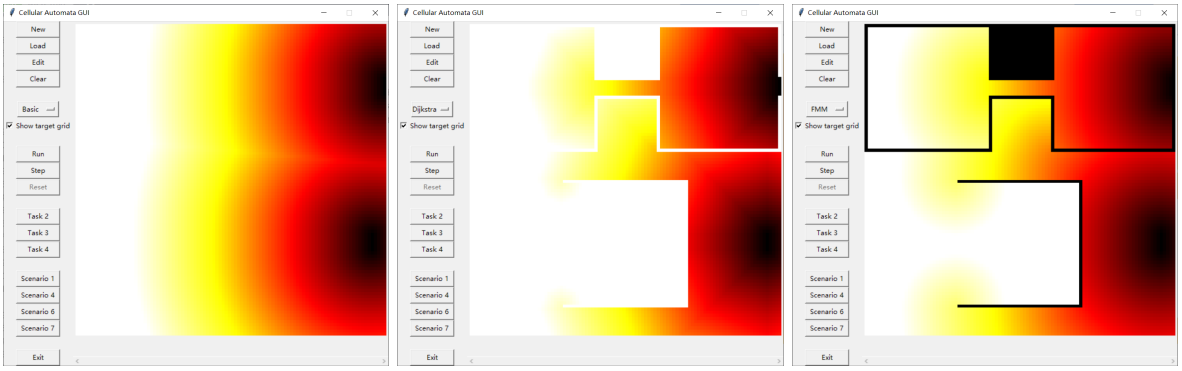


(a) At the beginning of the simulation (b) In the middle of the simulation (c) Near the end of the simulation

Figure 13: With the Dijkstra flooding algorithm, pedestrians gained the ability to pass all obstacles and can pass both scenarios now

In order to have a more accurate computation of the distance field, we also implemented the Fast Marching Method by Sethian, using a third-party function provided in the library **scikit-fmm**. The process of flooding computation and the behavior of pedestrians are similar in both Dijkstra and FMM algorithms, for which and due to space constraints we are not presenting the pseudo-code and images of pedestrians again.

The following figure shows the different result of the distance field computed by the basic, Dijkstra and FMM algorithms (Figure 14). From which we can see that the basic algorithm failed to take any obstacles into consideration, the Dijkstra one shows a clear eight-maned star pattern, which was generated because each cell's distance is calculated on the basis of the eight neighbors' distances, rather than a real representation of distance towards the target. And the result from FMM algorithm has shown an accurate, smooth distance field, with values representing the true distance to the nearest target, which has the best performance among these three algorithms.



(a) The Basic Algorithm (b) The Dijkstra Flooding Algorithm (c) The Fast Marching Method

Figure 14: Result of the distance field computed by different algorithms

Report on task 5, Tests

TEST1: RiMEA scenario 1 (straight line)

The RiMEA scenario 1 consists in a single pedestrian with 40 cm body dimension located in the beginning of a 2m wide and 40m long corridor (Figure 15). This scenario tries to show that a pedestrian with a defined walking speed will cover the distance in the corresponding time period.

For this scenario, it has been consider that a cell has 40cm x 40cm dimensions. So that, the pedestrian of 40cm body dimension is located only in one cell. The corridor then will be 100 cells (100 x 40cm = 40m) away from the pedestrian and as the pedestrian has a walking speed of 1.3m/step = 3.25 cells/step, the pedestrian will have to reach the target after 31 steps (100/3.25 = 30.76).

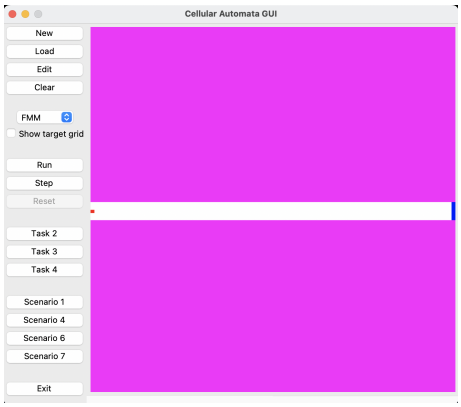
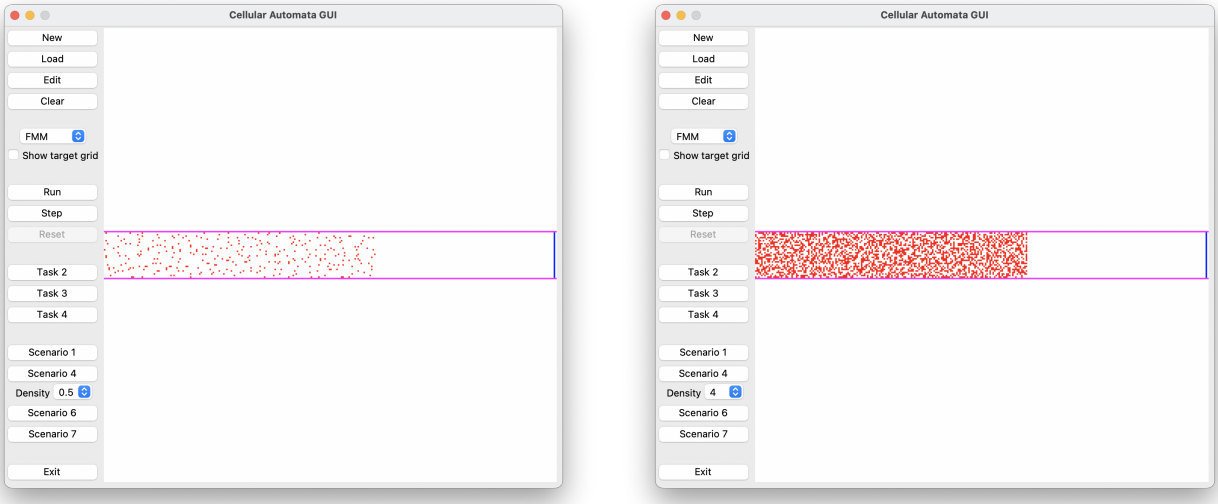


Figure 15: Initial setup of the scenario

In fact, after doing the simulation the pedestrian has reached the target after 31 steps.

TEST2: RiMEA scenario 4 (fundamental diagram: density vs. velocity or density vs. flow)

The RiMEA scenario 4 describes a corridor (1000 m long, 10 m wide) to be filled with different densities of persons with an equal as possible free walking speed (for example 1.2 – 1.4m/s): 0.5 P/m^2, 1 P/m^2, 2 P/m^2, 3 P/m^2, 4 P/m^2, 5 P/m^2 and 6 P/m^2. Considering that the original corridor is too long to implement and display the results efficiently, we hereby only select the 200m part in the middle of the corridor to test our algorithms. Based on the implemented function to fill in pedestrians according to given densities, we further provided GUI for choosing the aforementioned densities, and test the corresponding behaviors within the corridor.



(a) Density = 0.5 P/m^2 (b) Density = 4 P/m^2

Figure 16: GUI and changed environment for different densities

For the measuring points in the test scenarios, we transformed the original calculation of flow = speed × density into the number of pedestrians getting across a unit length of border into the measuring area within a unit time unit, i.e., the number of pedestrians passing the left border (for all pedestrians can only move towards right) in each second, divided by the length of the left border. This is based on the calculation that $(m/s) \times (P/m^2) = P/(s \times m)$.

We implemented the gathering information of pedestrians passing borders within the process of updating each pedestrians' positions. For each time a pedestrian's position is updated, it checks for each measuring points whether the pedestrian is the case, that the previous position is outside border (beyond left boundary), and the new position is within the measuring area. If so, then the current time is added into the dictionary containing all measuring points as keys and a list of time that all pedestrians counted passed through the boundary as values. After all pedestrians reached the destination, the dictionary is exported for aggregation of useful information and eventually calculating the flow.

Here we show the result of flow over time on all 3 measuring points for 3 different densities of pedestrians: 6P/m^2, 4P/m^2 and 2P/m^2, in the Figure 17. From which we can see that the system managed to capture the characteristic of "overcrowding", that is the non-linear relation between increase in density, flow and total time used for all pedestrians to reach the target.

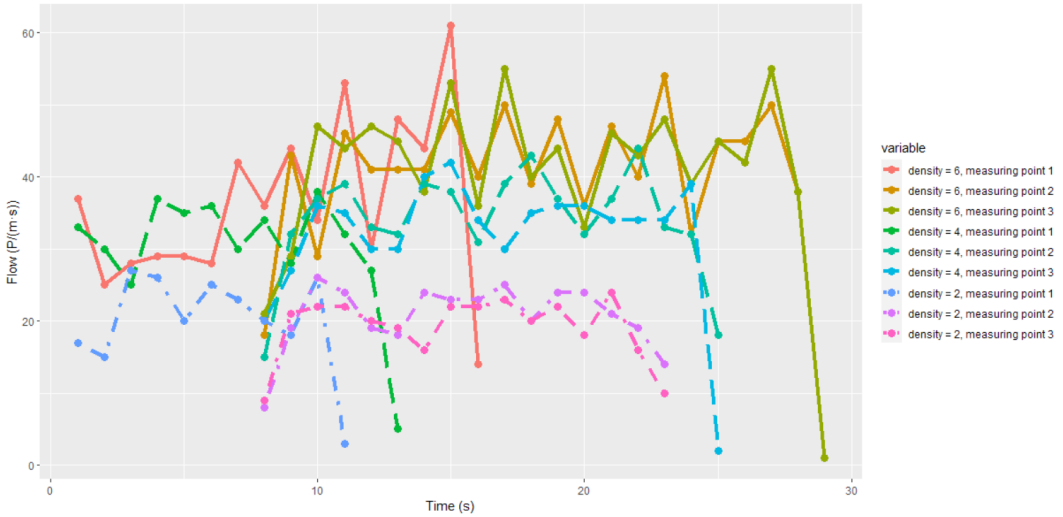
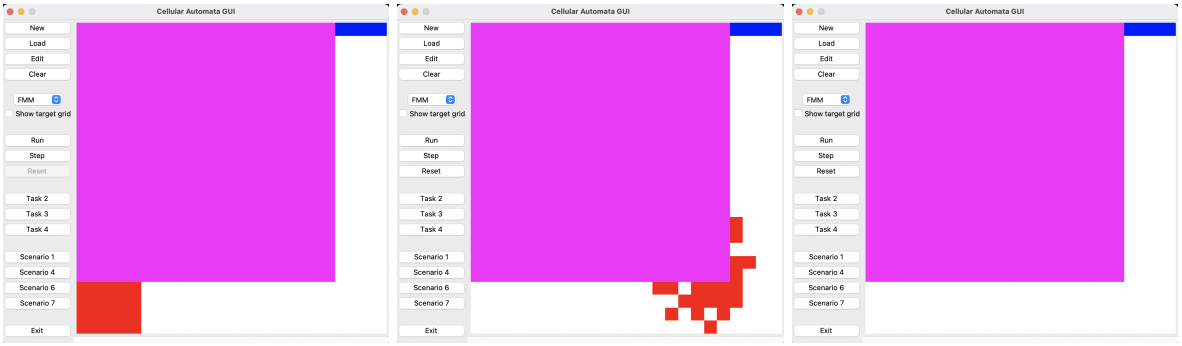


Figure 17: Flow over time for different densities, counted at 3 measuring points

As we can see, the increase of density from 2 to 4 has caused more increase in flow, and also less increase in total evacuation time, than the one of density from 4 to 6. It also seems that the total amount of area beneath the line of flow over time stays linear with the density, and the more dense pedestrians has gathered from the beginning, the less efficient they are in moving towards the target, and the longer time it takes for all of them to reach the target. Reasonable result like this indeed proved that our simulation is able to successfully display basic features of crowd dynamics.

TEST3: RiMEA scenario 6 (movement around a corner)

The RiMEA scenario 6 describes a group of 20 pedestrians moving towards a corner which turns to the left (Figure 18a) will successfully go around it without passing through walls. The objective of this scenario is to show that the pedestrians in the simulation can avoid the obstacles.



(a) Initial setup of the scenario (b) In the middle of the simulation (c) End of the simulation

Figure 18: Screenshoots showing the different stages of the simulation

In fact, the pedestrians can avoid the wall (Figure 18b) and reach the targets (Figure 18c).

TEST4: RiMEA scenario 7 (demographic parameters)

The RiMEA scenario 7 describes a group of 50 pedestrians aged between 18 and 80, with walking speeds distributed in accordance with Figure 19. The aim of this scenario is to show that the distribution of walking speeds in the simulation is consistent with the distribution in the table.

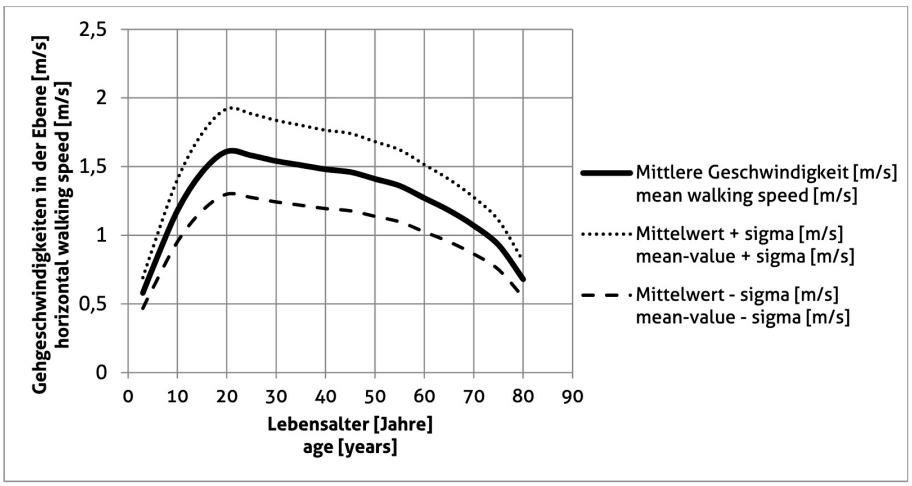


Figure 19: Walking speed in the plane as a function of age based on Weidmann [1]

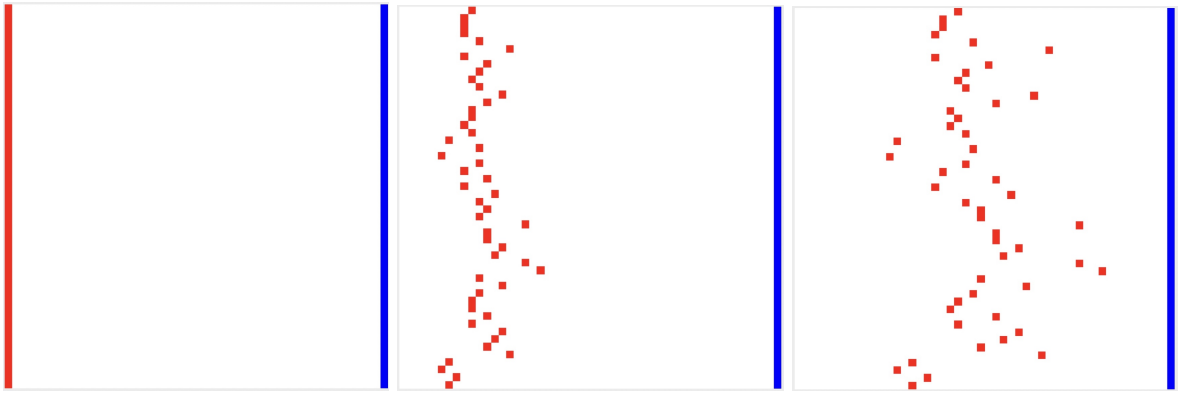
For this task, the main feature implemented was the assignment of the walking speeds to pedestrians depending on their ages. By taking a look at the Figure 19, it can be seen that the distribution does not follow any pattern, so the implementation had to be done in an approximate way. Therefore, for every different age interval, a mean walking speed and a standard deviation has been assigned (another option would have been taking every single age and attributing it its corresponding walking speed and sigma parameter, but defining age intervals should be enough if accuracy is not the main objective). By doing so, the walking speeds can be assigned with a normal distribution so that with a large enough number of pedestrians or different executions, on average, the results would be as similar as possible to the figure. The implementation of this feature is shown in Listing 1

```python
def assign_speeds(self):
    for pedestrian in self.pedestrians:
        age = pedestrian.age
        if age < 20:
            mean_speed = 1.5
            std = 0.3
        elif age >= 20 and age < 30:
            mean_speed = 1.6
            std = 0.25
        elif age >= 30 and age < 40:
            mean_speed = 1.5
            std = 0.25
        elif age >= 40 and age < 50:
            mean_speed = 1.4
            std = 0.25
        elif age >= 50 and age < 60:
            mean_speed = 1.3
            std = 0.2
        elif age >= 60 and age < 70:
            mean_speed = 1.1
            std = 0.1
        else:
            mean_speed = 0.9
            std = 0.1

    pedestrian._desired_speed = np.random.normal(mean_speed, std)
```

Listing 1: Normal distribution of walking speeds depending on different age intervals

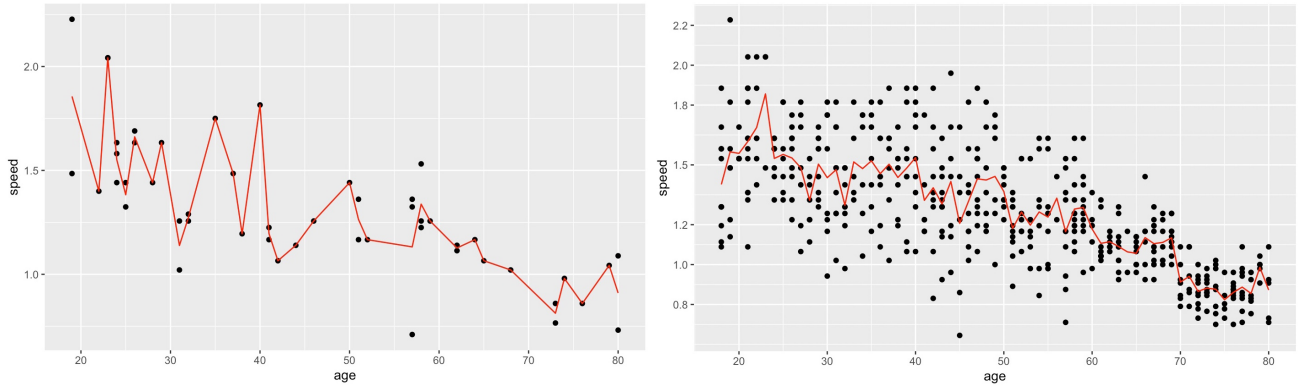
The scenario has dimensions 50x50, with 50 pedestrians placed in the first column and 50 targets placed in the last column, as shown in Figure 20a. As it can be observed in Figures 20b and 20c, the pedestrians are moving in different walking speeds as expected, but it must verified that the real walking speed and the assigned walking speed of each pedestrian is roughly the same. Since the measurement units are metres per second, the Euclidean distance unit has been considered as metres and each tick as a second. Therefore, the real speed has been calculated dividing the distance walked by the time spent until reaching the target.



(a) Initial setup of the scenario (b) Start of the simulation (c) Middle of the simulation

Figure 20: Different stages of the simulation of the scenario 7

Now, moving on to the results of the simulation, by looking at the Table 1, it can be noticed that there is not much of a difference between the expected speed and the real speed, which means that the implementation for the walking speed is quite accurate. As for the relation between the walking speed and the age of the pedestrians, the experiment done consisted in executing multiple simulations (with randomized ages in each execution) and saving the results, in order to obtain a plot similar to the Figure 19. This statistical test is appropriate since, as it was said before, with the defined normal distribution, the results obtained over a large number of observations should make a good representation of the plot to be recreated. By comparing the obtained scatter plot displayed in Figure 21b and the original plot, it can be noticed that there are similarities in the tendency of the mean walking speed as well as in its deviation. Therefore, it has been concluded that both the experiment and the simulation were successful.



(a) Scatter plot with 50 observations

(b) Scatter plot with 500 observations

Figure 21: Scatter plots of the results of simulation of the scenario 7.

Id	Age	Expected speed	Real speed
34	19	2.2626140044332383	2.227272727272727
28	23	2.0700413935501727	2.0416666666666665
33	23	2.0679226648511855	2.0416666666666665
5	40	1.8606775661769244	1.8148148148148149
45	35	1.7793918769286883	1.75
11	26	1.728591068450894	1.6896551724137931
31	24	1.6550880474807992	1.6333333333333333
36	26	1.6829926254717111	1.6333333333333333
42	29	1.64597148257722	1.6333333333333333
24	24	1.5837790879854388	1.5806451612903225
32	58	1.5405490546592486	1.53125
12	37	1.499876674569721	1.4848484848484849
43	19	1.5110510201690972	1.4848484848484849
22	24	1.4629414084157466	1.4411764705882353
29	28	1.4824884818394477	1.4411764705882353
30	50	1.4600818630545707	1.4411764705882353
40	25	1.4605886952023632	1.4411764705882353
7	22	1.4333495247419161	1.4
26	51	1.3864354104170407	1.3611111111111112
44	57	1.3829543615268625	1.3611111111111112
27	57	1.3334203074494406	1.3243243243243243
35	25	1.3569925704783334	1.3243243243243243
4	32	1.2973633189369191	1.2894736842105263
8	59	1.2776870293887876	1.2564102564102564
10	58	1.258068408827545	1.2564102564102564
18	31	1.282620815397896	1.2564102564102564
20	32	1.2686869546813793	1.2564102564102564
37	46	1.2866345006167508	1.2564102564102564
16	41	1.2385384732818703	1.225
25	58	1.2539416513406914	1.225
0	38	1.2219406662223529	1.1951219512195121
9	51	1.1871016878253722	1.1666666666666667
14	64	1.1694649731151314	1.1666666666666667
38	41	1.172539973490902	1.1666666666666667
41	52	1.1703506965304746	1.1666666666666667
13	44	1.1630744989726405	1.1395348837209303
39	62	1.1519485072143882	1.1395348837209303
15	62	1.116107492441249	1.1136363636363635
2	80	1.09641011670902	1.0888888888888888
1	65	1.068861145013333	1.065217391304348
21	42	1.0883556402223342	1.065217391304348
3	79	1.0426807338195663	1.0425531914893618
6	31	1.039429651889844	1.0208333333333333
23	68	1.0273602842593716	1.0208333333333333
48	74	0.9835593918854721	0.98
46	76	0.8684178118198739	0.8596491228070176
49	73	0.8621006300863451	0.8596491228070176
17	73	0.7705855651335983	0.765625
47	80	0.7419466240094087	0.7313432835820896
19	57	0.7113355523168826	0.7101449275362319

Table 1: Results of a simulation of the scenario 7

References

[1] Weidmann U., Transporttechnik der Fußgänger, Schriftenreihe des Institut für Verkehrsplanung, Transporttechnik, Strassen- und Eisenbahnbau Nr. 90, S.35-46, Zürich, January 1992.

[2] RiMEA. Guideline for Microscopic Evacuation Analysis. RiMEA e.V., 3.0.0 edition, 2016. `www.rimea.de`.