

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Placing Police Officers in Circle Pits Effectively

Johanna Hedlund Lindmar, Leo Horne, Michele Pagani, Zuowen
Wang

Zurich
December 2017

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

J.lindmar

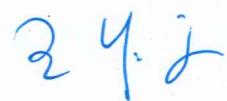
Johanna Hedlund Lindmar

LHorne

Leo Horne



Michele Pagani



Zuowen Wang

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PLACING POLICE OFFICERS IN CIRCLE PITS EFFECTIVELY

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

HEDLUND LINOMAR

HORNE

PAGANI

WANG

First name(s):

JOHANNA MARIEKE

LEO

MICHELE

ZUOWEN

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

ZÜRICH, DECEMBER 17, 2017

Signature(s)

J. Lindmar
J. Horne
M. Pagani
Z. Wang

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Contents

1 Abstract	7
2 Individual Contributions	7
3 Introduction and Motivations	7
3.1 Research Question	7
4 Description of the Model	7
4.1 Explanation of the Silverberg Model	7
4.2 Adaptations and Additions to the Silverberg Model	8
4.2.1 Centripetal force	8
4.2.2 Removal of noise	8
4.2.3 Repulsive force	9
4.2.4 Effect of policemen	9
4.2.5 Participation condition	9
4.2.6 Calculating the danger an individual is in	10
5 Implementation	10
5.1 Implementation of an Individual	10
5.2 Development of the Main Data Structure	10
5.2.1 Requirements for the main data structure	10
5.2.2 Design of the data structure	11
5.2.3 Implementation in Java	11
5.3 Implementation of the Model	11
5.3.1 Initializing the simulation	12
5.3.2 Running one timestep	12
5.3.3 Animating the simulation	12
5.4 Graphical Code	12
5.5 Configuring the Simulation	12
5.6 Collecting Data	13
6 Experiment & Methods	13
6.1 Determining the Parameters α , γ , μ , ϵ , ζ , and η	13
6.2 Configurations of Police Officers Used	13
6.3 Method for Data Collection	13
6.4 Methods Used for Analysis of Data	13
7 Simulation Results and Discussion	14
7.1 Effectiveness of Exemplary Police Officer Configurations	14
7.1.1 Configuration: Arrow	14
7.1.2 Configuration: Big Circle	14
7.1.3 Configuration: Center	15
7.1.4 Configuration: Circle	15
7.1.5 Configuration: Control	16
7.1.6 Configuration: Corner	16
7.1.7 Configuration: Cross	17
7.1.8 Configuration: C-shape	17
7.1.9 Configuration: Diagonal	18
7.1.10 Configuration: Double Half-Line	18
7.1.11 Configuration: U-shape	19
7.1.12 Configuration: Vertical Line	19

7.2	Comparison of Police Configurations	19
7.3	Issues with our Analysis	20
7.4	Limits to our Model	21
7.5	Judgment of the Danger Function	21
7.6	Comments on the Results	21
8	Summary and Outlook	22
9	References	22
Appendix A Launching the Simulation		23
Appendix B Explanation of Configuration File Format		23
Appendix C Configuration Files Used		25
C.1	Arrow	25
C.2	Big Circle	26
C.3	Center	27
C.4	Circle	28
C.5	Control	29
C.6	Corner	30
C.7	Cross	31
C.8	C-shape	32
C.9	Diagonal	33
C.10	Double Half-Line	34
C.11	Vertical Line	35
C.12	U-Shape	36
Appendix D Phase Diagrams		37
D.1	Arrow	37
D.2	Big Circle	37
D.3	Center	38
D.4	Circle	38
D.5	Corner	39
D.6	Control	39
D.7	Cross	40
D.8	C-Shape	40
D.9	Diagonal	41
D.10	Double Half-Line	41
D.11	U-Shape	42
D.12	Vertical Line	42
Appendix E Java Code		43
E.1	Main Code	43
E.2	Automatic Simulation	43
E.3	Control Panel	47
E.4	Data Collection	53
E.5	Individual	59
E.6	Position Matrix	61
E.7	Simulation	66
E.8	Graphical User Interface	80
E.9	Simulation Panel	82

Appendix F Python Code	87
F.1 Time Evolution of Participation Rate and Danger	87
F.2 Creation of Phase Diagrams	89

1 Abstract

Concerts are dangerous, in particular the “moshpits”. The aim of this paper is to reduce the hazard introducing “police officers” among the crowd, arranged as efficiently as possible. We developed a model based on Silverberg et al.’s studies. We used it to simulate the effects of police officers on circle pits. Interesting results emerged: namely, the simulations showed a central positioning of officer barely slowed down people, whereas an arrangement blocking off an entire section of the arena quickly stopped them.

2 Individual Contributions

The authors contributed equally to this work.

3 Introduction and Motivations

If you have ever been to a rock concert, you probably know the term “mosh pit”. That means people dance and run together in a form of crowd with the loud music. People often forget how dangerous it could be – even fatal – for individuals in the crowd.

Our group wanted to reduce the danger in the crowd movement. We set up different configurations corresponding to different security arrangement. We developed a simulation based on Java and ran fifteen thousand rounds of simulations for each configuration. Although we did not collect real-world data, with the help of modeling the crowd using existing physical models – based on a study from Silverberg et al –, we drew interesting conclusions based on our simulation results.

We extracted three major factors from the data, namely average danger level in time evolution, participating rate and danger levels of individuals. Then we use python scripts to generate three sorts of graphs correspondent to configurations we have chosen.

3.1 Research Question

Our research question is therefore: what is the most efficient placement of police officers among the crowd at a heavy metal concert so that the formation of circle pits is stopped swiftly (in the least amount of time) and safely (subjecting both the participants and non-participants of the circle pit to the least possible amount of danger)?

4 Description of the Model

The model is based on a paper by Silverberg et al. [1]. We will briefly explain the chief ideas of their model before explaining which adaptations and additions we had to make. In the following sections, we will refer to this model as the *Silverberg model*.

4.1 Explanation of the Silverberg Model

The Silverberg model is essentially a social force model: each individual is subject to a certain number of forces, which are composed of *individual interests* and *environmental influences* [2]. The motion of the crowd is calculated analogously to a many-particle system: Together with the initial conditions, the time evolution of a system of N particles is determined by Newton’s law, which allows us to compute the acceleration of one particle a_j out of the sum of the forces F_i acting on it:

$$\sum_i F_i = m a_j \quad (1)$$

whereby masses are assumed to be the same for all individuals and therefore not considered in the model.

One of the forces used in the Silverberg model is the repulsive force that acts between two individuals as soon as they collide with each other. It acts by increasing the distance between the colliding people (equation 2). The social force model also considers a force that accelerates or decelerates the individuals towards a specific *preferred speed*: a propulsive force is added which is proportional to the difference of the preferred speed and the actual current speed (equation 3). To simulate the flocking behavior of the members of a circle pit, a flocking force that simply averages the velocities of the individuals within a certain radius r_{flock} is added (equation 4). The authors of [1] also added a random noise vector to the individuals to simulate the randomness of human movements (equation 5).

Putting all this together, Silverberg et al. obtained the following set of equations governing the movements of individual i :

$$\vec{F}_i^{\text{repulsion}} = \begin{cases} \epsilon \left(1 - \frac{r_{ij}}{2r_0}\right)^{\frac{5}{2}} \hat{r}_{ij} & \text{for } r_{ij} < 2r_0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$\vec{F}_i^{\text{propulsion}} = \mu(v_0 - v_i)\hat{v}_i \quad (3)$$

$$\vec{F}_i^{\text{flocking}} = \alpha \sum_{j=0}^{N_i} \vec{v}_j \Bigg/ \left\| \sum_{j=0}^{N_i} \vec{v}_j \right\| \quad (4)$$

$$\vec{F}_i^{\text{noise}} = \vec{\eta}_i \quad (\vec{\eta}_i \text{ is random}) \quad (5)$$

where r_0 is the radius of an individual, r_{ij} is the center-to-center distance between two individuals i and j , v_0 is the preferred speed of an individual, and v_i is the instantaneous speed of individual i . N_i is the number of neighbors of i within the radius r_{flock} . A circumflex ($\hat{\cdot}$) above a variable denotes a unit vector. ϵ , μ , and α are real numbers representing the relative magnitudes of these forces.

4.2 Adaptations and Additions to the Silverberg Model

The Silverberg model focuses on simulating moshpits. To adapt it to circle pits and to simulate the effects of the policemen on the crowd, we had to make a couple changes to it

4.2.1 Centripetal force

A commonly stated goal while participating in a circle pit is to move towards the center of the circle. According to the social force model, this *individual interest* can be considered as a further acting force on the individual. We therefore chose to consider a centripetal force that pulls each participating individual towards one specific point in the center \vec{c} (which is the same for each participating individual). The centripetal force on individual i can be described by equation 6:

$$\vec{F}_i^{\text{centripetal}} = \gamma \frac{\vec{c} - \vec{x}_i}{\|\vec{c} - \vec{x}_i\|} \quad (6)$$

where \vec{x}_i is the position vector of individual i and γ is a real number representing the relative magnitude of the centripetal force.

4.2.2 Removal of noise

The Silverberg model is designed to simulate general coordinated crowd phenomena, which also includes the mosh pits. In a mosh pit, people move in random directions with the aim of colliding with their neighbors as often as possible. The random noise that is added to the force

therefore fulfills a considerable function. Since we are limiting our analysis to circle pits and since people generally do not aim to collide with other people while participating in a circle pit, we decided not to implement the random noise.

4.2.3 Repulsive force

The repulsive force that is used in the Silverberg model acts between two individuals as soon as they touch each other. This choice of force is sensible when analyzing general mosh pits in which the participating people do not try to avoid collisions (and even aim for them). Since collisions are considered undesirable in circle pits, our choice of the repulsive force leads to a repulsion even if two individuals approach within a certain radius of each other, even though they are not yet touching. This can be led back to an additional *individual interest*. We consider two separate cases: either two individuals i and j are already “overlapping” (i.e. they are occupying the same physical space) or they are merely in proximity to each other. This leads to the following set of equations for the repulsive force:

$$\vec{F}_i^{\text{repulsion}} = \begin{cases} 500\epsilon\hat{r}_{ij} & \text{if } r_{ij} < 2r_0 \\ \frac{\epsilon\vec{r}_{ij}}{r_{ij}(2r_0 - r_{ij})} & \text{if } r_{ij} < 2r_1 \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

The reason for splitting the calculation of the repulsive force into these categories is to discourage the overlapping of individuals much more strongly than their being close to each other. In equation 7, r_0 remains the radius of one individual, but r_1 is another constant that represents the radius within which being in proximity to another individual incurs a repulsive force.

4.2.4 Effect of policemen

The effect of a policeman that is placed within the crowd is to prevent people in its direct environment to participate in the circle pit. That means that as soon as an individual is closer to a policeman than a given radius, its participating state will automatically change to “is not participating”, with all the consequences discussed in the subsection 4.2.5.

4.2.5 Participation condition

Contrary to the Silverberg model, the number of participants in the circle pit is not fixed. An individual’s will to participate is influenced by its neighborhood: conditions — such the number of people participating around him — will increase, or respectively, decrease it. This *will* is represented by the boolean variable `isParticipating`, which follows two conditions:

1. If the individual is not participating and has at least `minParticipantNeighbors` other participating individuals around him, he will start participating
2. If the individual is participating and has less than `minCirclePitSize` other participating people around him, he will stop participating.

If a person is participating, he is subject to all the forces described by our model. Otherwise, he is not subject to the flocking force and the centripetal force and his *preferred speed* is considerably reduced.

In total, the force \vec{F}_i acting on individual i is calculated as follows:

$$\vec{F}_i = \begin{cases} \vec{F}_i^{\text{repulsion}} + \vec{F}_i^{\text{propulsion}} + \vec{F}_i^{\text{flocking}} + \vec{F}_i^{\text{centripetal}} & \text{if } i \text{ is participating in the circle pit} \\ \vec{F}_i^{\text{repulsion}} + \vec{F}_i^{\text{propulsion}} & \text{otherwise} \end{cases} \quad (8)$$

4.2.6 Calculating the danger an individual is in

Since one of our goals is to analyze how much danger individuals are subject to in a circle pit and minimize it through the clever placement of police officers and the arena, we need a way to assess how much danger an individual is in.

An individual i can be said to be in danger if the norm of the force \vec{F}_i acting upon it is large. Due to equation 1, a high force implies an increase of velocity and a high acceleration, both of which are dangerous. Furthermore, the danger is increased if i is at a very dense spot in the arena (i.e. has many neighbors). We therefore define the function $\text{density}(i)$ as the amount of neighbors of i within the radius r_{flock} . Putting this together, we obtain the following danger function:

$$\text{danger}(i) = \frac{\|\vec{F}_i\|}{\zeta} + \frac{\text{density}(i)}{\eta} \quad (9)$$

where ζ and η are real numbers that must be chosen so that $\|\vec{F}_i\|$ and $\text{density}(i)$ contribute approximately equally to $\text{danger}(i)$ (they depend on the values chosen for α , ϵ , μ , and γ).

5 Implementation

We decided to implement our model in Java because MATLAB was too slow to be able to animate the simulation in real time.

5.1 Implementation of an Individual

An individual is represented by the `Individual` class. This class simply stores the position, velocity, and preferred speed of the individual. Furthermore, it stores whether the individual is participating in the circle pit, the parameters necessary to calculate the danger the individual is in (norm of force and density) and the current danger level.

There are also convenience methods to facilitate certain tasks like finding the distance between two individuals or between the individual and a point.

5.2 Development of the Main Data Structure

5.2.1 Requirements for the main data structure

In order to implement our model, we needed a data structure for the individuals that could do the following things efficiently:

1. Get the neighbors of a given individual (we should not have to traverse the entire data structure to find them)
2. Move an individual in the arena
3. Place an individual at an arbitrarily precise position in the arena

5.2.2 Design of the data structure

To do this, we initially considered using position matrix like the one used in the cellular automaton model (where each individual x is located at a particular cell (i, j) of the matrix). This would allow us to efficiently find the neighbors of x , as we simply need to look at the cells around position (i, j) . However, implementing such a matrix would not be possible due to the large number of individuals involved (by our estimates, even a simulation with 500 individuals would have resulted in a matrix taking up approximately 2 GB of RAM). Furthermore, one of the drawbacks to this method would be that the number of positions an individual can be at at any given time is limited to the number of cells of the matrix.

We therefore decided to split the matrix up into sectors. Each sector can contain several individuals at the position indicated by their position property (the positions are relative to the entire matrix, not the sector). This also does not affect the ability to efficiently find the neighbors of an individual x , since if the partition into sectors is fine-grained enough, we can simply look at the sectors in the vicinity of the sector containing x , which would not incur too large of a performance loss. Each sector contains a specific coordinate space, so the position of an individual must be in the coordinate space of its sector. For example, in our implementation, each sector contains 10 coordinate positions, so any individuals with a position $(i, j) \in [0, 10) \times [0, 10)$ are in the sector at position $(0, 0)$ in the matrix (i.e., the sector in the upper left corner). This approach also allows individuals to be located at arbitrarily precise locations (subject only to the machine precision ε).

5.2.3 Implementation in Java

In our implementation, such a sector is actually a `java.util.ArrayList` to which we can add as many individuals as we want. The precise location of each individual within its sector is defined by its position property.

The final result is a data structure based on a matrix of `ArrayList`s, whereby each `ArrayList` is a list containing all the `Individual` objects in the sector represented by that `ArrayList`. We also defined certain convenience methods and properties to simplify the usage of the data structure, including:

- A `Sector` inner class to represent a sector (defined by a row and a column index within the matrix)
- An `ArrayList` of all the individuals currently in the matrix to allow easy iteration over all the individuals in the matrix
- Getter methods for getting the `ArrayList` at a particular position (i, j) or at the location specified by a `Sector` object
- An “add” method to add an individual to the matrix at the proper sector
- Methods to transform (x, y) coordinates to a sector
- A method to get the neighbors of a specified individual

This functionality is implemented in the `PositionMatrix` class. The precise functionality of each method is described in the Javadoc comments.

5.3 Implementation of the Model

Using the `PositionMatrix` and `Individual` classes, implementing the model was relatively straightforward. The entire simulation is implemented in the `Simulation` class, which has methods to initialize the simulation, to run one timestep of the simulation and to run the simulation as a live animation.

5.3.1 Initializing the simulation

All the necessary parameters of the simulation are read via the constructor of the `Simulation` class. After that, we simply need to initialize the position matrix. To do this, we simply generate individuals with a random position and velocity and then add them to the position matrix. Each individual participates in the circle pit with a certain probability.

5.3.2 Running one timestep

In this method, we run the simulation for all individuals in one time step. In order to achieve this, we iterate over all the individuals and calculate the joint force \vec{F}_i acting upon them, which according to equation 1 gives the second derivative of the position. To integrate this differential equation numerically, we make use of the *Leapfrog Method*. This method is based on the idea of updating the positions and velocities at interleaved time points. It is known to be less affected by error accumulation due to rounding errors (i.e. more numerically stable). The formulas and parameters we use for calculating \vec{F}_i are based on equation 8 derived from the Silverberg model [1] and our own additions and adaptations of it.

We also update the individuals' levels of danger based on an appropriate scale of joint force and density in two ways: we calculate a *discrete danger level* (DDL) ($DDL \in \{0, \dots, 6\}$) is a discrete measure of the danger level useful for visualizing the danger level) and a *continuous danger level* (CDL) corresponding to $danger(i)$ from equation 9. We use different levels of colors to represent the DDLs. The two factors contributing to the DDL, density and joint force, can be activated or deactivated respectively by two check boxes in the GUI. This helps the users to see different impacts from these two factors.

5.3.3 Animating the simulation

Animating the simulation was very simple: every 50 milliseconds, we run one timestep of the simulation and update the graphics. If necessary, we also collect data (see Section 5.6).

5.4 Graphical Code

This subsection describes the code necessary to create the GUI and animate the simulation on the screen.

We used the Swing framework to create the graphics. The main window is simply a `JFrame` containing two `JPanels`: one for the simulation and one for the controls pertaining to the model.

Drawing the simulation is very simple: we simply need to iterate over all the individuals of the matrix and draw them based on their position property, with some additional scaling to fit the window size. The animation is achieved simply by periodically running one timestep and then refreshing the window.

The controls were also very simple to implement: each one of them simply changes the corresponding field of the current `Simulation` instance.

The main window is implemented in the `SimulationGUI` class, the animation is done in `SimulationPanel`, and the panel with the controls is in `ControlPanel`.

5.5 Configuring the Simulation

To be able to efficiently vary the parameters and police officer configurations, we needed a way to efficiently specify how a simulation should be run so that it could be run automatically. To this end, we defined how a configuration should be run using configuration files. The format of these configuration files is explained in Appendix B.

5.6 Collecting Data

In order to perform our analysis, we needed a way to automatically collect data. We defined two classes for this: `DataCollector` and `AutomaticSimulator`. `DataCollector` simply runs the simulation by repeatedly running one timestep and gathers the positions and velocities acting on the individuals as well as the forces acting on them and their densities according to the density function. Furthermore, we output the CDLS as well as the average CDL, maximum CDL and median CDL. `AutomaticSimulator` runs several simulations with various random seeds using the configuration files specified in a file called `configs.sim` and uses `DataCollector` to run one instance of the simulation and collect data on that simulation.

6 Experiment & Methods

6.1 Determining the Parameters α , γ , μ , ϵ , ζ , and η

We had to find a suitable configuration of these parameters in order to make the simulation closer to reality. This set of parameters is applied to every configuration of policemen, since varying would exceed the bounds of this study (lead to too much data). By experimenting with different values, we established that the values $\alpha = 700$, $\gamma = 600$, $\mu = 10$, and $\epsilon = 2$ generate a reasonably accurate-looking circle pit simulation. Having fixed these values, we could determine ζ and η by looking at the average force and density and selecting them so that $||\vec{F}_i||$ and $\text{density}(i)$ contribute approximately equally to $\text{danger}(i)$. We selected $\zeta = 10000$ and $\eta = 50$.

6.2 Configurations of Police Officers Used

We decided to test eleven police officer configurations. The configuration files for these configurations are given in Appendix C.

Each configuration has exactly ten police officers so that the effectiveness of the configurations can be easily compared.

6.3 Method for Data Collection

We collected data once every second of the simulation, i.e. every 20 timesteps. We also chose 50 different random seeds for the initial velocities and positions of the individuals.

6.4 Methods Used for Analysis of Data

To analyze the data we collected, we plotted two graphs for each arrangement of police officers, that show the time evolution of quantities describing how the circle pit situations evolve. Those graphs showed the median of the *danger level* along the time, averaged over the various simulations with different random seeds. That did not only allow a good and immediate comprehension about the efficiency of a certain configuration, but also gives us a good method to compare the methods to each other. The y-axis of the graph showing the median of the danger level with respect to the time as for example seen in 1, is chosen so that median danger = 1 corresponds to the maximum value, which in our cases is achieved shortly after the police officers are inserted to the crowd.

We also generated a graph showing the participation rate over time of all the configurations. The participation rate equals one if all individuals in the crowd are joining the circle pit and it is zero as soon as nobody is participating anymore.

Furthermore, we created phase diagrams showing the danger “hot spots” at a specific time, averaged over all the random seeds. We generated separate phase diagrams to show the influence of force and the danger of density of the total danger level of the individual.

7 Simulation Results and Discussion

7.1 Effectiveness of Exemplary Police Officer Configurations

In the graphs below, if we compare the graphs with the danger level to the graphs with the participating rate, it seems that when the participation rate is zero, the danger level hovers at around 0.47 (just below 0.5).

7.1.1 Configuration: Arrow

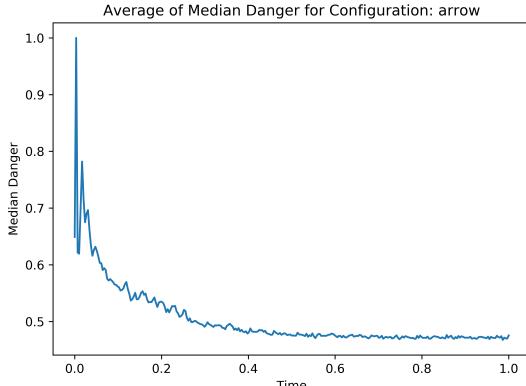


Figure 1

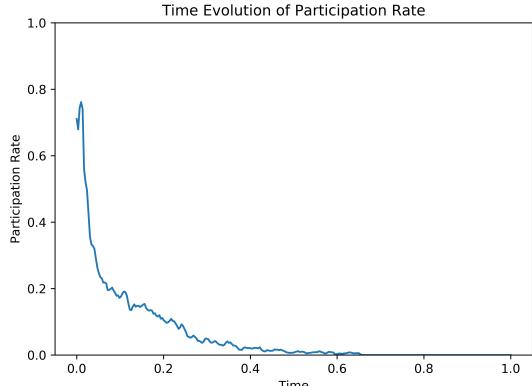


Figure 2

The configuration “arrow” is similar to the configuration ”double half line”. And it is also very good at decreasing the participation rate, since the two lines of policemen stop quite a lot of people. From the median danger aspect this configuration is also very useful since the level converges to 0.5 within a short period of time.

7.1.2 Configuration: Big Circle

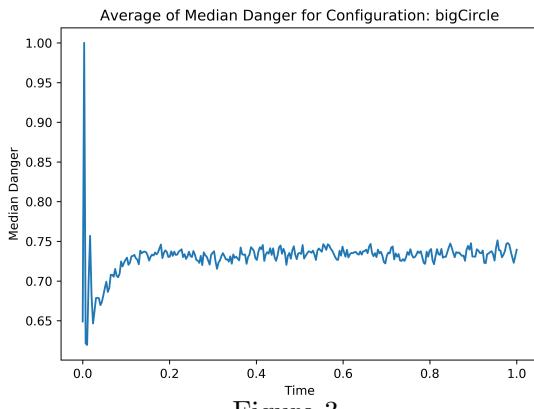


Figure 3

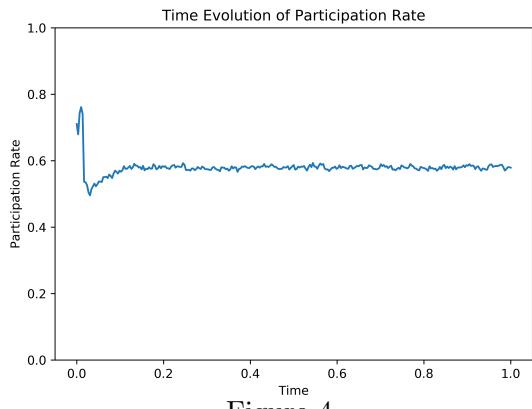


Figure 4

The configuration “big circle” is not very effective. First of all, the policemen are all placed near the middle of the arena, so the circle pit can still move around the policemen. Also, since the line of policemen outlining the big circle is not complete, the participants of the circle pit can still slip in between the gaps where there are policemen, thus contaminating the people inside the circle and rendering the circle of policemen quite useless.

7.1.3 Configuration: Center

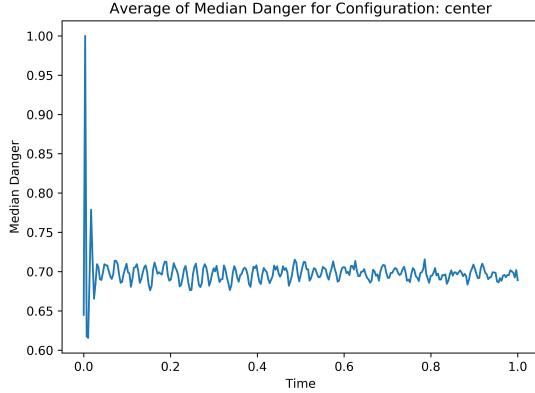


Figure 5

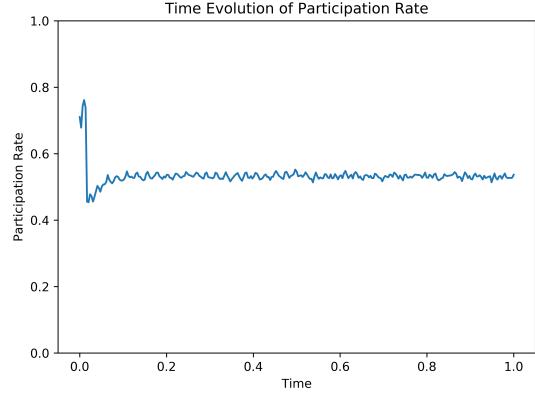


Figure 6

The configuration “center” is also does not work very well. Since all the policemen are in the middle, they do not affect the crowd circling around them. Thus, both median danger level and participation rate stay at a high level (0.7 and 0.45 respectively).

7.1.4 Configuration: Circle

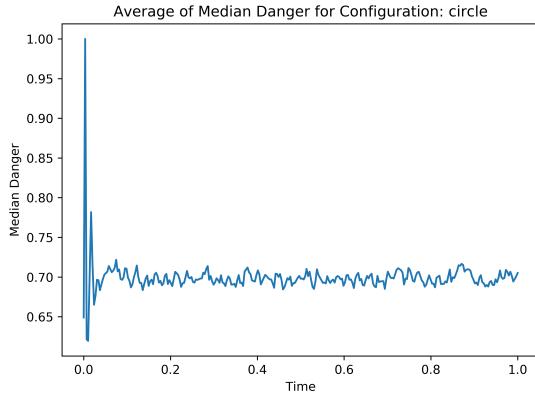


Figure 7

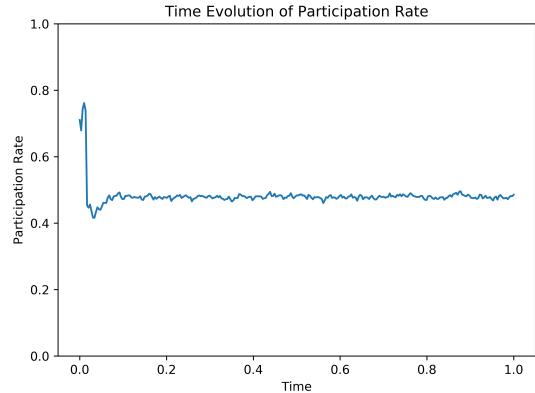


Figure 8

The “circle” configuration is not very effective at stopping the circle pit because the policemen are only placed at the center of the arena (similarly to the “center” configuration). The circle pit can still rotate around the cluster of policemen. Moreover, the non-participants inside the circle of policemen often are squeezed very tightly since they experience repulsive forces from all sides of the circle due to the influx of participating members that suddenly stop participating, making the density in the circle very high (and therefore making it very dangerous).

7.1.5 Configuration: Control

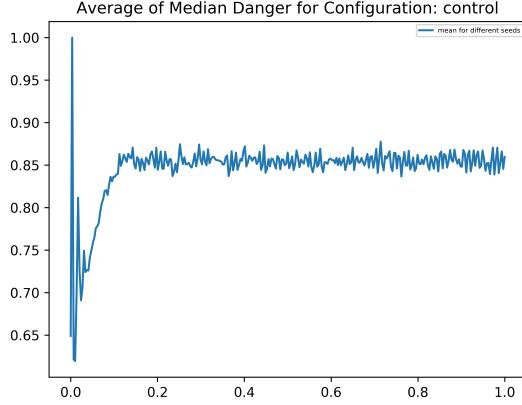


Figure 9

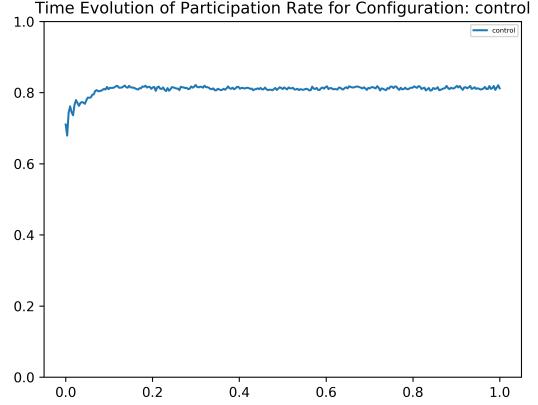


Figure 10

The configuration “control” works as a control group to show that the policemen really make a difference in controlling the crowd movement. The danger level without police officers controlling the crowd is much higher than the danger level with police officers, even if the police officer configuration did not succeed in stopping the crowd. The participating rate is also much higher than the other configurations.

7.1.6 Configuration: Corner

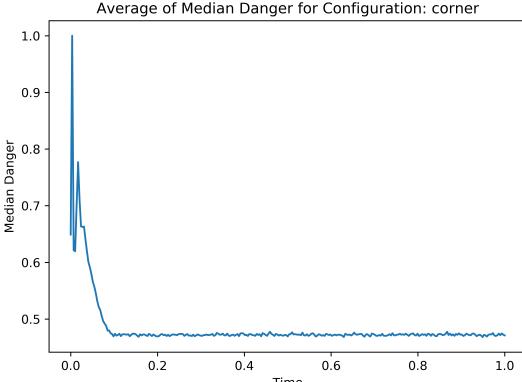


Figure 11

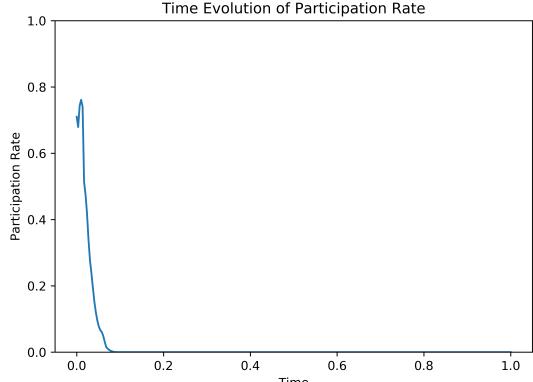


Figure 12

The “corner” configuration proved to be very effective at stopping the circle pit without causing a large amount of danger. Since one entire corner is blocked off by the police and since the individuals move freely, at some point, all the participating individuals will have moved out of that corner. This means that now anyone entering the corner will enter a “region” of the arena where no one is participating. After a while, every participating individual has passed through the corner and the circle pit is stopped. Since the circle pit tends to accumulate in one big mass (i.e. not distributed evenly across the arena), an individual exiting from the corner does not join the circle pit again (the “mass” of participating individuals is at the “entrance” to the corner).

7.1.7 Configuration: Cross

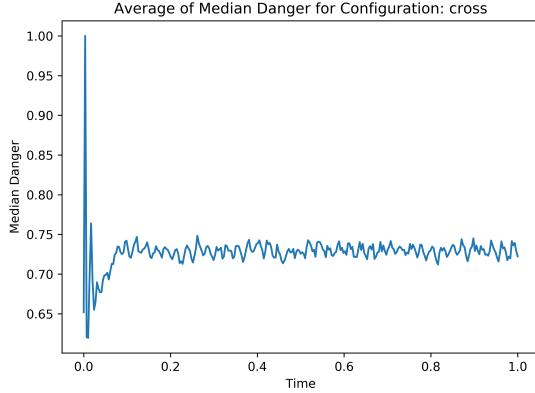


Figure 13

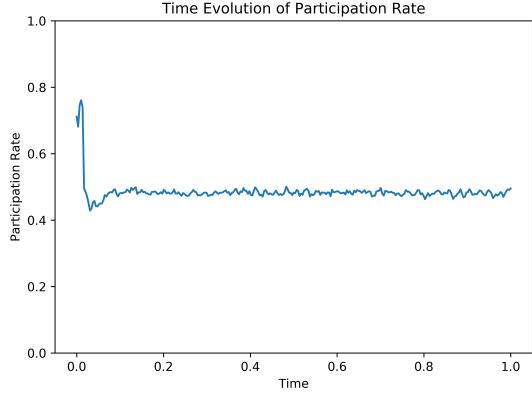


Figure 14

The “cross” configuration did not prove to be very effective. The circle pit was never stopped (the participation rate never reached 0) and individuals remained at a very high level of danger (around 0.73). This can be explained by the fact that the police officers are only present in the center of the arena. The individuals are still free to circle around the cross formed by the policemen. Furthermore, after passing one “arm” of the cross, individuals can still be “contaminated” by individuals circling around the cross.

7.1.8 Configuration: C-shape

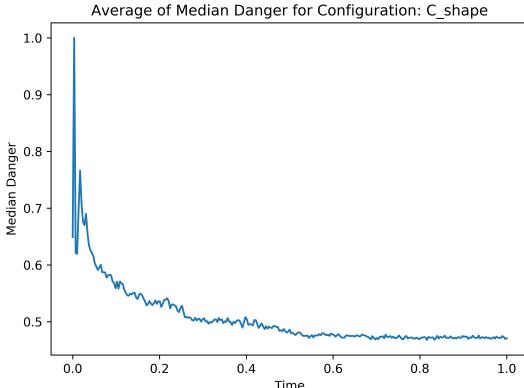


Figure 15

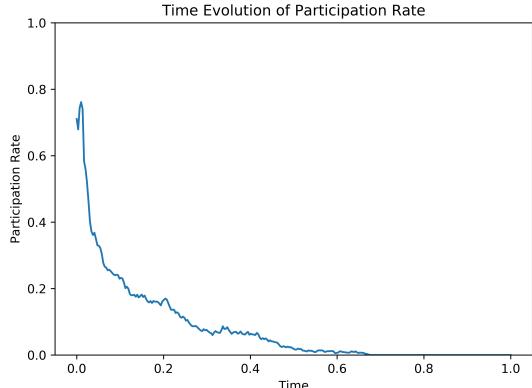


Figure 16

The “C-shape” configuration seems to be quite useful. The circle pit stopped rather quickly, namely the rate of participation goes to zero rapidly. Also the average of median danger declined in short time as well. The two curves are similar to the the configuration “Double Half-Line”. But we observed the crowd is slightly split into two discontinuous parts.

7.1.9 Configuration: Diagonal

7.1.9

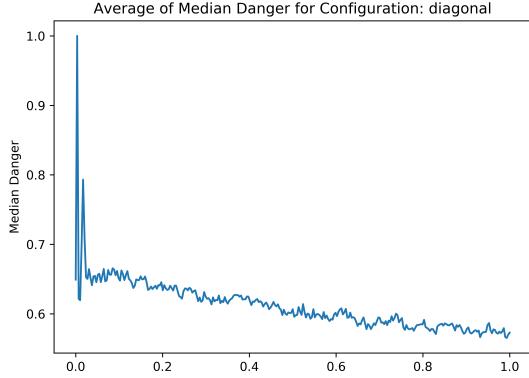


Figure 17

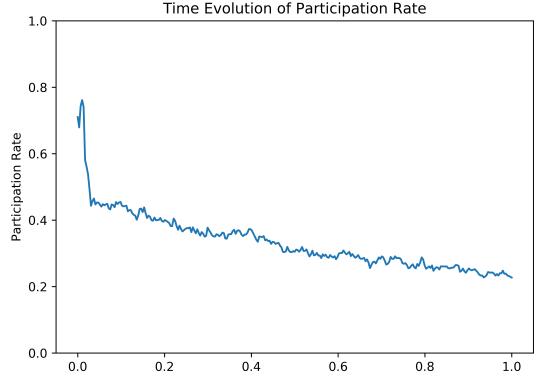


Figure 18

The “diagonal” configuration works not very well with decreasing the median danger (to around 0.6). The circle pit was also never stopped. This can be explained by the fact that the police officers are standing on the diagonal of the arena. The individuals can still rotate and move like without policemen, but decrease their speed every time passing the diagonal. The area of influence of the policemen is a square and these squares are arranged in a diagonal line. This means that two squares always touch corners (i.e. the lower right corner of one square touches the upper left corner of the next square etc.). This means that at the points where two squares are touching, the policemen have almost no effect. The circle pit can therefore still be propagated across these interstices, and the diagonal configuration is almost always unable to stop the circle pit.

7.1.10 Configuration: Double Half-Line

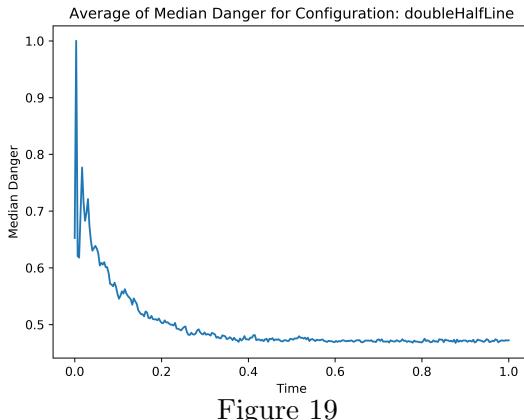


Figure 19

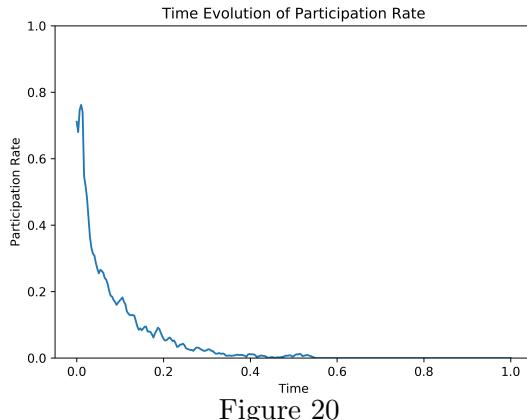


Figure 20

The configuration ”double half-line” is very effective. It decreases both the median danger level and the participating rate to a very low level within a short time. One reason for this is that the area created by the policemen effectively stores many people and forces them stop participating. However, we noticed while running the simulation that the circle pit has a tendency to split up into two separate direction after crossing the double half-line and collides at the double half-line. This possibly contributes to the fact that danger remains fairly high and only tapers off gradually compared to other configurations.

7.1.11 Configuration: U-shape

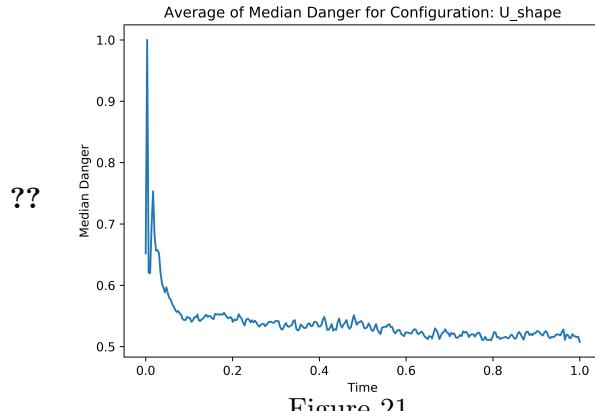


Figure 21

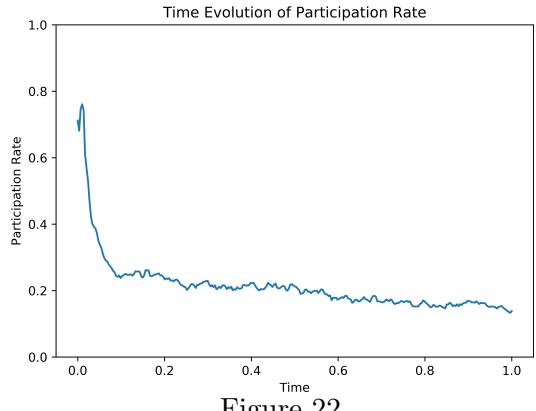


Figure 22

The configuration "U-shape" is very effective in decreasing of the median danger level, since it quickly stops almost half of the people within the U shape. But the entire group never stops (participating rate never goes to zero) because there will be a small group of people keeps on moving outside the area of U shape. It is likely that increasing the size of the U would be much more effective because more people would get "trapped" in the U.

7.1.12 Configuration: Vertical Line

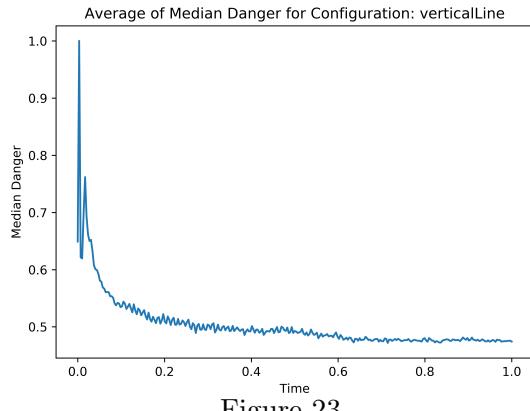


Figure 23

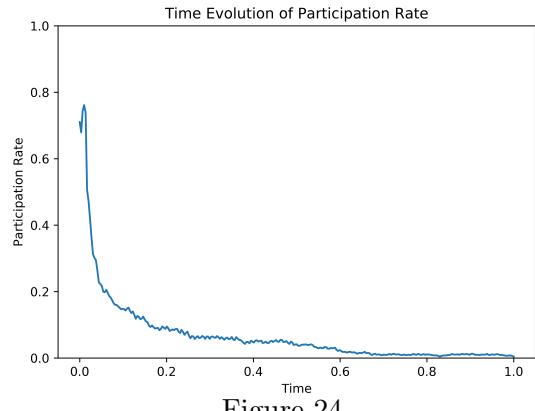


Figure 24

The configuration "vertical line" is very powerful in stopping the crowd. Initially, there are participants on both sides of the line. However, after both sides cross the line several times, the amount of participants dwindles significantly until there are so few participants on either side of the line that they do not influence "newcomers" having just crossed over the line from the other side. Because of this, the participation rate very quickly converges to a level below 0.5.

7.2 Comparison of Police Configurations

Since the aim of this simulation was to determine the most efficient placement of policemen, we will now compare the different configurations we tested.

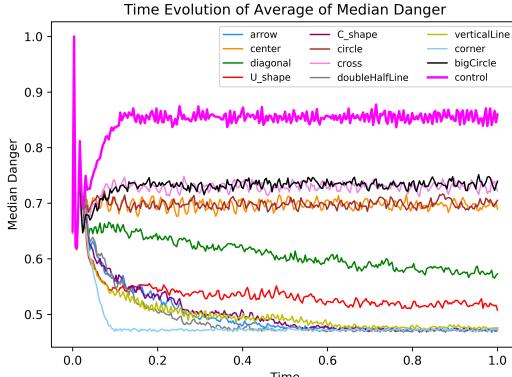


Figure 25

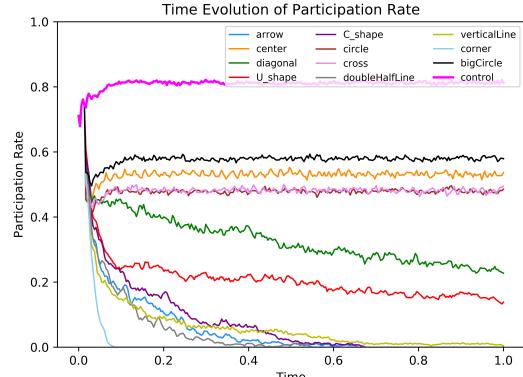


Figure 26

Figure 25 shows a graph containing the danger levels of all the configurations we tested. As mentioned before, all the configurations induce a lower danger level than the control configuration, even if the circle pit was not stopped.

Based on Figure 25, we classified the configurations into three categories:

1. Circle pit eliminated (danger level below 0.5)
2. Circle pit not eliminated, but danger level at end below 0.6
3. Circle pit not eliminated and final danger level between 0.6 and 0.8

Looking at Figures 25 and 26, we can easily see that the configurations “arrow”, “C-shape”, “corner”, “diagonal”, and “double half-line” belong to the first category. “Corner” stands out from the other in that it decreases the participation rate to zero considerably faster than the other configurations. These configurations all block off a section of the arena through an impenetrable wall of police offices large enough to substantially reduce the amount of participating individuals passing through it. Furthermore, the wall is placed in such a way that individuals emerging from the wall are far away from the mass of participating individuals and do not feel any motivation to join the circle pit again.

The configurations “U-shape” and “diagonal” fall within the second category. Although they do not manage to bring the participation rate to zero, they keep the amount of danger at a fairly safe level. As mentioned in sections 7.1.9 and ??, they both reduce the amount of participating individuals, but do not provide enough coverage over the entire arena (in the case of “U-shape”) or have interstices that allow the circle pit to seep through the line of policemen (in the case of “diagonal”).

The third category contains the configurations “big circle”, “center”, “circle”, and “cross”. These configurations all have the property that the policemen are clustered at the center of the arena so that the crowd can still form a circle pit around them. Therefore, the participation rate and danger level do not decrease as with the other configurations, but remain at a fairly stable level.

In conclusion, we can see that all of the configurations from the first category fulfill the needs of our research question. However, “corner” seems to be particularly well suited. Also, the mere presence of police officers is enough to substantially reduce the danger level and participation rate at the rock concert (compared to the “control” configuration with no police officers).

7.3 Issues with our Analysis

For our analysis, we had to collect data once every second instead of every timestep since collecting data every timestep would have lead to around 250 GB of data to analyze as well as slowing down the simulation considerably. Ideally, we would have to rerun the simulation on a supercomputer while collecting data every timestep to obtain more precise results.

7.4 Limits to our Model

In our model, we are only simulating the appearance of one single circle pit with respect to a pre-defined center. In reality, often more than one circle pit occur simultaneously and, since they mostly happen spontaneously, the center emerges naturally and may even change with time. Moreover, our experiment omits phenomena that would have to be considered in a real-life situation. First of all, the danger the police is in while regulating the circle pit is not considered. It could very well be that the “corner” configuration (which was very effective at stopping the circle pit) puts the police officers in so much danger that it would be practically infeasible to implement such security measures at a real concert. Related to this, even if one of our configurations could be implemented in real life, the police officers would probably not be able to remain in formation due to the flow of the circle pit. Also, in real life, not every individual would react to the police officers the same way. In our model, an individual under the influence of a police officer immediately stops participating. In real life, certain individuals might be more daring and might continue participating even when faced with a police officer. Finally, using our model, we could only test a finite amount of police officer configurations. In our implementation, the police officers can only be at 100 different positions. They also have the exact same area of influence, whereas in real life, certain police officers might have a more pronounced effect on the crowd than others.

7.5 Judgment of the Danger Function

Looking at Figures 25 and 26, we can see that the danger function described by equation 9 was well-chosen. The curves for the danger level and the participation rate correspond quite closely, which makes sense due to the fact that a circle pit is obviously associated with more danger.

7.6 Comments on the Results

All in all, our results were somewhat unexpected (in fact, we only included the “corner” configuration because we thought it might be interesting, not because we suspected it might be an effective configuration). Also, configurations that we thought would do well actually turned out not to work at all (for example with diagonal). However, in each case, the results can be easily explained. In section 7.1.6 we explained why we think “corner” turned out to be so effective and also explained the major flaw with “diagonal” in section 7.1.9.

Furthermore, we discovered that the circle pit is actually quite sensitive to small changes in the configurations. For example, the “U-shape” and “C-shape” configurations are very similar (one is the rotation of the other; the motivation behind both of them was to provide a “hollow” that the members of the circle pit would run into, thereby stopping the circle pit). However, “U-shape” did not stop the circle pit whereas “C-shape” did.

Also, asymmetric configurations such as “C-shape” and “arrow” were able to stop the circle pit, even though they were both designed with the same “hollow” idea as mentioned previously. This means that even though the circle pit did not always move counter-clockwise, these configurations were still effective.

Finally, we would like to note that configuration with a double line of police officers seemed to be a little more effective than configurations with only a single line. For example, most of the time the danger level for “double half-line” was lower than that of “vertical line” (see Figure 25), whereby we can consider “double half-line” as a doubled version of half of the “vertical line” configuration.

8 Summary and Outlook

To summarize, as seen in Figures 25 and 26, we were able to compare the quality of different police configurations by observing the time evolution of the danger level and the participation rate.

We found out that the configuration “corner” (a configuration in which the police officers completely block off one of the corners of the arena) was by far the most effective.

On the other hand, we found that placing police officers near the center of the arena was the least effective because the individuals could simply circle around the police officers and continue participating.

The following ideas would have exceeded the bounds of our project but would be very interesting to consider: first of all, it would have been exciting to test our simulation with varying numbers of police officers so as to determine the relative effectiveness of single police officers. Also, as an alternative measure of crowd control, we also would have liked to implement barriers that the members of the audience could not traverse (and no social forces between people on opposite sides of the wall are present).

However, all in all, we are fairly satisfied with our results since we were able to answer our research question to a reasonable extent. For us, it was a challenge at first to get the code to run (for example, we had trouble defining a functioning version of the repulsive force since the individuals always overlapped). At first, we did not quite know where our project was headed, but after our simulation ran properly, we were able to think about the analysis of our results and find a path to a satisfactory answer to our research question.

9 References

- [1] J. L. Silverberg, M. Bierbaum, J. P. Sethna, and I. Cohen, “Collective Motion of Moshers at Heavy Metal Concerts,” *Physical Review Letters*, vol. 110, no. 228701, 2013.
- [2] D. Helbing and P. Molnár, “Social Force Model for Pedestrian Dynamics,” *Physical Review E*, vol. 51, no. 4282, 1995.

Appendix A: Launching the Simulation

Before launching the program, you must decide whether to run an automated simulation or a manual simulation. To do so, comment out the corresponding section in `Main.java`. In the example below, we are running an automated simulation.

```
1 // Run a manual simulation
2 // Simulation simulation = new Simulation(2, 10, 700, 600, 500,
3 //                                         8, 0.01, 0.5, 6, 1, 3);
4 // simulation.runManualSimulation();
5
6 // Run an automatic simulation
7 AutomaticSimulator simulator = new AutomaticSimulator();
8 simulator.run();
```

If you are running an automated simulation, you must also specify which configuration files from the `configs` folder to test by listing their names (without the trailing `.config`) in the `configs.sim` file. The program can then be launched in one of the following ways:

1. Importing the Eclipse/IntelliJ IDEA project located in the `java_source` folder into Eclipse or IntelliJ IDEA and running the target `Main`
2. Executing the following commands from the `java_code` folder in a terminal (assuming the Java compiler `javac` and the Java Runtime Environment (JRE) are installed)

```
1 $ javac -classpath . -sourcepath src -d . src/Main.java
2 $ java Main
```

Appendix B: Explanation of Configuration File Format

The format for the `.config` files is given in the example below. Note that in an actual `.config` file, the comments (starting with `#`) must be removed.

```
1 # This is a guide to show how to write a config file. In a real config file,
2 # all lines starting with '#' MUST be removed.
3 # Have a look at Appendix C to see a real config file.
4 # Each config file must start with a one-line comment:
5 This comment briefly explains the configuration.
6 # These next lines are fairly self-explanatory, they are the same as in the
7 # Simulation class.
8 epsilon:                      2      # Strength of repulsive force
9 mu:                           10     # Strength of propulsive force
10 alpha:                        700    # Strength of flocking force
11 gamma:                        600    # Strength of centripetal force
12 number_of_people:              500    # Initial number of people in the arena
13 flockRadius:                  8      # r_flock (radius in which to search for
14                                # neighbors for the flocking force)
15 dt:                            0.01   # Timestep
16 percentParticipating:          0.5    # Proportion of people initially participating
17                                # in the circle pit
18 rParticipating:                6      # Radius in which to search for participating
19                                # neighbors
```

```

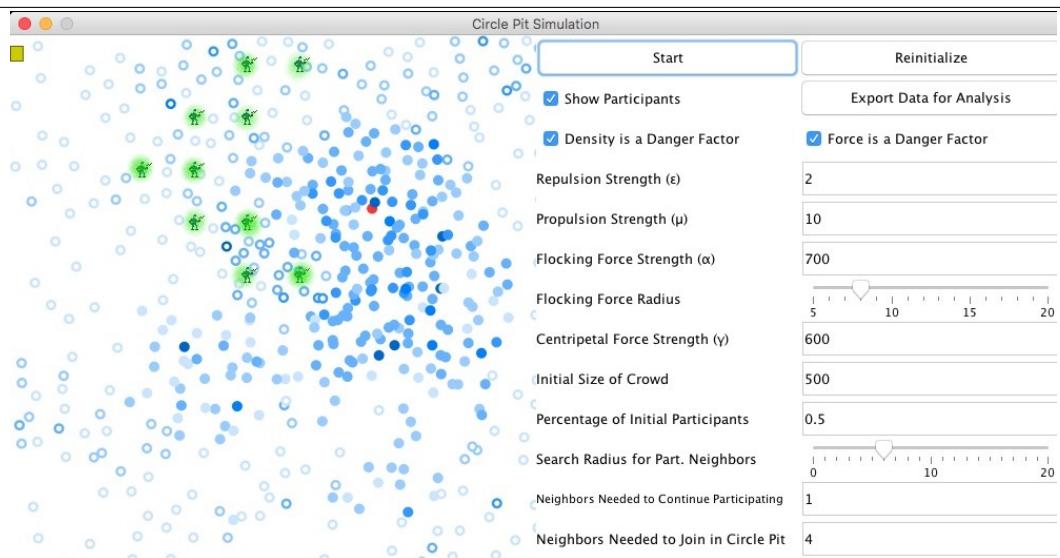
20 minCirclePitSize:           1      # Min. number of neighbors needed to continue
21                                         # participating in the circle pit
22 minParticipatingNeighbors: 3      # Min. number of neighbors needed to join in
23                                         # circle pit
24 dataCollectionInterval:    2000   # After how many milliseconds to collect data
25 insertPoliceAfter:          0      # After how many timesteps (not milliseconds!)
26                                         # police should be inserted.
27 collectDataThisManyTimes:  3      # How many times to collect data
28 useThisManySeeds:           2      # How many seeds to test
29 # The next section is a "graphical" representation of where the policemen are
30 # located. Each policeman is denoted by an 'x'. The representation below must
31 # include the borders of the matrix and should be 10 spaces wide and 10 lines
32 # long.
33 policeConfig:
34 -----
35 |   x   |
36 |   x   |
37 |   x   |
38 |   x   |
39 |   x   |
40 |   x   |
41 |   x   |
42 |   x   |
43 |   x   |
44 |   x   |
45 -----

```

Appendix C: Configuration Files Used

C.1: Arrow

```
1 A policeman configuration in the shape of a left-pointing arrow. Policemen are added after 5 se
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 numberOfWorkers: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |   xx   |
20 |   xx   |
21 |   xx   |
22 |   xx   |
23 |   xx   |
24 |
25 |
26 |
27 |
28 |
29 -----
```

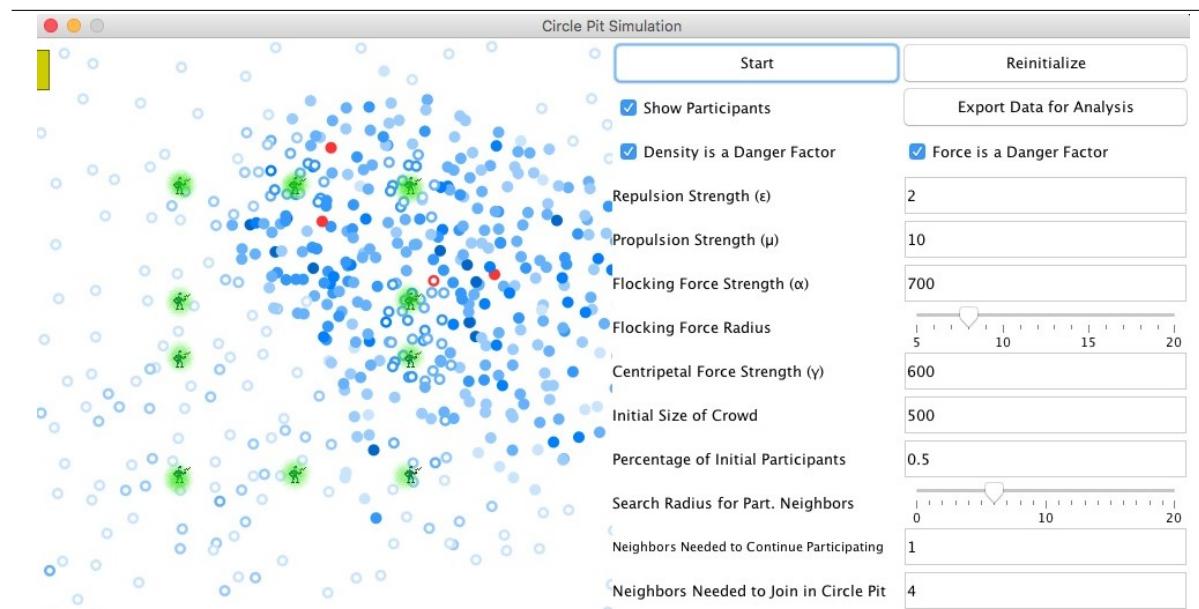


C.2: Big Circle

```

1 A "circle" of policemen larger than the 'circle' config file. Policemen are added after 5 seconds
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |   x x x |
20 |           |
21 |   x   x |
22 |           |
23 |   x   x |
24 |           |
25 |           |
26 |   x x x |
27 |           |
28 |           |
29 -----

```

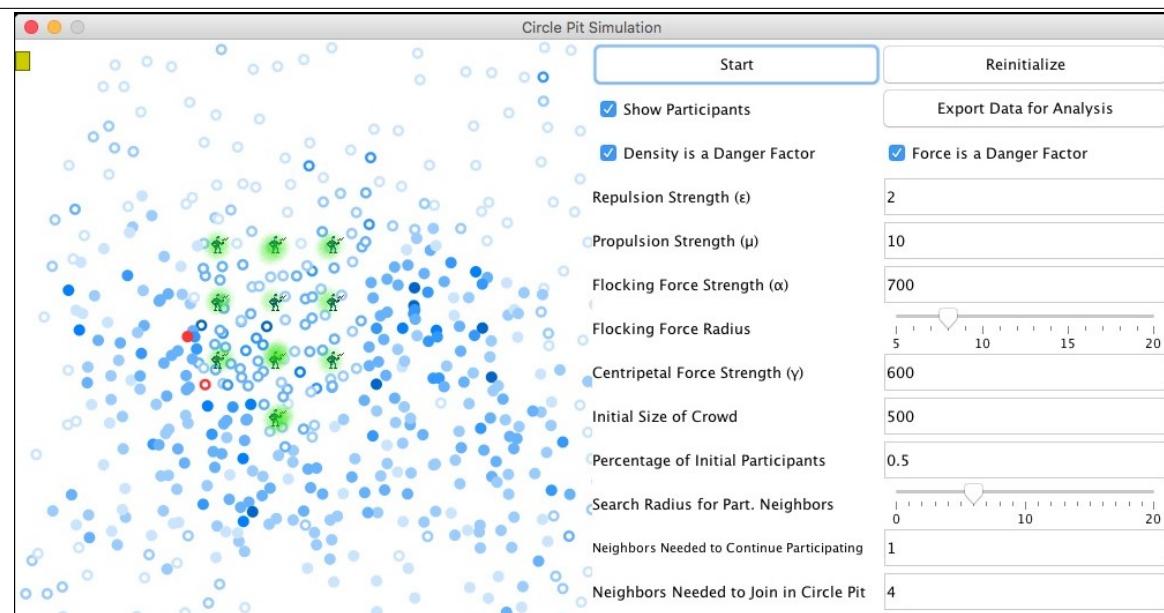


C.3: Center

```

1 A configuration with all the policemen grouped in the center of the arena. Policemen are added
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |
20 |
21 |
22 |     XXX
23 |     XXX
24 |     XXX
25 |     X
26 |
27 |
28 |
29 -----

```

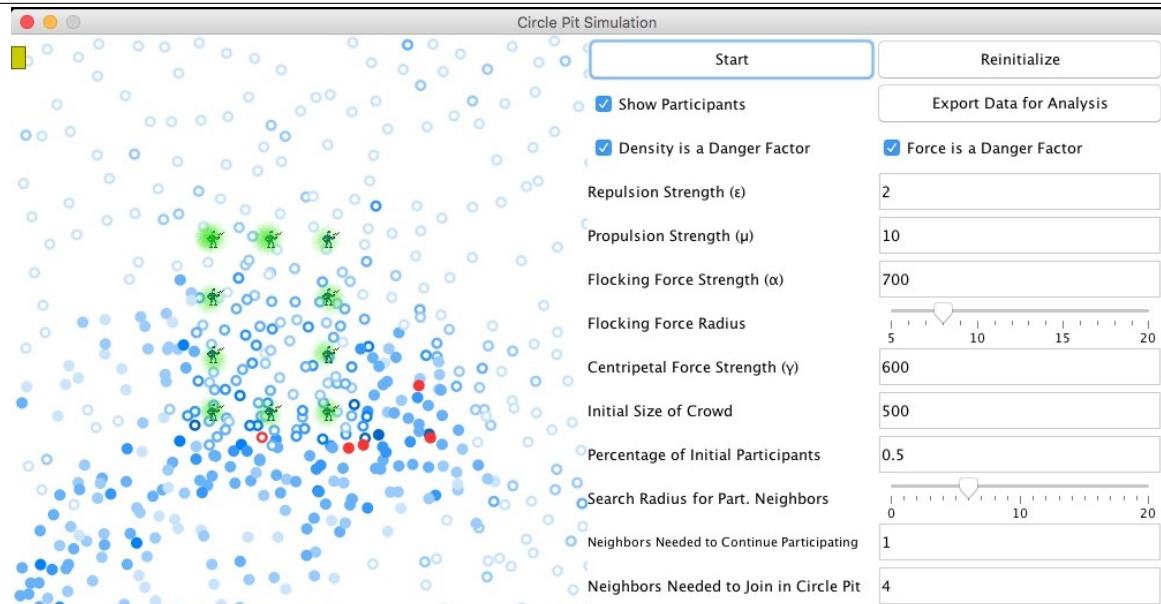


C.4: Circle

```

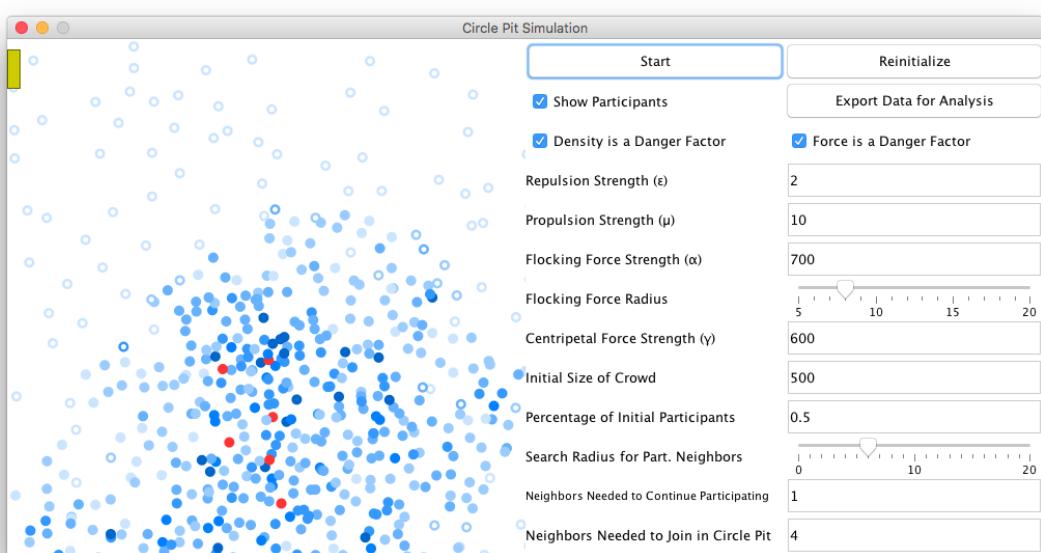
1 A "circle" of policemen in the center of the arena. Policemen are added after 5 seconds.
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |
20 |
21 |
22 |     XXX
23 |     X X
24 |     X X
25 |     XXX
26 |
27 |
28 |
29 -----

```



C.5: Control

```
1 A configuration with no policemen.  
2 epsilon: 2  
3 mu: 10  
4 alpha: 700  
5 gamma: 600  
6 number0fPeople: 500  
7 flockRadius: 8  
8 dt: 0.01  
9 percentParticipating: 0.5  
10 rParticipating: 6  
11 minCirclePitSize: 1  
12 minParticipatingNeighbors: 4  
13 dataCollectionInterval: 1000  
14 insertPoliceAfter: 100  
15 collectDataThisManyTimes: 300  
16 useThisManySeeds: 50  
17 policeConfig:  
18 -----  
19 |  
20 |  
21 |  
22 |  
23 |  
24 |  
25 |  
26 |  
27 |  
28 |  
29 -----
```

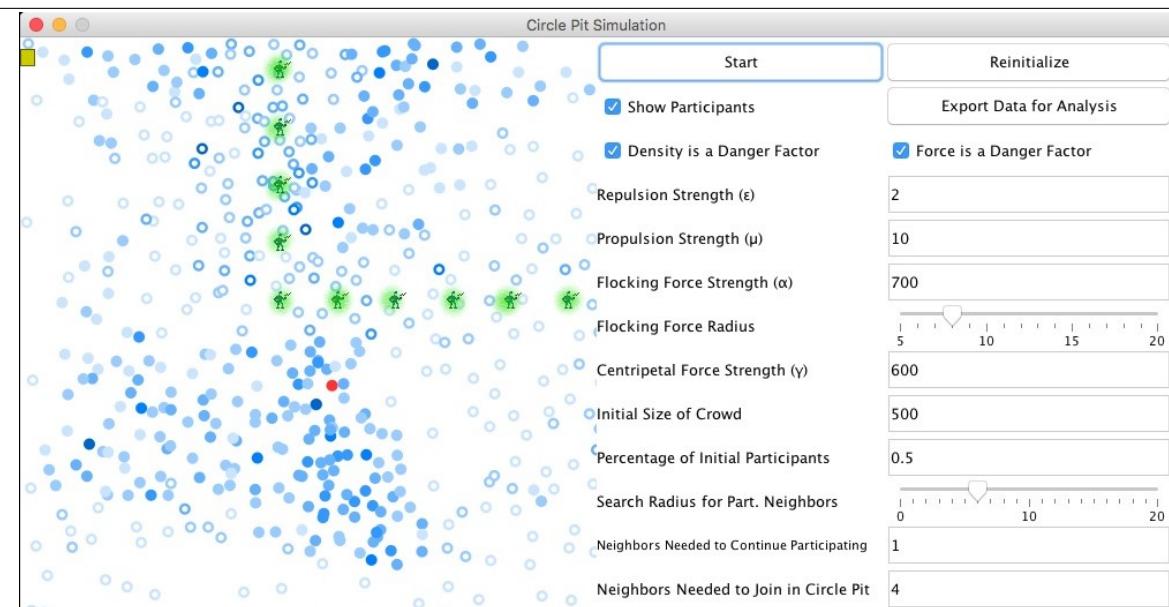


C.6: Corner

```

1 A line of policemen completely blocking off one corner of the arena. Policemen are added after
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |   X   |
20 |   X   |
21 |   X   |
22 |   X   |
23 | XXXXXX |
24 |
25 |
26 |
27 |
28 |
29 -----

```

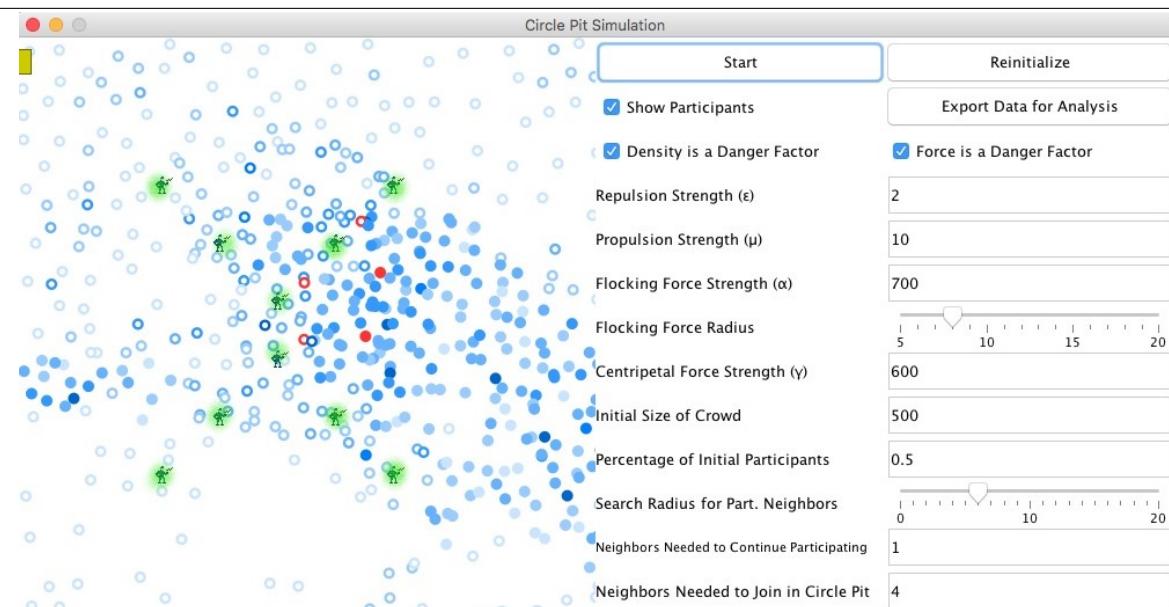


C.7: Cross

```

1 Policemen resembling a cross in the center of the arena. Policemen are added after 5 seconds.
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |   X   X |
20 |   X X   |
21 |     X   |
22 |     X   |
23 |   X X   |
24 |   X   X |
25 |           |
26 |           |
27 |           |
28 |           |
29 -----

```

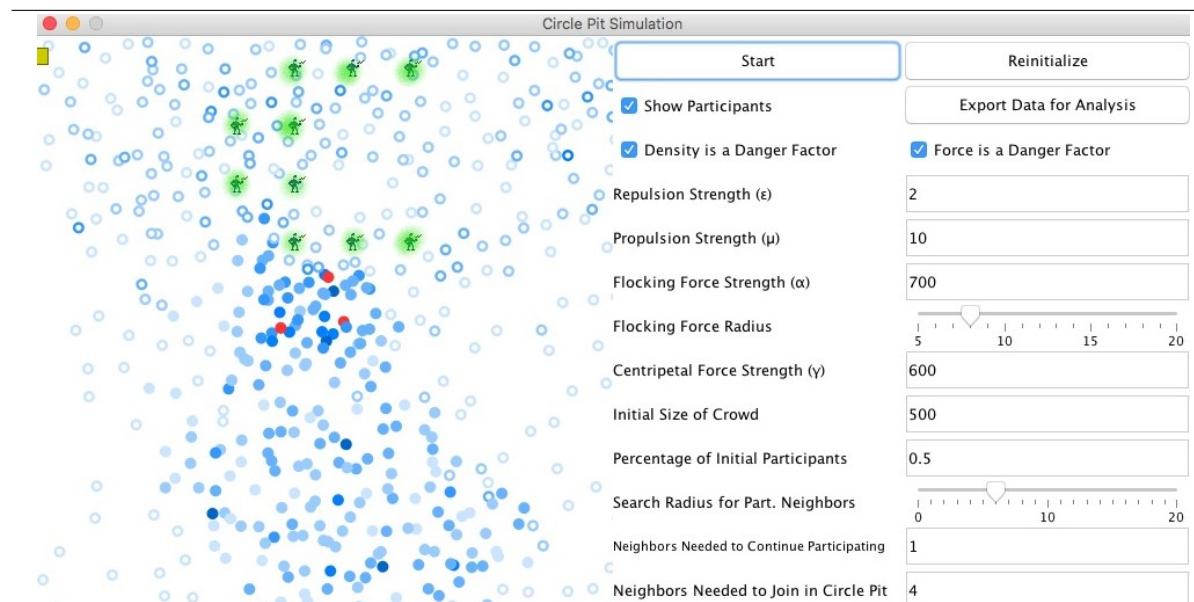


C.8: C-shape

```

1 A policeman configuration resembling a 'C'. Policemen are added after 5 seconds.
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |     xxx |
20 |     xx  |
21 |     xx  |
22 |     xxx |
23 |
24 |
25 |
26 |
27 |
28 |
29 -----

```

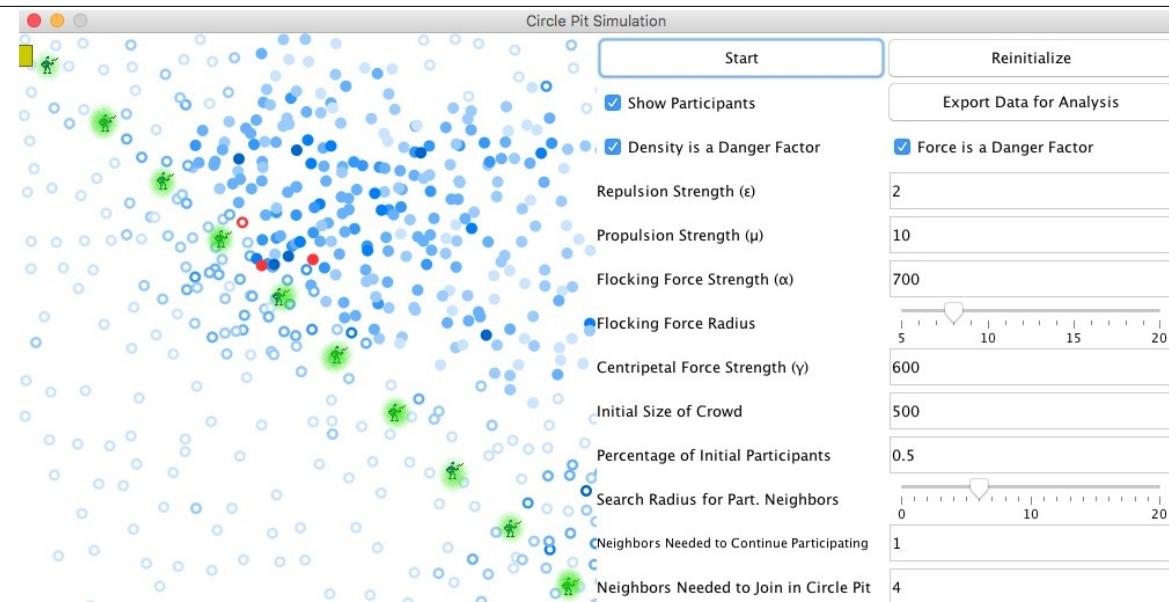


C.9: Diagonal

```

1 A diagonal line of policemen from upper left to lower right. Policemen are added after 5 seconds
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 | x
20 | x
21 | x
22 | x
23 | x
24 | x
25 | x
26 | x
27 | x
28 | x |
29 -----

```

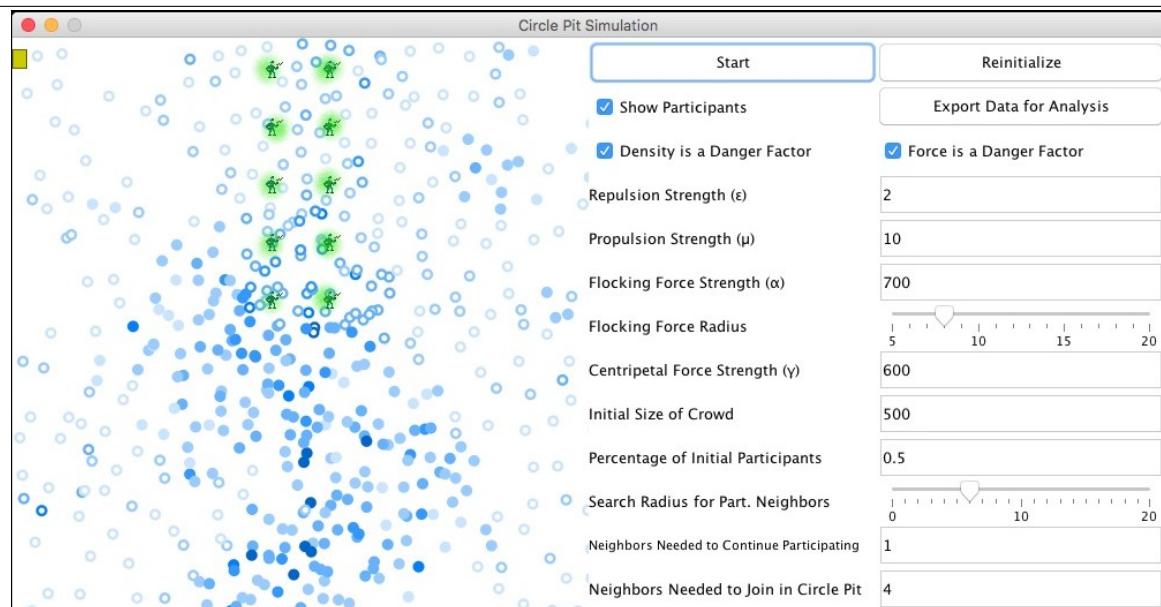


C.10: Double Half-Line

```

1 A double half-line of policemen at the top center of the arena. Policemen are added after 5 sec
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 number0fPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |   XX   |
20 |   XX   |
21 |   XX   |
22 |   XX   |
23 |   XX   |
24 |
25 |
26 |
27 |
28 |
29 -----

```

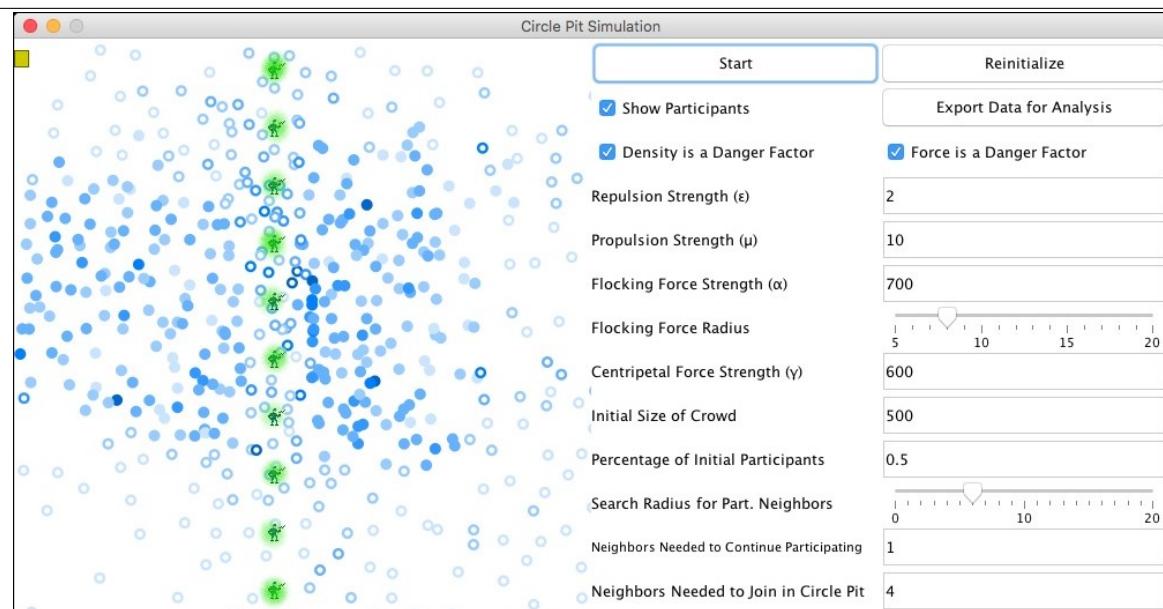


C.11: Vertical Line

```

1 A vertical line of policemen in the center of the arena. Policemen are added after 5 seconds.
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 numberOfPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 | X |
20 | X |
21 | X |
22 | X |
23 | X |
24 | X |
25 | X |
26 | X |
27 | X |
28 | X |
29 -----

```

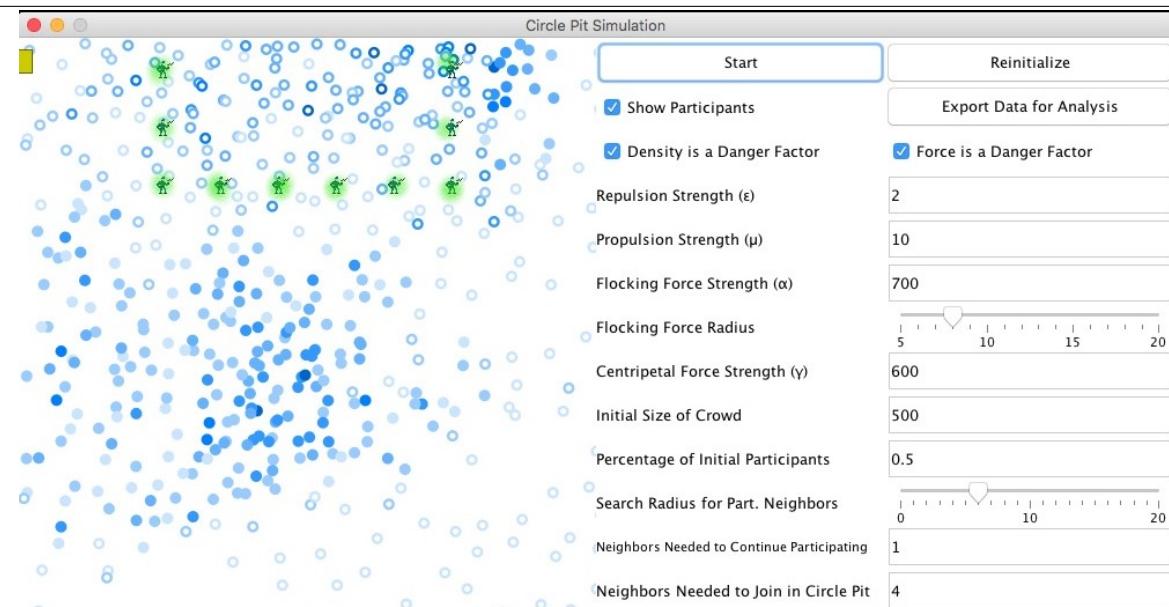


C.12: U-Shape

```

1 A configuration of police officers resembling a 'U' at the top center of the arena. Policemen a
2 epsilon: 2
3 mu: 10
4 alpha: 700
5 gamma: 600
6 numberOfPeople: 500
7 flockRadius: 8
8 dt: 0.01
9 percentParticipating: 0.5
10 rParticipating: 6
11 minCirclePitSize: 1
12 minParticipatingNeighbors: 4
13 dataCollectionInterval: 1000
14 insertPoliceAfter: 100
15 collectDataThisManyTimes: 300
16 useThisManySeeds: 50
17 policeConfig:
18 -----
19 |   x   x   |
20 |   x   x   |
21 | xxxxxx |
22 |
23 |
24 |
25 |
26 |
27 |
28 |
29 -----

```



Appendix D: Phase Diagrams

The following appendix shows phase diagrams of where the danger is located at two time points, one time point right after the point when the police were inserted and one time point after the circle pit had been mostly stopped.

D.1: Arrow

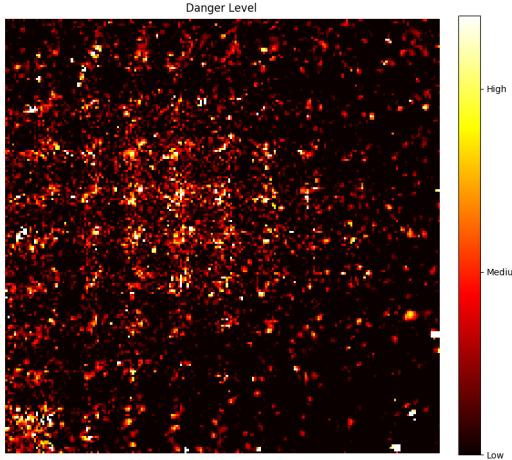


Figure 27: *Arrow's configuration* phase diagram after fifteen time steps, global danger level has already decreased significantly, with a peak in the first quadrant – where the inner wings of the guards stays.

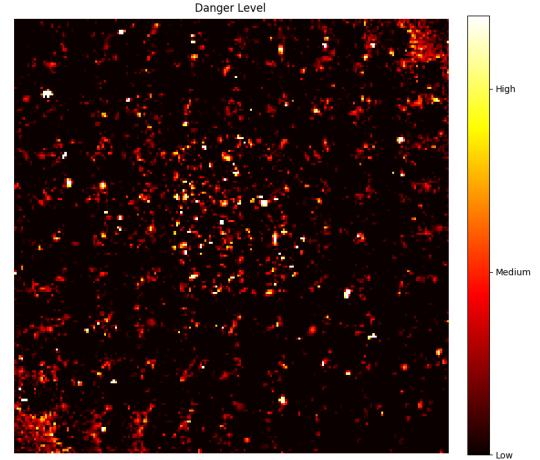


Figure 28: Phase diagram shot after 110 steps, almost a third compared to the second shot of other configurations as *arrow's arrangement* already killed the circle pit by the time.

D.2: Big Circle

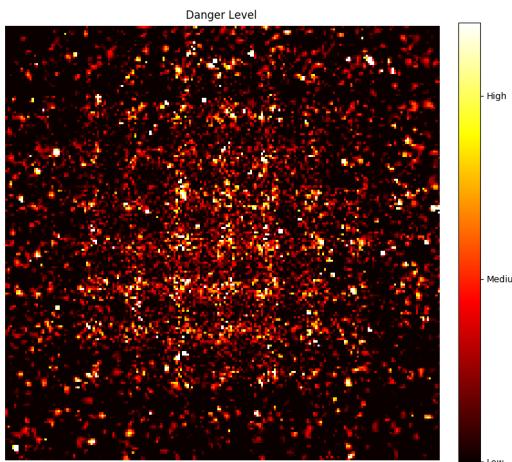


Figure 29: *Big Circle's configuration* phase diagram after ten time steps. It is possible to distinguish a slight darker circumference: the perimeter created by the guards.

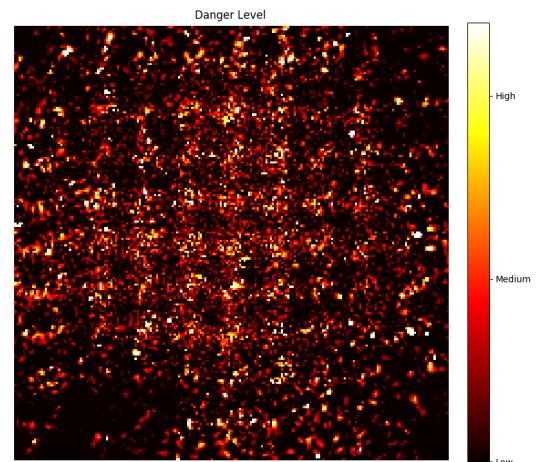


Figure 30: Phase diagram taken at the 290th time step, danger level is still high all above the plane.

D.3: Center

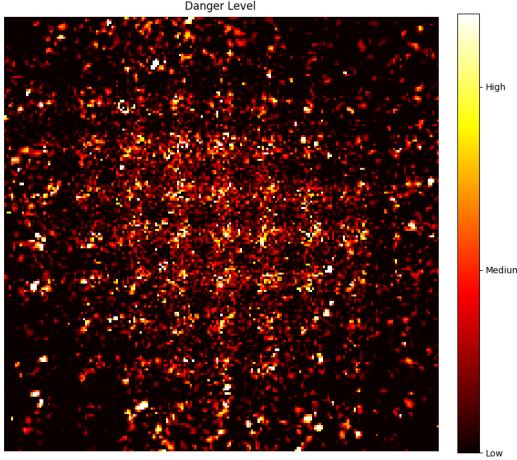


Figure 31: Phase diagram shot after ten steps w.r.t. the *center's arrangement* of guards. Danger is concentrated in the middle of the diagram; people – attracted there by the centripetal force – stop as they reach the police officers, producing a *high-density zone*.

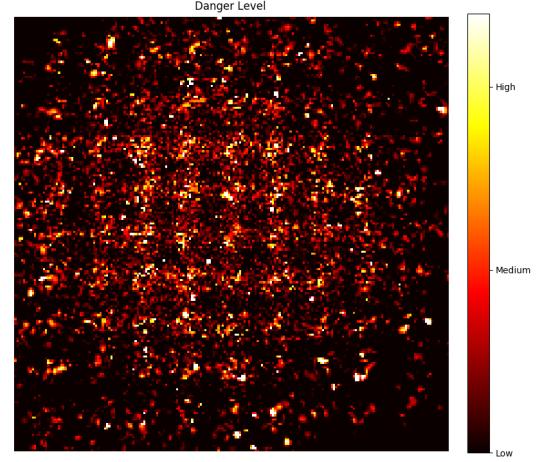


Figure 32: Image of the phase diagram took after 200 time steps: the pit is still spinning.

D.4: Circle

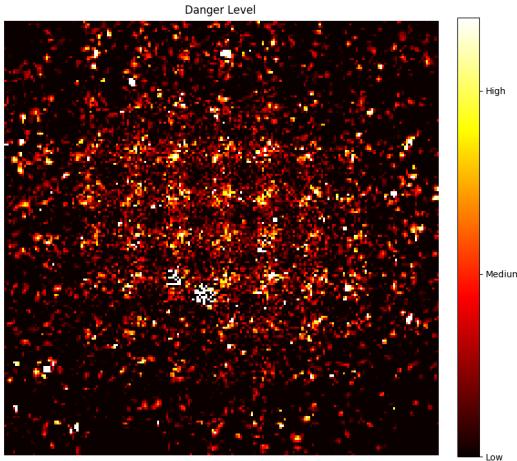


Figure 33: *Circle's disposition* of the guards phase diagram, after ten steps. As this configuration is similar to the previous two, the diagram resemble theirs: higher danger levels concentrated in the center.

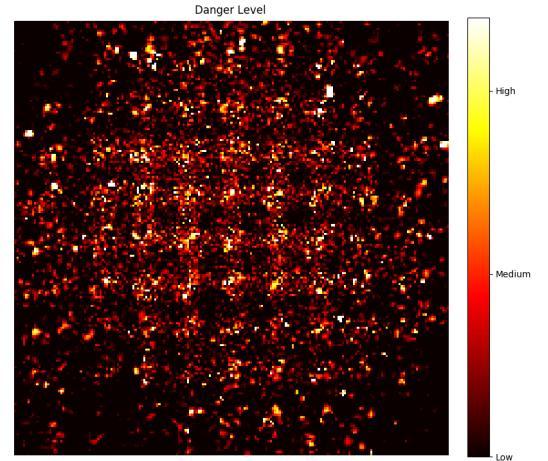


Figure 34: Phase diagram at the two hundred ninetieth step: as for the preceding diagrams yellow is still painting the plan, meaning people didn't stop spinning.

D.5: Corner

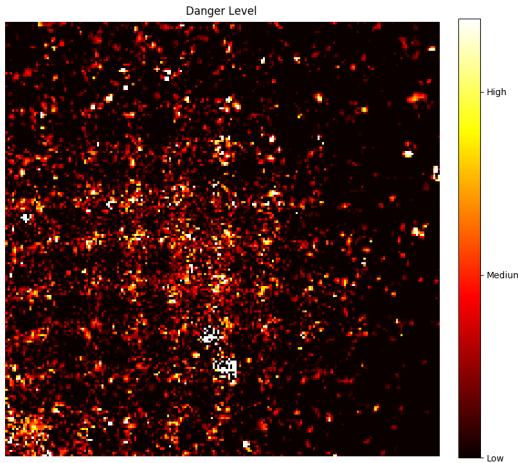


Figure 35: Phase diagram of the *corner's configuration* at the tenth time step. Danger has already strongly decreased, but for a curious spot in the middle of the lower part of the plane.

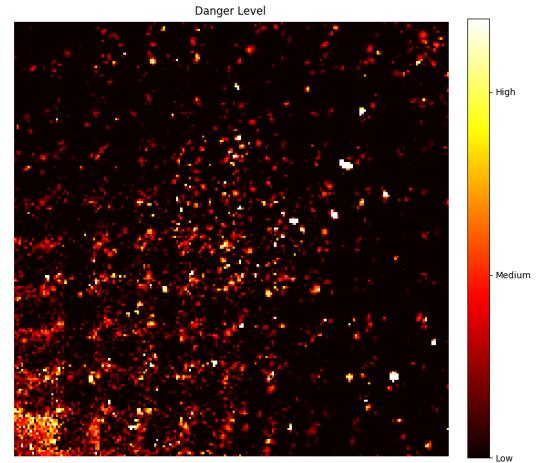


Figure 36: Phase diagram after thirty time steps, circle pit is almost dead by now. *Corner's arrangement* of police officers seems to be the most effective among the eleven we tried.

D.6: Control

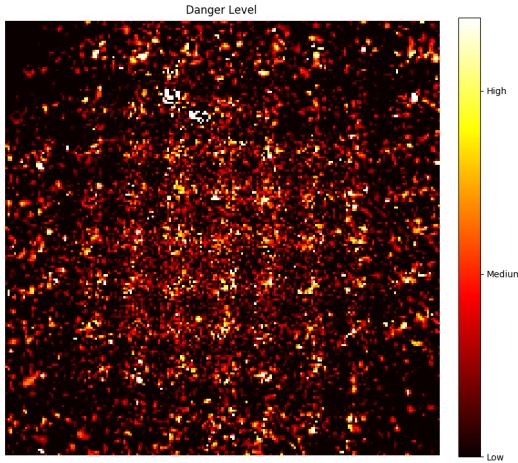


Figure 37: Phase diagram of a simulation run without *guards* at the most common time step w.r.t. the other phase diagrams, namely ten. Danger is overall high.

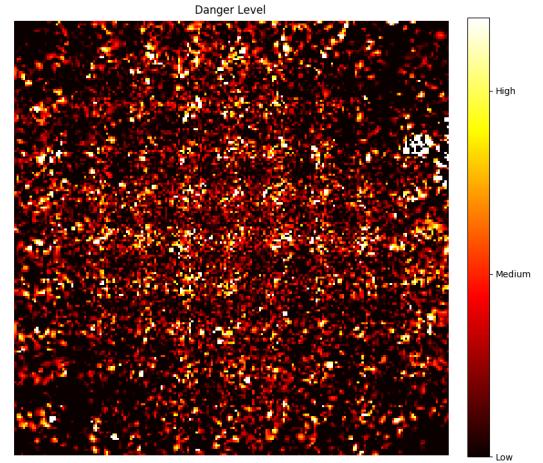


Figure 38: Phase diagram after two hundred ninety steps. Without guards, people keep running.

D.7: Cross

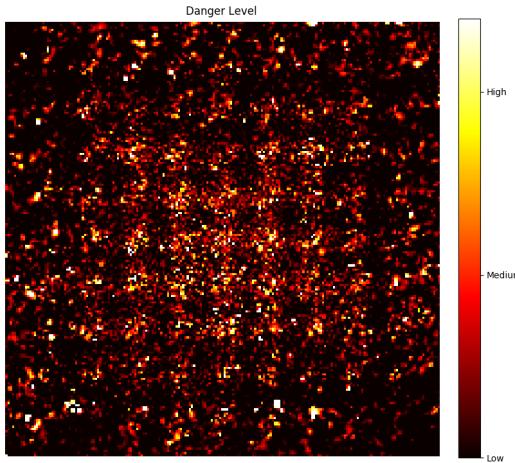


Figure 39: *emphCross'* profile depicted in a phase diagram at the twelfth time step. Global danger level is under the media and it is well distributed overall the space.

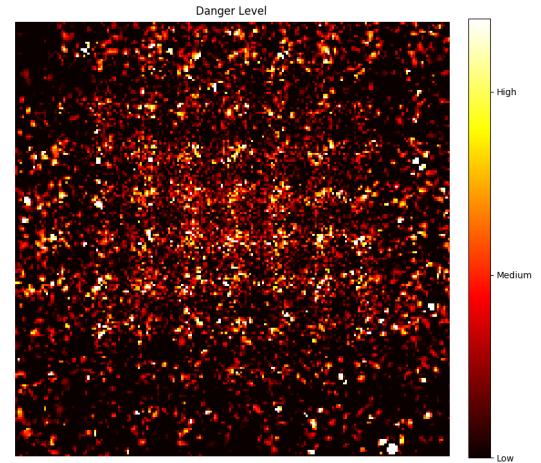


Figure 40: *Cross'* profile depicted in a phase diagram at the twelfth time step. Global danger level is under the media and it is well distributed overall the space

D.8: C-Shape

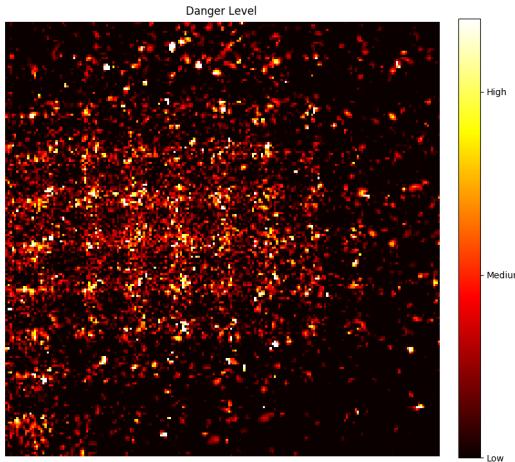


Figure 41: *C-shape configuration* phase diagram after nine time steps. Comparing it to the *arrow configuration* – which a similar shape has – danger is more homogeneously allocated. Moreover it has no danger concentration over the police officers.

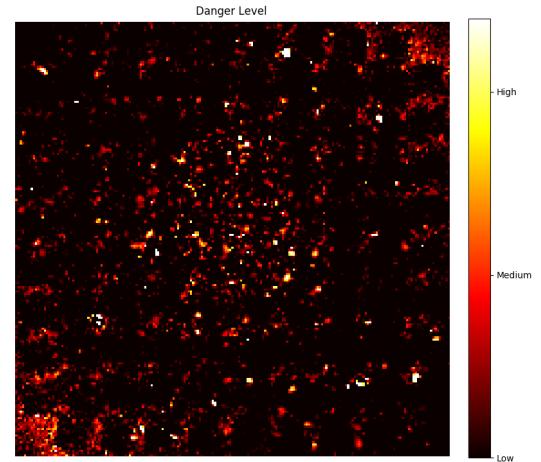


Figure 42: Phase diagram at hundred fifty time steps. *C-shape* arrangement has a good effectiveness, it has indeed by the time stopped the circle pit.

D.9: Diagonal

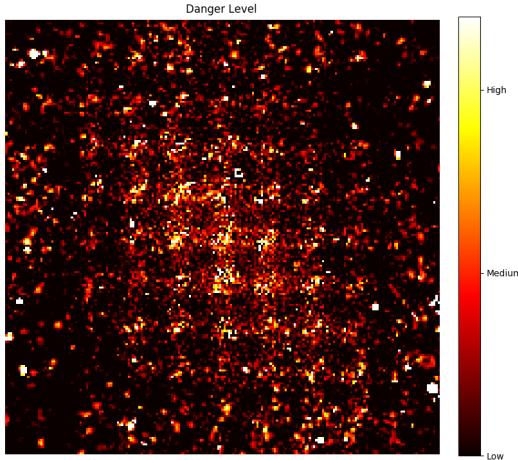


Figure 43: Phase diagram of *diagonal configuration* after six time step. Here is visible the yellow-red shape exactly over the position of the *guards*

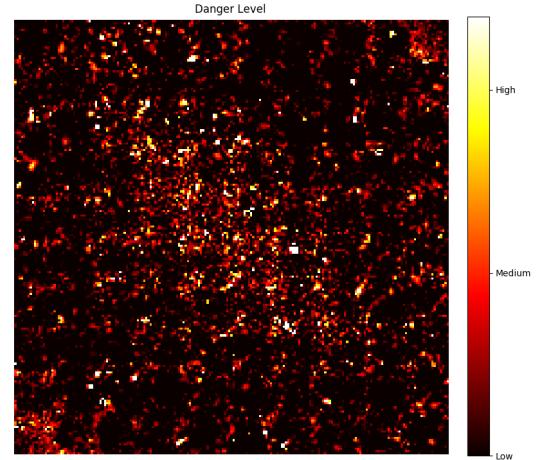


Figure 44: Shot of phase diagram at time step two hundred and ninety. The “shadow” above the police officers is still visible. People are spinning less, but the pit keeps living.

D.10: Double Half-Line

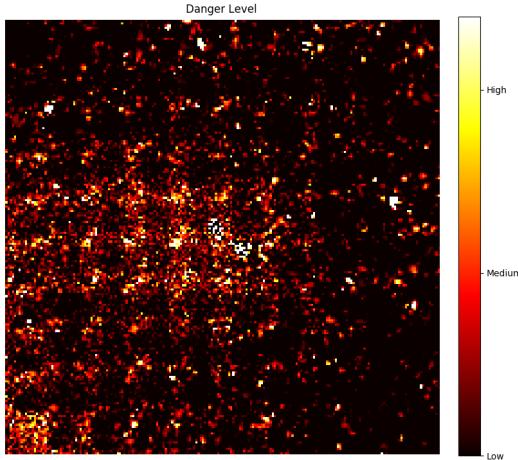


Figure 45: Phase diagram after fifteen time steps, danger level decreases rapidly. *Double Half-Line* is a relatively efficient configuration. Its effect is similar to the *C-shape* configuration.

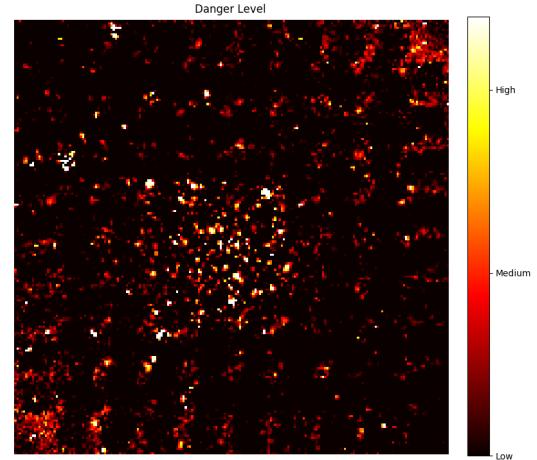


Figure 46: Phase diagram after one hundred thirty time steps, circle pit is almost dead by now. *Double Half-Line* is a relatively efficient configuration. Its effect is similar to the *C-shape* configuration.

D.11: U-Shape

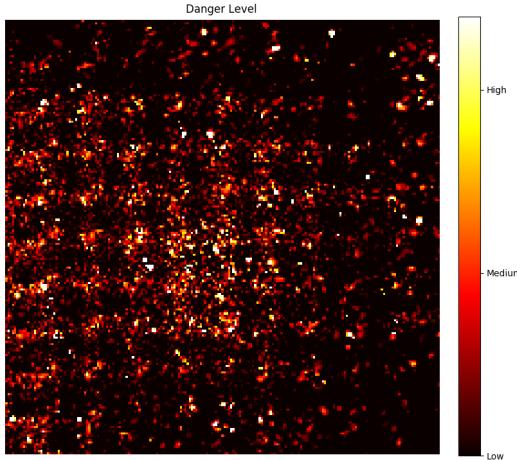


Figure 47: Phase diagram after ten time steps, the degree of danger declines very fast. *U-shape* is a very efficient configuration.

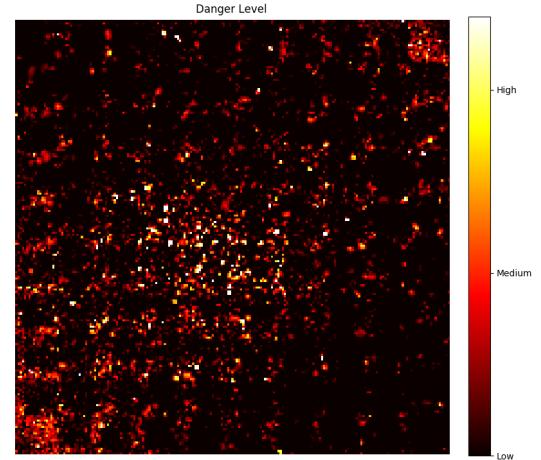


Figure 48: Phase diagram at time step two hundred ninety. People have almost stopped to spin: not the most efficient arrangement of *guards*, but it slowly slowed down everybody in the pit.

D.12: Vertical Line

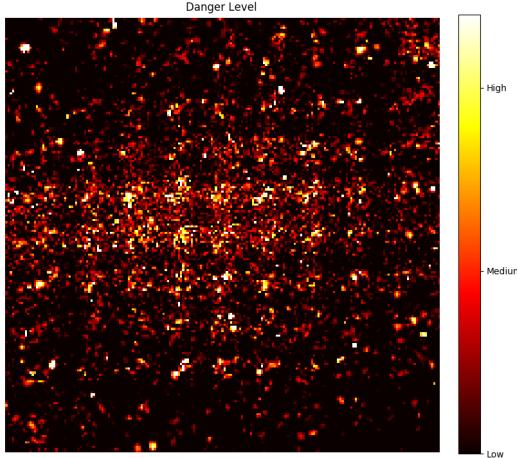


Figure 49: Phase diagram w.r.t. *vertical line configuration* at the tenth time step. Opposite to diagonal line, higher danger level seems to arrange himself on a line perpendicular to the line of police officers.

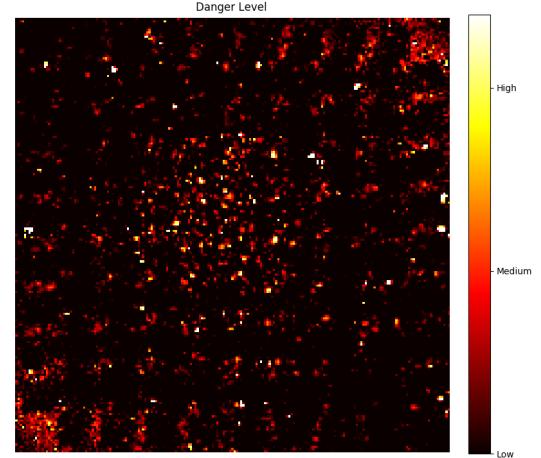


Figure 50: Last phase diagram, shot after two hundred eighty steps. As for the previous arrangement of *guards* circle pit has stopped, but late in time.

Appendix E: Java Code

E.1: Main Code

```
1 import java.io.FileNotFoundException;
2
3 public class Main {
4     public static void main(String[] args) throws FileNotFoundException {
5         // Run a manual simulation
6         // Simulation simulation = new Simulation(2, 10, 700, 600, 500, 8,
7         //                                         0.01, 0.5, 6, 1, 3);
8         // simulation.runManualSimulation();
9
10        // Run an automatic simulation
11        AutomaticSimulator test = new AutomaticSimulator();
12        test.run();
13    }
14 }
```

E.2: Automatic Simulation

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Arrays;
4 import java.util.LinkedList;
5 import java.util.Queue;
6 import java.util.Scanner;
7
8 /**
9  * An object that reads the configs given in <code>configs.sim</code> and
10 * executes the corresponding simulations while dumping data for analysis.
11 */
12 public class AutomaticSimulator {
13
14     private Queue<String> configNames = new LinkedList<String>();
15         // The configs to simulate
16     private Simulation simulation;
17
18     public AutomaticSimulator() throws FileNotFoundException {
19         readConfigNames();
20     }
21
22 /**
23 * Runs the automatic simulation.
24 *
25 * @throws FileNotFoundException either if <code>configs.sim</code> could
26 *                                 not be found or if a configuration
27 *                                 specified in <code>configs.sim</code> could
28 *                                 not be found.
29 */
30     public void run() throws FileNotFoundException {
31         // Keep going while not all configurations have been tested
32         while (!configNames.isEmpty()) {
33             String config = configNames.poll();
34             Config c = readConfigFile(config);
35             newSimulation(c);
36             // Poll the simulation to see if it is done yet (really ugly but
37             // oh well...)
38             while (!simulation.isCurrentIterationFinished) {
39                 try {
40                     Thread.sleep(1000);
41                 } catch (InterruptedException e) {
42                     e.printStackTrace();
43                 }
44             }
45             simulation.isCurrentIterationFinished = false;
46         }
47         System.exit(0);
48     }
49
50 // Reads the configurations to simulate from configs.sim
```

```

51 private void readConfigNames() throws FileNotFoundException {
52     Scanner scanner = new Scanner(new File("configs.sim"));
53     scanner.nextLine();
54     while (scanner.hasNext()) {
55         configNames.add(scanner.next());
56     }
57     System.out.println("Starting simulation with configs: " + configNames);
58 }
59
60 // Reads a configuration file
61 private Config readConfigFile(String config) throws FileNotFoundException {
62     Scanner scanner =
63         new Scanner(new File("configs/" + config + ".config"));
64     scanner.nextLine(); // Skip comment on first line
65     double[] data = new double[15];
66     Config c = new Config();
67     c.name = config;
68     c.name = config;
69     for (int i = 0; i < data.length;
70          i++) { // Read parameters from config file
71         scanner.next();
72         data[i] = scanner.nextDouble();
73     }
74     c.readNumbersFromArray(data);
75     scanner.nextLine(); // Move to line after all the number values
76     scanner.nextLine(); // Move over line with "policeSectors: "
77     scanner.nextLine(); // Move over upper boundary of ASCII-based
78                     // representation of matrix
79     for (int i = 0; i < 10; i++) {
80         String line = scanner.nextLine();
81         for (int j = 0; j < 10; j++) {
82             if (line.charAt(j + 1) ==
83                 'x') { // j+1 because of leading |, 'x' means there
84                 // is a policeman there
85                 c.policeSectors[j][i] = true;
86             }
87         }
88     }
89     System.out.println("Current config: " + config);
90     System.out.println("    data = " + Arrays.toString(data));
91     System.out.println("    policeSectors = ");
92     for (int i = 0; i < c.policeSectors.length; i++) {
93         System.out.println("        " + Arrays.toString(c.policeSectors[i]));
94     }
95     return c;
96 }
97
98 private void newSimulation(Config c) {
99     if (simulation == null) {
100         // Initialize the simulation with data from the config file
101         simulation = new Simulation(c.epsilon, c.mu, c.alpha, c.gamma,
102                                     c.numberOfPeople, c.flockRadius, c.dt,

```

```

102             c.percentParticipating,
103             c.rParticipating, c.minCirclePitSize,
104             c.minParticipatingNeighbors);
105         simulation.createWindow();
106         simulation.createNewTimer(new DataCollector(simulation, c.name,
107                                         c.dataCollectionInterval,
108                                         c.insertPoliceAfter,
109                                         c.amountOfSeeds,
110                                         c.collectionTimes,
111                                         c.policeSectors));
112         simulation.start();
113     } else {
114         // Reinitialize all the parameters with data from the config file
115         simulation.stop();
116         simulation.epsilon = c.epsilon;
117         simulation.alpha = c.alpha;
118         simulation.mu = c.mu;
119         simulation.gamma = c.gamma;
120         simulation.numberOfPeople = c.numberOfPeople;
121         simulation.flockRadius = c.flockRadius;
122         simulation.dt = c.dt;
123         simulation.percentParticipating = c.percentParticipating;
124         simulation.rParticipating = c.rParticipating;
125         simulation.minCirclePitSize = c.minCirclePitSize;
126         simulation.minParticipatingNeighbors = c.minParticipatingNeighbors;
127         simulation.createNewTimer(new DataCollector(simulation, c.name,
128                                         c.dataCollectionInterval,
129                                         c.insertPoliceAfter,
130                                         c.amountOfSeeds,
131                                         c.collectionTimes,
132                                         c.policeSectors));
133         simulation.basicReset();
134         simulation.start();
135     }
136 }
137
138 // A class that stores all the parameters from a config file
139 private class Config {
140     String name;
141     double epsilon, alpha, mu, gamma, dt, percentParticipating;
142     int numberOfPeople, flockRadius, rParticipating, minCirclePitSize,
143         minParticipatingNeighbors, dataCollectionInterval,
144         insertPoliceAfter, collectionTimes, amountOfSeeds;
145     boolean[][] policeSectors = new boolean[10][10];
146
147     private void readNumbersFromArray(double[] array) {
148         assert array.length == 15;
149         epsilon = array[0];
150         mu = array[1];
151         alpha = array[2];
152         gamma = array[3];

```

```
153     numberOfPeople = (int) array[4];
154     flockRadius = (int) array[5];
155     dt = array[6];
156     percentParticipating = array[7];
157     rParticipating = (int) array[8];
158     minCirclePitSize = (int) array[9];
159     minParticipatingNeighbors = (int) array[10];
160     dataCollectionInterval = (int) array[11];
161     insertPoliceAfter = (int) array[12];
162     collectionTimes = (int) array[13];
163     amountOfSeeds = (int) array[14];
164 }
165 }
166 }
```

E.3: Control Panel

```
1 import javax.swing.*;
2 import javax.swing.event.ChangeEvent;
3 import javax.swing.event.ChangeListener;
4 import java.awt.*;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7 import java.beans.PropertyChangeEvent;
8 import java.beans.PropertyChangeListener;
9
10 public class ControlPanel extends JPanel
11     implements PropertyChangeListener, ChangeListener {
12
13     private Simulation simulation;
14     private SimulationPanel simulationpanel;
15     private JButton startPauseButton;
16     private JButton restartButton;
17     private JButton exportDataButton;
18     private JCheckBox enableDensity;
19     private JCheckBox enableForce;
20     private JCheckBox showParticipants;
21     private JLabel epsilonLabel, muLabel, alphaLabel, flockRadiusLabel,
22         gammaLabel, initialParticipantsLabel, percLabel, rPartLabel,
23         sizeLabel, minNeighborsLabel;
24     private JFormattedTextField epsilonField, muField, alphaField, gammaField,
25         initialParticipantsField, percField, sizeField, minNeighborsField;
26     private JSlider rPartSlider, flockRadiusSlider;
27
28     private boolean paused = true;
29
30     public ControlPanel(Simulation simulation,
31                         SimulationPanel simulationpanel) {
32         this.simulation = simulation;
33         this.simulationpanel = simulationpanel;
34         setPreferredSize(new Dimension(500, 230));
35         setBackground(Color.WHITE);
36         setLayout(new GridLayout(13, 2));
37
38         initComponents();
39         addListeners();
40         addComponents();
41     }
42
43     private void initComponents() {
44         startPauseButton = new JButton("Start");
45         restartButton = new JButton("Reinitialize");
46         showParticipants = new JCheckBox("Show Participants");
47         showParticipants.setSelected(true);
48         exportDataButton = new JButton("Export Data for Analysis");
49
50         enableDensity = new JCheckBox("Density is a Danger Factor");
```

```

51    enableDensity.setSelected(true);
52    enableForce = new JCheckBox("Force is a Danger Factor");
53    enableForce.setSelected(true);
54
55    epsilonLabel = new JLabel("Repulsion Strength (\u03b5)");
56
57    epsilonField = new JFormattedTextField();
58    epsilonField.setValue(simulation.epsilon);
59    epsilonField.setColumns(10);
59
60    muLabel = new JLabel("Propulsion Strength (\u03bc)");
61
62    muField = new JFormattedTextField();
63    muField.setValue(simulation.mu);
64    muField.setColumns(10);
65
66    alphaLabel = new JLabel("Flocking Force Strength (\u03b1)");
67
68    alphaField = new JFormattedTextField();
69    alphaField.setValue(simulation.alpha);
70    alphaField.setColumns(10);
71
72    flockRadiusLabel = new JLabel("Flocking Force Radius");
73    flockRadiusSlider = new JSlider();
74    flockRadiusSlider.setValue((int) simulation.flockRadius);
75    configureSlider(flockRadiusSlider, 1, 5, 5, 2 * Simulation.SECTOR_SIZE);
76
77    gammaLabel = new JLabel("Centripetal Force Strength (\u03b3)");
78
79    gammaField = new JFormattedTextField();
80    gammaField.setValue(simulation.gamma);
81    gammaField.setColumns(10);
82
83    initialParticipantsLabel = new JLabel("Initial Size of Crowd");
84
85    initialParticipantsField = new JFormattedTextField();
86    initialParticipantsField.setValue(simulation.numberOfPeople);
87    initialParticipantsField.setColumns(10);
88
89    percLabel = new JLabel("Percentage of Initial Participants");
90
91    percField = new JFormattedTextField();
92    percField.setValue(simulation.percentParticipating);
93    percField.setColumns(10);
94
95    rPartLabel = new JLabel("Search Radius for Part. Neighbors");
96
97    rPartSlider = new JSlider();
98    rPartSlider.setValue((int) simulation.rParticipating);
99    configureSlider(rPartSlider, 1, 10, 0, 2 * Simulation.SECTOR_SIZE);
100
101

```

```

102     sizeLabel = new JLabel("Neighbors Needed to Continue Participating");
103
104     String fontFamily = sizeLabel.getFont().getFamily();
105     sizeLabel.setFont(new Font(fontFamily, Font.PLAIN, 11));
106
107
108     sizeField = new JFormattedTextField();
109     sizeField.setValue(simulation.minCirclePitSize);
110     sizeField.setColumns(10);
111
112     minNeighborsLabel = new JLabel(
113         "Neighbors Needed to Join in Circle Pit");
114
115     minNeighborsField = new JFormattedTextField();
116     minNeighborsField.setValue(simulation.minParticipatingNeighbors);
117     minNeighborsField.setColumns(10);
118
119 }
120
121 private void configureSlider(JSlider slider, int minorSpacing,
122                             int majorSpacing, int lowerLimit,
123                             int upperLimit) {
124     slider.setMajorTickSpacing(majorSpacing);
125     slider.setMinorTickSpacing(minorSpacing);
126     slider.setMinimum(lowerLimit);
127     slider.setMaximum(upperLimit);
128     slider.setPaintLabels(true);
129     slider.setPaintTicks(true);
130 }
131
132 private void addListeners() {
133     startPauseButton.addActionListener(new ActionListener() {
134         /* @Override */
135         public void actionPerformed(ActionEvent e) {
136             if (paused) {
137                 simulation.getSimulationTimer().start();
138                 startPauseButton.setText("Pause");
139             } else {
140                 simulation.getSimulationTimer().stop();
141                 startPauseButton.setText("Start");
142             }
143             paused = !paused;
144         }
145     });
146
147     restartButton.addActionListener(new ActionListener() {
148         /* @Override */
149         public void actionPerformed(ActionEvent e) {
150             startPauseButton.setText("Start");
151             paused = true;
152             simulation.resetMatrix();

```

```

153     }
154 );
155
156     showParticipants.addActionListener(new ActionListener() {
157         /* @Override */
158         public void actionPerformed(ActionEvent e) {
159             simulationpanel.shouldShowParticipants =
160                 !simulationpanel.shouldShowParticipants;
161             getParent().repaint();
162         }
163     });
164
165     exportDataButton.addActionListener(new ActionListener() {
166         /* @Override */
167         public void actionPerformed(ActionEvent e) {
168             simulation.exportData();
169         }
170     });
171
172     enableDensity.addActionListener(new ActionListener() {
173         /* @Override */
174         public void actionPerformed(ActionEvent e) {
175             if (!simulation.enableDensity) {
176                 simulation.enableDensity = true;
177             } else {
178                 simulation.enableDensity = false;
179                 for (Individual individual : simulation.getMatrix()
180                     .getIndividuals()) {
181                     individual.dangerLevel = 0;
182                 }
183             }
184         }
185     });
186
187     enableForce.addActionListener(new ActionListener() {
188         /* @Override */
189         public void actionPerformed(ActionEvent e) {
190             if (!simulation.enableForce) {
191                 simulation.enableForce = true;
192             } else {
193                 simulation.enableForce = false;
194                 for (Individual individual : simulation.getMatrix()
195                     .getIndividuals()) {
196                     individual.dangerLevel = 0;
197                 }
198             }
199         }
200     });
201
202     epsilonField.addPropertyChangeListener("value", this);
203     alphaField.addPropertyChangeListener("value", this);

```

```

204     flockRadiusSlider.addChangeListener(this);
205     muField.addPropertyChangeListener("value", this);
206     gammaField.addPropertyChangeListener("value", this);
207     initialParticipantsField.addPropertyChangeListener("value", this);
208     percField.addPropertyChangeListener("value", this);
209     rPartSlider.addChangeListener(this);
210     sizeField.addPropertyChangeListener("value", this);
211     minNeighborsField.addPropertyChangeListener("value", this);
212 }
213
214 private void addComponents() {
215     add(startPauseButton);
216     add(restartButton);
217     add(showParticipants);
218     add(exportDataButton);
219     add(enableDensity);
220     add(enableForce);
221     add(epsilonLabel);
222     add(epsilonField);
223     add(muLabel);
224     add(muField);
225     add(alphaLabel);
226     add(alphaField);
227     add(flockRadiusLabel);
228     add(flockRadiusSlider);
229     add(gammaLabel);
230     add(gammaField);
231     add(initialParticipantsLabel);
232     add(initialParticipantsField);
233     add(percLabel);
234     add(percField);
235     add(rPartLabel);
236     add(rPartSlider);
237     add(sizeLabel);
238     add(sizeField);
239     add(minNeighborsLabel);
240     add(minNeighborsField);
241
242     setVisible(true);
243 }
244
245 /**
246  * Called when a field's "value" property changes.
247  */
248 /* @Override */
249 public void propertyChange(PropertyChangeEvent e) {
250     Object source = e.getSource();
251     if (source == epsilonField) {
252         simulation.epsilon =
253             ((Number) epsilonField.getValue()).doubleValue();
254     } else if (source == muField) {

```

```

255     simulation.mu = ((Number) muField.getValue()).doubleValue();
256 } else if (source == alphaField) {
257     simulation.alpha = ((Number) alphaField.getValue()).doubleValue();
258 } else if (source == gammaField) {
259     simulation.gamma = ((Number) gammaField.getValue()).doubleValue();
260 } else if (source == initialParticipantsField) {
261     simulation.numberOfPeople =
262         ((Number) initialParticipantsField.getValue()).intValue();
263 } else if (source == percField) {
264     simulation.percentParticipating =
265         ((Number) percField.getValue()).doubleValue();
266 } else if (source == rPartSlider) {
267     simulation.rParticipating =
268         ((Number) rPartSlider.getValue()).doubleValue();
269 } else if (source == sizeField) {
270     simulation.minCirclePitSize =
271         ((Number) sizeField.getValue()).intValue();
272 } else if (source == minNeighborsField) {
273     simulation.minParticipatingNeighbors =
274         ((Number) minNeighborsField.getValue()).intValue();
275 }
276 }
277
278 /* @Override */
279 public void stateChanged(ChangeEvent e) {
280     JSlider source = (JSlider) e.getSource();
281     if (source == rPartSlider && !source.getValueIsAdjusting()) {
282         simulation.rParticipating = (double) rPartSlider.getValue();
283     } else if (source == flockRadiusSlider &&
284                 !source.getValueIsAdjusting()) {
285         simulation.flockRadius = (double) flockRadiusSlider.getValue();
286     }
287 }
288 }
```

E.4: Data Collection

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import java.io.*;
4 import java.util.ArrayList;
5 import java.util.List;
6
7 /**
8 * Class that automatically runs the simulation for one police configuration and
9 * harvests data from it.
10 */
11 public class DataCollector implements ActionListener {
12
13     private final int MAX_TIME; // Max amount of timepoints to dump data for
14     private final int MAX_SEEDS; // Max amount of seeds to test
15     private final int dataCollectionInterval; // How often to collect data
16     public boolean hasInsertedPolice = false; // True if police inserted
17     private Simulation simulation; // Simulation for which to collect data
18     private PrintWriter fileCounterWriter; // Writes number of timepoints and
19                                         // number of seeds
20     private PrintWriter outWriter; // Writes data
21     private int seedCounter = 0; // Counts how many seeds have already been
22                               // tested
23     private int timeCounter = 0; // Counts how many timepoints we have
24                               // already dumped data for
25     private int realTimeElapsed = 0; // Amount of timesteps of the simulation
26                               // elapsed
27     private int insertPoliceAfterCounter = 0; // Counter to test if we should
28                                         // insert police
29     private boolean stillWaitingToInsertPolice = true; // True if police not
30                                         // inserted yet
31     private int insertPoliceAfter; // How many timesteps to wait before
32                               // inserting police
33     private boolean[][] policeSectors = new boolean[10][10]; // Sectors with
34                                         // police
35
36     private String configurationName; // Name of the configuration we are
37                                         // currently testing
38
39     public DataCollector(Simulation simulation, String configurationName,
40                         int dataCollectionInterval, int insertPoliceAfter,
41                         int maxSeeds, int maxTime, boolean[][] policeSectors) {
42         this.simulation = simulation;
43         this.dataCollectionInterval = dataCollectionInterval;
44         this.insertPoliceAfter = insertPoliceAfter;
45         this.policeSectors = policeSectors;
46         MAX_TIME = maxTime;
47         MAX_SEEDS = maxSeeds;
48         // Files are named as out_[seed#]_[time#].py in a subfolder for this
49         // configuration
50         File file = new File(configurationName + "/out_0_0.py");
```

```

51     file.getParentFile().mkdirs();
52     try {
53         fileCounterWriter = new PrintWriter(
54             configurationName + "/counter.py");
55     } catch (FileNotFoundException e) {
56         e.printStackTrace();
57     }
58     try {
59         outWriter = new PrintWriter(configurationName + "/out_0_0.py",
60                                     "UTF-8");
61     } catch (FileNotFoundException e) {
62         e.printStackTrace();
63     } catch (UnsupportedEncodingException e) {
64         e.printStackTrace();
65     }
66     this.configurationName = configurationName;
67     File initFile = new File(configurationName + "/__init__.py");
68     try {
69         initFile.createNewFile();
70     } catch (IOException e) {
71         e.printStackTrace();
72     }
73 }
74
75 /**
76 * Iterates over the given list and, for each element of that list, adds the
77 * value of the field with the given name to a numpy array that is written
78 * to a file using the provided PrintWriter. The resulting array is named
79 * <code>description</code>.
80 *
81 * @param list      The list from which to gather the data
82 * @param propertyName The property that we want to output
83 * @param writer      The writer with which to write the data
84 * @param description The name of the numpy array
85 */
86 public static void writeNumPyArray(List list, String propertyName,
87                                     PrintWriter writer, String description) {
88     writer.print(description + " = array([" +
89     boolean firstTime = true;
90     try {
91         for (Object object : list) {
92             Object value = object.getClass().getField(propertyName).get(
93                 object);
94             if (firstTime) writer.print(value);
95             else writer.print(", " + value);
96             firstTime = false;
97         }
98         writer.println("])");
99         writer.flush();
100    } catch (NoSuchFieldException e) {
101        e.printStackTrace();

```

```

102     } catch (IllegalAccessException e) {
103         e.printStackTrace();
104     }
105 }
106
107 /*@Override*/
108 public void actionPerformed(ActionEvent e) {
109     PositionMatrix matrix = simulation.getMatrix();
110     realTimeElapsed +=
111         Simulation.TIMESTEP; // Check how much time has elapsed to
112         // see if we should dump data
113     if (!simulation.isCurrentIterationFinished) {
114         simulation.runOneTimestep();
115         simulation.repaint();
116         if (stillWaitingToInsertPolice) {
117             insertPoliceAfterCounter++;
118         }
119         if (insertPoliceAfterCounter >=
120             insertPoliceAfter) { // Check if the appropriate amount
121                 // of time before inserting police
122                 // has elapsed
123             stillWaitingToInsertPolice = false;
124         }
125         if (!stillWaitingToInsertPolice &&
126             !hasInsertedPolice) { // Insert the police
127             simulation.setMonitoredSectors(policeSectors);
128             this.hasInsertedPolice = true;
129         }
130     }
131     // Dump data if now is the right time
132     if (realTimeElapsed % dataCollectionInterval == 0 &&
133         !simulation.isCurrentIterationFinished) {
134         realTimeElapsed = 0;
135
136         outWriter.println("from numpy import *");
137         outWriter.flush();
138
139         List<Individual> individuals = matrix.getIndividuals();
140
141         // Write data on positions of individuals
142         writeNumPyArray(individuals, "x", outWriter, "x");
143         writeNumPyArray(individuals, "y", outWriter, "y");
144
145         // Write data on danger levels of individuals
146         writeNumPyArray(individuals, "dangerLevel", outWriter,
147                         "dangerLevel");
148         writeNumPyArray(individuals, "continuousDangerLevel", outWriter,
149                         "continuousDangerLevel");
150
151         writeIsParticipatingData(
152             matrix); // Write data on which individuals are

```

```

153     // participating
154     writeNumPyArray(individuals, "f", outWriter,
155                     "F"); // Write data on force acting on individuals
156     writeNumPyArray(individuals, "density", outWriter,
157                      "density"); // Write data on density surrounding
158     // individuals
159     writeAverageDanger(matrix);
160     writeMaxDanger(matrix);
161     writeMedianDanger(matrix);
162
163     // If we have dumped data enough times, restart with a new seed
164     if (timeCounter >= MAX_TIME) {
165         timeCounter = 0;
166         simulation.setSeed(seedCounter);
167         seedCounter++;
168         simulation.restartSimulation();
169         hasInsertedPolice = false;
170         stillWaitingToInsertPolice = true;
171         insertPoliceAfterCounter = 0;
172     }
173     // If we have tested enough seeds, finish the simulation
174     if (seedCounter >= MAX_SEEDS) {
175         fileCounterWriter.println("n = " + MAX_TIME);
176         fileCounterWriter.println("m = " + MAX_SEEDS);
177         fileCounterWriter.flush();
178         fileCounterWriter.close();
179         outWriter.flush();
180         outWriter.close();
181         simulation.isCurrentIterationFinished = true;
182     } else {
183         resetWriters(configurationName + "/out_" + seedCounter + "_" +
184                     timeCounter + ".py");
185         timeCounter++;
186     }
187 }
188
189 // Writes 1 to a numpy array if individual is participating, else 0
190 private void writeIsParticipatingData(PositionMatrix matrix) {
191     outWriter.print("isParticipating = array([");
192     boolean firstTime = true;
193     for (Individual individual : matrix.getIndividuals()) {
194         if (firstTime) outWriter.print(
195             (individual.isParticipating ? 1 : 0));
196         else outWriter.print(", " + (individual.isParticipating ? 1 : 0));
197         firstTime = false;
198     }
199     outWriter.println("])");
200     outWriter.flush();
201 }
202
203

```

```

204 // Writes the average danger of all the individuals
205 private void writeAverageDanger(PositionMatrix matrix) {
206     outWriter.print("averageDanger = ");
207     double averageDanger = 0;
208     for (Individual individual : matrix.getIndividuals()) {
209         averageDanger += individual.f / 10000 + individual.density / 50;
210     }
211     averageDanger /= matrix.getIndividuals().size();
212     outWriter.println(averageDanger);
213     outWriter.flush();
214 }
215
216 // Writes the max danger of all the individuals
217 private void writeMaxDanger(PositionMatrix matrix) {
218     outWriter.print("maxDanger = ");
219     double maxDanger = Double.MIN_VALUE;
220     for (Individual individual : matrix.getIndividuals()) {
221         maxDanger = Math.max(maxDanger, individual.f / 10000 +
222                             individual.density / 50);
223     }
224     outWriter.println(maxDanger);
225     outWriter.flush();
226 }
227
228 private void writeMedianDanger(PositionMatrix matrix) {
229     outWriter.print("medianDanger = ");
230     // Check if there is an even amount of individuals
231     ArrayList<Individual> individuals =
232         (ArrayList<Individual>) matrix.getIndividuals();
233     int nOver2 = individuals.size() / 2;
234
235     double median = (individuals.size() % 2 == 0) ?
236         (getNthSmallestContinuousDangerLevel(individuals, nOver2) +
237          getNthSmallestContinuousDangerLevel(individuals,
238                                              nOver2 + 1)) / 2 :
239         getNthSmallestContinuousDangerLevel(individuals, nOver2);
240
241     outWriter.println(median);
242     outWriter.flush();
243 }
244
245 // QuickSelect algorithm
246 private static double getNthSmallestContinuousDangerLevel(
247     ArrayList<Individual> individuals, int n) {
248     double result;
249     double pivot;
250
251     // 3 ArrayLists for elements smaller than pivot, equal to pivot,
252     // larger than pivot
253     ArrayList<Individual> lessThanPivot = new ArrayList<Individual>();
254     ArrayList<Individual> greaterThanPivot = new ArrayList<Individual>();

```

```

255     ArrayList<Individual> equalToPivot = new ArrayList<Individual>();
256
257     // Select a random pivot
258     pivot = individuals.get((int) (Math.random() *
259                             individuals.size())).continuousDangerLevel;
260
261     // Add each element of the given list to the appropriate category
262     for (Individual individual : individuals) {
263         if (individual.continuousDangerLevel < pivot) {
264             lessThanPivot.add(individual);
265         } else if (individual.continuousDangerLevel > pivot) {
266             greaterThanPivot.add(individual);
267         } else {
268             equalToPivot.add(individual);
269         }
270     }
271
272     // Recurse into the appropriate category or return the pivot
273     if (n < lessThanPivot.size()) {
274         result = getNthSmallestContinuousDangerLevel(lessThanPivot, n);
275     } else if (n < lessThanPivot.size() + equalToPivot.size()) {
276         result = pivot;
277     } else {
278         result = getNthSmallestContinuousDangerLevel(greaterThanPivot, n -
279                                         lessThanPivot.size() - equalToPivot.size());
280     }
281
282     return result;
283 }
284
285 // Make sure we are writing to the appropriate file
286 private void resetWriters(String newOutFileName) {
287     try {
288         outWriter.close();
289         outWriter = new PrintWriter(newOutFileName, "UTF-8");
290     } catch (FileNotFoundException e3) {
291         e3.printStackTrace();
292     } catch (UnsupportedEncodingException e3) {
293         e3.printStackTrace();
294     }
295 }
296 }
```

E.5: Individual

```
1  /**
2   * Represents an individual.
3   */
4  public class Individual {
5      /**
6       * The x position of this individual.
7       */
8      public double x;
9      /**
10      * The y position of this individual.
11      */
12      public double y;
13      /**
14      * The x velocity of this individual.
15      */
16      public double vx;
17      /**
18      * The y velocity of this individual.
19      */
20      public double vy;
21      /**
22      * True if the individual is participating in the circle pit.
23      */
24      public boolean isParticipating;
25      /**
26      * The size of the individual.
27      */
28      public double radius = 2;
29      /**
30      * The preferred speed of the individual.
31      */
32      public double preferredSpeed = 30;
33      /**
34      * The danger level that the individual is currently at. Ranges from 0 to 6.
35      */
36      public int dangerLevel;
37      /**
38      * The norm of the force currently acting on the individual.
39      */
40      public double f;
41      /**
42      * Number of neighbors of the individual.
43      */
44      public double density;
45      /**
46      * Continuous (rather than discrete) measure of the danger level of an
47      * individual.
48      */
49      public double continuousDangerLevel;
50
```

```

51     public Individual(double x, double y, double vx, double vy,
52                         boolean isParticipating, int dangerLevel) {
53         this.x = x;
54         this.y = y;
55         this.vx = vx;
56         this.vy = vy;
57         this.isParticipating = isParticipating;
58         this.dangerLevel = dangerLevel;
59     }
60
61     public Individual(double[] position, double[] velocity,
62                         boolean isParticipating, int dangerLevel) {
63         this(position[0], position[1], velocity[0], velocity[1],
64               isParticipating, dangerLevel);
65     }
66
67     public double[] getPosition() {
68         return new double[]{x, y};
69     }
70
71     public double[] getVelocity() {
72         return new double[]{vx, vy};
73     }
74
75     public double distanceTo(Individual other) {
76         double dx = Math.abs(x - other.x);
77         double dy = Math.abs(y - other.y);
78
79         return Math.sqrt(dx * dx + dy * dy);
80     }
81
82     public double distanceTo(double[] point) {
83         double dx = Math.abs(x - point[0]);
84         double dy = Math.abs(y - point[1]);
85
86         return Math.sqrt(dx * dx + dy * dy);
87     }
88
89     @Override
90     public String toString() {
91         return "Individual: position = (" + x + ", " + y + "), velocity = (" +
92                 vx + ", " + vy + "), isParticipating = " + isParticipating;
93     }
94 }
```

E.6: Position Matrix

```
1 import java.util.ArrayList;
2 import java.util.LinkedList;
3 import java.util.List;
4
5 /**
6  * Data structure representing the position matrix in the simulation.
7 */
8 public class PositionMatrix {
9
10    /**
11     * The width of the matrix
12     */
13    public final int width;
14    /**
15     * The height of the matrix
16     */
17    public final int height;
18    /**
19     * The size of one sector
20     */
21    public final int sectorSize;
22    /**
23     * A boolean matrix to indicate whether sector (i, j) is under police
24     * surveillance
25     */
26    public boolean[][] isPoliceAtSector;
27    // The actual matrix
28    private LinkedList<Individual>[][] matrix;
29    // A List containing all the individuals, makes iterating over them easier
30    private ArrayList<Individual> individuals;
31
32    /**
33     * Creates a new PositionMatrix
34     *
35     * @param width      The desired width of the matrix
36     * @param height     The desired height of the matrix
37     * @param sectorSize The desired sector size
38     */
39    public PositionMatrix(int width, int height, int sectorSize) {
40        this.height = height;
41        this.width = width;
42        this.sectorSize = sectorSize;
43
44        // Initialize the matrix
45        matrix = (LinkedList<Individual>[][])
46            new LinkedList[width][height];
47        for (int i = 0; i < width; i++) {
48            for (int j = 0; j < height; j++) {
49                matrix[i][j] = new LinkedList<Individual>();
50            }
51        }
52    }
53}
```

```

51     individuals = new ArrayList<Individual>();
52     isPoliceAtSector = new boolean[width][height];
53 }
54
55 /**
56 * Gets an individual at position (i, j) in the matrix.
57 *
58 * @param i The row of the individual
59 * @param j The column of the individual
60 * @return The individual at (i, j)
61 */
62 public List<Individual> get(int i, int j) {
63     return matrix[i][j];
64 }
65
66 /**
67 * Gets an individual at the specified sector in the matrix.
68 *
69 * @param sector The sector that the individual is in.
70 * @return The individual at the given sector.
71 */
72 public List<Individual> get(Sector sector) {
73     return matrix[sector.row][sector.col];
74 }
75
76 /**
77 * Gets a list of all the individuals.
78 *
79 * @return The list of the individuals.
80 */
81 public List<Individual> getIndividuals() {
82     return individuals;
83 }
84
85 /**
86 * Adds a new individual to the matrix at the proper position.
87 *
88 * @param individual The individual to be added
89 */
90 public void add(Individual individual) {
91     Sector sector = getSectorForCoords(individual);
92     matrix[sector.row][sector.col].add(individual);
93     individuals.add(individual);
94 }
95
96
97 public void removeAndAdd(Individual individual, Sector oldSector,
98                         Sector newSector) {
99     matrix[oldSector.row][oldSector.col].remove(individual);
100    matrix[newSector.row][newSector.col].add(individual);
101 }

```

```

102
103 /**
104 * Gets the neighbors of an individual.
105 *
106 * @param individual The individual for whom to search for neighbors
107 * @param radius The radius in which to search
108 * @return A <code>java.util.List</code> of the neighbors.
109 */
110 public List<Individual> getNeighborsFor(Individual individual,
111                                         double radius) {
112     ArrayList<Individual> neighbors = new ArrayList<Individual>(10);
113     ArrayList<Sector> sectorsToSearch = new ArrayList<Sector>(9);
114     Sector sector = getSectorForCoords(individual);
115     sectorsToSearch.add(sector);
116
117     // Look around the current sector
118     for (int i = -1; i <= 1; i++) {
119         for (int j = -1; j <= 1; j++) {
120             Sector newSector = getSectorForCoords(individual.x + i * radius,
121                                                 individual.y +
122                                                 i * radius);
123             if (!sector.equals(newSector)) {
124                 sectorsToSearch.add(newSector);
125             }
126         }
127     }
128
129     // Look for neighbors at the appropriate distance in the sectors to
130     // search
131     for (Sector sectorToSearch : sectorsToSearch) {
132         List<Individual> possibleNeighbors = get(sectorToSearch);
133         for (Individual neighbor : possibleNeighbors) {
134             if (individual.distanceTo(neighbor) < radius) {
135                 neighbors.add(neighbor);
136             }
137         }
138     }
139
140     return neighbors;
141 }
142
143 /**
144 * Gets the sector for the given coordinates.
145 *
146 * @param x The x coordinate
147 * @param y The y coordinate
148 * @return A <code>Sector</code> object corresponding the the given
149 * coordinates.
150 */
151 public Sector getSectorForCoords(double x, double y) {
152     int sectorX = (int) x / sectorSize;

```

```

153     int sectorY = (int) y / sectorSize;
154
155     // Make sure the sector is not out of bounds
156     if (sectorX < 0) sectorX = 0;
157     else if (sectorX >= width) sectorX = width - 1;
158
159     if (sectorY < 0) sectorY = 0;
160     else if (sectorY >= height) sectorY = height - 1;
161
162     return new Sector(sectorX, sectorY);
163 }
164
165 /**
166 * Gets the sector that the given individual is in.
167 *
168 * @param individual The individual for whom to get the sector
169 * @return The sector that the individual is in.
170 * @see PositionMatrix#getSectorForCoords(double, double)
171 */
172 public Sector getSectorForCoords(Individual individual) {
173     return getSectorForCoords(individual.x, individual.y);
174 }
175
176 /**
177 * Gets the sector that the given individual is in given an array of {x, y}
178 * coordinates.
179 *
180 * @param position The array of coordinates
181 * @return The sector corresponding to <code>position</code>
182 * @see PositionMatrix#getSectorForCoords(double, double)
183 */
184 public Sector getSectorForCoords(double[] position) {
185     return getSectorForCoords(position[0], position[1]);
186 }
187
188 /**
189 * Checks if the given sector is monitored by the police.
190 */
191 public boolean isSectorMonitored(Sector sector) {
192     return isPoliceAtSector[sector.row][sector.col];
193 }
194
195 /**
196 * Adds the given list of sectors to be monitored by the police.
197 *
198 * @param policeSectors A comma separated list of sectors, e.g. 0, 0, 1, 1,
199 *                      2, 2
200 */
201 public void setMonitoredSectors(int... policeSectors) {
202     for (int i = 0; i < policeSectors.length; i += 2) {
203         isPoliceAtSector[policeSectors[i]][policeSectors[i + 1]] = true;

```

```
204     }
205 }
206
207 /**
208 * An inner class that represents one sector at a particular location in the
209 * matrix.
210 */
211 public class Sector {
212     int row; // Row of the matrix
213     int col; // Column of the matrix
214
215     public Sector(int row, int col) {
216         this.row = row;
217         this.col = col;
218     }
219
220     public boolean equals(Sector other) {
221         return row == other.row && col == other.col;
222     }
223 }
224 }
```

E.7: Simulation

```
1 import javax.swing.*;
2 import java.awt.event.ActionEvent;
3 import java.awt.event.ActionListener;
4 import java.io.FileNotFoundException;
5 import java.io.PrintWriter;
6 import java.io.UnsupportedEncodingException;
7 import java.util.List;
8 import java.util.Random;
9
10 /**
11  * Class representing one or multiple simulations of a circle pit.
12 */
13 public class Simulation {
14
15     /**
16      * Size of one sector of the matrix
17      */
18     public static final int SECTOR_SIZE = 10;
19
20     /**
21      * The duration of one timestep (in milliseconds)
22      */
23     public final static int Timestep = 50;
24
25     /**
26      * Strength of repulsive force
27      */
28     public double epsilon;
29
30     /**
31      * Strength of propulsion
32      */
33     public double mu;
34
35     /**
36      * Strength of flocking force
37      */
38     public double alpha;
39
40     /**
41      * Strength of the centripetal force
42      */
43     public double gamma;
44
45     /**
46      * Number of people in the simulation
47      */
48     public double numberOfPeople;
49
50     /**
51      * Radius within which velocity of neighbors has an effect on the flocking
52      * force
53      */
54     public double flockRadius;
55
56     /**
57      * Timestep of the simulation
58      */
59
```

```

51  public double dt;
52  /**
53   * Percentage of people initially participating in the circle pit
54   */
55  public double percentParticipating;
56  /**
57   * Radius within which the 'isParticipating' of neighbors affects the
58   * individual
59   */
60  public double rParticipating;
61  /**
62   * Minimum amount of people necessary to constitute a circle pit
63   */
64  public int minCirclePitSize;
65  /**
66   * Necessary number of neighbors participating to start participating
67   */
68  public int minParticipatingNeighbors;
69  /**
70   * Center of the matrix
71   */
72  public double[] center;
73  /**
74   * Window with the simulation
75   */
76  public SimulationGUI window;
77  /**
78   * Safe density level
79   */
80  public int safeDensity = 10;
81  /**
82   * Density danger level 1
83   */
84  public int density1 = 20;
85  /**
86   * Density danger level 2
87   */
88  public int density2 = 40;
89  /**
90   * safe Force danger level
91   */
92  public int safeForce = 1000;
93  /**
94   * Force danger level 1
95   */
96  public int force1 = 3000;
97  /**
98   * Density danger level 2
99   */
100 public int force2 = 4000;
101 /**

```

```

102     * True if force is considered to be a danger factor
103     */
104     public boolean enableForce = true;
105 /**
106     * True if density is considered to be a danger factor
107     */
108     public boolean enableDensity = true;
109 /**
110     * False while the current iteration of the simulation is running (used for
111     * automation)
112     */
113     public boolean isCurrentIterationFinished = false;
114
115     private PrintWriter writer; // Writes analysis data to a file
116
117     private double maxX; // Right-hand border of the terrain
118     private double maxY; // Bottom border of the terrain
119     private Timer timer; // Timer to run the simulation
120
121     private Random random = new Random(42); // To generate random numbers
122     private int fileCounter = 0; // Counts how many files were written
123     private PrintWriter fileCounterWriter;
124         // Writes information about how many files were written to a file
125
126     private PositionMatrix matrix;
127
128 /**
129     * Creates a new Simulation with the specified parameters.
130     *
131     * @param epsilon Strength of the repulsive force
132     * @param mu Strength of the propulsive force
133     * @param alpha Strength of the flocking force
134     * @param gamma Strength of the centripetal force
135     * @param numberOfPeople Number of people at the concert
136     * @param flockRadius Radius in which to look for neighbors
137     * for the flocking force
138     * @param dt Duration of one timestep
139     * @param percentParticipating Proportion of people initially
140     * participating in the circle pit
141     * @param rParticipating Radius in which to search for
142     * participating neighbors
143     * @param minCirclePitSize Minimum number of participating
144     * neighbors needed to continue
145     * participating in the circle pit
146     * @param minParticipatingNeighbors Minimum number of participating
147     * neighbors needed to join in the circle
148     * pit
149     */
150     public Simulation(double epsilon, double mu, double alpha, double gamma,
151                     double numberOfPeople, double flockRadius, double dt,
152                     double percentParticipating, double rParticipating,

```

```

153             int minCirclePitSize, int minParticipatingNeighbors) {
154     this.epsilon = epsilon;
155     this.mu = mu;
156     this.alpha = alpha;
157     this.gamma = gamma;
158     this.numberOfPeople = numberOfPeople;
159     this.flockRadius = flockRadius;
160     this.dt = dt;
161     this.percentParticipating = percentParticipating;
162     this.rParticipating = rParticipating;
163     this.minCirclePitSize = minCirclePitSize;
164     this.minParticipatingNeighbors = minParticipatingNeighbors;
165     try {
166         fileCounterWriter = new PrintWriter("special_counter.py");
167     } catch (FileNotFoundException e) {
168         e.printStackTrace();
169     }
170     initializeMatrix();
171 }
172
173 // Initializes the matrix with individuals with random velocity and
174 // position.
175 private void initializeMatrix() {
176     int matrixSize = 10;
177
178     // The maximum x and y values that an individual can have
179     maxX = matrixSize * SECTOR_SIZE;
180     maxY = matrixSize * SECTOR_SIZE;
181
182     center = new double[]{maxX / 2, maxY / 2};
183
184     matrix = new PositionMatrix(matrixSize, matrixSize,
185                                 SECTOR_SIZE);
186
187     for (int i = 0; i < numberOfPeople; i++) {
188         // Generate random coordinates
189         double[] coords = new double[2];
190         coords[0] = random.nextDouble() * SECTOR_SIZE * matrixSize;
191         coords[1] = random.nextDouble() * SECTOR_SIZE * matrixSize;
192
193         // Generate random velocity
194         double[] velocity = new double[]{random.nextDouble() - 0.5,
195                                         random.nextDouble() - 0.5};
196
197         // Decide whether individual is initially participating
198         boolean isParticipating =
199             random.nextDouble() < percentParticipating;
200
201         //levels of danger, 0 is safe, by default it's safe
202         int dangerLevel = 0;
203

```

```

204         Individual individual = new Individual(coords, velocity,
205                                         isParticipating,
206                                         dangerLevel);
207
208         // Add individual to appropriate sector
209         matrix.add(individual);
210     }
211 }
212
213 /**
214 * Resets the matrix of the simulation.
215 */
216 public void resetMatrix() {
217     stop();
218     basicReset();
219 }
220
221 /**
222 * Restarts the simulation.
223 */
224 public void restartSimulation() {
225     stop();
226     basicReset();
227     start();
228 }
229
230 /**
231 * Performs a basic reset of the simulation back to its initial state.
232 */
233 public void basicReset() {
234     initializeMatrix();
235     window.resetSimulationPanel();
236     window.repaint();
237 }
238
239 /**
240 * Stops the simulation.
241 */
242 public void stop() {
243     timer.stop();
244 }
245
246 /**
247 * Starts the simulation.
248 */
249 public void start() {
250     timer.start();
251 }
252
253 /**
254 * Creates a new <code>Timer</code> to run the simulation

```

```

255 *
256 * @param listener The <code>ActionListener</code> the new
257 *                  <code>Timer</code> should be based on
258 */
259 public void createNewTimer(ActionListener listener) {
260     timer = new Timer(TIMESTEP, listener);
261 }
262
263 /**
264 * Redraws the simulation.
265 */
266 public void repaint() {
267     window.repaint();
268 }
269
270 /**
271 * Sets the seed of the <code>Random</code> object used during the
272 * simulation.
273 *
274 * @param seed The seed to set the <code>Random</code> object to
275 */
276 public void setSeed(int seed) {
277     random = new Random(seed);
278 }
279
280 // Checks if the given neighbor at the given distance is participating
281 private boolean isNeighborParticipating(Individual neighbor,
282                                         double distance) {
283     return neighbor.isParticipating && distance < rParticipating;
284 }
285
286 /**
287 * Runs the simulation manually (so that the user can interact with it).
288 */
289 public void runManualSimulation() {
290     window = new SimulationGUI(this);
291
292     // Run a new frame every 50 milliseconds
293     timer = new Timer(50, new ActionListener() {
294         /*@Override*/
295         public void actionPerformed(ActionEvent e) {
296             runOneTimestep();
297             window.repaint();
298         }
299     });
300
301     window.setVisible(true);
302 }
303
304 /**
305 * Creates a new window that displays and animates the simulation.

```

```

306     */
307     public void createWindow() {
308         window = new SimulationGUI(this);
309         window.setVisible(true);
310     }
311
312     /**
313      * Runs the simulation automatically (for use for automatic data
314      * collection).
315      *
316      * @param name           The name of the folder in which to put
317      *                       generated data.
318      * @param dataCollectionInterval How often to collect data (in
319      *                               milliseconds)
320      * @param insertPoliceAfter The amount of time after which police
321      *                          should be inserted, if any
322      * @param amountOfSeeds    How many random seeds to test
323      * @param collectionTimes  How many times to collect data per seed
324      */
325     public void runAutomaticSimulation(String name, int dataCollectionInterval,
326                                         int insertPoliceAfter, int amountOfSeeds,
327                                         int collectionTimes) {
328         window = new SimulationGUI(this);
329
330         DataCollector collector = new DataCollector(this, name,
331                                               dataCollectionInterval,
332                                               insertPoliceAfter,
333                                               amountOfSeeds,
334                                               collectionTimes,
335                                               new boolean[10][10]);
336
337         timer = new Timer(TIMESTEP, collector);
338         timer.start();
339         window.setVisible(true);
340     }
341
342     /**
343      * Runs one timestep of the simulation.
344      */
345     public void runOneTimestep() {
346
347         // List of all the individuals in the matrix
348         List<Individual> individuals = matrix.getIndividuals();
349
350         // Iterate over each individual
351         for (Individual individual : individuals) {
352             // Sector where the individual is before updating position
353             PositionMatrix.Sector initialSector = matrix.getSectorForCoords(
354                 individual);
355             // Amount of neighbors participating with the radius rParticipating
356             int sumParticipating = 0;
357             // Forces acting upon individual

```

```

357     double[] F = {0.0, 0.0};
358     // Sum of the neighbor velocities
359     double[] sumOverVelocities = {0.0, 0.0};
360     // List of neighbors
361     List<Individual> neighbors = matrix.getNeighborsFor(individual,
362                                         flockRadius);
363     // Position and velocity of individual
364     double[] position = individual.getPosition();
365     double[] velocity = individual.getVelocity();
366     double r0 = 2 * individual.radius;
367
368     individual.dangerLevel = 0;
369     // Calculate the danger level
370     int numNeighbors = neighbors.size();
371
372     // We use the number of people in the neighbor list to represent
373     // density
374     if (enableDensity) {
375         if (numNeighbors > density2)
376             individual.dangerLevel = 3;
377         else if (numNeighbors > density1)
378             individual.dangerLevel = 2;
379         else if (numNeighbors > safeDensity)
380             individual.dangerLevel = 1;
381         else if (numNeighbors < safeDensity)
382             individual.dangerLevel = 0;
383     }
384
385     // ===== CALCULATION OF THE FORCES =====
386     for (Individual neighbor : neighbors) {
387         // Make sure we are not using the individual him/herself
388         if (neighbor == individual) {
389             continue;
390         }
391         double[] positionNeighbor = neighbor.getPosition();
392         double[] velocityNeighbor = neighbor.getVelocity();
393
394         double distance = individual.distanceTo(neighbor);
395
396         sumParticipating += isNeighborParticipating(neighbor,
397                                                 distance) ? 1 : 0;
398
399         // Repulsive force
400         // We only use neighbors within a radius of 2 * r0
401
402         if (distance < 2 * individual.radius) {
403             F[0] += epsilon * 500 * (individual.x - neighbor.x) /
404                     distance;
405             F[1] += epsilon * 500 * (individual.y - neighbor.y) /
406                     distance;
407         } else if (distance < 2 * r0) {

```

```

408     F[0] += -epsilon * ((1 / (distance - 2 * r0)) *
409                           (position[0] - positionNeighbor[0])) / distance;
410     F[1] += -epsilon * ((1 / (distance - 2 * r0)) *
411                           (position[1] - positionNeighbor[1])) / distance;
412 }
413
414     sumOverVelocities[0] += velocityNeighbor[0];
415     sumOverVelocities[1] += velocityNeighbor[1];
416 }
417
418 // Calculate the norm of the force
419 double jointForce = norm(F);
420
421 if (enableForce) {
422     if (jointForce > force2)
423         individual.dangerLevel += 3;
424     else if (jointForce > force1)
425         individual.dangerLevel += 2;
426     else if (jointForce > safeForce)
427         individual.dangerLevel += 1;
428     else if (jointForce < safeForce)
429         individual.dangerLevel += 0;
430 }
431
432 // Adjust the danger level if only one of the two options is enabled
433 if (enableForce ^ enableDensity) {
434     individual.dangerLevel *= 2;
435 }
436
437 // Decide if the individual is participating or not
438 if (sumParticipating >= minParticipatingNeighbors) {
439     individual.isParticipating = true;
440 }
441
442 if (sumParticipating < minCirclePitSize) {
443     individual.isParticipating = false;
444 }
445
446 if (matrix.isSectorMonitored(
447     matrix.getSectorForCoords(individual))) {
448     individual.isParticipating = false;
449 }
450
451 // Set preferred speed accordingly
452 individual.preferredSpeed =
453     individual.isParticipating ? 30 : 5 * random.nextDouble();
454
455 // Propulsion
456 // Makes the individual want to travel at their preferred speed
457 double vi = individual.preferredSpeed;
458 F[0] += -mu * (norm(velocity) - vi) * velocity[0] / norm(velocity);

```

```

459 F[1] += -mu * (norm(velocity) - vi) * velocity[1] / norm(velocity);
460
461 // Flocking
462 if (individual.isParticipating &&
463     !(sumOverVelocities[0] == 0 && sumOverVelocities[1] == 0)) {
464     double norm = norm(sumOverVelocities);
465     F[0] += alpha * sumOverVelocities[0] / norm;
466     F[1] += alpha * sumOverVelocities[1] / norm;
467 }
468
469 // Centripetal Force
470 if (individual.isParticipating) {
471     double distanceToCenter = individual.distanceTo(center);
472
473     // Normalized vector from center to individual
474     double[] r = new double[]{center[0] - individual.x,
475                             center[1] - individual.y};
476     r[0] /= distanceToCenter;
477     r[1] /= distanceToCenter;
478
479     F[0] += gamma * r[0];
480     F[1] += gamma * r[1];
481 }
482
483 // Add noise
484 // TODO: Generate noise
485
486 // Make sure that F does not become too large to fit into a double
487 double normOff = norm(F);
488
489 if (normOff > Long.MAX_VALUE) {
490     F[0] /= normOff * Long.MAX_VALUE;
491     F[1] /= normOff * Long.MAX_VALUE;
492 }
493
494 // ===== CALCULATE Timestep =====
495 // Using the leap-frog method to integrate the differential equation
496 // d^2y/dt^2 = rhs(y)
497
498 // Shifted initial velocity for leap-frog
499 double[] v_temp = new double[2];
500 v_temp[0] = velocity[0] + 0.5 * dt * F[0];
501 v_temp[1] = velocity[1] + 0.5 * dt * F[1];
502
503 // New position of the individual
504 double newX = position[0] + dt * v_temp[0];
505 double newY = position[1] + dt * v_temp[1];
506
507 // New velocity of the individual
508 double newVx = v_temp[0] + dt * F[0] / 2;
509 double newVy = v_temp[1] + dt * F[1] / 2;

```

```

510
511     // Make sure individuals rebound off the edges of the space
512     if (newX < 0 || newX > maxX) {
513         newVx = -newVx;
514         F[0] = -F[0];
515     }
516     if (newY < 0 || newY > maxY) {
517         newVy = -newVy;
518         F[1] = -F[1];
519     }
520
521     // Make sure they don't get stuck in an out-of-bounds area
522     if (newX >= 0 && newX <= maxX)
523         individual.x = newX;
524     if (newY >= 0 && newY <= maxY)
525         individual.y = newY;
526
527     individual.vx = newVx;
528     individual.vy = newVy;
529
530     // Add the individual to the correct sector
531     PositionMatrix.Sector newSector = matrix.getSectorForCoords(
532             individual);
533     if (!newSector.equals(initialSector)) {
534         matrix.removeAndAdd(individual, initialSector, newSector);
535     }
536
537     // Set the force acting on this individual and the amount of
538     // neighbors so that the danger level can be assessed
539     individual.f = norm(F);
540     individual.density = neighbors.size();
541     individual.continuousDangerLevel =
542             individual.f / 10000 + individual.density / 50;
543     }
544 }
545
546 /**
547 * Writes a small Python/Numpy script to allow analysis of the current
548 * situation.
549 */
550 public void exportData() {
551     try {
552         writer.close();
553         writer = new PrintWriter("special" + fileCounter + ".py", "UTF-8");
554     } catch (FileNotFoundException e) {
555         e.printStackTrace();
556     } catch (UnsupportedEncodingException e) {
557         e.printStackTrace();
558     } catch (NullPointerException e) {
559         try {
560             writer = new PrintWriter("special" + fileCounter + ".py",

```

```

561                                     "UTF-8");
562     } catch (FileNotFoundException e1) {
563         e1.printStackTrace();
564     } catch (UnsupportedEncodingException e1) {
565         e1.printStackTrace();
566     }
567 }
568 try {
569     fileCounterWriter.close();
570     fileCounterWriter = new PrintWriter("special_counter.py");
571 } catch (FileNotFoundException e) {
572     e.printStackTrace();
573 }
574 writer.println("from numpy import *");
575 writer.print("x = array([");
576 boolean firstTime = true;
577 for (Individual individual : matrix.getIndividuals()) {
578     if (firstTime) writer.print(individual.x);
579     else writer.print(", " + individual.x);
580     firstTime = false;
581 }
582 writer.println("])");
583 writer.flush();
584
585 writer.print("y = array([");
586 firstTime = true;
587 for (Individual individual : matrix.getIndividuals()) {
588     if (firstTime) writer.print(individual.y);
589     else writer.print(", " + individual.y);
590     firstTime = false;
591 }
592 writer.println("])");
593 writer.flush();
594
595 writer.print("dangerLevel = array([");
596 firstTime = true;
597 for (Individual individual : matrix.getIndividuals()) {
598     if (firstTime) writer.print(individual.dangerLevel);
599     else writer.print(", " + individual.dangerLevel);
600     firstTime = false;
601 }
602 writer.println("])");
603 writer.flush();
604
605 writer.print("isParticipating = array([");
606 firstTime = true;
607 for (Individual individual : matrix.getIndividuals()) {
608     if (firstTime) writer.print((individual.isParticipating ? 1 : 0));
609     else writer.print(", " + (individual.isParticipating ? 1 : 0));
610     firstTime = false;
611 }

```

```

612     writer.println("])");
613     writer.flush();
614
615     writer.print("F = array[");
616     firstTime = true;
617     for (Individual individual : matrix.getIndividuals()) {
618         if (firstTime) writer.print(individual.f);
619         else writer.print(", " + individual.f);
620         firstTime = false;
621     }
622     writer.println("]");
623     writer.flush();
624
625     writer.print("density = array[");
626     firstTime = true;
627     for (Individual individual : matrix.getIndividuals()) {
628         if (firstTime) writer.print(individual.density);
629         else writer.print(", " + individual.density);
630         firstTime = false;
631     }
632     writer.println("]");
633     writer.flush();
634     fileCounter++;
635     fileCounterWriter.println("n = " + fileCounter);
636     fileCounterWriter.flush();
637 }
638
639 // Calculates the norm of the given vector.
640 private double norm(double[] vector) {
641     return Math.sqrt(vector[0] * vector[0] + vector[1] * vector[1]);
642 }
643
644 /**
645  * Gets the matrix associated with this object.
646  *
647  * @return The <code>PositionMatrix</code> object the individuals are stored
648  * in.
649  */
650 public PositionMatrix getMatrix() {
651     return matrix;
652 }
653
654 /**
655  * Gets the <code>Timer</code> that is running this simulation.
656  *
657  * @return
658  */
659 public Timer getSimulationTimer() {
660     return timer;
661 }
662

```

```
663  /**
664   * Sets which sectors are monitored by the police.
665   *
666   * @param sectors A comma-separated list of comma-separated pairs of numbers
667   *                 representing sectors monitored by the police
668   */
669  public void setMonitoredSectors(int... sectors) {
670      matrix.setMonitoredSectors(sectors);
671  }
672
673 /**
674  * Sets which sectors are monitored by the police.
675  *
676  * @param sectors A <code>boolean[][]</code> array containing
677  *                 <code>true</code> at position (i, j) if a policeman is in
678  *                 that sector
679  */
680  public void setMonitoredSectors(boolean[][] sectors) {
681      matrix.isPoliceAtSector = sectors;
682  }
683 }
```

E.8: Graphical User Interface

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class SimulationGUI extends JFrame {
5
6     private SimulationPanel simulationPanel;
7     private ControlPanel controlPanel;
8     private Simulation simulation;
9
10    public SimulationGUI(Simulation simulation) {
11        this.simulation = simulation;
12        try {
13            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
14        } catch (ClassNotFoundException e) {
15            e.printStackTrace();
16        } catch (InstantiationException e) {
17            e.printStackTrace();
18        } catch (IllegalAccessException e) {
19            e.printStackTrace();
20        } catch (UnsupportedLookAndFeelException e) {
21            e.printStackTrace();
22        }
23        setTitle("Circle Pit Simulation");
24        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
25        setLayout(new BorderLayout());
26        setResizable(false);
27
28        initComponents();
29        addComponents();
30        pack();
31        setLocationRelativeTo(null);
32    }
33
34    private void initComponents() {
35        PositionMatrix matrix = simulation.getMatrix();
36        simulationPanel = new SimulationPanel(500, 500, simulation,
37                                              matrix.width * matrix.sectorSize,
38                                              matrix.height *
39                                              matrix.sectorSize);
40
41        controlPanel = new ControlPanel(simulation, simulationPanel);
42    }
43
44    private void addComponents() {
45        add(simulationPanel, BorderLayout.CENTER);
46        add(controlPanel, BorderLayout.EAST);
47    }
48
49    /**
50     * Resets the simulation panel.
```

```
51     */
52     public void resetSimulationPanel() {
53         simulationPanel.setIndividuals(simulation.getMatrix().getIndividuals());
54     }
55 }
```

E.9: Simulation Panel

```
1 import javax.imageio.ImageIO;
2 import javax.swing.*;
3 import java.awt.*;
4 import java.awt.event.MouseAdapter;
5 import java.awt.event.MouseEvent;
6 import java.awt.geom.Ellipse2D;
7 import java.awt.geom.Rectangle2D;
8 import java.awt.image.BufferedImage;
9 import java.io.File;
10 import java.io.IOException;
11
12 /**
13  * Draws the matrix into a JPanel.
14  */
15 public class SimulationPanel extends JPanel {
16
17     public boolean shouldShowParticipants = true;
18     private java.util.List<Individual> individuals;
19     private double xScalingFactor, yScalingFactor;
20     private Simulation simulation;
21     private BufferedImage image;
22
23     public SimulationPanel(int width, int height, Simulation simulation,
24                           int matrixWidth, int matrixHeight) {
25         this.individuals = simulation.getMatrix().getIndividuals();
26         xScalingFactor = width / matrixWidth;
27         yScalingFactor = height / matrixHeight;
28         this.simulation = simulation;
29         try {
30             image = ImageIO.read(new File("policeman.png"));
31         } catch (IOException e) {
32             e.printStackTrace();
33         }
34         this.setSize(new Dimension(width, height));
35         this.setPreferredSize(new Dimension(width, height));
36         this.setMinimumSize(new Dimension(width, height));
37         this.setBackground(Color.WHITE);
38         addListeners();
39         setFocusable(false);
40     }
41
42     private void addListeners() {
43         // Add a policeman on mouse click
44         this.addMouseListener(new MouseAdapter() {
45             @Override
46             public void mouseClicked(MouseEvent e) {
47                 double x = e.getX() / xScalingFactor;
48                 double y = e.getY() / yScalingFactor;
49                 PositionMatrix.Sector sector =
50                     simulation.getMatrix().getSectorForCoords(x, y);
```

```

51             simulation
52                 .getMatrix().isPoliceAtSector[sector.row][sector.col] =
53                     !(simulation
54                         .getMatrix().isPoliceAtSector[sector
55                             .row][sector.col]);
56             getParent().repaint();
57         }
58     );
59 }
60 /**
61 * Sets the individuals list to the given parameter.
62 */
63 public void setIndividuals(java.util.List<Individual> individuals) {
64     this.individuals = individuals;
65 }
66
67 // Increases the rendering quality of the simulation
68 private void increaseRenderingQuality(Graphics2D g2d) {
69     g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
70                         RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
71     g2d.setRenderingHint(RenderingHints.KEY_DITHERING,
72                         RenderingHints.VALUE_DITHER_ENABLE);
73     g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
74                         RenderingHints.VALUE_RENDER_QUALITY);
75     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
76                         RenderingHints.VALUE_ANTIALIAS_ON);
77     g2d.setRenderingHint(RenderingHints.KEY_FRACTIONALMETRICS,
78                         RenderingHints.VALUE_FRACTIONALMETRICS_ON);
79     g2d.setRenderingHint(RenderingHints.KEY_ALPHA_INTERPOLATION,
80                         RenderingHints.VALUE_ALPHA_INTERPOLATION_QUALITY);
81     g2d.setRenderingHint(RenderingHints.KEY_COLOR_RENDERING,
82                         RenderingHints.VALUE_COLOR_RENDER_QUALITY);
83     g2d.setRenderingHint(RenderingHints.KEY_STROKE_CONTROL,
84                         RenderingHints.VALUE_STROKE_PURE);
85 }
86
87 @Override
88 public void paintComponent(Graphics g) {
89     super.paintComponent(g);
90     int sumDanger = 0;
91     Graphics2D graphics2D = (Graphics2D) g.create();
92     increaseRenderingQuality(graphics2D); // Comment this out to
93                                         // potentially make animation
94                                         // smoother (if laggy)
95
96     // Draw each individual
97     for (Individual individual : individuals) {
98         double[] coords = individual.getPosition();
99         coords[0] *= xScalingFactor;
100        coords[1] *= yScalingFactor;

```

```

102
103     // Set the color depending on the danger level of the individual
104     int dangerLevel = individual.dangerLevel;
105     if (dangerLevel == 0) {
106         graphics2D.setColor(new Color(204, 229, 255));
107     } else if (dangerLevel == 1) {
108         graphics2D.setColor(new Color(154, 204, 255));
109     } else if (dangerLevel == 2) {
110         graphics2D.setColor(new Color(102, 178, 255));
111     } else if (dangerLevel == 3) {
112         sumDanger++;
113         graphics2D.setColor(new Color(51, 153, 255));
114     } else if (dangerLevel == 4) {
115         sumDanger += 2;
116         graphics2D.setColor(new Color(0, 128, 255));
117     } else if (dangerLevel == 5) {
118         sumDanger += 3;
119         graphics2D.setColor(new Color(0, 102, 204));
120     } else if (dangerLevel == 6) {
121         sumDanger += 4;
122         graphics2D.setColor(new Color(255, 51, 51));
123     }

124
125     graphics2D.fill(new Ellipse2D.Double(coords[0], coords[1],
126                                         individual.radius *
127                                         xScalingFactor,
128                                         individual.radius *
129                                         yScalingFactor));
130
131     // Fill the individual with a white circle if not participating
132     if (shouldShowParticipants) {
133         if (!individual.isParticipating) {
134             graphics2D.setColor(Color.WHITE);
135             graphics2D.fill(new Ellipse2D.Double(
136                 coords[0] + individual.radius * xScalingFactor / 4,
137                 coords[1] + individual.radius * yScalingFactor / 4,
138                 individual.radius * xScalingFactor / 2,
139                 individual.radius * yScalingFactor / 2));
140
141         }
142     }
143 }

144
145     // General Danger indicator
146     graphics2D.setColor(new Color(102, 102, 0));
147     graphics2D.fill(new Rectangle2D.Double(0, 10, 13, sumDanger / 5));
148     graphics2D.setColor(new Color(204, 204, 0));
149     graphics2D.fill(new Rectangle2D.Double(1, 11, 11, sumDanger / 5 - 2));
150
151     // Drawing the policemen
152     boolean[][] monitoredSectors = simulation.getMatrix().isPoliceAtSector;

```

```
153     for (int i = 0; i < monitoredSectors.length; i++) {
154         for (int j = 0; j < monitoredSectors[i].length; j++) {
155             if (monitoredSectors[i][j]) {
156                 graphics2D.drawImage(image,
157                     (int) ((i * 10 + 4) * xScalingFactor),
158                     (int) ((j * 10 + 4) * yScalingFactor),
159                     (int) (3 * xScalingFactor),
160                     (int) (3 * yScalingFactor), null);
161             }
162         }
163     }
164     g.dispose();
165 }
166 }
```

Appendix F: Python Code

F.1: Time Evolution of Participation Rate and Danger

```
1 # -*- coding: utf-8 -*-
2 from matplotlib.pyplot import *
3 from numpy import *
4
5 colors = array(
6     ['dodgerblue', 'darkorange', 'g', 'red', 'purple', 'brown', 'violet',
7      'grey', 'y', 'lightskyblue', 'black',
8      'magenta'])
9
10
11 def analyse_median_danger(test_set, i):
12     to_do = "from %s.counter import *" % test_set
13
14     exec(to_do, globals())
15
16     median_danger_time_evolution = zeros((n - 5, m))
17
18     t = linspace(0, 1, n - 5)
19     title('Time Evolution of Average of Median Danger')
20
21     for j in range(0, m): # iterating over different seeds
22         for k in range(0, n - 5): # iterating over time steps
23             filename = 'out_%s_%s' % (j, k)
24             exec("from %s.%s import *" % (test_set, filename),
25                  globals()) # importing data for given seed and time steps
26             median_danger_time_evolution[k, j] += medianDanger
27     y = average(median_danger_time_evolution, axis=1)
28     ylabel('Median Danger')
29     ymax = max(y)
30     if (test_set == 'control'):
31         linewidth_ = 2
32     else:
33         linewidth_ = 1.2
34     plot(t, y / ymax, colors[i], linewidth=linewidth_, label=test_set)
35
36
37 def analyse_is_participating(test_set, i):
38     ToDo = "from %s.counter import *" % test_set
39
40     exec(ToDo, globals())
41
42     is_participating_time_evolution = zeros(n - 5)
43     t = linspace(0, 1, n - 5)
44     ylim(0, 1)
45     title('Time Evolution of Participation Rate')
46     for j in range(0, m): # iterating over different seeds
47         for k in range(0, n - 5): # iterating over time steps
```

```

48     filename = 'out_%s_%s' % (j, k)
49     exec("from %s.%s import *" % (test_set, filename),
50           globals()) # importing data for given seed and time steps
51     is_participating_time_evolution[k] += sum(isParticipating) / (
52         500 * 50)
53
54     if (test_set == 'control'):
55         linewidth_ = 2
56     else:
57         linewidth_ = 1.2
58
59     plot(t, is_participating_time_evolution, colors[i], linewidth=linewidth_,
60          label=test_set)
61     ylabel('Participation Rate')
62
63
64 if __name__ == '__main__' and __package__ is None:
65     import sys, os.path as path
66
67     # Make sure we can import the out.py files
68     sys.path.append(path.dirname(path.dirname(path.abspath(__file__))))
69
70     # isParticipating: iterate over the configs listed in configs.sim
71     with open("configs.sim") as f:
72         next(f)
73         figure(figsize=(7, 5))
74         k = 0
75         for test_set in f:
76             analyse_is_participating(test_set.strip(), k)
77             k += 1
78
79         legend(prop={'size': 8}, fancybox=True, ncol=3, loc=1)
80         xlabel('Time')
81         savefig('isParticipating_timeEvolution.png', dpi=600)
82
83     # Median Danger: iterate over the configs listed in configs.sim
84     with open("configs.sim") as f:
85         next(f)
86         figure(figsize=(7, 5))
87         k = 0
88         for test_set in f:
89             analyse_median_danger(test_set.strip(), k)
90             k += 1
91
92         legend(prop={'size': 8}, fancybox=True, ncol=3, loc=1)
93         xlabel('Time')
94         savefig('medianDanger_timeEvolution.png', dpi=600)

```

F.2: Creation of Phase Diagrams

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy import interpolate
4
5
6 def phase_diagram(x, y, danger):
7     # levels to be considered safe/dangerous
8     safe_danger = 0.4
9     danger1 = 0.9
10    danger2 = 1.4
11
12    # interpolates the data to get a full grid
13    grid_x, grid_y = mgrid[0:99:200j, 0:99:200j]
14    points = (x, y)
15    grid_danger = interpolate.griddata(points, danger, (grid_x, grid_y),
16                                         method='cubic')
17
18    plt.figure()
19    # plots danger diagram
20    im_d = plt.imshow(grid_danger, extent=(0, 99, 0, 99), origin='upper',
21                      cmap='hot', vmin=safe_danger, vmax=danger2 + 0.2)
22    plt.axis('off')
23    cbar_d = plt.colorbar(im_d, fraction=0.046, pad=0.04, orientation='vertical',
24                          ticks=[safe_danger, danger1, danger2])
25    cbar_d.ax.set_yticklabels(['Low', 'Medium', 'High'])
26    plt.title('Danger Level')
27
28    plt.show()
29
30
31 # This program shows danger levels in a phase
32 # diagram as an average of a set of 'out.py' files
33
34 # Average function which looks for two equals points
35 def average_(points2, d2):
36     global continuous_danger_level_
37     global points
38     continuous_danger_level_ = np.append(continuous_danger_level_, d2)
39     points_[0] = np.append(points_[0], points2[0])
40     points_[1] = np.append(points_[1], points2[1])
41
42
43 # Imports and averages the data-set from the out files
44 def analyse2(test_set, time):
45
46     # Import the data from the out.py files
47     to_do = "from %s.counter import *" % test_set
48     exec(to_do, globals())
49
```

```

50 global points_
51 global continuous_danger_level_
52 exec("from %s.out_0_0 import *" % (test_set), globals())
53 points_ = [x, y]
54 continuous_danger_level_ = continuousDangerLevel
55
56 # Iterate over the seeds
57 for i in range(0, m):
58     try:
59         filename_ = 'out_%s_%s' % (i, time)
60         exec("from %s.%s import *" % (test_set, filename_), globals())
61     except ModuleNotFoundError:
62         break
63
64     # Average over the seeds
65     average_(x, y), continuousDangerLevel)
66
67 phase_diagram(points_[0], points_[1], continuous_danger_level_)
68
69
70 if __name__ == '__main__' and __package__ is None:
71     import sys, os.path as path
72
73     sys.path.append(path.dirname(path.dirname(path.abspath(__file__))))
74
75     # Iterate over all the config files in configs.sim
76     with open("configs.sim") as f:
77         next(f)
78         for test_set in f:
79             time = int(input('Choose time -> '))
80             analyse2(test_set.strip(), time)

```
