Large Scale Crowd Simulation

1. Overview

The last two decades have seen a dramatic rise in the number of techniques used in simulating real-world phenomena, from animating fluid and sound to modelling hyperelastic materials and human crowds [Bridson 2015; Lee 2010; Pelechano et al. 2016; Sifakis and Barbic 2012]. In many fields, this growth in the choice of simulation methods raises many questions for domain experts in choosing appropriate simulation techniques. This is especially true in human crowd simulation, where there are many simulation methods that all perform well in some scenarios, but none of which perform “best" in all scenarios. However, choosing the right simulation method can be very important. When crowds are used in games and movies, poor simulation performance can lead to unnatural crowd behaviour, quickly breaking the immersion. When crowd simulations are used to guide event planning and building design, poor simulation models can lead to bad practices affecting thousands of people.

As in many fields, domain experts in crowd simulations have a wealth of informal knowledge about what simulation approaches to use and when. Many experts will suggest not using reactive methods such as social forces or boids [Helbing et al. 2000; Reynolds 1987] unless the simulation task involves a small number of agents in sparse scenarios. Likewise, many researchers tend to feel geometric methods such as ORCA [van den Berg et al. 2011] are likely to struggle in medium-density scenarios where agile manoeuvres are important, but do

well in dense scenarios where efficient accounting for constraints is key. This kind of informal, unquantified knowledge is crucial to the proper choice of simulation method, but cannot always be reliably communicated in an objective and universally accessible fashion

Ultimately, our work provides a method for formalising and quantifying knowledge about what types of simulation techniques to use in different scenarios. To that end, we propose the following contributions:

- An agent-centric formulation of the entropy metric of [Guy et al. 2012] that allows the estimation of a simulation's local accuracy
- An analysis of key trade-offs between simulation methods, along with a learning-based approach to automatically predict the best (i.e., most accurate) simulation methods to use in a given scenario
- ***Crowd Space:*** Crowd Space refers to a conceptual framework where complex interactions within crowds are simplified and represented in a lower-dimensional space. This space is centred around individual agents (people or entities within the crowd) and captures a wide range of scenarios where these agents interact. By using a low-dimensional manifold, which is a simplified representation akin to a multidimensional map, Crowd Space enables researchers to understand and model diverse crowd behaviours such as movement patterns, social interactions, and emergent group dynamics. This approach allows for effective analysis and simulation of crowd dynamics while focusing on essential aspects of agent behaviour and interaction within the crowd

2. Literature Review

Our work is mostly based on the paper [Karamouzas, I., Sohre, N., Hu, R., & Guy, S. J. (2018). Crowd Space: A Predictive Crowd Analysis Technique. ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2018)] and [Guy, S. J., van den Berg, J., Liu, W., Lau, R., Lin, M. C., & Manocha, D. (2012). A Statistical Similarity Measure for Aggregate Crowd Dynamics.]

To facilitate our predictive analysis of various crowd simulation techniques, we must first establish an evaluation metric. Here, we wish to focus on the accuracy of a simulation technique; that is, how closely a simulation matches the behaviour of real pedestrians in similar scenarios. Much of the previous work in this area has focused on studying indirect measures of accuracy, such as comparing the local density of simulated agents to those found in a database of typical human motion as in [Lerner et al. 2010] and [Charalambous et al. 2014], or has used user-defined metrics to evaluate a path such as smoothness or number of collisions [Kapadia et al. 2009; Singh et al. 2009]. While these approaches are valuable metrics of path quality, they do not directly address the predictive accuracy a simulation method may have in a given scenario. Closer to our goal is the entropy metric proposed in [Guy et al. 2012], which uses a stochastic inference framework to directly estimate the prediction accuracy of a method given a set of pedestrian trajectories in a manner which is robust to the large amount of sensor noise present in real-world datasets.

The original framework assumes that the number of agents in the whole scenario is static throughout the dataset. In practice, agents will likely enter and leave the scene quite

frequently in any dataset lasting more than a few seconds. Also, the original entropy metric provides a single accuracy estimate per simulation method per dataset. In reality, the accuracy of a simulation varies spatially (per-agent), and over time (for a given agent). So to deal with it we propose a per-agent entropy metric which (1) allows agents to enter and exit a scene over time and (2) provides a per-agent, ego-centric estimate of how well a given crowd simulation method would predict the motion of each agent in the crowd.

3. Methodology

In this section, we first provide a brief overview of the data used and the preprocessing required to prepare the data for our algorithm. Following this, we present the details of the algorithms and simulation methods employed in calculating the entropy metric. Finally, we describe the simulation code used to calculate the entropy.

For the entropy metric calculation, we require real-world datasets from which we can extract the coordinates and velocities in each frame. We utilised a video depicting two agents crossing paths and exchanging positions.

Fig1. Two agents exchanging their positions

Extraction of 2D-Coordinates Of Agents

For the detection of the coordinates, We used the **YOLOv8** model with a frame rate of **125ms** or **8hz.** The YOLO architecture adopts the local feature analysis approach instead of examining the image as a whole, the objective of this strategy is mainly to reduce computational effort and enable real-time detection. To extract feature maps, convolutions are used several times in the algorithm.

Convolution is a mathematical operation that combines two functions to create a third. In computer vision and signal processing, convolution is often used to apply filters to images or signals, highlighting specific patterns. In convolutional neural networks (CNNs), convolution is used to extract features from inputs such as images. Convolutions are structured by Kernels (K), Strides (S) and paddings (p).

In terms of probability, the convolution operation can be understood as a weighted average or a weighted sum of random events, which can be interpreted as a probability distribution with two random variables X and Y with probability distributions pX(x) and pY(y). The convolution of X and Y is given by

$$(X*Y)(z) = \int pX(x)\, pY(z - x)\, dx$$

YOLO predicts bounding boxes and class probabilities for objects directly from the input image in a single forward pass, making it significantly faster.

The YOLOv8 architecture can be broadly divided into three main components:

Backbone: This is the convolutional neural network (CNN) responsible for extracting features from the input image. YOLOv8 uses a custom CSPDarknet53 backbone,

which employs cross-stage partial connections to improve information flow between layers and boost accuracy.

Neck: The neck, also known as the feature extractor, merges feature maps from different stages of the backbone to capture information at various scales. YOLOv8 utilises a novel C2f module instead of the traditional Feature Pyramid Network (FPN). This module combines high-level semantic features with low-level spatial information, leading to improved detection accuracy, especially for small objects.

Head: The head is responsible for making predictions. YOLOv8 employs multiple detection modules that predict bounding boxes, objectness scores, and class probabilities for each grid cell in the feature map. These predictions are then aggregated to obtain the final detections. The head of YOLOv8 consists of multiple convolutional layers followed by a series of fully connected layers. These layers are responsible for predicting bounding boxes, objectness scores, and class probabilities for the objects detected in an image

We discarded those frames in which one or no agents were detected to maintain consistency and using this algorithm we were able to capture the coordinates at **27** different timestamps.

All the coordinates and timestamps were recorded and stored in a NumPy array as a **.npy** file in the data folder in the GitHub repository.

The trajectory of both agents :

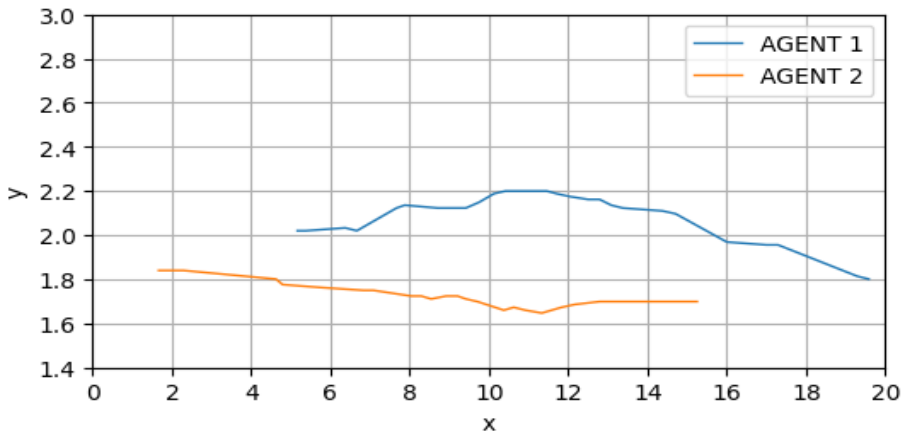

Fig2. The trajectory of both agents

Upon analysis of the x vs t and y vs t of any one agent, we found that there were not many noises present in our dataset and hence there was no need to apply any denoising filters.

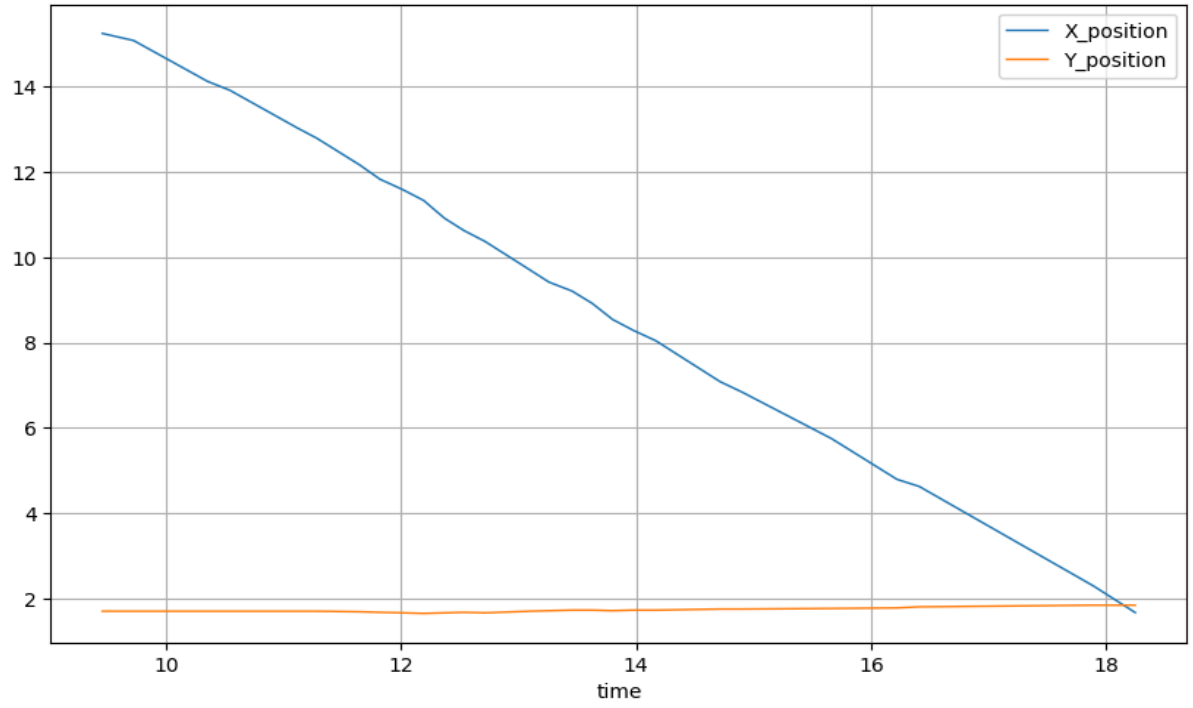

Fig3. Time variation of x and y coordinate of agent 1

Extraction of Velocities of Agents

We need the respective velocities of agents in the horizontal and vertical direction as it is needed for our algorithm later. For this, we use the **Central Difference Formula** with the 5-point stencil to calculate the first derivative of a function.

In one dimension, if the spacing between points in the grid is *h*, then the five-point stencil of a point *x* in the grid is :

$$\{x - 2h, x - h, x, x + h, x + 2h\}$$

The first derivative of a function *f* of a real variable at a point *x* can be approximated using a five-point stencil as:

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}$$

There for our case, we can represent the x and y position of the agent as a function of f and g dependent on time i.e,

$$f(t) = x\,,\; g(t) = y$$

so, h will become the time difference between frames.

Since for this algorithm, we need to give at least two initial values as from the equations above we can comprehend that (t > 2h), so we take the initial two values of velocity to be zero for both agents.

Algorithm 1: Central Difference Formula for Agent 1

Input : The x-coordinate of agent1 *x_1,* The y-coordinate of agent1 *y_1,* The number of timestamps *t*

Output: The numpy array file of x and y velocities of agent 1 *vx_1.npy* and *vy_1.npy.*

for each i ∈ 2,3…..,t-2 **do**

$$h = time[i+2] - time[i]$$

$$v\_x[i] = \frac{-x\_1[i+2]+8*x\_1[i+1]-8*x\_1[i-1]+x\_1[i-2]}{12*h}$$

$$v\_y[i] = \frac{-y\_1[i+2]+8*y\_1[i+1]-8*y\_1[i-1]+y\_1[i-2]}{12*h}$$

end for

do *np.save("vx_1.npy",v_x1)*

np.save("vy_1.npy",v_y1)

Similarly, we can calculate for the second agent.

Conversion of Units

We have the coordinates in pixels and need to convert them to S.I. units. Knowing the real-world coordinates and their corresponding pixel coordinates for four points, we can perform this conversion. The origin is assumed to be at the lower left corner of the window.

Real World Coordinates(in metres)	**Displayed Coordinates**(in pixels)
(0,0)	(0,0)
(20,0)	(480,0)
(20,3.50)	(480,272)
(0,3.50)	(0,272)

Therefore for a general displayed point (x,y) in pixels, the corresponding points in metres can be written as

$$(x, y) = \left(\left(\frac{x}{480} * 20\right), \left(\frac{y}{272} * 3.50\right)\right)$$

Due to the small magnitude of the velocity, we used metres per millisecond (m/ms) as its unit.

Formulation of Agent Centric Entropy metric

The approach we use is to first decompose real crowd data into small simulation tasks. we use an **Ensemble Kalman Smoother**(EnKS) as a state estimation technique. In this method, we also need a crowd simulation function to compare and for this, we use **Universal Power Law** Simulations.

After the ensemble of states has been estimated via EnKS, we can estimate the noise covariance, **Q**, which would maximise the likelihood of the observed trajectory using **Maximum Likelihood Estimation.**

After estimating Q we will use this value to calculate the entropy :

$$e(Q') = \frac{1}{2} * log((2\pi e)^d * det(Q')$$

Where d is the dimensionality of Q' and Q' is the covariance matrix obtained by averaging Q over all agents.

The algorithms are explained in detail below.

Ensemble Kalman Smoother(EnKS)

EnKS-based approach works by iteratively estimating the most likely internal agent state ak at timestep k based on external observations over the entire dataset.

For our case, we need some inputs which we need to implement it.

S - State Matrix The state of each agent was defined as a Four-dimensional vector of 2D position and 2D velocity. It is defined as an agent only.	$[x, y, v_x, v_y]$
H - Position Observation Function It is used to convert an instance of the state vector to the format found in the validation data. In our case, the function h simply returns the position component of the state vector	$h([x, y, v_x, v_y]) = [x, y, 0, 0]$
f **- Prediction Function**	The result of Universal Power Law simulations in the form of a matrix
Q - Process Noise Covariance Matrix It will be used for estimation and correction in the Maximum Likelihood Estimation step.	
X_k - State of k-th agent. The state of the agent comes as a result of simulation using **Universal Power Law.** It contains information about the state of all agents along with the ensemble and the time stamps at which we have to calculate the entropy metric.	Expressed in the form of a matrix = [ensemble size, k, S_k, timestamp]

EnKS takes a two-stage approach of first predicting the new state for the next timestep, **f (X(k))**, based on the prediction function (predict step), and then correcting this prediction based on the observations (correct step). The effect of simulation uncertainty is modelled through the use of noise. Rather than running a single simulation to predict the next agent state, **m(=2000)** simulations are run each with a sample of Gaussian noise with covariance **Q** added to the prediction. An agent's state, then, is represented by an ensemble of different states. (A similar step adds noise of fixed size **R** to the observations.) At each timestep, the

correct step fits a Gaussian to this ensemble of samples and updates the state estimation based on the Kalman gain equation.

Algorithm 2: EnKS

Input : Process noise covariance matrix(Q), no. of agents(num), ensemble size(m), H, no. of iterations(iteration), measurement noise(R), Real World State(Y)
Output: The predicted state (*N)*
//Predict Step
for each j ∈ 1,2…..,num **do**
for each i ∈ 0,1,..... m-1 **do**
for each s ∈ 0,1,2,3 **do**

$$N[j,\ i+1,\ s,\ iteration] = X[j,\ i,\ s,\ iteration] + rnd.gauss(0, Q[j,\ s,\ s])$$
$$z[j,\ i+1,\ s,\ iteration] = H[j,\ i,\ s,\ iteration] + rnd.gauss(0, R[j,\ s,\ s])$$

end for
end for
// Correct Step

$$z' = (1/m) * (\sum_{i \in z} i)$$

$$z\_cov = (1/m) * (\sum_{i \in z} (i - z') * (i - z)^T$$

$$N' = (1/m) * (\sum_{i \in z} i)$$

$$cross\_cov = (1/m) * \Sigma (N - N')(z - z')^T$$

for each i ∈ 1,2,..... m **do**

$$N[j,\ i+1,\ s,\ iteration] \mathrel{+}= (cross\_cov) * (z\_cov)^{-1} * (z - z')$$

end for
end for

Maximum Likelihood Estimation:

We can estimate the noise covariance, **Q**, which would maximise the likelihood of the observed trajectory using Maximum Likelihood Estimation.

Maximum likelihood estimation is a method that determines values for the parameters of a model. The parameter values are found such that they maximise the likelihood that the process described by the model produced the data that were actually observed.

Our approach assumes that the error between the predicted state and the observed state can be well-modelled by a Gaussian noise of covariance **Q** when performing the Maximum Likelihood Estimation.

i.e.

$$P(\boldsymbol{x}: \mu,\sigma) = L(\mu,\sigma: \boldsymbol{x}) = \left(\frac{1}{\sigma\sqrt{2\Pi}}\right) * exp\left(-\left(\frac{x-\mu}{\sqrt{2\Pi}}\right)^2\right)$$

We assume the noise to be zero meaned, and **Q** is the covariance matrix of the state. Therefore, the variance for a particular state can be determined from the diagonal elements of the covariance matrix.

for any S[i] , for particular likelihood, $\boldsymbol{Q[i,i]} = \sigma^2$

To maximise the likelihood we need to differentiate and find the optimal value of σ.

Instead of maximising the likelihood the general practice is to maximise the log-likelihood to reduce the mathematical complexity. This is absolutely fine because the natural logarithm is a monotonically increasing function. This means that if the value on the x-axis increases, the value on the y-axis also increases. This is important because it ensures that the maximum value of the log of the likelihood occurs at the same point as the original likelihood function.

Thus using the data points, we can minimise the log-likelihood and get the optimal value of Q.

Algorithm 3: Maximum Likelihood Estimation

Input: Estimated State Distribution (N), X, No. Of Iterations(iter)
Output: Process Noise Covariance Matrix(Q)
for each k ∈ 1,2…. num **do**
$Q[k] = 0,\ Q'[K] = 0$
for each i ∈ 1,2,....,m **do**
$Q[k] \mathrel{+}= (\frac{1}{m}) * (N[j, i, :, iter] - X[j, i, :, iter]) * (N[j, i, :, iter] - X[j, i, :, iter]$
end for
end for
for each k ∈ 1,2…. num **do**
for each i ∈ max(0,k-w),....., min(num,k+w) **do**
$Q'[k] \mathrel{+}= Q[i]$
$Q'[k] /= 2w + 1$
end for
end for

Universal Power Law Simulation.

It simulates the dynamics of a crowd using power-law principles. It employs agent-based modelling where each agent represents an individual in the crowd. The agents move according to certain rules, considering both their desired velocities and interactions with neighbouring agents.

w.r.t to agent simulation many tasks need to be done like finding the neighbours, calculating energy, time to collision, updating the states etc

The neighbours are determined by the Euclidean distance between agents, considering periodic boundary conditions (torus topology). For two agents i,j their positions are represented as c[i] and c[j],

$$\textit{distance calculation(d)} = c[i] - c[j]$$

We also have to adjust for the torus topology i.e if one agent leaves the window it has to reappear again,

Thus, for a window size of length ***s*** metres.

$$\text{if } d[0] > s/2 \Rightarrow d[0] = s - d[0]$$

Similarly, we can do this for d[1]

The squared distance $\ell 2$ is calculated, and if $\ell 2$ is within the specified sight range, agent j is considered a neighbour of agent i.

Now to calculate the time of collision(τ) between the two agents we consider the relative distance and relative velocity.

τ can be calculated by solving the below equation and taking the minimum root.

$$at^2 + bt + c = 0$$

where : $a = r_v * r_v$

$b = 2 * (r_v * p)$

$c = p*p - (r_a + r_b)^2$

$discr = b^2 - ac$

where r_v is the relative velocity of the two agents.

p is the relative position of the two agents.

We also have to adjust the distances w.r.t to torus topology.

This formula can be derived from the laws of motion.

We need to compute the differential energy between agents, which influences their avoidance behaviour. For this, first, we need the energy expression,

$$E(t) = \frac{B}{t^m} * exp(\frac{-t}{t_0})$$

Where B,t_0 are constants.

Now, differentiating it spatially we can get the differential energy or the avoidance force as,

$$i.e\ F_{av} = -\Delta_r(E) = -(k * exp(\frac{-t}{t_0}) * (r_v - (\frac{r_v p - pa}{\sqrt{discr}}) * (\frac{m}{t} + \frac{1}{t_0}))$$

Now using this force and velocities we can calculate the acceleration of the agents to update the positions and velocities for the next frame, and we can apply a cap on the maximum acceleration using the F_{av}.

Finding the Entropy Error

At each timestamp, our approach involves a sequential application of three rounds of Ensemble Kalman Smoother (EnKS) to iteratively estimate and correct the states of agents within the crowd dynamics. Following this, we employ Maximum Likelihood Estimation (MLE) to assess the error between simulated and predicted states. To ensure robustness and stability in our analysis, we further smooth the results using windowed averaging techniques. The entropy value, computed using parameter Q, is then averaged across all epochs to provide a comprehensive measure of crowd dynamic as done in [Guy, S. J., van den Berg, J., Liu, W., Lau, R., Lin, M. C., & Manocha, D., 2012, ACM Transactions on Graphics, Article No: 190]

The formula for entropy error : $e(Q') = \frac{1}{2} * log((2\pi e)^d * det(Q')$

Where d is the dimensionality of Q' and Q' is the covariance matrix obtained by averaging Q over all agents.

4. Results

for a window size of 6 metres and collision diameter of 0.5m, we get the range of entropy values as this :

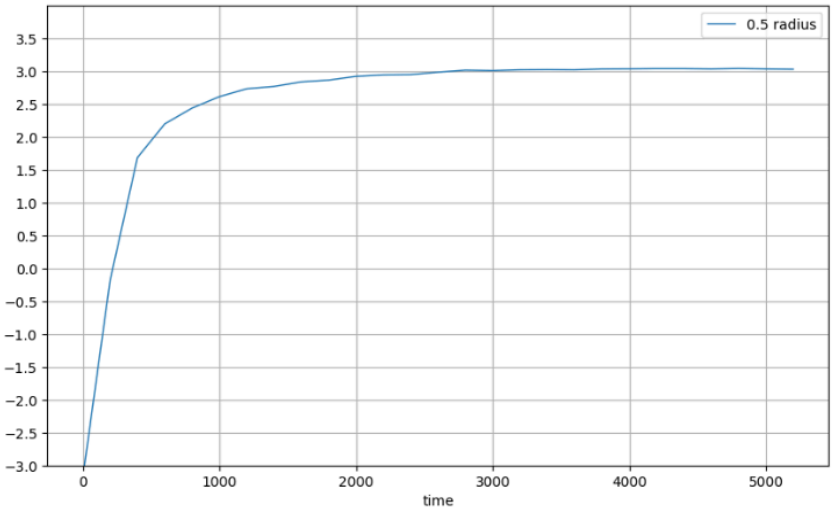

Fig 4. Average Entropy value for 0.5m agent radius

Thus, we can see that the entropy value over the time stamp is converging to a value of around **3**

Now, we will see the variation of entropy with different parameters.

Entropy vs radius.

Theoretically, if we see, for a fixed window size the entropy value should decrease with the decrease in radius as the agent has now more free space to move and hence the uncertainty in collision decreases, but also for collision avoidance we need the state and with decreasing radius, the certainty in the position decreases and hence there is a tradeoff between the increasing and decreasing uncertainty. Also, we have to keep in mind the torus topology which adjusts the state otherwise.

Collision Radius(in m)	**Entropy Value**
0.2	3.002433790511066
0.5	3.0378922750888537

1	3.054169547015123
1.4	3.0379668110160636
1.8	3.0466853063637434

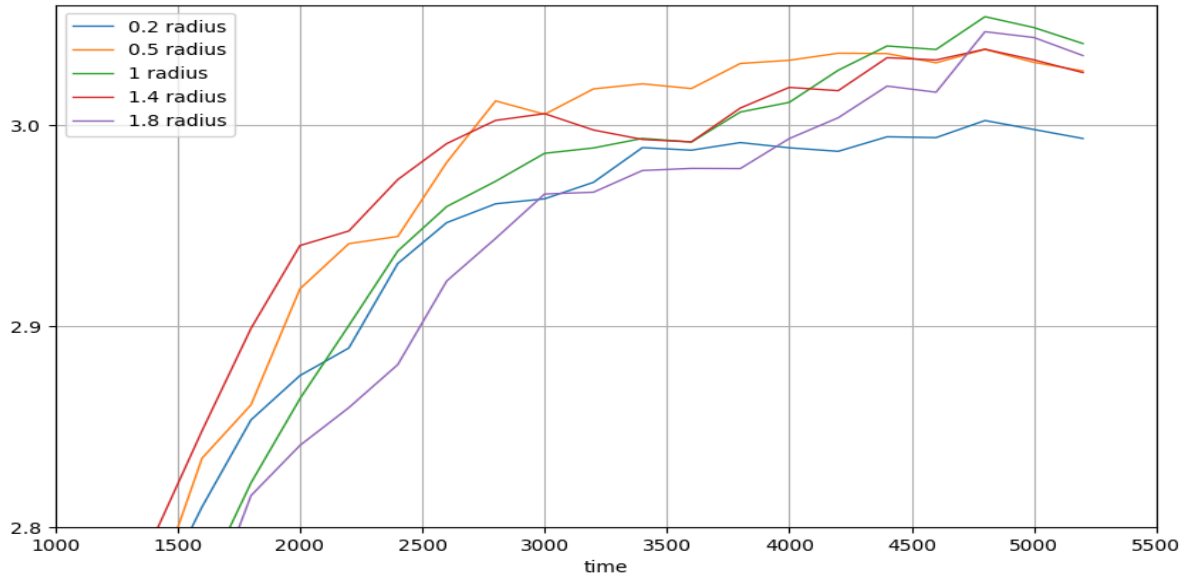

Fig 5. Variation of Average Entropy values with time for different radii

5. Further Improvements

As we can see the variation of the radius with the entropy values somewhat deviates from the actual results,

So, to further improve our results we also applied the maximum likelihood estimation algorithm for **R** (Measurement noise) which gave good entropy results

What we expect is that the entropy values for all radii at the beginning and the end should be nearly the same, as interactions are minimal at these points, accurately reflecting realistic conditions and should be maximum at the points when they are closest to each other and this is what we got after the improvements.

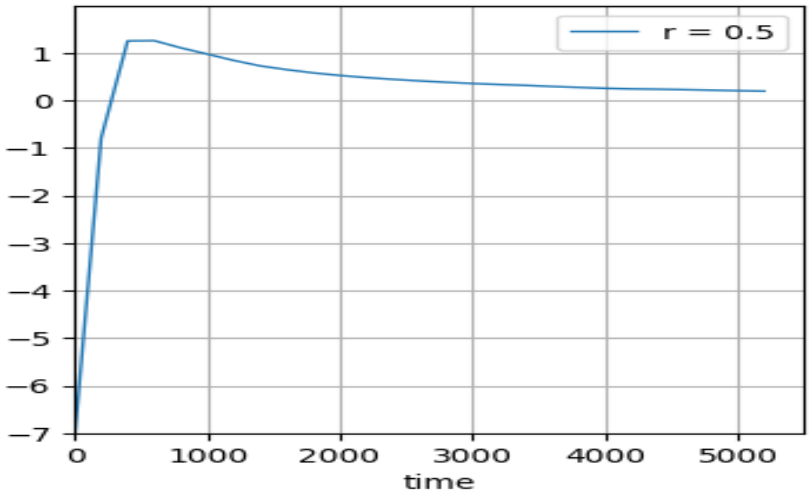

Fig 7. Average entropy value with 0.5m agent radius

This is the graph of entropy values for different radii, here we can observe the start and end of the graph for all radii are almost the same, adhering to the principles,

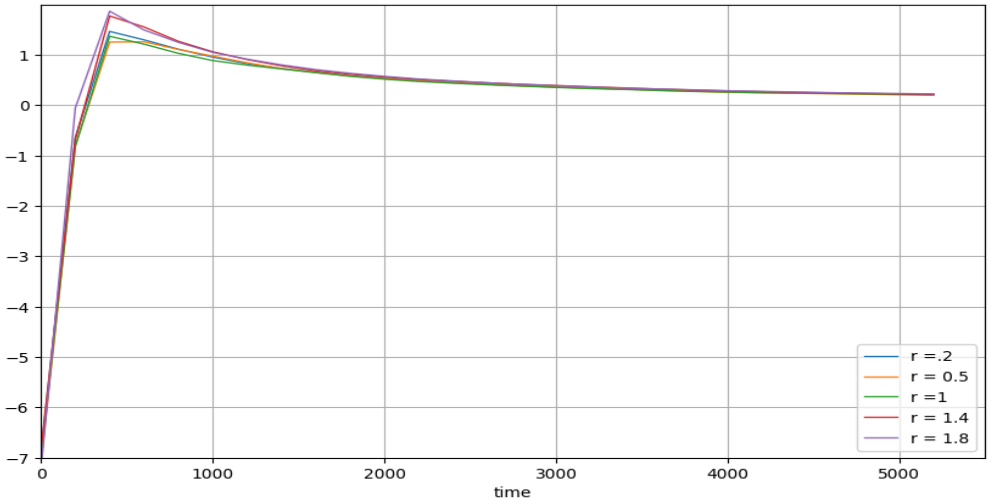

Fig 8. Variation of Average Entropy value with time for different radii

This is the variation of peaks of different radii when they are closest to each other, here we can observe that $peak \propto \frac{1}{radius}$ and it agrees with the theoretical fact also, as the chance of

collision avoidance increases when the size of agents increases due to the constant size of the window.

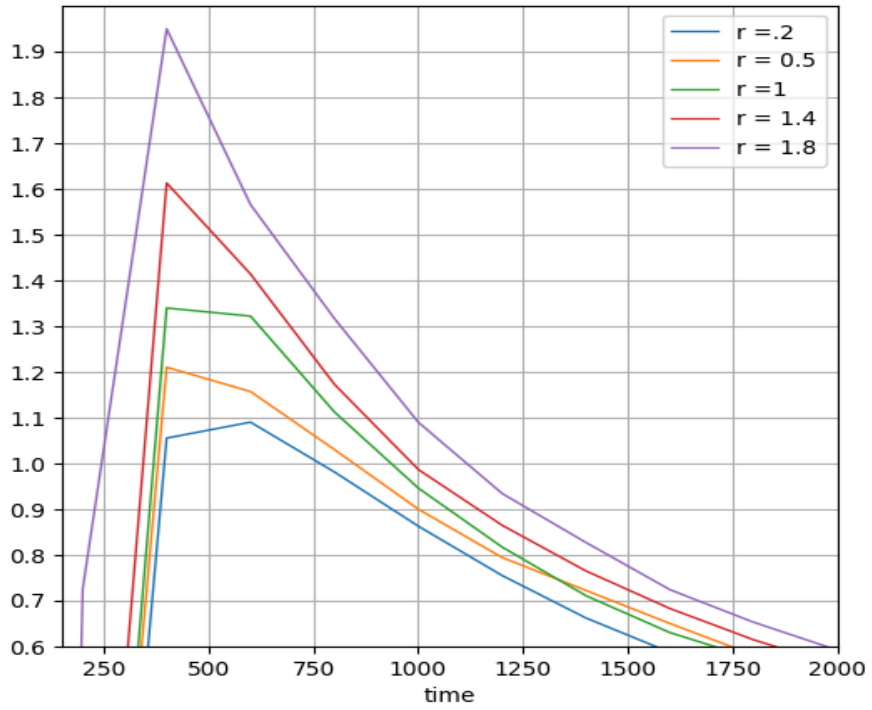

Fig 9. Variation of Entropy with radii at the point of minimum separation

Collision Radius(in m)	Entropy Value
0.2	1.08941426
0.5	1.2095394
1	1.3388229
1.4	1.61207529
1.8	1.94886984

Thus, with these adjustments, we get fairly good improvements.

Comparison of Our Entropy Metric with the previous work

In prior research, entropy was calculated as a single average value across the entire dataset, rather than being evaluated at individual timestamps. This approach provided an average entropy value of around **2.01** for agents with a 1-metre radius, particularly at the timestamp when they were in closest proximity to each other. In contrast, our advanced algorithm computes entropy at each specific timestamp, offering a more granular and accurate assessment of crowd dynamics. This method allows for a more detailed and insightful analysis of agent interactions over time, demonstrating superior performance in capturing the intricacies of crowd behaviour

Simulation Results with UPL

Fig 10. Agents starting the journey

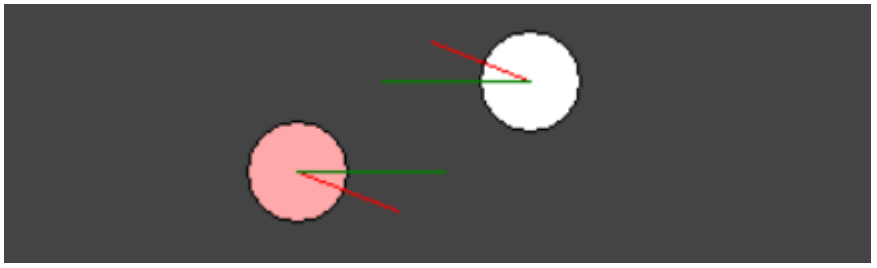

Fig 11. Agents closest to each other

Fig 12. Agents ending their journey

6. References:

Ioannis Karamouzas, Nick Sohre, Ran Hu, and Stephen J. Guy. 2018. Crowd Space: A Predictive Crowd Analysis Technique. ACM Trans. Graph. 37, 6, Article 186 (November 2018), 14 pages. https://doi.org/10.1145/3272127.3275079

Guy, S. J., van den Berg, J., Liu, W., Lau, R., Lin, M. C., & Manocha, D. (2012). A Statistical Similarity Measure for Aggregate Crowd Dynamics. ACM Transactions on Graphics. Article No: 190, Pages 1 - 11

Robert Bridson. 2015. Fluid simulation for computer graphics. CRC Press.
Panayiotis Charalambous and Yiorgos Chrysanthou. 2014. The PAG Crowd: A Graph Based Approach for Efficient Data-Driven Crowd Simulation. Computer Graphics Forum 33, 8 (2014), 95–108

Jur van den Berg,Ming C.Lin, and Dinesh Manocha.2008. Reciprocal Velocity Obstacles for real-time multi-agent navigation. In IEEE International Conference on Robotics and Automation. 1928–1935.
Jur van den Berg, Stephen J. Guy, Ming C. Lin, and Dinesh Manocha. 2011. Reciprocal n-body Collision Avoidance. In the International Symposium of Robotics Research. 3–19.

Shawn Singh, Mubbasir Kapadia, Petros Faloutsos, and Glenn Reinman. 2009. Steer Bench: a benchmark suite for evaluating steering behaviours. Computer Animation and Virtual Worlds 20, 5-6 (2009), 533–548