ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Analysis of Axelrod Tournament Meetings and Methodology

Group 03

Collaborators

Kostantinos Fotis Papadakis, kpapadak@ece.auth.gr

Grigoris Daios, grigorad@ece.auth.gr

Ioannis Georgiou Mousses, georgioui@ece.auth.gr

Sokratis Nazlidis, snazlidi@ece.auth.gr

Department of Electrical and Computer Engineering, AUTH

May 2025

Contents

Chapter 1

Introduction

1.1 The problem at hand

In the following report, we examine the dynamics of **evolutionary games**-games played over successive generations in large populations. These games consist of:

- A **base game**, such as the Iterated Prisoner's Dilemma.
- A **population** of players.
- A **pool of strategies** available to players.
- A **match**, defined as a play between two players.
- A **meeting**, which includes all possible matches in the population.

The evolutionary game progresses through a sequence of meetings, with each new generation adopting strategies based on performance. Initially, strategy adoption is random, but over time, better-performing strategies prevail. While some games converge to a single dominant strategy, others exhibit **oscillations**, **sensitivity to initial conditions**, or even **chaotic patterns**. This report focuses on two mechanisms that drive these evolutionary changes:

- **Imitation dynamics**-players mimic successful strategies from the previous generation.
- **Fitness dynamics**-strategies spread in proportion to their payoff.

Through the above models, we explore how strategy performance shapes long-term behavior in populations. It is ultimately revealed that cooperation is not always the inevitable outcome.

1.2 Related work

This report builds on earlier work by Axelrod and Hamilton (1981) [2], Axelrod and Dion (1988) [3], and the book "Evolutionary Game Theory " by J. McKenzie Alexander (2023) [1]. Furthermore,"Studies on Dynamics in the Classical Iterated Prisoner's Dilemma with Few Strategies" by Mathieu, Beaufils and Delahaye (1999) [4] questions

the assumption that cooperation always dominates,by presenting other complex dynamics and highlighting the role of initial conditions in final simulation outcomes.
For the Markov Theory part, we based our work on: "The Iterated Prisoner's Dilemma" [6], "Zero-Determinant Strategies in the Iterated Prisoner's Dilemma" [7] and "Iterated Prisoner's Dilemma contains strategies that dominate any evolutionary opponent" [8].

1.3 Quick start-Brief intro to our code

First and foremost, one can review our entire implementation here:
`https://github.com/Kou-ding/Prisoner-s-Dilemma`
Detailed presentation of the most important parts of the code can be found in the Appendix of this report.
Our entire **MATLAB toolbox** is based upon two sections:

- **Genaxel**: Here we simulate and analyze strategy evolution within the Axelrod tournament. The implementation facilitates dIfferent modes of fitness calculation and evolutionary dynamics.It also supports multiple tournament styles and population update mechanisms.
- **Markov**: This second part of the project was implemented in a completely dIfferent way. Object-oriented programming was not the basis here. The logic is implementing 1 on 1 tournaments of 3 players, where populations change on the basis of implemented rules and States.

1.3.1 How to run the project

1. Download the project locally
2. navigate to `Examples/` directory
3. Configure the `script.m` file to reflect the parameters you want to use
    - Set `custom = true` to manually configure all parameters.
    - Or set `custom = false` and load one of the pre-made meetings by uncommenting the corresponding `.fig` file under the **"Load your desired meeting"** section.

1.3.2 Project directory overview

Folder / File	Description
`Code/Genaxel/`	Genetic Axelrod tournament (OOP)
`Code/Markov/`	Markov-related modules
`Markov Theory/`	Theory documentation
`Markov Simulation/`	Simulation setup and code
`Documentation/`	Function-level code documentation
`Examples/`	Demos and `script.m` setup
`Report/`	Final report PDF and LaTeX source

1.4 Markov theory: Quick start

1.4.1 How to run

All the files required are inside the Markov directory. In function `initialize()` we can choose the total population N, the payoff matrix (R, S, T, P), the rounds per match for the theoretical (`roundsth`) and the simulation (`roundssim`) parts, the number of repetitions per initial state `numofrep`, the number of generations `ngens` needed for the simulation part and the vector `chosen` of the three strategies to compete. If the calculations are to be based on the expected average payoff per round, the value of the variable `predetermined` should be set to the empty string ' '. If exact calculations based on the evaluation of the score as a deterministic function of the game history are to be used, we can choose the competitors by setting an appropriate string value for `predetermined` (see `initialize()` for details). After the theoretical calculations are performed and the respective graphs drawn, the user is asked if he wishes to continue with the simulation part (may be time demanding). There is no simulation part if predetermined outcome meetings are used. Run the script `MarkovRun.m`.

1.4.2 How to add a new strategy with its strategy vector p

In `initialize()`, add the strategy's index and name in the map `names`, its name and strategy vector in map `ps` and its name and probability for the initial move to be C, in the map `pinitC`.

1.4.3 How to add a new predetermined outcome meeting

Use a new letter as a possible value of `predetermined` and add a new information message in the appropriate `switch-case` in `initialize()`. Describe the calculation of the payoffs in a match between any two players, in the `switch-case` of the function `deterministicPayoffs()`.

1.4.4 Structure of matrix `allstaterepgenNumOfPlayersHistory` and plotting of the time evolution of the population of each strategy

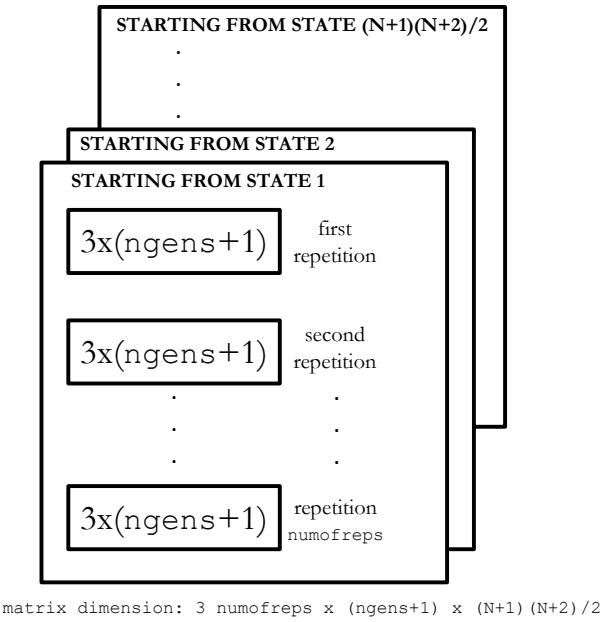


Figure 1.1: Structure of the population time evolution matrix.

`populationAnimation(allstatespergenNumOfPlayersHistory(4:6,:,20),chosen,names)` plots the state evolution in time, of the second repetition (4:6), starting from the initial state 20.

1.4.5 Function calls

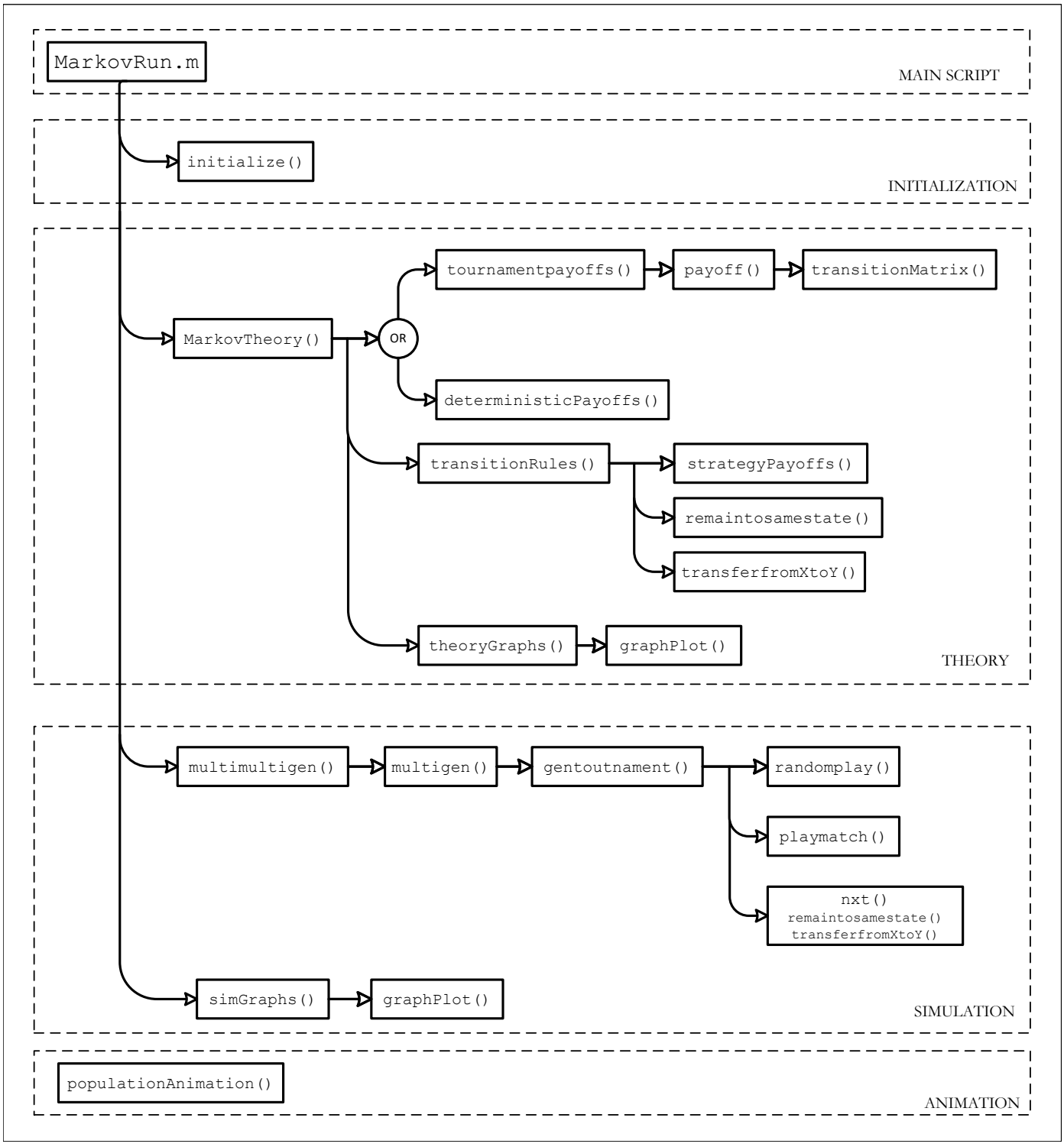


Figure 1.2: Function calls.

Chapter 2

Fitness Dynamics

2.1 Brief Overview

In the 1999's paper "Studies on Dynamics in the Classical Iterated Prisoner's Dilemma with Few Strategies" the authors imagine a Prisoner's dilemma like tournament, involving multiple dIfferent strategies. These strategies use a logical basis to derive their move against their opponent, ultimately choosing one out of the two following actions; either cooperate or defect. Each player plays a certain number of rounds with every other player. Once all encounters have been conducted we calculate the strategies' populations to start the next generation. However the encounters between players are not carried out one by one. To expedite the calculations of the Theoretical Fitness Axelrod Genetic Tournament, as we call it, we utilize the fact that when deterministic strategies play with each other, the emerging individual scores are always the same. Thus we can safely emerge to the score of each strategy by following the equation 2.1.

$$g_n(\mathbf{A}) = W_n(\mathbf{A})V(\mathbf{A}|\mathbf{A}) + W_n(\mathbf{B})V(\mathbf{A}|\mathbf{B}) + W_n(\mathbf{C})V(\mathbf{A}|\mathbf{C}) - V(\mathbf{A}|\mathbf{A}) \tag{2.1}$$

$$g_n(\mathbf{B}) = W_n(\mathbf{A})V(\mathbf{B}|\mathbf{A}) + W_n(\mathbf{B})V(\mathbf{B}|\mathbf{B}) + W_n(\mathbf{C})V(\mathbf{B}|\mathbf{C}) - V(\mathbf{B}|\mathbf{B}) \tag{2.2}$$

$$g_n(\mathbf{C}) = W_n(\mathbf{A})V(\mathbf{C}|\mathbf{A}) + W_n(\mathbf{B})V(\mathbf{C}|\mathbf{B}) + W_n(\mathbf{C})V(\mathbf{C}|\mathbf{C}) - V(\mathbf{C}|\mathbf{C}) \tag{2.3}$$

$$\tag{2.4}$$

$$\mathrm{t}(n) = W_n(\mathbf{A})g_n(\mathbf{A}) + W_n(\mathbf{B})g_n(\mathbf{B}) + W_n(\mathbf{C})g_n(\mathbf{C}) \tag{2.5}$$

$$W_{n+1}(\mathbf{A}) = \frac{\Pi W_n(\mathbf{A})g_n(\mathbf{A})}{\mathrm{t}(n)} \tag{2.6}$$

$$W_{n+1}(\mathbf{B}) = \frac{\Pi W_n(\mathbf{B})g_n(\mathbf{B})}{\mathrm{t}(n)} \tag{2.7}$$

$$W_{n+1}(\mathbf{C}) = \frac{\Pi W_n(\mathbf{C})g_n(\mathbf{C})}{\mathrm{t}(n)} \tag{2.8}$$

The acronyms used are explained in detail here:

- $g_n(A)$: Total score of strategy A per generation, on generation n
- $V(A|B)$: Score awarded to A when A encounters B

- $W_n(A)$: Population of strategy A, on generation n
- $t(n)$: The sum of the strategies' scores, on generation n

To enrich the Theoretical tournament we created two more ways to simulate a Tournament.

1. **Simulated Fitness Axelrod Genetic Tournament:** This is a computationally intensive but 1-to-1 accurate way of running the tournament. We make sure to meet each idividual player object with the other. To calculate the next generation's populations we use a fitness function 2.7.
2. **Simulated Imitation Axelrod Genetic Tournament** And this is an equally computationally intensive but 1-to-1 accurate way of running the tournament, though the next generation's populations are now calulated through imitation. K in number, randomly selected players, who do not belong to the optimal strategy, change strategy and adopt the the best scoring strategy. In case of multiple best scoring strategies, the picking procedure is made to be random.

2.2 Player Class

The player class encapsulates every attribute and method common to all strategies. The dIfferentiation occurs in the subclasses where we sometimes need to add attributes and/or methods. The move() method is implemented on each individual subclass/strategy. The common player attributes are:

- **index**: The index of the player in the game.
- **score**: The score of the player.
- **history**: A list of the player's moves for each round played (rows) when matched with dIfferent other players (columns) in the game.
- **move**: The move of the player in the current round. Can either be `0` (cooperate) or `1` (defect).

The methods implemented are simple setters and getter that give other classes of our program access to the attributes of the player class.

2.3 Strategies Index

2.3.1 Cooperate Class

Always cooperates.

2.3.2 Defect Class

Always defects.

2.3.3 Random Class

Makes a random move.

2.3.4 TitForTat Class

First cooperate, afterwards copy your opponent's last move.

2.3.5 Grim Class

Starts by cooperating, switches to permanent defection If the opponent ever defects. It tracks whether each opponent has defected using a flag array.

2.3.6 hard_tft Class

Cooperates on the first two moves, defects after at least one defection in the last two rounds, cooperates otherwise.

2.3.7 slow_tft Class

Cooperates on the first move, defects after two consecutive defections, and returns to cooperation after two consecutive cooperations by the opponent.

2.3.8 tf2t Class

Tit for 2 tat: Cooperates on the first move, defects after two consecutive opponent defections, cooperates otherwise.

2.3.9 mistrust Class

Defects on first move, then play what the opponent played on the previous move.

2.3.10 pavlov Class

Cooperates on the first move, then cooperates only If the two players made the same move.

2.3.11 per_CD Class

Periodically plays cooperate,defect.

2.3.12 per_kind Class

Periodically plays cooperate, cooperate, defect.

2.3.13 per_nasty Class

Periodically plays defect, defect, cooperate.

2.3.14 gradual Class

Cooperates on the first move, then defects n times after nth defections and calms down its opponent with two cooperations.

Algorithm 1 Gradual Strategy

```
For each opponent, track:
    total defections
    how many punishments remain
    how many calming cooperations remain
function SetMove(opponent's last move, opponent index, current round)
    if first round then
        Cooperate return
    end if
    if opponent defected then
        Increase total defections
    end if
    if currently punishing then
        Defect
        Decrease punish count return
    end if
    if currently calming then
        Cooperate
        Decrease calm count return
    end if
    if opponent just defected then
        Start punishing: defect as many times as total defections so far minus 1
        Then plan 2 calming cooperations
        Defect return
    end if
    Cooperate (default behavior)
end function
```

2.3.15 soft_majo Class

Plays the opponent's most played move, cooperates in case of equality.

2.3.16 per_ccccd Class

Periodically plays cooperate, cooperate, cooperate, cooperate, defect.

2.3.17 prober Class

Initially plays defect, cooperate, cooperate. On the third cooperate ascertains if the has opponent retaliated on rounds 2 and 3. If yes then the opponent isn't exploitable, thus continue as a tit-for-tat player. Otherwise the opponent is exploitable, thus continue as a defect player to maximize profits.

Algorithm 2 Prober Strategy: Brief overview

```
For each opponent, keep:
    State: what step we are in
    flag: whether the opponent is gullible
function SetMove(opponent's last move, opponent index)
    if State is 0 then
        Defect
        Go to State 1
    else if State is 1 then
        Cooperate
        Go to State 2
    else if State is 2 then
        Cooperate
        if opponent defected then
            Mark opponent as gullible
        end if
        Go to State 3
    else if State is 3 then
        if opponent defected then
            Mark opponent as gullible
        end if
        if opponent is gullible then
            Switch to tit-for-tat (copy opponent's move)
            Go to State 4
        else
            Switch to always defect
            Go to State 5
        end if
    else if State is 4 then
        Play tit-for-tat (copy opponent's last move)
    else if State is 5 then
        Always defect
    end if
end function
```

2.4 Axelrod Class

The Axelrod class plays out one generation of the game which comprises of a number of rounds. The class is initialized with the following parameters:

- **players**: A list of players that belong to play the game.
- **rounds**: The number of rounds the game is played.
- **currentRound**: The current round of the game.
- **payoffMatrix**: The matrix used to calculate the scores of the players based on their moves.

The methods of the class include the standard setters and getters but also materializes the main game loop. At first we set the current round and play the round using the `playRound()` method.

Algorithm 3 `playRound()` logic

```
function PLAYROUND
    for i ← i to N − 1 do
        for j ← i + 1 to N do
            Run the round between player i and player j
        end for
    end for
```

The `playRound()` logic can be seen below.
We run two nested loops to effectively simulating all encounters between players. It should be noted that the second loop begins at `i+1` to avoid enacting the same encounter twice.
Finally, the `encounter()` method is responsible for setting the moves of the two players involved, updating their history column which corresponds to the current round (row) and current opponent (column) and updating their current score.

Algorithm 4 `encounter()` logic

```
1: function ENCOUNTER(player1, player2, round)
2:     if round = 1 then
3:         each player sets move assuming no history
4:     else
5:         Each player sets move based on opponent's last move
6:     end if
7:     Assign payoffs to both players based on moves
8:     Update each player's history with their move
```

Now, in order to start the tournament, we have to convert the input arguments into actual players. The `InitPlayers()` method creates a map matching specIfic numbers with strategy constructors. The distinct populations determine how many times each strategy object should be created and the number of rounds contributes to the number of rows the history matrices should have.

Algorithm 5 The `initPlayers() logic`

```
Map strategy numbers to constructor functions
players ← empty list
for each (strategy, count) in (strategiesArray, populationsArray) do
    if strategy is valid then
        for i = 1 to count do
            Add new player using constructor(strategy) to players
        end for
    else
        ERROR(Invalid Strategy)
    end if
end for
for i = 1 to length(players) do
    Set players[i].index ← i
    Initialize history for players[i]
end for
return players =0
```

2.5 Genaxel Class

With the `genaxel` class, we simulate and analyze strategy evolution within the Axelrod tournament. The implementation facilitates dIfferent modes of fitness calculation and evolutionary dynamics.It also supports multiple tournament styles and population update mechanisms.

To begin with, the `TheoreticalFitness` function simulates evolutionary fitness for a population of strategies by conducting pairwise Axelrod tournaments. It computes a score matrix `V` where each strategy plays against every other, calculates a fitness score `Gn` for each strategy based on its interactions and current population, and then updates the next generation population `Wn` propportionally. The function supports multiple rounding schemes, which we explore further down in this report.

On the contrary, the `SimFitness` function collects indivisual scores for each strategy, and aggregates these to compute a total score. Based on each strategy's share of the total score, the function updates the population vector `Wn` for the next generation, scaling and rounding the values proportionally. This models evolutionary selection based on realtive performance in a full-population tournament setting.

Finally, in the `ImitationSim` function, each strategy's total score is calculated based on its interactions with others and normalized per individual to get average player scores. The strategy with the highest average score is identIfied, and up to `K` indivisuals from other non-zero populations are converted to imitate this best-performing strategy. This model "learns" by imitation, where successful strategies attract followers over time.

Algorithm 6 TheoreticalFitness logic

Require: obj, b strategies, pop0, T, rounding
Ensure: obj, Wn, V

Initialize $V \leftarrow$ payoff Matrix through oneVone , $Wn \leftarrow pop0$, $Gn \leftarrow 0$, $Tn \leftarrow 0$
$totalPlayers \leftarrow \sum Wn$
for $i = 1$ to $|\texttt{strategies}|$ **do**
    for $j = 1$ to $|\texttt{strategies}|$ **do**
        $V[i][j] \leftarrow$ score of player i
        $Gn[i] \leftarrow Gn[i] + V[i][j] \cdot Wn[j]$
    end for
    $Gn[i] \leftarrow Gn[i] - V[i][j]$
    $Tn \leftarrow Tn + Gn[i] \cdot Wn[i]$
end for
Compute new Wn using Gn, Tn, and chosen rounding method
    return obj, Wn, V

Algorithm 7 sim_fitness logic

```
1: function SIM_FITNESS(strategies, pop0, T, b)
2:     Initialize tournament with strategies and payoffs
3:     Run the tournament
4:     Set popscore to zero array
5:     Set totalscore to 0
6:     Set totalplayers to sum of pop0
7:     Set Wn to pop0
8:     Set counter to 1
9:     for each strategy i do
10:        for each player in strategy i do
11:            Add player's score to popscore[i]
12:            Increase counter by 1
13:        end for
14:        Add popscore[i] to totalscore
15:    end for
16:    for each strategy i do
17:        Set Wn[i] to floor(totalplayers * popscore[i] / totalscore)
18:    end for
       return Wn
```

Algorithm 8 imitation_sim logic

```
function IMITATION_SIM(strategies, pop0, K, T, b)
    Set Wn to pop0
    Create empty matrix V[strategy][strategy]
    Create score array Gn initialized to zeros
    Set oneVone to [1, 1]
    for each strategy i do
        for each strategy j do
            Create temporary player list with strategy i and j
            Initialize a new tournament with those players
            Run the tournament
            Set V[i][j] to score of strategy i against strategy j
            Add V[i][j] × Wn[j] to Gn[i]
        end for
        Subtract V[i][i] from Gn[i]
    end for
    Set popscore to Gn
    Compute average score per player: playerscore[i] = popscore[i] / Wn[i]
    Find maximum value in playerscore
    Find all indices with that value
    Randomly choose one indexBest among them
    Set Bestscore to playerscore[indexBest]
    Set newIndex to indexBest
    for i from 1 to K do
        Find all zero-population strategy indices
        if there are at least two non-zero populations then
            while newIndex equals indexBest or newIndex in zero-population list do
                Select newIndex randomly from strategies
            end while
            Decrease Wn[newIndex] by 1
            Increase Wn[indexBest] by 1
            Reset newIndex to indexBest
        end if
    end for
     return Wn
```

We noticed that the way we implemented `genaxel` was time consuming, because of recurring initialization of Axelrod tournament wich essentially calculated time and again th epayoff Matrix, `V`.
In the following, improved version, we made the function "oneVone" which is only called once, at the beginning of `genaxel` in `TourTheFit()`.

Algorithm 9 one_v_one

```
1: function ONE_V_ONE(strategies, T, b)
2:     Create matrix V of zeros with size [strategies][strategies]
3:     Set population array to [1, 1]
4:     for each strategy i do
5:         for each strategy j do
6:             Create temporary pair [strategy i, strategy j]
7:             Initialize a new tournament with that pair
8:             Run the tournament
9:             Set V[i][j] to the score of player 1
10:        end for
11:    end for
12:    Store V in object V
```

2.6 Script

The Script is our means to configure the program and the dIfferent kinds of tournaments we engineered. Through the script one can edit the following paramenters:

- `sim_mode`: The simulation mode. Can take one of 4 values:
  - `Axel`: Axelrod, is a simple Axelrod tournament.
  - `TourTheFit`: Tournament Theoretical Fitness, is a genetic algorithm implementing the Axelrod tournament for many generations. The mechanism for determining the next generation includes finding the ratio of the total score each strategy accumulated and distributing the total number of players to the individual strategies based on that ratio. The encounters here are only theoretical. We save on computation time by encountering the strategies and multiplying by their populations.
  - `TourSimFit`: Tournament Simulated Fitness, is a genetic algorithm implementing the Axelrod tournament for many generations. This algorithm's mechanism is the same as TourTheFit but this time the players have to be distinct and truly encounter one another. More computationally expensive but also able to accurately simulate random strategies.
  - `TourSimImit`: Tournament Simulated Imitation, genetic algorithm implementing the Axelrod tournament for many generations. The mechanism for determining the next generation has the suboptimal players converting to the best performing strategy.
- `meeting_mode`: Meeting mode, lets the user choose from a number of dIfferent meetings created as experiments by the authors of "" to highlight various interesting emerging States.

The user is also able to run custom tournaments by selecting the custom meeting mode and thereafter changing any of the following parameters:

- **strategies**: The strategies participating in the tournament.
- **populations**: The populations of each strategy.
- **payoffMatrix**: The matrix that determines the scores of the players.
- **rounds**: The number of rounds to be played each generation of the tournament.
- **generations**: The number of generations to be played in the tournament.
- **rounding**: The rounding method used to have integer populations each generations while, also, keeping the initial total population.
- **K**: The number of players that will imitate the best performing strategy.

Running the script.m file runs the tournament with the parameters set inside it. The results are plotted for visualization.

2.7 Meetings

Retaining the exact same number of players, each generation, requires rounding up the emerging number of players each strategy has at the end of each generation and then redistributing the decimal parts. Ultimately, the decimal parts in our 3-strategy meetings add up to either one or two, and the distribution method we found to be the most fair was:

- If the remainder sum is 1, we round up each strategy to the previous integer and distribute the decimal part to the strategy that is the closest to its next integer.
- If the remainder sum is 2, we round up each strategy to the previous integer and distribute the decimal part to the two strategies that are the closest to their next integer. 1 to each.
- If the remainder sum is 0, there are no decimal parts.

We also devised a similar method which, this time, distributes the decimal parts to the top two strategies in terms of total population. We used both methodologies to conduct our experiments. However, when comparing our results with the 1999's paper "Studies on Dynamics in the Classical Iterated Prisoner's Dilemma with Few Strategies" the graphs we produced were mostly coherent but ultimately dIfferent. These dIfferences can be largely attributed to the mutable nature of the simulations hidden in programming layers of abstraction and the non-disclosed method of rounding used by the authors of the paper. They only Stated that: "All divisions being rounded to the nearest lower integer.", which is not accurate based on their results which seem to be retaining their initial total population. This will be an attempt to replicate the paper's results while attributing our dIfferences and trying to make the same point the original authors were trying to make regardless of the dIfferences. Our rounding logic is presented in the next page.

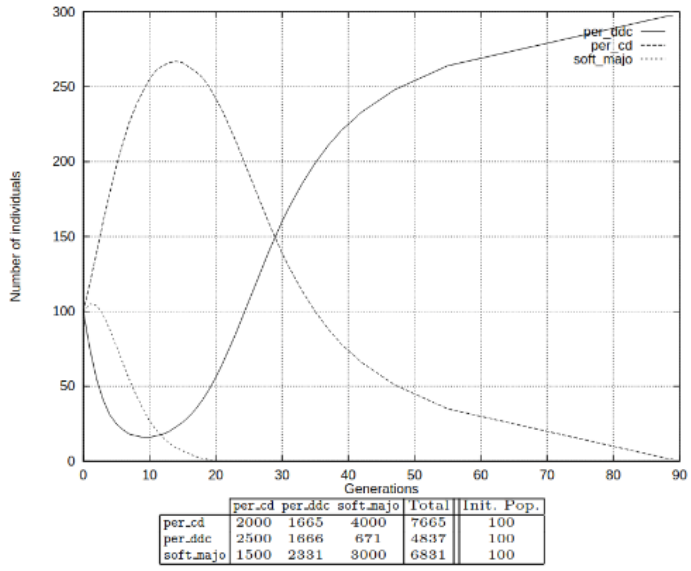
It is important to note that when comparing equal quantities there is no clear ranking between remainders. Thus the program decides where to distribute the decimal parts based on the order of the strategies. This could very well be consequential enough to skew the results in a different direction. Let's see how our results compare relative to the ones in the paper. Along with our own rounding methods we managed to reverse engineer the paper's method. It involves pinning the total population at the start of the code so as to only compute the strategies' score ratios. We then floor the emerging individual populations but no matter the decimal parts the total population of each round cannot become $constant\_total\_population - 3$. Since the results are exactly the same there isn't any commentation to be done. Thus we will only depict them as proof we recreated the paper's results.

Algorithm 10 Decimal redistribution logic

```
1: if rounding is paper then
2:     for each strategy do
3:         Calculate share of players
4:         Round down to nearest whole number
5:     end for
6: end if
7: if rounding is pop then
8:     for each strategy do
9:         Calculate share of players
10:        Save the decimal part
11:        Round down to whole number
12:    end for
13:    Count how many players have to be redistributed
14:    Sort strategies by largest population count
15:    for each leftover player do
16:        Give to strategies ranking highest
17:    end for
18: end if
19: if rounding is dec then
20:    for each strategy do
21:        Calculate share of players
22:        Save the decimal part
23:        Round down to whole number
24:    end for
25:    Count how many players have to be redistributed
26:    Sort strategies by largest decimal part
27:    for each leftover player do
28:        Give to strategy with largest decimal
29:    end for
30: end if
31: if rounding is off then
32:    for each strategy do
33:        Just calculate the exact, non-rounded share
34:    end for
35: end if=0
```

2.7.1 Defectors may be strong

This meeting is supposed to show how defecting more often can be beneficial, for the defector, in chaotic environments. When rounding using the "off" method soft majority does not get eradicated leading to different behavior. Same are the results we get with the "dec" method too, only this time we increase the number of generations to observe the entire phenomenon. It is evident that the oscillations converge to a stable periodic State.



	per_cd	per_ddc	soft_majo	Total	Init. Pop.
per_cd	2000	1665	4000	7665	100
per_ddc	2500	1666	671	4837	100
soft_majo	1500	2331	3000	6831	100

Figure 2.1: Defectors may be strong Original Plot

Strategy	**Population**
soft_majo	100
per_cd	100
per_ddc	100
Notes:	
Rounds = 1000	
Generations = 90	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

Table 2.1: Initial populations

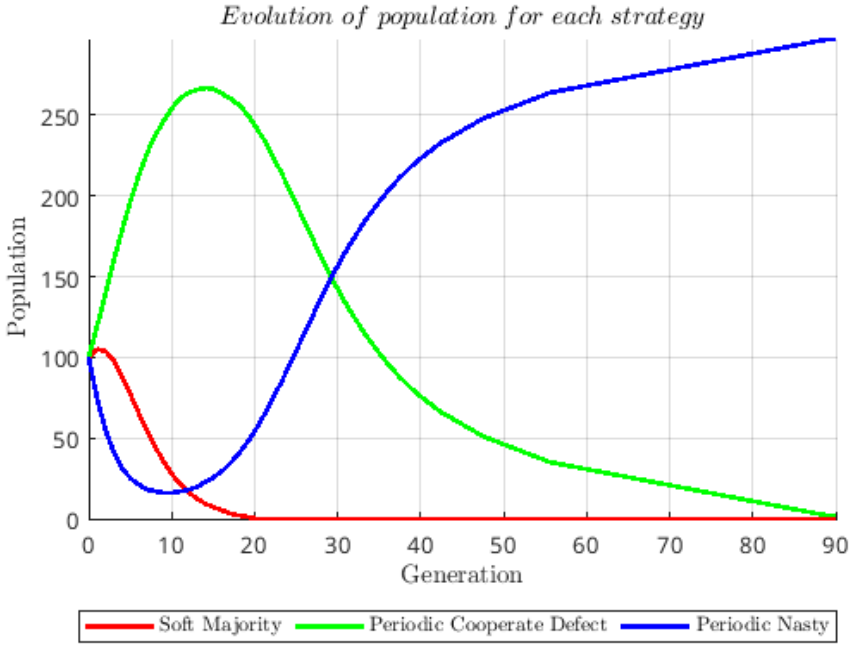


Figure 2.2: Defectors may be strong Paper Plot

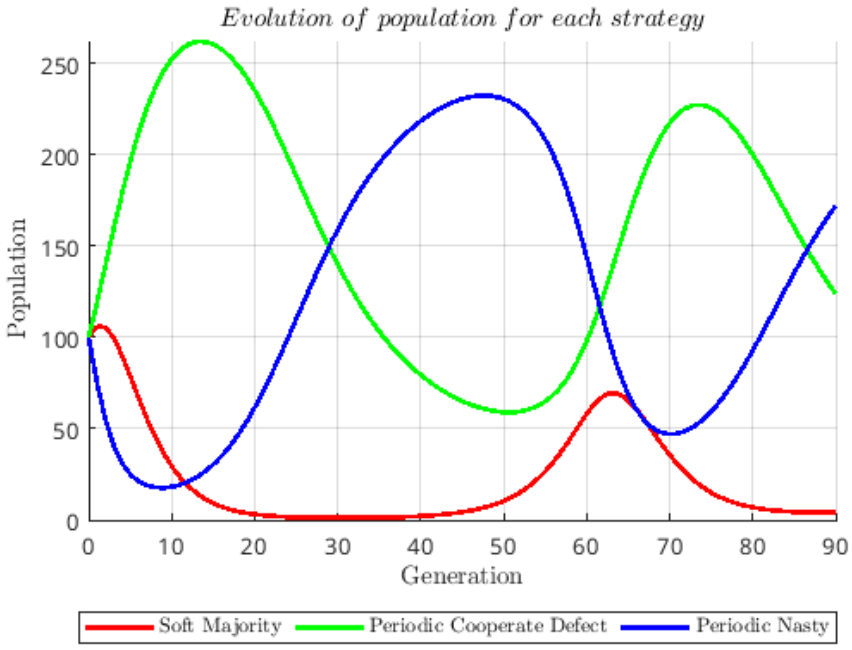


Figure 2.3: Defectors may be strong Off Plot

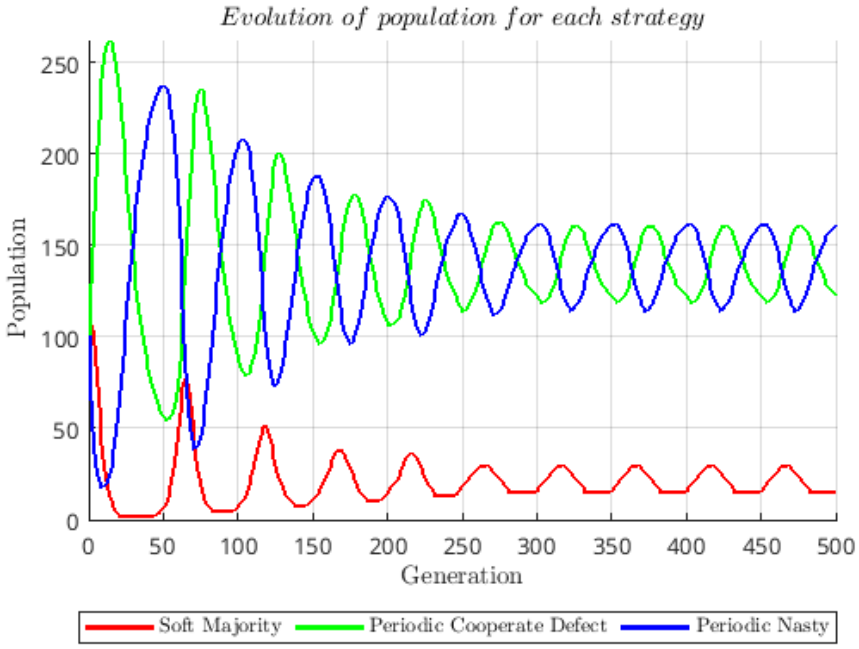


Figure 2.4: Defectors may be strong Dec Plot

When rounding using the "pop" method we replicate the results of the paper.

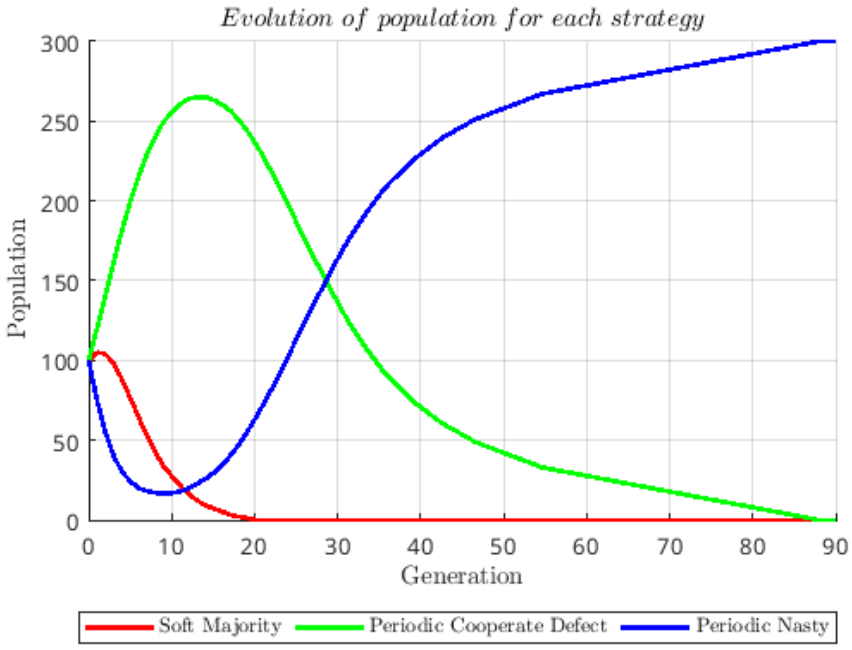
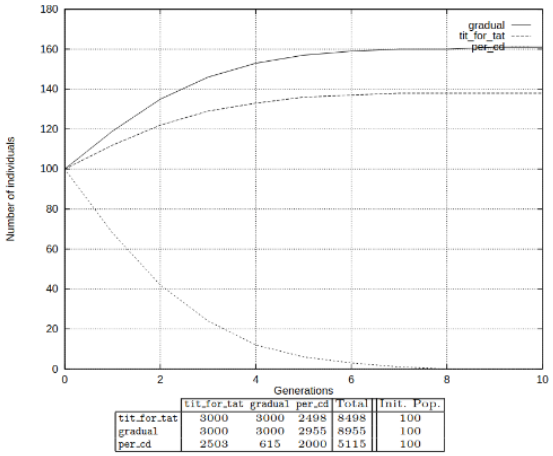


Figure 2.5: Defectors may be strong Pop Plot

2.7.2 Monotonous convergence

This meeting simulates clear monotonous convergence. The paper claims this is the most common outcome of the experiments they ran. Here all methods "pop", "dec" and "off" methods produced identical results recreating the paper's plots.



	tit_for_tat	gradual	per_cd	Total	Init. Pop.
tit_for_tat	3000	3000	2498	8498	100
gradual	3000	3000	2955	8955	100
per_cd	2503	615	2000	5115	100

Figure 2.6: Monotonous convergence Original Plot

Strategy	**Population**
tft	100
per_cd	100
gradual	100
Rounds = 1000	
Generations = 10	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

Table 2.2: Initial populations

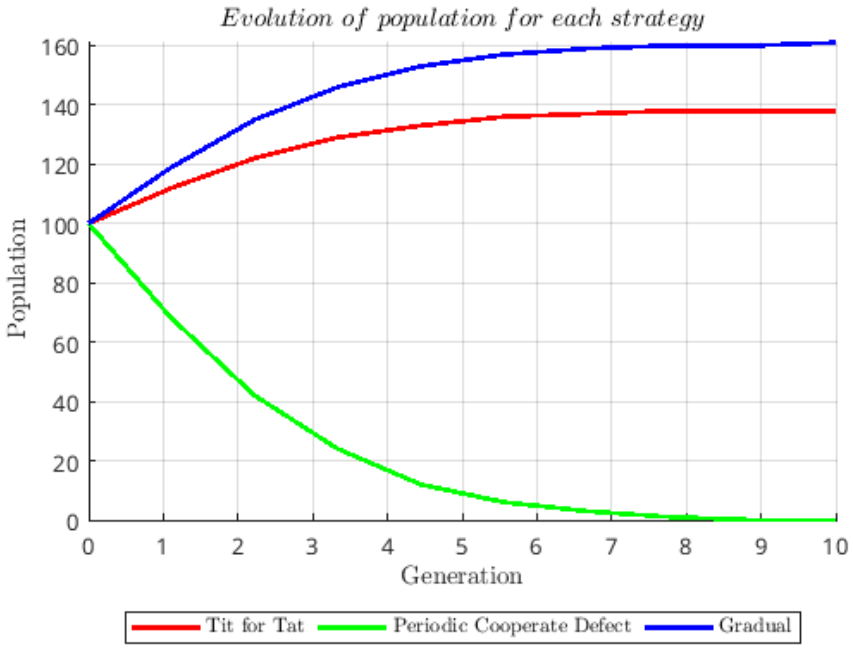


Figure 2.7: Monotonous convergence Paper Plot

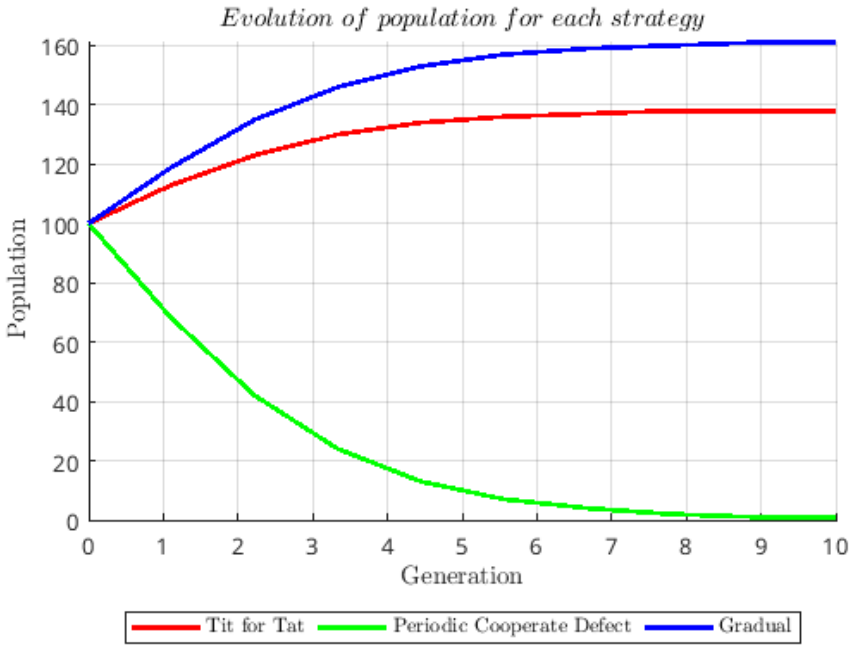


Figure 2.8: Monotonous convergence Dec Plot

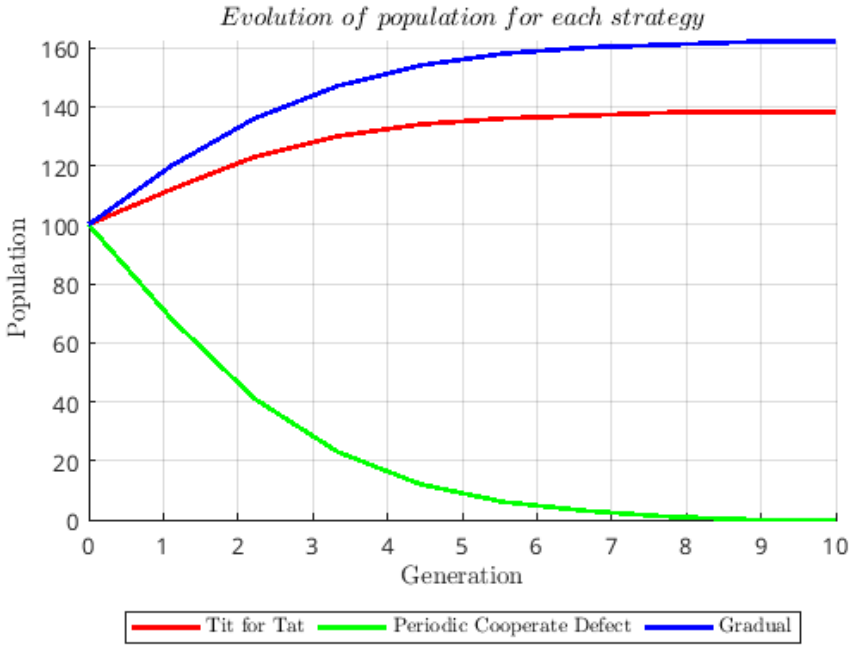


Figure 2.9: Monotonous convergence Pop Plot

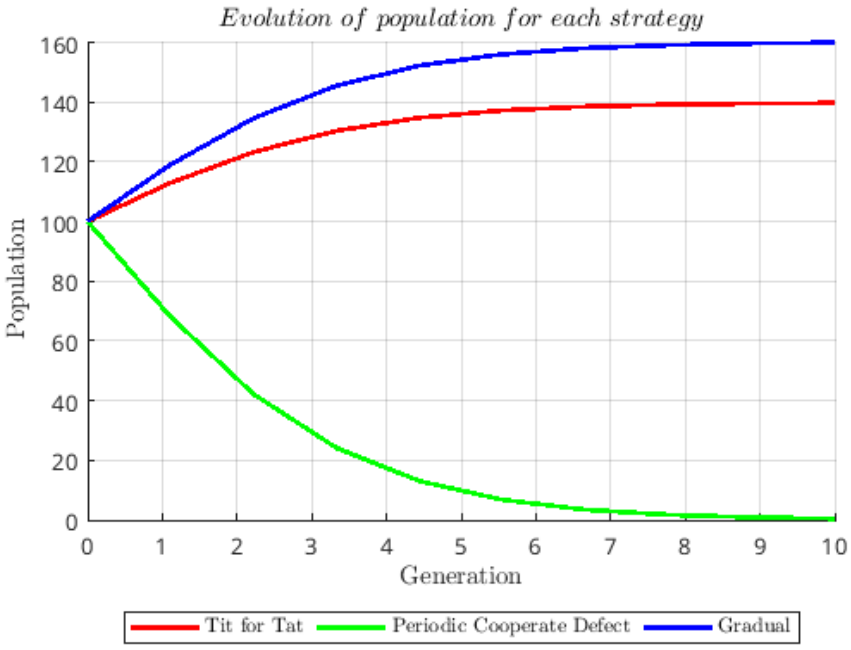


Figure 2.10: Monotonous convergence Off Plot

2.7.3 Attenuated oscillatory movements

Here we see decreasing oscillations that reach an equilibrium. All rounding methods are again very close to the paper.

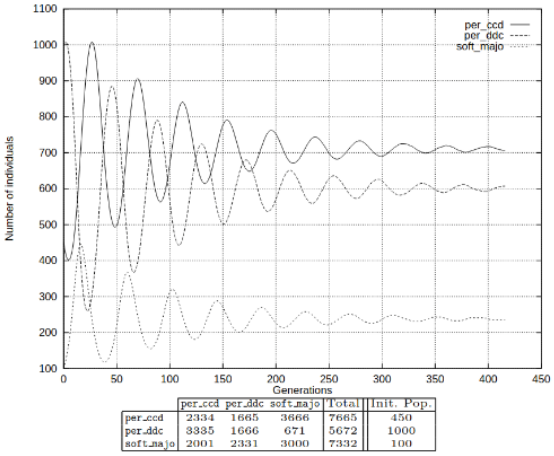


Figure 2.11: Attenuated oscillatory movements Original Plot

Strategy	Population
soft_majo	100
per_ccd	450
per_ddc	1000
Rounds = 1000	
Generations = 450	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

Table 2.3: Initial populations

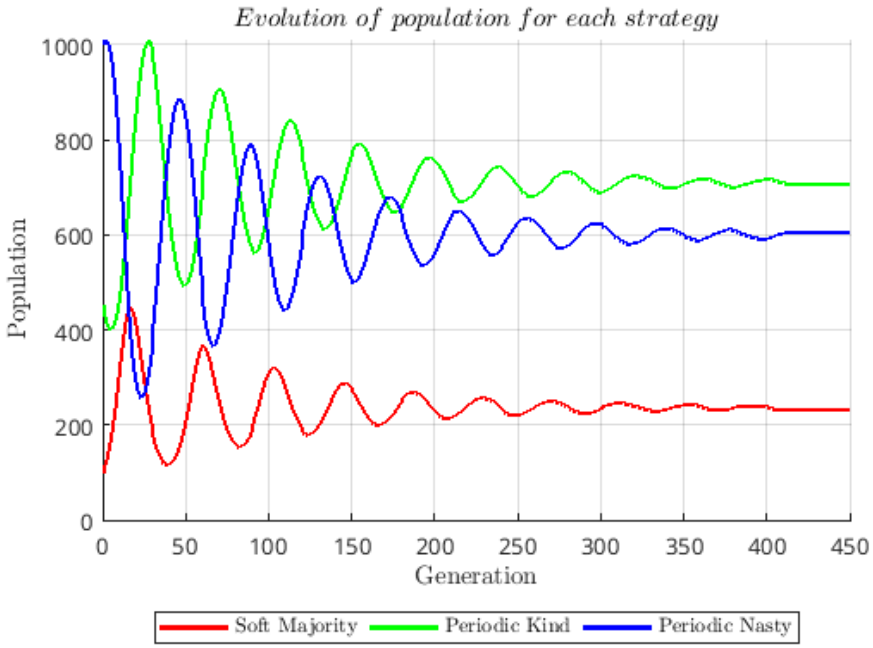


Figure 2.12: Attenuated oscillatory movements Paper Plot

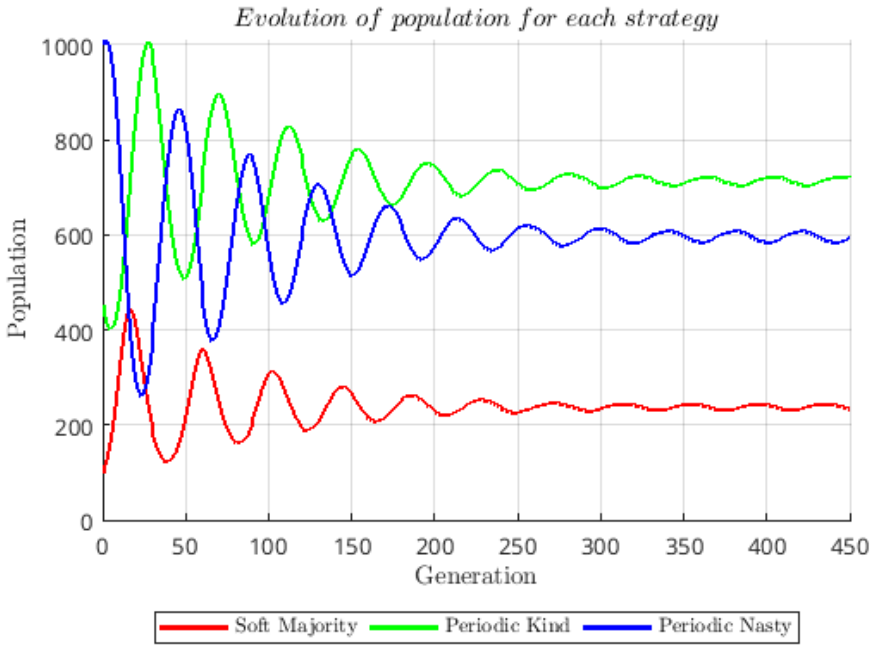


Figure 2.13: Attenuated oscillatory movements Dec Plot

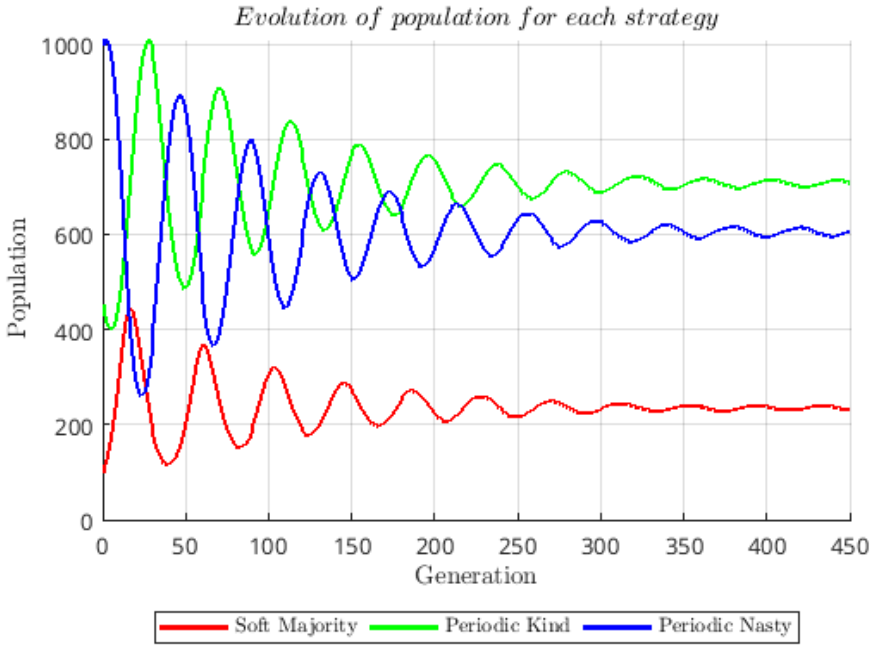


Figure 2.14: Attenuated oscillatory movements Pop Plot

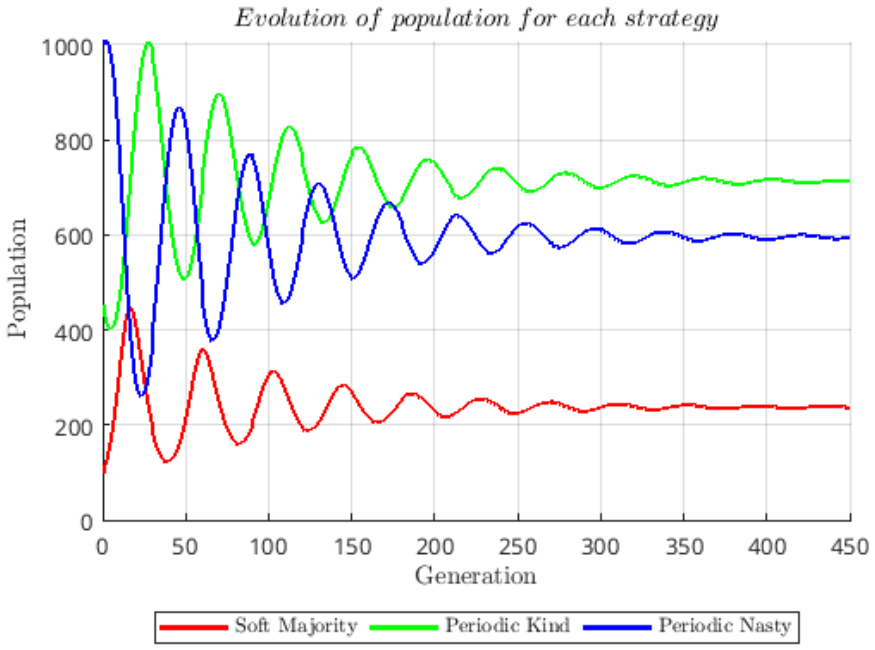


Figure 2.15: Attenuated oscillatory movements Off Plot

2.7.4 Periodic movements

The periodic movements meeting highlights the periodicity and constant amplitude of the oscillations. The "dec" rounding method comes closest to replicating the paper's results. The "pop" method produces oscillations that overlap due to the oscillations' bigger amplitude. Finally, the "off" method creates oscillations that soon enough reach to a stable State.

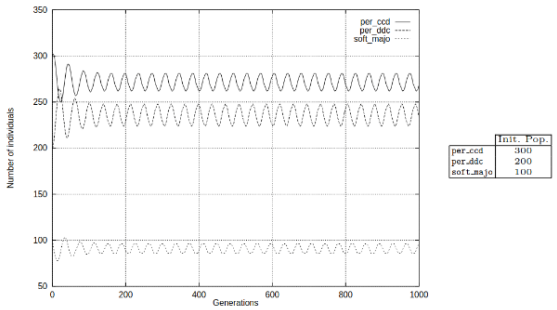


Figure 2.16: Periodic movements Original Plot

Strategy	Population
soft_majo	100
per_ccd	300
per_ddc	200
Rounds = 1000	
Generations = 1000	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

Table 2.4: Initial populations

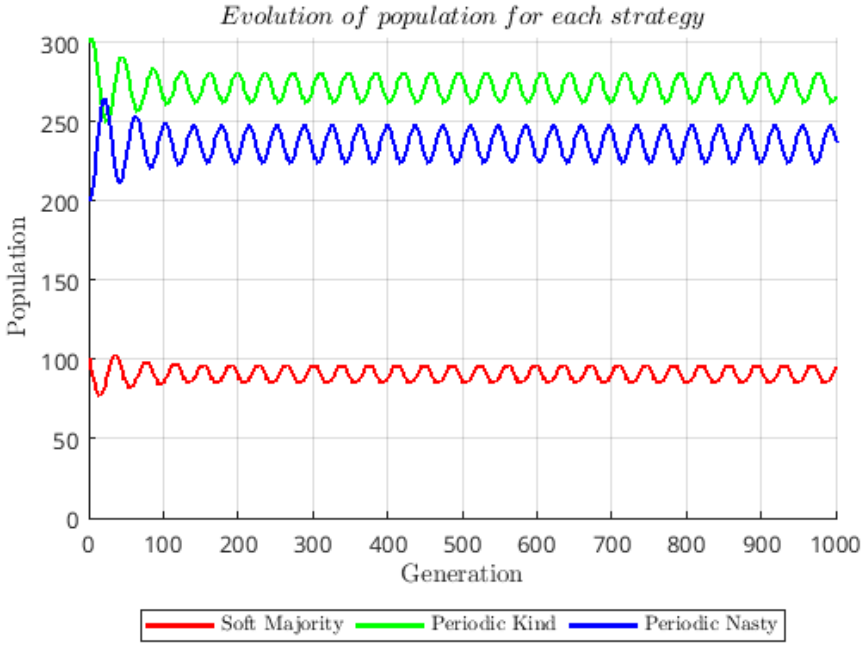


Figure 2.17: Periodic movements Paper Plot

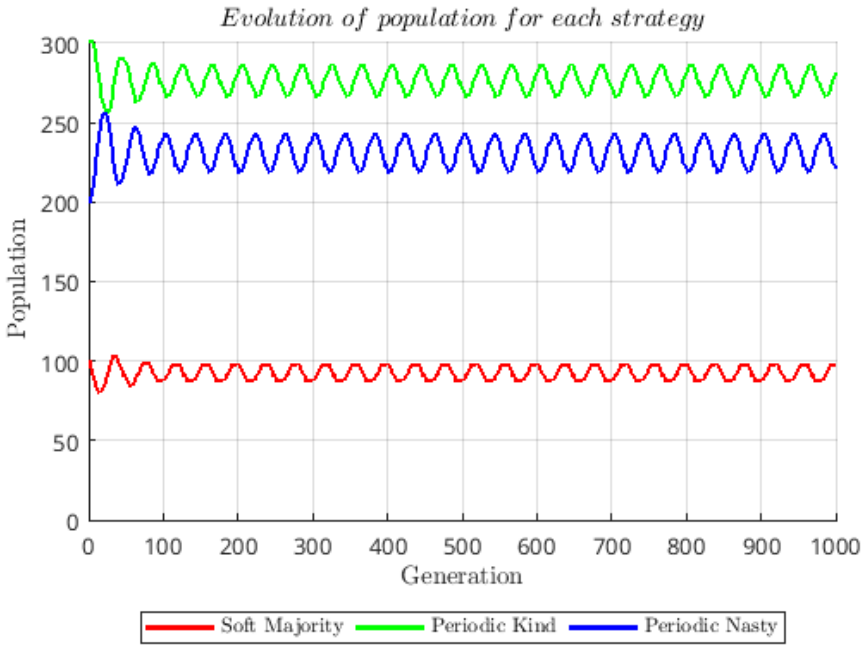


Figure 2.18: Periodic movements Dec Plot

Using the "pop" method increased the amplitude of the oscillations.

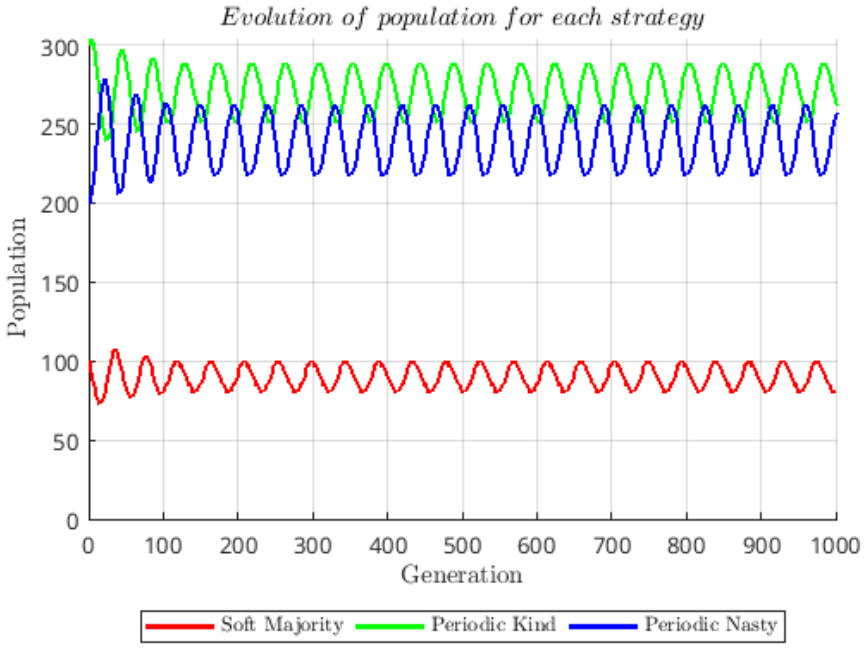


Figure 2.19: Periodic movements Pop Plot

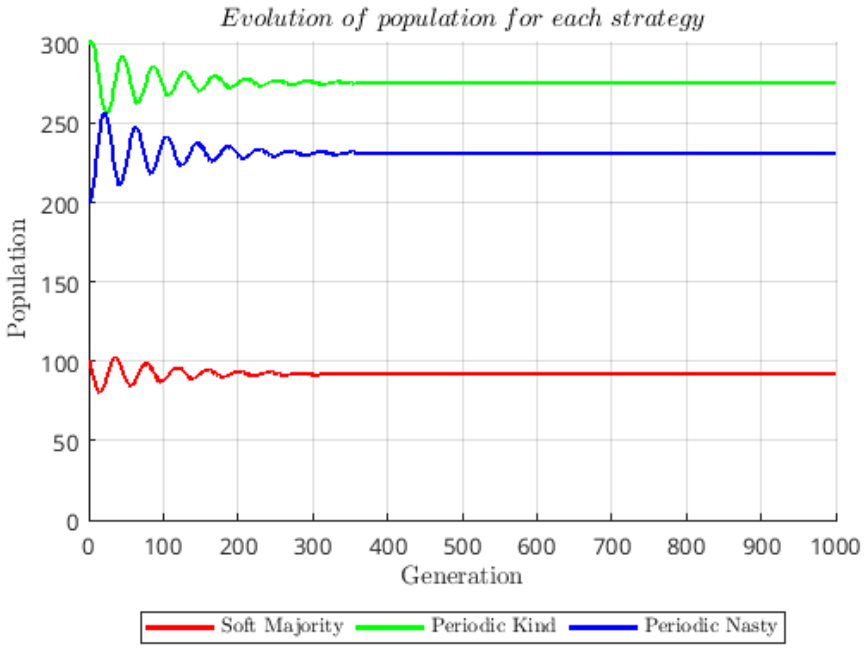


Figure 2.20: Periodic movements Off Plot

2.7.5 Increasing oscillations

The increasing oscillations effect in an environment where the total players remain constant can be attributed to Periodic Nasty and Periodic Cooperate Defect who seem to be getting their extra population from driving soft majority into extinction and from being out of phase with each other. "dec" and "off" methods seem be following the attenuated oscillatory movements wand the "pop" method is a bit slower at converging.

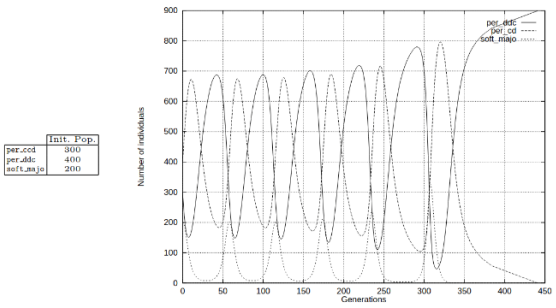


Figure 2.21: Increasing oscillations Original Plot

Strategy	Population
soft_majo	200
per_cd	400
per_ddc	300
Rounds = 1000 Generations = 450 Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

Table 2.5: Initial populations

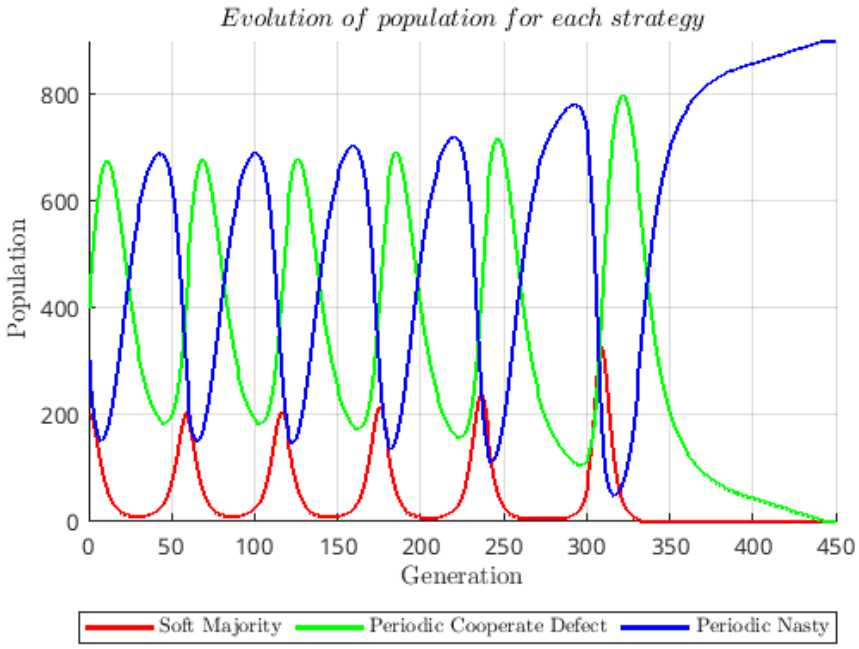


Figure 2.22: Increasing oscillations Paper Plot

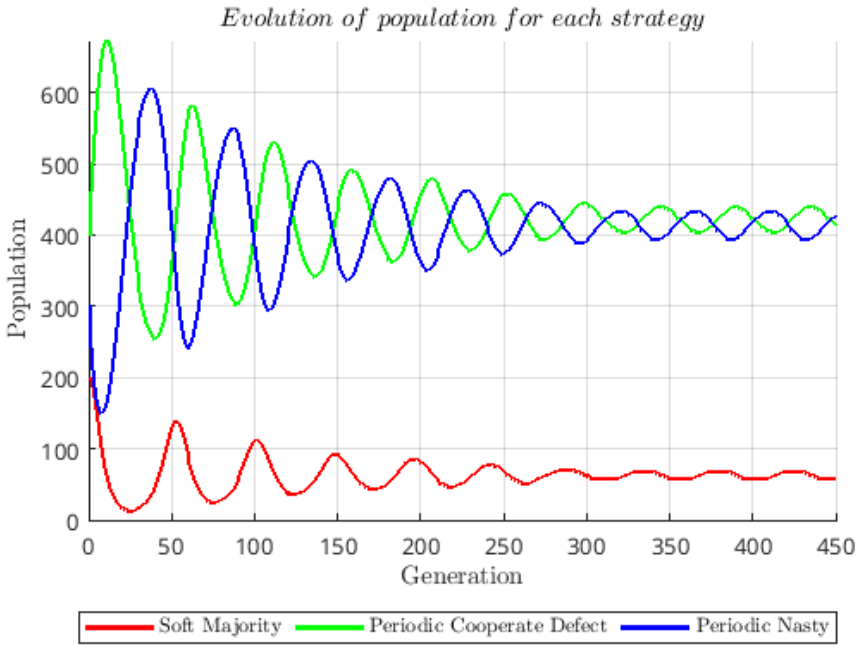


Figure 2.23: Increasing oscillations Dec Plot

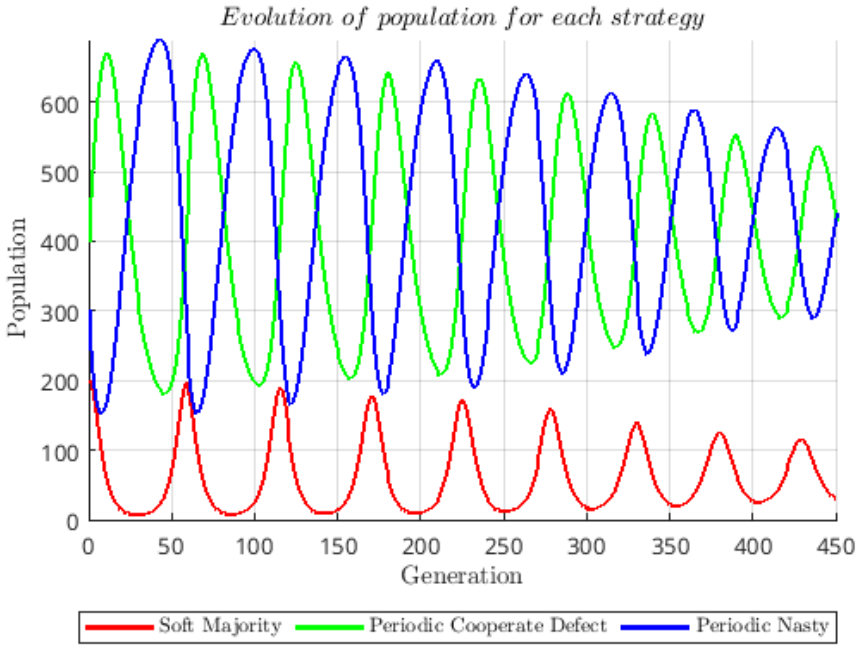


Figure 2.24: Increasing oscillations Pop Plot

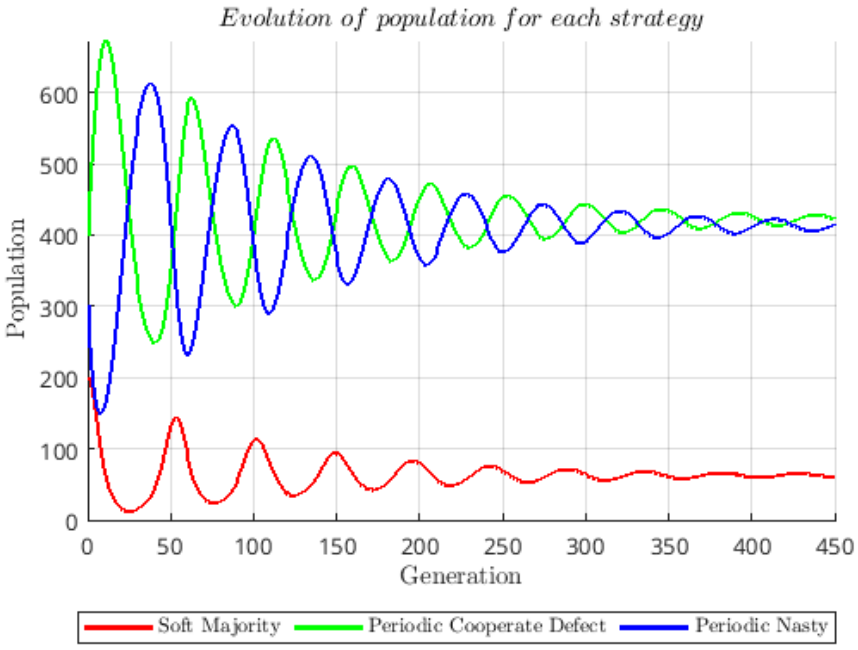
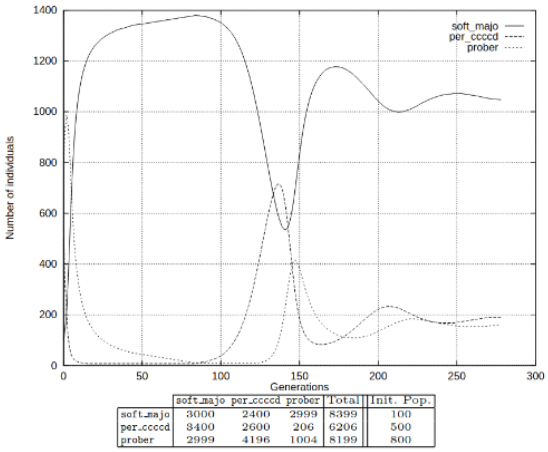


Figure 2.25: Increasing oscillations Off Plot

2.7.6 Disordered oscillations

When examining the disordered oscillations, "dec" and "off" methods almost recreate the paper's results at a more subtle way. "pop" very quickly concluded its tournament by having soft majority dominate the other two strategies by round 50.



	soft_majo	per_ccccd	prober	Total	Init. Pop.
soft_majo	3000	2400	2999	8399	100
per_ccccd	3400	2600	206	6206	500
prober	2999	4196	1004	8199	800

Figure 2.26: Disordered oscillations Original Plot

Strategy	**Population**
soft_majo	100
per_ccccd	500
prober	800
Rounds = 1000 Generations = 300 Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

Table 2.6: Initial populations

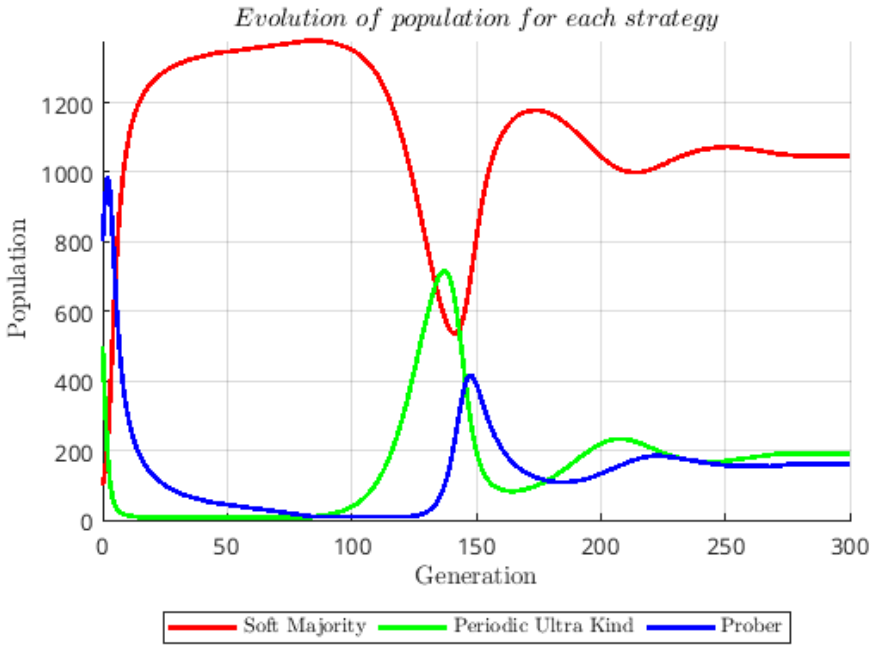


Figure 2.27: Disordered oscillations Paper Plot

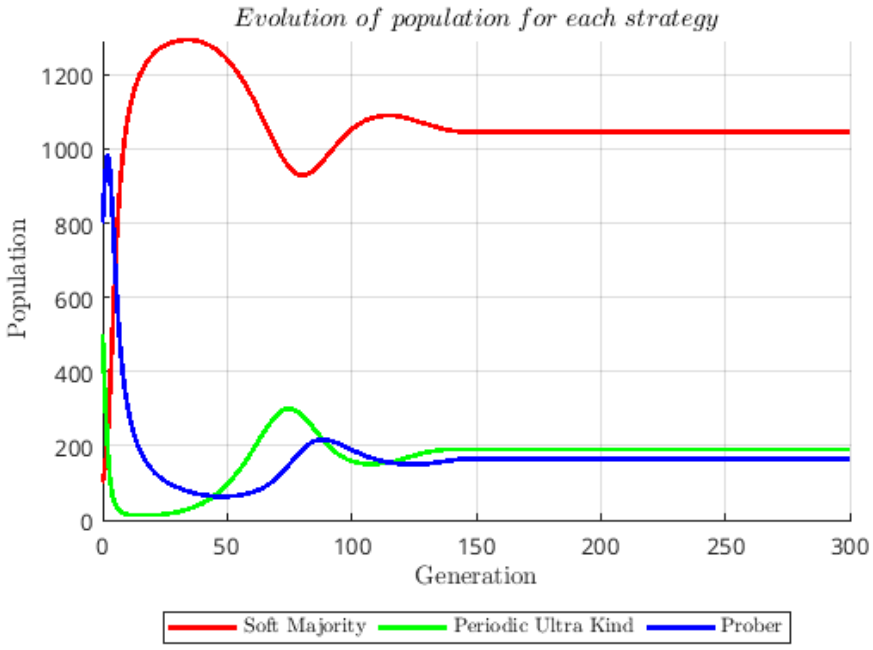


Figure 2.28: Disordered oscillations Dec Plot

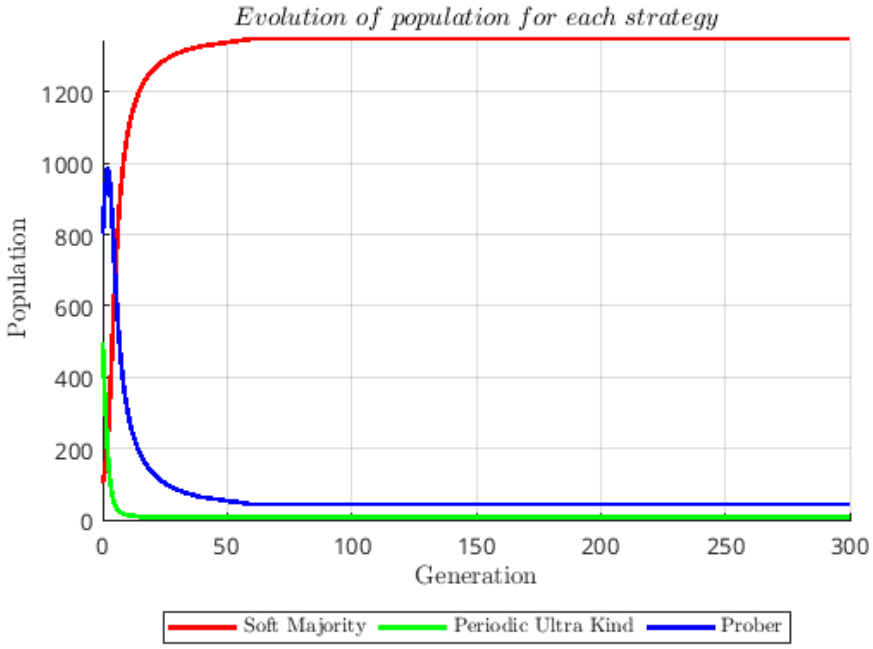


Figure 2.29: Disordered oscillations Pop Plot

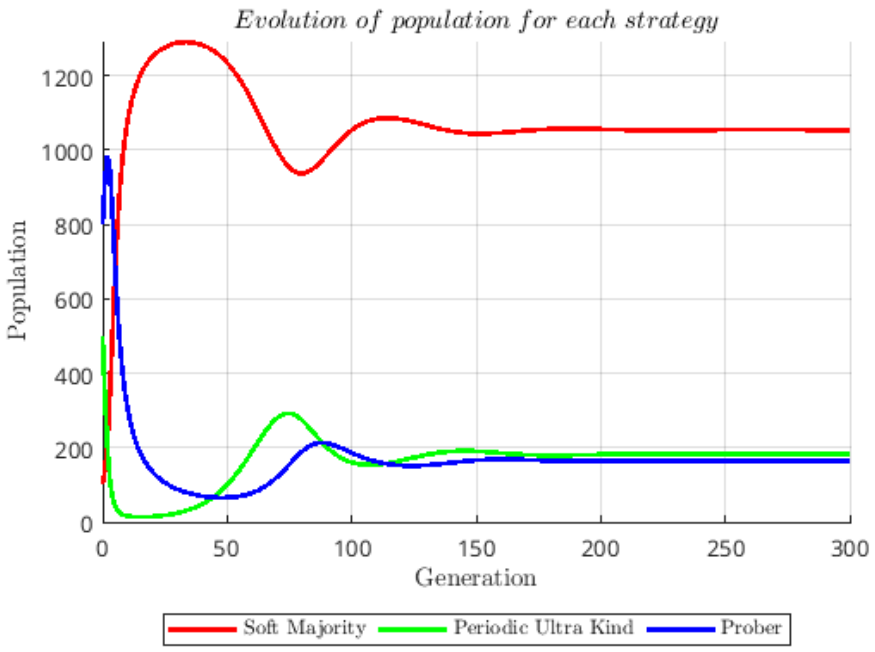


Figure 2.30: Disordered oscillations Off Plot

2.7.7 Population size sensitivity

Going forward the paper decides to explore the tournament's sensitivity in slight changes of various parameters starting with the initial population size. Other than the fact that the crucial point where the oscillations stop gets shifted at periodic defect-defect-cooperate population: 235 into 236, instead of 244 into 245, the plots "dec" and "pop" recreate the same phenomenon. Furthermore, a pattern that seems to be emerging is the non-oscillating State that the "off" method reaches. This can most likely be attributed to the fact that at some point the distribution of the decimal points becomes periodic in our own methods. "off" which doesn't redistribute the decimal parts has no problem converging to a final state.

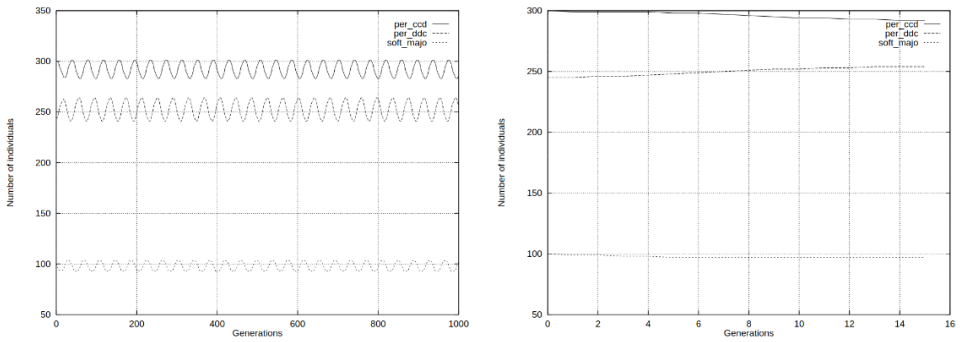
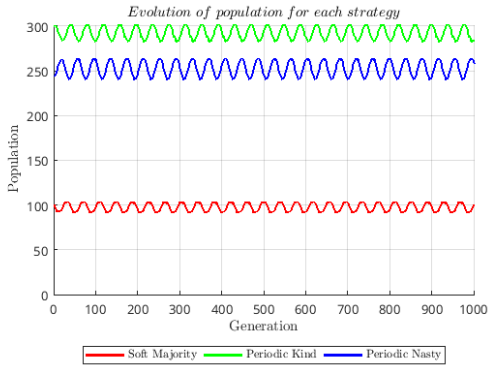


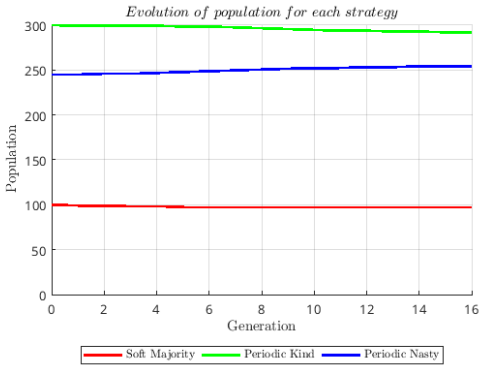
Figure 2.31: Population size sensitivity Original Plot

Strategy	**Population**
soft_majo	100
per_ccd	300
per_ddc	235 (or 244) $\rightarrow$ 236 (or 245)
Notes:	
Rounds = 1000	
Generations = 1000 $\rightarrow$ 16	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

Table 2.7: Initial populations

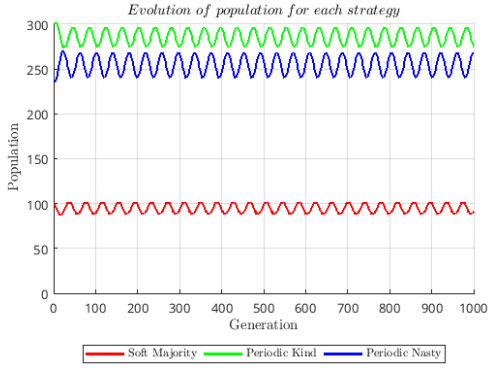


(a) Before

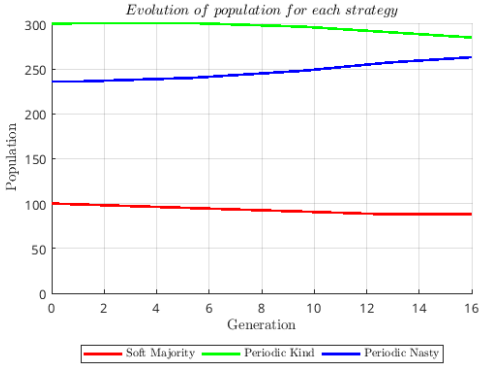


(b) After

Figure 2.32: Side-by-side comparison of population size sensitivity before and after Paper

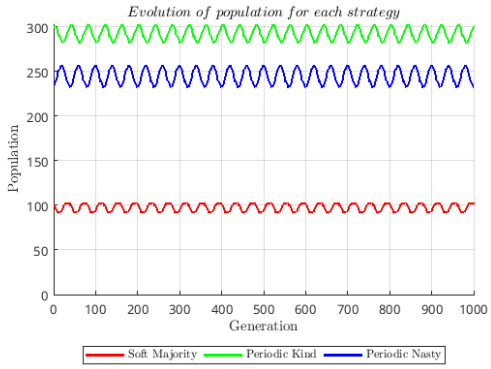


(a) Before

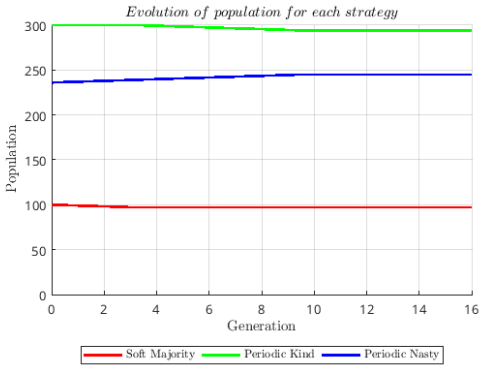


(b) After

Figure 2.33: Side-by-side comparison of population size sensitivity before and after Pop

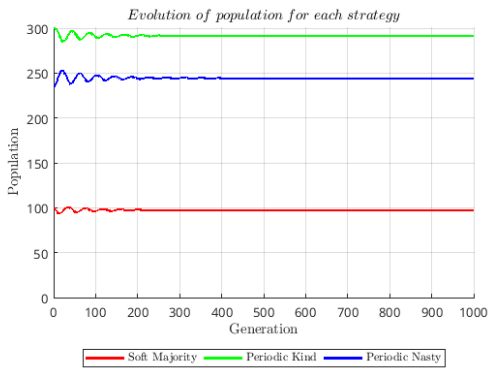


(a) Before

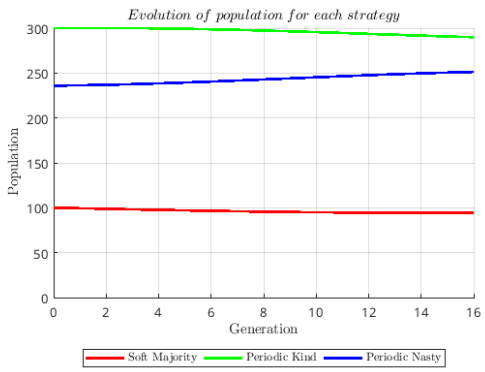


(b) After

Figure 2.34: Side-by-side comparison of population size sensitivity before and after Dec



(a) Before



(b) After

Figure 2.35: Side-by-side comparison of population size sensitivity before and after Off

2.7.8 Population size sensitivity 2

This is an example of generally oscillatory behavior becoming eager to converge through either smoother oscillations or steady states.

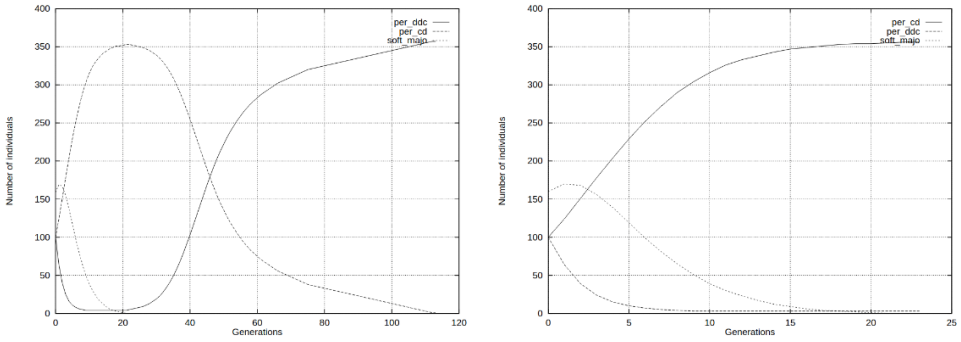
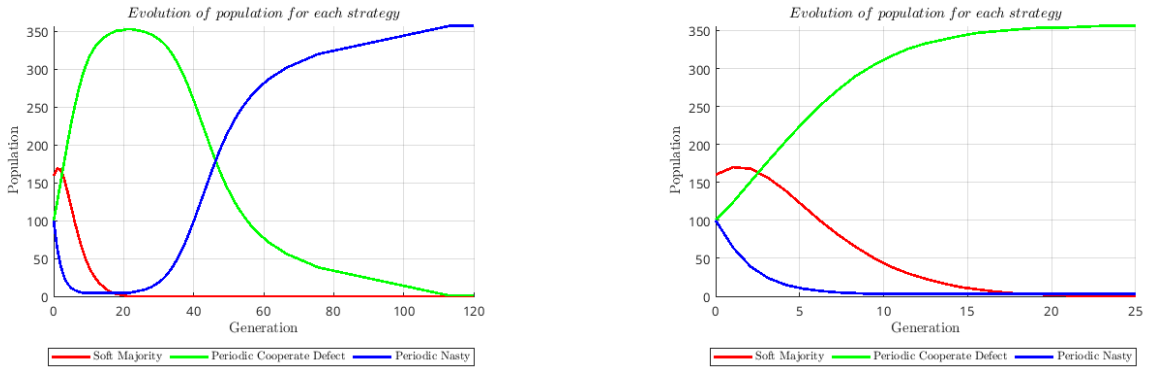


Figure 2.36: Population size sensitivity 2 Original Plot

Strategy	**Population**
soft_majo	100
per_ccd	300
per_ddc	159 → 160
Notes:	
Rounds = 1000	
Generations = 120 → 50	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

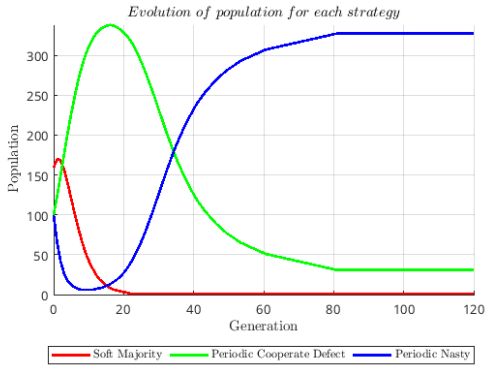
Table 2.8: Initial populations



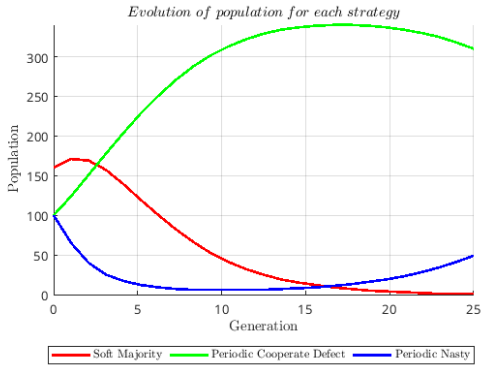
(a) Before

(b) After

Figure 2.37: Side-by-side comparison of population size sensitivity 2 before and after Paper method

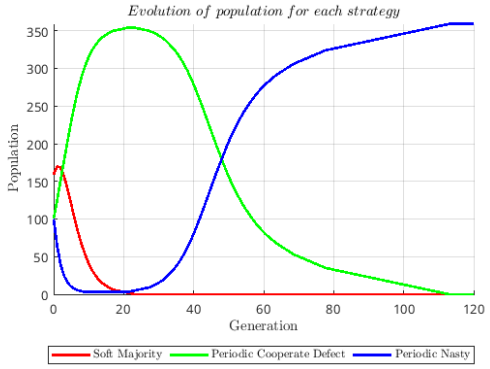


(a) Before

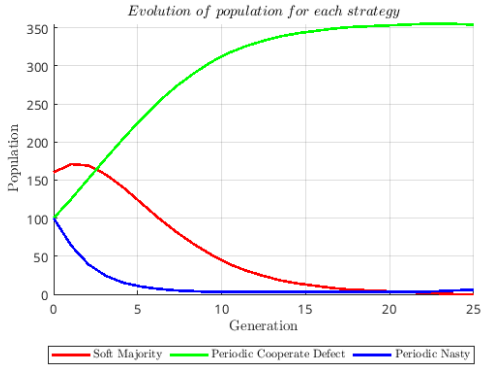


(b) After

Figure 2.38: Side-by-side comparison of population size sensitivity 2 before and after Dec method

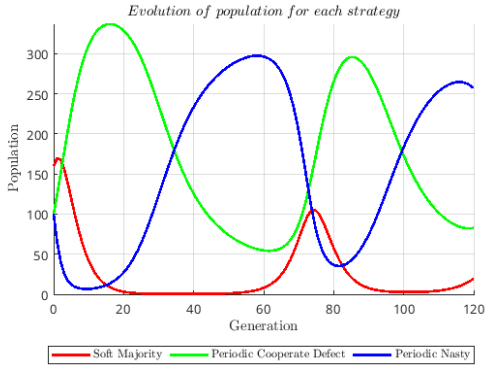


(a) Before

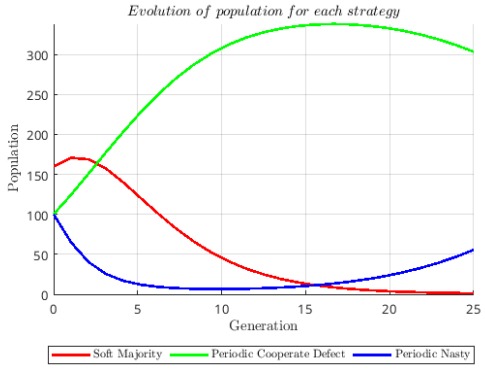


(b) After

Figure 2.39: Side-by-side comparison of population size sensitivity 2 before and after Pop method



(a) Before



(b) After

Figure 2.40: Side-by-side comparison of population size sensitivity 2 before and after Off method

2.7.9 Game length sensitivity

This experiment examines the case of changing the game's length or the number of rounds played by the players each generation. In the before State the peaks reached are higher in the paper compared to our example, however we manage to capture the change in converging speed and amplitude of the oscillations. The rounding methods that recreate the general shape of the plot are "dec" and "off". In the case of "pop" we notice periodic oscillations of the after state, which do not decrease in amplitude.

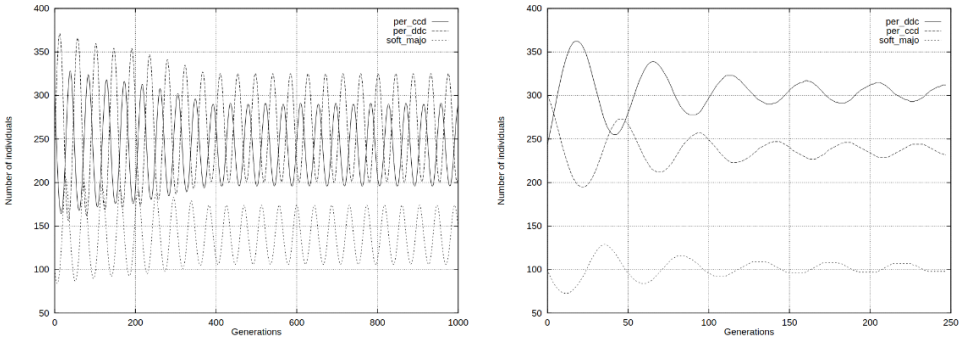
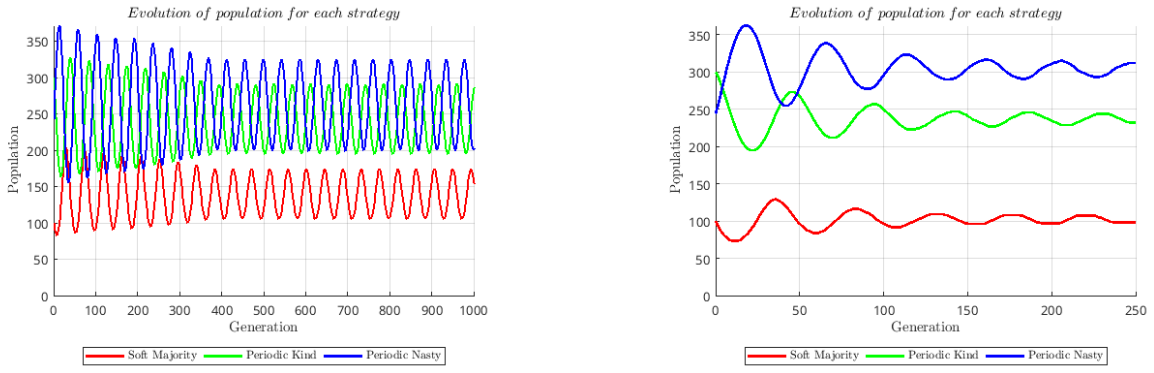


Figure 2.41: Game length sensitivity Original Plot

Strategy	**Population**
soft_majo	100
per_ccd	300
per_ddc	244
Notes:	
Rounds $= 6 \rightarrow 7$	
Generations $= 1000 \rightarrow 250$	
Matrix $= \begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

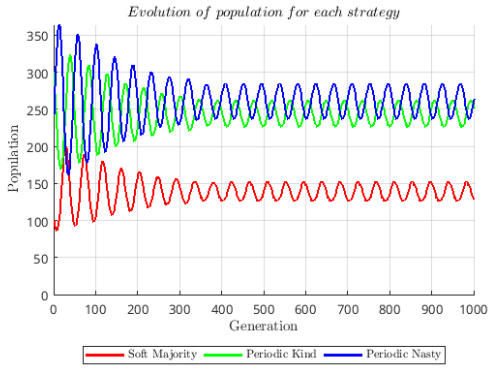
Table 2.9: Initial populations with additional notes



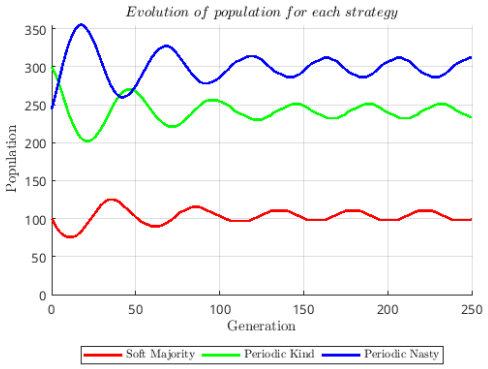
(a) Before

(b) After

Figure 2.42: Side-by-side comparison of game length sensitivity before and after Paper

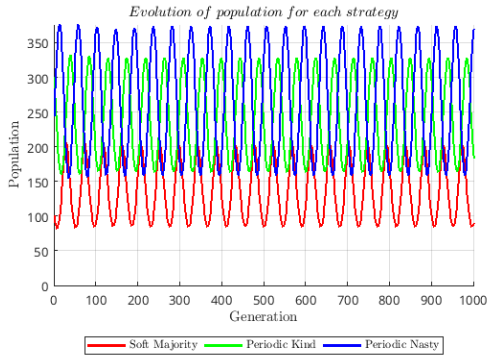


(a) Before



(b) After

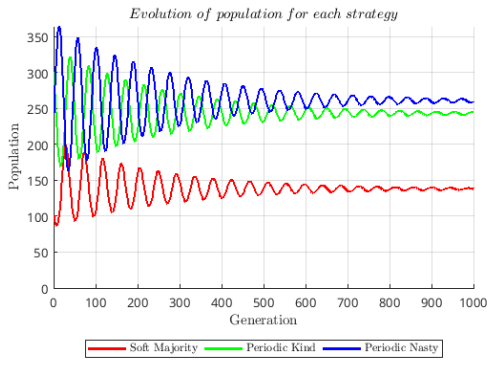
Figure 2.43: Side-by-side comparison of game length sensitivity before and after Dec



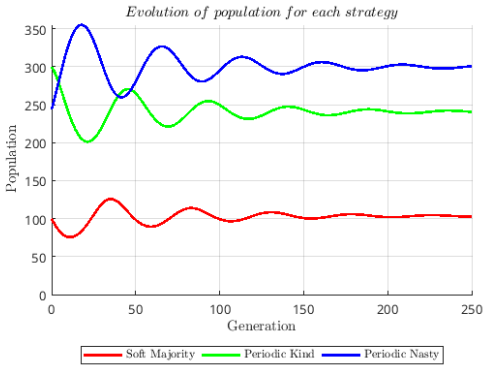
(a) Before

(b) After

Figure 2.44: Side-by-side comparison of game length sensitivity before and after Pop



(a) Before



(b) After

Figure 2.45: Side-by-side comparison of game length sensitivity before and after Off

2.7.10 Payoff matrix sensitivity

This meeting highlights the effects a slight change to one of the payoff matrix values can have on the results. In the before State we see increasing oscillations and after changing the defector's exploiting payoff these oscillations become periodic. We perfectly replicate the paper's results using all of our rounding methods. An interesting observation is that "pop" is reaching a final before State a lot quicker ($\approx$ 400 generations) than the other two methods ($\approx$ 1000 generations)

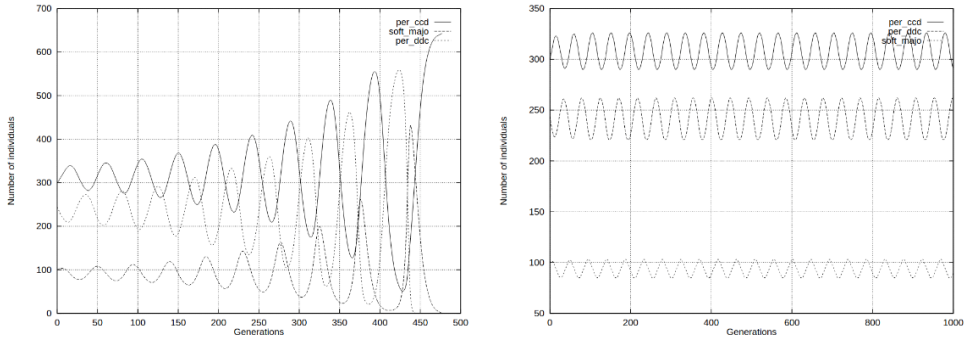
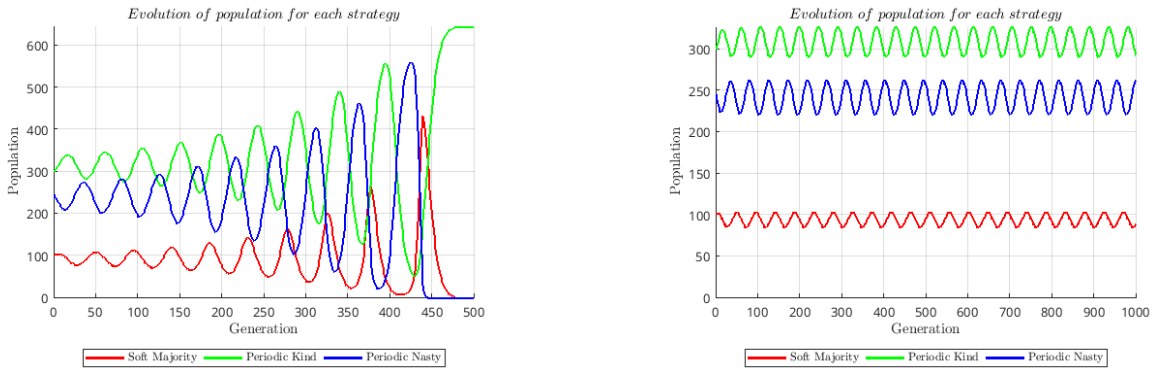


Figure 2.46: Payoff matrix sensitivity Original Plot

Strategy	**Population**
soft_majo	100
per_ccd	300
per_ddc	244
Notes:	
Rounds = 1000	
Generations = 500 $\rightarrow$ 1000	
Matrix = $\begin{bmatrix} 3 & 0 \\ 4.6 \rightarrow 4.7 & 1 \end{bmatrix}$	

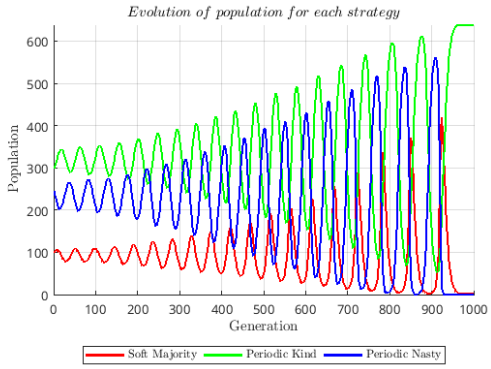
Table 2.10: Initial populations with additional notes



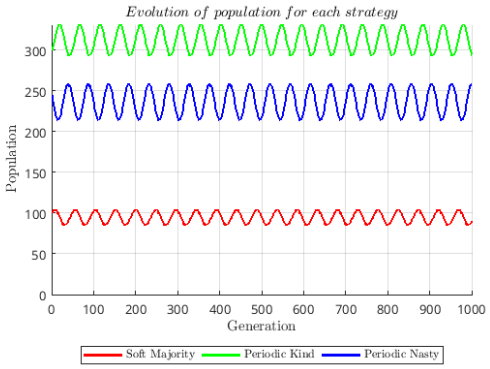
(a) Before

(b) After

Figure 2.47: Side-by-side comparison of payoff matrix sensitivity before and after Paper

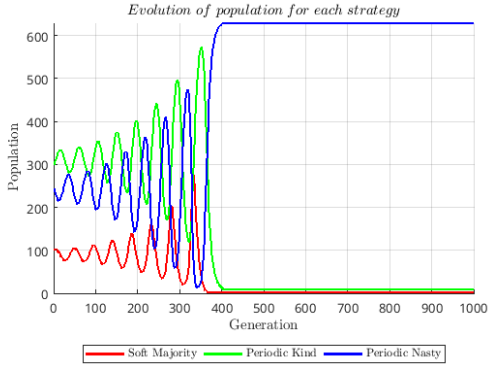


(a) Before

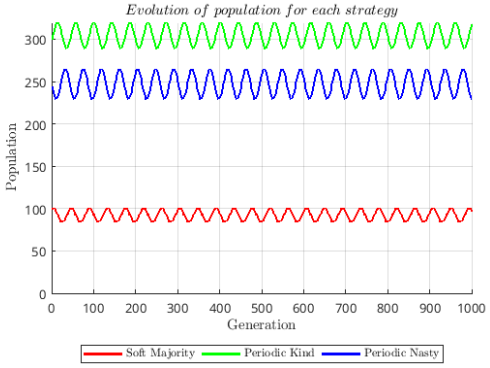


(b) After

Figure 2.48: Side-by-side comparison of payoff matrix sensitivity before and after Dec

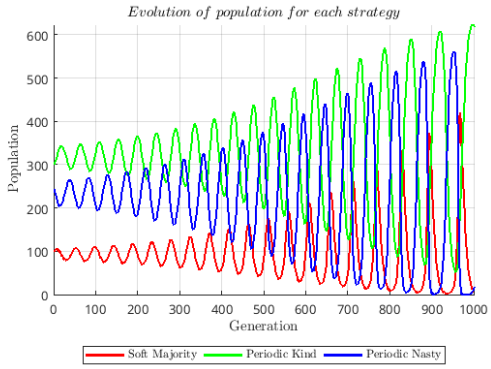


(a) Before

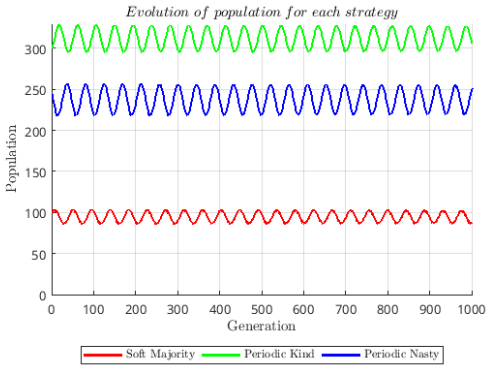


(b) After

Figure 2.49: Side-by-side comparison of payoff matrix sensitivity before and after Pop



(a) Before



(b) After

Figure 2.50: Side-by-side comparison of payoff matrix sensitivity before and after Off

2.7.11 Rounding method sensitivity

The final category of minor changes leading to different results is the rounding method. Though out the meetings analysis we have highlighted the importance of the rounding method since to replicate the paper's results we resorted to creating more than one methods, alternating between all of them to better illustrate the effects mentioned in the paper. On the first example we see periodic movements become attenuated oscillations and ultimately steady states. The rounding method used in the before state is "dec", "pop" and in the after state it is "off" or no rounding.

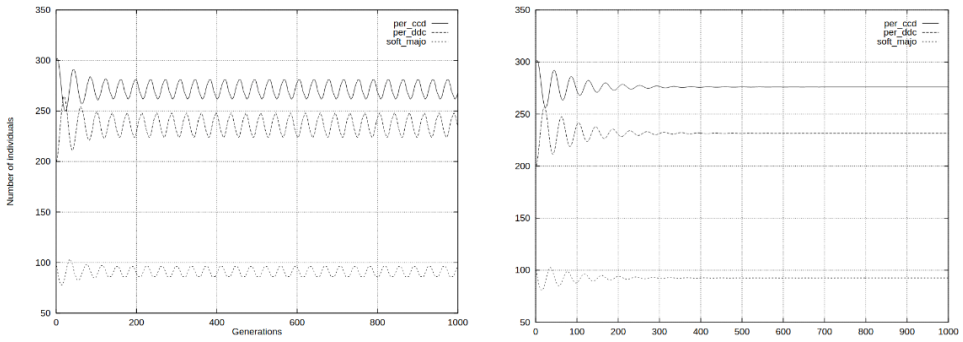
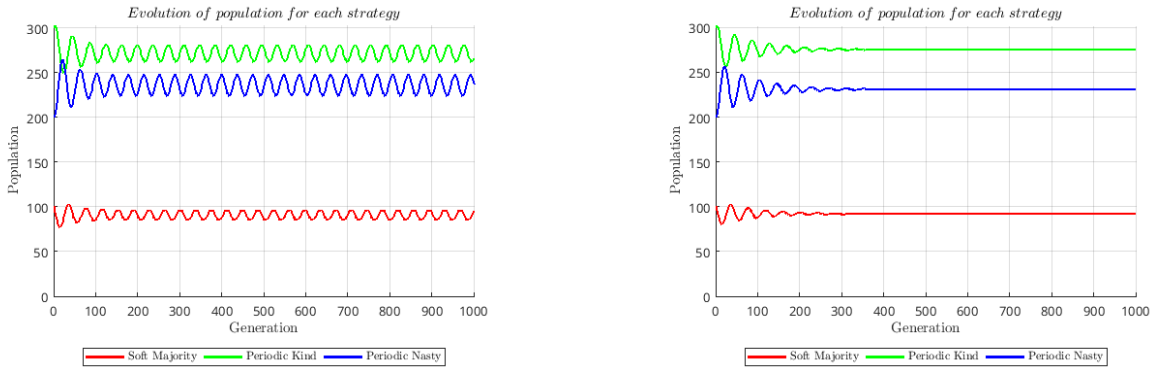


Figure 2.51: Rounding method sensitivity Original Plot

Strategy	**Population**
soft_majo	100
per_ccd	300
per_ddc	200
Notes:	
Rounds = 1000	
Generations = 1000	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

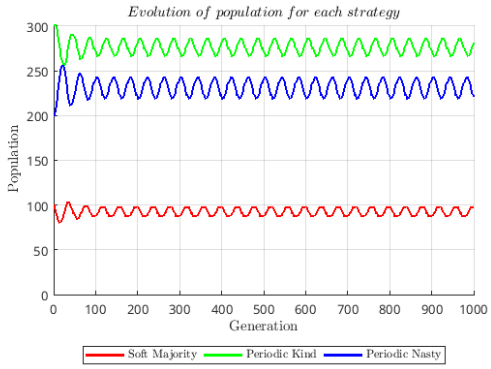
Table 2.11: Initial populations with additional notes



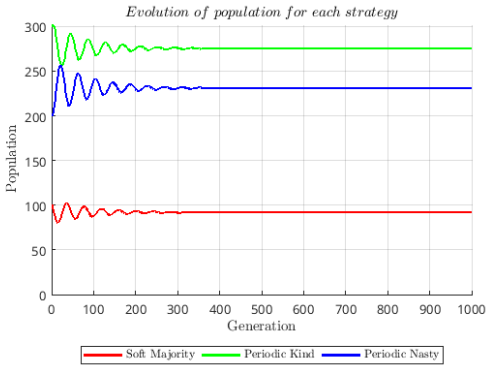
(a) Before

(b) After

Figure 2.52: Side-by-side comparison of rounding method sensitivity before (Paper) and after (Off)

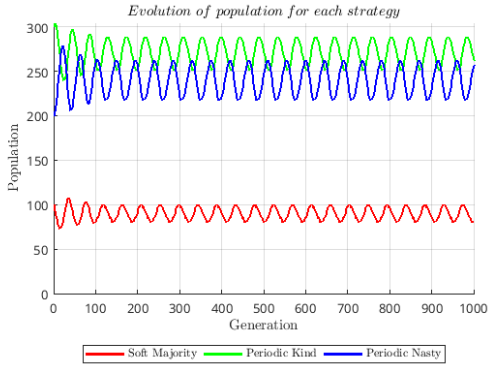


(a) Before

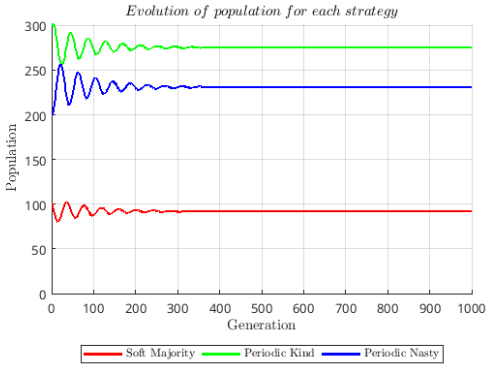


(b) After

Figure 2.53: Side-by-side comparison of rounding method sensitivity before (Dec) and after (Off)



(a) Before



(b) After

Figure 2.54: Side-by-side comparison of rounding method sensitivity before (Pop) and after (Off)

2.7.12 Rounding method sensitivity 2

The second example is best displayed using the "pop" method. Here we divide our initial populations by ten making rounding more important since we round an substantially larger section of the individual populations. The "off" and "dec" methods do not display the same behavior on the after state but rather they seem to be a micrography of the before state.

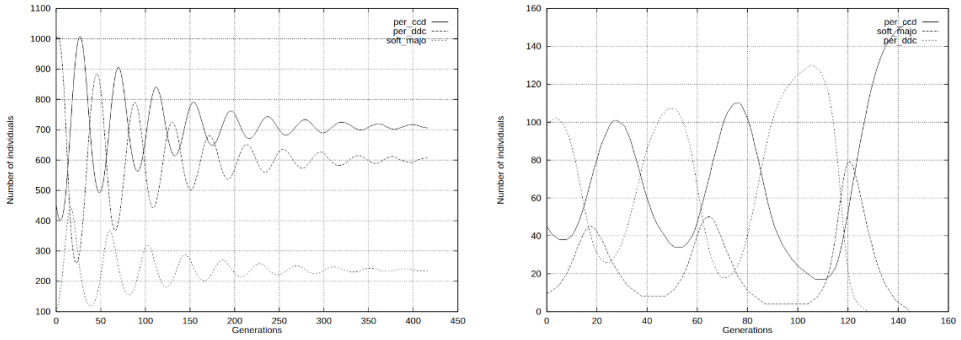
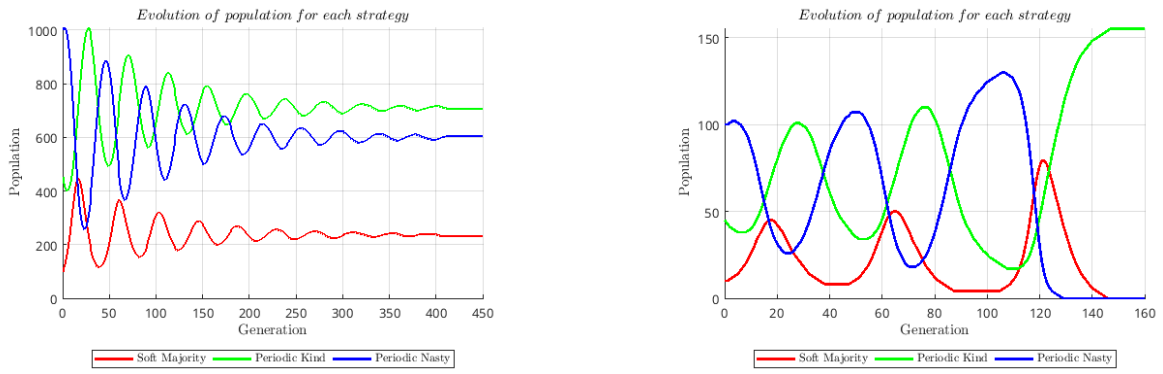


Figure 2.55: Rounding method sensitivity 2 Original Plot

Strategy	**Population**
soft_majo	$100 \rightarrow 10$
per_ccd	$450 \rightarrow 45$
per_ddc	$1000 \rightarrow 100$
Notes:	
Rounds = 1000	
Generations = $450 \rightarrow 160$	
Matrix = $\begin{bmatrix} 3 & 0 \\ 5 & 1 \end{bmatrix}$	

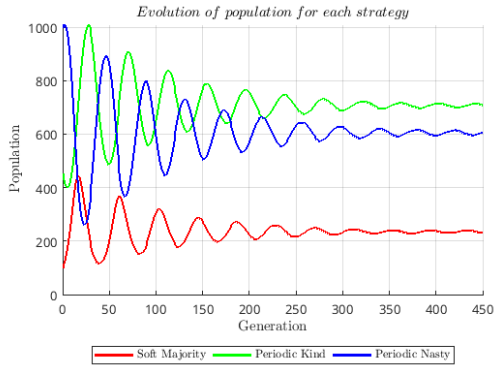
Table 2.12: Initial populations with additional notes



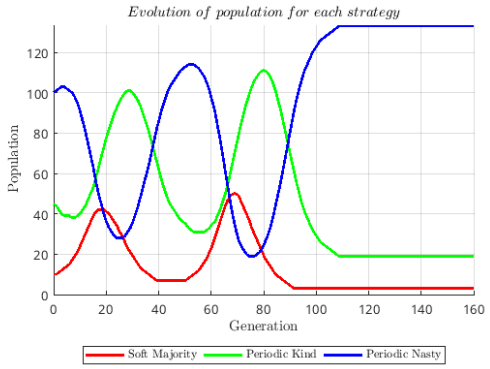
(a) Before

(b) After

Figure 2.56: Side-by-side comparison of rounding method sensitivity 2 before and after Paper

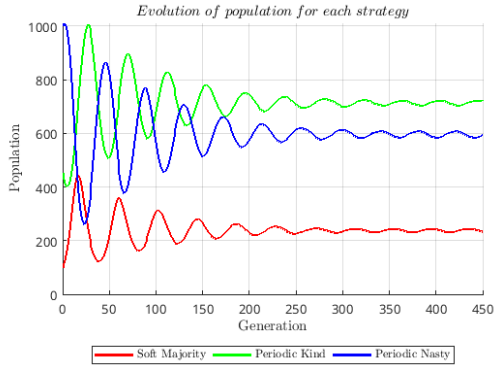


(a) Before

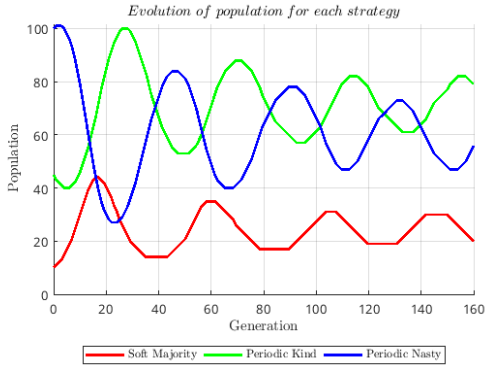


(b) After

Figure 2.57: Side-by-side comparison of rounding method sensitivity 2 before and after Pop

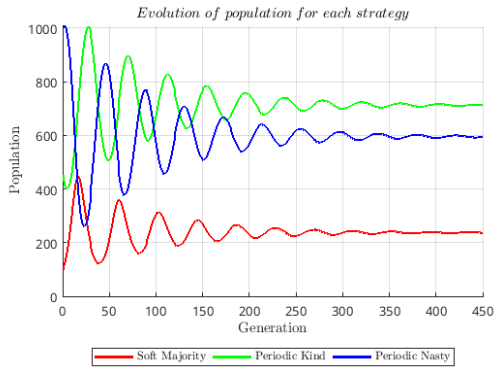


(a) Before

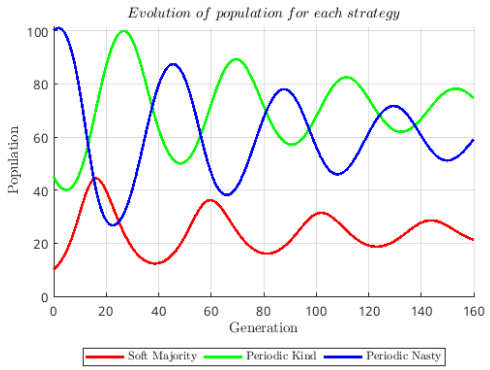


(b) After

Figure 2.58: Side-by-side comparison of rounding method sensitivity 2 before and after Dec



(a) Before



(b) After

Figure 2.59: Side-by-side comparison of rounding method sensitivity 2 before and after Off

Chapter 3

Imitation Dynamics

3.1 Imitation dynamics

In imitation dynamics, after a meeting among players adopting different strategies is concluded, a number of players adopt (imitate) one of the best performing strategies of the previous generation. The best performing strategies are those that received the highest payoff in the previous generation and we can have more than one of them. A strategy's payoff depends, not only on the specific strategies competing, but also on the proportion of players representing the strategy in a generation.

In what follows, we will restrict ourselves to three competing strategies with the total population of N players kept constant from generation to generation. We will also consider that after a meeting is concluded, a single player is chosen randomly, from the strategies that have non zero populations and that this player chooses randomly, with equal probability, to adopt one of the strategies that scored the most in that previous meeting.

The state of the system can be represented in this case by a triplet (n_1, n_2, n_3) of the players adopting each one of the three strategies, with $n_i \in \mathbb{Z},\ 0 \leq n_i \leq N,\ n_1+n_2+n_3 = N$. The evolution of the state through the different generations (extending the evolutionary game to run for an infinite number of steps), is a Markov chain. This Markov chain has in general a non deterministic state transition matrix M, due to the probabilistic adoption of one of the best strategies, by one player, at the end of a generation. In the sequel we will calculate theoretically and by simulation, this state transition matrix.

3.1.1 Theoretical approach

In order to calculate theoretically the state transition matrix M, one needs the population of each one of the three strategies at the beginning of a generation, the total payoff of each strategy during a meeting and the rules for transition from one state to the possible next states. For the transition rules, it suffices to know the strategies from which a player can leave, and the best scoring strategies in the meeting and then easily calculate the probabilities of transition for all the possibly adopted new strategies.

The total payoff of each strategy during a meeting equals the sum of the scores that individual players following that strategy can gain, during all two-by-two games in the

two-by-two round robin tournament. Consequently, it suffices to calculate the score of each player during a match between two players. This can of course be achieved by simulation, but also, for certain deterministic strategies this can be evaluated without simulating a match. For example, during an All-C vs All-D match, the players choose C and D, respectively at each round, and the All-C player's match score is SU, while the All-D player's match score is TU, with

	C	D
C	R R	S T
D	T S	P P

the payoff matrix and U the number of rounds per game. R is the Reward payoff, P the Punishment payoff, T the Temptation payoff and S the Sucker's payoff. $T > R > P > S$ guarantees that the Nash equilibrium is mutual defection, whereas $2R > T + S$ makes mutual cooperation the globally best outcome. However, this road of investigation cannot be extended to more involved strategies and certainly is not adequate for probabilistic strategies. Thereof, instead of calculating the exact score a player can gain during a match, either by simulation, or as a deterministic function of game histories in simple cases, we can instead use the expected average payoff, per round, of each player.

The expected average payoff per round can be calculated by using the fact that a match between two players can also be modeled as a finite-state Markov chain, where the system transits among the four states CC, CD, DC, DD. By adopting this method, we can extend the calculation of the state transition matrix even in the case of probabilistic strategies, like the zero determinant and extortionate strategies that will be presented hereafter.

3.1.1.1 Computation of expected average payoff per round, using Markov chains

In one round of the match, the outcome can be CC, CD, DC, or DD, where C = cooperation and D = defection. In each pair, the first letter represents the choice of the first player and the second the choice of the second player. In games where both players adopt strategies based on memory of the previous round (memory-one strategies), but also more generally in the case of finite-memory strategies, a match can be modeled as a finite-state Markov chain, where the system transitions between the four states CC, CD, DC, DD [5]. The conditional probabilities of any future event depend only on the current state and are independent of any past event.

Thus, the one-step transition matrix M has the form:

$$M = \begin{bmatrix} p(\text{CC}|\text{CC}) & p(\text{CC}|\text{CD}) & p(\text{CC}|\text{DC}) & p(\text{CC}|\text{DD}) \\ p(\text{CD}|\text{CC}) & p(\text{CD}|\text{CD}) & p(\text{CD}|\text{DC}) & p(\text{CD}|\text{DD}) \\ p(\text{DC}|\text{CC}) & p(\text{DC}|\text{CD}) & p(\text{DC}|\text{DC}) & p(\text{DC}|\text{DD}) \\ p(\text{DD}|\text{CC}) & p(\text{DD}|\text{CD}) & p(\text{DD}|\text{DC}) & p(\text{DD}|\text{DD}) \end{bmatrix}$$

For example, $p(\text{CD}|\text{DC}) = p(X_{k+1} = \text{CD}|X_k = \text{DC})$ is the probability that the next state is CD, given that the current state is DC (the first player chooses C and the second D, while their previous moves were D and C, respectively).

Assuming independence of the players' choices:

$$p(\text{CD}|\text{DC}) = p_X(C|\text{DC}) \cdot p_Y(D|\text{DC}) = p_X(C|\text{DC}) \cdot (1 - p_Y(C|\text{DC}))$$

where

$$p_X(C|\text{DC})$$

is the probability that the first player chooses C for his next move, when the previous state was DC, and

$$p_Y(D|\text{DC})$$

is the probability that the second player chooses D for his next move, when the previous state was DC. It is important to be able to calculate these conditional probabilities:

$$p_X(C|\text{CC}), p_X(C|\text{CD}), p_X(C|\text{DC}), p_X(C|\text{DD})$$

that is, the probabilities for a player to choose C given that the previous state was CC, or CD, or DC, or DD, which are dependent on the specific strategy that the player has adopted.

Let us define:

$$\mathbf{p} = (p_1, p_2, p_3, p_4) = (p_X(C|\text{CC}), p_X(C|\text{CD}), p_X(C|\text{DC}), p_X(C|\text{DD}))$$
$$\mathbf{q} = (q_1, q_2, q_3, q_4) = (p_Y(C|\text{CC}), p_Y(C|\text{CD}), p_Y(C|\text{DC}), p_Y(C|\text{DD}))$$

the strategy vectors for the first player, and the second player, respectively. The strategy vector $\mathbf{p}$ can be calculated for each strategy. For example:

- Random strategy: $\mathbf{p} = (0.5, 0.5, 0.5, 0.5)$
- All-C: $\mathbf{p} = (1, 1, 1, 1)$
- All-D: $\mathbf{p} = (0, 0, 0, 0)$
- Tit-For-Tat: $\mathbf{p} = (1, 0, 1, 0)$
- Generous Tit-For-Tat (GTFT): $\mathbf{p} = (1, q, 1, q)$, since, in comparison with Tit-for-Tat, when the opponent has played D, GTFT responds with D with a probability of $1 - q$
- Pavlov (Win State Lose Shift): $\mathbf{p} = (1, 0, 0, 1)$

Using these vectors, the one-step transition matrix M becomes:

$$M = \begin{bmatrix} p_1q_1 & p_1(1-q_1) & (1-p_1)q_1 & (1-p_1)(1-q_1) \\ p_2q_3 & p_2(1-q_3) & (1-p_2)q_3 & (1-p_2)(1-q_3) \\ p_3q_2 & p_3(1-q_2) & (1-p_3)q_2 & (1-p_3)(1-q_2) \\ p_4q_4 & p_4(1-q_4) & (1-p_4)q_4 & (1-p_4)(1-q_4) \end{bmatrix}$$

Each row of M must sum to 1. Let it be noted that in the second row, the system transits to a CD state (given the current state CC, CD, DC, DD) and thus the second player transits to a D state. For that reason, $q_3 = p_Y(C|\text{DC})$, which represents the probability that the second player chooses C, given the previous state was DC and not q_2 is used. Similar observations can be made about the third row.

It is also important to be able to calculate the unconditional (marginal) probabilities that the system will be driven in a CC, CD, DC or DD state, and how these are evolving through generations. Let v_k be the vector of unconditional probabilities at step k:

$$v_k = \big[p(X_k = \text{CC}), p(X_k = \text{CD}), p(X_k = \text{DC}), p(X_k = \text{DD})\big]$$

Then, using the law of total probability:

$$\begin{aligned} p(X_{k+1} = CD) &= p(X_{k+1} = CD \mid X_k = CC)\, p(X_k = CC) \\ &\quad + p(X_{k+1} = CD \mid X_k = CD)\, p(X_k = CD) \\ &\quad + p(X_{k+1} = CD \mid X_k = DC)\, p(X_k = DC) \\ &\quad + p(X_{k+1} = CD \mid X_k = DD)\, p(X_k = DD) \\ &= \big[p(CD \mid CC), p(CD \mid CD), p(CD \mid DC), p(CD \mid DD)\big] \cdot \begin{bmatrix} p(X_k = CC) \\ p(X_k = CD) \\ p(X_k = DC) \\ p(X_k = DD) \end{bmatrix} \end{aligned}$$

In conclusion, in the last equation where we used the simpler notation adopted in the transition matrix M, we have that $p(X_{k+1} = CD)$, which is an element of the vector v_{k+1}, equals the inner product of the second row of the one-step transition matrix M and the vector v_k. Therefore, overall, the temporal evolution of v_k follows $v_{k+1} = Mv_k$, and recursively $v_{k+1} = M^{k+1}v_0$, where v_0 is the vector of unconditional probabilities for the system to be in one of the states CC, CD, DC, DD at the step when the players make their initial moves. Thus, the matrix M^m is the matrix of conditional probabilities over m steps (the m-step transition matrix), and includes all the conditional probabilities for transitioning from any current state CC, CD, DC, DD to any other state CC, CD, DC, DD in m steps of the game (Chapman-Kolmogorov equations).

It is proven that when the Markov chain is irreducible and ergodic, the vector v_k of unconditional probabilities converges ($\lim_{k\to\infty} v_k = \pi$) to the stationary distribution π, which is also given by the convergent limit

$$\pi = \lim_{n\to\infty} \frac{1}{n} \sum_{k=0}^{n-1} v_0 M^k$$

When this limit exists, it is independent of v_0. Clearly, for π, it holds that $\pi M = \pi$ (obtained from the relation $v_{k+1} = Mv_k$ in the limit $k \to \infty$), and the sum of the elements of π is equal to 1, as it represents the unconditional probabilities, in the steady state, that the system is found in one of the states CC, CD, DC, DD.

Then, the elements of each column of the matrix M^m, for large m, approach the same value, since the conditional probability of transitioning to a state in m steps for large m becomes independent of the current state of the system. Knowing now, for a game between two opponents X and Y, the vector π with the long-term probabilities of the system being in any of the states CC, CD, DC, DD, it is clear that we can calculate the average expected pay-off per round for each of the two opponents, using the formulas:

$$s_X = \pi S_X = \pi \begin{bmatrix} R \\ S \\ T \\ P \end{bmatrix} \quad \text{and} \quad s_Y = \pi S_Y = \pi \begin{bmatrix} R \\ T \\ S \\ P \end{bmatrix}$$

respectively.

3.1.1.2 Theoretical calculation of the state transition matrix using Markov chains

In the sequel, we will consider a tournament between $M = 3$ strategies with an imitating behavior, let us call them A, B and C, with n_1, n_2 and n_3 being their respective population at a certain generation. During all generations, the total population remains constant at $N = n_1 + n_2 + n_3$. At each generation, we consider a two-by-two round robin tournament between all the players, where each player of strategy X plays a number of games with opponents Y of his strategy group, or with opponents from the other strategies, according to the table given hereafter:

XY	A	B	C
A	$\frac{n_1(n_1-1)}{2}$	$n_1 n_2$	$n_1 n_3$
B		$\frac{n_2(n_2-1)}{2}$	$n_2 n_3$
C			$\frac{n_3(n_3-1)}{2}$

From our previous analysis, considering the Markov chain of a certain game between two players, having as states all the possible outcomes of the game (CC, CD, DC, DD), we were able to calculate the mean expected payoff for each one of the two players. More precisely, having the strategy vectors

$$\mathbf{p} = (p_X(C|CC) \;\; p_X(C|CD) \;\; p_X(C|DC) \;\; p_X(C|DD))$$

and

$$\mathbf{q} = (p_Y(C|CC) \;\; p_Y(C|CD) \;\; p_Y(C|DC) \;\; p_Y(C|DD))$$

we can calculate the one-step transition matrix $M(\mathbf{p}, \mathbf{q})$, then the stationary distribution π from $\pi M = \pi$ and $\sum_i \pi(i) = 1$ and finally the expected average payoff per round for each one of the two players as

$$s_X = \pi S_X = \pi \begin{bmatrix} R \\ S \\ T \\ P \end{bmatrix}, s_Y = \pi S_Y = \pi \begin{bmatrix} R \\ T \\ S \\ P \end{bmatrix}$$

with

	C	D
C	R R	S T
D	T S	P P

the game payoff matrix. In a meeting (tournament) of 3 strategies, let us denote with

$$\mathbf{P} = [P^A_{AA}, \;\; P^B_{BB}, \;\; P^C_{CC}, \;\; P^A_{AB}, \;\; P^B_{AB}, \;\; P^A_{AC}, \;\; P^C_{AC}, \;\; P^B_{BC}, \;\; P^C_{BC}]$$

a vector comprising all the payoffs between two players. In $\mathbf{P}$, P^X_{XY} denotes the payoff for player X in a match between X and Y and P^Y_{XY} the payoff of player Y in the same

match. Obviously, considering all the matches in a meeting, if each match comprises U rounds, the total expected average payoff for each one of the three strategies is given by

$$\text{strategy A:}\quad SC_A = [n_1(n_1-1)P^A_{AA} + n_1 n_2 P^A_{AB} + n_1 n_3 P^A_{AC}]U$$

$$\text{strategy B:}\quad SC_B = [n_1 n_2 P^B_{AB} + n_2(n_2-1)P^B_{BB} + n_2 n_3 P^B_{BC}]U$$

$$\text{strategy C:}\quad SC_C = [n_1 n_3 P^C_{AC} + n_2 n_3 P^C_{BC} + n_3(n_3-1)P^C_{CC}]U$$

Note here that in a match between two players of the same strategy, the strategy payoff is $2P^X_{XX}$. We are now ready, for the specific choice of strategies A, B and C and for all combinations of populations $n_1,\ n_2,\ n_3$, such that $n_1+n_2+n_3 = N$, $n_i \in \mathbb{Z}$, $0 \le n_i \le N$, to calculate the total expected average payoffs for the three strategies ($SC_A,\ SC_B,\ SC_C$) and find which can be the next state. In general, being at a certain state ($n_1,\ n_2,\ n_3$), the next state can be one of the following: ($n_1,\ n_2,\ n_3$), or ($n_1+1,\ n_2-1,\ n_3$), or ($n_1+1,\ n_2,\ n_3-1$), or ($n_1-1,\ n_2+1,\ n_3$), or ($n_1,\ n_2+1,\ n_3-1$), or ($n_1-1,\ n_2,\ n_3+1$), or ($n_1,\ n_2-1,\ n_3+1$), as far as $n_1 \pm 1,\ n_2 \pm 1,\ n_3 \pm 1$ remain integers in the region $[0, N]$.

After ordering the three strategies in descending order, based on their scores SC_A, SC_B, SC_C in a meeting, and supposing that the strategies have n_1, n_2, n_3 players respectively in the current state, we have the following mutually independent cases for the movements of a player from a state to another one, together with their respective probabilities.

Relations between scores	Current state	Next state	Probability
Rule 00: SC_A, $SC_B = SC_C = 0$	$(n_1,\ 0,\ 0)$	$(n_1,\ 0,\ 0)$	1
Rule 01: $SC_A > SC_B$, $SC_C = 0$	$(n_1,\ n_2,\ 0)$	$(n_1+1,\ n_2-1,\ 0)$ $(n_1,\ n_2,\ n_3)$	$\frac{n_2}{n_1+n_2}$ $\frac{n_1}{n_1+n_2}$
Rule 02: $SC_A = SC_B$, $SC_C = 0$	$(n_1,\ n_2,\ 0)$	$(n_1,\ n_2,\ 0)$ $(n_1+1,\ n_2-1,\ 0)$ $(n_1-1,\ n_2+1,\ 0)$	$\frac{1}{2}$ $\frac{1}{2}\frac{n_2}{n_1+n_2}$ $\frac{1}{2}\frac{n_1}{n_1+n_2}$
Rule 03: $SC_A > SC_B > SC_C$	$(n_1,\ n_2,\ n_3)$	$(n_1,\ n_2,\ n_3)$ $(n_1+1,\ n_2-1,\ n_3)$ $(n_1+1,\ n_2,\ n_3-1)$	$\frac{n_1}{n_1+n_2+n_3}$ $\frac{n_2}{n_1+n_2+n_3}$ $\frac{n_3}{n_1+n_2+n_3}$
Rule 04: $SC_A = SC_B > SC_C$	$(n_1,\ n_2,\ n_3)$	$(n_1,\ n_2,\ n_3)$ $(n_1+1,\ n_2-1,\ n_3)$ $(n_1-1,\ n_2+1,\ n_3)$ $(n_1+1,\ n_2,\ n_3-1)$ $(n_1,\ n_2+1,\ n_3-1)$	$\frac{1}{2}\frac{n_1+n_2}{n_1+n_2+n_3}$ $\frac{1}{2}\frac{n_2}{n_1+n_2+n_3}$ $\frac{1}{2}\frac{n_1}{n_1+n_2+n_3}$ $\frac{1}{2}\frac{n_3}{n_1+n_2+n_3}$ $\frac{1}{2}\frac{n_3}{n_1+n_2+n_3}$
Rule 05: $SC_A > SC_B = SC_C$ (can be incorporated in rule 03)	$(n_1,\ n_2,\ n_3)$	$(n_1,\ n_2,\ n_3)$ $(n_1+1,\ n_2-1,\ n_3)$ $(n_1+1,\ n_2,\ n_3-1)$	$\frac{n_1}{n_1+n_2+n_3}$ $\frac{n_2}{n_1+n_2+n_3}$ $\frac{n_3}{n_1+n_2+n_3}$
Rule 06: $SC_A = SC_B = SC_C$	$(n_1,\ n_2,\ n_3)$	$(n_1,\ n_2,\ n_3)$ $(n_1+1,\ n_2-1,\ n_3)$ $(n_1+1,\ n_2,\ n_3-1)$ $(n_1-1,\ n_2+1,\ n_3)$ $(n_1,\ n_2+1,\ n_3-1)$ $(n_1-1,\ n_2,\ n_3+1)$ $(n_1,\ n_2-1,\ n_3+1)$	$\frac{1}{3}$ $\frac{1}{3}\frac{n_2}{n_1+n_2+n_3}$ $\frac{1}{3}\frac{n_3}{n_1+n_2+n_3}$ $\frac{1}{3}\frac{n_1}{n_1+n_2+n_3}$ $\frac{1}{3}\frac{n_3}{n_1+n_2+n_3}$ $\frac{1}{3}\frac{n_1}{n_1+n_2+n_3}$ $\frac{1}{3}\frac{n_2}{n_1+n_2+n_3}$

In the table above, we have supposed that a player is chosen randomly from the strategies which have non zero populations in the meeting and chooses randomly, with equal probability, to move to one of the strategies that scored the most in the meeting (more than one, if they draw).
For example, in the case $SC_A = SC_B = SC_C$ (Rule 06) the three strategies have equal total expected average payoffs, a player can be chosen randomly from anyone of the strategies and move to anyone of the strategies. More precisely, we can remain in the same state (n_1, n_2, n_3) if, either a player is chosen randomly from strategy A with probability $\frac{n_1}{n_1+n_2+n_3}$ and moves to the same strategy A with probability $\frac{1}{3}$, or a player is chosen from strategy B and remains to B with probability $\frac{1}{3}\frac{n_2}{1+n_2+n_3}$, or a player is chosen from strategy C and remains to C with probability $\frac{1}{3}\frac{n_3}{n_1+n_2+n_3}$, all the probabilities summing to $\frac{1}{3}$. In an analogous manner, we have calculated all the reported state transitions in the table with their respective probabilities. Note that Rule 05 can be incorporated in Rule 03.

Now that we can have the payoffs P^X_{XY}, P^Y_{XY} for all combinations of strategies in a meeting, we can, for each combination of possible strategy populations (integers $n_1,\ n_2,\ n_3$ such that $\sum_i n_i = N, n_i \in \mathbb{Z},\quad n_i \in [0, N]$), calculate the expected average payoffs $SC_A,\ SC_B,\ SC_C$ for each strategy and then apply the appropriate rule to obtain the next states to which the current state can transition, with the respective probabilities,

thus obtaining the state transition matrix. We can provide all possible combinations by changing n_1 from 0 to N, then for each n_1 changing n_2 from 0 to $N - n_1$ and obtaining for each combination of n_1, n_2 the respective n_3 from $n_3 = N - n_1 - n_2$. There is a total of $(N+1) + N + (N-1) + ... + 2 + 1 = \frac{((N+1)+1)(N+1)}{2} = \frac{(N+1)(N+2)}{2}$ combinations.

3.1.1.3 Zero determinant and extortionate strategies

For the three strategy meetings, we can choose between the well known strategies All-C, All-D, Random, Pavlov, Tit-for-Tat and Generous Tit-for-Tat. For these strategies we can easily calculate their strategy vector $\mathbf{p}$. Moreover, we can choose among some strategies which are called zero determinant strategies [8] [7] [6].

A player using such a strategy, can enforce a linear relation $\alpha s_X + \beta s_Y + \gamma = 0$, or $(s_X - P) = \chi(s_Y - P)$, $\chi \geq 1$ between his score s_X and his opponent's score s_Y. These strategies are based on the fact that it can be proven that, the inner product of a game's stationary distribution π and an arbitrary 4×1 vector $\mathbf{f} = [f_1, \ f_2, \ f_3, \ f_4]^T$ equals the determinant of a matrix:

$$\pi \cdot \mathbf{f} = \det\Big(\begin{bmatrix} -1+p_1q_1 & -1+p_1 & -1+q_1 & f_1 \\ p_2q_3 & -1+p_2 & q_3 & f_2 \\ p_3q_2 & p_3 & -1+q_2 & f_3 \\ p_4q_4 & p_4 & q_4 & f_4 \end{bmatrix} \Big) = \det[\overline{\mathbf{r}}(\mathbf{p},\mathbf{q}) \quad \tilde{\mathbf{p}}(\mathbf{p}) \quad \tilde{\mathbf{q}}(\mathbf{q}) \quad \mathbf{f}] \text{ with}$$

$\mathbf{p} = (p_1, \ p_2, \ p_3, \ p_4)$ and $\mathbf{q} = (q_1, \ q_2, \ q_3, \ q_4)$ the two strategy vectors.

Notice here that the second column $\tilde{\mathbf{p}}$ of the matrix is solely under the control of X while the third column $\tilde{\mathbf{q}}$ is solely under the control of Y, because they depend on $\mathbf{p}$ and $\mathbf{q}$, respectively, which X and Y, respectively, can choose at their will. Player X can choose $\mathbf{p}$ such that $\tilde{\mathbf{p}}$ becomes equal to $\mathbf{f}$, subsequently zeroing the determinant and obtaining $\pi \cdot \mathbf{f} = 0$ (hence the name zero determinant). By choosing $\mathbf{f} = \alpha S_X + \beta S_Y + \gamma \mathbf{1}$ with $\mathbf{1} = [1 \ \ 1 \ \ 1 \ \ 1]^T$, it can be proven that $\alpha s_X + \beta s_Y + \gamma = 0$, while by choosing $\mathbf{f} = (S_X - P\mathbf{1}) - \chi(S_Y - P\mathbf{1})$ it can be proven that $(s_X - P) = \chi(s_Y - P)$.

In the case $\alpha s_X + \beta s_Y + \gamma = 0$, one could obviously suppose that X could choose $\beta = 0$ and impose his score s_X to a desired value, but this is proven to be non feasible. On the contrary, it is absolutely feasible for X to choose $\mathbf{p}$ such that $\tilde{\mathbf{p}} = \mathbf{f} = \alpha S_X + \beta S_Y + \gamma \mathbf{1}$ and thus $\pi \cdot \mathbf{f} = 0$ and with the choice $\alpha = 0$, impose the score of Y to $s_Y - \frac{\gamma}{\beta}$, and this can be done independently of Y's strategy. In the same way, it is feasible for Y to choose $\mathbf{q}$ such that $\tilde{\mathbf{q}} = \mathbf{f} = \alpha S_X + \beta S_Y + \gamma \mathbf{1}$ and thus $\pi \cdot \mathbf{f} = 0$ and with the choice $\beta = 0$ impose the score of his opponent X to $s_X = -\frac{\gamma}{\alpha}$, independently of X's strategy.

We use for example in our MATLAB realization the strategies SET-2 which forces the opponent's payoff to be 2 regardless of what strategy the opponent uses, and SET-3. For SET-2, $\mathbf{p} = (p_X(C|CC) \ p_X(C|CD) \ p_X(C|DC) \ p_X(C|DD)) = (0.75 \ \ 0.25 \ \ 0.5 \ \ 0.25)$ and for SET-3 $\mathbf{p} = (1 \ \ 0.9 \ \ 0.15 \ \ 0.1)$. We also use the generous zero determinant strategy GEN-2 with $\mathbf{p} = (1 \ \ 0.5625 \ \ 0.5 \ \ 0.125)$.

Similarly, X can choose $\mathbf{p}$ such that $\tilde{\mathbf{p}} = \phi\mathbf{f} = \phi[(S_X - P\mathbf{1}) - \chi(S_Y - P\mathbf{1})]$, $\chi \geq 1$, $0 < \phi \leq \frac{P-S}{(P-S)+\chi(T-P)}$ and thus obtain $\pi \cdot \mathbf{f} = 0$ and impose an extortionate share of

payoffs $(s_X - P) = \chi(s_Y - P)$. We have used the extortionate strategies EXT-2 with $\chi = 2$ and $\mathbf{p} = (0.875 \;\; 0.4375 \;\; 0.375 \;\; 0)$ and EXT-5 with $\chi = 5$ and $\mathbf{p} = [0.68 \;\; 0.16 \;\; 0.36 \;\; 0]$. We note here that Tit-for-Tat results as an extortionate zero determinant strategy with the strategy vector $(1 \;\; 0 \;\; 1 \;\; 0)$, in the special case $\chi = 1$ and $\phi = \frac{1}{5}$, thus imposing $s_X = s_Y$, implying fairness.

3.1.1.4 MATLAB realization of the theoretical approach

Concerning now our MATLAB realization of the theoretical approach, in the `initialize()` function, we can choose the total population N, the payoff matrix $(R,\ S,\ T,\ P)$, the number of rounds per match, `roundsth` and the three strategies A, B, C to compete, with their respective strategy vectors $\mathbf{p}$ and the probability for each strategy to choose C as its first move. We also calculate all the different $\frac{(N+1)(N+2)}{2}$ states of the system. The main script to run is `MarkovRun.m`. Both `initialize()` and `MarkovRun.m` are also used in the simulation based approach to calculate the state transition matrix, as we will see hereafter.

`MarkovRun.m` calls `initialize()` and then calls the function `MarkovTheory()` to perform the theoretical calculations needed. To carry on the calculations in the case that the expected average payoff is used, `MarkovTheory()` calls the function `tournamentpayoffs()`, which returns the average expected payoff, for each one of the players in a match, for all the combinations of strategies (A vs A, B vs B, C vs C, A vs B, A vs C, B vs C). In order to calculate the payoffs, `tournamentpayoffs()` calls the function `payoff()`.

The function `payoff()` calculates the transition matrix M by calling the function `transitionMatrix()`, resolves the equations $\pi M = \pi$, $\sum_i \pi(i) = 1$ in order to obtain the stationary distribution vector π and then finds the average expected payoffs P_{XY}^X and P_{XY}^Y, as in section 3.1.1.1.

`MarkovTheory()` then calls function `transitionRules()`. In a double `for` loop all triplets (n_1, n_2, n_3) are created and for each one of them the function `strategyPayoffs()` is called to calculate the total average strategy payoffs $SC_A,\ SC_B$ and SC_C during a meeting. Both (n_1, n_2, n_3) and (SC_A, SC_B, SC_C) are used to examine which rule of section 3.1.1.2 is fired, which will then provide all the next states to which the transition can be performed from the current state (n_1, n_2, n_3), together with the respective transition probabilities.

The function `transitionRules()` returns the current states which are stored in the matrix `allcurrentstates`, their respective next states in the matrix `allnextstates`, and the respective probabilities in the matrix `allprobs`. Because from one current state we can transit to more than one next states, the elements of `allcurrentstates` are non unique.

To populate the matrices `allcurrentstates`, `allnextstates` and `allprobs`, the function `transitionRules()` uses function `remaintosamestate()` in the case the next state is the same as the current state and `transitionfromXtoY()` in the case a player is changing strategy.

Finally function `theoryGraphs()` is called by `MarkovRun.m`, where we find all the unique elements of `allcurrentstates`, in the matrix `allcurrentuniquestates`, which subsequently also comprises the unique names of all the states. We can then build the theoretical one-step transition matrix by finding to which index of the matrix `allcurrentuniquestates`, corresponds the current state `allcurrentstates(i)` and its respective next state, `allnextstates(i)`.

By using MATLAB's `dtmc`, we construct a Markov chain model and then plot the heatmaps of the one-step transition matrix M, the eight step transition matrix M^8, the k-step transition matrix M^k for a certain big k (for example $k = 100$) and a movie of how the heatmap of M^k changes as k changes from 0 to k and finally a digraph using MATLAB's `graphplot()`. Our function `graphPlot()` is also called in order to obtain the state transition graph in a different format.

3.1.1.5 Analysis and MATLAB realization in predetermined outcome meetings

Often, the scores players gain in a match can be evaluated as a deterministic function of the game history. In these cases we could use the exact results, instead of the expected average payoff per round values, moreover without any need for simulation of the game. For example in a match of U rounds between an All-D player and a Trigger player the game history is

$$\begin{array}{lcccccc} All-D: & D & D & D & D & ... & D \\ Trigger: & C & D & D & D & ... & D \end{array}$$

Then the match scores are $\begin{bmatrix} T \\ S \end{bmatrix} + (U-1)\begin{bmatrix} P \\ P \end{bmatrix}$, i.e. the score of the All-D player is $T + (U-1)P$, while the score of the Trigger player is $S + (U-1)P$. We have considered the cases of All-C vs All-D vs Trigger and CCD vs DDC vs Trigger meetings, as examples.

In the case of All-C vs All-D vs Trigger meetings, one can easilly obtain the following game scores:

All-C vs All-C: $\begin{bmatrix} RU \\ RU \end{bmatrix}$

All-D vs All-D: $\begin{bmatrix} PU \\ PU \end{bmatrix}$

Trigger vs Trigger: $\begin{bmatrix} RU \\ RU \end{bmatrix}$

All-C vs All-D: $\begin{bmatrix} SU \\ TU \end{bmatrix}$

All-C vs Trigger: $\begin{bmatrix} RU \\ RU \end{bmatrix}$

All-D vs Trigger: $\begin{bmatrix} T + (U-1)P \\ S + (U-1)P \end{bmatrix}$

In the more involved case CCD vs DDC vs Trigger, one can verify the following score

values:

$$\text{CCD vs CCD:} \quad \begin{bmatrix}(U\texttt{div}3)(2R+P)+(U\texttt{mod}3)R\\(U\texttt{div}3)(2R+P)+(U\texttt{mod}3)R\end{bmatrix}$$

with `div` being the quotient and `mod` the modulus (remainder) of a division. In an analogous way we get:

$$\text{DDC vs DDC:} \quad \begin{bmatrix}(U\texttt{div}3)(2P+R)+(U\texttt{mod}3)P\\(U\texttt{div}3)(2P+R)+(U\texttt{mod}3)P\end{bmatrix}$$

$$\text{Trigger vs Trigger:} \quad \begin{bmatrix}RU\\RU\end{bmatrix}$$

$$\text{CCD vs DDC:} \quad \begin{bmatrix}(U\texttt{div}3)(2S+T)+(U\texttt{mod}3)S\\(U\texttt{div}3)(2T+S)+(U\texttt{mod}3)T\end{bmatrix}$$

$$\text{CCD vs Trigger:} \quad \text{for } U=1: \begin{bmatrix}R\\R\end{bmatrix}$$

$$\text{for } U=2: \begin{bmatrix}2R\\2R\end{bmatrix}$$

$$\text{for } U=3: \begin{bmatrix}2R+T\\2R+S\end{bmatrix}$$

$$\text{for } U\geq 4: \begin{bmatrix}2R+T+[(U-3)\texttt{div}3](2S+T)+[(U-3)\texttt{mod}3]S\\2R+S+[(U-3)\texttt{div}3](2T+S)+[(U-3)\texttt{mod}3]T\end{bmatrix}$$

$$\text{DDC vs Trigger:} \quad \text{for } U=1: \begin{bmatrix}T\\S\end{bmatrix}$$

$$\text{for } U=2: \begin{bmatrix}T+P\\S+P\end{bmatrix}$$

$$\text{for } U=3: \begin{bmatrix}T+S+P\\T+S+P\end{bmatrix}$$

$$\text{for } U\geq 4: \begin{bmatrix}T+S+P+[(U-1)\texttt{div}3](2P+S)+[(U-1)\texttt{mod}3]P\\T+S+P+[(U-1)\texttt{div}3](2P+T)+[(U-1)\texttt{mod}3]P\end{bmatrix}$$

In the MATLAB realization, all the above calculations are performed by the function `deterministicPayoffs()`. More precisely, in the case a predetermined outcome meeting is chosen, the function `MarkovTheory()`, calls `deterministicPayoffs()` to do the exact calculations, instead of calling `tournamentpayoffs()` which is based on the expected average payoff per round.

3.1.2 Simulation based approach to the state transition matrix

The idea behind the simulation based approach to calculate the state transition matrix, is to simulate, starting from some initial state, a number of meetings between the competing strategies (generations in evolutionary terms), until an equilibrium state is reached

and repeat this process starting from any possible state as initial state, many times. If the number of the state transitions from generation to generation are recorded (one-step transitions) the one step state transition matrix can be built by calculating the relative frequency of these transitions. More precisely, the many repetitions, starting from the same state, lead to the construction of a certain row of the transition matrix (a normalization to obtain probabilities is required at the end). If one records the initial state and the equilibrium state reached after many generations, the k-step (k big) state transition matrix can be constructed in an analogous manner.

We simulated in MATLAB the evolutionary imitation dynamics. In function `initialize()` one can set up the number of rounds per match between two opponents (variable `roundssim`), the number of generations (meeetings) simulated until an equilibrium state is reached (variable `ngens`) and the number of repetitions starting from the same state (variable `numofreps`). The function `initialize()` is called by the main script `MarkovRun.m` and after the theoretical calculations described in section 3.1.1.2, or 3.1.1.5 finish, the user can choose to continue with the simulation approach.

At the lowest level, i.e. in function `randomplay()`, and in order to be able to also simulate zero determinant strategies, for a strategy in a game described by its strategy vector $\mathbf{p}$, we construct a $4 \times roundssim$ matrix `Cprobs`, where each row is a random vector of `roundssim` elements, '0's for C and '1's for D. In each one of the four row vectors, C is present with a probability $p(C|CC)$, $p(C|CD)$, $p(C|DC)$ or $p(C|DD)$, respectively. During a match between two players, depending on the outcome of the system's previous state (CC, CD, DC, or DD) a player can choose his response in the current round, by using an element of the appropriate row of this premade matrix. We can guarantee (up to rounding errors[1]) that the desired probabilities are exact when considering all the elements of the row and approximately exact when less elements of the row are used in a match. By simulation, we have concluded that this strategy, with the premade choices, works slightly better than using a random generator each time at the level of a round in a match, in order to produce C or D with a specified probability. MATLAB's function `randperm()` for random permutations of integers was used to create the premade choice matrices with the desired probabilities for the appearance of the choice C. For deterministic strategies like All-C, All-D, Pavlov, Tit-for-Tat which have only '0's or '1's in their strategy vectors $\mathbf{p}$, the results obtained are exact.

The function `playmatch()` is used to simulate a match of `roundssim` rounds between two players, following strategies with vectors $\mathbf{p}$ and $\mathbf{q}$, respectively, and with their respective premade choice matrices `CprobsX`, `CprobsY`. For zero determinant strategies, the first move of the player is randomly chosen between C or D with probability 0.5 for each. The function `playmatch()` also calculates the score of each player in a match, based on the game playoff matrix. The premade choice matrices are randomly recalculated for each different match between two players.

The function `gentournament()` realizes a meeting between three different strategies (a generation in evolutionary terms), where all matches between all possible pairs of players are played. For the number of matches between players belonging to two strategies, see the matrix in the beginning of section 3.1.1.2. By summing, it can be easily seen that

[1]For example, in a vector of four elements, we cannot achieve a probability of 70% for C.

there are $\frac{N(N-1)}{2}$ matches in a meeting, N being the total constant population. After a match is finished, the total score of each strategy is updated according to the outcome of the match, also in function `gentournament()`. Finally, the same function, given the current state and the meeting scores for each one of the three strategies, calls the function `nxt()`, which calculates the next state of the system. After a meeting, one player is chosen randomly from the strategies with a non zero number of players and chooses, by imitation, to follow one of the strategies that scored the most in the meeting. The details of changing strategy for one player after a meeting, were also described in section 3.1.1.2 and are realized in `nxt()`. Either a player moves from a strategy to another one (realized by the function `transferfromXtoY()` in `nxt()`), or we remain to the same state (realized by `remaintosamestate()`) The function `nxt()` has its own realizations of the functions `remaintosamestate()` and `transferfromXtoY()`.

In both these functions, `remaintosamestate()` and `transferfromXtoY()`, when a transition from the current state to a next state is realized, we increment by one the appropriate (row, column) element of the one-step transition matrix (depending on the transition). The rows of the matrix are at the end normalized to probabilities to transition, starting from one certain state, to the other states, in the script `MarkovRun.m`. The matrix is declared as `global Msim`.

The evolution from one generation to the next is realized in `multigen()`, which calls `gentournament()` `ngens` times in order to obtain the final equilibrium states, useful for the calculation of the k-step (k big) transition matrix `Mk`.

The function `multigen()` which simulates the evolution from generation to generation, is called `numofreps` times in the function `multimultigen()`, with the help of which we simulate many generation evolutions, starting each time from the same initial state. The k-step (k big) state transition matrix is constructed in function `multimultigen()`, by considering the initial and final states each time.

In the script `MarkovRun.m`, in a double `for` loop, we create all the possible states, given a population of N players ($\frac{(N+1)(N+2)}{2}$ states) and call `multimultigen()` for each different initial state to create each time a specific row of the k-step transition matrix. After the end of the double `for` loop, in `MarkovRun.m`, we call function `simGraphs()` to prepare the visual outcomes of the simulation. In `simGraphs()`, by using MATLAB's `dtcm()` we create Markov chain models for the one-step and k-step transition matrices and plot their digraphs with `graphplot()`. We also create heatmaps for the simulated transition matrices `Msim`, `Mk` and for their differences, $|\texttt{Msim} - \texttt{MTheory}|$, $|\texttt{Mk} - \texttt{MTheory}^{\texttt{ngens}}|$ from their respective theoretically calculated ones, `MTheory` and $\texttt{MTheory}^{\texttt{ngens}}$. We used as a metric of the difference between the two matrices, the sum of all the absolute values of the differences between all their respective elements. `simGraphs()` also calls `graphPlot()`, as in the theoretical part, to produce the state transition graph.

3.1.3 Experiments

In order to compare the theoretical and the simulation based results, we considered an experiment with the following characteristics: the three strategies to compete were GEN-2, SET-3 and TFT, with a total of $N = 6$ players, 1000 rounds per match for the

theoretical calculations, 100 rounds per match for the simulation calculations, 10 repetitions per each initial state, 60 generations per repetition, with a payoff matrix having $R = 3,\ T = 5,\ S = 0,\ P = 1$. With $\frac{N(N-1)}{2} = 15$ matches per meeting, a total of $100 \times 10 \times 60 \times 15 = 9 \cdot 10^5$ rounds are simulated, requiring approximately $2.05 \frac{msec}{round}$ or a total of 1842 sec of computer time with the hardware used.

In Figure 3.1 we report the theoretically calculated one-step transition matrix `MTheory`. In Figure 3.2 the heatmap of the simulated one-step transition matrix `Msim` and in Figure 3.3 the heatmap of the absolute value of their difference. Alike, in Figure 3.4 we give the heatmap of the theoretical k-step (k big, here `ngens`) transition matrix $\texttt{MTheory}^{\texttt{ngens}}$, in Figure 3.5 the heatmap of the calculated by simulation k-step (k big) transition matrix `Mk` and in Figure 3.6 the heatmap of the absolute value of their difference. In Figures 3.7 to 3.9 we report the digraphs of the theoretically calculated `MTheory` and of the calculated by simulation `Msim` and `Mk`, respectively. In Figure 3.10 we report the values of the metric per row of the matrices, for the one-step (blue color) and for the k-step (red color) simulated transition matrices. The metric was equal to 3.09. The state transition graphs are given in Figure 3.11 for the theoretical part and in Figure 3.12 for the simulated part. They are alike.

It is possible to plot the time evolution, through the different generations, of the population of each strategy. An example is given in Figure 3.13. In this purpose, during the simulation, a matrix named `allstatesrepgenNumOfPlayersHistory` is constructed in the functions `multigen()`, `multimultigen()` and the script `MarkovRun.m`. This matrix holds all the history of the population evolution through the generations for all repetitions, starting from any initial state and can be found in MATLAB's workspace, after the calculations finish. The function `populationAnimation()` is given, which can be executed in order to get an animation of the population evolution. For example, by executing `populationAnimation(allstatesrepgenNumOfPlayersHistory(4:6,:,20),chosen,names)` we plot the state evolution in time for the second repetition (rows 4 to 6) starting from state 20 (state (3,1,2)).

From Figures 3.4 and 3.5, we see that all the states after a number of generations converge, either to state 1 (state (0,0,6)), or to state 7 (state (0,6,0)), or to state 28 (state (6,0,0)). These are the equilibrium absorbing states, where the winner is TFT, or SET-3, or GEN-2, respectively. Most states tend to these absorbing states with probability 1, except for states 16, 4, 19 and 22. We find for example that state 16 (state (2,2,2)) converges to state 1, or 7, or 28, with equal probability $\frac{1}{3}$ as calculated theoretically, and with probabilities 0.3, 0.3 and 0.4, respectively, as calculated by simulation.

The differences between theory and simulation are given in the next table and are due to the limited number of repetitions (10) per initial state.

Initial StateFinal State	State 1 (0, 0, 6)	State 7 (0, 6, 0)	State 28 (6, 0, 0)	
State 16 (2, 2, 2)	1/3 0.3	1/3 0.3	1/3 0.4	Theory Simulation
State 4 (0, 3, 3)	0.5 0.6	0.5 0.4		Theory Simulation
State 19 (3, 0, 3)	0.5 0.8		0.5 0.2	Theory Simulation
State 22 (3, 3, 0)		0.5 0.5	0.5 0.5	Theory Simulation

The simulations become quite time demanding, when the total population augments. Indicative simulation times (for the same strategies) are given in the table hereafter, for the case $N = 9$.

Rounds per match	Generations	Repetitions per state	Simulation time	Metric
50	40	3	15 min	16
100	45	10	1 hrs 50 min	7.5
500	50	15	15 hrs	7.14

We report here results from the simulation of the last case given in the table above. Since in the case $N = 9$ there are 36 matches per meeting (generation), this gave an average of 4.1 msec per round of execution time in the hardware used. A simulation with 1000 rounds/match 50 generations and 100 repetitions per state would demand almost 210 hours ($\sim$ 9 days) of simulation time.

We report the heatmap of the simulated (`Msim` - Figure 3.14) and theoretically calculated (`MTheory` - Figure 3.15) one-step transition matrices and of the absolute value of their difference in Figure 3.16, which appears to be minimal. The same holds for the heatmaps of the simulated k-step (k big) transition matrix `Mk` (Figure 3.17), the respective theoretical one $\texttt{MTheory}^{\texttt{ngens}}$ (Figure 3.18) and the absolute value of their difference (Figure 3.19). The 8-step transition matrix $\texttt{MTheory}^{8}$ (probabilities of each state to transition to other states in 8 steps), is given in Figure 3.20. In Figure 3.21, we report the values of the metric per row of the matrices, for the simulated one-step `Msim` (blue color) and the k-step `Mk` (red color) transition matrix. The `MTheory`, `Msim` and `Mk` digraphs are given in Figures 3.22, 3.23 and 3.24, respectively. In Figure 3.25 we give the state transition graph based on `MTheory`.

In Figures 3.26 and 3.27, we give the digraphs for the one-step `Msim` and k-step `Mk` matrices, where in this case the competing strategies are All-C, All-D and Tit-for-Tat, with $N = 9$ players in total, using 50 rounds per match, 3 repetitions per initial state and 50 generations. Their respective state transition graph based on `MTheory` is given in Figure 3.28.

3.1.4 Graphs

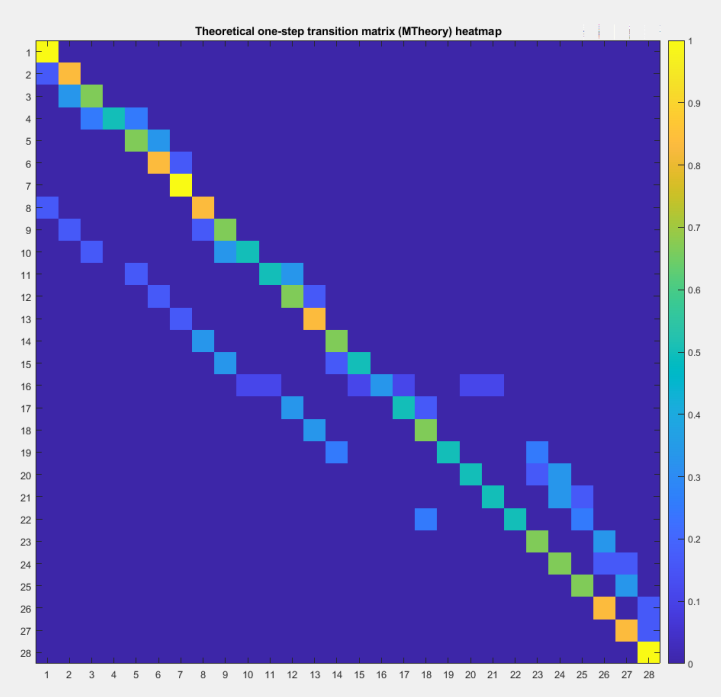


Figure 3.1: $N = 6$, GEN-2 vs SET-3 vs TFT. Heatmap of the theoretically calculated one-step transition matrix `MTheory`.

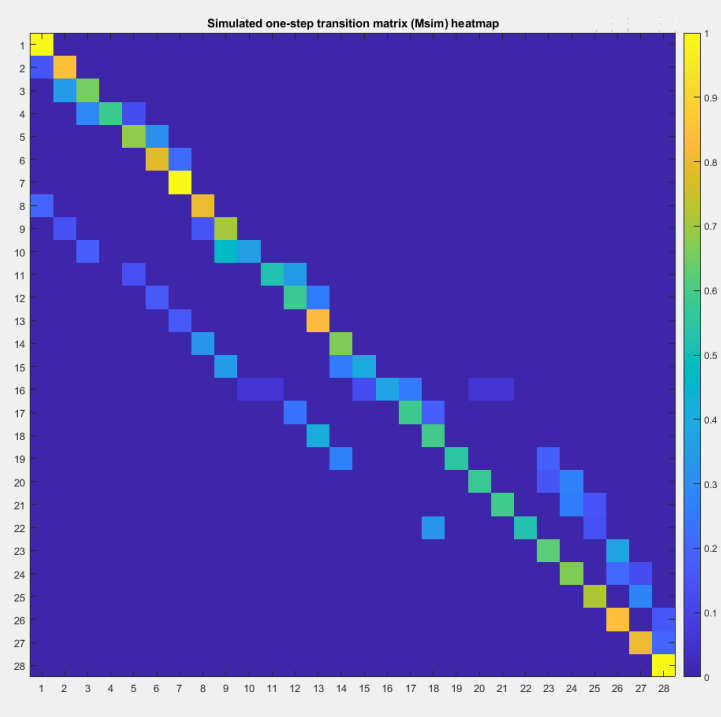


Figure 3.2: $N = 6$, GEN-2 vs SET-3 vs TFT. Heatmap of the simulated one-step transition matrix `Msim`.

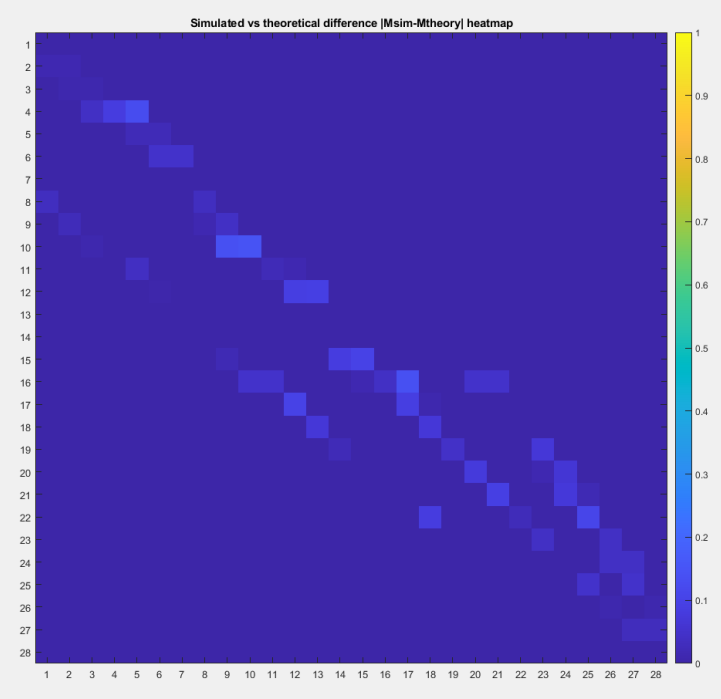


Figure 3.3: $N = 6$, GEN-2 vs SET-3 vs TFT. Heatmap of $|\texttt{Msim} - \texttt{MTheory}|$.

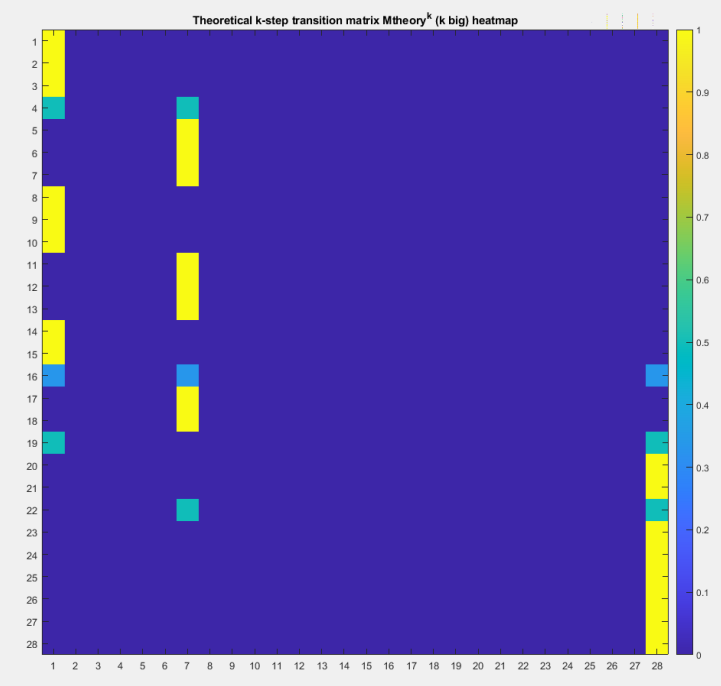


Figure 3.4: $N = 6$, GEN-2 vs SET-3 vs TFT. Heatmap of the theoretically calculated k-step (k big) transition matrix $\texttt{MTheory}^{\texttt{ngens}}$, $\texttt{ngens}= 60$.

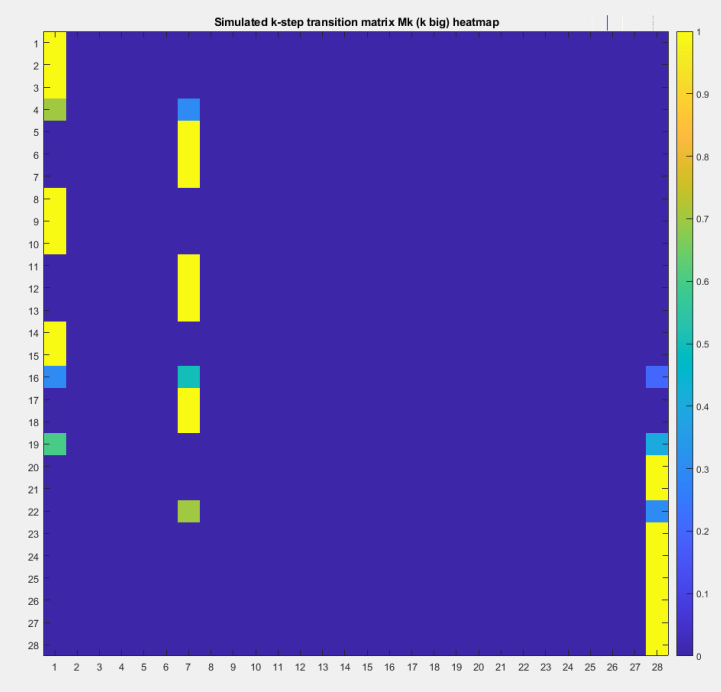


Figure 3.5: $N = 6$, GEN-2 vs SET-3 vs TFT. Heatmap of the simulated k-step (k big) transition matrix `Mk`.

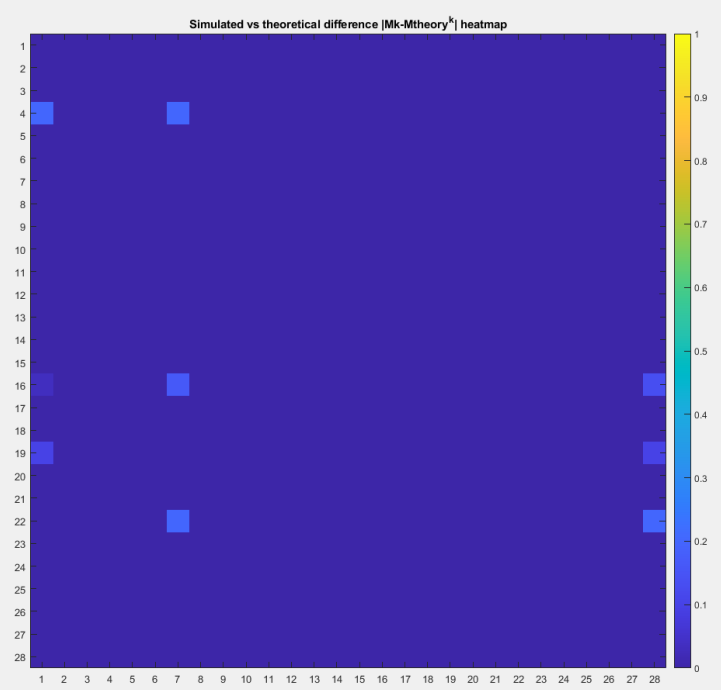


Figure 3.6: $N = 6$, GEN-2 vs SET-3 vs TFT. Heatmap of $|\texttt{Mk} - \texttt{MTheory}^{60}|$.

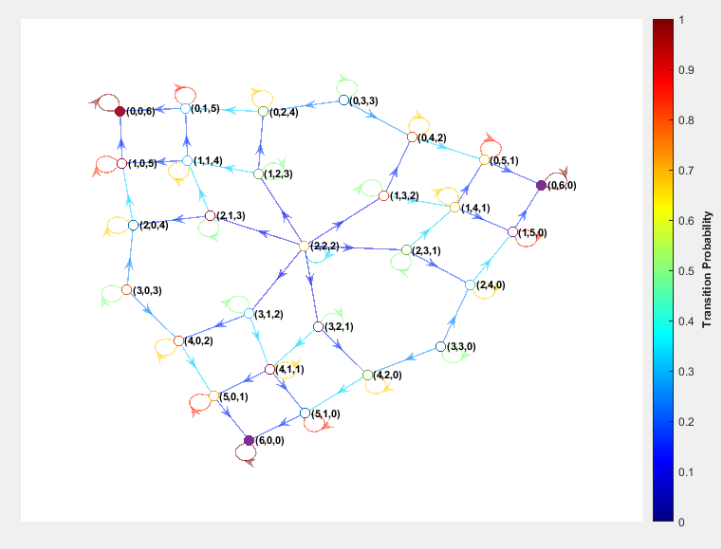


Figure 3.7: $N = 6$, GEN-2 vs SET-3 vs TFT. Digraph of the theoretically calculated one-step transition matrix `MTheory`.

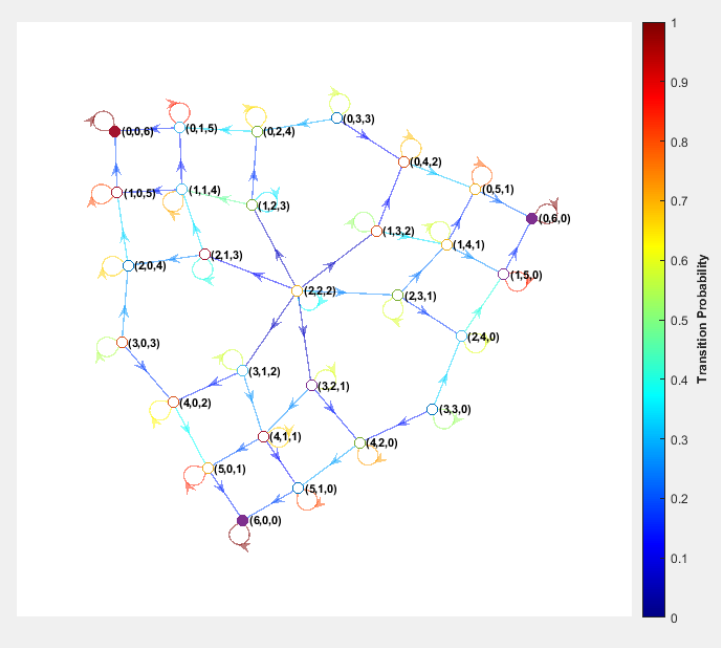


Figure 3.8: $N = 6$, GEN-2 vs SET-3 vs TFT. Digraph of the simulated one-step transition matrix `Msim`.

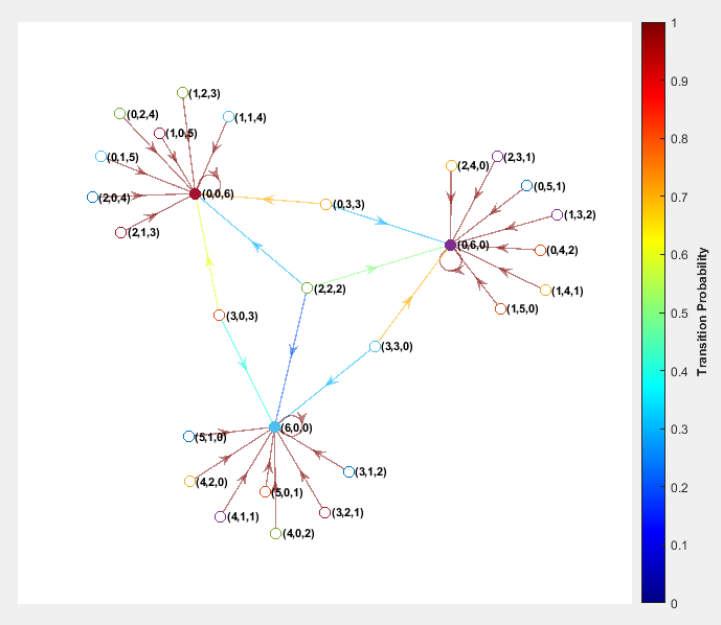


Figure 3.9: $N = 6$, GEN-2 vs SET-3 vs TFT. Digraph of the simulated k-step (k big) transition matrix `Mk`.

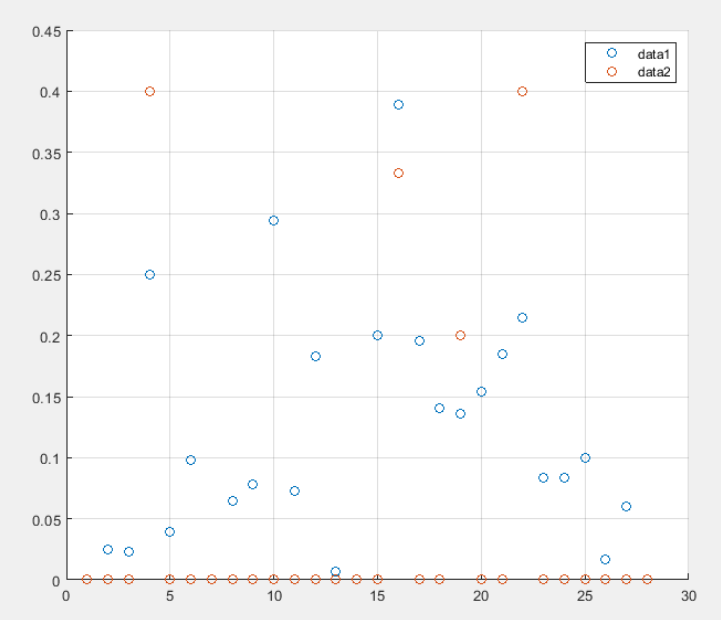


Figure 3.10: $N = 6$, GEN-2 vs SET-3 vs TFT. Scatter plot of the metric per row for the simulated one-step `Msim` (blue) and k-step `Mk` (red) transition matrices.

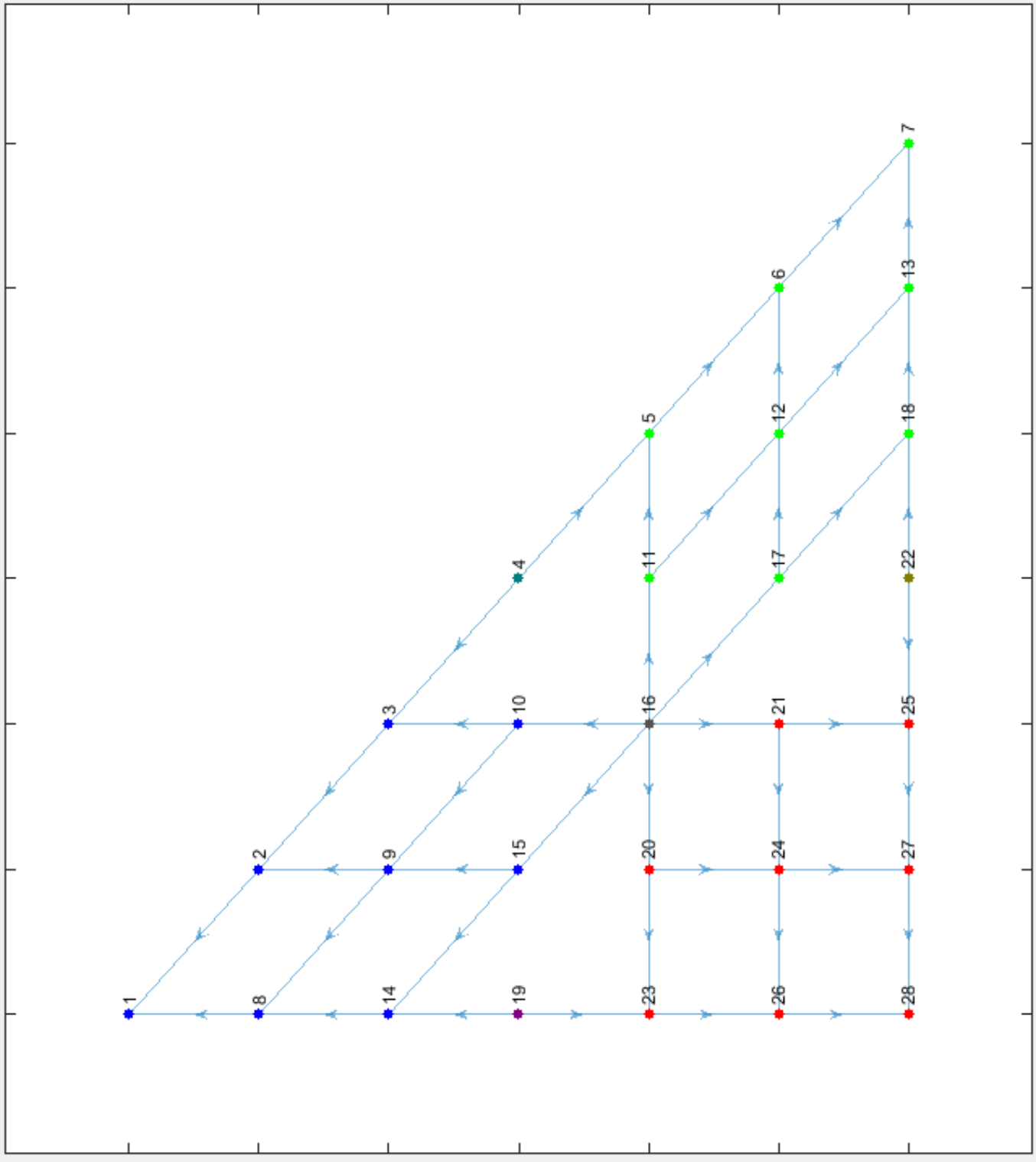


Figure 3.11: $N = 6$, GEN-2 vs SET-3 vs TFT. State transition graph based on `MTheory`.

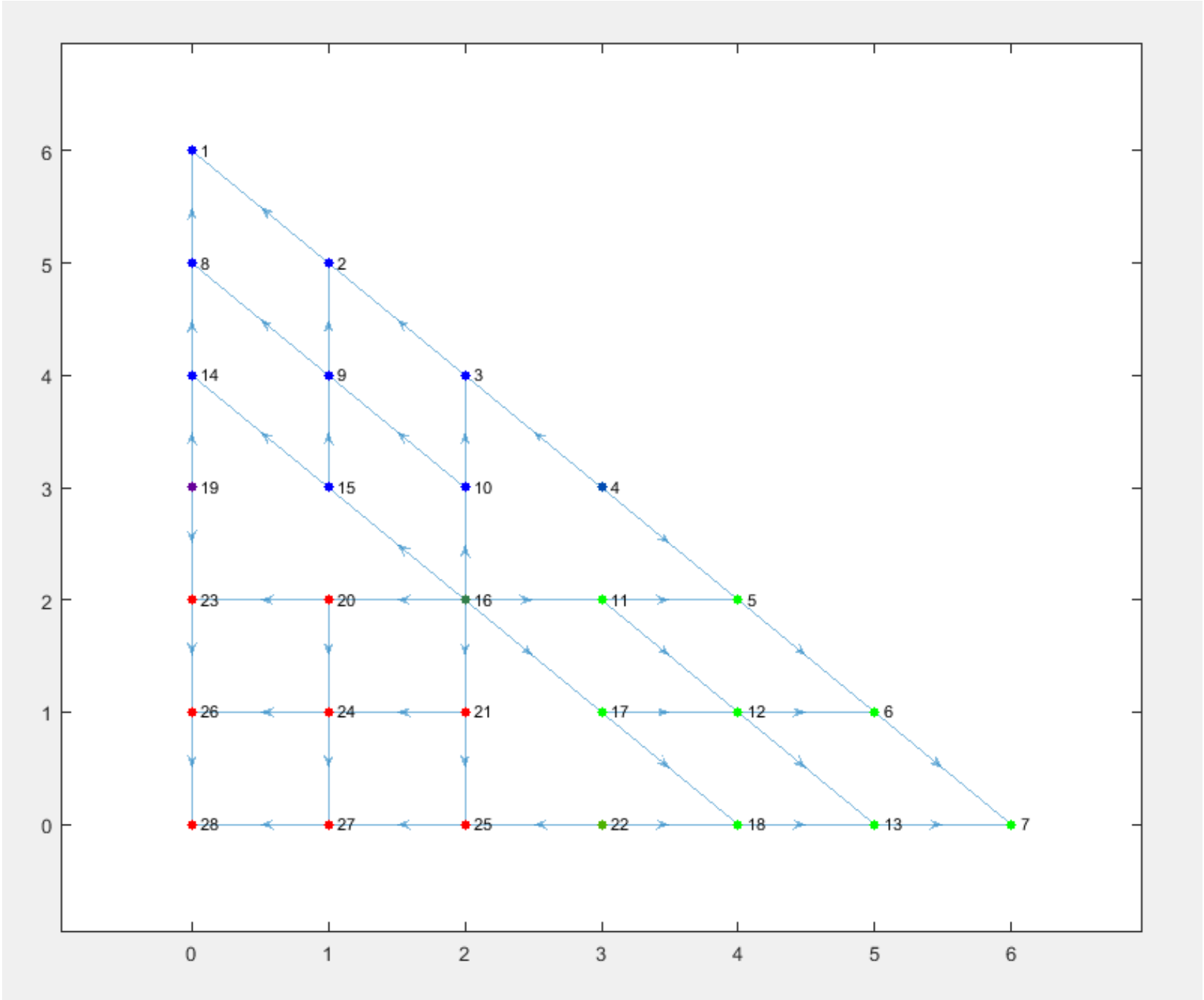


Figure 3.12: $N = 6$, GEN-2 vs SET-3 vs TFT. State transition graph based on `Msim`.

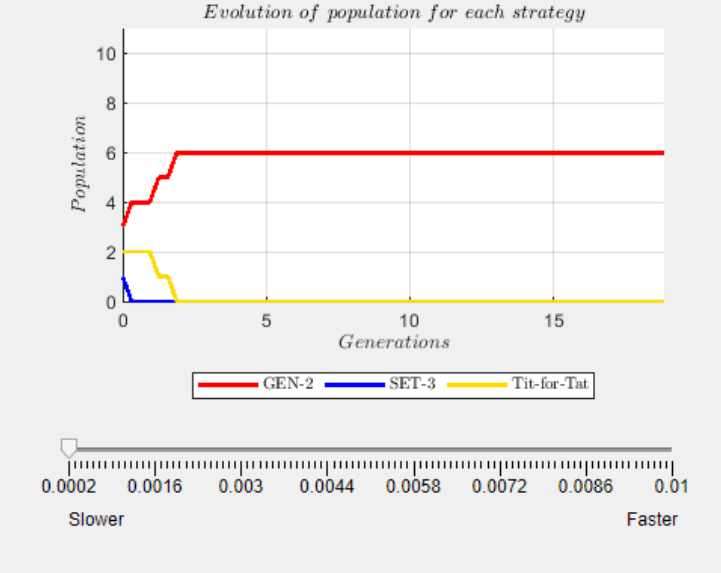


Figure 3.13: $N = 6$, GEN-2 vs SET-3 vs TFT. Example of the evolution of population for each strategy, starting from state (3,1,2).

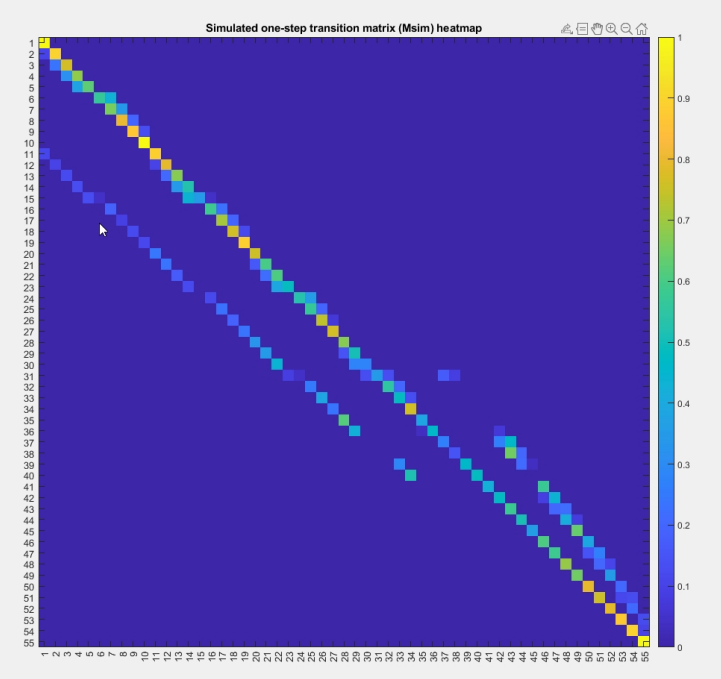


Figure 3.14: $N = 9$, GEN-2 vs SET-3 vs TFT. Simulated one-step transition matrix `Msim` heatmap.

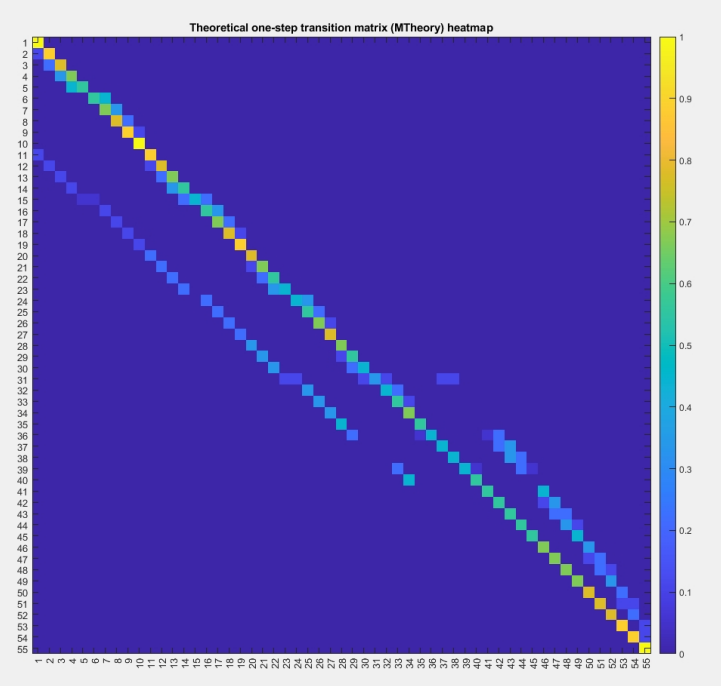


Figure 3.15: $N = 9$, GEN-2 vs SET-3 vs TFT. Theoretical one-step transition matrix `MTheory` heatmap.

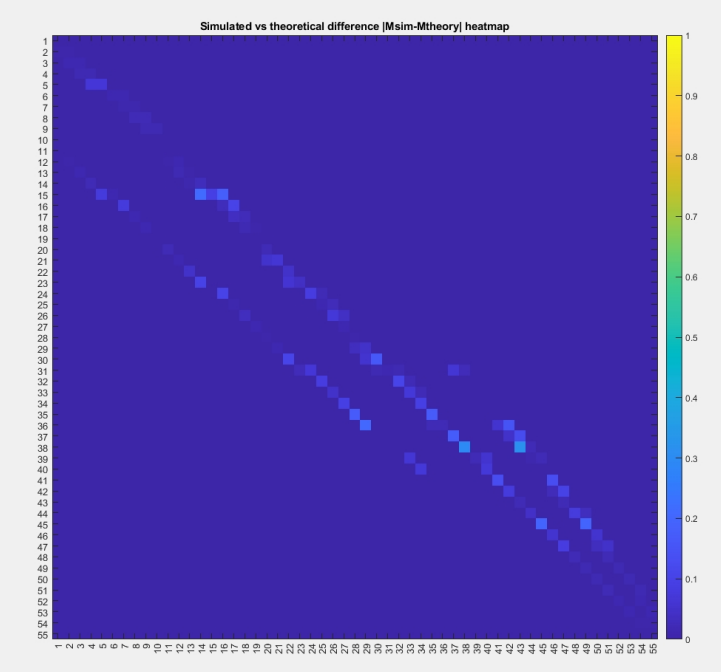


Figure 3.16: $N = 9$, GEN-2 vs SET-3 vs TFT. Heatmap of $|\texttt{Msim} - \texttt{MTheory}|$

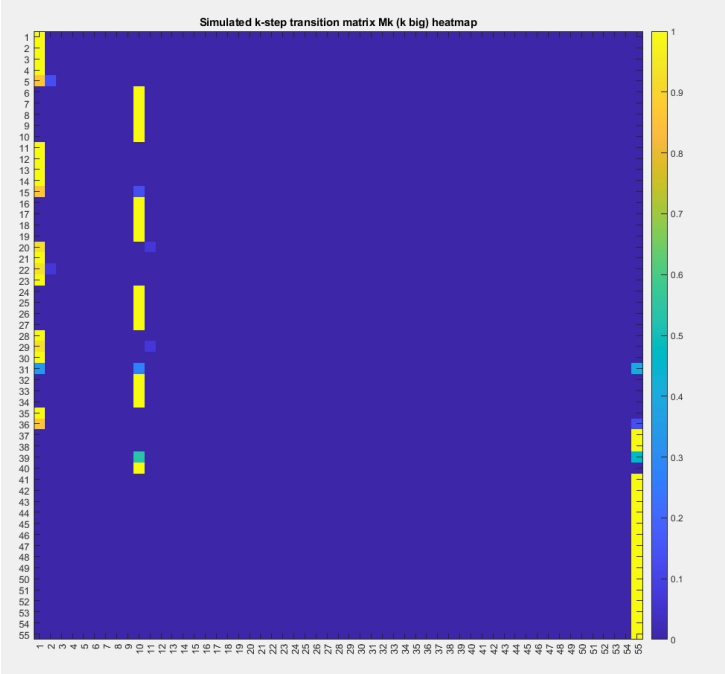


Figure 3.17: $N = 9$, GEN-2 vs SET-3 vs TFT. Simulated k-step transition matrix `Mk` heatmap.

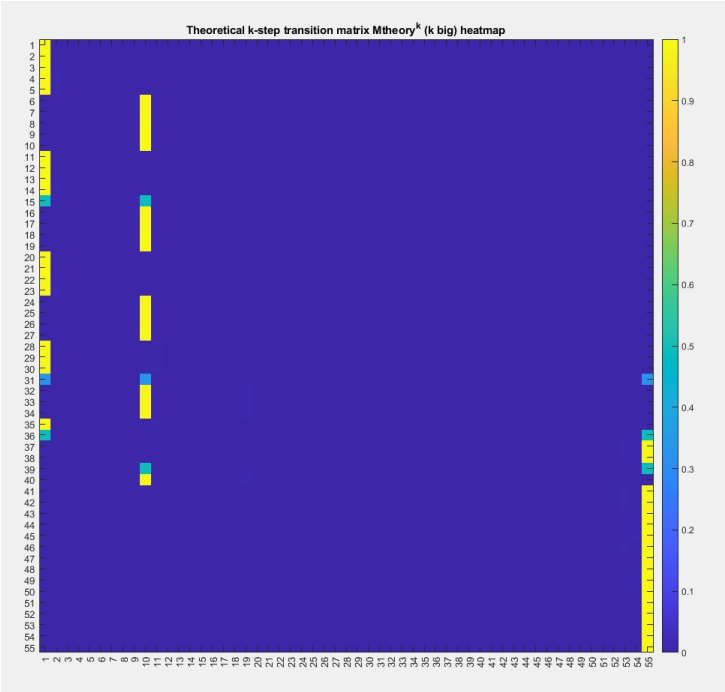


Figure 3.18: $N = 9$, GEN-2 vs SET-3 vs TFT. Theoretical k-step transition matrix (`MTheory`^{50} here).

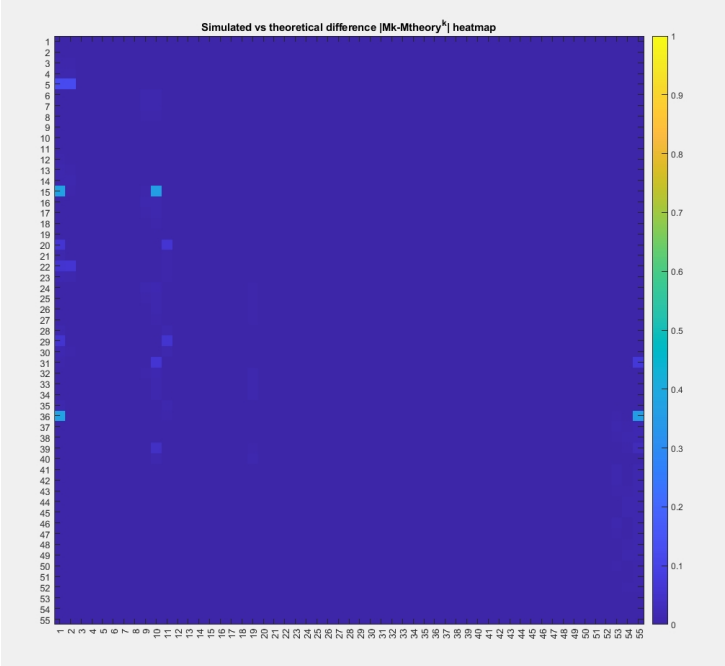


Figure 3.19: $N = 9$, GEN-2 vs SET-3 vs TFT. Heatmap of $|\texttt{Mk-MTheory}^{50}|$.

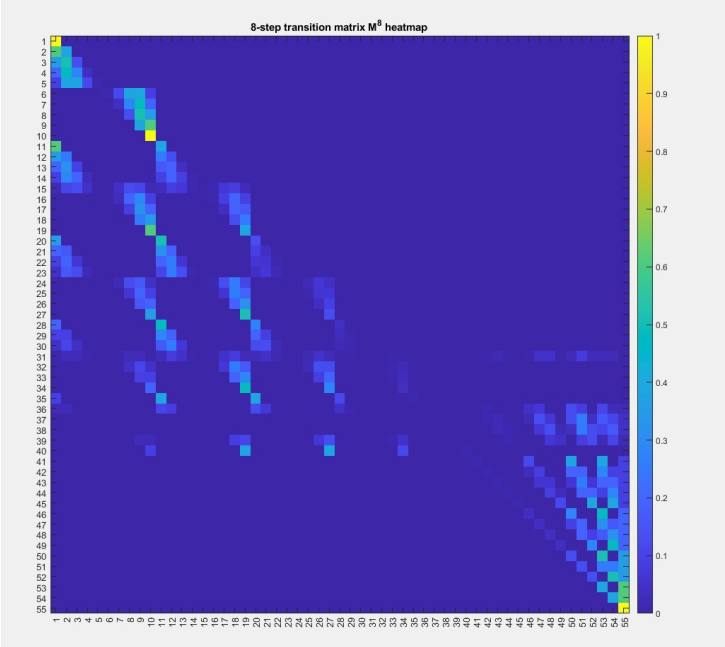


Figure 3.20: $N = 9$, GEN-2 vs SET-3 vs TFT. 8-step transition matrix $\texttt{MTheory}^8$. Probabilities of each state to transition to other states in eight steps.

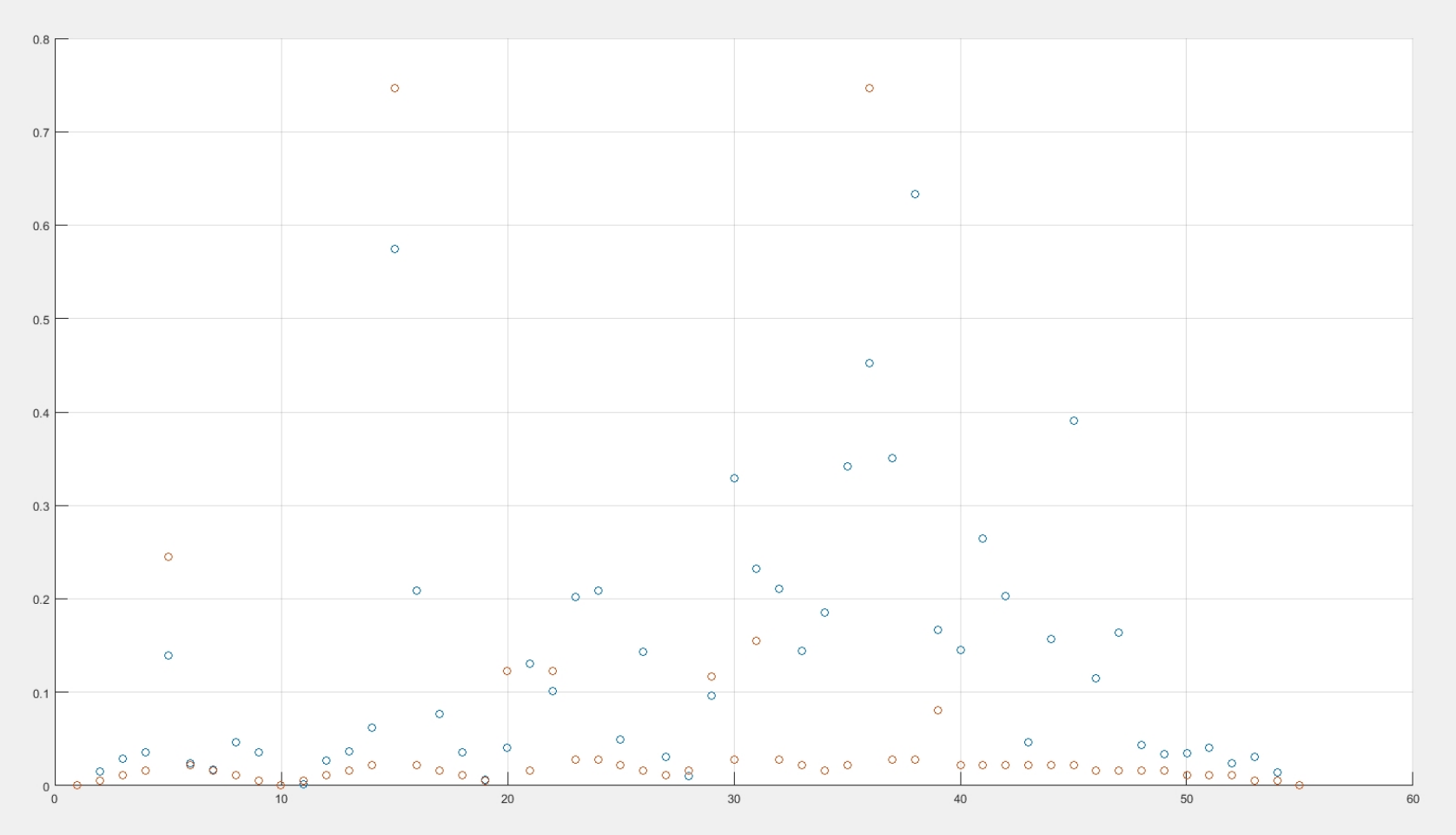


Figure 3.21: $N = 9$, GEN-2 vs SET-3 vs TFT. Scatter plot of the metric per row for the simulated one-step `Msim` (blue) and k-step `Mk` (red) transition matrices.

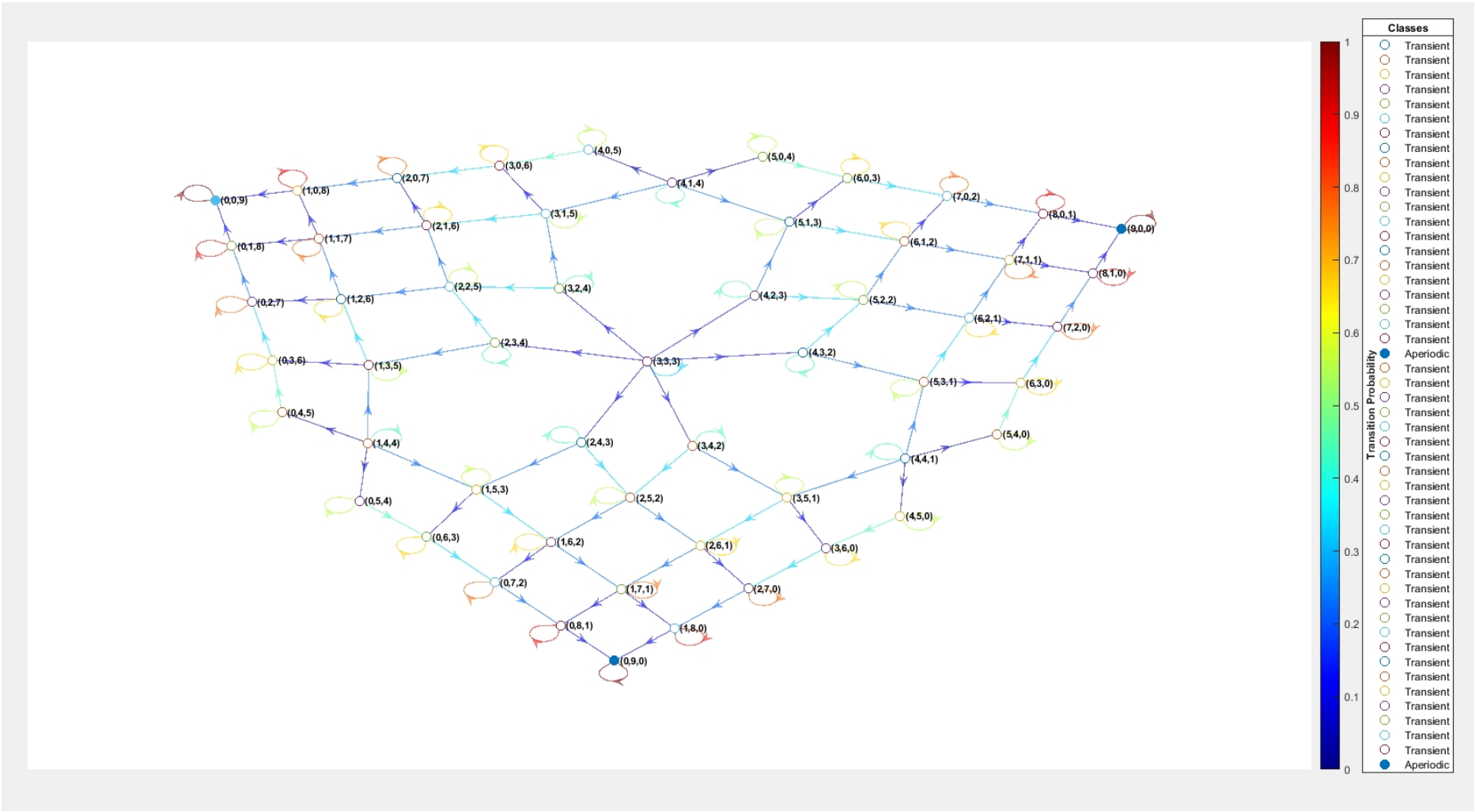


Figure 3.22: $N = 9$, GEN-2 vs SET-3 vs TFT. Digraph of the theoretically calculated one-step transition matrix `MTheory`.

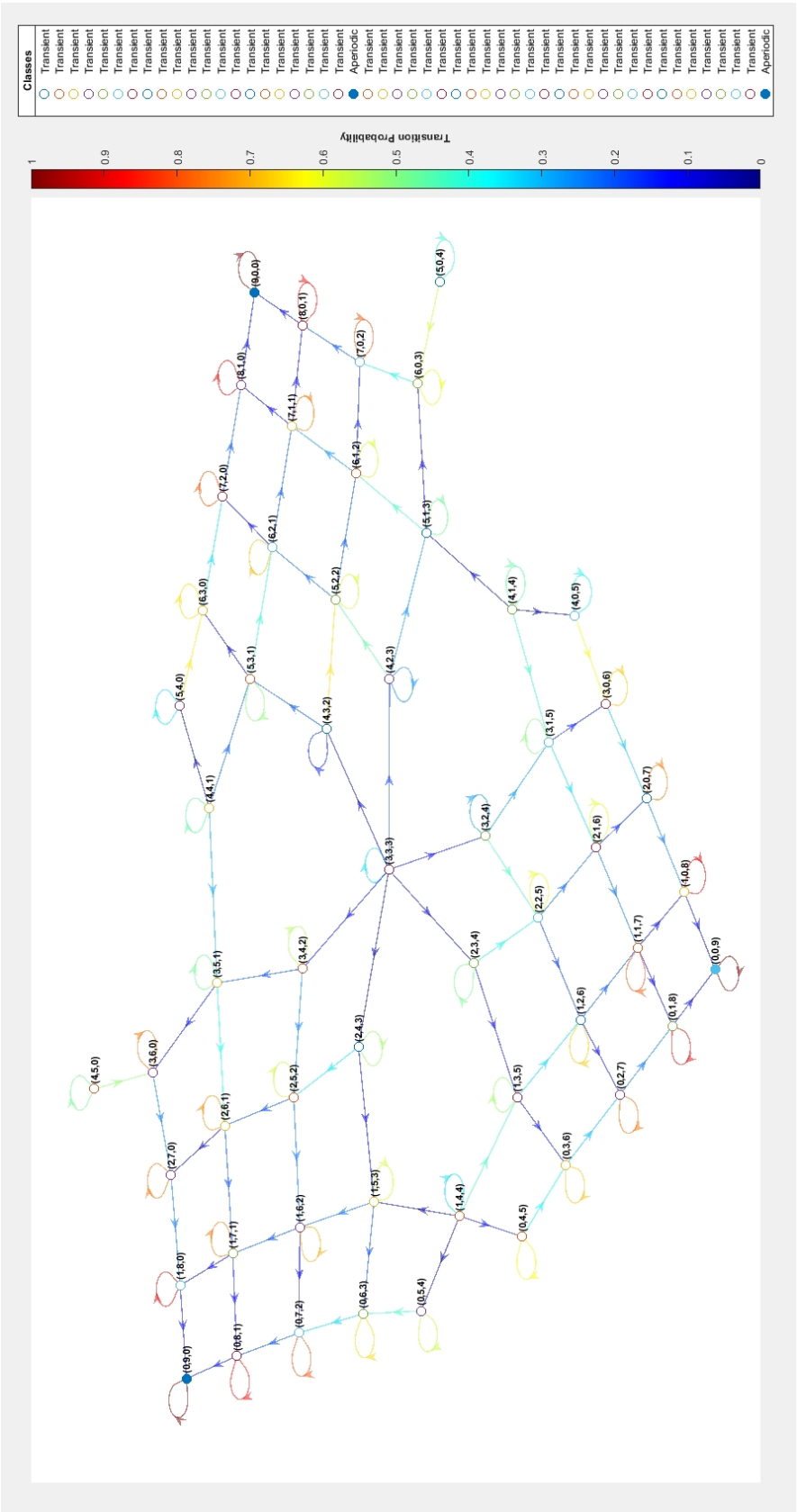


Figure 3.23: $N = 9$, GEN-2 vs SET-3 vs TFT. Digraph of the simulated one-step transition matrix `Msim`.

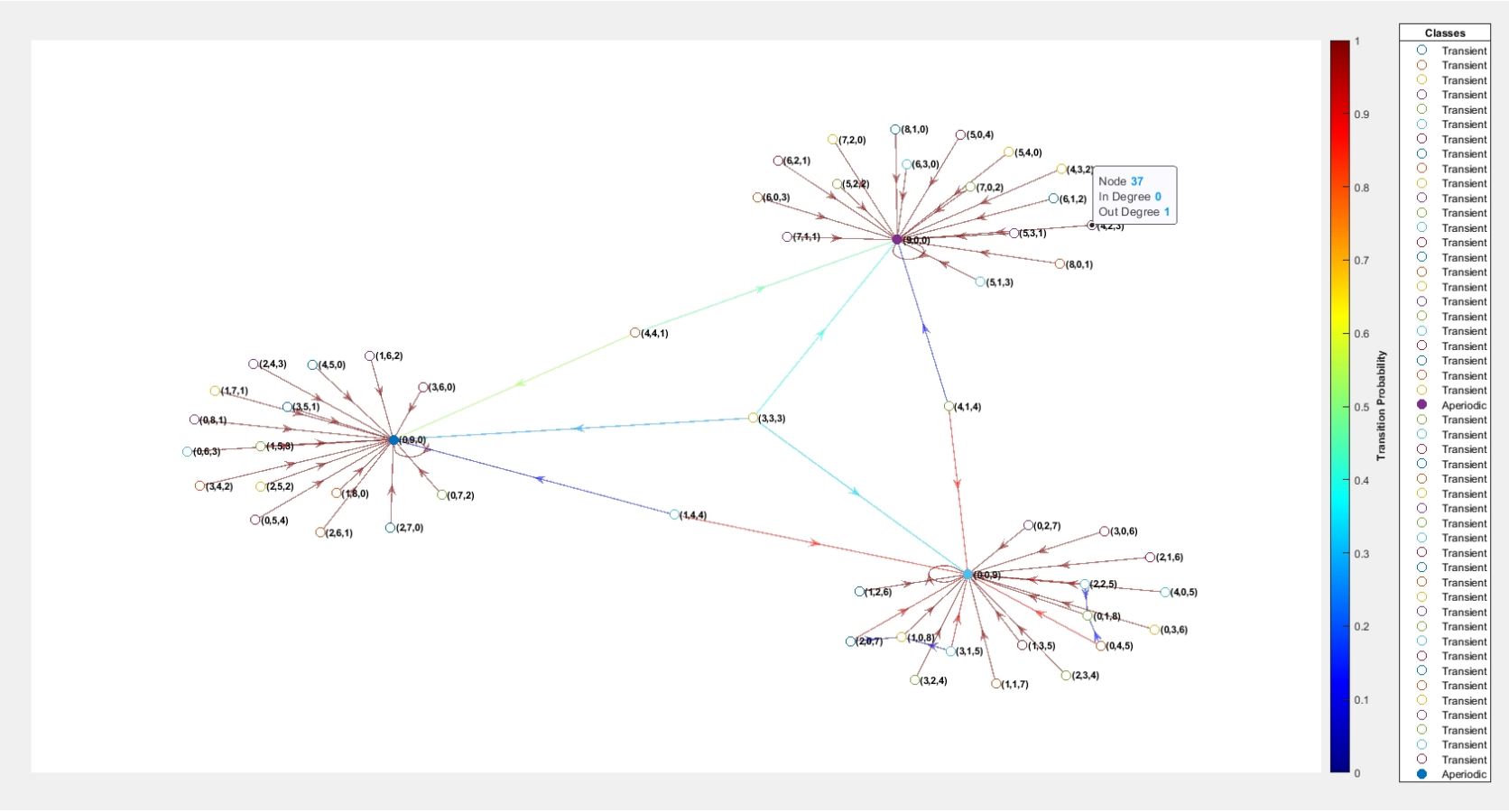


Figure 3.24: $N = 9$, GEN-2 vs SET-3 vs TFT. Digraph of the simulated k-step transition matrix `Mk`.

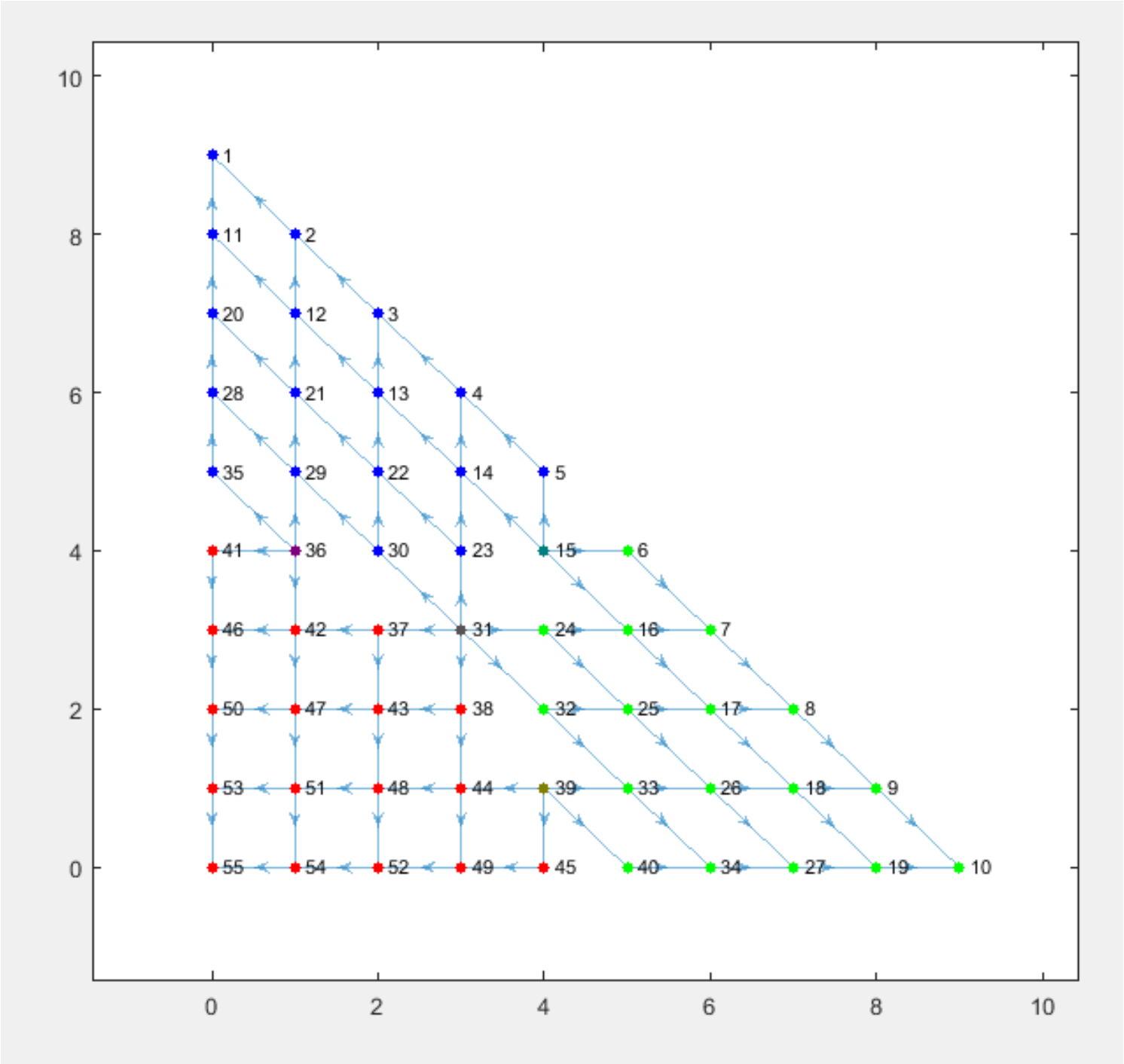


Figure 3.25: $N = 9$, GEN-2 vs SET-3 vs TFT. State transition graph based on `MTheory`.

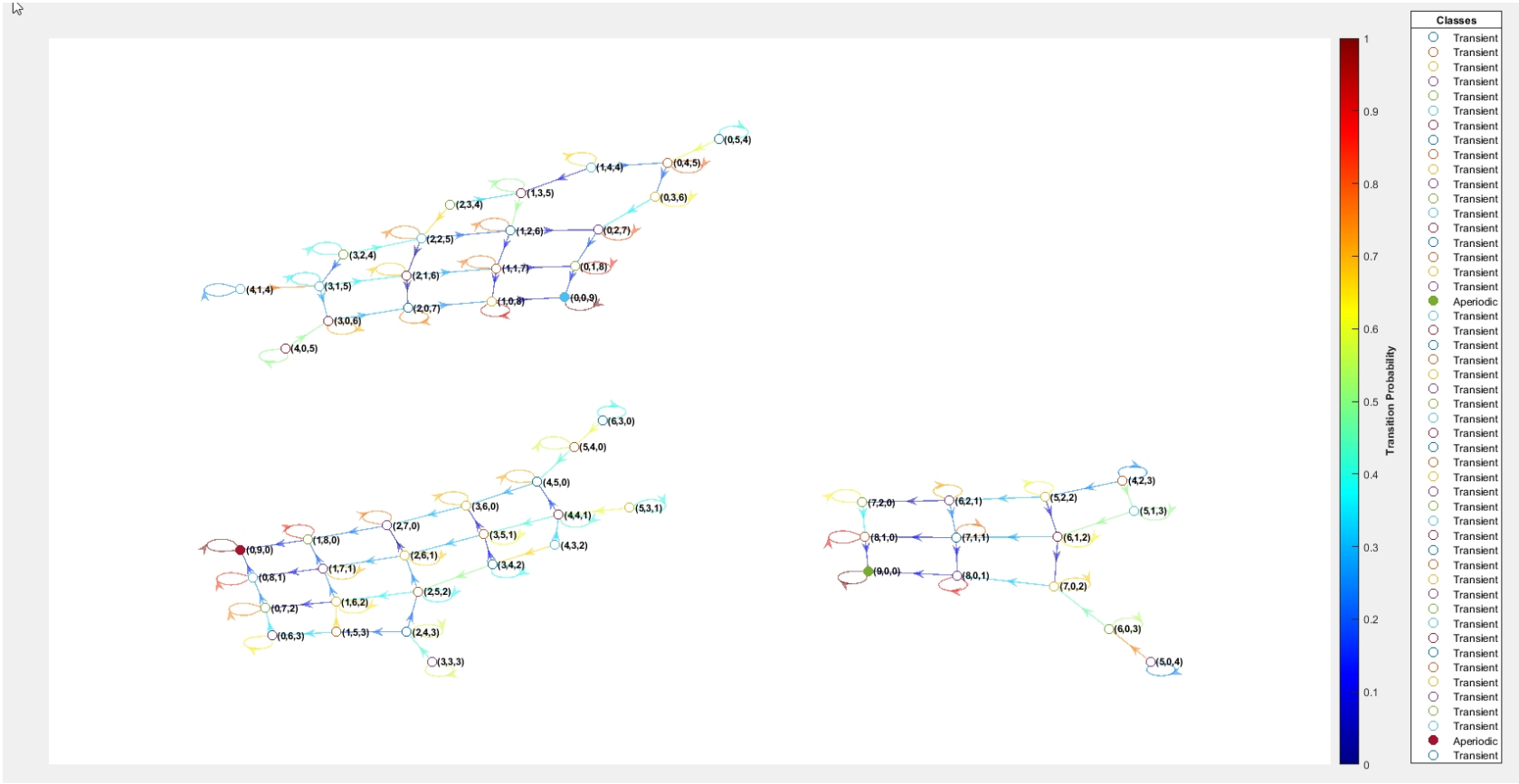


Figure 3.26: $N = 9$. Digraph of `Msim` in the case of All-C, All-D and Tit-for-Tat strategies.

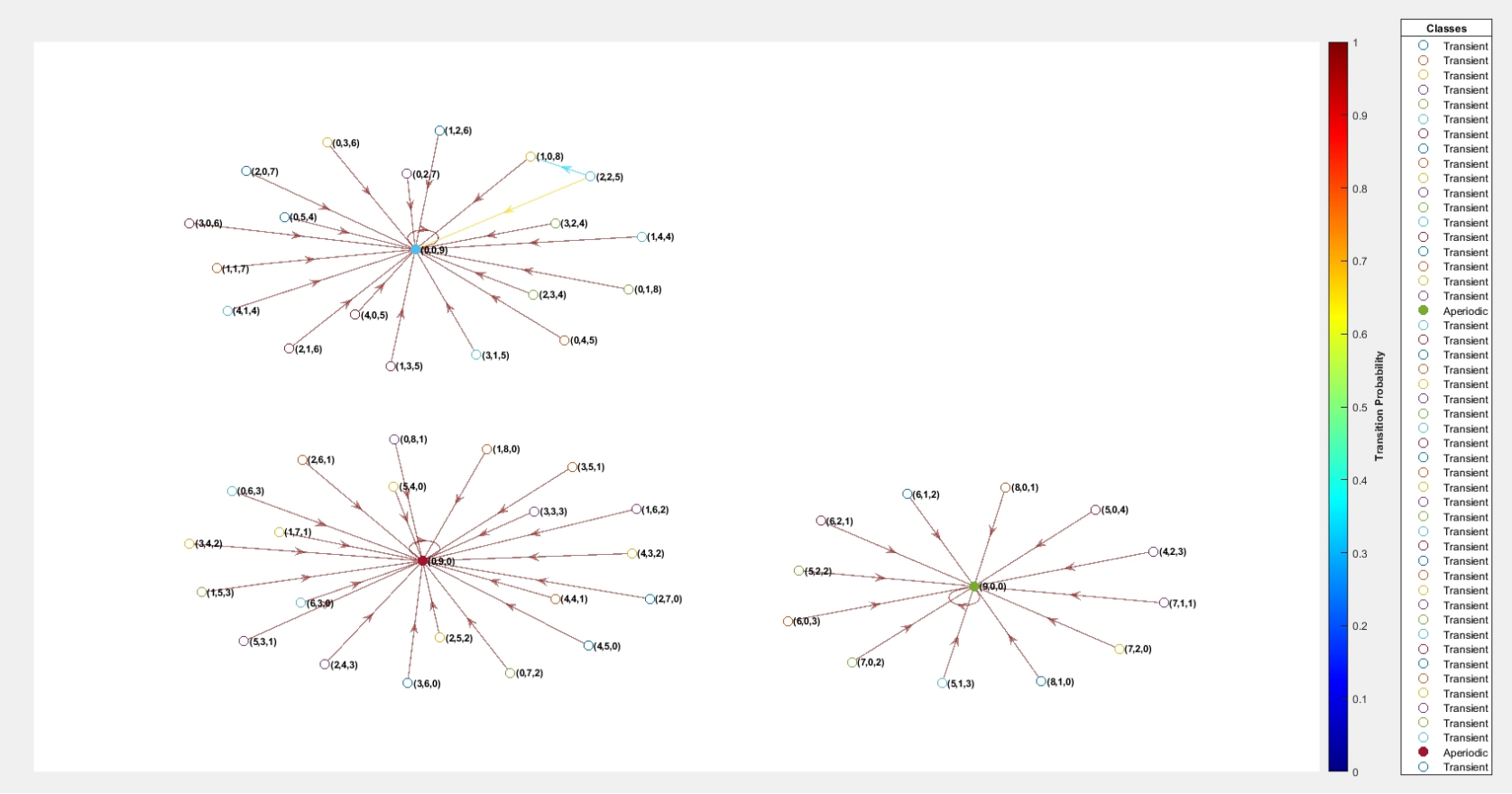


Figure 3.27: $N = 9$. Digraph of `Mk` in the case of All-C, All-D and Tit-for-Tat strategies.

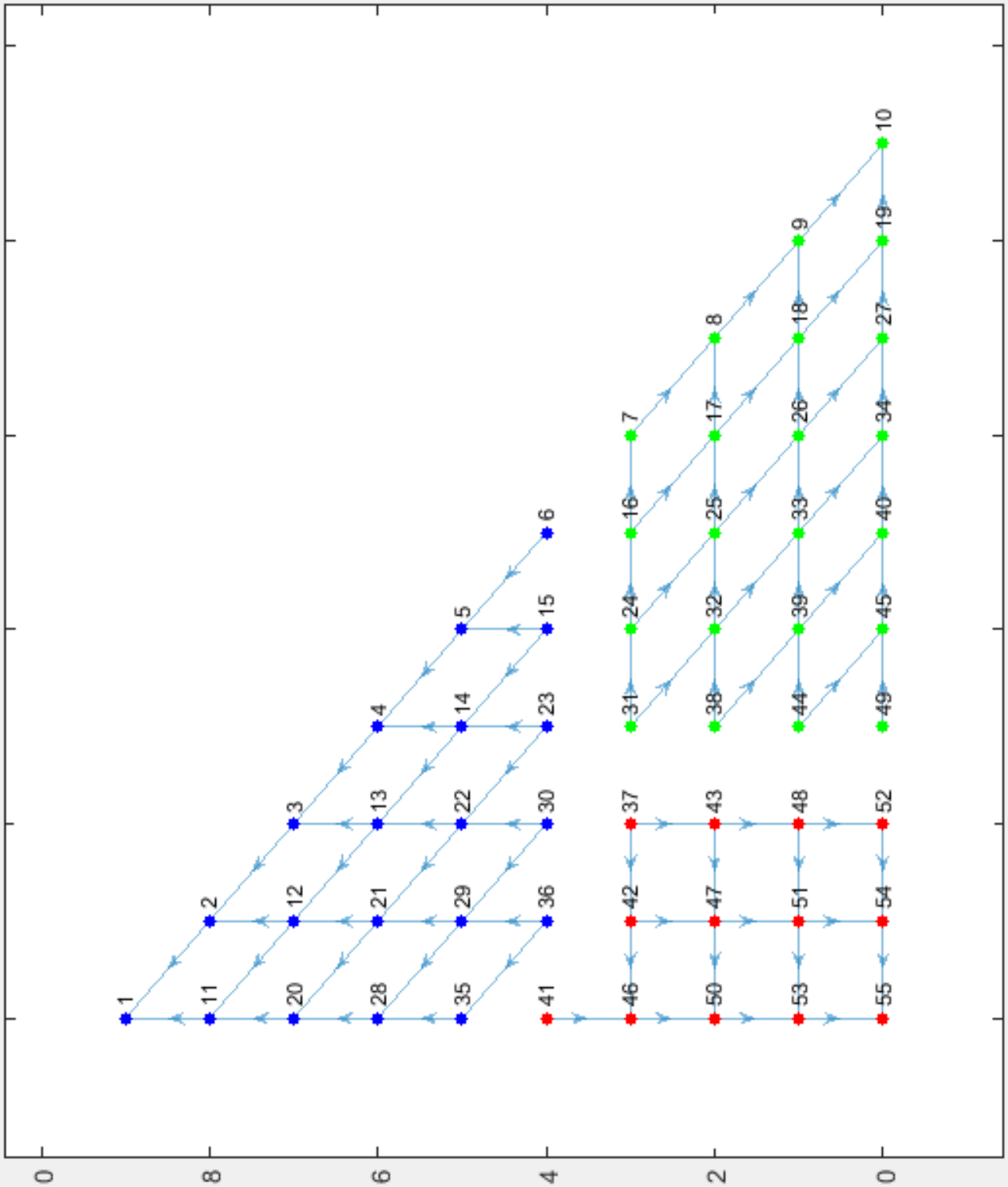


Figure 3.28: $N = 9$, All-C vs All-D vs Tit-for-Tat. State transition graph based on `MTheory`.

Chapter 4

References

Bibliography

[1] Alexander, J. McKenzie. *Evolutionary Game Theory*. Cambridge University Press, 2023.

[2] Axelrod, Robert, and William D. Hamilton. "The evolution of cooperation." *Science*, Vol. 211, No. 4489, pp. 1390–1396, 1981.

[3] Axelrod, Robert, and Douglas Dion. "The further evolution of cooperation." *Science*, Vol. 242, No. 4884, pp. 1385–1390, 1988.

[4] Mathieu, Philippe, Bruno Beaufils, and Jean-Paul Delahaye. "Studies on Dynamics in the Classical Iterated Prisoner's Dilemma with Few Strategies." *European Conference on Artificial Evolution*. Springer: Berlin Heidelberg, 1999.

[5] Hillier, Frederick S., and G. J. Lieberman. *Introduction to Operations Research*. 7th Edition, McGraw-Hill Higher Education, 2001. Chapter 16: Markov Chains. ISBN: 0-07-232169-5.

[6] Wolfram. "The Iterated Prisoner's Dilemma." `https://demonstrations.wolfram.com/TheIteratedPrisonersDilemma/`.

[7] Shulman. "Zero-Determinant Strategies in the Iterated Prisoner's Dilemma." *The n-Category Café Blog*. `https://golem.ph.utexas.edu/category/2012/07/zerodeterminant_strategies_in.html`.

[8] Press, William H., and Freeman J. Dyson. "Iterated Prisoner's Dilemma contains strategies that dominate any evolutionary opponent." *Proceedings of the National Academy of Sciences*, Vol. 109, No. 26, pp. 10409–10413, 2012. `https://www.pnas.org/doi/abs/10.1073/pnas.1206569109`.

Chapter 5

Appendix

5.1 Code Appendix

5.1.1 Player Class

Class Definition
Class Definition

`classdef player`

Class Attributes

```
index;       % Player's index
score;       % Player's score
history;     % Empty array Strategies x Rounds
move;        % Player's move
```

Methods

- **Constructor**

  `function obj = player()`

  Creates a Player.

- **Set Score**

  `function obj = setScore(obj, roundScore)`

  - `obj`: The player object.
  - `roundScore`: The score to add.
  - `returns obj`: The updated player.

  Adds the score to the player's total.

- **Get Score**

```
function score = getScore(obj)
```

  - `obj`: The player object.
  - `returns score`: The player's score.

- **Initialize History**

```
function obj = initHistory(obj, rounds, numberPlayers)
```

  - `obj`: The player object.
  - `rounds`: Total tournament rounds.
  - `numberPlayers`: Number of players.
  - `returns obj`: The updated player.

  Initializes the history matrix with zeros.

- **Set History**

```
function obj = setHistory(obj, round, opponentIndex, move)
```

  - `obj`, `round`, `opponentIndex`, `move`
  - `returns obj`: The updated player.

  Archives a move.

- **Get History**

```
function history = getHistory(obj)
```

  - `obj`: The player object.
  - `returns history`: The move history.

- **Get History Element**

```
function historyElement = getHistoryElement(obj, round, opponentIndex)
```

  - `obj`, `round`, `opponentIndex`
  - `returns historyElement`

  Gets a specIfic move from history.

- **Set Index**

```
function obj = setIndex(obj, index)
```

  - `index`: The player index.
  - `returns obj`: The updated player.

- **Get Index**

```
function index = getIndex(obj)
```

  - `obj`: The player object.
  - `returns index`

- **Get Move**

```
function move = getMove(obj)
```

  - `obj`: The player object.
  - `returns move`

Strategies

Strategies are subclasses of `player`. They implement a unique `move()` method and may have additional attributes. See `Documentation/strategies/` for details.

5.1.2 The Axel Function

Function Definition

```
function Axel(strategiesArray, populationsArray, matrix, rounds)
```

Parameters

- `strategiesArray`: Array that includes all the participating strategies.
- `populationsArray`: Array that specIfies the populations of each strategy in the strategiesArray.
- `matrix`: Tournament's payoff matrix.
- `rounds`: Number of tournament rounds.

Description

The **Axel** function starts a Tournament by creating an `axelrod` object and initializing the players and the tournament itself. It prints the results to the console.

5.1.3 The axelrod class

Class Definition

`classdef axelrod`

Class Attributes

```
players;       % Cell array of player objects
rounds;        % Total number of rounds in the tournament
currentRound;  % Current round of the tournament
payoffMatrix;  % Payoff matrix (2x2 cell array)
```

Methods

- **Constructor**

  `function obj = axelrod()`

  `returns obj`: The axelrod object.

  Creates an axelrod object.

- **Initialize Axelrod**

  `function obj = initAxel(obj, playersArray, matrix, rounds)`

  - `obj`: The axelrod object.
  - `playersArray`: Array of initialized players.
  - `matrix`: Payoff matrix.
  - `rounds`: Number of rounds.
  - `returns obj`: The updated object.

- **Initialize Players**

  `function obj = InitPlayers(obj, strategiesArray, populationsArray, rounds)`

  - `obj`: The axelrod object.
  - `strategiesArray`: Array of strategy indices.
  - `populationsArray`: Player counts per strategy.
  - `rounds`: Number of rounds.
  - `returns obj`: The updated object.

  Dynamically creates and initializes players.

- **Set Players**

  `function obj = setPlayers(obj, players)`

  - `obj`: The axelrod object.
  - `players`: Player objects.
  - `returns obj`: The updated object.

- **Get Players**

  `function players = getPlayers(obj)`

  `obj`: The axelrod object.
  `returns players`: The player array.

- **Set Rounds**

  `function obj = setRounds(obj, rounds)`

  - `obj`, `rounds`
  - `returns obj`

- **Get Rounds**

  `function rounds = getRounds(obj)`

  `returns rounds`: Total number of rounds.

- **Set Current Round**

  `function obj = setCurrentRound(obj, currentRound)`

  - `obj`, `currentRound`
  - `returns obj`

- **Get Current Round**

  `function currentRound = getCurrentRound(obj)`

  `returns currentRound`: Current round number.

- **Set Payoff Matrix**

```
function obj = setPayoffMatrix(obj, payoffMatrix)
```

  - `obj`, `payoffMatrix`
  - `returns obj`

- **Get Payoff Matrix Element**

```
function payoffMatrix = getPayoffMatrixElement(obj, row, column)
```

  - `obj`, `row`, `column`
  - `returns payoffMatrix`

  Retrieves a value using move indices (0: cooperate, 1: defect).

- **Encounter**

```
function [player1, player2] = encounter(obj, player1, player2, currentRound)
```

  - `obj`, `player1`, `player2`, `currentRound`
  - `returns player1, player2`

  Simulates and updates a match between two players.

- **Play Round**

```
function obj = playRound(obj)
```

  `returns obj`: Plays a full round of encounters.

- **Begin Tournament**

```
function obj = begin(obj)
```

  `returns obj`: Plays all rounds sequentially.

- **Print Results**

```
function printResults(obj, strategiesArray, populationsArray)
```

  - `obj`, `strategiesArray`, `populationsArray`
  - `returns void`

  Prints total scores for each strategy.

5.1.4 The genaxel Class

Class Definition

```
classdef genaxel
```

Class Attributes

```
static_totalplayers = 0; % Total number of players in the population
```

Methods

- **Constructor**

  ```
  function obj = genaxel()
  ```

  `returns obj`: The genaxel object.

  Creates a new genaxel object.

- **TheoreticalFitness**

  ```
  function [obj, Wn, V] = TheoreticalFitness(obj, b, strategies, pop0, T, rounding)
  ```

  - `obj`: The genaxel object.
  - `b`: The payoff matrix.
  - `strategies`: Array of strategies.
  - `pop0`: Initial population for generation 0.
  - `T`: Number of rounds per game.
  - `rounding`: Method for rounding population updates.
  - `returns Wn`: Population of the next generation.
  - `returns V`: Matrix of scores for each strategy against others.

  Runs a theoretical tournament to evaluate strategy fitness and updates populations accordingly.

- **SimFitness**

  ```
  function [obj, Wn] = SimFitness(obj, b, strategies, pop0, T)
  ```

  - `obj`: The genaxel object.
  - `b`: The payoff matrix.
  - `strategies`: Array of strategies.
  - `pop0`: Population at generation n.

- `T`: Number of rounds.
- `returns Wn`: Updated population.

Simulates a tournament with the given population and returns updated population sizes based on scores.

- **ImitationSim**

```
function [obj, Wn] = ImitationSim(obj, b, strategies, pop0, K, T)
```

  - `obj`: The genaxel object.
  - `b`: The payoff matrix.
  - `strategies`: Array of strategies.
  - `pop0`: Population of generation n.
  - `K`: Number of imitation steps.
  - `T`: Number of rounds.
  - `returns Wn`: Updated population.

  Applies an imitation rule where agents adopt successful strategies based on scores.

- **plotgen**

```
function obj = plotgen(obj, generations, popHistory, strategies)
```

  - `obj`: The genaxel object.
  - `generations`: Total number of generations.
  - `popHistory`: Matrix tracking population evolution.
  - `strategies`: Array of strategy indices.
  - `returns obj`

  Generates a plot showing how each strategy's population evolves across generations.

5.1.5 TourSimFit function

`function TourSimFit(b , strategies , pop0 , T , J)`

- **b**: The benefit parameter for the game.
- **strategies**: The array of available strategies (see details in *Examples/script.m*).
- **pop0**: The initial population distribution across the strategies.
- **T**: The number of rounds per tournament.
- **J**: The number of generations to simulate.

Runs a tournament using selected strategies and populations only, tracking individual scores and computing a total. Each strategy's population is then updated proportionally to its score, reflecting dynamic interactions in populations over time.

5.1.6 TourSimImit function

`function TourSimImit(b, strategies, pop0, K, T, J)`

- **b**: The payoff parameter(s) for the game.
- **strategies**: The array of available strategies (see details in *Examples/script.m*).
- **pop0**: The initial population distribution across the strategies.
- **K**: The size of the tournament / selection parameter.
- **T**: The number of rounds per tournament.
- **J**: The number of generations to simulate.

5.1.7 TourTheFit function

`TourTheFit(obj, b , strategies , pop0 , T , J , rounding)`

- **b**: The payoff parameter(s) for the game.
- **strategies**: The array populated by the strategy definitions.
- **pop0**: The initial population distribution across the strategies.
- **T**: The number of rounds per tournament.
- **J**: Number of generations to simulate.
- **rounding**: Controls how fractional population values are rounded.

Runs a round-robin tournament to compute payoffs, then updates the population using fitness-proportional rules based on Mathieu's formula. Scores are weighted by opponent populations, normalized, and rounded to determine the next generation's distribution.

5.2 Markov Theory Code Appendix

Script MarkovRun.m

Use

This is the main script ro run. Calls `initialize()` for initialization, `MarkovTheory()` to run the theretical part, `multimultigen()` to run the simulation part and `simGraphs()` to draw the simulation part graphs.

Function initialize()

```
[p1, p2, p3, payoff, N, roundsth, roundssim, numofreps,
alluniquestates, initprobs, ngens, predetermined,
names, chosen] = initialize()
```

`@ p1, @p2, @p3`: Strategy probability vectors.
`@ payoff`: Payoff matrix.
`@ N`: Total number of players.
`@ roundsth, @roundssim`: Rounds per match for the theoretical and simulation parts.
`@ numofreps`: Number of repetitions per initial state.
`@ alluniquestates`: All the unique states.
`@ initprobs`: Vector with the probabilities for each strategy to choose C as initial move.
`@ ngens`: Number of generations.
`@ predetermined`: Equals empty string '' to denote use of strategies with strategy vectors. Use 'A', 'B',... to denote use of predetermined outcome meetings.
`@ names`: Map with all the strategy names.
`@ chosen`: Vector with the indices of the strategies to compete.

Use

Initially setup parameters and choices.

Called by

Called by the main script `MarkovRun.m`

Function MarkovTheory()

```
[Ps, allcurrentstates,allnextstates,allprobs, MTheory, mcTheory]
   ↪ =
MarkovTheory(p1, p2, p3, payoff, N, roundsth, predetermined,
alluniquestates)
```

`@ Ps`: Match payoffs for any two-by-two combination of players.

`@ allcurrentstates, @ allnextstates, @ allprobs`: Contain the information for all state transitions. In the same row of these three matrices we have respectively the current state, the next state and the probability for a certain state transition.

`@ MTheory`: The theoretically calculated transition matrix.

`@ mcTheory`: The Markov chain model for MTheory.

`@ p1, @p2, @p3`: Strategy probability vectors.

`@ payoff`: Payoff matrix.

`@ N`: Total number of players.

`@ roundsth`: Rounds per match for the theoretical part.

`@ predetermined`: Equals empty string '' to denote use of strategies with strategy vectors. Use 'A', 'B',... to denote use of predetermined outcome meetings.

`@ alluniquestates`: All the unique states.

Use

Calls `tournamentpayoffs()`, or `determiniosticPayoffs()`, depending on whether we use strategies with strategy vectors or predetermined outcome meetings, `transitionRules()` and `theoryGraphs()` to perform theoretical calculations.

Called by

Called by `MarkovRun()`.

Function tournamentpayoffs()

```
Ps = tournamentpayoffs(p1, p2, p3, R, T, S, P)
```

`@ Ps`: Match payoffs for any two-by-two combination of players.

`@ p1, @p2, @p3`: Strategy vectors.

`@ R, @T, @S, @P`: Elements of the game payoff matrix.

Use

Calculate the total strategy payoffs during a meeting. Calls `payoff()`.

Called by

Called by `MarkovTheory()`.

Function payoff()

```
[M, pi, payoff1, payoff2] = payoff(p, q, R, S, T, P)
```

`@ M`: The state transition matrix.
`@ pi`: The stationary distribution π.
`@ payoff1, @payoff2`: The long-run expected average payoffs per round of the two players in a match.
`@ p, @q`: The strategy vectors of the two opponent strategies in a game.
`@ R, @T, @S, @P`: Elements of the game payoff matrix.

Use

Calculates the state transition matrix, the stationary distribution and the expected average payoffs per round. Calls `transitionMatrix()`.

Called by

Called by `tournamentpayoffs()`.

Function transitionMatrix()

```
M = transitionMatrix(p, q)
```

`@ M`: The state transition matrix.
`@ p, @q`: The strategy vectors of the two opponent strategies in a game.

Use

Calculates the state transition matrix.

Called by

Called by `payoff()`.

Function deterministicPayoffs()

```
Ps = deterministicPayoffs(R, T, S, P, roundsth, deterministic)
```

`@ Ps`: Match payoffs for any two-by-two combination of players.
`@ R, @T, @S, @P`: Elements of the game payoff matrix.
`@ roundsth`: Rounds per match for the theoretical part.
`@ deterministic`: Equals empty string '' to denote use of strategies with strategy vectors. Use 'A', 'B',... to denote use of predetermined outcome meetings.

Use

Calculates the match payoffs for any two-by-two combination of players in the case of predetermined outcome meetings.

Called by

Called by `MarkovTheory()`.

Function transitionRules()

```
[allcurrentstates, allnextstates, allprobs] =
transitionRules(Ps, N, roundsth)
```

`@ allcurrentstates, @ allnextstates, @ allprobs`: Contain the information for all state transitions. In the same row of these three matrices we have respectively the current state, the next state and the probability for a certain state transition.

`@ Ps`: Match payoffs for any two-by-two combination of players.

`@ N`: Total number of players.

`@ roundsth`: Rounds per match for the theoretical part.

Use

Find all possible state transitions, given strategy populations and payoffs. Calls `strategyPayoffs()` to calculate the total strategy payoffs during a meeting, `remaintosamestate()` and `transferfromXtoY()`.

Called by

Called by `MarkovTheory()`.

Function strategyPayoffs()

```
[SA, SB, SC] = strategyPayoffs(Ps, n, roundsth)
```

`@ SA, @ SB, @ SC`: The total payoffs, per generation, for the three strategies.

`@ Ps`: Match payoffs for any two-by-two combination of players.

`@ n`: n = [n1, n2, n3] is the vector of strategy populations.

`@ roundsth`: Rounds per match for the theoretical part.

Use

Returns the total expected average payoff for each one of the three strategies, for the current generation.

Called by

Called by `transitionRules()`.

Function remaintosamestate()

```
[allcurrentstates, allnextstates, allprobs] =
remaintosamestate(probability, n,
allcurrentstates, allnextstates, allprobs)
```

`@ allcurrentstates`, `@ allnextstates`, `@ allprobs`: Contain the information for all state transitions. In the same row of these three matrices we have respectively the current state, the next state and the probability for a certain state transition.

`@ probability`: The probability to remain to the same state.

`@ n`: n = [n1, n2, n3] is the current vector of strategy populations.

Use

Add a new row in `@ allcurrentstates`, `@ allnextstates`, `@ allprobs`, when we remain to the same state `@ n` with `@ probability`.

Called by

Called by `transitionRules()`.

Function transferfromXtoY()

```
[allcurrentstates, allnextstates, allprobs] =
transferfromXtoY(strategyX, strategyY, probability,
n, idx, allcurrentstates, allnextstates, allprobs)
```

`@ allcurrentstates`, `@ allnextstates`, `@ allprobs`: Contain the information for all state transitions. In the same row of these three matrices we have respectively the current state, the next state and the probability for a certain state transition.

`@ strategyX`: Index of the strategy to transfer the player from.

`@ strategyY`: Index of the strategy to transfer the player to.

`@ probability`: The probability for the transition to occur.

`@ n`: n = [n1, n2, n3] is the current vector of strategy populations.

`@ idx`: Vector of indices of the three strategies when sorted in descending order, with respect to their total expected average payoff in the current generation.

Use

Add a row in `@ allcurrentstates`, `@ allnextstates`, `@ allprobs`, with a new state transition, in the case we substract a player from `@ strategyX` and add it to `@ strategyY`.

Called by

Called by `transitionRules()`.

Function theoryGraphs()

```
[M, mcTheory] = theoryGraphs(allcurrentstates, allnextstates,
allprobs, N, alluniquestates)
```

@ M: The theoretically calculated transition matrix.

@ mcTheory: The Markov chain model for @ M.

@ allcurrentstates, @ allnextstates, @ allprobs: Contain the information for all state transitions. In the same row of these three matrices we have respectively the current state, the next state and the probability for a certain state transition.

@ N: Total number of players.

@ alluniquestates: All the unique states.

Use

Draws all the graphs for the theoretical part. Returns the theoretically calculated transition matrix @ M and its Markov chain model.

Called by

Called by MarkovTheory().

Function graphPlot()

```
graphPlot(mc, M, alluniquestates, N)
```

@ mc: The Markov chain model for the theoretically calculated transition matrix @ M.

@ M: The theoretically calculated transition matrix.

@ alluniquestates: All the unique states.

@ N: Total number of players.

Use

Draws the state transition graph, based on @ M.

Called by

Called by theoryGraphs().

Function multimultigen()

```
[row, m, repgenNumOfPlayersHistory] = multimultigen(p1, p2, p3,
initialstate, rounds, payoff, initprobs,
numofreps, alluniquestates, ngens)
```

@ row: The index of the row of the k-step (k big) transition matrix @ M when considering the @ numofreps repetitions, all starting from the same @ initialstate.

@ m: The row of the k-step (k big) transition matrix @ M when considering the @ numofreps repetitions, all starting from the same @ initialstate.

@ repgenNumOfPlayersHistory: Holds all the history of the evolution in time of the population of each strategy for all the repetitions starting from the same @ initialstate.

@ p1, @ p2, @p3: The strategy vectors of the three competing strategies.

@ initialstate: The initial state for all the @ numofreps repetitions.

@ rounds: Rounds per match for the simulation part.

@ payoff: Holds the game payoff matrix elements @ R, @ S, @ T, @ P.

@ initprobs: Vector with the probabilities for each strategy to choose C as initial move.

@ numofreps: Number of repetitions per initial state.

@ alluniquestates: All the unique states.

@ ngens: Number of generations.

Use

Performs the many repetitions of the generation evolution, all starting from the same @ initialstate. Calls, multigen().

Called by

Called by MarkovRun.m.

Function multigen()

```
[finalstate, genNumOfPlayersHistory] = multigen(p1, p2, p3,
initialstate, rounds, payoff, initprobs, alluniquestates, ngens)
```

@ finalstate: The final state reached, starting the generation evolution from @ initialstate.

@ genNumOfPlayersHistory: Holds the evolution in time of the population of each strategy for a specific repetition starting from the state @ initialstate.

@ p1, @ p2, @p3: The strategy vectors of the three competing strategies.

@ initialstate: The initial state for all the @ numofreps repetitions.

@ rounds: Rounds per match for the simulation part.

@ payoff: Holds the game payoff matrix elements @ R, @ S, @ T, @ P.

@ initprobs: Vector with the probabilities for each strategy to choose C as initial move.

@ alluniquestates: All the unique states.

@ ngens: Number of generations.

Use

Performs one of the repetitions of the generation evolution, starting from the @ initialstate. Calls gentournament().

Called by

Called by multimultigen().

Function gentournament()

```
[nextstate, tournamentscores] = gentournament(p1, p2, p3,
currentstate, rounds, payoff, initprobs, alluniquestates)
```

@ nextstate: The next state in the generation evolution.

@ tournamentscores: Vector with the total expected average payoffs of the competing strategies.

@ p1, @ p2, @p3: The strategy vectors of the three competing strategies.

@ currentstate: The current state.

@ rounds: Rounds per match for the simulation part.

@ payoff: Holds the game payoff matrix elements @ R, @ S, @ T, @ P.

@ initprobs: Vector with the probabilities for each strategy to choose C as initial move.

@ alluniquestates: All the unique states.

Use

Performs a meeting (generation) among three competing strategies. Returns the next state in the generation evolution and the total expected average payoff for each strategy, at the end of the meeting. Calls randomplay(), playmatch() and nxt().

Called by

Called by multigen().

Function randomplay()

```
Cprobs = randomplay(rounds, vectorp)
```

@ Cprobs: A 4 x @ rounds matrix with the premade choices, according to the conditional probabilities given in @ vectorp.

@ rounds: Rounds per match for the simulation part.

@ vectorp: The strategy vector of the specific strategy.

Use

Creates the matrix @ Cprobs with the premade choices with the conditional probabilities given in @ vectorp.

Called by

Called by gentournament().

Function playmatch()

```
[XYchoices, matchscores] = playmatch(rounds, firstX, CprobsX,
firstY, CprobsY, payoff)
```

@ XYchoices: A 2 x @ rounds matrix, with the choices of the two players during the match.

@ matchscores: Contains the scores the two players gained after the match.

@ rounds: Rounds per match for the simulation part.

@ firstX: First move of the first player.

@ CprobsX: A 4 x @ rounds matrix with the premade choices for the first player, according to the conditional probabilities of his strategy.

@ firstY: First move of the second player.

@ CprobsY: A 4 x @ rounds matrix with the premade choices for the second player, according to the conditional probabilities of his strategy.

@ payoff: Holds the game payoff matrix elements @ R, @ S, @ T, @ P.

Use

Plays a match between two players, returning their choices and scores.

Called by

Called by gentournament().

Function nxt()

```
nextstate = nxt(S, n, alluniquestates)
```

@ nextstate: Next state in the generation evolution.

@ S: Vector of the total expected average payoffs of the strategies during this generation.

@ n: n = [n1, n2, n3] is the current vector of strategy populations.

@ alluniquestates: All the unique states.

Use

Finds the next state of the generation evolution, given the current state @ n and the vector @ S of the total expected average payoffs of the strategies during this generation. Calls its own realizations of remaintosamestate() and transferfromXtoY().

Called by

Called by gentournament().

Function simGraphs()

```
simGraphs(Msim, Mk, MTheory, statenames, alluniquestates, ngens,
    ↪ N)
```

@ Msim: The simulated one-step state transition matrix.

@ Mk: The simulated k-step (k big) state transition matrix.

@ MTheory: The theoretically calculated one-step state transition matrix.

@ statenames: Vector containing all the unique state names as strings.

@ alluniquestates: All the unique states.

@ ngens: Number of generations.

@ N: Total number of players.

Use

Draws all the graphs for the simulation part. Calls graphPlot().

Called by

Called by `MarkovRun.m`.

Function populationAnimation()

```
populationAnimation(W, strategiesArray, namesofStrategies)
```

`@ W`: Matrix of dimensions 3 x (`@ ngens`+1), with the population evolution history to be animated.

`@ strategiesArray`: Vector containing the indices of the strategies to compete.

`@ namesofStrategies`: Map with the indices and names of all the strategies.

Use

Animation of the time evolution of the strategy populations.

Called by

Called from MATLAB's Command Window.