Solving Normal form games by building a Python package that uses a genetic algorithm.

Cesar Ivan Esparza Soto

September 2015
School of Mathematics
Cardiff University

A dissertation submitted in partial fulfilment of the requirements for MSc (in Operational Research and Applied Statistics) by taught programme.

Abstract

The literature review can be found in the following section. Some history of Game theory will be mentioned, Nash equilibrium is briefly mentioned but still important since the concept of equilibrium is important to discuss in game theory, simple concept about normal form games, history of evolutionary game theory and an explanation of what evolutionary game theory is and the introduction of the concept of equilibrium in evolutionary game theory which is known as evolutionary stable strategies(ESS), and a brief explanation of the similarities with Nash equilibrium.

Dedication

To my family

Declaration

I declare that..

Acknowledgements

I want to thank..

Contents

1 Introduction **3**
1.1 Introduction . . . 3
1.1.1 Project objectives . . . 3
1.1.2 Project structure . . . 4

2 Literature Review **5**
2.1 Game Theory . . . 5
2.1.1 Normal form games . . . 6
2.1.2 Nash equilibrium . . . 7
2.2 Evolutionary game theory . . . 8
2.2.1 Evolutionary stable strategy (ESS) . . . 11
2.2.2 Evolutionary stable strategy and Nash equilibrium . . . 12
2.3 Genetic algorithm and evolutionary game theory . . . 14
2.4 Agent-based modelling (ABM) and object oriented programming (OOP) 15
2.5 Genetic algorithm considerations for the project. . . . 18

3 Software development practices **20**
3.1 Test Driven Development . . . 20
3.2 Version Control . . . 21
3.3 Documentation . . . 23

4 Library **24**
4.1 Library (Package: Ablearn) . . . 24
4.1.1 Population module . . . 25
4.1.2 Environments module . . . 25
4.1.3 Algorithms module . . . 28
4.1.4 Simulation module . . . 30
4.1.5 Overall interaction view . . . 36

5 Application of the code **38**
5.1 How the simulation was built. . . . 38
5.2 Results . . . 40
5.2.1 Prisoner's Dilemma . . . 40
5.2.2 Matching Pennies . . . 49
5.2.3 Battle of sexes . . . 59
5.2.4 Hawk-dove . . . 68

5.2.5 Axelrod python package 73

6 Conclusion and Further work **79**
6.1 General conclusions about the results 79
6.1.1 Further work 80

A Git commands **82**

B Matching pennies simulation table **86**

C Battle of sexes simulation table **87**

D Hawk-dove simulation table **88**

E 'Axelrod' package strategies **89**

F Code **90**
F.1 Init file 90
F.2 Agents module 90
F.3 Environment module 91
F.4 Genetic module 93
F.5 Simulation module 96
F.6 Matrix tranponser module 103
F.7 Tests module 105
F.7.1 Simulation module test 105
F.7.2 Agents module test 108
F.7.3 Environment module test 108
F.7.4 Algorithm module test 110
F.7.5 Matrix transponser module test 115

List of Tables

2.1 Normal form game bi-matrix 7
2.2 Best responses for player 1 8
2.3 Best responses for player 2 8
2.4 Nash equilibrium with best responses. 8
2.5 General Symmetric Game. 11
2.6 Wolves Hunting 13

5.1 Prisoner's Dilemma 38
5.2 Matching Pennies 39
5.3 Battle of Sexes 39
5.4 Hawk-Dove 39
5.5 Configuration of simulations for prisoner's dilemma. 40
5.6 Initial Prisoners Dilemma distribution for simulation 1. 41
5.7 Initial Prisoners Dilemma distribution for simulation 2. 41
5.8 Initial Prisoners Dilemma distribution for simulation 3. 41
5.9 Generation 5 Prisoners Dilemma distribution for simulation 1. 41
5.10 Generation 5 Prisoners Dilemma distribution for simulation 2. 41
5.11 Generation 5 Prisoners Dilemma distribution for simulation 3. 42
5.12 Generation 10 Prisoners Dilemma distribution for simulation 1. 42
5.13 Generation 10 Prisoners Dilemma distribution for simulation 2. 42
5.14 Generation 10 Prisoners Dilemma distribution for simulation 3. 42
5.15 Generation 25 Prisoners Dilemma distribution for simulation 1. 43
5.16 Generation 25 Prisoners Dilemma distribution for simulation 2. 43
5.17 Generation 25 Prisoners Dilemma distribution for this simulation 3. . . . 43
5.18 Generation 50 Prisoners Dilemma distribution for simulation 1. 43
5.19 Generation 50 Prisoners Dilemma distribution for simulation 2. 43
5.20 Generation 50 Prisoners Dilemma distribution for simulation 3. 44
5.21 Prisoner's dilemma Nash equilibrium with best responses. 48
5.22 Matching pennies bimatrix payoff. 57
5.23 Table of initial values distribution for simulation 19, 5 rounds. 61
5.24 Table of initial values distribution for simulation 19, 25 rounds. 61
5.25 Battle of sexes bimatrix payoff. 67
5.26 Hawk-Dove combinations for simulations 8, 9, 12 and 14. 69
5.27 Hawk-Dove bimatrix payoff. 71
5.28 'Axelrod' package transformed payoff bimatrix for 8 strategies from Axelrod's tournament. 73

5.29 'Axelrod' package payoff bimatrix for basic strategies. 76
5.30 'Axelrod' package and 'ablearn' name equivalence. 77

B.1 Configuration of simulations for matching pennies. 86

C.1 Configuration of simulations for battle of sexes. 87

D.1 Configuration of simulations for hawk-dove. 88

List of Figures

3.1 Basic TDD diagram. 21

4.1 Diagram of population module. 25
4.2 Diagram of environments module. 27
4.3 Diagram of algorithms module. 30
4.4 Diagram of simulation module. 35
4.5 Diagram interaction of modules. 37

5.1 Prisoner's dilemma simulations 1, 2 and 3. 44
5.2 Prisoner's dilemma with mutation rate equal to 0 45
5.3 Prisoner's dilemma simulations 3, 4 and 5, with 5 rounds per generation . 45
5.4 Prisoner's dilemma simulations 5, 6 and 7, with 5 rounds per generation . 46
5.5 Prisoner's dilemma simulations 10, 11 and 12, with 5 rounds per generation 47
5.6 Prisoner's dilemma simulations 16, 17 and 18, with 5 rounds per generation 47
5.7 Simulation 1, 2 and 3 50
5.8 Simulation 10 with 5, 25 and 100 rounds per generation 50
5.9 Simulation 12 51
5.10 Simulation 13, 14 and 15 with 5 rounds per generation 51
5.11 Simulation 13, 14 and 15 with 25 rounds per generation 51
5.12 Simulation 13, 14 and 15 with 100 rounds per generation 52
5.13 Simulation 16, 17 and 18 with 5 rounds per generation 52
5.14 Simulation 16, 17 and 18 with 25 rounds per generation 53
5.15 Simulation 16, 17 and 18 with 100 rounds per generation 53
5.16 Simulation 19, 20 and 21 with 5 rounds per generation 53
5.17 Simulation 19, 20 and 21 with 25 rounds per generation 54
5.18 Simulation 19, 20 and 21 with 100 rounds per generation 54
5.19 Stack plots from simulations 19, 20 and 21 with 25 rounds per generation 54
5.20 Simulation 22, 23 and 24 with 5 rounds per generation 55
5.21 Simulation 22, 23 and 24 with 25 rounds per generation 55
5.22 Simulation 22, 23 and 24 with 100 rounds per generation 55
5.23 Simulation 25, 26 and 27 with 5 rounds per generation 56
5.24 Simulation 25, 26 and 27 with 25 rounds per generation 56
5.25 Simulation 25, 26 and 27 with 100 rounds per generation 56
5.26 Stack plots for simulation 27 with 5, 25 and 100 rounds per generation . 57
5.27 Graph and stack plot for simulation 4 with no inputed initial strategy, with 5 rounds per generation 59

5.28 Graph and stack plot for simulation 4 with no inputed initial strategy, with 25 rounds per generation . . . 60
5.29 Graph and stack plot for simulation 19 with no inputed initial strategy, with 5 rounds per generation . . . 60
5.30 Graph and stack plot for simulation 19 with no inputed initial strategy, with 25 rounds per generation . . . 61
5.31 Graph and stack plot for simulation 4 with initial strategy, with 5 rounds per generation . . . 62
5.32 Graph and stack plot for simulation 4 with initial strategy, with 5 rounds per generation . . . 62
5.33 Graph and stack plot for simulation 4 with initial strategy, with 25 rounds per generation . . . 62
5.34 Graph and stack plot for simulation 4 with initial strategy, with 25 rounds per generation . . . 63
5.35 Graph and stack plot for simulation 19 with initial strategy, with 5 rounds per generation . . . 63
5.36 Graph and stack plot for simulation 19 with initial strategy, with 5 rounds per generation . . . 63
5.37 Graph and stack plot for simulation 19 with initial strategy, with 25 rounds per generation . . . 64
5.38 Graph and stack plot for simulation 19 with initial strategy, with 25 rounds per generation . . . 64
5.39 Graph and stack plot for simulation 4 with initial strategy (0.7, 0.3), with 5 rounds per generation . . . 65
5.40 Graph and stack plot for simulation 19 with initial strategy (0.7, 0.3), with 5 rounds per generation . . . 65
5.41 Graph and stack plot for simulation 4 with initial strategy (0.3, 0.7), with 5 rounds per generation . . . 66
5.42 Graph and stack plot for simulation 19 with initial strategy (0.3, 0.7), with 5 rounds per generation . . . 66
5.43 Graphs for simulations 3, 4 and 5 . . . 68
5.44 Stackplots for simulations 3, 4 and 5 . . . 69
5.45 Graphs for simulations 8 and 9 . . . 70
5.46 Graphs for simulations 12 and 14 . . . 70
5.47 Stack plots for simulations 8 and 9 . . . 70
5.48 Stack plots for simulations 12 and 14 . . . 71
5.49 Stack plot with 100 time units produced with ‘axelrod’ package. . . . 74
5.50 Stackplots for simulations from ‘ablearn’ package. . . . 75
5.51 Stack plot with 1000 time units produced with ‘axelrod’ package for basic strategies. . . . 76
5.52 Stackplots for simulations from ‘ablearn’ package. . . . 77

A.1 Example of one file staged with git add. . . . 82
A.2 Example after using git commit asking for message for the commit. . . . 83
A.3 Example of general output with git diff and git diff HEAD - - stat (red color represent the deleted lines and green the added) . . . 83

A.4 Example diagram from Github of a master branch with a branch 1 for being merged. 84
A.5 Basic example of git log oneline. 85

Summary

Game theory is field that has been used more frequently nowadays in a wider range of fields. The expansion in fields of application, have led to variations in the original assumptions used for games in game theory. In this project a review of "evolutionary game theory" was presented, which is a subfield of game theory. Evolutionary game theory is the application of game theory to the concept of evolution. This was relevant for understanding, how seen from a biological sense, game theory can be applicable to concepts of survival of species. With this as a basic idea it is easier to understand the concept of the "genetic algorithm" used to create a python library for this project. The "genetic algorithm" used was simple, and it mainly uses concepts and assumptions from "evolutionary game theory". The algorithm basically consists in the creation of agents with characteristics. The approach taken for defining agents, was agent-based modelling, which main purpose is to simulate the how the interactions between different agents has an effect on the individual agent and in the whole system. The main characteristics for each agent are: each created agent has only one strategy which it will use until it is eliminated and by using this strategy the agent is capable of accumulating a utility. When the utility is high it will help it to endure during the simulation, and it will very likely be able to produce a copy of itself, consequently the strategy it passes on will be present in more agents during the simulation. On the other hand, agents with a low accumulated utility will probably disappear as the simulation progresses. Parameters that affected the simulation were introduced, such as death rate, mutation rate, exploitation rate and initial distribution of strategies. For being able to build a simulation with all of the features mentioned "Python" was used which an object oriented programming language. Once a Python library capable of running simulations was created some of the most known games in game theory like prisoner's dilemma, matching pennies, battle of sexes and hawk-dove (in evolutionary game theory) were simulated. All these games were tested with different combination of parameters, to observe their individual behaviour. And peculiarities for each game, according to their types of equilibria (mixed or pure) were observed. After these games were tested an additional game was tested. This game was produced with a Python package called "axelrod", which allows simulate tournaments with predetermined strategies. From "axelrod" a payoff matrix was taken, this payoff matrix was transformed into a bimatrix and introduced into the Python library created for this project. The "axelrod" package also produces a stack plot, the library for this project also produces a stackplot and both stack plots were compared. After producing the different outputs for the different games and also after comparing results with the results from "axelrod" package, the library built for this project which name is "ablearn" did produce expected results for the known games for game theory and produced somehow

similar stack plots to the stack plots produced by “axelrod” package.

Chapter 1

Introduction

1.1 Introduction

In nature, each living creature has specific characteristics that determine its role in its environment. From microscopic entities to blue whales, all organisms possess patterns that enable them to live, reproduce and die. Whilst the life cycle is a process common in all life forms, differences between them are clear. In the colder environments, living entities like polar bears with thick fur have different characteristics to animals in hotter environments as a camel's body temperature fluctuates with its surroundings. Arguably, the most significant difference is behaviour as it defines the role every existing organism has. Consider the example of a predator and its prey. Their roles can be subjective and the prey in this instance can become a predator to another. That is the essence of behaviour; how particular it is to each type of organism but with a final common goal, survival.

Humans are inherently complex and their basic rules of survival have changed. Survival appears to be secondary to superficial concerns, an example like eating has been belittled with the focal concern to survive. This peculiarity could be one of the main reasons for changes in human behaviour. The key word for me is rules. Rules dictate who we are and how we behave around others, and while following these rules we adapt to prevail in the environment. It is bold to say that we all play a game, life, and the rules are not only dictated by our natural environment but also by the social environment.

1.1.1 Project objectives

This project aims to give a general framework of a particular branch of game theory, evolutionary game theory, and to build a library in python to solve game theoretical problems presented in the normal form with considerations made from evolutionary game theory. The program built in Python will use a simplified genetic algorithm. The approach that will be used for modeling is agent based modeling (ABM), which will be introduced further in this work. Basic relevant knowledge about evolutionary game theory will be presented.

Python programming language will be used to build a library capable of solving some well known normal form games (games is strategic form) like prisoners' dilemma, matching pennies, battle of sexes and some others. A Python package will be used to simulate an experiment made by the professor in political science Robert Axelrod in 1980, who's goal was to observe how cooperation evolved in an environment where selfish behaviours existed when interacting through time.

1.1.2 Project structure

This project has the following structure. In the next section which is literature review, will have some background about game theory, which will include normal form games and Nash equilibrium; followed by an introduction to evolutionary game theory, a definition for evolutionary stable strategy and how it relates to Nash equilibrium; background information about genetic algorithm and how it relates to evolutionary game theory; background information about agent-based modeling and the object oriented programming style; finally the considerations made to use a genetic type algorithm in the project. Section three will explain the use of the approach test driven development for building code; an introduction to version control system Git for creating packages and documents in which others can contribute; general guidelines of what tools and webpage will be used for providing potential users the instructions and relevant information about the project. In section four an explanation of the modules contained in the library, the interaction between them, and the variables and methods contained within the modules will be given. Finally for section five, examples of the different outputs the python library provides will be presented with some different combinations of parameters that can be used, applied to some well known game theoretical games in the normal form.

Chapter 2

Literature Review

2.1 Game Theory

Game theory or theory of games is a wildly known subject, which studies the interaction of decisions. This interaction is given to the interdependence between the participants in an environment. Resulting from this there can be mainly two types of interaction cooperative or competitive [1]. The whole concept of game theory has a very wide application in many different sciences like economics, companies in a competing market, setting import and exporting tariffs, agreements in wages between employers and employees, auctions, to name some [2]; for political sciences Nolan McCarthy mentions examples such as raising more money than other candidates when running for a political position, what to do when the opposition is considered weak or strong, accepting new policies if the outcomes are not certain, an example is when a jury has to sentence a defendant among others [3]; in psychology, when studying the behavior of individuals with respect of others, although application of strict assumptions of game theory such as rationality been assumed in the same level for all participants still is a controversial topic for the application of game theory in many areas of social psychology [4] [5]; biology which will be mentioned further, related to evolutionary game theory, and other disciplines.

Studies that started exploring the potential solution of some games can be traced back to 1713 when James Waldegrave in a letter communicated a mixed solution for a two person game to his colleague Pierre Remond de Montmort [6] . Since Waldegrave many others did studies that now we relate to game theory [1]. We will start from the french mathematician Emile Borel who in a note in his work "La théorie du jeu et les équations, intégrales à noyau symétrique gauche" [7] mentioned:

"The problems of probability and analysis suggest themselves concerning the art of war, or economic or financial speculations, are not without analogy with problems concerning games, though they generally have a higher degree of complication".

In 1924 Borel on his note "On games that involve chance and the skill of the Players" [8] he mentions "the study of games that involve at once chance and the skill of the

players appears to me similarly able to furnish an opportunity for mathematical research, the applications of which might far surpass the limits of the restricted domain to which this first study is limited. /such research might be extended to very many questions in which psychological unknowns figure along with algebraic unknowns." Then Borel continues saying that the only author who had studied problems with this focus was Joseph Bertrand in his "Calcul des Probalites" in 1889, distinguishing between mathematical and psychological aspects in an example given of a game of baccarat, but goes on stating why Bertrand study was incomplete [8].

Game Theory was further researched and formally presented by the Hungarian mathematician John von Neumann in 1928 with his work Theory of Parlor Games, stating in the introduction that " any event given external conditions and the participants situation (provided the latter are acting of their own free will) may be regarded as a game of strategy if one looks at the effect it has on the participants." [9]. And in 1944 John von Neumann and Oskar Morgestern published "Theory of Games and Economic Behavior" in which they established theory of games of strategy as an instrument to study problems in the economic behavior [10]. Despite of the great contribution of von Neumann and Morgestern the application of game theory was limited. Game theory became widely used after the Doctoral dissertation from John F. Nash was published in 1950. Some of the main arguments from Nashs work are that he gives the possibility of analyzing games with more n-players, he introduces the concept of non-cooperative games having at least one equilibrium point, and gives the idea of a dynamical approach to study cooperative games [11]. The last point I mentioned refers to how a cooperative game can be reduced to non-cooperative form, because the idea of a cooperative game established by von Neumann and Morgenstern was that cooperation was given under the assumption described in Nashs words players can communicate and form coalitions which will be enforced by and umpire. [11] , but he proves this condition is not the only one that would define a cooperative game. The work from Nash opened the possibility of a broader application of the essential concept of game theory, many people have extended the fundamental concepts adopted from Nash's work and a lot of studies for the development of game theory have been made since.

2.1.1 Normal form games

Watson 2013[1] mentions that there are several mathematical ways to describe games. In game theory there are two most common ways to represent a game called extensive form and normal form. The extensive form which represents in the form of a game tree the different actions that can be taken by each player. Starts with an initial node from which it branches out representing the possible choices and then after each branch another node is placed where a second player has other choices that branch out, and keeps on until reaching the end of the tree where the payoff for each sequence(following the branches) for the path are represented. For the purpose of this project, the normal form representation of a game will be used and the basic information about it is the following. The normal form games, also known as strategic form of games, usually have the following elements:

- **Players** A finite set of N players.
- **Strategies:** A set S_i for each player $i \in N$. Which as we have discussed represents the action a player chooses.
- **Payoffs:** Payoff functions represented for each player represented by u_i: S_1 x S_2 x ... x $S_N \rightarrow \mathbb{R}$ .Which represent the payoff each player obtain after interacting with each other.

For the purpose of this project, the number of players will be $N = 2$ taking the following assumptions from Knight 2014 [12] we will represent the game with a **bi-matrix**. And we will assume that $\mathbf{S_1 = \{s_i \mid 1 \geq i \geq n\}}$ and $\mathbf{S_2 = \{s_j \mid 1 \geq j \geq n\}}$ this will be the **bi-matrix** for this game :

Player 2

		s_1	s_2	...	s_j
	r_1	$u_1(r_1,s_1)$, $u_2(r_1,s_1)$	$u_1(r_1,s_2)$, $u_2(r_1,s_2)$	...	$u_1(r_1,s_j)$, $u_2(r_1,s_j$
	r_2	$u_1(r_2,s_1)$, $u_2(r_2,s_1)$	$u_1(r_2,s_2)$, $u_2(r_2,s_2)$	...	$u_1(r_2,s_j)$, $u_2(r_2,s_j$
Player 1	.	.	.	...	.
	.	.	.	...	.
	.	.	.	...	.
	r_i	$u_1(r_i,s_j)$, $u_2(r_i,s_j)$	$u_1(r_i,s_j)$, $u_2(r_i,s_j)$	...	$u_1(r_i,s_j)$, $u_2(r_i,s_j)$

Table 2.1: Normal form game bi-matrix

We can see that each intersection of row and column has two utility functions. The first utility function represents the payoff for the row player and the second utility function represents the payoff for the column player. Utilities can be of type ordinal, which is used only to rank the alternatives from better to worse, or cardinal which indicates that the value assigned is meaningful and represents the satisfaction of the player [3].

2.1.2 Nash equilibrium

John Nash writes in his dissertation "... an equilibrium point is an n-tuple **s** so that each player's mixed strategy maximizes his payoff if the strategies of the others are held fixed. Thus each player's strategy is optimal against those of the others." [11]. After this he describes under what circumstances a mixed strategy behaves as a pure strategy. Nowadays the definition of Nash's equilibrium point is known as a Nash equilibrium and we will commonly find the phrase "no regrets" when describing the Nash equilibrium and this is because paraphrasing Nash, no player in the game could have had a better payoff given the action chosen by her opponent. For an N player normal form game Nash equilibrium Knight 2014[12] gives the following definition:

$$u_i(\hat{s}) \geq u_i(\bar{s}_i, \wedge s_{-i}) \forall i \tag{2.1}$$

In a two player game, this means each player i has a set of strategies S, and a pair of strategies $(\hat{r}, \hat{s})$ so pure Nash equilibrium can be defined as:

$$u_1(\hat{r}, \hat{s}) \geq u_1(r, \hat{s}) \forall r \in S_1 \text{ and } u_2(\hat{r}, \hat{s}) \geq u_2(\hat{r}, s) \forall r \in S_1 \tag{2.2}$$

An "easy" way to find Nash equilibria in games with pure Nash equilibria is by looking at the table in the normal form that holds the strategies and payoffs for each player, and underlining the best response for each player when the other player's strategies are held fixed. As an example we will use the prisoner's dilemma normal form, and first we hold fixed player's 2 strategies and we underline the best response for player 1 for each of player's 2 strategies.

		Player 2	
		Cooperate	Defect
Player 1	Cooperate	3, 3	0, 5
	Defect	$\underline{5}$, 0	$\underline{1}$, 1

Table 2.2: Best responses for player 1

In this first table we have that defect is the best response for player 1 in the case where player 2 chooses either of the strategies. Now we explore the options holding player 1's strategies fixed and we have the following.

		Player 2	
		Cooperate	Defect
Player 1	Cooperate	3, 3	0, $\underline{5}$
	Defect	5, 0	1, $\underline{1}$

Table 2.3: Best responses for player 2

And now we only take the strategies that are best responses for both players. Remembering that is the strategy where neither of the players have any reason to change from given that whatever the other player choice is it will always give them the best payoff available for the interaction.

		Player 2	
		Cooperate	Defect
Player 1	Cooperate	3, 3	0, 5
	Defect	5, 0	$\underline{1}$, $\underline{1}$

Table 2.4: Nash equilibrium with best responses.

2.2 Evolutionary game theory

A branch of game theory which is of great interest for the purpose of this project is 'Evolutionary game theory'. Evolutionary game theory has its roots in evolutionary biology. Even if this overview is not intended to be focused in biology some remarks from it are worth mentioning. In the 6th edition of Charles Darwin's work "Origin of species" he outlines that in nature there exist many struggles for existence. Some examples are the struggles of a species in the nature, between species and within species. Darwin stresses

that the most severe struggle might be within species if they become into competition [13]. Darwin 1872 mentions that there may be infinite varied diversities of structure for each being under the changing conditions of life, and continues saying that through the course of generations, there can occur variations that can give perhaps a slightly advantage to some beings giving them a higher chance of survival and procreation. The preservation of favourable characteristics and destruction of what he called injurious, was called 'natural selection' or 'survival of the fittest' [13]. I go this far because later on we can think of the concept of fitness, as the utility a player has in evolutionary game theory which in general terms can be called the fitness.

The scientist Ronald Fisher might have been the first to apply game theory to evolution in his study on sex ratios in 1930 [14] although at this time the formal definition of game theory had not been presented yet. Later in 1961 Lewontin discussed species playing against nature, and said that species should adopt the "maximin" strategy if nature presented worse-case scenarios [15]. In 1967 in an the article "Extraordinary sex ratios" wrote in 'Science' William D. Hamilton uses game theory to model local competition and frequency-dependent fitness values [15]. But perhaps the most important contribution in evolutionary game theory was the one made by Professor John Maynard and Dr. George Price. John Maynards attention was first caught by the article "Antlers, Intraspecific Combat, and Altruism", written by George Price in 1968 for 'Nature about ritualized behavior in animal contests [16] unpublished for being too long. Then in 1973 John Maynard and George Price published a joint paper On the logic of animal conflict in which the mathematical concept of an evolutionary stable strategy is established, and it applies concepts of game theory to the study of conflicts between animals [15]. The idea Maynard and Price presented was that concepts from game theory could be used to characterize eventually stable endpoints in the evolutionary process, with the concept of evolutionary stable strategy [17]. It can be said that the concept of evolutionary game theory was born with the ideas of John Maynard and George Price.

Nowadays evolution by natural selection can be thought of as a game, where some behavioural patterns (often referred to as phenotypes the equivalent to strategies when related to traditional game theory) from animals are more successful than others [18]. Now I can relate evolutionary game theory with traditional game theory in the some essential concepts. Animals in the biological concept are equivalent to the players (agents) that participate in a game; the environment in which animals interact is comparable to the set of rules that regulate interaction in the traditional concept of games; as mentioned before the heritable phenotypes of animals can be thought of as the strategies that players use in the traditional concept of games; a tricky concept to relate to evolutionary game theory is the one of payoffs (utility) in traditional game theory for this I will refer to how Maynard and Price defined it in 'The logic of animal conflict' and this was as the contribution the contest has made to the reproductive success of the organism (agent) [19] which could be the expressed in terms of fitness [13] the fitness in an organism (agent) directly influences the frequency of the strategy in the population [20], Maynard and Price take in account three factors: the advantages of winning compared to losing, disadvantage of being seriously injured and disadvantage of wasting time and energy in the contest [19] this are not usually considered in games but I consider important mentioning. Another very important concept is equilibrium, in some evolutionary games

the existence of evolutionary stable strategies (ESS) , the mathematical representation of ESS will be presented later in this project. Roughly we can consider an ESS as a strategy that predominates in frequency in evolutionary games through time and that in the case of the emergence of a mutated strategy is not invaded (threatened to be reduce in number) this concept of ESS has similarities with the concept of Nash equilibria as seen by in the sense that both can be "no-regret" strategies when in a population a Nash or an ESS is played "no individual can benefit from unilaterally changing their strategy" [20].

Thinking of how the idea of evolutionary game theory was conceived from behavior of animals, comparing it to the traditional game theory, there are some characteristics that distinguish them from each other. First and perhaps one of the most relevant assumptions in traditional game theory is that every player is rational which means they make rational decisions to maximize their profits, they are also aware of the possible payoffs of the other players and that other players are rational, and the rational players are aware of the game rules, evolutionary game theory does not make such assumption of rational players, instead the strategies are hard wired to them in other words they are assigned to each player with no possibility of the player choosing. Traditional game theory is about choosing from different strategies looking to optimize the payoffs, whilst evolutionary game theory is used to determine strategies that will endure through time. Traditional game theory as said before has a set of strategies from which players can choose, whilst in evolutionary game theory the strategies are already defined (given that they are inherited), although there can be present some occasional mutations. Also in evolutionary game theory there will be groups of players that possess the same set of strategies and the same related payoff for these strategies, in traditional game theory each player has their own set of strategies and their own associated payoffs per strategy [20]. The application of evolutionary game theory in different areas of study has grown, and with this some assumptions change. For example in the biological application players do not choose their strategies and never change them, unlike in the economic application the players are people, who can choose and change their strategies (Samuelson 1997)[21].

What is evolutionary game theory?

It can be defined as the combination of some game theoretical concepts, with the concepts of natural selection in evolution. It is a change in the focus from traditional game theory, because the main goal of evolutionary game theory is to observe the stable equilibria and how it changes through time with the interactions between the organisms (players) with their own different behaviours (strategies), instead of only focusing in optimizing outcomes for a single game. Something important to note about the interaction is that in evolutionary biology and evolutionary game theory, the concerning interaction is between individuals of the same species. In this sense we can identify two main approaches to evolutionary game theory [22]. The first approach is the static approach which is directly derived from the work of Maynard and Price, the main tool for analyzing is the ESS. The second approach through the study of the population dynamics (change in density of existing strategies) and of how the strategies evolve in the model built [22].

2.2.1 Evolutionary stable strategy (ESS)

To give an easier explanation of ESS, the concept will be analysed with a symmetric game. It is important to mention that the games that will be used in this project are two player games. When describing ESS the most frequent example used is Hawk-Dove, first used by Maynard and Price in their paper "The logic of animal conflict" in 1973 which has been mentioned before. It is worth mentioning that this game has the structure of the well-known prisoner's dilemma game. The 'Prisoner's dilemma was formally presented by Albert Tucker to psychology students in Stanford University in 1950. As we may know prisoners dilemma is a game in which two individuals who committed a crime are interrogated in separate rooms, and they are not able to exchange information with each other. So they are presented with 2 options (strategies), they can confess or not. And the interaction between the choices each one have go as follow. One confesses and the other does not, both of them confess or neither confess. Each interaction has a pay-off. From this point on the use of the word 'player' and I will refer to the players of the games we describe as agents. First a general definition of the ESS should be done. To illustrate this it will be used as a guide the approach from Easley & Kleinberg in "Networks, Crowds and Markets: Reasoning about a Highly connected World"[23] in definition of ESS since it is considered suitable for this project. We know that a strategy is evolutionarily stable when a population of agents can resist the invasion of emergent mutated agents (new strategy). As mentioned before in evolutionary biology as well as in evolutionary game theory, fitness can be defined as reproductive success [23]. Therefore agents with higher fitness value are majority in a population or in time will become majority, whereas agents will low fitness value will be minority and in time with a very high probability will disappear. The fitness value is obtained using the pay-off values from each interaction. An explanation will be given using the table 2.5 to illustrate an example, the table is taken from Easley & Kleinberg 2010 [23]. For a symmetric strategic two-player game we have the following bimatrix.

		Agent 2	
		X	Y
Agent 1	X	a, a	b, c
	Y	c, b	d, d

Table 2.5: General Symmetric Game.

We assume that there exists a large population of agents that always take action X. Now we suppose that within the population appears a small group that takes action Y and the fraction representing the number of agents in this group is ϵ. Since ϵ represents the number of agents that choose Y, we can say that 1 - ϵ is the fraction of the agents that choose X. We will assume then that the probability of an agent using X encountering another agent at uses X is 1 - ϵ, and with a mutated that uses Y the probability is ϵ. With these and the values from the figure 2.5 we build the payoff equation for the agent that use strategy X as follows:

$$(1 - \epsilon)a + \epsilon b \tag{2.3}$$

And with the same values from the table and the proportions in the population, we build

the pay-off equation for the agents using the strategy Y as follows:

$$(1-\epsilon)c + \epsilon d \tag{2.4}$$

For X to be an evolutionary stable strategy, we need:

$$(1-\epsilon)a + \epsilon b > (1-\epsilon)c + \epsilon d \tag{2.5}$$

For X to be an ESS a > c for small values of ϵ, on the other hand when the values of ϵ are closer to 1, for X to be ESS a = c and b > d. Since the pay-offs for each agents are the result from the interactions, we should understand the following [23]:

- For X to be an ESS, the pay-off for interacting with X should be greater or at least equal to the pay-off that a strategy Y gets when interacting with X.
- Or for X to be an ESS when X and Y get the same pay-off while interacting with X, X needs to get a better pay-off when interacting with Y than Y interacting with Y.

We see that if the previous does not hold, then $(1-\epsilon)a + \epsilon b < (1-\epsilon)c + \epsilon d$ and therefore X is not an ESS. So far we have defined the possibility of having a pure strategy dominating in time. There also exist a case where pure strategies can be played with different probabilities, this will prevent from having a pure strategy equilibria causing an evolutionarily stable mixed strategy. I consider important to mention that when only a pure strategy is played by any entity, it means that it is palyed with a probability of 1 (100%) , if a mixed strategy is to be played it means that proportion is distributed among the possible effective strategies. In Osborne 2004[24], we can find a fragment of Maynard Smith's work where he explains that if randomizing between strategies offers an advantage it can evolve in time, therefore it is possible to be inherited from parents.

2.2.2 Evolutionary stable strategy and Nash equilibrium

To gain a better understanding of the concepts linking ESS and Nash equilibrium, let us remember what Nash equilibrium is. An action (strategy) profile s^*, where no player i can do better by choosing a different action (strategy) from s_i^*, holding players j action (strategy) s_j^* fixed [24]. Including the notions of rationality we already know. Now a similar explanation to the one given in 'Networks, Crowds, and Markets: Reasoning about a Highly Connected World' by Easley and Kleinberg[23] will be used. Looking back at table 2.5, we can take (X, X) as a Nash equilibrium, therefore we assume X is a best response to both players. And this would mean the following according to the arbitrary values we assumed:

$$a \geq c \tag{2.6}$$

The conditions for a strategy to be evolutionarily stable are the following:

$$a > c, \text{ or } a = c \text{ and } b > d \tag{2.7}$$

If $a > c$ then X would be a strict Nash equilibrium, on the other hand if $a = c$ *and* $b > d$ the condition is not a strict but still a Nash equilibrium.From Easley 2010 [23] we can draw the following conclusions:

- *If X (any strategy s) is not a Nash equilibrium, then X is not evolutionarily stable.*
- *If X (any strategy s) is evolutionarily stable, then X is a Nash equilibrium.*
- *If X (any strategy s) is a Nash equilibrium, but $a = c$ and $b < d$ then X is not evolutionarily sta*

Hoping to provide a more understandable explanation of the previous, an example with values will be used and since evolutionary game theory is originated from evolutionary biology, as many literature that defines ESS, we will use an example where the players are animals of a certain type.

Let us assume that there is a population of wolves all of the same size, but at some point in time a mutation occurs and some bigger wolves result from this. This mutation now represents the fraction ϵ of the population, and the rest of the population (small size wolves) are represented by 1 - ϵ. These species of wolves as many others hunt in packs and given this cooperative nature the small size wolves share the prey in exactly half. Nevertheless the big sized wolves cannot afford to be so equally sharing, they need a bigger share of the prey since their body needs more nutrients, also given their size they have to make a greater effort when catching the prey. These two issues have made the big wolves aggressive to the other wolves whenever they compete for a share of the prey. The small size wolf avoids conflicts with other wolves, so if a big wolf attacks him after catching the prey they leave, but even with this situation most of the times the small size wolf still has a small share of the prey. A different scenario is given when two big wolves hunt a prey, they will fight each other until they are left seriously injured. This situation is costly for both. The expected pay-offs of these interactions are represented in the following table:

	Wolf size	
	Small	Big
Small	5, 5	1, 7
Big	7, 1	2, 2

Table 2.6: Wolves Hunting

Now we will determine if the small size wolf is an evolutionarily stable strategy, for this we will use the pay-off formulas previously defined, and we have that in this population of wolves, the small size have a pay-off of:

$$(1 - \epsilon)5 + 1\epsilon = 5 - 4\epsilon \tag{2.8}$$

While the population of bigger wolves have the following expected pay-off:

$$(1 - \epsilon)7 + 2\epsilon = 7 - 5\epsilon \tag{2.9}$$

For small values of ϵ we see that the expected pay-off of the big wolves is higher than the small wolves pay-off. This means that small wolves are not evolutionarily stable. We can also determine if the big wolves are evolutionarily stable. We now assume that in a population of big wolves, some mutation of small wolves appear in the population in a

fraction equal to ϵ. Therefore the population of big wolves is now 1 - ϵ. Now we determine the expected pay-off for big size wolves in this kind of population:

$$(1-\epsilon)2 + 7\epsilon = 2 + 5\epsilon \tag{2.10}$$

And in this population, the mutated small sized wolves will have an expected pay-off of:

$$(1-\epsilon)1 + 5\epsilon = 1 + 4\epsilon \tag{2.11}$$

In this population, we can see that the expected pay-off of the big sized wolves is greater than the one for small size wolves. This means that the big size wolf population is evolutionarily stable and a strict Nash Equilibrium.

2.3 Genetic algorithm and evolutionary game theory

Algorithms inspired in evolution for optimization and machine learning, have been the subject of interest for many. After the second world war von Neumann showed great interest in artificial intelligence, his main studies were focused and can be said that originated what now known as cellular automata [25]. The mathematician Nils Barricelli wrote simulations of evolutionary processes, in the interest of creating artificial life in computers using nature processes. He beccame the first to write a genetic algorithm software, and his first work about it was published in 1954 [26]. Another person who would use computer programs to simulate evolution was the biologist Alexander Fraser. Fraser was interested in evolution, but in the real world evolution happened to slowly for this reason he decided to use simulations created in digital computers. Fraser wrote in 1957 "Simulation of genetic systems by automatic digital computers" and with these he became the first person to use computer simulations with the purpose of simulation biological evolution [26]. Other scientists also used computers to simulate evolutionary processes like Han-Joachim Bremermann his first paper about it was a published in 1958 entitled " The evolution of intelligence". George Box, was another scientist who worked in the field. Interesting about Box is that his main interest was not artificial life or evolution, his interest was more in solving real life problems. The first paper he published that can be related to genetic algorithm was in 1957 and the title was "Evolutionary operation: A method for increasing industrial productivity"[26]. Box states that living things advance in means of two mechanisms: 1. Genetic variability and 2. Natural selection [27], and in raw terms he associates a new discovery of a process to a mutation and using or discarding different less efficient processes as natural selection. George Friedman outlined in 1956 how a "selective feedback computer" might evolve electrical circuits to maintain internal temperature of a mobile robot [25], this work being similar to nowadays genetic algorithm [26].In 1966 Lawrence Fogel along with Alvin Owens and Michael Walsh wrote a book about genetic algorithms with the title "Artificial Intelligence through Simulated evolution" [26]. In 1960s and 1970s James Holland with students and colleagues from University of Michigan developed the genetic algorithm [28]. Holland's goal was to study the phenomenon of adaptation of processes in natural systems and develop ways in which

the mechanics of adaptation might be imported to computer systems. Holland introduced a population-based algorithm with crossover, inversion and mutation [28].

Genetic algorithm is a type of evolutionary algorithm. In general terms genetic algorithm can be seen as an imitation of biological evolution for problem solving. A genetic algorithm combines the survival of the fittest with structured yet randomized information exchange. Genetic algorithms are no simple random walk, they exploit historical information to speculate on new search points expecting improved performance. The main research in genetic algorithms revolves around robustness, the balance between eficiency and efficacy for survival in different environments [27] Genetic algorithms are blind, they perform an effective search only requiring payoff values associated with individual strings [27]. Melanie 1999 mentions that although there no widely accepted definition for genetic algorithm in the evolutionary - computational community, most methods self called "genetic algorithm" have in common the use of population of chromosomes, selection according to fitness, crossover to produce new offsprings, and random mutation of the new offspring.

2.4 Agent-based modelling (ABM) and object oriented programming (OOP)

When building a computer simulations focused in social studies there are two concepts that can be helpful to make the simulation process easier to build: Agent-based modeling (ABM) and object oriented programming(OOP). Agent-based modeling is a good approach to have a concept of how to represent the elements that are in someway the subjects of study and object oriented programming (OOP) can help to represent agents with there individual characteristics, their environment and any other factors that have an impact in the intended study, given its versatile style for variables manipulation.

Agent-Based Modelling (ABM)

Generally speaking in ABM agents are entities created to represent entities of the real world, and make them interact to study their behavior. One of the first known scientists to show interest in the concept of entities was John von Neumann [29]. He thought in the future it would be useful to have artificial machines that could reproduce autonomously, in order to represent objects such as celestial objects [29]. Given to the suggestion of his colleague Stanislaw Ulam, von Neumann developed a simple model of cellular automaton. In which each cell take one of multiple states, and then the changes in it are based on the history of previous states from the cell and neighboring cells [30].

In 1970 Martin Gardner published a game by the british mathematician John Conway. Which was a simplified model of cellular automata applied into what he called Game of Life [30]. In which an infinite universe is divided into cells, each cell interact with the neighboring cells (8 cells around it), according to a set of established rules creating complicated patterns according to simple states of the cells, which could be in one of

two states (alive or dead) [31]. In 1973 in a joint paper published in 'Nature' (Nature Publishing Group) by Professor Maynard Smith and George R. Price describe the results of simulations made, which can be considered an example of an agent-based model [32].

In 1971 the economist Thomas Schelling used a chessboard in which by moving pennies and dimes he represented what he described in his work "Models of segregation" [30]. Finally another important work that used agents were the simulations made by the political scientist Robert Axelrod in 1984 [30], in which he asked game theorists from different disciplines to submit a strategy each, these strategies were used for simulating a tournament of a type of game called 'prisoners dilemma'. Each strategy interacted with the other strategies, during a number repetitions and for each repetition a determined number of rounds [33] the objective was to see if 'cooperative' strategies can evolve in an environment where 'selfish' strategies exist.

The previously mentioned works are some of the most relevant that involve this agent based modeling. Agent-based models (ABM) are being increasingly used to model complex systems [34]. Robert Axelrod in his book 'The complexity of Cooperation defines ABM as the simulation of agents and their interactions. This type of modeling is different from the traditional type, and it is a type of simulation that is viewed as 'bottom-up [35] used for understanding properties of social systems [35], by representing complex behaviours in the hopes of emulating some specific aspect. There are 3 ways for doing science according to Axelrod, which are induction, deduction and agent-based modeling, the latter starts like deduction with a set of explicit assumptions then it generates simulated data that can be analized inductively. And unlike induction the data produced comes from specific rules rather than real world data [35]. Joshua Epstein points ABM as "a new tool for empirical research" [36]. So it is very important to remember that our final goal with ABM is not to recreate reality, its rather for demonstrating a principal and understand how it works. There are at least three main components to take in account when building an ABM, the number of agents with characteristic variables (agents contain data together with methods which act of the data), a set of rules forthe interaction and the environment in which the interaction takes place [37] this will be described in further in the description of the simulation. In agent based modeling what matters is not how decisions are made but how the interaction is given [37].

A very important step when building a simulation is to choose a programming language. Axelrod mentions that a good agent based model should achieve three goals: validity this is for the program to implement the model correctly, usability which involves the fact that the person building the model and other users can run the program, interpret the output and understand how it works, and the third point is extendability which means the program should be adaptable for new uses in the future [35]. For the type of simulation that was built for this project, object oriented programming (OOP) is appropriate. In the recent years this language has gained popularity, but it can be traced for a long way back. Before the concept of OOP was well established, all computing languages were procedural languages, which means that it looked like the program was all contained in one long procedure all data and the logic were presented in a long code.

Object Oriented Programming (OOP)

OOP's concern is on 'objects' [38] this objects can be anything one can think of i.e. an action, a thing, a person, a place, of course they have to be well defined. Object is the basic element. Objects possess attributes of procedures and data. Storing the data in variables and responds to messages by executing procedures (in Python called methods). The program is divided in individual objects (modules) each one can be viewed as an abstract data type. Each of the objects contain their own methods and data [38]. Communication between objects requesting action can be called message. The purpose of this is that each object represent parts of whole program, but by breaking it down in modules each can be thought of as a particular action which we can relate in an easier how ideas and actions are structured in our everyday life. As an example we can think of an apple which would be our object and this apple contains attributes such as color, size, weight, etc. but it can also contain methods such as growing, changing color, falling from a tree, etc. This way of conceptualizing ideas in such conventional way in relation to our everyday life make it a perfect candidate for using in simulations. In this project OOP would simplify structuring the ABM, in which it was mentioned before we have so very defined concepts (agents, rules and environments).

To be able to work with OOP there is an important concept to understand, the difference between class and object. A class can be thought of as a general description or blueprints of something but is not the thing itself. What the class is defining are abstract ideas of what was mentioned before, 'attributes and 'methods, formally a class is defined as a template from which objects are created [39]. The next concept is object, would be the "tangible" instance of the initial template which is the class[40]. To understand this correctly we can think of the class as the blueprints of something we want to build, and the object is that thing we wanted to build from the blueprints. When creating an object the abstract ideas from the class, become specific characteristics of the object. A relevant property is inheritance, which allows a class to inherit methods and attributes from another class, this makes the class that inherits a subclass of the class it inherits from, its important to mention that this subclass can redefine inherited methods and can add methods that can differentiate it from the class it inherited from [39]. Some other relevant properties from OOP are dynamic binding, which is not exclusive for OOP, means that the binding of operator to a particular operation takes place at the run time. Encapsulation is another and it describes the scope of unrestricted reference to the attributes of an object. An object can examine and modify its own attributes, and allows access to its attributes to other objects through accessing functions allowing it to have control over any changes requested from other variables. Data abstraction refers to how any object can be required for any information, and the fact that who requests gets what he/she asked for [38].

With the created code when running the simulation individual agents with individual characteristics are created (for this project the characteristics are that they can posses a strategy and everytime they intreract they are able to accumulate an individual utility), and they are set to interact in a certain way under determined rules, the final goal will be to have a general perspective of how the interaction of these individual objects leads

to a resulting conclusion. In this case we will be able to observe how agents represent different strategies and what is the result of the interaction which will be how one strategy dominates another or how they interact and find a balance. The language chosen for writing the code is Python which is a high level programming language, designed by Guido van Rossum. It serves general programming purposes and it is a very effective object oriented language.

2.5 Genetic algorithm considerations for the project.

For this project the code that was developed does not use all the features described for a genetic algorithm. The characteristics for the algorithm used resemble more those of evolutionary game theory. Where the agents are created at the beginning of the simulation and are assigned with a specific strategy which they will continue to use until they are eliminated or until the simulation ends. For this the agents created do not posses a chain of phenotypes from which they will occasionally choose and play a different strategy. In our simulation the strategy is assigned randomly and this assures that at least every possible strategy have representatives during the simulation. Additionally we use the concept of asexual reproduction, which means there is no two parents for each "newborn" agent, but instead it is copy of one of the individuals that had the higher utility(fitness) during the generation (or other possible choices under a criteria described later). For convenience a constant population number is maintained during the whole simulation, so the same number of agents that will be eliminated for not being efficient, will be the same number of agents that will be created for the next step of the simulation. The concept of mutation allows us to not always select the individual with the highest payoff to reproduce, but introduces the possibility of another agent with a less efficient strategy to appear. In addition to this concept of mutation. The use a variable called exploitation rate is introduced mainly because of the article by James March "Exploration and Exploitation in Organizational Learning" in which he names two concepts that can or should be taken in account when making a choice in the organizational aspect. But these concepts were considered valid for the purpose of our simulations, because the simulations made by the code are random, and though randomness should help to cope with any potential bias, the random interactions may favour one strategy over another potentially good potential strategy. March 1991 mentions that exploration includes things captured by terms such as search, variation, risk taking, experimentation, play, flexibility, discovery and innovation: whilst exploitation includes refinement, choice, production, efficiency, selection, implementation and execution[41]. In raw terms exploration implies looking for another feasible alternative, in the case of our code we can do this in any percentage of the population, or choosing the alternative that appears to generate the highest utility in every run. In the code it is implemented as an additional decision parameter to give more possibility to other less efficient strategies during a generation to reproduce. An exploitation rate of 1 (100%) means that only the agent with highest payoff has a possibility of reproducing itself and a 0 (0%) means that the whole population is considered a candidate for reproducing itself for the next generation, and this decision is let to be handled by a random choice function. For the purpose of this project, we

will be focusing on the interaction between the same species. The model we will use is simplified, but yet it contains sufficient information to display the expected results. And to confirm this after some of the simulations for games, Nash equilibria and ESS will be calculated.

Chapter 3

Software development practices

In this section a description on how the code was built will be given. The focus used to build the code was a discipline in computer programming called test driven development, a description of what it is will be given. Version control system will be explained, what is a version control system and the elemental use of the version control system Git will be explained, which is used for interaction between different people that contribute to a single project.

3.1 Test Driven Development

The idea of OOP goes well with a programming discipline test-driven development (TDD) which is being used more commonly. TDD is very similar to the concept of test first development (TDF) described by Kent Beck in his book "Extreme programming", and the main idea is to build a test to assert that the result from a small piece of the application code will be reached, this is very likely to make the programmer think of what is the result she wishes to obtain and very likely as consequence build a very efficient code. Usually a fundamental step of TDF is described as "write a test that will fail", then under that idea build the functional part of the code that will make that test pass. Test driven development (TDD) follows this same idea, but adds an additional ingredient which is refactoring. In the book "Test-driven development: an example" Kent talks about a TDD "mantra" which is described in 3 simple steps, the first two are the same principles as the ones of TDF. The first step he calls it "Red" and says to write a small test that doesn't work, the second step "Green" is to make the test work (by writing the functional code), and the last step "Refactor" which is getting rid of all possible redundancies in the code without affecting the functional code [42]. The testing is usually automated unit testing, the concept of unit testing is useful when using OOP, since the intention of OOP is to keep a clear and functional code with a certain degree of abstraction. Unit testing is testing each small component of a program. It is important to say that TDD is intended to test the code in small steps, and not the final application of the code. So in TDD testing and designing the code go together in small steps to produce a final simple and testable code.

Figure 3.1: Basic TDD diagram.

For testing in this code **unittest** test module contained in the standard Python library was used. **unittest** from Python supports some important concepts for testing for example test fixture, test case, test suite and test runner [43]. For the code given that each function of each module was tested we used the smallest unit for testing which is the class **TestCase**. Although there exist many methods for testing in **TestCase**, for this code the main methods used when testing were the following (taken from van Rossum 2003[43]):

- **assertEqual:** Asserts an element a is equal to an element b (a == b) if the values are not equal the test fails.
- **assertNotEqual:** Asserts an element a is not equal to an element b (a !=b) if the values are equal the test fails.
- **assertTrue:** Asserts an argument passed is True (bool(x) is True) test fails if argument is not True.
- **assertFalse:** Asserts an argument passed is False (bool(x) is False) test fails if argument is not False.
- **assertIn:** Assert an element a is contained in b (a in b) test fails if the element a is not part of b (i.e. a = 3, and b = [1, 5, 6, 8 ,10], value of a (3) is not contained in the list so the test fails).

3.2 Version Control

When teaming with other people with common goals for developing any kind of project, one of the most important elements is communication. There should be a common agreement for a communication channel so every element of the team can be updated with any changes made during the project. Developing a project in computer programming is not the exception, and from some years back computer programmers have been using

very efficient communication channels. Some very useful tool are version control systems (VCS). The main goal from a VCS is to keep track of the development of a project with a very intelligent system that allows people (programmers) to track changes in a project through time. It keeps record of changes that have been made to a project and allows to retrieve the information from those other points in time. VCS basically can create a backups for every changes made to a project, revert specific files, the whole project to a previous state, compare changes over time, see who did last modifications that could be causing a problem, who introduced an issue, when among other [44]. Version controlled systems can be local, centralized or distributed, to collaborate in a project nowadays the most popular and perhaps effective choice is to use distributed version control systems in which the users fully mirror the set of files or directories under version control commonly known as 'repository' [44]. This means that each contributor has a copy of the full repository and if the main server has a malfunction and information is lost, it can be restored by any contributor.

The VCS used for this project was Git. Git was authored and developed by the creator of Linux kernel, Linus Trovalds in 2005 and many other developers from the Linux community after a commercial, break down between the community that developed Linux kernel and the company that provided them DVCS called BitKeeper [44] .

Git thinks of data like snapshots of a miniature filesystem, to take this snapshots Git has a command that is called commit which can be used to record in different points of time. One could think of commits as some form of milestones for the individual and the collective part of the project. An important fact about Git is that since all the history of the project being developed is in each users local disk, the speed to browse around the history of the project is very efficient. This also allows the user to work even if there is no access to the network where the repository is.

For this project only basic, but still essential features of Git were used. The webpage where the repository for the project was stored is Github, a link to the code can be found in appendix F. For using Github, first one has to create a user name and then a repository in the webpage . Which is relatively easy by following the instructions given in the webpage. It essentially asks to give a name to the repository and an optional description, the repository can be public or private, for this project it was set as public. It is advisable to create a README file for describing the project. At the end of the steps one selects create repository and the repository is created. Setting up Git in the pc is not hard, for this Git bash was used which is a built environment shell for Windows and enables Windows to use the Git commands. There is a lot of information about how to set up Git bash in the web which involves generating a key, by adding a command in bash along with which I added the email address used when signing up in Github in this case and then follow the steps and it is set. To start working with a repository from Github either our own or someone else's. We copy an URL that appears on the right bar in the page from the repository we intend to work with. This URL was pasted into the Git bash in the same line after the following command **git clone** and with this a local clone of the repository is created. Once the local directory for the project is in the pc, we can start working with it. Every file that will be included in the project should be in that directory. To see the basic Git commands with a short description used for this project

go to appendix *A* Git Commands. If further information is required a link to Github can be found in appendix *A*.

3.3 Documentation

A good practice when creating a package is to put together **Documentation** about it. The **Documentation** must contain relevant information about the package and how to use it. The **Documentation** for this package can be found in: **http://ablearn.readthedocs.org**, and as it can be seen from the link it is held in a very useful webpage for sharing information and particularly documentation from packages called **http://readthedocs.org Read the Docs**. An advantage of using **ReadTheDocs** is that it can be linked to a **GitHub** repository, and it can be set to automatically update whenever new information from the repository is available. To be able to build a correct structure that could be used by **Read the Docs** a tool named **Sphinx** was used. **Sphinx** is a tool that was originally designed to help create documentation for Python, and now it can be used for other languages. It helps to create basic files to give the initial structure to the **Documentation**, by building an index which acts as the main view for the **Documentation**. After this as many number of .rst files can be created, as required to populate the **Documentation** file. It is recommended to create a directory with the name of **docs** or **doc** that will hold the files created and all the additional files the user creates to be included in the documentation. An .rst is the extension for **reStructuredText**. A **reStructuredText** is a markup syntax that is useful for doing inline program documentation, webpages designs and simple text documents [45]. The files created can contain any kind of information, and by referencing the created file in the index it automatically generates it as a **section** of the **Documentation**. The **Documentation** for this package was built using the format previously mentioned **reStructuredText** (rst), the name given to the **Documentation** directory was **docs**. After creating a basic structure, the directory was pushed to a branch in **GitHub**, and when the branch was merged with the master branch, the changes made automatically updated the **ReadTheDocs** page for the **Documentation**. The result is a nice looking and readable output that is relatively easy to understand [45].

Chapter 4

Library

4.1 Library (Package: Ablearn)

An overview of the library built for this package is given in this section, explaining how the different modules contained work. The name chosen for the package is 'ablearn' and it stands for agent-based learning, the intention is to build a package that can be used to solve different types of game theory problems and for different people to be able to contribute for the expansion of this package in the future and therefore have a wider range in application at some point in time. The library is the file that contains all the components of the code. Because of the object oriented properties that Python facilitates, the code is segmented into 6 modules.Two of these modules will not be commented on ('tests' and 'axl') information about them will be found in the DOCUMENTATION WEBPAGE and a link to it can be found in appendix F. According to their function the order in describing the remaining modules is not relevant. The word 'simulation' will be used to represent when the code is 'run', in addition to the meaning it will have when describing the 'Simulation module'.

When building a package in a modular form, a main directory for the package has to be created. Inside this directory subdirectories for each module have to be contained and an additional file for python to be able to execute the directory. This additional file is should be named __init__ with extension .py, and it should contain import commands for the subdirectories that the package will execute. An example of the __init__ file for 'ablearn' package can be found in appendix F. Each subdirectory also contains an __init__.py file this file is needed so that python can identify which subdirectories are required to be executed in the package. Each subdirectory as mentioned before representsa module for the package. In this project, each module has the code to create a new class, from which an instance or several instances will be created and will interact during the simulations (runs of the program).

4.1.1 Population module

The population module is intended to create agents that will be used in the simulation. The reason of being called agents instead of the classic name player, is because of the intended focus in agent-based modeling as mentioned before. It was built in a very generic way so it can be used, if possible, for any other type of interaction and algorithm in the future. The basic information contained in this class is very generic and it's attributes can be modified when using it. This module contains the instructions to create a `class Agent` the class has an initializatin(`__init__`) method that takes as parameters strategies, utility and the possibility to add a label. After the initialization, another method is presented `increment_utility` which is set for incrementing the agents utility, the criteria for this `increment_utility` will be explained in the 'Environment' module.

- `Strategies:` Each created agent will be assigned a strategy
- `Utility:` The utility each created agent generates after each interaction with another agent.
- `Label:` The possibility of adding a label to each created agent, to track their performance.

Figure 4.1: Diagram of population module.

4.1.2 Environments module

The environments module for this project is the representation of the environment in which the agents will interact.The environment created is called `bimatrix_random_environment` and has only two characteristics. It is set to make two agents interact by pairing them randomly and it also sets the rules that these paired agents use to interact. Some of the methods contained in this module make use of the random module from python. This module is set to be imported when the environment module is executed.

Bimatrix_random_environment creates a `class BiMatrixRandomEnv`, named after the characteristics of the environment of interaction which are that the data is given in the form of a bimatrix and that the agents will interact in a random way. This class has an initialization (`__init__`) method that takes as parameters `number_of_agents` and `bimatrix`, within the initialization some variables are defined, these variables along with the parameters will now be explained:

- `number_of_agents:` Input by user, total population of agents regardless of the type of agent(i.e. row agent or column agent).
- `bimatrix` Input by user, bimatrix of payoffs(can be symmetric or assymetric).
- `number_of_row_agents:` Result from dividing by 2 the previously input value number_of_agents. And gives the number of row agents.
- `number_of_col_agents:` Result from dividing by 2 the previously input value number_of_agents. And gives the number of column agents.
- `number_of_row_strategies:` Number of strategies that will be available for row agents. Calculated by counting how many rows the bimatrix has.
- `number_of_col_strategies:` Number of strategies that will be available for column agents. Calculated by counting how many columns the bimatrix has.
- `row_strategies:` List containing the available strategies for row agents.
- `col_strategies:` List containing the available strategies for column agents.

After the initialization, a method `interact` is defined. This method first defines a variable called `pairs` which is assigned to a function `randomly_pair_agents` that will be explained later. It also contains a `''for''loop` this loop within other things contains a variable which is set to a function `strategies_to_utilities` which will be explained, variables in the **"for" loop** are the following:

- `ra`: Used in the code during for loops instead of `row_agents`.
- `ca`: Used in the code during for loops instead of `col_agents`.
- `pairs`: Variable created to represent the group of paired `row_agents` with `col_agents`.
- `utility`: Variable used to obtain the utilities resulting from the interaction from each pair of agents (**row_agents and col_agents**).
- `Agent.increment_utility`: The increment utility function is defined in the population model. The structure for in this for loop is as follows:
  agent.increment_utility(utility[x]) and what it does is assign the function increment utility to an agent can be `ra (row_agent)` or`ca (col_agent)`, the `utility` in parenthesis was assigned in the previous variable, and it is only calling the value with the position x in the list. Given that we only have 2 types of players (we are using a bimatrix) x can be either 0 or 1.

Following `interact` the method previously mentioned `strategies_to_utilities` is defined. This method is in charge of obtaining the specific pair of utilities (assigned to row and

column) from the `bimatrix`. It then returns these values to the `utility` variable in the **`interact`** method and interact uses it to assign the utilities to each agent.

After **`strategies_to_utilities`** the method **`randomly_pair_agents`** is defined. This method is used by **`interact`** too, and what it does is that the previously created row and column agents that are contained in lists are randomly selected (one of each type) and then paired so they can interact. The following diagram shows in general terms the structure of the bimatrix random environment module:

Figure 4.2: Diagram of environments module.

4.1.3 Algorithms module

The **algorithms module** is a module that is meant to contain different algorithms by which the created **agents** from **population module** will be processed. For this project **algorithms module** contains a **genetic algorithm**. The **genetic algorithm** is named **genetic** and it creates a `class Genetic`. In the initialization (`__init__`) takes the following parameters:

- `generations`: Are the number of generations (times) the whole code will be run.
- `rounds_per_generation`: Are the number of times the interaction of the agents will be given within each generation.
- `death_rate`: Represents the proportion of agents that will be eliminated after each generation has run.
- `mutation_rate`: This parameter is given as a limit for agents with the highest accumulated utility not being reproduced. A `mutation_rate` value is established and is compared against a `random.random()` number. Every time the `mutation_rate` is higher than the value given by the `random.random()` a mutation will occur. This comparison occurs every time a new agent is created when all the interactions within a generation have happened. This means that each new agent is passed under this condition. The numbers given by `random.random()` are decimal numbers, which means that the range for `mutation_rate` should also be in decimal numbers from **0 to 1**.
- `exploitation_rate`: Is a variable introduced to determine if the concept of *exploitation or exploratio* will be used. Where a value of 1 (100%) represents a total exploitation, this means reproducing only the agent with the highest utility for each type of agent (`row_agents` and `col_agents`) and a value of 0 (0%) is used to consider the whole population of agents.
- `initial_strategy_distribution`: Is an optional parameter that will allow to the user to introduce the initial proportions of the strategies, meaning what percentage of the population each strategy will be represented by. The distribution will be introduced as a form of a list within '[ ]' with commas separating the different values. The proportions that are input have to add up to 1 since the number 1 represents 100% of the population i.e. [0.7, 0.3] added up are equal to 1, this means that the strategy with the first position in the bimatrix will start will be represented by 70% of the total population of the created agents and the strategy in the second position will be represented by the remaining 30% of the created agents in the population at the beginning of the simulation. If a particular `initial_strategy_distribution` is not needed, the 'list' with the proportions does not need to be introduced and the program will automatically set the distribution in almost equal proportions according to the number of strategies.

Following the initialization (`__init__`), the method `assign_strategies` is found. This method takes parameters **agents**, which according to the type of agent in turn will be represented by `row_agents` or `col_agents` and the parameter `agents_strategies`, also which according to the type of agent in turn will be represented by `row_strategies` or `col_strategies`. This

method is used after the different types of agents are created in the `initialization` of the **`class BiMatrixRandomEnv`** of the module `environments`. The **`assign_strategies`** method assigns a random strategy for each agent according to the strategies available given their type through a **"for" loop** and **`random.choice`**`(agents_strategies)` this random choice selects from the `row_strategies` or `col_strategies` randomly according to the type of agent.

After the (**`assign_strategies`**) method in the code, the method alternative method in case a specific initial distribution is required **`assign_strategies`**`_initial_distribution` is found. This method takes parameters **agents**, which according to the type of agent in turn will be represented by `row_agents` or `col_agents`, the parameter `agents_strategies`, also which according to the type of agent in turn will be represented by `row_strategies` or `col_strategies`, the parameter `number_of_agents` which is the total number of agents in the simulation, and the parameter `agent_initial_distribution` which is a list that holds the proportions in which the strategies are required to appear at the beginning of the simulation. This method has the same purpose as **`assign_strategies`** with the only difference that when selecting an initial strategy distribution the program will create the exact proportion of agents strategies that is required. After creating a list of agents with strategies in the proportions required it will be found that the agents were created in order, to continue with the random environment required the method `random.shuffle` from the module `random` is used. This way the order of the agents will not be a specific when interacting with each other in the second generation.

After the **`assign_strategies`** method, comes a method named **`kill_agents`**. This method takes as parameters `agents, number_of_agents, number_of_x_agents`, and `agents_strategies`. In this method a variable `number_of_deaths_per_generation` is created, which determines the number of agents that will be eliminated for each type of agent group (`row_agents` or `col_agents`) it does so by taking the integer value from the product of the multiplication of the variables `death_rate number_of_x_agents`(this last variable represents the total number of row_agents or col_agents). The value `number_of_deaths_per_generation` is used in a **"while" loop** to eliminate the agents while the condition is not satisfied. Within the "while" loop the agents (`row_agents` or `col_agents)` are sorted according to their accumulated `utility` from smallest to highest and the one with the lowest accumulated `utility` is deleted from the list of the corresponding type of agents.

After the method **`kill_agents`** comes the method **`reproduce_agents`**. This method takes as parameters `agents, number_of_agents, number_of_x_agents` and `agents_strategies`. A variable `choice` is created by taking the integer value of (`((number_of_agents / 2)* (exploitation_rate − 1)+ 0`
this value is used within the "while" loop that follows. The "while" loop used in this method works while the existing number of row or col agents is less than the number of row or col agents created at the beginning of the simulation. Within this loop, the agents are sorted according to their accumulated `utility` values from low to high, and a **`copy.deepcopy`** function is used to create a copy of an agents chosen with **`random.choice`**`(agents[choice:])` where the variable `choice` will indicate the start of the range of agents that can be chosen within the list of agents. After this within the same loop and "if" statement is presented which condition is that if the `mutation_rate` value is greater than a **`random.random`**`()` number, the agent that will be reproduced can be any agent (row or col agent) with any

value of accumulated `utility` except for the agent (row or col agent) with the highest accumulated `utility` value of that generation. The following diagram shows the general structure of the genetic algorithm module:

Figure 4.3: Diagram of algorithms module.

4.1.4 Simulation module

The **simulation module** is a module that is makes all the other modules interact with each other. For this project **simulation module** is named `simulation` and it creates a `class Simulation`. In the initialization (`__init__`) takes the parameters used in the other modules in addition to another parameter named `tog` which stands for "type of game". There are 7 predetermined games within the code, which can be selected by typing the correct value of `tog`. `tog` and the other parameters taken by the class `Simulation` have the following characteristics:

- `tog`: Stands for "type of game" and it serves to pick one of the available choices of bimatrix that have been coded in. The values that can be selected have to go between single quotation marks and are the following: `'pd'` an example of prisoner's dilemma, `'psr'` an example of the game paper-scissors-rock, `'mp'` an example of matching pennies, `'bos'` an example of battle of sexes game, `'hd'`, `'sh'` an example of stag hunt game, `'cs'` an example of choosing sides game and `'axl'` an example emulating the first tournament from Robert Axelrod with 8 out of the 14 original strategies. This strategies are taken from the axelrod package available for python and with the following strategies: *Tit for Tat, Grofman, Shubik, Grudger, Davis, Feld, Joss, Tullock and a Rand*
- `na`: Stands for the value of `number_of_agents` explained in the `bimatrix_random_environment` module.
- `ge`: Stands for the value of `generations` explained in the `genetic` algorithm module. And is the number of generations (times) the whole code will be run.

- `rpg`: Stands for the value of `rounds_per_generation` explained in the `genetic` algorithm module. Is the number of times the interaction of the agents will be given within each generation.
- `dr`: Stands for `death_rate` explained in the `genetic` algorithm module. And represents the proportion of agents that will be eliminated after each generation has run.
- `mr`: Stands for `mutation_rate` explained in the `genetic` algorithm module. And this parameter is given as a limit for agents with the highest accumulated utility not being reproduced. A `mutation_rate` value is established and is compared against a `random.random()` number. Every time the `mutation_rate` is higher than the value given by the `random.random()` a mutation will occur. This comparison occurs every time a new agent is created when all the interactions within a generation have happened. This means that each new agent is passed under this condition. The numbers given by `random.random()` are decimal numbers, which means that the range for `mutation_rate` should also be in decimal numbers from **0 to 1**.
- **er:** Stands for `exploitation_rate` explained in the **genetic** algorithm module. And it is a variable introduced to determine if the concept of *exploitation or exploration* will be used. Where a value of 1 (100%) represents a total exploitation, this means reproducing only the agent with the highest utility for each type of agent (`row_agents` `and` `col_agents`) and a value of 0 (0%) is used to consider the whole population of agents.
- **isd:** Stands for `initial_strategy_distribution` explained in the **genetic** algorithm module. It takes a list of decimal numbers which should add up to 1, each value representing the proportion for each strategy available in the `bimatrix`. In here is where the user inputs the initial distribution desired for the strategies. At the beginning of the simulation this value will be used by the instance created from the `class Genetic` from the genetic module to assign the initial strategies to each agent.

In the initialization (`__init__`) method an instance of the `class Genetic` from the genetic module is created, an instance for the `class Agents` from the population module is created also and when selecting the `tog` value, an instance of the `class BiMatrixRandomEnv` from the environment module is created. After the instances are created within the initialization method, the functions `ga.assign_strategies()` or `ga.assign_strategies_initial_distribution()` from the genetic algorithm module is used to assign the values for the strategies for each type of agent (`row_agents` and `col_agents`) that have been created through the instance of the module environment can be used, the methods are conditioned under an if statement which checks if an initial distribution was introduced or not. After three lists with the names `row_accumulated_strategies`, `col_acummulated_strategies` and `strategies_history` are created respectively and the purpose for each list is the following:

- `row_accumulated_strategies`: Creates a list, that within itself contains a certain number of individual lists according to the length of `row_strategies` which is the number of strategies available for the `row_agents` obtained from the module environment.
- `col_accumulated_strategies`: Creates a list, that within itself contains a certain number of individual lists according to the length of `col_strategies` which is the number

of strategies available for the `col_agents` obtained from the module environment.

- `strategies_history`: Creates a list, that within itself contains a certain number of individual lists according to the length of `row_strategies` if `row_strategies` $\geq$ `col_strategies`, else it creates a number of lists according to the number of `col_strategies`. This list will help to create the stack plot, and what it does is it stores the total numbers of each type of strategy (not taking in account the type of agent) in a list that is generated each generation and all these lists created per generation are contained within a list that helps create the stack plot.

After the initialization method has created the different instances from the other modules, a method with the name `run(plot=False, stack=False)` is presented. This method is used to start running the whole simulation (all the modules). The method takes an attribute `plot` which is set by default as `False` and an attribute `stack` which is also set by default as `False`, if using the method `run(plot=True, stack=True)`, the simulation will plot the results from each generation in a graph and at the end of the simulation will produce a stack plot representing how the number of agents for each strategy behaved. The graph and stack plot are produced with the module `matplotlib` from python and the functions used are **pyplot and cm(color map)**. The graph is set in interactive mode by the function `ion( )`. The limits for the graph's y axis are set `ylim(0, 1)` and for the stack plot the y axis shows the total number of agents, the limit for the x axis for graph and stack plot are the number of `generations`. The x axis for both is labeled `xlabel(''Generations")` and the y axis is labeled `ylabel(''Probability'')` for graphs and `ylabel(''Agents'')` for stack plots. For the coloring the lines representing the different strategies for each type of agents in the graphs color map method from mathplotlib is used, for the `row_accumulated_strategies[ ]` the color map copper is used and for the `col_accumulated_strategies[ ]` the color map winter is used, how these `accumulated_strategies` values were obtained for graphs will be explained later, the thing to know about them is that each of them represent the proportion within a type of agent that is using a certain strategy. The graph indicates how the population of the different types of agents with strategies varied through time, the value of interest from this is the strategy the types of agents are using. While the stack plot does not make a distinction between types of agents and it shoes the total number of each type of strategy in the total population through the whole simulation.

After the code for creating the graph a function `generations_passing( )` is presented, this function takes attributes `generations`, `rounds_per_generation`, `number_of_agents`, `row_agents`, `col_agents`, and `plot` and takes us to the next method.

After the `run` method, there is a method `generations_passing` that takes attributes `generations`, `row_agents`, `col_agents`, `number_of_agents`, `plot`. In this method there is a "for" loop that loops until the `generations` value is reached, within this loop exist two other "for" loops. he first one loops around the `row_agents` and `col_agents` setting for each agent the `utility` value to 0. The second "for" loop loops according to the `number_of_rounds` chosen at the beginning of the simulation. Within this loop we find the function `interact( )` from the module environment, explained before, which in general terms makes the `row_agents` and `col_agents` interact with each other and as a result obtains the payoff for each type of agent and an attributed `utility`. The loop from the rounds causes this process to repeat several times and when the last round is played all the different agents have an accu-

mulated utility product of the repeated interactions. When the second loop concludes a function `distributions( )` follows. This function is for the method `distributions` that takes attributes `number_of_agents`, `row_agents`, `col_agents`, `row_strategies`, `col_strategies`, and`plot`.

Before the method `distributions` there are two methods, the first is `proportion_classified_strategies` that takes attributes `agents` and `strategies`. This method will do some calculations when the code is using the method `distributions`. The method first creates a varibale `frequency_per_strategy_per_agent` with an empty list. Then it has a "main" "for" loop that loops around the variable `strategies` (this variable can be `row_strategies` or `col_strategies` according to which values are being used) of the `agent` in turn and appends a value of 0 to the list. Then an "secondary" loop is contained within the previous loop of `strategies`, this loops around the agents in turn (can be `row_agents` or `col_agents`) and contains an "if" statement, which conditions if the strategy from the `agent` in turn is equal to the counter in the "main" loop for (`strategies`) the last value of the list `frequency_per_strategy_per_agent` will increment by 1. When both loops finish the function returns a list with all the values that are contained in the `frequency_per_strategy_per_agent`, each of those values are devided by the number agents. This values are the proportion that strategy represents in the type of agents in turn. It is important to note that the variable `agents` taken by this function can be either `row_agents` or `col_agents` and the variable `strategies` can be either `row_strategies` or `col_strategies`.

The second method before `distributions` method is `total_classified_strategies` that takes attributes `row_agents`, `row_strategies`, `col_agents` and `col_strategies`. This method will do some calculations when the code is using the method `distributions` for getting the total number of each type of strategy. The method first creates a varibale `frequency_per_strategy_for_sum` with an empty list. Then it has a "main" "for" loop that loops around the variable `row_strategies` and appends a value of 0 to the list. Then an "secondary" loop is contained within the previous loop of `row_strategies`, this loops around the `row_agents` and contains an "if" statement, which conditions if the strategy from the `row_agents` is equal to the counter in the "main" loop for (`row_strategies`) the last value of the list `frequency_per_strategy_for_sum` will increment by 1. When both loops finish the function returns a list with all the values that are contained in the `frequency_per_strategy_for_sum` and repeats the same process for `col_agents` with their corresponding `col_strategies`. This values are accumulated in strategies, so there is no distinction between the type of agents only between type of strategy regardless if it is from `row_agents` or `col_agents`. It is important to mention that the loops for both types of agents are conditioned with an 'if' statement, where if the `row_agents` have more or equal types of strategies `row_strategies` than `col_agents`the first for loop to start producing the list in the variable `frequency_per_strategy_for_sum` will be the `row_agents`' loop, else if the `col_agents` have more types of strategies `col_strategies` than `row_agents`, the `col_agents` loop will go first. This conditional statement is used in case that there is a non symmetric type of game where `col_agents` have a higher number of `col_strategies` than `row_agents` of its own type of strategies.

When the method `distributions` starts first it creates a variable `row_strategies_distribution` this variable contains a function `proportion_classified_strategies( )` that takes as at-

tributes `row_agents` and `row_strategies`. And then has a "for" loop which loops around the number of `row_strategies` and appends into the list created at the initialization `row_accumulated_strategies` the resulting value from the `row_strategies_distribution`. According to which strategy it belongs, it appends the value to the corresponding list. Then the method creates second variable which has the same process as the previous, but with `col_agents`. The variable it creates is `col_strategies_distribution` this variable contains a function `proportion_classified_strategies( )` that takes as attributes `col_agents` and `col_strategies`. Then has a "for" loop which loops around the number of `col_strategies` and appends into the list created at the initialization `col_accumulated_strategies` the resulting value from the `col_strategies_distribution`. According to which strategy it belongs, it appends the value to the corresponding list. Then a third variable `total_strategies_per_generation` is created. This variable contains a function `total_classified_strategies( )` that takes as attributes `row_agents`, `row_strategies`, `col_agents` and `col_strategies`. And then has a "for" loop which loops around the number of lists contained in `strategies_history` and appends into the list created at the initialization `strategies_history` the resulting value from the `total_strategies_per_generation`. According to which generation it belongs, it appends the value to the corresponding list. After these three variables, and the append of `row_strategies_distribution[ ]` to `row_accumulated_strategies[ ]` and `col_strategies_distribution[ ]` to `col_accumulated_strategies[ ]` is finished. The method then **prints** four lines, the first one contains the following text *"Row players' strategy distribution:"*, the second line prints the list `row_strategies_distribution`, the third line prints the text *"Column players' strategy distribution:"*, and the fourth line prints the list `col_strategies_distribution`. After these lines come two lines, each with the function **`kill_agents( )`** from the genetic algorithm module. One of which is for eliminating `row_agents` and the other for eliminating `col_agents`. Then we have other two lines, each with the function **`reproduce_agents( )`** from the genetic algorithm module. One to create the `row_agents` that were previously eliminated until reaching the required number of `row_agents` and the other to create `col_agents` until reaching the required number of `col_agents`. After the last **`reproduce_agents( )`** function, a code to update the graph is run (if it was chosen to graph at the beginning of the simulation). The whole cycle is repeated until the simulation reaches the number of `generations` that was established at the beginning. The following diagram shows the general structure of the simulation module:

Figure 4.4: Diagram of simulation module.

4.1.5 Overall interaction view

Now we will have a general overview of how an actual simulation runs. Simulation contains the messages for all the other modules and with this it gives the sequence in which the modules work basically following the steps below:

- **1.** Create agents without a strategy, and with a utility equal to 0 according to the initial number of agents and the given initial distribution. If no initial distribution is entered it will create agents in equal proportions.
- **2.** Then a random strategy is assigned according to the available strategies from the payoff matrix.
- **3.** Then the different agent strategies interact, and accumulate a utility. Each interaction is equivalent to a round, so the interactions are repeated according to the number of rounds established at the beginning of the simulation.
- **4.** The proportions of the agents present in the population are calculated to be plotted.
- **5.** Agents with low utilities are eliminated according to the death rate, since a constant size of population is desired the same number of agents that were eliminated are created. The new agents are created according to the mutation rate and exploitation rate established at the beginning of the simulation. With low mutation rate and high exploitation rate the agents with highest utility are more likely to be reproduced, this means the new agents will have the same strategy as the agents with highest utility.
- **6.** Points 2, 3, 4 and 5 describe the sequence of the code for each generation. And will be repeated according to the number of generations established at the beginning of the simulation.
- **7.** After the iteration through the number of generations has finished, the final plot describing the behaviour of the strategies and a stackplot representing the proportions of the strategies in the populations are presented.
- **8.** The simulation ends.

The following diagram represents in a graphic way how the modules interact:

Figure 4.5: Diagram interaction of modules.

Chapter 5

Application of the code

In this chapter the runs of the program will be discussed. What was consider when running a simulation. An explanation of what games were simulated, and of the results against the known techniques to indicate how the program outputs our expected results when running it with known classic game theory games. A comparison of the results given by the library 'Axelrod' from python with results from this code 'Ablearn'.

5.1 How the simulation was built.

After building the code the objective is to determine if the program gives us the expected results. It is known that an evolutionary stable strategy is a strategy that survives through time. And as it was mentioned before there are certain conditions that can be verified in order to identify and evolutionary stable strategy. The following tables represent some of the most well known examples of games in the normal form.

As said before, when using the code for simulating there are different variables that can be input which will have an impact in the resulting output. The input rates that have a greater impact are death rate, which determines how many agents will be eliminated per generation; mutation rate, which will make the most efficient strategy in accumulating utility in each generation reproduce but with different characteristics (strategy); exploitation rate, which can be modified so that only the agent with highest utility will reproduce or to take a percentage of the population to be considered for reproducing.

Perhaps the most known is the the prisoner's dilemma which was mentioned before and it basically has two Nash equilibria strategies one in which both cooperate and the other in which non cooperate.

		Prisoner 2	
		Cooperate	Defect
Prisoner1	Cooperate	3, 3	0, 5
	Defect	5, 0	1, 1

Table 5.1: Prisoner's Dilemma

Matching pennies which is a pure conflict zero-sum game in which the winner of the game takes all and the loser ends up losing her share. Tthere is no scenario where both players can agree in a strategy. Player 1 preference of choice is from tossing a coin involves getting 2 heads or 2 tails, whilst player 2 prefers having a mixed combination. The equilibrium for this game is a mixed Nash equilibrium.

		Player 2	
		Heads	Tails
Player 1	Heads	1, -1	-1, 1
	Tails	-1, 1	1, -1

Table 5.2: Matching Pennies

The battle of sexes game is a coordination game in which both agents cannot exchange information about what option out of two to choose, they both have a preferred strategy, but if they both choose their preferred strategy they have no payoff from it because they rather choose the same strategy and concur, even if it has a higher payoff for one of them than the other. In the example from the table there is a representation of preferences, and it should be assumed that it is a couple trying to decide where to go, the female agent prefers going to the opera, whilst the male agent prefers going to watch football. But it can be seen that if they both end up choosing different strategies from each other they get a payoff of 0.

		Female	
		Opera	Football
Male	Opera	1, 2	0, 0
	Football	0, 0	2, 1

Table 5.3: Battle of Sexes

And the hawk-dove game. This game is often used in evolutionary game theory. And it represents the 2 strategies an agent can choose, and the result of the interaction. There is an aggressive strategy which is the hawk strategy and a passive strategy which is the dove. When both agents choose to play hawk, the payoff they get is 0, the explanation is that since they are both aggressive, the possible payoff they could have had from the resource they are competing for is not greater than the cost they pay for playing this strategy against each other. When they both choose dove, they split in equal parts the resource and the payoff they receive is the same for both. When one plays hawk and the other dove, the agent using hawk strategy gets a higher payoff than the one using the dove strategy.

	Hawk	Dove
Hawk	0, 0	3, 1
Dovel	1, 3	2, 2

Table 5.4: Hawk-Dove

5.2 Results

In this section the python package created ('ablearn') will be used to solve the games described in the previous section. And a brief analysis over the results displayed in the form of graphs and stack plots will be done. In addition the python package 'axelrod' will be used to produce a specific output (payoff matrix) and to produce a stack plot. The created package 'ablearn' will use the output (payoff matrix) from 'axelrod' package to produce a stack plot. Both stack plots will be compared and briefly commented on.

5.2.1 Prisoner's Dilemma

The following table indicates how the simulations for prisoner's dilemma were run.

Num.	Population	Generations	Rounds per Gen.	Death Rate	Mutation Rate	Exploitation Rate
1	1000	50	50	0.1	0.1	1
2	1000	50	25	0.1	0.1	1
3	1000	50	5	0.1	0.1	1
4	1000	50	5	0.5	0.1	1
5	1000	50	5	0.9	0.1	1
6	1000	50	5	0.9	0.5	1
7	1000	50	5	0.9	0.9	1
8	1000	50	5	0.1	0.5	1
9	1000	50	5	0.1	0.9	1
10	1000	50	5	0.1	0.1	0.5
11	1000	50	5	0.5	0.1	0.5
12	1000	50	5	0.9	0.1	0.5
13	1000	50	5	0.1	0.5	0.5
14	1000	50	5	0.5	0.5	0.5
15	1000	50	5	0.9	0.5	0.5
16	1000	50	5	0.1	0.9	0.5
17	1000	50	5	0.5	0.9	0.5
18	1000	50	5	0.9	0.9	0.5
19	1000	50	5	0.1	0.1	0.1
20	1000	50	5	0.5	0.1	0.1
21	1000	50	5	0.9	0.1	0.1
22	1000	50	5	0.1	0.5	0.1
23	1000	50	5	0.5	0.5	0.1
24	1000	50	5	0.9	0.5	0.1
25	1000	50	5	0.1	0.9	0.1
26	1000	50	5	0.5	0.9	0.1
27	1000	50	5	0.9	0.9	0.1
28	1000	50	5	0.5	0.5	1
29	1000	50	5	0.5	0.9	1

Table 5.5: Configuration of simulations for prisoner's dilemma.

For Prisoner's dilemma in this project it is not considered relevant to vary the number of generations to test for any changes. This is because the interaction of the agents and their strategies is given in the level of rounds. In other words, the rounds give the proportion of strategies for each generation. A variation in the number of generations may be considered when a clear pattern in the data is not easy to identify. However, it is considered important to determine if the number of rounds make a difference in the results. Hence the simulation will be tested with different number of rounds. It will be done under the conditions 1, 2 and 3 of table 5.5.

The simulations start by assigning the following proportions to row and column agents to the strategies cooperate and defect.

	Cooperate	Defect
Row Agents	0.49	0.51
Column Agents	0.502	0.498

Table 5.6: Initial Prisoners Dilemma distribution for simulation 1.

	Cooperate	Defect
Row Agents	0.489	0.514
Column Agents	0.524	0.476

Table 5.7: Initial Prisoners Dilemma distribution for simulation 2.

	Cooperate	Defect
Row Agents	0.512	0.488
Column Agents	0.49	0.51

Table 5.8: Initial Prisoners Dilemma distribution for simulation 3.

The 3 simulations start with very similar proportions for each strategy for agent type. They all start close to 0.5 so there is not a significant difference at this stage.

By generation 5 it can be seen that the strategy 'Defect' is dominating.

	Cooperate	Defect
Row Agents	0.348	0.652
Column Agents	0.24	0.76

Table 5.9: Generation 5 Prisoners Dilemma distribution for simulation 1.

	Cooperate	Defect
Row Agents	0.25	0.75
Column Agents	0.378	0.622

Table 5.10: Generation 5 Prisoners Dilemma distribution for simulation 2.

	Cooperate	Defect
Row Agents	0.35	0.65
Column Agents	0.314	0.686

Table 5.11: Generation 5 Prisoners Dilemma distribution for simulation 3.

For simulation 2 in generation 5 it can be seen that the proportions are slightly different than first simulation. For row agents the proportion increased and is very close to the proportion of the column agents in generation 5 from the previous simulation 1 and for the column agents decreased and is very close to the proportion for row agents in the simulation 1. In simulation 3 for this generation the value for strategy 'Defect' for row agents is slightly higher than simulation 1 and slightly smaller than simulation 2. For the same strategy in column agents compared to simulation 1 it decreased slightly and to simulation 2 it increased. The differences presented between simulations do not appear significant for this generation.

By generation 10 it can be seen that the dominance of strategy 'Defect' is almost absolute in the 3 simulations.

	Cooperate	Defect
Row Agents	0.054	0.946
Column Agents	0.074	0.926

Table 5.12: Generation 10 Prisoners Dilemma distribution for simulation 1.

	Cooperate	Defect
Row Agents	0.018	0.982
Column Agents	0.094	0.906

Table 5.13: Generation 10 Prisoners Dilemma distribution for simulation 2.

	Cooperate	Defect
Row Agents	0.082	0.918
Column Agents	0.07	0.93

Table 5.14: Generation 10 Prisoners Dilemma distribution for simulation 3.

In simulation 2 for generation 10 with respect to simulation 1, a small increase in the strategy 'Defect' for row agents, and a small decrease in the same strategy for column agents. For simulation 3 strategy 'Defect' for row agents is slightly decreased with respect to the value in simulation 1 and 2. For the same strategy in column agents it increased slightly when compared to simulation 1 and 2. In this generation for the 3 different simulations, the value for strategy 'Defect' are above 0.9, but again there is no obvious pattern so the variations may be given by the effect of the randomized variables for each simulation.

In generation 25 it can be seen that the condition persists and the strategy 'Defect' for both types of agents dominates in frequency.

	Cooperate	Defect
Row Agents	0.068	0.932
Column Agents	0.054	0.946

Table 5.15: Generation 25 Prisoners Dilemma distribution for simulation 1.

	Cooperate	Defect
Row Agents	0.045	0.955
Column Agents	0.047	0.953

Table 5.16: Generation 25 Prisoners Dilemma distribution for simulation 2.

	Cooperate	Defect
Row Agents	0.072	0.928
Column Agents	0.054	0.972

Table 5.17: Generation 25 Prisoners Dilemma distribution for this simulation 3.

In simulation 2 for generation 25, the proportions increased slightly for strategy defect for both agents in relation to simulation 1. For simulation 3 the proportions are slightly smaller for the strategy 'Defect' for row agents than in simulations 1 and 2. For same strategy in column agents is slightly higher than values from simulations 1 and 2. Still all the values for strategy 'Defect' are above 0.9.

For generation 50, the last generation we see that strategy 'Defect' dominates the strategy 'Cooperate' for all simulations.The graphs from the 3 different simulations are presented.

	Cooperate	Defect
Row Agents	0.01	0.99
Column Agents	0.044	0.956

Table 5.18: Generation 50 Prisoners Dilemma distribution for simulation 1.

	Cooperate	Defect
Row Agents	0.024	0.976
Column Agents	0.08	0.92

Table 5.19: Generation 50 Prisoners Dilemma distribution for simulation 2.

	Cooperate	Defect
Row Agents	0.02	0.98
Column Agents	0.028	0.972

Table 5.20: Generation 50 Prisoners Dilemma distribution for simulation 3.

(a) Prisoner's dilemma simulation 1, 50 rounds.

(b) Prisoner's Dilemma simulation 2, 25 rounds.

(c) Prisoner's Dilemma simulation 3, 5 rounds.

Figure 5.1: Prisoner's dilemma simulations 1, 2 and 3.

In simulation 2 for this generation, both proportions decreased slightly. Since the changes in the proportions are very small for this project it is not consider to be a significant difference between using 50 or 25 rounds. For simulation 3 the value for row agent in strategy 'Defect' is slightly higher compared to simulation 2 and smaller from the one in simulation 1. For column agents for the same strategy the value is higher in comparison to the values from simulations 1 and 2. After the 3 runs for the simulations, it can be seen in the graphs from each simulation that they all have a similar pattern. And after analysing the different specific points (generations), the variations between them do not appear to be caused by the number of rounds. The strategies through the generation have a similar behaviour, the strategy 'Defect' for row and column agents dominate the strategy 'Cooperate'. The strategy 'Cooperate' in the 3 different simulations starts declining from as early as the second generation, and around generation 10 it reaches the lowest values continuing to oscillate between 0.02 and 0.06 for the rest of the simulation. Again it is important to mention that the length of the simulations in terms of generations do not appear to be very relevant, since we can see that the behaviour for both strategies is very similar from generation 10 to generation 50 (when the simulation ends). For this reason in prisoner's dilemma simulations the configuration with 5 rounds per generation will used. When there is a small increment in the 'Cooperate' or 'Defect' strategy, this is given to the mutation rate, that is allowing more of either to appear, but since it is very low (0.1) it does not change the behaviour significantly. The following graphs in figure 5.2 show the behaviour of the the proportion of the strategies without mutation rate.

(a) 50 rounds (b) 25 rounds (c) 5 rounds

Figure 5.2: Prisoner's dilemma with mutation rate equal to 0

From the previous figure 5.2 it can be seen that no variations occur during the simulations. This is because mutation rate allows the less effective strategy 'Cooperate' to be present throughout the generations, and without mutation they will just disappear as early as the 5th generation.

Let us continue to try the different combinations of values that are shown in the table 5.5. Simulations 3, 4 and 5 have constant values for mutation rate with 0.1 and exploitation rate of 1. It is important to note that in these first 3 simulations the value that varies is death rate, and that the smaller the death rate, the gap between strategies 'Defect' and 'Cooperate' is larger as can be seen in the graphs from figure 5.3:

(a) Simulation 3 (b) Simulation 4 (c) Simulation 5

Figure 5.3: Prisoner's dilemma simulations 3, 4 and 5, with 5 rounds per generation

The following graphs from simulations 5, 6 and 7 have a high death rate of 0.9, this means that it allows the creation off 90% of new agents each generation while eliminating the least efficient. These three graphs have mutation rates of 0.1, 0.5 and 0.9 respectively with an exploitation rate of 1. And the behaviour can be seen in figure 5.4:

(a) Simulation 5 (b) Simulation 6 (c) Simulation 7

Figure 5.4: Prisoner's dilemma simulations 5, 6 and 7, with 5 rounds per generation

It can be seen from the previous figure 5.4 that when the mutation rate is lower the variation for each strategy is higher, even if the proportions are close to 0.5. There are evident high peaks and low valleys for both strategies. In the three simulations the strategy 'Defect' dominates in general, but in simulation 5 for generations 5, 7, 17, 23, 27, 34, 36 and 48, the strategy 'Cooperate' briefly dominates 'Defect' in fact it only dominates in this generations in the preceding and next generation the values are lower. This brief dominance from 'Cooperate' may be given to a higher interaction between this type of strategy between each other, in consequence they accumulate a higher utility which allows them to reproduce in the following generation. Because of the high death rate they reproduce quickly. But this means that in the following generation there is a higher probability for 'Defect' to encounter with 'Cooperate', so 'Defect' accumulates more utility and 'Cooperate' quickly dies out. In the other 2 simulations we see how as the mutation level increases the proportion of the strategies become more stable. By looking at the graphs 1, 8 and 9 have a similar behaviour where as the mutation rate increases, less peaks and valleys can be seen.

For the following graphs resulting from simulations 10, 11 and 12 an death rate is increasing in 0.1, 0.5 and 0.9 respectively. For all 3 simulations there are constant values for mutation rate of 0.1 and exploitation rate of 0.5. As the death rate increases, as expected the gap between the number of strategies representing 'Defect' and 'Cooperate' reduces. But it can also be seen that as the as death rate increases given the value of 0.5 for exploitation rate the peaks and valleys increase, they do not get to the point where 'Cooperate' dominates 'Defect', but it is important to note that with a low mutation rate and a lower exploitation rate than 1 which in these cases is 0.5 strategies that earn lower utility during the interaction are allowed to reproduce in the same probability as the one with highest utility.

(a) Simulation 10 (b) Simulation 11 (c) Simulation 12

Figure 5.5: Prisoner's dilemma simulations 10, 11 and 12, with 5 rounds per generation

When comparing these graphs with the graphs from simulation 3, 4 and 5 which have an increasing value of death rate and a mutation rate of 0.1 they behave in a similar way. But when comparing graphs from simulation 5 and 12 with each other. They both have a death rate of 0.9 and a mutation rate of 0.1, but a different exploitation rate value of 1 and 0.5 respectively. We can see that since a lower exploitation rate makes other strategies reproduce with the same probability as the strategy with the highest accumulated utility from the generation, a larger gap between strategies is present. Meaning that even if 'Cooperate' accumulates a high utility, if there is 'Defect' strategy within the 50% of the population that is allowed to reproduce, it has a possibility to reproduce. This 'randomization' gives an equilibrium between the presence of strategies in the population. Simulations 19, 20 and 21 have the same values as these other groups of simulations, but an exploitation rate of 0.1, which makes the graph from simulation 21 behave very similar to the graph from simulation 12.

For the following graphs representing simulations 16, 17 and 18 the value that changes is death rate 0.1, 0.5 and 0.9 respectively. With constant values of mutation rate 0.9 and exploitation rate 0.5. As seen before the effect of the death rate reflects with a reducing gap between the type of strategies as death rate increases.

(a) Simulation 16 (b) Simulation 17 (c) Simulation 18

Figure 5.6: Prisoner's dilemma simulations 16, 17 and 18, with 5 rounds per generation

A particular characteristic can be seen in the graph for simulation 16, it reaches the lowest value for 'Cooperate' strategy later than other simulations with the same death rate. Graphs from simulations 3, 8, 9, 10, 13, 19 and 22 they all reach the lowest value for 'Cooperate' around generation 10. Simulation 16 reaches it closer to generation 20,

we see a similar behaviour for simulation 25. What this two simulations have in common other than death rate value, is their mutation rate is 0.9. Since mutation rate allows any strategy to reproduce except for the agent with strategy with that accumulated the highest utility, this delays the 'Cooperate' strategy for reaching its lowest value. The remaining simulations, behave in a similar way to the one that have been discussed so far. It can be seen that the variable that acts as an enabler for the other variables to have an effect is the death rate.

Nash Equilibrium

As calculated before in this project. The Nash Equilibrium for the prisoner's dilemma game can be found by looking at the normal form table and select the best responses for each player. The results from the simulations give the strategy "Defect" as stable and this is also a Nash equilibrium as seen in the following normal form table for the game.

		Player 2	
		Cooperate	Defect
Player 1	Cooperate	3, 3	0, 5
	Defect	5, 0	$\underline{1}$, $\underline{1}$

Table 5.21: Prisoner's dilemma Nash equilibrium with best responses.

ESS

Now its time to find the ESS for the prisoner's dilemma game. Earlier it was explained how to calculate the evolutionarily stable strategy (ESS) when a mutant strategy appears in the population in the proportion ϵ. It will be assumed the strategy 'Cooperate' is the population and the invader is the strategy 'Defect'. Thus the payoff for the strategy 'Cooperate' is the following:

$$(1-\epsilon)3 + 0\epsilon = 3 - 3\epsilon$$

And the payoff for the strategy 'Defect' is:

$$(1-\epsilon)5 + 1\epsilon = 5 - 4\epsilon$$

The strategy 'Defect' appears to be greater than 'Cooperate'. Now since the simulation gives almost equal starting proportions to each strategy it be will assumed that the 'invading' strategy 'Defect' appears in a proportion of ϵ=0.5. And results in the following:

$$3 - 3(0.5) = 1.5$$

And for the strategy 'Defect' is:

$$5 - 4(0.5) = 3$$

The payoff for 'Defect' is greater than the payoff for 'Cooperate' thus it can be said that 'Cooperate' is not an evolutionarily stable strategy. Its time to check if the strategy 'Defect' is evolutionarily stable. And therefore it is assumed that 'Cooperate is the invader. It appears that 'Defect' might be evolutionarily stable given the previous test. But it will be calculated anyway. First calculating the expected payoff for the strategy 'Defect':

$$(1-\epsilon)1 + 5\epsilon = 1 + 4\epsilon$$

And the payoff for the strategy 'Defect' is:

$$(1-\epsilon)0 + 3\epsilon = 3\epsilon$$

Again it appears that the strategy 'Defect' has a greater expected payoff than 'Cooperate'. it will be assumed that the 'invading' strategy 'Cooperate' appears in a proportion of ϵ=0.5. And the calculation for the strategy 'Defect' is as follows:

$$1 + 4(0.5) = 3$$

And for the strategy 'Cooperate' is:

$$3(0.5) = 1.5$$

Again the strategy'Defect' has a higher expected payoff, so it is an evolutionarily stable strategy (ESS). With this it can be noticed that the strategy 'Defect' is both a Nash equilibrium and an ESS. And this supports the results from the simulation.

5.2.2 Matching Pennies

For the simulations in matching pennies, generations with 5, 25 and 100 rounds per generation were made. The table for the configurations is similar to the one used in Prisoner's dilemma simulations, and it can be found in appendix B .

For simulations 1, 2 and 3 with 5, 25 and 100 rounds per generation respectively. It can be seen that as the rounds per generation increase the amplitude of the wave grows. This means that with more rounds per generations each strategy has a greater chance of reaching a higher number in population, by reproducing when accumulating a high utility. This is also given to the low death rate, the low mutation rate and high exploitation rate which allows the most efficient strategy in the generation to reproduce almost every change of generation. But also when a population of a certain strategy increases in this kind of game, it is likely that a strategy that is earns high payoff when interacting with the strategy with high population will grow in population too given that it is more likely to encounter its counterpart and this cycle repeats with all the available strategies and we can see a cycle.

(a) Simulation 1 (b) Simulation 2 (c) Simulation 3

Figure 5.7: Simulation 1, 2 and 3

The behaviour for the graphs for each simulation is similar with an increasing amplitude until simulation 10 for the three different number of rounds. For simulation 10 the amplitude increases when having a lower exploitation rate in combination with low death rate (0.1) and low mutation rate (0.1) in a very similar way for the different number of rounds. This behaviour is because the exploitation rate allows the 50% of the population to reproduce with the same probability. So almost any strategy can reproduce unless it has the lowest accumulated utility, which means they will be eliminated. And the low death rate allows them to accumulate in a greater number.

(a) 5 rounds (b) 25 rounds (c) 100 rounds

Figure 5.8: Simulation 10 with 5, 25 and 100 rounds per generation

In simulation 12 the death rate of 0.9 increases the frequency in oscillations when combined with the 0.5 exploitation rate, all with a very similar amplitude. Which means that the population of certain strategy increases fast and is eliminated fast given the high death rate, and the broad possibilities of different agents reproducing with the exploitation rate of 0.5.

(a) 5 rounds

(b) 25 rounds

(c) 100 rounds

Figure 5.9: Simulation 12

In simulation 13, 14 and 15 there is an interesting behaviour when an increasing death rate of 0.1, 0.5 and 0.9 respectively is combined with the constant values of 0.5 for mutation rate and 0.5 for exploitation rate.

(a) Simulation 13

(b) Simulation 14

(c) Simulation 15

Figure 5.10: Simulation 13, 14 and 15 with 5 rounds per generation

(a) Simulation 13

(b) Simulation 14

(c) Simulation 15

Figure 5.11: Simulation 13, 14 and 15 with 25 rounds per generation

(a) Simulation 13 (b) Simulation 14 (c) Simulation 15

Figure 5.12: Simulation 13, 14 and 15 with 100 rounds per generation

An increasing amplitude with the different increasing number of rounds can be seen. Nevertheless in simulation 15 with death rate of 0.9 the amplitude for all of them decreases. This is given to the mutation rate of 0.5 in combination with 0.5 exploitation rate, these values make the creation of new agents with a high possibility of randomness, so the agent with the highest accumulated utility has same less probability of reproducing than others because the mutation rate gives opportunity to any agent strategy except for that with the highest accumulated payoff. And a similar behaviour is expected when death rate is 0.9 with mutation rate of 0.9, which will practically rule out the possibility of the best agent strategy with the highest accumulated utility to reproduce.

Simulation 16, 17 and 18 have constant values of mutation rate for 0.9 and exploitation rate of 0.5 with increasing death rate of 0.1, 0.5 and 0.9 respectively. For 16 and 17 with increasing death rate it can be see when comparing with 5, 25 and 100 rounds per generation, the amplitude increases. Something interesting with the high value of 0.9 in death rate that they virtually have the same amplitude.

(a) Simulation 16 (b) Simulation 17 (c) Simulation 18

Figure 5.13: Simulation 16, 17 and 18 with 5 rounds per generation

(a) Simulation 16 (b) Simulation 17 (c) Simulation 18

Figure 5.14: Simulation 16, 17 and 18 with 25 rounds per generation

(a) Simulation 16 (b) Simulation 17 (c) Simulation 18

Figure 5.15: Simulation 16, 17 and 18 with 100 rounds per generation

As expected with a high mutation rate (0.9) and high death rate (0.9) the amplitude is smaller, as seen in graphs from simulation 15 and in graphs from simulation 18.

For simulations 19, 20 and 21 we have a low constant value of 0.1 mutation rate and 0.1 exploitation rate, and increasing death rate 0.1, 0.5 and 0.9. The amplitude behaves the same with 5, 25 and 100 rounds, and the frequency increases for all the cases with the increasing death rate.

(a) Simulation 19 (b) Simulation 20 (c) Simulation 21

Figure 5.16: Simulation 19, 20 and 21 with 5 rounds per generation

(a) Simulation 19 (b) Simulation 20 (c) Simulation 21

Figure 5.17: Simulation 19, 20 and 21 with 25 rounds per generation

(a) Simulation 19 (b) Simulation 20 (c) Simulation 21

Figure 5.18: Simulation 19, 20 and 21 with 100 rounds per generation

This behaviour is given to the fact that the mutation rate is low, and the low exploitation (0.1) rate allows any strategy to reproduce with the same probability including the agent strategy with the highest accumulated utility, and it makes the selection is not as random than when also having a high mutation rate, but still many agents are candidates to reproduce. So the high frequency in the graphs with high death rate is present because a high number of agents are allowed to reproduce each generation and also the same number are eliminated. So the brief dominance of an agent strategy is given to the random factor. Only for to see how the strategies look in proportion in the population when the frequency increases, the stack plots from simulations 19, 20 and 21 with 25 rounds per generation are shown in the following figure 5.19:

(a) Simulation 19 with 25 rounds (b) Simulation 20 with 25 rounds (c) Simulation 21 with 25 rounds

Figure 5.19: Stack plots from simulations 19, 20 and 21 with 25 rounds per generation

It can be seen in figure 5.19 how the frequency increases as the death rate increases, reflecting how death rate affects the periods of dominance from a strategy. Also the behaviour from the stack plots reflects the mixed Nash equilibrium. Because it can be seen how when a strategy is dominating the counterpart appears and dominates it.

Simulations 22, 23, 24 have an increasing death rate of 0.1, 0.5 and 0.9 respectively, with constant mutation rate of 0.5 and exploitation rate of 0.1. With death rate 0.1 we see an expected increase in the amplitude when increasing number of rounds. With death rate of 0.5 we see an increase in the frequency and a reduced amplitude for the graphs for 5, 25 and 100 rounds. And with death rate 0.9 a much smaller amplitude is perceived and still with a high frequency.

(a) Simulation 22 (b) Simulation 23 (c) Simulation 24

Figure 5.20: Simulation 22, 23 and 24 with 5 rounds per generation

(a) Simulation 22 (b) Simulation 23 (c) Simulation 24

Figure 5.21: Simulation 22, 23 and 24 with 25 rounds per generation

(a) Simulation 22 (b) Simulation 23 (c) Simulation 24

Figure 5.22: Simulation 22, 23 and 24 with 100 rounds per generation

As expected the only difference between simulations with 5, 25 and 100 rounds is an increasing amplitude when increasing number of rounds. Also the expected reduced amplitude when death rate is increased since there is a high mutation rate and a relatively low exploitation rate, population is renewed almost disregarding the performance of each agent strategy.

With increasing death rate of 0.1, 0.5 and 0.9 for simulation 25, 26 and 27 respectively; and constant mutation rate of 0.9 and exploitation rate of 0.1 for the three simulations. A similar behaviour as the previous simulations (22, 23 and 24) analised, can be seen in the following graphs and stackplots for simulations 25, 26 and 27.

(a) Simulation 25 (b) Simulation 26 (c) Simulation 27

Figure 5.23: Simulation 25, 26 and 27 with 5 rounds per generation

(a) Simulation 25 (b) Simulation 26 (c) Simulation 27

Figure 5.24: Simulation 25, 26 and 27 with 25 rounds per generation

(a) Simulation 25 (b) Simulation 26 (c) Simulation 27

Figure 5.25: Simulation 25, 26 and 27 with 100 rounds per generation

The amplitude increases as the number of rounds per generation are increased in 5, 25 and 100. For these simulations the increasing amplitude is smaller, because of the random factor that the mutation rate introduces (and how it excludes the agent strategy with highest accumulated utility). It should also be noted the incrementing frequency when using 0.5 and 0.9 death rate. And for simulation 27, it can be seen that for the different number of rounds the amplitude and frequency are similar for all, these can be attributed to the high mutation rate and high death rate. It is also important to mention that the low exploitation rate in combination with high mutation rate will have an effect on the simulation, and therefore all the agent strategies will be present in very similar proportions in the population. The following stack plots in figure 5.26 can be observed to have a better visual understanding:

(a) 5 rounds (b) 25 rounds (c) 100 rounds

Figure 5.26: Stack plots for simulation 27 with 5, 25 and 100 rounds per generation

Strategy 1 represents 'Heads' and strategy 2 represents 'Tails'. It can be seen that each strategy during the whole simulation is present in the total population in a proportion very close to 0.5.

Nash Equilibrium

The Nash Equilibrium for the matching pennies game will be calculated next, since it cannot be determined just by looking at the normal form table and selecting the best responses since this game has a mixed Nash equilibrium.

		Player 2	
		Heads	Tails
Player 1	Heads	1, -1	-1, 1
	Tails	-1, 1	1, -1

Table 5.22: Matching pennies bimatrix payoff.

The mixed Nash equilibrium is calculated by making equal the expected payoff when playing 'heads' to the expected payoff of playing 'tails'. It will be assumed that the probability for player 1 of playing 'heads' is σ and for playing tails 1 - σ, resulting in the following:

$$(-1)(\sigma) + (1-\sigma) = (1)(\sigma) + (-1)(1-\sigma)$$
$$-4\sigma = -2$$
$$\sigma = 1/2$$

This means that player 1 will play 'heads' with a probability of 1/2 , and also means she will be playing tails with 1/2. And now it is time to calculate the probability of playing 'heads; for player 2 with probability σ and 'tails' with probability 1 - σ.

$$(1)(\sigma) + (1-\sigma)(-1) = (-1)(\sigma) + (1)(1-\sigma)$$
$$4\sigma = 2$$
$$\sigma = 1/2$$

This means that player 2 when playing against player 1 will play' heads' 1/2 of the times and in consequence play 'tails' in the other 1/2 of the times. This means that the Nash equilibria is when:

((1/2, 1/2), (1/2, 1/2))

ESS

Now let us find the ESS for the matching pennies game. In the same way the prisoner's dilemma ESS was calculated. It will be assumed the strategy 'Heads' is the population and the invader is the strategy 'Tails'. Thus the payoff for the strategy 'Heads' is the following:

$$(1-\epsilon)(1) + (-1)\epsilon = 1 - 2\epsilon$$

And the payoff for the strategy 'Tails' is:

$$(1-\epsilon)(-1) + (1)\epsilon = 2\epsilon - 1$$

Both strategies appear to be equivalent unless there is a very small value of ϵ.But since the simulation gives almost equal starting proportions to each strategy it will be assumed that the 'invading' strategy 'Tails' appears in a proportion of ϵ=0.5. Resulting in the following:

$$1 - 2(0.5) = 0$$

And for the strategy 'Defect' is:

2(0.5)- 1 = 0

So the resulting expected payoff for both is 0 thus it can be said that 'Heads' is not an evolutionarily stable strategy, Given that it does not dominate 'Tails'. Next step is to check if the strategy 'Tails' is evolutionarily stable. And therefore it is assumed that 'Heads' is the invader. The expected payoff of the strategy 'Tails' is:

$$(1-\epsilon)(-1) + (1)(\epsilon) = 2\epsilon - 1$$

And the payoff for the strategy 'Heads' is:

$$(1-\epsilon)(1)+(-1)(\epsilon)=1-2\epsilon$$

Again it appears that they are equal. So 'Tails' is not an ESS either. But it will be calculated it anyway assuming that the 'invading' strategy 'Heads' appears in a proportion of ϵ=0.5. For the strategy 'Tails' the following is the expected payoff for this proportions :

$$1 - 2(0.5) = 0$$

And for the strategy 'Heads' is:

$$2(0.5) - 1 = 0$$

The strategy 'Tails' is not an ESS either. And this explains the behaviour in the graphs, how through time both strategies appear to interact and oscilate in the middle (0.5)

5.2.3 Battle of sexes

For battle of sexes the Nash equilibrium will be calculated in the next section and it will be seen that there are 2 non symmetric Nash equilibria and a mixed Nash equilibrium. To see how these 3 equilibria behave the stack plots and graphs from 2 simulations will be shown. No other characteristics for this simulation will be analyzed, only the behaviour derived from Nash equilibria which will be explained. In this section the name of the strategy 'Opera' will be 'strategy 1' since the output of our python code shows it like this and for 'Football' the name 'strategy 2' will be used. They will be used interchangeably along this section. The table for simulations can be found in the appendix C, and it is similar to prisoner's dilemma and matching pennies tables. We will use the configuration for simulations 4 and 19, with 5 and 25 rounds per generation. First we will see the behaviour of these simulations without inputing an initial distribution, which means a random intial distribution will be set by python. It will usually be close to 0.5 for each strategy, and the resulting graphs for this test are the following: For simulation 4:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.27: Graph and stack plot for simulation 4 with no inputed initial strategy, with 5 rounds per generation

(a) 25 rounds

(b) 25 rounds stackplot

Figure 5.28: Graph and stack plot for simulation 4 with no inputed initial strategy, with 25 rounds per generation

From the previous figures 5.27 and 5.28 it can be seen that rounds per generation that rounds per generation appears to allow strategies to have a more clear dominance with the configuration of simulation 4. But the important observation comes with the set of graphs and stack plots shown in figures 5.29 and 5.30 for simulation 19 without setting an initial distribution. For simulation 19 these are the resulting graphs and plots for 5 and 25 rounds per generation:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.29: Graph and stack plot for simulation 19 with no inputed initial strategy, with 5 rounds per generation

(a) 25 rounds

(b) 25 rounds stack plot

Figure 5.30: Graph and stack plot for simulation 19 with no inputed initial strategy, with 25 rounds per generation

For the previous graphs and stack plots in figures 5.29 and 5.30 it appears that the number of rounds per generation has an impact on which strategies will dominate through time. Even if in simulation 4 this was not true. The number of rounds per generation do not influence in which strategy will be dominating. What happened in simulation 19 was that python does not assign exactly 0.5 for each strategy the tables 5.23 and 5.24 will show what was the initial distribution for simulation 19 with 5 and 25 rounds per generation.

	Strategy 1	Strategy 2
Row agents	0.476	0.524
Col agents	0.514	0.486

Table 5.23: Table of initial values distribution for simulation 19, 5 rounds.

	Strategy 1	Strategy 2
Row agents	0.5	0.5
Col agents	0.494	0.506

Table 5.24: Table of initial values distribution for simulation 19, 25 rounds.

From the previous tables it can be seen that there does not appear to be a pattern determining which strategy will be dominating. This may suggest that when there is no significant difference between the distributions, the dominating strategy will be determined by random factors which can work in favour of either strategy. To prove this initial strategies distribution will be set 0.5 for each strategy for simulation 4 and simulation 19 will be used.

The following figures 5.31 and 5.32 graphs can be found where the behaviour when using an initial distribution of 0.5 for each strategy for simulation 4 can be seen:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.31: Graph and stack plot for simulation 4 with initial strategy, with 5 rounds per generation

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.32: Graph and stack plot for simulation 4 with initial strategy, with 5 rounds per generation

By looking at the previous stack plots and graphs in figures 5.31 and 5.32 for simulation 4 with 5 rounds per generation, it can be seen that for each run a different strategy was dominating. It is important to remember that an initial strategy distribution of 0.5 for each has been set. Now simulation 4 with 25 rounds per generation with 0.5 as initial proportion for each strategy is run and the following graphs and stackplots show the behaviour after running it twice with the same parameters:

(a) 25 rounds

(b) 5 rounds stackplot

Figure 5.33: Graph and stack plot for simulation 4 with initial strategy, with 25 rounds per generation

(a) 25 rounds

(b) 5 rounds stackplot

Figure 5.34: Graph and stack plot for simulation 4 with initial strategy, with 25 rounds per generation

From the previous stackplots and graphs in figures 5.33 and 5.34 it can be seen that the behaviour for simulation 4 with 25 rounds per generation is also random. So it can be said that with equal initial proportions for each strategy the behaviour of which will dominate is random. Now we check to see if this behaviour is true for simulation 19.

Simulation 19 with 5 rounds per generation and 0.5 initial proportion for each strategy, run twice with the same parameters gives the following graphs and stack plots from figures 5.35 and 5.36:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.35: Graph and stack plot for simulation 19 with initial strategy, with 5 rounds per generation

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.36: Graph and stack plot for simulation 19 with initial strategy, with 5 rounds per generation

For simulation 19 with 25 rounds per generation and 0.5 initial proportion for each strategy can be found in the following graphs and stack plots from figure 5.37 and 5.38:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.37: Graph and stack plot for simulation 19 with initial strategy, with 25 rounds per generation

(a) 25 rounds

(b) 25 rounds stackplot

Figure 5.38: Graph and stack plot for simulation 19 with initial strategy, with 25 rounds per generation

These reinforces the idea that with equal initial distribution, any of the strategies which are equilibrium can dominate in time. Given its mixed strategy equilibrium. Next we will see how when using a more favourable initial distribution for one of the strategies will help that strategy to dominate the other through time.

It can be noted that the number of rounds per generation does not influence. Now simulation 4 and simulation 19 with 5 rounds per generation will be run, and each with an initial strategy distribution of 0.7 for strategy 1 and 0.3 for strategy 2.

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.39: Graph and stack plot for simulation 4 with initial strategy (0.7, 0.3), with 5 rounds per generation

And for simulation 19 with the same distribution:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.40: Graph and stack plot for simulation 19 with initial strategy (0.7, 0.3), with 5 rounds per generation

From the previous stack plots from figures 5.39 and 5.40 it can be seen that the strategy that dominates is strategy 1, which had the highest proportion in the initial distribution. Now we will change the proportions to favour strategy 2. Now it will be calculated for simulation 4 and simulation 19, each with an initial strategy distribution of 0.3 for strategy 1 and 0.7 for strategy 2:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.41: Graph and stack plot for simulation 4 with initial strategy (0.3, 0.7), with 5 rounds per generation

And for simulation 19 with the same distribution:

(a) 5 rounds

(b) 5 rounds stackplot

Figure 5.42: Graph and stack plot for simulation 19 with initial strategy (0.3, 0.7), with 5 rounds per generation

Strategy 2 appears to be the dominating strategy by looking at the previous stack plots in figures 5.41and5.42. This behaviour can be seen with any configuration of simulations from the table, when choosing a higher initial proportion for one of the two strategies in the 'Battle of sexes' game. And for the this game is the most relevant behaviour to analyze. The next step is to calculate the Nash equilibria and ESS.

Nash Equilibrium

Nash equilibrium for the battle of sexes game is calculated by looking at the normal form table and it can be noticed that there are 2 non symmetric Nash equilibria, this means that they do not play the same strategy; and there can also be found a mixed Nash equilibrium.

		Female	
		Opera	Football
Male	Opera	$\underline{2}$, $\underline{1}$	0, 0
	Football	0, 0	$\underline{1}$, $\underline{2}$

Table 5.25: Battle of sexes bimatrix payoff.

Now the mixed Nash equilibrium is calculated by making equal the expected payoff when going to the Opera with the expected payoff of watching Football. It will be assumed that the probability for the Male player of going to the Opera is σ and for watching Football 1 - σ, resulting in the following:

$$(2)(\sigma) + (1 - \sigma)(0) = (0)(\sigma) + (1)(1 - \sigma)$$

$$3\sigma = 1$$

$$\sigma = 1/3$$

This means that the Male player chooses to go to the opera with the probability of 1/3 , and also means he will prefer watching football 2/3. And now we calculate the probability of going to the Opera for the Female player with probability σ and for watching Football with probability 1 - σ.

$$(1)(\sigma) + (1 - \sigma)(0) = (0)(\sigma) + (2)(1 - \sigma)$$

$$3\sigma = 2$$

$$\sigma = 2/3$$

This means that the Female player will choose to go to the Opera with a probability of 2/3 and in consequence only chooses to watch football the other 1/3. This means that the Nash equilibria is when:

$$((1/3,\ 2/3),\ (2/3,\ 1/3))$$

ESS

Now to find the ESS for this game. It will be assumed that going to the 'Opera' is the population and the invader is the strategy to watch 'Football'. Thus the payoff for the strategy 'Opera' is the following:

$$(1 - \epsilon)(2) + (0)\epsilon = 1 - 2\epsilon$$

And the payoff for the strategy watching 'Football' is:

$$(1 - \epsilon)(0) + (1)\epsilon = \epsilon$$

With a value of $\epsilon \leq 0.3$ 'Opera' is an ESS. But since the simulation runs every time with almost equal starting proportions to each strategy it will be assumed that the 'invading' strategy 'Football' appears in a proportion of ϵ=0.5. And the calculation is as follows:

$$1\text{- } 2(0.5) = 0$$

And for the strategy 'Football' is:

$$(0.5) = 0.5$$

The expected payoff for 'Football' is greater thus it can be said that 'Opera' is not an evolutionarily stable strategy, given that it does not dominate 'Football'. Now a check to see if the strategy 'Football' is evolutionarily stable. And therefore it is assumed that 'Opera' is the invader. The expected payoff of the strategy 'Football' is:

$$(1 - \epsilon)(1) + (0)(\epsilon) = 1 - \epsilon$$

And the payoff for the strategy 'Opera' is:

$$(1 - \epsilon)(0) + (2)(\epsilon) = 2\epsilon$$

So it can be seen that 'Football' is not an ESS either, given that under certain values of ϵ it will not be dominant. Now it is calculated assuming that the 'invading' strategy 'Opera' appears in a proportion of ϵ=0.5. For the strategy 'Football':

$$1 \text{ - } (0.5) = 0.5$$

And for the strategy 'Opera' is:

$$2(0.5) = 1$$

The strategy 'Football' is not an ESS either. Each strategy will appear to be evolutionarily stable according to their probability of appearance in the population. But with the default settings for our simulation which give a proportion of 0.5 to each there is not a clear evolutionarily stable strategy at the beginning of each simulation.

5.2.4 Hawk-dove

For the hawk-dove game 5 rounds per generation were used for the different combinations of the table of simulations. The most interesting behaviours from the combinations will be analyzed briefly. The simulation table is similar to those of the previous games but it is only ran with 5 rounds per generation and can be found in appendix D For simulations 3, 4 and 5 the resulting graphs are the following:

(a) Simulation 3 (b) Simulation 4 (c) Simulation 5

Figure 5.43: Graphs for simulations 3, 4 and 5

(a) Simulation 3 (b) Simulation 4 (c) Simulation 5

Figure 5.44: Stackplots for simulations 3, 4 and 5

From the previous figures 5.43 and 5.44 it can be said the dominating strategies are strategy 1 for row agents and strategy 2 for column players. With an ascending death rate of 0.1, 0.5 and 0.9 for simulation 3 , 4 and 5 respectively. The gap between the dominating and dominated strategies reduces as the death rate increases. As in prisoner's dilemma this happens because as more agents are allowed to be eliminated the dominance they do not have possibility of accumulating a high population. In the stack plots 5.44 for these 3 games it can be seen that strategy 1 and 2 are close to having an equal distribution throughout the simulations.

Now simulations 8, 9, 12 and 14 will be run. Where it can be seen that the strategies that dominate are strategy 2 for row agents and strategy 1 for column agents. These four simulations have a varied combination of factors, as presented in the following table 5.26.

	Death Rate	Mutation Rate	Exploitation rate
Simulation 8	0.1	0.5	1
Simulation 9	0.1	0.9	1
Simulation 12	0.9	0.1	0.5
Simulation 14	0.5	0.5	0.5

Table 5.26: Hawk-Dove combinations for simulations 8, 9, 12 and 14.

A pattern does not appear to exist, only that randomness is higher because of the increasing mutation rate, decreasing exploitation rate or combination of both situations. The graphs for the simulations can be found in figures 5.45 and 5.46 are the following:

(a) Simulation 8

(b) Simulation 9

Figure 5.45: Graphs for simulations 8 and 9

(a) Simulation 12

(b) Simulation 14

Figure 5.46: Graphs for simulations 12 and 14

The stack plots from figures 5.47 and 5.48 it can be seen how the behaviour from the simulations 3, 4 and 5 prevails also despite of which combination of strategies are played in agents. Also it can be noted that strategy 1 and strategy 2 (hawk and dove strategies) tend to be close to equally distributed along the whole simulation. An interesting initial behaviour for simulation 8 and 9 can be seen but when running it for 100 simulations it eventually stabilizes and behaves as equally distributed. In the case of simulation 12 there are obvious changes and it does not appear to be as stable as the others, but this is given to the high death rate, and the 0.5 exploitation rate. This gives an expected behaviour where the population will look for stability, but it the high death rate and the randomness of reproducing that gives the relatively low exploitation rate, will generate some irregular patterns but it can be seen that it is close to an equal distribution for strategies.

(a) Simulation 8

(b) Simulation 9

Figure 5.47: Stack plots for simulations 8 and 9

(a) Simulation 12

(b) Simulation 14

Figure 5.48: Stack plots for simulations 12 and 14

With these a conclusion can be made that, as will be see when calculating Nash equilibria, randomness determines which of the 2 existing Nash equilibria will be the result for this game. Now the Nash equilibria and the ESS for this game will be determined. Unlike prisoner's dilemma, in hawk and dove game if an 'agressive' strategy is present in the population, the agents playing against that strategy do better by using 'dove' strategy than retaliating.

Nash Equilibrium

The Nash equilibrium for the hawk-dove game will now be found. The Nash equilibrium for the this game, can be found by looking at the normal form table and select the best responses for each player. There are 2 Nash equilibria in this game.

		Player 2	
		Hawk	Dove
Player 1	Hawk	0, 0	$\underline{3}$, $\underline{1}$
	Dove	$\underline{1}$,$\underline{3}$	2, 2

Table 5.27: Hawk-Dove bimatrix payoff.

The mixed Nash equilibrium is calculated by making equal the expected payoff when playing 'Hawk' to the expected payoff of playing 'Dove'. It will be assumed that the probability for player 1 of playing 'Hawk' is σ and for playing 'Dove' is 1 - σ, resulting in the following:

$$(0)(\sigma) + (1-\sigma)(3) = (1)(\sigma) + (2)(1-\sigma)$$

$$-2\sigma = -1$$

$$\sigma = 1/2$$

This means that player 1 will play 'Hawk' 1/2 of the times, and also means she will be playing 'Dove' with a probability of 1/2. And now to calculate the probability of playing 'Hawk' for player 2 with probability σ and playing 'Dove' with probability 1 - σ.

$$(0)(\sigma) + (1-\sigma)(3) = (1)(\sigma) + (2)(1-\sigma)$$

$$4\sigma = 2$$

$$\sigma = 1/2$$

This means that player 2 when playing against player 1 will play 'Hawk' with a probability of 1/2 and in consequence play 'Dove' with the remaining probability of 1/2. This means that the Nash equilibria is when:

$$((1/2,\ 1/2),\ (1/2,\ 1/2))$$

Therefore it can be seen that there exist 2 non symmetric Nash equilibria and a mixed Nash equilibrium.

ESS

Now it will be found the ESS for this game. It is assumed the strategy 'Hawk' is the population and the invader is the strategy 'Dove'. Thus the payoff for the strategy 'Hawk' is the following:

$$(1-\epsilon)(0) + (3)\epsilon = 3\epsilon$$

And the payoff for the strategy 'Dove' is:

$$(1-\epsilon)(1) + (2)\epsilon = 1 + \epsilon$$

It can be seen that with certain values of $\epsilon > 5$ Hawk is an ESS. Since the simulation every time gives almost equal starting proportions to each strategy it will be assumed that the 'invading' strategy 'Dove' appears in a proportion of ϵ=0.5. And calculates as follows:

$$3(0.5) = 1.5$$

And for the strategy 'Dove' is:

$$1 + (0.5) = 1.5$$

So the expected payoff for both are equal thus we can say that 'Hawk' is not an evolutionarily stable strategy with an initial distribution of 0.5, given that it does not dominate 'Dove'. Now it is checked if the strategy 'Dove' is evolutionarily stable. And therefore 'Hawk' is assumed to be the invader. First the expected payoff for the strategy 'Dove' is:

$$(1-\epsilon)(0) + (3)(\epsilon) = 3\epsilon$$

And the payoff for the strategy 'Hawk' is:

$$(1-\epsilon)(1) + (2)(\epsilon) = 1 + \epsilon$$

It can be seen a similar situation as the previous. So it can be assumed that 'Dove' is not an ESS either. But it will be calculated anyway assuming that the 'invading' strategy 'Hawk' appears in a proportion of ϵ=0.5. And the following is the result for the strategy 'Dove':

$$3(0.5) = 1.5$$

And for the strategy 'Hawk' is:

$$1 + (0.5) = 1.5$$

The strategy 'Dove' is not an ESS either under this distribution.

5.2.5 Axelrod python package

Axelrod's tournament

Now a python package called 'axelrod' will be used. This package was originated from the concept of Robert Axelrod's tournaments. Robert Axelrod is a doctor in political science which as briefly mentioned before, who was interested in how cooperative strategies can persist in a population through time in an environment where hostile strategies exist. Axelrods first tournament had originally 14 strategies, which were submitted by known game theorists at the time the tournament was played. From 'axelrod's' package in python a simulation with 8 strategies was run only for observation purposes. A description for the characteristics of each strategy used can be found in appendix E. From the 'axelrod' library there can be run an ecological variant, this produces a stack plot which we was compared to the stack plot produced by the code 'ablearn' from this project. In 'axelrod's' python library, the ecological shows how a strategy performs in the population through time giving how likely it is for a strategy to reproduce [46]. Instructions of how to use 'Axelrod's' python library can be found in the package documentation page, a link to it can be found in appendix E and in the references section.

The strategies used were mentioned in the library's description, as a reminder they will be listed again: Tit for Tat, Grofman, Shubik, Grudger, Davis, Feld, Joss and Tullock. With the axelrod library we get a payoff matrix from playing, which was transformed into the following bimatrix:

	Tit for Tat	**Grofman**	**Shubik**	**Grudger**	**Davis**	**Feld**	**Joss**	**Tullock**
Tit for Tat	3 ,3	1.7414 , 1.759	3 , 3	3 , 3	3 , 3	1.392 , 1.417	1.2814 , 1.3065	1.626 , 1.648
Grofman	1.759 , 1.7414	1.775 , 1.775	1.143 , 1.975	0.750 , 2.143	0.881 , 2.148	1.549 , 1.861	1.662 , 1.777	1.592 , 1.849
Shubik	3 , 3	1.975 , 1.143	3 , 3	3 , 3	3 , 3	1.338 , 1.338	1.079 , 1.086	1.348 , 1.302
Grudger	3 , 3	2.143 , 0.7505	3 , 3	3 , 3	3 , 3	1.2479 , 1.2479	1.083 , 1.085	1.293 , 1.2
Davis	3 , 3	2.148 , 0.881	3 , 3	3 , 3	3 , 3	1.236 , 1.238	1.104 , 1.119	1.32 , 1.23
Feld	1.417 , 1.392	1.861 , 1.549	1.338 , 1.338	1.247 , 1.247	1.238 , 1.236	1.315 , 1.315	1.159 , 1.171	1.375 , 1.36
Joss	1.306 , 1.281	1.777 , 1.662	1.086 , 1.079	1.085 , 1.083	1.119 , 1.104	1.171 , 1.159	1.115 , 1.115	1.444 , 1.38
Tullock	1.648 , 1.626	1.849 , 1.592	1.302 , 1.348	1.2 , 1.293	1.23 , 1.32	1.36 , 1.375	1.38 , 1.444	1.484 , 1.484

Table 5.28: 'Axelrod' package transformed payoff bimatrix for 8 strategies from Axelrod's tournament.

The 'axelrod' package plays 10 repetitions of 200 turns each. And the resulting ranking for the strategies was in the following order: 'Tit For Tat', 'Shubik', 'Davis', 'Grudger', 'Tullock', 'Feld', 'Grofman', 'Joss'. The stack plot from 'axelrod' package was produced, this stack plot uses a probabilistic method to estimate how the strategies will evolve in time periods. The following stack plot shown in figure 5.49 from 'axelrod' package is estimated with 100 time units.

Figure 5.49: Stack plot with 100 time units produced with 'axelrod' package.

The stack plots in figure 5.50 were produced with the 'ablearn' package, with the values that considered standard for this project which are 0.1 death rate, 0.1 mutation rate and 1 exploitation rate.

(a) Stackplot 100 rounds

(b) Stackplot 25 rounds

(c) Stackplot 5 rounds

Figure 5.50: Stackplots for simulations from 'ablearn' package.

It can be seen that they have a different result from the predicted behaviour of the stack plot produced from 'axelrod' package in figure 5.49. This is because the agents from 'ablearn' do not change their strategies, they only reproduce and the resulting agent may posses the same strategy as the agent that was considered for its creation or it may posses any other according to the mutation and exploitation rate, this gives the observed resulting interaction. In 'axelrod' package each type of strategy takes in consideration previous interactions with agents and can shift to a convenient action according to their individual criteria, while 'ablearn' is pure interaction of agents which have a defined static behaviour with agents in this same condition.

Basic strategies

With the 'axelrod' package also a variable called 'basic_strategies' was run, which automatically generates the following set of strategies: Alternator, Cooperator, Defector, Random and Tit For Tat. The description in this strategies can be found in the documentation webpage for 'axelrod' package which can be found in appendix **??** also a brief description will be added in the appendix E.

	Alternator	Cooperator	Defector	Random	Tit For Tat
Alternator	2 , 2	4 , 1.5	0.5 , 3	2.302 , 2.217	2.515 , 2.49
Cooperator	1.5 , 4	3 , 3	0 , 5	1.495 , 4.002	3 , 3
Defector	3 , 0.5	5 , 0	1 , 1	3.07 , 0.482	1.019 , 0.994
Random	2.217 , 2.302	4.002 , 1.495	0.482 , 3.07	2.245 , 2.245	2.275 , 2.264
Tit For Tat	2.49 , 2.515	3 , 3	0.994 , 1.019	2.264 , 2.275	3 , 3

Table 5.29: 'Axelrod' package payoff bimatrix for basic strategies.

The following environmental stack plot shown in figure 5.51 was the result from the 'axelrod' package for the basic strategies:

Figure 5.51: Stack plot with 1000 time units produced with 'axelrod' package for basic strategies.

It can be seen that the strategy'Tit for Tat' will eventually dominate all the other strategies,

With the payoff matrix that 'axelrod' package outputs, the simulation with 100 generations with values for death rate of 0.1, mutation rate of 0.1 and exploitation rate of 1 was run with 100, 25 and 5 rounds per generation, the resulting stack plots can be found in the following figure 5.52.

(a) Stackplot 100 rounds

(b) Stackplot 25 rounds

(c) Stackplot 5 rounds

Figure 5.52: Stackplots for simulations from 'ablearn' package.

To be able to compare the stack plots from 'ablearn' with the stack plot from 'axelrod' package, it is important to remember the order the strategies were input for 'ablearn' since the program does not give us the name of the strategy (this will be one of the future improvements to be introduced into 'ablearn' package). The following table 5.30 will clarify what each strategy in 'ablearn' is related to.

Axelrod	**Ablearn**
Alternator	Strategy 1
Cooperator	Strategy 2
Defector	Strategy 3
Random	Strategy 4
Tit For Tat	Strategy 5

Table 5.30: 'Axelrod' package and 'ablearn' name equivalence.

By comparing the stack plots in figure 5.52 produced by 'ablearn' with the stack plot produced by 'axelrod' in figure 5.51, it can be seen that stack plot 5.52*c* shows results more similar to 'axelrod' stack plot. Where strategy 'Random' has a brief increase during the initial generations, 'Tit for Tat' increases and dominates the other strategies, and all strategies except for 'Tit for Tat' are reduced in time. What makes 'ablearn' stack plots behave slightly different is that the concept of mutation rate allows strategies that could

disappear in a long run to remain, since it gives the same possibility to any strategy to have a copy of itself.

By analyzing strategies from Axelrod's tournament and the basic strategies from the package 'axelrod' it can be seen that the package 'ablearn' produces similar behaviours to the expected under the parameters of 0.1 death rate, 0.1 mutation rate and 1 exploitation rate, when using the payoff matrix produced by 'axelrod' package.

Chapter 6

Conclusion and Further work

6.1 General conclusions about the results

Game theory has proven to give relevant results for many fields for a long period of time now, even if its application nowadays is mostly in economics. Evolutionary game theory which was derived from game theory, even if less complex assumptions can give equally good and accurate results, and nowadays has had more acceptance and its application has grown too. This suggests complex questions can be responded by answers where simpler criteria is assumed. Even if 'ablearn' package at this stage can only solve a limited type of games, it has proven to give reliable results. The algorithm used in 'ablearn' is simple, but given the number of iterations it would be long and not a very practical method for calculating manually. With all the available tools and advancement in technology this long process is made in less time and with less complications. From the analysis of the simulations' outputs it can be noted that death rate had a big influence. A low death rate allows agents with certain strategy (most commonly the dominant strategy) to have a higher presence in the population and when the rate is high agent with certain strategy (most commonly dominant strategy) still have a higher presence in the population but it is not as significant since in every generation a big number is eliminated and the agents with dominant strategies may not always be the only ones able to be reproduced. This difference in which agents with dominant strategies may not always be the ones to be reproduced, highly depends on the other two remaining rates (mutation rate and exploitation rate). As it can be seen between mutation and exploitation rate, the first has a higher impact in the outcome. One of the reasons is because of the costraint it adds to the dominant strategy agents (not allowing them to reproduce), another reason is that mutation rate acts as a second criteria so even after an agent has been selected to reproduce by the exploitation rate it is still possible that the mutation rate changes the selected agent and one with a different strategy may be chosen. With a high mutation rate even a possible overwhelming dominant strategy will not have an absolute presence in the population, although it will still be dominant. As the death rate increases in combination with a high mutation rate, the distribution of the agents with different strategies will tend to be equal for all in proportions. Exploitation rate is not very relevant when it is in combination with a high mutation rate, but with a low mutation rate the effect from

it can be noticed. But combined with a low mutation rate and a relatively high death rate, it can give equal opportunity to any agent with any strategy to reproduce if the exploitation rate is low. Another important thing that could be observed how results could be affected by adding an initial strategy distribution into a game where 2 pure Nash equilibria and a mixed Nash equilibria exist. It could be observed in the graphs and the stack plots that when giving a higher value in the initial distribution to one of the strategies, that strategy became the dominant strategy. When comparing the resulting stack plots from 'ablearn' package's with the resulting stack plot from 'axelrod' pacakge's, it could be seen that two different techniques were being used to estimate the behaviour of the strategies, but the results only slightly different. In a first view the stack plot from 'axelrod' seems smoother that is because it uses a probabilistic method to calculate the values, on the other hand 'ablearn' does all the iterations, the stochasticity produced is given the random interactions and the randomness that mutation and exploitation rate add to the reproduction of the strategies.

6.1.1 Further work

The package was created with the intention that contributions, from others in the field interested in the package, can be made to enhance it. Whilst in this project a code was built for creating agent based models that will use learning algorithms to solve games, the PYTHON functionalities are vast and possibilities for further developments are endless. One of the advantages of using a VCS, in this project Git is used, is that contributors can continually update work; in addition the tools for comparing contributions, checking for possible conflicts within the code and the communication offered by websites like Github make contributing package development easier. This thriving community provides opportunities for distributing packages and getting acknolowledgement to attract other enthusiasts and encourage contribution. This project is a contirbution to this community and with hopes in the future I along with others will improve it.

The results that 'ablearn' package in it's current state provides, are reliable for users to see how strategies in symmetric normal form games perform through time. It helps to identify in a graphic form Nash equilibria and evolutionary stable strategies. With the different combinations that can be used as input (number of generations, number of rounds, death rate, mutation rate, exploitation rate and distribution) it is able to produce the results of various different scenarios and should help the user how under different conditions the behaviour of ,dominating or dominated, strategies changes.

There are many possibilities for making the code aplicable to the variety of existing type of games. Some of the examples include developing the package so that it can solve a variety of games with n-players, with other types of environments and interactions i.e. instead of agents interacting randomly with any other agent, allow them to interact only with a certain number of agents which can create a network environment. Another possible enhancement includes incorporating the option of analysing historical data from a certain type of agents and generating a prediction of an outcome i.e. taking the records from the time runners in the same category and that run the same distance in races under different conditions such as weather, place and other possible relevant variables, and be

able to generate a prediction of which runner would be more likely to win a race against the other runners under certain conditions. The main purpose is to explore as many available algorithms to solve as many types of games as possible. Also it would be desirable that after a person reads this project an interest is drawn towards algorithms that can be applied in game theory and python's language broad potential applications,

Appendix A

Git commands

- **git status:** Used to see the what the status on the working directory. It will display two sections, one with the staged changes that are ready to be committed and another section displaying the changes that have not been staged, along with a small legend on the left side of the name of the file, that indicates what changes have been done to that file (deleted, modified, new file, etc.)
- **git add:** Used to stage the changes we have made in the directory for the local repository clone specific files. Before using this command all changes that have been made in the directory are considered unstaged. This means that any new file added or any already existing file that has been modified has to be staged with this command in order to include them in the next commit, any changes made that have not been staged with **git add** will not be considered in the next commit.

Figure A.1: Example of one file staged with git add.

- **git commit:** Takes a snapshot of the status of the local cloned repository, of course the files that considers are the ones that $textbfgitadd$ was previously used on. Creates a new point in the history of the project. Once this command is run, it commonly opens a text editor where I introduced the changes that were made in the files that are being changed in this commit.

Figure A.2: Example after using git commit asking for message for the commit.

- **git diff:** Using this command directly displays the content that has been changed in the project since the last commit and that are not staged for the next commit. If the command is run **git diff - - cached** it will display the changes that have been staged. If we wish to see staged and unstaged changes together we use **git diff HEAD** and if there is no interest in watching the specific changes but we want to display more information than just the one presented by **git status** we add **- - stat** after the different options we had with **git diff** [**options**] **- - stat** and this displays a summary of the changes.

Figure A.3: Example of general output with git diff and git diff HEAD - - stat (red color represent the deleted lines and green the added) .

- **git fetch** [**remote-name**]**:** This command pulls all the data from the remote project that is not contained the current copy we hold in our local repository clone. Basically used to update with all the changes other users (contributors) to the project have made, but it does not merge it with the work we have done because this has to be done manually. The name given to the remote repository we created a clone from is usually origin.

- **git branch:** Without any arguments the command gives a list of the existing branches in the local repository.The command followed with the name of the branch **git branch (branchname)** will create a branch with that given name out of the main project line (master), in the last commit made before the branch. So if we continue working on the master and then we switch to the branch all the changes

made to that point will be reverted to the context where the branch was created. If we wish to delete a local branch we use **git branch -d (branchname)**. To delete a remote branch we use the command **git push (remote-name) :(branchname)**.

- **git checkout (branchname):** This command is used to change between the different existing branches. A way of creating a branch and changing to it at the same time is a shortcut provided by Git **git checkout -b (branchname)**.
- **git merge:** This command is used to merge the changes that have been made in a branch are complete. And what is usually done is change into the main project line (master) with **git checkout (name of the branch we want to merge into)** and then use **git merge (name of the branch wished to be merged in)**. This way the changes, if there exists no conflict, will be merged into the main project line (master).
- **git pull:** Basically does first a **git fetch** immediately followed by a **git merge** from the tracked branch into the branch we are currently in. This command basically does the same as those two commands, but it can if there are changes that may cause a problem, it might make the process a bit more complex.

Figure A.4: Example diagram from Github of a master branch with a branch 1 for being merged.

- **git push:** This command helps to update the local changes we have made to the remote repository. And the way to use it is **git push (name given to the remote) (name of th**
We just need to be careful to not overwrite changes.
- **git log:** This command displays all the commit messages that have been previously used in the state of the project we currently are in. There is an alternative command that displays how the project has been modified up to the point we currently are in and is **git log - - oneline - - decorate - - graph - - all**

Figure A.5: Basic example of git log oneline.

Webpage for Github: **https://github.com/**

Appendix B

Matching pennies simulation table

Num.	Population	Generations	Rounds per Gen.	Death Rate	Mutation Rate	Exploitation Rate
1	1000	100	100	0.1	0.1	1
2	1000	100	25	0.1	0.1	1
3	1000	100	5	0.1	0.1	1
4	1000	100	100, 25, 5	0.5	0.1	1
5	1000	100	100, 25, 5	0.9	0.1	1
6	1000	100	100, 25, 5	0.9	0.5	1
7	1000	100	100, 25, 5	0.9	0.9	1
8	1000	100	100, 25, 5	0.1	0.5	1
9	1000	100	100, 25, 5	0.1	0.9	1
10	1000	100	100, 25, 5	0.1	0.1	0.5
11	1000	100	100, 25, 5	0.5	0.1	0.5
12	1000	100	100, 25, 5	0.9	0.1	0.5
13	1000	100	100, 25, 5	0.1	0.5	0.5
14	1000	100	100, 25, 5	0.5	0.5	0.5
15	1000	100	100, 25, 5	0.9	0.5	0.5
16	1000	100	100, 25, 5	0.1	0.9	0.5
17	1000	100	100, 25, 5	0.5	0.9	0.5
18	1000	100	100, 25, 5	0.9	0.9	0.5
19	1000	100	100, 25, 5	0.1	0.1	0.1
20	1000	100	100, 25, 5	0.5	0.1	0.1
21	1000	100	100, 25, 5	0.9	0.1	0.1
22	1000	100	100, 25, 5	0.1	0.5	0.1
23	1000	100	100, 25, 5	0.5	0.5	0.1
24	1000	100	100, 25, 5	0.9	0.5	0.1
25	1000	100	100, 25, 5	0.1	0.9	0.1
26	1000	100	100, 25, 5	0.5	0.9	0.1
27	1000	100	100, 25, 5	0.9	0.9	0.1
28	1000	100	100, 25, 5	0.5	0.5	1
29	1000	100	100, 25, 5	0.5	0.9	1

Table B.1: Configuration of simulations for matching pennies.

Appendix C

Battle of sexes simulation table

Num.	Population	Generations	Rounds per Gen.	Death Rate	Mutation Rate	Exploitation Rate
1	1000	100	100	0.1	0.1	1
2	1000	100	25	0.1	0.1	1
3	1000	100	5	0.1	0.1	1
4	1000	100	25, 5	0.5	0.1	1
5	1000	100	25, 5	0.9	0.1	1
6	1000	100	25, 5	0.9	0.5	1
7	1000	100	25, 5	0.9	0.9	1
8	1000	100	25, 5	0.1	0.5	1
9	1000	100	25, 5	0.1	0.9	1
10	1000	100	25, 5	0.1	0.1	0.5
11	1000	100	25, 5	0.5	0.1	0.5
12	1000	100	25, 5	0.9	0.1	0.5
13	1000	100	25, 5	0.1	0.5	0.5
14	1000	100	25, 5	0.5	0.5	0.5
15	1000	100	25, 5	0.9	0.5	0.5
16	1000	100	25, 5	0.1	0.9	0.5
17	1000	100	25, 5	0.5	0.9	0.5
18	1000	100	25, 5	0.9	0.9	0.5
19	1000	100	25, 5	0.1	0.1	0.1
20	1000	100	25, 5	0.5	0.1	0.1
21	1000	100	25, 5	0.9	0.1	0.1
22	1000	100	25, 5	0.1	0.5	0.1
23	1000	100	25, 5	0.5	0.5	0.1
24	1000	100	25, 5	0.9	0.5	0.1
25	1000	100	25, 5	0.1	0.9	0.1
26	1000	100	25, 5	0.5	0.9	0.1
27	1000	100	25, 5	0.9	0.9	0.1
28	1000	100	25, 5	0.5	0.5	1
29	1000	100	25, 5	0.5	0.9	1

Table C.1: Configuration of simulations for battle of sexes.

Appendix D

Hawk-dove simulation table

Num.	Population	Generations	Rounds per Gen.	Death Rate	Mutation Rate	Exploitation Rate
1	1000	100	100	0.1	0.1	1
2	1000	100	25	0.1	0.1	1
3	1000	100	5	0.1	0.1	1
4	1000	100	5	0.5	0.1	1
5	1000	100	5	0.9	0.1	1
6	1000	100	5	0.9	0.5	1
7	1000	100	5	0.9	0.9	1
8	1000	100	5	0.1	0.5	1
9	1000	100	5	0.1	0.9	1
10	1000	100	5	0.1	0.1	0.5
11	1000	100	5	0.5	0.1	0.5
12	1000	100	5	0.9	0.1	0.5
13	1000	100	5	0.1	0.5	0.5
14	1000	100	5	0.5	0.5	0.5
15	1000	100	5	0.9	0.5	0.5
16	1000	100	5	0.1	0.9	0.5
17	1000	100	5	0.5	0.9	0.5
18	1000	100	5	0.9	0.9	0.5
19	1000	100	5	0.1	0.1	0.1
20	1000	100	5	0.5	0.1	0.1
21	1000	100	5	0.9	0.1	0.1
22	1000	100	5	0.1	0.5	0.1
23	1000	100	5	0.5	0.5	0.1
24	1000	100	5	0.9	0.5	0.1
25	1000	100	5	0.1	0.9	0.1
26	1000	100	5	0.5	0.9	0.1
27	1000	100	5	0.9	0.9	0.1
28	1000	100	5	0.5	0.5	1
29	1000	100	5	0.5	0.9	1

Table D.1: Configuration of simulations for hawk-dove.

Appendix E

‘Axelrod’ package strategies

Link: *http://axelrod.readthedocs.org/en/latest/index.html*

Appendix F

Code

F.1 Init file

```
from population.agents import *
import environments
from algorithms.genetic import *
import tests
from simulation.simulation import *
from axl.matrix import *
```

F.2 Agents module

```
class Agent:
    """
    A generic class for agents that will be used by the library
    """

    """ INITIALIZATION """
    def __init__(self, strategies=False, label=False): # The properties each agent will have.
        self.strategies = strategies # Strategy the agent will use.
        self.utility = 0 # Utility the agent will obtain.
        self.label = label # If required a label to track a specific agent.

    def increment_utility(self, amount=1): # How much their utility will increase
        self.utility += float(amount) # Utility for each agent will increase in this amount.
```

F.3 Environment module

```
import ablearn
import random

class BiMatrixRandomEnv():
        """ This class will create the row and column agents for the 2 player game
                , and will make them interact (compete) with each other randomly in order
                to obtain which strategy for each type of agent is the one that generates
                more utility(fittest)
        """

        """ INITIALIZATION"""

        def __init__(self, number_of_agents, bimatrix): # The needed input for this class.

                self.number_of_agents = number_of_agents # Total number of agents before determining what type of agents they are.
                self.number_of_row_agents = number_of_agents / 2 # Split number of agents and determine how many row agents.
                self.number_of_col_agents = number_of_agents / 2 # Split number of agents and determine how many column agents.

                self.bimatrix = bimatrix # For our interactions the bimatrix gives the strategy payoff for each type of agent.

                self.number_of_row_strategies = len(bimatrix) # Count the number of strategies available for each row agent.
                self.number_of_col_strategies = len(bimatrix[0]) # Count the number of strategies available for each column agent.

                self.row_strategies = range(self.number_of_row_strategies) # A strategy number for row agents is assigned.
                self.col_strategies = range(self.number_of_col_strategies) # A strategy number for column agents is assigned

                self.row_agents = [ablearn.Agent() for k in
                                range(self.number_of_row_agents)] # Creates row agents without strategies
                self.col_agents = [ablearn.Agent() for k in
```

```
                                range(self.number_of_col_agents)] # Creates column agents without strategies

        def interact(self): # This function will make row agents interact (compete) against column agents.
            """ Agents interact and get a resulting utility from the interaction"""
            pairs = self.randomly_pair_agents() # Pairs a row agent with a column agent.
            for ra, ca in pairs:
                utility = self.strategies_to_utilities([ra.strategies, ca.strategies]) # Variable that is equivalent to the utility each row and column agent get from the interaction.
                ra.increment_utility(utility[0]) # Increments each row agent's utility.
                ca.increment_utility(utility[1]) # Increments each column agent's utility.

        def strategies_to_utilities(self, strategy): # Transform value of strategy to the corresponding utility.
            """ Utility from each interacting pair of agents for each agents is determined"""
            return self.bimatrix[strategy[0]][strategy[1]] # Returns the payoff pair equivalent to the pair of utilities one for each type of agent.

        def randomly_pair_agents(self): # Randomly pairs agents
            """ Each row agent and column agent will be paired with their counterparts """
            random.shuffle(self.row_agents) # Randomly shuffle the row agents in the list.
            random.shuffle(self.col_agents) # Randomly shuffle the column agents in the list.
            return zip(self.row_agents, self.col_agents) # Create pairs of random row and column agents.
```

F.4 Genetic module

```
import random
import ablearn
import copy



class Genetic():
        """
    This class acts as a simple Genetic Algorithm for interaction of row and column players
        """

        """"            INITIALIZATION            """
        def __init__(self, generations, rounds_per_generation, death_rate,
                                mutation_rate, exploitation_rate, initial_strategy_distribution=False): # This are the needed input for this class
                self.generations = generations # Number of generations. Times the simulated environment will run.
                self.rounds_per_generation = rounds_per_generation # The number of time the simulation will run for each geneartion.
                self.death_rate = death_rate # Equivalent to the percentage of agents that will die at the end of each generation. An equal number of agents with random or the best strategies will be created for the next generation.
                self.mutation_rate = mutation_rate # Mutation in this code allows to create an agent with different strategy than the best strategy in the previous generation. A high value will allow our new agents to mutate more often, and a low will do the opposite.
                self.exploitation_rate = exploitation_rate # This value will give us the percentage of the population that will not pass their strategies to the new agents. A value of 1 (100%) will allow only the agent with the highest utility to pass on her strategy to the new agents.

        def assign_strategies(self, agents, agents_strategies): # This function is run once the column and row agents are created, and it will randomly assign them a strategy to use.
                """ Assigning random strategies to agents"""
                for a in agents: # Iterate through agents.
                        a.strategies=random.choice(agents_strategies) # Assign a random strategy to each agent.
```

```
                    #print a.strategies

        def assign_strategies_initial_distribution(self, agents, agents_strategies, number_of_agents, agent_initial_distribution):
            self.final_agent_strategies = []
            agents_number_of_strategies=[int(agent_initial_distribution[k]*int(number_of_agents / 2)) for k in range(len(agents_strategies))]
            for k in range(len(agents_strategies)):
                for s in range(agents_number_of_strategies[k]):
                    if k==agents_strategies[k]:
                        self.final_agent_strategies.append(agents_strategies[k])
            random.shuffle(self.final_agent_strategies)
            #print self.final_agent_strategies
            for a in agents:
                a.strategies=self.final_agent_strategies[agents.index(a)]

        def kill_agents(self, agents, number_of_agents, number_of_x_agents, agents_strategies): # This function will kill the calculated number of agents from the death rate after each generation.
            self.number_of_deaths_per_generation = int(self.death_rate * (number_of_x_agents)) # This variable represents the number of agents that will be killed after each generation.
            """ Eliminating row and column agents with lowest utilities """
            d = 0
            while d < self.number_of_deaths_per_generation:
                agents.sort(key=lambda x: x.utility) # Sort agents according to utility from low to high.
                del agents[0] # Kills agents with lowest utility (payoff)
                d += 1

        def reproduce_agents(self, agents, number_of_agents, number_of_x_agents, agents_strategies): # Creates each generation the equal number of agents killed per generation.
            choice = int((number_of_agents / 2) * (self.exploitation_rate - 1 ) + 0.05) - 1 # Calculates how many agents will be taken in account for passing on their strategies to the new agents.
            """ Reproducing row and column agents with highest utilities """
```

```
        while len(agents) < number_of_x_agents:
            agents.sort(key=lambda x: x.utility) # Sort agents according to utility (payoff) from low to high.
            agents.append(copy.deepcopy(random.choice(agents[choice:]))) # Creates an agent and assigns her a strategy according to the exploitation rate.
            if random.random() < self.mutation_rate: # How frequently the new agents will be mutated, and assigned a different strategy from the best of the previous generation.
                agents[-1].strategies = random.choice([s for s in agents_strategies if s != agents[-1].strategies]) # Select a strategy for each new agent different from the best of the current generation.
```

F.5 Simulation module

```
from __future__ import division # include __future__ module in this case so a specific division can give decimal points.
import random
import ablearn
import matplotlib.pyplot as pl
import matplotlib.cm as cm
from collections import defaultdict
from matplotlib.patches import Rectangle

#import axelrod.matrix as axl

class Simulation:

    def __init__(self,tog, na, ge, rpg, dr, mr, er, isd=False):
        global env, ga, plt
        self.tog = tog
        self.na = na
        self.ge = ge
        self.rpg = rpg
        self.dr = dr
        self.mr = mr
        self.er = er
        self.isd = isd
        ag = ablearn.Agent()
        if self.tog=='pd': # prisoners' dilemma
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                bimatrix=[[[3, 3], [0, 5]],
                                          [[5, 0], [1, 1]]])
        elif self.tog =='psr': # paper-scissors-rock
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                bimatrix=[[[0, 0], [-1, 1], [1,-1]],
                                            [[1,-1], [0, 0], [-1, 1]],
                                            [[-1, 1], [1, -1], [0,0]]])
        elif self.tog=='mp': # matching pennies
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                bimatrix=[[[1,-1], [-1,1]],
                                        [[-1,1], [1,-1]]])
        elif self.tog=='bos': # battle of sexes
```

```
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                 bimatrix=[[[2, 1], [0, 0]],
                                           [[0, 0], [1, 2]]])
        elif self.tog=='hd': # chicken/hawk and dove
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                 bimatrix=[[[0, 0], [3, 1]],
                                           [[1, 3], [2, 2]]])
        elif self.tog=='sh': # stag hunt
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                 bimatrix=[[[5, 5], [0, 1]],
                                           [[1, 0], [1, 1]]])
        elif self.tog=='cs': # choosing sides
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                 bimatrix=[[[1, 1], [0, 0]],
                                           [[0, 0], [1, 1]]])
        elif self.tog=='axl':
            env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(
                                number_of_agents=self.na,
                                bimatrix=[[[3.0, 3.0], [1.7414999999999998, 1.759],
                                [3.0, 3.0], [3.0, 3.0], [3.0, 3.0], [1.3924999999999998, 1.4175],
                                [1.2814999999999999, 1.3065000000000002], [1.6260000000000001,
                                1.6485000000000003]],
                                [[1.759, 1.7414999999999998], [1.7752499999999998,
                                1.7752499999999998],
                                [1.1430000000000002, 1.9754999999999998], [0.7505, 2.1430000000000002],
                                 [0.881, 2.1485], [1.549, 1.8615], [1.6625, 1.7774999999999999],
                                 [1.592, 1.8495000000000001]], [[3.0, 3.0],
                                 [1.9754999999999998, 1.1430000000000002], [3.0, 3.0],
                                 [3.0, 3.0], [3.0, 3.0], [1.3380000000000003,
```

```
                                    1.3380000000000003],
                                 [1.0790000000000002, 1.0864999999999998],
                                 [1.348, 1.3029999999999997]], [[3.0, 3.0],
                                 [2.1430000000000002, 0.7505], [3.0, 3.0],
                                 [3.0, 3.0], [3.0, 3.0],
                                    [1.2479999999999998,
                                    1.2479999999999998],
                                 [1.083, 1.0855000000000001], [1.293,
                                    1.2005000000000003]],
                                [[3.0, 3.0], [2.1485, 0.881], [3.0, 3.0],
                                   [3.0, 3.0], [3.0, 3.0],
                                [1.236, 1.2385], [1.1045,
                                   1.1195000000000002], [1.32, 1.23]],
                                [[1.4175, 1.3924999999999998], [1.8615,
                                   1.549],
                                [1.3380000000000003, 1.3380000000000003],
                                [1.2479999999999998, 1.2479999999999998],
                                [1.2385, 1.236], [1.31575, 1.31575],
                                   [1.1590000000000003, 1.1715],
                                [1.3755000000000002, 1.3604999999999998]],
                                [[1.3065000000000002, 1.2814999999999999],
                                [1.7774999999999999, 1.6625],
                                   [1.0864999999999998,
                                   1.0790000000000002],
                                [1.0855000000000001, 1.083],
                                   [1.1195000000000002, 1.1045],
                                [1.1715, 1.1590000000000003], [1.115,
                                   1.115],
                                [1.4449999999999998, 1.3800000000000003]],
                                [[1.6485000000000003, 1.6260000000000001],
                                [1.8495000000000001, 1.592],
                                   [1.3029999999999997, 1.348],
                                [1.2005000000000003, 1.293], [1.23, 1.32],
                                [1.3604999999999998, 1.3755000000000002],
                                [1.3800000000000003, 1.4449999999999998],
                                [1.4842500000000003, 1.4842500000000003]]])

        else:
            env = ablearn.environments.bimatrix_random_environment.
                BiMatrixRandomEnv(
                                number_of_agents=self.na, bimatrix=[[[1,1],
                                    [1,1]],
                                                                    [[1,1],

                                                                        [1,1]]])
```

```
        ga = ablearn.algorithms.genetic.Genetic(generations=self.ge, rounds_per_generation=self.rpg,
                                death_rate=self.dr, mutation_rate=self.mr, exploitation_rate=self.er)
        if self.isd == False:
            ga.assign_strategies(env.row_agents, env.row_strategies)
            ga.assign_strategies(env.col_agents, env.col_strategies)
        else:
            ga.assign_strategies_initial_distribution(env.row_agents, env.row_strategies, env.number_of_agents, self.isd)
            ga.assign_strategies_initial_distribution(env.col_agents, env.col_strategies, env.number_of_agents, self.isd)

        self.row_accumulated_strategies = [[] for r in range(len(env.row_strategies))]
        self.col_accumulated_strategies = [[] for c in range(len(env.col_strategies))]

        if len(env.row_strategies) == len(env.col_strategies):
            self.strategies_history = [[] for r in range(len(env.row_strategies))]
        else:
            longer = max(env.row_strategies, env.col_strategies)
            self.strategies_history = [[] for r in range(len(longer))]


    def run(self, plot=False, stack=False):
        if plot==True:
            pl.ion()
            pl.ylim(0, 1)
            pl.xlabel("Generations")
            pl.ylabel("Probability")
            for k in range(len(env.row_strategies)):
                c = cm.copper((k + 1) / len(env.row_strategies))
                pl.plot(self.row_accumulated_strategies[k], color=c, label="Row strategies: %s" % (k + 1))
            for k in range(len(env.col_strategies)):
                c = cm.winter((k + 1) / len(env.row_strategies))
                pl.plot(self.col_accumulated_strategies[k], color=c, label="Col strategies: %s" % (k + 1))
            #pl.legend(bbox_to_anchor=(0.7, 0.85), loc=2, prop={'size':7})
            pl.legend(loc=2, prop={'size':7})
            pl.draw()

```

```
        self.generations_passing(ga.generations, ga.rounds_per_generation, env.number_of_agents, env.row_agents, env.col_agents, plot, stack)

    def generations_passing(self, generations, rounds_per_generation, number_of_agents, row_agents, col_agents, plot, stack):
        for g in range(generations):
            print "\nGeneration %i of %i" % (g + 1, generations)
            for a in range(int(number_of_agents / 2)):
                row_agents[a].utility = 0
                col_agents[a].utility = 0
            for r in range(rounds_per_generation):
        #         print "\tRound %i of %i" % (r + 1, rounds)
                env.interact()
            self.distributions(env.number_of_agents, row_agents, col_agents, env.row_strategies, env.col_strategies, plot)

        if stack==True:
            b = range(len(self.strategies_history))
            for k in range(len(self.strategies_history)):
                b[k] = 'Strategy %r' % (k + 1)

            print self.strategies_history
            print b

            fig, ax = pl.subplots()
            stack_coll = ax.stackplot(range(1, generations + 1), self.strategies_history[0:], label=str(b[0:]))
            proxy_rects = [Rectangle((0, 0), 1, 1, fc=pc.get_facecolor()[0]) for pc in stack_coll]
            ax.set_xlabel('Generations')
            ax.set_xlim(1, generations)
            ax.set_ylabel('Agents')
            ax.set_title('Stackplot')
            ax.legend(proxy_rects, b)
            pl.show()

        if plot==True:
            pl.show(block=True)

    def proportion_classified_strategies(self, agents, strategies):
        frequency_per_strategy_per_agent = []
        for s in strategies:
            frequency_per_strategy_per_agent.append(0)
            for a in agents:
                if a.strategies == s:
```

```
                    frequency_per_strategy_per_agent[-1] += 1
        return [f / len(agents) for f in frequency_per_strategy_per_agent]

    def total_classified_strategies(self, row_agents, row_strategies,
        col_agents, col_strategies):
        frequency_per_strategy_for_sum = []

        if row_strategies >= col_strategies:
            for rs in row_strategies:
                frequency_per_strategy_for_sum.append(0)
                for ra in row_agents:
                    if ra.strategies == rs:
                        frequency_per_strategy_for_sum[rs] += 1
            for cs in col_strategies:
                for ca in col_agents:
                    if ca.strategies == cs:
                        frequency_per_strategy_for_sum[cs] += 1
        else:
            for cs in col_strategies:
                frequency_per_strategy_for_sum.append(0)
                for ca in col_agents:
                    if ca.strategies == cs:
                        frequency_per_strategy_for_sum[cs] += 1
            for rs in row_strategies:
                for ra in row_agents:
                    if ra.strategies == rs:
                        frequency_per_strategy_for_sum[rs] += 1

        return [f for f in frequency_per_strategy_for_sum]

    def distributions(self, number_of_agents, row_agents, col_agents,
        row_strategies, col_strategies,plot):
        self.row_strategies_distribution = self.
            proportion_classified_strategies(row_agents, row_strategies)
        for rs in range(len(row_strategies)):
             self.row_accumulated_strategies[rs].append(self.
                row_strategies_distribution[rs])
        self.col_strategies_distribution = self.
            proportion_classified_strategies(col_agents, col_strategies)
        for cs in range(len(col_strategies)):
             self.col_accumulated_strategies[cs].append(self.
                col_strategies_distribution[cs])

        self.total_strategies_per_generation = self.
            total_classified_strategies(row_agents, row_strategies,
            col_agents, col_strategies)
```

```
        for sh in range(len(self.strategies_history)):
            self.strategies_history[sh].append(self.total_strategies_per_generation[sh])

        print "\n Row players' strategy distribution:"
        print "\t", self.row_strategies_distribution
        print "\n Column players' strategy distribution:"
        print "\t", self.col_strategies_distribution

        ga.kill_agents(env.row_agents, env.number_of_agents, env.number_of_row_agents, env.row_strategies)
        ga.kill_agents(env.col_agents, env.number_of_agents, env.number_of_col_agents, env.col_strategies)
        print "Row agents eliminated", (number_of_agents / 2) - len(env.col_agents)
        print "Col agents eliminated", (number_of_agents / 2) - len(env.row_agents)
        ga.reproduce_agents(env.row_agents, env.number_of_agents, env.number_of_row_agents, env.row_strategies)
        ga.reproduce_agents(env.col_agents, env.number_of_agents, env.number_of_col_agents, env.col_strategies)

        if plot==True:
            for k in range(len(env.row_strategies)):
                c = cm.copper((k + 1) / len(env.row_strategies))
                pl.plot(self.row_accumulated_strategies[k], color=c, label="Row strategies: %s" % (k + 1))
            for k in range(len(env.col_strategies)):
                c = cm.winter((k + 1) / len(env.row_strategies))
                pl.plot(self.col_accumulated_strategies[k], color=c, label="Col strategies: %s" % (k + 1))
            pl.draw()
```

This module is used to generate a bimatrix from the matrix generated by the 'axelrod' package.

F.6 Matrix tranponser module

```
class Transform:
    """ This class is used to transform the output payoff matrix from
        Axelrod's package
    """

    """ INITIALIZATION """
    def __init__ (self, matrix):
        self.matrix = matrix
        self.building_big_list = [] # Creates an empty list where all the
            strategies will be placed in.
        self.sorting_into_lists = [[[] for m in range(len(matrix))] for r
            in range(len(matrix))] # Creates a list containing a list for
            each pair of strategies row and column.

    """  This function assigns the payoff for each row and column agent
        when interacting with each other."""
    def sorting_strategies(self):
        for a in range(len(self.matrix)): # Explores the number of row
            strategies payoffs in the matrix.
            for b in range(len(self.matrix)): # Explores the number of
                strategies payoffs for each row agent.
                r = self.matrix[a][b] # Separates each strategy payoff
                self.building_big_list.append(r) # Appends each strategy
                    payoff into a list that in the end will contain all the
                    available strategies payoffs

        e = 0
        for c in range(len(self.matrix)): # Iterates through each empty
            list created in sorting into lists variable which contain a list
             within itself.
            for d in range(len(self.matrix)): # Iterates through every
                empty list created within the lists in sorting into lists.
                s = self.building_big_list[e] # Obtains the corresponding
                    row value from building big list.
                self.sorting_into_lists[c][d].append(s) # Assigns the row
                    values.
                e += 1

        f = -(len(self.building_big_list))
```

```
        for c in range(len(self.sorting_into_lists)): # Iterates through each list that are now equivalent to a row in our bimatrix.
            for d in range(len(self.matrix)): # Iterates through each list that contains a row strategy payoff assigned previously.
                s = self.building_big_list[f] # Selects the corresponding column strategy payoff.
                self.sorting_into_lists[d][c].append(s) # Assigns the column strategy payoff.
                f += 1
        print self.sorting_into_lists # Prints our new Bimatrix
```

In this section the tests built for the different modules are presented.

F.7 Tests module

F.7.1 Simulation module test

```
import unittest
import ablearn
import random

class SimulationTesting(unittest.TestCase):
    def test_initialisation_initial_parameters_and_assigning_initial_strategies(self):
        env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(number_of_agents=50, bimatrix=[[[-1,1],[1,-1]], [[1,-1], [-1,1]]])
        ga = ablearn.algorithms.genetic.Genetic(generations=2, rounds_per_generation=5, death_rate=0.5, mutation_rate=0.1, exploitation_rate=0.1, initial_strategy_distribution=False)
        sim = ablearn.simulation.simulation.Simulation(tog='mp', na=50, ge=2, rpg=5, dr=0.5, mr=0.1, er=0.1)

        self.assertEqual(sim.tog, 'mp')
        self.assertEqual(sim.na, 50)
        self.assertEqual(sim.ge, 2)
        self.assertEqual(sim.rpg, 5)
        self.assertEqual(sim.dr, 0.5)
        self.assertEqual(sim.mr, 0.1)
        self.assertEqual(sim.er, 0.1)
        self.assertEqual(sim.row_accumulated_strategies, [[], []])
        self.assertEqual(sim.col_accumulated_strategies, [[], []])
        self.assertTrue(sim.strategies_history)

        for s in range(len(env.row_agents)):
            self.assertTrue(env.row_agents[s].strategies in env.row_strategies)
            self.assertTrue(env.col_agents[s].strategies in env.col_strategies)

        #self.assertEqual(env.row_agents[4].strategies, 1)
        #self.assertEqual(env.col_agents[1].strategies, 1)

        sim.run()
```

```
        self.assertNotEqual(env.row_agents[0].strategies, False)
        self.assertNotEqual(env.col_agents[0].strategies, False)


    def test_generations_passing_assert_possible_values_for_utilities_in_types_of_agents(self):
        env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(number_of_agents=100, bimatrix=[[[-1,1], [1,-1]], [[1,-1], [-1,1]]])
        ga = ablearn.algorithms.genetic.Genetic(generations=2, rounds_per_generation=5, death_rate=0.3, mutation_rate=0.5, exploitation_rate=0.9, initial_strategy_distribution=False)
        sim = ablearn.simulation.simulation.Simulation(tog='2', na=50, ge=2, rpg=5, dr=0.5, mr=0.1, er=0.1)

        ga.assign_strategies(env.row_agents, env.row_strategies)
        ga.assign_strategies(env.col_agents, env.col_strategies)


        sim.generations_passing(ga.generations, ga.rounds_per_generation, env.number_of_agents, env.row_agents, env.col_agents, plot=False, stack=False)
        env.interact()
        a = [env.row_agents[i].utility for i in range(len(env.row_agents))]
        b = [env.col_agents[i].utility for i in range(len(env.col_agents))]

        self.assertIn(1, a)
        self.assertIn(-1, a)
        self.assertIn(1, b)
        self.assertIn(-1, b)

    def test_proportion_classified_strategies_for_creating_lists_of_accumulated_strategi
    (self):
        env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(number_of_agents=100, bimatrix=[[[3,3], [0,5]], [[5,0], [1,1]]])
        ga = ablearn.algorithms.genetic.Genetic(generations=2, rounds_per_generation=1, death_rate=0.1, mutation_rate=0.1, exploitation_rate=0.9, initial_strategy_distribution=False)
        sim = ablearn.simulation.simulation.Simulation(tog='mp', na=100, ge=2, rpg=5, dr=0.5, mr=0.1, er=0.1)

        self.assertEqual(sim.row_accumulated_strategies, [[], []])
```

```
        self.assertEqual(sim.col_accumulated_strategies, [[], []])

        random.seed(1)
        ga.assign_strategies(env.row_agents, env.row_strategies)
        ga.assign_strategies(env.col_agents, env.col_strategies)
        sim.generations_passing(2, 2, 100, env.row_agents, env.col_agents, plot=False, stack=False)

        self.assertNotEqual(sim.row_accumulated_strategies, [[], []])
        self.assertNotEqual(sim.col_accumulated_strategies, [[], []])

    def test_total_classified_strategies_values_for_stackplot(self):
        env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(number_of_agents=100, bimatrix=[[[3,3], [0,5]], [[5,0], [1,1]]])
        ga = ablearn.algorithms.genetic.Genetic(generations=2, rounds_per_generation=1, death_rate=0.1, mutation_rate=0.1, exploitation_rate=0.9, initial_strategy_distribution=False)
        sim = ablearn.simulation.simulation.Simulation(tog='mp', na=100, ge=2, rpg=5, dr=0.5, mr=0.1, er=0.1)

        self.assertEqual(sim.strategies_history, [[], []])

        random.seed(1)
        ga.assign_strategies(env.row_agents, env.row_strategies)
        ga.assign_strategies(env.col_agents, env.col_strategies)
        sim.generations_passing(2, 2, 100, env.row_agents, env.col_agents, plot=False, stack=False)

        self.assertNotEqual(sim.strategies_history, [[], []])
        self.assertEqual(sim.strategies_history[0][0]+sim.strategies_history[1][0], 100)
        self.assertEqual(sim.strategies_history[0][1]+sim.strategies_history[1][1], 100)

    def test_distributions_not_empty_lists(self):
        env = ablearn.environments.bimatrix_random_environment.BiMatrixRandomEnv(number_of_agents=100, bimatrix=[[[-1,1], [1,-1]], [[1,-1], [-1,1]]])
        ga = ablearn.algorithms.genetic.Genetic(generations=2, rounds_per_generation=5, death_rate=0.5, mutation_rate=0.1, exploitation_rate=0.1, initial_strategy_distribution=False)
        sim = ablearn.simulation.simulation.Simulation(tog='2', na=50, ge=2, rpg=5, dr=0.5, mr=0.1, er=0.1)

        sim.run()
```

```
        ga.assign_strategies(env.row_agents, env.row_strategies)
        ga.assign_strategies(env.col_agents, env.col_strategies)
        env.interact()

        sim.distributions(env.number_of_agents, env.row_agents, env.
            col_agents, env.row_strategies, env.col_strategies, plot=False)

        self.assertNotEqual(sim.row_strategies_distribution, [[], []])
        self.assertNotEqual(sim.col_strategies_distribution, [[], []])
```

F.7.2 Agents module test

```
import unittest
import ablearn

class AgentTesting(unittest.TestCase):

    def test_initialisation(self):
        strategies = ['R', 'P', 'S']
        agent = ablearn.Agent(strategies)
        self.assertEqual(agent.strategies, strategies)
        self.assertEqual(agent.utility, 0)
        self.assertFalse(agent.label)

    def test_labelling(self):
        strategies = ['R', 'P', 'S']
        agent = ablearn.Agent(strategies, label=5)
        self.assertEqual(agent.strategies, strategies)
        self.assertEqual(agent.utility, 0)
        self.assertEqual(agent.label, 5)

    def test_increment_utility(self):
        strategies = ['R', 'P', 'S']
        agent = ablearn.Agent(strategies, label=5)
        self.assertEqual(agent.utility, 0)
        agent.increment_utility()
        self.assertEqual(agent.utility, 1)
        agent.increment_utility(5)
        self.assertEqual(agent.utility, 6)
        agent.increment_utility(-3)
        self.assertEqual(agent.utility, 3)
        agent.increment_utility(0)
        self.assertEqual(agent.utility, 3)
```

F.7.3 Environment module test

```python
import unittest
import ablearn
import random

class EnvironemntTesting(unittest.TestCase):
    def test_initialisation(self):
        bimatrix =[[[1, 2], [4, 5]], [[5, 1], [6, 1]], [[5, 1], [8, 2]]]
        env = ablearn.environments.BiMatrixRandomEnv(number_of_agents=20,
                bimatrix=bimatrix)
        self.assertRaises(AttributeError, lambda: len(env.agents))
        self.assertEqual(env.number_of_row_agents, 10)
        self.assertEqual(env.number_of_col_agents, 10)
        self.assertEqual(set([agent.__class__.__name__ for agent in
            env.row_agents]), set(['Agent']))
        self.assertEqual(set([agent.__class__.__name__ for agent in
            env.col_agents]), set(['Agent']))

    def test_strategies_to_utilities(self):
        bimatrix =[[[1, 2], [4, 5]], [[5, 1], [6, 1]], [[5, 1], [8, 2]]]
        env = ablearn.environments.BiMatrixRandomEnv(number_of_agents=20,
                bimatrix=bimatrix)
        test_strategies = [[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
        for strat in test_strategies:
            self.assertEqual(env.strategies_to_utilities(strat),
                    bimatrix[strat[0]][strat[1]])

    def test_randomly_pair_agents(self):
        bimatrix =[[[1, 2], [4, 5]], [[5, 1], [6, 1]], [[5, 1], [8, 2]]]
        env = ablearn.environments.BiMatrixRandomEnv(number_of_agents=20,
                bimatrix=bimatrix)
        pairs = env.randomly_pair_agents()
        self.assertEqual(len(pairs), env.number_of_agents / 2)
        self.assertTrue(all([set([pair[0]]) < set(env.row_agents) and
                        set([pair[1]]) < set(env.col_agents) for pair in
                            pairs]))
        self.assertEqual(set([pair[0] for pair in pairs]), set(env.
            row_agents))
        self.assertEqual(set([pair[1] for pair in pairs]), set(env.
            col_agents))

    def test_interact(self):
        bimatrix =[[[1, 2], [4, 5]], [[5, 1], [6, 1]], [[5, 1], [8, 2]]]
        env = ablearn.environments.BiMatrixRandomEnv(number_of_agents=20,
                bimatrix=bimatrix)

        for ra, ca in zip(env.row_agents, env.col_agents):
```

```
            ra.strategies = random.choice(env.row_strategies)
            ca.strategies = random.choice(env.col_strategies)
            self.assertEqual(ra.utility, 0)
            self.assertEqual(ca.utility, 0)

        env.interact()

        for ra, ca in zip(env.row_agents, env.col_agents):
            self.assertEqual(ra.utility, bimatrix[ra.strategies][ca.strategies][0])
            self.assertEqual(ca.utility, bimatrix[ra.strategies][ca.strategies][1])
```

F.7.4 Algorithm module test

```
import unittest
import ablearn
import random


class GATesting(unittest.TestCase):  #This class is to test the Genetic Algotihm

        def test_for_initialisation_without_initial_strategy_distribution(self):        #This method is to test the initial settings for the algorithm.
                generations = 1
                rounds_per_generation = 1
                death_rate = 0.1
                mutation_rate = 0.5
                exploitation_rate = 0.5
                choice = int(20 / 2 * (1 - exploitation_rate) + 0.05) - 1

                gen = ablearn.algorithms.genetic.Genetic(generations, rounds_per_generation,
                death_rate, mutation_rate, exploitation_rate)

                self.assertEqual(gen.generations, generations)
                self.assertEqual(gen.rounds_per_generation, rounds_per_generation)
                self.assertEqual(gen.death_rate, death_rate)
                self.assertEqual(gen.mutation_rate, mutation_rate)
                self.assertEqual(gen.exploitation_rate, exploitation_rate)

        def test_assign_strategies(self):
                generations = 1
```

```
27                  rounds_per_generation = 1
28                  death_rate = 0.1
29                  number_of_deaths_per_generation = int(death_rate * 10)
30                  mutation_rate = 0.5
31                  exploitation_rate = 0.5
32                  choice = int(20 / 2 * (1 - exploitation_rate) + 0.05) -1
33
34                  gen = ablearn.algorithms.genetic.Genetic(generations,
                        rounds_per_generation,
35                                                  death_rate, mutation_rate,
                                                        exploitation_rate)
36                  bimatrix =[[[1, 2], [4, 5]], [[5, 1], [6, 1]], [[5, 1], [8,
                         2]]]
37                  env = ablearn.environments.BiMatrixRandomEnv(
                        number_of_agents=20, bimatrix=bimatrix)
38
39                  random.seed(1)
40                  for ra, ca in zip(env.row_agents, env.col_agents):
41                          ra.strategy = 0
42                          ca.strategy = 0
43
44                  gen.assign_strategies(env.row_agents, env.row_strategies)
45                  gen.assign_strategies(env.col_agents, env.col_strategies)
46                  # Asserting row agents random strategies
47                  self.assertEqual(env.row_agents[2].strategy, 1)
48                  self.assertEqual(env.row_agents[1].strategy, 2)
49                  self.assertEqual(env.row_agents[0].strategy, 0)
50                  # Asserting col agents random strategies
51                  self.assertEqual(env.col_agents[3].strategy, 1)
52                  self.assertEqual(env.col_agents[2].strategy, 0)
53                  self.assertEqual(env.col_agents[1].strategy, 0)
54                  self.assertEqual(env.col_agents[0].strategy, 1)
55
56                  self.assertEqual(len(str(env.row_agents[0].strategy)), 1) #
                        (env.row_agents[0].strategy, 1)
57                  self.assertEqual(len(str(env.col_agents[0].strategy)), 1) #
                        (env.col_agents[3].strategy, 1)
58
59          def test_for_initialisation_without_initial_strategy_distribution(
                self):
60                  generations = 1
61                  rounds_per_generation = 1
62                  death_rate = 0.6
63                  mutation_rate = 0.5
64                  exploitation_rate = 0
65                  choice = int(20 / 2 * (1 - exploitation_rate) + 0.05) - 1
```

```
66
67                  gen = ablearn.algorithms.genetic.Genetic(generations,
                        rounds_per_generation,
68                                                  death_rate, mutation_rate,
                                                        exploitation_rate,
                                                        initial_strategy_distribution=
                                                        True)
69                  env = ablearn.environments.BiMatrixRandomEnv(
                        number_of_agents=20,
70                                                  bimatrix=[[[1, 2], [4, 5]], [[5,
                                                        1], [6, 1]], [[5, 1], [8, 2]]])
71
72                  self.col_stras=[0,1]
73                  self.row_stras=[0,1]
74                  self.col_stra_dist=[0.45,0.55]
75                  self.row_stra_dist=[0.1,0.9]
76                  self.number_of_agents=200
77                  self.number_of_row_agents=100
78                  self.number_of_col_agents=100
79                  self.col_numb_stra=[int(self.col_stra_dist[k]*self.
                        number_of_row_agents) for k in range(len(self.col_stras)
                        )]
80                  self.row_numb_stra=[int(self.row_stra_dist[k]*self.
                        number_of_col_agents) for k in range(len(self.row_stras)
                        )]
81
82                  gen.final_agent_strategies = []
83                  gen.assign_strategies_initial_distribution(env.row_agents,
                        self.row_stras,
84                                          self.number_of_agents, self.row_stra_dist)
85                  # Asserting row agents correct strategies distribution
86                  self.assertEqual(gen.final_agent_strategies.count(0), 10)
87                  self.assertEqual(gen.final_agent_strategies.count(1), 90)
88
89                  gen.final_agent_strategies = []
90                  gen.assign_strategies_initial_distribution(env.col_agents,
                        self.col_stras,
91                                          self.number_of_agents, self.col_stra_dist)
92                  # Asserting col agents correct strategies distribution
93                  self.assertEqual(gen.final_agent_strategies.count(0), 45)
94                  self.assertEqual(gen.final_agent_strategies.count(1), 55)
95
96          def test_kill_agents(self):
97                  generations = 1
98                  rounds_per_generation = 1
99                  death_rate = 0.6
```

```
            mutation_rate = 0.5
            exploitation_rate = 0
            choice = int(20 / 2 * (1 - exploitation_rate) + 0.05) - 1

            gen = ablearn.algorithms.genetic.Genetic(generations, rounds_per_generation,
                                death_rate, mutation_rate, exploitation_rate)


            env = ablearn.environments.BiMatrixRandomEnv(number_of_agents=20,
            bimatrix=[[[1, 2], [4, 5]], [[5, 1], [6, 1]], [[5, 1], [8, 2]]])

            random.seed(2)
            utilities = [25, 6, 8, 100, 12, 7, 9, 1000, 51, 60]
            for r in env.row_agents:r.utility=random.choice(utilities)
            for c in env.col_agents:c.utility=random.choice(utilities)

            # Before kill function number of agents is 10 for each.
            self.assertEqual(len(env.row_agents), 10)
            self.assertEqual(len(env.col_agents), 10)

            gen.kill_agents(env.row_agents, env.number_of_agents, env.number_of_row_agents, env.row_strategies)
            gen.kill_agents(env.col_agents, env.number_of_agents, env.number_of_col_agents, env.col_strategies)

            # After kill function with 0.6 death rate, number of agents is 4 for each.
            self.assertTrue(gen.number_of_deaths_per_generation)
            self.assertEqual(len(env.row_agents), 4)
            self.assertEqual(len(env.col_agents), 4)

        def test_reproduce_agents(self):
            generations = 1
            rounds_per_generation = 1
            death_rate = 0.6
            mutation_rate = 0.9 # set mutation rate at 0 to be able to test exploitation.
            exploitation_rate = 1 # exploitation set to 1 (100%) meaning the agent with highest utility will be reproduced.
            choice = int((20 / 2) * (exploitation_rate - 1) + 0.05) - 1

```

```
        gen = ablearn.algorithms.genetic.Genetic(generations, rounds_per_generation,
                                                 death_rate, mutation_rate, exploitation_rate)
        bimatrix =[[[1, 2], [4, 5]], [[5, 1], [6, 1]], [[5, 1], [8, 2]]]
        env = ablearn.environments.BiMatrixRandomEnv(number_of_agents=20,
        bimatrix=bimatrix)

        random.seed(2) # Assign utilities to the agents to be able to test.
        utilities = [25, 6, 8, 100, 12, 7, 9, 1000, 51, 60]
        for r in env.row_agents:r.utility=random.choice(utilities)
        for c in env.col_agents:c.utility=random.choice(utilities)

        gen.kill_agents(env.row_agents, env.number_of_agents, env.number_of_row_agents, env.row_strategies)
        gen.kill_agents(env.col_agents, env.number_of_agents, env.number_of_col_agents, env.col_strategies)

        self.assertEqual(len(env.row_agents), 4) # After kill command and before reproduce, length of row agents 10.
        self.assertEqual(len(env.col_agents), 4) # After kill command and before reproduce, length of row agents 10.

        gen.reproduce_agents(env.row_agents, env.number_of_agents, env.number_of_row_agents, env.row_strategies)
        gen.reproduce_agents(env.col_agents, env.number_of_agents, env.number_of_col_agents, env.col_strategies)

        self.assertEqual(len(env.row_agents), 10) # After reproduce command, length of row agents 10.
        self.assertEqual(len(env.col_agents), 10) # After reproduce command, length of column agents 10.

        # Testing for exploitation rate at 100% for row agents, last and second to last row agents have the same utility.
        self.assertEqual(env.row_agents[-1].utility, env.row_agents[-2].utility )
        # Testing for exploitation rate at 100% for column agents, last and second to last column agents have the same utility.
        self.assertEqual(env.col_agents[-1].utility, env.col_agents[-2].utility )
```

F.7.5 Matrix transponser module test

```
import unittest
import ablearn
import random

class TransformMatrixTesting(unittest.TestCase):
    def test_initialisation(self):
        """ Create 2 empty lists where we will place the values. The first one
            will contain all the values in the given bimatrix and the second one
            will contain a number of lists according the pairs of payoffs we expect
            to see once the values are sorted
        """

        """Use prisoners dilemma payoffs and place them in the same configuration
            as axelrod's package outputs the payoff values.
        """
        axl = ablearn.axl.matrix.Transform(matrix=[[3,0],[5,1]])

        self.assertEqual(axl.building_big_list, [])
        self.assertTrue(axl.sorting_into_lists)

    def test_sorting_strategies(self):
        """ Since it is a simple matrix it is know how the values are supposed
            to be shown. So we test that the values will be in the following
            position: [[[3, 3], [0, 5]], [[5, 0],[1, 1]]].
        """
        axl = ablearn.axl.matrix.Transform(matrix=[[3,0],[5,1]])

        axl.sorting_strategies()

        self.assertEqual(len(axl.building_big_list), 4)
        self.assertEqual(axl.building_big_list[0], 3)
        self.assertEqual(len(axl.sorting_into_lists), 2)
        self.assertEqual(len(axl.sorting_into_lists[0]), 2)

        self.assertEqual(axl.sorting_into_lists[0][0], [3,3])
        self.assertEqual(axl.sorting_into_lists[0][1], [0,5])
        self.assertEqual(axl.sorting_into_lists[1][0], [5,0])
        self.assertEqual(axl.sorting_into_lists[1][1], [1,1])
```

Bibliography

[1] Joel Watson. *Strategy: An introduction to game theory.* 2013.

[2] Robert Gibbons. *A primer in game theory.* Harvester Wheatsheaf, 1992.

[3] Nolan McCarty and Adam Meirowitz. *Political game theory: an introduction.* Cambridge University Press, 2007.

[4] Amnon Rapoport. Game theory: contributions to the study of human cognition. , 6(2):142–167, 1999.

[5] Andrew M Colman. Cooperation, psychological game theory, and limitations of rationality in social interaction. *Behavioral and brain sciences*, 26(02):139–153, 2003.

[6] M Hykšová. Several milestones in the history of game theory. *inJubiläenChance oder Plage*, pages 49–56, 2004.

[7] Emile Borel. La théorie du jeu et les équations intégrales à noyau symétrique. *Comptes Rendus de lAcadémie des Sciences*, 173(1304-1308):58, 1921.

[8] Emile Borel. On games that involve chance and the skill of the players. *Econometrica: Journal of the Econometric Society*, pages 101–115, 1953.

[9] J von Neumann. On the theory of parlor games. *Mathematische Annalen*, 100:295–320, 1928.

[10] John Von Neumann and Oskar Morgenstern. *Theory of games and economic behavior.* Princeton university press, 2007.

[11] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[12] Vincent Knight. Nash equilibrium. *Class Notes*, 1, 2014.

[13] Charles Darwin and Encyclopaedia Britannica. The origin of species by means of natural selection. 1872.

[14] Mark Pallen. *The rough guide to evolution.* Dorling Kindersley Ltd, 2009.

[15] Karl Sigmund. John maynard smith and evolutionary game theory. *Journal of theoretical biology*, 68:7–10, 2004.

[16] J Maynard Smith. Evolution and the theory of games: in situations characterized by conflict of interest, the best strategy to adopt depends on what others are doing. *American Scientist*, pages 41–45, 1976.

[17] John M McNamara and Franz J Weissing. Evolutionary game theory. *Social behaviour: Genes, ecology and evolution*, pages 109–33, 2010.

[18] Fiona Carmichael. *A guide to game theory*. Pearson Education, 2005.

[19] J Maynard Smith and GR Price. The logic of animal conflict. *Nature*, 246:15, 1973.

[20] Thomas L Vincent and Joel S Brown. *Evolutionary game theory, natural selection, and Darwinian dynamics*. Cambridge University Press, 2005.

[21] Larry Samuelson. *Evolutionary games and equilibrium selection*, volume 1. Mit Press, 1998.

[22] Alexander J McKenzie and Edward N Zalta. Evolutionary game theory. 2009.

[23] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[24] Martin J Osborne. *An introduction to game theory*, volume 3. Oxford University Press New York, 2004.

[25] Margaret Ann Boden. *Mind as machine: A history of cognitive science*. Oxford University Press, 2006.

[26] Dan Simon. *Evolutionary optimization algorithms*. John Wiley & Sons, 2013.

[27] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989, 1989.

[28] Mitchell Melanie. An introduction to genetic algorithms. *Cambridge, Massachusetts London, England, Fifth printing*, 3, 1999.

[29] Uri Wilensky and William Rand. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. MIT Press, 2015.

[30] Marco A Janssen and Elinor Ostrom. Empirically based, agent-based models. *Ecology and Society*, 11(2):37, 2006.

[31] Paul Rendell. A turing machine in conway's game life. *http://www. cs. ualberta. ca/˜ bulitko F*, 2, 2001.

[32] J Maynard Smith. The theory of games and the evolution of animal conflicts. *Journal of theoretical biology*, 47(1):209–221, 1974.

[33] Rober Axelrod. *The Evolution of Cooperation*. Basic Books, Inc., Publishers New York, 1984.

[34] Francesco C Billari, Thomas Fent, Alexia Prskawetz, and Jürgen Scheffran. Agent-based computational modelling: an introduction. In *Agent-Based Computational Modelling*, pages 1–16. Springer, 2006.

[35] Robert M Axelrod. *The complexity of cooperation: Agent-based models of competition and collaboration*. Princeton University Press, 1997.

[36] Joshua M Epstein. Agent-based computational models and generative social science [generative social science studies in agent-based computational modeling]. *Introductory Chapters*, 2007.

[37] Charlotte Bruun. Agent-based computational economics-an introduction. *Dept of Economics, Politics and Public Administration, Aaiborg University, Denmark*, 2004.

[38] Bhanu Prasad Pokkunuri. Object oriented programming. *ACM Sigplan Notices*, 24(11):96–101, 1989.

[39] Richard P. Ten Dyke and John C. Kunz. Experiences with object-oriented group support software development. *IBM Systems Journal*, 28(3):465–478, 1989.

[40] Francesco Luna and Benedikt Stefansson. *Economic Simulations in Swarm: Agent-based modelling and object oriented programming*, volume 14. Springer Science & Business Media, 2012.

[41] James G March. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991.

[42] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[43] Guido Van Rossum and Fred L Drake. *Python language reference manual*. Network Theory, 2003.

[44] Scott Chacon. *Pro git*. Apress, 2009.

[45] Docutils. restructuredtext markup syntax and parser component of docutils. Accessed 2015-09-08.

[46] et al Knight, Vincent. Axelrod. Accessed 2015-09-01.