Lecture with Computer Exercises: Modelling and Simulating Social Systems HS18

Modelling Human Trail Systems and Implementation in a Real World Problem

Lukas Bircher (13-939-566),
Robin Stähli (13-914-239),
Christoph Martin (14-928-741)

Zürich, 8th December 2018

Contents

List of Figures

List of Tables

Abstract

Goal of this thesis is to reproduce the results and model from the paper *Modelling the evolution of human trail systems* from Dirk Helbing et al. and to review if the model is capable of reproducing the trail systems that emerged in the real world. The program which was derived from the mathematical model, was able to reproduce the exact same scenario as shown in the paper. Therefore it was used on more complex structures such as a real world park. The goal was to implement boundary conditions similiar to the reality, to get results which reflect the trails which have formed in real life. This was possible to some extent, the simulation showed different behaviour with different visibility factors, as it should. But the exact same trail system did not emerge. This might be due to small errors in pedestrian distribution. Further on, the complexity of the system is shown. Even slight changes are capable of alternating the whole trail system. This problem can be seen when the size of the simulated field changes. All parameters depend on the size of the field.

Overall the program is able to reproduce the reality quite good. Therefore it could be used as a clue to build new pathways in parks or cities. The downside is, that it is a computationally heavy program. If one wants to use it for a big system, it will take a long time.

1 Introduction

The code was developed in several different sessions. In the first two sessions we focused on building a functioning script, using the information given in the paper *Modelling the evolution of human trail systems* by Dirk Helbing, Joachim Keltsch and Péter Molnàr. All formulas used, were taken from the afore mentioned paper.
The following sessions were intended to implement a graphical response. The last step was, to show the principle of the code in a real world case.

The paper from Helbing et al. (1997) aims to describe and simulate so called self-organization phenomena of human beings, with mathematical models. This model is thought to help predict optimal paths for urban planning of cities, parks, malls and furthermore of emergency exits.
These cases can be simulated very realistically, as shown in previous studies. Helbing et al. showed in their paper, that pedestrians aim to take the shortest possible path. The effect of trail properties has little influence on the trail formation. This paper extended the typical approach to simulate trail systems to an *active walker model*, which also takes environmental changes into account and is therefore more precise [1].

2 Mathematical Model and Code

2.1 Mathematical Model

The mathematical model defines several functions to calculate the trail after a certain amount of iterations. One of the functions is the *ground structure* at a time and place t and r respectively. The ground structure factor represents how attracted a pedestrian is to a certain place on the ground. Path in parks or sidewalks are labeled with a very high ground structure factor $G(t, r)$.
The ground structure changes with the amount of pedestrians stepping on it. This increases the likelihood of another pedestrian choosing the same path. The amount of destruction per step on a groundfield is called *intensity*. Helbing et al. define the intensity as

$$I(r) * (1 - G(t, r)/G_{max}(r)) \tag{1}$$

where G_{max} is the upper limit of destruction of the trail. When the maximum is reached, this means that all vegetation (grass for example), is destroyed and nothing more is left to be destroyed. Therefore following pedestrians are attracted by the visible trail in the grass.
As the grass regrows, the trail disappears which might lead to a further modification of the trail system. The factor of regrow is called *weathering rate* or $1/T(r)$ where $T(r)$ is the durability of the trail. The higher $T(r)$, the slower the grass regrows and the smaller the weathering rate.
Change in ground structure is thus calculated as follows

$$\frac{dG(r, t)}{dt} = \frac{1}{T(r)}[G_0(r) - G(r, t)] + I(r)[1 - \frac{G(r, t)}{G_{max}(r)}] \sum_{\alpha} \delta(r - r_{\alpha(t)}) \tag{2}$$

where G_0 is defined as the starting condition of the field e.g. grass everywhere in the park. r_α is the position of the current pedestrian.
$\delta(r - r_\alpha)$ is the Dirac function. Its properties are, that it is either zero iff r is not equal to r_α and one, if they are equal.

To calculate, if a pedestrian is attracted to a certain segment of the trail, a function called *trail potential* is introduced. The potential

$$V_{tr}(r_\alpha, t) = \int d^2 r e^{\frac{-|r - r_\alpha|}{\sigma(r_\alpha)}} G(r, t) \tag{3}$$

depends on the distance of the pedestrian to the segment and also on the visibility. In the function, this is taken into account in the exponential factor. The trail potential is also a function of the ground condition, therefore depends on the amount of people already stepped on a certain field segment.

If there are no natural gradients like hills and we only focus on a planar field, then the walking direction can be calculated from two values. The first important value is the destination the pedestrian receives. Second value is the ground attraction or trail potential calculated beforehand. Important to note is, that the gradient of the trail potential needs to be normalized. The orientation is calculated as follows

$$e_\alpha(r_\alpha, t) = \frac{d_\alpha - r_\alpha + \nabla_{r_{\alpha,t}} V_{tr}(r_\alpha, t)}{|d_\alpha - r_\alpha + \nabla_{r_{\alpha,t}} V_{tr}(r_\alpha, t)|} \tag{4}$$

which is taken to calculate the equation of motion of each pedestrian

$$\frac{dr_\alpha}{dt} = v_\alpha^0 e_\alpha(r_{\alpha,t}) \tag{5}$$

where v_α^0 is the predefined velocity of the pedestrian.
Equation 1 through 5 were taken from the paper of Helbing et al. [1].

2.2 Implementation in the Code

The ground structure $G(r,t)$ is implemented as an array with the size of the park. The values in the array represent either areas with grass (value = 0) or areas with a fully developed trail (value = 1). Everything in between zero and one denotes a not fully evolved trail.
In figure 1, one cycle of the program can be seen. The position of a pedestrian is updated, therefore the ground structure has to be updated as well. Then the overall field is evaluated and the pedestrian "decides" which pixel he will choose in his next iteration.

This decision depends on his visibility and the attractiveness of closeby trails and the direction in where his destination is. The three vectors for destination, attractiveness of closeby trails and the thereof calculated displacement vector can be seen in figure 2. This orientation vector is calculated as in formula 4.

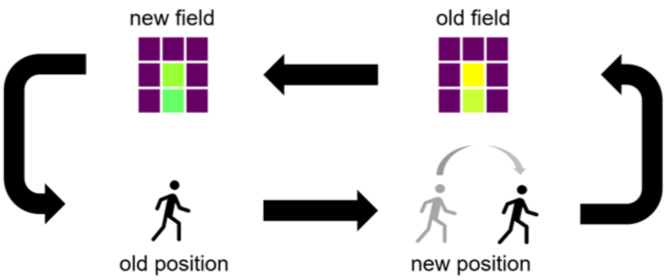


Figure 1: This image shows one iteration step of a single pedestrian

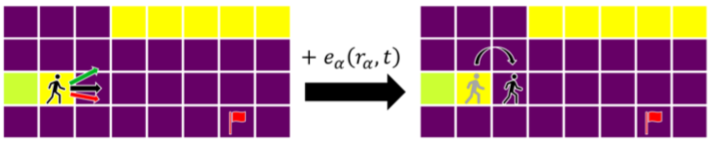


Figure 2: Vectors of displacement, attractiveness and destination

2.3 CPU Parallelization

As almost all new computers have multicore processors, it is obvious to parallelize the written program to speed it up and use the full potential. As this code is programmed in python, it is not that easy to rewrite the code for several cores. The reason for this is that python has a global interpreter lock implemented. This is usually a good thing, as it prevents harmful interactions between threads. But for this purpose, we need python to execute several statements at the same time [3].

Not all parts of the code have been parallelized, only the parts which need the most computing power. This is the equation 3. To adapt the program to the computer used (or the amount of cores available), it will get the amount of parallel cores the CPU has. With the multiprocessing library it is simple to parallelize a whole function or a loop. But before the parallelization, it is important to check if it is possible for this certain function. What is meant by this is, that if a function needs its previous values, you cannot parallelize it, as the calculations will be made on different cores.

2.4 Calculation on EULER

EULER is one of the scientific computer clusters of the ETH. Its purpose is to be used for computationally heavy programs such as image analysis, biological models and many different simulations [4].
As this simulation takes a long time on a four core computer, it makes sense to run it on the EULER cluster, where more than four cores can be used. For our calculations we used up to 32 cores simultaneously. According to the job info of the batch system of

euler, the workload was evenly spread over all cores, with a utilization of 90%.

2.5 Parameter Dependencies

To show the validity of the code it was tested on the same scenario as in the paper of Helbing et al.. A quadratic field consisting of 30 by 30 pixels and 3 starting points was used. If the code is correct, it should output results reaching form a direct way system to a minimal way system.
The output is very similar to the findings in the paper, it can therefore be assumed, that the code is correct. The result can be found in the table 1 on page 9.

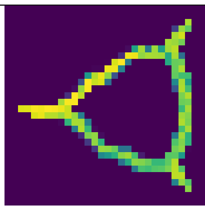
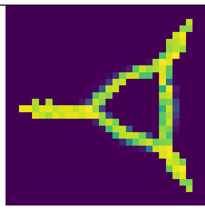
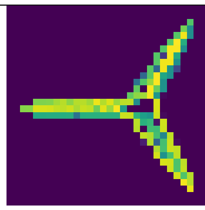
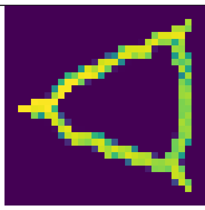
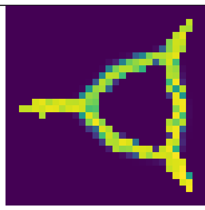
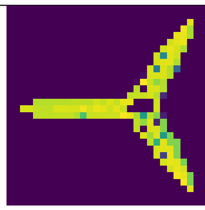
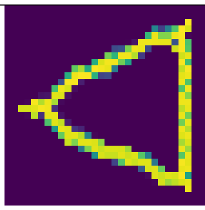
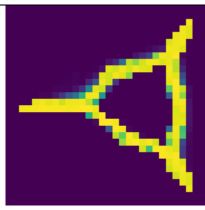
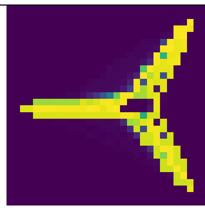
Durability / σ	1	2	4
250	 $\kappa = 87.5$	 $\kappa = 43.75$	 $\kappa = 21.875$
500	 $\kappa = 175$	 $\kappa = 87.5$	 $\kappa = 43.75$
1000	 $\kappa = 350$	 $\kappa = 175$	 $\kappa = 87.5$

Table 1: 3 x 3 Images with different settings for sigma (σ) and durability (T) and constantly same setting for intensity (I = 0.35)

In the paper, kappa is described as the sole influence on the system and therefore on the results. It is defined as $\kappa = I * T/\sigma$ [1]. However, the images in the table 1 suggest otherwise. The pictures on the diagonal axis, from the top left to the bottom right, have the same Kappa. Yet the images do not show the same path system, even after 4000 iterations, where the system is already in steady state.
This makes the variable definition more difficult because more independent variables have to be set.

3 Real World Application

3.1 Park Description

In the following simulations all adjustable ground parameters have been adjusted to the size of the park to get meaningful results. This includes the number of pedestrians, the durability and the intensity.

To compare the code results with real life conditions, a park was searched. It should show the typical human trail systems. Since parks in Switzerland seem to avoid these arbitrary trails with prebuilt concrete trails and paths, other countries were searched for better examples. A good park was found in Alcala de Henares, Spain [2].

The park fulfils most criterias for the code to work properly. Namely the park has a rectangular shape, the trails are nicely visible and there are no obstacles inside the park. The good visibility of the trail is probably owed to the dry weather, which does not allow the grass to grow back quickly. However there are small issues like a small house on the bottom left and a fence on the right bottom of the park as can be seen in the figure 3. The assumption regarding these issues is, that they are neglectable, what must be verified on the resulting trails of the program.

Figure 3: The image shows the obstacles in the park (house, parking space and fence) as well as the surrounding pavement

3.2 Park Implementation

The images of the park had to be preprocessed to work with our program, therefore had to be converted to grayscale. Inkscape was used to recreate the park trails and surrounding as a vector graphic, which was exported to a PNG image which can be read by the python script. The resolution was chosen to be 150 x 152 px, so small paths can still be seen but that the image and therefore the computing size is not too large. The process of the image processing can be seen in figure 4.

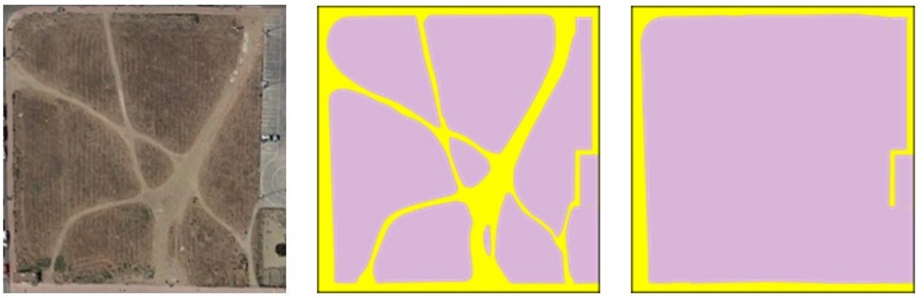


Figure 4: Image of the park on the left, colored base model with paths in the middle, colored base model without paths on the right side

Regarding the model including the paths, the house on the bottom left was ignored and marked as grass area. The pavement surrounds the park, except on the right side, since there is a fence set up which does not allow passing. The parking lot was split into two. One, the street where people can pass easily without any obstruction and two the parking lots itself, which will obstruct the pedestrians from passing, when a car is parked. Therefore the lots are marked as grass.
The roundabout at the end of the parking area was modelled as a thin paved way for pedestrians. Either of the base model images were used as the starting condition G_0 in the code.

3.3 Pedestrian Distribution

Starting points for the pedestrians were needed. The analysis of the surroundings can be seen in the figure 5 on page 12.

The pedestrian start and destination points are divided into 6 locations and each spawn location represents the access to a zone next to the park. The chosen park with its location in Spain was not accessible for monitoring and counting of pedestrians walking through the park. Therefore the analysis of the spawn points based on research on google maps [2] as well as on further information from a local resident who grew up in this area.
All zones except zone 5 are residential areas. This zone is a big athletic field and the main access to the local bus connections. Zone 3 is also important for the public transport system and represents the way to the train station. The residential area in zone 6 is small, however a preschool and a side entrance to the local supermarket (big white building between zone 1 and 6 in figure 5) are located in this zone which is the only supermarket nearby.
The fact that the small residential zone 6 contains the supermarket leaded to the approximation that zone 6 will be treated equally important as the three bigger residential zones (1, 2 and 4). On the other hand, zone 3 and 5 are more attractive because they are very important for the public transport. Furthermore as zone 5 contains the biggest

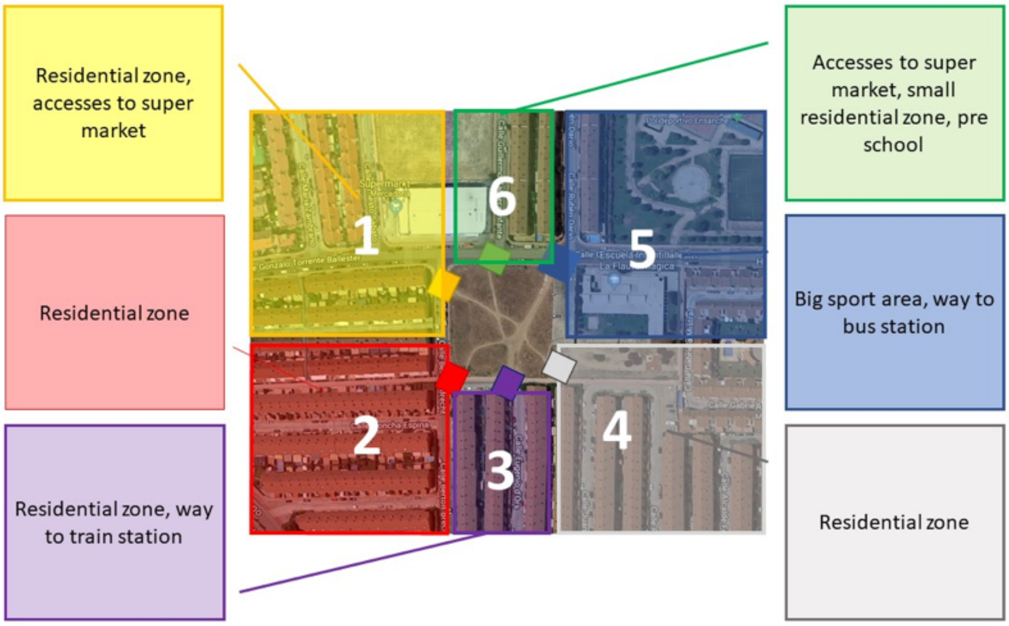


Figure 5: The surrounding of the park is subdivided into 6 zones. Each zone has its own access location depicted as a small square in the related colour, which represent the spawn points in the model [2].

public sport field in the surroundings, it gathers a higher importance comparing to the other zones. This led to the pedestrian distribution as shown in table 2. The amount of spawn points is the importance multiplied by factor 2.
To get a more realistic model, the spawn points are spread over an area, rather than a single pixel. This represents people who do not enter the park at a single point, but rather in a certain area.

Zone	Importance	Spawn Points
1	1	2
2	1	2
3	4	8
4	1	2
5	4	8
6	1	2

Table 2: Importance of different areas on the map of Alcala

4 Results and Discussion

To see whether the program builds similar paths as in the park in Alcala on its own, a plain field was input into it (right image of the figure 4 on page 11). The same sigmas as in the triangular case are used, but scaled to match the bigger field. It has a size of $150 \times 150px$. The durability was adapted to the bigger map as well. It has to be increased, so the paths do not decrease as fast as in the smaller map.
The images in the table 3 resulted after 8000 iterations. Overall, 10 pedestrians were walking on the field at the same time. Therefore a total amount of 80000 steps were made.

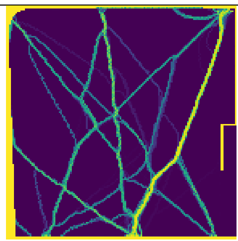
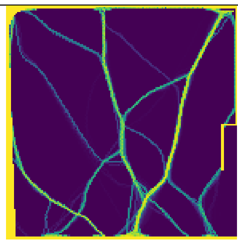
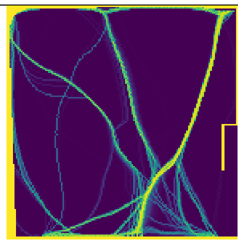
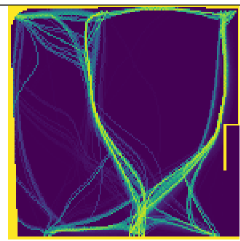
σ	1	3	6	12
Durab. 2000	 $\kappa = 700$	 $\kappa = 233.33$	 $\kappa = 116.66$	 $\kappa = 58.33$

Table 3: Trail formation on a plain field with dimensions and boundary conditions similar to the park in Alcala

The images in the table 3 show the transition between the direct way system on the left to the minimal way system on the right. It can be observed that the trails of the real park lay in between the $\sigma = 3$ and $\sigma = 6$. The trails built from scratch are similar but not identical to the real trail system.
This error might be due to the estimation of the distribution of pedestrians. The amount of people starting from one end or in one street was estimated, not counted. Therefore the small difference can be led back to this. The initially made assumptions, about the house, the fence and the parking space can be confirmed. In neither of the Images, a conflict can be found.

In a second simulation the goal was to find out, if the path system, as it exists on the field in Alcala, is a stable equilibrium. The picture in the middle of figure 4 on page 11 was taken as base image. The results can be seen in the table 4 on page 14. It can be observed that the calculated pedestrians did accept the pre-existing trails, if the sigma is around three. Above, minimal way systems form and below, direct way systems evolve. Therefore, the state which forms in reality is in between a direct and a minimal way system. This could mean that the park is not in a equilibria and will change further on.

Overall, it is rather hard to find good values which work together. The problem is, that one set of values does not work for different problems, nor does it work for different

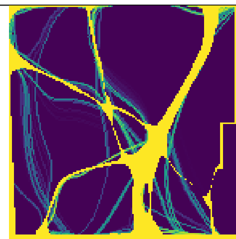
σ	1	3	6	12
Durab. 2000	 $\kappa = 700$	 $\kappa = 233.33$	 $\kappa = 116.66$	 $\kappa = 58.33$

Table 4: Trail formation on the field with pre defined real life paths

field sizes. The bigger the simulation, the more time is needed for calculation. Therefore it is harder to find matching parameters for bigger simulations. The best option is to find variables via brute force methods and simply test several different value pairs.
For the triangle, over 4000 value pairs were tested on the EULER cluster, until matching parameters were found. These values helped as a clue for further simulations.
The park in Alcala used another 200 iterations of variables to find good ones.

5 Summary and Outlook

The goal of the thesis was to reproduce the model and some of the results from the paper of Dirk Helbing et al. It was shown, that the model works when implemented correctly. It can even be used to simulate real world trail systems, which was shown with the park in Alcala.

One problem is the finding of parameter sets. As there are many parameters which can be adjusted, it can take a long time to find a correct set. It would be great to generate a lookup table for different scenarios and image sizes. This would reduce the computational power needed.

This model could be used as a guidance for an architect who has to plan a park. It can predict which routes will be chosen the most, therefore the paved pathways can be set up in these locations. This can help building new parks inside cities, arranging shelves in supermarkets or managing tables in big offices.

Further extensions to the code could be made by including a potential field. This could be used to model mountains or differences in altitude in small parks.
Another extension should tackle the problem of how to implement obstacles, which either restricts the vision of the pedestrians or blocks the path.

Bibliography

[1] Dirk Helbing, Joachim Keltsch & Péter Molnàr. *Modelling the evolution of human trail systems*. Macmillan Publishers Ltd, Nature, 1997.

Retrieved from URL

[2] Google Maps. *Park in Alcala de Henares, Spain*. 2018.
`https://www.google.com/maps/place/Alcalá+de+Henares,+Madrid,+Spanien/@40.4980571,-3.370361,150m/data=!3m1!1e3!4m5!3m4!1s0xd424bd8c669e60b:0x555eb1454b21723!8m2!3d40.4819791!4d-3.3635421`

[3] Sebastian Raschka. *An introduction to parallel programming using Python's multiprocessing module*. 2014.
`https://sebastianraschka.com/Articles/2014_multiprocessing.html`

[4] ETH Zurich. *Getting started with clusters*. 2016.
`https://scicomp.ethz.ch/wiki/Getting_started_with_clusters`

Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import random
import os
import matplotlib
from os import path
import time
from joblib import Parallel, delayed
import multiprocessing
import cv2
import timeit

run_time = time.strftime("%Y%m%d_%H%M") ## Time of Run of the programm, will be used for the name of the folder


##############################################################################
## To do first!
## 1.Write correct path 'imgpath' for load file of park and save data 'newpath'
## 2.Choose your parameter in the sequence of parameter
## 3.If you want simulate other parks than Alcala change respwan points (p0) in function respwan(pedestrians)
##############################################################################


##############################################################################
## Path to load and save file
## Write correct path 'imgpath' for load file of park and save data 'newpath'
## Example:

## Example for mac
imgpath = '/Users/username/folder/Alcala_green.png' ## loads file from park you want to evaluate
newpath = r'/Users/username/folder/images/{}'.format(run_time) ## saves all generated data in this folder

## For Windows

#imgpath = "C:\\Users\\Lukas\\Documents\\1_Schule\\Modelling_and_Simulating_Social_Systems\\Alcala_130xpx_137yps.png"##
# loads file from park you want to evaluate
#newpath = "C:\\Users\\Lukas\\Documents\\1_Schule\\Modelling_and_Simulating_Social_Systems\\Test_Save_Plot\{}".
```

```
# format(run_time) ## saves all generated data in this folder

##############################################################################


##############################################################################
## Parameters for run
## Choose your parameter in the sequence of parameter
iterations = 800 # amount of cycles
Intensity = 0.35 # Destruction of Trail (I)
Durability = 2000 # Growth rate / regeneration (T) The higher, the slower the
    grass grows
sigma = 12 # Visibility
v0 = 1 # Velocity of pedestrian
ped_num = 10 # Number of Pedestrians

##############################################################################


##############################################################################
## observe dimensions of park you want to evaluate and prepars pic for model
img = cv2.imread(imgpath,0) # read in file of park to matrix (grayscale)
Parkshape = img.shape #size of image
print("Pixel size of the park: ",Parkshape)
img = img/255 # color of pixel (RGB) to value between 0 to 1
invmat = np.ones([Parkshape[0],Parkshape[1]]) # Matrix with ones in size of
    immge
ParkMat = invmat-img # Invertion of image of park
Pedestrianrecord = np.zeros([Parkshape[0],Parkshape[1]]) # generats map were
    pedestrian have walked during the hole run

##############################################################################


##############################################################################
##initialization of data for main
pedestrians = np.ones((ped_num, 4)) # Array where the start/actual-x postition
    , start/actual-y postition,
# destination-x und destination-y position of pedestrian are saved
field = Parkshape # field area
G_0 = ParkMat # ground structure
G_akt = ParkMat # actual ground structure
G_max = np.asarray(np.ones((field[0], field[1]), dtype=float)) # max of ground
     structure
## intialization of data for multithreding process
num_cores = multiprocessing.cpu_count() # read in number of your cpu cores
```

```
print("Number of your cpu cores: ",num_cores)
##############################################################################


##############################################################################
## generates new folder to save data
if not os.path.exists(newpath):
    os.makedirs(newpath)
##############################################################################


##############################################################################
## generates a log file and write in all characteristics of the run
kappa = (Intensity * Durability) / sigma
Lambda = (v0 * Durability) / sigma

file = os.path.join(newpath, "Run_Characteristic.txt")
file1 = open(file, "w")
file1.write("Date and Time: {0}\n".format(run_time))
file1.write("Intensity: {0}\n".format(Intensity))
file1.write("Durability: {0}\n".format(Durability))
file1.write("Sigma: {0}\n".format(sigma))
file1.write("V0: {0}\n".format(v0))
file1.write("Kappa: {0}\n".format(kappa))
file1.write("Lambda: {0}\n".format(Lambda))
file1.write("Field Size: {0}\n".format(field))
file1.write("Number of Pedestrians: {0}\n".format(ped_num))
file1.write("Iterations: {0}\n".format(iterations))
file1.close()
##############################################################################


##############################################################################
## Function calculates equation of environmental changes and calculation of new ground structure
##
## Inputs: Intensitiy; Durability; G_0 (Ground structure); G_akt (actual field structure),
# G_max (max of ground structure); pedestrians
## Outputs: G_akt (actual field structure)
def dG(intens, durab, g0, g_ak, g_m, ped):
    d_g = np.asarray(np.zeros((field[0], field[1]), dtype=float)) # Array to save changes of ground structure
    d_g1 = 1 / durab * np.asarray(g0 - g_ak) # Calculates regeneration of ground structure
    g_ak = g_ak + d_g1 # Update actual ground structure
```

```
    i = 0
    while i < (ped.shape[0]): # for all pedestirans
        x = int(round(ped[i, 0]))
        y = int(round(ped[i, 1]))
        d_g[x, y] = intens * (1 - (g_ak[x, y] / g_m[x, y])) # Calculates
            distruction of pedestrians
        i = i + 1
        g_ak[x, y] = d_g[x, y] + g_ak[x, y]# Update of actual ground structure
    return g_ak
##############################################################################



##############################################################################
## Sub-Function calculates trail potential
##
## Inputs: x and y position to calculate trail potential; Sigma (Visibility);
    G_akt (actual field structure),
## Outputs: vtr (trail potential)
def d_r(x, y, G_ak, sig):
    vtr = 0
    for i in range(0, field[0]):
        for k in range(0, field[1]):
            dist = np.sqrt((i - x) ** 2 + (k - y) ** 2) # distance calculation
                for trail potential
            vtr = vtr + np.exp(-dist / sig) * G_ak[i, k] # trail potential
                calculation
    vtr = vtr/(field[0]*field[1]) #scaling to field size
    return vtr
##############################################################################


##############################################################################
## Sub-Function calculates derivation of trail potential in vertical direction
##
## Inputs: i (number of pedestrian to calculate derivation); pedestrian ;
    Sigma (Visibility);
# G_akt (actual field structure)
## Outputs: derivation of vtr (trail potential) in vertical direction
def vec_parallel_vt1(i, ped, G_ak, sig):
    x = int(round(ped[i, 0]))
    y = int(round(ped[i, 1]))
    Vtr1 = d_r(x + 1, y, G_ak, sig)
    Vtr3 = d_r(x - 1, y, G_ak, sig)
    dVtr1 = (Vtr1 - Vtr3) / 2
    return dVtr1
```

```
##############################################################################


##############################################################################
## Sub-Function calculates derivation of trail potential in horizontal
    direction
##
## Inputs: i (number of pedestrian to calculate derivation); pedestrian ;
    Sigma (Visibility);
# G_akt (actual field structure)
## Outputs: derivation of vtr (trail potential) in horziontal direction
def vec_parallel_vt2(i, ped, G_ak, sig):
    x = int(round(ped[i, 0]))
    y = int(round(ped[i, 1]))
    Vtr2 = d_r(x, y + 1, G_ak, sig)
    Vtr4 = d_r(x, y - 1, G_ak, sig)
    dVtr2 = (Vtr2 - Vtr4) / 2
    return dVtr2
##############################################################################


##############################################################################
## Sub-Function calculates vertical part of vector to destination of
    pedestrian
##
## Inputs: i (number of pedestrian to calculate destination vector);
    pedestrian
## Outputs: d_dest1(vertical part of vector to destination of pedestrian)
def vec_parallel_d1(i, ped):
    d_dest1 = ped[i, 2] - ped[i, 0]
    return d_dest1
##############################################################################


##############################################################################
## Sub-Function calculates horizontal part of vector to destination of
    pedestrian
##
## Inputs: i (number of pedestrian to calculate destination vector);
    pedestrian
## Outputs: d_dest2 (horizontal part of vector to destination of pedestrian)
def vec_parallel_d2(i, ped):
    d_dest2 = ped[i, 3] - ped[i, 1]
    return d_dest2
##############################################################################

```

```python

################################################################################
## Function is used to parallelize the calculation of the pedestrian direction and calls
# with multicore operataion the Sub-Functions
##
## Inputs: pedestrian ; Sigma (Visibility); G_akt (actual field structure)
## Outputs: Columne of all calculated vectors
def vec(ped, G_ak, sig):
    results1 = Parallel(n_jobs=num_cores)(delayed(vec_parallel_vt1)(i, ped, G_ak, sig) for i in range(0, ped.shape[0]))
    results2 = Parallel(n_jobs=num_cores)(delayed(vec_parallel_vt2)(i, ped, G_ak, sig) for i in range(0, ped.shape[0]))
    results3 = Parallel(n_jobs=num_cores)(delayed(vec_parallel_d1)(i, ped) for i in range(0, ped.shape[0]))
    results4 = Parallel(n_jobs=num_cores)(delayed(vec_parallel_d2)(i, ped) for i in range(0, ped.shape[0]))
    d_Vtr = np.column_stack((results1, results2)) #Stacks vector of trail potentials
    d_dest = np.column_stack((results3, results4)) #Stacks vector of destination
    return {"d_dest": d_dest, "d_Vtr": d_Vtr} #Columne of all calculated vectors
################################################################################


################################################################################
## Function calculates the effective vector of the pedestrian displacement
##
## Inputs: pedestrian ; Sigma (Visibility); G_akt (actual field structure); v0 (pedestrian velocity)
## Outputs: epdest (all pedestrians displacement vectors)
def d_pos(ped, G_ak, sig, v0):
    epdest = np.zeros((ped.shape[0], 2), dtype=float)
    result= vec(ped, G_ak, sig)
    d_dest = result["d_dest"]
    dVtr = result["d_Vtr"]
    for i in range(0, ped.shape[0]):
        length_dest = np.sqrt((d_dest[i, 0]) ** 2 + (d_dest[i, 1]) ** 2) # length destination vector
        length_pot = np.sqrt((dVtr[i, 0]) ** 2 + (dVtr[i, 1]) ** 2) #length trail potential vector
        if length_dest == 0:
            d_dest[i, :] = d_dest[i,:]
        else:
            d_dest[i, :] = d_dest[i, :] / length_dest #standardization of
```

```
            destination vector if >0
        dVtr[i, :] = dVtr[i, :] / length_pot #standardization of trail potential vector

        d_dest = d_dest * 1.5 # prioritization of destination vector to ensure that pedestrian reaches destination
        lengthl = np.sqrt((d_dest[i, 0] + dVtr[i, 0]) ** 2 + (d_dest[i, 1] + dVtr[i, 1]) ** 2) #length of combined
        # vector (destination + potential vector)
        epdest[i, 0] = (d_dest[i, 0] + dVtr[i, 0]) / (lengthl) * v0 # pedestrian vertical displacement vector
        epdest[i, 1] = (d_dest[i, 1] + dVtr[i, 1]) / (lengthl) * v0 # pedestrian horizontal displacement vector
    return epdest
###############################################################################


###############################################################################
## Function for respawning of the pedestrians when they reach their destination
## Attention respwan point are spezific for park
## Inputs: pedestrian
## Outputs: pedestrian (updated pedestrian array)
def respawn(ped):
    for i in range(0, ped.shape[0]):
        p_0 = [[3, 3], [6, 3],
               [Parkshape[0] - 3, 22], [Parkshape[0] - 3, 25],
               [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 12], [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 15],
               [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 18], [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 21],
               [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 12], [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 15],
               [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 18], [Parkshape[0] - 3, Parkshape[1] / 3 * 2 - 21],
               [Parkshape[0] - 3, Parkshape[1] - 13], [Parkshape[0] - 3, Parkshape[1] - 16],
               [3, Parkshape[1] - 3], [3, Parkshape[1] - 6], [3, Parkshape[1] - 9], [3, Parkshape[1] - 12],
               [3, Parkshape[1] - 3], [3, Parkshape[1] - 6], [3, Parkshape[1] - 9], [3, Parkshape[1] - 12],
               [3, Parkshape[1] / 5 * 2], [3, Parkshape[1] / 5 * 2 + 3]] # respwan points specifig for park alcala

        if (abs(round(ped[i, 0]) - round(ped[i, 2]))) < 1.5 and abs(round(ped[i, 1]) - round(ped[i, 3])) < 1.5: #when
```

```
            # pedestrian reach destination

            ped[i, 0], ped[i, 1] = random.choice(p_0) #random choice of location in p0 for start position
            ped[i, 2], ped[i, 3] = random.choice(p_0) #random choice of location in p0 for destination position
            while ((ped[i, 0] == ped[i, 2]) and (ped[i, 1] == ped[i, 3])): # prohibits that destination is equal
                # to start
                ped[i, 2], ped[i, 3] = random.choice(p_0)
    return ped
#############################################################################


#############################################################################
#############################################################################
## main from which all the functions are called
pic_number = 1 # image number
start = timeit.timeit() # start counter
for j in range(0, iterations):
    respawn(pedestrians) # respawn pedestrians that reched the destination
    start = time.perf_counter()
    G_akt = dG(Intensity, Durability, G_0, G_akt, G_max, pedestrians) # update the field matrix
    d_v = d_pos(pedestrians, G_akt, sigma, v0) # calculate walking direction of all pedestrians
    for i in range(0, pedestrians.shape[0]): # update the prositions of the pedestrians
        pedestrians[i, 0] = pedestrians[i, 0] + d_v[i, 0]
        pedestrians[i, 1] = pedestrians[i, 1] + d_v[i, 1]
        if j == (iterations-1):
            pedestrians[i, 0], pedestrians[i, 1] = [2,2]
        Pedestrianrecord[int(pedestrians[i, 0]), int(pedestrians[i, 1])] += 1.0 #Adds plus one at location of
        # the all pedestrians
    if j % 10 == 0: # saves trail image every 10 iterations
        plt.imsave(path.join(newpath, "State_{0}.png".format(pic_number)), G_akt) # saves image at location 'newpath'
        pic_number = pic_number + 1
    if j % 100 == 0: # saves absolulte/normalized trail image every 100 iterations
        print ("Saved Picture")
        Pedestrianrecordnorm = Pedestrianrecord / (np.amax(Pedestrianrecord))
        plt.imsave(path.join(newpath, "Pedcontroll_norm_{0}.png".format(j)), Pedestrianrecordnorm) # save
        # normalized trail image
```

```
        Pedestrianrecordnorm = np.clip(Pedestrianrecord, 0, 1)
        plt.imsave(path.join(newpath, "Pedcontroll_clip_{0}.png".format(j)),
            Pedestrianrecordnorm) # save
        # absolute trail image
end = timeit.timeit()
print ("Time used for calculation: ",end-start,"[s]")
##############################################################################
##############################################################################
```