MCS Project
Cellular automata: Fire evacuation simulation

Urlich Icimpaye, Jakob Häggström, Oscar Jacobson

May 2022

1 Introduction

This project aims to use a simple cellular automation model presented in the paper *Cellular automaton model for evacuation process with obstacles* [1] to simulate expected behaviours and gather insights about the event of an evacuation of a room. This is done firstly in the same way as in the paper [1] to gain a baseline understanding of the model. Secondly some parameters were adjusted to get some understanding of how the model operates under different circumstances. Lastly an extension in the form of a person-to-person interaction was added to further investigate some properties and applications of the model. The cellular automata only has a few simple rules, but despite this, it does remarkably well at describing the general characteristics of an evacuation.

The model aims to simulate the real life situation of evacuating a room. This is done by dividing the room up in equally sized squares and have them all represent some aspect of the real life event. The squares are represented as doors, floor space, walls and obstacles that together makes the base upon which the simulations are carried out. The empty and walkable spaces are all indexed to replicate a field directed towards the available doors helping the simulated people move towards an exit.

2 Model selection

The model aims to replicate a room filled with pedestrians walking towards an exit. The room will be a N x N dimensional grid of pixels where each pixel represents the space taken up by one person. In [3] referenced in [1] a person in a crowded room takes up around a 0.4 meter by 0.4 meter area, each square in our model therefore also represents a 0.4m by 0.4m area in the room. Assuming the average walking speed of a person to be 1 m/s the Δt in our simulations is set to be 0.4 seconds per timestep.

2.1 Model floor

The cellular automata was simulated on a N x N grid of pixels, called the floor. The pixels are individually assigned a state depending on the timestep and defining rules of the model. There are walls, doors, obstacles and floor space. Every floor space is assigned a number depending on its location with respect to doors. A door is a floor space with the number 1 assigned to it. A wall is a floor space with the number 500 assigned to it. A obstacle, like a wall, is a floor space with the number 500 assigned to it. Every floor space can in simulations host a singular person per timestep. This, in theory, includes walls and obstacles, but as to be explained this will never happen due to the rules of the model.

Every room or instance of a simulation is started out with generating a floor. In the floor the locations of all doors and obstacles will firstly be decided upon. The algorithm will then assign numbers to every empty cell depending on its location relative to the placed doors. The generation of numbers in the floor field take obstacles and walls into account.

In [1] there were two different floor measures taken into consideration. One using distances in only two dimensions and one using euclidean distances. The two different measurements have some effect on the simulations and the differences is discussed in [1]. A conclusion drawn in that paper states that as we allow diagonal movement over cells, the most accurate measurement of movement makes use of euclidean distances. The addition of euclidean distances make a difference in for example, moving diagonally (taking one step left and one step down) compared to taking two steps in the same direction. In real life the Pythagorean theorem describes the measure of diagonal length and will be included in a simplified way. [4]

The implemented algorithm for the generation of a floor executes as following:

1. The chosen coordinates of doors are given the value 1.
2. The chosen coordinates of obstacles are given the value 500.
3. All the vertically or horizontally neighbouring floor spaces are given a value of $N + 1$.
4. All the diagonally neighbouring floor spaces are given a value of $N + \gamma$.
5. If a floor space is already assigned a value, the lowest value is chosen.
6. Starting at the doors and iterating outwards, this is repeated until all spaces are filled.
7. The walls are created around the existing floor spaces and given the values 500.

Because of our choice of euclidean distances, γ is chosen to be 1.5 which is the value used in [1] and is a great fit as the Pythagorean theorem suggest that the diagonal distance should be $\sqrt{2} \approx 1.41$.

	7,5	8	8,5	9	9,5	10	11	12	13	14	15	16	17	18	19	20	21	
	6,5	7	7,5	8	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5	16,5	17,5	18,5	19,5	20,5	
	5,5	6	6,5	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
	4,5	5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5	16,5	17,5	18,5	19,5	
	3,5	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5	16,5	17,5	18,5	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5	16,5	17,5	18,5	
	3,5	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
	4,5	5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5	16,5	17,5	18,5	19,5	
	5,5	6	6,5	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
	6,5	7	7,5	8	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5	16,5	17,5	18,5	19,5	20,5	
	7,5	8	8,5	9	9,5	10	11	12	13	14	15	16	17	18	19	20	21	

Figure 1: Standard 14 x 17 floor plan without obstacles, with floor field showing.

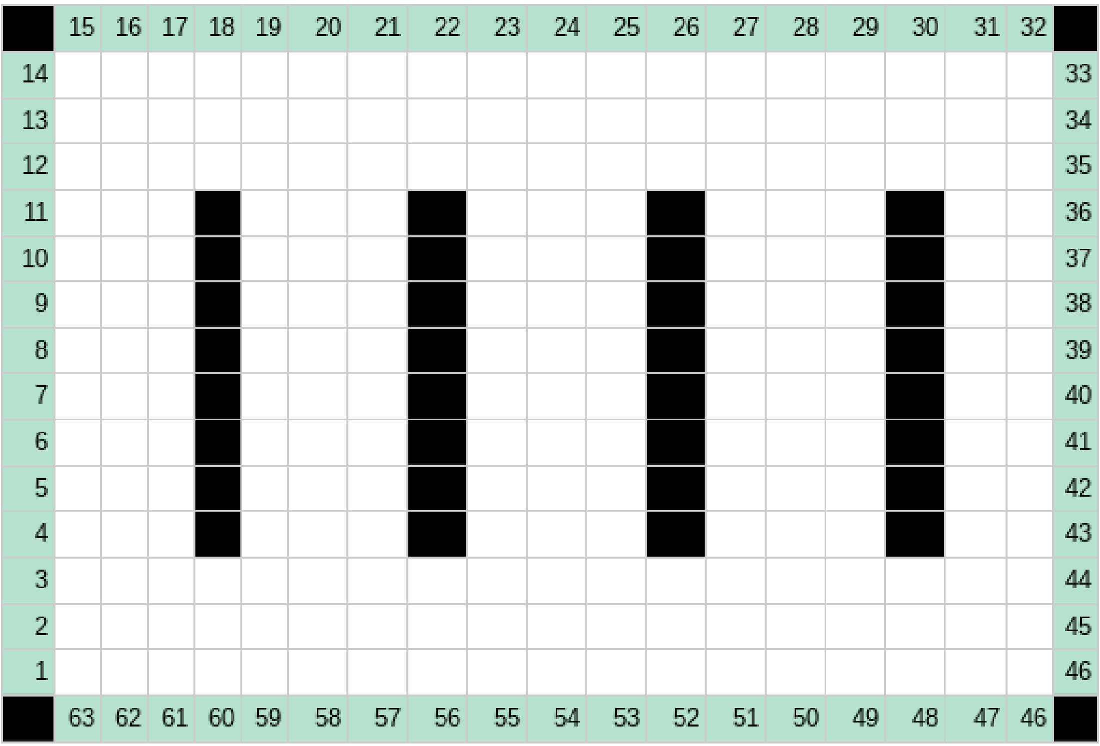

Figure 2: Standard 14 x 18 floor plan with obstacles, with door positions showing.

2.2 Model behaviour

At the start of every simulation, after the floor is generated, every empty floor space has a chance to get a person. These people follow a set of rules as in [1]. The basic rules are:

- At every iteration every person decides where to move.
- With no disturbance, a person will move to the neighbouring cell with the lowest assigned value.
- If two neighbouring cells have the same value, the choice between them is randomized.
- A person cannot move into an occupied floor space.
- If two people are trying to move into the same space two random numbers are drawn, the winner with the smallest number moves and the loser stays in the same place.
- Every person has a 5% chance to not move at all (Because of panic).

In the original paper, a panic factor was simulated by making every person have a 5% chance to not move at all. This rule will later be examined in the section *Modifying a parameter* as well as replaced and reworked in the section *Extension* where we examine panic and person-to-person interactions.

3 Simulation results

3.1 Room without obstacles

An example of a simulation can be found in *gif1.gif.*

3.2 Room with obstacles

An example of a simulation can be found in *gif2.gif.*

3.3 General findings

As seen in the previous sections, the behaviour of the model can be very easily mapped. There are a few assumptions previously made about the model that easily can be changed. For example, the current timestep Δt is set to be 0.4 seconds. If this does not match with the reality of a very similar situation the only thing needed to be changed is the assumed Δt. Other examples of simple tuning points of the model is, the rate at which people exit a door, the size taken up by every person or the speed at which a person walks in the case of an emergency. This is one of the big advantages of using a simple model like the automata used in this project.

Furthermore, we can see in *gif1.gif* that the individuals tend to group around the exit door since there is a limit of how many individuals that can exit the room per tick. This makes the door width interesting in this case, since a wider door might enable a higher exit flow. Also in *gif2.gif*, we can see that individuals that are initially behind an obstacle that is perpendicular to a door have an significant disadvantage of escaping. This makes the door position a parameter of interest when having a dense furnished room as in *gif2.gif*, since there might be placements that are more optimal than others.

A second very obvious parameter to examine is the effect panic has on the evacuation time. Without any implemented panic the model is by definition deterministic. Any disturbances in the simulations caused by added randomness will therefore effect the evacuation times.

4 Modifying a parameter

As far as simulating a full evacuation of a room the simulations up until now serves as a good baseline. These should for maximum accuracy and relevancy be compared to actual evacuation times of similar "simple" rooms to calibrate the assumed parameters of the model, like Δt and the assumed distance between squares. The simulations in this section was performed with a 40% likelihood of any empty square to be occupied by a pedestrian. This works out to be about 100 people in total. And they were repeated 10 times for each configuration.

4.1 Results of panic parameters

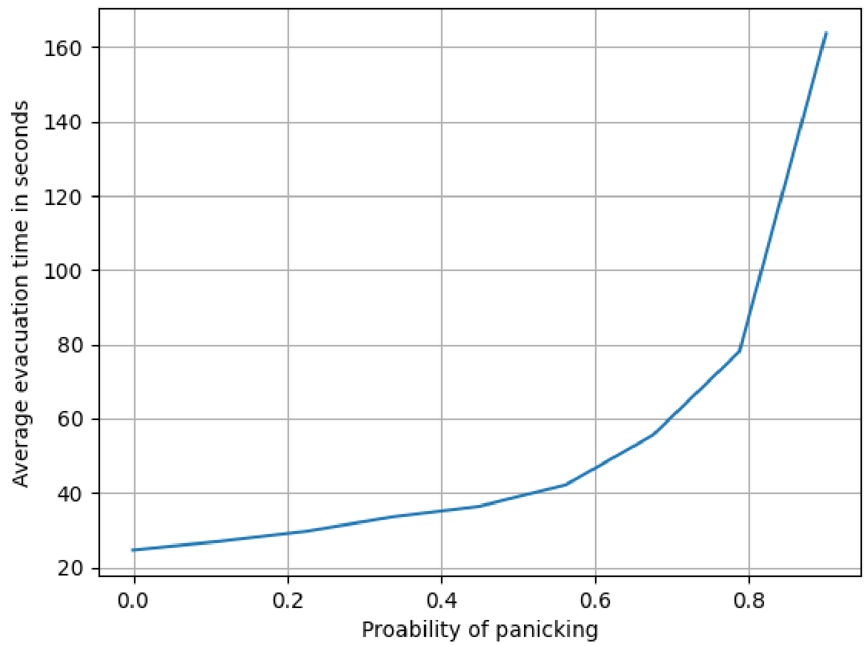

Figure 3: Average evacuation time for 100 people while varying "simple" panic parameter. Using every tick as 0.4 seconds.

4.2 Results of door width

As discussed in section 3.3, the width of the door is of interest. The door width was varied between 1 to 14 squares.

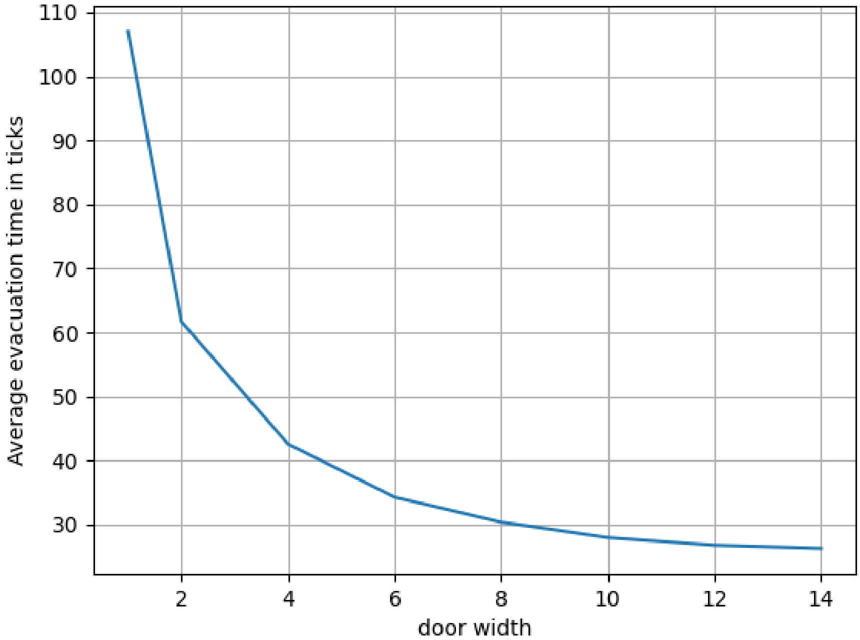

Figure 4: door width vs evacuation time

As seen in figure 4, the evacuation time seems to decrease with an decreasing door width.

4.3 Results of door position in furnished room

In this section the door position was varied as visualised in figure 2.

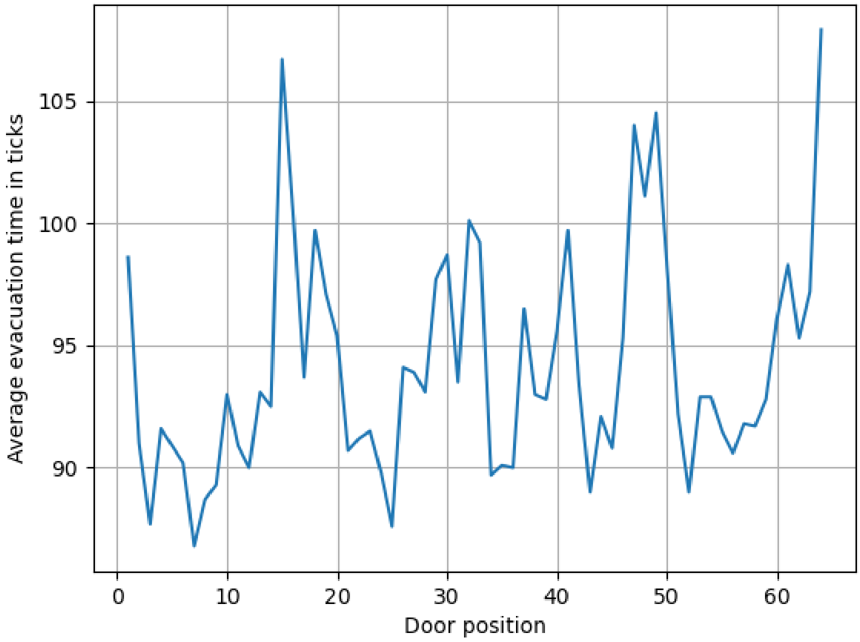

Figure 5: Result of average simulation time versus the position of the door.

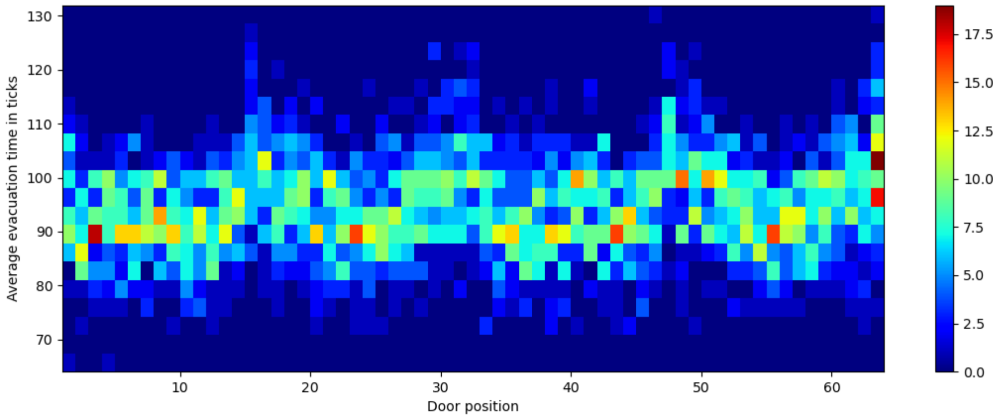

Figure 6: Phase transition plot of the actual simulation time versus the position of the door.(There is a typo on the y-label, it is the actual evacuation time, not the average.

As seen in figure 5, the evacuation time seems to be shorter if you place the door in a central square on any wall (Number 8 and number 24 are best). This makes sense as symmetry would help the simulation to allow a more even flow of people. People might still be walking towards an unoccupied door if its placed in a bad spot which would cause time to be wasted.

5 Extension

To further examine the model a set of rules for panic was introduced. Replacing the 5% chance to not move at all, the new rules were as follows:

- Every person can have three states, *Panicking*, *Calming* or *Neutral*.
- A *panicking* person will ether spread panic with a probability λ or stop panicking with a probability $1 - \lambda$.
- If a *panicking* person gets to spread panic that person will choose one neighboring person that is neutral to spread it to. And the *panicking* person continues to remain in it's position.
- If a *panicking* stops panicking, that person becomes neutral.
- A *neutral* person can only become *panicking* if a neighbouring person that's *panicking* chooses them. Otherwise they continue to moves accordingly with the rules stated in 2.2.
- A *calming* person cannot become *panicked*.
- A *calming* person will cause one panicking neighbouring person to become *neutral* in every iteration.

5.1 The impact of λ

These simulations was performed in a similar manner as in section 3.1, where the room was empty and the population was initiated in the same manner. The conditions for the different states were 50% panicking, 20 % calming and the rest were neutral. An example simulation is shown in *gif3.gif*, where the red squares are panicking individuals, green calming and blue neutral. The behaviour of the simulations for varying λ was further investigated, and the evacuation time was recorded 50 times for each parameter.

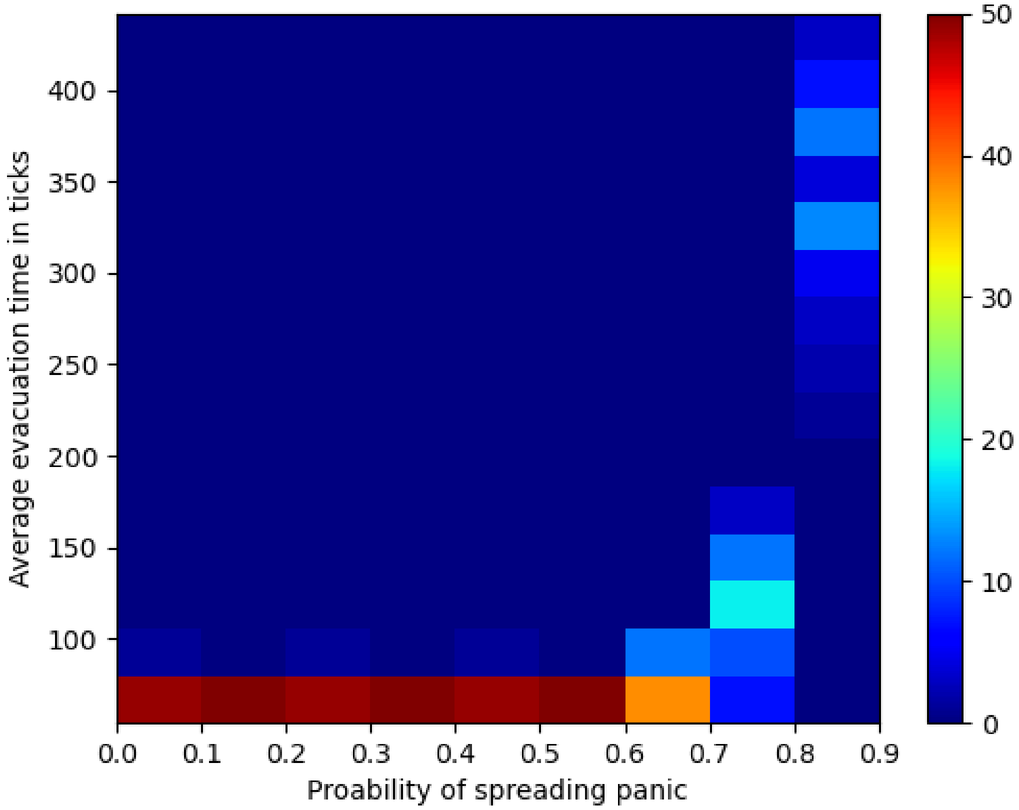

Figure 7: Phase transition plot of evacuation time and λ. (There is a typo on the y-label, it is the actual evacuation time, not the average.)

6 Conclusions

The basic model serves as an excellent baseline for a project like this, as it is very simple and have many parameters that are assumed and therefore very easy to calibrate to better make it follow real life. If a simulation is too fast the Δt can easily be changed. If a room, obstacle or door behaves too small or too large compared to reality the pixels are easily changeable and the assumed distance between them can be changed to match reality. As our project was set out to explore the model itself, the fitting to real world scenarios has not been carried out more than the initial assumptions made.

Our experiments showed very similar results to that of the original paper [1] in the initial stages of testing. Showing a very similar trend when examining the width of doors as well as the effect of obstacles and "simple" panic. The width and the position of the door seemed to have a significant impact on the result, which might be valuable knowledge if examined further. Building onto this model, adding the extension of a person-to-person interaction and expanding on the concept of panic in the model. Though this extension did not show great differences comparing to the original model. The spreading of panic increases evacuation time as seen in figure 7 which is also hinted at by figure 3. The extension would require an improvement in order to properly model interaction with the environment as well as interactions between people. An example of this would be to implement an agent-based simulation that was discussed in *Agent-based simulation of affordance-based human behaviors in emergency evacuation* [5]. Which allows fore more in depth concepts and parameters to be explored.

References

[1] Cellular automaton model for evacuation process with obstacles, A.Varas, M.D.Cornejo, D.Mainemer, B.Toledo, J.Rogan, V.Muñoz, J.A.Valdivia. Available from: https://www.sciencedirect.com/science/article/pii/S0378437107003676

[2] Cellular Automata Lecture, Fiona Skerman, Uppsala University. Available from: https://fskerman.github.io/2022:L2.pdf

[3] C. Burstedde, K. Klauck, A. Schadschneider, J. Zittartz, Simulation of pedestrian dynamics using a two-dimensional cellular automaton, Physica A 295 (2001) 507–525.

[4] Euclidean distance, Wikipedia, 2022-06-01. Available from: https://en.wikipedia.org/wiki/Euclidean_distance

[5] Agent-based simulation of affordance-based human behaviors in emergency evacuation, J. Joo et al. Available from: https://www.sciencedirect.com/science/article/pii/S1569190X12001724

Figure Appendix

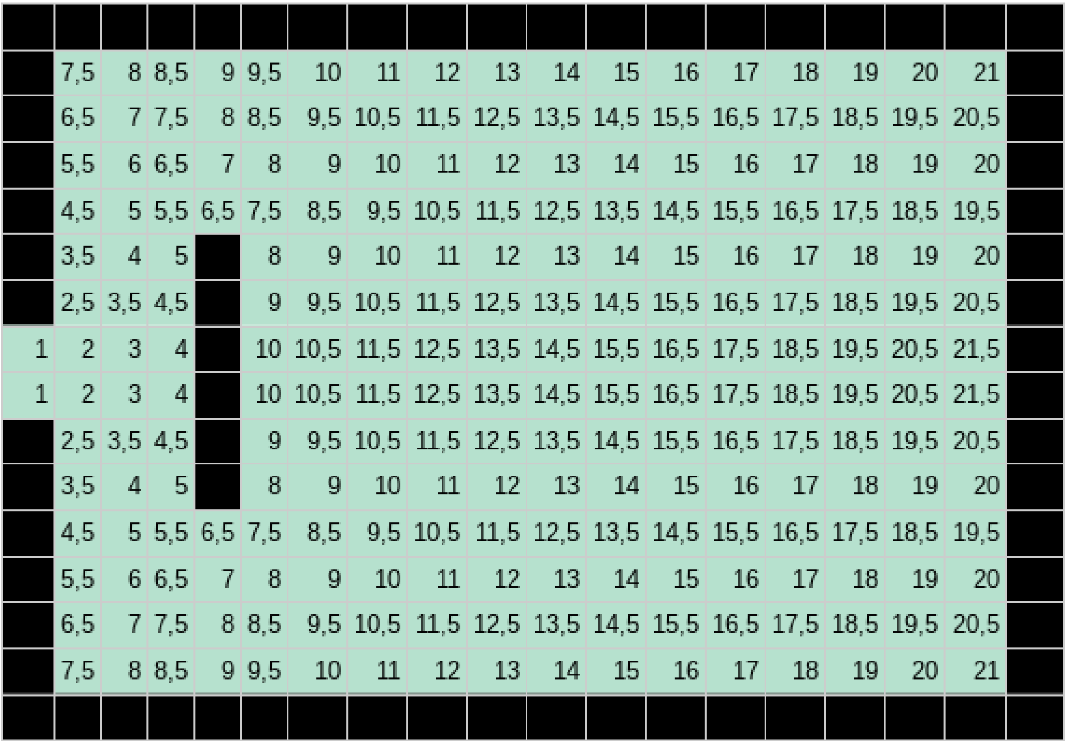

Figure 8: Standard 14 x 17 floor plan with obstacles, with floor field showing.

Code Appendix

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from celluloid import Camera
import os


MAIN_DIR = os.path.split(os.path.abspath(__file__))[0]


def set_elements_rec(mesh, row, col):

    def filter_func(el, mesh = mesh):

        if (el[0] < 1 or el[1] < 1 or el[0] > mesh.shape[0] - 2 or el[1] > /
            mesh.shape[1] - 2):

            return False

        if (mesh[el[0],el[1]] == 1 or mesh[el[0],el[1]] == 500):

                return False

        return True
```

```


    diag_ind = [(row - 1, col - 1), (row - 1, col + 1), (row + 1, col - 1), (row + /
        1, col + 1)]
    hor_ind = [(row - 1, col), (row, col + 1), (row, col - 1), (row + 1, col)]
    diag_ind = list(filter(filter_func, diag_ind))
    hor_ind = list(filter(filter_func, hor_ind))
    neighbor_ind = diag_ind + hor_ind
    set_elements(mesh, row, col, diag_ind, lamb = 1.5)
    set_elements(mesh, row, col, hor_ind, lamb = 1)



    while True:

        new_neighbours = []
        mesh_old = mesh.copy()
        for temprow, tempcol in neighbor_ind:

            diag_ind = [(temprow - 1, tempcol - 1), (temprow - 1, tempcol + 1), /
                (temprow + 1, tempcol - 1), (temprow + 1, tempcol + 1)]
            hor_ind = [(temprow - 1, tempcol), (temprow, tempcol + 1), (temprow, /
                tempcol - 1), (temprow + 1, tempcol)]

            try:
                diag_ind = list(filter(filter_func, diag_ind))
                hor_ind = list(filter(filter_func, hor_ind))
            except:
                raise ValueError(f"{(temprow, tempcol)}")
            templs = diag_ind + hor_ind
            set_elements(mesh, temprow, tempcol, diag_ind, lamb = 1.5)
            set_elements(mesh, temprow, tempcol, hor_ind, lamb = 1)

            for tempind in templs:
                if tempind not in neighbor_ind and tempind not in new_neighbours:
                    new_neighbours.append(tempind)

        bool_matr = (mesh_old == mesh)
        bool_matr = bool_matr.flatten()
        if all(bool_matr):
            break

        neighbor_ind = new_neighbours


def set_elements(mesh, row, col, indicies, lamb = 1):

    for ind in indicies:

        if mesh[ind[0],ind[1]] == 500:
            continue
        elif mesh[ind[0],ind[1]] !=0:
            mesh[ind[0],ind[1]] = min([mesh[ind[0],ind[1]], mesh[row,col] + lamb])
        else:

            mesh[ind[0],ind[1]] = mesh[row,col] + lamb

def draw_mesh_art(rows,cols, door_pos, lamb):
    mesh = 500 * np.ones(((rows + 2), cols + 2))
    mesh[1:rows + 1, 1:cols + 1] = 0
```

```

    for i in range(4, 16 + 4, 4):

        mesh[4:12,i] = 500

    for door in door_pos:
        mesh[door[0], door[1]] = 1

    for ax in range(4):

        doorposition_vec = np.where(mesh[:,0] == 1)[0]
        if len(doorposition_vec) != 0:

            for door in doorposition_vec:

                if mesh[door,1] != 0:

                    mesh[door, 1] = min([mesh[door, 1], mesh[door, 0] + 1])
                else:
                    mesh[door, 1] = mesh[door, 0] + 1

                if mesh[door + 1, 1] != 0:

                    mesh[door + 1, 1] = min([mesh[door + 1, 1], mesh[door, 0] + lamb])
                else:
                    mesh[door + 1, 1] = mesh[door, 0] + lamb

                if mesh[door - 1, 1] != 0:

                    mesh[door - 1, 1] = min([mesh[door - 1, 1], mesh[door, 0] + lamb])
                else:
                    mesh[door - 1, 1] = mesh[door, 0] + lamb



                set_elements_rec(mesh, door, 1)


        mesh = np.rot90(mesh)

    return mesh


def request_move(neigbourhood, people, indicies, curr_pos):

    temp_neigbourhood = neigbourhood.copy()
    temppeople = people.copy()

    possible_moves = []
    floor_val = []

    for i, row in enumerate(temppeople):
        for j, col in enumerate(row):
            if col == 0 or indicies[i][j] == curr_pos:
                possible_moves.append(indicies[i][j])
                floor_val.append(temp_neigbourhood[i, j])

    comb_list = [[dist, ind] for dist, ind in sorted(zip(floor_val, /
        possible_moves))]
    min_val = [comb_list[0]]
```

```
    for i in range(1, len(comb_list)): # If there are several squares with the /
        same distance.
        if min_val[0][0] == comb_list[i][0]:
            min_val.append(comb_list[i])
        else:
            break


    if len(min_val) > 1: # If there are several squares with the same distance.

        ind = np.random.choice(list(range(len(min_val))))

        return (min_val[ind][1][0], min_val[ind][1][1])

    else:

        return (min_val[0][1][0],min_val[0][1][1])

def panic(prob):
    return np.random.choice([0,1], size = 1, p = [1 - prob, prob])


def handle_requests(people_new, requests):

    requests = sorted(requests, key = lambda x: x[1]) # Sort the requests on the /
        destination

    ind = 0
    nr_requests = len(requests)


    while ind < nr_requests:
        temp_req_lst = [requests[ind]]
        ind += 1


        while ind < nr_requests and requests[ind][1] == temp_req_lst[0][1]:
            temp_req_lst.append(requests[ind])

            ind += 1



        temp_num_req = len(temp_req_lst)
        if temp_num_req > 1: # One gets to move there are several requests for one /
            square.

            rand_ind = np.random.choice(list(range(temp_num_req)))

            lucky_man = temp_req_lst.pop(rand_ind)

            people_new[lucky_man[1][0], lucky_man[1][1]] = 1
            people_new[lucky_man[0][0], lucky_man[0][1]] = 0


            for unlucky_man in temp_req_lst:

                people_new[unlucky_man[0][0], unlucky_man[0][1]] = 1

```

```
      else:
         people_new[temp_req_lst[0][1][0], temp_req_lst[0][1][1]] = 1
         if temp_req_lst[0][0] != temp_req_lst[0][1]:
            people_new[temp_req_lst[0][0][0], temp_req_lst[0][0][1]] = 0

   return people_new



def step(mesh, people, panic_prob, door_pos):

   requests = []
   dims = people.shape
   for door in door_pos:
      people[door[0], door[1]] = 0

   for i in range(1, dims[0] - 1):
      for j in range(1, dims[1] - 1):

         if people[i, j] == 0 or people[i, j] == 500:
            continue

         if panic(panic_prob):
            requests.append([(i,j), (i,j)])

         else:
            indicies = [[[x,y] for y in range(j-1 , j + 2)] for x in range(i-1, i /
                + 2)]

            move_request = request_move(mesh[i-1:i+2,j-1:j+2], /
                people[i-1:i+2,j-1:j+2], indicies, [i,j])
            requests.append([(i,j), move_request])


   return handle_requests(people, requests)


def run_sim(mesh_dim, panic_prob, time_steps, door_pos,lamb, gen_gif = False, /
    filename = None):
   np.random.seed()

   floor_mesh = draw_mesh_art(mesh_dim[0],mesh_dim[1], door_pos, lamb)
   people_mesh = 500 * np.ones((mesh_dim[0] + 2,mesh_dim[1] + 2), dtype = int)
   people_mesh[1:mesh_dim[0] + 1, 1:mesh_dim[1] + 1] = 0

   for i in range(4, 16 + 4, 4):

      people_mesh[4:12,i] = 500

   people_mesh[people_mesh == 0] = np.random.choice([0,1], size = /
       people_mesh[people_mesh == 0].shape, p = [0.60,0.40])


   for door in door_pos:
      people_mesh[door[0], door[1]] = 0

   T = time_steps
   t = 0
   if gen_gif:
      fig = plt.figure()
```

```
        camera = Camera(fig)
        plot_mesh(people_mesh, fig)
        camera.snap()

    while t < T:

        people_mesh = step(floor_mesh, people_mesh, panic_prob, door_pos)
        t += 1
        if gen_gif:
            plot_mesh(people_mesh, fig)
            camera.snap()

        if len(people_mesh[people_mesh == 1]) == 0:
            break

    if gen_gif:
        animation = camera.animate()
        animation.save(os.path.join(MAIN_DIR,'gifs',filename))

    return t


def plot_mesh(mesh, fig):
    color_dict = {0: np.array([255, 255, 255]), 1: np.array([255, 0, 0]), 500: /
        np.array([0, 0, 0])}
    res = [[color_dict[int(col)] for col in row] for row in mesh]
    res = np.array(res)

    if not fig:
        plt.figure()

    plt.imshow(res)

    if not fig:
        plt.show()




if __name__ == "__main__":

    # case left side: increasing col from zero, divide row.
    # case bottom: decreaseing rows from max to zero. divide columns
    # case top: increasng rows form zero. divide cols
    # case right: decreasing cols from zero, divide rows.
    # door_pos = [[4,0], [2,11]]
    # mesh = draw_mesh_art(14,18, door_pos, 1.5)



    # tot_doors = [[[7,0]]]

    # ind1 = 6
    # ind2 = 9
    # i = 1
    # while ind1 > 0:
    # templist = tot_doors[i] + [[ind1, 0]] + [[ind2, 0]]
    # ind1 -= 1
    # ind2 += 1
    # i += 1
```

```
    # tot_doors.append(templist)


    tot_doors = []

    for i in range(14, 0, -1):
        tot_doors.append([[i, 0]])
    for i in range(1,19):
        tot_doors.append([[0, i]])

    for i in range(1,15):
        tot_doors.append([[i, 19]])

    for i in range(18,0, -1):
        tot_doors.append([[15, i]])



    # panic_prob_vec = np.linspace(0, 0.9, 9)
    panic_prob_vec = [0.05]
    num_rep = 10
    filename = 'example_furnished3.gif'
    evac_time_vec = []

    for doors in tot_doors:
        tempvec = []
        for i in range(num_rep):
            evacuation_time = run_sim((14,18), panic_prob_vec[0], 2000, doors, 1.5)
            tempvec.append(evacuation_time)
        evac_time_vec.append(np.mean(tempvec))

    # evacuation_time = run_sim((14,18), panic_prob_vec[0], 500, [[7,0], [7,19]], /
        1.5, True, filename)

    dor = np.array(list(range(len(evac_time_vec)))) + 1

    plt.figure()
    plt.plot(dor, evac_time_vec)
    plt.xlabel('Door position')
    plt.ylabel('Average evacuation time in ticks')
    plt.grid(True)
    plt.show()
```

Listing 1: The code for the first model.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from celluloid import Camera
import os

MAIN_DIR = os.path.split(os.path.abspath(__file__))[0]
color_dict = {0: np.array([255, 255, 255]), 500: np.array([0, 0, 0])}
state_dict = {0: np.array([0, 0, 255]), 1: np.array([255, 0, 0]), 2: np.array([0, /
    255, 0])}

class person:

    def __init__(self, pos, state):
        self.pos = pos
        self.state = state
```

```
def set_elements_rec(mesh, row, col):

    def filter_func(el, mesh = mesh):

        if (el[0] < 1 or el[1] < 1 or el[0] > mesh.shape[0] - 2 or el[1] > /
            mesh.shape[1] - 2):

            return False

        if (mesh[el[0],el[1]] == 1 or mesh[el[0],el[1]] == 500):

                return False

        return True


    diag_ind = [(row - 1, col - 1), (row - 1, col + 1), (row + 1, col - 1), (row + /
        1, col + 1)]
    hor_ind = [(row - 1, col), (row, col + 1), (row, col - 1), (row + 1, col)]
    diag_ind = list(filter(filter_func, diag_ind))
    hor_ind = list(filter(filter_func, hor_ind))
    neighbor_ind = diag_ind + hor_ind
    set_elements(mesh, row, col, diag_ind, lamb = 1.5)
    set_elements(mesh, row, col, hor_ind, lamb = 1)



    while True:

        new_neighbours = []
        mesh_old = mesh.copy()
        for temprow, tempcol in neighbor_ind:

            diag_ind = [(temprow - 1, tempcol - 1), (temprow - 1, tempcol + 1), /
                (temprow + 1, tempcol - 1), (temprow + 1, tempcol + 1)]
            hor_ind = [(temprow - 1, tempcol), (temprow, tempcol + 1), (temprow, /
                tempcol - 1), (temprow + 1, tempcol)]

            diag_ind = list(filter(filter_func, diag_ind))
            hor_ind = list(filter(filter_func, hor_ind))
            templs = diag_ind + hor_ind
            set_elements(mesh, temprow, tempcol, diag_ind, lamb = 1.5)
            set_elements(mesh, temprow, tempcol, hor_ind, lamb = 1)

            for tempind in templs:
                if tempind not in neighbor_ind and tempind not in new_neighbours:
                    new_neighbours.append(tempind)

        bool_matr = (mesh_old == mesh)
        bool_matr = bool_matr.flatten()
        if all(bool_matr):
            break

        neighbor_ind = new_neighbours


def set_elements(mesh, row, col, indicies, lamb = 1):

```

```
    for ind in indicies:

        if mesh[ind[0],ind[1]] == 500:
            continue
        elif mesh[ind[0],ind[1]] !=0:
            mesh[ind[0],ind[1]] = min([mesh[ind[0],ind[1]], mesh[row,col] + lamb])
        else:

            mesh[ind[0],ind[1]] = mesh[row,col] + lamb

def draw_mesh_art(rows,cols, door_pos, lamb):
    mesh = 500 * np.ones(((rows + 2), cols + 2))
    mesh[1:rows + 1, 1:cols + 1] = 0

    for i in range(4, 16 + 4, 4):

        mesh[4:12,i] = 500


    for door in door_pos:
        mesh[door[0], door[1]] = 1

    for ax in range(4):

        doorposition_vec = np.where(mesh[:,0] == 1)[0]
        if len(doorposition_vec) != 0:

            for door in doorposition_vec:

                if mesh[door,1] != 0:

                    mesh[door, 1] = min([mesh[door, 1], mesh[door, 0] + 1])
                else:
                    mesh[door, 1] = mesh[door, 0] + 1

                if mesh[door + 1, 1] != 0:

                    mesh[door + 1, 1] = min([mesh[door + 1, 1], mesh[door, 0] + lamb])
                else:
                    mesh[door + 1, 1] = mesh[door, 0] + lamb

                if mesh[door - 1, 1] != 0:

                    mesh[door - 1, 1] = min([mesh[door - 1, 1], mesh[door, 0] + lamb])
                else:
                    mesh[door - 1, 1] = mesh[door, 0] + lamb



                set_elements_rec(mesh, door, 1)


        mesh = np.rot90(mesh)

    return mesh

def request_move(neigbourhood, people, indicies, curr_pos):

    temp_neigbourhood = neigbourhood.copy()
    temppeople = people.copy()
```

```python

    possible_moves = []
    floor_val = []

    for i, row in enumerate(temppeople):
        for j, col in enumerate(row):
            if col == 0 or indicies[i][j] == curr_pos:
                possible_moves.append(indicies[i][j])
                floor_val.append(temp_neigbourhood[i, j])

    comb_list = [[dist, ind] for dist, ind in sorted(zip(floor_val, possible_moves))]
    min_val = [comb_list[0]]

    for i in range(1, len(comb_list)): # If there are several squares with the same distance.
        if min_val[0][0] == comb_list[i][0]:
            min_val.append(comb_list[i])
        else:
            break


    if len(min_val) > 1: # If there are several squares with the same distance.

        ind = np.random.choice(list(range(len(min_val))))

        return (min_val[ind][1][0], min_val[ind][1][1])

    else:

        return (min_val[0][1][0],min_val[0][1][1])

def panic(prob):
    return np.random.choice([0,1], size = 1, p = [1 - prob, prob])


def handle_requests(people_new, requests,person_vec):

    requests = sorted(requests, key = lambda x: x[1]) # Sort the requests on the destination

    ind = 0
    nr_requests = len(requests)


    while ind < nr_requests:
        temp_req_lst = [requests[ind]]
        ind += 1


        while ind < nr_requests and requests[ind][1] == temp_req_lst[0][1]:
            temp_req_lst.append(requests[ind])

            ind += 1



        temp_num_req = len(temp_req_lst)
        if temp_num_req > 1: # One gets to move there are several requests for one square.
```

```
            rand_ind = np.random.choice(list(range(temp_num_req)))

            lucky_man = temp_req_lst.pop(rand_ind)

            people_new[lucky_man[1][0], lucky_man[1][1]] = 1
            people_new[lucky_man[0][0], lucky_man[0][1]] = 0
            temp_ind = find_person_at(person_vec, lucky_man[0])
            person_vec[temp_ind].pos = lucky_man[1]


            for unlucky_man in temp_req_lst:

                people_new[unlucky_man[0][0], unlucky_man[0][1]] = 1

        else:
            people_new[temp_req_lst[0][1][0], temp_req_lst[0][1][1]] = 1
            if temp_req_lst[0][0] != temp_req_lst[0][1]:
                temp_ind = find_person_at(person_vec, temp_req_lst[0][0])
                person_vec[temp_ind].pos = temp_req_lst[0][1]
                people_new[temp_req_lst[0][0][0], temp_req_lst[0][0][1]] = 0

    return people_new, person_vec


def find_person_at(person_vec, pos):

    for i, person in enumerate(person_vec):
        if person.pos == pos:
            return i

def step(mesh, people, panic_spead_prob, door_pos, person_vec):

    requests = []
    state_update_vec = []
    dims = people.shape
    for door in door_pos:
        people[door[0], door[1]] = 0

    num_persons = len(person_vec)

    pind = num_persons - 1

    while pind > 0:
        temppos = person_vec[pind].pos

        if [temppos[0], temppos[1]] in door_pos:
            person_vec.pop(pind)

        pind -= 1

    for i in range(1, dims[0] - 1):
        for j in range(1, dims[1] - 1):

            if people[i, j] == 0:
                continue

            p_ind = find_person_at(person_vec, (i,j))
            indicies = [[[x,y] for y in range(j-1 , j + 2)] for x in range(i-1, i + /
                2)]
```

```
            if person_vec[p_ind].state == 1:

                requests.append([(i,j), (i,j)])
                if panic(panic_spead_prob):

                    possible_moves = []

                    for indx in indicies:
                        for indy in indx:
                            if people[indy[0],indy[1]] != 500 and /
                                 people[indy[0],indy[1]] != 0 and indy != [i,j]:

                                possible_moves.append(indy)


                    if len(possible_moves) > 0:
                        tempind = np.random.choice(list(range(len(possible_moves))))
                        spread = possible_moves[tempind]
                        state_update_vec.append([(spread[0], spread[1]), 1])
                else:
                    state_update_vec.append([(i, j), 0])

            else:

                if person_vec[p_ind].state == 2:

                    possible_moves = []

                    for indx in indicies:
                        for indy in indx:
                            if people[indy[0],indy[1]] != 500 and /
                                 people[indy[0],indy[1]] != 0 and indy != [i,j]:

                                possible_moves.append(indy)


                    if len(possible_moves) > 0:
                        tempind = np.random.choice(list(range(len(possible_moves))))
                        calm = possible_moves[tempind]
                        state_update_vec.append([(calm[0], calm[1]), 0])


                move_request = request_move(mesh[i-1:i+2,j-1:j+2], /
                     people[i-1:i+2,j-1:j+2], indicies, [i,j])
                requests.append([(i,j), move_request])

    for pos, status in state_update_vec:

        p_ind = find_person_at(person_vec, pos)

        if person_vec[p_ind].state != 2:

            person_vec[p_ind].state = status


    return handle_requests(people, requests, person_vec)


def run_sim(mesh_dim, panic_prob, time_steps, door_pos,lamb, init_panic, /
     init_calm, gen_gif = False, filename = None):
```

```
    np.random.seed()

    floor_mesh = draw_mesh_art(mesh_dim[0],mesh_dim[1], door_pos, lamb)
    people_mesh = 500 * np.ones((mesh_dim[0] + 2,mesh_dim[1] + 2), dtype = int)

    for door in door_pos:
          people_mesh[door[0], door[1]] = 0

    people_mesh_c = people_mesh.copy()
    people_mesh_c[1:mesh_dim[0] + 1, 1:mesh_dim[1] + 1] = 0

    people_mesh[1:mesh_dim[0] + 1, 1:mesh_dim[1] + 1] = np.random.choice([0,1], /
        size = mesh_dim, p = [0.60,0.40])
    indx,indy = np.where(people_mesh == 1)

    people_indices = np.concatenate((indx.reshape(-1,1),indy.reshape(-1,1)), axis /
        = 1)

    person_vec = []

    for ind in people_indices:
       state = np.random.choice([0,1,2], p = [1 - init_panic - init_calm, /
           init_panic, init_calm])
       person_vec.append(person((ind[0], ind[1]), state))


    T = time_steps
    t = 0
    if gen_gif:
       fig = plt.figure()
       camera = Camera(fig)
       plot_mesh(people_mesh_c, person_vec, fig)
       camera.snap()

    while t < T:

       people_mesh, person_vec = step(floor_mesh, people_mesh, panic_prob, /
           door_pos, person_vec)
       t += 1
       if gen_gif:
          plot_mesh(people_mesh_c, person_vec, fig)
          camera.snap()

       if len(people_mesh[people_mesh == 1]) == 0:
          break

    if gen_gif:
       animation = camera.animate()
       animation.save(os.path.join(MAIN_DIR,'gifs',filename))

    return t


def plot_mesh(mesh, person_vec, fig):

    res = [[color_dict[int(col)] for col in row] for row in mesh]
    res = np.array(res)

    for person in person_vec:
       temppos = person.pos
```

```
        res[temppos[0], temppos[1]] = state_dict[person.state]


    if not fig:
        plt.figure()

    plt.imshow(res)

    if not fig:
        plt.show()




if __name__ == "__main__":

    # case left side: increasing col from zero, divide row.
    # case bottom: decreaseing rows from max to zero. divide columns
    # case top: increasng rows form zero. divide cols
    # case right: decreasing cols from zero, divide rows.
    # door_pos = [[4,0], [2,11]]
    # mesh = draw_mesh_art(14,18, door_pos, 1.5)


    door_pos = [[8,0], [7,0]]
    num_rep = 50
    filename = 'sim3.gif'
    evac_time_vec = []
    panic_prob_vec = np.linspace(0, 0.9, 9)
    init_panic = 0.5
    init_calm = 0.2
    # evacuation_time = run_sim((14,18), panic_spread, 300, door_pos, 1.5, /
        init_panic, init_calm, True, filename)



    for freeze_prob in panic_prob_vec:
        tempvec = []
        for i in range(num_rep):
            evacuation_time = run_sim((14,18), freeze_prob, 2000, door_pos, 1.5, /
                init_panic, init_calm)
            evac_time_vec.append([freeze_prob,evacuation_time])


    # plt.figure()
    # plt.plot(panic_prob_vec, evac_time_vec)
    plt.show()
    tot_res = np.array(evac_time_vec)
    plt.hist2d(tot_res[:,0], tot_res[:,1], bins=(9,15), cmap=plt.cm.jet)
    plt.ylabel("Average evacuation time in ticks")
    plt.xlabel("Proability of spreading panic")
    plt.colorbar()
    plt.show()
```

Listing 2: The code for the extended model.