ETHlogo

Lecture with Computer Exercises:

Modelling and Simulating Social Systems with MATLAB

Project Report

|  |
| --- |
| Crowd Simulation  … |

Samuel Oberholzer & Philipp Lütolf

Zürich

11.05.2014

IMPORTANT

You MUST include the ETH declaration of originality here; it is available for download on the course website or at

http://www.ethz.ch/faculty/exams/plagiarism/index\_EN;

It can be printed as pdf and should be filled out in handwriting.

Agreement for free-download

We hereby agree to make our source code of 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.

|  |  |
| --- | --- |
| Samuel Oberholzer | Philipp Lütolf |

1. Table of content
2. Abstract

The goal of simulation is to model big crowds in different locations and identify the dangerous spots. Our model is programmed in Matlab, is agent – based and in continuous space. The physics are based on the ‘social force model’ from Helbing. To identify the dangerous spots, the density of the people is computed and illustrated in the simulation. In this report two different locations are considered.

We expect the most critical spots to be obstacles, corners, intersections and other bottlenecks where the crowd is disrupted. Therefore we chose to consider a crossroad setting and a curve. The locations are simulated with and without obstacles to identify the differences. To conclude our model we compared our results to projects from previous semesters which are mainly implemented as cellular models.

1. Individual contributions

The work on this project was shared evenly between both authors because we always worked together.

1. Introduction and Motivations

Big events with a lot of people bear a great risk of mass panic. At the love parade in Duisburg 2010, 21 people died and 541 people were injured due to a crowd disaster. One critical factor of such tragedies is the arrangement of the location. We want to model big crowds and identify the dangerous spots in a specified environment. We try to validate the results of projects from earlier semesters with a different approach. We state the following questions:

RESEARCH QUESTIONS

1. Description of the Model
   1. Social force model

The model we chose to use is based on the ‘social force model’ developed by D. Helbing and P. Molnar ADDIN EN.CITE <EndNote><Cite><Author>P. D. Helbing</Author><Year>1995</Year><DisplayText> [1]</DisplayText><record><ref-type name="Journal Article">17</ref-type><contributors><authors><author>P. D. Helbing</author></authors></contributors><titles/><title>Social force model for pedestrian dynamics</title><periodical/><volume>51</volume><dates><year>1995</year><pub-dates/></dates></record></Cite></EndNote> [1]. It describes the “social forces” acting on a pedestrian in a crowd. Each pedestrian in a crowd can be represented with a point in space, the speed of the pedestrians can be described by the equation and the acceleration by .

D. Helbing and P. Molnar added a fluctuation term which include random variations of the behaviour.

One term of describes the driving force which accelerates the pedestrian towards the desired velocity , another term consists of the repulsion force from other pedestrians and obstacles and the last force outlines attractive effects .

* 1. Driving Force (Acceleration force)

The driving force accelerates the pedestrian towards the desired velocity

* 1. Pedestrian Interactions
  2. Boundary Interactions

1. Implementation

The social force model is implemented in Matlab as an agent – based model in continuous space. The implementation consists of three parts. First, the testModel.m file where all the computation is done and the data is saved. Secondly, a simulate function, that plots the saved data and makes a video out of it. And third, the maxPeopleOnSquare function, which visualizes the simulation data for analyzing it.

6.1 Initialization and Simulation (the testModel.m file)

The implementation of the core file (testModel.m) roughly follows the Pseudocode shown below:

—Initialization—

Set walls and waypoints

Set start positions of agents

—Simulation—

FOR each frame of the simulation

FOR every pedestrian in the system do

Set up next destination and calculate desired direction

Calculate acceleration force and influence on velocity

FOR each wall element do

Check distance to element

Calculate wall force of closest element and influence on velocity

END

FOR every other agent of the matrix do

Calculate pedestrian force and influence on velocity

END

Update position according to calculated velocity

save agent in a updated matrix

END

set matrix to updated matrix

save data for plotting

END

6.1.1 Initialization

The implementation allows to specify and run the simulation on different maps, only depending on the initialization of the walls, waypoints and agents. Two situations were specified to investigate our research questions.

**6.1.1.1 Walls and Waypoints**

Walls are specified by a start- and endpoint. So every wall needs to store x and y position for two points, means four values per wall. Obstacles can be placed into the scene as well by placing them as walls.

Because it’s unlikely, that all agents head to a single point in space, a new concept of waypoints is introduced. Here, waypoints are more like „waylines“ and stored in the same format as walls. It’s a line that fixes the next destination of an agent. The shortest path to this line is used to calculate the desired direction of an agent. If an agent reaches the last waypoint, he respawns at his starting area.

**6.1.1.2 Agents**

TODO: IMG AGENT

There is one matrix containing all the agents of the simulation. Every agent consists of eight different values. These are the agent’s position and its current velocity, its desired velocity which is Gaussian-distributed at 1.3+-0.1 m/s and its type that determines to which group of agents the agent belongs and therefore where its starting area and destination is. Further, which waypoint the agent currently is aiming for, and what its average speed is.

The positions and waypoints for the initialization depend on the situation. Here, the following two situations were specified:

**6.1.1.3 The ‚cross‘ situation**

TODO: IMG CROSS SITUATION

The walls are defined as seen in the picture above. Agents are initialized randomly distributed in a 5x5 area on the left side and on the top (outside the picture). Their goal is to reach the other side of the cross-way. To a achieve this, the way points were set as seen in the picture.

The crowd flow in this situation is investigated with and without the obstacles seen in the picture and the outcome is compared.

**6.1.1.4 The ‚curve‘ situation**

TODO: IMG CURVE SITUATION

In this situation, the goal of the agents is to follow the street and make it to the other end. One half starts at the bottom-left end and the other half at the top-right. Here also obstacles were added to compare the crowd flow with obstacles to the one without. The waypoints are set as seen in the picture to overcome the obstacles.

6.1.2 Simulation

In order to realistically simulate movement of agents in continuous space, they need to be updated a lot. Time between these updates was chosen at 0.05s, which corresponds to 20 frames per second. If the time between the updates is chosen too high, the steps of the agents are larger and important factors influencing their movement could be skipped. For example if an agent moves towards a wall and the update occurs shortly before the agent is pushed back by the wall force, the agent could be past the wall in the next update.

DT IMG

In every step, all the agents have to be updated. These updates depend on the current waypoint they are aiming for, the walls, and the positions of other agents. So for every agent is done the following things:

**6.1.2.1 Setting the next destination**

The distance from the agent to the next waypoint is computed with *vectorFromWall*. Is the distance small enough, the next waypoint is set as new destination.

If it was the last waypoint, the agent has reached its destination and is re-initialized at the starting area. Like this, the flow of incoming agents can be maintained without having to create new ones and therefore performance is tuned. The desired direction is computed again using *vectorFromWall* so the acceleration force can be computed.

The function *vectorFromWall* takes a waypoint and the agent’s position and gives back the normalized vector pointing to this waypoint and the distance to it.

**6.1.2.2 The acceleration force**

The acceleration force is computed using *accelerationF*. Then the force is added to the current velocity of the agent.

The function *accelerationF* takes the desired direction, current velocity, average speed and the desired velocity and returns the calculated acceleration force according to formula (TODO================).

**6.1.2.2 The wall force**

First, the nearest wall is searched using *vectorFromWall* again, then the wall force is computed using *wallF*. The wall force is added to the velocity afterwards

The function *wallF* takes the distance to a wall and the normalized vector pointing to it. According to formula (TODO===============) the wall force is computed.

**6.1.2.3 The pedestrian force**

Every other pedestrian has a little influence on the agent. Therefore, the pedestrian force resulting from every other agent is computed using *pedestrianF* and added to the velocity of the agent. Note that the loop over all the agents uses the matrix from the last step, so updates on all agents are made simultaneously.

The function *pedestrianF* computes the pedestrian force using the positions of two agents as arguments according to the formula (TODO==============).

**6.1.2.4 Position update and saving data**

After all the forces contributing to the new velocity have been computed, the position of the agent is updated. Additionally, the matrix containing information about how many people are in a square meter is updated.

After all the agents have been updated, the whole matrix is updated simultaneously. The positions of the agents are stored separately so that the simulation run can be plotted later on.

6.2 Plotting the data and making a video (simulate.m)

*function simulate(filename,mode,savevideo)*

In this function, the saved data from the file ‚filename’ is visualized. The x positions of all the agents are plotted against there y position. Combined with the plot of the walls matrix this results in the visual simulation of the situation. If ,mode‘ is passed a 1, the whole plot is provided with a background picture. This background picture shows how many people are standing in each square meter. The whole plot is saved to a video file called ,savevideo’.

6.3 Overview of the result (maxPeopleOnSquare.m)

*function maxPeopleOnSquare(inputfile)*

This function creates a figure with two subplots to summarize the data of the plot. Input data is loaded from the file ,inputfile’. The first plot visualizes what the maximum number of people standing in a square meter was at each frame of the video. The second plot shows for each square meter of the map, what the maximum number of people on it was throughout the whole video.

7. Simulation Results and Discussion

1. Summary and Outlook
2. References

ADDIN EN.REFLIST

|  |  |
| --- | --- |
| [1] | P. M. D. Helbing, "Social force model for pedestrian dynamics," *Physical Review E,* vol. 51, no. 5, 1995. |