Comparison of Models for Macroscopic Pedestrian Motion

Gianluca Oberreit and Vadim Silaev
(Dated: December 2025)

We explore 2 models for simulating pedestrian motion. The first is based on the deterministic social force model, and the second on a Monte Carlo model of motion. We implement and test these two models separately, and then compare the results of these models by using both of them to simulate the same situation. We conclude that the results of both models are fairly similar, indicating that the emergent behaviour of pedestrians in this case is not dependent on their individual motion. We also observe that the social force model is much more efficient than the Monte Carlo one, running orders of magnitude faster.

I. INTRODUCTION

Crowd dynamics represents a complex interdisciplinary field bridging statistical physics, sociology, and computer science. Understanding how groups of individuals move, interact, and organize in confined spaces has critical implications for public safety, urban planning, and emergency management. The intrinsic complexity of the problem leaves one without choice but to simplify human behavior to a physical or numerical model that can be analyzed. Existing reviews discuss treating large crowd systems with lattice-based, gas-kinetic, hydrodynamical models, cellular automata methods, and many others.

This particular report studies the implementation of two distinct models, useful for describing crowd dynamics: a fully deterministic social force model and Monte Carlo model for queuing scenarios.

This dual investigation allows us to compare two fundamentally different approaches: one deterministic and force-based, the other stochastic and statistical. Through systematic implementation, we assess each model's computational efficiency, parameter sensitivity, and ability to reproduce known crowd phenomena e. g. organized lane formation in corridors or emergent queuing behaviors around service points. The comparative analysis provides insights into when each modeling paradigm is most appropriate and what trade-offs exist between physical realism, computational cost, and predictive accuracy in crowd dynamics simulations.

II. SOCIAL FORCE MODEL

The social force model, primarily developed by D. Helbing and P. Molnár, assumes that pedestrians have encountered most situations before and have thus developed predetermined strategies to react in the most efficient way to a situation. Their movement can therefore be considered fully automatic and predictable, with each individual pedestrian changing its acceleration to maintain its average speed in its desired direction and avoid collisions.

The model implemented here follows the approach of Gareth William Parry's masters' thesis, which will from now on, in this section, be referred to as our GWP.

A. Theory

We model a pedestrian α of radius r_α as having a position $\vec{p}_\alpha(t) \in \mathbb{R}^2$, a destination $\vec{d}_\alpha \in \mathbb{R}^2$, a velocity $\vec{v}_\alpha(t) \coloneqq \frac{d\vec{p}_\alpha}{dt}$, and an acceleration (hereafter also called total force) $\vec{f}_\alpha(t) \coloneqq \frac{d\vec{v}_\alpha}{dt}$. The driving force is defined as follows:

$$\vec{f}_\alpha = \vec{f}_\alpha{}^0(\vec{v}_\alpha) + \vec{f}_{\alpha B}(\vec{p}_\alpha) + \sum_{\beta\neq\alpha} f_\beta(\vec{p}_\alpha, \vec{v}_\alpha, \vec{p}_\beta)$$

Where the first term $\vec{f}_\alpha{}^0$ denotes the driving force, which describes the desire to move in a certain direction with a certain velocity, the second term $\vec{f}_{\alpha B}$ denotes the repulsive interactions with boundaries and the third, $\sum_{\beta\neq\alpha} \vec{f}_{\alpha\beta}$ the repulsive interactions with other pedestrians.

The driving force describes each pedestrian's desire to move to their destination with their desired speed v_α^0. Letting $v_\alpha^0(t)$ be the desired speed, $\vec{e}_\alpha(t) \coloneqq \frac{\vec{d}_\alpha - \vec{p}_\alpha}{|\vec{d}_\alpha - \vec{p}_\alpha|}$ the desired direction and $\vec{v}_\alpha^0 \coloneqq v_\alpha^0 \vec{e}_\alpha$ the desired velocity, we then obtain the following expression

$$\vec{f}_\alpha{}^0(t) = \frac{1}{\vec{\tau}_\alpha}(\vec{v}_\alpha^0(t) - \vec{v}_\alpha(t))$$

with $\vec{\tau}_\alpha$ the relaxation time within which deviations from the desired velocity $\vec{v}_\alpha^0$ are corrected. To simulate impatience in case of delay, we define the desired speed as follows:

$$v_\alpha^0(t) = (1 - n_\alpha)v_\alpha^0(0) + n_\alpha(t)v_\alpha^{\max}$$

Where we have introduced the maximum desired velocity $v_\alpha^{\max}$ and the impatience $n_\alpha(t) = 1 - \bar{v}_\alpha(t)/v_\alpha^0(t)$, with $\bar{v}_\alpha$ the average speed in the desired direction of motion.

The pedestrian interactions are described by

$$\begin{aligned}\vec{f}_{\alpha\beta}(t) \quad &= A_\alpha^1 \exp\left[\frac{\vec{r}_{\alpha\beta} - \vec{d}_{\alpha\beta}}{B_\alpha^1}\right] n_{\alpha\beta} \cdot F_{\alpha\beta} \\ &+ A_\alpha^2 \exp\left[\frac{\vec{r}_{\alpha\beta} - \vec{d}_{\alpha\beta}}{B_\alpha^2}\right] n_{\alpha\beta}\end{aligned}$$

Where the second term is a collision term which stops pedestrians from crossing each other, whilst the first term describes the tendency of pedestrians to respect other pedestrians' distances. A^i_α and B^i_α, $i \in \{1, 2\}$ are respectively the interaction strength and range, $\vec{d}_{\alpha\beta} \coloneqq |\vec{p}_\alpha - \vec{p}_\beta|$, $\vec{r}_{\alpha\beta} \coloneqq r_\alpha + r_\beta$, and $n_{\alpha\beta} \coloneqq \frac{\vec{p}_\alpha(t) - \vec{p}_\beta(t)}{\vec{d}_{\alpha\beta}(t)}$. The factor $F_{\alpha\beta}$ simulates the anisotropic behaviour of pedestrians, where they react more to things in their field of vision than outside of it. it is defined as:

$$F_{\alpha\beta} = \lambda_\alpha + (1 - \lambda_\alpha)\frac{1 + \cos(\phi_{\alpha\beta})}{2}$$

with $\cos(\phi_{\alpha\beta}) = -n_{\alpha\beta} \cdot \vec{v}_\alpha(t)/|\vec{v}_\alpha(t)|$ and λ_α is the anisotropic character of the pedestrians.

The force due to a boundary is defined by:

$$\vec{f}_{\alpha\beta}(\vec{p}_\alpha) = A_{\alpha B} \exp\left[\frac{r_\alpha - \vec{d}_{\alpha B}}{B_{\alpha B}}\right] n_{\alpha B}$$

With $\vec{d}_{\alpha B}$ the distance between the boundary and the pedestrian, $n_{\alpha B}$ the unit vector pointing from the boundary to α. In the case of multiple boundaries, there are two possible ways of determining the total boundary interaction: superposition of all the interactions or keeping the one with the shortest distance only.

B. Implementation

The Social force simulation is implemented as a single class `SocialForce`. This includes force calculations, the state of the pedestrians and the solver.

The single class structure was made to simplify interaction with the solver, `scipy.integrate.ODE45` (Runge-Kutta-Fehlberg method), which is an explicit $RK4$ algorithm with an $O(h^5)$ error estimator, allowing for an adaptive timestep. This is the same algorithm as that used by GWP.

At each step of the solver, the desired velocities are calculated, the change in position is set to be the velocity and the change in velocity is set to be the total calculated force.

The possible effect of a boundary is calculated as an exponential which depends on the smallest distance between that boundary and the pedestrian. One can choose whether to have the total force due to the boundaries be a superposition of all the forces or just the force from the nearest boundary.

For the pedestrian destinations, the destination is often not a point but rather some positional threshold, such as the end of a corridor. To simulate this we place a destination at a very large distance, which is effectively infinite, and set the range of the destination so that it coincides closely with the needed threshold.

It also often happens that a pedestrian has waypoints to its destination. For instance, a pedestrian that needs to pass a corner will first aim for the corner, and then for their final destination. This is implemented by assigning an array of destinations to each pedestrian, each with a range, so that when the pedestrian is within that range of their current destination, the current destination is updated to the next one.

1. Parameters

We used the same parameters as GWP. Namely

- radius = 0.3
- initial desired speed = 3.14
- maximum desired speed = desired speed * 1.3
- relaxation time = 0.5
- anisotropic character = 0.75
- $A_1 = 0$
- $B_1 = 0.3$
- $A_2 = 2$
- $B_2 = 0.2$
- $A_B = 5$
- $B_B = 0.1$

Where one can reintroduce units by matching dimensions to meters and seconds.

2. Numerical optimisation

All the force calculations are almost exclusively done using `numpy` array operations. We have also reimplemented the most computationally heavy part of the loop, the repulsive force calculation which is of complexity $O(n^2)$, using `numba` and a Verlet sphere to reduce computational cost. Since the `numpy` code is still faster[1] until around 250 pedestrians, we have implemented a switcher that only uses the `numba` codes for higher numbers of pedestrians. For 500 pedestrians, this results in a better than 2× speed improvement.

3. Postprocessing

Postprocessing helper functions were defined in the `social_force/postprocessing` directory of the project to aid in the creation of plots and videos of the simulations.

[1] On a framework laptop 13 with adm ryzen 5 7640 cpu

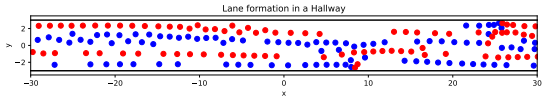

FIG. 1. Lane formation in bidirectional flow. Pedestrians line up behind each other to minimise time delays. 250 pedestrians were placed half on the left (blue) and half on the right (red) sides of the corridors and their destinations placed far on the opposite side. See `social_force/situation_tests/hallways.py` to reproduce.

C. Testing

1. *Verification*

The implementation was continuously verified during development by the inclusion of `assert` statements checking whether the dimensions of the arrays match those expected. Individual methods were tested against hand derived results, with the methods often originally written in many subparts, and the code was run line by line on simple cases and the results compared with the theoretically expected values.

2. *Validation*

The implementation was first validated with some simple tests:

1. Does a pedestrian go towards their destination.
2. Do pedestrians avoid each other.
3. Does a boundary stop a pedestrian from passing.
4. Do pedestrians successfully go from one destination to the next

Once the basic validation is completed, more complex examples are tested. First, pedestrians in a hallway successfully show lane formation as illustrated in fig. 1. This is expected behaviour, since it is easier to follow someone who has made a path into a crowd than to make one's own path. It also reflects GWP's results.

Second, pedestrian flows which intersect at an angle show striping, as as shown in fig. 2.

Thirdly, in opposing flows at a bottleneck, such as a doorway, we observe that each side goes through in alternating batches. This is best seen using the video produced with `social_force/situation_tests/doorway.py`.

Fourthly, at a 90 degree corner, the pedestrians slow down before reaccelerating after having passed the corner. This can also be observed as a video using the `social_force/situation_tests/corner.py` file.

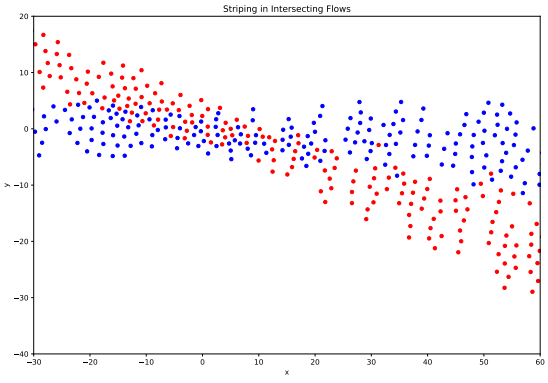

FIG. 2. Striping in intersecting flows. We observe that each pedestrian flow gets separated into "stripes" to facilitate crossing. The blue travel horizontally from left to right and the red diagonally from top left to bottom right. See `social_force/situation_tests/intersections.py` to reproduce.

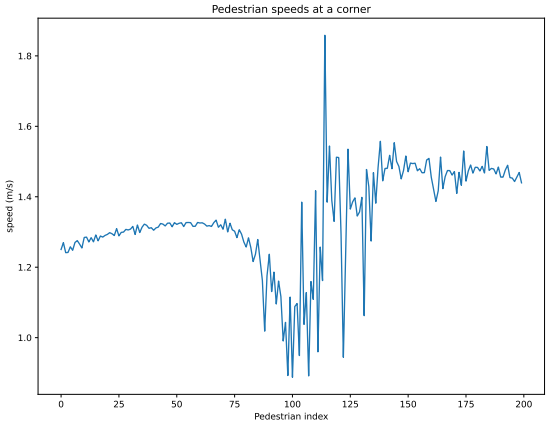

FIG. 3. Shockwaves in 90° corners. The pedestrian index is by initial position. 0 is the furthest from the corner at $t = 0$ and 200 the closest. Pedestrians are slowed down at the corner, and reaccelerate after to make up for lost time. This velocity snapshot is taken at $t = 15$ so that around half the pedestrians have passed the corner. See `social_force/situation_tests/corner.py` to reproduce.

III. MONTE CARLO MODEL

Another approach to modeling crowd dynamics, proposed by F. Piazza, does not consider pedestrians as deterministic entities but describes their behavior through a simple Monte Carlo model. This approach proves particularly useful in queuing scenarios where the mechanisms driving people's motion are either unpredictable or too complex to be effectively computed numerically. In implementing this model, we focus on a problem of serving dynamics, where all agents move toward the same goal (a central counter) until being served one by one in order of their distance to the goal.

A. Theory

The model treats crowd dynamics as two distinct processes operating on different time scales. The serving process represents a deterministic sequence where the agent closest to the goal is served and subsequently removed from the system. The rearrangement process constitutes a stochastic relaxation mechanism considered instantaneous compared to the serving timescale, allowing the remaining agents to redistribute themselves in response to the removal. For computational simplicity and analytical tractability, we adopt a circular geometry where the counter is placed at the origin. Agents are modeled as impenetrable circular disks with either homogeneous or heterogeneous radii distribution.

The Monte Carlo rearrangement follows specific algorithmic procedures. At each rearrangement step, we perform a predetermined number of displacement cycles M_I. During each displacement cycle (j), agents are considered in random order and attempt moves according to a defined rule. Each agent first moves a specified distance δ_j directly toward the counter. Then, with a certain probability p, an additional displacement with the same magnitude δ_j is added in a random direction making an angle $\alpha \in [-\pi/2, \pi/2]$ with the radial direction. This non-radial component represents agents' attempts to exploit local density fluctuations for faster progress toward their goal, mimicking realistic navigation strategies in crowded environments. A proposed move is accepted only if the new position lies within the circular boundary and no overlap occurs with any other agent. The displacement magnitudes are dynamically adjusted after each cycle (j) to maintain a target acceptance rate, typically around fifty percent.

The model requires careful initialization to ensure physical consistency and statistical reliability. The initialization procedure follows three sequential stages. Agents are first arranged on a regular square lattice within a circle of predetermined radius, with lattice spacing calculated to approximate optimal distribution for the given number of agents. Only lattice points falling within the circular boundary are retained, with exactly the required number of points selected randomly if the lattice yields excess positions. Agent radii are then rescaled to achieve a precise initial area fraction, ensuring the system's initial packing fraction exactly matches the prescribed value regardless of the specific lattice spacing or radius distribution. Finally, an initial relaxation of thousands of Monte Carlo steps is performed with purely random moves, transforming the ordered lattice into a physically realistic fluid-like configuration. Multiple independent configurations generated through different relaxation sequences serve as statistically independent initial conditions for the serving dynamics.

The complete serving dynamics algorithm proceeds systematically. After initialization, the system undergoes iterative serving cycles. In each cycle, the agent with minimal distance to the origin is identified, its serving time recorded, and it is removed from the system. Following each removal, Monte Carlo rearrangement continues until the system reaches a dynamically stable configuration after fixed number of rearrangement cycles.

The model's parameters have clear physical interpretations. The probability of non-radial moves p controls the balance between goal-directed motion and exploratory behavior. The area fraction φ determines crowd density and packing constraints. The size heterogeneity parameter Δr quantifies the diversity in agent sizes, with zero corresponding to monodisperse systems.

B. Implementation

1. *Structure*

The theoretical model was implemented in Python through the CrowdMCImproved class. Agent radii are generated based on the heterogeneity parameter Δr, then rescaled to achieve the target area fraction φ. Initial positions are created using square lattice placement within a circular boundary of radius R, followed by Monte Carlo relaxation with M_I random moves to reach a fluid-like configuration. In our case M_I is fixed to be equal to 5000

During the serving phase, the algorithm repeatedly identifies and removes the agent closest to the center, then rearranges the remaining agents using $M = 200$ Monte Carlo displacement cycles. Movement attempts combine a radial step toward the origin with a probabilistic non-radial step controlled by parameter p. Collision detection enforces hard-disk exclusion, and an adaptive step size maintains an optimal acceptance rate near 50%.

Visualization features include gradient coloring of agents by size using matplotlib colormaps, enabling intuitive analysis of size-dependent behavior. The code supports batch execution across parameter sets and saves graphical outputs automatically.

2. *Computational efficiency*

The computational efficiency of the implemented Monte Carlo algorithm reveals notable limitations, primarily stemming from its $O(n^2)$ scaling due to pairwise collision detection. In each movement attempt, the algorithm must check potential overlaps between the moving agent and all other active agents, resulting in significant computational overhead as system size increases.

The algorithm's serial nature represents a fundamental limitation. Despite the inherent parallelism of Monte Carlo methods, where individual agent moves are largely independent, the current implementation processes agents sequentially. This fails to exploit modern multi-core architectures, leaving significant potential performance gains untapped. The requirement for random ordering of move attempts further complicates parallelization strategies.

These inefficiencies collectively limit the practical scale of simulations. While the algorithm successfully models systems with several hundred agents, extending to realistic crowd sizes of thousands or tens of thousands would require substantial optimization or alternative approaches. Future implementations could benefit from spatial partitioning techniques like cell lists or neighbor lists to reduce collision detection complexity, alongside parallel processing and more efficient data structures to address the identified bottlenecks.

C. Testing

While testing the algorithm we focused mainly on two different cases: a homogeneous case ($N = 300$, $\varphi = 0.6$, $p = 0.2$, $\Delta r = 0$) and a heterogeneous case ($N = 150$, $\varphi = 0.6$, $p = 0.2$, $\Delta r = 0.2$). We produced animations describing the serving process in these cases step by step (See `MC_crowd_sim/states` and `MC_crowd_sim/crowd_simulation_results /states_N_300_hom` for animations or fig. 4 for snapshots). Hypotheses we aimed to verify were whether serving time relates to initial distance following the quadratic queuing law, and whether in heterogeneous cases smaller agents reach the goal faster.

After visually verifying visually correct behaviour of the pedestrians in the simulation we were able to produce some statistics to verify certain serving laws and discuss the impact of model's parameters.

The most prominent feature of all simulations' results is it following on average the sequential serving law, which states that number of serving steps should depend on the initial distance from the counter as $n = N(\frac{d}{R})^2$, n - number of serving steps, N - number of agents, d - initial distance from the counter of the served agent, R - outer boundary radius. We can observe the validity of this law by looking at the simulation results with $N = 300$, $\varphi = 0.6$, $p = 0.2$, $\Delta r = 0$ (see fig. 5)

Introducing heterogeneity to the radii distribution of agents gives us the opportunity to observe another effect. Agents with smaller radii turn out to be slightly faster on average in reaching the counter than bigger ones. In our case of $N = 150$, $\varphi = 0.6$, $p = 0.2$, $\Delta r = 0.2$ and 10 inde-

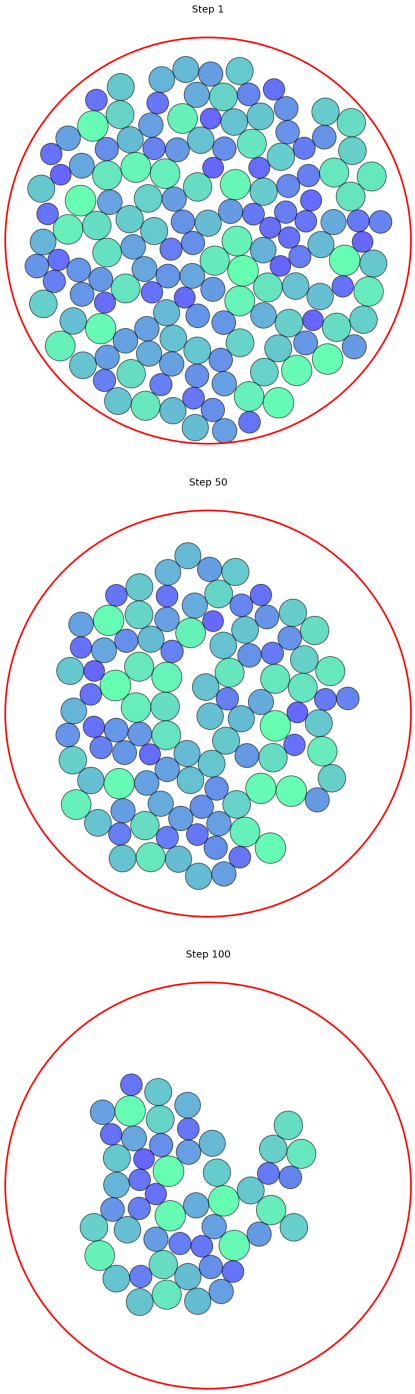

FIG. 4. Snapshots of pedestrian configurations during 1 (top panel), 50 (central panel), 100 (lower panel) step in case with $N = 150$, $\varphi = 0.6$, $p = 0.2$, $\Delta r = 0.2$. Colormap is used to highlight different radii of the agents. See `MC_crowd_sim/script.py` to reproduce.

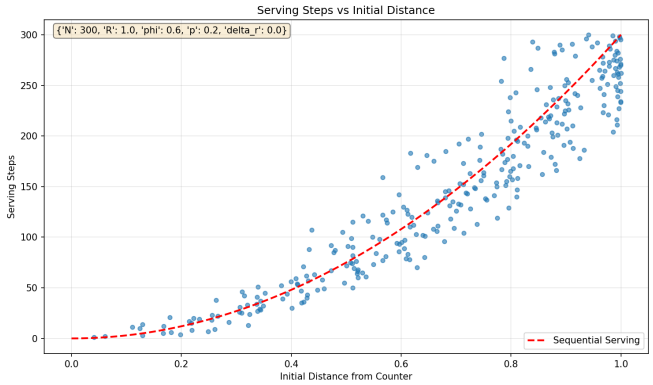

FIG. 5. Normalized serving time with respect to initial distance in the case of $N = 300$, $\varphi = 0.6$, $p = 0.2$, $\Delta r = 0$. Sequential serving law is added for reference. See `MC_crowd_sim/script.py` to reproduce.

pendent simulation runs the average time steps needed to reach the goal for agents with radii smaller than average was around 5% less than for agents with radii bigger than average. See fig. 6 for graphical distribution of serving times with respect to initial distance for inhomogeneous case.

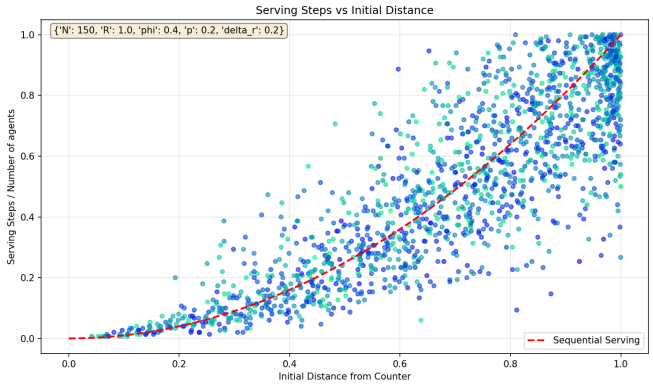

FIG. 6. Normalized serving time with respect to initial distance in the case of $N = 150$, $\varphi = 0.6$, $p = 0.2$, $\Delta r = 0.2$. Sequential serving law is added for reference. The same colormap to indicate agents' radii as in fig. 4 is used. See `MC_crowd_sim/script.py` to reproduce.

IV. COMPARISON OF THE MODELS

We now wish to compare the two models. The simplest case is to run with the social force model a similar simulation to the Monte Carlo model. We do first do this with 300 pedestrians, each with the same radius, covering 60% of the circle area.

In the social force model, this is done by taking random starting positions for the pedestrians in a circle of radius 45, with first destination the origin. When a pedestrian is within range of the first destination, we change their position and destination to a point on a circle of radius 1e6, so that they are effectively removed from the simulation, and they are outside the Verlet sphere of any other pedestrians.

The results of these two simulations can be seen in `MC_crowd_sim/crowd_simulation_results/state_N_300_hom/states_anim.gif` (obtained by `MC_crowd_sim/script.py`) for the Monte Carlo simulation and `social_force/videos/montecarlo.mp4` (obtained with `social_force/analysis/central_exit_sf.py`) for the social force simulation. We observe similar behaviour, as one would expect.

A very big difference between the two models is the social force simulation runs in around 5 seconds, as opposed to the Monte Carlo, which takes orders of magnitude longer.

V. CONCLUSION

We conclude that, considering the various simulations that were presented here, the social force model is both more versatile and more efficient than the Monte Carlo model. We do note the result that the overall behaviour of pedestrians is obtained both with deterministic pedestrian and with a Monte Carlo pedestrian model, which raises the question of whether one could simplify the social force model for an even more efficient simulation, while maintaining valid macroscopic results..

A. Organisation

We organised development by assigning the social force model to Gianluca Oberreit and the Monte Carlo model to Vadim Silaev. This made the separation of work very easy and minimised possible conflicts and misunderstandings. This also allowed for, in the case of problems being encountered, consultation with the other as a more unbiased observer.

B. Difficulties

In the development of the social force model, the main problems came from conflicting information being given in the social force presentation part of GWP's thesis and their matlab implementation. Another big problem in the beginning was pedestrians clipping through walls due to a missing pair of parentheses in the `boundary_force` calculation…Other than this one, not many bugs were encountered due to the deliberate slow nature of development, using `raise` assertions to check all the `numpy` broadcasting, explicitly writing the broadcasting using `None` indices to not rely on implicit broadcasts, and general step by step testing.

While developing Monte Carlo model, the most significant difficulties came from inherent sequential nature of the proposed algorithm, which made it inefficient for crowd sizes relevant for testing and greatly prolongated the duration of the overall process

Appendix A: Instructions on running the Code

1. Social Force Model

All the complex validation test cases (bidirectional flow in a hallway, striping in flows at an angle, bidirectional flow through a doorway, unidirectional flow around a 90° corner) can be run by running the appropriate python files in the `social_force/situation_tests` directory of the repository (respectively `hallways.py`, `intersections.py`, `doorway.py`, `corner.py`). One can select whether to display an image or view a video of the simulation by commenting and uncommenting the appropriate lines at the bottom of the file (`postprocessing.plotting.plot`, `postprocessing.plotting.movie`). For the plot of velocities in the 90° corner, one must comment the 2 `postprocessing` lines and remove the triple quotes around the final plotting code of the file.

In general, one can run a custom simulation by following the steps of the other files. For n pedestrians, that is

1. Initialising the class with n.
2. Creating an $(n, 2)$ `numpy` array of initial positions
3. Optionally creating an $(n, 2)$ `numpy` array of initial velocities
4. Creating an $(n, m, 2)$ `numpy` array of initial destinations. That is, an array of m destinations for each pedestrians.
5. Optionally creating an $(n, m, 1)$ array of destination ranges, which indicate when a pedestrian moves on to its next destination.
6. Running the class' `init_pedestrians` method with the previously defined parameters.
7. Optionally, create a $(b, 2, 2)$ array of boundaries, characterised by their start and end points, with b the number of boundaries. Then run the `set_boundaries` method with this parameter and, optionally, the `boundary_selection` parameter set to "superpose" to use superposition of boundaries instead of nearest boundary selection for the boundary force calculation. If "superpose" is selected, one can also set `boundary_verlet_sphere` to some `float` value, so that only boundaries within the Verlet sphere are accounted for.
8. Run the `init_solver` method with parameters `t_bound` the maximal time the solver will attain (this can be set to something higher than the greatest time we want to run the simulation to). Optionally, one can also set `rtol`, `atol` and `max_step`, respectively the maximal relative tolerance, absolute tolerance and timestep that the RK45 solver will use.
9. Run the `run` method, with output `(time, results)`, with optional parameters `t_bound` the maximum time to which the solver will run, `print_freq`, every how many timesteps the number of timesteps passed is printed (this can be set to `False` or `None` for no printing) and `to_save`, a tuple of values to save. These can be accessed in the `results` dictionary, the second value of the output tuple. The following strings are taken into account if in `to_save`:
   - `"positions"`
   - `"velocities"`
   - `"desired_speeds"`
   - `"desired_directions"`
   - `"total_forces"`
   - `"repulsive_forces"`
   - `"driving_forces"`
   - `"boundary_forces"`

2. Monte Carlo model

All the simulation results can be produced by running `MC_crowd_sim/script.py`. There are two options built in to run either: one or many independent simulations with different parameters, or running a set of simulations with different parameters but different starting configurations to acquire statistics from multiple runs. Choosing the running mode is done by uncommenting relevant lines in "`if __name__ == "__main__":`" condition.

For running an independent simulation you must call the function "`run_complete_simulation_once()`" while specifying desired parameters, seed, and what results do you want to be shown or saved: initialisation, serving statistics or/and snapshots. After running the simulation results will be saved in `crowd_simulation_results/`.

For running a series of simulations you must call the function "`run_complete_simulations()`" and specifying number of consecutive runs as well as desired parameters and seed.