King's College London

Department of Informatics

Rock Paper Scissors Lizard Spock : An Evolutionary Games Analysis

Raphaël Rolnik (Student no.: 23131461)

Module name: **Agent-Based Modelling**

Module code: **7CCSMAMF**

Submission: **Friday 29th March 2024**

Lecturer: **Dr Maria Polukarov & Dr Katie Bentley**

Contents

List of Figures

1 Introduction and Motivation

We examine the agent-based modelling aspects arising from the well-known game Rock Paper Scissors (RPS). From initially scrutinising the original simple two-person game by using Python Mesa, we delve into the evolutionary dynamics of Rock Paper Scissors (RPS) and its complex extension, Rock Paper Scissors Lizard Spock (RPSLS), by using a spatial model in NetLogo. We aim to comprehend how growing tactics and rules affect the game's equilibrium, cyclic behaviour, and stable strategy evolution. By doing so, our idea is to start by looking at basic game theory and then move on to more complex evolutionary theories, similar to the structure of this module.

We begin by focusing on RPS's simplicity, which highlights the strategies' cyclical supremacy. Next, we investigate RPSLS. This extension leads to a reassessment of dominance patterns and evolutionary stability within the game's framework, in addition to complicating the strategic environment. With our NetLogo simulations, we analyse how these game modifications influence population behaviours, incorporating unique situations like cannibalism to test the durability and flexibility of strategies against evolutionary challenges. This study highlights the fundamental concepts of evolutionary game theory, underpinning the delicate interplay of competitive and cooperative forces within dynamic environments. Our exploration aims to reveal the complex dynamics sparked by simple alterations in rules, providing a deeper understanding of the mechanisms driving evolutionary stability and transformation.

2 Definition of the Game

2.1 Rock Paper Scissors

Generally, RPS is a simple hand game that two people play. The players simultaneously form one of three shapes with an outstretched hand. The possible shapes are rock (a fist), paper (an open hand), and scissors (a fist with the index and middle fingers extended, forming a V). The game has a simple set of rules that determine the winner: rock crushes scissors, scissors cut paper, and paper covers rock. If both players choose the same shape, the game is tied and is usually played again to break the tie.

2.2 Rock Paper Scissors and Lizard Spock

RPSLS is an extension of the traditional Rock Paper Scissors game, designed to reduce the chances of a tie by introducing two additional gestures: Lizard and Spock. This variation was popularized by the TV show 'The Big Bang Theory' and adds a layer of complexity with its expanded set of rules. An overview of how the gestures perform against each other may be seen below:

- Rock crushes Scissors (as before) and crushes Lizard.
- Paper covers Rock (as before) and disproves Spock.
- Scissors cuts Paper (as before) and decapitates Lizard.
- Lizard poisons Spock and eats Paper.
- Spock smashes Scissors and vaporizes Rock.

By implementing these criteria, RPSLS keeps the original game's cyclical structure while providing a greater range of possible outcomes and decreasing the likelihood that players will select the same gesture. With five alternatives available to each participant, the game becomes more complex as each choice wins against two other gestures, loses to two, and ties with none. (Kang et al. 2013)

3 RPS as a Simple Two-Person Game

3.1 Pure and Random Strategies

To implement an initial RPS played by two agents against each other, we utilised the Python Mesa agent-based modelling framework.[1] With the help of customised implementations or built-in core components, Mesa allows easy development of agent-based models, visualisation of them using a browser-based interface, and analysis of the results using Python's data analysis capabilities. Its objective is to be a Python-based substitute for Repast, NetLogo, or MASON. (Kazil, Masad, and Crooks 2020) The main objective of our initial probe here is to understand which repercussion distinguished strategies have on each other, given the absence of a dominant strategy in RPS.

Initially, we implemented a way to run the simulations by enabling each agent to either choose a 'pure' strategy, like always playing rock, as an example or to play a random strategy, where each move is randomly picked. Unsurprisingly, no further information on the game dynamics could be drawn from these simulations, as pure strategies always performed as they should, depending on the opponent's move, and random strategies were indeed random, resulting in approx. 1/3 of the games being tied, 1/3 of the games being won by agent 1, and vice versa, 1/3 of the games being won by player 2. Given that humans face difficulties creating actual random sequences, as mentioned in Gilovich, Vallone, and Tversky (1985), it must be considered that in a real RPS game, the outcome of a random strategy will likely deviate from the observed outcome in our simulation.[2]

3.2 Enhanced Strategies

As a subsequent step, we enhanced the model by incorporating three advanced strategies, namely these are: Beat-Last Move, Win-Stay Lose-Shift (WSLS), and a Tit for Tat variant for RPS, which originates from the context of the famous Prisoner's Dilemma. Beat-Last Move anticipates an opponent's tendency to repeat their last move, selecting the option that would have won against it in the subsequent round. WSLS operates on a simple yet dynamic principle: if a move results in a win (or a tie), it is repeated in the next round; if it results in a loss, a different move is chosen, rewarding success and adapting to failure. Tit for Tat is implemented in its most classical form: it simply mirrors the opponent's last move. This approach is based on the psychological principle of imitation. It presumes that adapting to the opponent's previous action could lead to a stable equilibrium over many rounds, which is a form of mutual cooperation or prediction of repeated behaviour. When these strategies face-off, intriguing dynamics emerge. Beat-Last Move finds an advantage against WSLS by predicting and countering its repetition of winning moves. The best outcome WSLS can obtain is a tie. The interaction between Beat-Last Move and Tit for Tat turns into a dance of action and reaction, with the outcome hinging on the sequence of initial moves. Similarly to WSLS, the best outcome in the long term Tit for Tat can yield is a tie. Moreover, Tit for Tat, with its mirroring strategy, introduces a different dynamic against WSLS. While WSLS tries to establish a winning pattern, Tit for Tat's approach of directly copying the last move can disrupt WSLS's strategy or lead to predictable outcomes that WSLS might exploit. Since in our simulation settings, WSLS exclusively alters its strategy in the event of a loss, the two strategies (in the long term) result in repeating ties, as both strategies neutralise each other.

3.3 Findings from the Simple Two-Players Game

From the simulations conducted, it may be deduced that in any case where no random or pure strategy is played, it might make sense for one agent to stubbornly choose Beat-Last Move as its strategy, given that compared to the other presented strategies. It usually, in the long run, results in at least a tie. However, according to Zhang, Moisan, and Gonzalez (2020) the strategy receiving the most empirical support is WSLS, which directly contrasts our findings. An explanation for this contradiction might be the setting of the specific chosen strategies. Moreover, Zhang, Moisan, and

[1] The source code of our model is given in Appendix A.

[2] A screenshot of the server running 100 steps of two agents playing a random strategy can be seen in Appendix B.

Gonzalez (2020, p. 272) also found out "that humans' behavior are very heterogeneous, and cannot conclusively be described by a commonly claimed heuristic of WSLS." The circumstance that our simulation is carried out by deterministic machines and not by 'stochastic' humans also represents a potential explanation for this contradiction.

Whereas, it is indeed quite interesting that Beat-Last Move[3] outperforms the other non-random strategies. This finding is not 'statistically' robust for the aforementioned reasons. Nevertheless, from a learning perspective, it was appealing to examine an alternative to the well-known NetLogo framework, which is simple to use and comes with some auxiliary features, such as different charting techniques.

4 RPS(LS) as a Spatial Model

Given the boundaries encountered in the simple two-person game, our goal is to explore traditional game dynamics further within a spatially structured environment. Our objective is to go beyond the simple two-agent implementation and probe which dynamics may evolve when many agents face situations of evolutionary competition and strategy adaptation. Further, we want to clarify which insights from evolutionary game theory may be applied to our simulations.

4.1 The Original NetLogo Spatial Model

4.1.1 RPS in NetLogo

The original model works in the following way. Each patch is coloured either red, blue, green or black. A list of swap, select and reproduce events are generated based on the rates given in the UI. These events are randomly shuffled and assigned to one patch each. The patch randomly selects one of its four neighbours (as in a Von Neumann neighbourhood) and applies said event. A swap event changes the colour of the original patch with that of the target. A select event allows the original patch to compete in RPS with the target based on its colour. Whether the original patch beats the target or not is determined in a separate function. If the original patch beats the target, the target's colour will become black, and the opposite happens otherwise. Finally, a reproduction event allows for replication into a black patch. If the target is black, the original patch will be replicated, and otherwise, the opposite will happen.

It is important to note that, at this point, we no longer have in hands a traditional RPS game in which two players with different strategies try to beat each other. Instead, what we have is an evolutionary game with simple mechanics. We will explore it as such.

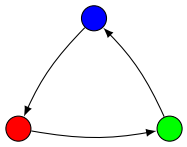


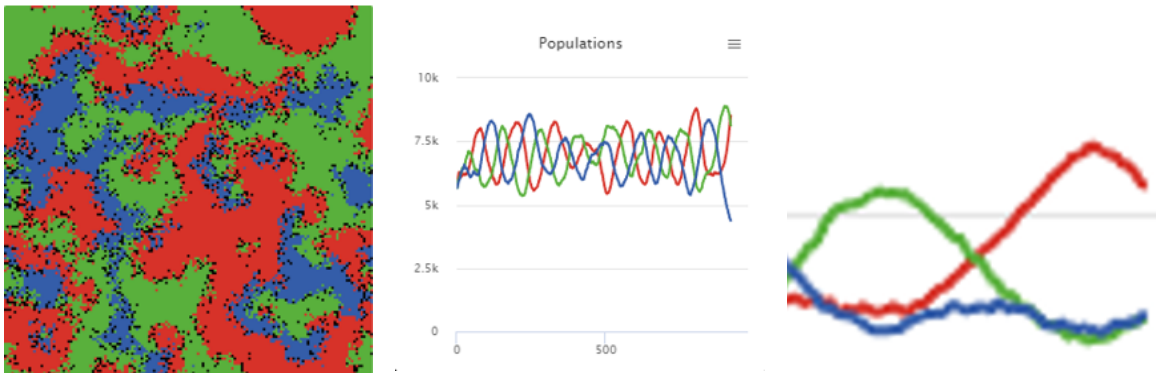
Figure 1: RPS Rules

4.1.2 Normal behaviour

In general, the model exhibits a regular cyclic behaviour among its colour-coded patches, representing different strategies in a Rock Paper Scissors (RPS) framework. On average, each colour — symbolizing rock, paper, or scissors — covers one-third of the board, as illustrated in Figure 2 (a). This distribution demonstrates an evolutionary stable strategy (ESS), where no single strategy dominates permanently. A closer examination of Figure 2 (b) and (c) reveals dynamic interactions: when green, the most populous at its peak, encounters red, its population decreases as red increases, a result of direct competition and game mechanics like reproduction and swapping. However, green's

[3]For at least 1000 simulation steps.

growth is eventually limited by the scarcity of blue patches and its prey, while blue's population remains relatively stable due to minimal engagement. The cyclical pattern — blue to green to red — reflects the inherent randomness of the model, illustrating a complex balance among competing strategies.



(a) Patch Coverage (b) Cyclic Behaviour of Populations (c) Zoomed in Cyclic Behaviour

Figure 2: RPS Analysis with NetLogo

4.2 RPSLS as a Spatial Model

4.2.1 RPSLS in NetLogo

To enhance the existing RPS three-choice model to an RPSLS five-choice model, our code has been significantly enhanced to include the elements of Lizard and Spock, extending beyond the traditional confines of Rock Paper Scissors. Initially confined to a palette of red, blue, green, and black to denote various states, the introduction of yellow and orange brings Lizard and Spock into the fray. In addition to this colour extension, the core function that evaluates the results between competing colours has also been updated. Adapted from the straightforward dynamics of Rock Paper Scissors, this function now navigates a more intricate web of interactions, integrating new rules where Red (Rock) overcomes Orange (Lizard) and Green (Scissors), Yellow (Spock) bests Red (Rock) and Green (Scissors), Orange (Lizard) outmanoeuvres Yellow (Spock) and Blue (Paper), Green (Paper) defeats Orange (Lizard) and Red (Rock), and Blue (Paper) cuts down Yellow (Spock) and Red (Rock). This expansion not only expands the strategic horizon but also adds a new level of complexity and interaction that revitalises the dynamics of the game and player engagement.

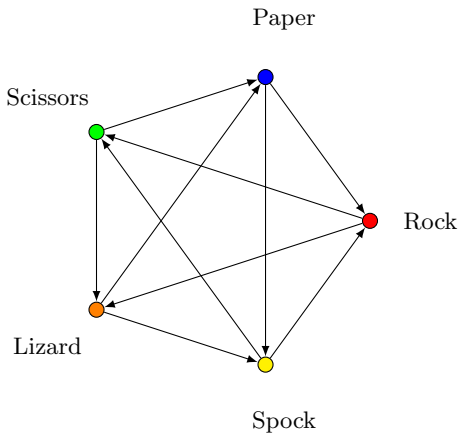


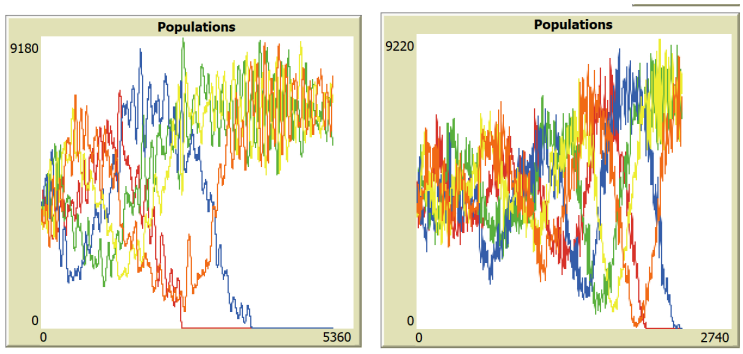
Figure 3: RPSLS Rules

4.2.2 Normal behaviour

In our expanded Rock Paper Scissors Lizard Spock (RPSLS) model, with the game's default settings, from Figure 4 (a), we observe intricate and unpredictable behaviour among the competing colours. The addition of complex rules — where each player can defeat two others but also be defeated by two — introduces a delicate balance. When this equilibrium is disturbed, as seen when a colour like red is eliminated early, a cascading effect on the game's dynamics ensues, leading to a domino effect where the absence of one player destabilises others. Through extensive simulation, a pattern emerges: the sequence of colour eliminations seems influenced by the order of their defeat, suggesting a cyclical nature to the game's balance of power.

This cyclical behaviour highlights a significant aspect of the model's dynamics, where colours take turns dominating and being eliminated. Notably, the size of our simulation grid appears to influence these outcomes, with smaller sizes potentially misrepresenting the dynamics due to limited space for the colours to interact. Despite computational limitations, this cycle of dominance and elimination provides insights into the strategic depth of the RPSLS model.

Further investigation into the model's behaviour with adjusted settings, like a reduced swap rate to 0.5%, confirms the robustness of these dynamics (See Figure 4 (b)). Even under these modified conditions, the cyclic pattern of dominance and elimination persists, underscoring the expanded game model's inherent complexity and intriguing equilibrium.



(a) High Variance (b) Lowered Swap Rate

Figure 4: RPSLS Analysis with NetLogo

4.3 Static Cannibalism

In the realm of evolutionary games, introducing unconventional concepts might lead to intriguing outcomes. Therefore, we experimented with the idea of enabling a form of 'cannibalism' within the game's mechanics to probe if any interesting patterns arise. This modification allows a colour, or 'species,' to 'eat' or dominate itself, potentially transforming internal dynamics. By incorporating a simple line of code that permits self-targeting in the competitive function, we essentially introduced cannibalistic behaviour. This change was hypothesised to induce a survival disadvantage, symbolised by the emergence of black squares within clusters of the same colour, representing internal competition or cannibalism. For now, we implement cannibalism on RPS and not on the enhanced RPSLS; the rationale for this is that this section aims at the introduction of cannibalism between agents rather than the rules of the game.[4]

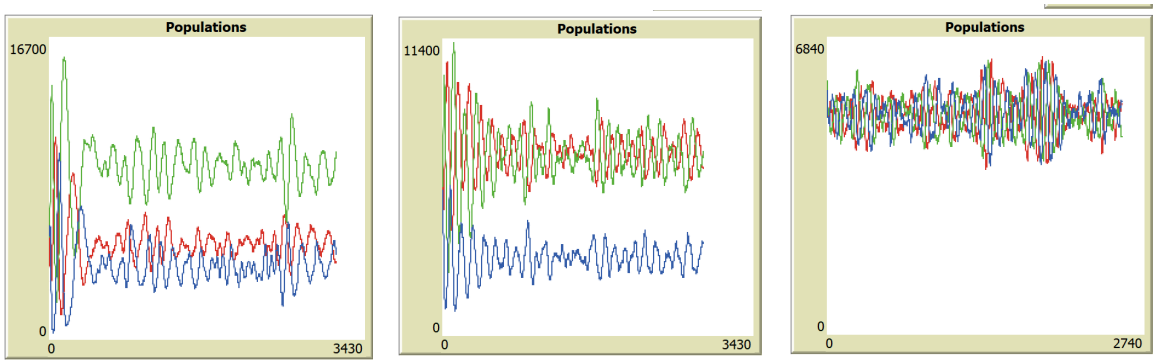
Our observations from Figure 5 (a) disclosed slight effects on the game's ecosystem. Initially, the green population maintained dominance, followed closely by red and blue, which alternated positions. This pattern suggests that cannibalism does not straightforwardly correlate with a decrease

[4]The source code used may be found in Appendix D. The code framework originates from Head, Grider, and Wilensky (2017), yet we altered it to fit our purposes

in population fitness. For instance, red, the first cannibal, did not immediately fall to a disadvantage. Instead, its interaction with green and blue under cannibalistic conditions required a deeper analysis. To decode the dynamics, it is essential to recall the basic food chain within our game: red preys on green, green on blue, and blue on red. When red adopts cannibalism, it inadvertently obstructs its own expansion by consuming potential allies, slowing its assault on the green. This self-limitation provides green, which preys on blue, an edge, as its primary threat is now partially self-occupied. Conversely, blue, red's prey, finds itself in a precarious position, squeezed between a dominant green and a self-consuming red.

Introducing a second cannibal, blue, in Figure 5 (b) further complicated the ecosystem. The equilibrium shifted slightly, bringing red and green to near-parity in population levels, while blue lagged behind. This scenario hinted at a nuanced balance where cannibalism's value is context-dependent, influenced by the presence and behaviour of other cannibals.

In Figure 5 (c), we push our experiment to its limit by introducing cannibalism across all three colours. This ultimate test yielded an ecosystem where no single colour could secure a lasting advantage. The populations entered a state of flux, constantly cycling without a clear dominator. This outcome reinforces the concept that in a balanced system, even seemingly detrimental traits like cannibalism can contribute to dynamic stability.



(a) Red Cannibals (b) Red and Blue Cannibals (c) All Three Cannibals

Figure 5: Cannibalism in RPSLS

4.4 Cannibalism as a Positive Trait

4.4.1 Cannibalism in Nature

In the previous section, we saw that being a single cannibal results in the domination of another colour in terms of population size, while the cannibal colour only slightly beats another colour. We can view this as a negative trait, as another colour gains far more than you, and your population declines, but perhaps there are some cases in which cannibalism could have a positive impact. This will be examined in the following.

In nature, there are examples of species, from single-celled organisms to mammals, where cannibalism is a trait aiding in areas such as longevity, size of offspring, etc. A paper by Cushing, Henson, and Hayward (2015) explored this idea with a two-stage population model motivated by an example from colonial seabirds, where a lack of resources resulted in an increase in egg cannibalism. Their model predicts that if adult survival is going towards extinction, the mean level of cannibalism is raised such that the population will not go extinct.

We will attempt to reproduce such results, even with our much cruder RPSLS evolutionary model, but first, a description of the setting is required.

4.4.2 Setting

The need for cannibalism can come from times of stress that cause the population to decline. In the model of Cushing, Henson, and Hayward (2015), food resource availability is the stress factor which harms the population of seabirds. However, in our model, we create a scenario where blue is not a strong strategy to play, 'forcing' it to extinction. We will see this in the Experimentation part.

Now, to allow the cannibalism trait to develop due to some stress, they use bifurcation theory to determine whether the extinction equilibrium suggests that the population is unstable and heading to collapse. Whereas we simply enable cannibalism when the population drops by a certain percentage from its starting value, in the scenario of certain extinction described in the above paragraph.

Their two-stage model allows an adult subset of the population to feed on a youngling subset to give the adults more energy and, therefore, longevity. Our model will use the method of Blue beating Blue, resulting in having more black squares surrounding their population as a buffer and allowing them to cling to their own 'food'(Yellow), thus increasing longevity. As displayed in Figure 6 (a), the blue patch has far more black squares, acting as a buffer for Orange, which is trying to eat it. Also, it can cling to some yellow patches to its north.[5]

4.4.3 Experimentation

Blue to Extinction

Having the original rules for every other tick, and in the other half, Red not being able to beat Green, forces Blue to extinction in all runs of our model so far. The reasoning behind this is that 50% of the time, the Red beats Green rule is removed, which allows for Green to have an advantage in population size, and Green beats Blue. For example, the phenomenon described can be seen in the zoomed-in section in Figure 6 (c). Moreover, it can be inferred that the increase in the Green population is correlated with a sudden decline in the Blue population. Yet, with five colours involved, deducing outcomes becomes significantly more complex.

Blue's Survival with Cannibalism

Now, with cannibalism enabled, if Blue's population is threatened, they are able to 'fight off' extinction by increasing its longevity through cannibalism. Figure 6 (d) contains an instance of Blue holding out long enough until the dynamics change enough for their population to increase again. After many ticks have passed, in Figure 6 (e), we see a new stable strategy form that resembles the Evolutionary Stable Strategy (ESS) in the original RPS evolutionary game. The reason for Yellow's extinction is that Yellow beats Green while Green has become extinct previously, with Yellow having one less colour to beat, it is dominated by the other colours which are now back to normal. In the next section, we will see how often extinction is prevented.

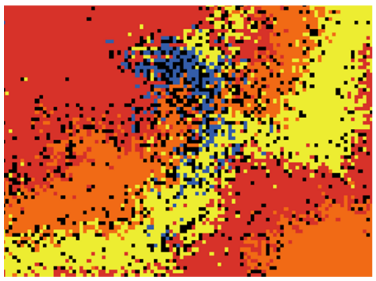
4.4.4 Results

We have seen that cannibalism can be a positive trait in our evolutionary model in circumstances where a population is pushed to extinction. Let's take a closer look at how close to extinction you should be to activate your cannibalistic property.

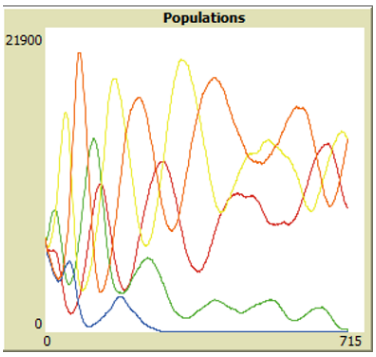
It usually takes at least 4000 ticks for a stable strategy to form. In Figure 7, we see the chance of Blue's population avoiding extinction based on the size of the population at which Blue become cannibals. In the base case, where Blue does not become a cannibal, the survival chance is 0% by the design of the game. However, there seems to be an optimal population size for Blue to become cannibals, maximising their survival rate, somewhere around a population size of 1000; this can be explored further with a 1-dimensional optimisation problem and Monte-Carlo simulations.

Becoming cannibals too early, with a population exceeding 1000, seems to damage Blue's chances; as we saw before in the Static Cannibalism section, being a cannibal results in a lower population

[5]Note: When Blue again reaches a healthy population over the specified percentage of the initial, it stops being a cannibal.

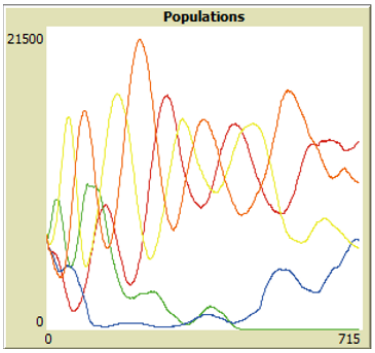


(a) Blue Cannibal Patch

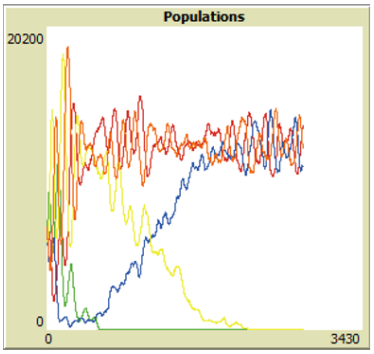


(b) Blue Going Extinct

(c) Blue to Extinction dynamics



(d) Blue Longevity



(e) New ESS Post Blue Longevity

Figure 6: Positive Cannibalism

size. Additionally, becoming cannibals when your population is already quite low is not optimal either, as Blue is already being attacked so much that cannibalism provides little fighting chance to avoid extinction.[6] Their trajectory to extinction is too steep to manage.

Some limitation to consider is that our game-playing lattice is only 100x100 due to computational limitations; other research suggests a usage of 1000x1000 (Szolnoki and Perc 2016) as there is a big influence on where groups of colours form on the board (Yang and Park 2023). Furthermore, we only ran 20 simulations for each population size above. These are far too few, but this could be solved with the help of Monte-Carlo-style simulations. We should also note that our model does not portray any real-world examples and only merely resembles that of Cushing, Henson, and Hayward (2015).

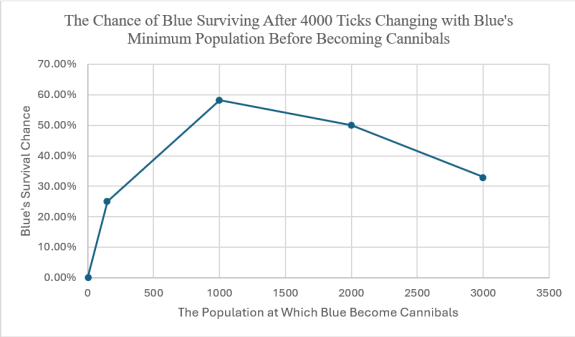


Figure 7: Optimal Population to Become Cannibal

[6] See Figure 7.

5 Conclusion

Our academic 'journey' began with a fundamental exploration of simple game theory and two-player strategies within the traditional framework of RPS. This initial phase served as a crucial stepping stone, allowing us to grapple with the basics of competitive dynamics and strategy optimisation. A key point was that, in theory, the best strategy for humans would be a random strategy. However, humans are not versed in creating random sequences, which diminishes the success rate of playing a random strategy. Moreover, the analysis disclosed that for our settings Beat-Last Move outperformed Tit for Tat or WSLS. The simplicity of RPS, combined with agent-based modelling techniques using Python Mesa, provided us with a solid foundation, yet it also presented us with a steep learning curve as we transitioned to the more complex evolutionary algorithms required for the expanded game of RPSLS explored through NetLogo.

The extension to a spatial evolutionary game and the incorporation of the new RPSLS rules marked a significant shift in our approach. We moved from studying traditional game theory to testing out the unpredictable dynamics of an agent-based model. We began by analysing normal behaviour and then incorporating the idea of cannibalism and its impact on new strategies and behaviours that could alter population dynamics and equilibrium states. Then, search through the literature and find an interesting theory to model. Our understanding tackled not only technical but also conceptual aspects of evolutionary game theory, challenging us to explore notions of dominance, survival, and stability.

Conclusively, this paper not only illustrates our methodological progression from basic game theory to sophisticated evolutionary simulations but also shows our academic journey. We transitioned from understanding simple deterministic outcomes in two-player scenarios to appreciating the complexity and unpredictability of evolutionary strategies in a spatially structured environment. Our exploration into RPSLS has illuminated the intricate relationship between competition and cooperation, strategy and counterstrategy, providing valuable insights into the mechanisms that drive the evolution of complex systems.

Bibliography

Cushing, J. M., S. M. Henson, and J. L. Hayward (2015). "An Evolutionary Game-Theoretic Model of Cannibalism". In: *Natural Resource Modeling* 28.4, pp. 497–521. DOI: 10.1111/nrm.12079. URL: https://doi.org/10.1111/nrm.12079.

Gilovich, Thomas, Robert Vallone, and Amos Tversky (1985). "The hot hand in basketball: On the misperception of random sequences". In: *Cognitive Psychology* 17, pp. 295–314.

Head, B., R. Grider, and U. Wilensky (2017). *NetLogo Rock Paper Scissors model*. http://ccl.northwestern.edu/netlogo/models/RockPaperScissors. Accessed: 2024-03-21. Evanston, IL.

Kang, Yibin et al. (2013). "A golden point rule in rock–paper–scissors–lizard–spock game". In: *Physica A: Statistical Mechanics and its Applications* 392.11, pp. 2652–2659. ISSN: 0378-4371. DOI: https://doi.org/10.1016/j.physa.2012.10.011. URL: https://www.sciencedirect.com/science/article/pii/S0378437112008916.

Kazil, Jackie, David Masad, and Andrew Crooks (2020). "Utilizing Python for Agent-Based Modeling: The Mesa Framework". In: *Social, Cultural, and Behavioral Modeling*. Ed. by Robert Thomson et al. Cham: Springer International Publishing, pp. 308–317. ISBN: 978-3-030-61255-9.

Szolnoki, Attila and Matjaž Perc (2016). "Competition of tolerant strategies in the spatial public goods game". In: *New Journal of Physics* 18.8, p. 083021. DOI: 10.1088/1367-2630/18/8/083021. URL: https://iopscience.iop.org/article/10.1088/1367-2630/18/8/083021.

Yang, Ryoo Kyung and Junpyo Park (2023). "Evolutionary dynamics in the cyclic competition system of seven species: Common cascading dynamics in biodiversity". In: *Chaos, Solitons and Fractals*. DOI: https://doi.org/10.1016/j.chaos.2023.112603. URL: https://www.sciencedirect.com/science/article/pii/S0960077923008500?fr=RR-2&ref=pdf_download&rr=86439d4d9cd263b1.

Zhang, Hanshu, Frederic Moisan, and Cleotilde Gonzalez (2020). "Paper-Rock-Scissors: an exploration of the dynamics of players' strategies". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 64.1. doi: 10.1177/1071181320641063, pp. 268–272. ISSN: 1071-1813. DOI: 10.1177/1071181320641063. URL: https://journals.sagepub.com/doi/abs/10.1177/1071181320641063.

Appendix

Appendix A

Listing 1: Two Player RPS in Python Mesa

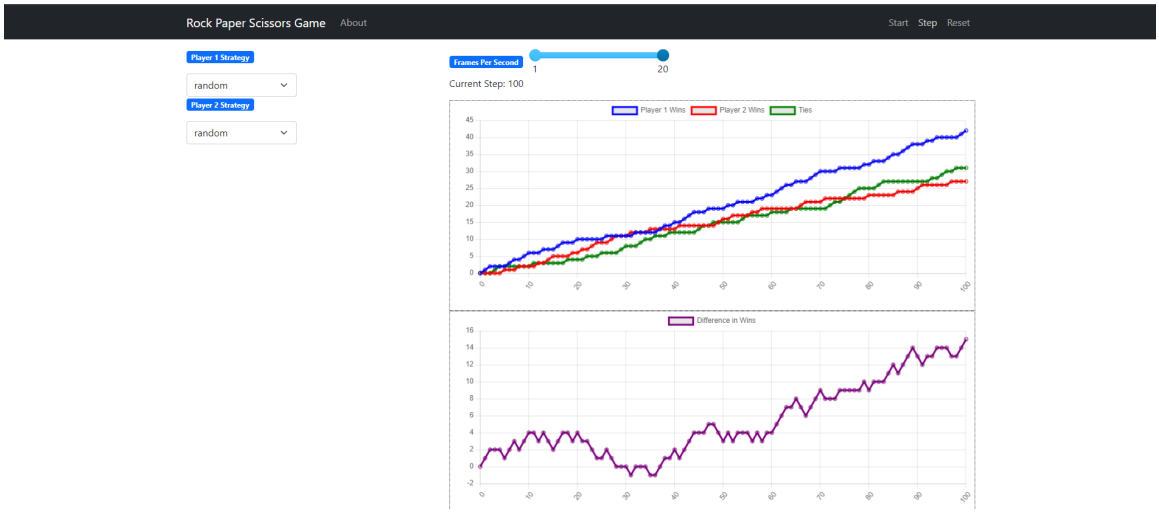
```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.datacollection import DataCollector
from mesa.visualization.modules import ChartModule, PieChartModule
from mesa.visualization.ModularVisualization import ModularServer
from mesa.visualization import Choice
import random

# Names the 3 pure strategies of RPS
CHOICES = ["rock", "paper", "scissors"]

# Defines the strategies including the pure strategies
STRATEGIES = ["random", "tit-for-tat", "beat-last-move", "win-stay-lose-
    ↪ shift"] + CHOICES

# Determines the winner of each round "step"
def determine_winner(player1_choice, player2_choice):
    if player1_choice == player2_choice:
        return 0  # Tie
    elif (player1_choice == "rock" and player2_choice == "scissors") or
        ↪ \
         (player1_choice == "scissors" and player2_choice == "paper") or
             ↪  \
         (player1_choice == "paper" and player2_choice == "rock"):
        return 1  # Player 1 wins
    else:
        return 2  # Player 2 wins

#Returns the choice that beats the given move of the previous round
def beat_last_move(move):

    if move == "rock":
        return "paper"

    elif move == "paper":
        return "scissors"

    elif move == "scissors":
        return "rock"

# Class of the agents
class PlayerAgent(Agent):
    def __init__(self, unique_id, model, strategy="random"): # Sets
        ↪ random as the default strategy on the server
        super().__init__(unique_id, model)
        self.wins = 0  # Track the number of wins
        self.strategy = strategy
        self.choice = None
        self.last_choice = None
```

```
        self.last_result = None  # Track the outcome of the last round (
            ↪ win, lose, tie)
        self.choice_history = []

    # Determines which move to make, based on the strategy
    def decide(self):
        # Determines who is the opponent
        opponent = self.model.player1 if self.unique_id == 2 else self.
            ↪ model.player2

        if self.strategy == "win-stay-lose-shift":
            # If this is the first round, or if the agent won or tied
                ↪ the last round, it stays with its last choice (if any)
            if self.last_result in [None, 1, 0]:
                self.choice = self.last_choice if self.last_choice else
                    ↪ random.choice(CHOICES)
            else:  # If the agent lost the last round, it shifts to the
                ↪ next choice
                next_index = (CHOICES.index(self.last_choice) + 1) % len
                    ↪ (CHOICES)
                self.choice = CHOICES[next_index]

        elif self.strategy == "tit-for-tat":
            # For the first round, or if no history, choose randomly
            if not opponent.choice_history:
                self.choice = random.choice(CHOICES)
            else:
                # Mimic the opponent's last round choice
                self.choice = opponent.choice_history[-1]

        elif self.strategy == "beat-last-move":
            # For the first round, or if no history, choose randomly
            if not opponent.choice_history:
                self.choice = random.choice(CHOICES)
            else:
                # Choose what would have beaten the opponent's last
                    ↪ round choice
                self.choice = beat_last_move(opponent.choice_history
                    ↪ [-1])

        elif self.strategy == "random":
            # For random strategy choices
            self.choice = random.choice(CHOICES)

        else:
            # For pure strategy choices
            self.choice = self.strategy

    # This function keeps track on the choices of each agent
    def finalize_choice(self):
        self.last_choice = self.choice
        self.choice_history.append(self.choice)

# Class for the ABM
```

```
class RPSModel(Model):

    def __init__(self, player1_strategy="random", player2_strategy="
        ↪ random"):
        super().__init__()
        self.schedule = RandomActivation(self)
        self.player1 = PlayerAgent(1, self, strategy=player1_strategy)
        self.player2 = PlayerAgent(2, self, strategy=player2_strategy)
        self.schedule.add(self.player1)
        self.schedule.add(self.player2)
        self.ties = 0

        # Data collector for the server graphs
        self.datacollector = DataCollector(
            {
                "Player 1 Wins": lambda m: m.player1.wins,
                "Player 2 Wins": lambda m: m.player2.wins,
                "Difference in Wins": lambda m: m.player1.wins - m.
                    ↪ player2.wins,
                "Ties": lambda m: m.ties,
            }
        )

    # Increment "step" function
    def step(self):

        # First, all agents decide their move
        self.player1.decide()
        self.player2.decide()

        # Determine the winner based on current decisions
        winner = determine_winner(self.player1.choice, self.player2.
            ↪ choice)

        # Update wins and the last_result for each player based on the
            ↪ winner
        if winner == 1:
            self.player1.wins += 1
            self.player1.last_result = 1
            self.player2.last_result = 2
        elif winner == 2:
            self.player2.wins += 1
            self.player1.last_result = 2
            self.player2.last_result = 1
        else:
            self.ties += 1
            self.player1.last_result = 0
            self.player2.last_result = 0

        # After deciding and resolving the round, finalise choices for
            ↪ history
        self.player1.finalize_choice()
        self.player2.finalize_choice()
```

```
        # Collect data after all actions are finalised
        self.datacollector.collect(self)

        # Log the choices to debug
        print(f"Round {self.schedule.steps}: Player 1 chose {self.
            ↪ player1.choice}, Player 2 chose {self.player2.choice}")

# Creates the choice dropdown fields on the LHS
model_params = {
    "player1_strategy": Choice(
        name="Player 1 Strategy",
        value="random",
        choices=STRATEGIES,
        description="Select Player 1's strategy",
    ),
    "player2_strategy": Choice(
        name="Player 2 Strategy",
        value="random",
        choices=STRATEGIES,
        description="Select Player 2's strategy",
    ),
}

# Creates the server containing the graphs
server = ModularServer(
    RPSModel,
    [
    ChartModule([{"Label": "Player 1 Wins", "Color": "Blue"},
                 {"Label": "Player 2 Wins", "Color": "Red"},
                 {"Label": "Ties", "Color": "Green"}],
                data_collector_name='datacollector'),
    ChartModule([{"Label": "Difference in Wins", "Color": "Purple"}],
                data_collector_name='datacollector'),
    PieChartModule([{"Label": "Player 1 Wins", "Color": "Blue"},
                 {"Label": "Player 2 Wins", "Color": "Red"},
                 {"Label": "Ties", "Color": "Green"}],
                data_collector_name='datacollector')
    ],
    "Rock Paper Scissors Game",
    model_params
)

# Ensure this port is free or change it to an available one
server.port = 8521

# Runs the server locally
server.launch()
```

Appendix B



Appendix C

Listing 2: Positive Cannibalism in NetLogo

```
globals [
  blue-patch-count
]


to setup
  clear-all
  ask patches [
    ; Start populations at roughly even levels.

    set blue-patch-count 0 ;

    set pcolor one-of [ red green blue orange yellow black ]

    if pcolor = blue [ ; Check if the patch is blue
      set blue-patch-count blue-patch-count + 1 ; Increment the count if the
          ↪  patch is blue
    ]

  ]
  reset-ticks
end

to go
  ; This model uses an event-based approach. It calculates how many events
      ↪ of each type
  ; should occur each tick and then executes those events in a random order.
      ↪  Event type
  ; could be signified by any constant. Here, we use a the numbers 0, 1, and
      ↪  2 to signify
  ; event type, but then store those numbers in variables with clear names
      ↪ for readability.
  let swap-event 0
  let reproduce-event 1
```

```
  let select-event 2


  set blue-patch-count count patches with [ pcolor = blue ]


  ; Note that we have to compute the number of global events rather than the
      ↪  number of
  ; actions that each individual patch performs since the execution of those
      ↪  events has
  ; to be random between the patches. That is, a single patch can't perform
      ↪ all of their
  ; actions in one go: suppose they end up executing 5 swaps in one tick.
      ↪ The swaps would
  ; be shuffling things around locally rather than allowing for an organism
      ↪ to travel
  ; multiple steps.
  ; Hence, this code creates a list with an entry for each event type that
      ↪ should occur
  ; this tick, then shuffles that list so that the events are in a random
      ↪ order. We then
  ; iterate through the list, and random neighboring patches run the
      ↪ corresponding event.
  let repetitions count patches / 3 ; At default settings, there will be an
      ↪ average of 1 event per patch.
  let events shuffle (sentence
    n-values random-poisson (repetitions * swap-rate)      [ swap-event ]
    n-values random-poisson (repetitions * reproduce-rate) [ reproduce-event
        ↪  ]
    n-values random-poisson (repetitions * select-rate)    [ select-event ]
  )

  foreach events [ event ->
    ask one-of patches [
      let target one-of neighbors4
      if event = swap-event      [ swap target ]
      if event = reproduce-event [ reproduce target ]
      if event = select-event    [ select target ]
    ]
  ]
  tick
end


; Raph Trying to add color swapping
;

; Determine whether or not I beat TARGET

to-report beat? [ target ]


  ;let bluePatchCount count-patches-with-color blue


  if blue-patch-count < 1500 [            ; CASE IF BLUE POPULATION IS DYING
      ↪  !!!!!!
```

```
  report (pcolor = red and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = blue) or
         (pcolor = blue  and [ pcolor ] of target = red) or
         (pcolor = red and [ pcolor ] of target = orange) or
         (pcolor = yellow and [ pcolor ] of target = red) or
         (pcolor = orange and [ pcolor ] of target = yellow) or
         (pcolor = yellow and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = orange) or
         (pcolor = orange and [ pcolor ] of target = blue) or
         (pcolor = blue and [ pcolor ] of target = yellow) or

         (pcolor = blue and [ pcolor ] of target = blue)

]
ifelse blue-patch-count >= 1500 [         ; CASE IF BLUE IS HEALTHY
   ↪ POPULATION


         let random-number random-float 1

           ifelse random-number < 0.5
           [
               ; Code to be executed when the condition is true

            report ;(pcolor = red and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = blue) or
         (pcolor = blue  and [ pcolor ] of target = red) or
         (pcolor = red and [ pcolor ] of target = orange) or
         (pcolor = yellow and [ pcolor ] of target = red) or
         (pcolor = orange and [ pcolor ] of target = yellow) or
         (pcolor = yellow and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = orange) or
         (pcolor = orange and [ pcolor ] of target = blue) or
         (pcolor = blue and [ pcolor ] of target = yellow)



           ]
           [
               ; Code to be executed when the condition is false
                report (pcolor = red and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = blue) or
         (pcolor = blue  and [ pcolor ] of target = red) or
         (pcolor = red and [ pcolor ] of target = orange) or
         (pcolor = yellow and [ pcolor ] of target = red) or
         (pcolor = orange and [ pcolor ] of target = yellow) or
         (pcolor = yellow and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = orange) or
         (pcolor = orange and [ pcolor ] of target = blue) or
         (pcolor = blue and [ pcolor ] of target = yellow)

       ]


        ;(pcolor = blue and [ pcolor ] of target = blue)

 ]
 [
```

```
    ; Handle the case where ticks < 150 but there's no else block, you can
        ↪ add specific behavior here if needed
  ]









end









;



; Patch procedures

; Swap PCOLOR with TARGET.
to swap [ target ]
  let old-color pcolor
  set pcolor [ pcolor ] of target
  ask target [ set pcolor old-color ]
end

; Compete with TARGET. The loser becomes blank.
to select [ target ]
  ifelse beat? target [
    ask target [ set pcolor black ]
  ] [
    if [ beat? myself ] of target [
      set pcolor black
    ]
  ]
end

; If TARGET is blank, reproduce on that patch. If I'm blank, TARGET
   ↪ reproduces on my patch.
to reproduce [ target ]
  ifelse [ pcolor ] of target = black [
    ask target [
      set pcolor [ pcolor ] of myself
    ]
  ] [
    if pcolor = black [
      set pcolor [ pcolor ] of target
    ]
  ]
end



; Determine whether or not I beat TARGET
;to-report beat? [ target ]
```

```
;  report (pcolor = red   and [ pcolor ] of target = green) or
;         (pcolor = green and [ pcolor ] of target = blue) or
;         (pcolor = red and [ pcolor ] of target = red)
;end


; Utility procedures

to-report rate-from-exponent [ exponent ]
  report 10 ^ exponent
end

to-report swap-rate
  report rate-from-exponent swap-rate-exponent
end

to-report reproduce-rate
  report rate-from-exponent reproduce-rate-exponent
end

to-report select-rate
  report rate-from-exponent select-rate-exponent
end

; Convert the given rate to a percentage of how much that action happens
to-report percentage [ rate ]
  report 100 * rate / (swap-rate + reproduce-rate + select-rate)
end




; Counting patches population


to-report count-patches-with-color [col]
  report count patches with [pcolor = col]
end
```

Appendix D

Listing 3: Static Cannibalism in NetLogo

```
Taken from Netlogo:



to setup
  clear-all
  ask patches [
    ; Start populations at roughly even levels.
    set pcolor one-of [ red green blue black ]
  ]
  reset-ticks
end

to go
```

```
  ; This model uses an event-based approach. It calculates how many events
    ↪ of each type
  ; should occur each tick and then executes those events in a random order.
    ↪  Event type
  ; could be signified by any constant. Here, we use a the numbers 0, 1, and
    ↪  2 to signify
  ; event type, but then store those numbers in variables with clear names
    ↪ for readability.
  let swap-event 0
  let reproduce-event 1
  let select-event 2

  ; Note that we have to compute the number of global events rather than the
    ↪  number of
  ; actions that each individual patch performs since the execution of those
    ↪  events has
  ; to be random between the patches. That is, a single patch can't perform
    ↪ all of their
  ; actions in one go: suppose they end up executing 5 swaps in one tick.
    ↪ The swaps would
  ; be shuffling things around locally rather than allowing for an organism
    ↪ to travel
  ; multiple steps.
  ; Hence, this code creates a list with an entry for each event type that
    ↪ should occur
  ; this tick, then shuffles that list so that the events are in a random
    ↪ order. We then
  ; iterate through the list, and random neighboring patches run the
    ↪ corresponding event.
  let repetitions count patches / 3 ; At default settings, there will be an
    ↪ average of 1 event per patch.
  let events shuffle (sentence
    n-values random-poisson (repetitions * swap-rate)      [ swap-event ]
    n-values random-poisson (repetitions * reproduce-rate) [ reproduce-event
      ↪  ]
    n-values random-poisson (repetitions * select-rate)    [ select-event ]
  )

  foreach events [ event ->
    ask one-of patches [
      let target one-of neighbors4
      if event = swap-event      [ swap target ]
      if event = reproduce-event [ reproduce target ]
      if event = select-event    [ select target ]
    ]
  ]
  tick
end

; Patch procedures

; Swap PCOLOR with TARGET.
to swap [ target ]
  let old-color pcolor
  set pcolor [ pcolor ] of target
  ask target [ set pcolor old-color ]
end

; Compete with TARGET. The loser becomes blank.
```

```
to select [ target ]
  ifelse beat? target [
    ask target [ set pcolor black ]
  ] [
    if [ beat? myself ] of target [
      set pcolor black
    ]
  ]
end

; If TARGET is blank, reproduce on that patch. If I'm blank, TARGET
  ↪ reproduces on my patch.
to reproduce [ target ]
  ifelse [ pcolor ] of target = black [
    ask target [
      set pcolor [ pcolor ] of myself
    ]
  ] [
    if pcolor = black [
      set pcolor [ pcolor ] of target
    ]
  ]
end

; Determine whether or not I beat TARGET
to-report beat? [ target ]
  report (pcolor = red   and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = blue) or
         (pcolor = blue  and [ pcolor ] of target = red)
end

; Utility procedures

to-report rate-from-exponent [ exponent ]
  report 10 ^ exponent
end

to-report swap-rate
  report rate-from-exponent swap-rate-exponent
end

to-report reproduce-rate
  report rate-from-exponent reproduce-rate-exponent
end

to-report select-rate
  report rate-from-exponent select-rate-exponent
end

; Convert the given rate to a percentage of how much that action happens
to-report percentage [ rate ]
  report 100 * rate / (swap-rate + reproduce-rate + select-rate)
end


; Copyright 2017 Uri Wilensky.
; See Info tab for full copyright and license.
@#$#@#$#@
GRAPHICS-WINDOW
```

```
310
10
771
472
-1
-1
3.0
1
10
1
1
1
0
1
1
1
-75
75
-75
75
1
1
1
ticks
30.0

BUTTON
5
10
100
43
NIL
setup
NIL
1
T
OBSERVER
NIL
NIL
NIL
NIL
1

BUTTON
115
10
210
43
NIL
go\n
T
1
T
OBSERVER
NIL
NIL
NIL
NIL
0
```

```
SLIDER
5
55
210
88
swap-rate-exponent
swap-rate-exponent
-1
1
0.0
0.1
1
NIL
HORIZONTAL

SLIDER
5
100
210
133
reproduce-rate-exponent
reproduce-rate-exponent
-1
1
0.0
0.1
1
NIL
HORIZONTAL

SLIDER
5
145
210
178
select-rate-exponent
select-rate-exponent
-1
1
0.0
0.1
1
NIL
HORIZONTAL

PLOT
5
190
305
470
Populations
NIL
NIL
0.0
10.0
0.0
10.0
true
```

```
false
"" ""
PENS
"default" 1.0 0 -2674135 true "" "plot count patches with [ pcolor = red ]"
"pen-1" 1.0 0 -10899396 true "" "plot count patches with [ pcolor = green ]"
"pen-2" 1.0 0 -13345367 true "" "plot count patches with [ pcolor = blue ]"

MONITOR
215
50
305
95
swap-%
percentage swap-rate
2
1
11

MONITOR
215
95
305
140
reproduce-%
percentage reproduce-rate
2
1
11

MONITOR
215
140
305
185
select-%
percentage select-rate
2
1
11

@#$#@#$#@
## WHAT IS IT?

This model explores the role of movement and space in a three species
    ↪ ecosystem. The system consists of three species, represented by red
    ↪ patches, green patches, and blue patches, which compete over space.
    ↪ The interactions between the species are based on the game Rock-Paper-
    ↪ Scissors. That is, red beats green, green beats blue, and blue beats
    ↪ red. Organisms compete with their neighbors, move throughout the
    ↪ environment, and reproduce. These interactions result in spiral
    ↪ patterns whose size and stability depends on the movement rate of the
    ↪ organisms.

The model is written in an event-based fashion, to reflect the formulation
    ↪ of the published model. See HOW IT WORKS and EXTENDING THE MODEL.

## HOW IT WORKS
```

```
Each patch can be occupied by one of three species or can be blank. The
    ↪ species are represented by three colors: red, green, and blue. Each
    ↪ tick, the following types of events happen at defined average rates:

- Select event: Two random neighbors compete with each other. In competition
    ↪ , red beats green, green beats blue, and blue beats red, like in rock
    ↪ paper scissors. The losing patch becomes blanks.
- Reproduce event: Two random neighbors attempt to reproduce. If one of the
    ↪ neighbors is blank, it acquires the color of the other. Nothing
    ↪ happens if neither neighbor is blank.
- Swap event: Two random neighbors swap color. This represents the organisms
    ↪  moving.

The exact number of, for instance, swap events that occur each tick is drawn
    ↪  from a [Poisson distribution](https://en.wikipedia.org/wiki/
    ↪ Poisson_distribution) with mean equal to `(count patches) * 10 ^ swap-
    ↪ rate-exponent`. A Poisson distribution defines how many times a
    ↪ particular event occurs given an average rate for that event assuming
    ↪ that the occurrences of that event are independent. Here, the
    ↪ occurrences of the events are approximately independent since they're
    ↪ being performed by different organisms.

The events occur in a random order involving random pairs of neighbors.

## HOW TO USE IT

Press SETUP to initialize the model and GO to run it.

SWAP-RATE-EXPONENT, REPRODUCE-RATE-EXPONENT, and SELECT-RATE-EXPONENT each
    ↪ control the rate at which their respective actions are performed.
    ↪ There will be an average of `count patches * 10 ^ rate-exponent`
    ↪ events each tick for each event type. This means that increasing a
    ↪ slider by `1.0` will result in that event type occurring 10 times more
    ↪  often, no matter what the other sliders are set to. The SWAP-%,
    ↪ REPRODUCE-%, and SELECT-% monitors indicate what percentage of events
    ↪ will be swap, reproduce, and select events (respectively) each tick.

The POPULATIONS plot shows how much of each organism there is over time.

## THINGS TO NOTICE

Running the model quickly results in a collection of interconnected spirals
    ↪ in which each species is chasing another species.

Global population levels of each of the species oscillate over time.

## THINGS TO TRY

- What happens as you increase SWAP-RATE-EXPONENT? What happens to the shape
    ↪  and size of the spirals? Why does this happen?
- What happens as you decrease SWAP-RATE-EXPONENT?
- Can you find parameter settings that result in the extinction of one of
    ↪ the species? What happens to the other two species?

## EXTENDING THE MODEL

- Try generalizing the model to more than three species.
- The model is written in an event-based manner. Try rewriting it in a more
    ↪ idiomatic agent-based way.
```

```
## NETLOGO FEATURES

The model makes heavy use of the ‘random-poisson‘ primitive. This primitive
    ↪ is useful when modeling events that happen at various rates.
    ↪ Furthermore, this model uses a technique wherein a list of events is
    ↪ produced and shuffled to simulate the occurrence of each event at each
    ↪  rate while still allowing the events to occur in arbitrary orders.

## RELATED MODELS

Wolf Sheep Predation shows a simpler predator prey model.

## CREDITS AND REFERENCES

Reichenbach, T., Mobilia, M., & Frey, E. (2008). Self-organization of mobile
    ↪  populations in cyclic competition. Journal of Theoretical Biology,
    ↪ 254(2), 368  383  . https://www.sciencedirect.com/science/article/pii/
    ↪ S0022519308002464?via%3Dihub

Reichenbach, T., Mobilia, M., & Frey, E. (2007). Mobility promotes and
    ↪ jeopardizes biodiversity in rock-paper-scissors games. Nature,
    ↪ 448(7157), 1046  1049  . https://doi.org/10.1038/nature06095

## HOW TO CITE

If you mention this model or the NetLogo software in a publication, we ask
    ↪ that you include the citations below.

For the model itself:

* Head, B., Grider, R. and Wilensky, U. (2017).  NetLogo Rock Paper Scissors
    ↪  model.  http://ccl.northwestern.edu/netlogo/models/RockPaperScissors.
    ↪   Center for Connected Learning and Computer-Based Modeling,
    ↪ Northwestern University, Evanston, IL.

Please cite the NetLogo software as:

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

## COPYRIGHT AND LICENSE

Copyright 2017 Uri Wilensky.

![CC BY-NC-SA 3.0](http://ccl.northwestern.edu/images/creativecommons/byncsa
    ↪ .png)

This work is licensed under the Creative Commons Attribution-NonCommercial-
    ↪ ShareAlike 3.0 License.  To view a copy of this license, visit https
    ↪ ://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to
    ↪ Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305,
    ↪ USA.

Commercial licenses are also available. To inquire about commercial licenses
    ↪ , please contact Uri Wilensky at uri@northwestern.edu.

<!-- 2017 Cite: Head, B., Grider, R. -->
```

```
@#$#@#$#@
default
true
0
Polygon -7500403 true true 150 5 40 250 150 205 260 250

airplane
true
0
Polygon -7500403 true true 150 0 135 15 120 60 120 105 15 165 15 195 120 180
    ↪  135 240 105 270 120 285 150 270 180 285 210 270 165 240 180 180 285
    ↪ 195 285 165 180 105 180 60 165 15

arrow
true
0
Polygon -7500403 true true 150 0 0 150 105 150 105 293 195 293 195 150 300
    ↪ 150

box
false
0
Polygon -7500403 true true 150 285 285 225 285 75 150 135
Polygon -7500403 true true 150 135 15 75 150 15 285 75
Polygon -7500403 true true 15 75 15 225 150 285 150 135
Line -16777216 false 150 285 150 135
Line -16777216 false 150 135 15 75
Line -16777216 false 150 135 285 75

bug
true
0
Circle -7500403 true true 96 182 108
Circle -7500403 true true 110 127 80
Circle -7500403 true true 110 75 80
Line -7500403 true 150 100 80 30
Line -7500403 true 150 100 220 30

butterfly
true
0
Polygon -7500403 true true 150 165 209 199 225 225 225 255 195 270 165 255
    ↪ 150 240
Polygon -7500403 true true 150 165 89 198 75 225 75 255 105 270 135 255 150
    ↪ 240
Polygon -7500403 true true 139 148 100 105 55 90 25 90 10 105 10 135 25 180
    ↪ 40 195 85 194 139 163
Polygon -7500403 true true 162 150 200 105 245 90 275 90 290 105 290 135 275
    ↪  180 260 195 215 195 162 165
Polygon -16777216 true false 150 255 135 225 120 150 135 120 150 105 165 120
    ↪  180 150 165 225
Circle -16777216 true false 135 90 30
Line -16777216 false 150 105 195 60
Line -16777216 false 150 105 105 60

car
false
0
```

```
Polygon -7500403 true true 300 180 279 164 261 144 240 135 226 132 213 106
    ↪ 203 84 185 63 159 50 135 50 75 60 0 150 0 165 0 225 300 225 300 180
Circle -16777216 true false 180 180 90
Circle -16777216 true false 30 180 90
Polygon -16777216 true false 162 80 132 78 134 135 209 135 194 105 189 96
    ↪ 180 89
Circle -7500403 true true 47 195 58
Circle -7500403 true true 195 195 58

circle
false
0
Circle -7500403 true true 0 0 300

circle 2
false
0
Circle -7500403 true true 0 0 300
Circle -16777216 true false 30 30 240

cow
false
0
Polygon -7500403 true true 200 193 197 249 179 249 177 196 166 187 140 189
    ↪ 93 191 78 179 72 211 49 209 48 181 37 149 25 120 25 89 45 72 103 84
    ↪ 179 75 198 76 252 64 272 81 293 103 285 121 255 121 242 118 224 167
Polygon -7500403 true true 73 210 86 251 62 249 48 208
Polygon -7500403 true true 25 114 16 195 9 204 23 213 25 200 39 123

cylinder
false
0
Circle -7500403 true true 0 0 300

dot
false
0
Circle -7500403 true true 90 90 120

face happy
false
0
Circle -7500403 true true 8 8 285
Circle -16777216 true false 60 75 60
Circle -16777216 true false 180 75 60
Polygon -16777216 true false 150 255 90 239 62 213 47 191 67 179 90 203 109
    ↪ 218 150 225 192 218 210 203 227 181 251 194 236 217 212 240

face neutral
false
0
Circle -7500403 true true 8 7 285
Circle -16777216 true false 60 75 60
Circle -16777216 true false 180 75 60
Rectangle -16777216 true false 60 195 240 225

face sad
false
0
```

```
Circle -7500403 true true 8 8 285
Circle -16777216 true false 60 75 60
Circle -16777216 true false 180 75 60
Polygon -16777216 true false 150 168 90 184 62 210 47 232 67 244 90 220 109
    ↪ 205 150 198 192 205 210 220 227 242 251 229 236 206 212 183

fish
false
0
Polygon -1 true false 44 131 21 87 15 86 0 120 15 150 0 180 13 214 20 212 45
    ↪  166
Polygon -1 true false 135 195 119 235 95 218 76 210 46 204 60 165
Polygon -1 true false 75 45 83 77 71 103 86 114 166 78 135 60
Polygon -7500403 true true 30 136 151 77 226 81 280 119 292 146 292 160 287
    ↪ 170 270 195 195 210 151 212 30 166
Circle -16777216 true false 215 106 30

flag
false
0
Rectangle -7500403 true true 60 15 75 300
Polygon -7500403 true true 90 150 270 90 90 30
Line -7500403 true 75 135 90 135
Line -7500403 true 75 45 90 45

flower
false
0
Polygon -10899396 true false 135 120 165 165 180 210 180 240 150 300 165 300
    ↪  195 240 195 195 165 135
Circle -7500403 true true 85 132 38
Circle -7500403 true true 130 147 38
Circle -7500403 true true 192 85 38
Circle -7500403 true true 85 40 38
Circle -7500403 true true 177 40 38
Circle -7500403 true true 177 132 38
Circle -7500403 true true 70 85 38
Circle -7500403 true true 130 25 38
Circle -7500403 true true 96 51 108
Circle -16777216 true false 113 68 74
Polygon -10899396 true false 189 233 219 188 249 173 279 188 234 218
Polygon -10899396 true false 180 255 150 210 105 210 75 240 135 240

house
false
0
Rectangle -7500403 true true 45 120 255 285
Rectangle -16777216 true false 120 210 180 285
Polygon -7500403 true true 15 120 150 15 285 120
Line -16777216 false 30 120 270 120

leaf
false
0
Polygon -7500403 true true 150 210 135 195 120 210 60 210 30 195 60 180 60
    ↪ 165 15 135 30 120 15 105 40 104 45 90 60 90 90 105 105 120 120 120 105
    ↪  60 120 60 135 30 150 15 165 30 180 60 195 60 180 120 195 120 210 105
    ↪ 240 90 255 90 263 104 285 105 270 120 285 135 240 165 240 180 270 195
    ↪ 240 210 180 210 165 195
```

```
Polygon -7500403 true true 135 195 135 240 120 255 105 255 105 285 135 285
    ↪ 165 240 165 195

line
true
0
Line -7500403 true 150 0 150 300

line half
true
0
Line -7500403 true 150 0 150 150

pentagon
false
0
Polygon -7500403 true true 150 15 15 120 60 285 240 285 285 120

person
false
0
Circle -7500403 true true 110 5 80
Polygon -7500403 true true 105 90 120 195 90 285 105 300 135 300 150 225 165
    ↪  300 195 300 210 285 180 195 195 90
Rectangle -7500403 true true 127 79 172 94
Polygon -7500403 true true 195 90 240 150 225 180 165 105
Polygon -7500403 true true 105 90 60 150 75 180 135 105

plant
false
0
Rectangle -7500403 true true 135 90 165 300
Polygon -7500403 true true 135 255 90 210 45 195 75 255 135 285
Polygon -7500403 true true 165 255 210 210 255 195 225 255 165 285
Polygon -7500403 true true 135 180 90 135 45 120 75 180 135 210
Polygon -7500403 true true 165 180 165 210 225 180 255 120 210 135
Polygon -7500403 true true 135 105 90 60 45 45 75 105 135 135
Polygon -7500403 true true 165 105 165 135 225 105 255 45 210 60
Polygon -7500403 true true 135 90 120 45 150 15 180 45 165 90

sheep
false
15
Circle -1 true true 203 65 88
Circle -1 true true 70 65 162
Circle -1 true true 150 105 120
Polygon -7500403 true false 218 120 240 165 255 165 278 120
Circle -7500403 true false 214 72 67
Rectangle -1 true true 164 223 179 298
Polygon -1 true true 45 285 30 285 30 240 15 195 45 210
Circle -1 true true 3 83 150
Rectangle -1 true true 65 221 80 296
Polygon -1 true true 195 285 210 285 210 240 240 210 195 210
Polygon -7500403 true false 276 85 285 105 302 99 294 83
Polygon -7500403 true false 219 85 210 105 193 99 201 83

square
false
0
```

```
Rectangle -7500403 true true 30 30 270 270

square 2
false
0
Rectangle -7500403 true true 30 30 270 270
Rectangle -16777216 true false 60 60 240 240

star
false
0
Polygon -7500403 true true 151 1 185 108 298 108 207 175 242 282 151 216 59
    ↪ 282 94 175 3 108 116 108

target
false
0
Circle -7500403 true true 0 0 300
Circle -16777216 true false 30 30 240
Circle -7500403 true true 60 60 180
Circle -16777216 true false 90 90 120
Circle -7500403 true true 120 120 60

tree
false
0
Circle -7500403 true true 118 3 94
Rectangle -6459832 true false 120 195 180 300
Circle -7500403 true true 65 21 108
Circle -7500403 true true 116 41 127
Circle -7500403 true true 45 90 120
Circle -7500403 true true 104 74 152

triangle
false
0
Polygon -7500403 true true 150 30 15 255 285 255

triangle 2
false
0
Polygon -7500403 true true 150 30 15 255 285 255
Polygon -16777216 true false 151 99 225 223 75 224

truck
false
0
Rectangle -7500403 true true 4 45 195 187
Polygon -7500403 true true 296 193 296 150 259 134 244 104 208 104 207 194
Rectangle -1 true false 195 60 195 105
Polygon -16777216 true false 238 112 252 141 219 141 218 112
Circle -16777216 true false 234 174 42
Rectangle -7500403 true true 181 185 214 194
Circle -16777216 true false 144 174 42
Circle -16777216 true false 24 174 42
Circle -7500403 false true 24 174 42
Circle -7500403 false true 144 174 42
Circle -7500403 false true 234 174 42
```

```
turtle
true
0
Polygon -10899396 true false 215 204 240 233 246 254 228 266 215 252 193 210
Polygon -10899396 true false 195 90 225 75 245 75 260 89 269 108 261 124 240
    ↪  105 225 105 210 105
Polygon -10899396 true false 105 90 75 75 55 75 40 89 31 108 39 124 60 105
    ↪ 75 105 90 105
Polygon -10899396 true false 132 85 134 64 107 51 108 17 150 2 192 18 192 52
    ↪  169 65 172 87
Polygon -10899396 true false 85 204 60 233 54 254 72 266 85 252 107 210
Polygon -7500403 true true 119 75 179 75 209 101 224 135 220 225 175 261 128
    ↪  261 81 224 74 135 88 99

wheel
false
0
Circle -7500403 true true 3 3 294
Circle -16777216 true false 30 30 240
Line -7500403 true 150 285 150 15
Line -7500403 true 15 150 285 150
Circle -7500403 true true 120 120 60
Line -7500403 true 216 40 79 269
Line -7500403 true 40 84 269 221
Line -7500403 true 40 216 269 79
Line -7500403 true 84 40 221 269

wolf
false
0
Polygon -16777216 true false 253 133 245 131 245 133
Polygon -7500403 true true 2 194 13 197 30 191 38 193 38 205 20 226 20 257
    ↪ 27 265 38 266 40 260 31 253 31 230 60 206 68 198 75 209 66 228 65 243
    ↪ 82 261 84 268 100 267 103 261 77 239 79 231 100 207 98 196 119 201 143
    ↪  202 160 195 166 210 172 213 173 238 167 251 160 248 154 265 169 264
    ↪ 178 247 186 240 198 260 200 271 217 271 219 262 207 258 195 230 192
    ↪ 198 210 184 227 164 242 144 259 145 284 151 277 141 293 140 299 134
    ↪ 297 127 273 119 270 105
Polygon -7500403 true true -1 195 14 180 36 166 40 153 53 140 82 131 134 133
    ↪  159 126 188 115 227 108 236 102 238 98 268 86 269 92 281 87 269 103
    ↪ 269 113

x
false
0
Polygon -7500403 true true 270 75 225 30 30 225 75 270
Polygon -7500403 true true 30 75 75 30 270 225 225 270
@#$#@#$#@
NetLogo 6.4.0
@#$#@#$#@
@#$#@#$#@
@#$#@#$#@
@#$#@#$#@
@#$#@#$#@
default
0.0
-0.2 0 0.0 1.0
0.0 1 1.0 0.0
0.2 0 0.0 1.0
```

```
link direction
true
0
Line -7500403 true 150 150 90 180
Line -7500403 true 150 150 210 180
@#$#@#$#@
1
@#$#@#$#@
```

Modifications:

RPSLS

```
to-report beat? [ target ]


  report (pcolor = red and [ pcolor ] of target = green) or
             (pcolor = green and [ pcolor ] of target = blue) or
             (pcolor = blue  and [ pcolor ] of target = red) or
             (pcolor = red and [ pcolor ] of target = orange) or
             (pcolor = yellow and [ pcolor ] of target = red) or
             (pcolor = orange and [ pcolor ] of target = yellow) or
             (pcolor = yellow and [ pcolor ] of target = green) or
             (pcolor = green and [ pcolor ] of target = orange) or
             (pcolor = orange and [ pcolor ] of target = blue) or
             (pcolor = blue and [ pcolor ] of target = yellow)

end
```

Cannibalism

```
to-report beat? [ target ]
  report (pcolor = red   and [ pcolor ] of target = green) or
         (pcolor = green and [ pcolor ] of target = blue) or
         (pcolor = blue  and [ pcolor ] of target = red) or
         (pcolor = red  and [ pcolor ] of target = red) ;or
;         (pcolor = blue  and [ pcolor ] of target = blue) or
;         (pcolor = green  and [ pcolor ] of target = green)

end
```