Department of Electronic Engineering

BEng Project Report

2020/2021

Student Name: Noah Williams

Project Title: "You're it!": Mathematical and computational modelling of children's playground chasing games.

Supervisors: John Bissell & Mohammad Nasr Esfahani

Department of Electronic Engineering
University of York
Heslington
York
YO10 5DD

Contents

ABSTRACT – In this report we discuss the potential of modelling two children's playground games: chain tig and stuck-in-the-mud, in a controlled environment using both deterministic and agent-based models. We look at the history of these models and how they have evolved with time and their present-day applications. We derive mathematical solutions for both of them, and design simulations that can be used to predict and measure the transition rates for experiments with variable parameters. We compare the results from each method of modelling, and show that once past a certain complexity level, stochastic techniques such as agent-based modelling becomes a much preferred way of modelling dynamic systems with human behavioural influences.

1| Introduction

It is a safe assumption that playground games that involve chasing, often with additional rules, are enjoyed by children all around the world, with the most common of which being named any variation of 'tig', 'tag', 'it', 'tick', 'touch', and so on. The premise is that some number of players, usually one or two depending on the overall group size, aim to chase the remaining players with the intent to touch them and transfer the status of 'it', thereby becoming a chaser while the other player joins the ranks of the runners. An article by Steve Roud [1] estimates that these types of games originate from early seventeenth century Britain and have been increasingly popular since, even branching out into other variants, many of which are described by Opie 1969 [2]. Throughout this report I will be exploring methods of modelling two specific versions most commonly known as chain tig and stuck-in-the-mud using both deterministic or compartmental models and agent-based models. The reason for doing so is to study these games in a controlled environment, where we can experiment with theoretical parameters to how the resultant behaviour is affected, and the differences between each method of modelling. This research could be further useful when applied to outside dynamic systems such as epidemiology. A paper by W. Casey [3] investigates signalling games to understand deceptive mimicry with relations to epidemiology, and while this is not an example of a children's game, it does show us an example of a simulations of human behaviour and the applications it can have in outside practices such as those I will be exploring myself.

2| Background and literature review

When conducting research and experiments in a certain topic, it is important to explore the existing literature and its origins within that field. This is to gain an understanding of its history and therefore areas that have already been investigated which can in turn be used to build knowledge and relevant arguments to be discussed.

2.1 Compartmental models

In epidemiology, the most widely used way to simplify the modelling of infections over time is to divide the population into a small number of compartments [4], which each have properties assigned by the model. In the early 20^{th} century various public health physicians led the development of these models, one of which was Hamer [5] who hypothesized that the probability of infection over a discrete period of time was proportional to the number of infected individuals multiplied by the number susceptible in a population. To account for this, he suggested a 'mass action law' to account for this rate of new infections since which has become a basic part of compartmental modelling of any variation. These ideas were further expanded upon by Kermack and McKendrick [6], who formed a deterministic epidemic model involving three compartments for people who are susceptible S, infected I, and recovered R, as well as being one of the first epidemic

models to incorporate randomness. Their basic model is sufficient enough to describe a simple epidemic, though does not factor in enough variables to be used for explaining endemic disease transmission or repeated events. In spite of this, the S.I.R. model is the most basic and well known, being the standard that many more complex models are derived from. Following the suggestion from Hamer, a transition rate must be introduced between compartments in order to account for varying parameters and how this will affect the resulting model.

For some infections such as the cold, the recovered compartment becomes irrelevant due to them not providing an individual with immunity upon recovery. Due to this, new S.I.S, or susceptible-infected-susceptible models were developed as shown by Murray [7].

The importance of these models and their applications are discussed by Bissell [8][9], in which he goes into depth about modelling the spread of socially determined behaviours, including those related to health issues. In particular he applies it to the rate at which smokers abandon their habit and the rate of relapse, which gives us a good grasp on the range of applications that these models have.

2.2 Agent based models

An agent-based model (ABM) is classified as a stochastic computational system for simulating a number of agents and their interactions with no human input, and how their behaviour affects the result of the entire system. Since these models are random, a single simulation will more than likely not show an accurate depiction of whatever system is being modelled, which is where we can use Monte Carlo sampling. Monte Carlo sampling [10] is the process of relying on repetitions of random sampling to gain a more reliable depiction of whatever is being sampled, meaning the more repeats the greater the dependability. One of the earliest examples of an agent-based model was one produced by T. C. Schelling [11] to model segregation, and the dynamics of discriminatory choices made by the agents.

Since its early days, agent-based modelling techniques have seen increased usage, including modern day applications discussed by Bonabeau in [12], such as simulating flow patterns for evacuations. Bonabeau also provides insights into some issues with ABM, one of which being that the agents used are generally modelling human behaviour but are unable to account for irrational choices made by humans in a real-life situation. Another use of ABMs discussed by Bissell [13] is to visualise the decline of British bank population, in which he finds that from a mean path of 1000 samples, resulting data matches the expected value of being a discrete analogue to the exponential model. Finally, a more relevant example is a study done by H. Situngkir [14], in which agent-based models are used to simulate the spreading rate of influenza in certain areas of Indonesia. The data gathered from this model is enough to describe possible action to help prevent such events from occurring, and potentially preventing a pandemic.

3| Deterministic models of chasing games

A 2012 technical report by J. J. Bissell [15] discusses mathematical modelling of children's playground games, in which he develops deterministic models for tig, chain-tig and stuck-in-the-mud. The reason for doing so is to view chasing games in a completely controlled context and to illustrate how minor changes to initial conditions can lead to differences in the outcomes.

3.1 Chain tig

Chain tig consists of a population *N*, with a designated number of both *Z* 'chasers' and *X* 'runners', where $Z + X = N$. The premise of this game is that as soon as a chaser makes contact with a runner's body, the runner is then added to the ranks of the chasers. In this way, the chaser population increases at a rate equal to or above zero ($dZ/dt \geq 0$) until eventually there exist no more runners, at which point the game concludes. Due to the nature of this game, we can determine that there will be a constant positive proportionality present between both the proportion of runners remaining *X/N* and the total chasers *Z*, which we will designate as the transition rate ν. We can also define variables for the proportion of each population to normalise them, as follows

$$\frac{dx}{dt} = -\nu xz \text{ and } \frac{dz}{dt} = +\nu xz \quad (3.1)$$

where

$$x = \frac{X}{N} \text{ and } z = \frac{Z}{N} \quad (3.2)$$

Due to the logical solution of $x + z = 1$, we can reduce (3.1) further to

$$\frac{dx}{dt} = -\nu x(x-1) \quad (3.3)$$

$$\int_{x(0)}^{x(t)} \frac{dx}{x(x-1)} = \int_0^t \nu(s)ds \quad (3.4)$$

where *s* is a dummy variable used to suggest time dependence in ν. Solving this integral by utilising partial fractions, we end up with the solution

$$x(t) = \left[\frac{x(0)}{x(0) + z(0)e^{\int_0^t \nu(s)ds}}\right] \quad (3.5)$$

For where the transition rate ν is constant, becomes

$$x(t) = \frac{x(0)}{x(0) + z(0)e^{\nu t}} \quad (3.6)$$

This solution interestingly shows that when at $t=0$, if there is any non-zero proportion of chasers (i.e., $z(0) \neq 0$);

$$\lim_{t\to\infty} x(t) = 0 \text{ and } \lim_{t\to\infty} z(t) = 1 \quad (3.7)$$

which implies that over an infinite duration, the proportion of chasers will always reach the maximum value of 1, meaning the game will always, eventually, conclude.

3.2 Stuck-in-the-mud

Stuck-in-the-mud is another variant of tig which, similarly to chain tig, begins with an initial population *N* with an assigned number of *Z* chasers and *X* runners but instead of transferring chaser status, once a runner is caught they become stuck in place. Due to the nature of this ruleset, the total

number of chasers never changes, but the total number of runners may vary due to the newly introduced population of stuck agents *Y*. For this model, $Y = M - X$, where *M* is the total players that are not chasers ($M = N - Z$). The *Y* population can be decreased by a runner coming into contact with a stuck individual, in which case the stuck agent will once again become a runner. Following equation (3.2), we can immediately normalise these populations to *M* so that

$$x = \frac{X}{M} \text{ and } y = \frac{Y}{M} \qquad (3.8)$$

From this and what we can expect that due to the mechanics of the game, we can assume

$$\frac{dx}{dt} = +\beta xy - \gamma x \qquad (3.9)$$

$$\frac{dy}{dt} = -\beta xy + \gamma x \qquad (3.10)$$

where β and γ define positive proportionalities of release and capture respectively, and where *Z* is incorporated into γ. Interestingly, this model is a direct equivalent to the S.I.S model for epidemiology discussed in section 2.1. Furthermore, as with chain-tig, the logical fact that $x + y = 1$ means that this problem can again be reduced to one dimension;

$$\frac{dx}{dt} = -\beta x^2 - (\gamma - \beta)x \qquad (3.11)$$

From this, we can express as a Bernoulli equation

$$\frac{dx}{dt} + \alpha(t)x + \beta(t)x^2 = 0 \qquad (3.11)$$

where $$\alpha(t) = \gamma(t) - \beta(t) \qquad (3.12)$$

can in turn be solved for exact solutions where α = 0 by integrating equation (3.11), and where α ≠ 0 by utilising the dummy variables r and s, and integrating factor $f(t) = f(0)e^{-\int_0^t \alpha(s)ds}$:

$$\mathrm{x(t)} = \begin{cases} \dfrac{\mathrm{x(0)}}{\mathrm{e}^{\int_0^t \alpha(s)ds} + \mathrm{x(0)} \int_0^t \beta(s)\mathrm{e}^{\int_s^t \alpha(r)dr}ds} & \text{for } \alpha \neq 0 \\ \dfrac{\mathrm{x(0)}}{1 + \mathrm{x(0)} \int_0^t \beta(s)ds} & \text{otherwise,} \end{cases} \qquad (3.13)$$

We can then continue to assume constant α and β values based on the model being expressed, resulting in

$$\mathrm{x(t)} = \begin{cases} \dfrac{\mathrm{x(0)}\alpha e^{-\alpha t}}{\alpha + x(0)\beta[1 - e^{-\alpha t}]} & \text{for } \alpha \neq 0 \\ \dfrac{\mathrm{x(0)}}{1 + \mathrm{x(0)}\beta \mathrm{t}} & \text{otherwise,} \end{cases} \tag{3.14}$$

Therefore, we can see that if $\beta > \gamma$, then $\alpha < 0$ due to equation (3.12), and the limits become

$$\lim_{t\to\infty} x(t) = \left(1 - \frac{\gamma}{\beta}\right) \text{ and } \lim_{t\to\infty} y(t) = \left(\frac{\gamma}{\beta}\right) \tag{3.15}$$

whereas $\beta \leq \gamma$ results in $\alpha \geq 0$, which leads to limits of

$$\lim_{t\to\infty} x(t) = 0 \text{ and } \lim_{t\to\infty} y(t) = 1 \tag{3.16}$$

implying that for there to be a conclusion where all runners become stuck, the release rate β must be surpassed by the rate at which they are caught γ.

4| Designing agent-based simulations of chasing games

While unable to account for various human factors such as general fatigue or psychological factors that would be present in real life examples of chasing games, stochastic techniques such as agent-based models offer us the opportunity to incorporate a realistic prediction of such behaviours. In this section I will be assessing how to design such models.

4.1 Agents moving in a domain

For our simulations of *n* timesteps, we desire a predetermined number of agents to all exist within a domain of size X×Y. This is achieved by assigning individual random pairs of *x* and *y* coordinates within the domain, which are then translated into a cartesian vector

$$\underline{r}(t_n) = \big(x(t_n), y(t_n)\big) \tag{4.1}$$

where t_n defines the position at the n^{th} timestep. This is illustrated in its most basic form in Fig. 1, showing a single agent static at the point $\underline{r} = (x, y)$.

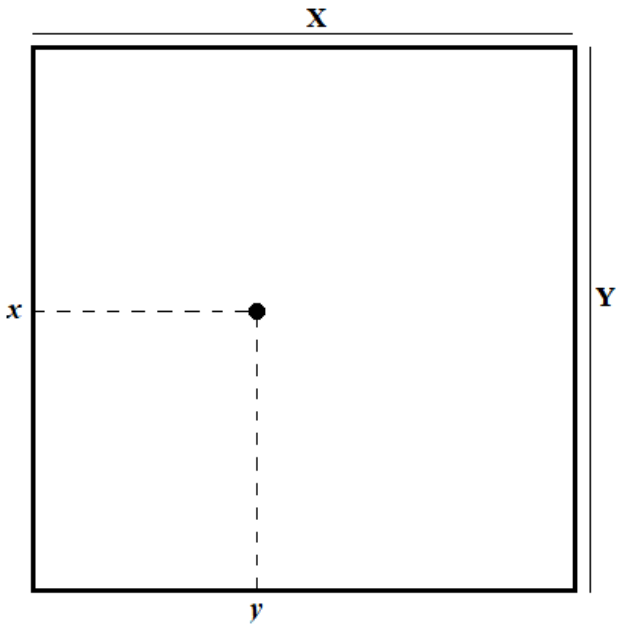

Fig. 1 A single agent on a canvas of size X×Y, with a cartesian coordinate of (x,y)

From here, we can incorporate movement into our agents by assigning a set mean speed <v> and random angle of rotation $\hat{r}$, and using them to calculate the subsequent position of each individual, as shown in equation (4.2).

$$\underline{r}(t_{n+1}) = \underline{r}(t_n) + \Delta\underline{r} \qquad (4.2)$$

where

$$\Delta\underline{r} = (< \mathrm{v} > \Delta t) \times \hat{r} \qquad (4.3)$$

4.2 Agent based model for chain tig and stuck-in-the-mud

The primary mechanic required when designing chasing games is the ability to transfer status through contact, meaning that now that we have agents that are free to move in a set domain, these agents must then be able to detect when another unit is within a certain radius. This is achieved by setting an impact parameter *b* where, if a chaser and a runner come within each other's range, the runner will be assigned the chaser status for chain-tig or become stuck during a game of stuck-in-the-mud. The same mechanic is used when a runner encounters a stuck agent in stuck-in-the-mud, where in this case the stuck player will be freed and become a runner once again. This impact parameter is displayed neatly in Fig. 2.

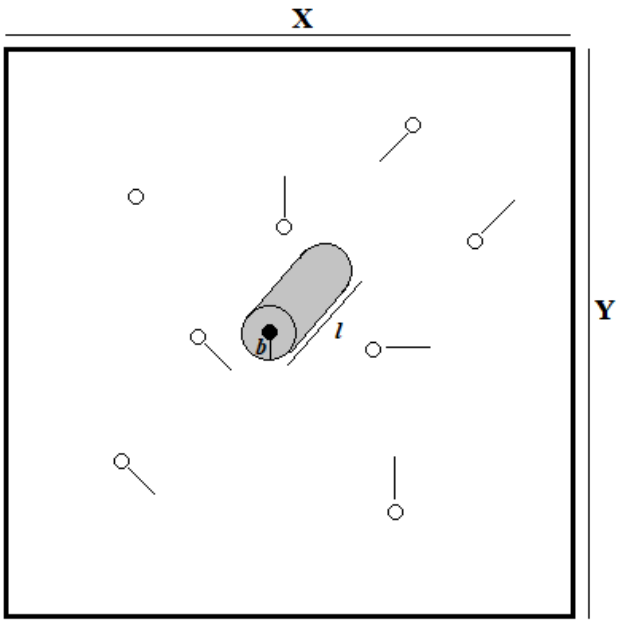

Fig. 2 Multiple agents moving in random directions, with a single chaser with an impact parameter of 'b' and distance travelled of 'l'

Also seen in Fig. 2 is distance l travelled by an agent in timeframe δt, and is calculated by

$$l = < \mathrm{v} > \ \delta t \qquad (4.4)$$

which, using the distance travelled and impact parameter, can be used to determine the area of influence a single chaser possesses over the time δt:

$$\begin{aligned} \delta A \ &\approx l \times 2b \\ &\approx 2b < \mathrm{v} > \ \delta t \end{aligned} \qquad (4.5)$$

4.3 Kinetic model for predicting transition rate nu

Now that I have working models for both chain-tig and stuck-in-the-mud, the next step is being able to predict the transition rate for simulations with set parameters. Using the value for area of influence of a single chaser determined in equation (4.5), the predicted number of runners caught by a single chaser over an amount of time can be calculated as

$$\Delta\tilde{x} = \delta A \ \times \frac{X}{A} \qquad (4.6)$$

therefore,

$$\Delta\tilde{x} = \frac{\delta A}{A} X \qquad (4.7)$$

with X being the total number of runners as previously defined in section 3.1. From this, we can easily work out the total number of runners captured in time δt by simply multiplying equation (4.7) by the total number of chasers.

$$\begin{aligned}\delta X &= -\delta\tilde{x} \times Z \\ &= -\frac{\delta A}{A} XZ \end{aligned} \qquad (4.8)$$

From here, I want to calculate the change in population of runners over time. I can do this by initially substituting my original value for δA, followed by dividing both sides by δt.

$$\delta X = -\left(\frac{2b < \mathrm{v} > \ \delta t}{A}\right) XZ \qquad (4.9)$$

$$\frac{\delta X}{\delta t} = -\left(\frac{2b < \mathrm{v} >}{A}\right) XZ \qquad (4.10)$$

While this equation appears to be complete, we would finally like to normalise the populations of both runners and chasers as we did in equation (3.2).

$$\frac{\delta\left(\frac{X}{N}\right)}{\delta t} = -\left(\frac{2b < \mathrm{v} >}{A}\right)\left(\frac{X}{N}\right) Z \qquad (4.11)$$

$$\begin{aligned}\frac{\delta x}{\delta t} &= -\left(\frac{2b < \mathrm{v} >}{A}\right) Nxz \\ &= -\left(\frac{2b < \mathrm{v} > N}{A}\right) xz \end{aligned} \qquad (4.12)$$

Here we have a normalised model for the change in runner population with regards to time. Looking at this closely, we can see a striking similarity to the deterministic model defined in equation (3.1), except in place of the constant ν we have an expression taking four constants as inputs. This value will be our kinetic model prediction for the impact parameter ν.

$$\frac{\delta x}{\delta t} = -\left(\frac{2b < \mathrm{v} > N}{A}\right) xz \qquad \text{and} \qquad \frac{dx}{dt} = -\nu xz \qquad (4.14)$$

$$\nu_{kinetic\ model} = \frac{2b < \mathrm{v} > N}{A} \qquad (4.15)$$

5| Simulations and results

After applying the concepts discussed in section 4, I coded models for both chain-tig and stuck-in-the-mud with random movement using Java and recorded the population of chasers, runners, and stuck agents at each timestep. This data was then plotted and further used to measure a practical value for the transition rate.

5.1 Monte Carlo sampling for chain tig

To show a reliable mean curve for my agent-based models, I utilised the Monte Carlo method of sampling discussed in section 2.2, as well as seen used in Bissell 2017 [13]. This involves taking a large number of simulations and plotting the mean field with standard deviation. I decided to use a total population size of 20 randomly placed agents each of two pixels in diameter

and therefore an impact parameter of three pixels. One of said agents is randomly assigned as chaser, leaving the other 19 as runners, each having a relative speed set to a fixed value of 10 pixels per timestep. To observe how population density affects the transition rate, I decided to vary the size of the window in which the simulation is being carried out. I used three window sizes of 100×100, 200×200 and 400×400, meaning the relative size of the agents is respectively 2%, 1%, and 0.5%. Beginning with a window size of 200×200, Fig. 3 demonstrates an average curve of 1000 individual runs, closely resembling that of what was expected from the deterministic model. I have included five random runs for both chaser and runner populations to demonstrate the variance between each of them. A high number of samples shows the rate of change for average population of chasers is at its highest between 40 and 130 seconds, with it slowing down from there, until coming to a maximum value at roughly 260 seconds. The inverse can also clearly be seen for the runner population.

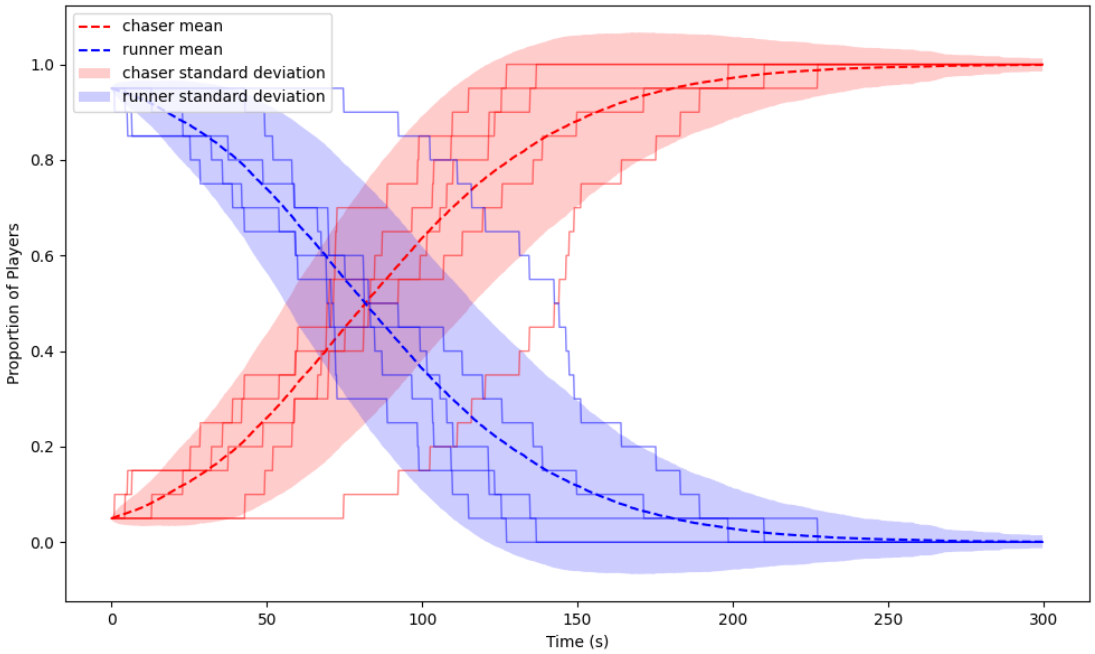

Fig. 3 Average of 1000 individual runs of chain tig with window size of 200×200, with the standard deviations for runners and chaser shown, as well as 5 random individual runs for each.

The next simulation had a window size of 100×100, a total area a quarter that of the first. As shown in Fig. 4, the chaser population predictably reaches a maximum value in a much shorter time of around 160 seconds, but the mean changes in population appear to approximately remain a similar shape to those in Fig. 3. The rate of change is much greater in this case due to a higher number of collisions largely due to the smaller area.

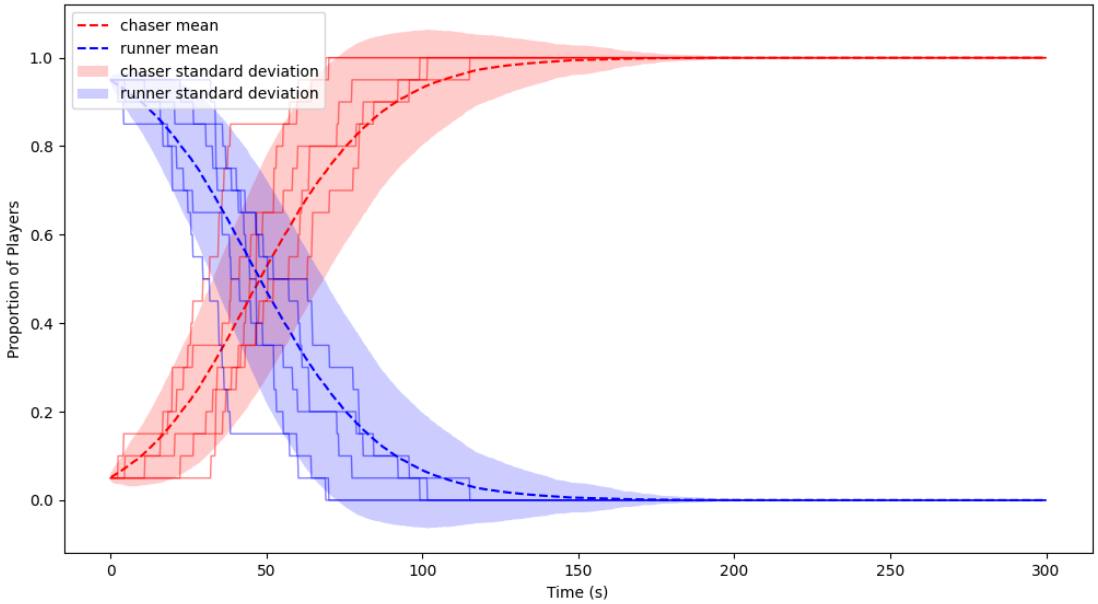

Fig. 4 Average of 1000 individual runs of chain tig with window size of 100×100, with the standard deviations for runners and chaser shown, as well as 5 random individual runs for each.

The final simulation I ran for chain tig was with an area of size 400×400, meaning a total area 16 times larger than the original simulation samples. The results shown in Fig. 5 indicate that a much greater runtime is needed for the populations to reach their maximum and minimum values. As discussed in section 3.1, a constant transition rate implies that as time tends towards infinity, the limits to the total proportion of chasers and runners are one and zero respectively. What this means is that no matter the parameters, all games will eventually conclude. Based upon each previous simulation and the fact that the maximum rate of change is yet to be reached, I would estimate that the runtime would need to be at least four times the length for games with an area of this size to eventually reach their final populations.

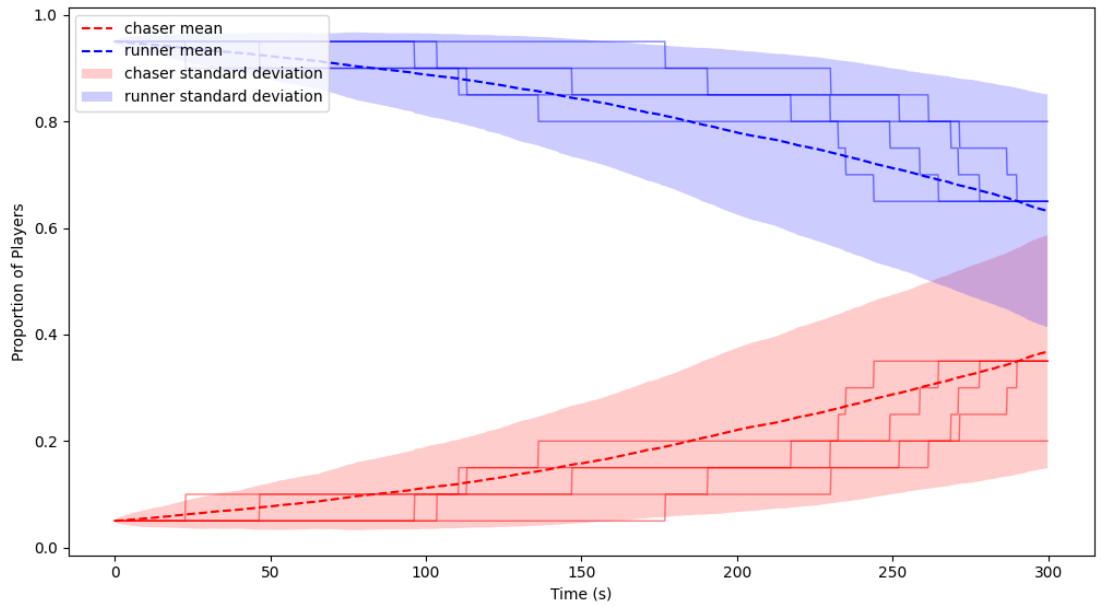

Fig. 5 Average of 1000 individual runs of chain tig with window size of 400×400, with the standard deviations for runners and chaser shown, as well as 5 random individual runs for each.

5.2 How to extract nu from data

After plotting the population fraction with respect to time, the next step was to calculate the gradient of the initial simulation. Fig. 6 shows the rate of change to time graph produced from this run, and as previously stated, the highest values for chaser rate of change lie roughly between 50 to 100 seconds, with the absolute maximum being about the crossover point, where chaser and runner populations are equal.

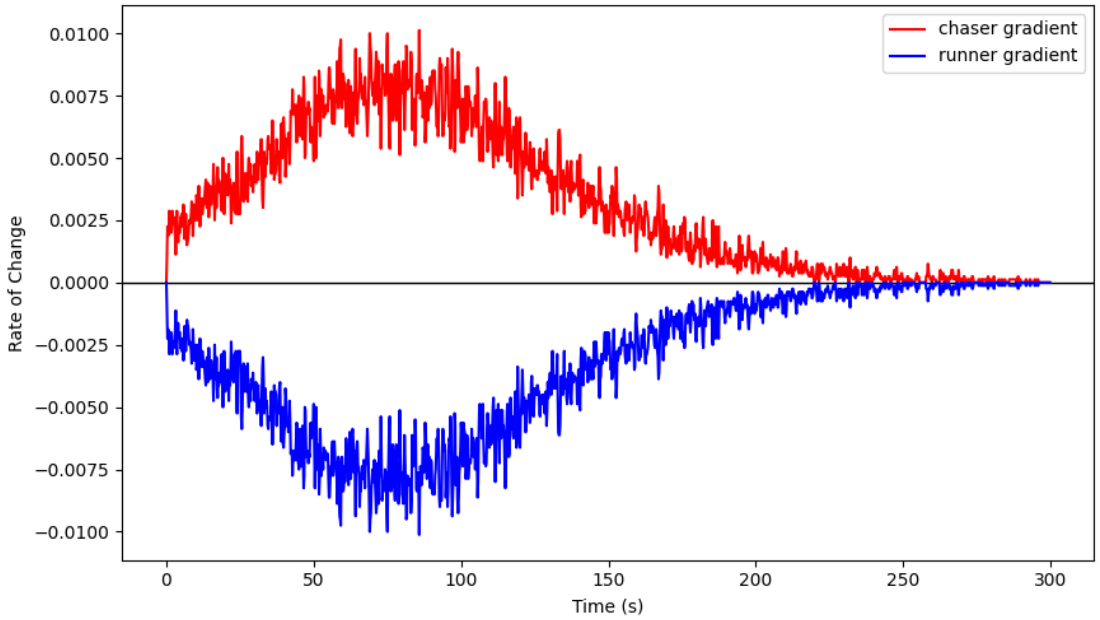

Fig. 6 Gradients of mean of 1000 individual runs with window size of 200×200 (Fig. 1).

Each of the three previous simulations followed the same process of calculating the gradient, which in turn was used to derive a measured value of the transition rate nu. This was achieved by following the deterministic model discussed in section 3.1, specifically equation (3.3), which when rearranged, results in

$$\nu_{measured} = -\left(\frac{1}{x(1-x)} \cdot \frac{\Delta x}{\Delta \mathrm{t}}\right) \tag{5.1}$$

where x is the normalised runner population.

Because my results are based on simulations of real data, values of nu are naturally quite noisy, particularly towards the beginning and end of each run. Because of this, I opted to disregard values outside of a certain interval where the mean population of both chasers and runners was between 0.2 and 0.8. Doing this allowed my results to focus on the area with the most significant rate of change alone. After this I was able to calculate a mean measured value for nu, as well as the standard deviation about that mean. Fig. 7 shows these measured transition rates for both runner and chaser populations, as well as comparing the mean measured value to the calculated value from the deterministic model discussed in section 3.1.

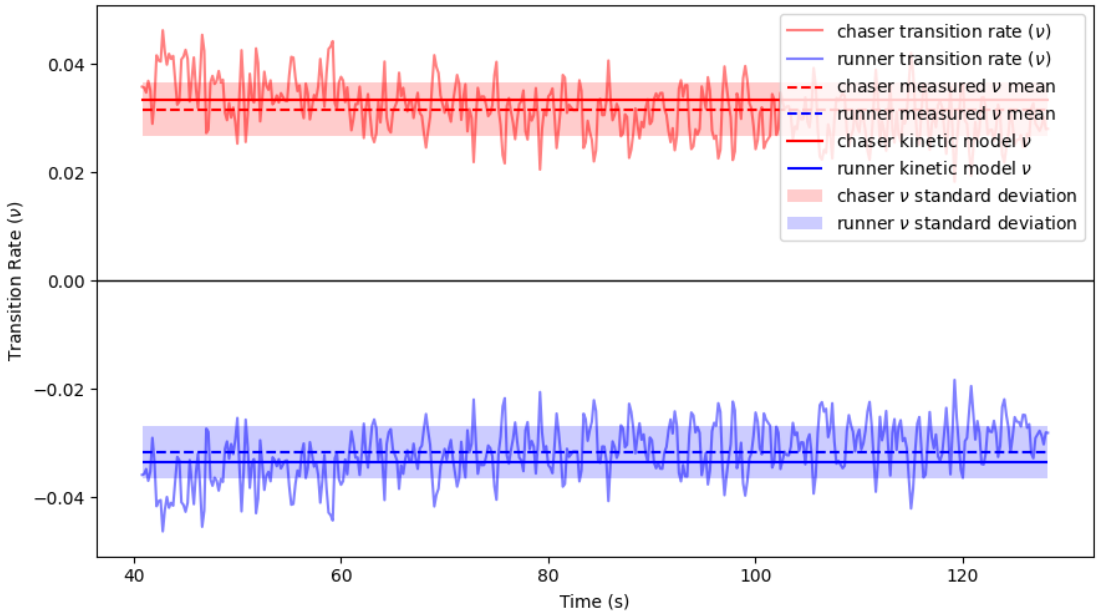

Fig. 7 Transition rates for domain size of 200×200, as well as the calculated deterministic model value with set parameters

Fig. 8 is the result of following the same process as the simulation with a domain size of 100×100, and has resulted in both the measured mean and the calculated model chaser values for nu being slightly higher than the previous example, and slightly more apart from each other.

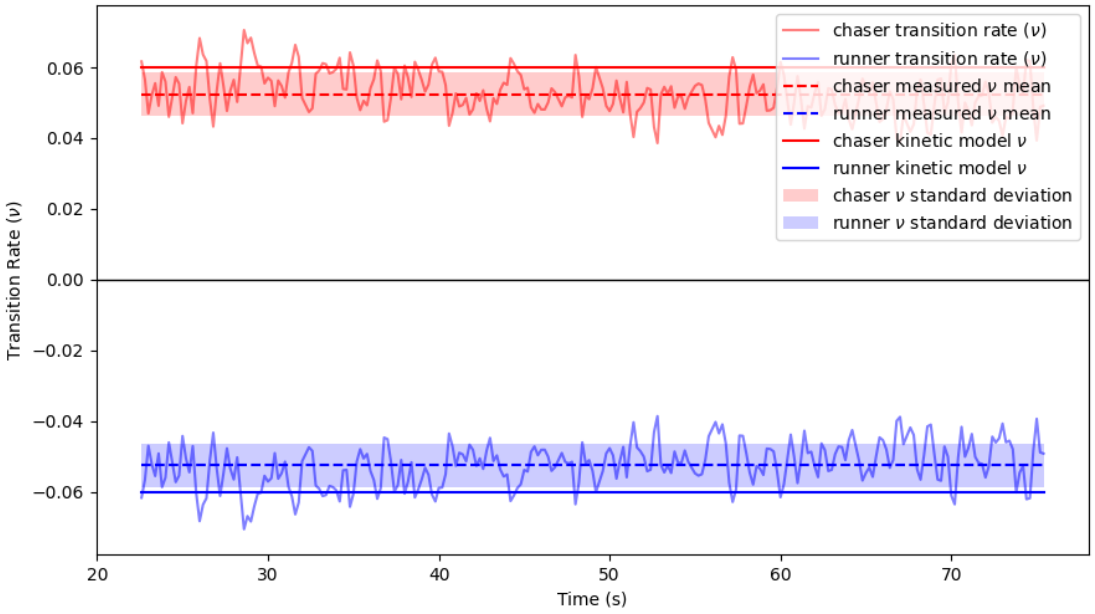

Fig. 8 Transition rates for domain size of 100×100, as well as the calculated deterministic model value with set parameters

Again, Fig. 9 shows the results of calculations on results for simulations with an area of 400×400. As opposed to Fig. 8, it appears that increasing domain size results in transition rates closer to zero, with measured and calculated nu being much more accurate to one another.

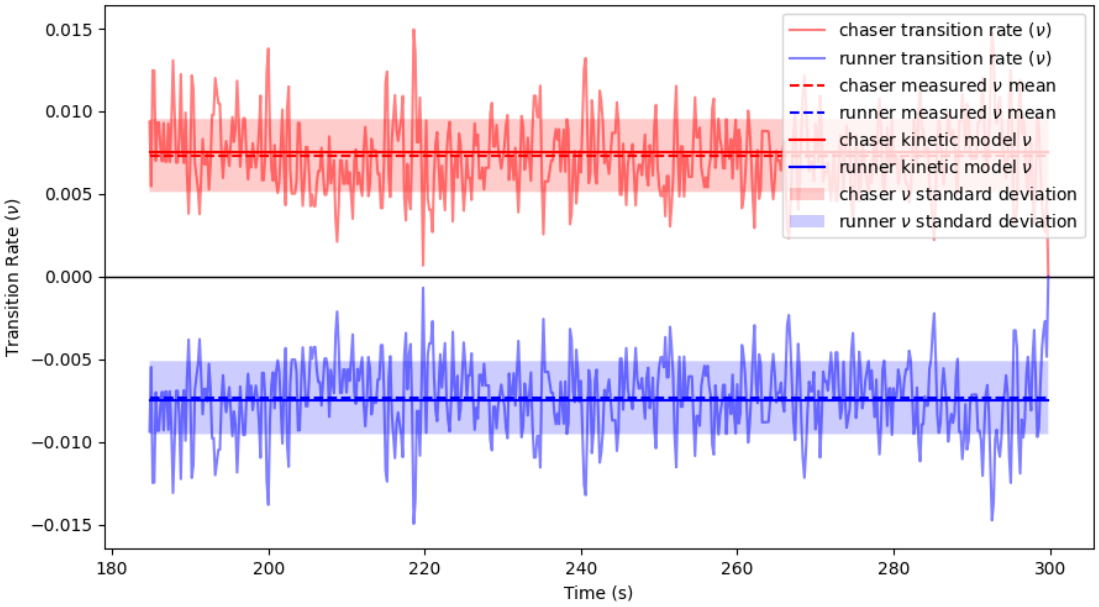

Fig. 9 Transition rates for domain size of 400×400, as well as the calculated deterministic model value with set parameters

5.3 Preliminary simulations of stuck-in-the-mud

When simulating stuck-in-the-mud, I used the same parameters the produced the results seen in Fig. 3 and multiplied the total game time by four to see if any changes occurred in each simulation given enough time. I began with a single chaser and 19 runners which, as shown in Fig. 10, came to an equilibrium point of around 90% of the population remaining runners.

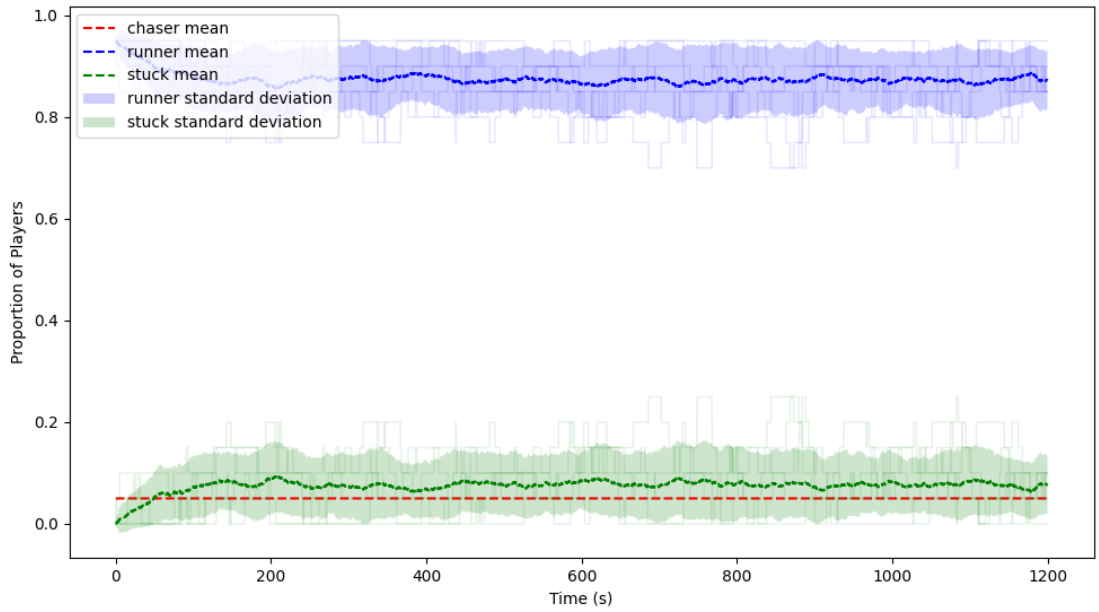

Fig. 10 Average of 100 individual runs of stuck-in-the-mud with 1 chasers and 19 runners initially, with the standard deviations for runners and stuck populations shown, as well as 5 random individual runs for each.

In the second simulation I ran, I increased the number of chasers to 5, and decreased the runners to 15 in order to make sure I kept the total population the same between experiments. The results of this simulation, Fig. 11, show a much more interesting pair of mean values in which the stuck population is more dominant by the time an equilibrium is reached, with a crossover point at about 200 seconds.

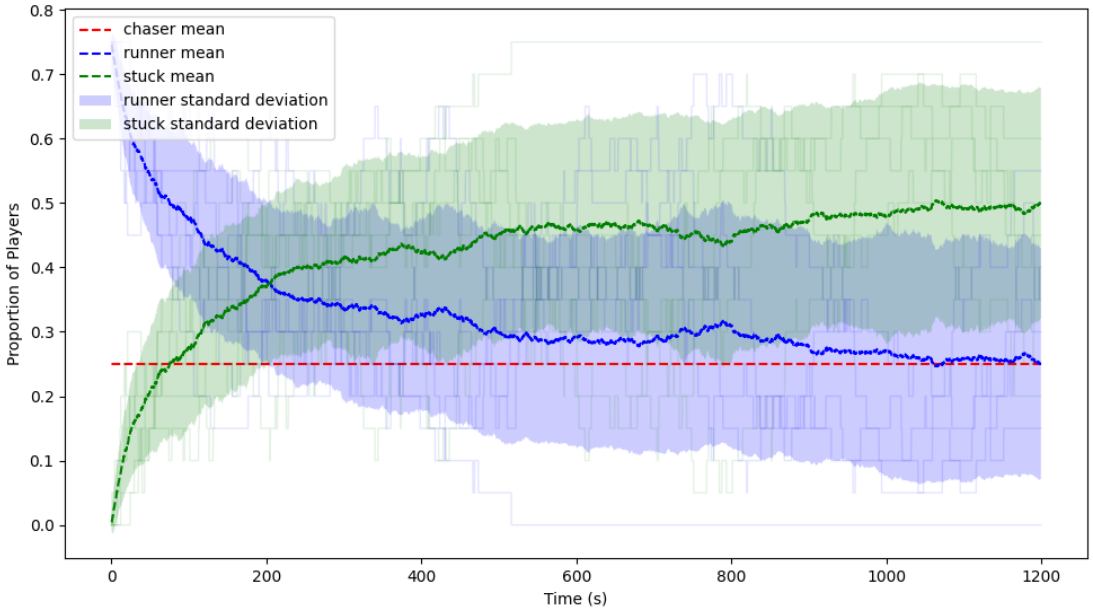

Fig. 11 Average of 100 individual runs of stuck-in-the-mud with 5 chasers and 15 runners initially, with the standard deviations for runners and stuck populations shown, as well as 5 random individual runs for each.

For my final simulation of stuck-in-the-mud, I began with chaser and runner populations equal at 10 agents each. With these initial populations, the game was finally able to conclude, with all runners ending up stuck at a time roughly around 500 seconds.

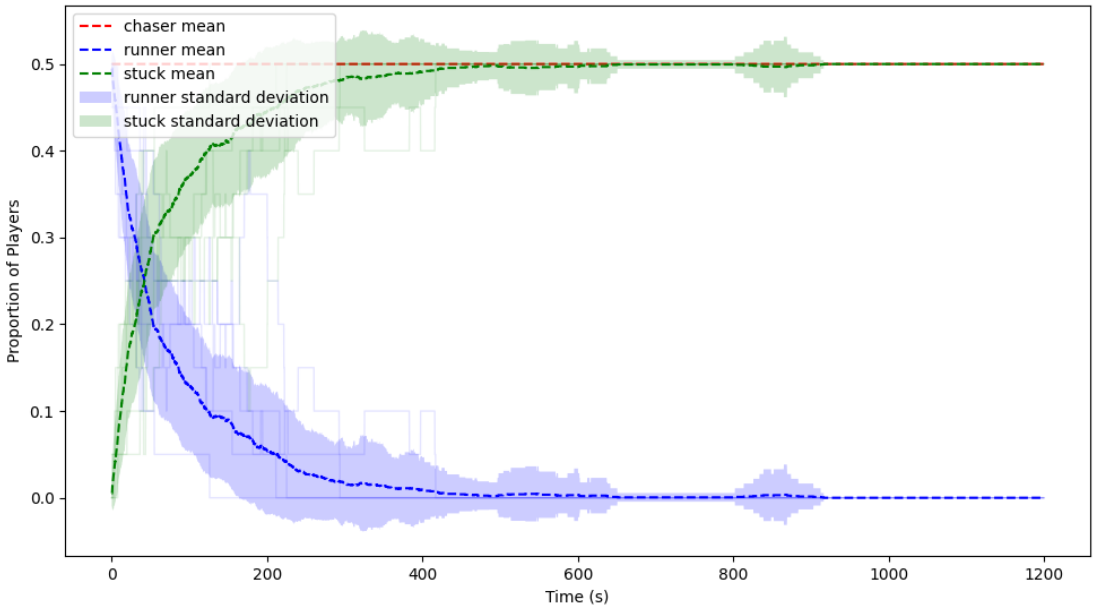

Fig. 12 Average of 100 individual runs of stuck-in-the-mud with 10 chasers and 10 runners initially, with the standard deviations for runners and stuck populations shown, as well as 5 random individual runs for each.

6| Discussion

After deriving equations for a deterministic model for chain tig in section 3.1 and designing and simulating an agent-based model for the same game in sections 4 and 5, I am now able to compare each method of modelling to one another and discuss benefits and flaws in both.

From the deterministic model, I derived that equation (3.6) should give an estimate for the normalised population of runners when given the initial runner and chaser populations. This curve matched the shape of the simulated mean curve seen in section 5.1, meaning that I can confidently say my simulation worked adequately.

Next, I decided to assess the differences produced by the kinetic model used to estimate transition rate nu, and those from the measured values for each simulation shown in section 5.2. In Fig. 7, Fig. 8, and Fig. 9, I have plotted both of these values so they can easily be compared to one another, and for the most part they look accurate, meaning the kinetic model is a viable method of estimating the transition rate. I did notice that as domain size increased, the difference between these two values became narrower. Due to the fact that this is only a sample of three simulations, I cannot come to a fair conclusion, however it may be interesting to try more domain sizes and plot them with the transition rates to observe any potential correlation. If I were to speculate a reason for this behaviour though, I would think it was due to the density of agents where for a smaller domain, the rate of change will be much higher, and therefore meaning there will be a higher margin of error.

My preliminary simulations for stuck-in-the-mud were interesting, as they seemed to correlate with the limits (3.15) (3.16) obtained from the deterministic model, in which I stated that unless the capture rate γ exceeds the release rate β, the game will never reach the point of all runners being stuck. I assume that we would see this being true if we calculated predictions and measured values for the capture and release rate for these simulations. I would expect the simulations seen in Fig. 10 and Fig. 11 to have higher release rates, and for Fig. 12 to have a higher capture rate, resulting in a quick victory for the chasers.

I believe that while the deterministic model was able to provide an accurate estimate for population dynamics, agent-based models are much more suited to demonstrate realistic human behaviour due to their inherent randomness, and because of this I think that they would also be applicable in areas of epidemiology. This is further supported by the example of H. Situngkir [14] applying an ABM in a very similar area.

7| Conclusions and Further Work

Throughout this project I have investigated two different methods for modelling children's playground games that have a focus on transfer of status through close proximity. Because of the nature of these games, I can relatively easily draw links to areas such as epidemiology, as infections spread in a similar way. I have done this by designing mathematical models for both chain tig and stuck-in-the-mud, as well as agent-based models for both. The reason for doing so was to have direct parallels between the two and determine if they had a place in accurately modelling population dynamics. After assessing the results from each of the models, I came to the conclusion that ABM would be the ideal way of modelling playground games. The agent-based model created for the purpose of this project is far from perfect, and therefore future work could include improving this to make it more accurate to real life populations. Some examples of this could be:

7.1 Adding intentionality into my agents

The agents in the existing model currently all move in random directions, which is obviously very unlike a chasing game. To improve on this, I could add mechanics such as chasers actually hunting runners, as well as the runners trying to find optimal paths away from close

chasers. I could also incorporate strategies between teammates in order to attempt to confuse and catch opponents by surprise.

7.2 Discussing real life models

Another thing that could be improved upon is the natural variety between different players. Some players would be faster, some would be bigger, some would tire quicker, etc. These are all human factors that could potentially have an effect on the outcome of some simulations, and it would be interesting to implement. If applying this to a model specifically for epidemiological purposes, these random factors could instead be vulnerability to infection, age, etc.

References

[1] 'Chasing games'[Online]. Available https://www.bl.uk/playtimes/articles/chasing-games

[2] I. A. Opie and P. Opie, *Children's Games in Street and Playground: Chasing, Catching, Seeking, Hunting, Racing, Duelling, Exerting, Daring, Guessing, Acting, Pretending*. Clarendon P., 1969, pp.89,110-111.

[3] W. Casey et al., 'How Signalling Games Explain Mimicry at Many Levels: From Viral Epidemiology to Human Sociology', *Res Sq*, Aug. 2020[Online]. Available http://dx.doi.org/10.21203/rs.3.rs-51959/v1

[4] Wikipedia contributors, *Compartmental models in epidemiology*, Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Compartmental_models_in_epidemiology&oldid=1021423575

[5] W. H. Hamer, 'Epidemic disease in England', *Lancet*, vol. 1, pp. 733–739, 1906.

[6] W. O. Kermack and A. G. McKendrick, 'A contribution to the mathematical theory of epidemics', *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, vol. 115, no. 772, pp. 700–721, Aug. 1927.

[7] J. D. Murray, Ed., *Mathematical Biology: I. An Introduction*. Springer, New York, NY, 2002.

[8] J. J. Bissell, 'Generalised Compartmental Modelling of Health Epidemics', in *Tipping Points*, unknown, 2015, pp. 1–20.

[9] C. Caiado et al., 'Compartmental modelling of social dynamics with generalised peer incidence', *Math. Models Methods Appl. Sci.*, vol. 24, no. 4, pp. 719–750, Apr. 2014.

[10] J. Brownlee, (2019, November.3), *A Gentle Introduction to Monte Carlo Sampling for Probability*. [Online]. Available: https://machinelearningmastery.com/monte-carlo-sampling-for-probability/

[11] T. C. Schelling, 'Dynamic models of segregation', *J. Math. Sociol.*, vol. 1, no. 2, pp. 143–186, Jul. 1971.

[12] E. Bonabeau, ‘Agent-based modeling: methods and techniques for simulating human systems’, *Proc. Natl. Acad. Sci. U. S. A.*, vol. 99 Suppl 3, pp. 7280–7287, May 2002.

[13] J. J. Bissell, ‘The decline in the British bank population since 1810 obeys a law of negative compound interest’, *Bus. Hist.*, vol. 59, no. 5, pp. 802–813, Jul. 2017.

[14] H. Situngkir, ‘Epidemiology Through Cellular Automata: Case of Study Avian Influenza in Indonesia’, *arXiv [nlin.CG]*, Mar. 17, 2004[Online]. Available http://arxiv.org/abs/nlin/0403035

[15] J. J. Bissell, ‘`Your It!' : Mathematical Modelling of Children's Playground Chasing Games’, 2012, pp. 2-6.

Appendices

Simulation.java

```
import javax.swing.*;

import drawing.Canvas;
import group.Group;
import tools.Utils;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class Simulation {

	//objects
	private JFrame frame;
	private Canvas canvas;

	//instantiating
	Group group = new Group();

	private final static int WINDOW_SIZE = 200;
	private final static int GAMEMODE = 1;
	private final static int NO_OF_RUNS = 100;
	private final static int RUNTIME = 6000;
	private final static double playerSize = (double)(2);

	private final int chaserCount = 10;
	private final int runnerCount = 10;

	private boolean continueRunning;
	private int runtimeCounter;
	private int runNumber;

	private double nuModel;

	private float[][] chaserArray = new float[RUNTIME][NO_OF_RUNS];
	private float[] meanC = new float[RUNTIME];
	private double[] standDevC = new double[RUNTIME];
	private double[] gradientC = new double[RUNTIME];
	private double[] nuC = new double[RUNTIME];

	private float[][] runnerArray = new float[RUNTIME][NO_OF_RUNS];
	private float[] meanR = new float[RUNTIME];
	private double[] standDevR = new double[RUNTIME];
```

```
	private double[] gradientR = new double[RUNTIME];
	private double[] nuR = new double[RUNTIME];

	private float[][] stuckArray = new float[RUNTIME][NO_OF_RUNS];
	private float[] meanS = new float[RUNTIME];
	private double[] standDevS = new double[RUNTIME];
	private double[] gradientS = new double[RUNTIME];


	@SuppressWarnings("unused")
	public Simulation() {
		runNumber = 0;

		setupGUI();

		for(int i = 0; i < NO_OF_RUNS; i++) {
			group.clearPlayers();
			for(int j = 0; j < chaserCount; j++) {
				group.addChaser(canvas, playerSize, playerSize +
(Math.random() * (WINDOW_SIZE - (playerSize * 2))), playerSize + (Math.random() *
(WINDOW_SIZE - (playerSize * 2))));
			}
			for(int j = 0; j < runnerCount; j++) {
				group.addRunner(canvas, playerSize, playerSize +
(Math.random() * (WINDOW_SIZE - (playerSize * 2))), playerSize + (Math.random() *
(WINDOW_SIZE - (playerSize * 2))));
			}
			gameLoop(runNumber);
			runNumber++;
			System.out.println(runNumber);
		}

		nuModel = (2 * (group.getImpactParameter() * playerSize) *
group.getSPEED() * (chaserCount + runnerCount)) / (WINDOW_SIZE * WINDOW_SIZE);

		for(int i = 0; i < RUNTIME; i++) {
			double squareDiffC = 0;
			double squareDiffR = 0;
			double squareDiffS = 0;
			double varianceC = 0;
			double varianceR = 0;
			double varianceS = 0;
			float time = (float)(group.getDeltaTime() * i);

			for(int j = 0; j < NO_OF_RUNS; j++) {
				chaserArray[i][j] = chaserArray[i][j] / group.players.size();
				runnerArray[i][j] = runnerArray[i][j] / group.players.size();

				meanC[i] = meanC[i] + chaserArray[i][j];
				meanR[i] = meanR[i] + runnerArray[i][j];

				if(GAMEMODE == 2) {
					stuckArray[i][j] = stuckArray[i][j] /
group.players.size();
					meanS[i] = meanS[i] + stuckArray[i][j];
				}
			}

			meanC[i] = meanC[i] / NO_OF_RUNS;
			meanR[i] = meanR[i] / NO_OF_RUNS;
```

```
                    if(GAMEMODE == 2) {
                          meanS[i] = meanS[i] / NO_OF_RUNS;
                    }

                    for(int j = 0; j < NO_OF_RUNS; j++) {
                          squareDiffC = Math.pow((chaserArray[i][j] - meanC[i]), 2);
                          squareDiffR = Math.pow((runnerArray[i][j] - meanR[i]), 2);
                          varianceC = varianceC + squareDiffC;
                          varianceR = varianceR + squareDiffR;

                          if(GAMEMODE == 2) {
                                squareDiffS = Math.pow((stuckArray[i][j] - meanS[i]),
2);
                                varianceS = varianceS + squareDiffS;
                          }
                    }

                    varianceC = varianceC / NO_OF_RUNS;
                    varianceR = varianceR / NO_OF_RUNS;

                    standDevC[i] = Math.sqrt(varianceC);
                    standDevR[i] = Math.sqrt(varianceR);

                    if(GAMEMODE == 2) {
                          varianceS = varianceS / NO_OF_RUNS;
                          standDevS[i] = Math.sqrt(varianceS);
                    }

              }

              for(int i = 0; i < RUNTIME; i++) {
                    float time = (float)(group.getDeltaTime() * i);

                    if(i == 0 || i == (RUNTIME-1)) {
                          gradientC[i] = 0;
                          gradientR[i] = 0;
                          if(GAMEMODE == 2) {
                                gradientS[i] = 0;
                          }
                    }
                    else {
                          //LOOK HERE LATER, CANT CALCULATE i+1
                          //gradientC[i] = (meanC[i+1] - meanC[i - 1]) / (2 * 0.2);
                          gradientC[i] = (meanC[i + 1] - meanC[i - 1]) / (2 * 0.2);
                          gradientR[i] = (meanR[i + 1] - meanR[i - 1]) / (2 * 0.2);
                          if(GAMEMODE == 2) {
                                gradientS[i] = (meanS[i + 1] - meanS[i - 1]) / (2 *
0.2);
                          }
                    }

                    if(meanC[i] > 0.2 && meanC[i] < 0.8) {
                          nuC[i] = gradientC[i] / (meanC[i] * (1 - meanC[i]));
                          nuR[i] = gradientR[i] / (meanR[i] * (1 - meanR[i]));

                          try {
                                FileWriter myWriter = new
FileWriter("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\nu.txt", true);
                                myWriter.write(time + "\t" + nuC[i] + "\t" + nuR[i] +
"\t" + nuModel + "\t" + -nuModel + "\n");
                                myWriter.close();
```

```
                                System.out.println("Successfully wrote to the file.");
                        } catch (IOException e) {
                                System.out.println("An error occurred.");
                                e.printStackTrace();
                        }
                }


                try {
                        FileWriter myWriter = new
FileWriter("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\chasers.txt",
true);
                        myWriter.write(time + "");
                        for(int j = 0; j < NO_OF_RUNS; j++) {
                                myWriter.write("\t" + chaserArray[i][j]);
                        }
                        myWriter.write("\n");
                        myWriter.close();
                        System.out.println("Successfully wrote to the file.");
                } catch (IOException e) {
                        System.out.println("An error occurred.");
                        e.printStackTrace();
                }

                try {
                        FileWriter myWriter = new
FileWriter("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\resultsC.txt",
true);
                        myWriter.write(time + "\t" + meanC[i] + "\t" + standDevC[i] +
"\t" + gradientC[i] + "\n");
                        myWriter.close();
                        System.out.println("Successfully wrote to the file.");
                } catch (IOException e) {
                        System.out.println("An error occurred.");
                        e.printStackTrace();
                }

                try {
                        FileWriter myWriter = new
FileWriter("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\runners.txt",
true);
                        myWriter.write(time + "");
                        for(int j = 0; j < NO_OF_RUNS; j++) {
                                myWriter.write("\t" + runnerArray[i][j]);
                        }
                        myWriter.write("\n");
                        myWriter.close();
                        System.out.println("Successfully wrote to the file.");
                } catch (IOException e) {
                        System.out.println("An error occurred.");
                        e.printStackTrace();
                }

                try {
                        FileWriter myWriter = new
FileWriter("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\resultsR.txt",
true);
                        myWriter.write(time + "\t" + meanR[i] + "\t" + standDevR[i] +
"\t" + gradientR[i] + "\n");
                        myWriter.close();
                        System.out.println("Successfully wrote to the file.");
```

```
                } catch (IOException e) {
                        System.out.println("An error occurred.");
                        e.printStackTrace();
                }

                if(GAMEMODE == 2) {
                        try {
                                FileWriter myWriter = new
FileWriter("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\stuck.txt", true);
                                myWriter.write(time + "");
                                for(int j = 0; j < NO_OF_RUNS; j++) {
                                        myWriter.write("\t" + stuckArray[i][j]);
                                }
                                myWriter.write("\n");
                                myWriter.close();
                                System.out.println("Successfully wrote to the file.");
                        } catch (IOException e) {
                                System.out.println("An error occurred.");
                                e.printStackTrace();
                        }

                        try {
                                FileWriter myWriter = new
FileWriter("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\resultsS.txt",
true);
                                myWriter.write(time + "\t" + meanS[i] + "\t" +
standDevS[i] + "\t" + gradientS[i] + "\n");
                                myWriter.close();
                                System.out.println("Successfully wrote to the file.");
                        } catch (IOException e) {
                                System.out.println("An error occurred.");
                                e.printStackTrace();
                        }
                }
        }
    }

    //method to set the GUI to my desired specifications
    private void setupGUI() {

        frame = new JFrame();
        frame.setTitle("Simulator");
        frame.setSize(WINDOW_SIZE + 16, WINDOW_SIZE + 39);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);


        canvas = new Canvas();

        frame.add(canvas);
    }

    //the main game loop, will loop until 'continueRunning' is False
    @SuppressWarnings("unused")
    private void gameLoop(int runNumber) {
        continueRunning = true;
        runtimeCounter = 0;

        while (continueRunning) {
```

```
                    group.Simulate(canvas, GAMEMODE);
                    chaserArray[runtimeCounter][runNumber] = group.chasingCount;
                    runnerArray[runtimeCounter][runNumber] = group.runningCount;
                    if(GAMEMODE == 2) {
                          stuckArray[runtimeCounter][runNumber] = group.stuckCount;
                    }
                    runtimeCounter++;

                    if(runtimeCounter >= RUNTIME) {
                          continueRunning = false;
                          Utils.pause(500);
                    }
             }
       }

       //main function
       @SuppressWarnings("unused")
       public static void main(String[] args) throws IOException {
             System.out.println();
             File arrayC = new
File("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\chasers.txt");
             arrayC.delete();
             arrayC.createNewFile();

             File resultsC = new
File("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\resultsC.txt");
             resultsC.delete();
             resultsC.createNewFile();

             File arrayR = new
File("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\runners.txt");
             arrayR.delete();
             arrayR.createNewFile();

             File resultsR = new
File("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\resultsR.txt");
             resultsR.delete();
             resultsR.createNewFile();

             File nu = new
File("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\nu.txt");
             nu.delete();
             nu.createNewFile();

             if(GAMEMODE == 2) {
                    File arrayS = new
File("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\stuck.txt");
                    arrayS.delete();
                    arrayS.createNewFile();

                    File resultsS = new
File("C:\\Users\\noah\\Desktop\\University\\proj\\Simulation\\resultsS.txt");
                    resultsS.delete();
                    resultsS.createNewFile();
             }

             new Simulation();

       }

}
```

Player.java

```
package group;

import drawing.Canvas;
import geometry.CartesianCoordinate;

//basic class of player, contains the basic methods including movement,
//collision, and drawing
public class Player {
	private Canvas myCanvas;

	protected CartesianCoordinate pos;
	private double angle;
	public double size;
	public boolean chasing;
	public boolean stuck;
	private boolean penDown;

	//gives current x position of the player
	public double getPositionX() {
		return pos.getX();
	}

	//gives current y position of the player
	public double getPositionY() {
		return pos.getY();
	}

	public Player(Canvas myCanvas) {
		this.myCanvas = myCanvas;
		this.pos = new CartesianCoordinate(0, 0);
		this.setAngle(0);
		this.chasing = false;
		this.stuck = false;
		this.penDown = false;
	}

	public void move(double i) {
		double x = pos.getX();
		double y = pos.getY();
		CartesianCoordinate oldPos = new CartesianCoordinate(pos.getX(),
pos.getY());
		x += i * Math.cos(Math.toRadians(getAngle()));
		y += i * Math.sin(Math.toRadians(getAngle()));
		pos = new CartesianCoordinate(x, y);
		if (this.penDown == true) {
			myCanvas.drawLineBetweenPoints(oldPos, pos);
		}

	}

	public void turn(double theta) {
		setAngle(getAngle() + theta);
	}

	//method to detect collision to the edges of the screen, as well as the obstacle
```

```
	//the player has been programmed to 'bounce' off at the same angle it came in at
	public void edgeCollision(double maxX, double maxY) {
		if (pos.getX() - size <= 0) {
			this.setAngle(0);
		}
		if (pos.getX() + size >= maxX) {
			this.setAngle(180);
		}
		if (pos.getY() - size <= 0) {
			this.setAngle(90);
		}
		if (pos.getY() + size >= maxY) {
			this.setAngle(270);
		}
	}

	//draws the player
	public void draw() {
		this.turn(-90);
		this.move(size/2);
		this.turn(90);
		this.putPenDown();
		for (int i = 0; i <= 360; i++) {
			if(chasing == true) {
				this.turn(90);
				this.move(size);
				this.putPenUp();
				this.move(-size);
				this.turn(-90);
				this.putPenDown();
			}
			this.move((size/2) * Math.toRadians(1));
			this.turn(1);
			//Utils.pause(1);

		}
		if(stuck == true) {
			this.turn(90);
			this.move(size);
			this.putPenUp();
			this.move(-size/2);
			this.turn(-90);
			this.move(size/2);
			this.putPenDown();
			this.move(-size);
			this.putPenUp();
			this.move(size/2);
			this.turn(-90);
			this.move(size/2);
			this.turn(90);
			this.putPenDown();
		}
		this.putPenUp();
		this.turn(90);
		this.move(size/2);
		this.turn(-90);


	}
```

```
	public void putPenUp() {
		this.penDown = false;
	}

	public void putPenDown() {
		this.penDown = true;
	}

	public double getAngle() {
		return angle;
	}

	public void setAngle(double angle) {
		this.angle = angle;
	}

}
```

DynamicPlayer.java

```
package group;

import drawing.Canvas;

//more advanced class inherited from the player class, includes methods
//to get and set angle and speed, as well as updating each player after
//a variable changes
public class DynamicPlayer extends Player {
	public static double SPEED = 10;
	public static double deltaTime = 0.2;
	public static double MAX_FOV = 80;
	private int counter;
	private double turnAng;

	//moves the player into a starting position other than 0,0
	public DynamicPlayer(Canvas canvas, double size, double xPosition, double
yPosition, double initialAngle, int initialCounter, boolean tagState) {
		super(canvas);
		this.size = size;
		move(xPosition);
		turn(90);
		move(yPosition);
		turn(-90);
		turn(initialAngle);
		draw();
		this.chasing = tagState;
	}

	//update method to change angle of movement of the player and
	//amount of steps it should take given the refresh rate
	public void update() {


		if (counter == 0) {
			counter = (int)(Math.random() * 20);
			//angular velocity is limited to a set FOV in front of the player,
			//in both positive and negative directions
			turnAng = (-(MAX_FOV/2) + (Math.random()*MAX_FOV));
```

```
            }
            else {
                  if(this.stuck == false) {
                        //allows for a gradual turn rather than snapping in altering
                        //directions
                        double theta = turnAng * deltaTime;
                        turn(theta);

                        double dist = SPEED * deltaTime;
                        move(dist);
                        counter--;
                  }
            }
      }
}
```

Group.java

```
package group;

import drawing.Canvas;

//more advanced class inherited from the player class, includes methods
//to get and set angle and speed, as well as updating each player after
//a variable changes
public class DynamicPlayer extends Player {
      public static double SPEED = 10;
      public static double deltaTime = 0.2;
      public static double MAX_FOV = 80;
      private int counter;
      private double turnAng;

      //moves the player into a starting position other than 0,0
      public DynamicPlayer(Canvas canvas, double size, double xPosition, double
yPosition, double initialAngle, int initialCounter, boolean tagState) {
            super(canvas);
            this.size = size;
            move(xPosition);
            turn(90);
            move(yPosition);
            turn(-90);
            turn(initialAngle);
            draw();
            this.chasing = tagState;
      }

      //update method to change angle of movement of the player and
      //amount of steps it should take given the refresh rate
      public void update() {


            if (counter == 0) {
                  counter = (int)(Math.random() * 20);
                  //angular velocity is limited to a set FOV in front of the player,
                  //in both positive and negative directions
                  turnAng = (-(MAX_FOV/2) + (Math.random()*MAX_FOV));
            }
            else {
```

```
                    if(this.stuck == false) {
                          //allows for a gradual turn rather than snapping in altering
                          //directions
                          double theta = turnAng * deltaTime;
                          turn(theta);

                          double dist = SPEED * deltaTime;
                          move(dist);
                          counter--;
                    }
              }
        }
}
```

graphingProgram.py

```
import matplotlib.pyplot as plt
import numpy as np

dataSingleC = np.loadtxt("chasers.txt")
dataC = np.loadtxt("resultsC.txt")
dataSingleR = np.loadtxt("runners.txt")
dataR = np.loadtxt("resultsR.txt")
dataNu = np.loadtxt("nu.txt")

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

plt.figure()

with open("chasers.txt") as f:
    n_cols = len(f.readline().split("\t"))
    print (n_cols)

for x in range(1, round(n_cols)):
    plt.plot(dataSingleC[:,0], dataSingleC[:,x], color='r', linewidth=1,
alpha=0.2)

plt.plot(dataC[:,0], dataC[:,1], linestyle = 'dashed', color='r', label="chaser
mean")
plt.fill_between(dataC[:,0], dataC[:,1] - dataC[:,2], dataC[:,1] + dataC[:,2],
facecolor='r', alpha=0.2, label="chaser standard deviation")

for x in range(1, round(n_cols)):
    plt.plot(dataSingleR[:,0], dataSingleR[:,x], color='b', linewidth=1,
alpha=0.2)

plt.plot(dataR[:,0], dataR[:,1], linestyle = 'dashed', color='b', label="runner
mean")
plt.fill_between(dataR[:,0], dataR[:,1] - dataR[:,2], dataR[:,1] + dataR[:,2],
facecolor='b', alpha=0.2, label="runner standard deviation")

plt.legend(loc="upper left")

plt.xlabel("Time (s)")
plt.ylabel("Proportion of Players")

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

plt.figure()
```

```
plt.axhline(y=0, color='black', linewidth=1)

plt.plot(dataC[:,0], dataC[:,3], linestyle = 'solid', color='r', label="chaser
gradient")
plt.plot(dataR[:,0], dataR[:,3], linestyle = 'solid', color='b', label="runner
gradient")

plt.legend(loc="upper right")

plt.xlabel("Time (s)")
plt.ylabel("Rate of Change")

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

plt.figure()

plt.axhline(y=0, color='black', linewidth=1)

plt.plot(dataNu[:,0], dataNu[:,1], linestyle = 'solid', color='r', alpha=0.5,
label="chaser transition rate ($\\nu$)")
plt.plot(dataNu[:,0], dataNu[:,2], linestyle = 'solid', color='b', alpha=0.5,
label="runner transition rate ($\\nu$)")

mean_lineC = np.array([np.mean(dataNu[:,1]) for i in range(len(dataNu[:,0]))])
mean_lineR = np.array([np.mean(dataNu[:,2]) for i in range(len(dataNu[:,0]))])

std_lineC = np.array([np.std(dataNu[:,1]) for i in range(len(dataNu[:,0]))])
std_lineR = np.array([np.std(dataNu[:,2]) for i in range(len(dataNu[:,0]))])

plt.plot(dataNu[:,0], mean_lineC, linestyle = 'dashed', color='r', label="chaser
measured $\\nu$ mean")
plt.plot(dataNu[:,0], mean_lineR, linestyle = 'dashed', color='b', label="runner
measured $\\nu$ mean")

plt.plot(dataNu[:,0], dataNu[:,3], linestyle = 'solid', color='r', label="chaser
kinetic model $\\nu$")
plt.plot(dataNu[:,0], dataNu[:,4], linestyle = 'solid', color='b', label="runner
kinetic model $\\nu$")

plt.fill_between(dataNu[:,0], mean_lineC - std_lineC, mean_lineC + std_lineC,
facecolor='r', alpha=0.2, label="chaser $\\nu$ standard deviation")
plt.fill_between(dataNu[:,0], mean_lineR - std_lineR, mean_lineR + std_lineR,
facecolor='b', alpha=0.2, label="runner $\\nu$ standard deviation")

print(np.mean(dataNu[:,1]))
print(np.mean(dataNu[:,2]))

plt.legend(loc="upper right")

plt.xlabel("Time (s)")
plt.ylabel("Transition Rate ($\\nu$)")

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

plt.show()
```

graphinProgramSIM.py

```
import matplotlib.pyplot as plt
import numpy as np

dataSingleC = np.loadtxt("chasers.txt")
dataC = np.loadtxt("resultsC.txt")
```

```
dataSingleR = np.loadtxt("runners.txt")
dataR = np.loadtxt("resultsR.txt")
dataSingleS = np.loadtxt("stuck.txt")
dataS = np.loadtxt("resultsS.txt")

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

plt.figure()

with open("chasers.txt") as f:
    n_cols = len(f.readline().split("\t"))
    print (n_cols)

plt.plot(dataC[:,0], dataC[:,1], linestyle = 'dashed', color='r', label="chaser
mean")

for x in range(1, round(n_cols)):
    plt.plot(dataSingleR[:,0], dataSingleR[:,x], color='b', linewidth=1,
alpha=0.1)

plt.plot(dataR[:,0], dataR[:,1], linestyle = 'dashed', color='b', label="runner
mean")
plt.fill_between(dataR[:,0], dataR[:,1] - dataR[:,2], dataR[:,1] + dataR[:,2],
facecolor='b', alpha=0.2, label="runner standard deviation")

for x in range(1, round(n_cols)):
    plt.plot(dataSingleS[:,0], dataSingleS[:,x], color='g', linewidth=1,
alpha=0.1)

plt.plot(dataS[:,0], dataS[:,1], linestyle = 'dashed', color='g', label="stuck
mean")
plt.fill_between(dataS[:,0], dataS[:,1] - dataS[:,2], dataS[:,1] + dataS[:,2],
facecolor='g', alpha=0.2, label="stuck standard deviation")

plt.legend(loc="upper left")

plt.xlabel("Time (s)")
plt.ylabel("Proportion of Players")

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

plt.figure()

plt.axhline(y=0, color='black', linewidth=1)

plt.plot(dataC[:,0], dataC[:,3], linestyle = 'solid', color='r', label="chaser
gradient")
plt.plot(dataR[:,0], dataR[:,3], linestyle = 'solid', color='b', label="runner
gradient")
plt.plot(dataR[:,0], dataS[:,3], linestyle = 'solid', color='g', label="stuck
gradient")

plt.legend(loc="upper left")

plt.xlabel("Time (s)")
plt.ylabel("Rate of Change")

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

plt.show()
```