Agent-Based Modeling: Analysis and Simulation with Python

Jennefer Maldonado
Department of Mathematics & Computer Science
Adelphi University
Garden City, NY 11530

May 21, 2021

Abstract

Agent-based modeling has applications in game theory, biology, business, economics, and social sciences. The goal of these models is to gain insight into the behavior of a population of agents following rules of a game or a system. Agents act in their own best self-interest playing the strategies that improve their status in the system. The prisoner's dilemma is an example of a game where agents acting in their own self-interest turn out to have sub-optimal results. There are three main strategies in the prisoner's dilemma, always cooperating, always defecting, and cooperating when your opponent does. Cooperating leads to the best possible outcome for both agents, but it is rarely the choice made in the dilemma. Studying the iterated version of the prisoner's dilemma provides insight into what strategies will survive and which ones will die out as a result of natural selection. Agent-based modeling software was readily available as its applications spread to multiple disciplines in the late 1900s and early 2000s. Most of this software is now outdated which leaves the question of what new techniques can be used for agent-based modeling. We have developed Python code to study the iterated prisoner's dilemma that will allow for a deeper understanding into the behavior of the agents and their strategies. Our code also is a new development to agent-based modeling software for a more updated version of a toolkit necessary to study these complex systems.

Contents

1 Introduction

1.1 What is the Prisoner's Dilemma?

The Prisoner's Dilemma became the most influential discovery in the history of game theory when Merrill Flood and Melvin Dresher created a simple game that challenged a part of the theory. The game can be described as follows, two prisoners are questioned on whether or not the other committed a crime. There are four cases that can occur. The first being if both prisoners confess to committing the crime, each prisoner will get jail time. The second and third cases are if one prisoner confesses and the second prisoner blames the first, the prisoner who confesses will get an extended jail sentence. The fourth and final case is if both prisoners state that the other committed the crime, they both receive a shortened jail sentence. This can be visualized as follows.

	Cooperate	Defect
Cooperate	3 years, 3 years	0 years, 5 years
Defect	5 years, 0 years	1 year, 1 year

Table 1: Example of a payoff matrix for the prisoner's dilemma

This is called a payoff matrix. We will describe more about payoff matrices in Section 2.4.2 and how they can be applied to game theory. A prisoner can cooperate and confess to the police or defect the blame onto the other. So if both prisoners confess they each get three years, if they both defect they each get one year, and if one confesses and is blamed the cooperator goes free and the defector gets five years. The prisoner's dilemma can be perceived as just a story, but it has many implications in mathematics and the real world as we will discuss in Section 1.6. If we look back at Table 1, the best outcome for both prisoners would be to defect ultimately blaming each other without incriminating themselves since it leads to the shortest jail sentence. We can believe this is the rational or "common sense" outcome, but the choice made by each prisoner is not influenced by the other. This means that each prisoner is acting in their own self interest. Looking at Table 1, with your own self interest in mind, the best chance to escape a prison sentence is to cooperate in hopes the other prisoner will defect. This mentality leads to both prisoners cooperating which leads to both being sentenced to 3 years in jail. Flood and Dresher believed that one of the people who could solve the prisoner's dilemma would be John von Neumann, by deriving a new theory to show the rational choice in the dilemma is cooperation. When a solution never came they both switched their beliefs to the fact that the prisoner's dilemma will never be solved. [20]

1.2 John von Neumann

John von Neumann is a member of the group of celebrity mathematicians that laypersons often associate with being one of the inventors of the digital computer. He always carried Chinese puzzles in his pockets and was never described as shy by his colleagues. His early interests lied in set theory and he created a definition of ordinal numbers that is still used today by the age of 20. As much as he was the life of the party, at times he would begin writing down his ideas in the middle of the dance floor thriving off of the noisiness and commotion. Von Neumann had a very sarcastic sense of humor as well as an incredible memory. He loves many types of children toys and after he obtained a box of Tinker Toys he realized one of his theoretical interests which was whether a machine could be built to reproduce itself. Von Neumann was a workaholic sleeping few hours during the night and working long hours when he was awake. In the years between 1925 and 1940, he quickly published multiple papers on logic, set theory,group theory, ergodic theory, and operator theory. Von Neumann mainly focused on applied mathematics during his career but not only focused on the actual mathematical theory but the philosophical undertones as well. By the late 1920s he was working on game theory, but focused on how the brain could be represented by the relays and circuits of a computer. Some historians believe that von Neumann's personal cynicism led him to study game theory instead of the rest of his interests but since game theory is a mathematical way to study conflict this is unlikely. Despite all

of his interests, von Neumann impacted game theory the most by publishing his 1928 paper On The Theory of Games of Strategy which distinguished game theory as a new field. [20]

1.3 John Nash

After von Neumann, John Nash became a leader in game theory developments. Nash studied mathematics at Princeton University where he created a game that was played with markers and hexagonal tiles or cells. He developed the game so that there was a correct way to play it but it is very difficult to discover the strategy. In 1952 the game was promoted with the name "Hex" by the Parker Brothers. Nash extended game theory to a place von Neumann did not which is the realm of non-cooperative games which means all alliances are forbidden. Even though von Neumann did study zero-sum games which follow the same principle, Nash studied non-zero-sum games and games with three or more players. One of von Neumann's important findings, the minimax theorem, showed that any two rational beings with completely opposing interests can settle on a rational course of action with confidence that the other will do the same. Nash was able to extend this result by stating the equilibrium solutions exist for non-zero-sum two-person games. Nash's strategy to studying non-cooperative games was to focus on "equilibrium points" which are defined as the payoffs where the players have no regrets. If a player were to choose a strategy where their opponent gains more points for choosing the same strategy they will have regrets in the post game analysis. So Nash picked the equilibrium point in this situation to be the one where the players have near balanced payoffs and at the post game analysis none would regret their choices. He argued that if a player were to change their strategy, it is not a rational strategy. Nash was able to prove that every two-person finite game has at least one equilibrium point. This means that often times equilibrium points will not always look rational and can have disagreeable qualities. [20] Nash was later awarded the Nobel Prize in Economics for this work more specifically known as Nash Equilibria.

1.4 Robert Axelrod

In 1980, Robert Axelrod conducted the most well-known study of the iterated prisoner's dilemma and it's strategies by running a group of computer tournaments. Axelrod is a professor of political science at the University of Michigan and works on interdisciplinary work including agent-based modeling. He wrote a book called *The Evolution of Cooperation* which is considered one of the significant discoveries of game theory. Axelrod stumbled upon game theory as a math major during his undergraduate years at the University of Chicago. Later he obtained his doctorate at Yale in political science where he studied the conflict of interest. Axelrod distinguished the iterated prisoner's dilemma from a one time dilemma by defining it as a dilemma that can give us information about the future. We can see the benefits of cooperation for the future, yet no one takes the future into consideration when making decisions regarding strategy prioritizing immediate gain. This means that for a one time game, defection is usually chosen but in an iterated dilemma, cooperation becomes a idea because players take into account the future and present state of the game. Axelrod brought together game theorists, psychologists, sociologists, political scientists, and economists to create their own "winning" strategies for the iterated prisoner's dilemma. Then these strategies were put into a computer simulated tournament to determine which one was the dominant strategy. [20] This experiment showed the "tit-for-tat" strategy was the strongest over cooperation and defection, we will discuss what this means more in the following sections.

1.5 Iterated Prisoner's Dilemma

The iterative prisoner's dilemma is an extension of the prisoner's dilemma because instead of focusing solely on one decision made per game, it focuses on multiple decisions made by the same players over multiple games. This is important in studying the changes in participant behavior throughout a game and like discussed prior, can emulate evolution similar to biological studies or long term patterns in a system. There are many strategies for the iterative dilemma that were proposed, more than just the all cooperation and defection strategies for the prisoner's dilemma we discussed in Section 1.1. They all performed poorly in comparison to the tit-for-tat strategy which allowed it to become of the three main strategies for the iterative prisoner's dilemma. The tit-for-tat strategy is different from the all cooperation and defection

strategies because it originally tries to cooperate with whatever it's opponent strategy is. If it's opponent defects, then the tit-for-tat strategy will defect as well, but if the opponent cooperates then tit-for-tat will continue to cooperate. The iterative prisoner's dilemma allows for researchers to see how the choices of the agents using the tit-for-tat strategy will change depending on their opponents choices. For this project we will focus on both the prisoner's dilemma and the iterated version of the game studying the outcomes of both, how to model them, and their applications in the real world.

1.6 Survey of Real World Applications

William Poundstone believes the discover of the prisoner's dilemma was a simple discovery. He compares the dilemma to air, stating that it has always been around us and barely noticed but waiting to be really seen. The earliest forms dilemmas similar to the prisoner's dilemma occur in the bible, the writings of Seneca, Hillel, Aristotle, Plato, and Confucius, and maybe more commonly known Thomas Hobbes's *The Leviathan*. In this section we will learn about the applications that the prisoner's dilemma is often used for. These topics span far away from tradition game theory and have impacts in a wide range of research fields.

1.6.1 Biology

Game theory has many implications into biological studies because it gives explanations to things such as biological cooperation and competition. Before game theory it was often too difficult to describe these scenarios. One example of biological cooperation is a type of bird that enters a crocodiles mouth to eat parasites. Both benefit heavily from cooperating with each other. The birds are able to eat and the crocodiles no longer have parasites in their mouths. Something else could easily happen: the crocodiles could eat the birds that land in their mouth. This is an example of what defection would represent. These are the types of scenarios that game theory made easier to understand.

Another important biological scenario is the inheritance of genes for sickle cell anemia. Sickle cell anemia causes the body to not get enough oxygen because there is a lack of healthy red blood cells. If an offspring inherits the gene from both parents then unfortunately develops sickle cell anemia. If an offspring inherits only one of the genes, they develop a strong immunity to malaria. The idea of how deadly genes have survived in humans can be studied in a similar way to the prisoner's dilemma described in Table 1. If both parents carry the gene it would be similar to both prisoners cooperating and ending up with a longer jail sentence. Therefore, the gene lives on in an offspring when both parent pass it on. It seems a little far from game theory because there are no strategies involved but the players choice is not always the most important aspect to game theory. Once you have a payoff matrix, those numbers determine what will happen no matter how much a player would like to maximize their earnings. Thus, genes will always randomly reproduce and be passed on given the series of criterion of natural selection. A factor that can change the random reproduction of the genes are mutations. Mutations are responsible for changes in the genetic code of species. An example very similar to what we will be modeling in later sections, is a species of animals where by nature they are sharers. They split the food amongst others and ensure the entire population has food to eat. A mutation can occur that turns the sharers into gorgers. The gorgers do not save food for the future or even share any with their fellow population. Let's say all of the stored food prepared by the sharers get ruined. This means they will all go hungry, since that was their main source of food. A high percentage of gorgers stay alive since they barely have went hungry before this time, due to their high consumption of food. These will be the next to reproduce offspring, and the offspring will have the same gorging trait. Eventually we may hit a point where the sharers become completely extinct, even if there were a few left, it would not be enough to completely bring back the entire population. Now, with a population of only gorgers they no longer have the ability to take from the sharers stored food. Now they will be worse off then before when the sharers were still in the population, since they will have to find their own food supply. This shows that it may not be true that the survival of the fittest strategy held in this scenario. This allowed scientists to discover what is called a "evolutionarily stable strategy". This means that all members of a population are genetically transmitted a behavior that is upheld because there is no other behavior to change it. Natural selection follows the behavior that leads to the highest chance of survival. We can see that being a sharer is better off for a larger population, but thinking about it without preference, we see that the gorgers survive at the expense of the sharers. This is now a prisoner's dilemma. Even for humans, defection is an evolutionarily

stable strategy and cooperation is not.

Robert Axelrod was the first to create a series of tournaments of the iterated prisoner's dilemma and discussed the idea of artificial selection. It was the idea described above, have a random population of different behaviors and strategies, reproduce the top scoring, and run the simulation again. He found that in the first few rounds of the simulations, the weak strategies died out, and the stronger strategies thrived. When simulations continued, the stronger strategy population had no weaker agents to get points off of and slowly began to die out. The mixed strategies that both cooperate and defect with their opponents were the only strategy in the population to survive. [20]

1.6.2 War Time Strategies

Early on von Neumann created a mathematical model that showed that the allies would win World War II. World War II was the first time game theory was introduced into war time strategies. Merrill Flood was given the assignment to study a location to drop a bomb on Japan. This problem is extremely complex due to the fact that if the United States consistently choose an important location to drop the bomb, then Japan would have heightened watch on these important areas. There is some game theory involved in this situation. Flood was given an early manuscript from one of von Neumann's books which helped his create a strategy that minimized the chance of the bomber being shot down. This strategy and some advice from von Neumann was given off to wartime officials who devised a list of the potential targets.

At the time that the prisoner's dilemma was discovered, the United States and Soviet Union had just began the nuclear arms race. This can be posed as a dilemma in this sense: both nations have to decide whether to create an arsenal of hydrogen bombs. Each nation wants to build them to be the strongest of the two, but there is not much to gain if both successfully build the bomb since they will be equally as strong. Both nations also lose money which is the cost of building a hydrogen bomb. Let's say building a bomb compares to defection in the prisoner's dilemma and cooperation means they do not build it. In this situation we see that both nations would prefer to not be attacked and this preference allows for the dilemma to exist in the first place. [20]

1.6.3 Economics

Oskar Morgenstern was an economist that worked close with von Neumann to bring the applications of game theory into economics. Some conflicts that occur in the field of economics can be viewed as games. For example two companies fighting for a contract, buyers bidding at an auction, or competing companies trying to sell a product to a consumer. These games take into account much second guessing which leads them to be optimal for analysis. Their initial publication on the topic, *Theory of Games and Economic Behavior*, became influential years after it was published although many economists did not read it. It became more of a pioneering work of economics and presented the games as models for economic interactions. Trying to model economic problems as games posed many challenges for von Neumann and Morgenstern since these problems could be much bigger than just a two person game. Economic problems often contain an arbitrary number of players which brings up the topic of N-player games and the minimax theorem we will discuss in Section 2.4.1. In economic games the number of computations increased exponentially as the number of players increased. Which caused this work to not be used at this time. [20] Nowadays we see large companies advertising product to consumers to try and outsell their competition in order to raise profits. If neither company was to advertise to consumers, their profits would always stay about the same. This could be written as a two player game or an $N-$player game depending on how many companies would be featured in the study. This is only one of the many applications in economics, but von Neumann and Morgenstern noticed early how these two different fields would work together.

1.6.4 Law

There are two important games with connections to the field of law. These are the zero-sum game and the nonzero-sum game. The zero-sum game was particularly important to von Neumann and is a game in particular where in a two person match one player will lose what the other player gains. Something to note about the zero-sum game is that all players are acting in their own self interest and want the best

possible outcome for themselves only. Nash explored the nonzero-sum game. The nonzero-sum game is the most like the prisoner's dilemma in that the rewards and losses for the two players are not even. The prisoner's dilemma itself is already an application into law. It describes how a district attorney has to handle sentencing depending if criminals want to confess to a crime or defect and blame their accomplice. The prisoner's dilemma can also be expanded to a N-player game. In this scenario, the optimal strategies come from the choice of more than one player and can have different payoff matrices depending on the jail time required.
If we take another example such as the prosecution and defense deciding which witness to question in a trial. To model this as a zero-sum game we define two strategies and note that an outcome the benefits one player, hurts another. The two strategies are questioning the witness A or questioning witness B. The prosecution is the representative who wants to maximize their payoff in this game so that they have the most evidence against the accused person. The defense would want to minimize their payoff since they want the least evidence found against their client. We can create a payoff matrix based on the evidence in the trial and then study the matrix to understand what would be the favorable outcomes for the both the prosecution and defense. [8]
The zero-sum and nonzero-sum game can be applied to many scenarios in law to weigh the options between choices made. This is an important analysis that can be made before or during a case to understand chances of winning or losing depending on rewards and losses found within games setup with payoff matrices.

2 Mathematical Analysis

2.1 Mathematical Background

In later sections we will see the importance of computer simulation for the iterated prisoner's dilemma and mathematically intractable scenarios, but for now we will discuss the more important topics in mathematics that allow for the computer simulations to be possible. It is crucial to understand the background before simulating hypotheses in all areas of research.

2.2 Differential Equations

In order to predict the future we need mathematical models. Calculus tells us that change is measured by the derivative of a function. A differential equation is an equation that relates one or more functions and their derivatives. Using these derivatives to form differential equations is called mathematical modeling. Mathematical modeling does not aim to produce a perfect replication of the real scenario but a near accurate approximation. Most of these models simulate change over time and how populations can evolve. A popular example is the rabbit and foxes population model. The population of rabbits depends on how many foxes are in the area, and the same goes for the fox population.
Differential equations can sometimes be accompanied by an initial condition. Initial conditions are used to solve for particular functions to predict scenarios under those given conditions. This allows for functions to vary under changing initial conditions. All of these particular solutions are what describe the behavior of a general solution for a differential equation. These associated problems are called initial-value problems and take the form of $\frac{dy}{dt} = f(t, y)$, $y(t_0) = y_0$. [4]

2.2.1 Slope Fields

Slope fields are a qualitative technique used to analyze differential equations. The pattern of the short lines that are produced allows the viewer to see the curve of the solution to a given equation. [11] Having a visual can provide a qualitative analysis of a differential equation when it is difficult to obtain a analytical or numerical solution.
Suppose we have an equation $\frac{dy}{dt} = f(t, y)$ and a function $y(t)$ is a solution. Let the graph of the equation pass through the point (t_1, y_1) such that $y_1 = y(t_1)$. Thus, our differential equation states that the derivative at $t = t_1$ is $t_1 = f(t_1, y_1)$. The slope of the tangent line of $y(t)$ at the point (t_1, y_1) is $f(t_1, y_1)$. [4]

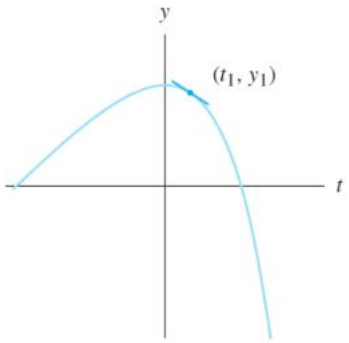

Figure 1.10
Slope of the tangent at the point (t_1, y_1) is given by the value of $f(t_1, y_1)$.

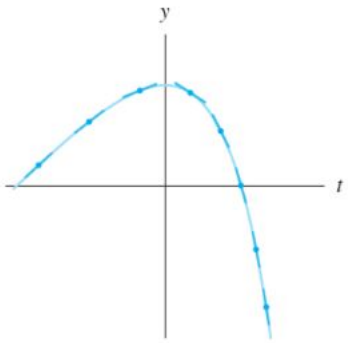

Figure 1.11
If $y = y(t)$ is a solution, then the slope of any tangent must equal $f(t, y)$.

Figure 1: Examples of how to graph tangent lines

These figures display the curve for only one solution curve to the differential equation and it's tangent lines. Now if we want to visualize multiple solutions to a first-order differential equation we can compute values of $\frac{dy}{dt} = f(t, y)$ at more points in the ty-plane. In Section 3.5.2, there is an example of how these slope fields can be generated using Python, as well as by hand.
To visualize an example of a completed slope field we can examine a simple differential equation, $\frac{dy}{dt} = 2t$. If we compute the integral of this equation we get, $y(t) = \int 2t\, dt = t^2 + c$. Therefore, the general solution to this differential equation is in the form $y(t) = t^2 + c$, where c is any constant.

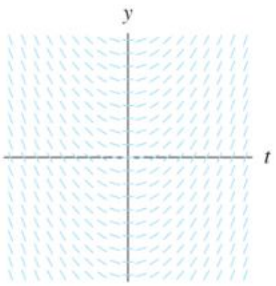

Figure 1.17
The slope field for

$$\frac{dy}{dt} = 2t.$$

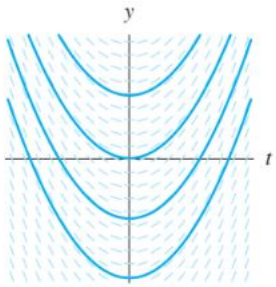

Figure 1.18
Graphs of solutions to

$$\frac{dy}{dt} = 2t$$

As we see in the diagram for the slope field of $\frac{dy}{dt} = 2t$, the curves resemble parabolas. There are infinitely many curves in the form $y(t) = t^2 + c$ as shown by the solution graph. All solutions to $\frac{dy}{dt} = 2t$ are parabolas with a vertical shift either upwards or downwards depending on the value of c. [4]

2.2.2 Euler's Method and Numerical Approximation

Numerical methods are used when we want to get qualitative data from differential equations that we cannot find exact solutions for. It would be beneficial to always have a exact solution, but having close approximations done by computers can be valuable to understanding the behavior of an equation. [4] The technique used to create slope fields is closely associated with a numerical approximation technique called

Euler's method. Mathematician Leonhard Euler (1707–1783), was the first to present the use of numerical methods to approximate solutions to differential equations. Euler's method is no longer used often in practice but it is an important method to understanding the more involved techniques more often used to solve differential equations.[6]
Suppose we have an initial-value problem, $\frac{dy}{dt} = f(t, y)$, $y(t_0) = y_0$. The idea behind Euler's method is to begin at a point, say (t_0, y_0), and calculate the tangent line at "steps" around this point. Every t value is one step apart and the step size is denoted as Δt. We can pick a value for Δt depending on how accurate we want our approximation to be and how many computations we are able to make.
The algorithm for Euler's Method goes as follows. We compute the derivative at the point (t_k, y_k), by plugging in our values for t and y into our equation. Next, we calculate the following point (t_{k+1}, y_{k+1}) by using two formulas.

$$t_{k+1} = t_k + \Delta t$$

$$y_{k+1} = y_k + f(t_k, y_k)\Delta t$$

We create line segments that start at (t_k, y_k) and goes through the point (t_{k+1}, y_{k+1}) with the slope of $f(t_k, y_k)$. [4] When we repeat this process over and over again we get a series of points, $(t_0, y_0), (t_1, y_2), (t_3, y_3)$, $\dots (t_k, y_k), (t_{k+1}, y_{k+1}), \dots$ that create line segments that approximate the solution curve.

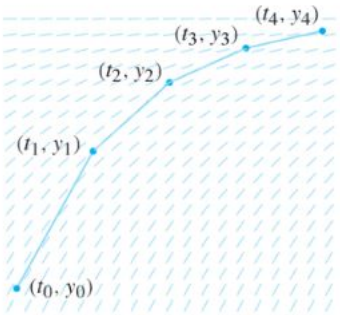

Figure 1.31
Stepping along the slope field.

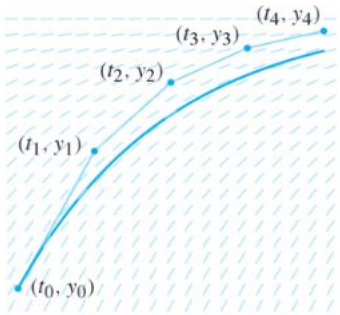

Figure 1.32
The graph of a solution and its approximation obtained using Euler's method.

In the two figures above we see the approximation of a solution curve using Euler's method against a slope field. Figure 1.31 shows the connected line segments forming a curve similar to ones on the slope field diagram. Figure 1.32 shows the actual solution and the Euler's method solution. The Euler's method solution curve is higher than the actual solution but it still provides us with a visual as to how the actual solution will act. If we chose a smaller value of Δt for this approximation using Euler's method we would have had more than five points like the figures above. More points yields a close approximation, but the number of computations to do this increases slowing down the efficiency of the machine you are using. There are now more advanced techniques than Euler's Method that take into account more than one slope.
Now that we see the connection between Euler's method and slope fields, it is also important to show where the method comes from. We wish to derive Euler's method for some initial value problem

$$\frac{dy}{dt} = f(t, y)$$

$$y(t_0) = y_0$$

We will add the extra condition that t is in a closed interval say, $[a, b]$. Assume we have evenly distributed points along $[a, b]$ such that for every $k = 0, 1, 2, 3, \dots, N$,

$$t_k = a + kh$$

Where N is the chosen number of divisions, a is the same value as in $[a, b]$ and h represents the distance between the points. This is defined as, $h = \frac{b-a}{N}$, or $h = t_{k+1} - t_k$, or also known as the step size.
An important theorem to this derivation is know as Taylor's Theorem. It was first described by Brook Taylor in 1715 but it is possible that others such as Issac Newton had discovered it prior. This theorem states that if a function f is continuously differentiable on some interval $[a, b]$, $f^{(n+1)}$ also exists on $[a, b]$, and $x_0 \in [a, b]$. Then for every $x \in [a, b]$ there exists a number call it $\xi(x)$ between x_0 and x such that,

$$f(x) = f(x_0) + f^{'}(x_0)(x - x_0) + \frac{f^{''}(x_0)}{2!}(x - x_0)^2 + ... + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)^{n+1}$$

We denote the first n terms as $P_n(x) = f(x_0) + f^{'}(x_0)(x - x_0) + \frac{f^{''}(x_0)}{2!}(x - x_0)^2 + ... + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$. This represents the nth Taylor Polynomial for f about x_0. The final term, $R_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)^{n+1}$ is the remainder term. The number $\xi(x)$ depends on the value of x at which $P_n(x)$ is being evaluated so that makes it a function of x. Thus, $\xi(x)$ should not be explicitly determined.
Now we can assume $Y(t)$ is a unique solution to the initial value problem above and has two continous derivatives in $[a, b]$ so that for every $k = 0, 1, 2, 3, \ldots, N$,

$$y(t_{k+1}) = y(t_k) + (t_{k+1} - t_k)y'(t_k) + \frac{(t_{k+1} - t_k)^2}{2}y''(\xi_k)$$

Since $h = t_{k+1} - t_i k$ we can rewrite this as,

$$y(t_{k+1}) = y(t_k) + hy'(t_k) + \frac{h^2}{2}y''(\xi_k)$$

Then because $y(t)$ satisfies the initial value problem ,

$$y(t_{k+1}) = y(t_k) + hf(t_k, y(t_k)) + \frac{h^2}{2}y''(\xi_k)$$

Euler's method finds a value approximately close to $y(t_k)$, call it w_k for every $k = 0, 1, 2, 3, \ldots, N$ by removing the final term $(R_n(x))$. Therefore we have shown Euler's method is represented by

$$y_{k+1} = y_k + f(t_k, y_k)\Delta t.$$

2.2.3 Normalizing Vectors

If we have a vector $\vec{v}$ and wish to change it's length to 1, then $\hat{v}$ is the normal vector. This vector $\hat{v}$ will have the same direction as $\vec{v}$, they will just differ in length. We will need to use the definition of the norm of a vector to find the normal vector. If we take a vector containing n elements such as, $\vec{v} = (v_1, v_2, v_3, \ldots, v_n)$ the norm is found by,

$$|\vec{v}| = \sqrt{v_1^2 + v_2^2 + v_3^2 + \ldots + v_n^2}$$

Now, to normalize $\vec{v}$,

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}$$

For a simple example, let $\vec{v} = (3, 4, 5)$. This means the norm of $\vec{v}$ is,

$$|\vec{v}| = \sqrt{3^2 + 4^2 + 5^2} = \sqrt{9 + 16 + 25} = \sqrt{50} = 5\sqrt{2}.$$

Using the formula for the normalization of a vector we can now compute $\hat{v}$.

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|} = \frac{(3, 4, 5)}{5\sqrt{2}} = \left(\frac{3}{5\sqrt{2}}, \frac{4}{5\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$$

Therefore, we have found that $\hat{v} = \left(\frac{3}{5\sqrt{2}}, \frac{4}{5\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$ which is the normalized vector of $\vec{v}$. This will become extremely useful later on in the discussion bout writing code for slope fields. In slope fields, we see vectors that are all of a normalized length in order to produce clear graphs of the solutions to a differential equation. Without normalizing the vectors the lines become cluttered and are drawn on top of each other. This is a tool that helps in data visualization and the analysis on graphs.

2.3 Agent Based Modeling

Agent based modeling is the way to model dynamical systems where agents that have different qualities are interacting in an environment. This kind of modeling emerged from cellular automaton which models a discrete dynamical system evolving on discrete-time and discrete space. The cells in these models are updated synchronously on a global time scale and some rule that influences cells to change based on its surrounding cells. [7] What distinguishes agent based models from cellular automaton, is that agent based models can update asynchronously depending on if they interact with other agents or their environment. These models also do not have a grid like structure and do not update on time steps but rather with event-cues. The reason why agent based models are used for simulations relating to biology and social sciences, is that the qualities of the agents being modeled and their environments, cannot be simplified to something like a cellular automaton.

Agent based models are important to describe the behavior of the agents as they respond to rules of their behavior and this can portray the behavior of instances such as insect colonies, immune responses, financial markets, neural computation, and even agents responses to rules similar to those such as the flocking behavior of birds. This distinguishes agent based models from classical mathematical models. These classical mathematical models often take form as systems of ordinary differential equations. These are inherently limited where agent based models have qualities programmed into them it is difficult if not impossible to assign a "quality" to a differential equation. If we look at a predator-prey model called the Lotka-Volterra system, it models the populations of rabbits and foxes with respect to each other. The population of rabbits grows at the rate that they reproduce and shrinks at the rate that the foxes hunt them, which depends on the number of foxes in the populations. The rate of foxes grows as the reproduce but with a constant food supply, and eventually die of old age. This model leaves out ideas of space and behavior. What were to happen if a fox gets closer to a rabbit? What happens if the rabbits and foxes don't cross paths? This is what makes agent based models so important. [3]

2.3.1 How to Create an Agent Based Model

There are key properties to remember when creating an agent based model. The first is that agents in this system have their own properties. In our case for the iterated prisoner's dilemma, our agents have the strategy they will use to play the game. The next is that each agent has a set way to act after an interaction with another agent. In our case, each agent gains a set number of points for their interaction depending on each agent's actions. The agent that uses the tit-for-tat strategy gains points but also switches it's strategy depending on its opposing agent's choice. Sometimes an agent may need some sort of memory to store other agent's actions during their interactions but for our project this was not necessary. There will need to be a specification on who each agent will interact with whether that means every single other agent in the system, only a selected group, or an undetermined number of other agents. For this project, each agent interacts with every agent, but new agents are introduced as the ones with a lower number of points are removed from the system. Finally, the system evolves over time. This project specifically relies on the number of rounds in a tournament of the game to determine how time passes. [3]

2.3.2 Genetic Algorithms

Genetic algorithms are based on the idea of providing a mathematical formula to how fast some genes would spread amongst a population. R.A. Fisher was the first to create this formulation by specifying a type of gene, viewed evolution by generations, and a mutation factor that modify offspring genes to create different generations. Natural selection is an important factor in Fisher's model. In this model natural selection is used to determine whether each individual is becoming of a higher fitness to carry the gene. It is also important to note that there can be many variations in mutations within a single species.

Genetic algorithms are used for problems that are not solvable by standard techniques such as protein folding, machine learning, and modeling complex adaptive systems, such as markets and ecosystems. Using a genetic algorithm requires there to be a search space represented by a set of strings. These strings can represent rules or behaviors, agents in a system, or organisms themselves. The genetic algorithm can be instantiated with predetermined strings or can simply be randomized. It then processes the strings and every generation that appears uncovers the form of the string that acts at a high fitness in the system. The finally, the algorithm

takes these superior strings and finds the improvements that were made to them to determine what caused a better fitness in the system. [10] Genetic algorithms pave the way for the discovery of what causes natural selection and what key factors evolve over time to allow agents in a system to thrive.

2.4 Game Theory

Game theory's foundation is looking at conflict by creating a theoretical framework to describe the actions between two actors. Game theory is often explained by conflicts of interest. A very simple example is two parents attempting to split a piece of cake evenly between two children. This will cause a conflict between the two because no matter how evenly the parents cut the cake, each child will believe the other got the larger piece. Both children are acting in their self interest because everyone wants more cake. The parents come up with a clever solution to let one child divide the cake and let the other chose what slice they want. This way neither child can complain because one divided the cake evenly and the other chose which piece they deemed larger. This applies to game theory because this is a situation where there can be a solution created solely based on each child's self interest in mind. [20]

2.4.1 Nash Equilibria and the Minimax Theorem

John von Neumann wanted an answer specifically to the question of whether there is a rational way to play every game. He ultimately decided to prove it mathematically and was able to show that there will always be a rational way a game is played by two players if their intentions are completely opposed. This became what is now known as the minimax theorem. This theorem applies to more than just the prisoner's dilemma. Its implications can be found in games such as ticktacktoe and chess, or any game where the players have opposing interests and there must be a winner and a loser. This also meant that now von Neumann could find a way to expand this to games where the interests are not entirely opposed and eventually to the fact that game theory could help understand any form of human conflict. [20] Nash equilibria represent the strategies in an N-player game where each player would have the best payoff against the $N-1$ other players of the game. This has to hold even if the other players announced what strategies they would choose. The Nash equilibria also can be considered stable points, since prior to the discovery of them, players would adjust their strategies to ensure a higher payoff their their opponents. [21]

2.4.2 Payoff Matrix

In game theory, a payoff matrix is a matrix that displays the strategies of one player in the rows and the other in the columns. Each cell determines the payoff that a player will receive for choosing that strategy versus their competitor. We will describe multiple examples and then explain the general formulation of a payoff matrix.

Specific Games

1. Prisoner's Dilemma
   The prisoner's dilemma is discussed in more detail in Section 1.1, but an important idea that occurs in games with similar payoff matrices to the prisoner's dilemma is **deadlock**. The payoff matrix for deadlock can be defined as follows.

	Cooperate	Defect
Cooperate	1,1	0,3
Defect	3,0	**2,2**

Table 2: Example of a payoff matrix for deadlock

With a quick analysis of this payoff matrix we see that defection always results in the highest payoff, either two for mutual defection or three for the only player to defect. In this scenario we want the highest payoff possible, whereas for the prisoners dilemma each prisoner wants the lowest payoff for the shortest jail sentence. Each player will defect in the hopes their opponent will cooperate but does not get too hurt from a mutual defection, only one point lower than hoped. Thus, there is no real reason to switch your strategy from defection to cooperation. When two players do not cooperate simply because they are hoping the other will causes deadlock. [20]

2. Chicken

The game of chicken can be pictured on a highway with two drivers driving heads on towards each other. Both drivers have an identical reaction time and turning radius. There is a moment where the drivers each have to decide whether or not they will swerve. Again this decision is made with out the influence of the other driver and each driver only gets one choice. The payoff matrix for a game of chicken is as follows.

	Swerve	Drive Straight
Swerve	2,2	**1,3**
Drive Straight	**3,1**	0,0

Table 3: Example of a payoff matrix for the game of chicken

The way the two drivers think about the outcomes is as we would expect. The worst possible outcome would be if both drivers do not swerve waiting for the other to do so. The theory behind the game is to not become a "chicken" by not swerving and having your competing driver swerve which guarantees 3 points. Both drivers get 2 points if they swerve and neither can be called a chicken. Chicken is different from the prisoner's dilemma because the most feared outcome is both drivers not swerving thus leading to both swerving being the best option. Whereas for the prisoners dilemma, the worst outcome is to cooperate while the other prisoner defects so defection is the best option. In a game of chicken, both drivers want to do the opposite of the what the other does. Another way to view a game of chicken is the volunteer's dilemma. In a game of chicken, when someone "volunteers" to swerve it is for the common good. If none of the drivers volunteer, both end up in car crash and if both volunteer each wishes they had driven straight. In the volunteer dilemma if no one volunteers it hurts the common good, but if someone does volunteer they benefit everyone with their decision. [20]

3. Stag Hunt and the Trust Dilemma

Before discussing the game of stag hunt we will start with a story of two students and their last day of school. A group of friends decide it will be a fun prank to come to school on the last day with a ridiculous haircut. Both students agree to get the haircut but as the night goes on each student begins to doubt whether or not the other will go through with the haircut. In this scenario the best possible outcome is if both student receive the haircut and pull a funny prank. Another good option is if both students do not get the haircut and everyone just simply forgets about their prank idea. If one student gets the haircut and the other does not, the former will be subject to ridicule by their friends and the latter looks like a jerk for not going through with it. What makes stag hunt a dilemma is the worry that one student will back out and if they do back out, the second student will want to back out too. [20] Stag hunt is also known as a "trust dilemma" which means that both students know they will obtain the best possible outcome if they both get the haircut but there is the possibility that they will be betrayed by the other. Therefore, this leads to players in a trust dilemma to decide how much trust they place in their opponent and this often stems from social uncertainty. This can mean many things such as uncertainty in intentions and actions or what the other player's intentions are. (trust dilemma page) Trust is needed in situations similar to the stag hunt so that mutual cooperation can occur and lead to the best payoff.

Jean-Jacques Rousseau's *A Discourse on Inequality* discusses the idea of early human societies having

hunting alliances. It takes two hunters to catch a stag but only one to hunt a hare. Therefore, mutual cooperation is need so that the hunting alliance walks away with the best option which is a stag.

	Hunt Stag	Chase Hare
Hunt Stag	**3,3**	0,2
Chase Hare	2,0	1,1

Table 4: Example of a payoff matrix for the stag hunt dilemma

Even though mutual cooperation is the best payoff there are still chances of defection due to the trust dilemma and the nature of humans to fall to social uncertainty.

General formulation

After talking about the prisoner's dilemma, chicken and stag hunt, we have seen multiple examples of what payoff matrices may look like. Now we will discuss a general case or a general form of what a payoff matrix is in regards to game theory. A payoff matrix is a table where the strategies of two players are displayed in rows and the columns respectively. The cells represent the payoff each player receives if the strategy is chosen. The first value represents the value for the player represented by the row. The payoff is the point value that a player gains or loses by executing a specific strategy. Payoff values and meanings vary depending on the game. Payoff matrices are crucial for game theory because a game can be summarized by its important information and allows for the quick analysis of dominant strategies and Nash equilibria. Below we can set up an default payoff matrix to understand how we could create one.

	Player 2 Strategy 1	Player 2 Strategy 2
Player 1 Strategy 1	$P1, P2$	$P1, P2$
Player 1 Strategy 2	$P1, P2$	$P1, P2$

Table 5: Example of an empty payoff matrix

In this example, Player 2 is represented by the columns and Player 1 is represented by the rows. Their payoffs are labeled as $P1$ and $P2$. Payoff matrices can represent zero-sum games as well as nonzero-sum games which allows for many scenarios to be summarized in this way. In the following sections we will see how to calculate the expected payoff values for different N-player games. [22]

2.5 The Two Player Game

2.5.1 Expected Payoff Values in an N-round Game of the Prisoner's Dilemma

We will now see how to determine the expected payoff values in a given iterated two player prisoner's dilemma game.

Proof:

We are given an iterated prisoners dilemma simulation with $N = 10$ rounds. If $P = 8$ of the population uses the tit-for-tat strategy, and $100 - P = 92$ plays the all defection strategy, then the tournament score of the tit-for-tat population will be greater than the tournament score of the defection population.

First let us examine the payoff matrix for this situation.

The tit-for-tat strategy is not in our payoff matrix. The reason for this is if we recall what the tit-for-tat strategy does is at first cooperates with it's opponent and only defects if its opponent does in the prior

	Cooperate	Defect
Cooperate	3,3	0,5
Defect	5,0	1,1

Table 6: Example of a payoff matrix for iterated prisoner's dilemma

round. Therefore the only payoff values needed are cooperate and defection. We can compute the expected values after $N = 10$ rounds. When an agent from the tit-for-tat population plays a match against another agent of the same strategy they will have

$$3 * N = 3 * 10 = 30$$

This is because two tit-for-tat players will always cooperate with each other. Now if a player with the tit-for-tat strategy plays against a defection player the expected values will be

$$(0 * 1) + 1 * (N - 1) = N - 1 = 10 - 1 = 9$$

This time the player receives 0 points for the initial cooperation but then gains points again for the mutual defection. Now if two players with the defection strategy play against each other the expected value will be

$$1 * N = 1 * 10 = 10.$$

Finally if an agent playing the defection strategy plays an agent using the tit-for-tat strategy we will have

$$(5 * 1) + (1 * (N - 1)) = 5 + 9 = 14.$$

Now we can compute the overall score after the 10 round tournament for the tit-for-tat agents.

$$\text{TFTTournamentScore} = (P - 1) * 30 + (100 - P) * 9$$
$$30P - 30 + 900 - 9P$$
$$21P + 870$$

We first used the score the tit-for-tat players if they are playing against each other first so $P - 1$ represents the unique tit-for-tat players they can pay against. Then the $100 - P$ represents all of the defection players they can encounter. Each of these values is represented by the total number of points they can receive for these matches. The tournament score for the defect players can be computed similarly.

$$\text{AllDTournamentScore} = P * 14 + (99 - P) * 10$$
$$14P + 990 - 10P$$
$$990 + 4P$$

They will encounter all P number of tit-for-tat players and $99 - P$ unique defection players. Now if we set these two expressions equal to each other we will see that we can solve for P.

$$21P + 870 = 990 + 4P$$
$$17P = 120$$
$$P = 7.05$$

We get that p must be greater than 7.05 so that the overall tournament score of the tit-for-tat players will overcome the tournament score for the defection players.

QED.

To find the general formula for this we can leave our formula in terms of N. Using the scores we have previously derived we can examine,

$$\text{TFTTournamentScore} = (P - 1) * (3N) + (100 - P) * (N - 1)$$

$$3NP - 3N + 100N - 100 - PN + P$$
$$2NP + 97N - 100 + P$$

$$\text{allDTournamentScore} = P * (5 + (N - 1)) + (99 - P) * (N)$$
$$5P + PN - P + 99N - PN$$
$$4P + 99N$$

We can set the derived equations equal to each other and solve for N,

$$2NP + 97N - 100 + P = 4P + 99N$$

$$2NP - 100 = 3P + 2N$$
$$2NP - 2N = 3P + 100$$
$$N = \frac{3P + 100}{2P - 2}$$

Thus, we have found a general solution for N.

2.6 The Three Player Game

After talking about the two player game, the next natural step would be the three player game. The table below describes the possible expected scores the agents can get depending on their opponent for 10 rounds.

	All-C	All-D	T-F-T
All-C	30	0	30
All-D	50	10	14
TFT	30	9	30

This can also be written in a matrix form which creates a system of equations we can solve to find the values of each populations. The values in the vector c, d, t, represent the All-C, All-D, and TFT populations respectively.

$$\begin{bmatrix} 30 & 0 & 30 \\ 50 & 10 & 14 \\ 30 & 9 & 30 \end{bmatrix} \begin{bmatrix} c \\ d \\ t \end{bmatrix} = \begin{bmatrix} 30c & 0 & 30t \\ 50c & 10d & 14t \\ 30c & 9d & 30t \end{bmatrix}$$

Using the equations for the tit-for-tat and defection populations, we can calculate the values needed for the population of tit-for-tat to exceed the other populations.

$$30c + 9d + 30t > 50c + 10d + 14t$$

Let $t = 1 - c - d$

$$30c + 9d + 30(1 - c - d) = 50c + 10d + 14(1 - c - d)$$
$$30c + 9d + 30 - 30c - 30d = 50c + 10d + 14 - 14c - 14d$$
$$30 - 21d = 36c + 14 - 4d$$
$$16 > 36c + 17d$$

We will use this equation later on to verify that the values we have chosen to represent the cooperation and defection populations will allow the tit-for-tat population to grow and take over.
Now if we want to investigate the tit-for-tat population in terms of the cooperation population we can examine the same equations with $d = 1 - t - c$,

$$30c + 9d + 30t = 50c + 10d + 14t$$

$$30c + 9(1 - t - c) + 30t = 50c + 10(1 - t - c) + 14t$$

$$30c + 9 - 9t - 9c + 30t = 50c + 10 - 10t - 10c + 14t$$
$$21c + 9 + 21t = 40c + 10 + 4t$$
$$-19c + 17t = 1$$
$$17t = 1 + 19c$$
$$t = \frac{1}{17} + \frac{19}{17}c$$

Now we have an equation for the tit-for-tat population in terms of the cooperation population. In order to get the tit-for-tat population alone, it is necessary to subtract the cooperation population value. This t value is the minimum value needed for the tit-for tat population to overtake all other populations in the simulation. Then we simply find d as stated before using $d = 1 - t - c$. Another thing to note would be, since this equation finds the minimum value for what t has to be, any value greater than what is found for t works as well.

2.6.1 Example of the Three Player Game Population

Let the cooperation population be 60% or $\frac{3}{5}$ of the population. This means that,

$$t = \frac{1}{17} + \frac{19}{17}\left(\frac{3}{5}\right) = \frac{62}{85}$$

This is roughly 73% of the population. Now this number seems too large. If we have the cooperation population as 60% and the tit-for-tat population at 73% we have already exceeded 100%. AN even more worrying result is that if you try to compute the value of d using $d = 1 - t - c$ we get $d = 1 - .73 - .6 = -.33$ which is not possible. To fix this, if we subtract the cooperation population from the tit-for-tat population we get $t = .73 - .6 = .13$ or 13% of the population which sounds a lot better. We can recompute d where $d = 1 - .13 - .6 = .27$ or 27%. Here we can verify these add up to 1, $c + d + t = .6 + .27 + .13 = 1$. Now we can use the first equation we found to see if after 10 rounds the tit-for-tat population will exceed the other populations.

$$16 > 36c + 17d = 36(.6) + 17(.27) = 26.19$$

Therefore, after 10 rounds the tit-for-tat population will not be able to exceed the other two populations. We will verify this result by a simulation in Section 4.2.

2.7 Evolution Results and Long Term Population Results

In the previous section, we saw an example of different population values for cooperation and defection that did not allow for the tit-for-tat population to grow and exceed both others. Since this was shown mathematically for 10 rounds, we are not able to see the changing populations over multiple rounds with evolution in place. In order to see if there would be a population change amongst the agents we would need to implement some sort of software that would handle the running of tournaments, where the lowest scoring agents would be removed. With this in place we would be able to see if long term the tit-for-tat population would exceed the cooperation and defection populations. In Sections 3.6 and Section 4 we will be able to show computationally that with these population values, initially the tit-for-tat population does not grow but over multiple tournaments it will exceed the other two populations eventually to a point where they will be completely removed from the system.

3 Software

3.1 Agent Based Modeling Software

Throughout our research, we were able to find already created software for agent based modeling. Most of these were created around the late 1900s and early 2000s which was the time that the topic became of interest to various fields other than mathematics. Many have been updated recently in 2019, others have

not been updated since shortly after it's creation, and others no longer have a known date of last update. The following software is the most recently updated and guided us to decided what software we would be using for this project.

3.2 Java Based Software

3.2.1 NetLogo

NetLogo is a multi-agent modeling environment that can be used to describe programming scenarios for agents to student from elementary school to graduate school, teachers, and researchers. Uri Wilensky was the programmer who created NetLogo in 1999 and he is the director of Northwestern University's Center for Connected Learning and Computer-Based Modeling (CCL). NetLogo was written in Scala and some Java. Scala is a object-oriented and a functional programming language mixed into one and it relies on static types to avoid errors in more complex code. This is the most recently updated agent-based modeling software being updated as recently as September of 2019. [14]

3.2.2 AScape

Ascape is less recently updated software, August 2010, for studying agent-based models. It is advertised to create models with less code than other software packages and make drastic changes in models with minimal code required. It is also supposed to have multiple kinds of modeling and visualization tools. Unlike NetLogo, Ascape is marketed towards non-programmers so that they can create a complex model design without much code to be written. It was also developed completely in Java and can run on any Java-enabled platform. [2]

3.2.3 MASON

MASON is a multiagent simulation library core designed in Java. It was designed in mind to become the basis for simulations in Java whether it is large or small scale. MASON is equipped with both 2D and 3D visualization. It was created by George Mason University's Evolutionary Computation Laboratory and the GMU Center for Social Complexity. The creators state that "MASON Stands for Multi-Agent Simulator Of Neighborhoods... or Networks... or something..." A benefit about MASON is that the models are completely independent from their visualization which allows for models to be modified at any time. It can also generate videos of simulations, charts and graphs, and output data streams. [19]

3.2.4 AnyLogic

AnyLogic is one of the more recently updated software packages in August of 2019. It focuses on the use of modeling and simulations to solve real-world problems by providing an important aspect of analysis across multiple disciplines. AnyLogic takes pride in their simulation models because unlike physical models, simulation modeling allows for a dynamic environment in 2D or 3D. Since agent-based modeling focuses on the active components of the system, agents can be defined as people, households, vehicles, equipment, products, or companies. Then connections between selected agents are defined,an environment is created, and a simulation can be run. The results occur from the interactions between agents or between agents and the environment. AnyLogic is able to combine system dynamics and agent based modeling in one software package to allow for their customers to achieve accurate and clear results. Their website is the most descriptive, including tutorials, documentation, training and events, and even conferences. [1]

3.2.5 GAMA Platform

GAMA (GIS Agent-based Modeling Architecture) is another software platform that models and develops agent-based simulations. Using the GAMA Platform a user designs their models by writing code in GAML which is a intuitive agent-based language. GAML is a Java based language. The GAMA Platform can handle large simulations up to millions of agents and can display 2D and 3D visualizations. According to the GAMA Platform's GitHub is was created in a general approach to be used for multiple application domains which include, transport, urban planning, epidemiology, and the environment. The GAMA platform was

developed specifically with scientists in mind who do not have a formal computer science background or none at all. A user just has to declare a species, assign a certain behavior, and add them to an environment. [9]

3.2.6 SARL

SARL is a programming language that is statically typed and agent oriented. It is based off of Java but improves on multiple aspects to create a clearer interface for agent-based modeling. SARL includes it's own statements specifically for agent programming, allows for lambda expressions, operator overloading, and powerful switch statements. Despite SARL being more readable and expressive than Java, it does not cause any issues between itself and Java. So code written in SARL works as expected in Java. SARL uses an Eclipse-based IDE that also works with the Eclipse Java Development Tools (JDT) so that all of the features of Java are still present. Something unique about SARL is even though it is related to Java, it can generate code in other languages such as Python. [17]

3.2.7 JABM- Java Agent-Based Modeling toolkit

JABM stands for Java Agent-Based Modeling and it is a toolkit for building agent-based models. It uses a "discrete-event simulation framework" and was originally created to aid in agent-based finance and economics research. In JABM, the agents are created an represented as Java objects and random attributes can be assigned with out writing Java code. [12]

3.3 Python Libraries

3.3.1 Mesa

Mesa is an agent-based modeling structure in Python. It's goal is to be the Python version of the previously discussed software packages in Section 3.2. Mesa comes prepared with components such as spatial grids and agent schedulers that allow it's users to quickly create agent-based models. It also has tools to visualize models with a browser-based interface and uses Python's data analysis tools to analyze results. [13]

3.3.2 NL4Py

Since there are not many options to simulate and create agent-based models in Python, the goal for NL4Py is to execute NetLogo models in a rapid and parallel way. It simply acts as controller software for NetLogo and gives the option to work with or without a graphical user interface NetLogo work-space control through Python. It was created when the demand surged for open-source computation and machine learning libraries in Python. NetLogo, as discussed in Section 3.2.1, is a popular and recently updated agent-based modeling package. Integrating NetLogo in with Python allows researchers to use statistical packages to analyze the output from models created. [15]

3.3.3 Why Python?

Python is an extremely powerful language that calculates computations in a much shorter time than others but this is not the main reason we chose to use it. As previously shown in Section 3.2 there are seven Java based agent base modeling packages spoken about but only two python libraries talked about in Section 3.3. The seven Java packages highlighted were a small subset from a larger collection that can be found available to use. This left the lack of similar software for other languages, especially new ones in recent years. Python has many libraries that expand it's functions from computations to now a data visualization tool. Even though agent based models rely heavily on object orientation as a method of creation, since Python allows for this technique and for powerful plotting tools it became an effective language to study agent based models.

3.4 Computational Mathematics

Computational mathematics is a multidisciplinary field that focuses on computation as a major component in research in mathematics, the sciences, and engineering. This is often achieved through algorithms and

numerical methods as well as using problem-solving techniques and methodologies to produce them. Computation aids in what was a typical theory and experimentation approach to advance scientific knowledge and experiments.
It is a similar field to applied mathematics where mathematical methods are applied to different research areas and expands to a different genre of topics such as partial and ordinary differential equations, linear algebra, operations research, discrete mathematics, and probability. Applied mathematics often creates mathematical models that are often used in industry to solve real-world problems. [16]

3.5 Graphing functions

3.5.1 Slope fields

There are many software packages across all programming languages that allow for the creation of slope fields. In order to graph these specifically in Python, we created functions to represent the functions of the derivatives in a differential equation. This made it easier to compute the slopes at a particular point. Next, we replicated Euler's Method in the code which computed the x and y values over a given number of steps. Finally, in order to plot the slope fields with a visible range of tangent lines, each vector had to be normalized before being plotted on the $xy-$axis. This code will be explained in more detail over the next few sections.

3.5.2 A Basic Example

To really understand this concept let us go through a simple example to see how a slope field is created. Let us examine,

$$\begin{pmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}$$

We can create a table to better understand what values are going to be graphed.

(x, y)	$\left(\frac{dx}{dt}, \frac{dy}{dt}\right)$
$(1, 1)$	$(-1, 1)$
$(0, 1)$	$(-1, 0)$
$(1, 0)$	$(0, 1)$
$(0, 0)$	$(0, 0)$
$(-1, 0)$	$(0, -1)$
$(0, -1)$	$(1, 0)$
$(-1, 1)$	$(-1, -1)$
$(1, -1)$	$(1, 1)$
$(-1, -1)$	$(1, -1)$

A rough sketch of this look like, At each point (x, y), we compute the values based on our equation and

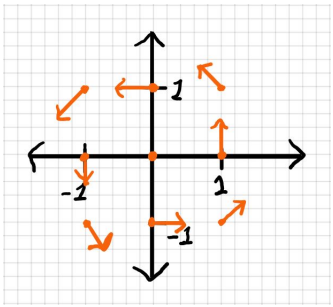

Figure 2: Rough sketch of differential equation slope field

then plot the slopes at each of these points. We see that at $(0, 0)$ there is no change which is portrayed as

just a point. At the point $(1, 1)$ we see that $\left(\frac{dx}{dt}, \frac{dy}{dt}\right) = (-1, 1)$. This means that the vector drawn will head towards the negative x direction and the positive y direction. Collectively the points create a almost circular shape, rotating in a counterclockwise direction. If we want to visualize the slopes on more (x, y) points, we can create Python code to do just that.

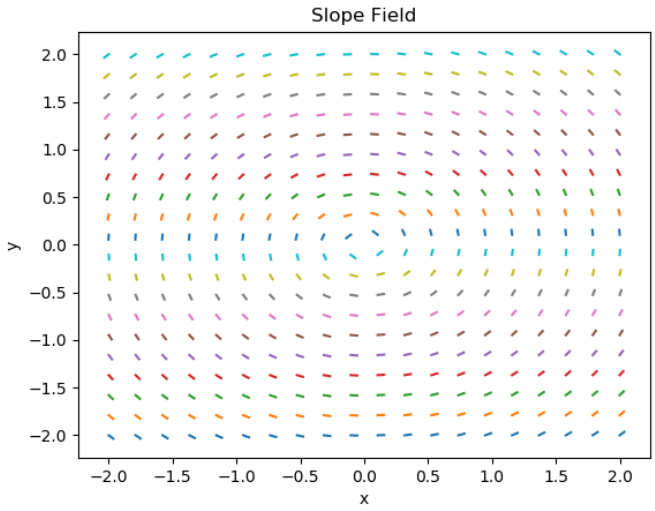

Figure 3: Python generated slope field of differential equation

In Figure 3, we see the same circular effect as our hand drawn slope field. The benefit of visualizing slope fields in Python is the accuracy. In the Figure 2, there is an approximation of slopes at points drawn in a plane, whereas here we see a more accurate picture. The Python graph also allows for many more vectors between points which can be changed depending on preference. In Figure 3, we chose 20 divisions on the x and y axis. Figure 3 does not show us the direction the vectors are pointing. We can make modifications so that we can get more information about our slope field.

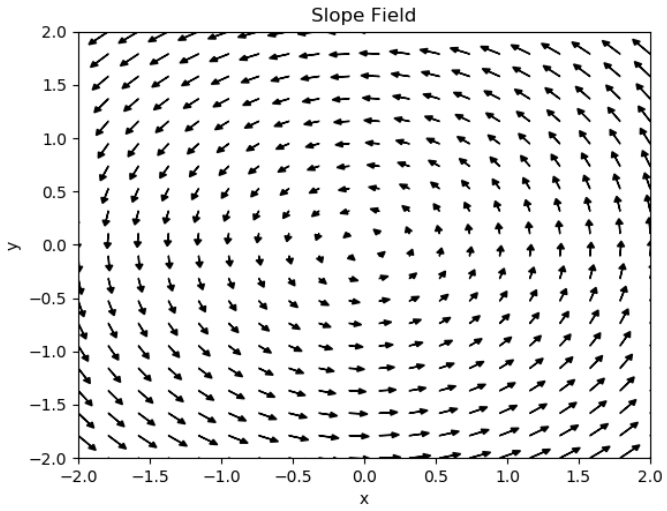

Figure 4: Python generated slope field of differential equation with direction

In the figure above, with a simple modification, we now have better qualitative data. We see the counter-clockwise movement of the vectors in the slope field.

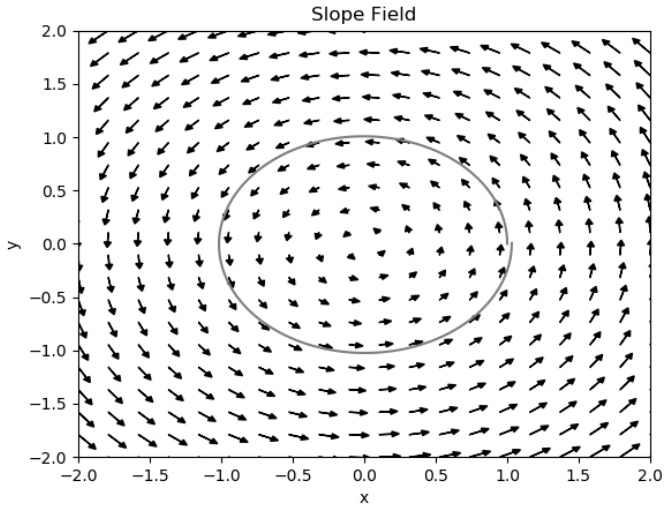

Figure 5: Python generated slope field of differential equation with a solution curve with initial condition $(x_0, y_0) = (1, 0)$

Finally, we can draw a curve to represent a solution given an initial condition. In Figure 5, there is a noticeable gap in the beginning and end of the solution curve. As the step size is decreased and the number of iterations is increase in Euler's method, this gets closer and closer to becoming a closed curve. This was done using Euler's method which will be discussed in the following section.

3.5.3 Euler's Method

Euler's Method is one of the known techniques to solve differential equations more specifically initial value problems. Although it is not used in practice. We discussed the algorithm from a mathematical standpoint in Section 2.2.2, the algorithm is implemented in code goes as follows.

```
# function for Euler's Method
# dx/dt = f(x,y)
# dy/dt = g(x,y)
# starting point on graph: (x0, y0)
# returns [ (x0,y0), (x1,y1), ... ]
#         [ (x0,x1,...), (y0,y1,...) ]
def eulersMethod(f, g, x0, y0, deltaT, numSteps):
    xValues = [x0]
    yValues = [y0]
    xCurrent = x0
    yCurrent = y0
    for i in range (0,numSteps):
        deltaX = f(xCurrent,yCurrent) * deltaT
        deltaY = g(xCurrent,yCurrent) * deltaT
        xNext = xCurrent + deltaX
        yNext = yCurrent + deltaY
        xValues.append( xCurrent )
        yValues.append( yCurrent )
        xCurrent = xNext
```

```
        yCurrent = yNext
    return [ xValues, yValues ]
```

The parameters for this function takes in a function f, a function g, a starting point (x_0, y_0), a value of Δt, and the number of steps to be taken. We begin by initializing a list of x values and y values with the initial point. Next we set a current point to be our current point to work with. Initially it is (x_0, y_0). Next, there is a loop that computes new points until the number of desired steps is reached. Here, we compute the next point with respect the function representing the derivative of it. Once the values for x and y are computed they are added to the list of x and y values and they become the new current values. The higher the number of steps allows for the algorithm to generate a solution closest to the analytic one if there is one. For this project specifically our slope field generator relied heavily on this function of Euler's method to compute the values of a solution curve as we saw in Figure 5.

3.6 Agent-Based Modeling

We spoke earlier about agent-based modeling in Section 2.3 in more detail. Here we will briefly review what these models are used for, their importance, and some applications. As well as discuss all the code we created for our agent-based models. The code can be found in the appendix or at
`github.com/jennmald/Iterative-Prisoner-s-Dilemma`

3.6.1 Purpose of Agent-Based Modeling

Agent-based modeling has many agents acting in a predetermined manner to their environment or other agents in the same environment. For our focus in the iterative prisoner's dilemma, agents need a strategy assignment to them. Every interaction they face with another agent is based on their game time strategy. Once the agent plays their move, whether they cooperate or defect, they score a number of points. These repetitive and competitive interactions between the agents outputs information about the system as a whole and it's behavior. This information is often manifested as patterns of behavior among the agents and in some real world simulations gives us an idea of how this situation would occur in real life. There are even studies where agent-based models have agents that evolve which can cause unexpected behaviors to occur this eventually can involve neural networks and evolutionary algorithms. For our project learning algorithms are not involved.[5] Agent based models have the ability to capture emergent phenomena, describe the system naturally, and is extremely flexible to changes to agents or the environment.

3.6.2 The Agent class

To begin the process of creating our own agent-based model for the iterated prisoner's dilemma, we created a class called Agent. The agent class has a constructor to initialize and agent in our system. This means that we are encoding the rules that each agent will follow and a name just for clarity. Since we are working in an iterated sense we also create a list to store the choices made each round of the game, a variable for the score at the end of each game, and then a variable for the tournament score for later on when we run multiple rounds of the game. An example of this is `allC = Agent(1,1,1, "All-C")`. The first 1 represents the probability of choosing to cooperate on the first round of the game. The second and third value represent the probability of choosing to cooperate and defect respectively. The reason here that the probability of choosing defection is 1 is because the choice method checks for the probability of cooperation first. If not it defects so that agent will always cooperate since that probability does not change and will only defect if the probability would change to zero. The method that determines chooses cooperation or defection uses a simple if statement to do this.
Another useful tool in this class is a clone function. This method can be called on an agent object which is then cloned and returned. This means something like `newAgent = allC.clone()`, would return a new agent with the same properties as the `allC` agent the method is called on. The agent class also comes with methods to reset the game score and tournament score if necessary.
In order to keep everything as organized as possible it was necessary to create a class for Agent utilities. These are helper methods that involve operations on agents that did not necessarily fit in the agent class and are used in other classes. It is important to note that the two methods in this class are static for this

reason. The first method in this class takes in a dictionary with a type of agent and a number to represent that quantity of that agent needed in the system. The method then creates a list of newly created agents with the same properties specified. The second method counts the number of agents of the same type and returns the count as a dictionary.

3.6.3 Playing the Prisoner's Dilemma (Game Class)

In order to begin the iterated simulation of the prisoner's dilemma a game class was necessary. This class is responsible for playing the game a certain number of rounds, comparing the predetermined choices made by each agent, and keeping score. The evaluate method takes in the choices from each player, compare them, and then returns the number of points for the agent the method is called on. If there is some choice that is not cooperate or defect the method will return an error message. The play method controls what agents are playing in the game and the number of rounds they will play against each other. It then utilizes a for loop to play each round, tally the score, and save the choice to game memory.

3.6.4 Iterated Prisoner's Dilemma (Evolution Class)

The evolution class is necessary for running a series of tournaments depending on the agent population size. The constructor of this method takes in a number of agents that will cooperate, defect, or use the tit-for-tat strategy. These become the initial population numbers for those agents. The initial count of number of rounds is set to zero in the constructor as well. The run method is responsible for playing a certain number of tournaments given a list with the selected agents. This method, similarly to the play method in the game class, uses a for loops to control the number of tournaments. After each tournament, this method counts the number of agents we still have after this simulation. It then picks the agents with the highest scores, removes half of the population with the lowest scores and clones the ones with the highest scores to be put back into the population. This is important data to the evolution process. We are now beginning to see how the populations are changing after a game is played. Only the agents with the top scores will make it back into the population, leaving out the ones who did not have the top scoring strategies. At the end of this method the agents are summarized in a list and returned in order to do data analysis.

3.6.5 Tournaments

As mentioned in the previous section, there is a way to run tournaments given a list of players. Every tournament is responsible for running a series of games depending on the population size. It is also responsible for updating the tournament scores in the agent class. The run method in the tournament class first does some type checking. This means that it determines if the players are in a list of dictionaries of agents. If it is the dictionary is then converted into just a list of agents for easier game play and conversion later on. Just for one more step of safety, the method checks whether or not the first element of the player list is an agent and throws an exception is this is not the case. The list of players is then shuffled, their tournament scores are all cleared, and then using two for loops the list is iterated through twice to ensure every player plays each other. Here the play method from the game class is utilized and each agent plays each other 10 times. Then the tournament score is incremented by the total game score after each game is played. At the end of the tournament, the list of agents is converted back into a dictionary of the agent type and count and is returned as output.

3.6.6 Data Visualization

To make this project more user-friendly an important component we added was a class for a graphical user interface. This class is in a separate file to distinguish the difference between the computational model and the visualization of it. For this the Python library tkinter was used in conjunction with matplotlib. The first element of this class that needed to be set up was the canvas. Once that was drawn to a desired size, an empty plot was placed to later be filled in with lines of the growing and shrinking agent populations. The x-axis represents the number of rounds that have passed and the y-axis represents the number of agents in the population. The axes can change depending on the user input for their choice in total number of agents and number of rounds they wish to play. In the graph there is also a legend to distinguish each line as a

different agent population. Underneath the graph, are a series of text boxes where the values for each agent population should be entered. The final text box is for the number of rounds the simulation will complete.

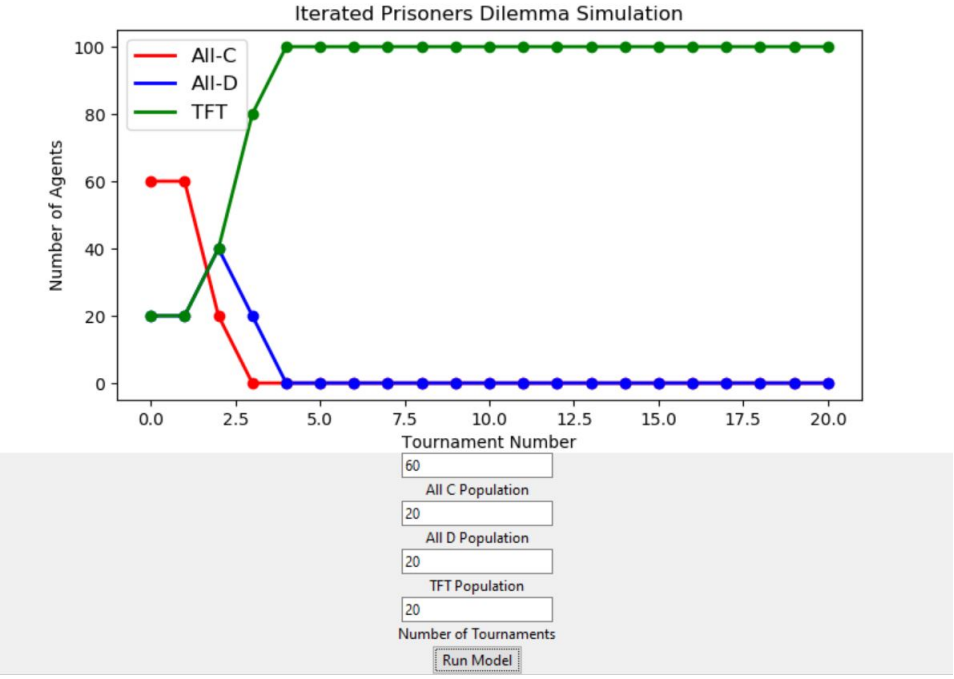

Figure 6: User view of the interface with given population values and number of tournament rounds

The "run model" button then calls a method of the same name to define the agents, create the evolution object, and run the simulation with the given information. This method also updates the data in order to have accurate plots. It adds the results of each tournament to the x and y data that is being graphed.

3.6.7 Steps to Simulating Evolution

In order to put all of these classes together to have a successful simulation of the iterated prisoner's dilemma there has to be many constraints defined. First the number of agents that will only cooperate, only defect, or use the tit-for-tat strategy must be defined. As well as a certain number of tournament rounds must be defined. The program then takes the rest into it's own hands by computing the necessary scores, determining the population numbers, and graphing the results for analysis. In the next section we will see how important these simulations are to our mathematical analysis above.

4 Simulations

4.1 Using a Simulation to Validate Calculations of Iterated 2-Player Game

In Section 2.5.1, we proposed the hypothesis that if we have an iterated prisoners dilemma simulation with $N = 10$ rounds, 8 agents play with the tit-for-tat strategy, and 92 agents play the defection strategy that the agents playing the tit-for-tat strategy will have a higher score. We proved mathematically that any value of the tit-for-tat population greater than a value of approximately 7.05 will gain a higher score than the defection population. Let's test this with a simulation, the cooperation population will be set to 0, defection will be set to 92, tit-for-tat will be set to 8, and the number of tournament rounds will be 10.

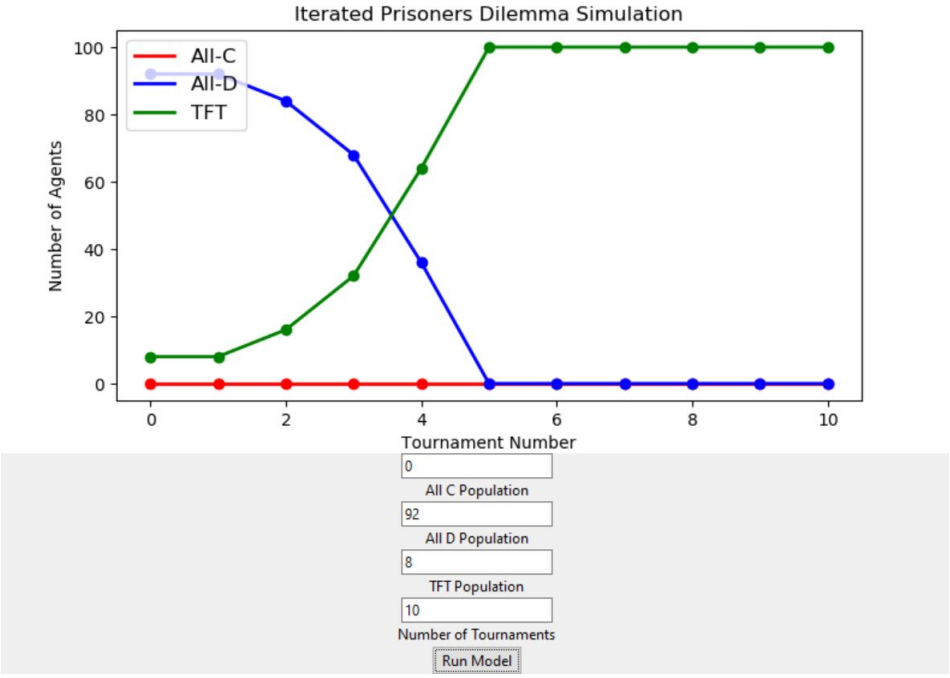

Figure 7: Simulation Results for $N = 10$ rounds, with population all-D = 92, TFT = 8, all-C = 0

In Figure 7, we see that defection population slowly begins to decrease around the same time the tit-for-tat population begins to increase at about 2 rounds. At 4 rounds total the populations cross and then the tit-for-tat population completely over comes the defection population. Like stated before the population of tit-for-tat agents has to be greater than 7.05 so let's examine what happens if the population is set for 7 TFT agents.

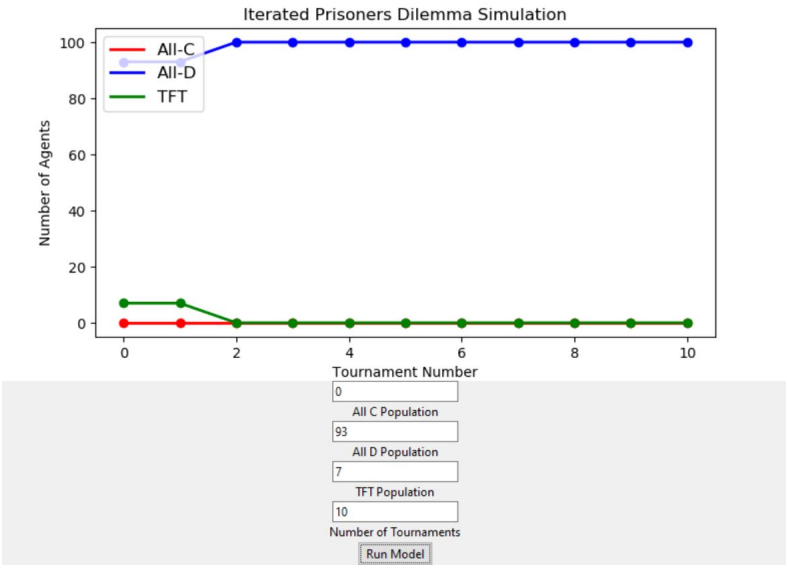

Figure 8: Simulation Results for $N = 10$ rounds, with population all-D = 93, TFT = 7, all-C = 0

Figure 8 has drastically different results. Here the tit-for-tat population completely dies out as a result of defection population taking the lead in every round. Since tit-for-tat defaults to cooperation initially, the agents with the strategy of all defection gain a initial boost of points and the tit-for-tat agents gain none. This means that the top scoring agents of the overall population will always be the all defection agents. Whereas with the previous simulation, since there is just one more tit-for-tat agent they manage to still stay in the population unlike this scenario.

4.2 Simulation of the Three Player Game

In Section 2.6, we showed that in a three player game of the iterated prisoners dilemma of 10 rounds, the equation for the tit-for-tat agents to defeat all other populations is $t > \frac{1}{17} + \frac{19}{17}c$. Mathematically we showed that even though we found a value of t that satisfies this equation, since the values of c and d didn't satisfy the second equation $16 > 36c + 17d$ the tit-for-tat population will not defeat the other populations in the system. This does not necessarily mean that over many tournaments with 10 round each this will be the case. Using our code, we can see what happens over 10 tournaments with 10 rounds each.

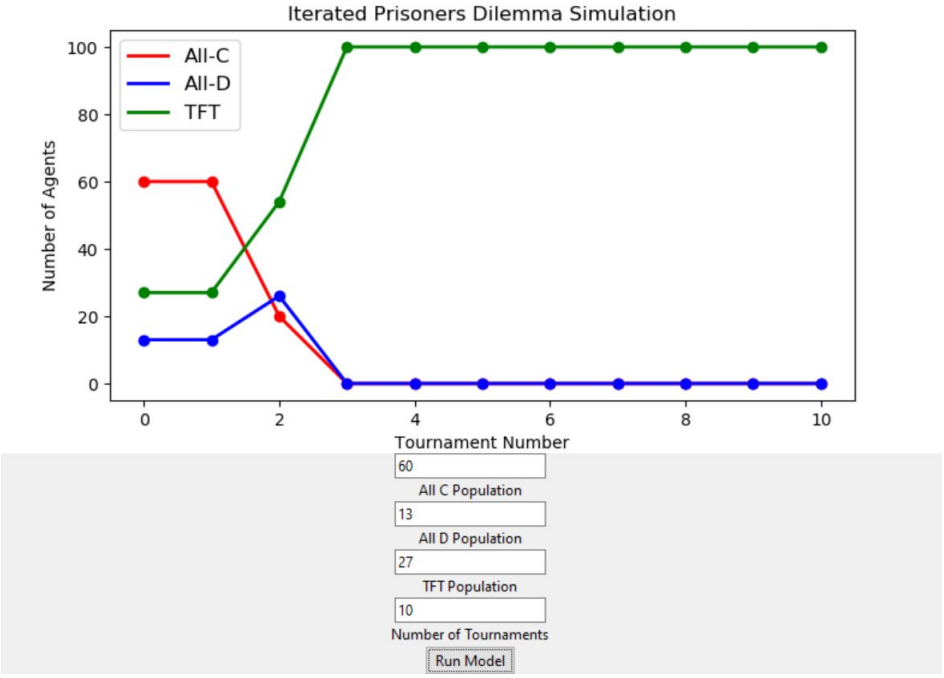

Figure 9: Simulation Results for $N = 10$ rounds, with population all-D = 13, TFT = 27, all-C = 60

Figure 9 shows us that the tit-for-tat population does not surpass the cooperation population until the second tournament, which proves our mathematical analysis to be correct. Another interesting observation is that the graph shows that not only does the tit-for-tat population surpass the cooperation population but also is responsible for the complete eradication of the cooperation and defection populations in the system. This is something that we would not have seen from our mathematical analysis and there are many real world problems that encounter this.

4.3 Simulating Mathematically Intractable Scenarios

There are many scenarios in the real world where scientists come up with situations that simply cannot be proven or are difficult to prove by hand. This is important especially for modeling complex systems in an environment where there are a large number of agents with behaviors than can evolve. For our project it

is important to note that the code will run with large population sizes and a large number of tournaments, but it is not always easy or practical to compute population values by hand like we did in Section 2. Lets take for example a simulation of 30 tournaments with 10 rounds each, a cooperation population of 298, and a defection and tit-for-tat population of 1.

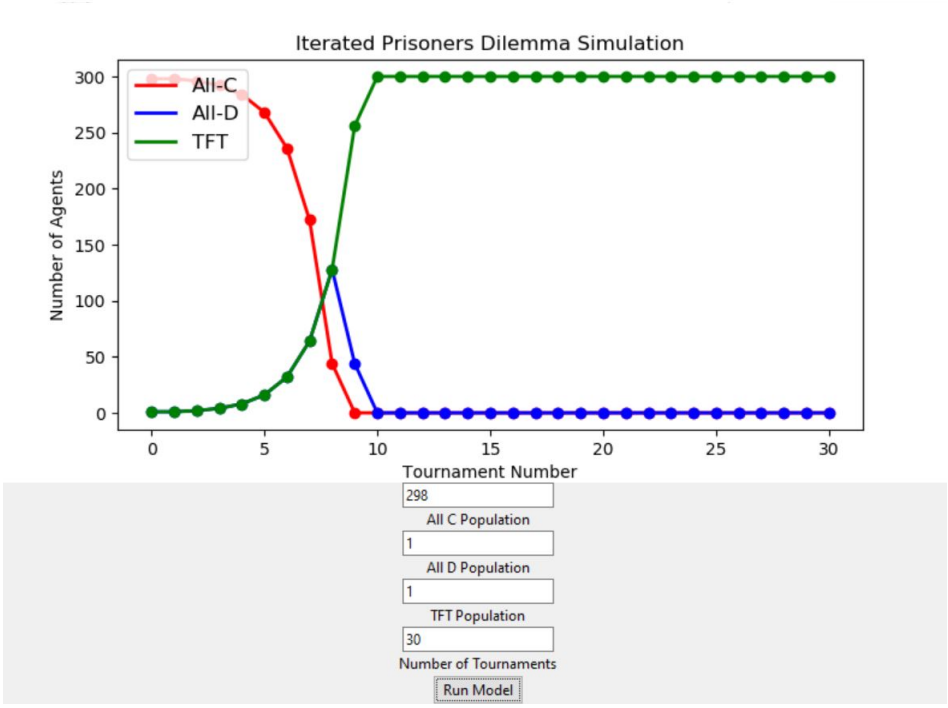

Figure 10: Simulation Results for $N = 30$ rounds, with population all-D = 1, TFT = 1, all-C = 298

Although this example may look a little silly, it is important to note the time it took for the cooperation population to be overcome by the tit-for-tat and defection populations. With a smaller population size it may only take a few tournaments to see changes in the systems overall populations. The cooperation population begins to shrink after three tournaments, but the tit-for-tat and defection populations do not begin to noticeably grow until tournament five. Typically with all the previous simulations in this section, they have been 10 tournaments each. If we were limited to that in this scenario we maybe have missed important information about the graph since it is still changing close to 10 tournaments in to the simulation. This raises awareness to the fact that simulations evolve over time and having computational power to conduct these simulations is important to having tangible results.

5 Conclusions and Future Work

5.1 Future Work

Although we did a study in agent-based modeling in Python, we only reached many of the surface level ideas. There are three main strategies touched upon in this study: all cooperation, all defection, and tit-for-tat. As we saw in earlier sections and through Axelrod's experiments, the tit-for-tat strategy always prevails given a high enough population. Our code does not run with other strategies besides those three. For future work, I would be interested in adapting the code to handle more than these strategies and determine if tit-for-tat is really as strong as it has always been deemed to be. It could also be interesting to attempt to create our own strategies and see how relatively strong and weak they are in comparison to those studied already. Another interesting direction for this project to lead to would be finding a way to adapt the code to a real world scenario. Although the code does study relative populations, it would have been interesting to see

if we would be able to study the dynamics of a crowd of humans moving through a large concert hall or different species of animals who follow similar behaviors that can be programmed into the code.

5.2 Summary

Since game theory, the prisoner's dilemma, and agent based modeling have applications in many unexpected fields, we see the emergence for a need for tools to analyze each of these topics. What we were able to accomplish is create a step towards the creation of agent based modeling software in Python. This was accomplished by studying the previously made software in Java and choosing the best qualities by comparing each. Before creating the software, to understand the best way to visualize the populations of agents and their patterns of behaviors we studied the mathematics behind all the strategies. The analysis of slope fields, Euler's method, and techniques to normalize vectors, helped create clear graphs that allowed us to visualize at what points in simulations populations grow and shrink. These graphs also allowed us to validate our proof results, as well as study scenarios that aren't computed so easily by hand. The influential scientists and mathematicians, such as von Neumann, Nash, and Axelrod, devised important parts of game theory that are all still used today to understand biological patterns, strategies for war, and economic strategies. Game theory will always have answers to situations that may not be understood out right, which continues to make it an important topic of study for generations to come.

6 Appendix

6.1 Evolution Code

```
#!/usr/bin/env python3
import random

####   Agent CLASS  ####
#      DESCRIPTION    #
# Stores information for #
# agents including probs #
# names, and scores.   #
# Creates clones of obj #
# Chooses C or D based #
# on probabilities and #
# the opponents choice #
##########################

class Agent:
    ## Initializing all the information needed to declare an agent ##
    def __init__(self, probFirst, probC, probD, name="Agent"):
        self.p1 = probFirst
        self.pC = probC
        self.pD = probD
        self.name = name
        self.gameMemory = [] # game only
        self.gameScore = 0
        self.tournamentScore = 0

    ## Creates a clone of a given Agent object ##
    def clone(self):
        newAgent = Agent(self.p1, self.pC, self.pD, self.name)
        return newAgent

    # randomly return "C" with probability p, else return "D"
    def randomCorD(self, p):
        r = random.random()
```

```
        if r < p:
            return "C"
        else:
            return "D"

    # Picks the value (C,D) based on probabilities and helper method
    def choose(self, opponent):
        # the game round number (1-based, not 0-based)
        #  can be calculated from the amount of data stored in memory
        currentRoundNumber = len(self.gameMemory) + 1
        if currentRoundNumber == 1:
            return self.randomCorD( self.p1 )
        else: # currentRoundNumber > 1
            previousRoundNumber = currentRoundNumber - 1
            previousRoundMemoryIndex = previousRoundNumber - 1
            opponentChoice = opponent.gameMemory[previousRoundMemoryIndex]
            if opponentChoice == "C":
                return self.randomCorD( self.pC )
            else:
                return self.randomCorD( self.pD )

    #Clears the GAME only, not to use for Tournaments
    def gameClear(self):
        self.gameMemory = []
        self.gameScore = 0

    #Clears the TOURNAMENT only, not to use for games
    def tournamentClear(self):
        self.tournamentScore = 0


####    GAME CLASS  ####
#      DESCRIPTION    #
# Plays a single game. #
# Computes points earned #
# by each player by    #
# their choices.       #
# -------------------- #
# Updates an agents    #
# game score and memory #
# in the Agent class   #
##########################
class Game():
    @staticmethod
    def evaluate(myChoice, opponentChoice):
        if myChoice == "C" and opponentChoice == "C":
            return 3
        elif myChoice == "C" and opponentChoice == "D":
            return 0
        elif myChoice == "D" and opponentChoice == "C":
            return 5
        elif myChoice == "D" and opponentChoice == "D":
            return 1
        else:
            return "Error"

    @staticmethod
    def play(agentA, agentB, numRounds):
        agentA.gameClear()
```

```
        agentB.gameClear()
        for i in range(1, numRounds+1):
            choiceA = agentA.choose(agentB)
            choiceB = agentB.choose(agentA)
            agentA.gameMemory.append(choiceA)
            agentB.gameMemory.append(choiceB)

            agentA.gameScore += Game.evaluate(choiceA, choiceB)
            agentB.gameScore += Game.evaluate(choiceB, choiceA)


#### AGENTUTILS CLASS ####
#      DESCRIPTION     #
# Extra methods to aid #
# in the creation and  #
# agent management of  #
# Tournaments and      #
# Evolution            #
##########################
class AgentUtils():
    # if given a dictionary of agent prototypes and counts,
    #  convert to actual list of agents
    @staticmethod
    def convertToAgentList( agentDictList ):
        agentList = []
        for agentDict in agentDictList:
            agent = agentDict["agent"]
            count = agentDict["count"]
            for i in range(0, count):
               agentList.append(agent.clone())
        return agentList

    ## Determines the number of each agent type in a list of agents ##
    @staticmethod
    def agentCounts(playerList):
        agentNames = []
        for agent in playerList:
            if agent.name not in agentNames:
                agentNames.append(agent.name)

        agentCounts = {}
        for name in agentNames:
            agentCounts[name] = 0
        for agent in playerList:
            agentCounts[agent.name] += 1
        return agentCounts

#### TOURNAMENT CLASS ####
#      DESCRIPTION     #
# Runs a series of games #
# depending on the     #
# population size      #
# -------------------- #
# Updates an agents    #
# tournament score in  #
# the Agent class      #
##########################

class Tournament():
```

```
    #runs tournament eliminating repeats and increments the score
    # returns: playerList
    @staticmethod
    def run( playerList ):

        # type checking:
        # if players is a list of dictionaries of agents,
        #  convert to list of agents
        if type( playerList[0] ) is dict:
            playerList = AgentUtils.convertToAgentList( playerList )

        # just in case
        if type( playerList[0] ) is not Agent:
            Exception("Player parameter does not contain agents")

        # shuffle for testing
        random.shuffle( playerList )

        for agent in playerList:
            agent.tournamentClear()

        for indexA in range(0, len(playerList)):
            for indexB in range(indexA+1, len(playerList)):
                agentA = playerList[indexA]
                agentB = playerList[indexB]
                Game.play(agentA, agentB, 10)
                agentA.tournamentScore += agentA.gameScore
                agentB.tournamentScore +=agentB.gameScore

        debug = False
        if (debug):
            # determine which agent names are present
            agentNames = []
            for agent in playerList:
                if agent.name not in agentNames:
                    agentNames.append(agent.name)

            # count how many of each agent name there is
            agentCounts = {}
            for name in agentNames:
                agentCounts[name] = 0
            for agent in playerList:
                agentCounts[agent.name] += 1
        return playerList


#### EVOLUTION CLASS ####
#      DESCRIPTION     #
# Runs a series of tour. #
# depending on the     #
# population size      #
# -------------------- #
#                      #
##########################
class Evolution():

    def __init__(self, init_C, init_D, init_TFT):
        self.round = 0
        self.init_C = init_C
```

```
        self.init_D = init_D
        self.init_TFT = init_TFT

        self.Cpop = init_C
        self.Dpop = init_D
        self.TFTpop = init_TFT

        self.round_history = []
        self.C_history = [self.init_C]
        self.D_history = [self.init_D]
        self.TFT_history = [self.init_TFT]


    def run( self, agentList, numOfTournaments ):
        # N times:
        for n in range(0, numOfTournaments+1):
            # incrementing the round number #
            self.round_history.append(n)
            # run a tournament with agentList
            agentList = Tournament.run(agentList)

            #Count the agents after each tournament here from the agentList
            ################################################################
            agentCounts = AgentUtils.agentCounts(agentList)

            if 'All-C' in agentCounts:
                self.C_history.append(agentCounts['All-C'])
            else:
                self.C_history.append(0)

            if 'All-D' in agentCounts:
                self.D_history.append(agentCounts['All-D'])
            else:
                self.D_history.append(0)

            if 'T-F-T' in agentCounts:
                self.TFT_history.append(agentCounts['T-F-T'])
            else:
                self.TFT_history.append(0)
            ################################################################

            # rank agents by tournament score
            agentList = sorted(agentList, key = lambda x: x.tournamentScore, reverse = True)
            # destroy bottom 50%, clone top 50%
            newAgentList = []
            half = int(len(agentList)/2)
            for agentIndex in range(0, half):
                agent = agentList[agentIndex]
                newAgentList.append( agent )
                newAgentList.append( agent.clone() )
            # set this to be new agent list
            agentList = newAgentList

        # summarize final population at end of evolution process
        return AgentUtils.agentCounts(agentList)
```

6.2 Data Visualization Code

```
#!/usr/bin/env python3
from tkinter import *
from tkinter.ttk import *
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use("TkAgg")
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

import numpy as np
from Evolution import *

class GUI():
    def __init__(self):
        root = Tk()

        figure = plt.figure( figsize=(8,4) )
        axes = plt.gca()
        canvas = FigureCanvasTkAgg(figure, master=root)
        canvas.get_tk_widget().grid(row=0, column=0)
        canvas.draw()

        # if you set a limit the graph will not dynamically resize
        # axes.set( xlim=(0,2), ylim=(0,4) )

        # plot the points
        # linestyle: solid, dashed, dotted, dashdot
        lineDataC = plt.plot([], [], linewidth=2, linestyle="solid", color="red", label="All-C")[0]
        lineDataD = plt.plot([], [], linewidth=2, linestyle="solid", color="blue", label="All-D")[0]
        lineDataTFT = plt.plot([], [], linewidth=2, linestyle="solid", color="green",
            label="TFT")[0]
        plt.legend(loc="upper left", fontsize=12)
        plt.xlabel('Tournament Number')
        plt.ylabel('Number of Agents')
        plt.title('Iterated Prisoners Dilemma Simulation')

        AllCLabel = Label(root, text = "All C Population")
        AllCEntry = Entry(root)
        AllCEntry.grid(row = 1, column = 0)
        AllCLabel.grid(row = 2, column = 0)

        AllDLabel = Label(root, text = "All D Population")
        AllDEntry = Entry(root)
        AllDEntry.grid(row=3, column=0)
        AllDLabel.grid(row=4, column=0)

        TFTLabel = Label(root, text = "TFT Population")
        TFTEntry = Entry(root)
        TFTEntry.grid(row=5, column=0)
        TFTLabel.grid(row=6, column=0)

        tLabel = Label(root, text = "Number of Tournaments")
        numofTEntry = Entry(root)
        numofTEntry.grid(row=7, column=0)
        tLabel.grid(row=8, column=0)


        runModelButton = Button(root, text="Run Model")
```

```
        runModelButton.grid(row=9, column=0)

        def runModel():
            allCP = int( AllCEntry.get() )
            allDP = int( AllDEntry.get() )
            TFTP = int( TFTEntry.get() )
            numOfTournaments = int( numofTEntry.get() )

            allC = Agent(1,1,1, "All-C")
            allD = Agent(0,0,0, "All-D")
            TFT  = Agent(1,1,0, "T-F-T")

            testPopulation = [{"agent":allC, "count":allCP}, {"agent":TFT, "count":TFTP},
                {"agent":allD, "count":allDP}]
            evolution = Evolution(allCP, allDP, TFTP)
            tempAgentCounts = evolution.run(testPopulation, numOfTournaments)

            for n in range(0, numOfTournaments+1):

                lineDataC.set_xdata( evolution.round_history[0:n+1] )
                lineDataC.set_ydata( evolution.C_history[0:n+1] )

                lineDataD.set_xdata( evolution.round_history[0:n+1] )
                lineDataD.set_ydata( evolution.D_history[0:n+1] )

                lineDataTFT.set_xdata( evolution.round_history[0:n+1] )
                lineDataTFT.set_ydata( evolution.TFT_history[0:n+1] )

                # also points!
                axes.plot( evolution.round_history[n-1], evolution.C_history[n-1], color="red",
                    marker="o" )
                axes.plot( evolution.round_history[n-1], evolution.D_history[n-1], color="blue",
                    marker="o" )
                axes.plot( evolution.round_history[n-1], evolution.TFT_history[n-1], color="green",
                    marker="o" )

                canvas.draw()

        runModelButton["command"] = runModel

        root.mainloop()

# run this program
gui = GUI()
```

7 Bibliography

References

[1] AnyLogic: Simulation Modelling Software Tools and Solutions,
https://www.anylogic.com/

[2] Ascape 5.6.0,
http://ascape.sourceforge.net/

[3] Eugene M. Izhikevich
Agent Based Modeling
http://www.scholarpedia.org/article/Agent_based_modeling

[4] Paul Blanchard, Robert L. Devaney, and Glen R. Hall.
Differential Equations.
Brooks/Cole and Cengage Learning, Boston, Massachusetts, 2012.

[5] Eric Bonabeau.
Agent-based modeling: Methods and techniques for simulating human systems.
PNAS May 14, 2002 99 (suppl 3) 7280-7287; https://doi.org/10.1073/pnas.082080899

[6] Richard L. Burden and J. Douglas Faires.
Numerical Analysis,. Ninth Edition
Brooks/Cole and Cengage Learning, Boston, Massachusetts, 2011.

[7] Cellular Automata, Eugene M. Izhikevich
http://www.scholarpedia.org/article/Cellular_automata

[8] Gabrielle T. Galante and Eizabeth M. Reid.
Applications of Game Theory to Law.
ΠME Journal, Vol. 15, No. 1, pp 23 - 30, 2019.

[9] GAMA Platform website,
http://gama-platform.org

[10] Genetic Algorithms, Eugene M. Izhikevich
http://www.scholarpedia.org/article/Genetic_algorithms

[11] Slope Fields,
https://www.geogebra.org/m/MJbBarpr

[12] Java Agent-Based Modelling toolkit,
https://jabm.sourceforge.io/

[13] Mesa Overview Documents Website,
https://mesa.readthedocs.io/en/master/overview.html

[14] Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[15] Gunaratne, C., Garibay I. (2018) *NL4Py: Agent-Based Modeling in Python with Parallelized NetLogo Workspaces* Complex Adaptive Systems Lab, University of Central Florida.

[16] What is Applied Mathematics and Computational Science?,
https://www.siam.org/careers/resources/details/what-is-applied-mathematics-and-computational-science

[17] SARL agent-oriented language website,
http://www.sarl.io.

[18] The SCALA Programming Language,
https://www.scala-lang.org/

[19] MASON: Multi-Agent Simulation Toolkit,
https://cs.gmu.edu/∼eclab/projects/mason/

[20] William Poundstone. *Prisoner's Dilemma.* Doubleday, New York, New York, 1992

[21] Holt, Charles A. and Roth, Alvin E.,
The Nash equilibrium: A perspective,
National Academy of Sciences, 2004
https://www.pnas.org/content/101/12/3999

[22] Payoff Matrix.
https://xplaind.com/953905/payoff-matrix