Report for exercise 1 from group E

Tasks addressed:	5
Authors:	Zhitao Xu (03750803) Vatsal Sharma (03784922) Minxuan He (03764584) Çağatay Gültekin (03775999) Gaurav Vaibhav (03766416)
Last compiled:	2023–11–02
Source code:	https://gitlab.lrz.de/mlcms-ex-group-e/mlcms-ex-group-e

The work on tasks was divided in the following way:

Zhitao Xu (03750803)	Task 1 Task 2	100% 100%
Vatsal Sharma (03784922)	Task 3	100%
Minxuan He (03764584)	Task 4	100%
Çağatay Gültekin (03775999)	Task 5	50%
Gaurav Vaibhav (03766416) **Project lead**	Task 5	50%

Report on task 1, Setting up the modeling environment

In this exercise, we will model and simulate a human crowd with a cellular automaton. The automaton is arranged in a two-dimensional grid of cells. We place pedestrians, targets and obstacles in different cells as an initial state and set some rules so that the pedestrians can move towards the targets and interact with each other and the environment. In this task, we successfully set up the modeling environment using Python and visualize it through a graphical user interface (GUI). In the following, the detailed explanations of the environment and the GUI will be introduced.

1.1 Pedestrian Class

We define a *Pedestrian* class to represent an individual pedestrian in the scenario. A pedestrian is initialized with its two attributes, i.e. *position* and *desired_speed*. The attribute *position* is a two-dimensional tuple representing the position of the pedestrian on the grid. The origin of the grid is located at the top-left corner, with the x-axis extending horizontally to the right and the y-axis extending vertically downward. For example, the position of the top-left grid cell is (0, 0). The cell to its right is (1, 0), and the cell below it is (0, 1).

The *Pedestrian* class has two methods, and they are introduced respectively as follows:

1. Method: *get_neighbors*

   This method computes all neighbors in an 8-cell neighborhood around the pedestrian's current position, and returns a list of neighbor cell indices. However, the cells whose positions are out of the range of the grid width and height are not considered as neighbors.

2. Method: *update_step*

   This method calls *get_neighbors* to get the list of positions of the neighbors of the current pedestrian. Then the method *terget_distance_grids* of the instance scenario, which will be introduced later, is called to compute the distances from each grid cell to its nearest target. After that, the method iterates through all the neighbors to find the neighbor cell closest to the target and updates the position attribute of the current pedestrian to match the position of this neighbor cell, effectively moving the current pedestrian to the location of that neighbor cell. During this process, the obstacles are excluded from the 'neighbors' by checking the values of the corresponding positions in the 'grid' matrix of the instance scenario. This is how we make the obstacle cells inaccessible. Besides, the pedestrians are allowed to occupy the same cell.

1.2 Scenario Class

The *Scenario* class is a critical component responsible for modeling and managing the simulation environment. It encompasses various aspects of the simulation, including the presence of obstacles, targets, the positions and movements of pedestrians, and methods for visualization.

Since each cell has 4 possible states, in order to materialize them into a matrix, we need to assign an integer to each state as its numerical ID, 0 for 'EMPTY', 1 for 'TARGET', 2 for 'OBSTACLE', and 3 for 'PEDESTRIAN'. Thus we define two dictionaries which set up a mapping between the states and their numerical IDs. We also define a dictionary which maps from the state to a 3-dimensional tuple corresponding to a RGB color. Besides, we specify the GRID_SIZE to represent the size of the visual grid.

The attributes and the methods are introduced as follows:

1. Attributes

   When the scenario instance is initialized, its *width* and *height* need to be specified. They define the size of the cellular automaton. The attribute *grid* is a two-dimensional Numpy array that represents the state of each cell in the scenario, including empty cells, targets, obstacles, and pedestrians. States are represented by numerical IDs. The attribute *pedestrians* is a list containing all pedestrians within the scenario, with each pedestrian being an instance of the *Pedestrian* class. The attribute *target_distance_grid* is a two-dimensional array storing the distances from each grid cell to its nearest target cell.

2. Method: *recompute_target_distances*

   This method recalculates the distances from every grid cell to its nearest target cell. This method is called during initialization and whenever changes occur in the scenario.

3. Method: *update_target_grid*

   This method computes the shortest distance from every grid cell to its nearest target cell in the scenario. We first collect the coordinates of all the targets according to the attribute *grid*. If there are no targets in the scenario at all, this method returns a matrix of the same shape of *grid* with all values to be zero. If the number of the targets is larger than zero, then we compute the distances from every cell to all the targets with *cdist* function from *Scipy*, obtaining a 2-D array of the shape (number of the targets, total number of cells). After that, we compute the minimum along the columns to get the distances from every cell to its nearest targets. Finally, the result is reshaped to the shape (width, height).

4. Method: *update_step*

   This method updates the positions of all pedestrians in the scenario, responsible for advancing pedestrian movements within the simulation.

5. Method: *cell_to_color*

   This is a static method that maps a cell's numerical ID to its corresponding color.

6. Method: *target_grid_to_image* and Method: *to_image*

   The method *target_grid_to_image* creates a visual representation of the distances from every cell to its nearest target based on the attribute *target_distance_grids*. The method *to_image* creates a visual representation of the scenario based on cell IDs in the grid, including pedestrians.

1.3 MainGUI Class

The *MainGUI* class is responsible for creating a graphical user interface and the visualization of the scenario and the movements of pedestrians. It facilitates user interaction by providing buttons for controlling the simulation, creating scenarios, and viewing the results.

1. Method: *start_gui*

   This method serves as the entry point for the GUI. It creates the main window and sets up the interface elements, including buttons and canvas for visualization. The GUI also contains a menu which is accessible from the top bar and includes options for creating a new scenario, restarting the simulation, and closing the GUI. A *Canvas* instance is created and it is the place where the scenario and its elements, such as pedestrians, obstacles, and targets, are visually represented. Then a *Scenario* instance is created. Its attribute *grid* is initialized with a zero matrix at the beginning. Based on the data we read from a JSON file, we have the positions of targets, obstacles, and pedestrians. Subsequently we set the corresponding entries in the *grid* to their numerical IDs. The attribute *pedestrians*, which essentially is a list of instances of *Pedestrian* class, is initialized with the positions of pedestrians. Besides, four buttons are available for user control:

   - Step simulation: This button allows the user to manually step the simulation, advancing each pedestrian one step at a time.
   - Automated step: This button initiates an automated simulation, moving pedestrians towards their targets until all targets are reached.
   - Restart simulation: This button restarts the simulation using the initial parameters.
   - Create simulation: This button opens a new window for users to input parameters to create a new scenario.

   A visualization of an example scenario is shown in Fig. 1 left. The pedestrians, targets, and obstacles are shown in red, blue, and magenta respectively.

2. Method: *get_scenario*

   This method returns a dictionary containing scenario parameters loaded from a JSON file. The parameters are organized as a dictionary containing the size of the automaton, positions of targets and obstacles, and positions and speeds of pedestrians.

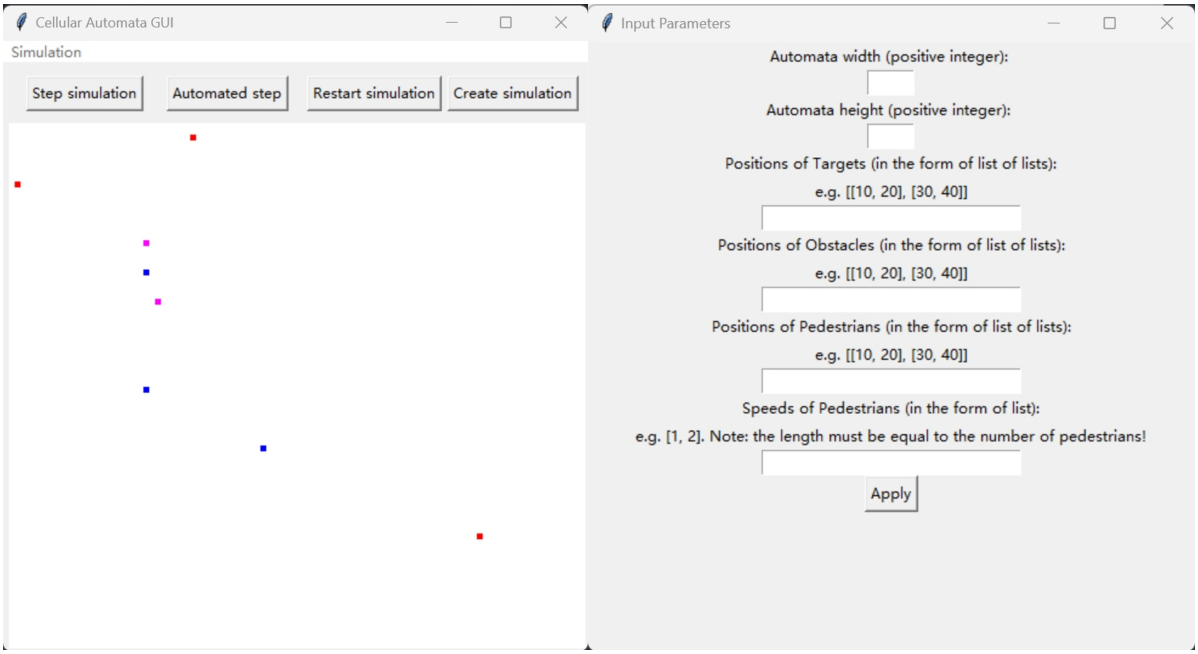


Figure 1: The GUI and the input window.

3. Method: *step_scenario*

   This method calls the *update_step* method of the scenario instance to move each pedestrian towards its target by one step and then calls the *to_image* method to visualize the changes.

4. Method: *automated_step*

   This method automatically calls the *step_scenario* method continuously until every pedestrian reaches a target by checking if a pedestrian's position coincides with a target position. Each time the *step_scenario* is called, we let the program pause for 0.3 seconds in order to make the movements of pedestrians perceivable.

5. Method: *restart_scenario*

   During the simulation, the positions of the pedestrians change when we click the 'Step simulation' or 'Automated step' button. This method restarts the scenario by setting the positions of pedestrians to their initial positions and then visualizes it.

6. Method: *create_scenario* and Method: *save_input_parameters*

   These two methods enable users to input scenario parameters to create a new scenario and visualize it. The method *create_scenario* opens a new input window, shown in Fig. 1 right, once we click on the button 'Create simulation'. The input window contains several widgets, including Labels and Entries. The Entries are used to obtain the text inputs from users. Above each Entry, there is a Label pointing out that what kinds of inputs and in what form they are expected. The parameters that the users can set include the width and the height of the automaton, the positions of targets, obstacles, pedestrians and the speeds of pedestrians. In such way, the scenario can be reset without changing the code. There is also an 'Apply' button at the bottom. Once we click on this button, the method *save_input_parameters* is called, resulting in the vanishing of the input window, and then the inputs are stored and written in the JSON file. Following this, the attributes of the instance scenario are updated based on the new parameters. Finally, the *to_image* method is called to visualize the new scenario.

1.4 Overall Flow

First, the GUI is initialized, presenting buttons and the initial scenario. Users can manually or automatically step the simulation. The code flow is organized as a loop to simulate pedestrian movement and display the results in real-time. Besides, users are allowed to restart the scenario or create a new scenario by providing

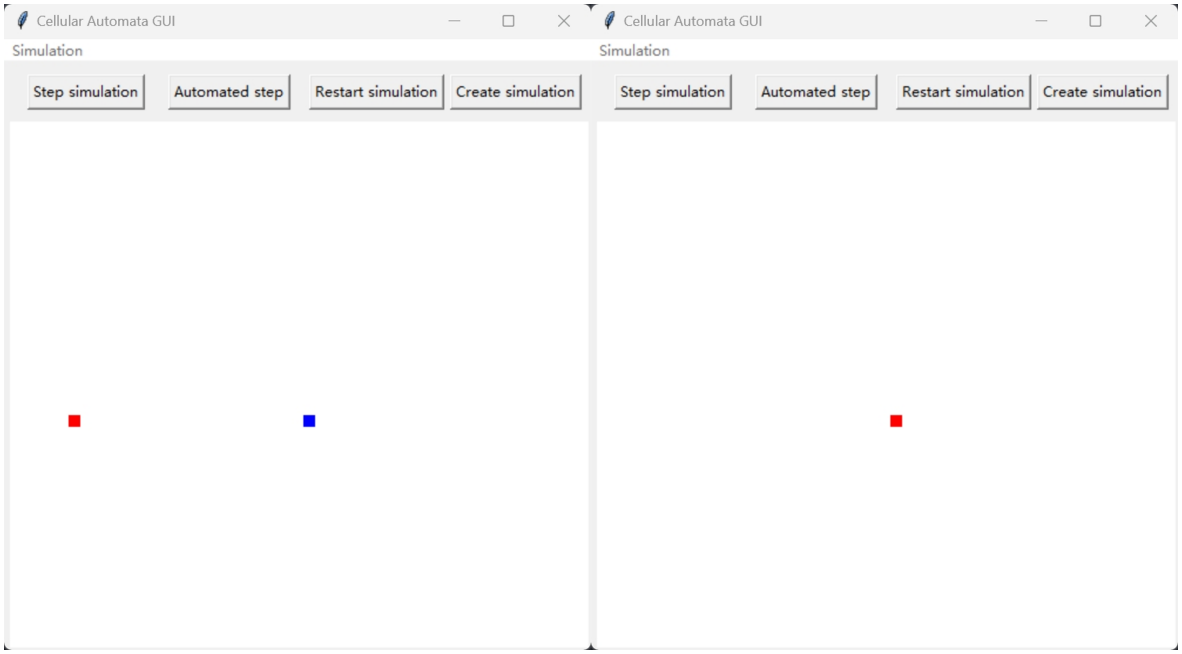


Figure 2: Visualization of the initial state and the final state.

parameters. Finally, the GUI can be closed by clicking on the 'close' option in the menu when users are done with the simulation.

Report on task 2, First step of a single pedestrian

As required, we set the scenario with 50 by 50 cells, and set a pedestrian at (5, 25) and a target at (25, 25) through the input window introduced above. We click on the button 'Apply', and then we have the scenario as shown in Fig. 2 left. We click on the 'Step simulation' for 25 times. We observe that the pedestrian moves horizontally towards the target and reaches the target after the 20th click. After that, the pedestrian stays at the target and the scenario remains unchanged as shown in Fig. 2 right.

Report on task 3, Interaction of pedestrians

This question introduces a new factor of "speed" to the setup of previous questions.
It bascially asks us to implement these features -

- A scenario with 5 pedestrians placed at the vertices of a regular pentagon, positioned around a single target.
- Target should exhibit an "absorbing" nature, that is a pedestrian should be removed as soon as it lands on the target cell.
- The word "distance" must point to the idea of "Euclidean distance" rather than the "number of cells", while implementing equal distances for all the pedestrians.
- A pedestrian should be able to traverse through its space arbitrarily with roughly the "same speed" in all 8 directions.

3.1 Scenario Setup

Since a specific scenario was required for the setup, all the pedestrian and target details were hard-coded and saved in a file named *scenarioq3.json*. These can be manually fed to the automaton system as and when required for demonstration.
The scenario grid size is taken to be **(99 x 99)**, with the target in cell indexed at **(49,49)**. Now an imaginary circle of 40 units radius can be visualised with the target as its center. Five pedestrians are spaced successively

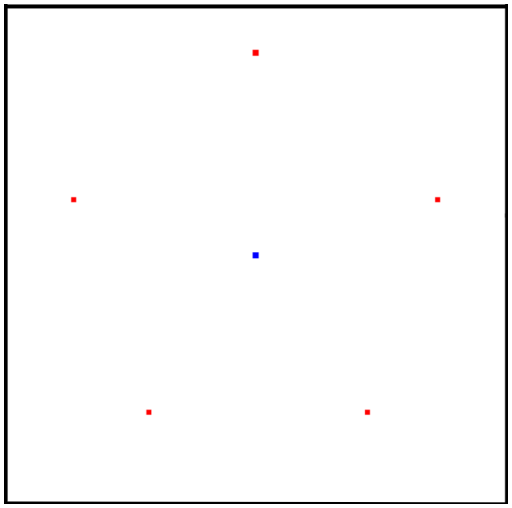


Figure 3: Five pedestrian setup

at the boundary of this circle, with a gap of 72° between them. Starting from 0° (right above the target) we have other 4 pedestrians placed at 72°, 144°, 216°and 288°.
We use basic trigonometry to estimate the coordinates for these 4 pedestrians, by splitting distances into their horizontal and vertical components and then placing them accordingly from the center target.
For example, the coordinates of the 1st pedestrian (leaving the one which is directly above the target) can be calculated as (49 + 40cos18°, 49 - 40sin18°).
Exact coordinates for the pedestrians can be found in the code files.

3.2 Absorbing nature of Target

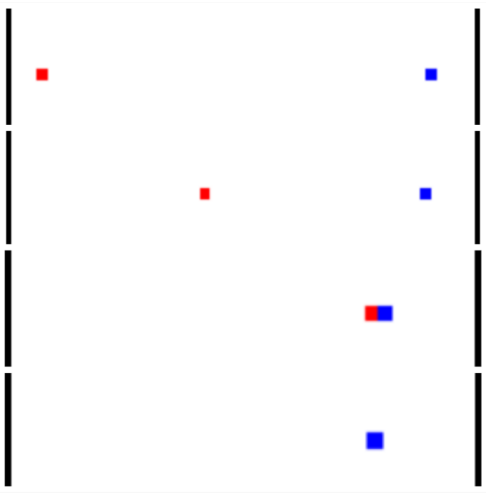


Figure 4: How the pedestrian(red) gets absorbed into the target(blue)

As stated earlier, this feature requires the pedestrian to "disappear" as soon as it lands on the target cell, during the simulation.
Every instance of the *Scenario* class, has a property named **pedestrians**, which is a list of pedestrians given by the user. These pedestrians themselves are instances of the *Pedestrian* class, having a **position** and **desired_speed** associated with them.
A pedestrian moves by comparing the distance_to_target value of all the 8 cells in its neighborhood and then jumps to the cell, having the least of it. So at any point, if the cell to which the pedestrian jumps, happens to

be the target cell, we simply remove the pedestrian from the scenario's **pedestrians** list.
Users need not worry about the "loss of pedestrian" from the scenario, as clicking the "Restart Simulation" button re-feeds the data of *scenario.json* file to the system, thus bringing the consumed pedestrian back into the setup.
See Fig. 4 for reference.

3.3 Configuring speeds of a pedestrian

In the previous question, a pedestrian made jumps to all of its 8 neighbor cells in the same time. This in turn resulted in different speeds as different "Euclidean distances" were covered in the same time interval.
The core idea lies in the fact, that neighbor cells which are diagonally placed to the pedestrian cell, are $\sqrt{2}$ times farther than the horizontally or vertically placed neighbor cells.

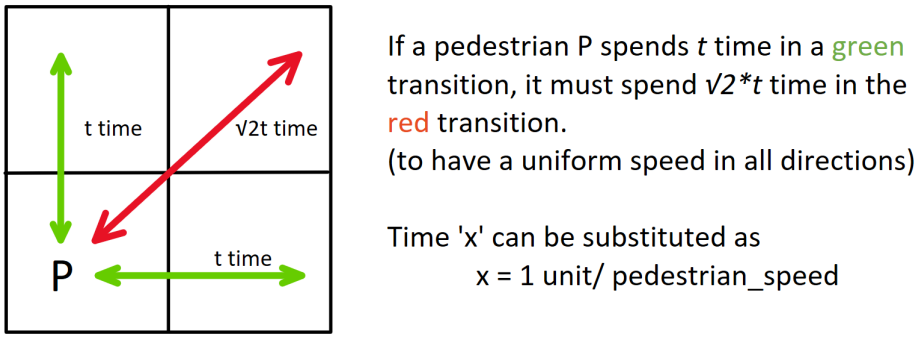


Figure 5: Explanation of speed adjustments

Hence a transition to the diagonal neighborhood must take $\sqrt{2}$ time longer than a transition to the vertical or horizontal neighborhood.

3.3.1 Implementation of new functionalities

1. Method: *new_automated_step* (MainGUI Class)

   An extended version of the previously implemented automated_step function. The function brings asynchronous programming into use and runs independent threads for all the pedestrians, where every pedestrian starts its journey, transitions to the most optimal neighbor, and checks whether it is a diagonal type neighbor or a horizontal-vertical type neighbor.
   If it is the diagonal neighbor, the function forces the corresponding thread to sleep for a time of $\sqrt{2}$ t and then make a jump to the respective cell, while in the other case thread sleeps for a duration of only t units (where t = 1/pedestrian_speed)
   That is for covering a longer distance, a pedestrian spends a longer time in making its transition than in the case where distance is shorter, so that the speeds can be equalized and made uniform.

2. Button: *Simulation involves speeds*

   A new toggle button by the name of "Simulation involves speeds" is introduced in the automaton UI. This button when clicked, implements *toggle_button* and *toggle_function* methods (described later in this section) and brings changes according to its state, set by the user.

3. Method: *toggle_button* (MainGUI Class)

   This function disables(enables) the "Step_Scenario" button in the automaton UI, when the check button is turned on(off). The "Step_Scenario" button allowed the user to move the pedestrians by 1 step at a time. This model worked only when all pedestrians made all their transitions (irrespective of the distance) in exactly the same 1-time step. But in the speed-adjusted version of the model, since the grid has discrete

cells, we cannot visualize a jump of 1 unit, when the jump is to be made to a diagonal cell. Hence we adopted a different approach by allowing a pedestrian to make a discrete jump of 1 unit to its horizontal-vertical neighborhood or $\sqrt{2}$ units to its diagonal neighborhood and spend an equivalent corresponding time accordingly.

Hence we cannot take time steps and have to go by a continuous simulation approach only. Thus the "Step_Scenario" button gets disabled as soon as the check button is checked on.

4. Method: *toggle_function* (MainGUI Class)

   This function replaces the "automated_step" function of the "Automated Step" button in UI with the "new_automated_step" function when check button is turned on. It reverts back to the former function when the check button is turned off.

3.3.2 Final Findings of Q3

SNo	Pedestrian Coordinates	Speed	Time taken to reach target
1	(49,9)	5	8.00s
2	(85,38)	5	8.11s
3	(71,80)	5	7.93s
4	(28,80)	5	8.02s
5	(13,38)	5	8.11s

Table 1: Final Observations

We can clearly see that all pedestrians roughly take the same amount of time (8 seconds) to traverse a "Euclidean distance" of 40 units when their speeds are set to 5.

Report on task 4, Obstacle avoidance

Task 4 is to implement different obstacle avoidance functions. We constructed two different scenarios, the "chicken test" and the "bottleneck test", and distributed 150 pedestrians randomly in a certain area, i.e. Figure 6. (Note: In the graphical interface, you must click on Task 4(DIJKSTRA)/Task 4(FMM)/Task 4(BASIC) in order to see the scenario. The three buttons represent the use of different algorithms.)

Figure 6: "Chicken Test" and "Bottleneck Test"

4.1 No Obstacle Avoidance

If no obstacle avoidance is implemented, pedestrians step on each other, because they will consider both other pedestrians and obstacles as blank cells, and step directly over the obstacles and go straight to the target, as shown in Figure 7.

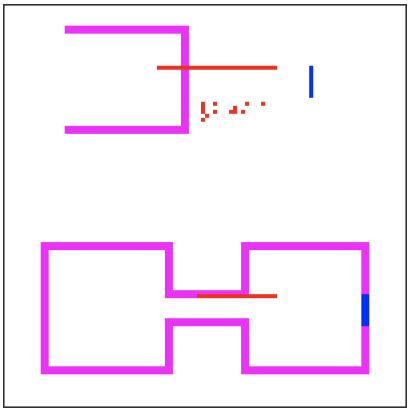
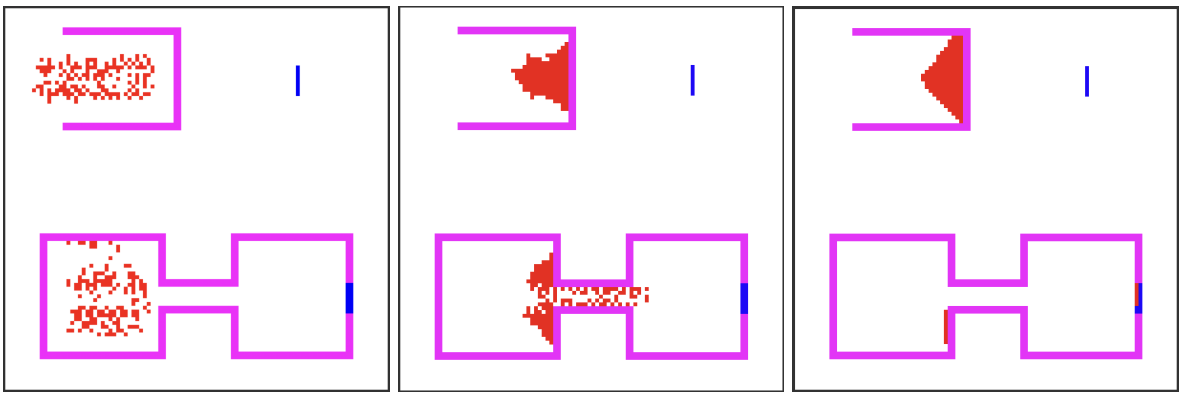


Figure 7: "Chicken Test" and "Bottleneck Test" without obstacle avoidance

4.2 Rudimentary Obstacle Avoidance

Rudimentary obstacle avoidance is implemented so that pedestrians do not step on each other and on obstacles. As shown in Figure 8, at the beginning of the simulation, all the pedestrians walk straight toward the target. For the "chicken test", the pedestrians stop when they hit the right wall because the basic algorithm only considers the direction of the nearest target. The "chicken test" failed. For the "bottleneck test", the pedestrians hit the rightmost wall first and then go down into the corridor. Most of the pedestrians pass the "bottleneck test", while a small number of pedestrians fail to reach the target. We can see that these pedestrians stop at the lower right wall of room 1 because this cell is the smallest Euclidean distance before entering the corridor.

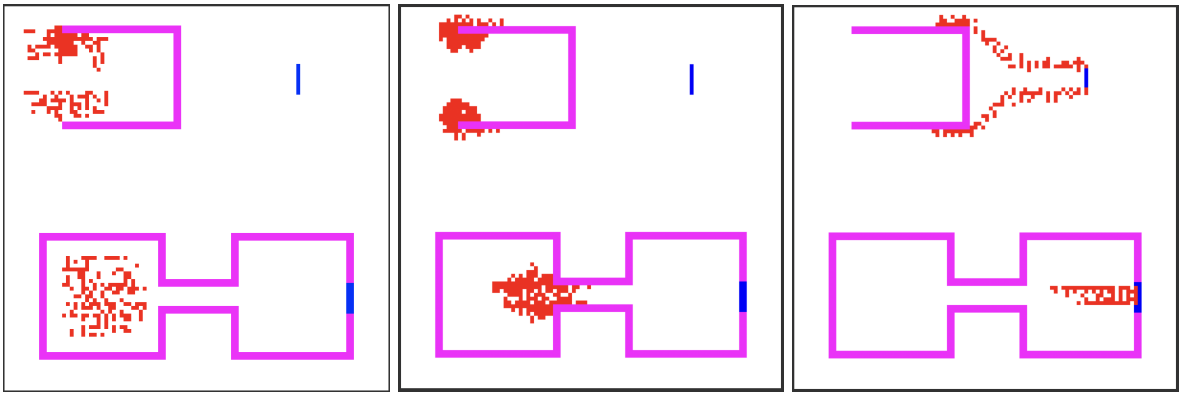


(a) It the beginning (b)In the middle (c) Near the end

Figure 8: "Chicken Test" and "Bottleneck Test" using basic algorithm

4.3 Dijkstra's Algorithm

Based on the problems encountered with the basic algorithm, we implemented dijkstra's algorithm.
From the process shown in Figure 9, we can see that the pedestrians passed the "chicken test" very well, the pedestrians bypassed the obstacles to reach the goal; for the "bottleneck test", the pedestrians did not walk to the rightmost wall first, but found the optimal path and did not move to unnecessary cells. Pedestrians can be seen moving through the corridor in an orderly fashion without crowding in room 1.



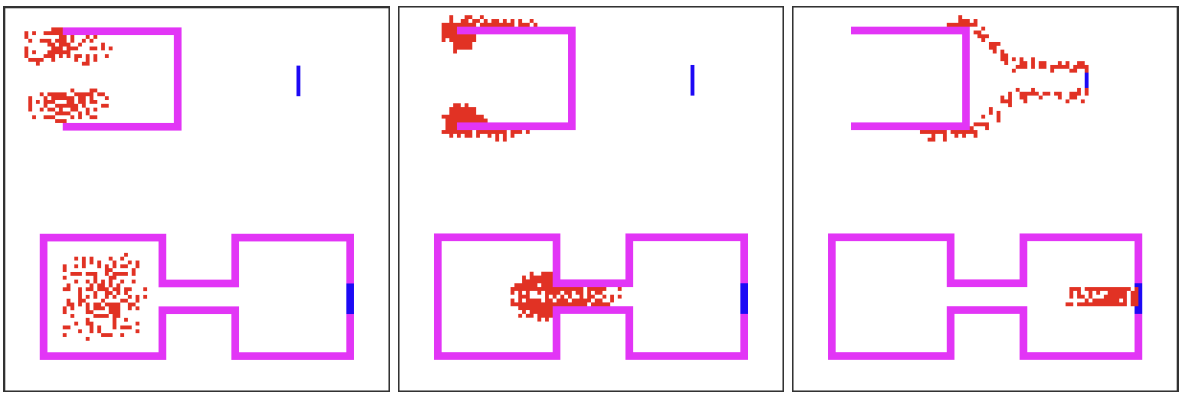
(a) It the beginning (b)In the middle (c) Near the end

Figure 9: "Chicken Test" and "Bottleneck Test" using dijkstra algorithm

Dijkstra's algorithm for finding the shortest distance from a specific target cell to all other cells in a grid or network. It operates in a stepwise manner, beginning with the target cells marked as closed and their distances initialized to zero. It then proceeds to explore neighboring cells, iteratively updating distances as it moves. The algorithm keeps track of which cells have been visited using a 'closed' array, and it maintains a 'distances' array to store the calculated distances. The process continues until either a specified limit is reached or there are no more available positions to explore.

4.4 Fast Marching Algorithm

The fast marching algorithm is primarily used for solving problems related to front propagation. It can find the shortest path while considering obstacles and constraints.
The results of the Fast Marching Algorithm are very similar to those of the Dijkstra Algorithm, passing both the "Chicken Test" and the "Bottleneck Test". (Figure 10)
First, we initialize two NumPy arrays, mask and distances, where mask represents obstacle locations in the grid, and distances are used to store distances from target cells. For each target cell found, it sets the distance of that cell in the distances array to 0, indicating that it's already at the target. If a cell is an obstacle, it sets the corresponding entry in the mask array to 1, indicating that it's obstructed. we use the scikit-fmm library (skfmm) to calculate the shortest distances.



(a) It the beginning (b)In the middle (c) Near the end

Figure 10: "Chicken Test" and "Bottleneck Test" using fast marching algorithm

Report on task 5, Tests

This task requires to test the implementations with four different scenarios from the Rimea guidelines.

5.1 [TEST1:] RiMEA scenario 1

5.1.1 Test Description

A pedestrian in a 2m wide and 40m long corridor with a defined walking speed will cover the distance in the correct time period 26-34 seconds.

Parameter	Values
Corridor length[m]	40
Corridor width[m]	2
No. of pedestrians	100
Speed	1.33 with 5 % deviation

Table 2: Test 1 Specifications

5.1.2 Simulation Model

Prior to this task, constant speeds were considered for the pedestrians. Pedestrians speed are sampled uniformly with mean speed 1.33 m/sec and standard deviation of 5percent. Pre-Movement time is ignored for our test simulation as we are just trying to prove pedestrians can walk with a certain speed, not the realization of evacuation situation. Also, the required travel time period (to be achieved) for the pedestrians is adjusted by the Pre-Movement time(1 sec). So, required travel time range for this test becomes 25-33 seconds instead of 26-34 seconds. No obstacles were placed in the test. The corridor and walking speeds were modelled according to the specifications (see table 2). One pedestrian was generated per simulation run, eliminating the side effects of other pedestrians.

5.1.3 Results

Figure. 11 presents this test setup where a pedestrian of defined walking speed is moving towards the end of corridor. Figure 12 confirms that in each of the 100 simulation runs, travel time of the pedestrians remains within the limit of its prescribed travel time 25-33 sec.

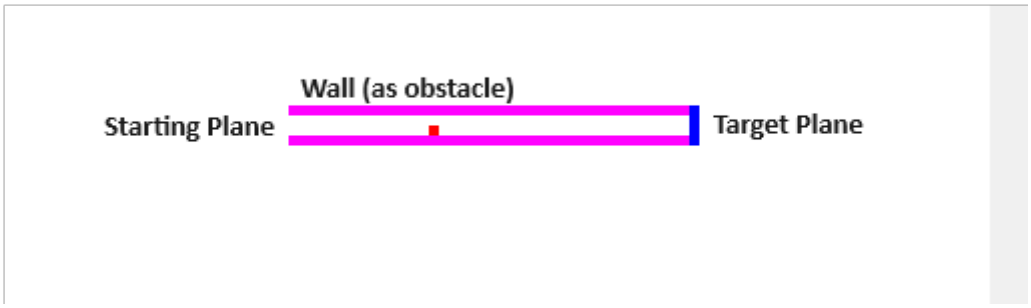


Figure 11: Screenshot of pedestrian moving in a straight corridor of 2m width and 40m long

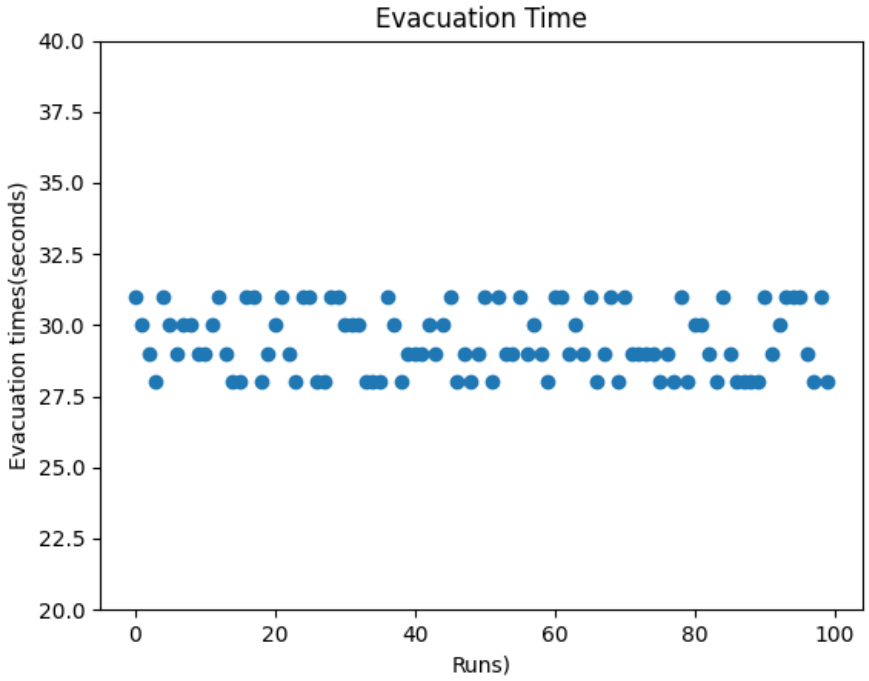


Figure 12: Evacuation Times (seconds) vs Simulation runs

5.2 [TEST2:] RiMEA scenario 4 (fundamental diagram)

.

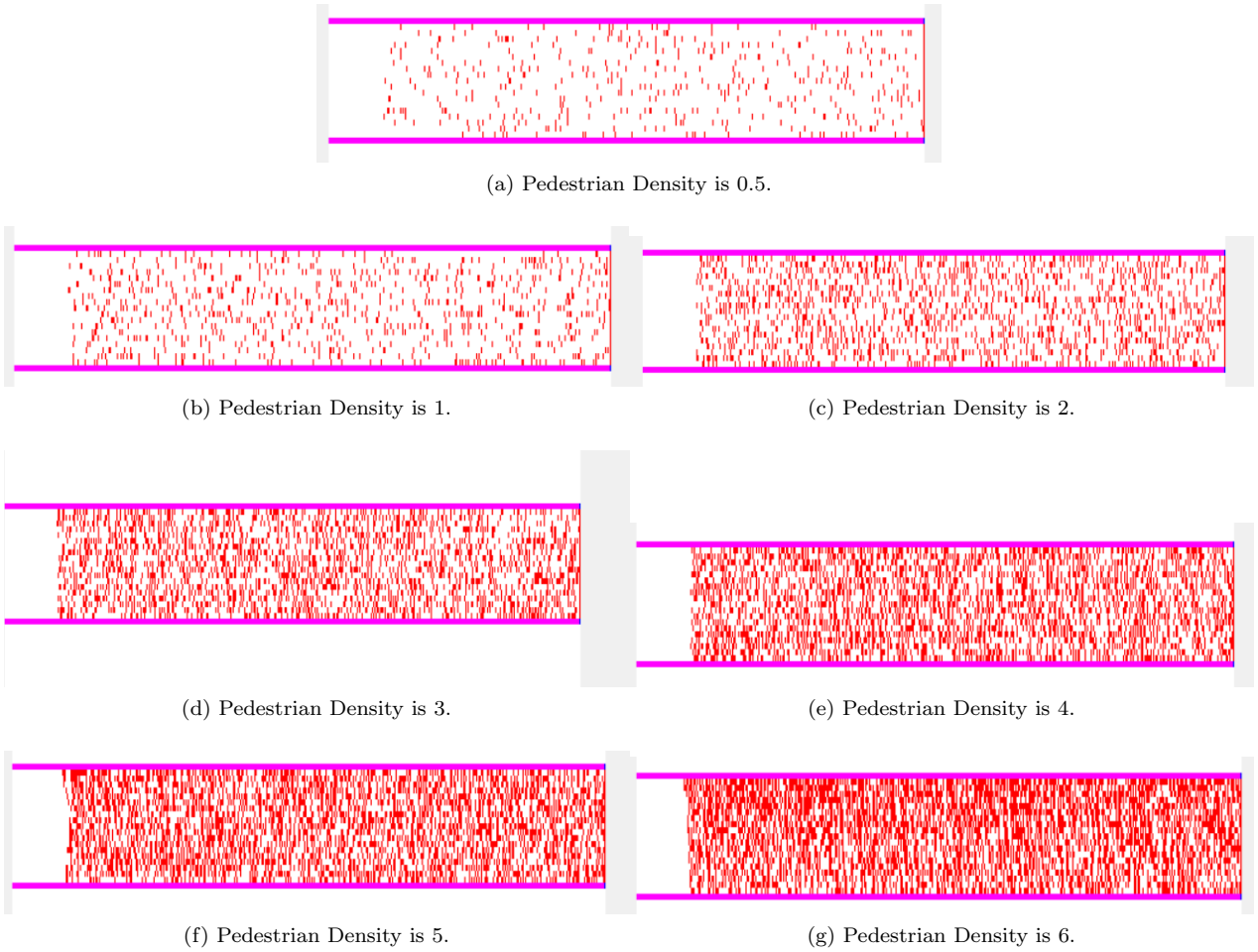
5.2.1 Problem Statement and Our Choices

The purpose of Test 2 is to confirm that, even if the starting speed of each pedestrian was set to the same value, crowd density affects the actual speed of pedestrians walking. As specified in the test description, we used three measuring sites. Following the simulation, a basic figure is produced that illustrates the relationship between average speed and density as well as flow rate value and density.

We did this test on a 1000x100 cell area. We did 10 + 60 simulation second tests. In every test, there were different pedestrian densities. These were 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. So, there was 800, 1600, 3200, 4800, 6400, 8000, and 9600 pedestrian accordingly in a 10000-cell area. In addition, in every density pedestrians are uniform and totally random which means the pedestrian initialization will be changed when you operate every time.

We did not significantly decrease the number of pedestrians because our system was sufficient to replicate the high number of pedestrians.

5.2.2 Test Results and Usage



(a) Pedestrian Density is 0.5.

(b) Pedestrian Density is 1.

(c) Pedestrian Density is 2.

(d) Pedestrian Density is 3.

(e) Pedestrian Density is 4.

(f) Pedestrian Density is 5.

(g) Pedestrian Density is 6.

Figure 13: Initial Pedestrians

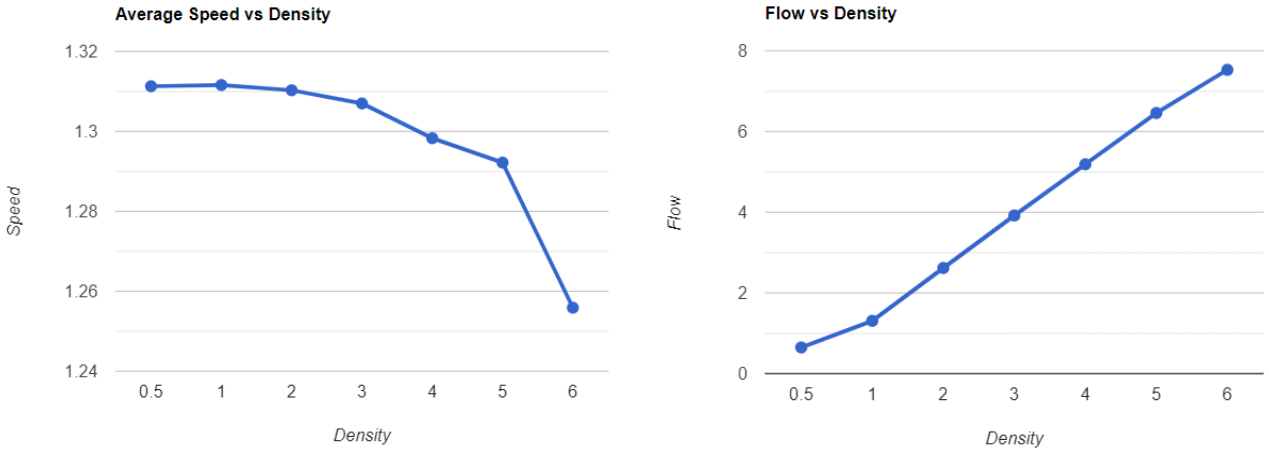
These images show the distributions of pedestrians after seventy simulation seconds. You can imagine here how density will affect the average speed of all pedestrians. It will decrease or will be the same.

Please adhere to the README guidelines for test 2 in task 5 in order to replicate test findings. After that, you can do simulations with different densities. After every simulation, you will get average speed and flow results. Then, you need to re-open the application to do another simulation with another density of pedestrians.

Density	Average Speed	Flow
0.5	1.3113	0.65
1	1.3116	1.31
2	1.3103	2.62
3	1.3070	3.92
4	1.2983	5.19
5	1.2922	6.46
6	1.2559	7.53

Table 3: Findings of tests on different densities

The results of experiments on various densities show that average speed is not much altered. However, it is very different in density six since there are people walking down the corridor almost filling up the whole area. On the other hand, computational flow is always increasing with the density.



(a) Average Speed vs Density

(b) Computational Flow vs Density

Figure 14: Test Results

The test results show that speed decreases as density increases because of the high density, and pedestrians begin to interfere much with one another. However, as density increases, flow also increases. The primary cause is that, speed does not alter much in response to density change.

Test 2 successful

5.3 [TEST3:] RiMEA scenario 6 (movement around a corner)

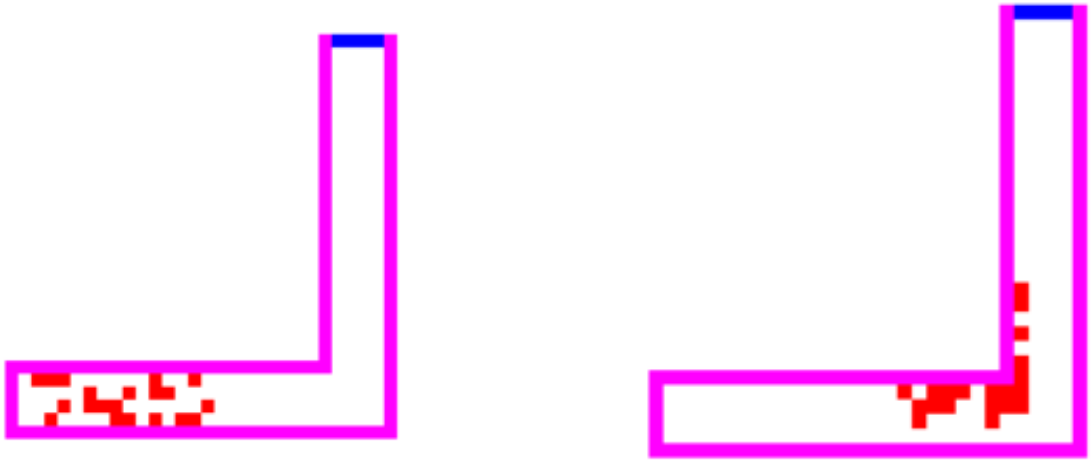
.

5.3.1 Problem Statement and Our Choices

.
The objective is to confirm that twenty people approaching a corner will be able to turn around without hitting any walls. In this test, we took every pedestrian as 40cm x 40cm. According to this, we built the obstacles. So, our obstacles' long sides became two hundred and fifty cells which was ten meters in the description. In this test, we preferred the Fast Marching Algorithm. So, every pedestrian can achieve the target in the stated conditions in the most effective way we use.

5.3.2 Test Results and Usage

.



(a) Initialization (b) Half-way to the target

Figure 15: Simulation Images

As you can see from these images, it started with uniformly distributed pedestrians. Then, pedestrians find the shortest way to reach the target without passing through the walls or creating a straight line. You can do your own test while using the basic application initialization stated in the README file. In every test, you could get different pedestrian initialization because our initialization is both uniform and random. Basically, you could do many tests while just clicking on Test 3, then Automated Step or Step Simulation.

Test 3 successful

5.4 [TEST4:] RiMEA scenario

5.4.1 Test Description

The task is to verify that the distribution of walking speeds according to the age in the simulation is consistent with the distribution in the Figure 16 Weidmann graph.

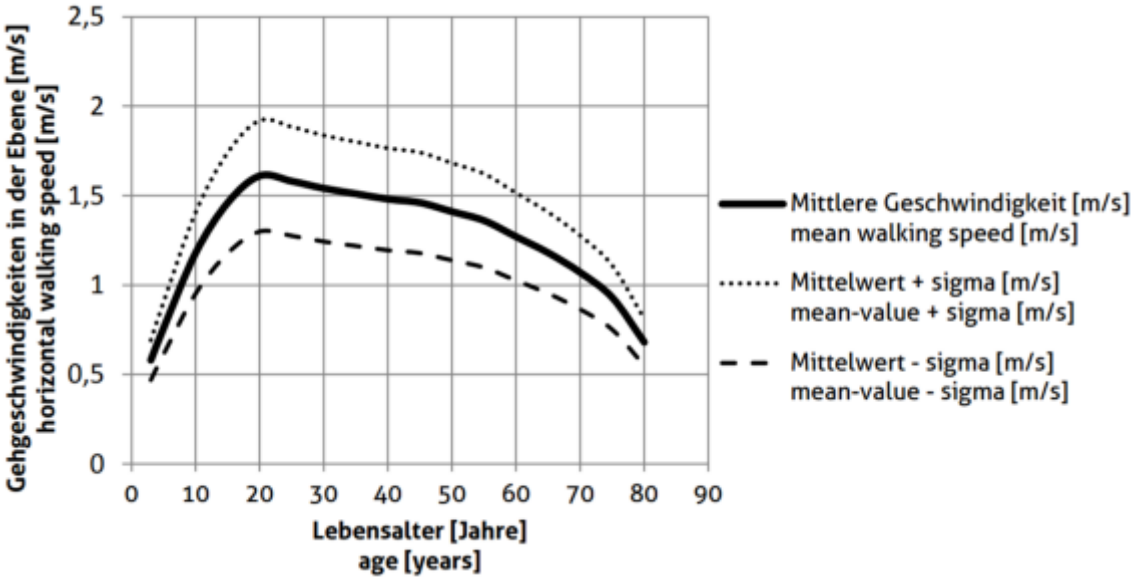


Figure 16: Horizontal walking speeds against pedestrian age

5.4.2 Simulation Model

The corridor (70m × 20m) and walking speeds are modelled according to the specifications (see Figure 17). For this test, 10 pedestrians each of 20, 30, 40, 50, 60 years of age have been considered. Their mean speeds and accounted standard deviations are provided in the table 3. The standard deviation considered is very narrow to produce more mean-centring data in order to map Weidmann graph better with smaller sample size of 50 pedestrians. Pedestrians are generated after every 3 step-simulation to mitigate congestion. If all 50 pedestrians move towards a single target at the same time, a bottleneck can occur. Probably, this would change their actual speed which is undesirable for this test.

Age Group	No of Pedestrians	Mean Speed[m/s]	Speed Deviation[m/s]
20	10	1.62	0.04
30	10	1.54	0.04
40	10	1.48	0.04
50	10	1.40	0.04
60	10	1.27	0.04

Table 4: Given Walking speed of each age group[Weidmann]

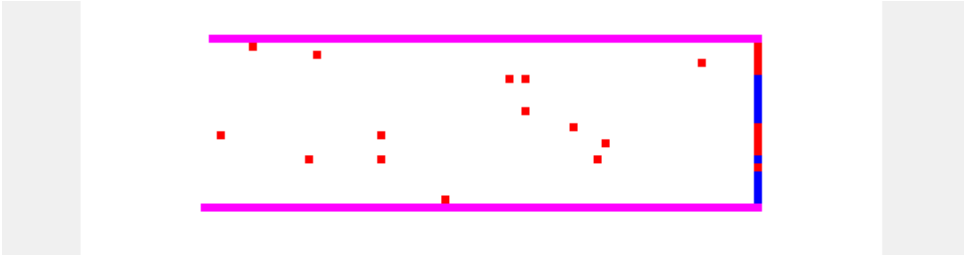


Figure 17: Screenshot of pedestrians of different age groups walking towards target plane

5.4.3 Results

Walking speeds of pedestrians are calculated by dividing the total distance travelled (over every time-step) by the total travel time. Figure 18 demonstrates that mean walking speed is decreasing starting from age 20 till

60 which is compatible with Weidmann documentation. [Note: Many simulations are overlapping, hence less scatter dots in the graph.]

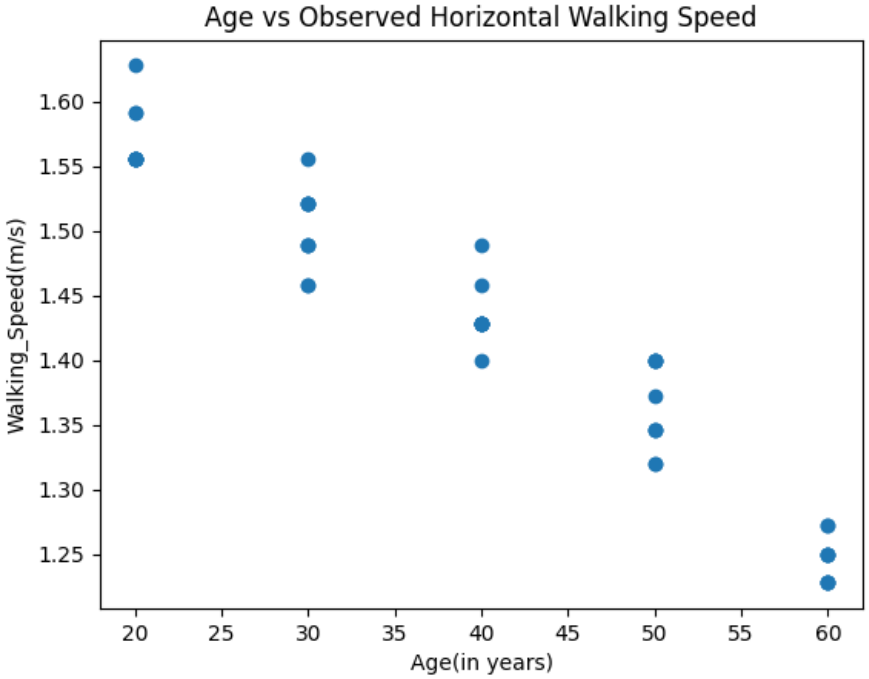


Figure 18: Walking Speeds of Pedestrians (demonstrates Weidmann graph)