Universiteit Utrecht

Faculty of Science

Symbolically Solving Two-Strategy Evolutionary Games in Python

Bachelor Thesis

Rinske Oskamp

Mathematics

Supervisor:

Dr. Artem Kaznatcheev
Utrecht University

June 13, 2024

Abstract

This thesis presents a tool for symbolically solving two-strategy evolutionary games. Evolutionary games are relevant in various application fields such as mathematical oncology, biology and economics. To develop this tool, the underlying mathematics of two-strategy evolutionary games were thoroughly examined, and the mathematical software system SageMath was addressed. Combining mathematical framework with SageMath's functionality, we created a program capable of symbolically solving evolutionary two-strategy games. In addition to the mathematical background and an introduction to SageMath, this thesis also includes a profound explanation of the code and examples of the application of the tool. Future directions for extending the tool to three-strategy games are also explored.

Contents

1 Introduction 1

2 Two-strategy evolutionary games 2
2.1 Replicator dynamics 2
2.2 The possible regimes 3
2.3 Solving a general game 3
2.4 Go vs. Grow 6
2.4.1 Constructing the payoff matrix 6
2.4.2 Solving the Go vs. Grow game 7

3 An introduction to SageMath 11
3.1 SageMath tutorial 11

4 Solving two-strategy games with SageMath 14
4.1 Input 14
4.2 Output 14
4.3 Customisation options in the code 15
4.3.1 Variables 15
4.3.2 Axis titles 15
4.3.3 Colours and titles of the regimes 16
4.4 Overview of the code 16
4.5 In-Depth Explanation of the code 17
4.5.1 Declaring input 17
4.5.2 Calculation of key values 17
4.5.3 Determining conditions for each regime 18
4.5.4 Plotting 20
4.5.5 Case: equilibrium independent of p 24

5 Case studies of using code 26
5.1 General game presenting as U vs. V 26
5.2 Go vs. grow game 27

6 Future directions 30

7 Conclusion 31

A Code 33

1 Introduction

One of the numerous applications of mathematics is in evolutionary game theory (EGT). EGT is a field that researches the interaction between organisms with different strategies or characteristics [1]. Unlike classic game theory, where participants choose the most optimal strategy, the strategies or traits in EGT are inherent to the organisms [1]. One significant application of EGT is in oncology, where cancer dynamics can be effectively described by evolutionary games. This field has contributed to the optimisation of adaptive cancer therapies [2]. Adaptive therapies adjust the medication doses based on the current state of the tumour, rather than following a predetermined schedule.

Gaining insight into these evolutionary games is achieved by investigating the behaviour of the game and how this behaviour changes when parameters are adjusted. There exist programming tools to help do this numerically for three-strategy games [3][4]. By numerically, we refer to demonstrating the behaviour of the game using specific numerical input values. In this thesis, we will develop a tool to solve such evolutionary games symbolically. This tool would visualise the regions the game presents certain behaviours, depending on the parameters of the game. This creates insight on the effect of these parameters.

Although EGT was originally developed in biological context, it now has applications in various fields such as economics and sociology. While we use an example from mathematical oncology in this thesis, the tool also can be applied on evolutionary games in different disciplines.

For this thesis project, we focused on solving two-strategy games. These describe dynamics, where there are two interacting populations. We developed a program that symbolically solves these evolutionary two-strategy games. We began by examining the mathematical foundations of these games. In Chapter 2, we will discuss how two-strategy games are mathematically solved and what it means to solve a game. We will analyse a general game and provide an example of an evolutionary game that describes cancer dynamics. To implement the code, we used SageMath, a software system that operates with a Python interface. Chapter 3 provides an introduction to SageMath, describing its functionalities and capabilities. In Chapter 4, we combine the concepts from the previous chapters to symbolically solve evolutionary two-strategy games using SageMath. We will explain our approach and the workings of the code. Subsequently, in Chapter 5 we will demonstrate how the code solves specific two-strategy games, using examples from Chapter 2. In the final chapter, Chapter 6, we will discuss potential future directions. We will briefly explore the possibility of extending the code to solve three-strategy games.

2 Two-strategy evolutionary games

In this chapter, we will examine two-strategy evolutionary games. We will explore what an evolutionary game is in biological and mathematical context. We will explain how to construct two-by-two payoff matrices for a game, what it means to solve a game symbolically and the methods used to do so. This will be done trough a general game and trough an example of an evolutionary game from oncology.

2.1 Replicator dynamics

Consider an environment with two populations, A and B. Let N_A and N_B denote the number of individuals in each population, where $p = \frac{N_A}{N_A+N_B}$. We use w_A and w_B to describe the offspring production rates for each population. The change in offspring can be described by the following equations:

$$\frac{dN_A}{dt} = w_A N_A, \tag{1}$$
$$\frac{dN_B}{dt} = w_B N_B. \tag{2}$$

We are interested in examining the change in the proportion p of population A:

$$\frac{dp}{dt} = \frac{d}{dt}\left[\frac{N_A}{N_A+N_B}\right]. \tag{3}$$

From the quotient rule follows [5]:

$$\frac{dp}{dt} = \frac{\frac{dN_A}{dt}(N_A+N_B) - N_A\frac{d}{dt}[N_A+N_B]}{(N_A+N_B)^2} \tag{4}$$
$$= \frac{\frac{dN_A}{dt}}{N_A+N_B} - \frac{N_A\frac{d}{dt}[N_A+N_B]}{(N_A+N_B)^2} \tag{5}$$
$$= \frac{w_A N_A}{N_A+N_B} - \frac{N_A}{N_A+N_B}\frac{w_A N_A + w_B N_B}{N_A+N_B}. \tag{6}$$

Recall $p = \frac{N_A}{N_A+N_B}$ and thus $(1-p) = \frac{N_B}{N_A+N_B}$, so follows

$$= w_A p - p(w_A p + (1-p)w_B) \tag{7}$$
$$= p(1-p)(w_A - w_B). \tag{8}$$

We define $\dot{p} = p(1-p)(w_A - w_B)$ as the replicator dynamics.

The interaction between population A and population B can be interpreted as a game, where A and B are different strategies and w_A and w_B are the payoff functions for these strategies. Such a game is called an evolutionary game.

2.2 The possible regimes

The replicator dynamics of an evolutionary game can exhibit different behaviours on the interval $p \in (0, 1)$. The system described by $\dot{p}$ can increase or decrease at different sections of the interval. In games with two strategies these behaviours come down to four situations, which we will call dynamic regimes. These regimes are visualised in Figure 1. In this figure an increase of the system is shown by an upward pointing arrow, while a decrease is shown by a downward pointing arrow.

In the left most regime the first strategy is stable. In biological context this means that eventually the proportion of the first population will become one, so the second population will die out. In the second regime in Figure 1 the second strategy is stable, so the first population will eventually die out. In the third regime of Figure 1 both of the strategies are unstable. This would mean the two populations would find stability with a proportion of $p = p^* \in (0, 1)$ for the first population. In the last regime of Figure 1 both of the strategies could become stable and the dynamics bifurcate around a proportion of p^*.

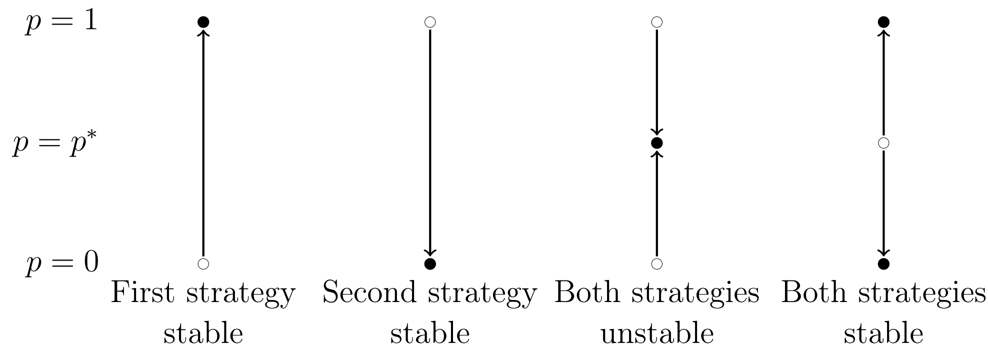


Figure 1: The four possible dynamic regimes in two-strategy evolutionary games. The open dots present unstable point and the black dots present stable points. The arrow defines if the system, described by the replicator dynamics, increases or decreases.

2.3 Solving a general game

The simplest form of the replicator dynamics $\dot{p}$ is when w_A and w_B are constant, then $\dot{p}$ describes exponential growth. We will look into what happens when w_A and w_B are linear. We define w_A and w_B based on functions R, S, T and F.

$$w_A = pR + (1 - p)S, \tag{9}$$
$$w_B = pT + (1 - p)F. \tag{10}$$

We can denote these as a matrix: $M = \left(\begin{smallmatrix} R & S \\ T & F \end{smallmatrix}\right)$. As mentioned in Section 2.1, this matrix can be interpreted as a payoff matrix for a game, where w_A and w_B are the payoff functions for each strategy. We can assume $R > F$ [6]. If we take a closer look we see that there are only two interesting variables. We can subtract F from every element without changing the structure of the game. In practice, this would mean that each time the game is played, it

collects an entry fee F for both players [6]. Next, we can rescale the matrix by dividing by $R-F$, this once again has no effect on the structure of the game.

$$\begin{pmatrix} R & S \\ T & F \end{pmatrix} = \begin{pmatrix} R-F & S-F \\ T-F & 0 \end{pmatrix} \tag{11}$$

$$= \begin{pmatrix} 1 & \frac{S-F}{R-F} \\ \frac{T-F}{R-F} & 0 \end{pmatrix}. \tag{12}$$

We introduce two new variables, $U = \frac{S-F}{R-F}$ and $V = \frac{T-F}{R-F}$. This results in a general matrix

$$\begin{pmatrix} 1 & U \\ V & 0 \end{pmatrix}. \tag{13}$$

The payoff functions for each strategy are given by $w_A = p + (1-p)U$ and $w_B = pV$, with p the proportion of population A. recall the replicator dynamics are defined by $\dot{p} = p(1-p)(w_A - w_B)$. We want to determine when each of the regimes explained in Section 2.2 occurs. It is apparent that the system increases when $\dot{p}$ is positive and $w_A > w_B$. The system decreases when $w_A < w_B$. Before we look into this, we will first examine for which value of p there is an equilibrium.

$$w_A = w_B, \tag{14}$$

$$p + (1-p)U = pV, \tag{15}$$

$$p = \frac{U}{U+V-1}. \tag{16}$$

We denote this equilibrium value by $p^* = \frac{U}{U+V-1}$. Next, we will determine the conditions under which $\dot{p}$ will be positive or negative. As we have stated, $\dot{p}$ is positive when $w_A > w_B$.

$$w_A > w_B, \tag{17}$$

$$p + (1-p)U > pV, \tag{18}$$

$$p(U+V-1) > U. \tag{19}$$

This inequality has two solutions. The conditions under which $\dot{p}$ is positive are:

$$\left(U+V<1,\, p > \frac{U}{U+V-1} = p^*\right), \tag{20}$$

$$\left(U+V>1,\, p < \frac{U}{U+V-1} = p^*\right). \tag{21}$$

In the same way we can determine that $\dot{p}$ is negative under the following conditions:

$$(U+V>1,\, p<p^*), \tag{22}$$

$$(U+V<1,\, p>p^*). \tag{23}$$

As mentioned earlier, p is the proportion of population A, so $0 < p < 1$. For the third and fourth regime in Figure 1 the equilibrium p^* lies in this interval. Therefore, we need to determine for which values of U and V p^* lies within or outside this interval. For $0 \leq p^* \leq 1$, there are two solutions:

$$(U \geq 0,\, V \geq 1), \tag{24}$$
$$(U \leq 0,\, V \leq 1). \tag{25}$$

$p^* < 0$ and $p^* > 1$ also have two solutions each. For $p^* < 0$ these are

$$(U > 0,\, U + V < 1), \tag{26}$$
$$(U < 0,\, U + V > 1). \tag{27}$$

For $p^* > 1$ these are

$$(U > 0,\, V < 1,\, U + V > 1), \tag{28}$$
$$(U < 0,\, V > 1,\, U + V < 1). \tag{29}$$

In our general game, we observe that the regime where both strategies are unstable, emerges when $\dot{p}$ is positive while $p < p^*$ and when $\dot{p}$ is negative when $p > p^*$. This corresponds with the third regime in Figure 1; the system decreases at the top where $p > p^*$ and increases at the bottom where $p < p^*$. In this case $0 \leq p^* \leq 1$ must hold. We saw $\dot{p} > 0$ while $p < p^*$ holds when $U + V > 1$ (21) and $\dot{p} < 0$ while $p < p^*$ holds when $U + V < 1$ (23), lastly $U \geq 0$ and $V \geq 1$ (24) or $U \leq 0$ and $V \leq 1$ (25) must hold for $0 \leq p^* \leq 1$. These restrictions combined result in the red area in figure 2.

We move on to the next regime, where both of the strategies are stable. For this regime $\dot{p} > 0$ when $p > p^*$ and $\dot{p} < 0$ when $p < p^*$. This corresponds with the fourth regime in Figure 1, and with equations (20) and (22). For this strategy $0 \leq p^* \leq 1$ also must hold so $U \geq 0$ and $V \geq 1$ (24) or $U \leq 0$ and $V \leq 1$ (25) apply again. These restrictions combined result in the blue area in figure 2.

In the third regime $\dot{p}$ always stays positive for all p in the interval $(1, 0)$. If $\dot{p}$ is dependent on p this means that besides $\dot{p} > 0$, either $p > p^*$ (20) and $p^* < 0$ (26,27) or $p < p^*$ (21) and $p^* > 1$ (28, 29). This can be visualised by imagining the black dot in the third regime in Figure 1 lays above $p = 1$ or the open dot in the last regime lays below $p = 0$. The mentioned restrictions combined result in the green area in figure 2. Note that this are actualy two distinct area's; above and below the $U + V = 1$ line.

The last regime we explore is when the second strategy is stable. The restrictions of this regime can be determined similarly to the previous regime. This results in the yellow area of Figure 2. This are again two distinct area's.

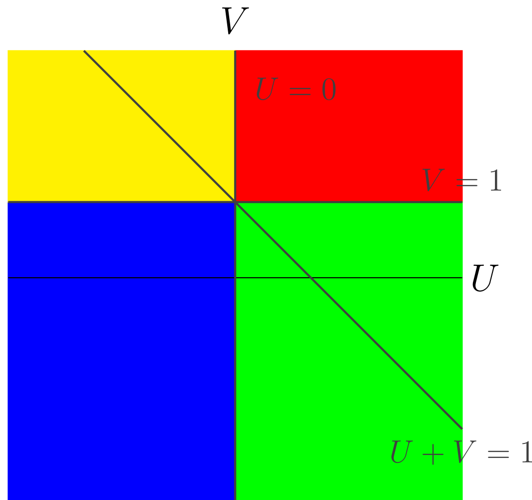


Figure 2: In this image it is shown when the four regimes, from Figure 1 occur for the general U vs. V game. Green correspond to the first regime, yellow to the second, red to the third and blue to the fourth. The grey lines denote the different area's.

2.4 Go vs. Grow

We will examine a specific evolutionary cancer game known as the "Go vs. Grow" game. In this context, the environment represents a tumor containing two types of cancer cells: cells capable of autonomous growth (AG) and cells with invasive characteristics (INV), enabling movement to different locations. In a medical context, these are cancer cells that either grow rapidly or have the potential to metastasise.

2.4.1 Constructing the payoff matrix

Two key parameters define this game: c represents the cost for movement for invasive (INV) cells, while b represents the maximum payoff attainable by a tumour cell under ideal conditions where resources like space or nutrients are limitless and not shared. We assume these payoffs are both non zero.

To construct the payoff matrix we analyse the outcomes when two cells interact. When two invasive cells encounter each other, one cell remains in place while the other relocates to alternative resources. The stationary cell receives a payoff of b, while the moving cell receives a payoff of $b-c$. It is assumed each cell has an equal probability of either remaining stationary or moving. Consequently, the payoff resulting from the interaction of two invasive cells is the average between the payoff for staying b and the payoff for moving $b-c$. When two growth cells interact, they must share the available resources, resulting in a payoff of $\frac{1}{2}b$. If a growth cell encounters a invasive cell, the invasive cell will seek new resources with a payoff of $b-c$, the growth cell will have a payoff of b. This analysis yields the following payoff matrix, with INV representing the first strategy and AG representing the second strategy [7].

$$\begin{array}{c} \\ \text{INV} \\ \text{AG} \end{array} \begin{array}{c} \begin{array}{cc} \text{INV} & \text{AG} \end{array} \\ \begin{pmatrix} \frac{1}{2}b + \frac{1}{2}(b-c) & b-c \\ b & \frac{1}{2}b \end{pmatrix} \end{array}. \tag{30}$$

We can simplify this matrix by rescaling, dividing every component by b. Note that we have assumed $b \neq 0$. This results in a more simple matrix, and shows that the most important parameter of the game is the ratio between c and b.

$$\begin{array}{c} \\ \text{INV} \\ \text{AG} \end{array} \begin{array}{c} \begin{array}{cc} \text{INV} & \text{AG} \end{array} \\ \begin{pmatrix} 1 - \frac{1}{2}\frac{c}{b} & 1 - \frac{c}{b} \\ 1 & \frac{1}{2} \end{pmatrix} \end{array}. \tag{31}$$

2.4.2 Solving the Go vs. Grow game

To solve the Go vs. Grow game we start by formulating the payoff functions w_{go} and w_{grow}.

$$w_{go} = p\left(1 - \frac{1}{2}\frac{c}{b}\right) + (1-p)\left(1 - \frac{c}{b}\right) \tag{32}$$

$$= \frac{1}{2}\frac{c}{b}p - \frac{c}{b} + 1, \tag{33}$$

$$w_{grow} = p + (1-p)\frac{1}{2} \tag{34}$$

$$= \frac{1}{2}p + \frac{1}{2}. \tag{35}$$

We determine for which value of p there is an equilibrium.

$$w_{go} = w_{grow}, \tag{36}$$

$$\frac{1}{2}\frac{c}{b}p - \frac{c}{b} + 1 = \frac{1}{2}p + \frac{1}{2}, \tag{37}$$

$$p\left(\frac{1}{2}\frac{c}{b} - \frac{1}{2}\right) = -\frac{1}{2} + \frac{a}{b}, \tag{38}$$

$$p = \frac{2c-b}{c-b}. \tag{39}$$

We denote this equilibrium value as $p^* = \frac{2c-b}{c-b}$. Next, we look at the conditions for $w_{go} > w_{grow}$.

$$\frac{1}{2}\frac{c}{b}p - \frac{c}{b} + 1 > \frac{1}{2}p + \frac{1}{2}, \tag{40}$$

$$p\left(\frac{1}{2}\frac{c}{b} - \frac{1}{2}\right) > \frac{c}{b} - \frac{1}{2}, \tag{41}$$

$$p\left(\frac{c-b}{2b}\right) > \frac{c}{b} - \frac{1}{2}, \tag{42}$$

$$p\left(\frac{c-b}{2b}\right) > \frac{2c-b}{2b}. \tag{43}$$

Note that for $b = c$, the inequality is non valid, so we assume $b \neq c$. Let us first consider the case where $b > 0$:

$$p(c-b) > 2c - b. \tag{44}$$

If $c > b$, the inequality can be solved as follows:

$$p > \frac{2c - b}{c - b} = p^*. \tag{45}$$

If $c < b$, $p < p^*$ follows. We now assume $b < 0$; then follows:

$$p(c - b) > 2c - b. \tag{46}$$

Which has solutions $p < p^*$ for $c > b$ and $p > p^*$ for $c < b$. To summarise, $w_{go} > w_{grow}$ has four solutions:

$$(b > 0,\ c > b,\ p > p^*), \tag{47}$$
$$(b < 0,\ c < b,\ p > p^*), \tag{48}$$
$$(b > 0,\ c < b,\ p < p^*), \tag{49}$$
$$(b < 0,\ c > b,\ p < p^*). \tag{50}$$

Similarly, we can determine the solutions for $w_{grow} > w_{go}$. This inequality also has four solutions:

$$(b > 0,\ c < b,\ p > p^*), \tag{51}$$
$$(b < 0,\ c > b,\ p > p^*), \tag{52}$$
$$(b > 0,\ c > b,\ p < p^*), \tag{53}$$
$$(b < 0,\ c < b,\ p < p^*). \tag{54}$$

We will now determine for which constrictions on b and c the different dynamic regimes occur. Recall Section 2.2 for an explanation of the four regimes.

First, we state when only the Go strategy is stable. This happens when $w_{go} > w_{grow}$ over the whole interval $p \in (0, 1)$. For the conditions (47) and (48) this would mean $p^* < 0$, because then $p > p^*$ always holds for $p \in (0, 1)$.

$$p^* < 0, \tag{55}$$
$$\frac{2c - b}{c - b} < 0, \tag{56}$$
$$\left(b > 0,\ \frac{b}{2} < c < b\right) \bigvee \left(b < 0,\ b < c < \frac{b}{2}\right). \tag{57}$$

Note these solutions for $p^* < 0$ contradict with (47) and (48). The situation where the equilibrium lies under zero does not occur. However, this regime also could occur when (49) and (50) hold, while $p^* > 1$.

$$p^* > 1, \tag{58}$$
$$\frac{2c - b}{c - b} > 1, \tag{59}$$
$$(b > 0,\ c > b) \bigvee (b > 0,\ c < 0) \bigvee (b \leq 0,\ c > 0) \bigvee (b \leq 0,\ c < b). \tag{60}$$

If we combine these restrictions with (49) and (50), we find that the Go strategy is stable and the Grow strategy is unstable for ($c < b$, $b > 0$, $c < 0$) or ($c > b$, $b < 0$, $c > 0$). Note $c < b$ and $c > b$ are implied by the other restrictions. This corresponds with the green area below in Figure 3a.

We will now examine the dynamic where only the Grow strategy is stable. This happens when $w_{grow} > w_{go}$ over the whole interval $p \in (0, 1)$. We apply the same method as above. First, we combine solutions (51) and (52) with $p^* < 0$, we find conditions $b > 0$, $\frac{b}{2} < c < b$ and $b < 0$, $b < c < \frac{b}{2}$. Thereafter we combine (53) and (54) with $p^* > 1$ and find $b > 0$, $c > b$ and $b < 0$, $c < b$. These four regions corresponds with the yellow area below in Figure 3b.

Lastly, we examine the dynamics with mixed strategies. When $w_{go} > w_{grow}$ for $p < p^*$, while $w_{grow} > w_{go}$ for $p > p^*$, there is a stable point in the equilibrium $p = p^*$. This happens exactly when $b > 0$ and $c < b$ (49, 51) or when $b < 0$ and $c > b$ (50, 52). In addition, p^* has to lie in the interval $(0, 1)$. First, we consider $p^* < 1$:

$$p^* < 1, \tag{61}$$

$$\frac{2\frac{c}{b} - 1}{\frac{c}{b} - 1} < 1, \tag{62}$$

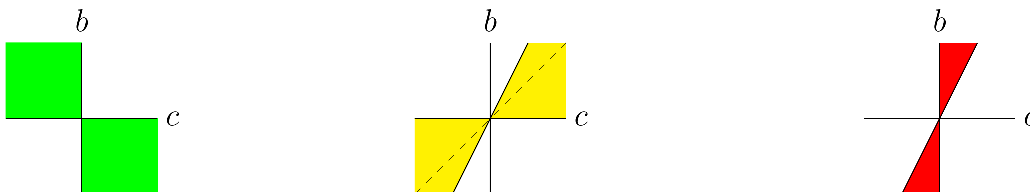
$$2c - b < c - b, \tag{63}$$

$$(b > 0,\ 0 < c < b) \bigvee (b < 0,\ b < c < 0). \tag{64}$$

Considering $p^* > 0$ as well, we obtain the following constraints for a and b, such that $0 < p^* < 1$:

$$(b > 0,\ 0 < c < \frac{b}{2}) \bigvee (b < 0,\ \frac{b}{2} < c < 0) \tag{65}$$

We can conclude that there is a stable point in the equilibrium for $b > 0$ and $0 < c < \frac{b}{2}$ and for $b < 0$ and $\frac{b}{2} < c < 0$. This corresponds with the red area in Figure 3c.



(a) The area's where the Go strategy is stable.

(b) The area's where the Grow strategy is stable.

(c) The area's where neither of the strategies is stable and there is a stable point in $p = p^*$.

Figure 3: The three occurring regimes of the Go vs. Grow game.

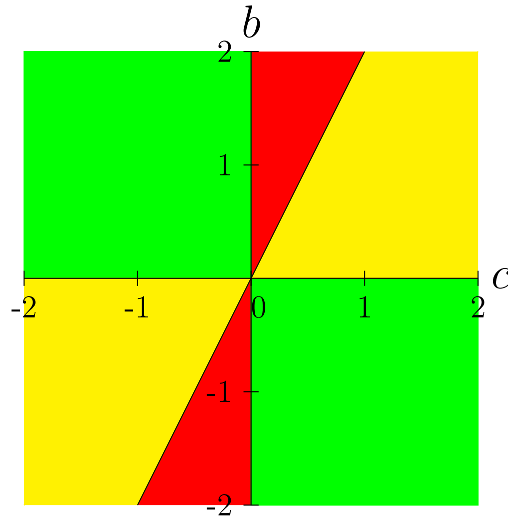


Figure 4: This image shows the regime plot for the Go vs. Grow game. The red area corresponds with the regime where both the Grow and Go strategy are unstable and there is a stable point in $p = p^*$. The Go cells are evolutionary stable in the green area and the Grow cells are evolutionary stable in the yellow area.

The final regime we will address, is the regime where both strategies are stable and the regime bifurcates around $p = p^*$. For this regime $w_{grow} > w_{go}$ for $p < p^*$, while $w_{go} > w_{grow}$ for $p > p^*$, both have to hold. This comes down to $b > 0$ and $c > b$ (47, 53) or $b < 0$ and $c < b$ (48, 54). But, for this regime p^* also has to be in the interval $(0, 1)$. The inequalities that define $p^* < 1$ (64) are in contradiction with the conditions for $w_{grow} > w_{go}$ for $p < p^*$, while $w_{go} > w_{grow}$ for $p > p^*$. We can conclude that in the Go vs. Grow game there are only three regimes. This result forms the dynamic regime plot in Figure 4.

3 An introduction to SageMath

The goal of this project was to develop a Python tool for symbolically solving two-strategy games. To achieve this, it was evident that we would need mathematical libraries. The most well-known and widely used mathematical library is Numpy. However, we quickly realised Numpy was insufficient for our project because it cannot solve equations symbolically. The next step was to explore Sympy, a library for symbolic mathematics. With Sympy, we were able to solve certain game matrices. However, Sympy is only able to solve inequalities in a single variable. This meant a program using Sympy could just be used to solve game matrices dependent on a single variable. The matrices from Chapter 2, could not be solved using this program. We continued seeking a way to preform more advanced calculations. In this search, we discovered SageMath. SageMath provided the necessary computational power to solve two-strategy games involving multiple variables.

SageMath proved to be a very powerful and useful mathematical software system. It integrates various existing open-source packages such as Sympy, Numpy, SciPy, and Matplotlib into a single interface [8]. Each of these packages, along with many others, has its own strengths but also certain limitations. By combining their strong aspects, SageMath becomes an exceptionally powerful mathematical tool.

SageMath excels in symbolic mathematics, offering extensive support for solving equations and inequalities, simplifying expressions, and performing symbolic integration and differentiation. For this thesis project, we primarily utilised its capabilities to solve equations and systems of inequalities.

3.1 SageMath tutorial

In this section, we will explore the features of SageMath by some examples. Sagemath works via the Python interface, so it mostly uses the usual Python commands. It uses "=" for assignment and "==", ">", "<", "<=" and ">=" for comparison, just like in Python. Some basic mathematical operations could look something like this:

```
1 sage: a = 2+3
2 5
3 sage: b = (2 + 4)/sqrt(7)
4 6/7*sqrt(7)
```

We will now discuss some functions provided by SageMath that are relevant for solving two-strategy evolutionary games. In almost every program where algebra or calculus are being used, the *expand* function proves to be useful. This function expands the brackets of an expression. In context of two-strategy games, this would be used, among other things, to define payoff functions. To calculate these functions it is necessary to first define relevant variables. Defining the payoff functions from the Go vs. Grow game from Section 2.4 in code would result in the lines below. To ensure $\frac{1}{2}$ is treated as an exact value and not as 0.5 we define a variable h for a half.

```
sage: c, b , p = var('c b p')
sage: h = Integer(1)/Integer(2)
sage: w_go = p*(1-h*c/b)+(1-p)*(1-c/b)
sage: w_go = w_go.expand()
1/2*c*p/b - c/b + 1
sage: w_grow = p*1+(1-p)*h
sage: w_grow = w_grow.expand()
1/2*p + 1/2
```

SageMath has a *solve* function to solve equations. This function takes two arguments: the equation you want solved and the variables for which you seek the solutions. If no value is provided to which the equation should be equal, it is solved for zero. Applying this method to solve the equation $w_{go} = w_{grow}$ (37) from our "Go to Grow" game yields the following result:

```
sage: solve(w_go==w_grow, p)
[p == (b - 2*c)/(b - c)]
```

The output of this *solve* function is a list with all the solutions of the equations. SageMath can also solve much more complex equations, but these are not relevant for this project. What is relevant, however, is solving inequalities. This works more or less the same as solving equations. We will apply this to inequalities $p^* < 0$ and $w_{go} > w_{grow}$ from Section 2.4.2 of last chapter.

```
sage: c, b , p, u, v = var('c b p u v')
sage: solve(1/2*c/b*p-c/b+1>1/2*p+1/2, p)
[[0 < b, -b*p + c*p + b - 2*c > 0], [b < 0, b*p - c*p - b + 2*c > 0]]
sage: solve(u/(u+v-1)<0, u, v)
[[-u + 1 < v, u < 0], [v < -u + 1, 0 < u]]
```

The output for the *solve* function is again a list with all the solutions of the inequality. All restrictions of a single solution are compiled into a list; thus, the output is a list of lists. For both equations and inequalities SageMath is also able to solve systems of multiple conditions. In case of the "Go vs Grow" game you could be interested in multiple conditions like $w_{go} > w_{grow}$ and $p^* < 0$ from Section 2.4.2.

```
sage: solve([1/2*c/b*p-c/b+1>1/2*p+1/2, (2*c/b-1)/(c/b-1)<0], b,c)
[[c < b, b < 2*c, 0 < c, -b*p + c*p + b - 2*c > 0],
[2*c < b, b < c, c < 0, b*p - c*p - b + 2*c > 0]]
```

In this thesis project, we have frequently discussed matrices. Therefore, it is useful to examine how SageMath can perform calculations on matrices.

```
23 sage: Go_Grow = Matrix([[1/2*b+1/2*(b-c), b-c], [b, 1/2*b]])
24 sage: M = Matrix([[2, 0], [1, 3]])
25 sage: Go_Grow + M
26 [b - 1/2*c + 2         b - c]
27 [        b + 1     1/2*b + 3]
28 sage: Go_Grow * M
29 [3*b - 2*c 3*b - 3*c]
30 [    5/2*b     3/2*b]
```

The examples provided above have hopefully given a clear understanding of what SageMath can do. In the next chapter, we will apply these functions to symbolically solve two-strategy evolutionary games.

4 Solving two-strategy games with SageMath

In this chapter, we will demonstrate how the SageMath functions discussed in Chapter 3 can be utilised to symbolically solve two-strategy evolutionary games. By solving, we refer to generating a plot that visually represents the regions in which the game exhibits a particular dynamic regime. In addition to generating this plot, it is also possible to obtain important values and calculations like discussed in Chapter 2. For instance, the equilibrium value p^* or the replicator dynamics $\dot{p}$ from an input game, can easily be retrieved.

To demonstrate how the code can be used and how the code works we begin by explaining the required format for the code input. Next, we describe the output the code produces. We also specify which values in the code can be adjusted to customise it. Then, we explain how the code works, first providing an overview and then detailing the entire code step by step. The complete code can be found in Appendix A.

4.1 Input

The input consists of four expressions, R, S, T and F, each dependent on a maximum of two variables, a and b. Which together form a matrix $\left(\begin{smallmatrix} R & S \\ T & F \end{smallmatrix}\right)$, like described in Section 2.3. These expressions are requested when you run the code. Alternatively, they can be defined directly in the code on lines 84-87. Make sure the input lines that are not used are removed or commented-out. The allowed input expressions are of the form:

$$\lambda a^x + \gamma b^y \text{ or } \lambda a^x \times \gamma b^y, \tag{66}$$

with $\lambda, \gamma \in \mathbb{R}$ and $x, y \in \{-1, 0, 1\}$.

4.2 Output

The output of the code will be a plot with a on the x axis and b on the y axis. This plot shows the different regions in which different dynamic regimes occur. The form of the output is a *.png* image which is saved on the user's computer. In the plot the regimes are labelled one to four, these correspond with the regimes described from left to right in Figure 1 in Section 2.2. The first regime is coloured green and refers to the regime where the first strategy is stable. The second regime is coloured yellow and refers to the regime where the second strategy is stable. The third regime is coloured red and refers to the regime where both strategies are unstable. The last regime is coloured blue and refers to the regime where both strategies are unstable.

To ensure the output file is saved correctly the user has to adjust line 283 of the code to the correct location and file name.

```
282 #Change to the correct location and name
283 P.save("<location>/<file_name>.png")
```

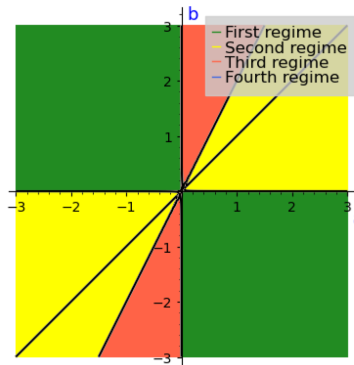


Figure 5: The output plot generated by the code when provided with the Go vs. Grow game matrix from Section 2.4 as input.

In Section 2.4 we solved the Go vs. Grow game. The output that is generated for this input game is presented in Figure 5. In Section 5.2, we will discuss how this image is generated.

4.3 Customisation options in the code

In the code, there are various variables that can be adjusted to customise the program. We will go through these variables and examine the effects they have.

4.3.1 Variables

In the current code, the main variables provided for solving the game are a and b. It is possible to have the input expressions depend on other variable names, this can be done as follows: Start by defining your variable names as SageMath variables by changing a and b to your variables in the following code line.

```
70  a, b, p = var('a b p')
```

In the remainder of the code a and b are frequently used as parameters in functions for solving or plotting. Instead of modifying all instances, it is much faster and easier to simply add "$a =$ *your_first_variable*" and "$b =$ *your_second_variable*" after declaring the input in lines 77 to 87. Note that the intermediate solutions will still contain a and b instead of the custom variables.

4.3.2 Axis titles

If the variables are adjusted, it may also be logical to change the axis titles. This can be easily done by adjusting the variables in lines 90 and 91.

```
89  #Adjust to variable names
90  x_titel = "c"
91  y_titel = "b"
```

The axis titles can be further customised in lines 273 and 274. Here, you can adjust the position and font size.

```
273 tx = text(x_titel, (scale+0.2,-0.5), fontsize=13)
274 ty = text(y_titel, (0.2,scale+0.2), fontsize=13)
```

4.3.3 Colours and titles of the regimes

For the regime plot the following colours provided by SageMath are used: “Forestgreen”, “Yellow”, “Tomato” and “Royalblue”. The colours can be customised by modifying the colour for the plot. Adjusting the colour for the plotted region is done by changing the value for *incol*. Modifying the colour for the legend can be done by changing the value for *color*. For the first regime, the colour is defined in lines 242 and 246. The lines for the remaining regimes can be found in Appendix A. SageMath offers dozens of named colours [9] and the ability to define custom colours [10].

```
240     #First Regime
241     for sol in plotlist1:
242         P += region_plot(sol, (a,-scale,scale),(b,-scale,scale), incol = 'tomato',
243         bordercol='black')
244
245     #Add a dummy plot for the legend
246     P += line([(0,0)], color='tomato', legend_label='First regime')
```

In the same line as the colour for the legend, the title of the respective regime can also be adjusted by modifying *legend_ label*.

4.4 Overview of the code

In this section, we will give a brief overview of the structure of the code. The first step is to define the input functions of the game we want to solve and convert them into Sage expressions (Section 4.5.1). Next, important values such as the payoff functions and the equilibrium value are calculated (Section 4.5.2), recall Chapter 2 for the definition of these values. The equilibrium value of the game may or may not depend on p. The code first handles the case where the equilibrium is not dependent on p. In this scenario, only two regimes are possible. The code determines for which restrictions on a and b each of the two regimes occur and plots the corresponding area’s with the appropriate colours (Section 4.5.5). For the remaining cases, where the equilibrium does depend on p, the code identifies the restrictions on a and b for which the game exhibits each regime. These conditions are placed in a list (Section 4.5.3). Before we plot these conditions, we need to apply some functions to prepare the lists for plotting. Finally, the four regimes are plotted with a and b on the x and y axes (Section 4.5.4).

4.5 In-Depth Explanation of the code

In this section we will give an in-dept explanation of all of the steps mentioned in the previous Section 4.4.

4.5.1 Declaring input

The first step of declaring the input is defining the game variables as Sage variables, as explained in Section 3.1. For the variables of the input game, we will use a and b. In addition we need to define the variable p, which is the proportion of the first strategy as defined in Chapter 2.

```
70 a, b, p = var('a b p')
```

Next, we declare $a, b, p \in \mathbb{R}$. This is important for solving equations and inequalities later on. This is done using the *assume* function provided by Sagemath.

```
73 assume(a, 'real')
74 assume(b, 'real')
75 assume(p, 'real')
```

We move on to declaring the actual input game matrix. This is done by declaring the four elements of the matrix in separate expressions. The allowed form of the input functions is given in Section 4.1. The input can be requested while running the program (lines 78-81) or directly in the code (lines 84-87), as mentioned before in Section 4.1. In principle, SageMath treats input as symbolic expressions by default, but to ensure this we use the function *SR()*.

```
77 # Define R, S, T, and F as Sage functions with input
78 R_expr = SR(input("Enter the function for R(a,b): "))
79 S_expr = SR(input("Enter the function for S(a,b): "))
80 T_expr = SR(input("Enter the function for T(a,b): "))
81 F_expr = SR(input("Enter the function for F(a,b): "))
82
83 # Define R, S, T, and F as Sage functions in code
84 R_expr = SR(<expr>)
85 S_expr = SR(<expr>)
86 T_expr = SR(<expr>)
87 F_expr = SR(<expr>)
```

4.5.2 Calculation of key values

In the next section of the code, the key values of the game are calculated. These values include the payoff functions for both strategies, the potential equilibrium point and the equation for

the replicator dynamics $\dot{p}$.

The payoff functions $w_1 = pR + (1-p)S$ and $w_2 = pT + (1-p)F$ are defined for each player in lines 95 and 96. Because the difference between w_1 and w_2 is frequently used to describe the direction of the system, we define $diff_w = w_1 - w_2$ in line 101. Using this variable, we define $\dot{p}$ in lines 103 and 104. Although the replicator dynamics $\dot{p}$ is not used further in the code, it remains an important expression for the game. Therefore, it is defined to allow the user to retrieve it. To simplify the expressions for w_1, w_2 and $\dot{p}$, we use the expand function provided by SageMath, explained in Section 3.1.

```
95  w_1 = p*R_expr + (1 - p) * S_expr
96  w_2 = p*T_expr + (1 - p) * F_expr
97
98  w_1 = w_1.expand()
99  w_2 = w_2.expand()
100
101 diff_w = w_1 - w_2
102
103 p_dot = p*(p-1)*(diff_w)
104 p_dot = p_dot.expand()
```

We continue by calculating the potential equilibrium value of the game. We calculate $w_1 = w_2$ by using the solve function provided by SageMath. The output of this solve function is a list of solutions, we named this list *p_ star_ list*.

```
106 p_star_list = solve(diff_w, p)
```

Note that this list will contain a maximum of one element, because R, S, T and F are non-quadratic. If this list is empty, this implies there is no solution, or the equilibrium does not depend on p. The next part of the code addresses this last case. We will discuss this later in Section 4.5.5. This part of the code utilises simpler versions of code components that will be explained in the following sections.

4.5.3 Determining conditions for each regime

Next, we will discuss the part of the code that determines restrictions for a and b. These restrictions indicate when each of the regimes appear. Recall Section 4.2 for the distinction of the four regimes. We will compile a list of the restrictions for each regime.

We start by retrieving the value of p^* from the list *p_ star_ list*. In this point in the code we already determined this list has exactly one element, refer to Section 4.5.5 for the case where the list is empty. This element is of the form $p == value$, so we can retrieve the value of p by taking the right side of first element of the list.

```
143     p_star = p_star_list[0].rhs()
```

Next, we solve the inequalities needed to define the different regimes. To determine when the system decreases or increases, we solve $w_1 - w_2 < 0$ and $w_1 - w_2 > 0$ for p. these solutions are saved in the lists *down*, for decrease, and *up*, for increase. The other relevant inequalities are ($p > p^*$, $0 < p^* < 1$), ($p < p^*$, $0 < p^* < 1$), $p^* > 1$ and $p^* < 0$, whose solutions are saved in the lists *pGp_star*, *pSp_star*, *p_starG1* and *p_starS0*, respectively. All these conditions are solved similar. In Chapter 2 we explained why these inequalities are relevant.

```
146     down = solve(diff_w < 0, p)
147     up = solve(diff_w > 0, p)
148 
149     pGp_star = solve([p>p_star, p_star>0, p_star<1],a,b)
150     pSp_star = solve([p<p_star, p_star>0, p_star<1],a,b)
151     p_starG1 = solve([p_star>1],a,b)
152     p_starS0 = solve([p_star<0],a,b)
```

We start with four empty lists for the four regimes, to which we will add the solutions of applicable inequalities. In Section 2.2, we explained when each of the regimes occur. The inequalities that for each regime have to satisfy are:

- Regime 1: The system increases while $p^* > 1$ and $p < p^*$ or the system increases while $p^* < 0$ and $p > p^*$.
- Regime 2: The system decreases while $p^* > 1$ and $p < p^*$ or the system decreases while $p^* < 0$ and $p > p^*$.
- Regime 3: The system decreases while $p > p^*$ or the system increases while $p < p^*$. For this regime p^* lies in the interval (0,1), so $0 < p^* < 1$.
- Regime 4: The system decreases while $p < p^*$ or the system increases while $p > p^*$. For this regime p^* lies in the interval (0,1), so $0 < p^* < 1$.

For every regime, we go through the restrictions of the lists *down*, *up*, *pGp_star*, *pSp_star*, *p_starG1* and *p_starS0* that apply. For instance, in the first regime, we determine when both $w_1 - w_2 > 0$ and $p^* > 1$ hold. This is done by first using a for loop to iterate through the solutions of *up* and *p_starG1* (lines 161 and 162). Then, the *solve* function is used to combine these restrictions on a and b with the additional restriction $p < p^*$, into a single solution (line 163). If there is at least one solution (i.e., the list is not empty) and this solution is not already in the list of the current regime, we add it to the list (lines 165 to 167). For the first regime we repeat this for the other case, where $p^* < 0$ and $p > p^*$ (lines 168 to 174). The code lines below demonstrate this for the first regime. The other regimes have similar code lines with different restrictions, which can be found in lines 176 to 219 in Appendix A.

```
161     for elem in up:
162         for sol in p_starG1:
163             eq = sol + elem + [p<p_star]
164             solution = solve(eq, a, b)
165             if solution != []:
166                 if solution not in regime1:
167                     regime1.append(solution)
168     for elem in up:
169         for sol in p_starS0:
170             eq = sol + elem + [p>p_star]
171             solution = solve(eq, a, b)
172             if solution != []:
173                 if solution not in regime1:
174                     regime1.append(solution)
```

4.5.4 Plotting

We now have obtained four lists with the correct restrictions for the corresponding regime. Before we can plot these we need to perform some operations on them. We start by cleaning up the list, for this we constructed a *cleanup_list* function consisting of three parts. Since the solve function outputs a list of lists, each solution is placed in an unnecessary extra list. To remove these, we created a simple line that directly adjusts the list.

```
5   def cleanup_list(regime_list) : #cleans up list before plotting
6
7       #removes dubbel list (brackets) around elements
8       regime_list = [sol for sublist in regime_list for sol in sublist]
```

The second part of the *cleanup_list* function ensures that every element of the list is in the same form, meaning each inequality should be expressed as a function of b. Therefore, we solve each element for b and return it to the list. We could not perform this step earlier, as restrictions depending solely on a might have been lost. We use two extra lists q and l to ensure only the restrictions that form a solution together are placed in a list together. In line 15 we take the first element of the first list of the solution, this is allowed because we solve single elements for b, so the *solve* function in this line always outputs a list of a single list with one element.

```
10  #Get everything in a "b=" form
11      q=[]
12      for sol in regime_list:
13          l = []
14          for elem in sol:
15              l.append(solve([elem],b)[0][0])
```

```
16         q.append(l)
17
18     regime_list = q
```

The final part of the *cleanup_list* function addresses the issue where SageMath occasionally leaves contradictory conditions in its list of solutions. This could result in pairs such as $2p + b > 0$ and $-2p - b > 0$, which obviously have no solutions. Therefore, we remove solutions from the list if they are identical except for a factor of -1. We iterate through the list and compare every element of each solution. If contradictions are found, the entire solution is added to a *helplist*. Thereafter, all of the solutions in this *helplist* are removed from the *regime_list*. Lastly, the cleaned-up regime list is returned.

```
20     #removes contradictions
21     helplist = []
22
23     for sol in regime_list:
24         for i in range(len(sol)-1):
25             for j in range(i + 1, len(sol)):
26                 if sol[i] == -1 * sol[j]:
27                     helplist.append(sol)
28
29     for sol in helplist:
30         regime_list.remove(sol)
31     return regime_list
```

To plot the four regimes we construct a plot list from each cleaned up list. This is done by a *plotlist* function. The function *plotlist* takes two parameters, the regime list and a list with the relations within these regime list. This relations list shows every inequality as a tuple with the decomposed relations. The inequality $1 + a > b$ would be shown as a tuple of the form $(1 + a$, 'gt', $b)$. The reason why this list is necessary will be addressed later on in this section. In lines 44 to 51, this list is created by the function *relations_list*, which takes the regime list. We iterate through the elements of each solution and form a tuple with the left side, the operator, and the right side of the inequality. This tuple is then added to the list of relations.

```
44 def relations_list(regime_list): #returns a list of decomposed relations
45     relations_list = []
46     for sol in regime_list:
47         relations = []
48         for elem in sol:
49             relations.append((elem.lhs(), get_operator(elem), elem.rhs()))
50         relations_list.append(relations)
51     return(relations_list)
```

As we see in line 49, we use a *get_ operator* function. The *get_ operator* function uses the build-in SageMath function *operator()*, but makes the output more aesthetic and easier to read.

```
34 def get_operator(x): #pretty output for operator
35     if x.operator() == operator.lt:
36         return "lt"
37     elif x.operator() == operator.gt:
38         return "gt"
39     elif x.operator() == operator.eq:
40         return "eq"
41     else:
42         return "zero"
```

We will now examine how the function *plotlist* works. This function begins with a empty plot list to which we will add the necessary inequalities. The function iterates through the inequalities of every solution in the constructed relations list (lines 55 to 57). Note that by the *cleanup_ list* function, we ensured the inequalities are solved for b. By iterating through the relations rather than the regime list itself, we can check for isolated a or b in the inequality. All such inequalities are added to the plot list per solution (lines 58 to 61). Each solution also includes restrictions for p, relating p^* to p, 0 and 1. These restrictions where used in composing the lists for the regimes. By the definition of the regimes in Section 4.5.3, we already know these constraints for each regime, allowing us to ignore them when composing the plot lists. Once all solutions have been processed, the complete plot list is returned.

```
53 def plotlist(relations_list, regime_list): #returns the plot list for a regime
54     plot_list = []
55     for i in range(len(relations_list)):
56         sol = []
57         for j in range(len(relations_list[i])):
58             eq = relations_list[i][j]
59             if (a in eq) or (b in eq):
60                 sol.append(regime_list[i][j])
61         plot_list.append(sol)
62     return plot_list
```

We apply all the aforementioned functions to get everything in order for the plot.

```
222     #Cleanup lists
223     regime1 = cleanup_list(regime1)
224     regime2 = cleanup_list(regime2)
225     regime3 = cleanup_list(regime3)
226     regime4 = cleanup_list(regime4)
227
```

```
228     #Extract a List of Decomposed Relations
229     relations1 = relations_list(regime1)
230     relations2 = relations_list(regime2)
231     relations3 = relations_list(regime3)
232     relations4 = relations_list(regime4)
233 
234     #Composes plot lists
235     plotlist1 = plotlist(relations1, regime1)
236     plotlist2 = plotlist(relations2, regime2)
237     plotlist3 = plotlist(relations3, regime3)
238     plotlist4 = plotlist(relations4, regime4)
```

For the actual plot we iterate through the plot list of every regime and add every region to the graphics P. This is done by the SageMath function *region_plot*. We provide this function with the inequalities that denotes the area to be filled, the boundaries on the axis for both axis, the colour for the area, and the colour for the boundaries given by the inequality. For the boundaries on the axis, which determine how large the plot is going to be, we defined a parameter *scale* in line 65. Since the region plot function does not support adding a legend, we use a dummy plot consisting of a line with length zero to add the colour and title to the legend. The colours in the legend will be shown by a short line segment, unfortunately it is not possible to get the legend to present squares of the colours.

```
241     for sol in plotlist1:
242         P += region_plot(sol, (a,-scale,scale),(b,-scale,scale), incol = 'tomato',
243         bordercol='black')
244 
245     #Add a dummy plot for the legend
246     P += line([(0,0)], color='tomato', legend_label='First regime')
```

These code lines are repeated for every regime. For the regimes, we used the colours provided by SageMath: “Forestgreen”, “Yellow”, “Tomato” and “Royalblue”, in that order.

The final step for the plot is to add the axis titles, set the legend options and save the image. Which is done in the following code lines. In the last line 283, the user has to adjust the location and file name.

```
272 #Axis labels
273 tx = text(x_titel, (scale+0.2,-0.5), fontsize=13)
274 ty = text(y_titel, (0.2,scale+0.2), fontsize=13)
275 P += tx
276 P += ty
277 
278 # Enable legend
279 P.set_legend_options(loc='upper right', back_color=(0.8, 0.8, 0.8),
```

```
280 shadow=False, handlelength=0.1, markerscale=5, font_size=12)
281
282 #Change to the correct location and name
283 P.save("<location>/<file_name>.png")
```

4.5.5 Case: equilibrium independent of p

In this last section, we handle the case where the equilibrium is not dependent on p. This case is a simplified version of the rest of the code because only two regimes are possible: one where the first strategy is stable and one where the second strategy is stable.These will be referred to as the "First regime" and the "Second regime", to maintain consistency with the rest of the code.

As previously mentioned in Section 4.5.2, we consider the case where *p_star_list* is empty, and thus begin by verifying this in line 108. Subsequently, we calculate the equilibrium as a function of a and b in lines 109 and 110. This ensures that the game is handled correctly if the equilibrium depends only on one of these variables. If the equilibrium depends on both both a and b or only on a, the list *a_star* will not be empty. The code handles this case first. If *a_star* is empty, but *b_star* is not, this case is addressed in a similar manner, as outlined in lines 127 to 141 in Appendix A.

```
108 if p_star_list==[]:
109     a_star =  solve(diff_w, a)
110     b_star = solve(diff_w, b)
111     if a_star!=[]:
```

Since there are only two regimes, we only need to calculate when the system increases or decreases. We do this by solving when the difference between w_1 and w_2 is greater than or less than zero. Because we are solving for only one inequality, the previously discussed functions *cleaunup_list*, *relations_list* and *plotlist* from Section 4.5.4, are not necessary. We plot the solutions for an increasing system in green and the solutions for a decreasing system in yellow. following the same method as in Section 4.5.4.

```
113         down = solve(diff_w < 0, a)
114         up = solve(diff_w > 0, a)
115         for sol in up:
116             P += region_plot(sol, (a,-scale,scale),(b,-scale,scale),
117             incol = 'forestgreen', bordercol='black')
118
119         P += line([(0,0)], color='forestgreen', legend_label='First regime')
120
121         for sol in down:
122             P += region_plot(sol, (a,-scale,scale),(b,-scale,scale),
```

```
123             incol = 'yellow', bordercol='black')
124
125         P += line([(0,0)], color='yellow', legend_label='Second regime')
```

We have now explained the purpose and functionality of each section of the code. The complete code can be found in Appendix A.

5 Case studies of using code

In the previous chapter, Chapter 4, we have discussed the code to symbolically solve two-strategy games. In this chapter we are going to apply this program to the games discussed in Chapter 2. In the first section we will show the plot which is generated when provided input from the general game from Section 2.3. We will focus on customising this plot using the variables U and V. In the second section we will discuss the Go vs. Grow game from Section 2.4. For this input game we will discuss the intermediate solutions, as well as the final plot.

5.1 General game presenting as U vs. V

We first recall the general matrix for the general game from Section 2.3

$$\begin{pmatrix} 1 & U \\ V & 0 \end{pmatrix}. \tag{67}$$

We adjust the code like explained in Section 4.3 to solve for U and V. We begin by replacing a and b with U and V in line 70 which defines the SageMath variables. We also declare $U, V \in \mathbb{R}$ in lines 73 and 74 of the code.

```
70 U, V, p = var('a b p')
71
72 #a,b and p are real numbers
73 assume(U, 'real')
74 assume(V, 'real')
75 assume(p, 'real')
```

Next we define our input expressions in the code and add $a = U$ and $b = V$.

```
84 R_expr = SR(1)
85 S_expr = SR(U)
86 T_expr = SR(V)
87 F_expr = SR(0)
88
89 a=U
90 b=V
```

Lastly, we rename the axis titles to U and V in lines 90 and 91.

```
92 x_titel = "U"
93 y_titel = "V"
```

This input and customisation result in the plot shown in Figure 6.

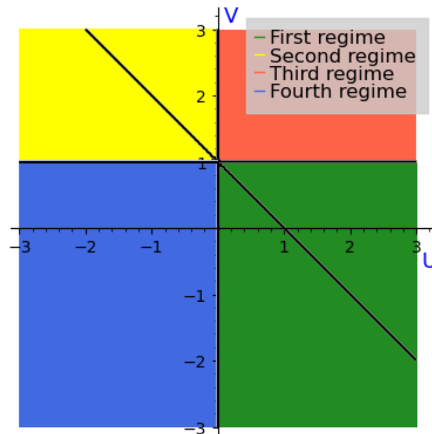


Figure 6: The output plot generated by the program when provided with the U vs. V representation of a general game matrix as input.

5.2 Go vs. grow game

In this section we will examine how the code generated a plot from the payoff matrix from Section 2.4:

$$\begin{pmatrix} 1-\frac{1}{2}\frac{c}{b} & 1-\frac{c}{b} \\ 1 & \frac{1}{2} \end{pmatrix}. \tag{68}$$

To run this example, we begin by defining the input expressions R, S, T and F. Note that we use the variable name a instead of c, as in Section 2.4. To ensure the game is solved symbolically, we transform the input expressions to symbolic expressions by the function *SR()*. We also define an additional variable $h = \frac{1}{2}$, this ensures the output to be exact.

```
77 # Define R, S, T, and F as Sage functions in code
78 h = Integer(1)/Integer(2)
79 R_expr = SR(1-h*a/b)
80 S_expr = SR(1-a/b)
81 T_expr = SR(1)
82 F_expr = SR(h)
```

The payoff functions and the equilibrium point of this game are defined in Section 2.4. For $c = a$ these are:

$$w_{go} = \frac{1}{2}\frac{a}{b}p - \frac{a}{b} + 1. \tag{69}$$

$$w_{grow} = \frac{1}{2}p + \frac{1}{2}. \tag{70}$$

$$p^* = \frac{2a-b}{a-b} \tag{71}$$

The payoff functions correspond to w_1 and w_2 in the code, respectively. When retrieved from the code, w_1 has output $\frac{ap}{2b} - \frac{a}{b} + 1$ and w_2 has output $\frac{1}{2}p + \frac{1}{2}$. These are equivalent to payoff

functions defined above. When asked for p^* the code returns $\frac{2a-b}{a-b}$, we observe once more this corresponds to our calculated value. In Section 2.4 we determined the constrictions for the four possible regimes. Recall that we determined the last regime does not occur. The restrictions for the first regime where:

$$(b > 0,\ a < 0), \tag{72}$$
$$(b < 0,\ a > 0). \tag{73}$$

The restrictions for the second regime where:

$$\left(b > 0,\ \frac{b}{2} < a < b\right), \tag{74}$$
$$\left(b < 0,\ b < a < \frac{b}{2}\right), \tag{75}$$
$$(b > 0,\ a > b), \tag{76}$$
$$(b < 0,\ a < b). \tag{77}$$

Lastly, the restrictions for the third regime where:

$$\left(b > 0,\ 0 < a < \frac{b}{2}\right), \tag{78}$$
$$\left(b < 0,\ \frac{b}{2} < a < 0\right). \tag{79}$$

In the code these constrictions are placed in lists. The output for the list for the first regime is

$$\begin{aligned}\text{regime1} = \big[&[a > 0, b < 0, -ap + bp + 2\,a - b > 0, -ap + bp + 2\,a - b > 0]\,,\\ &[a < 0, b > 0, ap - bp - 2\,a + b > 0, ap - bp - 2\,a + b > 0]\big].\end{aligned}$$

In these solutions the last two elements of each solution are equivalent to $p > p^*$ or $p < p^*$. These inequalities are removed before plotting by the function *plotlist*, explained in Section 4.5.4. The remaining inequalities correspond with our calculates inequalities (72) and (73). The output for the second regime is

$$\begin{aligned}\text{regime2} = \big[&[b < a, b > 0, -ap + bp + 2\,a - b > 0, -ap + bp + 2\,a - b > 0]\,,\\ &[a < b, b < 0, ap - bp - 2\,a + b > 0, ap - bp - 2\,a + b > 0]\,,\\ &[b < 2\,a, a < b, b > 0, -ap + bp + 2\,a - b > 0, -ap + bp + 2\,a - b > 0]\,,\\ &[b < a, 2\,a < b, b < 0, ap - bp - 2\,a + b > 0, ap - bp - 2\,a + b > 0]\big].\end{aligned}$$

For this regime the elements depending on p also denote $p > p^*$ or $p < p^*$. The remaining constrictions corresponds with (74), (75), (76) and (77). The output for the third regime is

$$\begin{aligned}\text{regime3} = \big[&[a > 0, 2\,a < b, b > 0, -ap + bp + 2\,a - b > 0]\,,\\ &[b < 2\,a, a < 0, b < 0, ap - bp - 2\,a + b > 0]\,,\\ &[b < 2\,a, a < 0, b < 0, -ap + bp + 2\,a - b > 0]\,,\\ &[a > 0, 2\,a < b, b > 0, ap - bp - 2\,a + b > 0]\big].\end{aligned}$$

Note this list contains each of the restrictions (78) and (79) twice, this is because the are added once for $w_{go} - w_{grow} > 0$, $p < p*$ and once for $w_{go} - w_{grow} < 0$, $p > p*$. In Section 2.4 is shown that these pairs of inequalities have the same solution. We can conclude the output of this regime corresponds with our mathematically derived values (78) and (79). The output for the last regime is

$$\text{regime4} = [\,] \,.$$

We found that the last regime does not occur, so it is correct that the list for regime 4 is empty.

Subsequently, the code will plot these identified regions, resulting in the dynamic regime plot shown in Figure 7. Note that the line $b = a$ is plotted because it is a boundary of the different regions that together form the second regime.

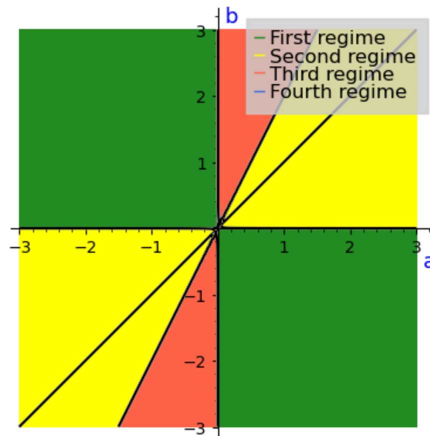


Figure 7: The output plot generated by the code when provided with the Go vs. Grow game matrix from Section 2.4 as input.

6 Future directions

In the preceding chapters, we explored evolutionary two-strategy games and demonstrated the application of SageMath in solving these games symbolically. In future projects this code could be expanded to solve three-strategy games. Three-strategy games introduce additional complexity due to the increased number of variables, requiring more creativity to produce meaningful two-dimensional plots.

A three-strategy game can be visualised as a triangle with each of the three strategies located at a corner as shown in Figure 8. Each corner represent the situation where the corresponding strategy has a proportion of one, i.e., makes up the entire population. The edges between two corners represent the behaviour of the game when only those two strategies are present, functioning like a two-strategy game like seen in Section 2.2. The middle of the triangle, however, can exhibit more complex behaviours. It is possible for two three-strategy games to have the exact same behaviours on the edges, while the middle presents totally different behaviours. For a deeper understanding of the mathematics involved in solving three-strategy games, please refer to the article by Bomze [11].

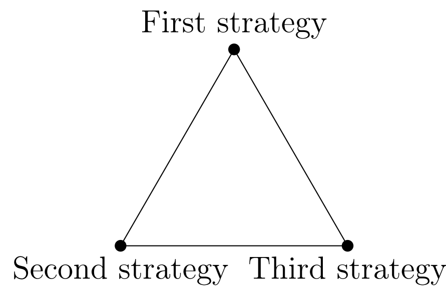


Figure 8: Visualisation of a three-strategy game. The corners refer to the state of the game where the proportion of the corresponding strategy is equal to one.

The first step in solving a three-strategy game is to solve the two-strategy games occurring on the edges. The code described in Chapter 4 is already able to solve these subgames. For a future project this intermediate solutions could be combined to solve the three-strategy game symbolically. As mentioned in the Introduction 1, there already exist programming tools to solve three-strategy games numerically [3][4]. However, we are especially interested in solving games symbolically, This creates more insight in the effects of the parameters in the game. Unfortunately, writing this code was not in scope of the current project.

There are many relevant three-strategy evolutionary games for which a symbolic solver could be helpful. One of these games is the "Go vs. Grow. vs. Gly" game [12] [13], which is an extension of the game discussed in Section 2.4. Other dynamics within oncology that can be described by evolutionary three-strategy games include the dynamics of Tumor-Stroma in multiple myeloma [14] or dynamics in metastatic castrate-resistant prostate cancer [15].

7 Conclusion

In this thesis, we developed a tool to symbolically solve two-strategy evolutionary games by combining mathematical theory with programming in SageMath. Although we primarily discussed two-strategy evolutionary games in a biological context, the tool could be used for two-strategy evolutionary games in all the application fields. The input for the tool are four expressions of the form $\lambda a^x + \gamma b^y$ or $\lambda a^x \times \gamma b^y$ which together form a game matrix $\left(\begin{smallmatrix} R & S \\ T & F \end{smallmatrix}\right)$. The output of the program is a plot with a and b on the axis, that shows the regions where different dynamic regimes occur.

To develop this tool we began by exploring the mathematics involved in solving two-strategy games. We explored how an environment with two populations can be described as an evolutionary game and which equations and values are important. We discussed what it means to solve such a game symbolically and how this is accomplished. This was demonstrated through a general game and a example from mathematical oncology.

Following this, we introduced SageMath. Through a brief tutorial, we covered the functions provided by SageMath which are important for solving two-strategy evolutionary games, providing the necessary tools to understand the final code. In this code, we symbolically solve two-strategy evolutionary games, by handling input, calculating key values, determining conditions for the different regimes and finally, plotting the results. Using examples we illustrated how this code can be used to symbolically solve two-strategy evolutionary games.

In future projects, this thesis could be extended to three-strategy games. We provided a brief insight on these games and the potential future directions of this project.

In conclusion, this thesis has not only provided a practical tool for symbolically solving two-strategy games but also offered insight into the mathematical background and the practical applications.

References

[1] B. Wölfl, H. te Rietmole, M. Salvioli, A. Kaznatcheev, F. Thuijsman, J. S. Brown, B. Burgering, and K. Staňková, *The contribution of evolutionary game theory to understanding and treating cancer* (2021).

[2] M. Gluzman, J. G. Scott, and A. Vladimirsky, *Optimizing adaptive cancer therapy: dynamic programming and evolutionary game theory* (2020).

[3] J. West, Y. Ma, A. Kaznatcheev, and A. Anderson, *Isomatrix: a framework to visualize the isoclines of matrix games and quantify uncertainty in structured populations.* (2020).

[4] I. Mirzaev, D. Williamson, and J. G. Scott, *Isomatrix: a framework to visualize the isoclines of matrix games and quantify uncertainty in structured populations.* (2018).

[5] A. Kaznatcheev, *Two conceptions of evolutionary games: reductive vs effective* (2017).

[6] A. Kaznatcheev, *Space of cooperate-defect games*, URL `https://egtheory.wordpress.com/2012/03/14/uv-space/`.

[7] A. Kaznatcheev, J. G. Scott, and D. Basanta, *Edge effects in game-theoretic dynamicsof spatially structured tumours* (2015).

[8] *Sagemath*, URL `https://www.sagemath.org/`.

[9] *List of named colors*, URL `https://matplotlib.org/stable/gallery/color/named_colors.html`.

[10] *2d graphics: Colors*, URL `https://doc.sagemath.org/html/en/reference/plotting/sage/plot/colors.html`.

[11] I. M. Bomze, *Lotka-volterra equation and replicator dynamics: A two-dimensional classification* (1983).

[12] A. Kaznatcheev, *Warburg effect and evolutionary dynamics of metastasis*, URL `https://egtheory.wordpress.com/2013/07/08/warburg-effect-and-evolutionary-dynamics-of-metastasis/`.

[13] D. Basanta, M. Simon, H. Hatzikirou, and A. Deutsch, *Evolutionary game theory elucidates the role of glycolysis in glioma progression and invasion.*

[14] J. Sartakhti, M. Manshaei, S. Bateni, and M. Archetti, *Evolutionary dynamics of tumor-stroma interactions in multiple myeloma* (2021).

[15] J. Zhang, J. J. Cunningham, J. S. Brown, and R. A. Gatenby, *Integrating evolutionary dynamics into treatment of metastatic castrate-resistant prostate cancer* (2017).

A Code

```
from sage.all import *

#-------Functions-------------------------------------------------------------------

def cleanup_list(regime_list) : #cleans up list before plotting

    #removes dubbel list (brackets) around elements
    regime_list = [sol for sublist in regime_list for sol in sublist]

    #Get everything in a "b=" form
    q=[]
    for sol in regime_list:
        l = []
        for elem in sol:
            l.append(solve([elem],b)[0][0])
        q.append(l)

    regime_list = q

    #removes contradictions
    helplist = []

    for sol in regime_list:
        for i in range(len(sol)-1):
            for j in range(i + 1, len(sol)):
                if sol[i] == -1 * sol[j]:
                    helplist.append(sol)

    for sol in helplist:
        regime_list.remove(sol)
    return regime_list


def get_operator(x): #Makes the output for operator more manageable
    if x.operator() == operator.lt:
        return "lt"
    elif x.operator() == operator.gt:
        return "gt"
    elif x.operator() == operator.eq:
        return "eq"
    else:
        return "zero"
```

```
43
44 def relations_list(regime_list): #Returns a list of decomposed relations
45     relations_list = []
46     for sol in regime_list:
47         relations = []
48         for elem in sol:
49             relations.append((elem.lhs(), get_operator(elem), elem.rhs()))
50         relations_list.append(relations)
51     return(relations_list)
52
53 def plotlist(relations_list, regime_list): #Returns the plot list for a regime
54     plot_list = []
55     for i in range(len(relations_list)):
56         sol = []
57         for j in range(len(relations_list[i])):
58             eq = relations_list[i][j]
59             if (a in eq) or (b in eq):
60                 sol.append(regime_list[i][j])
61         plot_list.append(sol)
62     return plot_list
63
64 #Needed for plotting
65 scale = 3
66 # Initialize the main plot
67 P = Graphics()
68
69 #-------Declaring input------------------------------------------------------------
70 a, b, p = var('a b p')
71
72 #a,b and p are real numbers
73 assume(a, 'real')
74 assume(b, 'real')
75 assume(p, 'real')
76
77 # Define R, S, T, and F as Sage functions with input
78 R_expr = SR(input("Enter the function for R(a,b): "))
79 S_expr = SR(input("Enter the function for S(a,b): "))
80 T_expr = SR(input("Enter the function for T(a,b): "))
81 F_expr = SR(input("Enter the function for F(a,b): "))
82
83 # Define R, S, T, and F as Sage functions in code
84 R_expr = SR(<expr>)
85 S_expr = SR(<expr>)
86 T_expr = SR(<expr>)
87 F_expr = SR(<expr>)
```

```
#Adjust to variable names
x_titel = "c"
y_titel = "b"

#-------Calculation of Key Values---------------------------------------------------

w_1 = p*R_expr + (1 - p) * S_expr
w_2 = p*T_expr + (1 - p) * F_expr

w_1 = w_1.expand()
w_2 = w_2.expand()

diff_w = w_1 - w_2

p_dot = p*(p-1)*(diff_w)
p_dot = p_dot.expand()

p_star_list = solve(diff_w, p)

if p_star_list==[]: #-------Case: eqiulibrium independent of p-------
    a_star =  solve(diff_w, a)
    b_star = solve(diff_w, b)
    if a_star!=[]:

        down = solve(diff_w < 0, a)
        up = solve(diff_w > 0, a)
        for sol in up:
            P += region_plot(sol, (a,-scale,scale),(b,-scale,scale),
            incol = 'forestgreen', bordercol='black')

        P += line([(0,0)], color='forestgreen', legend_label='First regime')

        for sol in down:
            P += region_plot(sol, (a,-scale,scale),(b,-scale,scale),
            incol = 'yellow', bordercol='black')

        P += line([(0,0)], color='yellow', legend_label='Second regime')

    if b_star!=[]:

        down = solve([(diff_w < 0)], b)
        up = solve([diff_w > 0], b)
        for sol in up:
            P += region_plot(sol, (a,-scale,scale),(b,-scale,scale),
```

```
133             incol = 'forestgreen', bordercol='black')
134 
135         P += line([(0,0)], color='forestgreen', legend_label='First regime')
136 
137         for sol in down:
138             P += region_plot(sol, (a,-scale,scale),(b,-scale,scale),
139             incol = 'yellow', bordercol='black')
140 
141         P += line([(0,0)], color='yellow', legend_label='Second regime')
142 else:
143 #-------Determining conditions for each regime---------------------------------------
144     p_star = p_star_list[0].rhs()
145 
146     down = solve(diff_w < 0, p)
147     up = solve(diff_w > 0, p)
148 
149     pGp_star = solve([p>p_star, p_star>0, p_star<1],a,b)
150     pSp_star = solve([p<p_star, p_star>0, p_star<1],a,b)
151     p_starG1 = solve([p_star>1],a,b)
152     p_starS0 = solve([p_star<0],a,b)
153 
154     regime1 = []
155     regime2 = []
156     regime3 = []
157     regime4 = []
158 
159 
160     #First Regime
161     for elem in up:
162         for sol in p_starG1:
163             eq = sol + elem + [p<p_star]
164             solution = solve(eq, a, b)
165             if solution != []:
166                 if solution not in regime1:
167                     regime1.append(solution)
168     for elem in up:
169         for sol in p_starS0:
170             eq = sol + elem + [p>p_star]
171             solution = solve(eq, a, b)
172             if solution != []:
173                 if solution not in regime1:
174                     regime1.append(solution)
175 
176     #Second Regime
177     for elem in down:
```

```
178         for sol in p_starG1:
179             eq = sol + elem + [p<p_star]
180             solution = solve(eq, a, b)
181             if solution != []:
182                 if solution not in regime2:
183                     regime2.append(solution)
184     for elem in down:
185         for sol in p_starS0:
186             eq = sol + elem + [p>p_star]
187             solution = solve(eq, a, b)
188             if solution != []:
189                 if solution not in regime2:
190                     regime2.append(solution)
191 
192     #Third Regime
193     for elem in down:
194         for sol in pGp_star:
195             solution = solve(sol + elem, a,b)
196             if solution != []:
197                 if solution not in regime3:
198                     regime3.append(solution)
199 
200     for elem in up:
201         for sol in pSp_star:
202             solution = solve(sol + elem, a, b)
203             if solution != []:
204                 if solution not in regime3:
205                     regime3.append(solution)
206 
207     #Fourth Regime
208     for elem in up:
209         for sol in pGp_star:
210             solution = solve(sol + elem, a, b)
211             if solution != []:
212                 if solution not in regime4:
213                     regime4.append(solution)
214     for elem in down:
215         for sol in pSp_star:
216             solution = solve(sol + elem, a, b)
217             if solution != []:
218                 if solution not in regime4:
219                     regime4.append(solution)
220 
221 #-------Plotting----------------------------------------------------------------
222     #Cleanup lists
```

```
223     regime1 = cleanup_list(regime1)
224     regime2 = cleanup_list(regime2)
225     regime3 = cleanup_list(regime3)
226     regime4 = cleanup_list(regime4)
227 
228     #Extract a List of Decomposed Relations
229     relations1 = relations_list(regime1)
230     relations2 = relations_list(regime2)
231     relations3 = relations_list(regime3)
232     relations4 = relations_list(regime4)
233 
234     #Composes plot lists
235     plotlist1 = plotlist(relations1, regime1)
236     plotlist2 = plotlist(relations2, regime2)
237     plotlist3 = plotlist(relations3, regime3)
238     plotlist4 = plotlist(relations4, regime4)
239 
240     #First Regime
241     for sol in plotlist1:
242         P += region_plot(sol, (a,-scale,scale),(b,-scale,scale), incol = 'forestgreen',
243         bordercol='black')
244 
245     #Add a dummy plot for the legend
246     P += line([(0,0)], color='forestgreen', legend_label='First regime')
247 
248     #Second Regime
249     for sol in plotlist2:
250         P += region_plot(sol, (a,-scale,scale),(b,-scale,scale), incol = 'yellow',
251         bordercol='black')
252 
253     #Add a dummy plot for the legend
254     P += line([(0,0)], color='yellow', legend_label='Second regime')
255 
256     #Third Regime
257     for sol in plotlist3:
258         P += region_plot(sol, (a,-scale,scale),(b,-scale,scale), incol = 'tomato',
259         bordercol='black')
260 
261     #Add a dummy plot for the legend
262     P += line([(0,0)], color='tomato', legend_label='Third regime')
263 
264     #Fourth Regime
265     for sol in plotlist4:
266         P += region_plot(sol, (a,-scale,scale),(b,-scale,scale), incol = 'royalblue',
267         bordercol='black')
```

```
268
269     #Add a dummy plot for the legend
270     P += line([(0,0)], color='royalblue', legend_label='Fourth regime')
271
272 #Axis labels
273 tx = text(x_titel, (scale+0.2,-0.5), fontsize=13)
274 ty = text(y_titel, (0.2,scale+0.2), fontsize=13)
275 P += tx
276 P += ty
277
278 # Enable legend
279 P.set_legend_options(loc='upper right', back_color=(0.8, 0.8, 0.8),
280 shadow=False, handlelength=0.1, markerscale=5, font_size=12)
281
282 #Change to the correct location and name
283 #P.save("<location>/<file_name>.png")
```