Report for exercise 2 from group A

Tasks addressed: 5
Authors: Hirmay Sandesara (03807348)
Yikun Hao (03768321)
Yusuf Alptigin Gün (03796825)
Fatih Özlügedik (03744764)
Subodh Pokhrel (03796731)
Last compiled: 2025–05–27

The work on tasks was divided in the following way:

Hirmay Sandesara (03807348)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	0%
	Task 5	15%
Yikun Hao (03768321)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	0%
	Task 5	70%
Yusuf Alptigin Gün (03796825)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	100%
	Task 5	0%
Fatih Özlügedik (03744764)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	0%
	Task 5	0%
Subodh Pokhrel (03796731)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	0%
	Task 5	15%

1 Task 1: Setting up the Vadere environment

1.1 Overview of the Vadere Environment

Vadere is a lightweight open-source simulation framework that offers pre-implemented versions of the most widely spread models for crowd simulation[4].

Vadere is implemented in Java and available for GNU-/Linux, MacOS and Windows. It reads in simulation parameters, like topography, an agent's (pedestrian's) radius and other parameters, from a human-readable JSON text file. The simulation results — usually x and y coordinates for each pedestrian and time step — are also written to text files. Because of this, text files can be written manually as input (see Section 3.2) and the output results can be processed by other programs (in Python, Matlab or other).[4]

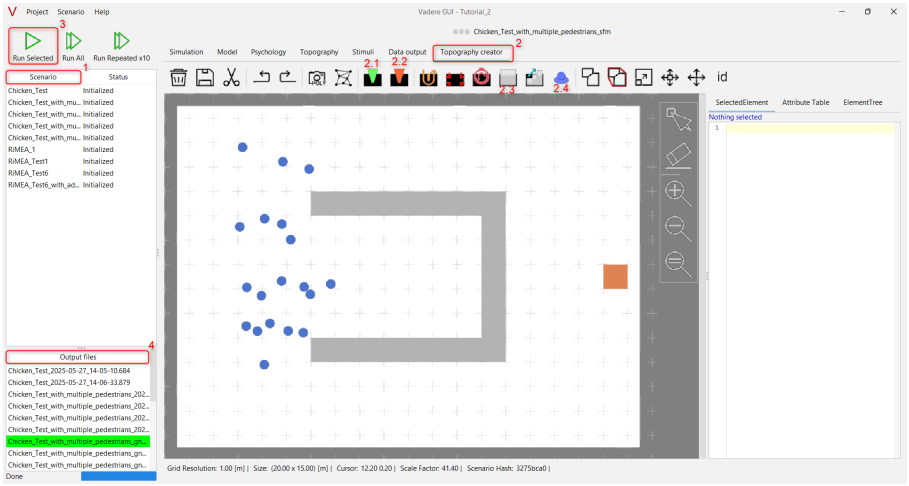


Figure 1: A snapshot of Vadere GUI

Figure 1 shows a snapshot of Vadere GUI with the most frequently used attributes and sections of the program highlighted with red boxes and numerically labeled.

1. Scenario A list of scenarios in the current project directory. Each scenario is a canvas with sources (2.1), targets (2.2), pedestrians (2.4), obstacles (2.3), and more.
2. Topology creator This is in the GUI playground to create scenarios.
3. Run selected Allows the selected scenarios to run
4. Output files The results of the simulation run with parameters chosen by the user to be saved

Additionally, Vadere provides a console version called *vadere-console.jar* that can be used to automate simulation runs; for example, run multiple instances or run on a remote computer. An implementation of this is seen in Section 3.1. For more information on Vadere, please see *Vadere: An open-source simulation framework to promote interdisciplinary understanding* by Kleinmeier et al. 2019[4].

1.2 Installation

1.2.1 Windows

Following the official documentation from Vadere, the software was set up to run in Windows by first installing OracleJDK (done for a simpler installation in comparison to OpenJDK) and Apache Maven 3.9. Then the downloaded and extracted release of Vadere could be opened in the GUI by double-clicking on vadere-gui.

1.2.2 macOS with ARM architecture

The Linux file didn't work on our system, despite numerous tries to debug. However, following the Video Tutorials: Opening Vadere in the IDE, cloning the repository, and building the JAVA code, we worked on this system. It did come with a few errors like 'Failed to load class "org.slf4j.impl.StaticLoggerBinder".' and OpenCL acceleration was not available, but simulations were easy to create, and everything worked for the required part.

1.3 Creation of RiMEA scenarios and Chicken Test

The 'Topology creator' tab of Vadere allows a user to add pedestrians, obstacles, sources, and targets, among others, to a canvas to simulate different scenarios of pedestrian movement. Using these features, three scenarios were created:

1. **RiMEA Test 1:** A pedestrian is placed in a 2-meter wide corridor with the target 40 meters away from it. In Figure 2, the pedestrian is the blue circle where as the target is the rectangular orange block.

Figure 2: RiMEA Test 1

In order to create the setup seen in Figure 2, a 44 m × 9 m canvas was set in Vadere. The pedestrian was placed at coordinated (1,4.5) and the target (bottom left position) at (41,4) with the size 1m × 1m. The speed of the pedestrian was set at 1.33 m/s. The model for movement was set to Optimal Steps Model (OSM).

2. **RiMEA Test 6:** The setup of RiMEA test 6 requires a horizontal corridor of 12 meters in length and 2 meters in width. This corridor had a sharp left turn at the end into another 12-meter-long, 2-meter-wide corridor. Out of the 12 meters of length of these two paths, 2 meters are shared in an overlapping zone where the sharp left turn is made by pedestrians. It is described by the test that 20 pedestrians should be uniformly distributed along 6 meters of the horizontal area (marked by the green rectangular source element seen in Figure 3). The pedestrians who spawn from the source move to the orange block at the top of the vertical corridor- the target.

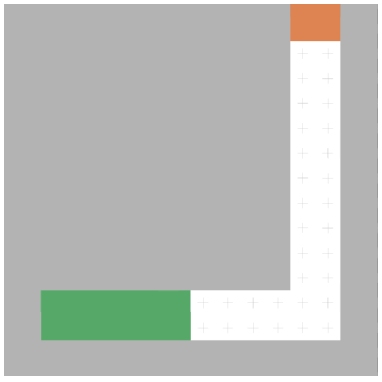
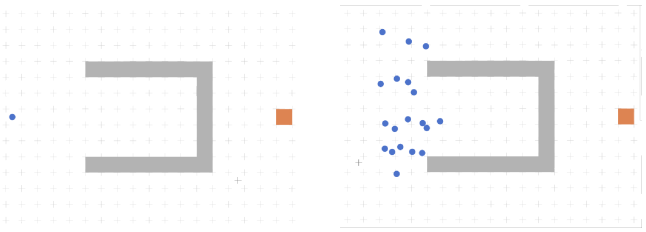


Figure 3: RiMEA Test 6

The setup of Figure 3 is created in Vadere on a 15 m × 15 m canvas with the areas other than the horizontal and vertical paths being obstacles (grey areas in the setup).

3. **Chicken Test:** A pedestrian is placed in a 20 m × 15 m canvas with a target placed at the same horizontal level. Between them, however, is a U-shaped obstacle with its opening facing the pedestrian. If the pedestrian moves straight towards the target, it is bound to go into the obstacle

and get stuck inside. The purpose of this test is to see how the pedestrian manages to reach the target, bypassing the obstacle. As in Figure 2, the blue circle in Figure 4 represents a pedestrian, whereas the orange rectangle is the target and the grey area is the obstacle. A second test was also created by adding more pedestrians to the setup in Figure 4a. This setup is seen in 4b.



(a) Chicken Test with one pedestrian

(b) Chicken Test with multiple pedestrians

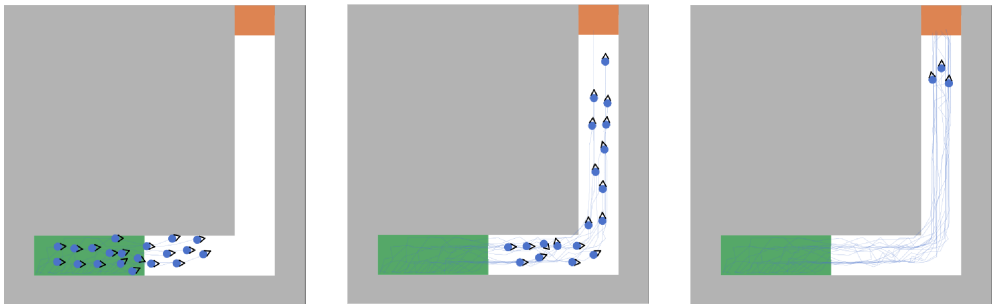
Figure 4: Chicken Tests

1.4 Simulation of RiMEA scenarios and Chicken Test

When the scenarios seen in Figures 2, 3 and 4 are run in Vadere using the Optimal Steps Model (OSM), the pedestrians move to the respective targets in each scenario as seen in Figures 5, 6 and 8 respectively. The blue line marks the path taken by the pedestrians.

Figure 5: RiMEA Test 1 simulation result

In Figure 5, the pedestrian moves in a straight line to the target in a total of 29.8 seconds. This falls in the expected range for RiMEA Test 1: 29-34 seconds and is no different than the simulation using cellular automata. The pre-movement time is ignored in this setup as the pedestrians are already assigned the target before running the simulation



(a) Early stage

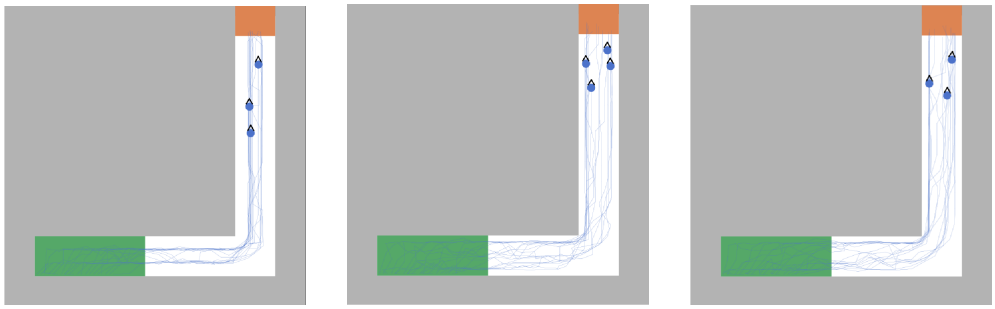
(b) Middle stage

(c) Ending stage

Figure 6: RiMEA Test 6 early at different stages into the simulation

For RiMEA Test 6, the result seen in Figure 6 shows the pedestrians emerging from the source, moving along the horizontal path, making a left turn into the vertical corridor, and moving towards the target. The simulation used the in-built model parameters of OS,M and it can be seen in the result that two pedestrians pass through the obstacles while making the turn. This is similar to the simulation using cellular automata, during which some pedestrians also passed through the obstacles. Pedestrians passing through obstacles is impossible and hints at an error in the parameters. And indeed changing some of the default parameters results in a more plausible situation where no pedestrians pass

through obstacles. Figure 7 shows different simulation results with parameters that are changed to avoid pedestrians moving through obstacles. In each scenario, only one parameter is changed with everything else kept to the default values.



(a) obstPotentialHeight = 20 (default = 8)

(b) obstPotentialWidth = 0.8 (default = 0.4)

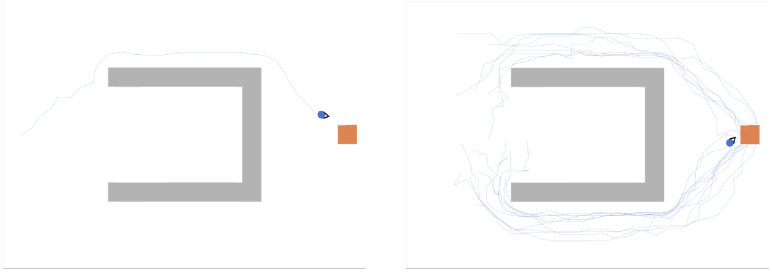
(c) stepCircleResolution = 12 (default = 4)

Figure 7: RiMEA Test 6 with parameters different to standard OSM in Vadere

1. **obstPotentialHeight:** This parameter defines the maximum strength of repulsion an obstacle exerts on a pedestrian. Increasing it means pedestrians are less likely to go towards and skim through the edge (or even go through).
2. **obstPotentialWidth:** This parameter defines the repulsion range. A lower range means that the repulsion is close to the obstacle, but is abrupt, in contrast to a higher range where there is repulsion at a greater distance, but softer.
3. **stepCircleResolution:** This parameter defines the directions into which a pedestrian considers their next path. Increasing this means a greater possibility of taking a different route than the one that goes through an obstacle.

The effect of these parameters and their differences are clearly visible in Figure 7. With *obstPotentialHeight = 20*, the pedestrians stay substantially away from the edge as the repulsion from it is high. Lowering *obstPotentialWidth* means the pedestrians can go closer to the walls due to the reduced range of repulsion, but don't go through the walls due to a high repulsion density at those close distances. Increasing *stepCircleResolution* has also prevented pedestrians from passing through the walls, as they have options that lead them away from going through the barriers.

When simulating the Chicken Test with Cellular Automata, it was seen that the pedestrian could not reach the target without obstacle avoidance. However, with Dijkstra's algorithm implemented, the pedestrian could navigate to the target by avoiding being stuck in the obstacle. The simulation on OSM is similar to a pedestrian going around the obstacle to reach the target.



(a) Chicken test with 1 pedestrian simulation result

(b) Chicken Test with multiple pedestrians simulation result

Figure 8: Chicken Test simulation result

It is seen in Figure 8b how the pedestrians that are already in the opening of the U-shaped obstacle are able to get out and go around the obstacle to reach the target. This is because the distance to target calculation is geodesic and therefore takes the obstacle into consideration, unlike an Euclidean distance measurement [4].

2 Simulation with a different model

The task involved running 3 different scenarios from the previous task:

- RiMEA scenario 1,
- RiMEA scenario 6,
- The Chicken test.

The scenarios were run using two different models: the Social Force Model (SFM), and the Gradient Navigation Model (GNM). These two models were compared to the Optimal Steps Model (OSM), which was used in the previous task. The description of these models and comparisons with OSM are provided in Subsections 2.1 and 2.2, in the order mentioned before. Lastly, we have combined the joint analysis of all three in Subsection 2.3. We'll start with a brief description of the OSM model.

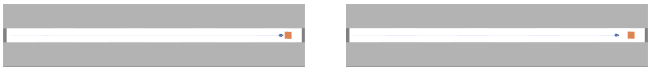
The Optimal Steps Model (OSM) is a model where pedestrians move towards their target by selecting steps that are 'optimal', specifically in their local neighbourhood [1]. The central idea revolves around pedestrians trying to minimise their objective function, which could be time or distance to a specified target. The target is defined by the 'floor field', i.e., a scalar function. This is typically represented by the time or distance to the final state or destination from any point in the defined environment. Hence, the pedestrians essentially evaluate the possible steps in their local neighbourhood that minimise the function described before. This results in a guiding mechanism where they can efficiently move towards their goal while implicitly handling obstacles, as they are represented with higher values for the function described [4].

2.1 Model: Social Force Model (SFM)

SFM describes the dynamics governing the pedestrians with the assumption that the motion of pedestrians is given by a combination of 'Social Forces'. These forces represent the external influences and internal motivations acting on an individual. SFM treats these forces in a way that is similar to physical forces in Newtonian mechanics. The sum of these forces results in an acceleration that changes the pedestrian's velocity and position over time. It aims to reproduce the emergent collective phenomena in pedestrian crowds, like jamming, oscillations at bottlenecks, and lane formation [3].

Observations based on the three scenarios are presented below:

- RiMEA scenario 1:

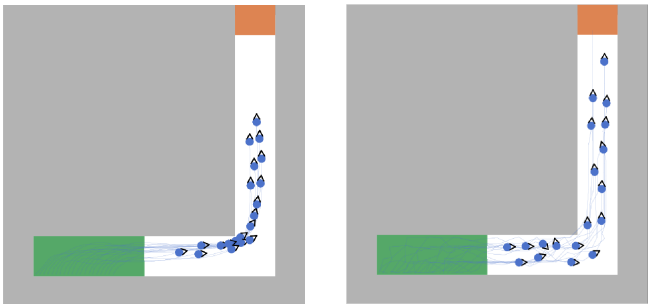


(a) SFM (b) OSM

Figure 9: Comparison between OSM and SFM for scenario 1

  - From Figure 9 it can be seen that our single pedestrian didn't follow a perfectly straight line, moving marginally downwards, then upwards before reaching the specified target.
  - OSM contrasts this, as the pedestrian followed the straight path without any deviation.
  - OSM was slightly faster, but time can be considered identical as they behave similarly.
  - The slightly peculiar behavior could be explained cause of there being a single pedestrian.

- RiMEA scenario 6:

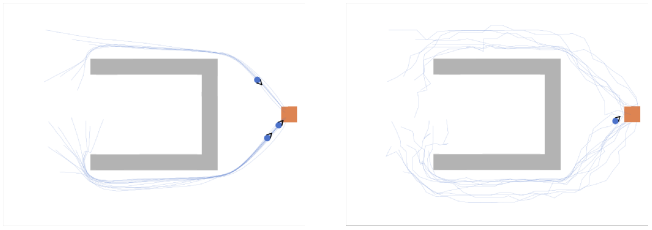


(a) SFM (b) OSM

Figure 10: Comparison between OSM and SFM for scenario 6

- Most of the pedestrians followed the same uniform path as the first pedestrian to reach the specified target, without passing through walls.
- Pedestrians seemed more organised, proceeding uniformly and naturally, compared to OSM method.
- From Figure 10 it can be seen that, pedestrians under FSM at times stopped or collided when it was not required.
- OSM was considerably faster than SFM.

- Chicken Test:



(a) SFM (b) OSM

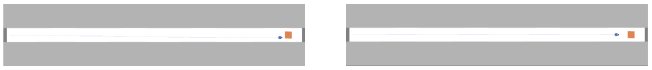
Figure 11: Comparison between OSM and SFM for the Chicken Test

- This particular behaviour was similar to the previous scenario, but different symmetrical paths were taken based on the initial state.
- In OSM, some collisions occurred, which resulted in SFM performing slightly better.
- From Figure 11 it can be seen that pedestrians in SFM seemed to know their path from the beginning, seeming more organised, direct, and intrinsically behaving; whereas in OSM, some of them collided with one another, and paths were not smooth.

2.2 Model: Gradient Navigation Model (GNM)

GNM is a microscopic pedestrian simulation model where the movements of such pedestrians are governed by gradients of a potential field. The 'floor field', as described earlier in the OSM, represents the desirability of different positions in the environment for a pedestrian trying to reach their target. Essentially, following the steepest descent of this field, guiding them towards their destination while intrinsically avoiding obstacles if the function is defined appropriately. GNM can incorporate various factors like individual walking speeds and can be modified to have pedestrian-pedestrian interactions. GNM is designed to be computationally efficient, especially by discretising space into a grid and pre-computing the static parts of the 'floor field' [2].

- RiMEA scenario 1:

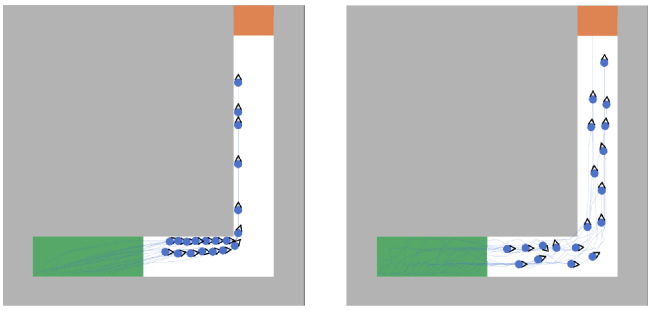


(a) GNM (b) OSM

Figure 12: Comparison between GNM and OSM for scenario 1

  - From Figure 12 it can be seen that, pedestrian didn't follow a straight line. It moved downwards until the specified target was reached.
  - Travel times were quite similar between GNM and OSM.
  - The peculiar behaviour where we saw a pedestrian turning upright before reaching the target.

- RiMEA scenario 6:

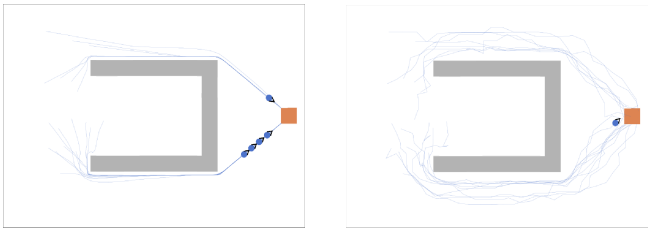


(a) GNM (b) OSM

Figure 13: Comparison between GNM and OSM for scenario 6

  - Almost all pedestrians ended up taking the same uniform path as the successful pedestrian candidate who reached the target first.
  - All of them appeared much more organised, compared to OSM.
  - From Figure 13 it can be seen that, if any obstructions occurred between pedestrians, it resulted in them waiting, instead of taking different paths.
  - This results in much wider time difference between the other models, as seen from Table 1.

- Chicken Test:



(a) GNM (b) OSM

Figure 14: Comparison between GNM and OSM for the Chicken Test

  - From Figure 14, it can be seen that pedestrians were able to turn around the obstacle, taking uniform paths. Unlike OSM, where collisions and non-uniform behaviour were seen.
  - Similar to the previous scenario, but here, different yet symmetrical paths were taken, based on the initial state of each pedestrian.
  - Peculiarly, some of them chose to avoid obstructing pedestrians rather than waiting.
  - Because of narrow passages, and pedestrians selecting different parts in OSM, blockages happen which doesn't happen with GNM.

2.3 Model: Combined Analysis

After making observations from 3 different scenarios with the three different models, the models quite significantly affect the path distribution of the pedestrians. Furthermore, depending on the scenario, some might have a considerable time advantage, and in some, all may be similar. The strictness of the model determined whether they avoided other pedestrians or not. Based on this definition, in our case, OSM was the least strict, whereas GNM was the most.

From Table 1, it can also be seen that time comparisons are made between the different models for our given scenarios.

Table 1: Time Comparisons between different models for our scenarios

Scenario	Model	time (in s)
Scenario 1	OSM	29.8
	SFM	30.4
	GNM	30.4
Scenario 6	OSM	27.23
	SFM	29.2
	GNM	36.4
Chicken Test	OSM	27.89
	SFM	29.3
	GNM	30.2

3 Using the console interface from Vadere

3.1 Running a scenario file through Vadere console

Moving on from GUI to console, *Test6.scenario* could be run and the output stored by the command:

```
java -jar vadere-console.jar scenario-run
--scenario-file "/path/to/scenario/file/file.scenario"
--output-dir="/path/to/output/directory"
```

Each file in the output directories from the GUI and console was compared using the command `comp .\output_directory_GUI\* .\output_directory_console5\* .` This command compares two files (in this case, all files from each directory) and prints the differences in the MS Powershell window [5]. The result of the comparison is seen in Figure 15.

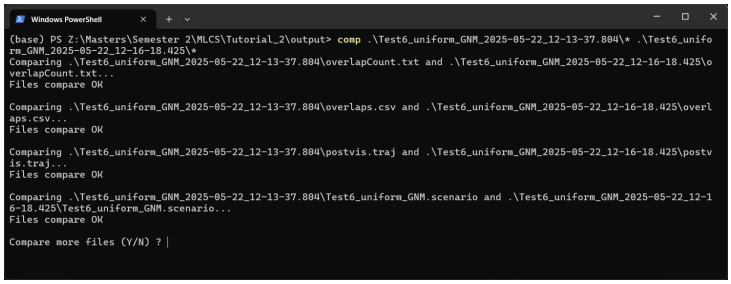


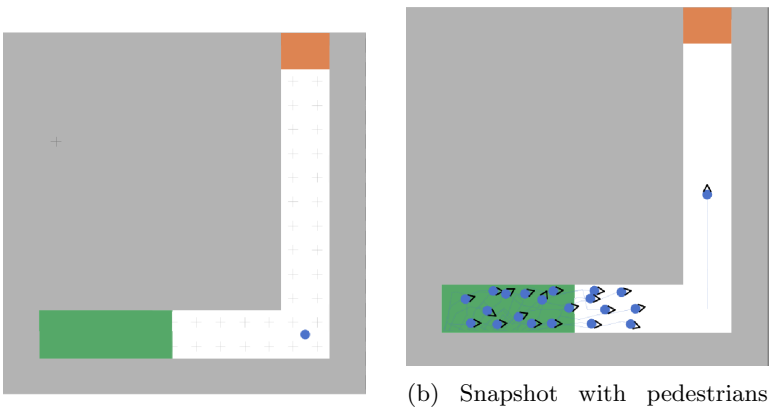
Figure 15: Comparison of output files from GUI and console

It is seen that the files generated as output in both methods have no difference whatsoever.

3.2 Adding pedestrians through code

In sections 1 and 2, scenarios were created in the Topology creator section of the GUI where different elements (such as pedestrians, targets, obstacles, etc) could be added to a canvas. Each canvas (a scenario) was stored as a JSON file. Editing or manually writing these scenarios without using GUI is possible as it merely is writing into a .json file in a structured manner.

The pedestrians in Vadere are defined as dynamicElements `scenario.topography.dynamicElements` in the JSON files. In the scenario file for RiMEA Test 6, a pedestrian is added using the Python program *add_pedestrian.py*. When the program is run, it reads the JSON file *RiMEA_Test_6* and adds a pedestrian with the same attributes as if it were added using Vadere GUI. This pedestrian is also assigned the same target as other pedestrians in the scenario and behaves the same. The setup and simulation of the new scenario can be seen in Figure 16.



(a) Setup with added pedestrian

(b) Snapshot with pedestrians moving to target

Figure 16: Setup and simulation of RiMEA Test 6 setup with an added pedestrian

The Python program is hard-coded for now to add only a single pedestrian with a single target. However, but a simple loop and a unique location, id, target id and size (among other variables) can be used as input to create multiple pedestrians at once.

With the addition of the pedestrian, the program also runs the simulation through the console and saves the output in a predetermined location. This is done by the lines:

```
vadere_cmd = [
    'java',
    '-jar',
    vadere_path,
    'scenario-run',
    '--scenario-file',
    scenario_path,
    '--output-dir',
    output_dir,
]

try:
    result = subprocess.run(vadere_cmd, check=True, capture_output=True, text=True)
    print("Vadere simulation completed successfully")
    print(result.stdout)
    ...
```

where `vadere\_path` is the path to `vadere-console`, `scenario\_path` is the path to the scenario file, and `output\_dir` is the path to the directory to store the output files.

This generates an output file with the simulation results. Comparing the results from the simulation run through the console and the same simulation also through the GUI outputs the exact same results as seen in Figure 17.

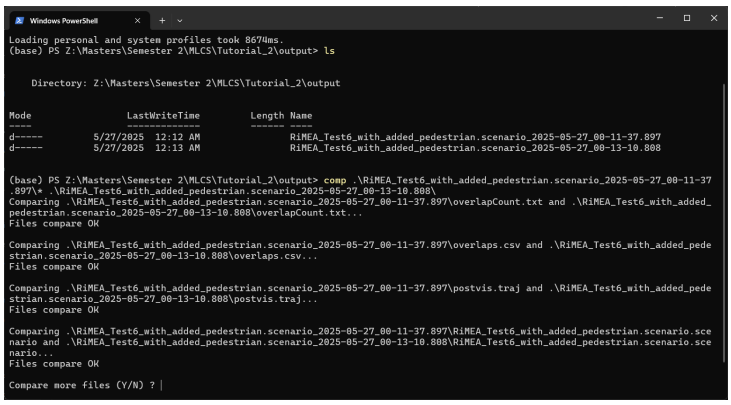


Figure 17: Comparison of output files from simulation run through console and GUI

4 Integrating a new model

4.1 SIR model in Vadere

The SIR model belongs to the package `org.vadere.simulator.models.groups.sir` and contains `SIRGroupModel`, `SIRGroup`, and `SIRType`. The `SIRType` contains an enumeration of the pedestrian states, infected, susceptible, recovered (Task 5), which are then managed through the other two classes. `SIRGroup` class acts as a container for group data and contains many helper functions that are crucial when managing the simulation via `SIRGroupModel`. Retrieving pedestrian IDs, group members' information, deleting and adding members to groups are some of the functionalities available. `SIRGroupModel` implements the core logic of the model, and it controls the simulation logic and interface with the update function and dynamic listeners. It initializes group structures, assigns infection states to incoming pedestrians, computes infection probabilities, and performs state transitions accordingly.

4.2 Analyzing

To analyze the infection spreading among the pedestrians, we need information about which pedestrians belong to which group at each time step of the simulation. As suggested in the sheet, we use the **FootStepGroupIDProcessor** of Vadere to get all this information into a CSV file. This CSV file precisely contains the information mentioned before, with each row corresponding to a pedestrian with its group type and the simulation time. This data will be used to create the graphs in part **4.5** when analyzing different scenarios.

4.3 Coloring Scheme

For the coloring scheme of the pedestrians, to not over-engineer the process, we use option b given in the sheet. In its original version, the `getGroupColor` assigns random colors to pedestrians through a color map. For compactness in visualization, we change this function and create a coloring scheme. By getting the group ID of a pedestrian using the `ped.getGroupIds().getFirst()` function, the function then returns a switch statement on the group ID, which assigns the following colors to each group:

- **Group 0, Red:** Infected pedestrian
- **Group 1, Blue:** Suspectible pedestrian
- **Group 2, Green:** Recovered pedestrian, which will be used in task 5.
- **Default Group, Gray:** Mostly for sanity check in the code

Now, we can group pedestrians by ID during simulation and visualize the process better in test scenarios.

4.4 Efficiency Improvements

To improve the efficiency of the simulation, using the `LinkedCellsGrid` class, we've to make some improvements in the `update` function of the `SIRGroupModel` class. Similar to the older version, it puts all pedestrians in a container using the topography and returns without any updates if there are no pedestrians. The difference starts after this step. Now, the function gets the bounds of the grid using the topography and turns the 2D bounds into a `VRectangle`. Together with the maximum infection distance of the simulation, this information is then passed to a constructor of the `LinkedCellsGrid`, creating a pseudo grid where it will perform neighboring checks. The maximum infection distance is used for the "cell size" parameter of the constructor as a heuristic, as the value here doesn't affect the underlying process since the grid is merely used to check neighbors, and the distance/visualization of the grid isn't important. After the grid creation, each pedestrian is added to the grid through a loop.
With this setup, now we loop through each pedestrian again, checking for the pedestrian group ID in each iteration. If the ID deems the pedestrian to not be susceptible, the function lets the loop continue to the next iteration since we won't be trying to possibly infect already infected (or recovered) pedestrians. Then, if the pedestrian is susceptible, the grid gets the neighbors of the pedestrian using the `getObjects` function. This way, with the addition of `LinkedCellsGrid`, we can now only check the neighbors of the pedestrian. Looping through these neighbors, the inner loop stays pretty similar to the original function. It checks if the neighbor is the pedestrian itself and continues to the next iteration if so. Then

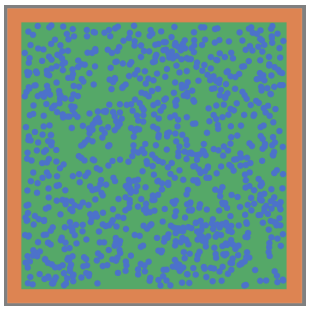
checks if the neighbor is infected (has the possibility to infect the initial pedestrian), and gets a random double value if so. If this value is smaller than the infection rate of the simulation, the initial susceptible pedestrian is removed from the simulation, and a new infected one is added in its place. Then the loop breaks since we don't need to do more checks.
What we've achieved through this structure is the improvement of the runtime. Before, two for loops inside one another were used, with all entries being run through in each loop, say **N** entries. With our new approach, even though there are still two loops, the inner loop is significantly smaller, say **M** entries. This significantly lowers the complexity of the function. Also, even though we've added another loop before the double loop, this won't have any effect on the overall complexity. Below are the complexities for both approaches.

- **Original Version:** $O(N) = O(N * N) = O(N^2)$
- **Revised Version:** $O(N) = O(N * M + N) \sim O(N)$ with $N >> M$

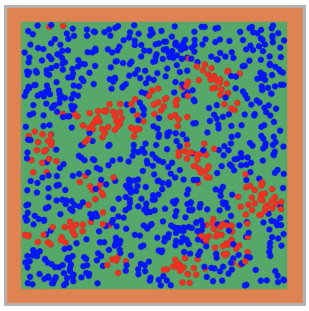
4.5 Test Scenarios and Results

4.5.1 Static 1000 Pedestrians:

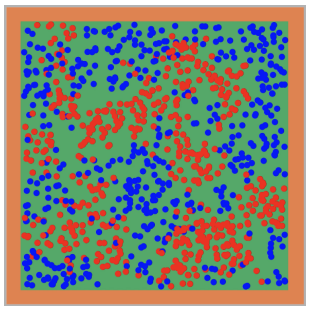
The steps given in the exercise are followed to create the scenario with 1000 pedestrians in a static position. Below can be seen some approximate images from the various points in the simulation. Also, the Dash/Plotly App visualization can be seen in figure 19. It takes about 25 seconds for half the population to get infected from the given seed. As another note, since this version of the infection mechanism only checks for neighbors when dealing with infections, it takes some time for the last infections to occur. Particularly, pedestrians in the lower right corner take quite a long time to get infected with a low infection rate (at this point 0.01), which is why the %100 infected timestamp is high.



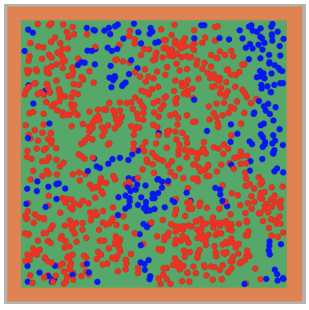
(a) Initial Distribution of 1000 Pedestrians, Timestamp: 0s



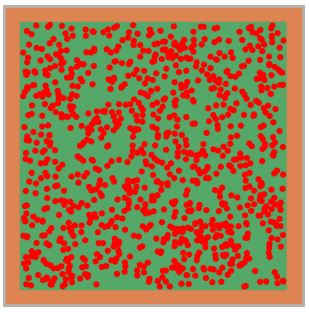
(b) Around %25 Pedestrians Infected, Timestamp: $\sim 15s$



(c) Around %50 Pedestrians Infected, Timestamp: $\sim 25s$



(d) Around %75 Pedestrians Infected, Timestamp: $\sim 38s$



(e) %100 Pedestrians Infected, Timestamp: $\sim 189s$

Figure 18: Static 1000 Pedestrian Test Simulation Steps

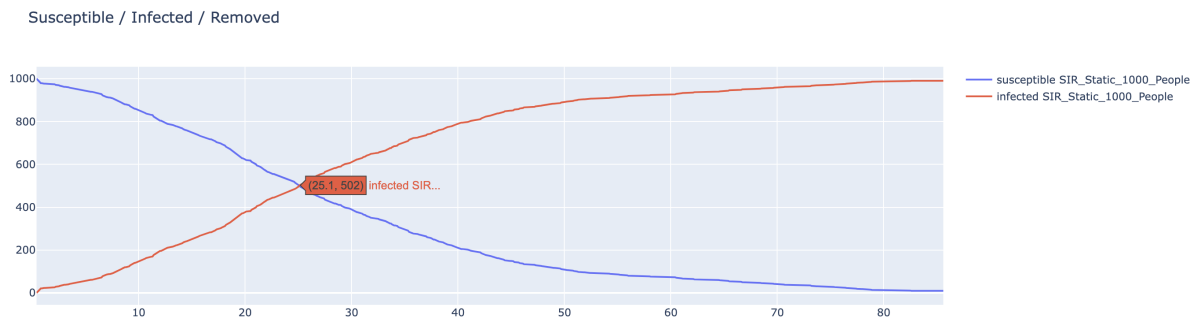
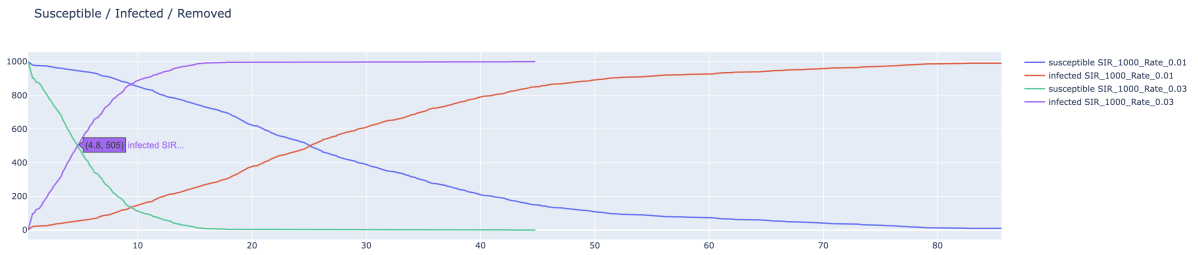


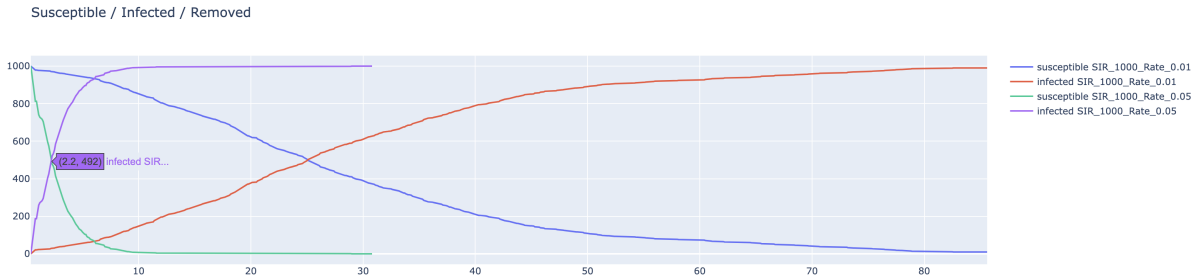
Figure 19: Visualization of the Infection Spread Among Pedestrians

4.5.2 Increased Infection Rate:

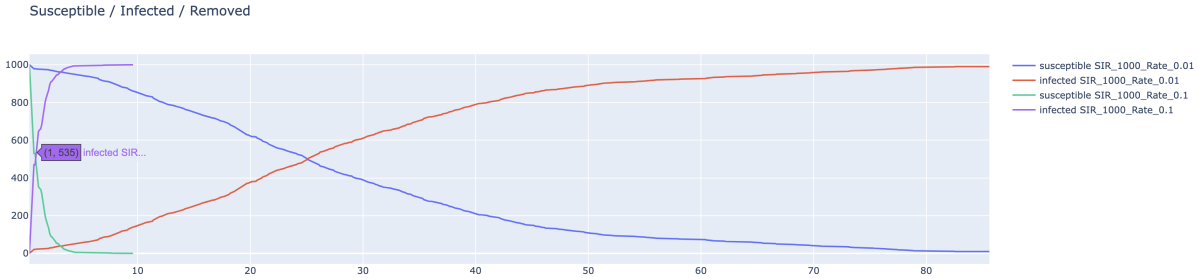
Below are some graph comparisons to see how much effect the increase in infection rate has on its spread. When the infection rate increases to 0.03, it takes about ~ 4.8 seconds for half of the population to get infected. If we keep increasing the infection rate, it gets to a point where it's nearly instant for the entire population to get infected; with half the population getting infected taking ~ 2.2 and ~ 1 seconds for infection rates of 0.05 and 0.1.



(a) Infection Rate 0.01 and 0.03 Comparison



(b) Infection Rate 0.01 and 0.05 Comparison

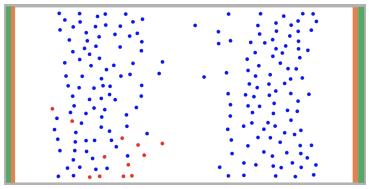


(c) Infection Rate 0.01 and 0.1 Comparison

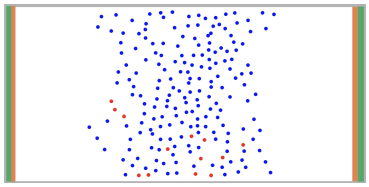
Figure 20: Infection Spread Differences Depending on Infection Rate

4.5.3 Corridor Scenario:

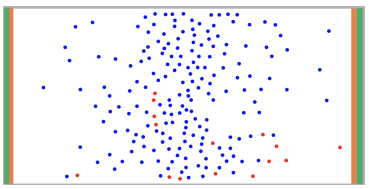
To create the corridor scenario, we place two sources and targets next to each other on each edge of the scenario, with each source having its target on the other side of the edge. Below can be seen some steps of the run of the scenario with an infection rate of 0.01. This scenario isn't so dense, and there is movement among pedestrians. Since the spread of infection is done through neighbors and the infection rate isn't so high, the infection doesn't spread quite effectively. We observe that only a total of 15 pedestrians get infected throughout the simulation, with the first 10 already starting infected.



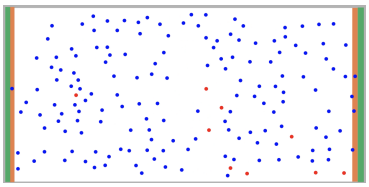
(a) Timestamp $\sim 9s$ of the Corridor Scenario



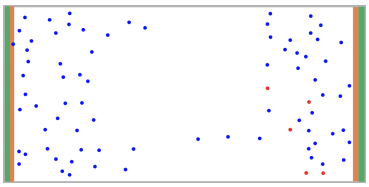
(b) Timestamp $\sim 14s$ of the Corridor Scenario



(c) Timestamp $\sim 21s$ of the Corridor Scenario



(d) Timestamp $\sim 29s$ of the Corridor Scenario



(e) Timestamp $\sim 36s$ of the Corridor Scenario

Figure 21: Corridor Scenario Simulation Steps

4.6 Decoupling Infection Rate and Time Step

In its current state, the update function uses the infection probability of the simulation in each update step, regardless of the simulation's time. To tackle this discretization issue, we need to incorporate the time of the simulation into the probability of the pedestrians getting infected. We start by adding the `lastSimTime` as a private variable to the `SIRGroupModel` class. This variable will track the simulation's time, giving us a way to check how much time has passed since the last update step. By making small changes in the update function, we can incorporate this variable. `lastSimTime` takes an initial value 0, thus it can be deemed assigned by the first simulation step. In each step, the update function now calculates the delta using `lastSimTime` and `simTimeInSec` (an input variable to the update function). Then, assigns `simTimeInSec` to `lastSimTime`. With the delta available, the function then gets the infection rate attribute from the simulation and calculates the probability of a pedestrian getting infected with the formula

$$\texttt{infectionProbThisStep} = 1.0 - (1.0 - \texttt{infectionRatePerSecond})^{\texttt{deltaTimeInSec}}$$

The `infectionProbThisStep` is then switched with the vanilla infection rate in the update function when checking for neighbors, and the decoupling of the simulation is achieved.

4.7 Possible Extensions

- **Expand Health Status:** The model approaches recovery of pedestrians with a very basic view. This can be extended to cover more realistic scenarios by adding a lot more groups/subgroups. Details about temporarily/fully recovered, (a)symptomatic, lightly/severely infected groups, and many more can be created to have more influence over the realism of the scenario.
- **Variation in Infection Risk:** The infection in the model spreads through neighbors with a uniform effect, which isn't the case in reality. This can be modelled better by taking into account things like pedestrian density, proximity, hotspots, environmental conditions, and, in general, having a varied infection risk depending on the scenario.

- **Interventions:** Incorporating real-world interventions into the model, such as vaccinations, mask usage, lockdowns, etc., can be crucial to incorporate more settings and realism into the model. These can affect various variables in the scenario, such as movement, behavior, and transmission.
- **Time-dependent recovery rate:** What is used as recovery rate is a fixed value. Meanwhile, in realities the recovery rate could be modeled as a function related to time. To better reflect real-world dynamics, the constant recovery rate r in the original model can be replaced by a time-dependent function r(t). This modification allows the model to capture changes in recovery behavior over time, such as medical intervention effects or evolving disease characteristics.
- **Post recovery mobility pattern:** In reality, the movement pattern might differ between groups. Taking susceptible group and recovered group as an example, the recovered group might reduce activity amount because of residual fatigue, and are more conscience about maintaining distance from others, especially in scenarios like supermarkets. This leads to adaptation of pedestrian interaction logic we implemented before. This point only makes sense when considering a re-infect possibility, which means under assumption that a recovered person might get re-infected. That is a further and more advanced SIR model to be build and evaluated in the future.

5 Analysis and visualization of results

5.1 Recovered State in `SIRGroupModel`

The recovered state of the SIR model is one of the pedestrian states in `SIRType`, which is implemented in Task 5. Adding recovered state yields a three-state model that now mirrors the classical SIR formulation. The recovered state represents the final stage in the SIR model, which is a unidirectional and irreversible state machine. Over time, individuals in the infected state gradually transition into the recovered state, which is absorbing, as no further state changes occur once this state is reached. The recovered state is added in `org/vadere/simulator/models/groups/sir/SIRType` as `ID_REMOVED` and is assigned to group ID 2. The transition logic is implemented in `SIRGroupModel`.

5.2 Recovery Dynamics

Within the context of this task, the SIR model functions as a sub-model applied by the OSM (Optimal Steps Model) to simulate infection dynamics among pedestrians. The basic logic is shown as 22. The increased rate of recovery is defined as $\frac{dR}{dt} = rI$. The first derivative of the recovered population with respect to time is always positive, which indicates that the recovered population increases monotonically until the number of infected individuals reaches zero.

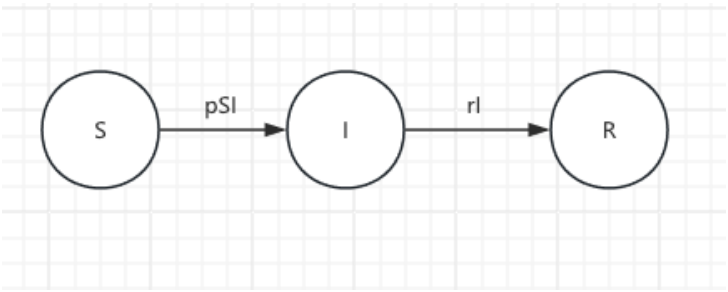


Figure 22: Basic Logic of SIR model

To implement recovery logic in `SIRGroupModel`, the recovery possibility of each step shall be calculated. We firstly get a pre-defined recovery rate in `AttributesSIRG` through `getRecoveryRate`. Then, as shown in part 4.6, the possibility of not recovery per second is `1.0 - RecoveryRatePerSecond`, and the possibility of non-recovery per time step is calculated via $1.0 - RecoveryRatePerSecond^{\Delta t}$, with Δt representing time step in seconds. As a final step, the recovery rate per time step is calculated via

$$\texttt{recoveryProbThisStep} = 1.0 - (1.0 - \texttt{RecoveryRatePerSecond})^{\Delta t}$$

Next step, the logic is fulfilled via `update` function in `SIRGroupModel`. In each time step for each pedestrian, check if the state is `infected`. If true, then generate a random number. When the recovery probability calculated from the last step is larger than this random number, the pedestrian is considered recovered in this step. As a result, the pedestrian is first removed from the current group using `elementRemoved(p)` and then reassigned to the removed group. The use of `continue` ensures that once a pedestrian recovers during a simulation step, they are excluded from any further infection checks. This effectively avoids redundant state evaluations.

5.3 Visualization of Recovered State

To add the recovered state visualization, we've to follow a similar procedure done in code for other SIR groups. The recovered state variable is added as a function input with its default value of 2 to the function `file_df_to_count_df`. Then, in the loop where the IDs for the simulation step are checked, we assign a new `current_state` as infected. This state is checked against the new state to see if it's different. If the new state is different and recovered, this means we've to make a pedestrian recover in the visualization. This is done by decreasing the count of infected and increasing the count of recovered by 1. This is all the necessary change to the `file_df_to_count_df` function. After this, we need to change the `create_folder_data_scatter` function a bit as well, since the data changed in `file_df_to_count_df` also needs to be visualized. By creating another scatter in the same way as the other groups, with just the IDs used in the creation being different, and returning this scatter with the other, the visualization includes the recovered state as well.

5.4 Tests with Recovered State

5.4.1 Static 1000 Pedestrians

The visualization of susceptible, infected, and recovered pedestrians through the simulation can be seen in 19. Since the simulation now contains a recovery state, it never reaches the full %100 infected state. Also, since the infection rate isn't so high, not all people get infected either, even though the infection and recovery rates are the same. Also, the pedestrians make a full recovery at the end, due to the way we define the recovered state.
In 24, we observe a mix of susceptible (blue), infected (red), and recovered (green) pedestrians. 24a shows the time when the first infected pedestrian got recovered, as the appearance of first green dot. 24b shows when recovered population equals to susceptible population at ~ 150. Although the infection spreads across the crowd, a large portion of the population remains uninfected due to the low infection rate, and many pedestrians successfully recover over time.

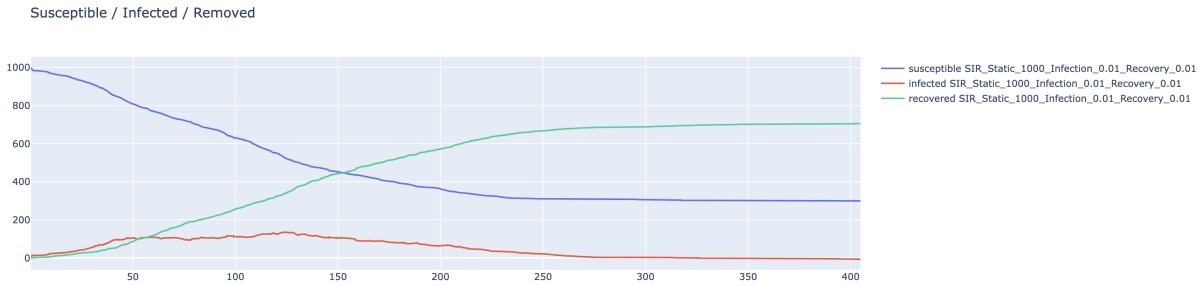
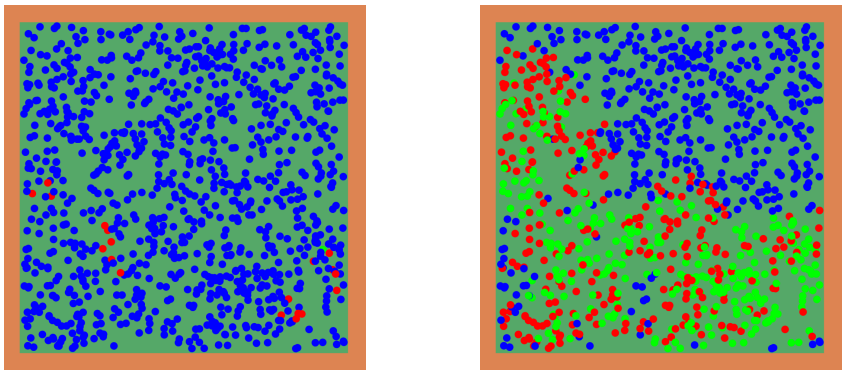


Figure 23: Visualization of Static 1000 Pedestrian with Infection Rate 0.01 and Recovery Rate 0.01



(a) Simulation time in secs: ~ 10 (b) Simulation time in secs: ~ 150

Figure 24: Infection Visualization with Infection Rate 0.01 and Recovery Rate 0.01

5.4.2 Experimenting with Infection and Recovery Rates

The purpose of this experiment is to investigate how different combinations of infection and recovery rates affect the spread of an disease within a static pedestrian population. By varying the infection rate and the recovery rate, we aim to observe changes in the dynamics of the SIR model. In this study, we continue to use the model of 1000 static pedestrians. For each scenario, we adjust the values of infection and recovery rates and visualize the settings through Vadere.

- **Visualization with Infection Rate 0.03 and Recovery Rate 0.03:**

  In this scenario we have both a higher infection rate as well as a higher recovery rate. As a result the susceptible population drops acutely as shown in 26a, and infected population reaches the peak of $\sim$ 60% of the overall population. As the recovery rate is also higher, we observe in 26b that almost all pedestrians are marked as recovered at $\sim 100s$ of the simulation. This variation indicates an epidemic model which is more infective but also more curable, the result is an acute and short epidemic.

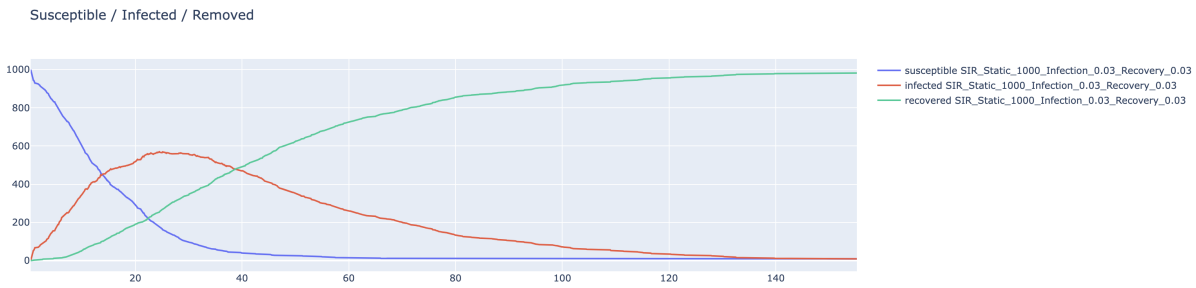
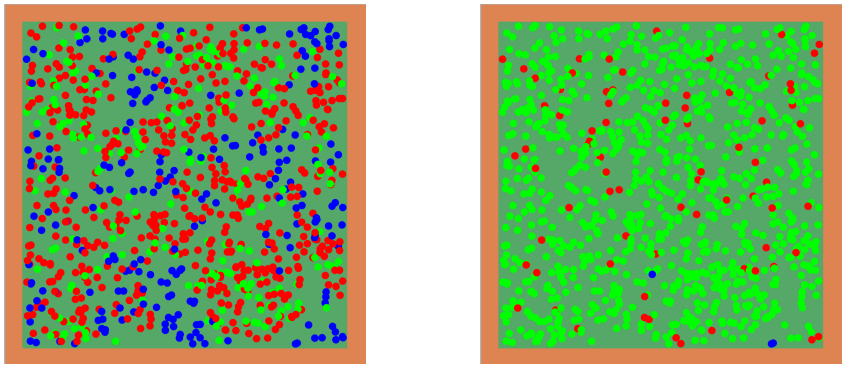


Figure 25: Visualization of Static 1000 Pedestrian with Infection Rate 0.03 and Recovery Rate 0.03



(a) Simulation time in secs: ~ 20 (b) Simulation time in secs: ~ 100

Figure 26: Infection Visualization with Infection Rate 0.03 and Recovery Rate 0.03

- **Visualization with Infection Rate 0.01 and Recovery Rate 0.03:**

  In this scenario we increase only the recovery rate while keeping the infection rate as in 5.4.1. Therefore, the infection rate is relatively low while the recovery rate is comparatively high. As a result, the infection spreads very slowly, and only a portion of the population becomes infected at any point in time, as shown in 28a. Eventually at $\sim 300s$, more than 30% of the pedestrian population remains uninfected (28b). This variation indicates an epidemic model which is less influential and highly recoverable, the result is a minor epidemic which is more controllable.

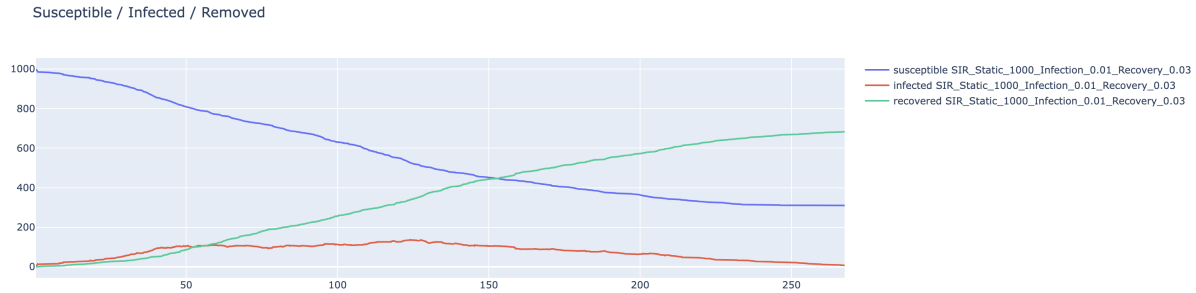
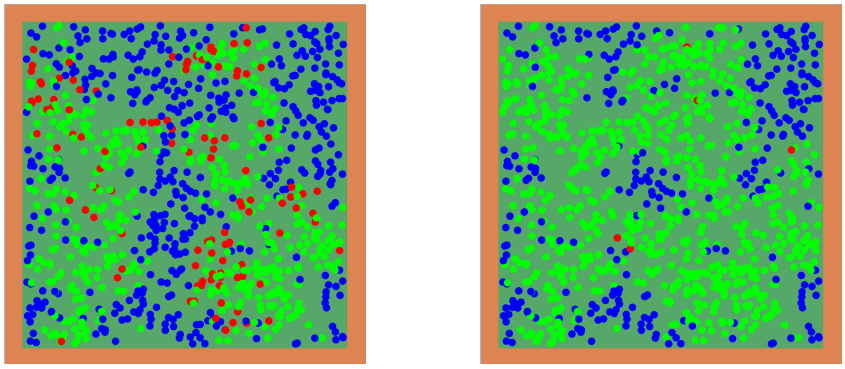


Figure 27: Visualization of Static 1000 Pedestrian with Infection Rate 0.01 and Recovery Rate 0.03



(a) Simulation time in secs: ~ 150 (b) Simulation time in secs: ~ 300

Figure 28: Infection Visualization with Infection Rate 0.01 and Recovery Rate 0.03

- **Visualization with Infection Rate 0.01 and Recovery Rate 0.1:**

  In this scenario, we have a very high recovery rate while keeping the infection rate as in 5.4.1. Therefore, the infection rate is very low in comparison to the recovery rate. As shown in 30a, a few pedestrians are recovered soon after the breakout of infection. And as in 30b, eventually less than 10% of the pedestrian population is infected. This variation indicates an epidemic model with minimal impact, and the result is a subtle disease.

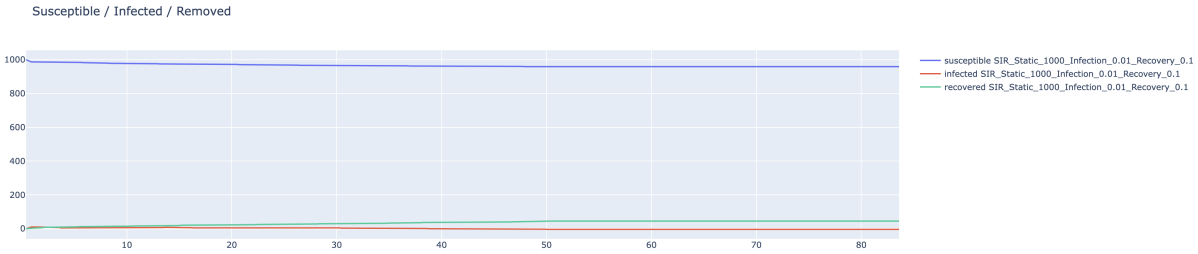
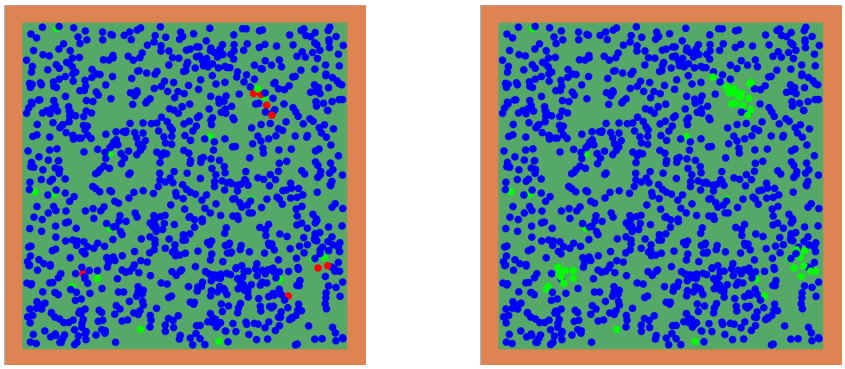


Figure 29: Visualization of Static 1000 Pedestrian with Infection Rate 0.01 and Recovery Rate 0.1



(a) Simulation time in secs: ~ 5 (b) Simulation time in secs: ~ 10

Figure 30: Infection Visualization with Infection Rate 0.01 and Recovery Rate 0.1

- **Visualization with Infection Rate 0.1 and Recovery Rate 0.01:**

  In this scenario, we have a very high infection rate while keeping the recovery rate as in 5.4.1. Therefore, the infection rate is very high in comparison to the recovery rate. As shown in 32a, the infection period is extremely fast, and at $\sim 14s$ all pedestrians are infected. And as in 32b, the recovery process is slow in comparison to the infection process, as half of the population recovered by $\sim 75s$. This variation indicates an epidemic model that is influential, and the result is a drastic and catastrophic epidemic.

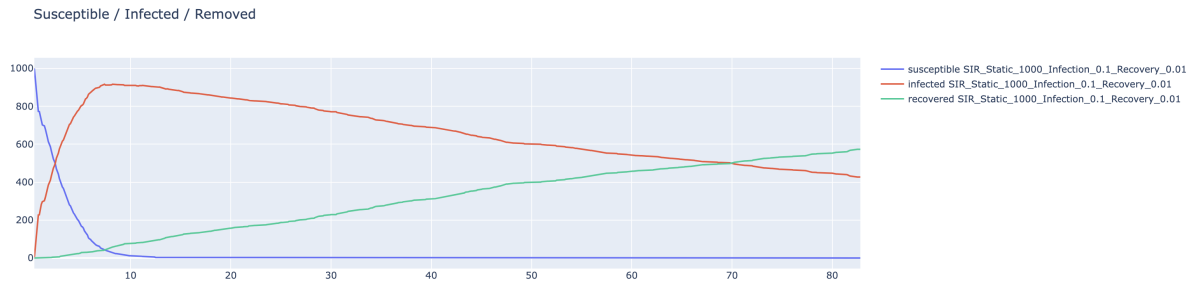
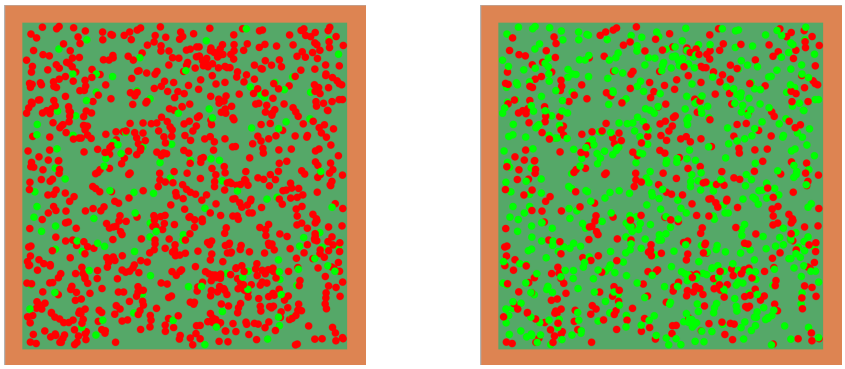


Figure 31: Visualization of Static 1000 Pedestrian with Infection Rate 0.1 and Recovery Rate 0.01



(a) Simulation time in secs: ~ 14 (b) Simulation time in secs: ~ 75

Figure 32: Infection Visualization with Infection Rate 0.1 and Recovery Rate 0.01

- **Comparison and Conclusion:** Some experiments with different infection and recovery rates can be seen in figures 25, 27, 29, and 31. The change in rates has different effects depending on how much of a change there is. For example, when the recovery rate is considerably higher in comparison to the infection rate, as is in figure 29, the infection doesn't even spread, and the pedestrians make a very fast recovery. When the recovery rate is relatively higher, as in 27, the infection does spread, but it's clear to see that recovery is good enough for not all pedestrians to get infected. Only around %75 of pedestrians get infected at some point during the simulation.

  When the rates are similar to each other, like in figure 20a, there's a level of infection spread depending on its rate, but a fully spread infection isn't the case before pedestrians start making a recovery. Comparing 23 and 25, with the same rates in themselves, but the infection rate is higher in 25. The difference is how much the infection is able to spread. The graph with the higher infection rate shows a higher curve for infected pedestrians, and also we can see that the lower infection rate causes not all pedestrians to be infected before the recovery happens. In addition, 27 with the same infection rate but slightly higher recovery rate shows a much faster recovery (nearly in half the time) compared to 23, but the graph in general looks similar. Finally, when the infection rate is significantly higher compared to recovery, as in 31, the infection spreads very fast, with its curve being very steep and the total number of infected pedestrians at one time being quite high. Though a full recovery still happens slowly (since the recovery rate is low).

5.5 Supermarket Scenario

In Vadere, a 30 meters $\times$ 30 meters canvas was taken and filled with obstacles, targets, and sources as seen in Figure 33. This is supposed to replicate a supermarket where the targets are shelves (and a cashier checkpoint also exists). The pedestrians spawn from the sources and, based on their pre-defined path, go to different shelves before heading to the checkout. In the simulation, 10 pedestrians are initially infected, and the infection rate and the recovery rate are 0.01.

The model parameter `pedPotentialPersonalSpaceWidth` is changed to see if it has any effect on how many people are infected as they go around the supermarket. For this, the sources were programmed to spawn 10, 15, and 20 pedestrians at once, and the `otential`ersonalSpaceWidth— set to 1.2 (default), 5, and 10.

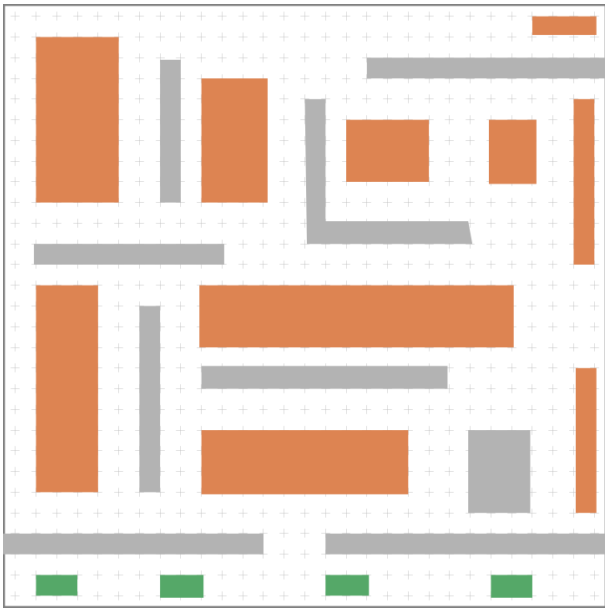
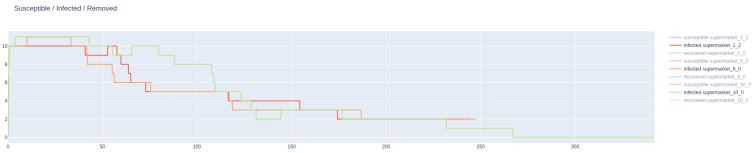
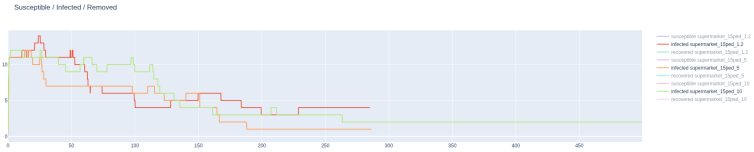


Figure 33: Supermarket simulation setup

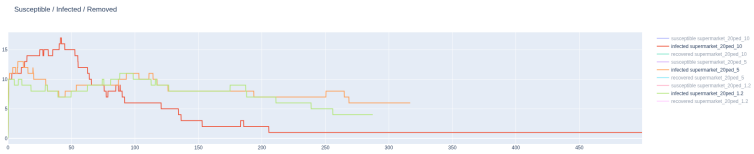
The result of the simulation is seen in Figure 34



(a) Number of infections for 10 pedestrians spawned by each source



(b) Number of infections for 15 pedestrians spawned by each source



(c) Number of infections for 20 pedestrians spawned by each source

Figure 34: Number of infections for pedestrians spawned by each source

It is visible from Figure 34 that there is some variance in the number of people infected at different times as `pedPotentialPersonalSpaceWidth`. This attribute changes the social-distancing area around each pedestrian, and although it slows down the total time taken by the pedestrians to reach the exit and leave, the 5-fold and 10-fold changes in the infection are not of such a big effect.

Additional tests

1. Test 1 Fewer pedestrians (5 per source) were used to see if reducing reduced the rate of infection. It was indeed true initially, as there were no new infections and the ones that were initially infected

recovered. This can be seen in 35.

Figure 35: Infection number with fewer pedestrians.

2. Test 2 This test allowed no infected pedestrians to enter the supermarket. This, as expected, caused no rise in infections and is a way to limit the spread. This can be seen in Figure 36.

Figure 36: Infection number with fewer pedestrians.

3. Test 3 The supermarket is closed, and no one can enter. There is no simulation needed as no infection is spread.

References

[1] Felix Dietrich, Gerta Köster, Michael Seitz, and Isabella von Sivers. Bridging the gap: From cellular automata to differential equation models for pedestrian dynamics. *Journal of Computational Science*, 5(5):841–846, September 2014.

[2] Felix Dietrich and Gerta Köster. Gradient navigation model for pedestrian dynamics. *Physical Review E*, 89(6), June 2014.

[3] Dirk Helbing and Péter Molnár. Social force model for pedestrian dynamics. *Physical Review E*, 51(5):4282–4286, May 1995.

[4] Benedikt Kleinmeier, Benedikt Zönnchen, Marion Gödel, and Gerta Köster. Vadere: An open-source simulation framework to promote interdisciplinary understanding. *Collective Dynamics*, 4:1–34, Sep. 2019. `https://collective-dynamics.eu/index.php/cod/article/view/A21`.

[5] Microsoft. Compare-object (microsoft.powershell.utility) - powershell. `https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/compare-object?view=powershell-7.5`, n.d. Accessed: 2025-05-22.