# Agent-Based Evolutionary Game Dynamics

# Agent-Based Evolutionary Game Dynamics

*A guide to implement and analyze Agent-Based Models within the framework of Evolutionary Game Theory*

*LUIS R. IZQUIERDO; SEGISMUNDO S. IZQUIERDO; AND WILLIAM H. SANDHOLM*

Cover picture by Dino Reichmuth

Save citation: BIB TXT RIS

This book was produced with Pressbooks (https://pressbooks.com) and rendered with Prince.

# Contents

## Part I. Introduction

## Part V. Agent-based models vs ODE models

## Appendices

# Dedication

To Carmen, Daniela, and Alejandra,
for being the meaning, the music,
the beauty, and the adventure of my life

To May, Gemma, and Luis
You are always on my mind

# 1. Purpose

The purpose of this book is to help you learn to **implement and analyze evolutionary models of social interactions in finite populations**. This goal may not be perfectly clear to you right now (and that is absolutely fine), so let us examine the main terms in the bold sentence above, one by one.

- By "**social interactions**" we mean interactions between individuals where the outcome for each individual depends not only on her own actions, but also on the actions of others. A simple example is the decision to drive on the left or on the right of a two-way road. Driving on your left is a good idea if everyone else also drives on their left, but not so good if everyone drives on their right. The interaction between a penalty kicker and the goal keeper is another example. A more complex example would be the decision to start a new collaboration with a partner or not and, if so, how much effort to put into the partnership. We will see in this book that *game theory* is a great framework to formalize and analyze this type of social interactions.

- By "**evolutionary model**" we mean a model where the individuals involved in the interaction may change their actions in time, according to some decision rule. An example of a decision rule could be: "Look at what other individuals are doing, identify who is doing best, and copy what she is doing (i.e., her action)". If individuals update their actions in time, the distribution of actions in the population changes or *evolves*. Evolutionary models at their core are just *dynamic* models, i.e. models where there is some change in time. We will see in this book that *evolutionary game theory* is a great framework to formalize and analyze this type of dynamic models of social interactions.

- By "**finite populations**" we mean that there is a finite number of individuals in our models. You may be wondering: what else could it be? Well, the point is that many models in evolutionary game theory assume that the population is infinite. Normally, this assumption is made for mathematical tractability, i.e., for analytical convenience. Here we will not impose that assumption. Our object of study will always be a model of a population which, in principle, could exist in the real world.

- By "**learn to implement evolutionary models**", we mean that we are going to learn how to code our evolutionary models in a programming language, using a computer. It is perfectly fine if you have never programmed before; actually, if that is the case, we believe that you are going to enjoy this book immensely. If, on the other hand, you have some programming experience, chances are that you will sail through many chapters of this book faster than you think. In this book we are going to use an agent-based (aka individual-based) approach, i.e. we are going to implement how individuals make their own decisions, how they *individually* change their actions in time, and we are going to see how they *individually* enjoy or suffer the consequences of their *individual* decisions. We will also be able to see how the distribution of actions in the population changes in time, according to the different assumptions that we wish to implement in our models.

- And finally, by "**learn to analyze evolutionary models**" we mean that we will do our very best to understand the dynamics of our models, i.e. we are going to find out *why* we observe what we observe, and try to derive conclusions about the logical implications of the assumptions we implement in our models.

The following paragraphs explain why the two skills we are going to learn with this book –i.e. model implementation and analysis– are key for scientific modeling.

## Model implementation

To use a scientific model rigorously, it is important to be fully aware of all the assumptions embedded in it, and also of the various alternative assumptions that could have been chosen. If we don't understand all the details of a model, we run the risk of over-extrapolating its scope and of drawing unsound conclusions. A great way to understand a model in depth is to *implement* it in computer code following an agent-based approach. We believe this is true regardless of whether the model is currently expressed in natural language (and may even exist only in your mind) or, alternatively, it is written down in mathematical language (e.g. using equations).

Coding a model expressed in natural language is very useful because it ensures that the model is both unambiguous and completely specified. A computer implementation of a model is necessarily *unambiguous* because the language used to code it (i.e. the programming language) is formal, so it does not allow ambiguities to infiltrate; symbols and instructions used in programming languages have always the same meaning regardless of context. A computer implementation of a model must also be *completely specified* before it can be run, since computers do not make assumptions by themselves. Thus, to execute a model in a computer, there cannot be any loose ends in the description of the model. This contrasts with models expressed in natural language, where it is easy to leave aspects of the model partially unspecified –often unintentionally–, since our brains are particularly good at using context to unconsciously fill the details. If we use natural language to describe a model, the audience may understand something slightly different from what we mean to communicate, and results may be driven by assumptions that we have not made explicit. By contrast, computers need all assumptions to be spelt out, and this requirement makes the process of scientific modeling more sound and rigorous.

If a model is written in the language of mathematics, the problems outlined in the paragraph above are no longer an issue. Thus, is it really worth implementing a mathematical model in computer code following an agent-based approach? We believe that, in many cases, it certainly is. The reason is that many mathematical models contain assumptions that are desirable for analytical tractability, but which also weaken the link between the model and the real world. These assumptions made for analytical convenience tend to elevate the mathematical model to a higher level of abstraction and aggregation. By contrast, the agent-based approach has the advantage of forcing the programmer to implement the microfoundations of a model explicitly, considering each individual agent as a separate entity. This requirement helps the modeler be aware of all the assumptions that are made in the mathematical model, and it also allows for an assessment of their significance.

## Model analysis

Once the model is implemented, the way to fully understand it is to *analyze* it. In this book, we will see several techniques that are useful to analyze finite-population evolutionary models, including Markov chains, Monte Carlo simulations, mean dynamics, stochastic stability analyses, diffusion approximations and pair approximations. For each of these techniques, we give a brief introduction, illustrate its usefulness with concrete examples, and provide references for the interested reader to learn more about it.

# 2. Structure of the book and potential courses

In this section we present an overview of the whole book and suggest different paths that readers may want to follow, depending on their particular interests and the time they want to spend on the book.

The book is divided into five parts, numbered in roman numerals:

- Part I. Introduction
- Part II. Our first agent-based evolutionary model
- Part III. Spatial interactions on a grid
- Part IV. Games on networks
- Part V. Agent-based models vs ODE models

These parts are largely independent and self-contained, except for Parts IV and V, which build on Part II. Each part contains several chapters. Each chapter corresponds to one webpage in the webbook.

Part I is somewhat special in the sense that it contains introductions to Evolutionary Game Theory (chapter I-2), Agent-based Modeling (chapter I-3), NetLogo (chapter I-4) and programming in NetLogo (chapter I-5). These introductions are self-contained and independent, so the reader who is already familiar with some of these topics can perfectly skip the corresponding chapters.

The modular design of the book allows for the configuration of different courses. The following lists provide some examples.

Courses on **Agent-Based Modeling (ABM)**:

- **Basic course on ABM**: Chapters I-3, I-4 and I-5, plus Part II. Optional: chapter I-2.
- **Course on ABM, including spatial interactions**: Parts I, II and III.
- **Course on ABM, including networks**: Parts I, II and IV.
- **Advanced course on ABM**: Parts I, II, III and IV.

Courses on **Evolutionary Game Theory (EGT)**:

- **Basic course on EGT**: Parts I and II.
- **Intermediate course on EGT**: Parts I, II and V.

- **Course on EGT, including spatial interactions**: Parts I, II, III and V.

- **Course on EGT, including networks**: Parts I, II, IV and V.

- **Advanced course on EGT**: The whole book.

The book contains more than 100 exercises with detailed solutions. The exercises are proposed at the end of most chapters, in a section titled "**Exercises**". The solutions for the exercises proposed in each Part are included in the last chapter of the corresponding Part, which is titled "**Answers to exercises**". These chapters are only present in the online webbook, and they are protected with a password. If you are not a student following a course based on this book, feel free to contact us to obtain access to these solutions.

# 3. Why NetLogo?

This book uses a programming language called NetLogo (Wilensky, 1999) to implement several models. In chapter I-4 we explain many of the features that make NetLogo our language of choice. Nonetheless, the main reason we chose NetLogo is that it is remarkably close to natural language. As a matter of fact, we believe that NetLogo language can perfectly be used as pseudo-code to communicate algorithms. This means that, in theory, a proficient programmer could use this book to implement the proposed models in her language of choice, rather than in NetLogo.

Given how easy it is to understand NetLogo code, admittedly, we have been very sparing in commenting our code. The reader may want to comment her own code much more than we do, and we definitely recommend this good practice. We did not comment our code much because we wanted to provide code as clean as possible, and also because comments are fairly personal; everyone has her own personal difficulties when programming and understanding code.

# 4. One book, many formats

This book has been written and formatted following a hands-on approach. It contains several (internal and external) hyperlinks, images, videos, code snippets, links to NetLogo programs, etc. The book comes in different formats:

- **Webbook**: This is the online version of the book. We believe this is the most convenient format for most readers. Its only drawback is that it requires access to the internet.

- **Digital PDF**: This a pdf document that you can download and use online or offline. This format preserves all the hyperlinks of the webbook, and these hyperlinks use the same colored fonts as in the webbook. Naturally, you will need to be online to follow the external links. Also, note that in this format all the videos have been replaced with hyperlinks, so you will need to be online to watch the videos.

- **Print PDF**: This is a pdf document that has no hyperlinks (nor videos). The hyperlinked (colored) words in the Digital PDF are not colored in this format, so readers who want to print the pdf may prefer this version to the Digital PDF.

- **EPUB**: This is an e-book file that can be read by most e-book readers. All images and videos are contained inside this file, so they can be seen without access to the internet.

In addition to these formats, we also provide Common Cartridge Files. These files contain web links to the different chapters of the book, and are useful to import parts of the book into Learning Management Systems such as Canvas, Moodle or Blackboard.

# 5. What about the programming and the math?

We are hopeful and confident that if you go through the whole text, implement the proposed models, and try to do some of the exercises included at the end of most chapters, you will master the art of implementing and analyzing agent-based evolutionary dynamics, using both computer simulation and mathematical analysis.

Nonetheless, we also have in mind two other types of less committed readers:

- If you are interested in learning how to analyze finite-population evolutionary models, and in their relation to other models in Evolutionary Game Theory, but **you do not wish to program**,

  then you should skip all the sections preceded by the NetLogo icon . These are the sections with a strong programming component.

- If you want to become proficient in coding agent-based models, but **you are not interested in learning how to analyze these models**, please do reconsider your preference. If your preference persists after careful reflection, you may want to skip the chapters titled "Analysis of these models" (which you will find at the end of Parts II, III and IV) and also chapter V-3. These are the chapters with a strong mathematical component.

Our hope is that, regardless of the discipline you are coming from, your background and your preferences, this book will help you learn new and exciting ways of understanding evolutionary systems.

# 6. History of the book and acknowledgments

The first steps of this book date back to 2016. That year, we —Segis and Luis— were awarded a grant to visit world-renowned Professor Bill Sandholm at the University of Wisconsin–Madison for three months. Having read his work, we had extremely high expectations for our in-person meeting with Bill.

Bill exceeded all our expectations by orders of magnitude in every conceivable way. Every minute spent with him was full of wisdom, generosity, awe, admiration, and joy. Every day spent working with him was worth a year on our own. Meeting Bill was, without a doubt, the decisive moment in our scientific careers.

*Segis, Bill and Luis in Madison (Wisconsin), in the Summer of 2016*

The primary goal of our visit was to design this book, and we hoped that Bill would review our proposal and provide some feedback. Not only did he fulfill our request with the greatest interest and professionalism, but he also offered his assistance in every way he could. Specifically, he introduced us to Steel Wagstaff, an extremely competent and inspiring instructional technology consultant who dedicates his life to doing wonderful things for wonderful reasons. Steel suggested we could use Pressbooks to publish this book, set everything up for us to begin writing, and instilled in us his passion for developing Open Educational Resources. Since then, Steel has been a constant source of motivation and support. This book would never have existed in its present form had we not met Steel.

After working with Bill for a couple of months, he honored us by accepting our invitation to contribute to the book as an author. Then, over the following years, we had the great privilege of collaborating with Bill on various projects, including this book. During those wonderful years, our admiration for Bill only grew, and every day we felt incredibly fortunate to have the opportunity to work alongside such an amazing scholar.

Then, on a fateful summer day in 2020, we received the devastating news that our dear mentor and friend had left this world.
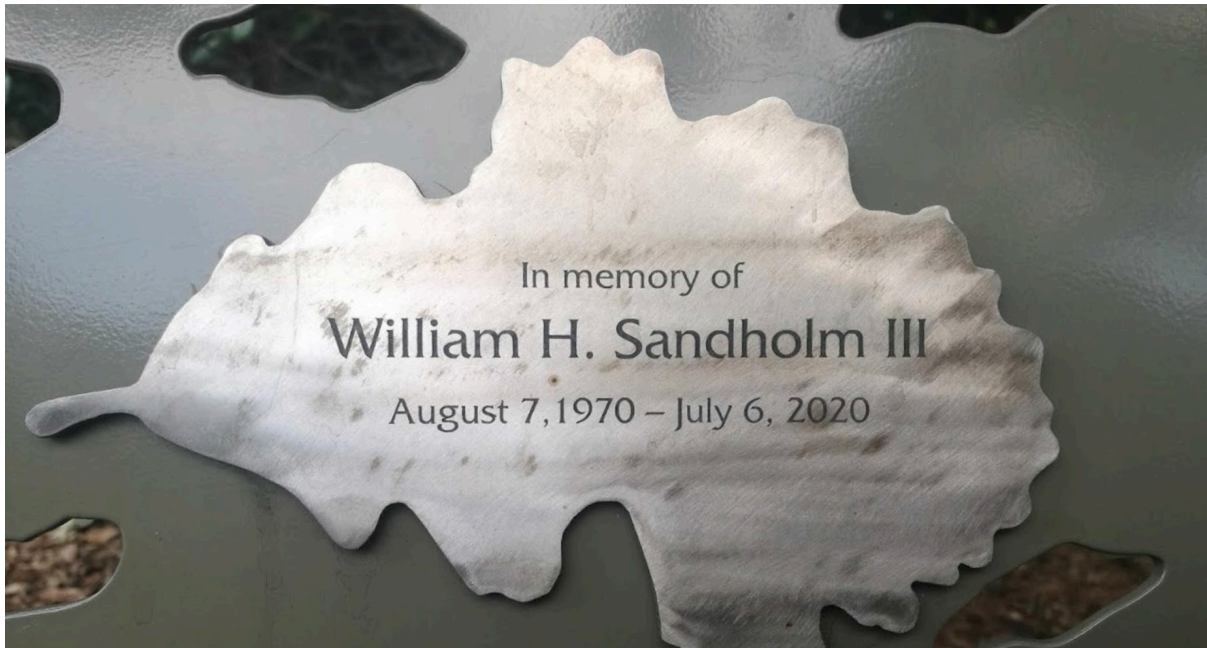
*Bench at the University of Wisconsin–Madison Arboretum*

After Bill's passing, it took us a long time to muster the courage and energy to continue working on a book that we had begun with him. Eventually, we found the strength to pull ourselves together and finish the book at the University of British Columbia, largely thanks to the wonderful support and care of Christoph and Claudia Hauert.

Our greatest hope now is that Bill would feel proud to see his name on the book's cover. If you find anything unclear or mistaken in this book, it is certainly our fault; if you discover something beautiful or inspiring, Bill is likely to be behind. We profoundly miss Bill's extraordinary intelligence, kindness, playful spirit, and commitment to scientific rigor. But, most importantly, when we think of him, we are filled with immense gratitude. We are truly thankful for the chance life afforded us to learn so much from such an incredible and remarkable friend.

Jörgen W. Weibull, Nacho Santos, Francisco Fatás-Villafranca, Koen Frenken, Isabel Almudí, Doina Olaru, Ryan McAllister, and Cesáreo Hernández for their pivotal roles in shaping our academic careers.

Finally, we would like to acknowledge our greatest debt. Over the past eight years, our beloved families have made tremendous sacrifices so we could write this book. Carmen and Gemma had to leave their jobs in Spain three times. Luis left his friends behind and followed us into the unknown, Daniela changed schools five times and Alejandra had to learn to walk on a transatlantic flight. They all had to endure countless nights sleeping on the floor of unfurnished student flats adorned with camping chairs. Most families we know would not have survived the ordeals our families have been through to support us. Only two families we know can do it with a smile on their face and remember all of it as a great adventure. We cannot thank them enough for that.

# PART I. INTRODUCTION

## I-1. Overview

# 1. What is this book about?

This book is a guide to implement and analyze **Agent-Based Models** within the framework of **Evolutionary Game Theory**, using a programming language called **NetLogo**.

Let us flesh out the main terms in the previous sentence. A *model* is an abstraction of the real world and, basically, *agent-based* **models** are models where individuals and their interactions are explicitly represented in the model (see fig. 1).
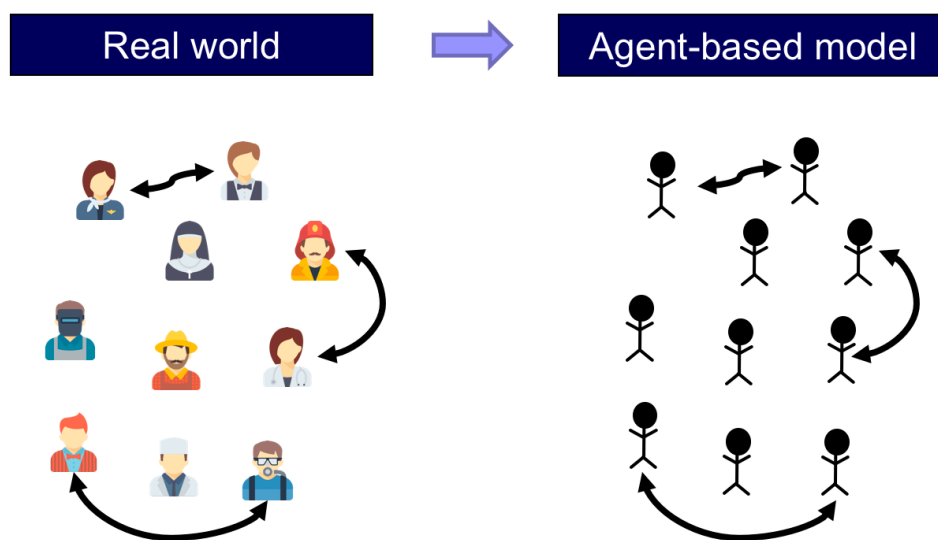


*Figure 1. In an agent–based model, the individual units of the real–world system to be modeled and their interactions are explicitly and individually represented in the model*

Agent-based models are most often implemented as computer programs, and the programming language we are going to use in this book is **NetLogo**. And finally, *game theory* is a formal theory devoted to studying interactions among individuals whose actions affect each other. In particular, *evolutionary* **game theory** studies populations of individuals who may change their actions in time.

In this book we are going to learn how to implement agent-based models in NetLogo, within the framework of evolutionary game theory, and we are also going to learn how to analyze these models using both computer simulation and mathematical analysis.

# 2. How is this book organized?

The book is divided into five parts:

> ## Structure of the book
>
> - Part I. Introduction
> - Part II. Our first agent-based evolutionary model
> - Part III. Spatial interactions on a grid
> - Part IV. Games on networks
> - Part V. Agent-based models vs ODE models

Each part contains several chapters; and each chapter corresponds to one webpage in the online webbook.

The following sections summarize each of the parts of the book. To do this, admittedly, we will have to use concepts that you may not be familiar with, just yet. Apologies for that, but do not worry. We will define and explain every concept we use here in the following chapters.

## 2.1. Part I. Introduction

This first part contains the following chapters:

> ## Part I. Introduction
>
> - Chapter I-1. Overview
> - Chapter I-2. Introduction to evolutionary game theory
> - Chapter I-3. Introduction to agent-based modeling
> - Chapter I-4. Introduction to NetLogo
> - Chapter I-5. The fundamentals of NetLogo

This very chapter you are now reading is "Chapter I-1. Overview". The rest of the chapters in this first part are introductions to the three main components of the book: evolutionary game theory (chapter I-2), agent-based modeling (chapter I-3), and NetLogo (chapter I-4 describes the main features of NetLogo, while chapter I-5 is a tutorial on programming in NetLogo). These introductions are self-contained and independent, so if you are already familiar with any of these topics, feel free to skip the corresponding chapters.

## 2.2. Part II. Our first agent-based evolutionary model

In Part II we implement our first agent-based evolutionary model, which is a simple model of a *well-mixed population*. The term *well-mixed population* refers to a population where all individuals are equally likely to interact with each other. Part II includes the following chapters:

> **Part II. Our first agent-based evolutionary model**
>
> - II-1. Our very first model
> - II-2. Extension to any number of strategies
> - II-3. Noise and initial conditions
> - II-4. Interactivity and efficiency
> - II-5. Analysis of these models
> - II-6. Answers to exercises

In chapter II-1, we implement a model where agents repeatedly play a symmetric 2-player 2-strategy game (chosen by the user) and follow a simple rule to update their strategies. The code for this first (fully-functional) model fits in one page. In chapter II-2 we extend this model to allow for games with any number of strategies. Then, in chapter II-3, we add two very useful features to the model: the possibility of setting initial conditions explicitly, and the possibility that revising agents select their strategy at random with a small probability (i.e., noise). In chapter II-4, we improve the interactivity of our model (i.e., the possibility of changing the value of parameters at runtime, with immediate effect on the dynamics of the model) and its efficiency (i.e., we modify the code of our model to make it run significantly faster).

Chapter II-5 does not extend the model anymore. Instead, this chapter focuses on how to analyze it using both computer simulation and mathematical analysis. In this chapter, we present different mathematical theories and approximations that can be used to analyze agent-based evolutionary models of well-mixed populations. In particular, we discuss the theory of Markov chains and we introduce various mathematical techniques such as the mean dynamic, the diffusion approximation and stochastic stability analyses.

Finally, in chapter II-6, we provide the solution to the 30 exercises proposed in this part.

## 2.3. Part III. Spatial interactions on a grid

Part III is self-contained and completely independent of Part II. In Part III, we show how to build and analyze agent-based models with *spatial structure*. The video below shows an illustrative run of a population of agents embedded on a 81×81 spatial grid. Each cell represents an agent. Agents play

a game called the Prisoner's Dilemma with their neighbors. In the video below, agents are colored according to their current strategy and their previous one.[1]

> 📹 *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=5#video-5-1*

In the models implemented in this part, agents do not interact with other agents with the same probability (as in Part II), but they interact only with those who are nearby in space. Populations where not every agent is equally likely to interact with each other are called *structured* populations (as opposed to *well-mixed* populations). Part III includes the following chapters:

## Part III. Spatial interactions on a grid

- III-1. Spatial chaos in the Prisoner's Dilemma
- III-2. Robustness and fragility
- III-3. Extension to any number of strategies
- III-4. Other types of neighborhoods and other decision rules
- III-5. Analysis of these models
- III-6. Answers to exercises

In chapter III-1, we implement a model where agents embedded on a grid repeatedly play a symmetric 2-player 2-strategy game (chosen by the user) with their spatial neighbors. The code for this first spatial model can perfectly fit in one page. In chapter III-2, we add three very useful features to the basic spatial model: noise, asynchronous strategy updating, and the possibility to choose whether agents play the game with themselves or not. We use these features to assess the robustness of various results in the literature. Then, in chapter III-3, we extend the model to allow for games with any number of strategies. In chapter III-4, we add two features that are crucial to assess the impact of space on evolutionary dynamics: the possibility to model different types of neighborhoods of arbitrary size, and the possibility to model different decision rules agents may use to update their strategy. The resulting model allows us to replicate many results in the literature.

Chapter III-5 does not extend the model anymore, but instead focuses on how to analyze it using both computer simulation and mathematical analysis. In this chapter, we explain what cellular automata are, and we discuss the pair approximation, a mathematical technique that is sometimes used to approximate the dynamics of evolutionary models on grids.

---

1. The settings for this simulation run are defined in the Discussion section of chapter III-2.

Finally, in chapter III-6, we provide the solution to the 29 exercises proposed in this part.

## 2.4. Part IV. Games on networks

In Part IV, we show how to implement and analyze agent-based models where players are connected through a network (fig. 2). In our models, the nodes in the network are the players, and each link connects two players.
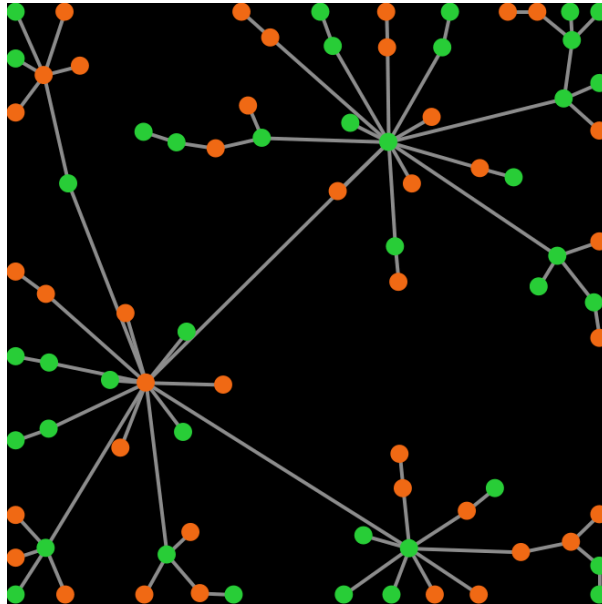


*Figure 2. Example of a network. Players are represented as circles and colored according to their strategy. There are links between some pairs of players*

In network models, we assume that players can only interact with their link-neighbors (i.e. those with whom the player shares a link), so link-neighbors are the only players that a player can observe or play with.

By using networks, we are able to generalize all the models previously developed in the book. Note that in Part II we implement models where every player could observe and play with every other player. Such models can be interpreted as network models where players are connected through a complete network, i.e., a network where everyone is linked to everybody else (fig. 3).
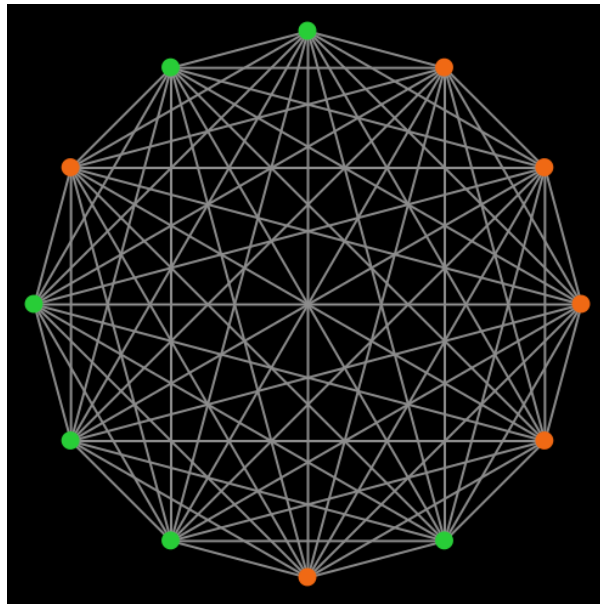
*Figure 3. A complete network with 12 nodes*

The spatial models developed in Part III can also be interpreted as network models. For instance, a grid model where adjacent cells (i.e., players) are neighbors (fig. 4a) corresponds to a square lattice network (fig. 4b).
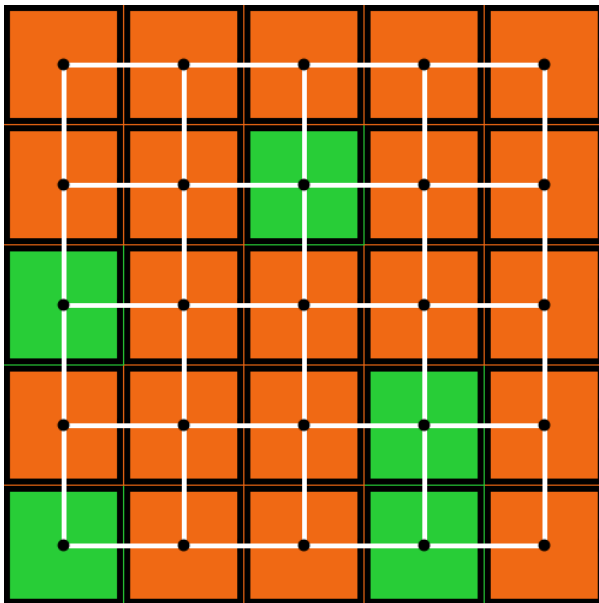


*(a) A grid where adjacent cells are neighbors. White lines connect neighbors*
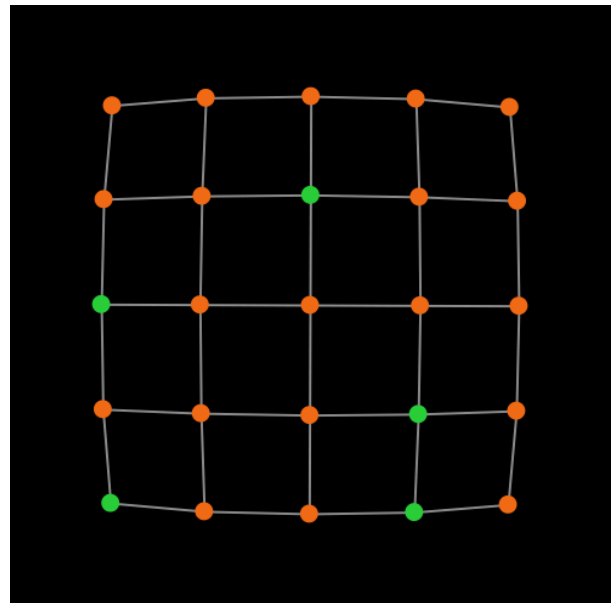


*(b) A square lattice network that corresponds to the grid on the left*

*Figure 4. (a) A grid model where adjacent cells are neighbors and (b) an equivalent representation as a square lattice network*

In this Part IV, we learn to implement models where players are connected through any arbitrary network. Part IV includes the following chapters:

<div style="background-color:#C8621A;color:white;padding:10px;">

## Part IV. Games on networks

</div>

- IV-1. The nxn game on a random network
- IV-2. Different types of networks
- IV-3. Implementing network metrics
- IV-4. Other ways of computing payoffs and other decision rules
- IV-5. Analysis of these models
- IV-6. Answers to exercises

In chapter IV-1, we modify the well-mixed population model implemented in Part II by embedding the players on a random network. In chapter IV-2, we extend our random-network model by adding the possibility of creating many other types of networks, such as small-world, preferential attachment, rings, stars, wheels, paths, etc. This allows us to assess the significance of network structure on evolutionary dynamics. Then, in chapter IV-3, we implement several functions to compute network metrics such as the density, the size of the largest component, the clustering coefficient and the degree distribution. Chapter IV-4 greatly generalizes the model by adding two features that can have a major impact on evolutionary dynamics on networks: the possibility to model different ways of computing payoffs and the possibility to model different decision rules that agents may use to update their strategy. With this final network model we are able to replicate and discuss many results in the literature.

Chapter IV-5 does not extend the model anymore. Instead, this chapter focuses on how to analyze it. In this chapter, we discuss four best practices that can help us to analyze evolutionary dynamics using computer simulation.

Finally, in chapter IV-6, we provide the solution to the 30 exercises proposed in this part.

## 2.5. Part V. Agent-based models vs ODE (Ordinary Differential Equation) models

Many models in the Evolutionary Game Theory literature are described as systems of Ordinary Differential Equations (ODEs). The most famous example is the Replicator Dynamics (Taylor and Jonker, 1978). These mathematical models are very different from the agent-based models we consider in this book. Our goal in Part V is to explain the relationship between these two kinds of models. We will see that ODE models in the literature can be seen as approximations to the dynamics of different agent-based models when the population is large (fig. 5).
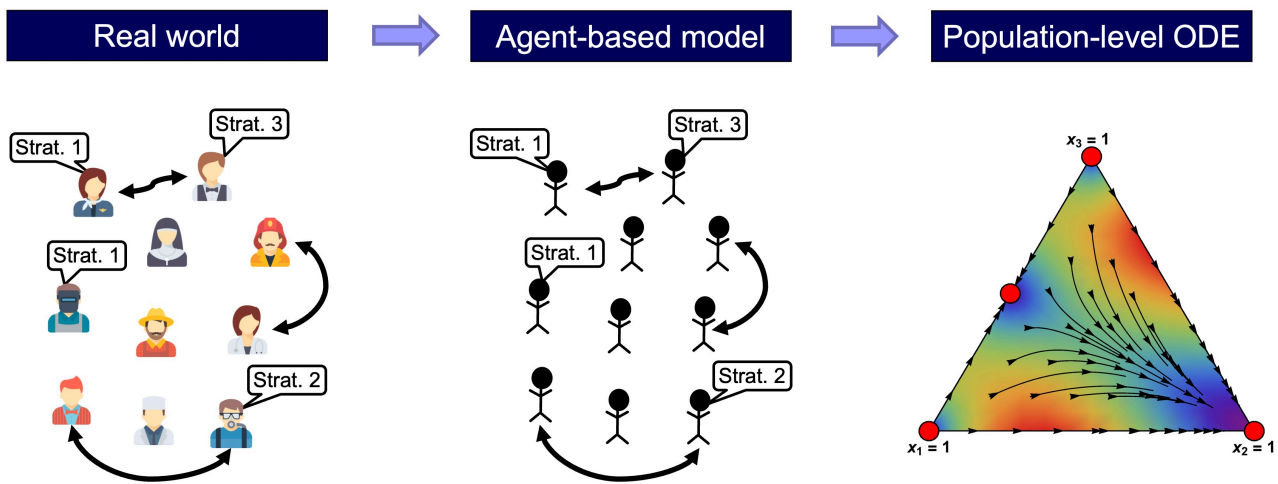
*Figure. 5. Agent–based models are described at the individual level, while ODE models are described at the population level*

Part V includes the following chapters:

## Part V. Agent-based models vs ODE models

- V-1. Introduction
- V-2. A rather general model for games played in well-mixed populations
- V-3. Mean Dynamics
- V-4. Answers to exercises

Chapter V-1 provides an introduction to the (many-to-one) relationship between agent-based models and ODE models. In chapter V-2, we generalize the well-mixed population model implemented in Part II by adding two important features: the possibility to model different ways of computing payoffs and the possibility to model different decision rules that agents may use to update their strategy. Then, in chapter V-3, we derive an ODE approximation (i.e., the mean dynamic) for every possible parameterization of our new (and fairly general) agent-based model. In this chapter we also learn how to numerically solve ODEs within NetLogo at runtime. This functionality allows us to compare agent-based simulations and their ODE approximations at runtime (fig. 6).
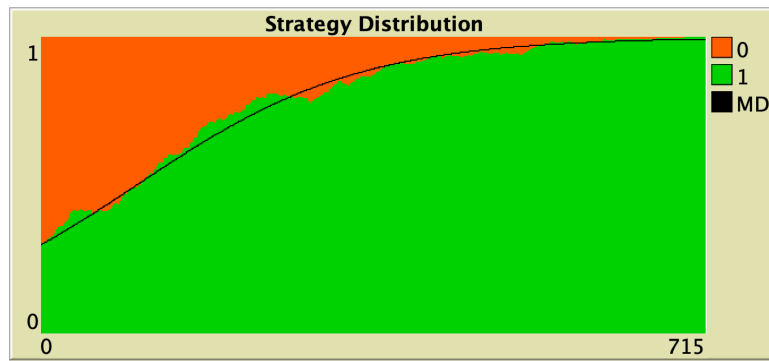
*Figure 6. Time series of the strategy distribution of an agent–based model, together with the solution of its mean dynamic (in black). The green area shows the fraction of agents using strategy 1 along time. The black line is the numerical solution of the corresponding ODE approximation to that fraction. The settings for this simulation run are defined in the caption of figure 3 in section 4.3 of chapter V–3*

Finally, in chapter V-4, we provide the solution to the 12 exercises proposed in this part.

## 2.6. Appendices

The book contains three appendices:

- In Appendix A-1, we explain that oftentimes there are different ways of implementing any particular model, and we comment on how to choose the most appropriate implementation for a given purpose.

- The book contains many code skeletons that provide a schematic bird's-eye view of the code. Figure 7 below shows an example. Appendix A-2 includes the legend to interpret these code skeletons.

- Finally, Appendix A-3 includes links to download each of the 22 NetLogo programs that are implemented in this book.
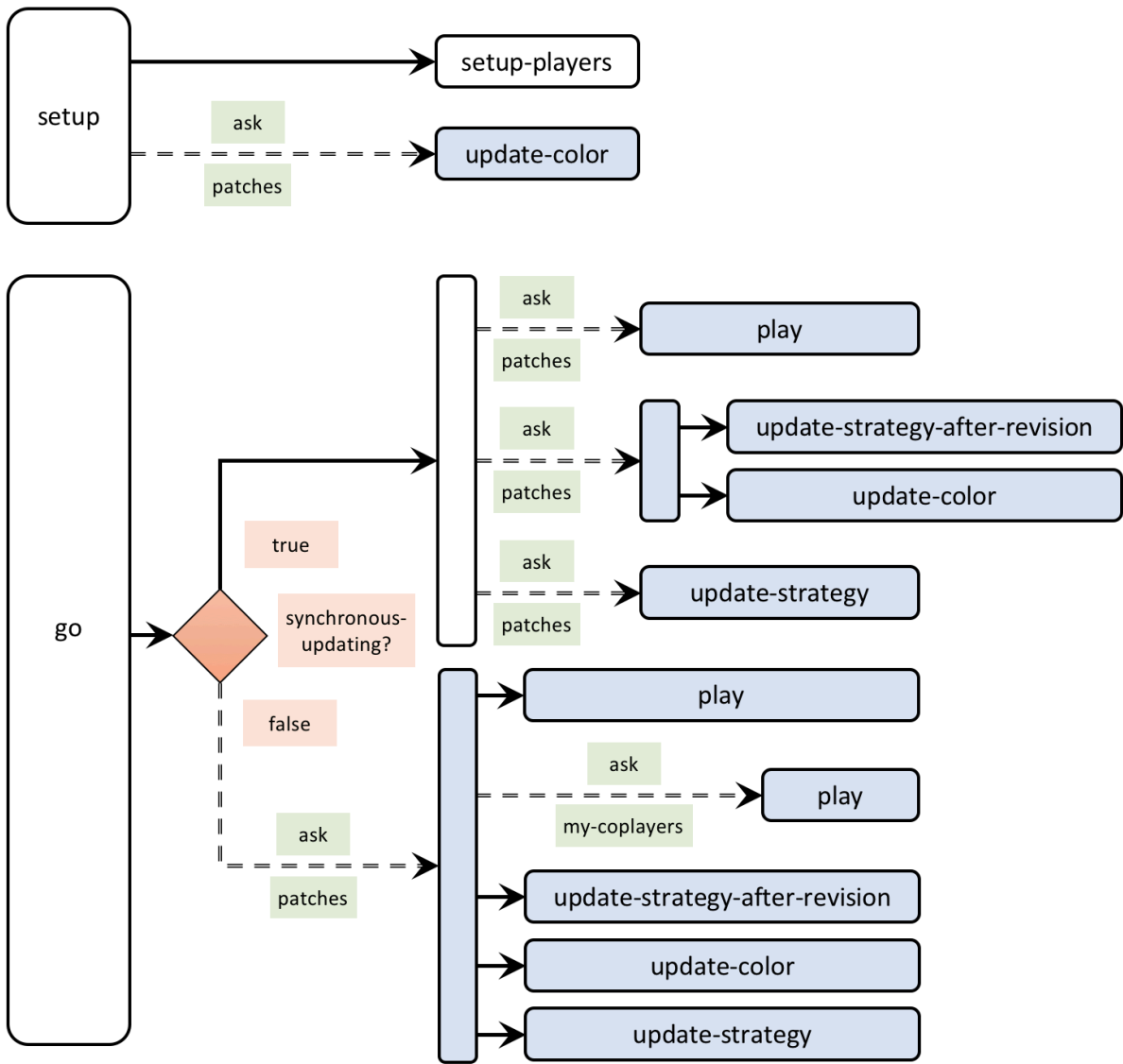
*Figure 7. Skeleton of the code implemented in chapter III-2*

## I-2. Introduction to evolutionary game theory

The goal of this chapter is to provide a brief introduction to Evolutionary Game Theory (EGT). Nonetheless, since EGT is a branch of a more general discipline called game theory, we believe it is useful to get familiar with the basic ideas underlying game theory first.

# 1. What is game theory?

Game theory is a discipline devoted to studying social interactions where individuals' decisions are interdependent, i.e. situations where the outcome of the interaction for any individual generally depends not only on her own choices, but also on the choices made by every other individual. Thus, several scholars have pointed out that game theory could well be defined as 'interactive decision theory' (Myerson, 1997, p. 1). Some examples of interactive decisions for which game theory is useful are:

1. Choosing the side of the road on which to drive.

2. Choosing WhatsApp or Facebook messenger as your default text messaging app.

3. Choosing which restaurant to go in the following situation:

   > Imagine that over breakfast your partner and you decided to have lunch together, but you did not specify where. Right before lunch time you discover that you left your cellphone at home, and there is no other way to contact your partner quickly. Fortunately, you are not completely lost: there are only two restaurants that you both love in town. Which one should you go to?

Social interactions like the ones outlined above are modeled in game theory as *games*. A game is an abstract representation of a social interaction which is meant to capture its most basic properties. In particular, a game typically comprises:

- the set of individuals who interact (called *players*),

- the different choices available to each of the individuals when they are called upon to act (named actions or *pure strategies*),

- the *information* individuals have at the time of making their decisions,

- and a *payoff* function that assigns a value to each individual for each possible combination of choices made by every individual (Fig. 1). In most cases, payoffs represent the preferences of each individual over each possible outcome of the social interaction,[1] though there are some

---

1. A common misconception about game theory relates to the roots of players' preferences. There is no assumption in game theory that dictates that players' preferences are formed in complete disregard of each other's interests. On the contrary, preferences in game theory are assumed to account for anything, i.e. they may include altruistic

evolutionary models where payoffs represent Darwinian fitness.

| | Player 2 | |
| --- | --- | --- |
| | Player 2 chooses **A** | Player 2 chooses **B** |
| Player 1 chooses **A** | **1 , 1** | **0 , 0** |
| Player 1 chooses **B** | **0 , 0** | **2 , 2** |

*Figure 1. Payoff matrix of a 2-player 2-strategy game. For each possible combination of pure strategies there is a corresponding pair of numbers (x , y) in the matrix whose first element x represents the payoff for player 1, and whose second element y represents the payoff for player 2*

Note that the payoff matrix shown in Fig. 1 could well be used for the three examples outlined above, assuming that there is a preferred side of the road, text messaging app or restaurant in town. To be sure, let us model the example about choosing a restaurant as a game, identifying each of its elements:

- The players would be you and your partner.
- Each of you may choose restaurant A or restaurant B.
- Neither of you have any information about the other's choice at the time of making your decision.
- Both of you prefer eating together rather than being alone, no matter where, and you both prefer restaurant B over restaurant A. Thus, the best possible outcome for both would be to meet at restaurant B; second best would be meeting at restaurant A; and any lack of coordination would be equally awful.[2]

The three examples above are very different in many aspects, but they all could be modeled using the same game. This is so because games are abstractions that are meant to capture the bare essentials of the original social interaction and, at least to some extent, the three examples above share the same strategic backbone.

Having seen this, it may not come as a surprise that the sort of issues for which game theory can be useful is impressively broad and diverse, including applications in international relations, resource management, network routing (of vehicles or information packages), voting systems, linguistics, law, distributed control, evolutionary biology, design of incentive systems, and business and environmental regulations.

---

motivations, moral principles, and social constraints (see e.g. Colman (1995, p. 301), Vega-Redondo (2003, p. 7), Binmore and Shaked (2010, p. 88), Binmore (2011, p. 8) or Gintis (2014, p. 7)).

2. Note that there is no inconsistency in being indifferent about outcomes {A, B} and {B, A}, even if you prefer restaurant B. It is sufficient to assume that you care about your partner as much as about yourself.

# 2. Traditional game theory

Game theory has nowadays various branches. Historically, the first branch to be developed was Traditional Game Theory (TGT) (von Neumann and Morgenstern (1944), Nash (1950), Selten (1965, 1975), Harsanyi (1967, 1968a, 1968b)). TGT is also the branch where most of the work has been focused, and the one with the largest representation in most game theory textbooks and academic courses.[3]

In TGT, payoffs reflect preferences and players are assumed to be rational, meaning that they act as if they have consistent preferences and unlimited computational capacity to achieve their well-defined objectives. The aim of the discipline is to study how these instrumentally rational players would behave in order to obtain the maximum possible payoff in the formal game.

A key problem in TGT is that, in general, assuming rational behavior for any one player rules out very few actions –and consequently very few outcomes– in the absence of strong assumptions about what players know about others' rationality, knowledge and actions. Hence, in order to derive specific predictions about how rational players would behave, it is often necessary to make very stringent assumptions about everyone's beliefs and their reciprocal consistency. If one assumes common knowledge of rationality and consistency of beliefs, then the outcome of the game is a *Nash equilibrium*, which is a set of strategies, one for each player, such that no player, knowing the other players' strategies in that set, could improve her expected payoff by unilaterally changing her own strategy (see Samuelson (1997, pp. 10-12) and Holt and Roth (2004) for several interpretations). An equivalent definition is the following: A Nash equilibrium is a strategy profile (i.e. one strategy for each player in the game) where every player is best responding to the strategies of the others.

Oftentimes, games have several Nash equilibria. As an example, the game depicted in Fig. 1 has three different Nash equilibria: the two strategy profiles where both players choose the same action (i.e. {A, A} and {B, B}), and a third equilibrium in *mixed* strategies, which means that players choose each action with a certain probability. In this third Nash equilibrium, both players choose action A with probability ⅔ (and action B with probability ⅓), a strategy that we denote (⅔, ⅓).

In this example, the equilibrium in mixed strategies is unsatisfactory for a number of reasons. First, since both actions can be chosen with strictly positive probability by each player, any observation of the actions actually taken by the two players would be consistent with this Nash equilibrium. Therefore, this equilibrium cannot be falsified by observing the outcome of the game. Another disappointing property of the Nash equilibrium in mixed strategies is the low payoff that players are expected to receive when they play it. In that equilibrium, outcome {B, B} would occur with probability ⅓ × ⅓, outcome {A, A} would occur with probability ⅔ × ⅔, and players would not coordinate with probability 2 × ⅓ × ⅔ , yielding a total expected payoff of ⅔ for each of them. Thus,

---

3. TGT can be divided further into cooperative and non-cooperative game theory. In *cooperative* game theory, it is assumed that players may negotiate binding agreements that can be externally enforced (by e.g. contract law). In *non-cooperative* game theory, such agreements cannot be enforced externally, so they are relevant only to the extent that abiding by them is in each individual's interest.

this equilibrium is worse than any of the other two Nash equilibria for *both* players. Finally, the mixed-strategy equilibrium does not seem to be very robust. Imagine that one of the players deviates from this equilibrium only slightly, by choosing action B with a probability marginally greater than ⅓. Then the other player's best response would be to choose action B with probability 1, and the deviator's best response to that reaction would be to play B with probability 1, too. Thus, this mixed-strategy equilibrium does not seem to be very stable.[4]

One could think that this diversity of equilibria, and the existence of the mixed-strategy equilibrium, may be partly an artifact of the fact that the game is played just once. It seems intuitive to think that if the game was played repeatedly, rational individuals would manage to coordinate in the (unique) *Pareto optimal*[5] outcome {B, B}, and the other suboptimal outcomes would not be observed in any Nash equilibrium. However, that natural intuition turns out to be wrong. To understand this, let us briefly review how repeated games are modeled in TGT.

In a *repeated game*, a certain basic game (called *stage game*) is played a number of rounds; the payoff obtained by each player in the repeated game is the sum of the (potentially discounted) payoffs obtained in each of the rounds. At any round, all the actions chosen by each of the players in previous rounds are known by everyone. A strategy in this repeated game is a complete plan of action for every possible contingency that may occur. For instance, in our coordination game, a possible strategy for the 3-round repeated game would be:

- At initial round $t = 1$, play B.
- At round $t > 1$, play B if the other player chose B at time $t - 1$. Otherwise play A.

Importantly, note that, even though the game is played repeatedly, the interaction only occurs once, since the strategies of the individuals dictate what to do in every possible history of the long game. Players could send their strategies by mail, and robots could implement them.

So, does repetition lead to sharper predictions about how rational players may interact? Not at all, rather the opposite. It turns out that when a game is repeated, the number of Nash equilibria generally multiplies, and there is a wide range of possible outcomes that can be supported by them. As an example, in our coordination game, any sequence formed by combining the three Nash equilibria of the stage game is a Nash equilibrium of the repeated game, and there are many more.[6]

The approach followed to model repeated interactions in Evolutionary Game Theory is rather different, as we explain below.

---

4. The same logic applies if one assumes that the deviation consists in choosing action B with a probability marginally *less* than ⅓. In this case, the other player's best response would be to choose action A with probability 1.

5. An outcome is Pareto optimal if it is impossible to make one player better off without making at least one other player worse off.

6. In general, any strategy profile which at every round prescribes the play of a Nash equilibrium of the stage game regardless of history is a (subgame perfect) Nash equilibrium of the repeated game. This can be easily proved using the one–shot deviation principle.

# 3. Evolutionary game theory

## 3.1. The beginnings

Some time after the emergence of traditional game theory, biologists realized the potential of game theory to formally study adaptation and coevolution of biological populations, particularly in contexts where the fitness of a phenotype depends on the composition of the population (Hamilton, 1967). The initial development of the evolutionary approach to game theory came with important changes on how the main elements of a game (i.e. players, strategies, information and payoffs) were interpreted and used:

- Players (who most often represented non-human animals) were assumed to be pre-programmed to play one given strategy, i.e. players were seen as mere carriers of a particular fixed strategy that had been genetically endowed to them and could not be changed during the course of the player's lifetime. As for the number of players, the main interest in early EGT was to study *large populations* of animals, where the actions of one single individual could not significantly affect the overall success of any strategy in the population.

- Strategies, therefore, were not assumed to be selected by players, but rather hardwired in the animals' genetic make-up. Strategies were, basically, phenotypes.

> The concept is couched in terms of a 'strategy' because it arose in the context of animal behaviour. The idea, however, can be applied equally well to any kind of phenotypic variation, and the word strategy could be replaced by the word phenotype; for example, a strategy could be the growth form of a plant, or the age at first reproduction, or the relative numbers of sons and daughters produced by a parent. Maynard Smith (1982, p. 10)

- Since strategies are not consciously chosen by players, but they are simply hardwired, information at the time of making the decision plays no significant role.

- Payoffs did not represent any order of preference, but Darwinian fitness, i.e. the expected reproductive contribution to future generations.

The main assumption underlying evolutionary thinking was that strategies with greater payoffs at a particular time would tend to spread more and thus have better chances of being present in the future. The first models in EGT, which were developed for biological contexts, assumed that this selection biased towards individuals with greater payoffs occurred at the population level, through a process of natural selection. As a matter of fact, early EGT models embraced a fairly direct interpretation of the essence of Wallace and Darwin's idea of evolution by natural selection.

> As many **more individuals of each species are born than can possibly survive**; and as, consequently, there is a frequently recurring struggle for existence, it follows that any being, if it **vary** however slightly in any manner profitable to itself, under the complex and sometimes varying conditions of life, will have a **better chance of surviving**, and thus be *naturally selected*. From the strong **principle of inheritance**, any **selected variety will tend to propagate its new and modified form**. Darwin (1859, p. 5)

The essence of this simple and groundbreaking idea could be algorithmically summarized as follows:

**IF**:

- More offspring are produced than can survive and reproduce, and
- variation within populations:
  - affects the fitness (i.e. the expected reproductive contribution to future generations) of individuals, and
  - is heritable,

**THEN**:

- evolution by natural selection occurs.

The key insight that game theory contributed to evolutionary biology is that, once the strategy distribution changes as a result of the evolutionary process, the relative fitness of the remaining strategies may also change, so previously unsuccessful strategies may turn out to be successful in the new environment, and thus increase their prevalence. In other words, the fitness landscape is not static, but it also evolves as the distribution of strategies changes.

An important concept developed in this research programme was the notion of *Evolutionarily Stable Strategy* (ESS), put forward by Maynard Smith and Price (1973) for 2-player symmetric games played by individuals belonging to the same population. Informally, a strategy **I** (for **I**ncumbent) is an ESS if and only if, when adopted by all members of a population, it enjoys a uniform invasion barrier in the sense that any other strategy **M** (for **M**utant) that could enter the population (in sufficiently low proportion) would obtain a strictly lower expected payoff in the postentry population than the incumbent strategy **I**. The ESS concept is a refinement of (symmetric) Nash equilibrium.

As an example, in the coordination game depicted in Fig. 1 both pure strategies are ESSs, but the mixed strategy ($\frac{2}{3}$ , $\frac{1}{3}$), corresponding to the symmetric Nash equilibrium in mixed strategies, is not an ESS. The intuition for this is clear: a population where every agent is playing strategy **I** = ($\frac{2}{3}$ , $\frac{1}{3}$) would be invadable by e.g. a small fraction of mutants playing action B (i.e. strategy (0,1)). This is so because the mutants would obtain the same payoff against the incumbents as the incumbents among

themselves (i.e. ⅔ on average), but a strictly greater payoff whenever they met other mutants (i.e. 2 for certain). Thus, natural selection would gradually favor the mutants over the incumbents.[7]

The basic ideas behind EGT –i.e. that strategies with greater payoffs tend to spread more, and that fitness is frequency-dependent– soon transcended the borders of biology and started to permeate many other disciplines. In economic contexts, it was understood that natural selection would derive from competition among entities for scarce resources or market shares. In other social contexts, evolution was often understood as *cultural* evolution, and it referred to dynamic changes in behavior or ideas over time (Nelson and Winter (1982), Boyd and Richerson (1985)).

## 3.2. An interpretation of evolutionary game theory where strategies are *explicitly selected* by individuals

Evolutionary ideas proved very useful to understand several phenomena in many disciplines, but –at the same time– it became increasingly clear that a *direct* application of the principles of *Darwinian* natural selection was not always appropriate for the study of (non-Darwinian) social evolution.[8] In many contexts, it seems more natural to assume that players are capable of adapting their behavior within their lifetime, occasionally revising their strategy in a way that tends to favor strategies leading to higher payoffs over strategies leading to lower payoffs. The key distinction is that, in this latter interpretation, strategies are selected at the individual level (rather than at the population level). Also, in this view of selection taking place at the individual level, payoffs do not have to represent Darwinian fitness anymore, but can perfectly well represent a preference ordering, and interpersonal comparisons of payoffs may not be needed. Following this interpretation, the algorithmic view of the process by which strategies with greater payoffs gradually displace strategies with lower payoffs would look as follows:

**IF**:

- Players using different strategies obtain different payoffs, and

- they occasionally revise their strategies (by e.g. imitation or direct reasoning over gathered information), preferentially switching to strategies that provide greater payoffs,

---

7. Note that mutants playing action A (i.e. strategy (1,0)) would also be able to invade the incumbent population.

8. As an example, note that payoffs interpreted as Darwinian fitness are added across different players to determine the relative frequency of different types of players (i.e. strategies) in succeeding generations. These interpersonal comparisons are inherent to the notion of biological evolution by natural selection, and pose no problems if payoffs reflect Darwinian fitness. However, if evolution is interpreted in cultural terms, presuming the ability to conduct interpersonal comparisons of payoffs across players may be controversial. In this link, you can watch a video that shows how unconvinced John Maynard Smith was by direct applications of the principles of *Darwinian* natural selection in Economics.

> **THEN**:
>
> - the frequency of strategies with greater payoffs will tend to increase (and this change in strategy frequencies may alter the future relative success of strategies).

In this interpretation, the canonical evolutionary model typically comprises the following elements:

- A population of agents,
- a game that is recurrently played by the agents,
- a procedure by which revision opportunities are assigned to agents, and
- a decision rule, which dictates how individual agents choose their strategy when they are given the opportunity to revise.

Note that this approach to EGT can *formally* encompass the biological interpretation, since one can always interpret the revision of a strategy as a death and birth event, rather than as a conscious decision. Having said that, it is clear that different interpretations may seem more natural in different contexts. The important point is that the framework behind the two interpretations is the same.

To conclude this chapter, let us revisit our coordination example (with payoff matrix shown in Fig. 1) in a population context. We will analyze two decision rules that lead to different results: *imitative pairwise-difference* and *best experienced payoff*.

- Under the *imitative pairwise-difference rule* (Helbing (1992), Schlag (1998)), a revising agent looks at another individual at random and considers imitating her strategy only if that strategy yields a higher expected payoff than his current strategy; in this case he switches with probability proportional to the payoff difference. It can be proved that the dynamics of this rule in large populations will tend to approach the state where every agent plays action B if the initial proportion of B-players is greater than ⅓, and will tend to approach the state where every agent plays action A if the initial proportion of B-players is less than ⅓ (Fig. 2).[9]



*Figure 2. Mean dynamic of the imitative pairwise-difference rule in a coordination game*

The following video shows some NetLogo simulations that illustrate these dynamics. In this book, we will learn to implement and analyze this model.[10]

---

9. To prove this statement, note that the mean dynamic of this decision rule is the well-known replicator dynamic (Taylor and Jonker, 1978).
10. See exercise 5 in chapter II-1 and exercise 3 in chapter II-5.

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=94#video-94-1*

*Simulation runs of the imitative pairwise-difference rule in coordination game [[1 0][0 2]]*

- Under the *best experienced payoff rule* (Osborne and Rubinstein (1998), Sethi (2000, 2021), Sandholm et al. (2019, 2020)), a revising agent tests each of the two strategies against a random agent, with each play of each strategy being against a newly drawn opponent. The revising agent then selects the strategy that obtained the greater payoff in the test, with ties resolved at random. It can be proved that the dynamics of this rule in large populations will tend to approach the state where every agent plays action B from any initial condition (other than the state where everyone plays A; see Fig. 3).[11]



*Figure 3. Mean dynamic of the best experienced rule in a coordination game.*

The following video shows some NetLogo simulations that illustrate these dynamics. In this book, we will learn to implement and analyze this model.[12]

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=94#video-94-2*

*Simulation runs of the best experienced payoff rule in coordination game [[1 0][0 2]]*

The example above shows that different decision rules can lead to very different dynamics in non-trivial ways. Both rules above tend to favor best-performing strategies, and in both the mixed-strategy Nash equilibrium is unstable. However, given an initial state where 80% of the population is playing strategy A, one of the rules will almost certainly lead the population to the state where everyone plays A, while the other rule will lead the population to the state where everyone plays B. In this book, we will learn a range of different concepts and techniques that will help us understand these differences.

---

11. This statement is a direct application of Proposition 5.11 in Sandholm et al. (2020). Izquierdo et al. (2022) prove that this dynamic leads to the efficient state in a larger class of games called single-optimum coordination games.
12. See exercise 6 in chapter II-1 and exercise 4 in chapter II-5.

Many engineering infrastructures are becoming increasingly complex to manage due to their large-scale distributed nature and the nonlinear interdependences between their components (Quijano et al., 2017). Examples include communication networks, transportation systems, sensor and data networks, wind farms, power grids, teams of autonomous vehicles, and urban drainage systems. Controlling such large-scale distributed systems requires the implementation of decision rules for the interconnected components that guarantee the accomplishment of a collective objective in an environment that is often dynamic and uncertain. To achieve this goal, traditional control theory is often of little use, since distributed architectures generally lack a central entity with access or authority over all components (Marden and Shamma, 2015).

The concepts developed in EGT can be very useful in such situations. The analogy between distributed systems in engineering and the social interactions analyzed in EGT has been formally established in various engineering contexts (Marden and Shamma, 2015). In EGT terms, the goal is to identify decision rules that will lead to desirable outcomes using limited local information only. As an example, at least in the coordination game discussed above, the *best experienced payoff* rule is more likely to lead to the most efficient outcome than the *imitative pairwise-difference* rule.

## 3.3. Take-home message

EGT is devoted to the study of the evolution of strategies in a population context where individuals repeatedly interact to play a game. Strategies are subjected to evolutionary pressures in the sense that the relative frequency of strategies which obtain higher payoffs in the population will tend to increase at the expense of those which obtain relatively lower payoffs. The aim is to identify which strategies are most likely to thrive in this "evolving ecosystem of strategies" and which will be wiped out, under different evolutionary dynamics. In this sense, note that EGT is an inherently dynamic theory.

There are two ways of interpreting the process by which strategies are selected. In biological systems, players are typically assumed to be pre-programmed to play one given strategy throughout their whole lifetime, and strategy composition changes by natural selection. By contrast, in socio-economic models, players are usually assumed capable of adapting their behavior within their lifetime, revising their strategy in a way that tends to favor strategies that provide greater payoffs at the time of revision.

Whether strategies are selected by natural selection or by individual players is rather irrelevant for the formal analysis of the system, since in both cases the interest lies in studying the evolution of

strategies. In this book, we will follow the approach which assumes that strategies are selected by individuals using a decision rule.

## 3.4. Relation with other branches

The differences between TGT and EGT are quite clear and rather obvious. TGT players are rational and forward-looking, while EGT players adapt in a fairly gradual and myopic fashion. TGT is a theory stated in terms of a one-time interaction: even if the interaction is a repeated game, this long game is played just once. In stark contrast, dynamics are at the core of EGT: the outcomes of the game shape the distribution of strategies in the population, and this change in distribution modifies the relative success of different strategies when the game is played again. Finally, TGT is mainly focused on the study of end-states and possible equilibria, paying hardly any attention to how such equilibria might be reached. By contrast, EGT is concerned with the evolution of the strategy composition in a population, which in some cases may never settle down on an equilibrium.

The branch of game theory that is closest to EGT is the Theory of Learning in Games (TLG). Like EGT, TLG abandons the demanding assumptions of TGT on players' rationality and beliefs, and assumes instead that players learn over time about the game and about the behavior of others (e.g. through reinforcement, imitation, or belief updating).

The process of learning in TLG can take many different forms, depending on the available information and feedback, and the way these are used to modify behavior. The assumptions made in these regards give rise to different models of learning. In most models of TLG, players use the history of the game to decide what action to take. In the simplest forms of learning (e.g. reinforcement or imitation) this link between acquired information and action is direct (e.g. in a stimulus-response fashion); in more sophisticated learning, players use the history of the game to form expectations or beliefs about the other players' behavior, and they then react optimally to these inferred expectations.[13]

The interpretation of EGT which assumes that players can revise their strategy is very similar to TLG. The main differences between these two branches are:

- EGT tends to study *large* populations of *small* agents, who interact *anonymously*. This implies that players' payoffs only depend on the distribution of strategies in the population, and also that any one player's actions have little or no effect on the aggregate success of any strategy at the population level. In contrast, TLG is mainly concerned with the analysis of small groups of players who repeatedly play a game among them, each of them in her particular role.

- The decision rules analyzed in EGT tend to be fairly simple and use information about the current state of the population only. By contrast, the sort of algorithms analyzed in TLG tend to be more sophisticated, and make use of the history of the game to decide what action to take.

---

13. Izquierdo et al. (2012) provide a succinct overview of some of the learning models that have been studied in TLG. For a more detailed account, see chapters 11 and 12 in Vega-Redondo (2003).

# 4. How can I learn game theory?

To learn more about game theory, we recommend the following material:

- Overviews:

  - Introductory: Colman (1995).

  - Advanced: Vega-Redondo (2003).

- Traditional game theory:

  - Introductory: Dixit and Nalebuff (2008).

  - Intermediate: Osborne (2004).

  - Advanced: Fudenberg and Tirole (1991), Myerson (1997), Binmore (2007).

- Evolutionary game theory:

  - Introductory: Maynard Smith (1982), Gintis (2009).

  - Intermediate: Hofbauer and Sigmund (2003), Sandholm (2009), Alexander (2023).

  - Advanced: Hofbauer and Sigmund (1998), Weibull (1995), Samuelson (1997), Sandholm (2010a).

  - Recent literature review: Newton (2018).

- The theory of learning in games:

  - Introductory: Vega-Redondo (2003, chapters 11 and 12).

  - Advanced: Fudenberg and Levine (1998), Young (2004).

# I-3. Introduction to agent-based modeling

In this chapter, we briefly explain what *agent-based modeling* (ABM) is about, including a paradigmatic example. We also try to clarify the relationship between evolutionary game theory and agent-based modeling, and offer some arguments in favor of implementing evolutionary models using an agent-based approach. Finally, we provide some references to other books that can be used to learn more about how to implement agent-based models.

## 1. What is agent-based modeling?

*Agent-based modeling* (ABM) is a methodology used to build formal models of real-world systems that are made up by *individual* units (such as e.g. atoms, cells, animals, people or institutions) which *repeatedly interact* among themselves and/or with their environment.

> **The essence of agent-based modeling**
>
> The defining feature of the agent-based modeling approach is that it establishes a direct and explicit correspondence
>
> - between the individual units in the target system to be modeled and the parts of the model that represent these units (i.e. the agents), and also
> - between the interactions of the individual units in the target system and the interactions of the corresponding agents in the model (figure 1).
>
> This approach contrasts with e.g. equation-based modeling, where entities of the target system may be represented via average properties or via single representative agents.

*Figure 1. In an agent-based model, the individual units of the real-world system to be modeled and their interactions are explicitly and individually represented in the model*

Thus, in an agent-based model, the individual units of the system and their repeated interactions are *explicitly* and *individually* represented in the model (Edmonds, 2001).[1] Beyond this, no further assumptions are made in agent-based modeling.

At this point, you may be wondering whether game theory is part of ABM, since in game theory players are indeed explicitly and individually represented in the models.[2] The key to answer that question is the last sentence in the box above, i.e. "*Beyond this, no further assumptions are made in agent-based modeling*". There are certainly many disciplines (e.g. game theory and cellular automata theory) that analyze models where individuals and their interactions are represented explicitly. The key distinction is that these disciplines make further assumptions, i.e. impose additional structure on their models. These additional assumptions constrain the type of models that are analyzed and, by doing so, they often allow for more accurate predictions and/or greater understanding within their (somewhat more limited) scope. Thus, when one encounters a model that fits perfectly into the framework of a particular discipline (e.g. game theory), it seems more appropriate to use the more specific name of the particular discipline, and leave the term "agent-based" for those models which satisfy the defining feature of ABM mentioned above and they do not currently fit in any more specific area of study.

The last sentence in the box also uncovers a key feature of ABM: its *flexibility*. In principle, you can make your agent-based model as complex as you wish. This has pros and cons. Adding complexity to your model allows you to study any phenomenon you may be interested in, but it also makes

---

1. These three videos by Bruce Edmonds and Michael Price and Uri Wilensky nicely describe what ABM is about.
2. The extent to which *repeated* interactions are *explicitly* represented in traditional game theoretical models is not so clear.

analyzing and understanding the model harder (or even sometimes practically impossible) using the most advanced mathematical techniques. Because of this, agent-based models are generally implemented in a programming language and explored using computer simulation. This is so common that the terms agent-based modeling and agent-based simulation are often used interchangeably. The following is a list of some features that traditionally have been difficult to analyze mathematically, and for which agent-based modeling can be useful (Epstein and Axtell, 1996):

- Agents' heterogeneity. Since agents are explicitly represented in the model, they can be as heterogeneous as the modeler deems appropriate.

- Interdependencies between processes (e.g. demographic, economic, biological, geographical, technological) that have been traditionally studied in different disciplines, and are not often analyzed together. There is no restriction on the type of rules that can be implemented in an agent-based model, so models can include rules that link disparate aspects of the world that are often studied in different disciplines.

- Out-of-equilibrium dynamics. Dynamics are inherent to ABM. Running a simulation consists in applying the rules that define the model over and over, so agent-based models almost invariably include some notion of time within them. Equilibria are never imposed a priori: they may emerge as an outcome of the simulation, or they may not.

- The micro-macro link. ABM is particularly well suited to study how global phenomena emerge from the interactions among individuals, and also how these emergent global phenomena may constrain and shape back individuals' actions.

- Local interactions and the role of physical space. The fact that agents and their environment are represented explicitly in ABM makes it particularly straightforward and natural to model local interactions (e.g. via networks).

As you can imagine, introducing any of the aspects outlined above in an agent-based model often means that the model becomes mathematically intractable, at least to some extent. However, in this book we will learn that, in many cases, there are various aspects of agent-based models that can be analytically solved, or described using formal approximation results. Our view is that the most useful agent-based models lie at the boundaries of theoretical understanding, and help us push these boundaries. They are advances sufficiently small so that simplified versions of them (or certain aspects of their behaviour) can be fully understood in mathematical terms –thus retaining its analytical rigour–, but they are steps large enough to significantly extend our understanding beyond what is achievable using the most advanced mathematical techniques available.

> In my personal (albeit biased) view, the best simulations are those which just peek over the rim of theoretical understanding, displaying mechanisms about which one can still obtain causal intuitions. Probst (1999)

# 2. What is an agent?

In this book we will use the term *agent* to refer to a distinct part of our (computational) model that is meant to represent a decision-maker. Agents could represent human beings, non-human animals, institutions, firms, etc. The agents in our models will always play a game, so in this book we will use the term agent and the term player interchangeably.

Agents have *individually-owned variables*, which describe their internal state (e.g. a strategy), and are able to conduct certain computations or tasks, i.e. they are able to run *instructions* (e.g. to update their strategy). These instructions are sometimes called decision rules, or rules of behavior, and most often imply some kind of interaction with other agents or with the environment.

The following are some of the individually-owned variables that the agents we are going to implement in this book may have:

- strategy (a number)
- payoff (a number)
- my-coplayers (the set of agents with whom this agent plays the game)
- color (the color of this agent)

And the following are examples of instructions that the agents in most of our models will be able to run:

- to play (play a certain game with my-coplayers and set the payoff appropriately)
- to update-strategy (revise strategy according to a certain decision rule)
- to update-color (set color according to strategy)

# 3. A paradigmatic example

In this section we present a model that captures the spirit of ABM. The model implements the main features of a family of models proposed by Sakoda (1949, 1971) and –independently– by Schelling (1969, 1971, 1978).[34] Specifically, here we present a computer implementation put forward by Edmonds and Hales (2005).[5]

In this model there are 133 blue agents and 133 orange agents who live in a 2-dimensional grid made

---

3. Hegselmann (2017) provides a detailed and fascinating account of the history of this family of models.

4. Our implementation is not a precise instance of neither Sakoda's nor Schelling's family of models, because unhappy agents in our model move to a *random* location. We chose this migration regime to make the code simpler. For details, see Hegselmann (2017, footnote 124). A faithful NetLogo implementation of the model described in Schelling (1971, pp. 154-158), which also includes other migration regimes, can be run online in this link (Izquierdo et al., 2022).

5. Izquierdo et al. (2009, appendix B) analyze this model as a Markov chain.

up of 20×20 cells (figure 2). Agents are initially located at random on the grid. The neighborhood of a cell is defined by the eight neighboring cells (i.e. the eight cells which surround it).[6]
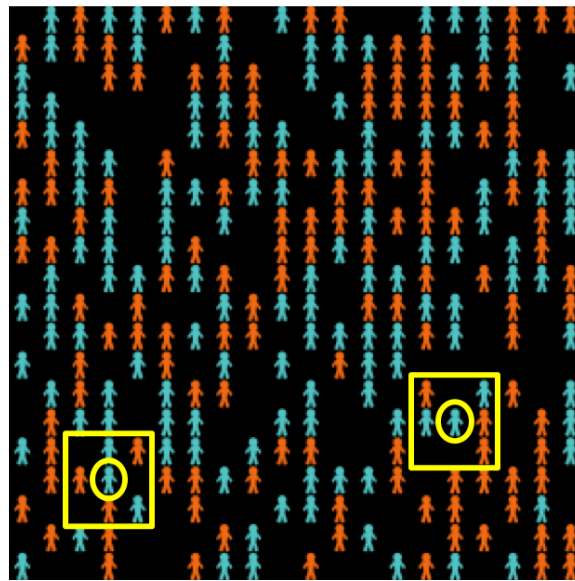


*Figure 2. Grid of Schelling–Sakoda model (20×20), with 133 blue agents, 133 orange agents. The agents in yellow circles have 2 out of 5 neighbors of the same color*

Agents may be happy or unhappy. An agent is happy if the proportion of other agents of its same colour in its neighbourhood is greater or equal than a certain threshold (*%-similar-wanted*), which is a parameter of the model; otherwise the agent is said to be unhappy. Agents with no neighbors are assumed to be happy regardless of the value of *%-similar-wanted*. In each iteration of the model one unhappy agent is randomly selected to move to a random empty cell in the lattice.

As an example, the two agents surrounded by a circle in figure 2 have 2 out of 5 neighbors of the same color as them, i.e. 40%. This means that in simulation runs where *%-similar-wanted* ≤ 40% these agents would be happy, and would not move. On the other hand, in simulations where *%-similar-wanted* > 40% these two agents would move to a random location.

---

6. Cells on a side have five neighbors and cells at a corner have three neighbors.

Now imagine that we simulate a society where agents require at least 60% of their neighbors to be of the same color as them in order to be happy (i.e. *%-similar-wanted* = 60%). These are pretty strong segregationist preferences, so one would expect a fairly clear pattern of spatial segregation at the end. The following video shows a representative run. You may wish to run the simulations yourself downloading this model's code.

*One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=40#video-40-1*

*Simulation run of Schelling–Sakoda model with %-similar-wanted = 60*

As expected, the final outcome of the simulation shows clearly distinctive ghettos. To measure the level of segregation of a certain spatial pattern we define a global variable named *avg-%similar*, which is the average proportion (across agents) of an agent's neighbors that are the same color as the agent. Extensive Monte Carlo simulation shows that a good estimate of the expected *avg-%similar* is about 95.7% when *%-similar-wanted* is 60%.

What is really surprising is that even with only mild segregationist preferences, such as *%-similar-wanted* = 40%, we still obtain fairly segregated spatial patterns (expected *avg-%similar* ≈ 82.7%). The following video shows a representative run.

*Simulation run of Schelling-Sakoda model with* %-similar-wanted *= 40*

And even with segregationist preferences as weak as *%-similar-wanted* = 30% (i.e. you are happy unless strictly less than 30% of your neighbors are of the same color as you), the emergent spatial patterns show significant segregation (expected *avg-%similar* ≈ 74.7%). The following video shows a representative run.

*Simulation run of Schelling-Sakoda model with* %-similar-wanted *= 30*

So this agent-based model illustrates how strong spatial segregation can result from only weakly segregationist preferences (e.g. trying to avoid an acute minority status). This model has been enriched in a number of directions (e.g. to include heterogeneity between and within groups),[7] but the implementation discussed here is sufficient to illustrate a non-trivial phenomenon that emerges from agents' individual choices and their interactions.

# 4. Agent-based modeling and evolutionary game theory

Given that models in Evolutionary Game Theory (EGT) comprise many *individuals* who repeatedly *interact* among them and occasionally revise their *individually-owned* strategy, it seems clear that agent-based modelling is certainly an appropriate methodology to build EGT models. Therefore, the question is whether other approaches may be more appropriate or convenient. This is an important issue, since nowadays most models in EGT are equation-based, and therefore –in general– more amenable to mathematical analysis than agent-based models. This is a clear advantage for equation-based models. Why bother with agent-based modeling then?

The reason is that mathematical tractability often comes at a price: equation-based models tend to incorporate several assumptions that are made solely for the purpose of guaranteeing mathematical tractability. Examples include assuming that the population is infinite, or assuming that revising agents are able to evaluate strategies' expected payoffs. These assumptions are clearly made for

---

7. See Aydinonat (2007).

mathematical convenience, since there are no infinite populations in the real world, and –in general– it seems more natural to assume that agents' choices are based on information obtained from experiences with various strategies, or from observations of others' experiences. Are assumptions made for mathematical convenience harmless? We cannot know unless we study models where such assumptions are not made. And this is where agent-based modelling can play an important role.

Agent-based modeling gives us the potential to build models closer to the real-world systems that we want to study, because in an agent-based model we are free to choose the sort of assumptions that we deem appropriate in purely scientific terms. We may not be able to fully analyze all aspects of the resulting agent-based model mathematically, but we will certainly be able to explore it using computer simulation, and this exploration can help us assess the impact of assumptions that are made only for mathematical tractability. In this way, we will be able to shed light on questions such as: how large must a population be for the mathematical model to be a good description of the dynamics of the finite-population model? and, how much do dynamics change if agents cannot evaluate strategies' expected payoffs with infinite precision?

# 5. How can I learn about agent-based modeling?

A wonderful classic book to learn the fundamental concepts and appreciate the kind of models you can build using ABM is Epstein and Axtell (1996). In this short book, the authors show how to build an artificial society where agents, using extremely simple rules, are able to engage in a wide range of activities such as sex, cultural exchange, trade, combat, disease transmission, etc. Epstein and Axtell's (1996) interdisciplinary book shows how complex patterns can emerge from very simple rules of interaction.

Epstein and Axtell's (1996) seminal book focuses on the fundamental concepts without discussing any code whatsoever. The following books are also excellent introductions to scientific agent-based modeling, and all of them make use of NetLogo: Gilbert (2007), Railsback and Grimm (2019), Wilensky and Rand (2015), Janssen (2020) and Smaldino (2023). Hamill and Gilbert (2016) discuss the implementation of several NetLogo models in the context of Economics. Most of these models are significantly more sophisticated than the ones we implement and analyze in this book.

# I-4. Introduction to NetLogo

This chapter gives an overview of NetLogo (Wilensky, 1999), the modeling platform that we use in this book to implement agent-based evolutionary models. Here we explain that NetLogo:

- is easy to learn,

- powerful,

- very well documented,

- allows the user to interact with the model while it is running,

- includes a tool (i.e., *BehaviorSpace*) that makes the automatic exploration of the parameter space of any model very easy,

- is open-source and free,

- can be run on multiple platforms and online,

- has an active and helpful community of users,

- there are excellent resources that can be used to learn it,

- includes several extensions that extend its basic functionalities,

- can be used to conduct experiments with real people, and

- can be linked with other software like R, Python, Mathematica or Matlab.

## 1. What is NetLogo?

> NetLogo is a well-written, easy-to-install, easy-to-use, easy-to-extend, and easy-to-publish-online environment. The entry level is simple enough and the tutorials provided in the package are straightforward and clear enough that anyone who can read and is comfortable using a keyboard and mouse could create their own models in a short time, with little or no additional instruction. Sklar (2007, p. 7)

NetLogo is a modeling environment designed for coding and running agent-based simulations.[1] Nowadays, there are many languages and software platforms that can be employed to create agent-

---

1. NetLogo was created by Uri Wilensky and is under continuous development at the Northwestern's Center for Connected Learning and Computer-Based Modeling. It is also important to acknowledge Seth Tisue, who "*worked meticulously to guarantee the quality of the NetLogo software*" (Wilensky and Rand, 2015, p. xxii) as lead developer for over a decade.

based models,[2] and at the time of writing NetLogo is the most widely used. We recommend NetLogo and will use it throughout this book for the many reasons we outline below.

## Easy to learn

> NetLogo stands out as the quickest to learn and the easiest to use. Gilbert (2007, p. 49)

The language used to code models within NetLogo –which is also called NetLogo– has been designed following a "Low Threshold, No Ceiling" philosophy (Wilensky and Rand, 2015). All reviews of the software highlight how easy it is to learn. To be concrete, we would estimate that an average scholar without previous coding experience can learn the basics of the language and be in a position to write a simple agent-based model after 2-4 days of work. Someone with programming experience could reduce the estimated time to 1-2 days.

One characteristic that makes the NetLogo language easy to learn is that it is remarkably close to natural language. As a matter of fact, NetLogo language could perfectly be used as pseudo-code to communicate algorithms implemented in other languages.

> Since NetLogo was designed to be easily readable, we believe that NetLogo code is about as easy to read as any pseudo-code we would have used. NetLogo also has the big advantage over pseudo-code of being executable, so the user can run and test the examples. (Wilensky and Rand, 2015, p. xiv)

NetLogo language is definitely simpler to use than e.g. Java or Objective-C, and can often reduce programming efforts significantly when compared with other languages.

## Powerful

> NetLogo has become a standard platform for agent-based simulation, yet there appears to be widespread belief that it is not suitable for large and complex models due to slow execution. Our experience does not support that belief. Railsback et al. (2017, abstract)

---

2. To our knowledge, the most up-to-date and comprehensive review of agent-based simulation software has been conducted by Abar et al. (2017), who compare 85 tools using a convenient tabular and chart format, and deem NetLogo both *easy to use* and also *appropriate to execute medium/large-scale simulations*. Another recent review that assesses and compares NetLogo with other platforms has been published by Kravari and Bassiliades (2015). There is also a wikipedia page set up by Nikolai and Madey (2009) which provides an up-to-date comparison of agent-based software toolkits. Finally, it is also possible to code agent-based models using general-purpose programming languages directly. In the context of evolutionary game theory, Isaac (2008) convincingly demonstrates how this can be easily done with Python.

NetLogo is powerful in that it can accommodate reasonably large and complex simulations, and its execution speed is more than acceptable for most purposes. NetLogo can easily run simulations with several tens of thousands of agents.

## Excellent documentation

> NetLogo is by far the most professional platform in its appearance and documentation. Railsback et al. (2006, p. 613)

One of the reasons why NetLogo is so easy to learn is that it is very well documented. The user manual includes three tutorials to help beginners get started, an excellent programming guide, and a comprehensive dictionary with the definitions of all NetLogo primitives, including examples of how to use them. NetLogo also comes with an extensive library of models from different disciplines (e.g. art, biology, chemistry, computer science, mathematics, networks, philosophy, physics, psychology, and other social sciences) and several code examples which succinctly illustrate particular features and coding techniques.

## Possibility to interact with the model at runtime

NetLogo is designed to allow the user to interact with the model during runtime in a variety of ways:

- By modifying parameter values at runtime, with immediate effect on the simulation. This feature is very convenient to assess the impact of different assumptions in the model and conduct exploratory work.

- By running commands in the middle of a run to e.g. create new agents, remove others, or make a subset of them take some action.

- By probing agents to see –and potentially set– the value of any of their individually-owned variables at any time.

## Automatic exploration of parameter space

NetLogo includes a software tool named BehaviorSpace (Wilensky and Shargel, 2002) which greatly facilitates running a model many times, systematically varying the desired parameter values, and keeping a record of the results of each run. Besides, computational experiments set up with BehaviorSpace can be run from the command line, i.e. without having to open NetLogo's graphical user interface. This feature is particularly useful for launching large-scale experiments in computer clusters.

## Open-source and free

NetLogo can be downloaded for free at http://ccl.northwestern.edu/netlogo/. Its source code is publicly hosted on GitHub at https://github.com/NetLogo/NetLogo, where users can open issues to request the implementation of new features or to report bugs.

## Multiplatform and online execution of models

NetLogo can run on Windows, Mac or Linux. Most modern computers will run NetLogo without any trouble. It can also be used online through NetLogo Web. NetLogo Web can also be used to create stand-alone versions of NetLogo models in HTML format. These self-contained versions can be run in any browser without having to install any software.[3]

## Great support and active user community

NetLogo developers are always happy to receive feedback and enhancement requests (at feedback@ccl.northwestern.edu), and bug reports (at bugs@ccl.northwestern.edu). There is also an active community of NetLogo users who post their questions and help each other at the NetLogo Forum, the NetLogo-Users Google group and on StackOverflow.

## Abundance of quality resources

At https://ccl.northwestern.edu/netlogo/resources.shtml you can find plenty of quality resources to learn NetLogo. These include references to textbooks, papers that make use of NetLogo, courses given at middle schools, high schools and Universities all around the world, competitions and tutorials. There are also many video tutorials on YouTube.

> This reviewer, who has used NetLogo for both research and teaching at several levels, highly recommends it for instructors from elementary school to graduate school and for researchers from a wide range of fields. Sklar (2007, p. 8)

## Extensions to fulfill specialised needs

Extensions are add-ons that extend the NetLogo language with new primitives created to fulfill specialised needs. Some of these extensions come bundled with NetLogo, some have been created by NetLogo developers but must be downloaded separately, and others have been created by third parties. Four representative examples of useful extensions that come with NetLogo are:

- The rnd extension, which provides efficient primitives to make random weighted selections, with or without replacement.
- The nw extension, which adds many primitives to generate networks, compute several network-related metrics, and import and export network data.
- The matrix extension, which adds a matrix data structure to NetLogo and several primitives to operate with it.
- The GIS (Geographic Information Systems) extension.

---

3. This can be done by uploading any NetLogo model to NetLogo Web and exporting it as HTML.

## Useful to conduct experiments with real people and for participatory modeling

The NetLogo release includes HubNet (Wilensky and Stroup, 1999), a technology that enables users to communicate and interact with each other through NetLogo. Thus, Hubnet can be very useful to run participatory simulations and experiments, in which human users can be part of the simulation and interact among themselves and with artificial agents.

## Happy to link with other software

> NetLogo is now a powerful tool widely used in science and we recommend it strongly, especially for those new to modeling and programming but also for serious scientists with software experience. Lytinen and Railsback (2012)

NetLogo can be linked with advanced software tools like R (R Core Team, 2019), Python (Python Software Foundation, 2019), Mathematica (Wolfram Research, Inc., 2019) or Matlab (The MathWorks, Inc., 2019). Specifically, using an R package called RNetLogo (Thiele (2014); Thiele et al. (2012a, 2012b, 2014)), it is possible to run and control NetLogo models from R, execute NetLogo commands, and obtain any information from a NetLogo model. The connector PyNetLogo (Jaxa-Rozen and Kwakkel, 2018) provides the same functionalty for Python, and the so-called Mathematica link (Bakshy and Wilensky, 2007) for Mathematica. The Mathematica link comes bundled as part of the latest NetLogo releases.

Conversely, one can also call R, Python and Matlab commands from within NetLogo using the R-Extension (Thiele and Grimm, 2010), the NetLogo Python extension (Head, 2018) and MatNet (Biggs and Papin, 2013) respectively.

# 2. How to learn NetLogo

To make the most of this book, we recommend you get familiar with the NetLogo environment and with NetLogo programming. This will normally take from a few hours to a couple of days, depending on your programming skills, and can be accomplished doing the following tasks:

- Download and install NetLogo following the instructions at https://ccl.northwestern.edu/netlogo/. In this book we will be using NetLogo version 6.4.0.[4]

- Go through the three tutorials in the NetLogo user manual, i.e.

  ◦ Tutorial #1: Models

---

4. Please, make sure you download version 6.1.0 or greater. NetLogo syntax changed significantly in version 6.0, and a little bit in 6.1.0. Version 6.4.0 introduced many new features in BehaviorSpace, but we will not be using these new features in this book.

- Tutorial #2: Commands

- Tutorial #3: Procedures

After having gone through the previous material, you will have obtained the required NetLogo background to follow this text without any problems. In the next chapter we review the main concepts of NetLogo and give an overview of the structure of most NetLogo models, using the Schelling-Sakoda model as an illustration.

# I-5. The fundamentals of NetLogo

This chapter provides a succinct overview of the fundamentals of NetLogo. It is strongly based on the excellent NetLogo user manual, version 6.4.0 (Wilensky, 2023). By no means do we claim originality on the content of this chapter; all credit should go to Uri Wilensky and his team. The following table provides links to the different aspects of NetLogo programming that we cover here.

| Very basics | More advanced | Final polishing |
|---|---|---|
| The three tabs | Ask | Consistency within procedures |
| Types of agents | Lists | Breeds |
| Instructions | Agentsets | Ticks and Plotting |
| Variables | Synchronization | Skeleton of many NetLogo models |
| | | The code for Schelling-Sakoda model |

Feel free to skip this chapter if you are already familiar with NetLogo. For future reference, you may wish to download our NetLogo quick guide, which is a 6-page pdf file containing the main concepts outlined here.

# 1. The three tabs

The main window of NetLogo contains three tabs, i.e. the interface tab, the info tab and the code tab (see figure 1).
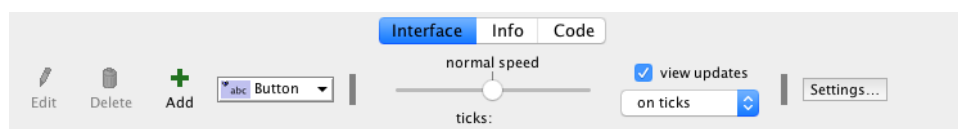


*Figure 1. Top bar of the NetLogo Interface tab, where you can select the tab you want to see*

**The Interface tab** is used to run the model. It often contains buttons, sliders, switches, plots... Most models include a button labeled setup, which is used to initialize the model, and another button labeled go, which is used to run the model.

**The Info tab** can be used to include the documentation of the model.

Finally, **the Code tab** contains most of the code of the model. We say *most* because in some models part of the code is included within the plots in the interface tab.

# 2. Types of agents
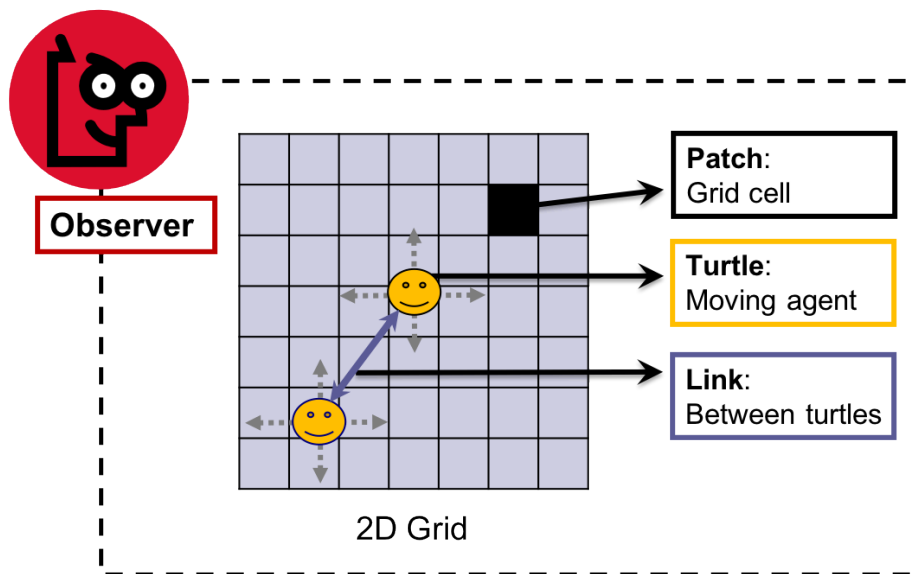


*Figure 2. The NetLogo world is made up of turtles, patches, links and the observer*

The NetLogo world is made up by four types of agents (see figure 2), i.e.:

- **Turtles**. Turtles are agents that can move.
- **Patches**: The NetLogo world is two-dimensional and is divided up into a grid of patches. Each patch is a square piece of "ground" over which turtles can move.
- **Links**: Links are agents that connect two turtles. Links can be directed (from one turtle to another turtle) or undirected (one turtle with another turtle).
- **The observer**: There is only one observer and it does not have a location. You can think of the observer as the conductor of the whole NetLogo orchestra.

Note that in many descriptions of agent-based models, the word *agent* is used only to refer to the turtles (i.e. the *mobile* agents in NetLogo), while patches and links are not considered *agents* (and the observer is not even mentioned). However, when reading NetLogo documentation, it is important to remember that these four types of entities are all agents in NetLogo.

# 3. Instructions

Instructions tell agents what to do. Three characteristics are useful to remember about instructions:

- Whether the instruction is **implemented by the user** (*procedures*), **or** whether it is **built into NetLogo** (*primitives*). Once you define a procedure, you can use it elsewhere in your program. The NetLogo Dictionary has a complete list of built-in instructions (i.e. primitives). The following code is an example of the implementation of procedure to setup:

```
to setup              ;; comments are written after semicolon(s)
  clear-all           ;; clear everything
  create-turtles 10   ;; make 10 new turtles
end                   ; (one semicolon is enough, but I like two)
```

The instruction to setup is a procedure (since it is implemented by us), whereas `clear-all` and `create-turtles` are both primitives (they are built into NetLogo).

Note that primitives are nicely colored, and you can click on them and press F1 to see their syntax, functionality, and examples. You may want to copy and paste the code above to see all this for yourself.

- Whether the instruction produces an **output** (*reporters*) **or not** (*commands*).

  ◦ A reporter computes a result and **reports** it. Most reporters are nouns or noun phrases (e.g. "average-wealth", "most-popular-girl"). These names are preceded by the keyword `to-report`. The keyword `end` marks the end of the instructions in the procedure.

```
to-report average-wealth          ;; this reporter returns the
  report mean [wealth] of turtles ;; average wealth in the
end                               ;; population of turtles
```

  ◦ A command is an action for an agent to carry out. Most commands begin with verbs (e.g. "create", "die", "jump", "inspect", "clear"). These verbs are preceded by the keyword `to` (instead of `to-report`). The keyword `end` marks the end of the procedure.

```
to go
  ask turtles [
    forward 1       ;; all turtles move forward one step
    right random 360 ;; and turn a random amount
  ]
end
```

Note that primitive commands are colored in `blue` while primitive reporters are colored in `purple`. Keywords are colored in `green`.

- Whether the instruction takes an **input** (or several inputs) **or not**. Inputs are values that the instruction uses in carrying out its actions.

```
to-report absolute-value [number]     ;; number is the input
  ifelse number >= 0            ;; if number is already non-negative
    [ report number ]           ;; return number (a non-negative value).
    [ report (- number) ]       ;; Otherwise, return the opposite, which
end                             ;; is then necessarily positive.
```

# 4. Variables

Variables are places to store values (such as numbers). A variable can be a *global* variable, a *turtle* variable, a *patch* variable, a *link* variable, or a *local* variable (local to a procedure). To change the value of a variable you can use the `set` command. If you don't set the variable to any value, it starts out storing a value of zero.

- **Global variables**: If a variable is a global variable, there is only one value for the variable, and every agent can access it. You can declare a new global variable either *in the Interface tab* –by adding a switch, a slider, a chooser or an input box– *or in the Code tab* –by using the `globals` keyword at the beginning of your code, like this:

```
globals [ n-of-strategies ]
```

- **Turtle, patch, and link variables**: Each turtle has its own value for every turtle variable, each patch has its own value for every patch variable, and each link has its own value for every link variable. Turtle, patch, and link variables can be *built-in* or *defined by the user*.

    - **Built-in** variables: For example, all turtles and all links have a `color` variable, and all patches have a `pcolor` variable. If you set this variable, the corresponding turtle, link or patch changes color. Other built-in turtle variables are `xcor`, `ycor`, and `heading`. Other built-in patch variables include `pxcor` and `pycor`. Other built-in link variables are `end1`, `end2`, and `thickness`. You can find the complete list in the NetLogo Dictionary.

    - **User-defined** turtle, patch and link variables: You can also define new turtle, patch or link variables using the `turtles-own`, `patches-own`, and `links-own` keywords respectively, like this:

```
turtles-own [ energy ]    ;; each turtle has its own energy
patches-own [ roughness ] ;; each patch has its own roughness
links-own   [ weight ]    ;; each link has its own weight
```

- **Local variables**: A local variable is defined and used only in the context of a particular procedure or part of a procedure. To create a local variable, use the `let` command. You can use this command anywhere. If you use it at the top of a procedure, the variable will exist throughout the procedure. If you use it inside a set of square brackets, for example inside an `ask`, then it will exist only inside those brackets.

```
to swap-colors [turtle1 turtle2]  ;; turtle1 and turtle2 are inputs
  let temp ([color] of turtle1)   ;; store the color of turtle1 in temp
  ask turtle1 [ set color ([color] of turtle2) ]
        ;; set turtle1's color to turtle2's color
  ask turtle2 [ set color temp ]
        ;; now set turtle2's color to turtle1's (original) color
end     ;; (which was conveniently stored in local variable "temp").
```

## Setting and reading the value of variables

Global variables can be read and set at any time by any agent. Every agent has direct access to her own variables, both for reading and setting. Sometimes you will want an agent to read or set a different agent's variable; to do that, you can use `ask` (which is explained in further detail later):

```
ask turtle 5 [ show color ]      ;; turtle 5 shows its color
ask turtle 5 [ set color blue ]  ;; turtle 5 becomes blue
```

You can also use `of` to make one agent read another agent's variable. `of` is written in between the variable name and the relevant agent (i.e. [*reporter*] `of` *agent*). Example:

```
show [color] of turtle 5 ;; observer shows turtle 5's color
```

Finally, a turtle can read and set the variables of the patch it is standing on directly, e.g.

```
ask turtles [ set pcolor red ]
```

The code above causes every turtle to make the patch it is standing on red. (Because patch variables are shared by turtles in this way, you cannot have a turtle variable and a patch variable with the same name –e.g. that is why we have `color` for turtles and `pcolor` for patches).

# 5. Ask

NetLogo uses the `ask` command to specify instructions that are to be run by turtles, patches or links. Usually, the observer uses `ask` to ask all turtles or all patches to run commands. Here's an example of the use of `ask` syntax in a NetLogo procedure:

```
to setup
  clear-all            ;; clear everything
  create-turtles 100   ;; create 100 new turtles with random heading
  ask turtles [        ;; ask them
    set color red      ;; to turn red and
    forward 50         ;; to move 50 steps forward
  ]
  ask patches [        ;; ask patches
    if (pxcor > 0) [   ;; with pxcor greater than 0
      set pcolor green ;; to turn green
    ]
  ]
end
```

You can also use `ask` to have an individual turtle, patch or link run commands. The reporters `turtle`, `patch`, `link`, and `patch-at` are useful for this technique. For example:

```
to setup
  clear-all              ;; clear the world
  create-turtles 3       ;; make 3 turtles
  ask turtle 0 [ fd 10 ] ;; tell the first one to go forward 10 steps
  ask turtle 1 [         ;; ask the second turtle (with who number 1)
    set color green      ;; ... to become green
  ]
  ask patch 2 -2 [       ;; ask the patch at (2,-2)...
    set pcolor blue      ;; ... to become blue
  ]
  ask turtle 0 [            ;; ask the first turtle (with who number 0)
    create-link-to turtle 2 ;; to link to turtle with who number 2
  ]
  ask link 0 2 [         ;; ask the link between turtle 0 and 2...
    set color blue       ;; ... to become blue
  ]
  ask turtle 0 [         ;; ask the turtle with who number 0
    ask patch-at 1 0 [   ;; ... to ask the patch to her east
      set pcolor red     ;; ... to become red
    ]
  ]
end
```

# 6. Lists

In the simplest models, each variable holds only one piece of information, usually a number or a string. Lists let you store multiple pieces of information in a single variable by collecting those pieces of information in a list. Each value in the list can be any type of value: a number, a string, an agent, an agentset, or even another list.

## Constant lists

You can make a list by simply putting the values you want in the list between brackets, e.g.:

```
set my-list [2 4 6 8]
```

## Building lists on the fly

If you want to make a list in which the values are determined by reporters, as opposed to being a series of constants, use the `list` reporter. The `list` reporter accepts two other reporters, runs them, and reports the results as a list.

```
set my-random-list list (random 10) (random 20)
```

To make shorter or longer lists, you can use the `list` reporter with fewer or more than two inputs, but in order to do so, you must enclose the entire call in parentheses, e.g.:

```
show (list random 10)
show (list (random 10) (turtle 3) "a" 30)  ;; inner () are not necessary
```

The `of` primitive lets you construct a list from an agentset (i.e. a set of agents). It reports a list containing each agent's value for the given reporter (syntax: [*reporter*] `of` *agentset*).

```
set fitness-list ([fitness] of turtles)
        ;; list containing the fitness of each turtle (in random order)
show [pxcor * pycor] of patches
```

See also: `n-values`, `range`, `sentence` and `sublist`.

## Reading and changing list items

List items can be accessed using `first`, `last` and `item`. The first element of a list is item 0. Technically, lists cannot be modified, but you can construct new lists based on old lists. If you want the new list to replace the old list, use `set`. For example:

```
set my-list [2 7 5 "Bob" [3 0 -2]]
  ;; my-list is now [2 7 5 "Bob" [3 0 -2]]
set my-list replace-item 2 my-list 10
  ;; my-list is now [2 7 10 "Bob" [3 0 -2]]
```

See also: `but-first`, `but-last`, `fput`, `lput`, `length`, `shuffle`, `position` and `remove-item`.

## Iterating over lists

To apply a function (procedure) on each item in a list, you can use `foreach` or `map`. The function to be applied is usually defined using anonymous procedures, with the following syntax:

```
[ [input-1 input-2 ...] -> code of the procedure ]
  ;; this syntax was different in versions before NetLogo 6.0
```

The names assigned to the inputs of the procedure (i.e. *input-1* and *input-2* above) may be used within the code of the procedure just like you would use any other variable within scope. You can use any name you like for these local variables (complying with the usual restrictions). An example of an anonymous procedure that implements the absolute value is:

```
[ [x] -> abs x ]  ;; you can use any symbol instead of x
[  x  -> abs x ]  ;; if there is just one input
                  ;; you do not need the square brackets
```

`foreach` is used to run a command on each item in a list. It takes as inputs the list and the command to be run on each element of the list, e.g.:

```
foreach [1.2 4.6 6.1] [ n -> show (word n " rounded is " round n) ]
  ;; output: "1.2 rounded is 1" "4.6 rounded is 5" "6.1 rounded is 6"
```

`map` is similar to `foreach`, but it is a reporter (it returns a list). It takes as inputs a list and a reporter; and returns an output list containing the results of applying the reporter to each item in the input list. As in `foreach`, procedures can be anonymous.

```
map [ element -> round element ] [1.2 2.2 2.7]        ;; returns [1 2 3]
```

Simple uses of `foreach`, `map`, `n-values`, and related primitives can be written more concise.

```
map round [1.2 2.2 2.7]
  ;; (see Anonymous procedures in Programming Guide)
```

Both `foreach` and `map` can take multiple lists as input; in that case, the procedure is run once for the first items of all input lists, once for the second items, and so on.

```
(map [[el1 el2] -> el1 + el2] [1 2 3] [10 20 30])  ;; returns [11 22 33]
(map + [1 2 3] [10 20 30])              ;; a shorter way of writing the same
```

See also: `reduce`, `filter`, `sort-by`, `sort-on`, and `-> (anonymous procedure)`.

# 7. Agentsets

An agentset is a set of agents; all agents in an agentset must be of the same type (i.e. turtles, patches, or links). An agentset is not in any particular order. In fact, it's always in a random order.[1] What's powerful about the agentset concept is that you can construct agentsets that contain only some agents. For example, all the *red* turtles, or the patches with positive `pxcor`, or all the links departing from a certain agent. These agentsets can then be used by `ask` or by various reporters that take agentsets as inputs, such as `one-of`, `n-of`, `with`, `with-min`, `max-one-of`, etc. The primitive `with` and its siblings are very useful to build agentsets. Here are some examples:

```
turtles with [color = red]        ;; all red turtles
patches with [pxcor > 0]          ;; patches with positive pxcor
[my-out-links] of turtle 0        ;; all links outgoing from turtle 0
turtles in-radius 3               ;; all turtles three or fewer patches away
other turtles-here with-min [size] ;; other turtles with min size on my patch
(patch-set self neighbors4)       ;; von Neumann neighborhood of a patch
```

Once you have created an agentset, here are some simple things you can do:

- Use `ask` to make the agents in the agentset do something.
- Use `any?` to see if the agentset is empty.
- Use `all?` to see if every agent in an agentset satisfies a condition.
- Use `count` to find out exactly how many agents are in the set.

Here are some more complex things you can do:

---

1. If you want agents to do something in a fixed order, you can make a list of the agents instead.

```
ask one-of turtles [ set color green ]
        ;; one-of reports a random agent from an agentset
ask (max-one-of turtles [wealth]) [ donate ]
        ;; max-one-of agentset [reporter] reports an agent in the
        ;; agentset that has the highest value for the given reporter
show mean ([wealth] of turtles with [gender = male])
        ;; Use of to make a list of values,
        ;; one for each agent in the agentset.
show (turtle-set turtle 0 turtle 2 turtle 9 turtles-here)
        ;; Use turtle-set, patch-set and link-set reporters to make new
        ;; agentsets by gathering together agents from a variety of sources
show (turtles with [gender = male]) = (turtles with [wealth > 10])
        ;; Check whether two agentsets are equal using = or !=
show member? (turtle 0) turtles with-min [wealth]
        ;; Use member? to see if an agent is a member of an agentset.
if all? turtles [color = red]     ;; use all? to see if every agent in the
  [ show "every turtle is red!" ] ;; agentset satisfies a certain condition
ask turtles [
  create-links-to other turtles-here ;; on same patch as me, not me,
    with [color = [color] of myself] ;; and with same color as me.
]
show [([color] of end1) - ([color] of end2)] of links
        ;; check everything's OK
```

# 8. Synchronization

When you ask a set of agents to run more than one command, each agent must finish all the commands in the block before the next agent starts. One agent runs all the commands, then the next agent runs all of them, and so on. As mentioned before, the order in which agents are chosen to run the commands is random. To be clear, consider the following code:

```
ask turtles [
  forward random 10 ;; move forward a random number of steps (0—9)
  wait 0.5          ;; wait half a second
  set color blue    ;; set your color to blue
]
```

The first (randomly chosen) turtle will move forward some steps, she will then wait half a second, and she will finally set her color to blue. Then, and only then, another turtle will start doing the same; and so on until all turtles have run the commands inside ask without being interrupted by any other turtle. The order in which turtles are selected to run the commands is random. If you want all turtles to move, and then all wait, and then all become blue, you can write it this way:

```
ask turtles [ forward random 10 ]
ask turtles [ wait 0.5 ]          ;; note that you will have to wait
ask turtles [ set color blue ]    ;; (0.5 * number-of-turtles) seconds
```

Finally, you can make agents execute a set of commands in a certain order by converting the agentset into a list. There are three primitives that help you do this: `sort`, `sort-by` and `sort-on`.

```
set my-list-of-agents sort-by [[t1 t2] -> [size] of t1 < [size] of t2] turtles
 ;; This sets my-list-of-agents to a list of turtles sorted in
 ;; ascending order by their turtle variable size. For simple orderings
 ;; like this, you can use sort-on, e.g.: sort-on [size] turtles
foreach my-list-of-agents [ ag ->
    ask ag [                ;; each agent undertakes the list of commands
    forward random 10   ;; (forward, wait, and set) without being
    wait 0.5            ;; interrupted, i.e. the next agent does not
    set color blue      ;; start until the previous one has finished.
  ]
]
```

# 9. Consistency within procedures

Some primitives in NetLogo can only be run by a certain type of agent. For instance, `forward` can only be run by turtles, since turtles are the only type of agent that can move. An easy way of knowing which type of agent can run a certain primitive is to find the primitive in the NetLogo Dictionary and look at the icon beneath the name of the primitive. If you click on `forward`, you will see the icon, which denotes turtles. The icons for the other types of agent are: for the observer, for patches, and for links. There are primitives that can be run by more than one type of agent. For instance, reporter `turtles-here` can be run by turtles and by patches.

The question that naturally comes to mind now is: How do we tell NetLogo what type of agent should run a certain procedure (which *we* implement)? The answer is simple: we don't. NetLogo infers that from the code of the procedure; we just have to be consistent. An example of inconsistency would be to code a procedure containing two primitives that can be run *only* by two different types of agents, as in the following example:

```
to setup
  create-turtles 10
  forward 1
end
```

If we implement this code, we obtain the following error message: "You can't use FORWARD in an observer context, because FORWARD is turtle-only" (see figure 3).
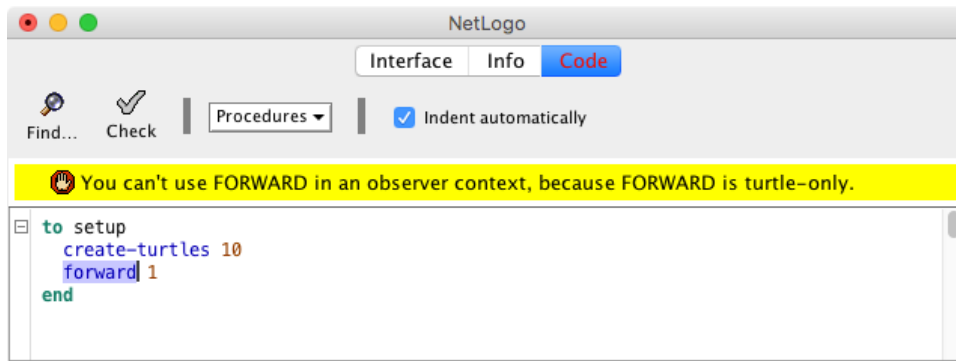
*Figure 3. Inconsistency error*

The reason is that NetLogo reads the primitive `create-turtles` and, since it can only be run by the observer, NetLogo infers that the procedure to setup will be run only by the observer, i.e. everything inside is in an observer context. Then, NetLogo reads the primitive `forward`, which can only be run by turtles, and throws the error.

We would obtain similar inconsistency errors if we tried to access individually-owned variables within procedures that can only be run by a type of agent that cannot access those variables, as in the following examples.

```
to setup
  create-turtles 10
  show xcor
end
;; Here we would obtain the error:
;; "You can't use XCOR in an observer context, because XCOR is turtle-only"
```

```
to setup
  create-turtles 10
  show pxcor
end
;; Here we would obtain the error:
;; "You can't use PXCOR in an observer context,
;; because PXCOR is turtle/patch-only"
```

Note that in the example above, NetLogo says that `pxcor` is "turtle/patch-only". This is because all patch variables can be directly accessed by any turtle standing on the patch (see section Variables above).

```
to setup
  create-turtles 10
  show end1
end
;; Here we would obtain the error:
;; "You can't use END1 in an observer context, because END1 is link-only"
```

# 10. Breeds

NetLogo allows you to have different types of turtles and different types of links. There are called breeds. Here we discuss breeds of turtles only, since breeds of links follow the same logic. Breeds are defined with the syntax:

```
breed [plural-name singular-name]
```

For instance, to define a breed of sellers and a breed of buyers, we would type the following at the top of our code:

```
breed [sellers seller]
breed [buyers buyer]
```

From then onwards, we could assign different individually-owned variables to each of the breeds, using the keywords `sellers-own` and `buyers-own`. Also, there are a number of primitives that are automatically added to the NetLogo language once you have defined a breed, such as `create-sellers`, `hatch-sellers`, `sprout-sellers`, `sellers-here`, `sellers-at`, `sellers-on`, and `is-seller?`.

# 11. Ticks and Plotting

In most NetLogo models, time passes in discrete steps called "ticks". NetLogo includes a built-in tick counter so you can keep track of how many ticks have passed. The current value of the tick counter is shown above the view. Note that –since NetLogo 5.0– ticks and plots are closely related.

You can write code inside the plots. Every plot and each of its pens have *setup* and *update* **code fields** where you can write commands. All these fields must be edited directly in each plot –i.e. in the interface, not in the code tab. To execute the commands written inside the plots, you can use `setup-plots` and `update-plots`, which run the corresponding fields in every plot and in every pen. However, in models that use the tick counter, these two primitives are not normally used because they are automatically triggered by tick-related commands, as explained below.

To use the tick counter, first you must `reset-ticks`; this command resets the tick counter to zero, sets up all plots (i.e. triggers `setup-plots`), and then updates all plots (i.e. triggers `update-plots`); thus, the initial state of the world is plotted. Then, you can use the `tick` command, which advances the tick counter by one and updates all plots.

See also: `plot`, `plotxy`, and `ticks`.

# 12. Skeleton of many NetLogo models

In most NetLogo models there are two basic procedures that are run by the observer: to setup and to go.

Procedure to setup is run just once at the beginning of the simulation, most often by clicking a button in the interface tab. In this procedure:

- we initialize the model from scratch using the primitive `clear-all`,
- we set up all initial conditions (this often implies creating several agents), and
- we finish with the primitive `reset-ticks`.

Procedure to go contains all the actions that will be executed repeatedly in the model. Some of these actions will be executed directly by the observer, while others will be run by the turtles, the patches or the links. In any case, procedure to go is run by the observer, so it is the observer who must ask the other agents to run the appropriate instructions, using the primitive `ask`. Most often, procedure to go contains the primitive tick, which advances the (discrete) NetLogo clock in one unit.

```
globals  [ … ]      ;; global variables (also defined with sliders, …)
turtles-own [ … ]  ;; user-defined turtle variables (also <breeds>-own)
patches-own [ … ]  ;; user-defined patch variables
links-own [ … ]    ;; user-defined link variables (also <link-breeds>-own)
…

to setup
  clear-all

  …
  setup-patches  ;; procedure where patches are initialized

  …
  setup-turtles  ;; procedure where turtles are created

  …
  reset-ticks
end
…

to go
  conduct-observer-procedure

  …
  ask turtles [conduct-turtle-procedure]

  …
  ask patches [conduct-patch-procedure]

  …
  tick ;; this will update every plot and every pen in every plot
end
…

to-report a-particular-statistic

  …
  report the-result-of-some-formula
end
```

# 13. The code for Schelling-Sakoda model

To conclude this chapter, we present some simple code that implements the Schelling-Sakoda model described in chapter I-3 "Introduction to agent-based modeling". The code we show here is simpler than the one used for the videos in chapter I-3, which is more efficient but less readable.[2] In the interface, we have used two sliders to define parameters *number-of-agents* and *%-similar-wanted* (see figure 4).



*Figure 4. Interface of a simple version of Schelling–Sakoda model*

---

2. Both implementations lead to exactly the same dynamics.

The code that goes in the code tab is shown below. You can download the whole model here (schelling-sakoda-simple.nlogo) and take this code as a test to check whether you are ready to proceed to the next chapter. If you can understand most of it, you are definitely prepared!

To work your way through the code, you will most likely have to use the NetLogo Dictionary intensively, and run small pieces of code in the Command Center (especially because the model includes several NetLogo primitives that we have not seen yet). You can also inspect individual turtles and make them run (turtle) instructions such as:

```
ask turtles-on neighbors [set label
"Hi!"]
```

You will have to type these instructions on the bottom line of the window that pops up when you inspect a turtle (see figure 5). To inspect a turtle, right-click on it, select the name of the turtle (e.g. turtle 21), and click on "inspect". Alternatively, you can just type the following instruction in the command center:

```
inspect turtle 21
```



*Figure 5. Window that pops up when you inspect a turtle. You can ask the turtle to execute instructions by typing them on the bottom line*

Developing these skills will be useful, since programming in NetLogo most often involves looking up the dictionary very often and testing short snippets of code. Once you have understood most of the code below we can start building our first agent-based evolutionary model in the next chapter!

```
;;;;;;;;;;;;;;;;;;
;;; VARIABLES ;;;
;;;;;;;;;;;;;;;;;;

turtles-own [
  happy?
]


;;;;;;;;;;;;;;;;;;;;;;;;;
;;; SETUP PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;

to setup
  clear-all
  setup-agents
  reset-ticks
end
to setup-agents
  set-default-shape turtles "person"
  ask n-of number-of-agents patches
    [ sprout 1 [set color cyan] ]
  ask n-of (number-of-agents / 2) turtles
    [ set color orange ]
  ask turtles [update-happiness]
end


;;;;;;;;;;;;;;;;;;;;;;
;;; MAIN PROCEDURE ;;;
;;;;;;;;;;;;;;;;;;;;;;

to go
  if all? turtles [happy?] [stop]
  ask one-of turtles with [not happy?] [move]
  ask turtles [update-happiness]
  tick
end


;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; TURTLES' PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;

to move
  move-to one-of patches with [not any? turtles-here]
end
to update-happiness
  let my-nbrs (turtles-on neighbors)
  let n-of-my-nbrs (count my-nbrs)
  let similar-nbrs (count my-nbrs with [color = [color] of myself])
  set happy? similar-nbrs >= (%-similar-wanted * n-of-my-nbrs / 100)
end
```

# PART II. OUR FIRST AGENT-BASED EVOLUTIONARY MODEL

# II-1. Our very first model

## 1. Goal

The goal of this chapter is to create our first agent-based evolutionary model in NetLogo. Being our first model, we will keep it simple; nonetheless, the model will already contain the four building blocks that define most models in agent-based evolutionary game theory, namely:

- a population of agents,

- a game that is recurrently played by the agents,

- a procedure that determines how strategy revision opportunities are assigned to agents, and

- a decision rule, which specifies how individual agents update their (pure) strategies when they are given the opportunity to revise.

In particular, in our model the number of (individually-represented) agents in the population will be chosen by the user. These agents will repeatedly play a symmetric 2-player 2-strategy game, each time with a randomly chosen counterpart. The payoffs of the game will be determined by the user. Agents will revise their strategy with a certain probability, also to be chosen by the user. The decision rule these agents will use is called *imitate-if-better*, which dictates that a revising agent imitates the strategy of a randomly chosen player, if this player obtained a payoff greater than the revising agent's.

This fairly general model will allow us to explore a variety of specific questions, like the one we outline next.

## 2. Motivation. Cooperation in social dilemmas

There are many situations in life where we have the option to make a personal effort that will benefit others beyond the personal cost incurred. This type of behavior is often termed "to cooperate", and can take a myriad forms: from paying your taxes, to inviting your friends over for a home-made dinner. All these situations, where cooperating involves a personal cost but creates net social value, exhibit the somewhat paradoxical feature that individuals would prefer not to pay the cost of cooperation, but everyone prefers the situation where everybody cooperates to the situation where no one does. Such counterintuitive characteristic is the defining feature of social dilemmas, and life is full of them (Dawes, 1980).

The essence of many social dilemmas can be captured by a simple 2-person game called the Prisoner's Dilemma. In this game, the payoffs for the players are: if both cooperate, R (Reward); if both defect, P (Punishment); if one cooperates and the other defects, the cooperator obtains S (Sucker) and the defector obtains T (Temptation). The payoffs satisfy the condition T > R > P > S. Thus, in a Prisoner's Dilemma, both players prefer mutual cooperation to mutual defection (R > P), but two

motivations may drive players to behave uncooperatively: the temptation to exploit (T > R), and the fear to be exploited (P > S).

Let us see a concrete example of a Prisoner's Dilemma. Imagine that you have $1000, which you may keep for yourself, or transfer to another person's account. This other person faces the same decision: she can transfer her $1000 money to you, or else keep it. Crucially, whenever money is transferred, the money doubles, i.e. the recipient gets $2000.

> Try to formalize this situation as a game, assuming you and the other person only care about money.

The game can be summarized using the payoff matrix in Fig. 1. To see that this game is indeed a Prisoner's Dilemma, note that transferring the money would be what is often called "to cooperate", and keeping the money would be "to defect".

|  |  | **Player 2** | |
|---|---|---|---|
|  |  | Keep | Transfer |
| **Player 1** | Keep | **1000 , 1000** | **3000 , 0** |
|  | Transfer | **0 , 3000** | **2000 , 2000** |

*Figure 1. Payoff matrix of a Prisoner's Dilemma game*

To explore whether cooperation may be sustained in a simple evolutionary context, we can model a population of agents who repeatedly play the Prisoner's Dilemma. Agents are either cooperators or defectors, but they can occasionally revise their strategy. A revising agent looks at another agent in the population and, if the observed agent's payoff is greater than the revising agent's payoff, the revising agent copies the observed agent's strategy. Do you think that cooperation will be sustained in this setting? Here we are going to build a model that will allow us to investigate this question… and many others!

# 3. Description of the model

In this model, there is a population of *n-of-players* agents who repeatedly play a symmetric 2-player 2-strategy game. The two possible strategies are labeled 0 and 1. The *payoffs* of the game are determined by the user in the form of a matrix $[[A_{00}\ A_{01}]\ [A_{10}\ A_{11}]]$, where $A_{ij}$ is the payoff that an agent playing strategy $i$ obtains when meeting an agent playing strategy $j$ ($i, j \in \{0, 1\}$).

Initially, the number of agents playing strategy 1 is a (uniformly distributed) random number between 0 and the number of players in the population. From then onwards, the following sequence of events –which defines a tick– is repeatedly executed:

1. Every agent obtains a payoff by selecting another agent at random and playing the game.

2. With probability *prob-revision*, individual agents are given the opportunity to revise their strategies. The decision rule –called ***imitate if better***– reads as follows:[1]

> Look at another (randomly selected) agent and adopt her strategy if and only if her payoff was greater than yours.

The model shows the evolution of the number of agents choosing each of the two possible strategies at the end of every tick.
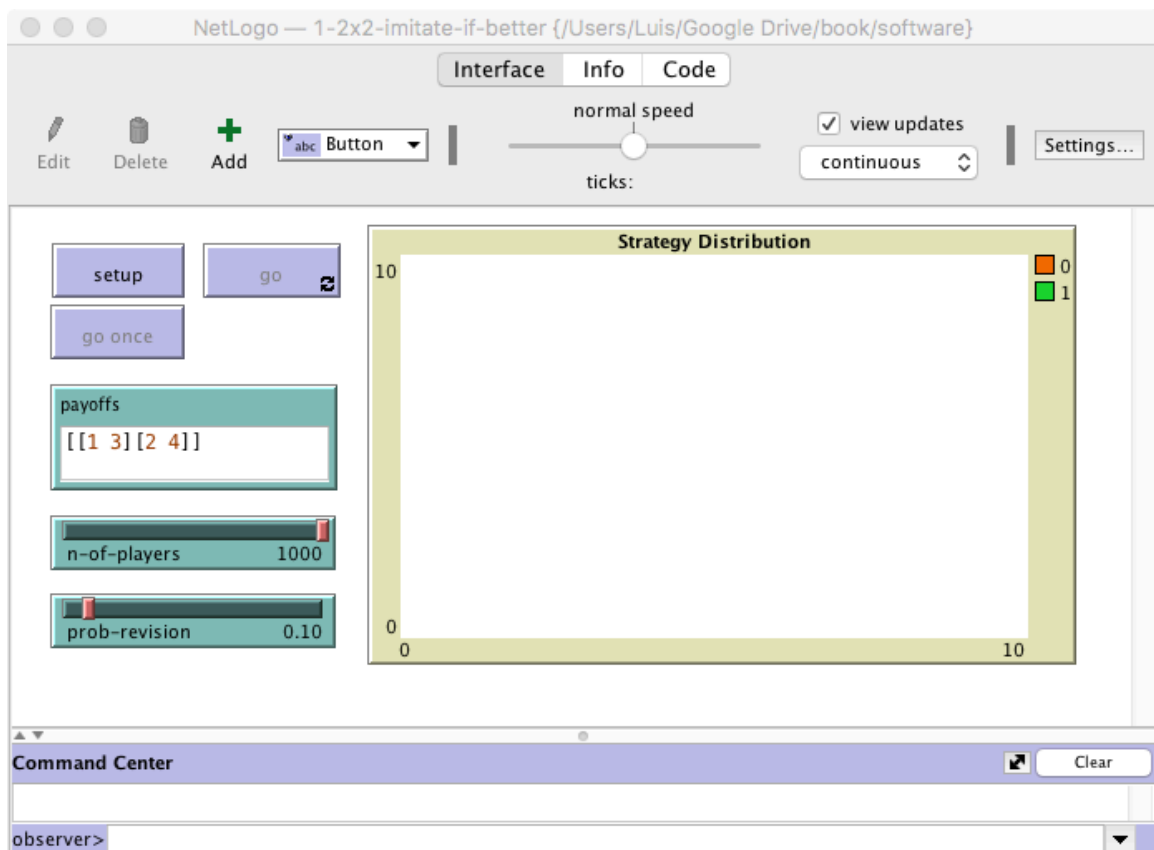
# 4. Interface design



*Figure 2. Interface design*

The interface (see figure 2) includes:

- Three buttons:

    1. One button named setup, which runs the procedure to setup.

    2. One button named go once, which runs the procedure to go.

---

1. This rule has been studied by Izquierdo and Izquierdo (2013) and Loginov (2021). Loginov (2021) calls this rule "imitate-the-better-realization".

3. One button named go, which runs the procedure to go indefinitely.

---

In the Code tab, write the procedures to setup and to go, without including any code inside for now.

---

```
to setup
  ;; empty for now
end

to go
  ;; empty for now
end
```

---

In the Interface tab, create a button and write setup in the "commands" box. This will make the procedure to setup run whenever the button is pressed.

Create another button for the procedure to go (i.e., write go in the commands box) with display name go once to emphasize that pressing the button will run the procedure to go just once.

Finally, create another button for the procedure to go, but this time tick the "forever" option. When pressed, this button will make the procedure to go run repeatedly until the button is pressed again.

---

- A slider to let the user select the number of players.

---

Create a slider for global variable *n-of-players*. You can choose limit values 2 (as the minimum) and 1000 (as the maximum), and an increment of 1.

---

- An input box where the user can write a string of the form [ [$A_{00}$ $A_{01}$] [$A_{10}$ $A_{11}$] ] containing the payoffs $A_{ij}$ that an agent playing strategy $i$ obtains when meeting an agent playing strategy $j$ ($i, j \in \{0, 1\}$).

---

Create an input box with associated global variable *payoffs*. Set the input box type to "String (reporter)". Note that the content of *payoffs* will be a string (i.e. a sequence of characters) from which we will need to extract the payoff numeric values.

---

- A slider to let the user select the probability of revision.

> Create a slider with associated global variable *prob-revision*. Choose limit values 0 and 1, and an increment of 0.01.

- A plot that will show the evolution of the number of agents playing each strategy.

> Create a plot and name it Strategy Distribution. Since we are not going to use the 2D view (i.e. the large black square in the interface) in this model, you may want to overlay it with the newly created plot.

# ▶ 5. Code

## 5.1. Initial skeleton of the code

Figure 3 below provides a schematic view of the code. We will use these code skeletons in many chapters of this book. You can find the legend for them in Appendix A-2.
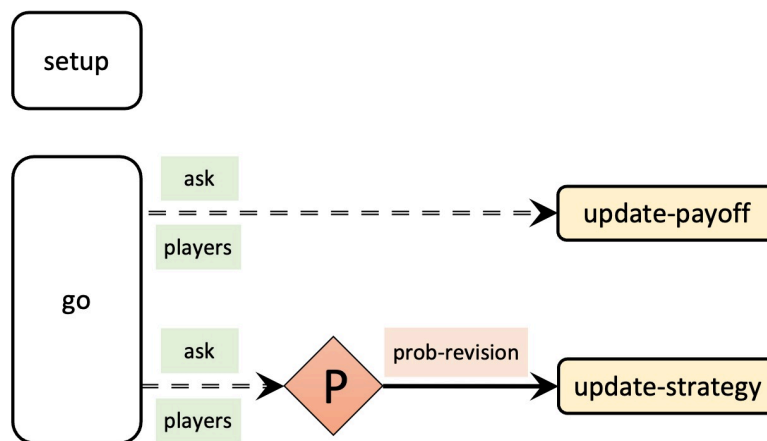


*Figure 3. Initial (and naive) skeleton of the code*

## 5.2. Global variables and individually-owned variables

First we declare the global variables that we are going to use and we have not already declared in the interface. We will be using a global variable named payoff-matrix to store the payoff values on a list, so the first line of code in the Code tab will be:

```
globals [payoff-matrix]
```

Next we declare a breed of agents called "players". If we did not do this, we would have to use the default name "turtles", which may be confusing to newcomers.

```
breed [players player]
```

Individual players have their own strategy (which can be different from the other agents' strategy) and their own payoff, so we need to declare these *individually-owned variables* as follows:

```
players-own [
  strategy
  payoff
]
```

## 5.3. Setup procedures

In the setup procedure we want:

- To clear everything up. We initialize the model afresh using the primitive `clear-all`:

  ```
  clear-all
  ```

- To transform the string of characters the user has written in the *payoffs* input box (e.g. "[[1 2][3 4]]") into a list (of 2 lists) that we can use in the code (e.g. [[1 2][3 4]]). This list of lists will be stored in the global variable named payoff-matrix. To do this transformation (from string to list, in this case), we can use the primitive `read-from-string` as follows:

  ```
  set payoff-matrix read-from-string payoffs
  ```

- To create *n-of-players* players and set their individually-owned variables to an appropriate initial value. At first, we set the value of payoff and strategy to 0:[2]

  ```
  create-players n-of-players [
    set payoff 0
    set strategy 0
  ]
  ```

  Note that the primitive `create-players` does not appear in the NetLogo dictionary; it has been automatically created after defining the breed "players". Had we not defined the breed "players", we would have had to use the primitive `create-turtles` instead.

  Now we will ask a random number of players (between 0 and *n-of-players*) to set their strategy to 1, using one of the most important primitives in NetLogo, namely `ask`. The instruction will be of the form:

---

2. By default, user-defined variables in NetLogo are initialized with the value 0, so there is no actual need to explicitly set the initial value of individually-owned variables to 0, but it does no harm either.

```
    ask AGENTSET [set strategy 1]
```

where AGENTSET should be a random subset of players.

To randomly select a certain number of agents from an agentset (such as players), we can use the primitive n-of (which reports another –usually smaller– agentset):

```
    ask (n-of SIZE players) [set strategy 1]
```

where SIZE is the number of players we would like to select.

Finally, to generate a random integer between 0 and *n-of-players* we can use the primitive random:

```
    random (n-of-players + 1)
```

The resulting instruction will be:

```
    ask n-of (random (n-of-players + 1)) players [set strategy 1]
```

- To initialize the tick counter. At the end of the setup procedure, we should include the primitive reset-ticks, which resets the tick counter to zero (and also runs the "plot setup commands", the "plot update commands" and the "pen update commands" in every plot, so the initial state of the model is plotted):

```
    reset-ticks
```

Thus, the code up to this point should be as follows:

```
globals [
  payoff-matrix
]

breed [players player]

players-own [
  strategy
  payoff
]

to setup
  clear-all
  set payoff-matrix read-from-string payoffs
  create-players n-of-players [
    set payoff 0
    set strategy 0
  ]
  ask n-of random (n-of-players + 1) players [set strategy 1]
  reset-ticks
```

```
    end

    to go

    end
```

## 5.4. Go procedure

The procedure to go contains all the instructions that will be executed in every tick. In this particular model, these instructions include

1.  asking all players to interact with another (randomly selected) player to obtain a payoff, and

2.  asking all players to revise their strategy with probability *prob-revision*.

To keep things nice and modular, we will create two separate procedures *to be run by players* named to update-payoff and to update-strategy. Procedure to update-payoff will update the payoff of the player running the procedure, while procedure to update-strategy will be in charge of updating her strategy. Writing short procedures with meaningful names will make our code elegant, easy to understand, easy to debug, and easy to extend... so we should definitely aim for that.

We now have to think very carefully about the order in which we are going to ask the players to update-payoff and to update-strategy. Should players update their strategies at the same time (i.e. synchronously), or sequentially (i.e. asynchronously)? Arguably, this is something that is not absolutely clear in the description of the model above. We chose to describe the model in that (admittedly ambiguous) way because it is common that the relative order in which agents run their actions is not absolutely clear in model descriptions. However, this is a very important issue with significant consequences, and it already takes some expertise even to only notice that a model description is ambiguous. The following sections will help us develop this expertise.

### A naive implementation

Let us start with the implementation that seems to follow the model description most closely, and which corresponds with the initial skeleton of the code shown in figure 3. First, all agents update their payoff; then, all agents update their strategies with probability *prob-revision*. Procedure to go would then look as follows:

```
ask players [update-payoff]
ask players [
  if (random-float 1 < prob-revision) [update-strategy]
]
```

Note that condition

```
    (random-float 1 < prob-revision)
```

will be true with probability *prob-revision*.

The implementation of procedure to go shown above seems natural and straightforward. However, it is faulty in a subtle but crucial way. The implementation above would do something that –most likely– the designer of the model did not intend.

To see this, think of the first agent who revises her strategy and changes it (in procedure to update-strategy). This agent would have her strategy changed, but her payoff would not change after the revision (because payoffs are only modified in procedure to update-payoff). Thus, her payoff would still correspond to a game played with her old strategy, i.e., her strategy before the revision took place. If, after this first revision, a second agent runs procedure to update-strategy and –having looked at the first agent's payoff– decides to imitate the first agent, this second agent will imitate the first agent's *new* strategy, which is a strategy that was not used to obtain the payoff on which the imitation is based.

Thus, with this first (and naive) implementation, some strategies may be imitated based on payoffs that have not been obtained with those strategies. In the following section we propose an implementation that solves this issue.

## A more functional implementation. Synchronous updating within the tick

In this section we propose an implementation of procedure to go that:

- guarantees that any imitation of a strategy is based on the payoff obtained with that strategy, and

- in our opinion, corresponds best with the description of the model above. Arguably, the description above seems to imply that revising agents within the tick update their strategies simultaneously.

To make sure that revising players within the tick update their strategies simultaneously, we need players to be able to compute their revised strategy, but they also must be able to keep their old one until all players have had the opportunity to revise their strategy. Therefore, we are going to need two *individually-owned* variables: one named strategy (for the strategy used to compute the payoffs), and another one named strategy-after-revision (for the strategy that agents will adopt after their revision). Thus, we are going to have to add the following line:

```
players-own [
  strategy
  strategy-after-revision ;; <== new line
  payoff
]
```

Similarly, we will also need two different procedures to be run by individual players: one named to update-strategy-after-revision (where agents will update their strategy-after-revision), and another one named to update-strategy (where agents will update their strategy, with the value of their strategy-after-revision). Procedure to update-strategy should be executed only after all agents have finished revising, to make sure that any imitation of any strategy is based on the payoff obtained with that strategy.

This is a major (and necessary) change to the skeleton of the code. The new skeleton of the code is shown in figure 4, which we recommend comparing with the initial (and faulty) skeleton shown in figure 3.
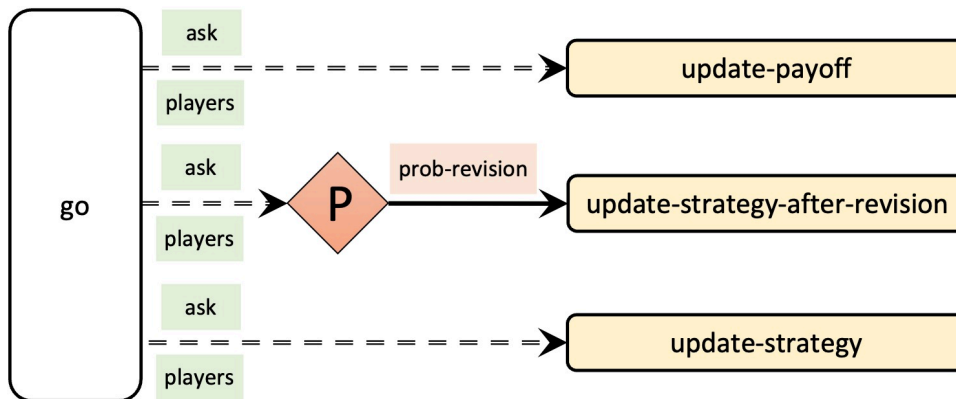


*Figure 4. Skeleton of the code for synchronous updating within the tick*

In terms of code, the implementation of procedure to go for synchronous updating within the tick would be as follows:

```
ask players [update-payoff]
ask players [
  if (random-float 1 < prob-revision) [
    update-strategy-after-revision
  ]
]
ask players [update-strategy]
```

Note that the last line of code above (where players update their strategy with the value of their strategy-after-revision and do nothing else in between) effectively implies that we update every agent's strategy at the same time *within the tick*, i.e., revisions are synchronous within the tick. Thus, with this implementation, the value of *prob-revision* allows us to control the fraction of agents who revise their strategies simultaneously, i.e. under exactly the same information.

Finally, having the agents go once through the code above will mark an evolution step (or generation), so, to keep track of these cycles and have the plots in the interface automatically updated at the end of each cycle, we include the primitive `tick` at the end of to go.

```
tick
```

## 5.5 Other procedures

### to update-payoff

Procedure to update-payoff is the procedure where agents update their payoff. Importantly, note that procedure to update-payoff *will be run by a particular player*. Thus, within the code of this procedure, we can access and set the value of player-owned variables strategy and payoff.

Here we want the player running this procedure (let us call her the running player) to play with some other player and get the corresponding payoff.[3] First, we will (randomly) select a counterpart and store it in a local variable named mate:

```
let mate one-of other players
```

Now we need to compute the payoff that the running player will obtain when she plays the game with her mate. This payoff is an element of the payoff-matrix list, which is made up of two sublists (e.g., [[1 2][3 4]]).

Note that the first sublist (i.e., `item 0 payoff-matrix`) corresponds to the case in which the running player plays strategy 0. We want to consider the sublist corresponding to the player's strategy, so we type:

```
item strategy payoff-matrix
```

In a similar fashion, the payoff to extract from this sublist is determined by the strategy of the running player's mate (i.e., [strategy] of mate). Thus, the payoff obtained by the running agent is:

```
item ([strategy] of mate) (item strategy payoff-matrix)
```

Finally, to make the running agent store her payoff, we can write:

```
set payoff item ([strategy] of mate) (item strategy payoff-matrix)
```

This line of code concludes the definition of the procedure to update-payoff.

### to update-strategy-after-revision

In this procedure, which is also *to be run by individual players*, we want the running player to look at some other random player (which we will call the observed-agent) and, if the payoff of the observed-agent is greater than her own payoff, set the value of her variable strategy-after-revision to the observed-agent's strategy. We do not want the revising agent to set the value of her variable strategy yet because, as explained above, this would imply that some imitations could then be based on the wrong payoffs.

To select a random player and store it in the local variable observed-agent, we can write:

```
let observed-agent one-of other players
```

---

3. In some evolutionary models, it is assumed that players are randomly *matched* in pairs to play the game. That would lead to a slightly different distribution of payoffs (especially for low population sizes). In exercise 7, we ask you to think about the changes we would have to make in our current code to model this random matching.

To compare the payoffs and, if appropriate, set the value of the revising agent's strategy-after-revision to the observed-agent's strategy, we can write:

```
if ([payoff] of observed-agent) > payoff [
  set strategy-after-revision ([strategy] of observed-agent)
]
```

This concludes the definition of the procedure to update-strategy-after-revision.

### to update-strategy

In this procedure, the running player will just update her variable strategy with the value of her variable strategy-after-revision. The code is particularly simple:

```
to update-strategy
  set strategy strategy-after-revision
end
```

## 5.6. Code in the plots

Finally, let us set up the plot to show the number of agents playing each strategy. This is something that can be done directly on the plot, in the Interface tab.

> Edit the plot by right-clicking on it, choose a color and a name for the pen showing the number of agents with strategy 0, and in the "pen update commands" area write:
>
> ```
> plot count players with [strategy = 0]
> ```

> Add a second pen to show the number of players with strategy 1.

## 5.7. Final fix

In principle, we have finished our model but, unfortunately, we have a small mistake. If you run our code now, you will see that something weird seems to happen on the first tick (see figure 5). Too many agents seem to change their strategies on the first tick, even if *prob-revision* is set to 0! Can you figure out what is going on? This is a tricky bug, but no-one said that the life of a rigorous agent-based modeler was going to be easy. Here we do not give medals for free. You gotta earn them! 😀
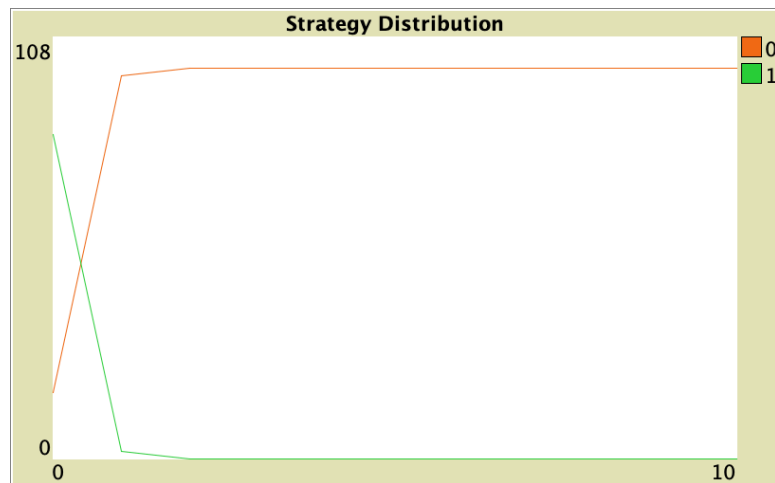
*Fig. 5. Simulation run with the current code*

### What is going on?

Well done if you ventured an answer! (even if your answer was wrong).

The problem with the current code is that we did not explicitly initialize the agents' variable strategy-after-revision, and in NetLogo, by default, user-defined variables are initialized with the value 0. Then, on the first tick, every agent will run procedure to update-strategy (even if they do not happen to run procedure to update-strategy-after-revision before), so many agents will (incorrectly) set their strategy to 0.

To fix this problem, we just have to properly initialize agents' variable strategy-after-revision when we create them, at procedure to setup:

```
to setup
  clear-all
  set payoff-matrix read-from-string payoffs
  create-players n-of-players [
    set payoff 0
    set strategy 0
  ]
  ask n-of random (n-of-players + 1) players [set strategy 1]
  ask players [set strategy-after-revision strategy]
    ;; the line above is needed to guarantee that agents
    ;; keep their initial strategy until
    ;; they revise their strategy for the first time.
    ;; Note that all agents will set their strategy to
    ;; strategy-after-revision at the end of procedure to go.
  reset-ticks
end
```

This concludes the definition of all the code in the Code tab, which by now should look as shown below.

## 5.8. Complete code in the Code tab

```
globals [
  payoff-matrix
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]

to setup
  clear-all
  set payoff-matrix read-from-string payoffs
  create-players n-of-players [
    set payoff 0
    set strategy 0
  ]
  ask n-of random (n-of-players + 1) players [set strategy 1]
  ask players [set strategy-after-revision strategy]
  reset-ticks
end

to go
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
end

to update-payoff
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end

to update-strategy-after-revision
  let observed-player one-of other players
  if ([payoff] of observed-player) > payoff [
    set strategy-after-revision ([strategy] of observed-player)
  ]
end

to update-strategy
```

```
    set strategy strategy-after-revision
  end
```

# 6. Sample runs

Now that we have the model, we can investigate the question we posed at the motivation above. Let strategy 0 be "Defect" and let strategy 1 be "Cooperate". We can use *payoffs* [[1 3][0 2]]. Note that we could choose any other numbers (as long as they satisfy the conditions that define a Prisoner's Dilemma), since our decision rule only depends on ordinal properties of payoffs. Let us set *n-of-players* = 100 and *prob-revision* = 0.1, but feel free to change these values.

If you run the model with these settings, you will see that in nearly all runs all agents end up defecting in very little time.[4] The video below shows some representative runs.

*One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=18#video-18-1*

Note that at any population state, defectors will tend to obtain a greater payoff than cooperators, so they will be preferentially imitated. Sadly, this drives the dynamics of the process towards overall defection.

# 7. Exercises

You can use the following link to download the complete NetLogo model: 2×2-imitate-if-better.nlogo.

---

4. All simulations will necessarily end up in one of the two absorbing states where all agents are using the same strategy. The absorbing state where everyone defects (henceforth D-state) can be reached from any state other than the absorbing state where everyone cooperates (henceforth C-state). The C-state can be reached from any state with at least two cooperators, so –in principle– any simulation with at least two agents using each strategy could end up in either absorbing state. However, it is overwhelmingly more likely that the final state will be the D-state. As a matter of fact, one single defector is extremely likely to be able to invade a whole population of cooperators, regardless of the size of the population.

**Exercise 1.** Consider a coordination game with payoffs [[3 0][0 2]] such that both players are better off if they coordinate in one of the actions (0 or 1) than if they play different actions. Run several simulations with 1000 players and probability of revision 0.1. (You can easily do that by leaving the button go pressed down and clicking the setup button every time you want to start again from random initial conditions.)



*Picture by Caleb Whiting*

Do simulations end up with all players choosing the same action? Does the strategy with a greater initial presence tend to displace the other strategy? How does changing the payoff matrix to [[30 0][0 2]] make a difference on whether agents coordinate on 0 or strategy 1?

P.S. You can explore this model's (deterministic) mean dynamic approximation with this program.

**Exercise 2.** Consider a Stag hunt game (Skyrms, 2001) with payoffs [[3 0][2 1]] where strategy 0 is "Stag" and strategy 1 is "Hare". Does the strategy with greater initial presence tend to displace the other strategy?

P.S. You can explore this model's (deterministic) mean dynamic approximation with this program.



*Picture by Ming Jun Tan*

**Exercise 3.** Consider a Hawk-Dove game with payoffs [[0 3][1 2]] where strategy 0 is "Hawk" and strategy 1 is "Dove". Do all players tend to choose the same strategy? Reduce the number of players to 100 and observe the difference in behavior (press the setup button after changing the number of players). Reduce the number of players to 10 and observe the difference.

P.S. You can explore this model's (deterministic) mean dynamic approximation with this program.

**Exercise 4.** Create a stand-alone version of the model we have implemented in this chapter. To do this, you will have to upload the model to NetLogo Web and then export it in HTML format.

 **Exercise 5.** Reimplement the procedure to update-strategy-after-revision so the revising agent uses the imitative pairwise-difference rule that we saw in chapter I-2.

 **Exercise 6.** Reimplement the procedure to update-strategy-after-revision so the revising agent uses the best experienced payoff rule that we saw in chapter I-2.

**Exercise 7.** In our current model, agents compute their payoff by selecting another agent at random and playing the game. Note that this other (randomly selected) agent does not store the payoff of the interaction. By contrast, in some other evolutionary models, it is assumed that agents *are randomly matched in pairs* to play the game (with both members of the pair keeping record of the payoff obtained in the interaction). Can you think about how we could implement this alternative way of computing payoffs? We provide a couple of hints below:

> **Hints to implement random matching**
>
> - Naturally, the main changes will take place in procedure to update-payoff, but some other changes in the code may be necessary.
>
> - In particular, we find it useful to define a new individually-owned variable named played?. For us, this is a boolean variable that keeps track of whether the agent has already played the game in the current tick or not. Thus, this variable would have to be set to false at the beginning of the tick (in procedure to go).
>
> - The built-in reporter `myself` will be useful at the time of asking your mate to set her own payoff.

## II-2. Extension to any number of strategies

## 1. Goal

Our goal here is to extend the model we have created in the previous chapter –which accepted games with 2 strategies only– to allow for (2-player symmetric) games with any number of strategies.

## 2. Motivation. Rock, paper, scissors

The model we will develop in this chapter will allow us to explore games such as Rock-Paper-Scissors. Can you guess what will happen in our model if agents are matched to play Rock-Paper-Scissors and they keep on using the *imitate-if-better* rule whenever they revise?

## 3. Description of the model

In this model, there is a population of *n-of-players* agents who repeatedly play a symmetric 2-player game with any number of strategies. The *payoffs* of the game are determined by the user in the form of a matrix [ [$A_{00}$ $A_{01}$ ... $A_{0n}$] [$A_{10}$ $A_{11}$ ... $A_{1n}$] ... [$A_{n0}$ $A_{n1}$ ... $A_{nn}$] ] containing the payoffs $A_{ij}$ that an agent playing strategy $i$ obtains when meeting an agent playing strategy $j$ ($i, j \in \{0, 1, ..., n\}$). The number of strategies is inferred from the number of rows in the payoff matrix.

Initially, players choose one of the available strategies at random (uniformly). From then onwards, the following sequence of events –which defines a tick– is repeatedly executed:

1. Every agent obtains a payoff by selecting another agent at random and playing the game.

2. With probability *prob-revision*, individual agents are given the opportunity to revise their strategies. The decision rule –called **imitate if better**– reads as follows:

   > Look at another (randomly selected) agent and adopt her strategy if and only if her payoff was greater than yours.

   All agents who revise their strategies within the same tick do it simultaneously (i.e. synchronously).

The model shows the evolution of the number of agents choosing each of the possible strategies at the end of every tick.

## ⏩ 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).
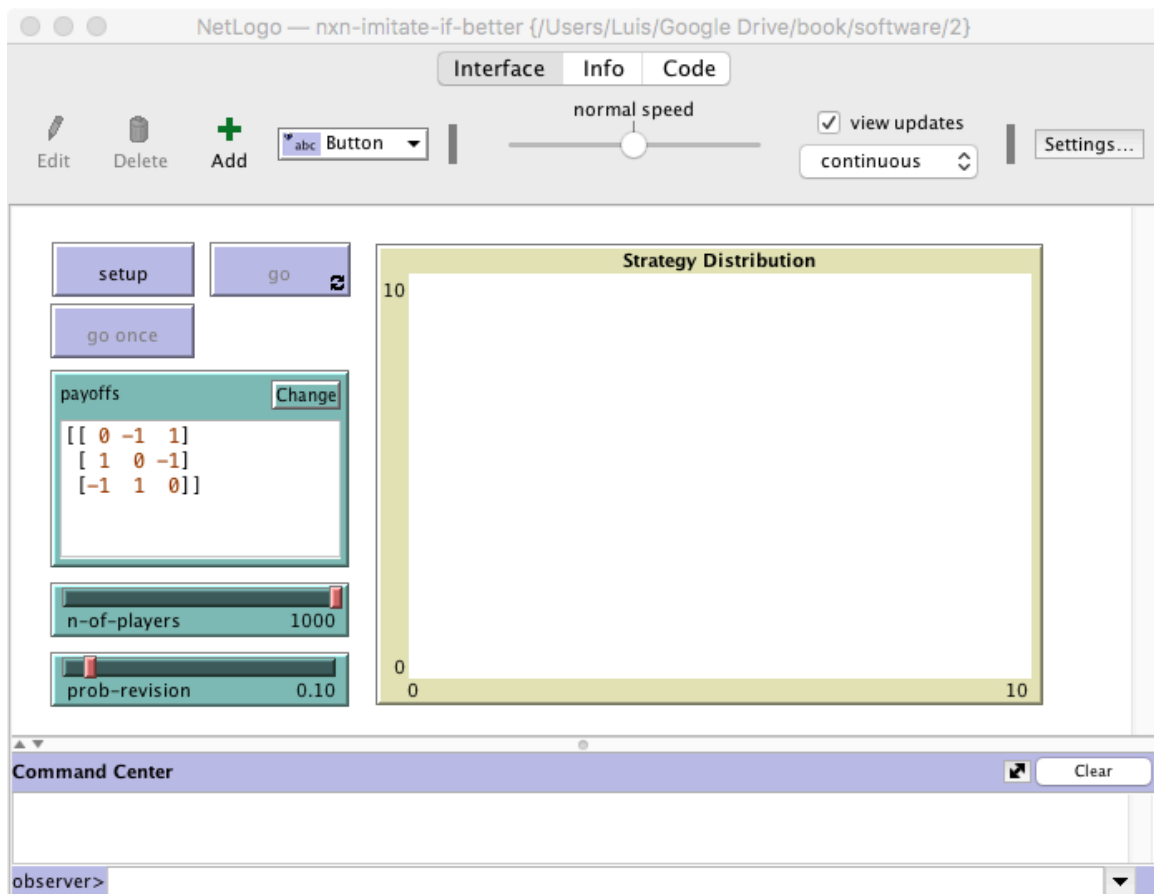
*Figure 1. Interface design*

The new interface (see figure 1 above) requires just two simple modifications:

- Make the *payoffs* input box bigger and let its input contain several lines.

> In the Interface tab, select the input box (by right-clicking on it) and make it bigger.
> Then edit it (by right-clicking on it) and tick the "Multi-Line" box.

- Remove the "pens" in the Strategy Distribution plot. Since the number of strategies is unknown until the payoff matrix is read, we will need to create the required number of "pens" via code.

> In the Interface tab, edit the Strategy Distribution plot and delete both pens.

# ▶ 5. Code

## 5.1. Skeleton of the code
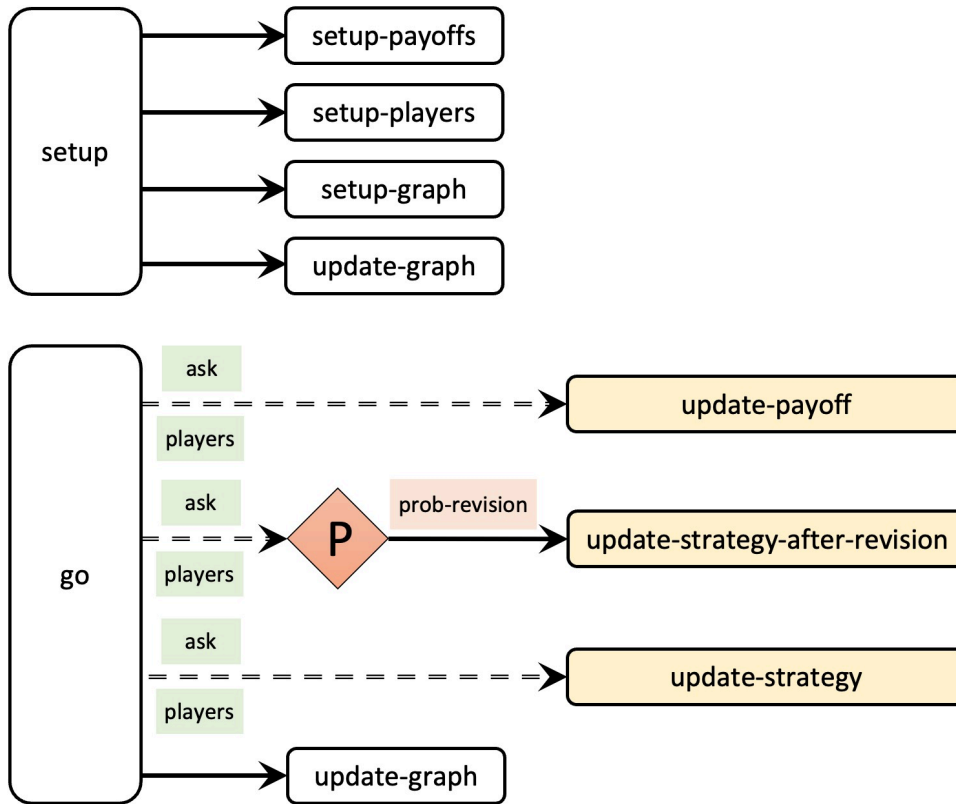


*Figure 2. Skeleton of the code*

## 5.2. Global variables and individually-owned variables

It will be handy to have a variable store the number of strategies. Since this information will likely be used in various procedures, it makes sense to define the new variable as global. A natural name for this new variable is n-of-strategies. The modified code will look as follows:

```
globals [
  payoff-matrix
  n-of-strategies
]
```

## 5.3. Setup procedures

The current setup procedure is the following:

```
to setup
  clear-all
  set payoff-matrix read-from-string payoffs
  create-players n-of-players [
```

```
     set payoff 0
     set strategy 0
   ]
   ask n-of random (n-of-players + 1) players [set strategy 1]
   ask players [set strategy-after-revision strategy]
   reset-ticks
 end
```

Note that the code in the current setup procedure performs several unrelated tasks –namely clear everything, set up the payoffs, set up the players, and set up the tick counter–, and now we will need to set up the graph as well (since we have to create as many pens as strategies). Let us take this opportunity to modularize our code and improve its readability by creating new procedures with descriptive names for groups of related instructions, as follows:

```
to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
end
```

## to setup-payoffs

The procedure to setup-payoffs will include the instructions to read the payoff matrix, and will also set the value of the global variable n-of-strategies. We will use the primitive `length` to obtain the number of rows in the payoff matrix.

```
to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end
```

## to setup-players

The procedure to setup-players will create the players and set the initial values for their individually-owned variables. The initial payoff will be 0 and the initial strategy will be a random integer between 0 and (n-of-strategies – 1). We must not forget to initialize their strategy-after-revision too.

```
to setup-players
 create-players n-of-players [
    set payoff 0
    set strategy (random n-of-strategies)
    set strategy-after-revision strategy
 ]
end
```

## to setup-graph

The procedure to setup-graph will create the required number of pens –one for each strategy– in the Strategy Distribution plot. To this end, we must first specify that we wish to work on the Strategy Distribution plot, using the primitive `set-current-plot`.

```
set-current-plot "Strategy Distribution"
```

Then, for each strategy $i \in \{0, 1, ..., ($n-of-strategies$ - 1)\}$, we do the following tasks:

1. Create a pen with the name of the strategy. For this, we use the primitive `create-temporary-plot-pen` to create the pen, and the primitive `word` to turn the strategy number into a string.

   ```
   create-temporary-plot-pen (word i)
   ```

2. Set the pen mode to 1 (bar mode) using `set-plot-pen-mode`. We do this because we plan to create a stacked bar chart for the distribution of strategies.

   ```
   set-plot-pen-mode 1
   ```

3. Choose a color for each pen. See how colors work in NetLogo.

   ```
   set-plot-pen-color 25 + 40 * i
   ```

Now we have to actually loop through the number of each strategy, making $i$ take the values 0, 1, ..., (n-of-strategies $- 1$). There are several ways we can do this. Here, we do it by creating a list [0 1 2 ... (n-of-strategies $- 1$)] containing the strategy numbers and going through each of its elements. To create the list, we use the primitive `range`.

```
range n-of-strategies
```

The final code for the procedure to setup-graph is then:

```
to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end
```

## to update-graph

Procedure to update-graph will draw the strategy distribution using a stacked bar chart, like the one shown in figure 3 below. This procedure is called at the end of setup to plot the initial distribution of strategies, and then also at the end of procedure to go, to plot the strategy distribution at the end of every tick.
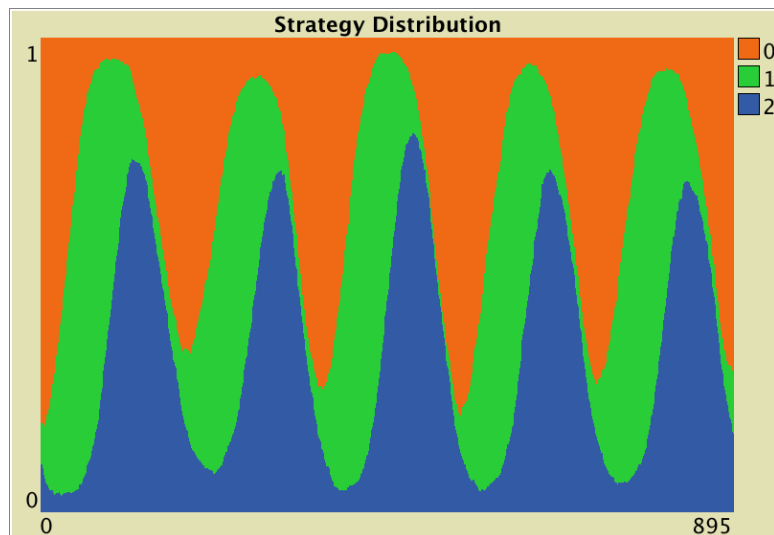
*Figure 3. Example of stacked bar chart showing the strategy distribution as ticks go by*

We start by creating a list containing the strategy numbers [0 1 2 … (n-of-strategies – 1)], which we store in local variable strategy-numbers.

```
let strategy-numbers (range n-of-strategies)
```

To compute the (relative) strategy frequencies, we apply to each element of the list strategy-numbers, i.e. to each strategy number, the operation that calculates the fraction of players using that strategy. To do this, we use primitive `map`. Remember that `map` requires as inputs a) the function to be applied to each element of the list and b) the list containing the elements on which you wish to apply the function. In this case, the function we wish to apply to each strategy number (implemented as an anonymous procedure) is:

```
[n -> ( count (players with [strategy = n]) ) / n-of-players]
```

In the code above, we first identify the subset of players that have a certain strategy (using `with`), then we count the number of players in that subset (using `count`), and finally we divide by the total number of players n-of-players. Thus, we can use the following code to obtain the strategy frequencies, as a list:

```
map [n -> count players with [strategy = n] / n-of-players] strategy-numbers
```

Finally, to build the stacked bar chart, we begin by plotting a bar of height 1, corresponding to the first strategy. Then we repeatedly draw bars on top of the previously drawn bars (one bar for each of the remaining strategies), with the height diminished each time by the relative frequency of the corresponding strategy. The final code of procedure to update-graph will look as follows:

```
to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
      count players with [strategy = n] / n-of-players
    ] strategy-numbers
```

```
    set-current-plot "Strategy Distribution"
    let bar 1
    foreach strategy-numbers [ n ->
      set-current-plot-pen (word n)
      plotxy ticks bar
      set bar (bar - (item n strategy-frequencies))
    ]
    set-plot-y-range 0 1
  end
```

## 5.4. Go procedure

The only change needed in the go procedure is the call to procedure to update-graph, which will draw the fraction of agents using each strategy at the end of every tick:

```
to go
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-graph
end
```

## 5.5. Other procedures

Note that there is no need to modify the code of to update-payoff, to update-strategy-after-revision, or to update-strategy.

## 5.6. Complete code in the Code tab

The Code tab is ready!

```
globals [
  payoff-matrix
  n-of-strategies
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]
```

```
to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end

to setup-players
 create-players n-of-players [
    set payoff 0
    set strategy (random n-of-strategies)
    set strategy-after-revision strategy
 ]
end

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

to go
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-graph
end

to update-payoff
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end

to update-strategy-after-revision
  let observed-player one-of other players
  if ([payoff] of observed-player) > payoff [
    set strategy-after-revision ([strategy] of observed-player)
```

```
     ]
  end

  to update-strategy
    set strategy strategy-after-revision
  end

  to update-graph
    let strategy-numbers (range n-of-strategies)
    let strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers

    set-current-plot "Strategy Distribution"
    let bar 1
    foreach strategy-numbers [ n ->
      set-current-plot-pen (word n)
      plotxy ticks bar
      set bar (bar - (item n strategy-frequencies))
    ]
    set-plot-y-range 0 1
  end
```

## 5.7. Code inside the plots

Note that we take care of all plotting in the update-graph procedure. Thus there is no need to write any code inside the plot. We could instead have written the code of procedure to update-graph inside the plot, but given that it is somewhat lengthy, we find it more convenient to group it with the rest of the code in the Code tab.

# 6. Sample run

Now that we have implemented the model, we can explore the behavior of a population who are repeatedly matched to play a Rock-Paper-Scissors game. To do that, let us use payoff matrix [[0 -1 1][1 0 -1][-1 1 0]], a population of 500 agents and a 0.1 probability of revision. The following video shows a representative run with these settings.

*One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=98#video-98-1*

Note that soon in the simulation run, one of the strategies will get a greater share by chance (due to the inherent randomness of the model). Then, the next strategy (modulo 3) will enjoy a payoff advantage, and thus will tend to be imitated. For example, if "Paper" is the most popular strategy, then agents playing "Scissors" will tend to get higher payoffs, and thus be imitated. As the fraction

of agents playing "Scissors" grows, strategy "Rock" becomes more attractive... and so on and so forth. These cycles get amplified until one of the strategies disappears. At that point, one of the two remaining strategies is superior and finally prevails. The three strategies have an equal change of being the "winner" in the end, since the whole model setting is symmetric.

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-imitate-if-better.nlogo.

**Exercise 1.** Consider a Rock-Paper-Scissors game with payoff matrix [[0 -1 1][1 0 -1][-1 1 0]]. Here we ask you to explore how the dynamics are affected by the number of players *n-of-players* and by the probability of revision *prob-revision*. Explore simulations with a small population (e.g. *n-of-players* = 50) and with a large population (e.g. *n-of-players* = 1000). Also, for each case, try both a small probability of revision (e.g. *prob-revision* = 0.01) and a large probability of revision (e.g. *prob-revision* = 0.5).



*Picture by Liane Metzler*

How do your insights change if you use payoff matrix [[0 -1 10][10 0 -1][-1 10 0]]?

**Exercise 2.** Consider a game with payoff matrix [[1 1 0][1 1 1][0 1 1]]. Set the probability of revision to 0.1. Press the setup button and run the model for a while (then press the setup button again to change the initial conditions). Can you explain what happens?

**Exercise 3.** How would you create the list [0 1 2 ... (n-of-strategies – 1)] using `n-values` instead of `range`?

**Exercise 4.** Implement the procedure to setup-graph:

1. using the primitive `repeat` instead of `foreach`.
2. using the primitive `while` instead of `foreach`.
3. using the primitive `loop` instead of `foreach`.

**Exercise 5.** Reimplement the procedure to update-strategy-after-revision so the revising agent looks at five (randomly selected) other agents and copies the strategy of the agent with the highest payoff (among these five observed agents). Resolve ties as you wish.

**Exercise 6.** Reimplement the procedure to update-strategy-after-revision so the revising agent

selects the strategy that is the best response to (i.e. obtains the greatest payoff against) the strategy of another (randomly) observed agent. This is an instance of the so-called *sampling best response decision rule* (Sandholm (2001), Kosfeld et al. (2002), Oyama et al. (2015)). Resolve ties as you wish.

# II-3. Noise and initial conditions

## 1. Goal

Our goal is to extend the model we have created in the previous chapter by adding two features that will prove very useful:

- The possibility of setting initial conditions explicitly. This is an important feature because initial conditions can be very relevant for the evolution of a system.

- The possibility that revising agents select a strategy at random with a small probability. This type of noise in the revision process may account for experimentation or errors in economic settings, or for mutations in biological contexts. The inclusion of noise in a model can sometimes change its dynamic behavior dramatically, even creating new attractors. This is important because dynamic features of a model –such as attractors, cycles, repellors, and other patterns– that are not robust to the inclusion of small noise may not correspond to relevant properties of the real-world system that we aim to understand. Besides, as a positive side-effect, adding small amounts of noise to a model often makes the analysis of its dynamics easier to undertake.

## 2. Motivation. Noise in rock, paper, scissors

In the previous chapter we saw that simulations of the Rock-Paper-Scissors game under the *imitate-if-better* decision rule end up in a state where everyone is choosing the same strategy. Can you guess what will happen in this model if we add a little bit of noise?

## 3. Description of the model

In this model, there is a population of *n-of-players* agents who repeatedly play a symmetric 2-player game with any number of strategies. The *payoffs* of the game are determined by the user in the form of a matrix [ [$A_{00}$ $A_{01}$ ... $A_{0n}$] [$A_{10}$ $A_{11}$ ... $A_{1n}$] ... [$A_{n0}$ $A_{n1}$ ... $A_{nn}$] ] containing the payoffs $A_{ij}$ that an agent playing strategy $i$ obtains when meeting an agent playing strategy $j$ ($i, j \in \{0, 1, ..., n\}$). The number of strategies is inferred from the number of rows in the payoff matrix.

Initial conditions are set with parameter *n-of-players-for-each-strategy*, using a list of the form [$a_0$ $a_1$ ... $a_n$], where item $a_i$ is the initial number of agents with strategy $i$. Thus, the total number of agents is the sum of all elements in this list. From then onwards, the following sequence of events –which defines a tick– is repeatedly executed:

1. Every agent obtains a payoff by selecting another agent at random and playing the game.

2. With probability *prob-revision*, individual agents are given the opportunity to revise their strategies. In that case, with probability *noise*, the revising agent will adopt a random

strategy; and with probability (1 – *noise*), the revising agent will choose her strategy following the ***imitate if better*** rule:

> Look at another (randomly selected) agent and adopt her strategy if and only if her payoff was greater than yours.

All agents who revise their strategies within the same tick do it simultaneously (i.e. synchronously).

The model shows the evolution of the number of agents choosing each of the possible strategies at the end of every tick.

# 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).
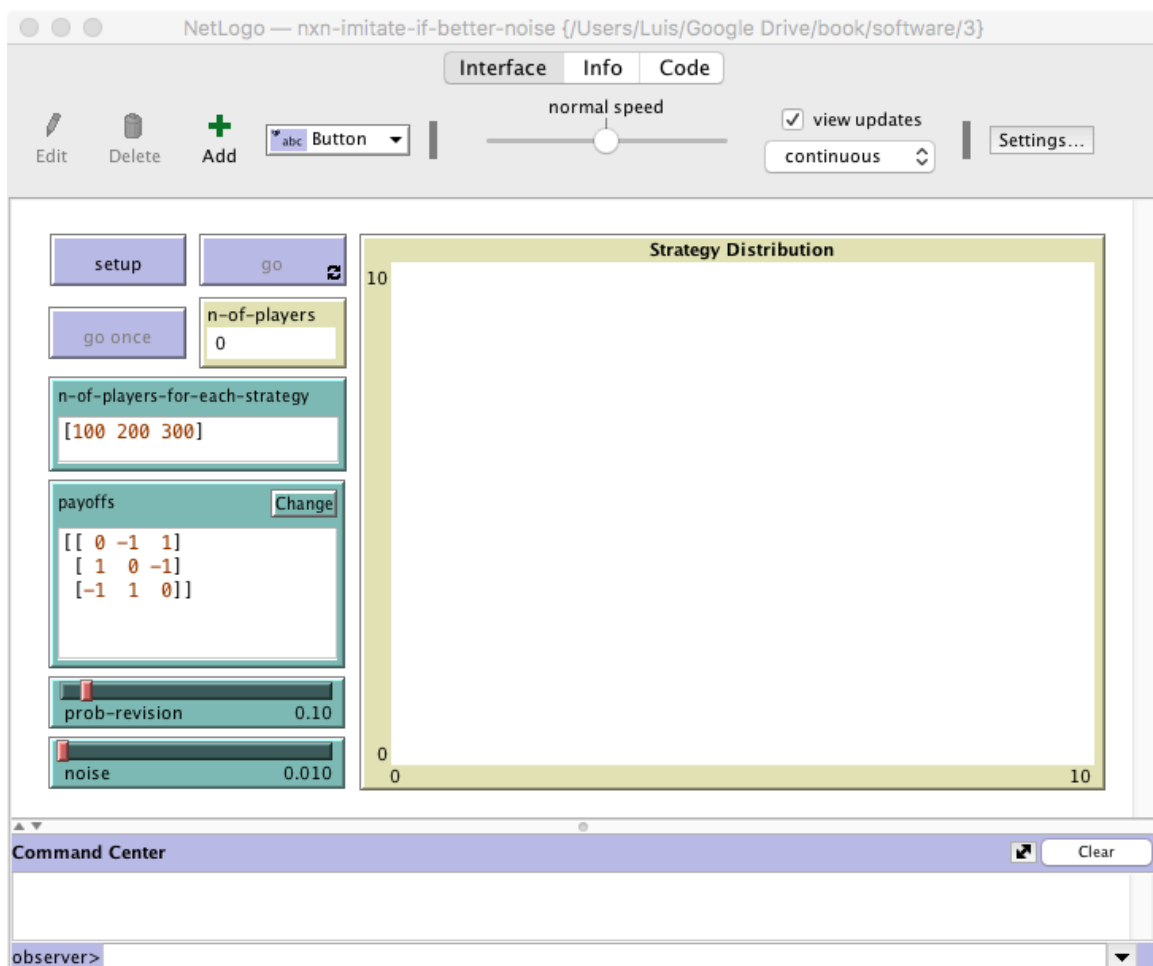


*Figure 1. Interface design*

The new interface (see figure 1 above) requires a few simple modifications:

- Create an input box to let the user set the initial number of players using each strategy.

> In the Interface tab, add an input box with associated global variable *n-of-players-for-each-strategy*. Set the input box type to "String (reporter)".

- Note that the total number of players (which was previously set using a slider with associated global variable *n-of-players*) will now be computed totaling the items of the list *n-of-players-for-each-strategy*. Thus, we should remove the slider, and include the global variable n-of-players in the Code tab.

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
]
```

- Add a monitor to show the total number of players. This number will be stored in the global variable n-of-players, so the monitor must show the value of this variable.

> In the Interface tab, create a monitor. In the "Reporter" box write the name of the global variable n-of-players.

- Create a slider to choose the value of parameter *noise*.

> In the Interface tab, create a slider with associated global variable *noise*. Choose limit values 0 and 1, and an increment of 0.001.

# 🚩 5. Code

## 5.1. Skeleton of the code
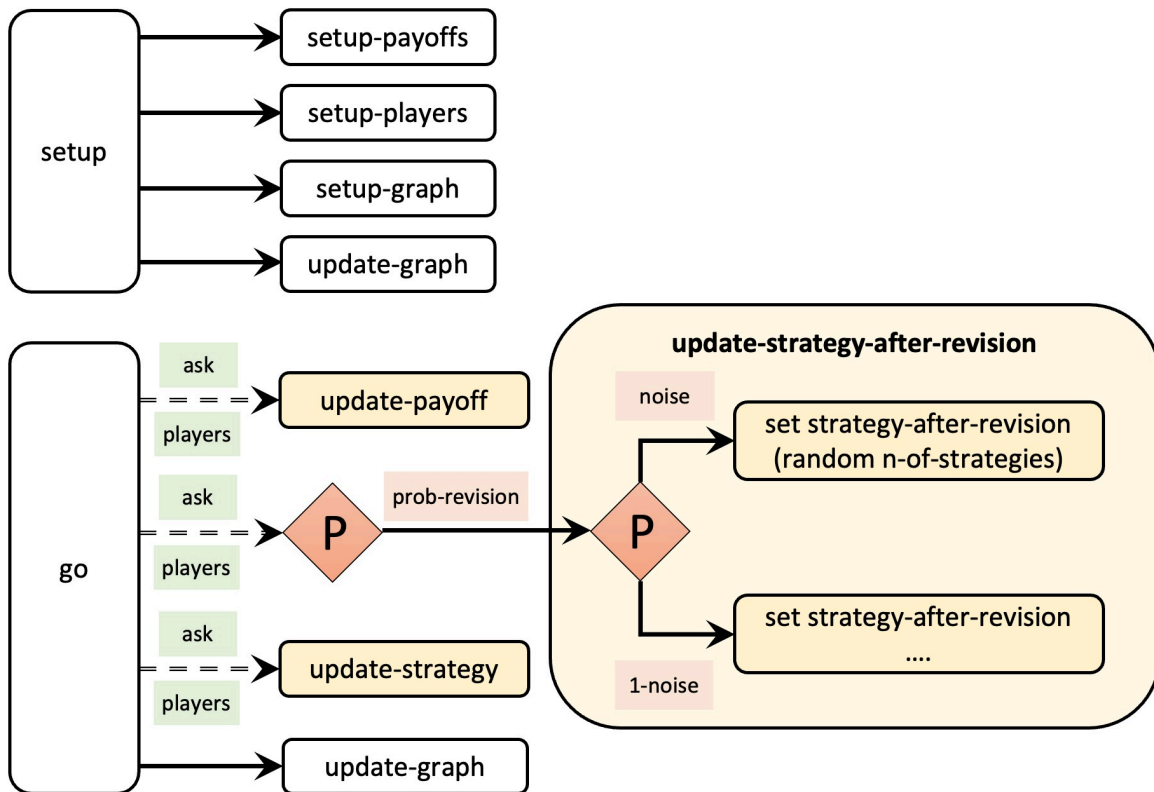


*Figure 2. Skeleton of the code*

## 5.2. Global variables and individually-owned variables

The only change required regarding user-defined variables is the inclusion of global variable n-of-players in the Code tab, as explained in the previous section.

## 5.3. Setup procedures

To read the initial conditions specified with parameter *n-of-players-for-each-strategy* and set up the players accordingly, it is clear that we only have to modify the code in procedure to setup-players. Note that making our code modular, by implementing short procedures with specific tasks and meaningful names, makes our life easy at the time of extending the model.

### to setup-players

Since the content of parameter *n-of-players-for-each-strategy* is a string, the first we should do is to turn it into a list that we can use in our code. To this end, we use the primitive `read-from-string` and store its output in a new local variable named initial-distribution, as follows:

```
let initial-distribution
  read-from-string n-of-players-for-each-strategy
```

Next, we can check that the number of elements in the list initial-distribution matches the number of possible strategies (i.e. the number of rows in the payoff matrix stored in payoff-matrix), and issue a warning message otherwise, using primitive `user-message`. Naturally, this is by no means compulsory, but it is a thoughtful touch that will make our program more user-friendly. To this end, we can use the code below.

```
if length initial-distribution != length payoff-matrix [
  user-message (word "The number of items in\n"
      ;; "\n" is used to jump to the next line
    "n-of-players-for-each-strategy (i.e. "
    length initial-distribution "):\n"
    n-of-players-for-each-strategy
    "\nshould be equal to the number of rows\n"
    "in the payoff matrix (i.e. "
    length payoff-matrix "):\n"
    payoffs
  )
]


;; It is not necessary to show the user
;; the value of n-of-players-for-each-strategy
;; and payoffs again,
;; but when creating an error message,
;; it is good practice to give the user
;; as much information as possible,
;; so the error can be easily corrected.
```

Now, let us create as many players using each strategy as indicated by the values in the list initial-distribution. For instance, if initial-distribution is [5 10 15], we should create 5 players with strategy 0, 10 players with strategy 1, and 15 players with strategy 2. Since we want to perform a task for each element of the list, primitive `foreach` will be handy.

Besides going through each element on the list using `foreach`, we would also like to keep track of the position being read on the list, which is the corresponding strategy number. For this, we create a counter i which we start at 0:

```
let i 0
foreach initial-distribution [ j ->
  create-players j [
    set payoff 0
    set strategy i
    set strategy-after-revision strategy
  ]
  set i (i + 1)
]
```

Finally, let us set the value of the global variable n-of-players:

```
    set n-of-players count players
```

The line above concludes the definition of procedure to setup-players, and the implementation of the user-chosen initial conditions.

## 5.4. Go and other main procedures

To implement the choice of a random strategy with probability *noise* by revising agents, we have to modify the code of procedure to update-strategy-after-revision. At present, the code of this procedure looks as follows:

```
to update-strategy-after-revision
  let observed-player one-of other players
  if ([payoff] of observed-player) > payoff [
    set strategy-after-revision ([strategy] of observed-player)
  ]
end
```

We can implement the noise feature using primitive `ifelse`, whose structure is

```
ifelse CONDITION
   [ COMMANDS EXECUTED IF CONDITION IS TRUE ]
   [ COMMANDS EXECUTED IF CONDITION IS FALSE ]
```

In our case, the CONDITION should be true with probability *noise*. Bearing all this in mind, the final code for procedure to update-strategy-after-revision could be as follows:

```
to update-strategy-after-revision
  ifelse random-float 1 < noise
  ;; the condition is true with probability noise
    [ ;; code to be executed if there is noise
      set strategy-after-revision (random n-of-strategies)
    ]
    [ ;; code to be executed if there is no noise
      let observed-player one-of other players
      if ([payoff] of observed-player) > payoff [
        set strategy-after-revision ([strategy] of observed-player)
      ]
    ]
end
```

## 5.5. Complete code in the Code tab

The Code tab is ready!

```
globals [
  payoff-matrix
```

```
    n-of-strategies
    n-of-players
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]

to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end

to setup-players
  let initial-distribution
    read-from-string n-of-players-for-each-strategy
  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]

  let i 0
  foreach initial-distribution [ j ->
    create-players j [
      set payoff 0
      set strategy i
      set strategy-after-revision strategy
    ]
    set i (i + 1)
  ]

  set n-of-players count players
end
```

```
to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

to go
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-graph
end

to update-payoff
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      let observed-player one-of other players
      if ([payoff] of observed-player) > payoff [
        set strategy-after-revision ([strategy] of observed-player)
      ]
    ]
end

to update-strategy
  set strategy strategy-after-revision
end

to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
      count players with [strategy = n] / n-of-players
    ] strategy-numbers

  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
```

```
     set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end
```

# 6. Sample run

Now that we have implemented the model, we can use it to answer the question posed above: Will adding a bit of noise change the dynamics of the Rock-Paper-Scissors game under the *imitate-if-better* decision rule? To do that, let us use the same setting as in the previous chapter, i.e. *payoffs* = [[0 -1 1][1 0 -1][-1 1 0]] and *prob-revision* = 0.1. To have 500 agents and initial conditions close to random, we can set *n-of-players-for-each-strategy* = [167 167 166]. Finally, let us use *noise* = 0.01. The following video shows a representative run with these settings.

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=103#video-103-1*

As you can see, noise dampens the amplitude of the cycles, so the monomorphic states where only one strategy is chosen by the whole population are not observed anymore.[1] Even if at some point one strategy went extinct, noise would bring it back into existence. Thus, the model with *noise* = 0.01 exhibits an everlasting pattern of cycles of varying amplitudes. This contrasts with the model without noise, which necessarily ends up in one of only three possible final states.

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-imitate-if-better-noise.nlogo.

---

1. In this model with noise, every state will be observed at some point if we wait for long enough, but long enough might be a really long time (e.g. centuries).

**Exercise 1.** Consider a Prisoner's Dilemma with payoffs [[2 4][1 3]] where strategy 0 is "Defect" and strategy 1 is "Cooperate". Set *prob-revision* to 0.1 and *noise* to 0. Set the initial number of players using each strategy, i.e. *n-of-players-for-each-strategy*, to [0 200], i.e., everybody plays "Cooperate". Press the setup button and run the model. While it is running, move the *noise* slider slightly rightward to introduce some small noise. Can you explain what happens?



*Picture by Danielle MacInnes*

**Exercise 2.** Consider a Rock-Paper-Scissors game with payoff matrix [[0 -1 1][1 0 -1][-1 1 0]]. Set *prob-revision* to 0.1 and *noise* to 0. Set the initial number of players using each strategy, i.e. *n-of-players-for-each-strategy*, to [100 100 100]. Press the setup button and run the model for a while. While it is running, click on the *noise* slider to set its value to 0.001. Can you explain what happens?

**Exercise 3.** Consider a game with payoff matrix [[1 1 0][1 1 1][0 1 1]]. Set *prob-revision* to 0.1, *noise* to 0.05, and the initial number of players using each strategy, i.e. *n-of-players-for-each-strategy*, to [500 0 500]. Press the setup button and run the model for a while (then press the setup button again to change the initial conditions). Can you explain what happens?

**Exercise 4.** Consider a game with *n* players and *s* strategies, with *noise* equal to 1. What is the infinite-horizon probability distribution of the number of players using each strategy?

**Exercise 5.** Imagine that you'd like to run this model faster, and you are not interested in the plot. This is a common scenario when you want to conduct large-scale computational experiments. What lines of code could you comment out?

**Exercise 6.** Note that you can modify the values of parameters *prob-revision* and *noise* at runtime with immediate effect on the dynamics of the model. How could you implement the possibility of changing the number of players in the population with immediate effect on the model?

# II-4. Interactivity and efficiency

## 1. Goal

Our goal in this chapter is to improve the interactivity and the efficiency of our model.

By **interactivity** we mean the possibility of changing the value of parameters at runtime, with immediate effect on the dynamics of the model. This feature is very convenient for exploratory work. In this chapter, we will implement the necessary functionality to let the user change the number of agents in the population at runtime.

By **efficiency** we mean implementing the model in such a way that it can be executed using as little time and memory as possible. In this chapter, we will modify the code of our model slightly to make it run significantly faster.

Oftentimes there is a trade-off between interactivity and efficiency: making the model more interactive generally implies some loss of efficiency. Nonetheless, sometimes we can find ways of implementing a model more efficiently without compromising its interactivity.

It is also important to be aware that –most often– there is also a trade-off between efficiency and code readability. The changes required to make our model run faster will frequently make our code somewhat less readable too. Uri Wilensky –the creator of NetLogo– and William Rand do not recommend making such comprises:

> However, it is important that your code be readable, so others can understand it. In the end, computer time is cheap compared to human time. Therefore, it should be noted that, whenever there is a possibility of trade-off, clarity of code should be preferred over efficiency. Wilensky and Rand (2015, pp 219–20)

Our personal opinion is that this decision is best made case by case, taking into account the objectives and constraints of the whole modelling exercise in the specific context at hand. Our hope is that, after reading this book, you will be prepared to make these decisions by yourself in any specific situation you may encounter.

## 2. Motivation. The impact of population size

The dynamics of many evolutionary models strongly depend on the number of agents in the population. Can you guess how the population size affects the dynamics of the *imitate-if-better* decision rule with noise in the Rock-Paper-Scissors game? In this chapter we will implement the possibility of changing the population size at runtime, a feature that will greatly facilitate the exploration of this question.

# 3. Description of the model

We will not make any modification on the formal model our program implements. Thus, we refer to the previous chapter to read the description of the model. The only paragraph we add (about the program itself) is the following:

> The number of players in the simulation can be changed at runtime with immediate effect on the dynamics of the model, using parameter *n-of-players*:
>
> - If *n-of-players* is reduced, the necessary number of (randomly selected) players are killed.
>
> - If *n-of-players* is increased, the necessary number of (randomly selected) players are cloned.
>
> Thus, the proportions of agents playing each strategy remain the same on average (although the actual effect of this change is stochastic).

# 4. Interactivity

Note that we can already modify the value of parameters *prob-revision* and *noise* at runtime, with immediate effect on the dynamics of the model. This is so because the values of these variables are used directly in the code. Parameter *prob-revision* is used only in procedure to go, in the following line:

```
if (random-float 1 < prob-revision) [update-strategy-after-revision]
```

And parameter *noise* is used only in procedure to update-strategy-after-revision, in the following line:

```
ifelse (random-float 1 < noise)
```

Whenever NetLogo reads the two lines of code above, it uses the current values of the two parameters. Because of this, we can modify the parameters' values on the fly and immediately see how that change affects the dynamics of the model.

By contrast, changing the value of parameter *n-of-players-for-each-strategy* at runtime will have no effect whatsoever. This is so because parameter *n-of-players-for-each-strategy* is only used in procedure to setup-players, which is executed at the beginning of the simulation –triggered by procedure to setup– and never again.

To enable the user to modify the population size at runtime, we should create a slider for the new parameter *n-of-players*. Before doing so, we have to remove the declaration of the global variable n-of-players in the Code tab, since the creation of the slider implies the definition of the variable as global.

```
globals [
  payoff-matrix
  n-of-strategies
  ;; n-of-players        <== we remove this line
]
```

After creating the slider for parameter *n-of-players*, we could also remove the monitor showing n-of-players from the interface, since it is no longer needed. Another option (see figure 1 below) is to use that same monitor to display the value of the ticks that have gone by since the beginning of the simulation. To do this, we just have to write the primitive `ticks` (instead of n-of-players) in the "Reporter" box of the monitor.



*Figure 1. Interface design*

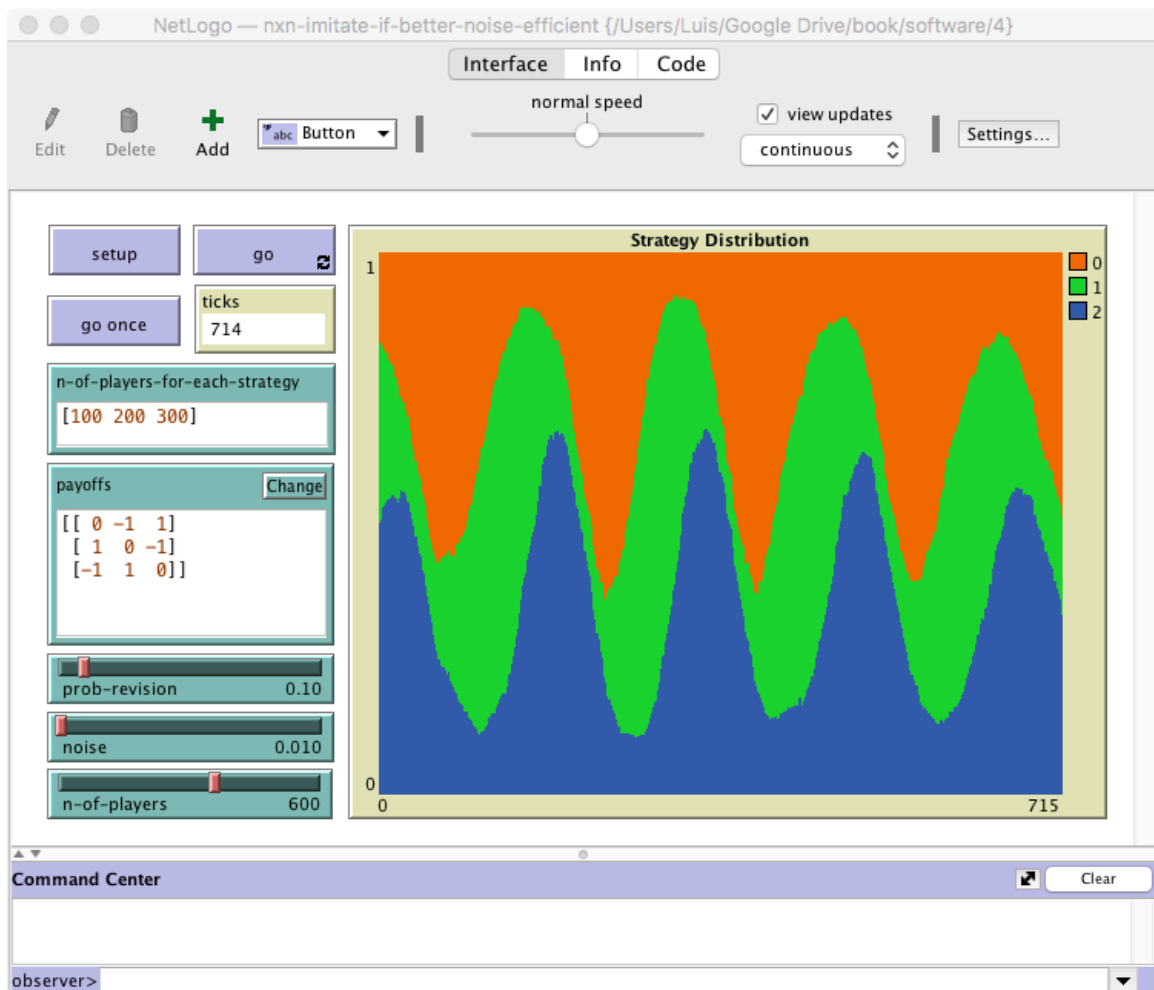The next step is to implement a separate procedure to check whether the value of parameter *n-of-players* differs from the current number of players in the simulation and, if it does, act accordingly. We find it natural to name this new procedure to update-n-of-players, and one possible implementation would be the following:

```
to update-n-of-players
  let diff (n-of-players - count players)
```

```
    if diff != 0 [
      ifelse diff > 0
      [ repeat diff [ ask one-of players [hatch-players 1] ] ]
      [ ask n-of (- diff) players [die] ]
    ]
  end
```

Note the use of primitives `hatch-players` and `die` to clone and kill agents respectively. The difference between primitives `hatch-players` and `create-players` is important. Hatching is an action that only individual agents (i.e. "turtles" and breeds of "turtles", in NetLogo parlance) can execute. By contrast, only the observer can run `create-turtles` and `create-<breeds>` primitives.

Finally, we should include the call to the new procedure at the beginning of to go.

```
to go
  update-n-of-players        ;; <== new line
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]

  tick
  update-graph
end
```

And with this, we're ready to go! Give it a try, and enjoy the good progress you are making!

Figure 2 below shows the skeleton of our current procedure to go, highlighting the only substantial change we have made so far.
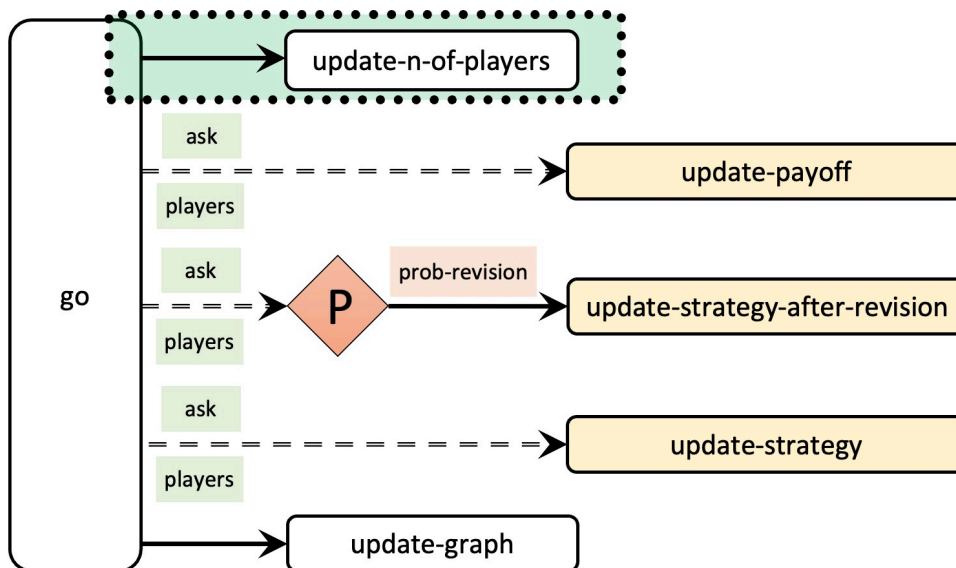
*Figure 2. Skeleton of procedure to go in model nxn-imitate-if-better-noise-interactive-profiler.nlogo. The dashed green rectangle highlights the main addition*

# 5. Efficiency

Naturally, to make a model run faster, one can always untick the "view updates" box on the Interface tab. The same effect can be achieved by pushing the speed slider –situated in the middle of the interface toolbar– to its rightmost position, or by using primitive `no-display` in the code. This is a must in models that do not make use of the view, like the ones we are programming here, since it implies a significant speed-up at no cost. Thus, in our model, we should include the primitive `no-display` in procedure to setup, right after `clear-all`.

```
to setup
  clear-all
  no-display      ;; <== new line
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
end
```

But beyond this simple piece of advice, in general, how can we make our model run faster? In our view, the very first thing we have to do is to find out where our model spends most of the time. The following section shows how to do this.

## 5.1. Measuring execution speed of different parts of the code

There are two simple ways to measure execution speed in NetLogo. One is using primitives `reset-timer` and `timer`. For instance, to time how long it takes to have every agent carry out the operation:

```
  other players
```

we could write the following reporter:

```
to-report time-other-players
  reset-timer
  ask players [let temporary-var other players]
  report timer
end
```

The second –more advanced– way of measuring execution speed involves the Profiler Extension, which comes bundled with NetLogo. This extension allows us to see how many times each procedure in our model is called during a run and how long each call takes. The extension is simple to use and it is well documented at its own website. To use it in our model, we should include the extension at the beginning of our code, as follows:

```
extensions [profiler]
```

Then we could execute the following procedure, borrowed from the Profiler Extension documentation page. (Please, remember that we have to include the new procedure *after* the declaration of variables.)

```
to show-profiler-report
  setup                  ;; set up the model
  profiler:start         ;; start profiling
  repeat 1000 [ go ]     ;; run something you want to measure
  profiler:stop          ;; stop profiling
  print profiler:report  ;; print the results
  profiler:reset         ;; clear the data
end
```

Once the procedure is implemented, you can run it by typing its name in the Command Center.

The profiler report includes the inclusive time and the exclusive time for each procedure. **Inclusive time** is the time the simulation spends running the procedure, i.e. since the procedure is entered until it finishes. **Exclusive time** is the time passed since the procedure is entered until it finishes, but does not include any time spent in other user-defined procedures which it calls.

Let us see an example of the output printed by `show-profiler-report` with our current model (nxn-imitate-if-better-noise-interactive-profiler.nlogo). We model Rock-Paper-Scissors game (*payoffs* = [[0 -1 1][1 0 -1][-1 1 0]]), initial distribution *n-of-players-for-each-strategy* = [200 200 200], *prob-revision* = 0.1 and *noise* = 0.01.

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name                 Calls Incl T(ms) Excl T(ms) Excl/calls
UPDATE-PAYOFF        600000  13569.169  13569.169      0.023
UPDATE-STRATEGY      600000   7745.460   7745.460      0.013
UPDATE-STRATEGY-AFT...  60179    796.219    796.219      0.013
GO                     1000  22746.879    462.636      0.463
UPDATE-GRAPH           1000    170.606    170.606      0.171
UPDATE-N-OF-PLAYERS    1000      2.788      2.788      0.003


Sorted by Inclusive Time
GO                     1000  22746.879    462.636      0.463
UPDATE-PAYOFF        600000  13569.169  13569.169      0.023
UPDATE-STRATEGY      600000   7745.460   7745.460      0.013
UPDATE-STRATEGY-AFT...  60179    796.219    796.219      0.013
UPDATE-GRAPH           1000    170.606    170.606      0.171
UPDATE-N-OF-PLAYERS    1000      2.788      2.788      0.003


Sorted by Number of Calls
UPDATE-PAYOFF        600000  13569.169  13569.169      0.023
UPDATE-STRATEGY      600000   7745.460   7745.460      0.013
UPDATE-STRATEGY-AFT...  60179    796.219    796.219      0.013
GO                     1000  22746.879    462.636      0.463
UPDATE-GRAPH           1000    170.606    170.606      0.171
UPDATE-N-OF-PLAYERS    1000      2.788      2.788      0.003
END PROFILING DUMP
```

In the example above we can see –among other things– that:

- Simulations spend most of the time executing procedure to update-payoff (13 569 / 22 746 ≈ 60%). This procedure is fairly simple (it only takes 0.023 ms each time it is executed), but the problem is that it is called 600 000 times. This makes sense, since there are 600 agents in this simulation, each of them runs to update-payoff once every tick, and we ran the model for 1000 ticks (600 × 1 × 1000 = 600 000).

- After procedure to update-payoff, simulations spend most of the time running procedure to update-strategy. This procedure is extremely simple, but it is also called 600 000 times. All in all, this procedure takes 7 745 / 22 746 ≈ 34% of simulation time.

- Our implementation to allow the user to modify the number of agents at runtime hardly takes any computing time (just 2.788 ms, i.e., 2.788 / 22 746 ≈ 0.01% of simulation time).

With this information in mind, our next step is to try to identify inefficiencies in our code. These inefficiencies often take one of two possible forms:

- Computations that we conduct but we do not use at all.

- Computations that we conduct several times despite knowing that their outputs will not change.

Let us see an example of each of these inefficiencies in our current code.

## 5.2. Example of computations that we conduct but do not use

Note that in this model we make all agents update their payoff in every tick, but we only use the payoffs obtained by the revising agents and by the agents they observe. Thus, we can make the model run faster by asking only revising and observed agents to update their payoff. One way of implementing this efficiency improvement would be to modify the code of procedures to go and to update-strategy-after-revision as follows:

```
to go
  update-n-of-players
  ;; ask players [update-payoff]      <== We remove this line
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-graph
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      let observed-player one-of other players

      update-payoff                     ;; <== new line
      ask observed-player [update-payoff] ;; <== new line

      if ([payoff] of observed-player) > payoff [
        set strategy-after-revision ([strategy] of observed-player)
      ]
    ]
end
```

These changes, which are summarized in Figure 3 below, will make simulations with low *prob-revision* run much faster.
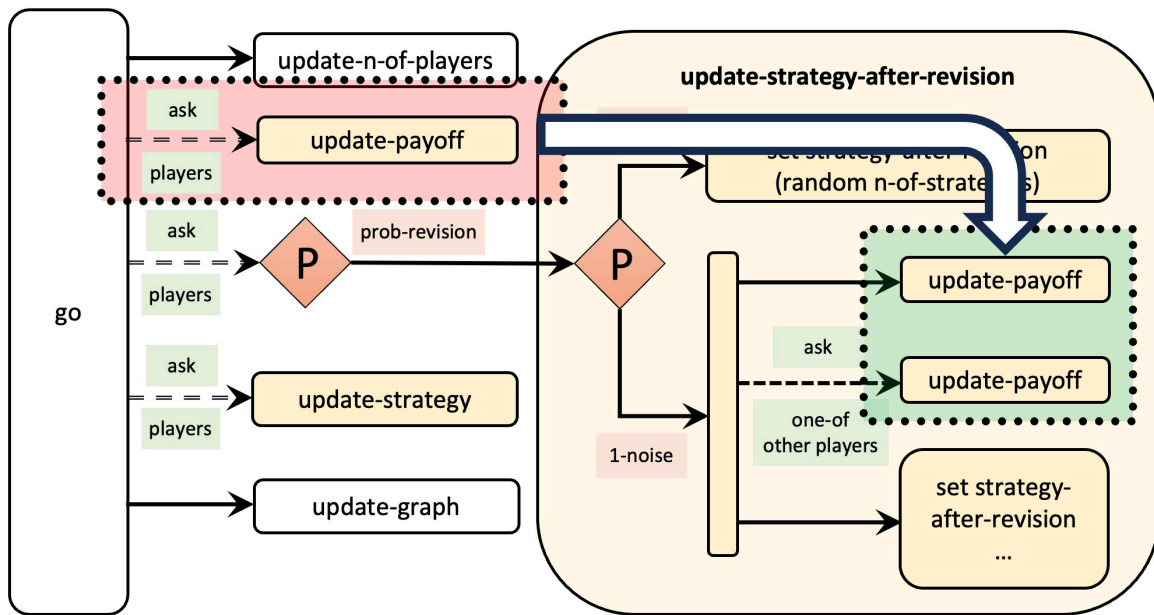
*Figure 3. Skeleton of procedure* to go *in model*
*nxn-imitate-if-better-noise-efficient-but-more-than-once-profiler.nlogo. The dashed red rectangle*
*highlights the code that has been removed. The dashed green rectangle highlights the addition*

In any case, we should double check that we have made our model faster using the profiler extension. The output for the new model (nxn-imitate-if-better-noise-efficient-but-more-than-once-profiler.nlogo) is:

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name              Calls  Incl T(ms)  Excl T(ms)  Excl/calls
UPDATE-STRATEGY   600000   8056.915    8056.915      0.013
UPDATE-PAYOFF     118634   3013.853    3013.853      0.025
UPDATE-STRATEGY-AFT...  59894  4866.334  1852.482    0.031
GO                  1000  13623.073     373.725      0.374
UPDATE-GRAPH        1000    320.337     320.337      0.320
UPDATE-N-OF-PLAYERS 1000      5.762       5.762      0.006


Sorted by Inclusive Time
GO                  1000  13623.073     373.725      0.374
UPDATE-STRATEGY   600000   8056.915    8056.915      0.013
UPDATE-STRATEGY-AFT...  59894  4866.334  1852.482    0.031
UPDATE-PAYOFF     118634   3013.853    3013.853      0.025
UPDATE-GRAPH        1000    320.337     320.337      0.320
UPDATE-N-OF-PLAYERS 1000      5.762       5.762      0.006


Sorted by Number of Calls
UPDATE-STRATEGY   600000   8056.915    8056.915      0.013
UPDATE-PAYOFF     118634   3013.853    3013.853      0.025
UPDATE-STRATEGY-AFT...  59894  4866.334  1852.482    0.031
GO                  1000  13623.073     373.725      0.374
UPDATE-GRAPH        1000    320.337     320.337      0.320
UPDATE-N-OF-PLAYERS 1000      5.762       5.762      0.006
END PROFILING DUMP
```

From the report, we can draw the following inferences:

- We have significantly reduced the number of calls to procedure to update-payoff, from 600 000 down to 118 634. This number makes sense, since there were 600 agents in this simulation, *prob-revision* was 0.1, *noise* was 0.01, each revision without noise makes two calls to the procedure, and we ran the model 1000 ticks (600 × 0.1 × 0.99 × 2 × 1000 = 118 800). Note, however, that for *prob-revision* greater than 0.5, the number of calls would be greater with the new implementation. In general, the expected number of calls to procedure to update-payoff in the new model will be (2 × *prob-revision* × *n-of-players* × ticks).

- Our change in the code has reduced the simulation time considerably, from 22 746 down to 13 623 ms.

- Now simulations spend most of the time running procedure to update-strategy. This procedure is extremely simple, but it is called 600 000 times. All in all, this procedure takes 8 056 / 13 623 ≈ 59% of simulation time.

The changes we conducted have made our model run much faster (since *prob-revision* was 0.1 < 0.5). However, note that, by making these changes, we have implemented a slightly different model. In the original implementation, agents would only update their payoff *once every tick*, but in the new implementation, the same agent could update her payoff several times in the same tick (if, for instance, she is observed several times). This leads to a (very slightly) different distribution of payoffs. Whether this is crucial or not is a matter that would depend on the scientific question we want to answer. In any case, do not worry! We can fix this problem and, at the same time, make our model even more efficient! Let us see how!

Our goal then is to make sure that agents execute procedure to update-payoff at most once in each tick. To make that happen, we have to keep track of whether each individual agent has already executed the procedure in the current tick or not. To that end, we could define a new player-owned variable named played? that will equal **true** if the agent has already executed procedure to update-payoff in the current tick and **false** otherwise. We would also have to set the every players' value of this new variable played? to false at the beginning of the tick. The changes we have to implement in the model to make this work are highlighted below:

```
players-own [
  strategy
  strategy-after-revision
  payoff
  played? ;; <== new line
]

to setup-players
  ;; ... some lines of code ... ;;

  let i 0
  foreach initial-distribution [ j ->
    create-players j [
      set payoff 0
```

```
        set strategy i
        set strategy-after-revision strategy
        set played? false ;; <== new line
    ]
    set i (i + 1)
  ]
  set n-of-players count players
end

to go
  update-n-of-players
  ask players [set played? false] ;; <== new line
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-graph
end

to update-payoff
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
  set played? true ;; <== new line
end

to update-strategy-after-revision
  ifelse (random-float 1 < noise)
    [ set strategy (random n-of-strategies) ]
    [
      let observed-player one-of other players
      if not played? [update-payoff] ;; <== modified lines
      ask observed-player [if not played? [update-payoff]] ;; <==
      if ([payoff] of observed-player) > payoff [
        set strategy ([strategy] of observed-player)
      ]
    ]
end
```

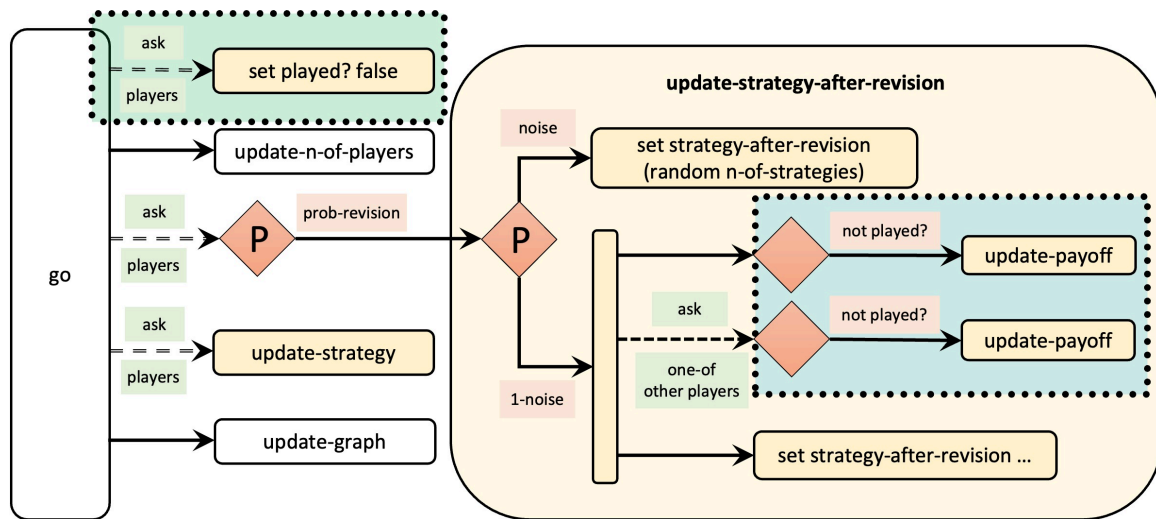The changes implemented above are summarized in Figure 4 below.

*Figure 4. Skeleton of procedure to go in model nxn-imitate-if-better-noise-efficient-played-profiler.nlogo. The dashed green rectangle highlights the main addition in the code. The dashed blue rectangle highlights the main modification in the code*

With this new implementation, the expected number of calls to procedure to update-payoff will be less than (2 × *prob-revision* × *n-of-players* × ticks) –especially so if *prob-revision* is high– and never more than (*n-of-players* × ticks). Thus, the new implementation reduces the number of calls to procedure to update-payoff regardless of the value of *prob-revision*, and it implements the original model precisely. Thus, one would think that we have made our model even faster but... is that true? Let us check with the profiler extension.

The output for the new model (nxn-imitate-if-better-noise-efficient-played-profiler.nlogo) is:

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name             Calls  Incl T(ms)  Excl T(ms)  Excl/calls
GO                1000  21457.028    8050.356       8.050
UPDATE-STRATEGY  600000  7716.470    7716.470       0.013
UPDATE-PAYOFF    110916  3871.700    3871.700       0.035
UPDATE-STRATEGY-AFT...  60322  5557.386  1685.686    0.028
UPDATE-GRAPH       1000   127.998     127.998       0.128
UPDATE-N-OF-PLAYERS 1000    4.817       4.817       0.005


Sorted by Inclusive Time
GO                1000  21457.028    8050.356       8.050
UPDATE-STRATEGY  600000  7716.470    7716.470       0.013
UPDATE-STRATEGY-AFT...  60322  5557.386  1685.686    0.028
UPDATE-PAYOFF    110916  3871.700    3871.700       0.035
UPDATE-GRAPH       1000   127.998     127.998       0.128
UPDATE-N-OF-PLAYERS 1000    4.817       4.817       0.005


Sorted by Number of Calls
UPDATE-STRATEGY  600000  7716.470    7716.470       0.013
UPDATE-PAYOFF    110916  3871.700    3871.700       0.035
UPDATE-STRATEGY-AFT...  60322  5557.386  1685.686    0.028
GO                1000  21457.028    8050.356       8.050
UPDATE-GRAPH       1000   127.998     127.998       0.128
```

```
       UPDATE-N-OF-PLAYERS           1000        4.817        4.817        0.005
       END PROFILING DUMP
```

The following table shows the simulation times and the number of calls to procedure to update-payoff of the different models we have programmed in this chapter up until now, in chronological order:

| Model's name | Summary | Number of calls to procedure to update-payoff | Simulation time (ms) |
|---|---|---|---|
| nxn-imitate-if-better-noise-interactive-profiler.nlogo | Baseline. All players update their payoff in every tick | 600 000 | 22 746 |
| nxn-imitate-if-better-noise-efficient-but-more-than-once-profiler.nlogo | Only revisers and the players they observe update their payoff. Some players may update their payoff more than once in one tick | 118 634 | 13 623 |
| nxn-imitate-if-better-noise-efficient-played-profiler.nlogo | The number of times that procedure to update-payoff is executed is minimal. Implementation with boolean variable named played? | 110 916 | 21 457 |

Table 1. Simulation times and number of calls to procedure to update–payoff of all the models implemented in this chapter up until now

It may come as a surprise that the last changes we made, which ensure that the model calls procedure to update-payoff as few times as possible, actually make the model much slower, from 13 623 ms up to 21 457 ms! Oh, no! How is that possible? We are calling procedure to update-payoff fewer times! (We went from 118 634 down to 110 916.) Take a look at the profiler report above and try to come up with an answer... This is a tricky one...

Why is the last model so slow?

As you can see in the report, the exclusive time spent in procedure to go each time it is executed has increased a lot: from 0.374 to 8.050 ms. The only change we have made in this procedure has been to include the following line:

```
ask players [set played? false] ;; <== new line
```

This line is not as harmless as it seems in current versions of NetLogo, and it is the main responsible for the increase in simulation time.

Our goal now is to implement the same model, but without having to *explicitly* tell every agent at the beginning of every tick that they have not played yet.

One way we can do this is by using a player-owned variable named tick-I-played-last (instead of boolean variable played?) that will store the tick number at which the player updated her payoff for the last time. Then, we would ask players to update their payoff only if the value of their tick-I-played-last is less than the current tick. The key difference with the previous model is that we do not have to set the value of this new variable tick-I-played-last every tick for every player.

Model nxn-imitate-if-better-noise-efficient-tick-I-played-last-profiler.nlogo implements these changes, which are summarized in Figure 5 below.
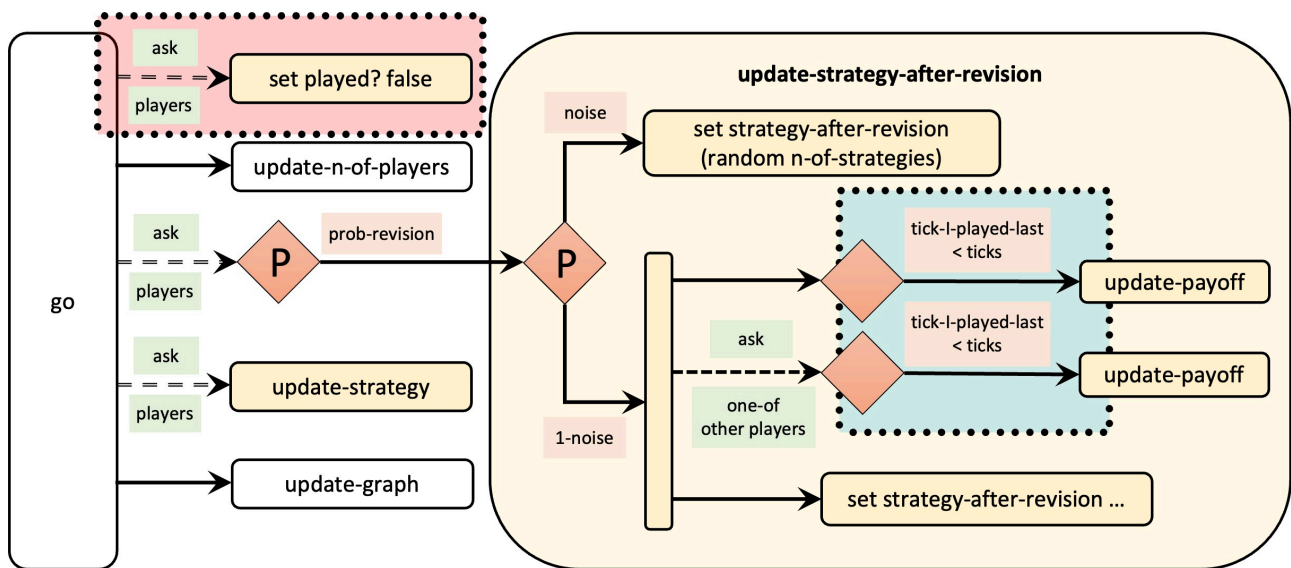


*Figure 5. Skeleton of procedure to go in model nxn-imitate-if-better-noise-efficient-tick-I-played-last-profiler.nlogo. The dashed red rectangle highlights the code that has been removed. The dashed blue rectangle highlights the main modification in the code*

All the necessary changes in the code are highlighted below:

```
players-own [
  strategy
  strategy-after-revision
  payoff
  ;; played?            <== deleted line
  tick-I-played-last ;; <== new line
]

to setup-players

  ;; ... some lines of code ... ;;

  let i 0
  foreach initial-distribution [ j ->
    create-players j [
      set payoff 0
      set strategy i
```

```
      set strategy-after-revision strategy
      ;; set played? false        <== deleted line
      set tick-I-played-last -1 ;; <== new line
    ]
    set i (i + 1)
  ]

  set n-of-players count players
end

to go
  update-n-of-players
  ;; ask players [set played? false] <== deleted line
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-graph
end

to update-payoff
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
  ;; set played? true           <== deleted line
  set tick-I-played-last ticks ;; <== new line
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      let observed-player one-of other players
      ;; vvv modified lines below vvv
      if (tick-I-played-last < ticks) [update-payoff]
      ask observed-player [
        if (tick-I-played-last < ticks) [update-payoff]
      ]

      if ([payoff] of observed-player) > payoff [
        set strategy-after-revision ([strategy] of observed-player)
      ]
    ]
end
```

The output of profiler for the new model (nxn-imitate-if-better-noise-efficient-tick-I-played-last-profiler.nlogo) is:

```
BEGIN PROFILING DUMP
```

```
Sorted by Exclusive Time
Name                 Calls  Incl T(ms)  Excl T(ms)  Excl/calls
UPDATE-STRATEGY      600000   7462.968    7462.968      0.012
UPDATE-PAYOFF        110211   3927.434    3927.434      0.036
UPDATE-STRATEGY-AFT...  59846   5527.475    1600.041      0.027
GO                     1000  13416.363     279.186      0.279
UPDATE-GRAPH           1000    144.769     144.769      0.145
UPDATE-N-OF-PLAYERS    1000      1.964       1.964      0.002


Sorted by Inclusive Time
GO                     1000  13416.363     279.186      0.279
UPDATE-STRATEGY      600000   7462.968    7462.968      0.012
UPDATE-STRATEGY-AFT...  59846   5527.475    1600.041      0.027
UPDATE-PAYOFF        110211   3927.434    3927.434      0.036
UPDATE-GRAPH           1000    144.769     144.769      0.145
UPDATE-N-OF-PLAYERS    1000      1.964       1.964      0.002


Sorted by Number of Calls
UPDATE-STRATEGY      600000   7462.968    7462.968      0.012
UPDATE-PAYOFF        110211   3927.434    3927.434      0.036
UPDATE-STRATEGY-AFT...  59846   5527.475    1600.041      0.027
GO                     1000  13416.363     279.186      0.279
UPDATE-GRAPH           1000    144.769     144.769      0.145
UPDATE-N-OF-PLAYERS    1000      1.964       1.964      0.002
END PROFILING DUMP
```

It is then corroborated that this last version is the most efficient (comparable to the second one if *prob-revision* is low) and it implements the original model exactly. The table below shows the simulation times and the number of calls to procedure to update-payoff of all the models implemented in this chapter up until now:

| Model's name | Summary | Number of calls to procedure to update-payoff | Simulation time (ms) |
|---|---|---|---|
| nxn-imitate-if-better-noise-interactive-profiler.nlogo | Baseline. All players update their payoff in every tick | 600 000 | 22 746 |
| nxn-imitate-if-better-noise-efficient-but-more-than-once-profiler.nlogo | Only revisers and the players they observe update their payoff. Some players may update their payoff more than once in one tick | 118 634 | 13 623 |
| nxn-imitate-if-better-noise-efficient-played-profiler.nlogo | The number of times that procedure to update-payoff is executed is minimal. Implementation with boolean variable named played? | 110 916 | 21 457 |
| nxn-imitate-if-better-noise-efficient-tick-I-played-last-profiler.nlogo | The number of times that procedure to update-payoff is executed is minimal. Implementation with variable named tick-I-played-last | 110 211 | 13 416 |

*Table 2. Simulation times and number of calls to procedure to update–payoff of all the models implemented in this chapter up until now*

Finally, note that, looking at the output printed by `show-profiler-report`, it still seems wasteful to make all agents update their strategies, even when they do not revise them. In exercise 5 below, we ask you to think about how you would make only revising agents call procedure to update-strategy.

## 5.3. Example of computations that we conduct several times when once would do

Let us now focus on the second type of inefficiency pointed out above. Can you identify any computations that we repeatedly conduct in every tick, even though its result does not change?

Note that we undertake the computation:

```
other players
```

several times in every tick, but we could conduct it just once for each agent in each simulation. To be sure, we conduct that operation every time an agent computes her payoff in to update-payoff:

```
let mate one-of other players
```

And also every time an agent revises her strategy in to update-strategy-after-revision:

```
let observed-agent one-of other players
```

This computation may not sound very expensive, but if the number of agents is large, it may well be (see exercise 3 below). To make the model run faster, we could create an individually-owned variable named e.g. other-players, as follows

```
players-own [
  strategy
  strategy-after-revision
  payoff
  tick-I-played-last
  other-players
]
```

And then we should set the new individually-owned variable other-players to the appropriate value only once at the beginning of each simulation (at the end of procedure to setup-players).

```
ask players [ set other-players other players ]
```

Since we may change the number of players at runtime, we should also include the line above in the block of code where we clone or kill agents in procedure to update-n-of-players, i.e.

```
to update-n-of-players
  let diff (n-of-players - count players)
  if diff != 0 [
    ifelse diff > 0
    [ repeat diff [ ask one-of players [hatch-players 1] ] ]
```

```
      [ ask n-of (- diff) players [die] ]
      ask players [set other-players other players]
   ]
  end
```

Once we have done that, in the two lines of code where we had the reporter

```
   other players
```

we should write other-players instead. These changes will make simulations with many players run faster. To find out how much faster, we should use the profiler extension again. The report of profiler for the new model (nxn-imitate-if-better-noise-efficient-tick-I-played-last-and-other-players-profiler.nlogo) is:

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name                 Calls  Incl T(ms)  Excl T(ms)  Excl/calls
UPDATE-STRATEGY      600000   7923.625    7923.625      0.013
UPDATE-PAYOFF        110079   3063.731    3063.731      0.028
UPDATE-STRATEGY-AFT...  59798   4255.300    1191.569      0.020
GO                     1000  12607.398     283.696      0.284
UPDATE-GRAPH           1000    141.295     141.295      0.141
UPDATE-N-OF-PLAYERS    1000      3.482       3.482      0.003

Sorted by Inclusive Time
GO                     1000  12607.398     283.696      0.284
UPDATE-STRATEGY      600000   7923.625    7923.625      0.013
UPDATE-STRATEGY-AFT...  59798   4255.300    1191.569      0.020
UPDATE-PAYOFF        110079   3063.731    3063.731      0.028
UPDATE-GRAPH           1000    141.295     141.295      0.141
UPDATE-N-OF-PLAYERS    1000      3.482       3.482      0.003

Sorted by Number of Calls
UPDATE-STRATEGY      600000   7923.625    7923.625      0.013
UPDATE-PAYOFF        110079   3063.731    3063.731      0.028
UPDATE-STRATEGY-AFT...  59798   4255.300    1191.569      0.020
GO                     1000  12607.398     283.696      0.284
UPDATE-GRAPH           1000    141.295     141.295      0.141
UPDATE-N-OF-PLAYERS    1000      3.482       3.482      0.003
END PROFILING DUMP
```

We can see that the execution time of procedures to update-payoff and to update-strategy-after-revision has been reduced, as expected, but the overall simulation time has not decreased substantially. The table below summarizes the simulation times and the number of calls to procedure to update-payoff of all the models implemented in this chapter:

| Model's name | Summary | Number of calls to procedure to update-payoff | Simulation time (ms) |
|---|---|---|---|
| nxn-imitate-if-better-noise-interactive-profiler.nlogo | Baseline. All players update their payoff in every tick | 600 000 | 22 746 |
| nxn-imitate-if-better-noise-efficient-but-more-than-once-profiler.nlogo | Only revisers and the players they observe update their payoff. Some players may update their payoff more than once in one tick | 118 634 | 13 623 |
| nxn-imitate-if-better-noise-efficient-played-profiler.nlogo | The number of times that procedure to update-payoff is executed is minimal. Implementation with boolean variable named played? | 110 916 | 21 457 |
| nxn-imitate-if-better-noise-efficient-tick-I-played-last-profiler.nlogo | The number of times that procedure to update-payoff is executed is minimal. Implementation with variable named tick-I-played-last | 110 211 | 13 416 |
| nxn-imitate-if-better-noise-efficient-tick-I-played-last-and-other-players-profiler.nlogo | We also use variable other-players | 110 079 | 12 607 |

*Table 3. Simulation times and number of calls to procedure to update–payoff of all the models implemented in this chapter*

## 5.4. Other tips to improve the efficiency of NetLogo code

Railsback et al. (2017) give several guidelines to identify slow parts of NetLogo code and make them run faster, providing specific examples for agent-based models written in NetLogo.

## 5.5. Take-home message

The main lesson we would like you to take home from this chapter is methodological: if you want to make a model run faster, you should definitely use the profiler extension, i.e. start by looking at the data. This is because NetLogo, like any other programming language, has been optimized to compute things in a certain way, and this way may be different from other programming languages you may be familiar with, and may even change from one version of NetLogo to the next. Thus, our most important piece of advice is: look at the evidence by using the profiler extension and try different ways of coding. By doing so, you will develop skills that will help you implement your models more efficiently.

# ⬢ 6. Complete code in the Code tab

```
extensions [profiler]

globals [
  payoff-matrix
  n-of-strategies
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
  tick-I-played-last
  other-players
]

to show-profiler-report
  setup                    ;; set up the model
  profiler:start           ;; start profiling
  repeat 1000 [ go ]       ;; run something you want to measure
  profiler:stop            ;; stop profiling
  print profiler:report    ;; print the results
  profiler:reset           ;; clear the data
end

to setup
  clear-all
  no-display
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end

to setup-players
  let initial-distribution read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
```

```
        "in the payoff matrix (i.e. "
        length payoff-matrix "):\n"
        payoffs
      )
    ]

    let i 0
    foreach initial-distribution [ j ->
      create-players j [
        set payoff 0
        set strategy i
        set strategy-after-revision strategy
        set tick-I-played-last -1
      ]
      set i (i + 1)
    ]

    set n-of-players count players
    ask players [ set other-players other players ]
end

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

to go
  update-n-of-players
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-graph
end

to update-payoff
  let mate one-of other-players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
  set tick-I-played-last ticks
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      let observed-player one-of other-players
```

```
        if (tick-I-played-last < ticks) [update-payoff]
        ask observed-player [
          if (tick-I-played-last < ticks) [update-payoff]
        ]

        if ([payoff] of observed-player) > payoff [
          set strategy-after-revision ([strategy] of observed-player)
        ]
      ]
end

to update-strategy
  set strategy strategy-after-revision
end

to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers

  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end

to update-n-of-players
  let diff (n-of-players - count players)
  if diff != 0 [
    ifelse diff > 0
    [ repeat diff [ ask one-of players [hatch-players 1] ] ]
    [ ask n-of (- diff) players [die] ]
    ask players [ set other-players other players ]
  ]
end
```

# 7. Sample run

Now that we can change the population size at runtime, we can easily explore the question posed above: How does population size affect the dynamics of the *imitate-if-better* decision rule with noise in the Rock-Paper-Scissors game? To do that, let us use the same setting as in the previous chapters (i.e. *payoffs* = [[0 -1 1][1 0 -1][-1 1 0]] and *prob-revision* = 0.1), start with a small population of 60 agents (*n-of-players-for-each-strategy* = [20 20 20]), and then, increase *n-of-players* up to 2000 at runtime. The following video shows a representative run with these settings, where we increased the population size from 60 to 2000 at tick 4000.

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=1061#video-1061-1*

As you can see, when the number of agents is small, the population consistently follows cycles of large amplitude among the three strategies. The cycles are so wide that sometimes one or even two strategies go extinct for a while. In stark contrast, when the population is large, the cycles get much smaller and the population tends to linger around the state where each strategy is used by approximately a third of the population.[1]

# 8. Exercises

You can use the following link to download the last NetLogo model we implemented in this chapter: nxn-imitate-if-better-noise-efficient-tick-I-played-last-and-other-players-profiler.nlogo.

**Exercise 1.** In this chapter we have improved both the interactivity and the efficiency of our model. Can you quantify how much faster the last version of our code runs compared to the previous one (nxn-imitate-if-better-noise)? For the sake of concreteness, use 1000-tick simulations with *payoffs* = [[0 -1 1][1 0 -1][-1 1 0]], initial distribution *n-of-players-for-each-strategy* = [300 300 300], *prob-revision* = 0.1 and *noise* = 0.01.

*Picture by Romain Peli*

---

1. The state where all strategies are equally represented is a globally asymptotically stable state of the mean dynamics of this model (which provides a good approximation for models with large populations). See solution to Exercise 2.3.2.

**Exercise 2.** In this chapter we have drastically reduced the number of times that procedure to update-payoff is called. Can you derive an analytical approximation to the percentage of calls to this procedure that the new implementation is expected to save, as a function of *prob-revision* and *n-of-players*? (You may consider *noise* = 0 initially).

Plot of an approximation to the expected percentage of calls to procedure to update-payoff that are saved

The plot below shows an approximation to the expected percentage of calls to procedure to update-payoff that the last three NetLogo models implemented in this chapter save (i.e., the efficient models that guarantee that players execute this procedure at most once in each tick), as a function of *prob-revision*, for different values of the number of players *n-of-players*. It is interesting to see that this function is not very sensitive to the number of players *n-of-players*. It is assumed that *noise* = 0.
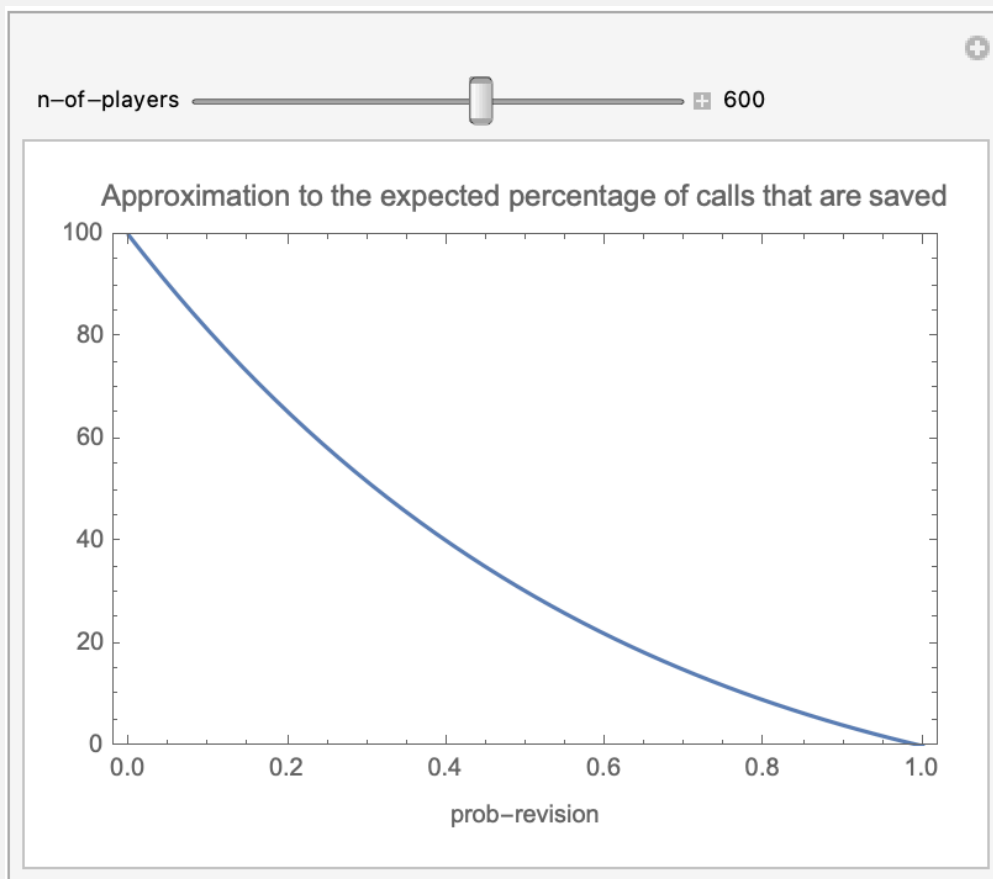
**Exercise 3.** In this chapter we have reduced the number of times the computation `other players` is conducted by creating an individually-owned variable (named other-players). To compare these two approaches, write a short NetLogo program where 10000 agents conduct this operation.

**Exercise 4.** What changes would you have to make in the code so revising agents within the tick update their strategies sequentially (in random order), rather than simultaneously?

> Hint to implement asynchronous strategy updating
>
> It is possible to do this by making a minor change in procedure to go, without touching the rest of the code.

**Exercise 5.** What changes would you have to make in the code so it is only revising agents that call procedure to update-strategy? Use the profiler extension to find out whether the new model is faster. To facilitate the comparison, use the same parameter settings as in Table 3.

> Hint to implement a version where only revising agents call procedure to update-strategy

```
let revising-players players with [random-float 1 < prob-revision]
```

# II-5. Analysis of these models

## 1. Two complementary approaches

Agent-based models are usually analyzed using computer simulation and/or mathematical analysis.

- The computer simulation approach consists in running many simulations –i.e. sampling the model many times– and then, with the data thus obtained, trying to infer general patterns and properties of the model.

- Mathematical approaches do not look at individual simulation runs, but instead analyze the rules that define the model directly, and try to derive their logical implications. Mathematical approaches use deductive reasoning only, so their conclusions follow with logical necessity from the assumptions embedded in the model (and in the mathematics employed).

These two approaches are complementary, in that they can provide fundamentally different insights on the same model. Furthermore, there are synergies to be exploited by using the two approaches together (see e.g. Izquierdo et al. (2013, 2019), Seri (2016), Hilbe and Traulsen (2016), García and van Veelen (2016, 2018) and Hindersin et al. (2019)).

Here we provide several references to material that is helpful to analyze the agent-based models we have developed in this Part II of the book, and illustrate its usefulness with a few examples. Section 2 below deals with the computer simulation approach, while section 3 addresses the mathematical analysis approach.

## 2. Computer simulation approach

The task of running many simulation runs –with the same or different combinations of parameter values– is greatly facilitated by a tool named BehaviorSpace, which is included within NetLogo and is very well documented at NetLogo website. Here we provide an illustration of how to use it.

Consider a coordination game defined by payoffs [[1 0][0 2]], played by 1000 agents who simultaneously revise their strategies with probability 0.01 in every tick, following the imitate if better rule without noise. This is the model implemented in the previous chapter, and it can be downloaded here (nxn-imitate-if-better-noise-efficient.nlogo). This model is stochastic and we wish to study how it *usually* behaves, departing from a situation where both strategies are equally represented. To that end, we could run several simulation runs (say 1000) and plot the average fraction of 1-strategists in every tick, together with the minimum and the maximum values observed across runs in every tick. An illustration of this type of graph is shown in figure 1. Recall that strategies are labeled 0 and 1, so strategy 1 is the one that can get a payoff of 2.
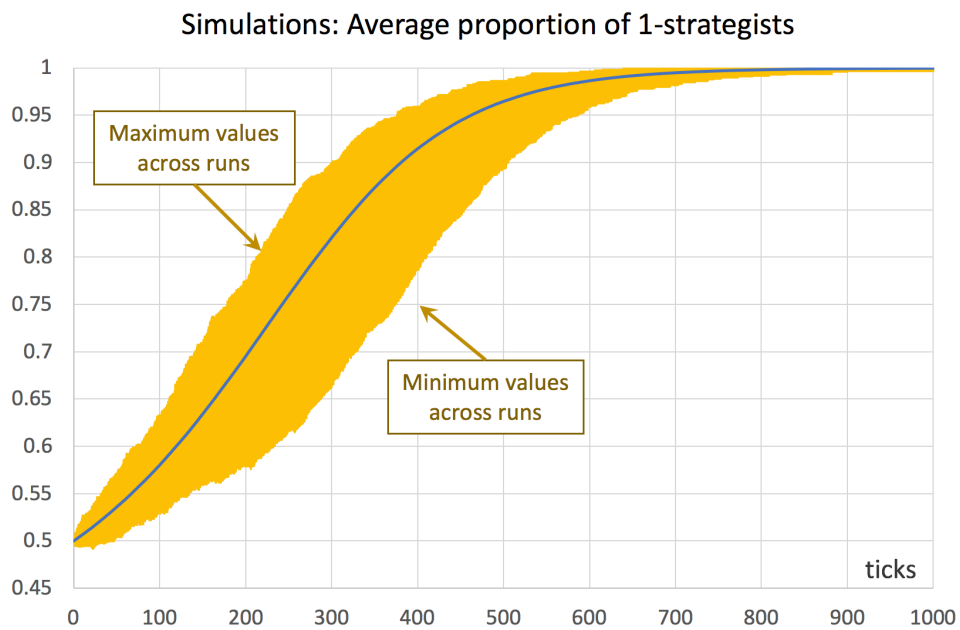
*Figure 1. Average proportion of 1-strategists in an experiment of 1000 simulation runs. Orange error bars show the minimum and maximum values observed across the 1000 runs. Payoffs [[1 0][0 2]]; prob-revision 0.01; noise 0; initial conditions [500 500]*

To set up the computational experiment that will produce the data required to draw figure 1, we just have to go to *Tools* (in the upper menu of NetLogo) and then click on *BehaviorSpace*. The new experiment can be set up as shown in figure 2.
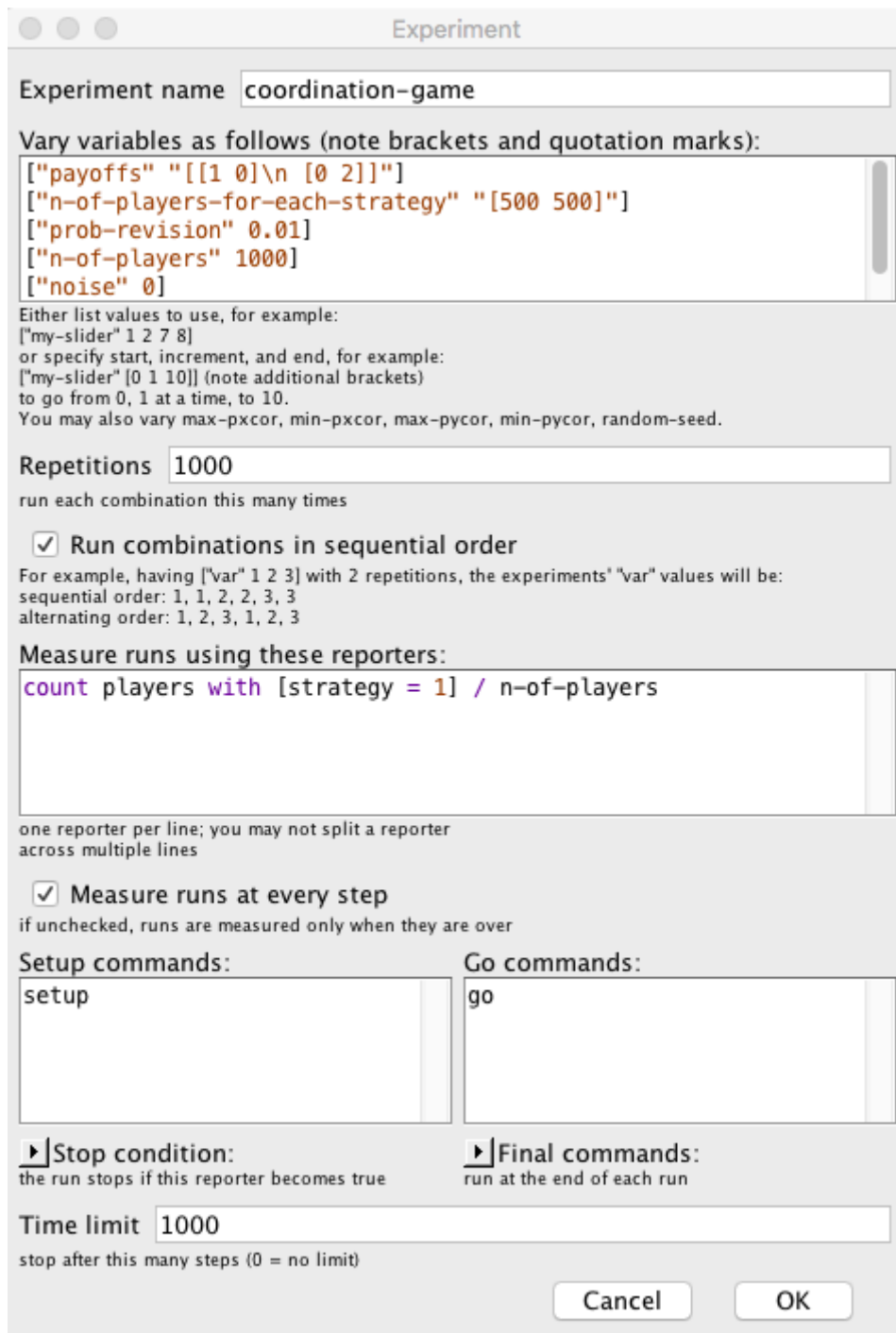
*Figure 2. Experiment setup in BehaviorSpace*

In this particular experiment, we are not changing the value of any parameter, but doing so is straightforward. For instance, if we wanted to run simulations with different values of *prob-revision* –say 0.01, 0.05 and 0.1–, we would just write:

```
[ "prob-revision" 0.01 0.05 0.1 ]
```

If, in addition, we would like to explore the values of *noise* 0, 0.01, 0.02 ... 0.1, we could use the syntax for loops, [*start increment end*], as follows:

```
[ "noise" [0 0.01 0.1] ] ;; note the additional brackets
```

If we made the two changes described above, then the new computational experiment would comprise 33000 runs, since NetLogo would run 1000 simulations for each combination of parameter values (i.e. 3 × 11).

The original experiment shown in figure 2, which consists of 1000 simulation runs only, takes a couple of minutes to run. Once it is completed, we obtain a *.csv* file with all the requested data, i.e. the fraction of 1-strategists in every tick for each of the 1000 simulation runs – a total of 1001000 data points. Then, with the help of a pivot table (within e.g. an Excel spreadsheet), it is easy to plot the graph shown in figure 1. A similar graph that can be easily plotted is one that shows the standard error of the average computed in every tick (see figure 3).[1]
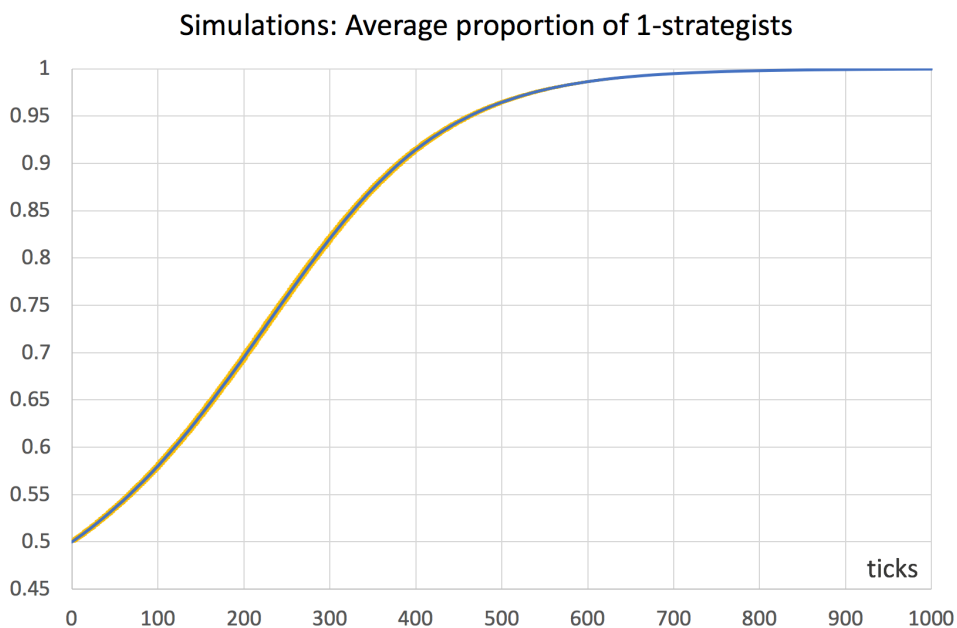


*Figure 3. Average proportion of 1-strategists in an experiment of 1000 simulation runs. Orange error bars show the standard error. Payoffs: [[1 0][0 2]]; prob-revision: 0.01; noise 0; initial conditions [500 500]*

# 3. Mathematical analysis approach. Markov chains

From a mathematical point of view, agent-based models can be usefully seen as time-homogeneous Markov chains (see Gintis (2013) and Izquierdo et al. (2009) for several examples). Doing so can make evident many features of the model that are not apparent before formalizing the model as a Markov chain. Thus, our first recommendation is to learn the basics of this theory. Karr (1990), Kulkarni (1995, chapters 2-4), Norris (1997), Kulkarni (1999, chapter 5), and Janssen and Manca (2006, chapter 3) are all excellent introductions to the topic.

All the models developed in this Part II can be seen as time-homogeneous Markov chains on the finite space of possible strategy distributions. This means that the number of agents that are using

---

1. The standard error of the average equals the standard deviation of the sample divided by the square root of the sample size. In our example, the maximum standard error was well below 0.01.

each possible strategy is all the information we need to know about the present –and the past– of the stochastic process in order to be able to –probabilistically– predict its future as accurately as it is possible. Thus, the number of possible states in these models is $\binom{N+s-1}{s-1}$, where $N$ is the number of agents and $s$ the number of strategies.[2]

In some simple cases, a full Markov analysis can be conducted by deriving the transition probabilities of the Markov chain and operating directly with them. Section 3.1 illustrates how this type of analysis can be undertaken on models with 2 strategies where agents revise their strategies sequentially.

However, in many other models a full Markov analysis is unfeasible because the exact formulas can lead to very cumbersome expressions, or may even be too complex to evaluate. This is often the case if the number of states is large.[3] In such situations, one can still take advantage of powerful approximation results, which we introduce in section 3.2.

## 3.1. Markov analysis of 2-strategy evolutionary processes where agents switch strategies sequentially

In this section we study 2-strategy evolutionary processes where agents switch strategies sequentially. For simplicity, we will asume that there is one revision per tick, but several revisions could take place in the same tick as long as they occurred sequentially. These processes can be formalized as birth-death chains, a special type of Markov chains for which various analytical results can be derived.

Note that, in the model implemented in the previous chapter (and simulated in section 2 above), agents do not revise their strategies sequentially, but simultaneously within the tick. The difference between these two updating schemes are small for low probabilities of revision, so the formal analysis presented here will be useful to analyze the computer model as long *prob-revision* is not too high.

### 3.1.1. Markov chain formulation

Consider a population of $N$ agents who repeatedly play a symmetric 2-player 2-strategy game. The two possible strategies are labeled 0 and 1. In every tick, one random agent is given the opportunity to revise his strategy, and he does so according to a certain decision rule (such as the imitate if better rule, the imitative pairwise-difference rule or the best experienced payoff rule).

Let $X_k$ be the proportion of the population using strategy 1 at tick $k$. The evolutionary process described above induces a Markov chain $\{X_k\}$ on the state space $S^N = \{0, \frac{1}{N}, \ldots, 1\}$. We do not have to keep track of the proportion of agents using strategy 0 because there are only two strategies, so the two proportions must add up to one. Since there is only one revision per tick, note that there are only three possible transitions: one implies increasing $X_k$ by $\frac{1}{N}$, another one implies decreasing

---

2. This result can be easily derived using a "stars and bars" analogy.

3. As an example, in a 4-strategy game with 1000 players, the number of possible states (i.e. strategy distributions) is $\binom{1000+4-1}{4-1} = 167,668,501.$

$X_k$ by $\frac{1}{N}$, and the other one leaves the state unchanged. Let us denote the transition probabilities as follows:

$$p(x) = \mathbb{P}(X_{k+1} = x + \tfrac{1}{N} \mid X_k = x)$$

$$q(x) = \mathbb{P}(X_{k+1} = x - \tfrac{1}{N} \mid X_k = x)$$

Thus, the probability of staying at the same state after one tick is:

$$\mathbb{P}(X_{k+1} = x \mid X_k = x) = 1 - p(x) - q(x)$$

This Markov chain has two important properties: the state space $S^N = \{0, \frac{1}{N}, \ldots, 1\}$ is endowed with a linear order and all transitions move the state one step to the left, one step to the right, or leave the state unchanged. These two properties imply that the Markov chain is a birth-death chain. Figure 4 below shows the transition diagram of this birth-death chain, ignoring the self-loops.
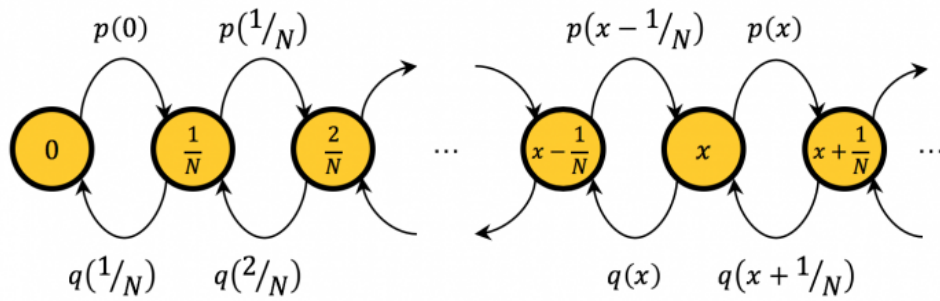


Figure 4. *Transition diagram of a birth–death chain*

The transition matrix $P$ of a Markov chain gives us the probability of going from one state to another. In our case, the elements $P_{ij} = \mathbb{P}(X_{k+1} = \frac{j-1}{N} \mid X_k = \frac{i-1}{N})$ of the transition matrix are:

$$
P = \begin{pmatrix}
1 - \sum\limits_{j\neq 1} P_{1j} & p(0) & 0 & 0 & \cdots & 0 \\
q(\frac{1}{N}) & 1 - \sum\limits_{j\neq 2} P_{2j} & p(\frac{1}{N}) & 0 & \cdots & 0 \\
0 & q(\frac{2}{N}) & 1 - \sum\limits_{j\neq 3} P_{3j} & p(\frac{2}{N}) & \ddots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & q(\frac{N-1}{N}) & 1 - \sum\limits_{j\neq N} P_{Nj} & p(\frac{N-1}{N}) \\
0 & \cdots & 0 & 0 & q(\frac{N}{N}) & 1 - \sum\limits_{j\neq N+1} P_{N+1\,j}
\end{pmatrix}
$$

In our evolutionary process, the transition probabilities $p(x)$ and $q(x)$ are determined by the decision rule that agents use. Let us see how this works with a specific example. Consider

the coordination game defined by payoffs [[1 0][0 2]], played by agents who sequentially revise their strategies according to the imitate if better rule (without noise). This is very similar to the model we have simulated in section 2 above. The only difference is that now we are assuming revisions take place sequentially, while in the model simulated in section 2 agents revise their strategies simultaneously within the tick (with probability 0.01). Assuming that the fraction of agents that revise their strategies simultaneously is low (in this case, about 1%), the difference between the formal model and the computer model will be small.[4]

Let us derive $p(x) = \mathbb{P}(X_{k+1} = x + \frac{1}{N} \mid X_k = x)$. Note that the state increases by $\frac{1}{N}$ if and only if the revising agent is using strategy 0 and he switches to strategy 1. In the game with payoffs [[1 0][0 2]], this happens if and only if the following conditions are satisfied:

- the agent who is randomly drawn to revise his strategy is playing strategy 0 (an event which happens with probability $(1 - x)$),

- the agent that is observed by the revising agent is playing strategy 1 (an event which happens with probability $\frac{xN}{N-1}$; note that there are $xN$ agents using strategy 1 and the revising agent observes *another* agent, thus the divisor $N - 1$), and

- the observed agent's payoff is 2, i.e. the observed agent –who is playing strategy 1– played with an agent who was also playing strategy 1 (an event which happens with probability $\frac{xN-1}{N-1}$; note that the observed agent plays with *another* agent who is also playing strategy 1).

Therefore:

$$p(x) = (1 - x)\frac{xN}{N-1}\frac{xN-1}{N-1}$$

Note that, in this case, the payoff obtained by the revising agent is irrelevant.

We can derive $q(x) = \mathbb{P}(X_{k+1} = x - \frac{1}{N} \mid X_k = x)$ in a similar fashion. Do you want to give it a try before reading the solution?

## Computation of $q(x)$

Note that the state decreases by $\frac{1}{N}$ if and only if the revising agent is using strategy 1 and he switches to strategy 0. In the game with payoffs [[1 0][0 2]], this happens if and only if the following conditions are satisfied:

- the agent who is randomly drawn to revise his strategy is playing strategy 1 (an event which happens with probability $x$),

---

4. We give you a hint to program the asynchronous model in exercise 4 of chapter II-4.

- the agent that is observed by the revising agent is playing strategy 0 (an event which happens with probability $\frac{(1-x)N}{N-1}$; note that there are $(1-x)N$ 0-strategists and the revising agent observes *another* agent, thus the divisor $N-1$),

- the revising agent's payoff is 0, i.e. the revising agent played with an agent who was playing strategy 0 (an event which happens with probability $\frac{(1-x)N}{N-1}$; note that the revising agent plays with *another* agent, thus the divisor $N-1$).

- the observed agent's payoff is 1, i.e. the observed agent, who is playing strategy 0, played with an agent who was also playing strategy 0 (an event which happens with probability $\frac{(1-x)N-1}{N-1}$; note that the observed agent plays with *another* agent who is also playing strategy 0).

Therefore:

$$q(x) = x\frac{(1-x)N}{N-1}\frac{(1-x)N}{N-1}\frac{(1-x)N-1}{N-1}$$

With the formulas of $p(x)$ and $q(x)$ in place, we can write the transition matrix of this model for any given $N$. As an example, this is the transition matrix $P$ for $N = 10$:

$$P = \begin{pmatrix}
1. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0.089 & 0.911 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0.123 & 0.857 & 0.020 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0.121 & 0.827 & 0.052 & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0.099 & 0.812 & 0.089 & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0.069 & 0.808 & 0.123 & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 0.040 & 0.812 & 0.148 & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0.017 & 0.827 & 0.156 & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.004 & 0.857 & 0.138 & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0 & 0.911 & 0.089 \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1.
\end{pmatrix}$$

And here's a little *Mathematica*® script that can be used to generate the transition matrix for any $N$:

```
n = 10;
p[x_] := (1 - x) (x n/(n - 1)) ((x n - 1)/(n - 1))
q[x_] := x (((1 - x)n)/(n - 1))^2 ((1 - x)n - 1)/(n - 1)

P = SparseArray[{
```

```
    {i_, i_} -> (1 - p[(i - 1)/n] - q[(i - 1)/n]),
    {i_, j_} /; i == j - 1 -> p[(i - 1)/n],
    {i_, j_} /; i == j + 1 -> q[(i - 1)/n]
    }, {n + 1, n + 1}];


MatrixForm[P]
```

### 3.1.2. Transient dynamics

In this section, we use the transition matrix $P$ we have just derived to compute the transient dynamics of our two-strategy evolutionary process, i.e. the probability distribution of $X_k$ at a certain $k \geq 0$. Naturally, this distribution generally depends on initial conditions.

To be concrete, imagine we set some initial conditions, which we express as a (row) vector $a_0$ containing the initial probability distribution over the $N + 1$ states of the system at tick $k = 0$, i.e. $a_0 = \left(a_0(0), a_0(\frac{1}{N}), \ldots, a_0(\frac{N-1}{N}), a_0(1)\right)$, where $a_0(i) = \mathbb{P}(X_0 = i)$. If initial conditions are certain, i.e. if $X_0 = x_0$, then all elements of $a_0$ are 0 except for $a(x_0)$, which would be equal to 1.

Our goal is to compute the vector $a_k = \left(a_k(0), a_k(\frac{1}{N}), \ldots, a_k(\frac{N-1}{N}), a_k(1)\right)$, which contains the probability of finding the process in each of the possible $N + 1$ states at tick $k$ (i.e. after $k$ revisions), having started at initial conditions $a_0$. This $a_k$ is a row vector representing the probability mass function of $X_k$.

To compute $a_k$, it is important to note that the $t$-step transition probabilities $\mathbb{P}(X_{k+t} = y \mid X_k = x)$ are given by the entries of the $t$th power of the transition matrix, i.e.:

$$(P^t)_{ij} = \mathbb{P}(X_{k+t} = \tfrac{j-1}{N} \mid X_k = \tfrac{i-1}{N})$$

Thus, we can easily compute the transient distribution $a_k$ simply by multiplying the initial conditions $a_0$ by the $k$th power of the transition matrix $P$:

$$a_k = a_0 P^k$$

As an example, consider the evolutionary process we formalized as a Markov chain in the previous section, with $N = 100$ imitate if better agents playing the coordination game [[1 0][0 2]]. Let us start at initial state $X_0 = 0.5$, i.e. $a_0 = (0, \ldots, 0, 1, 0, \ldots, 0)$, where the solitary 1 lies exactly at the middle of the vector $a_0$ (i.e. at position $(\frac{N}{2} + 1)$). Figure 5 shows the distributions $a_{100}, a_{200}, a_{300}, a_{400}$ and $a_{500}$.
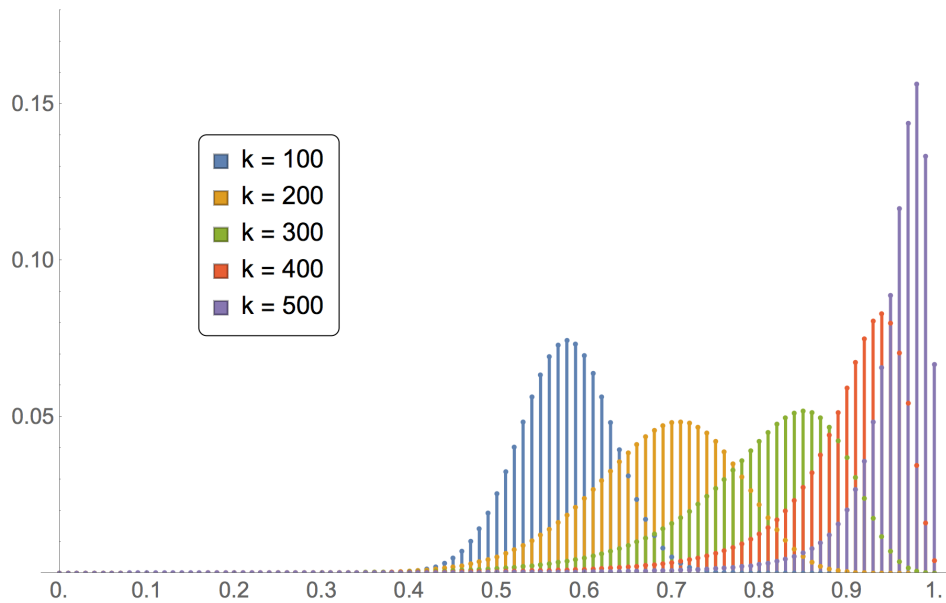
*Figure 5. Probability mass function of $X_k$ at different ticks. Number of agents $N = 100$. Initial conditions $X_0 = 0.5$*

To produce figure 5, we have computed the transition matrix with the previous *Mathematica*® script (having previously set the number of agents to 100) and then we have run the following two lines:

```
a0 = UnitVector[n + 1, 1 + n/2];
ListPlot[Table[a0.MatrixPower[N@P, i], {i, 100, 500, 100}], PlotRange -> All]
```

Looking at the probability distribution of $X_{500}$, it is clear that, after 500 revisions, the evolutionary process is very likely to be at a state where most of the population is using strategy 1. There is even a substantial probability (~6.66%) that the process will have reached the absorbing state where everyone in the population is using strategy 1. Note, however, that all the probability distributions shown in figure 5 have full support, i.e. the probability of reaching the absorbing state where no one uses strategy 1 after 100, 200, 300, 400 or 500 is very small, but strictly positive. As a matter of fact, it is not difficult to see that, given that $N = 100$ and $X_0 = 0.5$ (i.e. initially there are 50 agents using strategy 1), $a_k(0) = \mathbb{P}(X_k = 0 \mid X_0 = 0.5) > 0$ for any $k \geq 50$.

Finally, to illustrate the sensitivity of transient dynamics to initial conditions, we replicate the computations shown in figure 5, but with initial conditions $X_0 = 0.4$ (figure 6) and $X_0 = 0.3$ (figure 7).
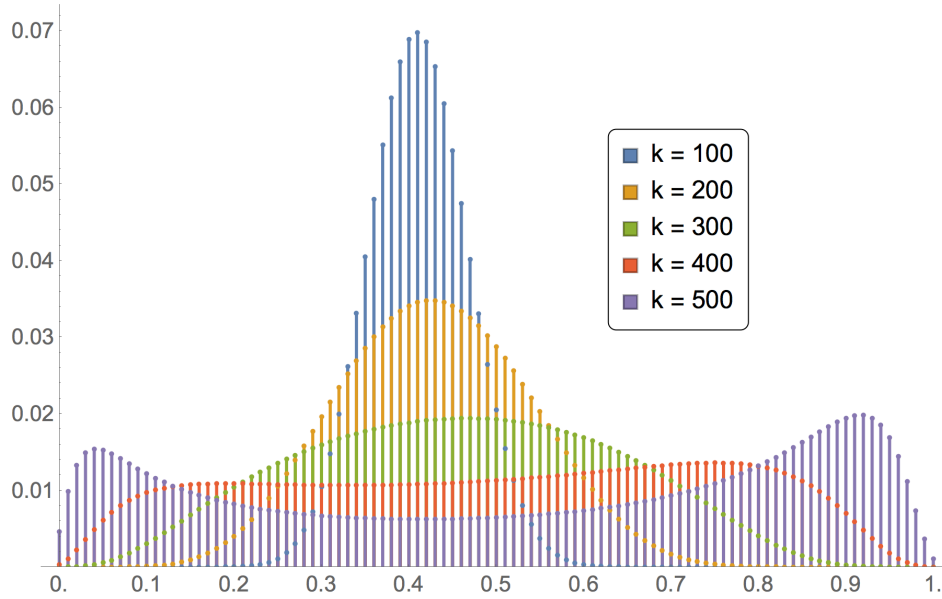
*Figure 6. Probability mass function of $X_k$ at different ticks. Number of agents $N = 100$. Initial conditions $X_0 = 0.4$*
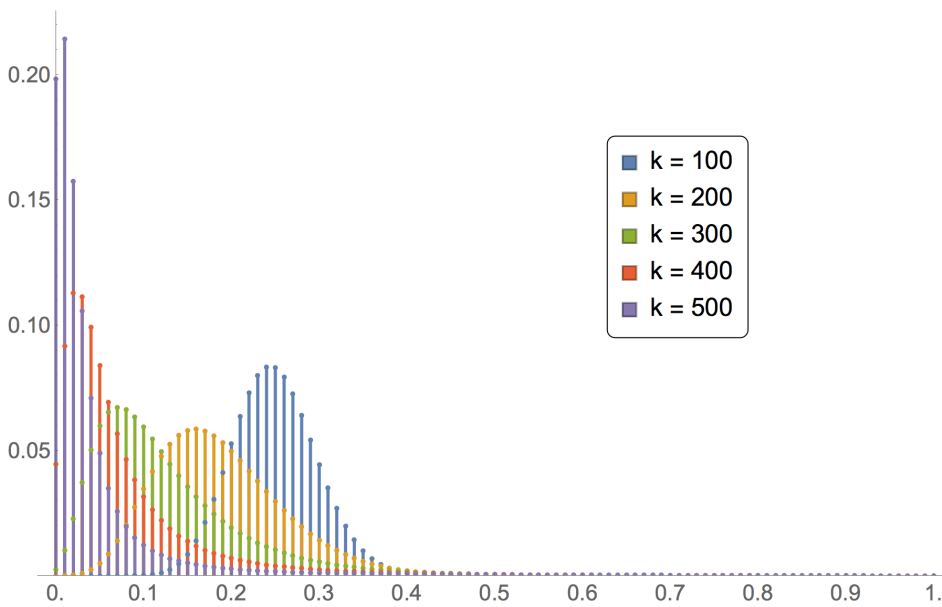


*Figure 7. Probability mass function of $X_k$ at different ticks. Number of agents $N = 100$. Initial conditions $X_0 = 0.3$*

Besides the probability distribution of $X_k$ at a certain $k \geq 0$, we can analyze many other interesting properties of a Markov chain, such as the expected hitting time (or first passage time) of a certain state $i \in S$, which is the expected time at which the process first reaches state $i$. For general Markov chains, this type of results can be found in any of the references mentioned at the beginning of section 3. For birth-death chains specifically, Sandholm (2010a, section 11.A.3) provides simple formulas to compute expected hitting times and hitting probabilities (i.e. the probability that the birth-death chain reaches state $i$ before state $j$).

### 3.1.3. Infinite-horizon behavior

In this section we wish to study the infinite-horizon behavior of our evolutionary process, i.e. the distribution of $X_k$ when the number of ticks $k$ tends to infinity. This behavior generally depends on initial conditions, but we focus here on a specific type of Markov chain (irreducible and aperiodic) whose limiting behavior does not depend on initial conditions. To understand the concepts of irreducibility and aperiodicity, we recommend you read any of the references on Markov chains provided at the beginning of section 3. Here we just provide sufficient conditions that guarantee that a (time-homogeneous) Markov chain is irreducible and aperiodic:

> **Sufficient conditions for irreducibility and aperiodicity of time-homogeneous Markov chains**
>
> - If it is possible to go from any state to any other state in one single step ( $P_{ij} > 0$ for all $i \neq j$) and there are more than 2 states, then the Markov chain is irreducible and aperiodic.
> - If it is possible to go from any state to any other state in a finite number of steps, and there is at least one state in which the system may stay for two consecutive steps ($P_{ii} > 0$ for some $i$), then the Markov chain is irreducible and aperiodic.
> - If there exists a positive integer $m$ such that $(P^m)_{ij} > 0$ for all $i$ and $j$, then the Markov chain is irreducible and aperiodic.

If one sees the transition diagram of a Markov chain (see e.g. Figure 4 above) as a directed graph (or network), the conditions above can be rewritten as:

- The network contains more than two nodes and there is a directed link from every node to every other node.
- The network is strongly connected and there is at least one loop.
- There exists a positive integer $m$ such that there is at least one walk of length $m$ from any node to every node (including itself).

The 2-strategy evolutionary process we are studying in this section is not necessarily irreducible if there is no noise. For instance, the coordination game played by *imitate-if-better* agents analyzed in section 3.1.2 is not irreducible. That model will eventually reach one of the two absorbing states where all the agents are using the same strategy, and stay in that state forever. The probability of ending up in one or the other absorbing state depends on initial conditions (see Figure 8).[5]

---

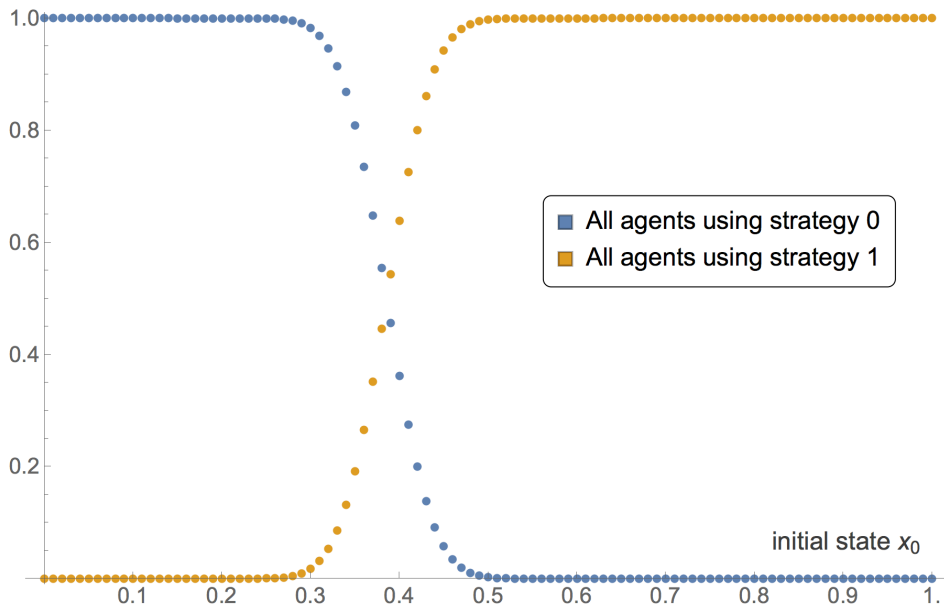5. These probabilities are sometimes called "fixation probabilities".

*Figure 8. Probability of ending up in each of the two absorbing states for different initial states $x_0$, in the coordination game [[1 0][0 2]] played by $N = 100$ imitate-if-better agents*

However, if we add noise in the agents' decision rule –so there is always the possibility that revising agents choose any strategy–, then it is easy to see that the second sufficient condition for irreducibility and aperiodicity above is fulfilled.[6]

Generally, in irreducible and aperiodic Markov chains $\{X_k\}$ with state space $S$ (henceforth IAMCs), the probability mass function of $X_k$ approaches a limit as $k$ tends to infinity. This limit is called the limiting distribution, and is denoted here by $\mu$, a vector with components $\mu(i)$ which denote the probability of finding the system in state $i \in S$ in the long run. Formally, in IAMCs the following limit exists and is unique (i.e. independent of initial conditions):

$$\lim_{k\to\infty} \mathbb{P}(X_k = i) = \mu(i) > 0 \ \text{ for all } i \in S.$$

Thus, in IAMCs the probability of finding the system in each of its states in the long run is strictly positive and independent of initial conditions. Importantly, in IAMCs the limiting distribution $\mu$ coincides with the occupancy distribution $\mu^*$, which is the long-run fraction of the time that the IAMC spends in each state.[7] This means that we can estimate the limiting distribution $\mu$ of a IAMC using the computer simulation approach by running just one simulation for long enough (which enables us to estimate $\mu^*$).

---

6. In terms of the transition probabilities $p(x)$ and $q(x)$, adding noise implies that $p(x) > 0$ for $x < 1$ (i.e. you can always move one step to the right unless $x$ already equals 1), $q(x) > 0$ for $x > 0$ (i.e. you can always move one step to the left unless $x$ already equals 0) and $p(x) + q(x) < 1$ for all $x \in [0, 1]$ (i.e. you can always stay where you are).

7. Formally, the occupancy of state $i$ is defined as:
$$\mu^*(i) = \lim_{k\to\infty} \frac{\mathbb{E}(N_i(k))}{k+1}$$
where $N_i(k)$ denotes the number of times that the Markov chain visits state $i$ over the time span $\{0, 1, \ldots, k\}$.

In any IAMC, the limiting distribution $\mu$ can be computed as the left eigenvector of the transition matrix $P$ corresponding to eigenvalue 1.[8] Note, however, that computing eigenvectors is computationally demanding when the state space is large. Fortunately, for irreducible and aperiodic birth-death chains (such as our 2-strategy evolutionary process with noise), there is an analytic formula for the limiting distribution that is easy to evaluate:[9]

$$\mu(i) = \mu(0) \prod_{j=1}^{Ni} \frac{p(\frac{j-1}{N})}{q(\frac{j}{N})} \;\; \text{for } i \in \{0, \tfrac{1}{N}, \ldots, 1\}$$

where the value of $\mu(0) = \left( \sum_{h=0}^{N} \prod_{j=1}^{h} \frac{p(\frac{j-1}{N})}{q(\frac{j}{N})} \right)^{-1}$ is derived by imposing that the elements of $\mu$ must add up to 1. This formula can be easily implemented in *Mathematica*®:

```
μ = Normalize[FoldList[Times, 1, Table[p[(j-1)/n]/q[j/n],{j, n}]], Total]
```

Note that the formula above is only valid for irreducible and aperiodic birth-death chains. An example of such a chain would be the model where a number of imitate if better agents are playing the coordination game [[1 0][0 2]] with noise. Thus, for this model we can easily analyze the impact of noise on the limiting distribution. Figure 9 illustrates this dependency.
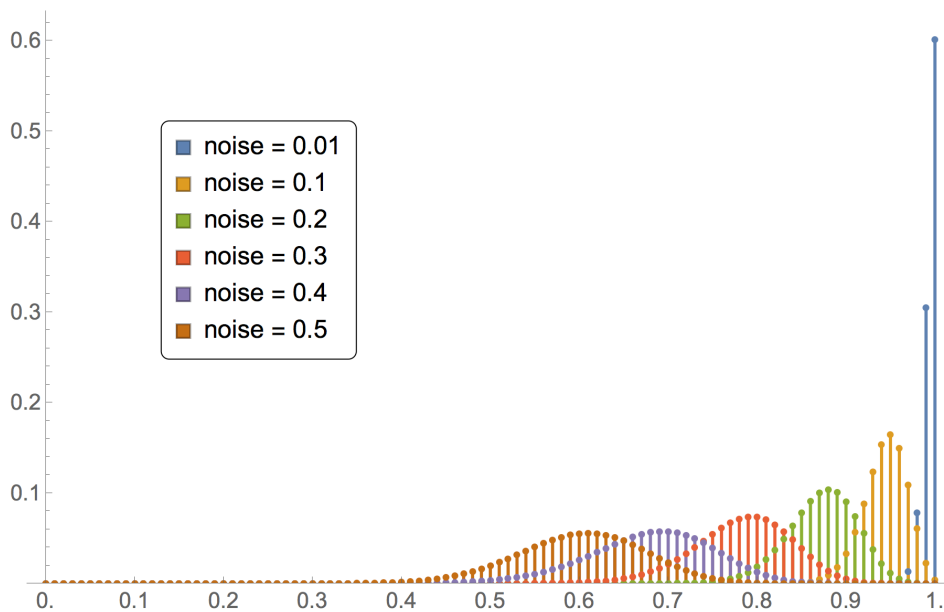


*Figure 9. Limiting distribution $\mu$ for different values of noise in the coordination game [[1 0][0 2]] played by $N = 100$ imitate–if–better agents*

Figure 9 has been created by running the following *Mathematica*® script:

---

8. The second-largest eigenvalue modulus of the transition matrix $P$ determines the rate of convergence to the limiting distribution.

9. For the derivation of this formula, see e.g. Sandholm (2010a, example 11.A.10, p. 443).

```
n = 100;


p[x_, noise_]:= (1-x)((1-noise)(x n/(n-1))((x n - 1)/(n-1)) + noise/2)
q[x_, noise_]:= x((1-noise)(((1-x)n)/(n-1))^2 ((1-x)n-1)/(n-1) + noise/2)


μs = Map[Normalize[
        FoldList[Times, 1, Table[p[(j-1)/n, #] / q[j/n, #], {j, n}]]
        , Total]&, {0.01, 0.1, 0.2, 0.3, 0.4, 0.5}];


ListPlot[μs, DataRange->{0, 1}, PlotRange->{0, All}, Filling -> Axis]
```

The limiting distribution of birth-death chains can be further characterized using results in Sandholm (2007).

## 3.2. Approximation results

In many models, a full Markov analysis cannot be conducted because the exact formulas are too complicated or because they may be too computationally expensive to evaluate. In such cases, we can still apply a variety of approximation results. This section introduces some of them.

### 3.2.1. Deterministic approximations of transient dynamics when the population is large. The mean dynamic

When the number of agents is sufficiently large, the mean dynamic of the process provides a good deterministic approximation to the dynamics of the stochastic evolutionary process over finite time spans. In this section we are going to analyze the behavior of our evolutionary process as the population size $N$ becomes large, so we make this dependency on $N$ explicit by using superscripts for $X_k^N$, $p^N(x)$ and $q^N(x)$.

Let us start by illustrating the essence of the mean dynamic approximation with our running example where $N$ *imitate-if-better* agents are playing the coordination game [[1 0][0 2]] without noise. Initially, half the agents are playing strategy 1 (i.e. $X_0^N = 0.5$). Figures 10, 11 and 12 show the expected proportion of 1-strategists $\mathbb{E}(X_k^N \mid X_0^N = 0.5)$ against the number of revisions $k$ (scaled by $N$), together with the 95% band for $X_k^N$, for different population sizes.[10] Figure 10 shows the transient dynamics for $N = 100$, figure 11 for $N = 1000$ and figure 12 for $N = 10000$. These figures show exact results, computed as explained in section 3.1.2.

---

10. For each value of $k$, the band is defined by the smallest interval $\{i, j\}$ that leaves less than 2.5% probability at both sides, i.e. $\mathbb{P}(X_k^N < i \mid X_0^N = 0.5) < 0.025$ and $\mathbb{P}(X_k > j \mid X_0^N = 0.5) < 0.025$, with $i, j \in S^N$.
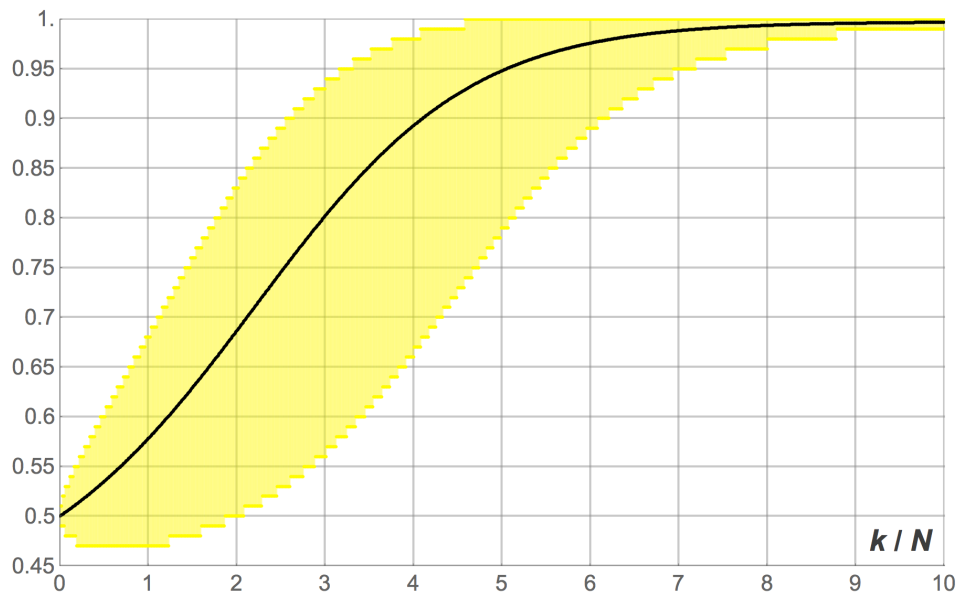
*Figure 10. Expected proportion of agents playing strategy 1 and its 95% band (in yellow). Model: coordination game [[1 0][0 2]] played by $N = 100$ imitate-if-better agents. Initially, half the population is using strategy 1*
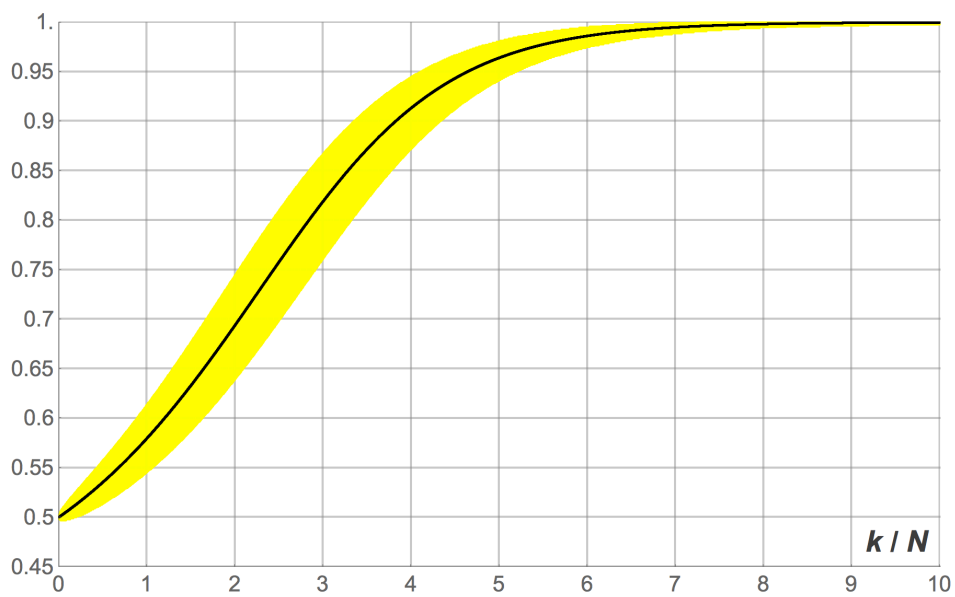


*Figure 11. Expected proportion of agents playing strategy 1 and its 95% band (in yellow). Model: coordination game [[1 0][0 2]] played by $N = 1000$ imitate-if-better agents. Initially, half the population is using strategy 1*

*Figure 12. Expected proportion of agents playing strategy 1 and its 95% band (in yellow). Model: coordination game [[1 0][0 2]] played by $N = 10000$ imitate-if-better agents. Initially, half the population is using strategy 1*

Looking at figures 10, 11 and 12 it is clear that, as the number of agents $N$ gets larger, the stochastc evolutionary process $\{X_k^N\}$ gets closer and closer to its expected motion. The intuition is that, as the number of agents gets large, the fluctuations of the evolutionary process around its expected motion tend to average out. In the limit when $N$ goes to infinity, the stochastic evolutionary process is very likely to behave in a nearly deterministic way, mirroring a solution trajectory of a certain ordinary differential equation called the *mean dynamic*.

To derive the mean dynamic of our 2-strategy evolutionary process, we consider the behavior of the process $\{X_k^N\}$ over the next $dt$ time units, departing from state $x$. We define one unit of clock time as $N$ ticks, i.e. the time over which every agent is expected to receive exactly one revision opportunity. Thus, over the time interval $dt$, the number of agents who are expected to receive a revision opportunity is $Ndt$. Of these $Ndt$ agents who revise their strategies, $p^N(x)Ndt$ are expected to switch from strategy 0 to strategy 1 and $q^N(x)Ndt$ are expected to switch from strategy 1 to strategy 0. Hence, the expected change in the number of agents that are using strategy 1 over the time interval $dt$ is $(p^N(x) - q^N(x))Ndt$. Therefore, the expected change in the proportion of agents using strategy 1, i.e. the expected change in state at $x$, is

$$dx = \tfrac{1}{N}\left(p^N(x) - q^N(x)\right)Ndt = \left(p^N(x) - q^N(x)\right)dt$$

Note that the transition probabilities $p^N(x)$ and $q^N(x)$ may depend on $N$. This does not represent a problem as long as this dependency vanishes as $N$ gets large. In that case, to deal with that dependency, we take the limit of $p^N(x)$ and $q^N(x)$ as $N$ goes to infinity since, after all, the mean dynamic approximation is only valid for large $N$. Thus, defining

$$p^\infty(x) = \lim_{N \to \infty} p^N(x)$$

and

$$q^{\infty}(x) = \lim_{N \to \infty} q^N(x)$$

we arrive at the mean dynamic equation:

$$\tfrac{d}{dt}x = \dot{x} = p^{\infty}(x) - q^{\infty}(x)$$

As an illustration of the usefulness of the mean dynamic to approximate transient dynamics, consider the simulations of the coordination game example presented in section 2. We already computed the transition probabilities $p^N(x)$ and $q^N(x)$ in section 3.1.1:

$$p^N(x) = (1-x)\tfrac{xN}{N-1}\tfrac{xN-1}{N-1}$$

$$q^N(x) = x\tfrac{(1-x)N}{N-1}\tfrac{(1-x)N}{N-1}\tfrac{(1-x)N-1}{N-1}$$

Thus, the mean dynamic reads:

$$\dot{x} = p^{\infty}(x) - q^{\infty}(x) = (1-x)x^2 - x(1-x)^3 = x(x-1)\left(x^2 - 3x + 1\right)$$

where $x$ stands for the fraction of 1-strategists. The solution of the mean dynamic with initial condition $x_0 = 0.5$ is shown in figure 13 below. It is clear that the mean dynamic provides a remarkably good approximation to the average transient dynamics plotted in figures 1 and 3.[11] And, as we have seen, the greater the number of agents, the closer the stochastic process will get to its expected motion.

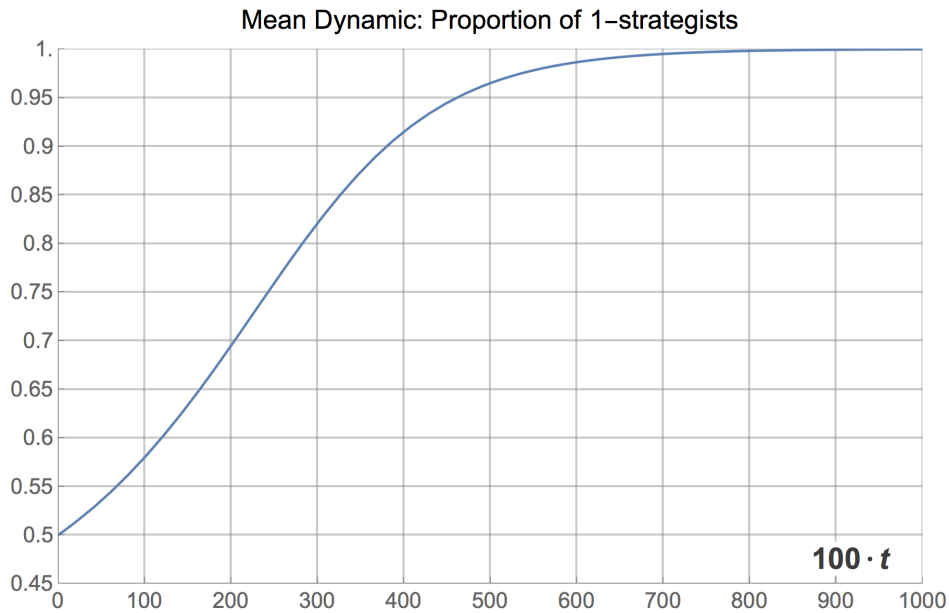Mean Dynamic: Proportion of 1–strategists



Figure 13. Trajectory of the mean dynamic of the example in section 2, showing the proportion of 1-strategists as a function of time (rescaled to match figures 1 and 3)

---

11. Note that one unit of clock time in the mean dynamic is defined in such a way that each player expects to receive one revision opportunity per unit of clock time. In the model simulated in section 2, *prob-revision* = 0.01, so one unit of clock time corresponds to 100 ticks (i.e. 1 / *prob-revision*).

Naturally, the mean dynamic can be solved for many different initial conditions, providing an overall picture of the transient dynamics of the model when the population is large. Figure 14 below shows an illustration, created with the following *Mathematica*® code:

```
Plot[
  Evaluate[
   Table[
    NDSolveValue[{x'[t] == x[t] (x[t] - 1) (x[t]^2 - 3 x[t] + 1),
       x[0] == x0}, x, {t, 0, 10}][ticks/100]
   , {x0, 0, 1, 0.01}]
  ], {ticks, 0, 1000}]
```
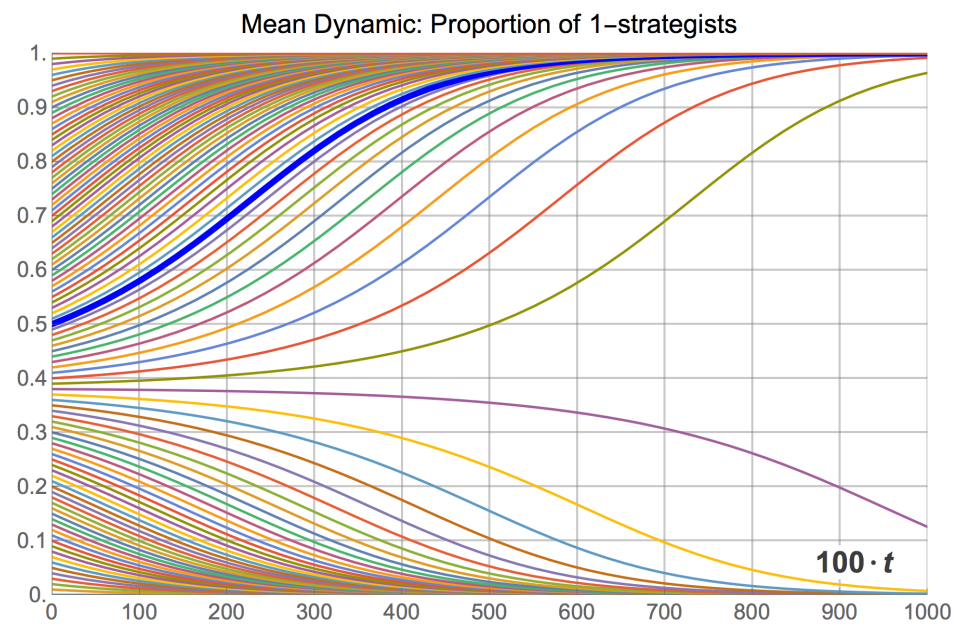


*Figure 14. Trajectories of the mean dynamic of the example in section 2, showing the proportion of 1-strategists as a function of time (rescaled to match figures 1 and 3) for different initial conditions*

The cut-off point that separates the set of trajectories that go towards state $x = 1$ from those that will end up in state $x = 0$ is easy to derive, by finding the rest points of the mean dynamic:

$$\dot{x} = 0 = x(x - 1)\left(x^2 - 3x + 1\right)$$

The three solutions in the interval $[0, 1]$ are $x = 0$, $x = 1$ and $x = \frac{1}{2}\left(3 - \sqrt{5}\right) \approx 0.382$.

In this section we have derived the mean dynamic for our 2-strategy evolutionary process where agents switch strategies sequentially. Note, however, that the mean dynamic approximation is valid for games with any number of strategies and even for models where several revisions take place simultaneously (as long as the number of revisions is fixed as $N$ goes to infinity or the probability of revision is $O(\frac{1}{N})$).

It is also important to note that, even though here we have presented the mean dynamic approximation in informal terms, the link between the stochastic process and its relevant mean

dynamic rests on solid theoretical grounds (see Benaïm & Weibull (2003), Sandholm (2010a, chapter 10) and Roth & Sandholm (2013)).

Finally, to compare agent-based simulations of the *imitate-if-better* rule and its mean dynamics in 2×2 symmetric games, you may want to play with the purpose-built demonstration titled Expected Dynamics of an Imitation Model in 2×2 Symmetric Games. And to solve the mean dynamic of the *imitate-if-better* rule in 3-strategy games, you may want to use this demonstration. In chapter V-3, we derive the mean dynamic for many other decision rules besides *imitate-if-better*, and we provide a NetLogo model that numerically solves the mean dynamic for different settings at runtime.

### 3.2.2. Diffusion approximations to characterize dynamics around equilibria

"Equilibria" in finite population dynamics are often defined as states where the expected motion of the (stochastic) process is zero. Formally, these equilibria correspond to the rest points of the mean dynamic of the original stochastic process. At some such equilibria, agents do not switch strategies anymore. Examples of such static equilibria would be the states where all agents are using the same strategy under the *imitate-if-better* rule. However, at some other equilibria, the expected flows of agents switching between different strategies cancel one another out (so the expected motion is indeed zero), but agents keep revising and changing strategies, potentially in a stochastic fashion. To characterize the dynamics around this second type of "equilibria", which are most often interior, the diffusion approximation is particularly useful.

As an example, consider a Hawk-Dove game with payoffs [[2 1][3 0]] and the *imitate-if-better* decision rule without noise. The mean dynamic of this model is:

$$\dot{x} = x(1-x)(1-2x)$$

where $x$ stands for the fraction of 1-strategists, i.e. "Hawk" agents.[12] Solving the mean dynamic reveals that most large-population simulations starting with at least one "Hawk" and at least one "Dove" will tend to approach the state where half the population play "Hawk" and the other play "Dove", and stay around there for long. Figure 15 below shows several trajectories for different initial conditions.

---

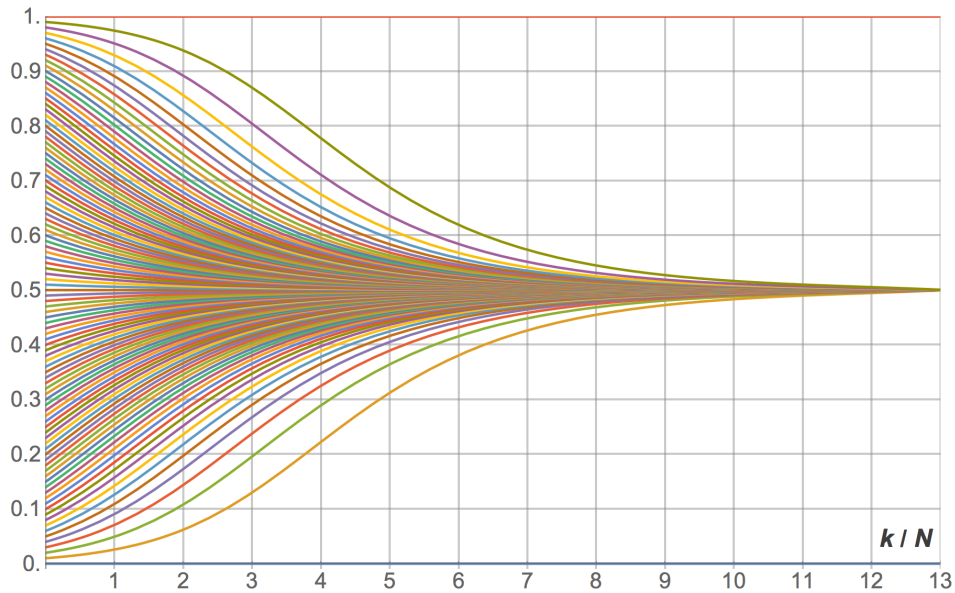12. For details, see Izquierdo and Izquierdo (2013) and Loginov (2021).

*Figure 15. Trajectories of the mean dynamic of an imitate–if–better Hawk–Dove game, showing the proportion of "Hawks" as a function of time for different initial conditions. One unit of time corresponds to $N$ revisions*

Naturally, simulations do not get stuck in the half-and-half state, since agents keep revising their strategy in a stochastic fashion (see figure 16). To understand this stochastic flow of agents between strategies near equilibria, it is necessary to go beyond the mean dynamic. Sandholm (2003) shows that –under rather general conditions– stochastic finite-population dynamics near rest points can be approximated by a diffusion process, as long as the population size $N$ is large enough. He also shows that the standard deviations of the limit distribution are of order $\frac{1}{\sqrt{N}}$.

To illustrate this order $\frac{1}{\sqrt{N}}$, we set up one simulation run starting with 10 agents playing "Hawk" and 10 agents playing "Dove". This state constitutes a so-called "Equilibrium", since the expected change in the strategy distribution is zero. However, the stochasticity in the decision rule and in the matching process imply that the strategy distribution is in perpetual change. In the simulation shown in figure 16, we modify the number of players at runtime. At tick 10000, we increase the number of players by a factor of 10 up to 200 and, after 10000 more ticks, we set *n-of-players* to 2000 (i.e., a factor of 10, again). The standard deviation of the fraction of players using strategy "Hawk" (or "Dove") during each of the three stages in our simulation run was: 0.1082, 0.0444 and 0.01167 respectively. As expected, these numbers are related by a factor of approximately $\frac{1}{\sqrt{10}}$.
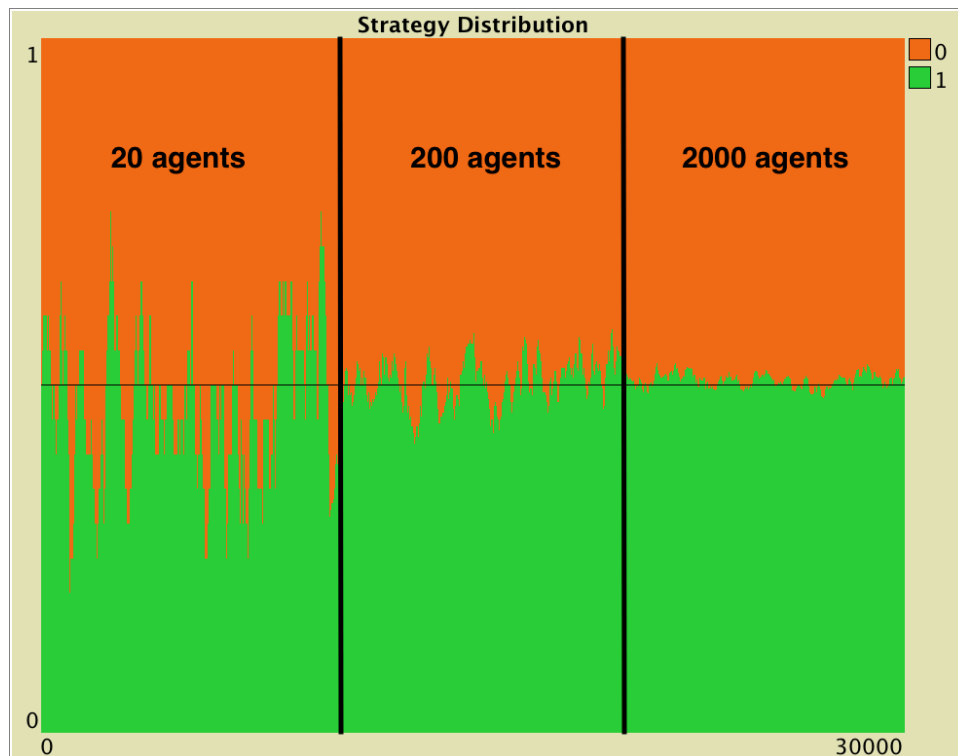
*Figure 16. A simulation run of an imitate-if-better Hawk–Dove game, set up with 20 agents during the first 10000 ticks, then 200 agents during the following 10000 ticks, and finally 2000 agents during the last 10000 ticks. Payoffs: [[2 1][3 0]]; prob-revision: 0.01; noise 0; initial conditions [10 10]*

As a matter of fact, Izquierdo et al. (2019, example 3.1) use the diffusion approximation to show that in the large $N$ limit, fluctuations of this process around its unique interior rest point $x = \frac{1}{2}$ are approximately Gaussian with standard deviation $\frac{1}{2\sqrt{N}}$.

### 3.2.3. Stochastic stability analyses

In the last model we have implemented, if *noise* is strictly positive, the model's infinite-horizon behavior is characterized by a unique stationary distribution regardless of initial conditions (see section 3.1 above). This distribution has full support (i.e. all states will be visited infinitely often) but, naturally, the system will spend much longer in certain areas of the state space than in others. If the noise is sufficiently small (but strictly positive), the infinite-horizon distribution of the Markov chain tends to concentrate most of its mass on just a few states. Stochastic stability analyses are devoted to identifying such states, which are often called *stochastically stable states* (Foster and Young, 1990), and are a subset of the absorbing states of the process without noise.[13]

To learn about this type of analysis, the following references are particularly useful: Vega-Redondo (2003, section 12.6), Fudenberg and Imhof (2008), Sandholm (2010a, chapters 11 and 12) and Wallace and Young (2015).

To illustrate the applicability of stochastic stability analyses, consider our *imitate-if-better* model

---

13. There are a number of different definitions of stochastic stability, depending on which limits are taken and in what order. For a discussion of different definitions, see Sandholm (2010a, chapter 12).

where agents play the Hawk-Dove game analyzed in section 3.2.2 with some strictly positive *noise*. It can be proved that the only stochastically stable state in this model is the state where everyone chooses strategy Hawk.[14] This means that, given a certain population size, as *noise* tends to 0, the infinite-horizon dynamics of the model will concentrate on that single state.

An important concern in stochastic stability analyses is the time one has to wait until the prediction of the analysis becomes relevant. This time can be astronomically long, as the following example illustrates.

## 3.2.4 A final example

> A fundamental feature of these models, but all too often ignored in applications, is that the asymptotic behavior of the short-run deterministic approximation need have no connection to the asymptotic behavior of the stochastic population process. Blume (1997, p. 443)

Consider the Hawk-Dove game analyzed in section 3.2.2, played by $N = 30$ *imitate-if-better* agents with noise = 10^-10, departing from an initial state where 28 agents are playing Hawk. Even though the population size is too modest for the mean dynamic and the diffusion approximations to be accurate, this example will clarify the different time scales at which each of the approximations is useful.

Let us review what we can say about this model using the three approximations discussed in the previous sections:

- **Mean dynamic**. Figure 15 shows the mean dynamic of this model without noise. The noise we are considering here is so small that the mean dynamic looks the same in the time interval shown in figure 15.[15] So, in our model with small noise, for large $N$, the process will tend to move towards state $x = 0.5$, a journey that will take about $13N$ revisions for our initial conditions $X_0 = \frac{28}{30} \approx 0.93$. The greater the $N$, the closer the stochastic process will be to the solution trajectory of its mean dynamic.

- **Diffusion approximation**. Once in the vicinity of the unique interior rest point $x = 0.5$, the diffusion approximation tells us that –for large $N$– the dynamics are well approximated by a Gaussian distribution with standard deviation $\frac{1}{2\sqrt{N}}$.

- **Stochastic stability**. Finally, we also know that, for a level of noise low enough (but strictly positive), the limiting distribution is going to place most of its mass on the unique stochastically stable state, which is $x = 1$. So, *eventually*, the dynamics will approach its limiting distribution,

---

14. To be precise, here we are considering stochastic stability *in the small noise limit*, where we fix the population size and take the limit of *noise* to zero (Sandholm, 2010a, section 12.1.1). The proof can be conducted using the concepts and theorems put forward by Ellison (2000). Note that the radius of the state where everyone plays Hawk is 2 (i.e. 2 mutations are needed to leave its basin of attraction), while its coradius is just 1 (one mutation is enough to go from the state where everyone plays Dove to the state where everyone plays Hawk).

15. The only difference is that, in the model with noise, the two trajectories starting at the monomorphic states eventually converge to the state $x = 0.5$, but this convergence cannot be appreciated in the time interval shown in figure 15.

which –assuming the noise is low enough– places most of its mass on the monomorphic state $x = 1$.[16]

Each of these approximations refers to a different time scale. In this regard, we find the classification made by Binmore and Samuelson (1994) and Binmore et al. (1995) very useful (see also Samuelson (1997) and Young (1998)). These authors distinguish between the short run, the medium run, the long run and the ultralong run:

> By the short run, we refer to the initial conditions that prevail when one begins one's observation or analysis. By the ultralong run, we mean a period of time long enough for the asymptotic distribution to be a good description of the behavior of the system. The long run refers to the time span needed for the system to reach the vicinity of the first equilibrium in whose neighborhood it will linger for some time. We speak of the medium run as the time intermediate between the short run *[i.e. initial conditions]* and the long run, during which the adjustment to equilibrium is occurring. Binmore et al. (1995, p. 10)

Let us see these different time scales in our Hawk-Dove example. The following video shows the exact transient dynamics of this model, computed as explained in section 3.1.2. Note that the video shows all the revisions up until $k = 400$, but then it moves faster and faster. The blue progress bar indicates the number of revisions already shown.

*One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=105#video-105-1*

*Transient dynamics of a model where $N = 30$ imitate-if-better agents are playing a Hawk-Dove game, with noise=$10^{-10}$. Each iteration corresponds to one revision*

In the video we can distinguish the different time scales:

- The short run, which is determined by the initial conditions $X_0 = \frac{28}{30} \approx 0.93$.
- The medium run, which in this case spans roughly from $k = 0$ to $k \approx 13N = 390$. The dynamics of this adjustment process towards the equilibrium $x = 0.5$ can be characterized by the mean dynamic, especially for large $N$.
- The long run, which in this case refers to the dynamics around the equilibrium $x = 0.5$, spanning roughly from $k \approx 13N = 390$ to $k \approx 10^7$. These dynamics are well described by the diffusion approximation, especially for large $N$.

---

16. Using the analytic formula for the limiting distribution of irreducible and aperiodic birth-death chains provided in section 3.1.3, we have checked that $\mu(1) > 0.9$ for $N = 30$ and noise = $10^{-10}$.

- The ultra long run, which in this case is not really reached until $k \gtrsim 10^{10}$. It is not until then that the limiting distribution becomes a good description of the dynamics of the model.

It is remarkable how long it takes for the infinite horizon prediction to hold force. Furthermore, the wait grows sharply as $N$ increases and also as the level of noise decreases.[17] These long waiting times are typical of stochastic stability analyses, so care must be taken when applying the conclusions of these analyses to real world settings.

In summary, as $N$ grows, both the mean dynamic and the difussion approximations become better. For any fixed $N$, eventually, the behavior of the process will be well described by its limiting distribution. If the noise is low enough (but strictly positive), the limiting distribution will place most of its mass on the unique stochastically stable state $x = 1$. But note that, as $N$ grows, it will take exponentially longer for the infinite-horizon prediction to kick in (see Sandholm and Staudigl (2018)).

Note also that for the limiting distribution to place most of its mass on the stochastically stable state, the level of noise has to be sufficiently low, and if the population size $N$ increases, the maximum level of noise at which the limiting distribution concentrates most of its mass on the stochastically stable state decreases. As an example, consider the same setting as the one shown in the video, but with $N = 50$. In this case, the limiting distribution is completely different (see figure 17). A noise level of $10^{-10}$ is not enough for the limiting distribution to place most of its mass on the stochastically stable state when $N = 50$.
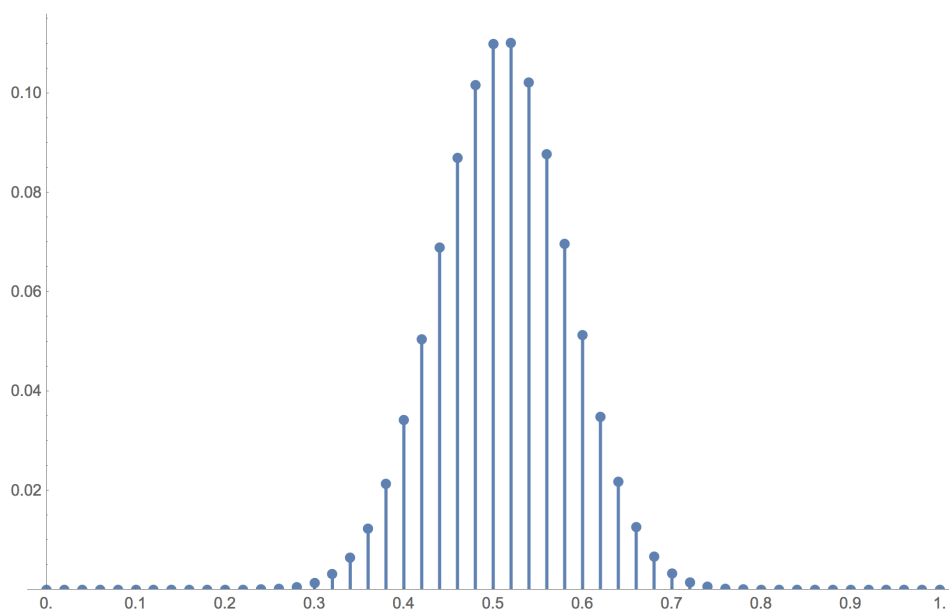


Figure 17. Limiting distribution of a model where $N = 50$ imitate-if-better agents are playing a Hawk–Dove game, with noise = $10^{-10}$

Figure 17 has been created by running the following *Mathematica*® script:

---

17. Using tools from large deviations theory, Sandholm and Staudigl (2018) show that –for large population sizes $N$– the time required to reach the boundary is of an exponential order in $N$.

```
n = 50;
noise = 10^-10;

p[x_, noise_] := (1-x)((1-noise)((x n)/(n-1))((1-x)n/(n-1)) + noise/2)
q[x_, noise_] := x((1-noise)(((1-x)n)/(n-1))(x n - 1)/(n-1) + noise/2)

μ = Normalize[
      FoldList[Times, 1,
        Table[p[(j-1)/n, noise] / q[j/n, noise],{j, n}]]
    , Total];

ListPlot[μ, DataRange->{0, 1}, PlotRange->{0, All}, Filling -> Axis]
```

To conclude, Figure 18 and Figure 19 below show the transient distributions of this model with $N = 30$ and $N = 50$ respectively, for different levels of noise μ, and at different ticks $k$. The distributions at the far right, for $k = 10^{11}$, are effectively equal to the asymptotic ones.
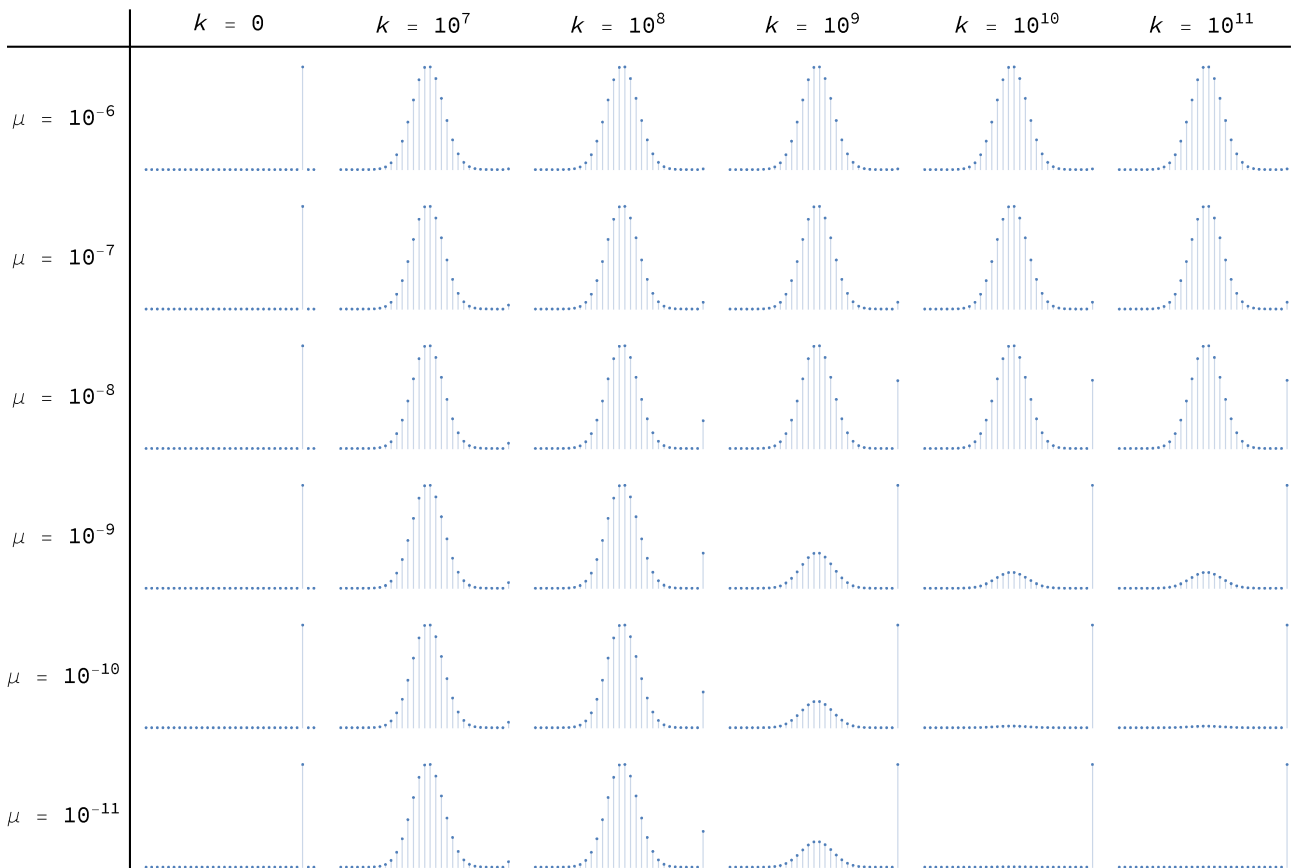


*Figure 18. Distributions of a model where $N = 30$ imitate-if-better agents are playing a Hawk-Dove game, with different levels of noise μ, and at different ticks $k$*
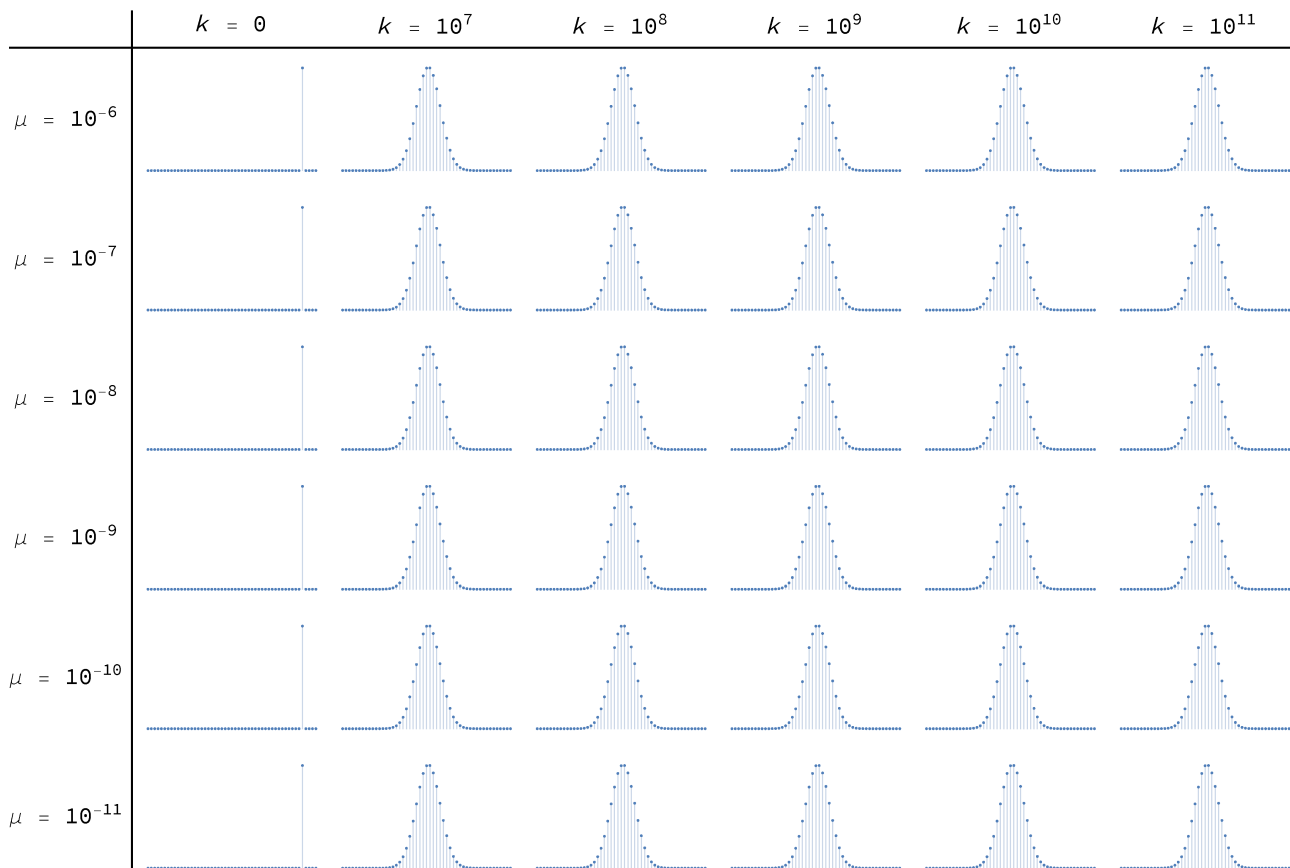
*Figure 19. Distributions of a model where $N = 50$ imitate-if-better agents are playing a Hawk-Dove game, with different levels of noise μ, and at different ticks $k$*

Figure 18 and Figure 19 have been created by running the following *Mathematica*® script:

```
n = 30;

noiseLevels = Table[10^(-i), {i, 6, 11}];
ticks = Join[{0}, Table[10^i, {i, 7, 11}]];

initialDistribution = UnitVector[n + 1, Round[0.93*n] + 1];

p[x_, noise_] := (1-x)((1-noise)((x n)/(n-1))((1-x)n/(n-1)) + noise/2)
q[x_, noise_] := x((1-noise)(((1-x)n)/(n-1))(x n - 1)/(n-1) + noise/2)

P[noise_] := SparseArray[{
    {i_, i_} -> (1 - p[(i - 1)/n, noise] - q[(i - 1)/n, noise]),
    {i_, j_} /; i == j - 1 -> p[(i - 1)/n, noise],
    {i_, j_} /; i == j + 1 -> q[(i - 1)/n, noise]},
    {n + 1, n + 1}];

FormatNice[i_]:=
  If[i == 0., 0, ScientificForm[N@i, NumberFormat -> (10^#3 &)]];

TableForm[
 Table[
  ListPlot[
    initialDistribution . MatrixPower[N@P[noise], finalTimestep],
```

```
   Filling -> Axis, PlotRange -> All, ImageSize -> Tiny, Axes -> False],
  {noise, noiseLevels}, {finalTimestep, ticks}
  ],
TableHeadings -> {
  Map[Row[{"μ = ", FormatNice[#]}] &, noiseLevels],
  Map[Row[{"k = ", FormatNice[#]}] &, ticks]},
TableAlignments -> Center, TableSpacing -> {3, 1}]
```

# 4. Exercises

**Exercise 1.** Consider the evolutionary process analyzed in section 3.1.2. Figure 5 shows that, if we start with half the population using each strategy, the probability that the whole population will be using strategy 1 after 500 revisions is about 6.66%. Here we ask you to use the NetLogo model implemented in the previous chapter to estimate that probability. To do that, you will have to set up and run an experiment using BehaviorSpace.

**Exercise 2.** Derive the mean dynamic of a Prisoner's Dilemma game for the imitate if better rule.

**Exercise 3.** Derive the mean dynamic of the coordination game discussed in chapter I-2 (with payoffs [[1 0][0 2]]) for the imitative pairwise-difference rule.

**Exercise 4.** Derive the mean dynamic of the coordination game discussed in chapter I-2 (with payoffs [[1 0][0 2]]) for the best experienced payoff rule.

**Exercise 5.** For the best experienced payoff rule, derive the mean dynamic of the 2-player $n$-strategy (single-optimum) coordination game with the following payoff matrix:

$$\begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 2 & 0 & \ldots & 0 \\ 0 & 0 & \ddots & & \vdots \\ \vdots & \vdots & & n-1 & 0 \\ 0 & 0 & \ldots & 0 & n \end{pmatrix}$$

**Exercise 6.** Once you have done exercise 5 above, prove that, in the derived mean dynamics, the state where every player chooses the efficient strategy (which provides a payoff of $n$) attracts all trajectories except possibly those starting at the other monomorphic states in which all players use the same strategy.

# PART III. SPATIAL INTERACTIONS ON A GRID

<div style="text-align: center; border: 1px solid black; background-color: #d3d3d3; padding: 10px;">

# III-1. Spatial chaos in the Prisoner's Dilemma

</div>

## 1. Goal

The goal of this chapter is to learn how to build agent-based models with *spatial structure*. In models with spatial structure, agents do not interact with all other agents with the same probability, but they interact preferentially with those who are nearby.[1]

More generally, populations where some pairs of agents are more likely to interact with each other than with others are called *structured* populations. This contrasts with the models developed in the previous Part, where all members of the population were equally likely to interact with each other.[2] The dynamics of an evolutionary process under random matching can be very different from the dynamics of the same process in a structured population. In social dilemmas in particular, population structure can play a crucial role (Gotts et al. (2003), Hauert (2002,[3] 2006), Roca et al. (2009a, 2009b)).[4]

## 2. Motivation. Cooperation in spatial settings

In the previous Part, we saw that if agents play the Prisoner's Dilemma in a population where all members are equally likely to interact with each other, then defection prevails. Here we want to explore whether adding spatial structure may affect that observation. Could cooperation be sustained if we removed the unrealistic assumption that all members of the population are equally likely to interact with each other? To shed some light on this question, in this chapter we will implement a model analyzed by Nowak and May (1992, 1993).

## 3. Description of the model

In this model, there is a population of agents arranged on a 2-dimensional lattice of "patches". There is one agent in each patch. The size of the lattice, i.e. the number of patches in each of the two

---

1. Note that in most evolutionary models there are two types of neighborhoods for each individual agent A:

   - the set of agents with whom agent A plays the game, and
   - the set of agents that agent A may observe at the time of revising his strategy.

   Most often these two sets coincide for each individual agent, but that is not necessarily the case (see e.g. Ohtsuki et al. (2007a, 2007b)).

2. Populations where all members are equally likely to interact with each other are sometimes called *well-mixed populations*.

3. See Roca et al. (2009b) for an important and illuminating discussion of this paper.

4. Christoph Hauert has an excellent collection of interactive tutorials on this topic at his site EvoLudo (Hauert 2018).

dimensions, can be set by the user. Each patch has eight neighboring patches (i.e. the eight cells which surround it), except for the patches at the boundary, which have five neighbors if they are on a side, or three neighbors if they are at one of the four corners.

Agents repeatedly play a symmetric 2-player 2-strategy game, where the two possible strategies are labeled C (for Cooperate) and D (for Defect). The payoffs of the game are determined using four parameters: *CC-payoff*, *CD-payoff*, *DC-payoff*, and *DD-payoff*, where *XY-payoff* denotes the payoff obtained by an X-player who meets a Y-player.

The initial percentage of C-players in the population is *initial-%-of-C-players*, and they are randomly distributed in the grid. From then onwards, the following sequence of events –which defines a tick– is repeatedly executed:

1.  Every agent plays the game with all his neighbors (once with each neighbor) and with himself (Moore neighborhood). The total payoff for the player is the sum of the payoffs in these encounters.

2.  All agents *simultaneously* revise their strategy according to the **imitate the best neighbor** decision rule, which reads as follows:

    > Consider the set of all your neighbors plus yourself; then adopt the strategy of one of the agents in this set who has obtained the greatest payoff. If there is more than one agent with the greatest payoff, choose one of them at random to imitate.
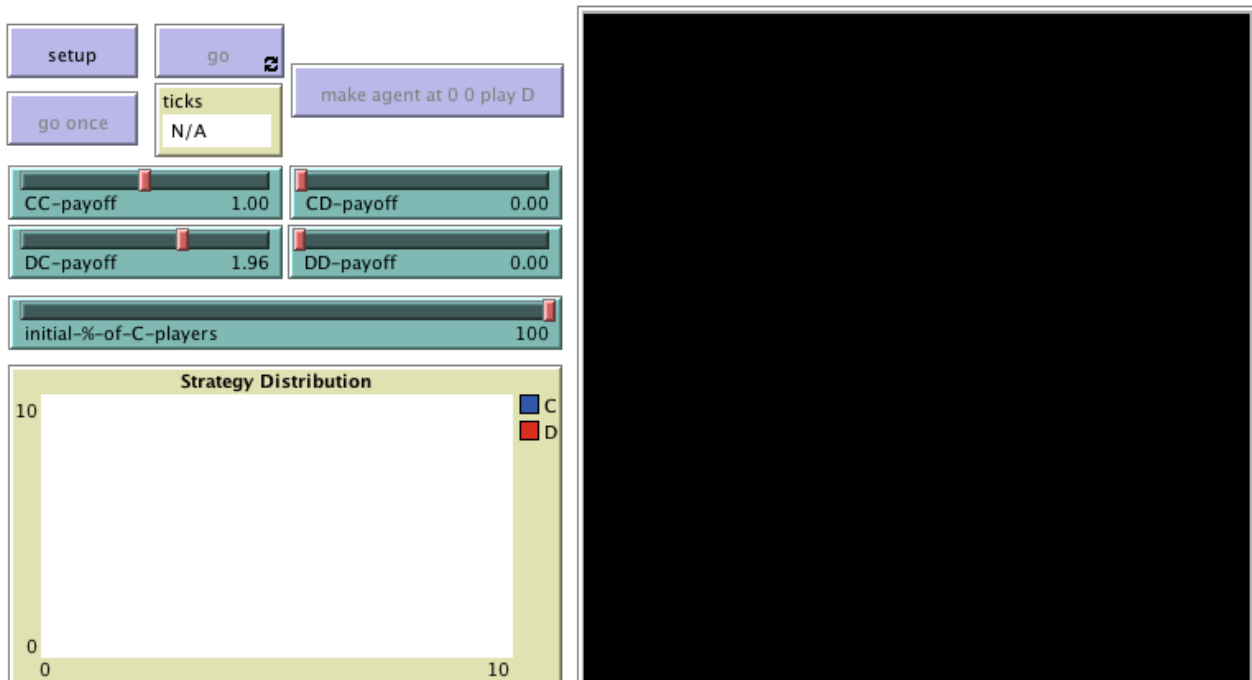
# 4. Interface design



*Figure 1. Interface design*

To define each agent's neighborhood, in this chapter we will use the 2-dimensional grid already built in NetLogo, often called "the world". This will make our code simpler and the visualizations nicer.

The interface (see figure 1 above) includes:

- The 2D view of the NetLogo world (i.e. the large black square in the interface), which is made up of patches. This view is already on the interface by default when creating a new NetLogo model.

> Choose the dimensions of the world by clicking on the "Settings…" button on the top bar, or by right-clicking on the 2D view and choosing *Edit*. A window will pop up, which allows you to choose the number of patches by setting the values of `min-pxcor`, `max-pxcor`, `min-pycor` and `max-pycor`. You can also determine the patches' size in pixels, and whether the grid wraps horizontally, vertically, both or none (see Topology section). You can choose these parameters as in figure 2 below:
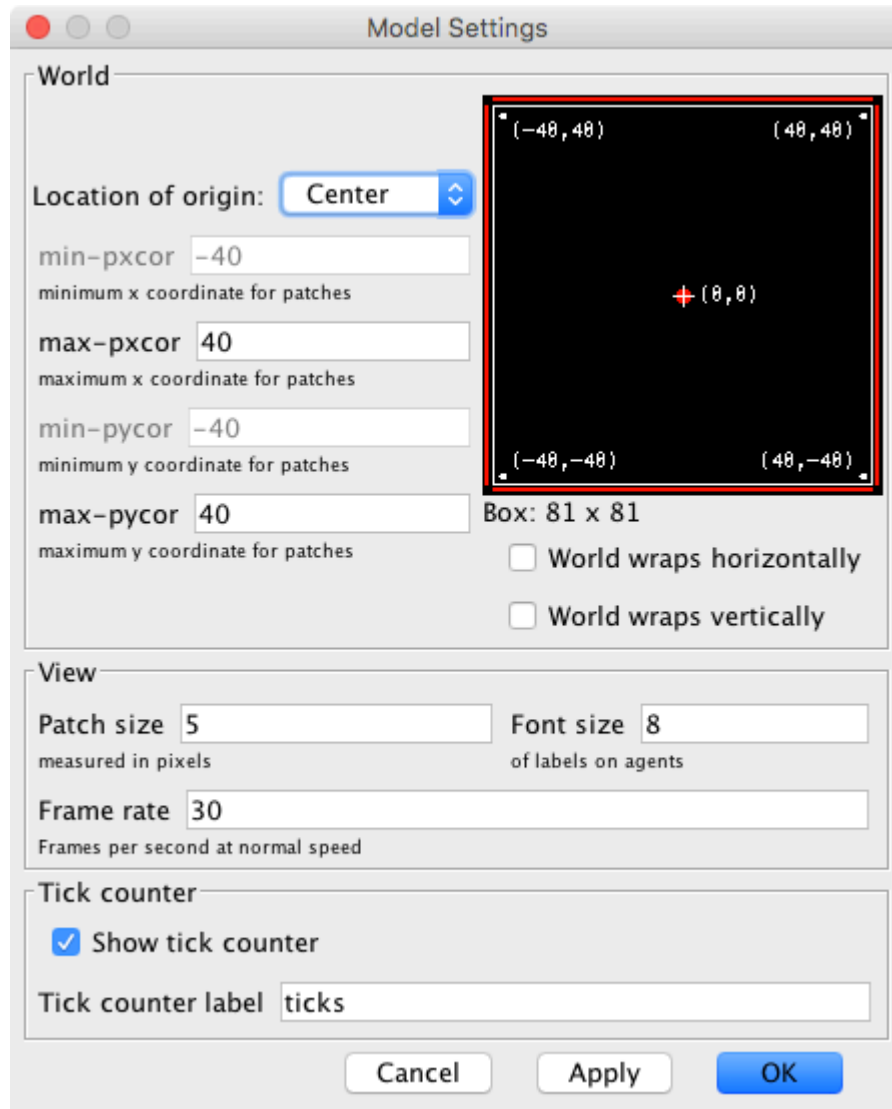
*Figure 2. Model settings*

- Three buttons:

    1. One button named setup, which runs the procedure to setup.
    2. One button named go once, which runs the procedure to go.
    3. One button named go, which runs the procedure to go indefinitely.

In the Code tab, write the procedures to setup and to go, without including any code inside for now. Then, create the buttons, just like we did in previous chapters.

Note that the interface in figure 1 has an extra button labeled make agent at 0 0 play

D. You may wish to include it now. The code that goes inside this button is proposed as Exercise 2.

- Four sliders, to choose the payoffs for each possible outcome (CC, CD, DC, DD).

Create the four sliders with global variable names *CC-payoff*, *CD-payoff*, *DC-payoff*, and *DD-payoff*. Remember to choose a range, an interval and a default value for each of them. You can choose minimum 0, maximum 2 and increment 0.01.

- A slider to let the user select the initial percentage of C-players.

Create a slider for global variable *initial-%-of-C-players*. You can choose limit values 0 (as the minimum) and 100 (as the maximum), and an increment of 1.

- A plot that will show the evolution of the number of agents playing each strategy.

Create a plot and name it Strategy Distribution.

# 5. Code

## 5.1. Skeleton of the code

Figure 3 below provides a schematic view of the code. You can find the legend for code skeletons in Appendix A-2.
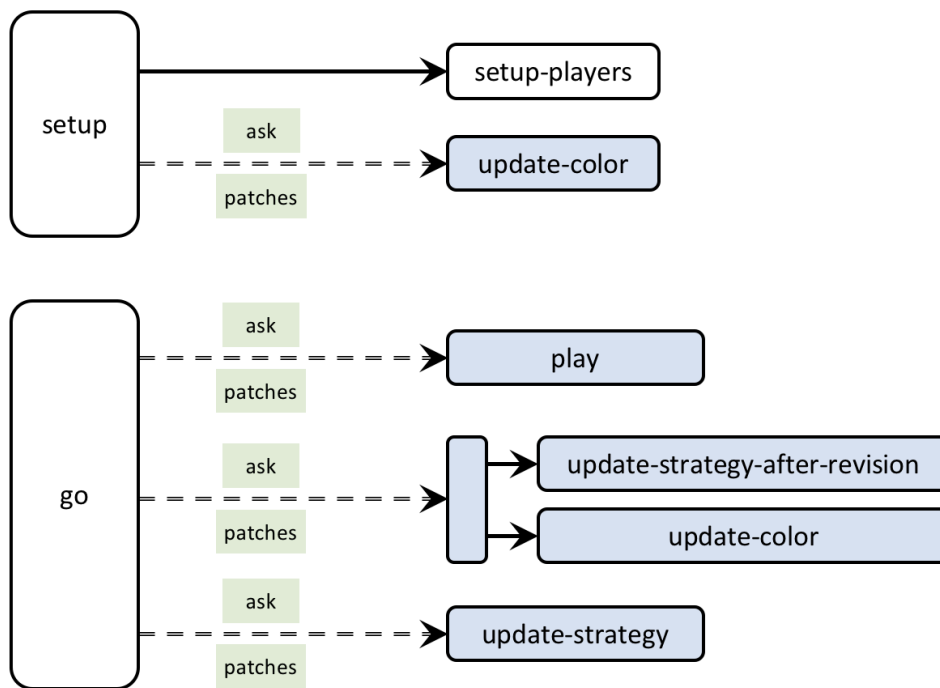
*Figure 3. Skeleton of the code*

## 5.2. Global variables and individually-owned variables

We will not need any global variables besides those defined with the sliders in the interface.

Note that in this model there is a one-to-one correspondence between our immobile players and the patches they live in. Thus, there is no need to create any turtles (i.e. NetLogo mobile agents) in our model. We can work only with patches, and our code will be much simpler and readable.

Thus, we can make the built-in "patches" be the players, identifying each patch with one player. These patches already exist in NetLogo, making up the world, so we do not need to create them. Having said that, we do need to associate with each patch all the information that we want it to carry. This information will be:

- Whether the patch is a C-player or D-player. For efficiency and code readability we can use a boolean variable to this end, which we can call C-player? and which will take the value true or false.

- Whether the patch will be a C-player or a D-player after its revision. For this purpose, we may use the boolean variable C-player?-after-revision. This is needed because we want to model *synchronous* updating, i.e. we want all patches to change their strategy at the same time. To do this, first we will ask all patches to compute the strategy they will adopt after the revision and, once all patches have computed their next strategy, we will ask them all to switch to it at the same time.

- The total payoff obtained by the patch playing with its neighbours. We can call this variable payoff.

- For efficiency, it will also be useful to endow each patch with the set of neighbouring patches plus itself. The reason is that this set will be used many times, and it never changes, so it can be

computed just once at the beginning and stored in memory. We will store this set in a variable named my-nbrs-and-me.

- The following variable is also defined for efficiency reasons. Note that the payoff of a patch depends on the number of C-players and D-players in its set my-nbrs-and-me. To spare the operation of counting D-players, we can calculate it as the number of players in my-nbrs-and-me (which does not change in the whole simulation) minus the number of C-players. To this end, we can store the number of players in the set my-nbrs-and-me of each patch as an individually-owned variable that we naturally name n-of-my-nbrs-and-me.

Thus, this part of the code looks as follows:

```
patches-own [
  C-player?
  C-player?-after-revision
  payoff
  my-nbrs-and-me
  n-of-my-nbrs-and-me
]
```

## 5.3. Setup procedures

In the setup procedure we will:

1. Clear everything up, so we initialize the model afresh, using the primitive `clear-all`:

   ```
   clear-all
   ```

2. Set initial values for the variables that we have associated to each patch. We can set the payoff to 0,[5] and both C-player? and C-player-last? to false (later we will ask some patches to set these values to true). To set the value of my-nbrs-and-me, NetLogo primitives `neighbors` and `patch-set` are really handy.

   ```
   ask patches [
     set payoff 0
     set C-player? false
     set C-player?-after-revision false
     set my-nbrs-and-me (patch-set neighbors self)
     set n-of-my-nbrs-and-me (count my-nbrs-and-me)
   ]
   ```

3. Ask a certain number of randomly selected patches to be C-players. That number depends on the percentage *initial-%-of-C-players* chosen by the user and on the total number of patches,

---

5. By default, user-defined variables in NetLogo are initialized with the value 0, so there is no actual need to explicitly set the initial value of individually-owned variables to 0, but it does no harm either.

and it must be an integer, so we can calculate it as:

```
round (initial-%-of-C-players * count patches / 100)
```

To randomly select a certain number of agents from an agentset (such as patches), we can use the primitive `n-of` (which reports another –usually smaller– agentset). Thus, the resulting instruction will be:

```
ask n-of (round (initial-%-of-C-players * count patches / 100)) patches [
  set C-player? true
  set C-player?-after-revision true
]
```

4. Color patches according to the four possible combinations of values of C-player? and C-player?-after-revision. The color of a patch is controled by the NetLogo built-in patch variable `pcolor`. A first (and correct) implementation of this task could look like:

```
ask patches [
  ifelse C-player?-after-revision
  [
    ifelse C-player?
    [set pcolor blue]
    [set pcolor lime]
  ]
  [
    ifelse C-player?
    [set pcolor yellow]
    [set pcolor red]
  ]
]
```

However, the following implementation, which makes use of NetLogo primitive `ifelse-value` is more readable, as one can clearly see that the only thing we are doing is to set the patch's `pcolor`.

```
ask patches [
  set pcolor
    ifelse-value C-player?-after-revision
    [ifelse-value C-player? [blue] [lime]]
    [ifelse-value C-player? [yellow] [red]]
]
```

5. Reset the tick counter using `reset-ticks.`

Note that:

- Points 2 and 3 above are about setting up the players, so, to keep our code nice and modular, we could group them into a new procedure called to setup-players. This will make our code

more elegant, easier to understand, easier to debug and easier to extend, so let us do it!

- The operation described in point 4 above will be conducted every tick, so we should create a separate procedure to this end that we can call to update-color, to be run by individual patches. Since this procedure is rather secondary (i.e. our model could run without this), we have a slight preference to place it at the end of our code, but feel free to do it as you like, since the order in which NetLogo procedures are written in the Code tab is immaterial.

Thus, the code up to this point should be as follows:

```
patches-own [
  C-player?
  C-player?-after-revision
  payoff
  my-nbrs-and-me
  n-of-my-nbrs-and-me
]

to setup
  clear-all
  setup-players
  ask patches [update-color]
  reset-ticks
end

to setup-players
  ask patches [
    set payoff 0
    set C-player? false
    set C-player?-after-revision false
    set my-nbrs-and-me (patch-set neighbors self)
    set n-of-my-nbrs-and-me (count my-nbrs-and-me)
  ]
  ask n-of (round (initial-%-of-C-players * count patches / 100)) patches [
    set C-player? true
    set C-player?-after-revision true
  ]
end

to go

end

to update-color
  set pcolor
    ifelse-value C-player?-after-revision
      [ifelse-value C-player? [blue] [lime]]
      [ifelse-value C-player? [yellow] [red]]
end
```

## 5.4. Go procedure

The procedure to go contains all the instructions that will be executed in every tick. In this particular model, we will ask each player (i.e. patch):

1. To play with its neighbours in order to calculate its payoff. For modularity and clarity purposes, we should do this in a new procedure named to play.

2. To compute the value of its next strategy and store it in the variable C-player?-after-revision. In this way, the variable C-player? will keep the strategy with which the current payoff has been obtained, and we can update the value of C-player?-after-revision without losing that information, which will be required by neighboring players when they compute their next strategy. To keep our code nice and modular, we will do this computation in a new procedure called to update-strategy-after-revision.

3. To update its color according to their C-player? and C-player?-after-revision values, using the procedure to update-color.

4. To update its strategy (i.e. the value of C-player?). We will do this in a separate new procedure called to update-strategy.

We should also mark the end of the round, or tick, after all players have updated their strategies, using the primitive `tick`, which increases the tick counter by one, and updates the graph on the interface. Thus, by now the code of procedure to go should look as follows:

```
to go
  ask patches [ play ]
  ask patches [
    update-strategy-after-revision
      ;; here we are not updating the agent's strategy yet
    update-color
  ]
  ask patches [ update-strategy ]
    ;; now we update every agent's strategy at the same time
  tick
end
```

## 5.5 Other procedures

### to play

In procedure to play we want patches to calculate their payoff. This payoff will be the number of C-players in the set my-nbrs-and-me times the payoff obtained with a C-player, plus the number of D-players in the set times the payoff obtained with a D-player.

We will store the number of C-players in the set my-nbrs-and-me in a local variable that we can name n-of-C-players. The number can be computed as follows:

```
let n-of-C-players count my-nbrs-and-me with [C-player?]
```

Note that if the calculating patch is a C-player, the payoff obtained when playing with another C-

player is *CC-payoff*, and if the calculating patch is a D-player, the payoff obtained when playing with a C-player is *DC-payoff*. Thus, in general, the payoff obtained when playing with a C-player can then be obtained using the following code:

```
ifelse-value C-player? [CC-payoff] [DC-payoff]
```

Similarly, the payoff obtained when playing with a D-player is:

```
ifelse-value C-player? [CD-payoff] [DD-payoff]
```

Taking all this into account, we can implement procedure to play as follows:[6]

```
to play
  let n-of-C-players count my-nbrs-and-me with [C-player?]
  set payoff n-of-C-players *
                (ifelse-value C-player? [CC-payoff] [DC-payoff]) +
            (n-of-my-nbrs-and-me - n-of-C-players) *
                (ifelse-value C-player? [CD-payoff] [DD-payoff])
end
```

## to update-strategy-after-revision

In this procedure, which will be run by individual patches, we want the patch to compute its next strategy, which will be the strategy used by one of the patches with the maximum payoff in the set my-nbrs-and-me. To select one of these maximum-payoff patches, we may use primitives one-of and with-max as follows:

```
one-of (my-nbrs-and-me with-max [payoff])
```

Now remember that strategy updating in this model is *synchronous*, i.e. every player revises his strategy at the same time. Thus, we want each patch to adopt the strategy that was used by the selected maximum-payoff patch when it played the game, i.e. before any strategy revision may have taken place. This strategy is stored in variable C-player?. With this, we conclude the code of procedure to update-strategy-after-revision.

```
to update-strategy-after-revision
  set C-player?-after-revision
      [C-player?] of one-of my-nbrs-and-me with-max [payoff]
end
```

Another (equivalent) implementation of this procedure, which makes use of primitive max-one-of is the following.

---

6. The parentheses around the first `ifelse-value` block are necessary since NetLogo 6.1.0 (see https://ccl.northwestern.edu/netlogo/docs/transition.html#changes-for-netlogo-610).

```
  to update-strategy-after-revision
    set C-player?-after-revision
        [C-player?] of max-one-of my-nbrs-and-me [payoff]
  end
```

## to update-strategy

This is a very simple procedure where the patch just updates its strategy (stored in variable C-player?) with the value of C-player?-after-revision. This update is not conducted right after having computed the value of C-player?-after-revision to make the strategy updating *synchronous*.

```
  to update-strategy
    set C-player? C-player?-after-revision
  end
```

# 5.6. Complete code in the Code tab

The Code tab is ready!

```
patches-own [
  C-player?
  C-player?-after-revision
  payoff
  my-nbrs-and-me
  n-of-my-nbrs-and-me
]

to setup
  clear-all
  setup-players
  ask patches [update-color]
  reset-ticks
end

to setup-players
  ask patches [
    set payoff 0
    set C-player? false
    set C-player?-after-revision false
    set my-nbrs-and-me (patch-set neighbors self)
    set n-of-my-nbrs-and-me (count my-nbrs-and-me)
  ]
  ask n-of (round (initial-%-of-C-players * count patches / 100)) patches [
    set C-player? true
    set C-player?-after-revision true
  ]
end

to go
  ask patches [ play ]
```

```
  ask patches [
    update-strategy-after-revision
      ;; here we are not updating the agent's strategy yet
    update-color
  ]
  ask patches [ update-strategy ]
    ;; now we update every agent's strategy at the same time
  tick
end

to play
  let n-of-C-players count my-nbrs-and-me with [C-player?]
  set payoff n-of-C-players *
                  (ifelse-value C-player? [CC-payoff] [DC-payoff]) +
            (n-of-my-nbrs-and-me - n-of-C-players) *
                  (ifelse-value C-player? [CD-payoff] [DD-payoff])
end

to update-strategy-after-revision
  set C-player?-after-revision
      [C-player?] of one-of my-nbrs-and-me with-max [payoff]
end

to update-strategy
  set C-player? C-player?-after-revision
end

to update-color
  set pcolor
    ifelse-value C-player?-after-revision
      [ifelse-value C-player? [blue] [lime]]
      [ifelse-value C-player? [yellow] [red]]
end
```

## 5.7. Code in the plots

We will use blue color for the number of C-players and red for the number of D-players.

> To complete the Interface tab, edit the graph and create the pens as in the image below:
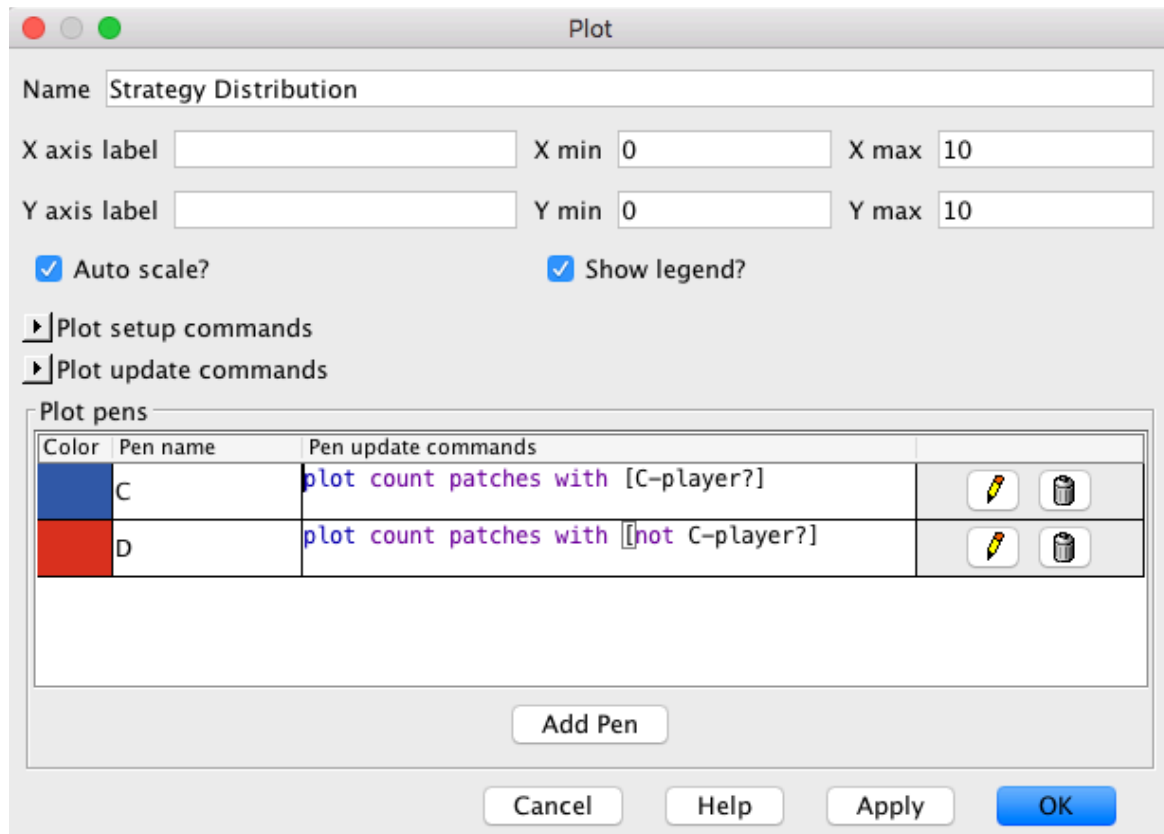
*Figure 4. Plot settings*

# 6. Sample runs

We can use the model we have implemented to shed some light on the question that we posed at the motivation above. We will use the same parameter values as Nowak and May (1992), so we can replicate their results: *CD-payoff* = *DD-payoff* = 0, *CC-payoff* = 1, *DC-payoff* = 1.85, and *initial-%-of-C-players* = 90.[7] An illustration of the sort of patterns that this model generates is shown in the video below.

> ▶️ *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=108#video-108-1*

---

7. Some authors make *CD-payoff* = *DD-payoff*, so they can parameterize the game with just one parameter, i.e. *DC-payoff*. Note, however, that the resulting game lies at the border between a Prisoner's Dilemma and a Hawk-Dove (aka Chicken or Snowdrift) game. Making *CD-payoff* = *DD-payoff* is by no means a normalization of the Prisoner's Dilemma, but a restriction which reduces the range of possibilities that can be studied.

As you can see, both C-players and D-players coexist in this spatial environment, with clusters of both types of players expanding, colliding and fragmenting. The overall fraction of C-players fluctuates around 0.318 for most initial conditions (Nowak and May, 1992). Thus, we can see that adding spatial structure can make cooperation be sustained even in a population where agents can only play C or D (i.e. they cannot condition their actions on previous moves).

Incidentally, this model is also useful to see that a simple 2-player 2-strategy game in a two-dimensional spatial setting can generate chaotic and kaleidoscopic patterns (Nowak and May, 1993). To illustrate this, let us use the same payoff values as before, but let us start with all agents playing C, i.e. *initial-%-of-C-players* = 100.

When you click on setup, the whole world should look blue, since all agents are C-players. If you now click on go, nothing should happen, since all agents are playing the same strategy and the strategy updating is imitative. To make things interesting, let us ask the agent at the center to play D. You can do this by typing the following code at the Command Center (i.e. the line at the bottom of the NetLogo screen) *after* clicking on setup:

```
ask patch 0 0 [set C-player? false]
```

If you now click on go, you should see the following beautiful patterns:

> 
> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=108#video-108-2*

# 7. Exercises

You can use the following link to download the complete NetLogo model: 2×2-imitate-best-nbr.nlogo.

Exercise 1. Let us run a (weak) Prisoner's Dilemma game with payoffs *DD-payoff* = *CD-payoff* = 0, *CC-payoff* = 1 and *DC-payoff* = 1.7. Set the *initial-%-of-cooperators* to 90. Run the model and observe the evolution of the system as you gradually increase the value of *DC-payoff* from 1.7 to 2. If at any point all the players adopt the same strategy, press the setup button again to start a new simulation. Compare your observations with those in fig. 1 of Nowak and May (1992). Note: To use the same dimensions as Nowak and May (1992), you can change the location of the NetLogo world's origin to the bottom left corner, and set both the max-pxcor and the max-pycor to 199. You may also want to change the patch size to 2.

Exercise 2. Create a button to make the patch at 0 0 be a D-player. You may want to label it make agent at 0 0 play D. This button will be useful to replicate some of the experiments in Nowak and May (1992, 1993).

**Exercise 3.** Replicate the experiment shown in figure 3 of Nowak and May (1992). Note that you will have to make the NetLogo world be a 99 × 99 square lattice.

**Exercise 4.** Implement the following extension to Nowak and May (1992)'s model, proposed by Mukherji et al. (1996):

> With a small probability ε, each player errs and chooses evenly between strategies C and D; with probability 1-ε, the player follows the Nowak and May update rule.

You may wish to rerun the sample run above with a small value for ε. You may also want to replicate the experiment shown in Mukherji et al. (1996, fig. 1).

**Exercise 5.** Implement the following extension to Nowak and May (1992)'s model, proposed by Mukherji et al. (1996):

> During each period, players fail to update their previous strategy with a small probability, θ.

You may wish to rerun the sample run above with a small value for θ. You may also want to replicate the experiment shown in Mukherji et al. (1996, fig. 1).

**Exercise 6.** Implement the following extension to Nowak and May (1992)'s model, proposed by Mukherji et al. (1996):

> After following the Nowak and May update rule, each cooperator has a small independent probability, φ, of cheating by switching to defection.

You may wish to rerun the sample run above with a small value for φ. You may also want to replicate the experiment shown in Mukherji et al. (1996, fig. 1).

# III-2. Robustness and fragility

## 1. Goal

Our goal in this chapter is to extend the model we have created in the previous chapter by adding three features that will prove very useful:

- Noise, i.e. the possibility that revising agents select a strategy at random with a small probability.

- Self-matching, i.e. the possibility to choose whether agents are matched with themselves to play the game or not.

- Asynchronous strategy updating, i.e. the possibility that agents revise their strategies *sequentially* –rather than *simultaneously*– within the same tick.[1]

These three features will allow us to assess the robustness of our previous computational results.

## 2. Motivation. Robustness of cooperation in spatial settings

In the previous chapter, we saw that spatial structure can induce significant levels of cooperation in the Prisoner's Dilemma, at least for some parameter settings. In particular, we saw that with *CD-payoff* = *DD-payoff* = 0, *CC-payoff* = 1, *DC-payoff* = 1.85, the overall fraction of C-players fluctuates around 0.318 for most initial conditions (Nowak and May, 1992). Here we wonder how robust this result is to changes in some of the model assumptions. In particular, we would like to study what happens…

- if we add a bit of noise,

- if agents do not play the game with themselves,

- if strategy updating is asynchronous, rather than synchronous, or

- if we use *DD-payoff* = 0.1 (rather than *DD-payoff* = 0), making the game a true Prisoner's Dilemma.

## 3. Description of the model

The model we are going to develop here is a generalization of the model implemented in the previous chapter. In particular, we are going to add the following three parameters:

---

1. There are different ways one can implement asynchronicity. Here we implement what Cornforth et al. (2005) call "Random Asynchronous Order". Under this scheme, at each tick all agents revise their strategy in a random order.

- *noise*. With probability *noise*, the revising agent will adopt a random strategy; and with probability (1 – *noise*), the revising agent will choose her strategy following the imitate the best neighbor rule. Thus, if *noise* = 0, we recover the model implemented in the previous chapter.

- *self-matching?*. If *self-matching?* is true, agents play the game with themselves, just like before. On the other hand, if *self-matching?* is false, agents do not play the game with themselves.

- *synchronous-updating?*. If *synchronous-updating?* is true, agents update their strategies *simultaneously*, just like before. On the other hand, if *synchronous-updating?* is false, agents play and update their strategies *sequentially*, i.e. one after another. In this latter case, all agents revise their strategies in every tick in a random order.

Everything else stays as described in the previous chapter.

# 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).
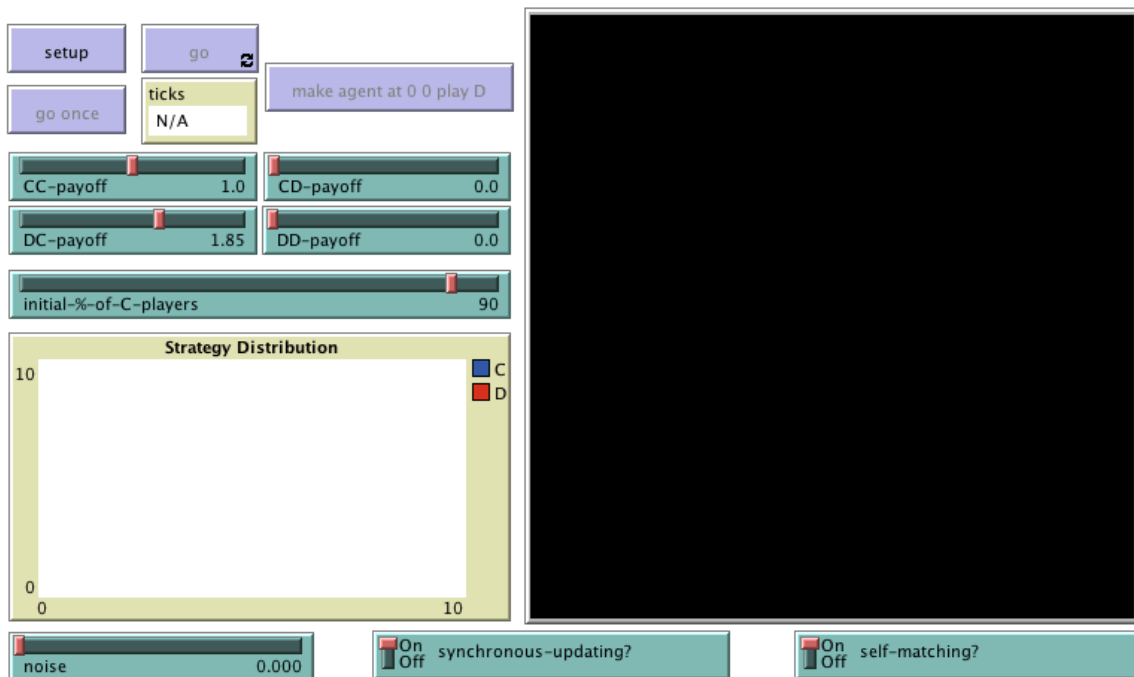


*Figure 1. Interface design*

In the new interface (see figure 1 above), we just have to add one slider for the new parameter *noise*, and two switches: one for parameter *synchronous-updating?* and another one for parameter *self-matching?*. We have added these elements at the bottom of the interface, but feel free to place them wherever you like.

# ▶ 5. Code

## 5.1. Skeleton of the code
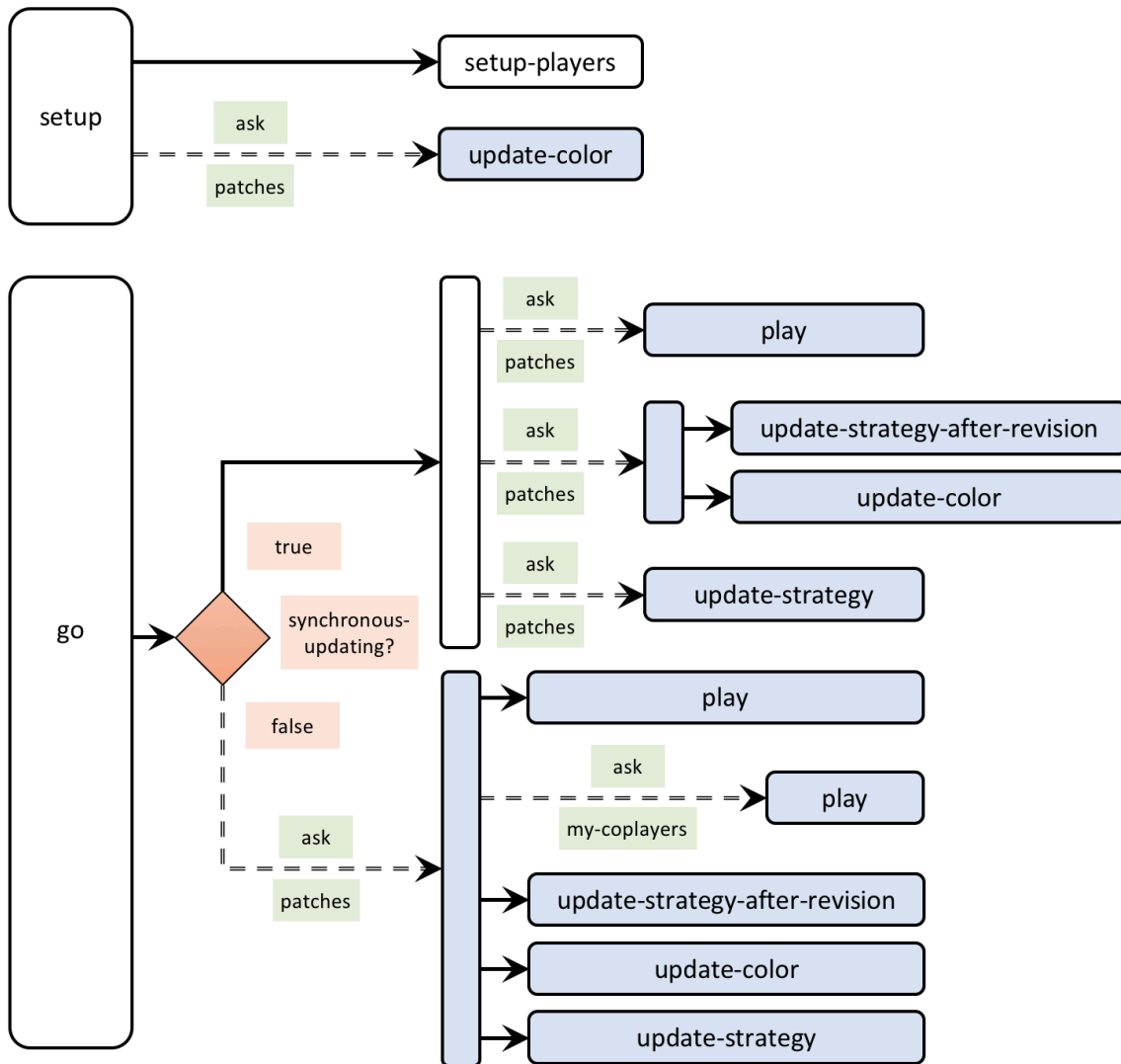


*Figure 2. Skeleton of the code*

## 5.2. Extension I. Adding noise to the decision rule

Recall that the implementation of the decision rule is conducted in procedure to update-strategy-after-revision. At present, the code of this procedure looks as follows:

```
to update-strategy-after-revision
  set C-player?-after-revision
     [C-player?] of one-of my-nbrs-and-me with-max [payoff]
end
```

To implement the choice of a random strategy with probability *noise* by revising agents, we can use NetLogo primitive `ifelse-value` as follows:[2]

```
to update-strategy-after-revision
  set C-player?-after-revision ifelse-value (random-float 1 < noise)
    [ one-of [true false] ] ;; this is run with probability noise
    [ [C-player?] of one-of (my-nbrs-and-me with-max [payoff]) ]
end
```

The noise extension is now ready, so you may want to explore the impact of noise in this model.

## 5.3. Extension II. Playing the game with yourself or not

> Whether it is natural to include self-interactions in the theory depends on the biological assumptions underlying the model. In general, if each cell is viewed as being occupied by a single individual adopting a given strategy then it is natural to exclude self-interaction. However, if each cell is viewed as being occupied by a population, all of whose members are adopting a given strategy, then it may be more natural to include self-interaction. Killingback and Doebeli (1996, p. 1136)

In our model, agents will play the game with themselves or not depending on the value of the new parameter *self-matching?*. To implement this extension elegantly, we find it convenient to define a new patch variable named my-coplayers, which will store the agentset with which the patch will play. Thus, if *self-matching?* is true, my-coplayers will include the patch's neighbors plus the patch itself, while if *self-matching?* is false, my-coplayers will include only the patch's neighbors.

It will also be convenient to define another patch variable named n-of-my-coplayers, which will store the cardinality of my-coplayers for each patch. This is just for the same (efficiency) reasons we defined n-of-my-nbrs-and-me in the previous model. Now that we have variables my-coplayers and n-of-my-coplayers, patch variable n-of-my-nbrs-and-me will no longer be needed. Thus, the definition of patch-own variables in the Code tab will look as follows:

---

2. We could also implement the noise extension using the NetLogo primitive `ifelse`, but the use of `ifelse-value` makes it clear that the only thing we are doing in this procedure is to set the value of the patch variable C-player?-after-revision.

```
patches-own [
  C-player?
  C-player?-after-revision
  payoff
  my-nbrs-and-me
  my-coplayers           ;; <== new variable
  n-of-my-coplayers      ;; <== new variable
  ;; n-of-my-nbrs-and-me    <== not needed anymore
]
```

Now we have to set the value of the two new patch-own variables. Since these values will not change during the course of the simulation and they pertain to the individual players, the natural place to set them is in procedure to setup-players.

```
to setup-players
  ask patches [
    set payoff 0
    set C-player? false
    set C-player?-after-revision false
    set my-nbrs-and-me (patch-set neighbors self)

    ;; set n-of-my-nbrs-and-me (count my-nbrs-and-me)  <== not needed anymore
    ;; the following two lines are new
    set my-coplayers ifelse-value self-matching? [my-nbrs-and-me] [neighbors]
    set n-of-my-coplayers (count my-coplayers)
  ]
  ask n-of (round (initial-%-of-C-players * count patches / 100)) patches [
    set C-player? true
    set C-player?-after-revision true
  ]
end
```

Finally, we have to modify procedure to play so patches play with agentset my-coplayers, rather than with agentset my-nbrs-and-me.

```
to play
  let n-of-C-players count my-coplayers with [C-player?]
  set payoff
      n-of-C-players * (ifelse-value C-player? [CC-payoff] [DC-payoff]) +
      (n-of-my-coplayers - n-of-C-players) *
          ifelse-value C-player? [CD-payoff] [DD-payoff]
end
```

Note also that we have to replace the variable n-of-my-nbrs-and-me with n-of-my-coplayers when computing the payoff. You can now explore the consequences of not forcing agents to play the game with themselves!

## 5.4. Extension III. Asynchronous strategy updating

To implement asynchronous updating we will have to modify procedure to go. If *synchronous-updating?* is true, updating takes place just like before, so we can wrap the code we had in to go within an `ifelse` statement whose condition is the boolean variable *synchronous-updating?* , i.e.:

```
to go
  ifelse synchronous-updating?
    [
      ask patches [ play ]
      ask patches [
        update-strategy-after-revision
          ;; here we are not updating the agent's strategy yet
        update-color
      ]
      ask patches [ update-strategy ]
        ;; now we update every agent's strategy at the same time
    ]
    [
      ;; this is where we have to place the code
      ;; for asynchronous strategy updating
    ]
  tick
end
```

The implementation of sequential updating requires that every patch (in a random order) goes through the whole cycle of playing and updating its strategy without being interrupted. Note that, at the time of revising the strategy, agents will compare their payoff with their coplayers' payoffs, so before calling procedure update-strategy-after-revision we have to make sure that all these payoffs have been properly computed, i.e. we must ask the revising agent and her coplayers to play the game. So basically, each patch, in sequential order, must:

- play the game,
- ask its coplayers to play the game (so their payoffs are updated),
- run update-strategy-after-revision to compute its next strategy (C-player?-after-revision),
- update its color (now that we have access both to the current strategy C-player? and to the next strategy C-player?-after-revision)
- update its strategy, i.e. set the value of C-player? to C-player?-after-revision. This is done in procedure update-strategy.

Taking all this into account, the code in the procedure to go looks as follows:

```
to go
  ifelse synchronous-updating?
    [
      ask patches [ play ]
      ask patches [
        update-strategy-after-revision
          ;; here we are not updating the agent's strategy yet
        update-color
      ]
      ask patches [ update-strategy ]
        ;; now we update every agent's strategy at the same time
    ]
    [
      ask patches [
        play
        ask my-coplayers [ play ]
          ;; since your coplayers' strategies or
          ;; your coplayers' coplayers' strategies
          ;; could have changed since the last time
          ;; your coplayers played
        update-strategy-after-revision
        update-color
        update-strategy
      ]
    ]
  tick
end
```

## 5.5. Complete code in the Code tab

The Code tab is ready! Congratulations! You have implemented three important generalizations of the model in very little time.

```
patches-own [
  C-player?
  C-player?-after-revision
  payoff
  my-nbrs-and-me
  my-coplayers
  n-of-my-coplayers
]

to setup
  clear-all
  setup-players
  ask patches [update-color]
  reset-ticks
end

to setup-players
  ask patches [
```

```
    set payoff 0
    set C-player? false
    set C-player?-after-revision false
    set my-nbrs-and-me (patch-set neighbors self)
    set my-coplayers ifelse-value self-matching? [my-nbrs-and-me] [neighbors]
    set n-of-my-coplayers (count my-coplayers)
  ]
  ask n-of (round (initial-%-of-C-players * count patches / 100)) patches [
    set C-player? true
    set C-player?-after-revision true
  ]
end

to go
  ifelse synchronous-updating?
    [
      ask patches [ play ]
      ask patches [
        update-strategy-after-revision
          ;; here we are not updating the agent's strategy yet
        update-color
      ]
      ask patches [ update-strategy ]
        ;; now we update every agent's strategy at the same time
    ]
    [
      ask patches [
        play
        ask my-coplayers [ play ]
          ;; since your coplayers' strategies or
          ;; your coplayers' coplayers' strategies
          ;; could have changed since the last time
          ;; your coplayers played
        update-strategy-after-revision
        update-color
        update-strategy
      ]
    ]
  tick
end

to play
  let n-of-cooperators count my-coplayers with [C-player?]
  set payoff
      n-of-cooperators * (ifelse-value C-player? [CC-payoff] [DC-payoff]) +
      (n-of-my-coplayers - n-of-cooperators) *
          ifelse-value C-player? [CD-payoff] [DD-payoff]
end

to update-strategy-after-revision
  set C-player?-after-revision ifelse-value (random-float 1 < noise)
    [ one-of [true false] ]
    [ [C-player?] of one-of (my-nbrs-and-me with-max [payoff]) ]
```

```
    end

    to update-strategy
      set C-player? C-player?-after-revision
    end

    to update-color
      set pcolor
        ifelse-value C-player?-after-revision
         [ifelse-value C-player? [blue] [lime]]
         [ifelse-value C-player? [yellow] [red]]
    end
```

# 6. Sample runs

Now that we have implemented the extended model, we can use it to answer the questions posed in the motivation above. Let us see how the simulation we ran in the previous chapter (with *CD-payoff* = *DD-payoff* = 0, *CC-payoff* = 1, *DC-payoff* = 1.85, and *initial-%-of-C-players* = 90 in a 81×81 grid) is affected by each of the changes outlined in the motivation, one by one. We will refer to this parameterization as *the baseline setting*.

## What happens if we add a bit of noise?

If you run the model with noise, you will see that the level of cooperation diminishes drastically. Using BehaviorSpace, we have estimated that the percentage of cooperators in the regime where cooperators and defectors coexist drops from ~32% in the model without noise to ~15% if *noise* = 0.04. If *noise* = 0.05, the long-run fraction of cooperation is just ~3%, so nearly all cooperation is coming from the random strategy updates (which accounts for 2.5% of the cooperation).[3] The influence of noise in the baseline setting was pointed out by Mukherji et al. (1996).

## What happens if agents do not play the game with themselves?

The impact of *self-matching?* is also clear. When agents do not play the game with themselves, no cooperation can emerge in the baseline setting. If the initial fraction of cooperators is high, some small clusters of initial cooperators may survive, but these clusters disappear if we add a tiny bit of noise. As an illustration, the video below shows a simulation with *self-matching?* = false, *initial-%-of-C-players* = 99 and *noise* = 0.01.

---

3. The model with low noise seems to have two regimes, one where most agents are defecting and another one where cooperators and defectors coexist. Simulations that start with a low percentage of initial cooperators tend to move first to the mostly-defection regime, while simulations that start with higher proportions of initial cooperators tend to move to the coexistence regime. Note, however, that transitions from one regime to the other are always possible with noise, and therefore they will occur if we wait for long enough. Having said that, the time we would have to wait to actually see these transitions may be extremely long in some settings. Note also that the model with noise can be seen as an irreducible and aperiodic Markov chain (see sufficient conditions for irreducibility and aperiodicity). This means that the long-run dynamics of this model are independent of initial conditions.

Therefore, it turns out that playing with oneself is a necessary condition to obtain some cooperation in the baseline setting.

## What happens if strategy updating is asynchronous, rather than synchronous?

The impact of *synchronous-updating?* on cooperation is also clear. If agents update their strategies sequentially, rather than simultaneously, no cooperation whatsoever can be sustained in the baseline setting. This observation was pointed out by Huberman and Glance (1993). As a matter of fact, to eliminate cooperation in this setting, it is sufficient that only a small fraction of the population (~15%) do not synchronize (Mukherji et al., 1996).[4]

## What happens if we use *DD-payoff* = 0.1?

Increasing the value of *DD-payoff* to 0.1 (so the game becomes a real Prisoner's Dilemma) also eliminates the emergence of cooperation. If the initial fraction of cooperators is high, some small clusters of initial cooperators may survive, but these clusters disappear if we add some noise.[5] As an illustration, the video below shows a simulation with *DD-payoff* = 0.1, *initial-%-of-C-players* = 99 and *noise* = 0.01.

## Discussion

In this chapter we have discovered that the emergence of cooperation observed in the sample run of the previous chapter is not robust at all. Any of the four modifications we have explored is sufficient to destroy cooperation altogether. Having said that, the emergence of cooperation in the spatially embedded Prisoner's Dilemma is much more robust for lower values of *DC-payoff* (see Nowak et al. (1994a, 1994b, 1996)). As an example, consider a simulation with *DC-payoff* = 1.3, where we include the four modifications we have investigated, i.e. *noise* = 0.05, *self-matching?* = false, *synchronous-updating?* = false, and *DD-payoff* = 0.1. The other parameter values are the same as in our baseline simulation, i.e. *CD-payoff* = 0, *CC-payoff* = 1, and the grid is 81×81. Cooperation in this setting can indeed emerge and be sustained. The video below shows an illustrative run with initial

---

4. Newth and Cornforth (2009) analyze various other updating schemes in this model.

5. If *DD-payoff* $\geq$ 0.58, no clusters of initial cooperators can survive, even in the absence of noise.

conditions *initial-%-of-C-players* = 25. The long-run proportion of cooperators in this setting is greater than 50%.

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=110#video-110-3*

In chapter III-4 we will see that there is another assumption in this model that has a very important (positive) influence in the emergence of cooperation: the use of the imitate the best neighbor rule. But for now, let us take a step back and think about what we have learned in this chapter in general terms, i.e. beyond the specifics of this particular model.

In this chapter we have learned that assumptions that may seem irrelevant at first sight can actually play a crucial role in the dynamics of our models. Furthermore, there are often complex interactions between the effects of different assumptions. We have also learned that small changes in one parameter can lead to big changes in the dynamics of our models (see exercise 1 below for a striking example). Unfortunately, this sensitivity to seemingly small details is not the exception but the rule in agent-based models. For this reason, it is of utmost importance to always check the robustness of our computational results, to explore the parameter space adequately, and to keep our conclusions within the scope of what we have actually investigated, not beyond.

# 7. Exercises

You can use the following link to download the complete NetLogo model: 2×2-imitate-best-nbr-extended.nlogo.

**Exercise 1.** Roca et al. (2009a, fig. 10; 2009b, fig. 2) report a counterintuitive singularity that we can replicate with our model. To do so, modify the baseline setting (*CD-payoff* = *DD-payoff* = 0, *CC-payoff* = 1) by choosing *self-matching?* = false, make the world 100×100 with periodic (or 'wrap-around') boundaries, and set initial conditions *initial-%-of-C-players* = 50. Now compare the long-run fraction of cooperators for values of *DC-payoff* equal to 1.3999, 1.4 and 1.4001. What do you observe?

To understand this curious phenomenon, you may also want to run simulations with initial conditions *initial-%-of-C-players* = 100 and make use of our button labeled make agent at 0 0 play D.

P.S. One may wonder whether this singularity could be an artifact due to floating-point errors, since (1.4 + 1.4 + 1.4 + 1.4 + 1.4) ≠ 7 in the IEEE754 floating-point standard (which is the standard used in most programming languages, and in NetLogo in particular).[6] You can check that the singularity is not due to floating-point errors choosing an equivalent parameterization that is not prone to floating-point errors. Can you come up with an equivalent parameterization that uses only integers when computing payoffs?

*Photo by Tyler Easton on Unsplash*

**Exercise 2.** Consider the simulation run from the previous chapter which produced the beautiful kaleidoscopic patterns. How does each of the four modifications outlined in the motivation affect its dynamics?

**Exercise 3.** How can we parameterize our model to replicate the results shown in figure 2 of Killingback and Doebeli (1996, p. 1138)?

**Exercise 4.** What changes should we make in the code to be able to replicate figure 3 of Killingback and Doebeli (1996, p. 1139)? Note that in the model used to produce that figure, individual patches do not update their strategy with 5% probability.

---

6. Note that in our implementation of procedure to play we do not add individual payoffs but we multiply them, so we would not compute (1.4 + 1.4 + 1.4 + 1.4 + 1.4) but instead 5*1.4, which is indeed exactly equal to 7 in IEEE754 floating-point arithmetic. For more on the potential impact of floating-point errors on agent-based models, see Polhill et al. (2006) and Izquierdo and Polhill (2006).

Exercise 5. In section "Sample runs", when we added some noise to the baseline setting, we stated that the percentage of cooperators in the regime where cooperators and defectors coexist is about ~15% if *noise* = 0.04. Try to corroborate this estimation using BehaviorSpace.

Exercise 6. In our model, changing the value of *noise* has an immediate effect on the dynamics of the model at runtime. The same occurs with *synchronous-updating?*, but not with *self-matching?*. How can you make the model respond immediately to changes in *self-matching?* ? Try to do it in a way that does not affect the execution speed.

# III-3. Extension to any number of strategies

## 1. Goal

Our goal here is to extend the model we have created in the previous chapter –which accepted games with 2 strategies only– to model (2-player symmetric) games with any number of strategies.

## 2. Motivation. Spatial Hawk-Dove-Retaliator

The model we are going to develop in this chapter will allow us to explore games with any number of strategies. Thus, we will be able to model games like the classical Hawk-Dove-Retaliator (Maynard Smith, 1982, pp. 17-18), which is an extension of the Hawk-Dove game, with the additional strategy *Retaliator*. Retaliators are just like Doves, except in contests against Hawks. When playing against Hawks, Retaliators behave like Hawks. A possible payoff matrix for this symmetric game is the following:

|                | Hawk (H) | Dove (D) | Retaliator (R) |
|----------------|----------|----------|----------------|
| Hawk (H)       | –1       | 2        | –1             |
| Dove (D)       | 0        | 1        | 1              |
| Retaliator (R) | –1       | 1        | 1              |

Let us consider the population game where agents are matched to play the normal form game with payoffs as above.[1] The only Evolutionarily Stable State (ESS; see Thomas (1984) and Sandholm (2010a, section 8.3)) of this population game is the state (½H + ½D), with half the population playing Hawk and the other half playing Dove (Maynard Smith, 1982, appendix E; Binmore, 2013). Also, note that Retaliators are weakly dominated by Doves: they get a strictly lower expected payoff than Doves in any situation, except in those population states with no Hawks whatsoever (at which retaliators get exactly the same payoff as Doves).

Figure 1 below shows the best response correspondence of this game. Population states are represented in a simplex, and the color at any population state indicates the strategy that provides the highest expected payoff at that state: orange for Hawk, green for Dove, and blue for Retaliator. As an example, the population state where the three strategies are equally present, i.e. (⅓H + ⅓D +⅓R), which lies at the barycenter of the simplex, is colored in green, denoting that the strategy that provides the highest expected payoff at that state is Dove.

---

1. The payoff function of the associated population game is $F(x) = Ax$, where $x$ denotes the population state and $A$ denotes the payoff matrix of the normal form game. This population game can be obtained by assuming that every agent plays with every other agent.
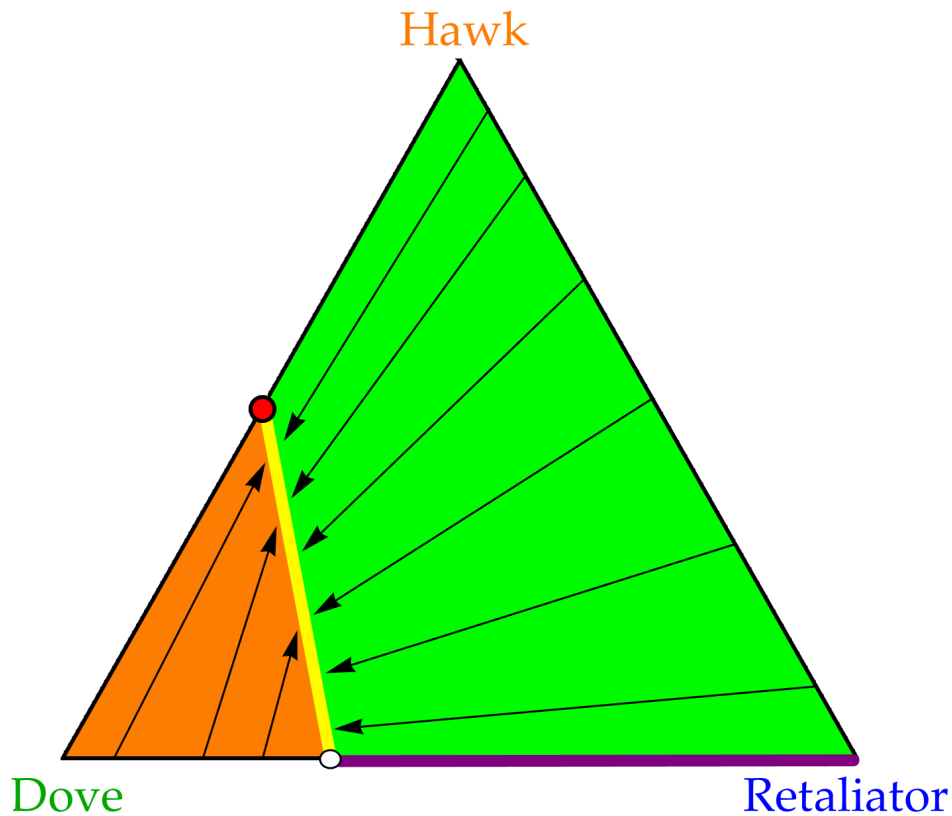
*Fig. 1. Best response correspondence for the Hawk–Dove–Retaliator game. Color indicates the strategy with the highest expected payoff at each population state. Arrows are just a visual aid that indicate the direction of the best response. The yellow line indicates that both Dove and Hawk are best response. The purple line indicates that both Dove and Retaliator are best response. All three strategies are best response at the white circle at (⅔D +⅓R). Finally, the unique ESS (½H + ½D) is indicated with a red circle*

We would like to study the dynamic stability of the unique ESS (½H + ½D) in spatial contexts. In unstructured populations, ESSs are asymptotically stable under a wide range of revision protocols (see e.g. Sandholm (2010a, theorem 8.4.7)), and in particular under the best response rule. Therefore, one might be tempted to think that in our spatial model with the imitate the best neighbor rule (including some noise to allow for the occasional entry of any strategy), simulations will tend to spend most of the time around the unique (½H + ½D) and Retaliators would hardly be observed. This hypothesis may be further supported by the fact that the area around the unique ESS where Retaliators are suboptimal is quite sizable. In no situation can Retaliators obtain a higher expected payoff than Doves, and departing from the unique ESS, at least one half of the population would have to be replaced (i.e. all the Hawks) for Retaliators to get the same expected payoff as Doves.

Having seen all this, it may come as no surprise that if we simulate this game with the random-matching model we implemented in Part II, retaliators tend to disappear from any interior population state. The following video shows an illustrative simulation starting from a situation where all agents are retaliators (and including some noise to allow for the entry of any strategy).[2]

---

2. The fact that the simulation tends to linger around the ESS is a coincidence, since the imitate if better rule depends only on ordinal properties of the payoffs. What is not a coincidence is that Retaliators (which are weakly dominated by Doves) are eliminated in the absence of noise (Loginov, 2021).

So, will space give Retaliators any chance of survival? Let's build a model to explore this question!

# 3. Description of the model

The model we are going to develop here is a generalization of the model implemented in the previous chapter. The new model will have a new parameter, *payoffs*, that the user can set to input a payoff matrix of the form [ [$A_{00}$ $A_{01}$ ... $A_{0n}$] [$A_{10}$ $A_{11}$ ... $A_{1n}$] ... [$A_{n0}$ $A_{n1}$ ... $A_{nn}$] ], containing the payoffs $A_{ij}$ that an agent playing strategy $i$ obtains when meeting an agent playing strategy $j$ ($i, j \in \{0, 1, ..., n\}$). The number of strategies will be inferred from the number of rows in the payoff matrix.

The user will also be able to set any initial conditions using parameter *n-of-players-for-each-strategy*, which will be a list of the form [$a_0$ $a_1$ ... $a_n$], where item $a_i$ is the initial number of agents playing strategy $i$. Naturally, the sum of all the elements in this list should equal the number of patches in the world.

Everything else stays as described in the previous chapter.

# 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).
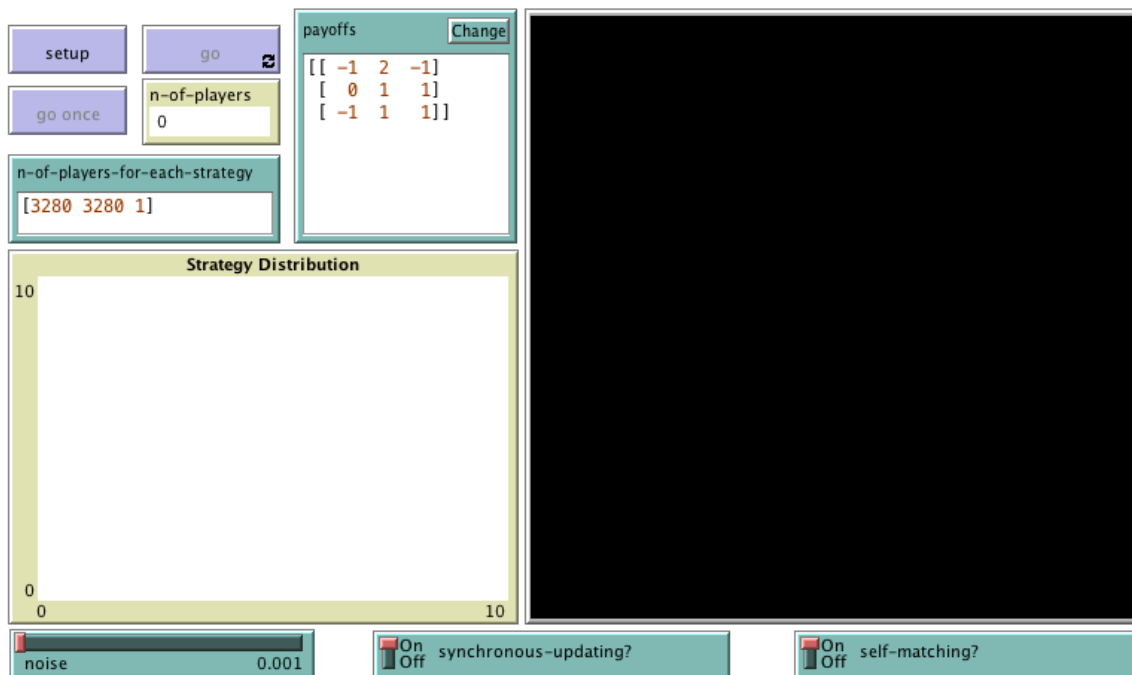
*Fig. 2. Interface design*

The new interface (see figure 2 above) requires the following modifications:

- Remove the sliders for parameters *CC-payoff*, *CD-payoff*, *DC-payoff*, *DD-payoff*, and *initial-%-of-C-players*. Since these sliders were our way of declaring the corresponding global variables, you will now get all sorts of errors, but don't panic, we will sort them out later.

- Remove the button labeled make agent at 0 0 play D. Yes, more errors, but let us do our best to stay calm; we will fix them in a little while.

- Add an input box for parameter *payoffs*.

  > Create an input box with associated global variable *payoffs*. Set the input box type to "String (reporter)" and tick the "Multi-Line" box. Note that the content of *payoffs* will be a string (i.e. a sequence of characters) from which we will have to extract the payoff numeric values.

- Create an input box to let the user set the initial number of players using each strategy.

  > Create an input box with associated global variable *n-of-players-for-each-strategy*. Set the input box type to "String (reporter)".

- Remove the "pens" in the Strategy Distribution plot. Since the number of strategies is unknown until the payoff matrix is read, we will need to create the required number of "pens" in the Code tab.

> Edit the Strategy Distribution plot and delete both pens.

- We have also modified the monitor. Before it showed the ticks and now it shows the number of players (i.e. the value of a global variable named n-of-players, to be defined shortly). You may want to do this or not, as you like.

# 5. Code

## 5.1. Skeleton of the code
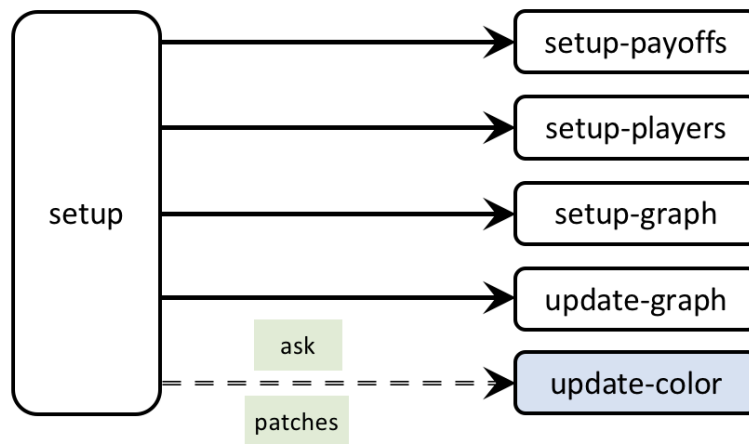


*Figure 3. Skeleton of the setup procedure*
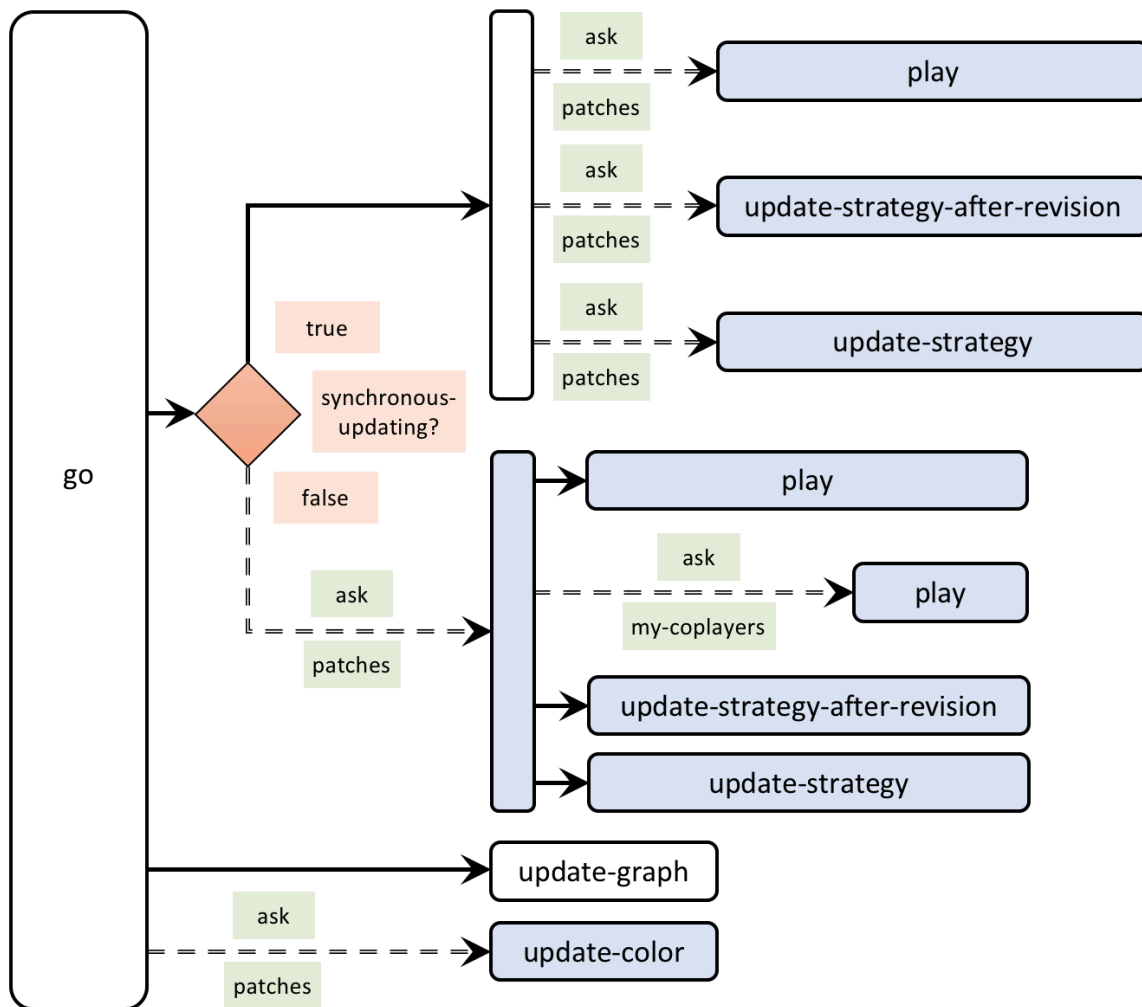
*Figure 4. Skeleton of the go procedure*

## 5.2. Global variables and individually-owned variables

First of all, we declare the global variables that we are going to use and we have not already declared in the interface. We will be using a global variable named payoff-matrix to store the payoff values on a list. It will also be handy to have a variable store the number of strategies and another variable store the number of players. Since this information will likely be used in various procedures and will not change during the course of a simulation, it makes sense to define the new variables as global. The natural names for these two variables are n-of-strategies and n-of-players:

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
]
```

Now we focus on the patches-own variables. We are going to need each individual patch to store its strategy and its strategy-after-revision. These two variables replace the previous C-player? and C-player?-after-revision. Thus, the code for patches-own variables looks as follows now:

```
patches-own [
  ;; C-player?                <== no longer needed
  ;; C-player?-after-revision  <== no longer needed
  strategy                 ;; <== new variable
  strategy-after-revision   ;; <== new variable
  payoff
  my-nbrs-and-me
  my-coplayers
  n-of-my-coplayers
]
```

## 5.3. Setup procedures

The current setup procedure looks as follows:

```
to setup
  clear-all
  setup-players
  ask patches [update-color]
  reset-ticks
end
```

Clearly we will have to keep this code, but additionally we will have to set up the payoffs and set up the graph (since the number of pens to be created depends on the payoff matrix now). To do this elegantly, we should create separate procedures for each set of related tasks; to setup-payoffs and to setup-graph are excellent names for these new procedures. Thus, the code of procedure to setup should include calls to these new procedures:

```
to setup
  clear-all
  setup-payoffs    ;; <== new line
  setup-players
  setup-graph      ;; <== new line
  reset-ticks

  update-graph     ;; <== new line
  ask patches [update-color]
end
```

Note that we have also included a call to another new procedure named to update-graph, to plot the initial conditions.[3] The code of procedure to setup in this model looks almost identical to the code of the same procedure in the model we developed in Part II. As a matter of fact, we will be able to reuse much of the code we wrote for that model. Let us now implement procedures to setup-payoffs, to

---

3. There is some flexibility in the order of the lines within procedure to setup. For instance, the call to procedure setup-graph could be made before or after executing reset-ticks.

setup-graph and to update-graph. We will also have to modify procedures to setup-players and to update-color.

## to setup-payoffs

The procedure to setup-payoffs will include the instructions to read the payoff matrix, and will also set the value of the global variable n-of-strategies. Looking at the implementation of the same procedure in the model we developed in Part II, can you implement procedure to setup-payoffs for our new model?

> Implementation of procedure to setup-payoffs.
>
> Yes, well done! We can use exactly the same code!
>
> ```
> to setup-payoffs
>   set payoff-matrix read-from-string payoffs
>   set n-of-strategies length payoff-matrix
> end
> ```

## to setup-players

The current procedure to setup-players looks as follows:

```
to setup-players
  ask patches [
    set payoff 0
    set C-player? false
    set C-player?-after-revision false
    set my-nbrs-and-me (patch-set neighbors self)
    set my-coplayers ifelse-value self-matching? [my-nbrs-and-me] [neighbors]
    set n-of-my-coplayers (count my-coplayers)
  ]
  ask n-of (round (initial-%-of-C-players * count patches / 100)) patches [
    set C-player? true
    set C-player?-after-revision true
  ]
end
```

This procedure will have to be modified substantially. In particular, the lines in bold in the code above include variables that do not exist anymore. But don't despair! Once again, to modify procedure to setup-players appropriately, the implementation of the same procedure in the model we developed in Part II will be invaluable. Using that code, can you try to implement procedure to setup-players in our new model?

Implementation of procedure to setup-players.

The lines marked in bold below are the only modifications we have to make to the implementation of this procedure from Part II.

```
to setup-players
  let initial-distribution read-from-string n-of-players-for-each-strategy
  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n" n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]
  ask patches [set strategy false]
  let i 0
  foreach initial-distribution [ j ->
    ask n-of j (patches with [strategy = false]) [
      set payoff 0
      set strategy i
      set strategy-after-revision strategy
      set my-nbrs-and-me (patch-set neighbors self)
      set my-coplayers ifelse-value self-matching?
        [my-nbrs-and-me] [neighbors]
      set n-of-my-coplayers (count my-coplayers)
    ]
    set i (i + 1)
  ]
  set n-of-players count patches
end
```

Finally, it would be a nice touch to warn the user if the total number of players in list *n-of-players-for-each-strategy* is not equal to the number of patches. One possible way of doing this is to include the code below, right before setting the patches' strategies to false.

```
if sum initial-distribution != count patches [
  user-message (word "The total number of agents in\n"
    "n-of-agents-for-each-strategy (i.e. "
    sum initial-distribution "):\n" n-of-players-for-each-strategy
    "\nshould be equal to the number of patches (i.e. "
    count patches ")"
  )
]
```

## to setup-graph

The procedure to setup-graph will create the required number of pens –one for each strategy– in the Strategy Distribution plot. Looking at the implementation of the same procedure in the model we developed in Part II, can you implement procedure to setup-graph for our new model?

Implementation of procedure to setup-graph.

Yes, well done! We can use exactly the same code!

```
to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end
```

## to update-graph

Procedure to update-graph will draw the strategy distribution using a stacked bar chart, just like in the model we implemented in Part II (see figure 3 in chapter II-2). This procedure is called at the end of setup to plot the initial distribution of strategies, and will also be called at the end of procedure to go, to plot the strategy distribution at the end of every tick.

Looking at the implementation of the same procedure in the model we developed in Part II, can you implement procedure to update-graph for our new model?

Implementation of procedure to update-graph.

Yes, well done! We only have to replace the word **players** in the previous code with **patches** in the current code.

```
to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
      count patches with [strategy = n] / n-of-players
    ] strategy-numbers
```

```
    set-current-plot "Strategy Distribution"
    let bar 1
    foreach strategy-numbers [ n ->
      set-current-plot-pen (word n)
      plotxy ticks bar
      set bar (bar - (item n strategy-frequencies))
    ]
    set-plot-y-range 0 1
  end
```

## to update-color

Note that in the previous model, patches were colored according to the four possible combinations of values of C-player? and C-player?-after-revision. Now that there can be many strategies, it seems more natural to use one color for each strategy. It also makes sense to use the same color legend as in the Strategy Distribution plot (see procedure to setup-graph). Can you try and implement the new version of to update-color?

Implementation of procedure to update-color.

Here we go!

```
  to update-color
    set pcolor 25 + 40 * strategy
  end
```

# 5.4. Go procedure

The current go procedure looks as follows:

```
  to go
    ifelse synchronous-updating?
      [
        ask patches [ play ]
        ask patches [
          update-strategy-after-revision
            ;; here we are not updating the agent's strategy yet
          update-color
        ]
```

```
      ask patches [ update-strategy ]
        ;; now we update every agent's strategy at the same time
    ]
    [
      ask patches [
        play
        ask my-coplayers [ play ]
          ;; since your coplayers' strategies or
          ;; your coplayers' coplayers' strategies
          ;; could have changed since the last time
          ;; your coplayers played
        update-strategy-after-revision
        update-color
        update-strategy
      ]
    ]
    tick
  end
```

In the previous version of the model, the call to update-color had to be done in between the calls to update-strategy-after-revision and update-strategy. Now that the patches' color only depends on their (updated) strategy, we should ask patches to run update-color at the end of procedure to go, after every patch has updated its strategy.

Finally, recall that we also have to run update-graph at the end of procedure to go, to plot the strategy distribution at the end of every tick. Thus, the code of procedure to go will be as follows:

```
to go
  ifelse synchronous-updating?
    [
      ask patches [ play ]
      ask patches [ update-strategy-after-revision ]
        ;; here we are not updating the agent's strategy yet
      ask patches [ update-strategy ]
        ;; now we update every agent's strategy at the same time
    ]
    [
      ask patches [
        play
        ask my-coplayers [ play ]
          ;; since your coplayers' strategies or
          ;; your coplayers' coplayers' strategies
          ;; could have changed since the last time
          ;; your coplayers played
        update-strategy-after-revision
        update-strategy
      ]
    ]
    tick
    update-graph                    ;; <== new line
```

```
    ask patches [update-color]    ;; <== new line
  end
```

## 5.5. Other procedures

### to play

In procedure to play the patch has to compute its payoff. For that, the patch must count how many of its coplayers are using each of the possible strategies. We can count the number of coplayers that are using strategy $i \in \{0, 1, ..., (\text{n-of-strategies} - 1)\}$ as:

```
count my-coplayers with [strategy = i]
```

Thus, we just have to run this little function for each value of $i \in \{0, 1, ..., (\text{n-of-strategies} - 1)\}$. This can be easily done using primitive `n-values`:

```
n-values n-of-strategies [ i -> count my-coplayers with [strategy = i] ]
```

The code above produces a list with the number of coplayers that are using each strategy. Let us store this list in local variable n-of-coplayers-with-strategy-?:

```
let n-of-coplayers-with-strategy-? n-values n-of-strategies [ i ->
    count my-coplayers with [strategy = i] ]
```

Now note that the relevant row of the payoff-matrix is the one at position strategy. We store this row in local variable my-payoffs:

```
let my-payoffs (item strategy payoff-matrix)
```

Finally, the payoff that the patch will get for each coplayer playing strategy *i* is the *i*-th element of the list my-payoffs, so we only have to multiply the two lists (my-payoffs and n-of-coplayers-with-strategy-?) element by element, and add up all the elements in the resulting list. To multiply the two lists element by element we use primitive `map`:

```
sum (map * my-payoffs n-of-coplayers-with-strategy-?)
```

With this, we have finished the code in procedure to play.

```
to play
  let n-of-coplayers-with-strategy-? n-values n-of-strategies [ i ->
    count my-coplayers with [strategy = i] ]
  let my-payoffs (item strategy payoff-matrix)
  set payoff sum (map * my-payoffs n-of-coplayers-with-strategy-?)
end
```

### to update-strategy-after-revision

Right now, procedure to update-strategy-after-revision is implemented as follows:

```
to update-strategy-after-revision
  set C-player?-after-revision ifelse-value (random-float 1 < noise)
    [ one-of [true false] ]
    [ [C-player?] of one-of (my-nbrs-and-me with-max [payoff]) ]
end
```

What changes do we have to make in this procedure?

Implementation of procedure to update-strategy-after-revision.

The only changes we have to make are highlighted in bold below:

```
to update-strategy-after-revision
  set strategy-after-revision ifelse-value (random-float 1 < noise)
    [ random n-of-strategies ]
    [ [strategy] of one-of (my-nbrs-and-me with-max [payoff]) ]
end
```

## to update-strategy

Right now, procedure to update-strategy is implemented as follows:

```
to update-strategy
  set C-player? C-player?-after-revision
end
```

What changes do we have to make in this procedure?

Implementation of procedure to update-strategy.

Keep up the excellent work!

```
to update-strategy
  set strategy strategy-after-revision
end
```

## 5.6. Complete code in the Code tab

The Code tab is ready!

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
]

patches-own [
  strategy
  strategy-after-revision
  payoff
  my-nbrs-and-me
  my-coplayers
  n-of-my-coplayers
]

to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
  ask patches [update-color]
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end

to setup-players
  let initial-distribution read-from-string n-of-players-for-each-strategy
  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n" n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]
  if sum initial-distribution != count patches [
    user-message (word "The total number of agents in\n"
      "n-of-agents-for-each-strategy (i.e. "
      sum initial-distribution "):\n" n-of-players-for-each-strategy
      "\nshould be equal to the number of patches (i.e. "
      count patches ")"
    )
```

```
    ]
    ask patches [set strategy false]
    let i 0
    foreach initial-distribution [ j ->
      ask n-of j (patches with [strategy = false]) [
        set payoff 0
        set strategy i
        set strategy-after-revision strategy
        set my-nbrs-and-me (patch-set neighbors self)
        set my-coplayers ifelse-value self-matching?
          [my-nbrs-and-me] [neighbors]
        set n-of-my-coplayers (count my-coplayers)
      ]
      set i (i + 1)
    ]
    set n-of-players count patches
  end

  to setup-graph
    set-current-plot "Strategy Distribution"
    foreach (range n-of-strategies) [ i ->
      create-temporary-plot-pen (word i)
      set-plot-pen-mode 1
      set-plot-pen-color 25 + 40 * i
    ]
  end

  to go
    ifelse synchronous-updating?
      [
        ask patches [ play ]
        ask patches [ update-strategy-after-revision ]
          ;; here we are not updating the agent's strategy yet
        ask patches [ update-strategy ]
          ;; now we update every agent's strategy at the same time
      ]
      [
        ask patches [
          play
          ask my-coplayers [ play ]
            ;; since your coplayers' strategies or
            ;; your coplayers' coplayers' strategies
            ;; could have changed since the last time
            ;; your coplayers played
          update-strategy-after-revision
          update-strategy
        ]
      ]
    tick
    update-graph
    ask patches [update-color]
  end
```

```
to play
  let n-of-coplayers-with-strategy-? n-values n-of-strategies [ i ->
    count my-coplayers with [strategy = i] ]
  let my-payoffs (item strategy payoff-matrix)
  set payoff sum (map * my-payoffs n-of-coplayers-with-strategy-?)
end


to update-strategy-after-revision
  set strategy-after-revision ifelse-value (random-float 1 < noise)
    [ random n-of-strategies ]
    [ [strategy] of one-of my-nbrs-and-me with-max [payoff] ]
end


to update-strategy
  set strategy strategy-after-revision
end


to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
    count patches with [strategy = n] / n-of-players ] strategy-numbers
  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end


to update-color
  set pcolor 25 + 40 * strategy
end
```

## 5.7. Code inside the plots

Note that we take care of all plotting in the update-graph procedure. Thus there is no need to write any code inside the plot. We could instead have written the code of procedure to update-graph inside the plot, but given that it is somewhat lengthy, we find it more convenient to group it with the rest of the code in the Code tab.

# 6. Sample runs

Now that we have implemented the model we can explore the dynamics of the spatial Hawk-Dove-Retaliator game! Will Retaliators survive in a spatial context? Let us explore this question using the parameter values shown in figure 2 above. Get ready... because the results are going to blow your mind!

Unbelievable! Retaliators do not only survive, but they are capable of taking over about half the population. Is this observation robust? If you modify the parameters of the model you will see that indeed it is. The following video shows an illustrative run with *noise* = 0.05, *synchronous-updating?* = false and *self-matching?* = false.

The greater level of *noise* means that more Hawks appear by chance. This harms Retaliators more than it harms Doves, but Retaliators still manage to stay the most prevalent strategy in the population. How can this be?

First, note that even though the state where the whole population is choosing Retaliator is not an ESS, it is a Neutrally Stable State (Sandholm, 2010a, p. 82). And, crucially, it is the only pure state that is Nash (i.e. the only pure strategy that is best response to itself). Note that in spatial contexts neighbors face similar situations when playing the game (since their neighborhoods overlap). Because of this, it is often the case that neighbors choose the same strategy, and therefore clusters of agents using the same strategy are common. In the Hawk-Dove-Retaliator game, clusters of Retaliators are more stable than clusters of Doves (which are easily invadable by Hawks) and also more stable than clusters of Hawks (which are easily invadable by Doves). This partially explains the amazing success of Retaliators in spatial contexts, even though they are weakly dominated by Doves.

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-imitate-best-nbr.nlogo.

Exercise 1. Killingback and Doebeli (1996, pp. 1140-1) explore the spatial Hawk-Dove-Retaliator-Bully game, with payoff matrix:

```
[[ -1  2 -1 2]
 [  0  1  1 0]
 [ -1  1  1 2]
 [  0  2  0 1]]
```

Do Retaliators still do well in this game?

Exercise 2. Explore the beautiful dynamics of the following monocyclic game (Sandholm, 2010a, example 9.2.2, pp. 329-30):

```
[[ 0 -1  0  0  1]
 [ 1  0 -1  0  0]
 [ 0  1  0 -1  0]
 [ 0  0  1  0 -1]
 [-1  0  0  1  0]]
```
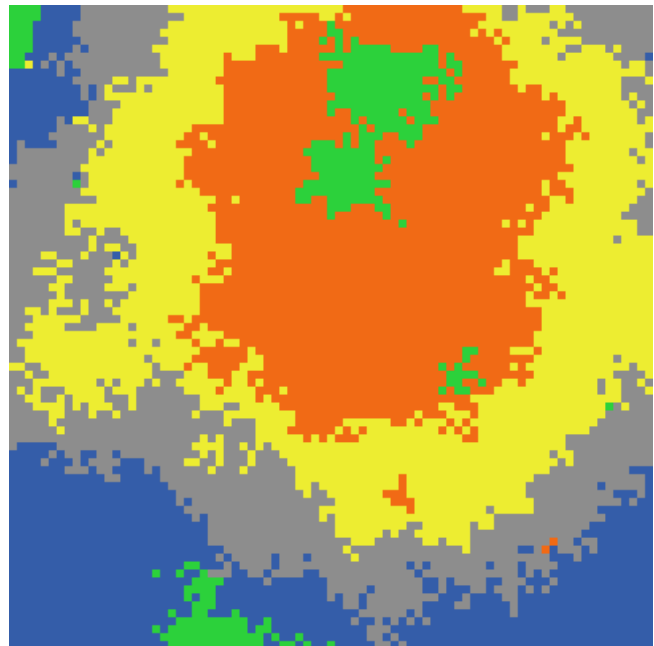


*Snapshot of a simulation run of a spatial monocyclic game with noise = 0.001, synchronous-updating? = false and self-matching? = true.*

Compare simulations with balanced initial conditions (i.e. all strategies approximately equally present) and with unbalanced initial conditions (e.g. only one strategy present at the beginning of the simulation). What do you observe?

Exercise 3. How can we parameterize our model to replicate the results shown in figure 4 of Killingback and Doebeli (1996, p. 1141)?

Exercise 4. In procedure to play we compute the list with the number of coplayers that are using each strategy as follows:

```
n-values n-of-strategies [ i -> count my-coplayers with [strategy = i] ]
```

Can you implement the same functionality using the primitive `map` instead of `n-values`?

Exercise 5. Reimplement the procedure to update-strategy-after-revision so the revising agent uses the imitative pairwise-difference rule adapted to networks, i.e. the revising agent looks at a random neighbor and copies her strategy only if the observed agent's average payoff is higher than the revising agent's average payoff; in that case, the revising agent switches with probability proportional to the payoff difference.

# III-4. Other types of neighborhoods and other decision rules

## 1. Goal

Our goal in this chapter is to extend the model we have created in the previous chapter by adding two features that are crucial to assess the impact of space on evolutionary models:

- The possibility to model different types of neighborhoods of arbitrary size. Besides Moore neighborhoods, we will implement Von Neumann neighborhoods, and both of them of any size.

- The possibility to model other decision rules besides the imitate the best neighbor rule. In particular, we will implement the *imitative pairwise-difference* rule, the *imitate if better* rule, the *imitative positive proportional* rule, and the *Fermi* rule.[1]

## 2. Motivation. The impact of decision rules

In chapter III-2 we explored the impact of different assumptions on the robustness of cooperation in spatial settings. However, one assumption we did not change was the decision rule (aka *update rule*). Roca et al. (2009a, 2009b) conducted an impressive simulation study of the effect of spatial structure in 2×2 games, and discovered that the decision rule can play a major role. In particular, they found that the imitate the best neighbor rule (which Roca et al. (2009a, 2009b) call "unconditional imitation") favors cooperation in the Prisoner's Dilemma more than any other decision rule they studied.

> In what concerns the Prisoner's Dilemma, the above results also prove that the promotion of cooperation in this game is not robust against changes in the update rule, because the beneficial effect of spatial lattices practically disappears for rules different from unconditional imitation, when seen in the wider scope of the ST plane. Roca et al. (2009b, p. 9)

In this chapter we are going to implement several decision rules. This will allow us to replicate Roca et al.'s (2009a, 2009b) results… and many others (see the proposed exercises).

---

1. The names given to the different rules follow Izquierdo et al. (2019).

# 3. Description of the model

The model we are going to develop here is a generalization of the model implemented in the previous chapter. In particular, we are going to add the following four parameters:

- *neighborhood-type*. This parameter is used to define the agents' neighborhood (both for playing and for strategy updating).[2] The parameter can have two possible values: "*Moore*" for a Moore neighborhood, or "*Von Neumann*" for a Von Neumann neighborhood.

- *neighborhood-range*. This parameter determines the range of the neighborhood. A patch's Moore neighborhood of range *r* consists of the patches within Chebyshev distance *r*. A patch's Von Neumann neighborhood of range *r* is composed of the patches within Manhattan distance *r*. Note that in our descriptions of decision rules below, we do not consider a patch to be a neighbor of itself unless otherwise stated.

- *decision-rule*. This parameter determines the decision rule that agents will follow. It will be implemented with a chooser, with five possible values:

  - "*best-neighbor*" (Nowak and May, 1992, 1993). This is the imitate the best neighbor rule, already implemented in our model.[3]

  - "*imitate-if-better-all-nbrs*". This is the imitate if better rule (Izquierdo and Izquierdo, 2013; Loginov, 2021) adapted to networks, where agents play with all their neighbors. Under this rule, the revising agent looks at a random neighbor and copies her strategy if and only if the observed neighbor's average payoff is higher than the revising agent's average payoff.

  - "*imitative-pairwise-difference*" (Hauert, 2002,[4] 2006). This is the imitative pairwise-difference rule we saw in chapter I-2 adapted to networks. Under this rule, the revising agent looks at a random neighbor and considers copying her strategy only if the observed neighbor's average payoff is higher than the revising agent's average payoff; in that case, the revising agent switches with probability proportional to the payoff difference.[5]

  - "*imitative-positive-proportional-m*" (Nowak et al., 1994a, b). Under this decision rule, the revising agent copies the strategy of one of her neighbors (including herself, in this case), selected with probability proportional to their total payoff raised to parameter *m*.[6] Parameter *m* controls the intensity of selection (see below). We do not allow for

---

2. Recall that the set of patches that a patch plays with also depends on the value of *self-matching?*.
3. Roca et al. (2009a, 2009b) call this decision rule "unconditional imitation" and Hauert (2002) calls it "best takes over". Also, note that Hauert (2002) resolves ties differently.
4. See Roca et al. (2009b) for an important and illuminating discussion of this paper.
5. Roca et al. (2009a, 2009b) call this decision rule "replicator rule" or "proportional imitation rule", though their implementation is not exactly the same as ours. They use total payoffs and we use average payoffs. The two implementations differ only if it is possible that two agents have different number of neighbors (e.g. as with non-periodic boundary conditions). Note, however, that Roca et al. (2009a, 2009b)'s experiments have periodic boundary conditions, i.e. all agents have the same number of neighbors, so for all these cases the two implementations are equivalent. Hauert (2002) calls this decision rule "imitate the better".

negative payoffs when using this rule.[7]

- ◦ "*Fermi-m*" (Szabó and Tőke, 1998; Traulsen and Hauert, 2009; Roca et al. 2009a, 2009b; Perc and Szolnoki, 2010, Adami et al., 2016). Under this decision rule, the revising agent $i$ looks at one of her neighbors $j$ at random and copies her strategy with probability $\frac{e^{m\pi_j}}{e^{m\pi_i}+e^{m\pi_j}}$, where $\pi_z$ denotes agent $z$'s average payoff and $m \geq 0.$[8] Parameter *m* controls the intensity of selection (see below).

- *m*. This is a parameter that controls the intensity of selection in decision rules *imitative-positive-proportional-m* and *Fermi-m* (see above). High values of *m* imply that selection is strongly based on payoffs, i.e. the agents with the highest payoffs will be chosen with very high probability. Lower values of *m* mean that the sensitivity of selection to payoffs is weak (e.g. if *m* = 0, selection among agents is random, i.e. independent of their payoffs).

Everything else stays as described in the previous chapter.

# ▶ 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).

---

6. It may make more sense to use *average* rather than *total* payoffs. We use total payoffs here to be able to replicate Nowak et al.'s (1994a, b) experiments. To use average payoffs, the change in the code is minimal.

7. Roca et al. (2009a, 2009b) call this decision rule with *m* = 1 "Moran rule". Their implementation is not exactly the same as ours, since they do allow for negative payoffs by introducing a constant. Note, however, that the two implementations are equivalent if payoffs are non-negative. Hauert (2002) calls this decision rule with *m* = 1 "proportional update".

8. Note that some authors (e.g. Szabó and Tőke, 1998) use *total* payoffs rather than *average* payoffs.
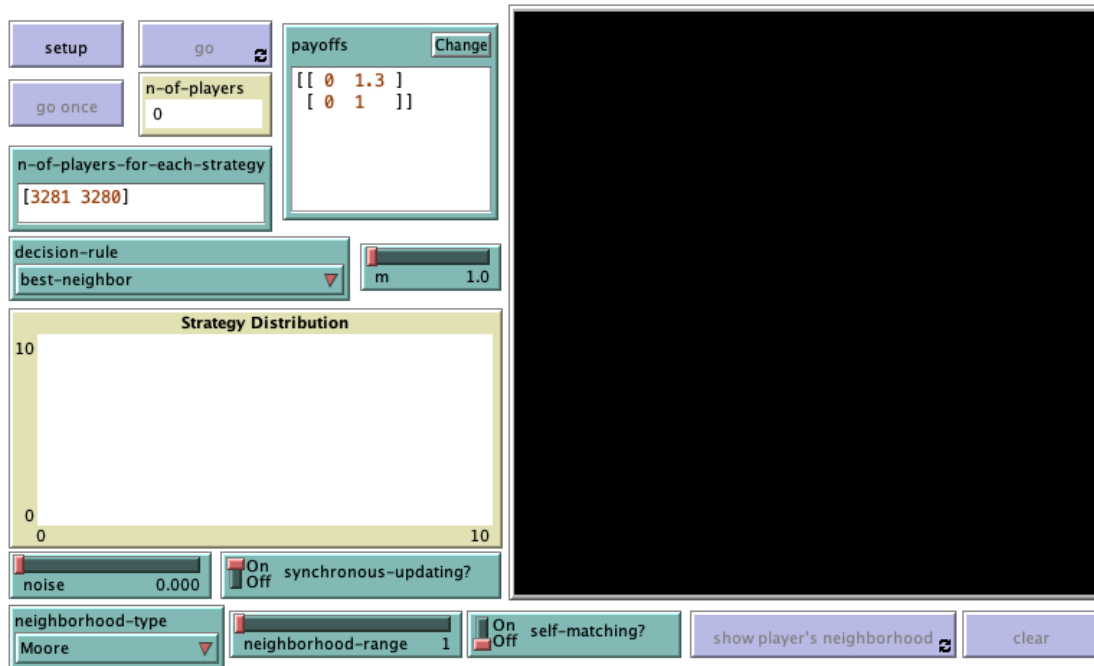
*Figure 1. Interface design*

In the new interface (see figure 1 above), we have to add:

- One chooser for new parameter *neighborhood-type* (with possible values "*Moore*" and "*Von Neumann*"), and a slider for parameter *neighborhood-range* (with *minimum = 1* and *increment = 1*).

- One chooser for new parameter *decision-rule* (with possible values "*best-neighbor*", "*imitate-if-better-all-nbrs*", "*imitative-pairwise-difference*", "*imitative-positive-proportional-m*" and "*Fermi-m*"), and a slider for parameter *m* (with *minimum = 0* and *increment = 0.1*).

We have also added a button (labeled show player's neighborhood) to see any patch's neighborhood on the 2D view and another button (labeled clear) to clear the display of the neighborhood.

# 5. Code

## 5.1. Skeleton of the code

The skeleton of the code for procedures to setup and to go is the same as in the previous model. In this chapter we will modify mainly the following two procedures:

- procedure to setup-players, to set the players' neighborhoods according to parameters *neighborhood-type* and *neighborhood-range* (see figure 2).
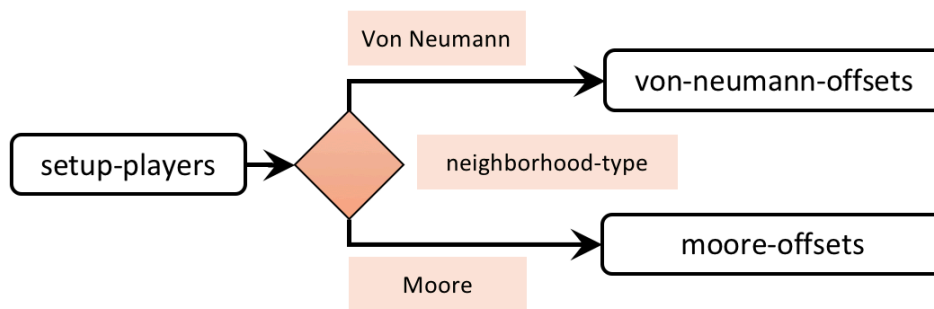
Figure 2. Calls to other procedures from procedure to setup–players

- procedure to update-strategy-after-revision, to run the decision rule indicated in parameter *decision-rule* (see figure 3).
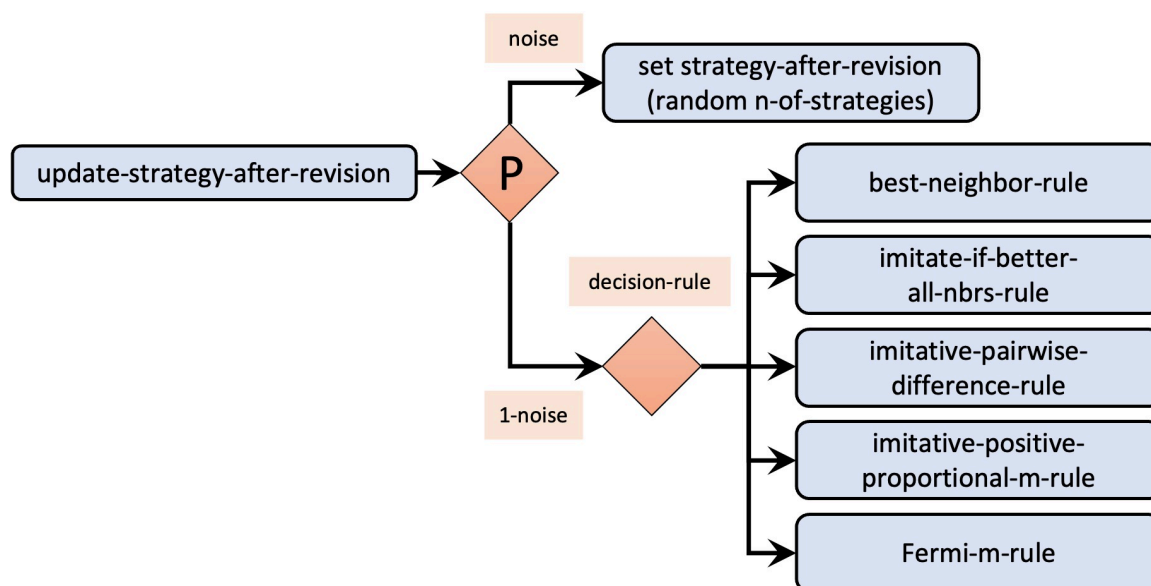


Figure 3. Calls to other procedures from procedure to update–strategy–after–revision

## 5.2. Extension I. Implementation of different neighborhoods

Recall that patches have the following two individually owned variables:

- my-coplayers, which contains the set of agents with whom the patch plays the game (and is affected by parameter *self-matching?*), and

- my-nbrs-and-me, which is the set of agents the player considers when revising its strategy.

These two agentsets are the same if *self-matching?* is true, and differ only in the focal patch if *self-matching?* is false. We will keep this distinction for any type of neighborhood.

To implement the different neighborhoods, primitive `at-points` will be very useful. This primitive reports a subset of a given agentset that includes only the agents on the patches at the given coordinates (relative to the calling agent). As an example, the following code gives a patch's Von Neumann neighborhood of range 1 (including the patch itself):

```
patches at-points [[-1 0] [0 -1] [0 0] [0 1] [1 0]]
```

And the following code gives a patch's Moore neighborhood of range 1 (including the patch itself):

```
patches at-points [[-1 -1] [-1 0] [-1 1] [0 -1] [0 0] [0 1] [1 -1] [1 0] [1
1]]
```

Thus, the key is to implement procedures that report the appropriate lists of relative coordinates.[9]
Let's do this!

## Implementation of relative coordinates for Moore neighborhoods

Our goal here is to implement a procedure that reports the appropriate list of relative coordinates for Moore neighborhoods, for any given range and letting the user choose whether the list should include the item [0 0] or not. Let us call this reporter to-report moore-offsets. Note that this reporter takes two inputs, which we can call r (for range) and include-center?.

```
to-report moore-offsets [r include-center?]
   ;; code to be written
end
```

Below we provide some examples of what reporter to-report moore-offsets should produce:

- If r = 1 and include-center? is true:

    ```
    [[-1 -1] [-1 0] [-1 1] [0 -1] [0 0] [0 1] [1 -1] [1 0] [1 1]]
    ```

- If r = 1 and include-center? is false:

    ```
    [[-1 -1] [-1 0] [-1 1] [0 -1] [0 1] [1 -1] [1 0] [1 1]]
    ```

- If r = 2 and include-center? is true:

    ```
    [[-2 -2] [-2 -1] [-2 0] [-2 1] [-2 2] [-1 -2] [-1 -1] [-1 0] [-1 1] [-1
    2] [0 -2] [0 -1] [0 0] [0 1] [0 2] [1 -2] [1 -1] [1 0] [1 1] [1 2] [2
    -2] [2 -1] [2 0] [2 1] [2 2]]
    ```

Note that the effect of input include-center? is just to include or exclude [0 0] from the output list, so we can worry about that at the end of the implementation. To produce the list of Moore offsets for range r, we can start building the list l = [ –r, –r+1, …, r-1, r ] and then build 2-item lists with each element of the list l together with each element of the same list l. Thus, let us build the list l = [ –r, –r+1, …, r-1, r ]:

```
let l (range (- r) (r + 1))
```

To build 2-items lists with first element *el1* and second element each of the elements of list l, we can use primitive map as follows:

---

9. Our implementation is heavily based on code example titled "Moore & Von Neumann Example", which you can find in NetLogo models library. Note, however, that this code example does not compute the correct neighborhoods if the center of the world lies at the edge or at a corner of the world (at least in version 6.4.0 and previous ones).

```
    map [el2 -> list el1 el2] l
```

And now we should make *el1* be each of the elements of list l. For this we can use primitive map again:

```
    map [ el1 -> map [el2 -> list el1 el2] l] l
```

The only problem now is that we have several nested lists. For example, the code above for l = [ -1, 0, 1 ] produces:

```
    [[[-1 -1] [-1 0] [-1 1]] [[0 -1] [0 0] [0 1]] [[1 -1] [1 0] [1 1]]]
```

Note that the outmost list contains three sublists, each of which contains three 2-item lists. We can get rid of the extra lists using primitives reduce and sentence:

```
    let result reduce sentence map [ el1 -> map [el2 -> list el1 el2] l] l
```

The code above creates the list of required offsets, including [0 0]. Now we just have to remove [0 0] if and only if input include-center? is false. Thus, the final code for reporter to-report moore-offsets is as follows:

```
    to-report moore-offsets [r include-center?]
      let l (range (- r) (r + 1))
      let result reduce sentence map [ el1 -> map [el2 -> list el1 el2] l] l
      report ifelse-value include-center?
        [ result ]
        [ remove [0 0] result ]
    end
```

## Implementation of relative coordinates for Von Neumann neighborhoods

Our goal here is to implement a procedure that reports the appropriate list of relative coordinates for Von Neumann neighborhoods. Let us call this reporter to-report von-neumann-offsets and let us call its inputs r (for range) and include-center?, just like before.

```
    to-report von-neumann-offsets [r include-center?]
        ;; code to be written
    end
```

Note that, for any given r and include-center?, the list of offsets for a Von Neumann neighborhood is a subset of the list of offsets for the corresponding Moore neighborhood. In particular, the list of Von Neumann offsets is composed of the elements [*el1 el2*] in the list of Moore offsets that satisfy the condition $|el1| + |el2| \leq r$ . Thus, we can create the corresponding list of Moore offsets and filter those coordinates that satisfy the condition using primitive filter. Do you want to give it a try?

Yes, well done!

```
to-report von-neumann-offsets [r include-center?]
  let moore-list (moore-offsets r include-center?)
  report filter [l -> abs first l + abs last l <= r] moore-list
end
```

## Putting everything together

Now that we have implemented to-report moore-offsets and to-report von-neumann-offsets, we can use them in procedure to setup-players to create the right neighborhood for each patch. The only lines we have to modify are the ones highlighted in bold below:

```
to setup-players
  let initial-distribution read-from-string n-of-players-for-each-strategy
  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n" n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]
  if sum initial-distribution != count patches [
    user-message (word "The total number of agents in\n"
      "n-of-agents-for-each-strategy (i.e. "
      sum initial-distribution "):\n" n-of-players-for-each-strategy
      "\nshould be equal to the number of patches (i.e. "
      count patches ")"
    )
  ]
  ask patches [set strategy false]
  let i 0
  let offsets ifelse-value (neighborhood-type = "Von Neumann")
        [ von-neumann-offsets neighborhood-range self-matching? ]
        [ moore-offsets neighborhood-range self-matching? ]
  foreach initial-distribution [ j ->
    ask n-of j (patches with [strategy = false]) [
      set payoff 0
      set strategy i
      set strategy-after-revision strategy
      set my-coplayers patches at-points offsets
```

```
      set n-of-my-coplayers (count my-coplayers)
      set my-nbrs-and-me (patch-set my-coplayers self)
    ]
    set i (i + 1)
  ]
  set n-of-players count patches
end
```

Note that variable my-coplayers may or may not contain the patch itself (depending on the value of parameter *self-matching?*), but variable my-nbrs-and-me will always contain it, since we are setting its value to (patch-set my-coplayers self). This is perfectly fine even if my-coplayers already contains the patch itself, since agentsets do not contain duplicates. Adding an agent *a* to an agentset that already contains that agent *a* has no effect whatsoever.[10]

## A nice final touch

Finally, we are going to write some code to let the user see the neighborhood of any patch in the 2D view by clicking on it. This is just for displaying purposes, so feel free to skip this if you're not really interested. Let us start by implementing a new procedure called to show-neighborhood as follows:

```
to show-neighborhood
  if mouse-down? [
    ask patch mouse-xcor mouse-ycor [
      ask my-coplayers [set pcolor white]
    ]
  ]
  display
end
```

You may want to read the documentation for primitives `mouse-down?`, `mouse-xcor` and `mouse-ycor`. Basically, this procedure paints in white the patches contained in the variable my-coplayers of the patch you click with your mouse. However, for this to work, the procedure must be running all the time. To do this, we can run this procedure within the button labeled show player's neighborhood in the interface, and make this button be a "forever" button.

> Insert the following code within the button labeled show player's neighborhood:
>
> ```
> with-local-randomness [show-neighborhood]
> ```

---

10. For instance, the following code reports true.

```
show (patch-set patch 0 0 patch 0 0) = (patch-set patch 0 0)
```

The use of primitive `with-local-randomness` guarantees that this piece of code does not interfere with the generation of pseudorandom numbers for the rest of the model.

> To clear the white paint, insert the following code within the button labeled clear (which should not be a "forever" button):
>
> ```
> with-local-randomness [ ask patches [update-color] ]
> ```

Figure 4 shows a patch's Von Neumann neighborhood of range 5 (excluding the patch itself) painted in white.
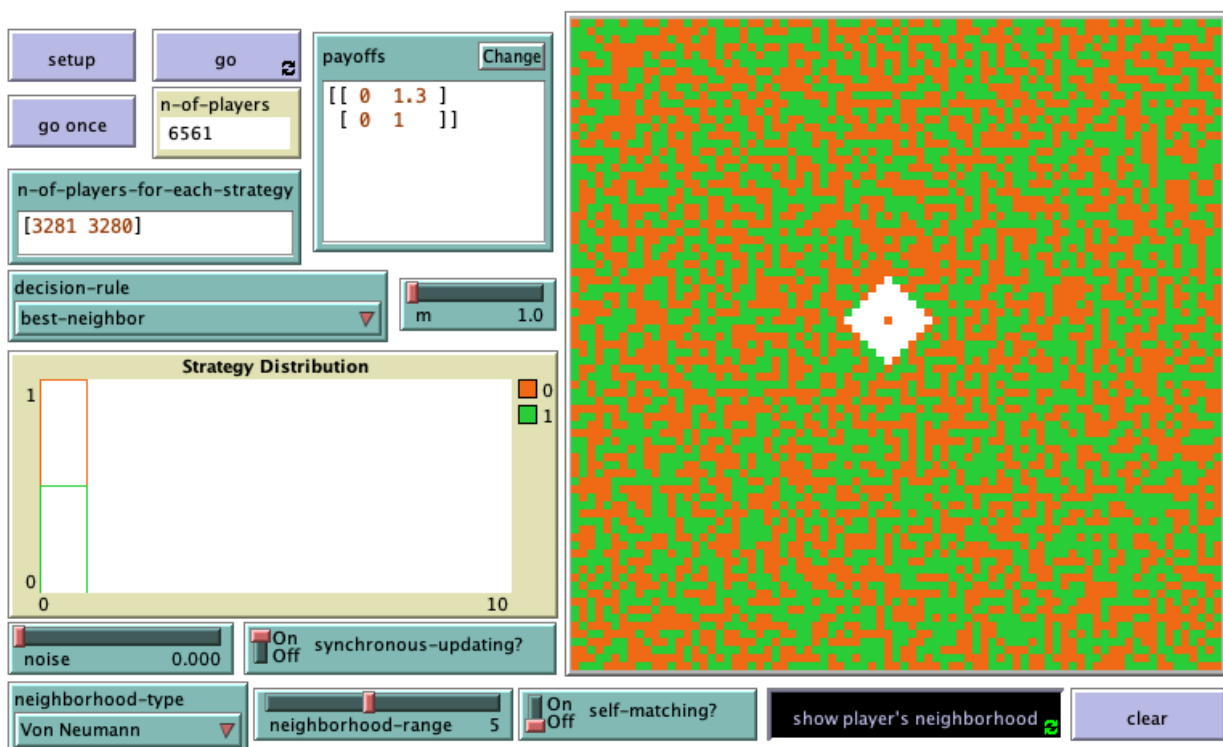


## 5.3. Extension II. Implementation of different decision rules

The implementation of the decision rule takes place in procedure to update-strategy-after-revision. Note that the effect of noise is the same regardless of the decision rule, so we can deal with noise in a unified way, regardless of which decision rule will be executed. One possible way of doing this is as follows:

```
to update-strategy-after-revision
  ifelse (random-float 1 < noise)
    [ set strategy-after-revision random n-of-strategies ]
    [
      ;; code to set strategy-after-revision
      ;; using the decision rule indicated by the user
```

```
      ;; through parameter decision-rule
    ]
  end
```

To make the implementation of decision rules elegant and modular, we should implement a different procedure for each decision rule. Let us call these procedures: best-neighbor-rule, imitate-if-better-all-nbrs-rule, imitative-pairwise-difference-rule, imitative-positive-proportional-m-rule and Fermi-m-rule. With these procedures in place, the code for procedure to update-strategy-after-revision would just look as follows:

```
to update-strategy-after-revision
  ifelse (random-float 1 < noise)
    [ set strategy-after-revision random n-of-strategies ]
    [ (ifelse
        decision-rule = "best-neighbor"
                  [ best-neighbor-rule ]
        decision-rule = "imitate-if-better-all-nbrs"
                  [ imitate-if-better-all-nbrs-rule ]
        decision-rule = "imitative-pairwise-difference"
                  [ imitative-pairwise-difference-rule ]
        decision-rule = "imitative-positive-proportional-m"
                  [ imitative-positive-proportional-m-rule ]
        decision-rule = "Fermi-m"
                  [ Fermi-m-rule ]
    )
    ]
end
```

Note that primitive `ifelse` can work with multiple conditions, just like switch statements in other programming languages.[11]

Now we just have to implement the procedures for each of the four decision rules. Do you want to give it a try? The first one is not very difficult.

Implementation of procedure to best-neighbor-rule

```
to best-neighbor-rule
  set strategy-after-revision
      [strategy] of one-of my-nbrs-and-me with-max [payoff]
```

---

11. This functionality was added in NetLogo 6.1.

```
    end
```

Implementing procedure to imitate-if-better-all-nbrs-rule is slightly more difficult, but we believe you can do it. Remember that players consider average payoffs, rather than total payoffs.

**Implementation of procedure to imitate-if-better-all-nbrs-rule**

```
to imitate-if-better-all-nbrs-rule
  let observed-player one-of other my-coplayers

  if ([payoff / n-of-my-coplayers] of observed-player) >
      (payoff / n-of-my-coplayers) [
    set strategy-after-revision ([strategy] of observed-player)
  ]
end
```

The implementation of procedure to imitative-pairwise-difference-rule is significantly more difficult than the previous one, but if you have managed to read this book until here, you certainly have what it takes to do it!

**Implementation of procedure to imitative-pairwise-difference-rule**

A possible implementation of this procedure is as follows:

```
to imitative-pairwise-difference-rule
  let observed-player one-of other my-coplayers
  ;; compute difference in average payoffs
  let payoff-diff ([payoff / n-of-my-coplayers] of observed-player
              - (payoff / n-of-my-coplayers))
  set strategy-after-revision
      ifelse-value
      (random-float 1 < (payoff-diff / max-payoff-difference))
        [ [strategy] of observed-player ]
        [ strategy ]
        ;; If your strategy is the better, payoff-diff is negative,
```

```
        ;; so you are going to stick with it.
        ;; If it's not, you switch with probability
        ;; (payoff-diff / max-payoff-difference)
   end
```

Note that we are using a new variable, i.e. max-payoff-difference, to make sure that the agent changes strategy with *probability* proportional to the average payoff difference. Thus, we should define it as global (since this max-payoff difference will not change over the course of a run), as follows:

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
  max-payoff-difference      ;; <== New line
]
```

We also have to set the value of this new variable. We can do that at the end of the setup-payoffs method, as follows:

```
to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
  ;; new lines below
  set max-payoff-difference
      (max-of-matrix payoff-matrix) - (min-of-matrix payoff-matrix)
end
```

Finally, note that we have implemented two new procedures to compute the minimum and the maximum value of a matrix:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; SUPPORTING PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;
;;; Matrices ;;;
;;;;;;;;;;;;;;;;;

to-report max-of-matrix [matrix]
  report max reduce sentence matrix
end

to-report min-of-matrix [matrix]
```

```
    report min reduce sentence matrix
  end
```

To implement procedure to imitative-positive-proportional-m-rule, there is an extension that comes bundled with NetLogo which will make our life much easier: `extension rnd`. To use it, you just have to add the following line at the beginning of your code

```
extensions [rnd]
```

Having loaded the `rnd` extension, you can use primitive `rnd:weighted-one-of`, which will be very handy. With this information, you may want to try to implement this short procedure yourself.

Implementation of procedure to imitative-positive-proportional-m-rule

```
to imitative-positive-proportional-m-rule
  let chosen-nbr rnd:weighted-one-of my-nbrs-and-me [ payoff ^ m ]
  set strategy-after-revision [strategy] of chosen-nbr
end
```

If you wanted to use *average* rather than *total* payoffs, you would only have to divide the payoff by n-of-my-coplayers.

To avoid errors when payoffs are negative and this rule is used, it would be nice to check that payoffs are non-negative, and if they are, let the user know. We can do this implementing the following procedure:

```
to check-payoffs-are-non-negative
  if min reduce sentence payoff-matrix < 0 [
    user-message (word
      "Since you are using decision-rule =\n"
      "imitative-positive-proportional-m,\n"
      "all elements in the payoff matrix\n"
      payoffs
      "\nshould be non-negative numbers.")
  ]
end
```

An appropriate place to call this procedure would be at the end of procedure to setup-payoffs, which would then be as follows:

```
  to setup-payoffs
    set payoff-matrix read-from-string payoffs
    set n-of-strategies length payoff-matrix
    ;; New lines below
    set max-payoff-difference
        (max-of-matrix payoff-matrix) - (min-of-matrix payoff-matrix)
    if decision-rule = "imitative-positive-proportional-m"
        [ check-payoffs-are-non-negative ]
  end
```

The implementation of procedure to Fermi-m-rule is not extremely hard after having seen the implementation of to imitative-pairwise-difference-rule, and realizing that $\frac{e^{m\pi_j}}{e^{m\pi_j}+e^{m\pi_i}} = \frac{1}{1+e^{-m(\pi_j-\pi_i)}}$.

Implementation of procedure to Fermi-m-rule

A possible implementation of this procedure is as follows:

```
  to Fermi-m-rule
    let observed-player one-of other my-coplayers
    ;; compute difference in average payoffs
    let payoff-diff ([payoff / n-of-my-coplayers] of observed-player
                    - (payoff / n-of-my-coplayers))
    set strategy-after-revision
      ifelse-value (random-float 1 < (1 / (1 + exp (- m * payoff-diff))))
        [ [strategy] of observed-player ]
        [ strategy ]
  end
```

## 5.4. Complete code in the Code tab

The Code tab is ready! Congratulations! With this model you can rigorously explore the effect of space in 2-player games.

```
  extensions [rnd]            ;; <== New line

  globals [
    payoff-matrix
    n-of-strategies
```

```
  n-of-players
  max-payoff-difference  ;; <== New line
]

patches-own [
  strategy
  strategy-after-revision
  payoff
  my-nbrs-and-me
  my-coplayers
  n-of-my-coplayers
]

to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
  ask patches [update-color]
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
  ;; New lines below
  set max-payoff-difference
     (max-of-matrix payoff-matrix) - (min-of-matrix payoff-matrix)
  if decision-rule = "imitative-positive-proportional-m"
     [ check-payoffs-are-non-negative ]
end

to setup-players
  let initial-distribution read-from-string n-of-players-for-each-strategy
  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n" n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]
  if sum initial-distribution != count patches [
    user-message (word "The total number of agents in\n"
      "n-of-agents-for-each-strategy (i.e. "
      sum initial-distribution "):\n" n-of-players-for-each-strategy
      "\nshould be equal to the number of patches (i.e. "
      count patches ")"
    )
  ]
```

```
    ask patches [set strategy false]
    let i 0
    let offsets ifelse-value (neighborhood-type = "Von Neumann")
         [ von-neumann-offsets neighborhood-range self-matching? ]
         [ moore-offsets neighborhood-range self-matching? ]
    foreach initial-distribution [ j ->
      ask n-of j (patches with [strategy = false]) [
        set payoff 0
        set strategy i
        set strategy-after-revision strategy
        set my-coplayers patches at-points offsets
        set n-of-my-coplayers (count my-coplayers)
        set my-nbrs-and-me (patch-set my-coplayers self)
      ]
      set i (i + 1)
    ]
    set n-of-players count patches
  end

  to setup-graph
    set-current-plot "Strategy Distribution"
    foreach (range n-of-strategies) [ i ->
      create-temporary-plot-pen (word i)
      set-plot-pen-mode 1
      set-plot-pen-color 25 + 40 * i
    ]
  end

  to go
    ifelse synchronous-updating?
      [
        ask patches [ play ]
        ask patches [ update-strategy-after-revision ]
          ;; here we are not updating the agent's strategy yet
        ask patches [ update-strategy ]
          ;; now we update every agent's strategy at the same time
      ]
      [
        ask patches [
          play
          ask my-coplayers [ play ]
              ;; since your coplayers' strategies or
              ;; your coplayers' coplayers' strategies
              ;; could have changed since the last time
              ;; your coplayers played
          update-strategy-after-revision
          update-strategy
        ]
      ]
    tick
    update-graph
    ask patches [update-color]
  end
```

```
to play
  let n-of-coplayers-with-strategy-? n-values n-of-strategies [ i ->
    count my-coplayers with [strategy = i] ]
  let my-payoffs (item strategy payoff-matrix)
  set payoff sum (map * my-payoffs n-of-coplayers-with-strategy-?)
end

to update-strategy-after-revision
  ifelse (random-float 1 < noise)
    [ set strategy-after-revision random n-of-strategies ]
    [ (ifelse
        decision-rule = "best-neighbor"
                  [ best-neighbor-rule ]
        decision-rule = "imitate-if-better-all-nbrs"
                  [ imitate-if-better-all-nbrs-rule ]
        decision-rule = "imitative-pairwise-difference"
                  [ imitative-pairwise-difference-rule ]
        decision-rule = "imitative-positive-proportional-m"
                  [ imitative-positive-proportional-m-rule ]
        decision-rule = "Fermi-m"
                  [ Fermi-m-rule ]
      )
    ]
end

to best-neighbor-rule
  set strategy-after-revision
      [strategy] of one-of my-nbrs-and-me with-max [payoff]
end

to imitate-if-better-all-nbrs-rule
  let observed-player one-of other my-coplayers
  if (([payoff / n-of-my-coplayers] of observed-player) >
      (payoff / n-of-my-coplayers) [
    set strategy-after-revision ([strategy] of observed-player)
  ]
end

to imitative-pairwise-difference-rule
  let observed-player one-of other my-coplayers
  ;; compute difference in average payoffs
  let payoff-diff ([payoff / n-of-my-coplayers] of observed-player
                - (payoff / n-of-my-coplayers))
  set strategy-after-revision
      ifelse-value (random-float 1 < (payoff-diff / max-payoff-difference))
        [ [strategy] of observed-player ]
        [ strategy ]
      ;; If your strategy is the better, payoff-diff is negative,
      ;; so you are going to stick with it.
      ;; If it's not, you switch with probability
      ;; (payoff-diff / max-payoff-difference)
end
```

```
to imitative-positive-proportional-m-rule
  let chosen-nbr rnd:weighted-one-of my-nbrs-and-me [ payoff ^ m ]
   ;; https://ccl.northwestern.edu/netlogo/docs/rnd.html#rnd:weighted-one-of
  set strategy-after-revision [strategy] of chosen-nbr
end


to Fermi-m-rule
  let observed-player one-of other my-coplayers
  ;; compute difference in average payoffs
  let payoff-diff ([payoff / n-of-my-coplayers] of observed-player
                  - (payoff / n-of-my-coplayers))
  set strategy-after-revision
    ifelse-value (random-float 1 < (1 / (1 + exp (- m * payoff-diff))))
      [ [strategy] of observed-player ]
      [ strategy ]
end


to update-strategy
  set strategy strategy-after-revision
end


to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
    count patches with [strategy = n] / n-of-players ] strategy-numbers
  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end


to update-color
  set pcolor 25 + 40 * strategy
end


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; SUPPORTING PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to-report moore-offsets [r include-center?]
  let l (range (- r) (r + 1))
  let result reduce sentence map [ el1 -> map [el2 -> list el1 el2] l] l
  report ifelse-value include-center?
    [ result ]
    [ remove [0 0] result ]
end


to-report von-neumann-offsets [r include-center?]
```

```
    let moore-list (moore-offsets r include-center?)
    report filter [l -> abs first l + abs last l <= r] moore-list
end


to show-neighborhood
  if mouse-down? [
    ask patch mouse-xcor mouse-ycor [
      ask my-coplayers [set pcolor white]
    ]
  ]
  display
end


to check-payoffs-are-non-negative
  if min reduce sentence payoff-matrix < 0 [
    user-message (word
      "Since you are using decision-rule =\n"
      "imitative-positive-proportional-m,\n"
      "all elements in the payoff matrix\n"
      payoffs
      "\nshould be non-negative numbers.")
  ]
end


;;;;;;;;;;;;;;;;;
;;; Matrices ;;;
;;;;;;;;;;;;;;;;;

to-report max-of-matrix [matrix]
  report max reduce sentence matrix
end


to-report min-of-matrix [matrix]
  report min reduce sentence matrix
end
```

# 6. Sample runs

## 6.1. Decision rules

Now that we have implemented the model, we can explore the impact of the decision rule on the promotion of cooperation in the spatially embedded Prisoner's Dilemma. Let us explore this question using the parameter values shown in figure 1 above. We start with the imitate the best neighbor rule. The simulation below shows a representative run.

*One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=1292#video-1292-1*

As you can see in the video above, with the imitate the best neighbor rule, high levels of cooperation are achieved (i.e. about 90% of the patches cooperate in the long run). But, how robust is this result to changes in the decision rule? To explore this question, let us run the same simulation with the other four decision rules we have implemented. The simulation below shows a representative run with the imitate-if-better-all-nbrs rule.

> ▶️ *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=1292#video-1292-2*

As you can see in the video, with the imitate-if-better-all-nbrs rule there is no cooperation at all. Defection takes over in about forty ticks. Let us see what happens with the imitative-pairwise-difference rule.

> ▶️ *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=1292#video-1292-3*

With the imitative-pairwise-difference rule, again, there is no cooperation at all in the long run. Let us now try the imitative-positive-proportional-m rule with $m$ = 1:[12]

> ▶️ *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=1292#video-1292-4*

We do not observe any cooperation at all with the imitative-positive-proportional-m rule (with $m$ = 1) either. Finally, let us try the Fermi-m rule with $m$ = 1 (or any other positive value, for that matter):[13]

> ▶️ *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=1292#video-1292-5*

---

12. As long as $m \leq 5$, results are very similar, but for greater values of $m$ (i.e. greater intensity of selection) cooperation starts to emerge.
13. In contrast with what happens under the imitative-positive-proportional-m rule, parameter $m$ does not seem to have a great influence in this case. This changes if *self-matching?* is turned on. In that case, for high values of $m$, cooperation can emerge.

And we observe universal defection again! Clearly, as Roca et al. (2009a, 2009b) point out, the imitate the best neighbor rule seems to favor cooperation in the Prisoner's Dilemma more than other decision rules.

## 6.2. Neighborhoods

What about if we change the type of neighborhood? You can check that the simulations shown in the videos above do not change qualitatively if we use Von Neumann neighborhoods rather than Moore neighborhoods. However, what does make a difference is the size of the neighborhood. Note that if the size is so large that agents' neighborhoods encompass the whole population, we have a global interaction model (i.e. a model without population structure). Thus, the size of the neighborhood allows us to study the effect of population structure in a gradual manner. Having seen this, we can predict that, for the imitate the best neighbor rule, if we increase the size of the neighborhood enough, at some point cooperation will disappear. As an exercise, try to find the minimum *neighborhood-range* (for both types of neighborhood) at which cooperation cannot be sustained anymore, with the other parameter values as shown in figure 1.

Critical neighborhood size at which cooperation does not emerge in the simulation parameterized as in figure 1.

- *neighborhood-type* = "*Moore*". With *neighborhood-range* = 1, we observe significant levels of cooperation (about 90%) cooperation, but with *neighborhood-range* ≥ 2, we observe no cooperation at all.

- *neighborhood-type* = "*Von Neumann*". With *neighborhood-range* = 1, we observe significant levels of cooperation (about 75%) cooperation. Interestingly, with *neighborhood-range* = 2, we observe even more cooperation (usually more than 90%), but with *neighborhood-range* ≥ 3, we observe no cooperation at all.

## 6.3. Discussion

We would like to conclude this chapter emphasizing that the influence of population structure on evolutionary dynamics generally depends on many factors that may seem insignificant at first sight, and whose effects interact in complex ways. This may sound discouraging, since it suggests that a simple general theory of the influence of population structure on evolutionary dynamics cannot be derived. On a more positive note, it also highlights the importance of the skills you are learning with this book. Without the aid of computer simulation, it seems impossible to explore this type of questions. But using computer simulation we can gain some understanding of the complexity and the beauty of these apparently simple –yet surprisingly intricate– systems. The following quote from Roca et al. (2009b) nicely summarizes this view.

> To conclude, we must recognize the strong dependence on details of evolutionary games on spatial networks. As a consequence, it does not seem plausible to expect general laws that could be applied in a wide range of practical settings. On the contrary, a close modeling including the kind of game, the evolutionary dynamics and the population structure of the concrete problem seems mandatory to reach sound and compelling conclusions. With no doubt this is an enormous challenge, but we believe that this is one of the most promising paths that the community working in the field can explore. Roca et al. (2009b, pp. 14-15)

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-imitate-best-nbr-extended.nlogo.

**Exercise 1.** How can we parameterize our model to replicate the results shown in figures 1b and 1d of Hauert and Doebeli (2004)?

**Exercise 2.** How can we parameterize our model to replicate the results shown in figures 1, 2, 6 and 7 of Nowak et al. (1994a)?[14]

**Exercise 3.** How can we parameterize our model to replicate the results shown in figure 1 of Szabó and Tőke (1998)? Note that Szabó and Tőke (1998) use *total* payoffs and we use *average* payoffs in our Fermi-m rule, but a clever parameterization can save you any programming efforts.



*A frame of the simulation shown in figure 11 of Nowak and May (1993)*

**Exercise 4.** How can we parameterize our model to replicate the results shown in figure 11 of Nowak and May (1993)?

**Exercise 5.** To appreciate the impact that *neighborhood-type* may have, run a simulation parameterized as in figure 1 above, but with *decision-rule* = "Fermi-m", *m* = 10, and *self-matching?* = true. Compare the results obtained using *neighborhood-type* = "Moore" and *neighborhood-type* = "Von Neumann". What do you observe?

**Exercise 6.** The following block of code (in procedure to update-strategy-after-revision) can be

---

14. Figures 1 and 2 in Nowak et al. (1994a) are the same as figures 1 and 2 in Nowak et al. (1994b).

replaced by one simple line using `run` (a primitive that can take a string containing the name of a command as an input, and it runs the command).

```
[ (ifelse
    decision-rule = "best-neighbor"
              [ best-neighbor-rule ]
    decision-rule = "imitate-if-better-all-nbrs"
              [ imitate-if-better-all-nbrs-rule ]
    decision-rule = "imitative-pairwise-difference"
              [ imitative-pairwise-difference-rule ]
    decision-rule = "imitative-positive-proportional-m"
              [ imitative-positive-proportional-m-rule ]
    decision-rule = "Fermi-m"
              [ Fermi-m-rule ]
  )
]
```

Can you come up with the simple line?

# III-5. Analysis of these models

The computable need not be predictable. Sigmund (1983, p. 54)

# 1. A much greater state space

Like most agent-based models, all the models developed in this Part III can be usefully seen as time-homogeneous Markov chains. Note, however, that in order to interpret them as Markov chains, the definition of the state of the system must be different from the one we used in Part II.

Recall that in Part II (where we studied unstructured populations), we could define the state of the system as the strategy distribution in the population, i.e. the number of agents that are using each possible strategy. This was all the information we needed about the present –and the past– of the stochastic process to be able to (probabilistically) predict its future as accurately as it is possible. By contrast, in grid models this information is not enough.

As an illustration, consider the following (deterministic) simulation runs, conducted with the first model developed in this Part III, using payoffs *DD-payoff* = *CD-payoff* = 0, *CC-payoff* = 1 and *DC-payoff* = 1.15, and periodic boundary conditions (see video 1 below). The three runs start with exactly four cooperators. The cooperators are arranged (a) in a square, (b) in L, and (c) in a line. As you can see, the three simulations lead to completely different dynamics: run (a) goes to universal cooperation, run (b) produces a glider of cooperators that moves indefinitely, and run (c) goes to universal defection in one tick.

| a) Four cooperators in a square | b) Four cooperators in L | c) Four cooperators in a line |
|---|---|---|
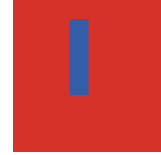
*Video 1: Simulations of model 2×2-imitate-best-nbr on a 80×80 torus, using payoffs DD-payoff = CD-payoff = 0, CC-payoff = 1 and DC-payoff = 1.15. Initially, there are only four cooperators arranged (a) in a square, (b) in L, and (c) in a line.*

Video 1 clearly illustrates that, in grid models, simply knowing how many agents are using each strategy is not enough to predict how the system will evolve. In general, we need to know the strategy of each individual agent (i.e. patch). Therefore, the size of the state space in grid models is much larger than in their well-mixed population counterparts. Let us see exactly how much. Recall that the number of possible states in the well-mixed population models implemented in the Part II was $\binom{N+s-1}{s-1}$, where $N$ is the number of agents and $s$ the number of strategies (see section II-5.3). In contrast, the number of possible states in the grid models implemented in this Part III is generally $s^N$. This is a major difference. For example, if $N = 100$ and $s = 2$, the size of the state space in the well-mixed population model would be $\binom{100+2-1}{2-1} = 101$, while the size of the state space in the grid model would be $2^{100} \approx 1.26765 * 10^{30}$.

Some of the models developed in this Part III can be seen as a particular class of Markov chains called cellular automata, so we deal with them separately in the following section.

# 2. Cellular automata

Cellular automata (CA) are discrete-time models that consist of a regular lattice of cells. In principle, the lattice can be of any finite number of dimensions. For instance, video 2 below shows a 2-dimensional CA that is called Conway's Game of Life.

*Video 2: Simulation of Conway's Game of Life on a 50×50 torus.*

In CA, each cell has a set of neighboring cells, defined by a certain neighborhood function (e.g. the neighborhood in the Game of Life is the Moore neighborhood). At any given tick, each cell is in one state out of a finite set (e.g., black or white in video 2 above). The state of each cell in the following tick is *deterministically* determined by the state of its neighbors and its own state in the current tick; and the function that updates the state of each cell is applied *synchronously* to every cell. As an example, the function that updates the cells' state in Conway's Game of Life reads as follows:

A white cell turns black if it has exactly three black neighbors, and a black cell stays black only if two or three of its neighbors are black. All other cells go white.

As you can see in video 2 above, cellular automata can produce astonishingly complex patterns from extremely simple deterministic rules.[1] This is true even for *elementary cellular automata*, which are 1-dimensional, two-state cellular automata where the neighborhood of each cell is just the two cells at either side (Wolfram, 1983, 2002). Amazingly, some elementary cellular automata, like rule 30, display chaotic behavior, and others, like rule 110, are even capable of universal computation (Cook, 2004).[2]

Going back to our models, note that all *synchronous* models implemented in this Part III that use a *deterministic* decision rule fit the definition of 2-dimensional cellular automata. In particular, the first model we implemented in this Part III (Nowak and May, 1992, 1993) is one of them.[3,4]

---

1. You can run your own simulations of the Game of Life using the following Mathematica script:

```
globalState = RandomInteger[1, {50, 50}];
Dynamic[ArrayPlot[
  globalState = Last[CellularAutomaton["GameOfLife", globalState, 1]]
]]
```

2. Two useful references to learn about the fascinating world of cellular automata are Berto and Tagliabue (2023) and Wolfram (2002). Sigmund (1983, chapter 2) and Mitchell (1998) are two wonderful thought-provoking gems, highly recommended for anyone interested in appreciating the important role of cellular automata in understanding self-reproduction and universal computation.

3. Note, however, that the neighborhood in the definition of the corresponding cellular automaton is not the (Moore) neighborhood used in the model. In the model, the strategy of a player depends on the *payoffs* of its Moore neighbors, which in turn depend on the *strategy* of the player's neighbors' neighbors. Thus, the neighborhood of the corresponding cellular automaton is the Moore neighborhood of range 2, i.e., the 5x5 square around each patch.

4. Szabó and Fáth (2007, sections 4.1 and 6.5) discuss this model in detail and review several other CA models in the context of evolutionary game theory.

In general, CA are surprisingly difficult to analyze. Even though they are completely deterministic, in many cases there seems to be no shorter way of predicting the state of the CA after *n* ticks than by actually running the whole simulation, i.e. there is no shortcut or possible theory that can save us from having to compute every single step from tick 0 to tick *n*.[5] This may come as a surprise, especially for CA defined on *finite* lattices like all the CA developed in this book. After all, if a CA is run on a finite grid, it is clear that, sooner or later, it must end up in a cycle of finite length (including single states, which can be seen as cycles of length 1). However, it is often the case that CA:

- have a huge number of possible end cycles,

- the length of the end cycles can vary substantially,

- the specific cycle where a simulation ends up may be very sensitive to initial conditions and to small changes in the model (such as the size and the topology of the grid) and,

- the number of ticks required to enter the end cycle may also be very sensitive to initial conditions and to small changes in the model.

The simulation runs displayed in Video 1 are a clear illustration of these points. Simulations (a) and (c) differ from (b) only in the location of one single cooperator at the beginning of the simulation. Nonetheless, the final state of run (a) is universal cooperation, the final state of run (c) is universal defection, and run (b) ends up in a cycle of length 80. In short, a change on initial conditions as small as it can be may lead to a change in the dynamics as large as it can be. You may want to run this model with different initial conditions to appreciate the great number of possible end cycles it can get to, and its sensitivity to initial conditions.

As another example, consider the Game of Life simulation displayed in video 2 above. Video 3 below shows a simulation run with the same initial condition except for the top leftmost cell, which was white in video 2 and is black in video 3 below.

> [video icon] *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=114#video-114-5*

*Video 3: Simulation of Conway's Game of Life on a 50×50 torus. The initial condition differs from the initial condition used in video 2 only in the cell at the top left.*

The simulation in video 2 reaches a 200-cycle (i.e., a cycle of length 200) after 467 ticks, while the simulation in video 3 reaches a completely different 2-cycle after 2083 ticks.

Besides initial conditions, there are other features we can change in CA, which may seem innocuous at first sight, but which may significantly affect their dynamics. To see this, consider the kaleidoscopic simulation we ran in section III-1.6. The simulation starts with one single defector in a sea of

---

5. Wolfram (2002, p. 739) calls this feature "computational irreducibility". However, to our knowledge, there is not a unique and uncontested formal definition of this concept.

cooperators. The run seems to go on forever but, given that the grid is finite, we know that the simulation will necessarily end up in a cycle. Table 1 below shows the tick at which the simulation enters its end cycle, together with the length of the end cycle and its average number of cooperators, for different grid sizes.[6] As you can see, changing slightly the size of the grid can also change the dynamics substantially (Nowak and May, 1993).

| Grid size | Tick at which the run enters the end cycle | Length of cycle | Average number of cooperators in the end cycle |
|---|---:|---:|---:|
| 49×49 | 1 140 | 1 | 0 (0%) |
| 51×51 | 737 | 2 | (2304+2500)/2 = 2402 (92.35%) |
| 53×53 | 331 | 1 | 0 (0%) |
| 55×55 | 752 | 1 | 0 (0%) |
| 57×57 | 4 770 | 2 | (2916+3136)/2 = 3026 (93.14%) |
| 59×59 | 33 | 1 | 0 (0%) |
| 61×61 | 3 900 | 4 | (48+48+48+48)/4 = 48 (1.29%) |
| 63×63 | 33 | 1 | 0 (0%) |
| 65×65 | 33 | 1 | 0 (0%) |
| 67×67 | 33 | 1 | 0 (0%) |
| 69×69 | 56 723 | 1 | 0 (0%) |
| 71×71 | 163 982 | 1 | 0 (0%) |
| 73×73 | 151 292 | 1 | 0 (0%) |
| 75×75 | 171 429 | 1 | 0 (0%) |
| 77×77 | 374 805 | 1 | 0 (0%) |
| 79×79 | 1 053 418 | 4 | (48+48+48+48)/4 = 48 (0.77%) |
| 81×81 | 1 547 640 | 1 | 0 (0%) |

*Table 1. End cycles of model 2×2-imitate-best-nbr run on a grid with fixed boundaries (box), for different grid sizes, using payoffs DD-payoff = CD-payoff = 0, CC-payoff = 1 and DC-payoff = 1.85. Initially, there is one single defector in the center of the grid. All runs for grids 1×1 to 47×47 end up in universal defection in less than 200 ticks*

Finally, let us see the impact of changing the boundary conditions, again in the kaleidoscopic setup. Table 2 below shows that changes in boundary conditions can impact both the type of cycle we end up, and also the time it takes to reach it, in ways that do not seem easy to grasp.

---

6. This is a slightly extended version of table 1 in Nowak and May (1993, p. 46).
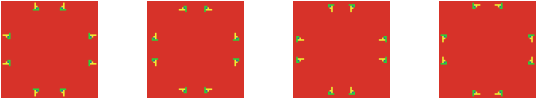
| Grid size | Fixed boundaries[7] (Box) | | Periodic boundaries (Torus) | |
|---|---|---|---|---|
| 43×43 | Universal defection from tick 111  | | 2-cycle from tick 324, with an average of 90.97% cooperators  | |
| 45×45 | Universal defection from tick 180 | | Universal defection from tick 939 | |
| 47×47 | Universal defection from tick 143 | | Universal defection from tick 8030 | |
| 49×49 | Universal defection from tick 1140 | | Universal defection from tick 430 | |
| 51×51 | 2-cycle from tick 737, with an average of 92.35% cooperators  | | Universal defection from tick 16,446  | |
| 53×53 | Universal defection from tick 331 | | Universal defection from tick 4853 | |
| 55×55 | Universal defection from tick 752 | | Universal defection from tick 14,946 | |
| 57×57 | 2-cycle from tick 4770, with an average of 93.14% cooperators  | | Universal defection from tick 3329  | |
| 59×59 | Universal defection from tick 33  | | 4-cycle from tick 45,406, with an average of 1.38% cooperators  | |
| 61×61 | 4-cycle from tick 3900, with an average of 1.29% cooperators  | | 4-cycle from tick 14,007, with an average of 1.29% cooperators  | |
| 63×63 | Universal defection from tick 33 | | Universal defection from tick 33 | |
| 65×65 | Universal defection from tick 33 | | Universal defection from tick 33 | |
| 67×67 | Universal defection from[7] tick 33 | | Universal defection from tick 286,379 | |

*Table 2. End cycles of model 2×2-imitate-best-nbr run on a grid, using payoffs DD-payoff = CD-payoff = 0, CC-payoff = 1 and DC-payoff = 1.85, for different grid sizes and boundary conditions (box vs. torus). Initially, there is one single defector in the center of the grid*

The previous examples illustrate how difficult it is to analyze CA, even when they are run on finite lattices. It seems that predicting their behavior by means of deductive mathematical analysis is only possible for particularly simple cases; in general, we have to resort to computer simulation. Bhattacharjee et al. (2020) review various tools to characterize CA and state the following:

> *In this section, we survey the characterization tools and parameters, developed till date to analyze the CAs. Tools are mainly developed for one-dimensional CAs, and for two or higher dimensional CAs, "run and watch" is the primary technique to study the behavior.* Bhattacharjee et al. (2020, p. 439)

To conclude this section on CA, let us emphasize that the dynamics displayed by many CA are also very sensitive to noise. Oftentimes, adding just a bit of stochasticity (e.g. by using random asynchronous updating rather than synchronous updating) drastically changes the generated patterns. See e.g. sample runs in chapter III-2. In such cases, we should be very careful about the conclusions we can draw from those CA. Our personal view is that deterministic dynamical patterns that are not robust to low levels of stochasticity may be interesting from a theoretical (or even aesthetical) point of view, but its usefulness as models of real-world systems is probably doubtful.

Interestingly, adding a little bit of stochasticity to CA, besides making them more realistic, often improves their mathematical tractability. As we saw in section II-5.3, stochasticity often turns reducible Markov chains into irreducible and aperiodic ones (sometimes called ergodic), which are more amenable to exploration and analysis, and their asymptotic dynamics do not depend on initial conditions.[8]

# 3. Models more amenable to mathematical analysis. The pair approximation

In chapter II-5, we saw that the dynamics of all the models developed in Part II could be approximated impressively well using various mathematical techniques, at least within certain ranges of parameter values (e.g. large populations and low revision probabilities). The main reason we could develop useful approximations for those models is that:

- the state of the system could be defined using a small set of continuous and bounded variables (i.e., the fraction of agents that are using each strategy), and

---

7. We use the term "fixed boundaries" to be consistent with Nowak and May's (1992, 1993) original papers, but other authors, such as Sayama (2015, p. 189), refer to this topology as "cut-off boundaries", and use the term "fixed boundaries" for a different topology.
8. In section II-5.3.1.3 we provided sufficient conditions for irreducibility and aperiodicity of time-homogeneous Markov chains.

- these variables evolved slowly (i.e., their values could change only slightly in each tick).

The spatial models we have developed in this book do not generally satisfy the two conditions above. This is specially evident for those that are cellular automata. In spatial models, the proportion of agents that are using each strategy is certainly not enough to define the state of the system (see, e.g., video 1). Thus, a possible move forward consists in adding to our approximations some extra variables that can capture some of the missing information that is necessary to predict how the system will evolve. Specifically, in grid models, the strategies of your neighbors are very relevant, so an additional set of variables that are certainly useful is the set of conditional probabilities that a neighbor of an *i*-player is a *j*-player. This is the approach followed by the so-called "Pair approximation", which we introduce in the next section.

## 3.1. Introduction to the pair approximation

The pair approximation (Nakamaru et al. (1997), Rand (1999), van Baalen (2000)) can be interpreted as an extension of the mean dynamic (section II-5.3.2.1) developed to be useful for structured populations. The goal of both techniques is to derive a system of (deterministic) differential equations that can approximate our model's stochastic dynamics, and both techniques are based on computing the *expected change of the stochastic process*.

To derive this expected change in unstructured populations, the only information we need is the distribution of strategies in the population, since this information is enough to define the state of the stochastic process. In unstructured populations, the fraction of agents with strategy *i*, i.e. the fraction of *i*-players, is also the probability that the next interaction of any agent will be with an *i*-player; and this is true *for every agent*. Thus, these fractions were the only variables we used in the mean dynamic.

By contrast, in structured populations, the probability of interacting with an *i*-player is not the fraction of *i*-players in the whole population; instead, this probability depends on the particular distribution of strategies in the agent's local neighborhood. Thus, in structured populations we need more information. The approach followed in the pair approximation is to include the conditional probabilities that a neighbor of an *i*-player is a *j*-player. These conditional probabilities, together with the distribution of strategies in the population, are the variables that are used in the pair approximation to estimate the expected evolution of the stochastic dynamics. These variables are by no means sufficient to compute the expected change in the state of the system exactly, but they can sometimes provide useful approximations.

## 3.2. Derivation of a pair approximation for regular undirected networks

In this section we derive a simple pair approximation for the dynamics of 2-player 2-strategy games in *regular undirected networks*, under different decision rules.
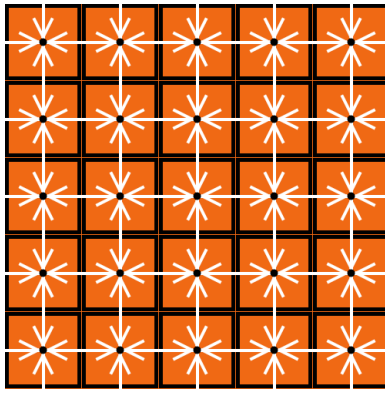
(a) *Von Neumann neighborhoods on a torus*    (b) *Moore neighborhoods on a torus*
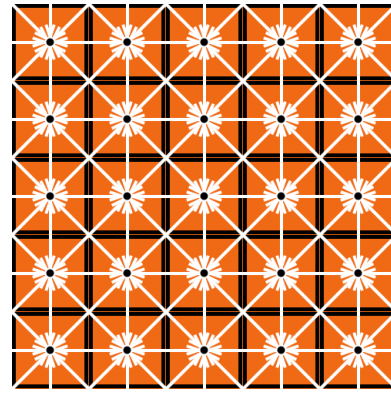
*Figure 1. The figure shows the regular networks that correspond to a grid with periodic boundaries using (a) Von Neumann neighborhoods and (b) Moore neighborhoods. There is a white link between any two patches that are neighbors*

A network is a set of nodes and a set of links. Nodes represent the agents and links connect agents that are neighbors. Figure 1 shows, in white, the links corresponding to a 2-dimensional grid with toroidal topology, using Von Neumann neighborhoods (fig. 1(a)) and using Moore neighborhoods (fig. 1(b)).

Here we consider *regular undirected networks* only. *Undirected* networks are those where the neighborhood relationship is symmetric, i.e., if agent *a* is a neighbor of agent *b*, then agent *b* is also a neighbor of agent *a*. *Regular* networks are networks where every node has the same degree (i.e., the same number of neighbors). Thus, the spatial grids we have considered here, with periodic boundaries, fit the definition of *regular undirected networks*.[9]

For simplicity, here we consider games with two strategies only. Let us denote them $A$ and $B$. Our goal is to derive a system of differential equations for the (joint) evolution of:

- the fraction of agents using strategy $A$, and

- the conditional probability that a neighbor of an $A$-player is an $A$-player.

The following section explains the notation we will use.

### 3.2.1. Notation

*Notation for nodes (i.e., agents)*

- $N$ is the number of agents.

- $k$ is the number of neighbors (or degree) of every agent.

- $[A]$ is the number of agents with strategy $A$, and $[B]$ is the number of agents with strategy $B$. Thus, $[B] = N - [A]$.

---

9. Hauert and Szabó (2005, section B) study dynamics on different regular undirected networks besides lattices, such as random regular graphs and regular small world networks.

- $x_A = \frac{[A]}{N}$ is the fraction of players using strategy $A$. Similarly, $x_B = \frac{[B]}{N}$. Therefore, $x_B = 1 - x_A$.

*Notation for links*

We follow the convention of counting links between agents as if they were directed, i.e., if an $A$-player and a $B$-player are neighbors, we count two links: one $A$–$B$ link and one $B$–$A$ link.[10] Thus, the total number of links is $kN$.

- $[AB]$ denotes the number of links from $A$-players to $B$-players. We define $[AA]$, $[BA]$ and $[BB]$ analogously. Thus, we have $[AA] + [AB] + [BA] + [BB] = kN$ and $[BA] = [AB]$.

- $x_{AB} = \frac{[AB]}{kN}$ denotes the fraction of links from $A$-players to $B$-players. We define $x_{AA}$, $x_{BA}$ and $x_{BB}$ analogously. Thus, we have $x_{AA} + x_{AB} + x_{BA} + x_{BB} = 1$ and $x_{AB} = x_{BA}$.

- $q_{A/A} = \frac{[AA]}{[AA]+[AB]} = \frac{[AA]}{k[A]} = \frac{x_{AA}}{x_A}$ denotes the fraction of neighbors of $A$-players that are $A$-players themselves. This is also the probability that a (random) neighbor of a (random) $A$-player is an $A$-player. $q_{B/A} = \frac{[AB]}{[AA]+[AB]} = \frac{x_{AB}}{x_A} = 1 - q_{A/A}$ is the probability that a random neighbor of an $A$-player is a $B$-player. We have equivalent definitions for $q_{A/B} = \frac{x_{AB}}{x_B} = \frac{x_A}{1-x_A}(1 - q_{A/A})$ and for $q_{B/B} = 1 - q_{A/B}$.

Importantly, note that all the considered variables $x_X$, $x_{XY}$ and $q_{X/Y}$ can be expressed as functions of $x_A$ and $q_{A/A}$.

## 3.2.2. Neighborhood configurations

In this section we derive the probability of different neighborhood configurations. We assume that the model is such that the only relevant information about the neighborhood is its distribution of strategies. With two strategies, this distribution can be characterized by the number of $A$-neighbors, $k_A$, with $0 \leq k_A \leq k$.

For an $A$-player, the probability of being in a neighborhood with $k_A$ $A$-neighbors can be approximated by the binomial distribution:

$$P_A(k_A) = \binom{k}{k_A} q_{A/A}^{k_A} (1 - q_{A/A})^{k-k_A}$$

And, for a $B$-player, the probability of being in a neighborhood with $k_A$ $A$-neighbors can be approximated by:

$$P_B(k_A) = \binom{k}{k_A} q_{A/B}^{k_A} (1 - q_{A/B})^{k-k_A}$$

It will also be useful to refine these probabilities for the cases where we already know that there is a player in the neighborhood using a strategy different from the focal agent's strategy. That is, if we

---

10. This convention is convenient for the formulas, but there are alternatives that are also valid.

already know that (at least) one of the neighbors of a focal $A$-player is a $B$-player, the conditional probability of the neighborhood configuration with $k_A$ $A$-players for the focal $A$-player can be approximated, for $k_A \leq k - 1$, by

$$P_{A,\mathrm{One}B}(k_A) = \binom{k-1}{k_A} q_{A/A}^{k_A} (1 - q_{A/A})^{k-1-k_A}$$

And, equivalently, for a $B$-player with at least one $A$-neighbor, the conditional probabilities, for $k_A \geq 1$, are given by

$$P_{B,\mathrm{One}A}(k_A) = \binom{k-1}{k_A-1} q_{A/B}^{k_A-1} (1 - q_{A/B})^{k-k_A}$$

Importantly, note that, given that $q_{A/B} = \frac{x_A}{1-x_A}(1 - q_{A/A})$, all the formulas in this subsection can also be expressed in terms of $x_A$ and $q_{A/A}$.

### 3.2.3. Derivation of the first equation: $\dot{x}_A = f_1(x_A, q_{A/A})$

As indicated before, our goal is to derive a system of differential equations for i) the fraction of agents using strategy $A$ and ii) the conditional probability that a neighbor of an $A$-player is an $A$-player, i.e.,

$$\dot{x}_A = f_1(x_A, q_{A/A})$$
$$\dot{q}_{A/A} = f_2(x_A, q_{A/A})$$

To derive the first equation, we follow the same approach we used to derive the mean dynamic (section II-5.3.2.1). We define one unit of clock time as the time over which every agent is expected to receive exactly one revision opportunity. Thus, over the time interval $dt$, the number of agents who are expected to receive a revision opportunity is $N dt$. Let $\mathbb{P}(A \rightarrow B)$ be the probability that a revising $A$-player switches to strategy $B$. Therefore, of the $N dt$ agents who revise their strategies, $x_A \mathbb{P}(A \rightarrow B) N dt$ are expected to switch from strategy $A$ to strategy $B$. Similarly, $x_B \mathbb{P}(B \rightarrow A) N dt$ agents are expected to switch from strategy $B$ to strategy $A$. Hence, the expected change in the number of agents that are using strategy $A$ over the time interval $dt$ is $(x_B \mathbb{P}(B \rightarrow A) - x_A \mathbb{P}(A \rightarrow B)) N dt$. Therefore, the expected change in the fraction of agents using strategy $A$ is:

$$dx_A = \tfrac{1}{N} \left( x_B \mathbb{P}(B \rightarrow A) - x_A \mathbb{P}(A \rightarrow B) \right) N dt = \left( x_B \mathbb{P}(B \rightarrow A) - x_A \mathbb{P}(A \rightarrow B) \right) dt$$

Thus,

$$\dot{x}_A = x_B \mathbb{P}(B \rightarrow A) - x_A \mathbb{P}(A \rightarrow B)$$

Note, however, that the probabilities in the formula above depend on the neighborhood configuration of the focal agent. Thus, we should consider all the possible neighborhood configurations and their corresponding probabilities, as follows:

$$\dot{x}_A = x_B \sum_{k_A=0}^{k} P_B(k_A) F_B(k_A) - x_A \sum_{k_A=0}^{k} P_A(k_A) F_A(k_A)$$

where $F_B(k_A)$ is the (conditional) probability that a revising $B$-player with $k_A$ $A$-neighbors switches

to strategy $A$, and $F_A(k_A)$ is the (conditional) probability that a revising $A$-player with $k_A$ $A$-neighbors switches to strategy $B$.

*Conditional switching probabilities $F_B(k_A)$ and $F_A(k_A)$*

Conditional switching probabilities depend on the number of $A$-neighbors $k_A$ and on the decision rule. Regarding the decision rule, here we focus on the imitative-pairwise-difference rule; we leave the derivation of the switching probabilities for other decision rules as exercises at the end of this chapter.

Under the imitative-pairwise-difference rule, the revising agent looks at a random neighbor and, if the observed neighbor has a greater average payoff, the reviser copies her strategy with probability proportional to the difference in average payoffs. Thus, let us start defining the payoff functions.

Consider first a revising $B$-player with $k_A$ $A$-neighbors. The average payoff to such a revising $B$-player is $\pi_B(k_A) = (k_A \pi_{BA} + (k - k_A)\pi_{BB})/k$.[11] The payoff to an $A$-neighbor of the revising $B$-player depends on the number $k'_A \leq k - 1$ of $A$-neighbors of this $A$-player (who has at least one $B$-neighbor), and this payoff is $\pi_A(k'_A) = (k'_A \pi_{AA} + (k - k'_A)\pi_{AB})/k$.

We can now derive $F_B(k_A)$, which is the probability that a revising $B$-player switches to strategy $A$. The probability that a random neighbor of a revising $B$-player with $k_A$ $A$-neighbors is an $A$-player is $\frac{k_A}{k}$. The payoff of this random $A$-neighbor depends on the number $k'_A$ of $A$-neighbors she has, with $0 \leq k'_A \leq k - 1$ (since we know that she has at least one $B$-neighbor, i.e. the reviser). The probability that the observed $A$-neighbor has $k'_A$ $A$-neighbors is $P_{A,\mathrm{OneB}}(k'_A)$. Therefore, the formula for $F_B(k_A)$ reads:

$$F_B(k_A) = \frac{k_A}{k} \sum_{k'_A=0}^{k-1} P_{A,\mathrm{OneB}}(k'_A) \frac{[\pi_A(k'_A) - \pi_B(k_A)]^+}{\Delta},$$

where $[x]^+ = \max(x, 0)$ and $\Delta > 0$ is a sufficiently large constant to guarantee that $\frac{[\pi_A(k'_A) - \pi_B(k_A)]^+}{\Delta} \leq 1$ in every possible case.[12]

Following the same reasoning, we can derive $F_A(k_A)$:

$$F_A(k_A) = \frac{k - k_A}{k} \sum_{k'_A=1}^{k} P_{B,\mathrm{OneA}}(k'_A) \frac{[\pi_B(k'_A) - \pi_A(k_A)]^+}{\Delta}$$

*Substitution*

By now, we have the formulas for every term in the equation:

$$\dot{x}_A = x_B \sum_{k_A=0}^{k} P_B(k_A) F_B(k_A) - x_A \sum_{k_A=0}^{k} P_A(k_A) F_A(k_A)$$

---

11. For decision rules that depend on total payoffs, rather than on average payoffs, we would just multiply by $k$.

12. The value of $\Delta$ should be at least $\left(\max(\pi_{AA}, \pi_{AB}, \pi_{BA}, \pi_{BB}) - \min(\pi_{AA}, \pi_{AB}, \pi_{BA}, \pi_{BB})\right)$; choosing any value greater than this will only imply a change of speed in the differential equations.

It only remains to express all these terms as functions of $x_A$ and $q_{A/A}$ to obtain the first equation of the pair approximation:

$$\dot{x}_A = f_1(x_A, q_{A/A})$$

### 3.2.4. Derivation of the second equation: $\dot{q}_{A/A} = f_2(x_A, q_{A/A})$

Recall that $q_{A/A} = \frac{x_{AA}}{x_A}$. Therefore,

$$\dot{q}_{A/A} = \frac{\dot{x}_{AA}}{x_A} - \frac{x_{AA}}{x_A^2}\dot{x}_A$$

Since we already have the formula for $\dot{x}_A = f_1(x_A, q_{A/A})$, we just need to derive $\dot{x}_{AA}$.

Remember that $x_{AA} = \frac{[AA]}{kN}$. Just like we did for the first equation, here we also consider the *events* that modify $[AA]$, taking into account all the possible neighborhood configurations with their corresponding probabilities. The expected change in the number of $A$–$A$ links $[AA]$ over the time interval $dt$ is:

$$d[AA] = \left(x_B \sum_{k_A=0}^{k} P_B(k_A)F_B(k_A)L_B^+(k_A) - x_A \sum_{k_A=0}^{k} P_A(k_A)F_A(k_A)L_A^-(k_A)\right)Ndt,$$

where $L_B^+(k_A) = 2k_A$ is the number of new $A$–$A$ links that appear when a $B$-player with $k_A$ $A$-neighbors switches to strategy $A$, and where $L_A^-(k_A) = 2k_A$ is the number of $A$–$A$ links that disappear when an $A$-player (with $k_A$ $A$-neighbors) adopts strategy $B$. Therefore, the expected change in $x_{AA}$ is:

$$dx_{AA} = \frac{d[AA]}{kN} = \frac{1}{k}\left(x_B \sum_{k_A=0}^{k} P_B(k_A)F_B(k_A)L_B^+(k_A) - x_A \sum_{k_A=0}^{k} P_A(k_A)F_A(k_A)L_A^-(k_A)\right)dt$$

Thus,

$$\dot{x}_{AA} = \frac{1}{k}\left(x_B \sum_{k_A=0}^{k} P_B(k_A)F_B(k_A)L_B^+(k_A) - x_A \sum_{k_A=0}^{k} P_A(k_A)F_A(k_A)L_A^-(k_A)\right)$$

Expressing the different terms as functions of $x_A$ and $q_{A/A}$, we obtain an equation of the form $\dot{x}_{AA} = f_3(x_A, q_{A/A})$. Finally, from $\dot{q}_{A/A} = \frac{\dot{x}_{AA}}{x_A} - \frac{x_{AA}}{x_A^2}\dot{x}_A$, we obtain the second equation of the pair approximation $\dot{q}_{A/A} = f_2(x_A, q_{A/A})$.

## 3.3. Solving the pair approximation. Examples

Just like we did with the mean dynamics, we can numerically solve the system of differential equations of the pair approximation. To this end, we have created a *Mathematica*® notebook that you can download and run using the free Wolfram Player. Figure 2 shows its interface.
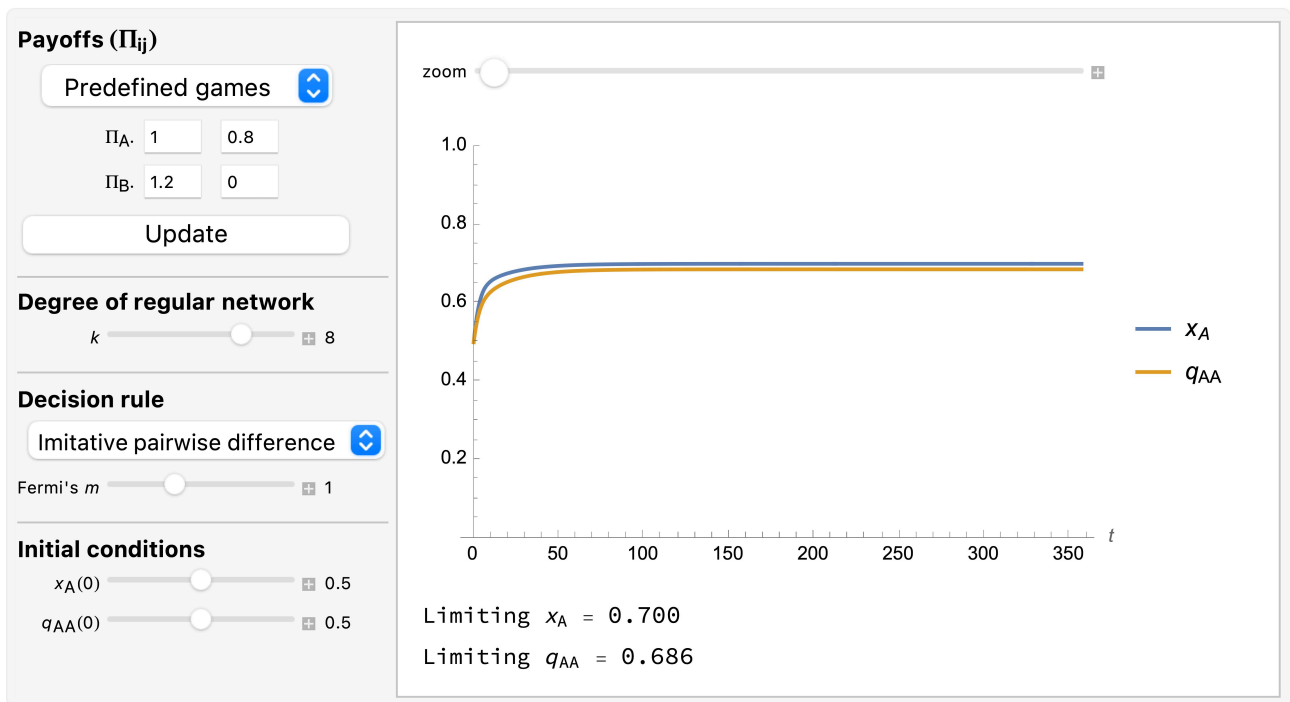
*Figure 2. Interface of the Mathematica® notebook created to solve the system of differential equations of the pair approximation (Izquierdo et al., 2024). You can download the notebook at https://github.com/ luis-r-izquierdo/pair-approximation*

As an example of a case where the pair approximation works well, we present some results for the snowdrift game (payoffs = [[1 0.8][1.2 0]]) played synchronously on a 100×100 torus using Moore neighborhoods ($k = 8$) and the imitative-pairwise-difference rule. Initial conditions are random, with half the population using strategy $A$ and the other half using strategy $B$, so $x_A(0) = 0.5$ and $q_{A/A}(0) \approx 0.5$. Figure 3 shows the solution trajectory of the pair approximation (see also Figure 2), together with the average, minimum and maximum values of the fraction of $A$-strategists at every tick, across 1000 simulation runs conducted with our nxn-imitate-best-nbr-extended model.
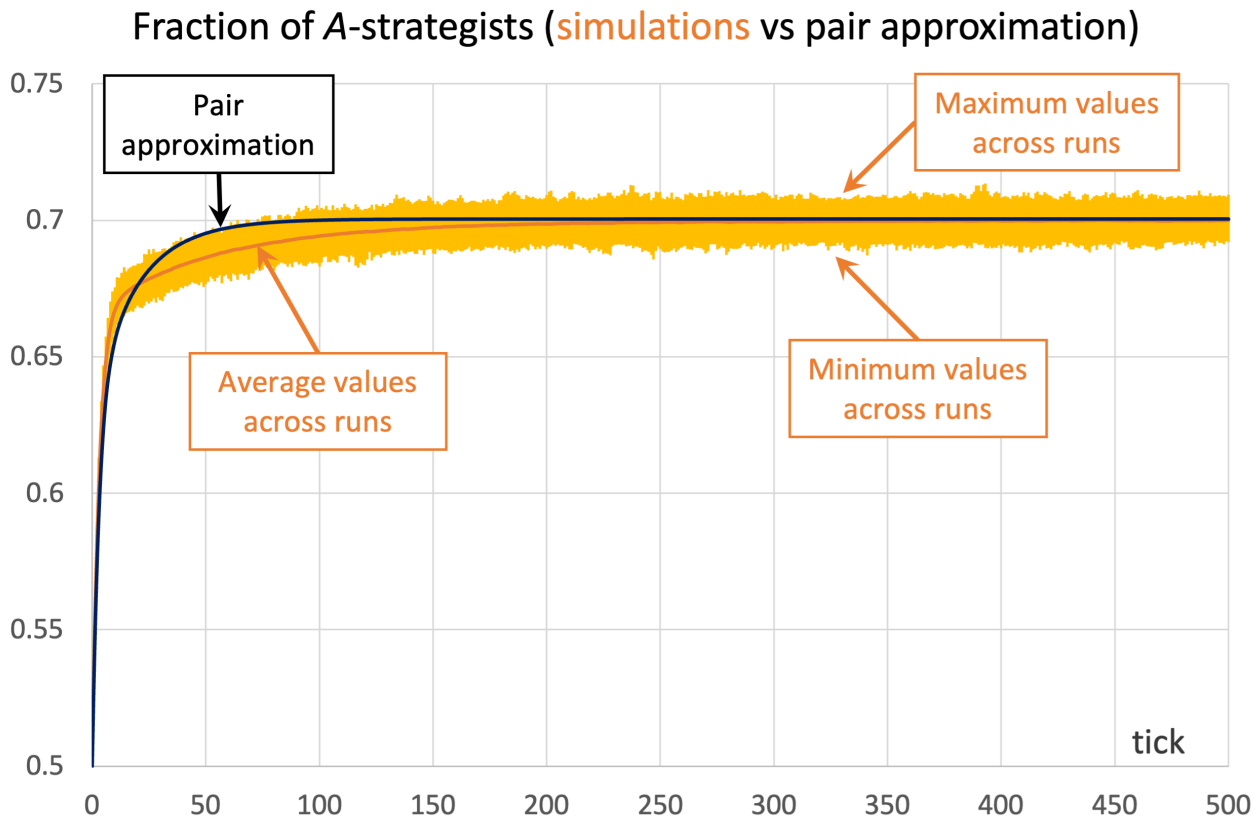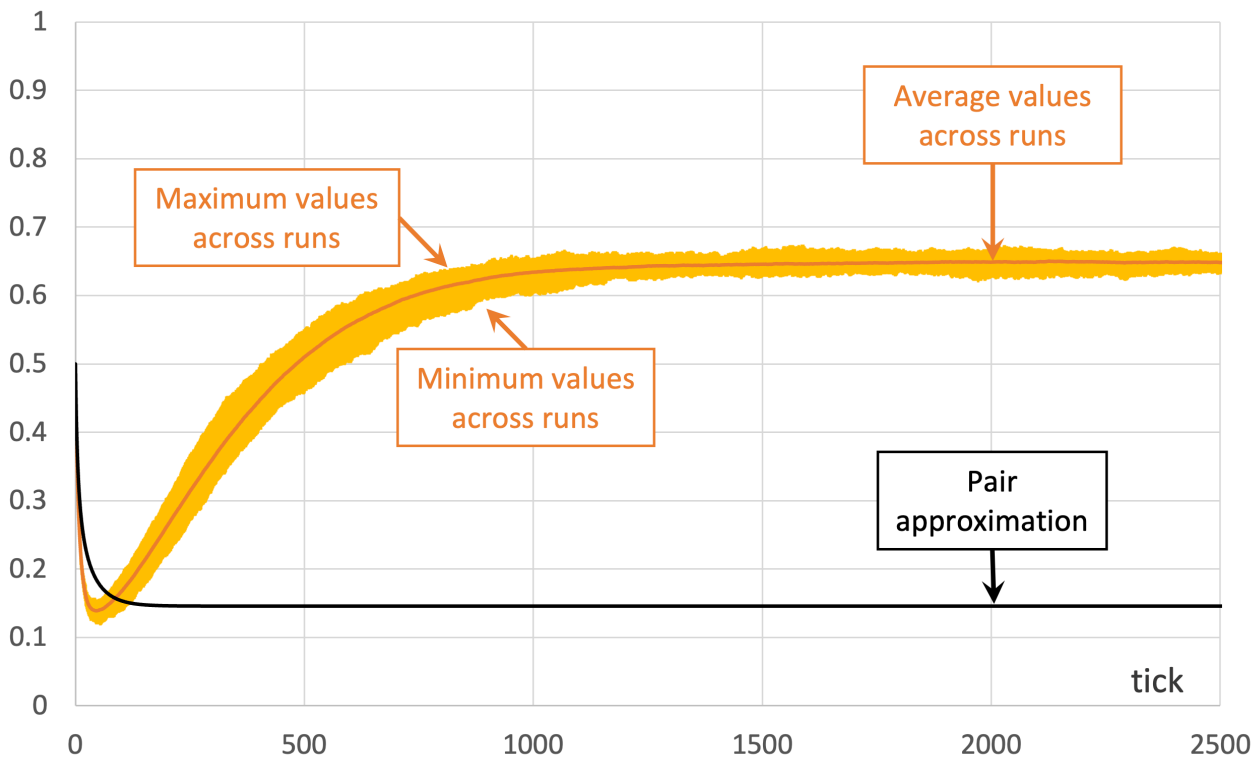
*Figure 3. Pair approximation and simulation results for the snowdrift game played synchronously on a 100×100 torus using Moore neighborhoods ($k = 8$) and the imitative-pairwise-difference rule. The black solid line shows the fraction of $A$-strategists according to the pair approximation, while the orange solid line shows the average values across 1000 simulation runs. The orange error bars show the minimum and maximum values observed across the 1000 runs. Payoffs [[1 0.8][1.2 0]]; initial conditions [5000 5000]*

As you can see in Figure 3, in this case the pair approximation works reasonably well, and its limiting value $x_A^\infty \equiv \lim_{t\to\infty} x_A(t) = 0.70$ provides a very good estimate of the long-term average value of the fraction of $A$-strategists.[13]

Unfortunately, the pair approximation does not always work so well, by any means. As an example, Figure 4 shows results for the same parameterization as in Figure 3, but using Von Neumann neighborhoods ($k = 4$) instead of Moore neighborhoods.

---

13. For asynchronous updating, the match between the pair approximation and the simulations is slightly worse but qualitatively similar. The long-term average values for synchronous and asynchronous updating, together with the pair approximation, are plotted in figure 1d of Hauert and Doebeli (2004), entering a cost-to-benefit ratio of 0.2.
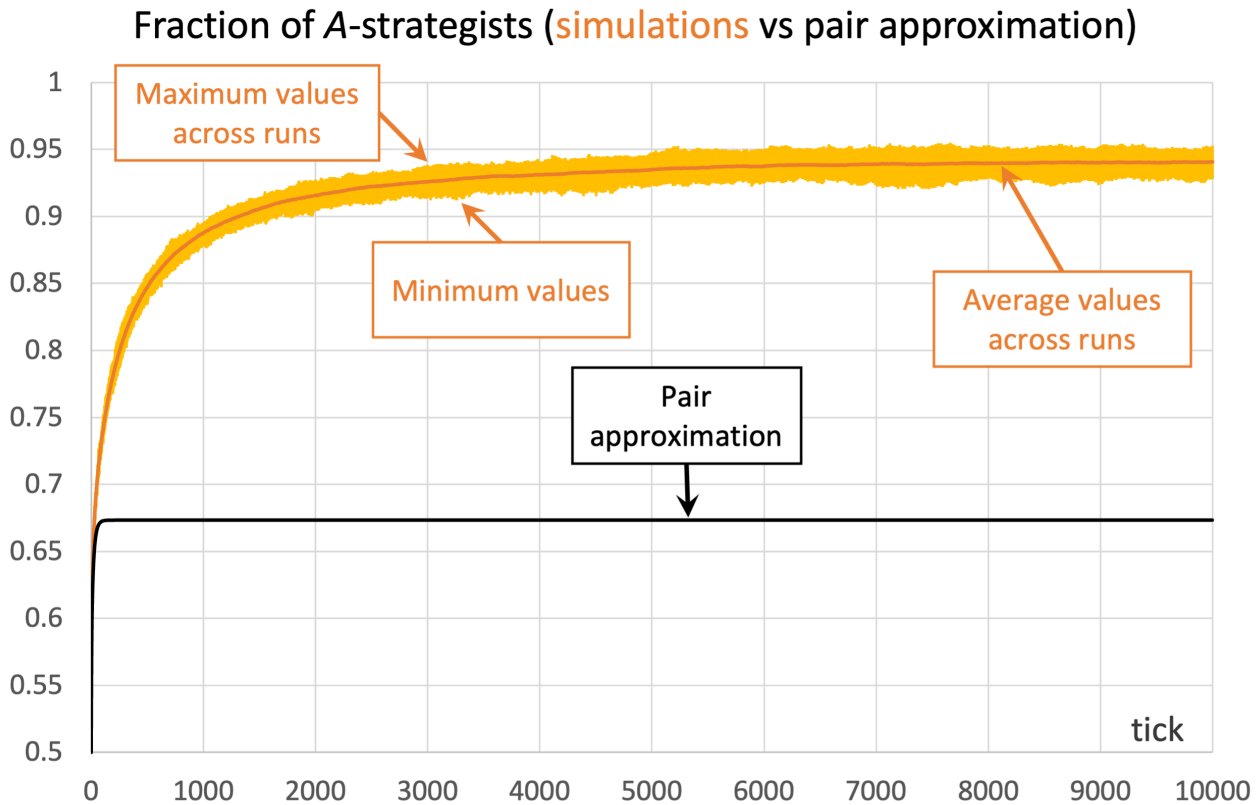
*Figure 4. Pair approximation and simulation results for the snowdrift game played synchronously on a 100×100 torus using Von Neumann neighborhoods ($k = 4$) and the imitative–pairwise–difference rule. The black solid line shows the fraction of $A$–strategists according to the pair approximation, while the orange solid line shows the average values across 100 simulation runs. The orange error bars show the minimum and maximum values observed across the 100 runs. Payoffs [[1 0.8][1.2 0]]; initial conditions [5000 5000]*

In the case shown in Figure 4, the pair approximation provides an estimate for the long-term average fraction of $A$-strategists of $x_A^\infty = 0.67$, which is far from the value of 0.94 obtained by running the stochastic model.[14]

We conclude this section with another example where the pair approximation does not work well. Consider the Prisoner's Dilemma with payoffs [[1 0][1.1 0.1]] played asynchronously on a 125×125 torus using Moore neighborhoods ($k = 8$) and the imitative-pairwise-difference rule. Initial conditions are such that $x_A(0) \approx q_{A/A}(0) \approx 0.5$. Figure 5 shows the comparison between the pair approximation and a computational experiment run with our nxn-imitate-best-nbr-extended model.[15] In this case, the pair approximation provides an estimate for the long-term average fraction of $A$-strategists of $x_A^\infty = 0.15$, while the value obtained from the computational experiment is 0.65.

---

14. These values are plotted in figure 1b of Hauert and Doebeli (2004), entering a cost-to-benefit ratio of 0.2.

15. A similar figure can be found in Fu et al. (2010, figure 4a, red lines), though they use different initial conditions. The limiting values ($x_A^\infty = 0.15$ and the value obtained from the computational experiment, 0.65) are also plotted in figure 6a of Fu et al. (2010), for $u = 0.1$. Fu et al. (2010) use total payoffs rather than average payoffs, but this difference is irrelevant when using the imitative-pairwise-difference rule on regular networks.

Figure 5. Pair approximation and simulation results for the Prisoner's Dilemma played asynchronously on a 125×125 torus using Moore neighborhoods ($k = 8$) and the imitative-pairwise-difference rule. The black solid line shows the fraction of $A$-strategists according to the pair approximation, while the orange solid line shows the average values across 100 simulation runs. The orange error bars show the minimum and maximum values observed across the 100 runs. Payoffs [[1 0][1.1 0.1]]; initial conditions [7812 7813]

The take-home message of these examples is that the pair approximation may or may not work well, and in most cases there is no simpler or shorter way of finding out whether it will work well other than running the computer simulations.

## 3.4. Discussion

Note that in the "well-mixed population" models we developed in Part II, every player was equally likely to interact with every other player, regardless of their strategies. Thus, in those models $q_{A|A} = q_{A|B} = x_A$.

By contrast, in spatial models, the values of $x_A$ and $q_{A|A}$ generally differ, and we often obtain better approximations if we track their joint evolution, as we do in the pair approximation. However, it seems clear that in most cases these two variables will not be enough to condense all the relevant information that can affect evolutionary dynamics on grids (or, more generally, on networks). To make this important point crystal clear, below we present different network configurations with exactly the same initial values of $x_A$ and $q_{A|A}$ (so the prediction of the pair approximation is exactly the same for both), but with completely different evolution.

Consider a toroidal grid with Von Neumann neighborhoods ($k = 4$). Agents play a coordination game with strategies $A$ (orange) and $B$ (green), and payoff matrix [[3 0][0 2]], using the imitate-if-better-all-

nbrs decision rule. You may assume synchronous or asynchronous updating. Initially, all players are using the green strategy $B$ except for a group of 8 agents who use strategy $A$. See the two initial configurations in Table 3 below for two examples. These initial configurations show only the part of the grid corresponding to the 8 orange $A$-strategists and their second neighbors; the numbers are the total payoff obtained by each player. You may assume that the grid is larger than the part shown in the figures, as long as the part not shown is green.

Table 3 shows two distinct initial configurations, each with 8 $A$-strategists and 8 undirected $A$–$A$ links, so they both have the same values for $x_A$ and $q_{A|A}$. However, configuration c1 is an absorbing state (i.e., it does not change over time), while configuration c2 leads to a complete invasion by orange strategy $A$.



Table 3. Two initial configurations with the same values for $x_A$ and $q_{A|A}$, and their corresponding final states

Similarly, initial configurations c3 to c6 in Table 4 share the same number of $A$-strategists (8) and of $A$–$A$ links (7 undirected links), so they all present the same values for $x_A$ and $q_{A|A}$. However, their dynamic evolution is completely different:

- Configuration c3 leads to the survival of the 2×2 orange square block and the disappearance of the other orange $A$-players.
- Configuration c4 leads to a complete invasion by orange strategy $A$.
- Configuration c5 leads to a complete invasion by green strategy $B$.
- Configuration c6 may lead to the invasion of orange $A$, to the invasion of green $B$, or to one of two other possible final states where both strategies are present.

Table 4. Four initial configurations with the same values for $x_A$ and $q_{A|A}$, and the final states they may lead to

It is then clear that variables $x_A$ and $q_{A|A}$ are not sufficient to condense all the relevant information

that can affect evolutionary dynamics on grids.[16] Thus, in general, the pair approximation presented here cannot provide a good description of the agent-based stochastic dynamics.

One may wonder whether adding more variables may lead to better approximations. There are different ways we can generalize and improve the pair approximation we have derived here. For instance, if the underlying network is such that neighboring nodes tend to have common neighbors (i.e., high clustering coefficient), we may refine the estimation of the probability of a neighbor's neighborhood configuration (i.e., our terms $P_{A,\text{One}B}(k'_A)$ and $P_{B,\text{One}A}(k'_A)$) by taking into account these correlations (see, e.g., Morita (2008)). Similarly, to estimate the probability $q_{X|AB}$ that a (random) neighbor of an $A$-player linked to a $B$-neighbor is an $X$-player (with $X \in \{A, B\}$), we could keep track of the expected evolution of triplets, such as [AAB]. This could well improve the quality of the approximation, but it would also increase the number of variables and equations in the system of ODEs, possibly making it less amenable to analysis (van Baalen, 2000).

In any case, the bottom line is that there is no guarantee whatsoever that adding more variables will eventually yield a good analytical approximation that will work in every case. While for unstructured populations there are mathematical theorems that guarantee that the stochastic process converges to the mean dynamic as the population size grows, this is not generally the case for pair approximations in structured populations. The "validity" of pair approximations is generally assessed via simulations... the very same simulations the approximation aims to describe.

---

16. Hauert and Szabó (2005, figure 5) show examples of different regular networks that share the same pair approximation but have very different dynamics.

# 4. Exercises

**Exercise 1.** Derive the pair approximation for the imitate-if-better-all-nbrs decision rule.

**Exercise 2.** Derive the pair approximation for the Fermi-m decision rule.

**Exercise 3.** Derive the pair approximation for the imitate the best neighbor decision rule. For simplicity, you may want to assume that a revising $B$-player will change her strategy only if no $B$-player achieves the maximum payoff in the neighborhood (and analogously for $A$-revisers). This is a slightly different way of resolving ties. Warning! This exercise is significantly harder than the two previous ones.

**Exercise 4.** Extend our nxn-imitate-best-nbr-extended model to include two monitors that show the values of $x_A$ and $q_{A|A}$.

**Exercise 5.** In exercise III-4.1, we parameterized our nxn-imitate-best-nbr-extended model to replicate the *simulation* results shown in figures



*Photo by Antony Hyson S on Unsplash*

1b and 1d of Hauert and Doebeli (2004). We can now replicate their *pair approximation* results too (i.e., the solid line in these figures). Please, use the Mathematica notebook that solves the pair approximation to compute a few values of the solid lines in these two figures.

**Exercise 6.** How can we parameterize our nxn-imitate-best-nbr-extended model and the Mathematica notebook that solves the pair approximation to replicate the results shown in figures 4a, 4d, 6a and 6c of Fu et al. (2010)?

# PART IV. GAMES ON NETWORKS

# IV-1. The nxn game on a random network

## 1. Goal

The goal of this chapter is to learn how to implement models where players are connected in a network (see figure 1). A network is a set of nodes and a set of links.[1] Links connect pairs of nodes. In our models, the nodes in the network will be the players, so each link connects two players.
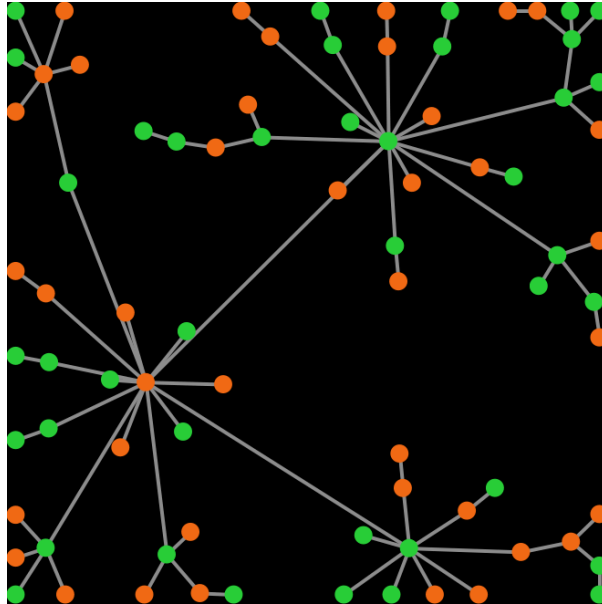


*Figure 1. Example of a preferential attachment network. Players are represented as circles. There are links between some pairs of players*

In this book, we will only use *undirected* links (which denote symmetric relations such as "being a sibling of"). Nonetheless, in NetLogo it is equally easy to implement models with directed links (for asymmetric relations, such as "being a parent of"), and also models with both types of links.

Here, we will use networks to limit the information that players can access. We assume that players can only interact with their link-neighbors (i.e. those with whom the player shares a link), so link-neighbors are the only players that a player can observe or play with. In this sense, networks define local neighborhoods of interaction, potentially different for each player.

By using networks, we will be able to generalize all the models previously developed in this book.

---

1. A more mathematical term for network is "graph". People tend to use the term "graph" when they study the mathematical structure, composed only of nodes and links. The term "network", rather than graph, is often used when nodes or links have some individual attributes or properties that are of interest. In this book we will use the term "network", but do feel free to use the term "graph" if you like. In graph theory, nodes are often called "vertices" and links are often called "edges".

Note that in Part II we implemented models where every player could observe and play with every other player. Such models can be interpreted as network models where players are connected through a complete network, i.e., a network where everyone is linked to everybody else. In Part III, we implemented models with spatial structure, i.e., models where players were embedded on a 2-dimensional grid and they could only interact with their spatial neighbors. Those models can be perfectly interpreted as network models. For instance, the spatial model where we used Von Neumann neighborhoods of radius 1 corresponds to a square lattice network. In this Part IV, we will learn to implement models where players are connected through any arbitrary network.

# 2. Motivation. A single-optimum coordination game

Consider the following 2-player 2-strategy single-optimum coordination game (which we discussed in chapter I-2):

<div align="center">

**Player 2**

|  | Player 2 chooses **A** | Player 2 chooses **B** |
|---|---|---|
| Player 1 chooses **A** | **1** , **1** | **0** , **0** |
| Player 1 chooses **B** | **0** , **0** | **2** , **2** |

**Player 1**

</div>

If you explore the dynamics of this game with the last model we developed in Part II, i.e. nxn-imitate-if-better-noise-efficient, you will see that a population of 100 agents using the *imitate if better* rule with low noise (e.g. *noise* = 0.03), starting with 70 A-strategists and 30 B-strategists, will almost certainly approach the inefficient state where everyone is choosing strategy A, and spend most of the time around it. In the video below, strategy A corresponds to strategy 0 (orange) and strategy B corresponds to strategy 1 (green).

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=341#video-341-1*

Note that in every model developed in Part II, every player can observe and play with every other player (i.e., the network of potential interactions is complete). Now imagine that instead of assuming that everyone can interact with everyone, we assume that each player has just a few contacts (say about 2 on average), and we also asume that these contacts are set at random, in the sense that any possible contact exists with the same probability (i.e., an Erdős–Rényi random network).

Do you think that this change on the network of potential interactions (from complete, to sparse and random) will make any difference? Players will still select other players to observe and to play *at random*. The difference will be that the set of players with which each player can interact throughout the course of the simulation will be much smaller (though still random).

Let us build a model to explore this question!

# 3. Description of the model

We depart from the program we implemented in chapter II-3 (nxn-imitate-if-better-noise). In chapter II-4, we saw that the computational speed of this program could be greatly increased. However, the speed boost came at the expense of making our code slightly less readable. Here we want to focus on code readability, so we believe it is better to start with the most natural implementation of the model, i.e. nxn-imitate-if-better-noise.

The only change we are going to make to the model that nxn-imitate-if-better-noise implements is to embed the players on a network created following the G(*n-of-players*, *prob-link*) Erdős–Rényi random network model. In this type of network model, each possible link between any two players is included in the network with probability *prob-link*, independently from every other link. The following is a full description of the model we aim to implement, highlighting the main changes:

In this model, there is a population of *n-of-players* agents who repeatedly play a symmetric 2-player game with any number of strategies. The *payoffs* of the game are determined by the user in the form of a matrix [ [$A_{00}$ $A_{01}$ ... $A_{0n}$] [$A_{10}$ $A_{11}$ ... $A_{1n}$] ... [$A_{n0}$ $A_{n1}$ ... $A_{nn}$] ] containing the payoffs $A_{ij}$ that an agent playing strategy *i* obtains when meeting an agent playing strategy *j* (*i*, *j* ∈ {0, 1, ..., *n*}). The number of strategies is inferred from the number of rows in the payoff matrix.

The initial strategy distribution is set with parameter *n-of-players-for-each-strategy*, using a list of the form [$a_0$ $a_1$ ... $a_n$], where item $a_i$ is the initial number of agents with strategy *i*. Thus, the total number of agents is the sum of all elements in this list.

**Agents are embedded on a network created following the G(*n-of-players*, *prob-link*) Erdős–Rényi random network model. The network is created once at the beginning of the simulation and it is kept fixed for the whole simulation. Players can only interact with their link-neighbors in the network.**

Once initial conditions are set and the network has been created, the following sequence of events –which defines a tick– is repeatedly executed:

1. Every agent obtains a payoff by selecting **one of her link-neighbors** at random and playing the game.

2. With probability *prob-revision*, individual agents are given the opportunity to revise their strategies. In that case, with probability *noise*, the revising agent will adopt a random strategy; and with probability (1 – *noise*), the revising agent will choose her strategy following the *imitate if better* rule, adapted for networks:

   > Look at **one (randomly selected) link-neighbor** and adopt her strategy if and only if her payoff was greater than yours. (And do nothing if you have no neighbors.)[2]

---

2. Note that, in this model, the agent observed by the revising agent may or may not be the same agent that the revising agent played with. In general, imposing that revising agents necessarily look at the same agent they played with leads to very different dynamics (see Hauert and Miękisz (2018)).

All agents who revise their strategies within the same tick do it simultaneously (i.e. synchronously).

The model shows the evolution of the number of agents choosing each of the possible strategies at the end of every tick. **The model also shows a representation of the network, with players colored according to their strategies.**

# 4. Interface design

We depart from the model we created in chapter II-3 (nxn-imitate-if-better-noise), so if you want to preserve it, now is a good time to duplicate it. The current interface looks as shown in figure II-3-1. Our goal now is to modify it so it looks as figure 2 below. We will place everything related to networks at the right side of the interface.
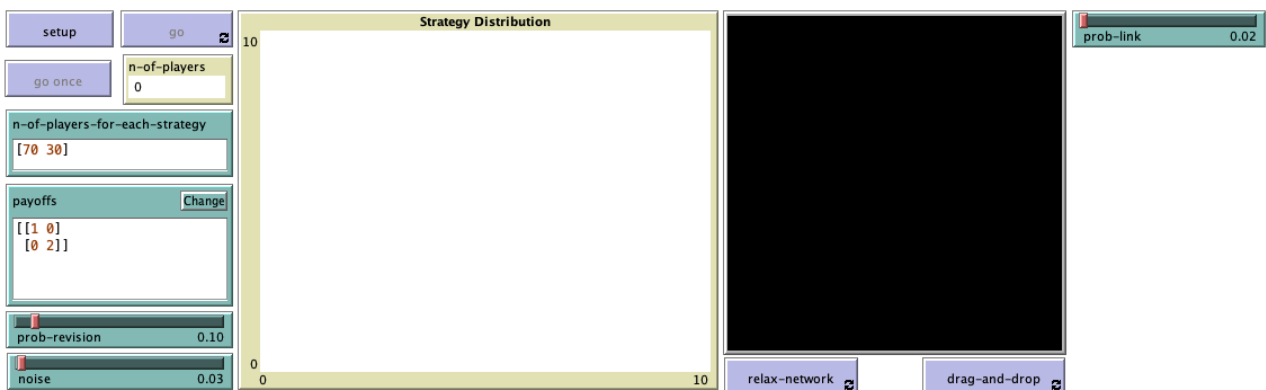


*Figure 2. Interface design*

Let us go through all the necessary changes:

- We should bring forward the 2D view of the NetLogo world (i.e. the large black square) to a place where we can see it. We will use this view to represent the network of players.

  Choose the dimensions of the world by clicking on the "Settings…" button on the top bar, by double-clicking on the 2D view, or by right-clicking on the 2D view and choosing *Edit*. A window will pop up, which allows you to choose the number of patches by setting the values of `min-pxcor`, `max-pxcor`, `min-pycor` and `max-pycor`. You can also determine the patches' size in pixels, and whether the grid wraps horizontally, vertically, both or none (see Topology section).

  We recommend unticking the boxes related to wrapping, to prevent the links of the network from going through the boundaries. Apart from that, feel free to choose any values you like for the other parameters. Our settings are shown in figure 3 below:
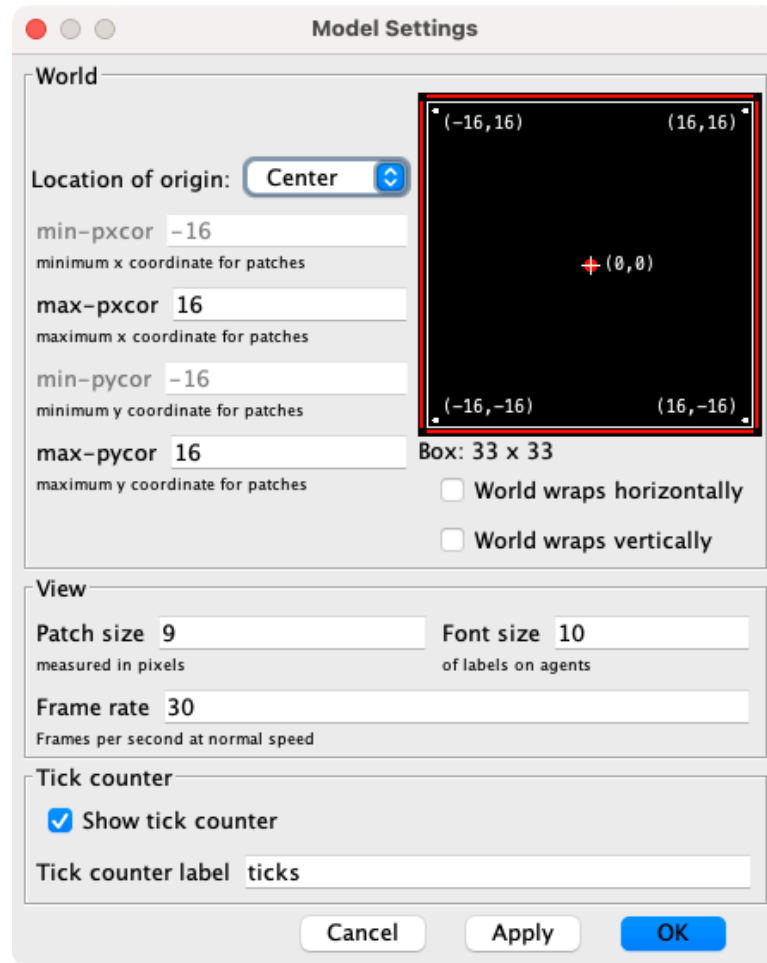
*Figure 3. Model settings*

- Let us create two new buttons, for procedures that deal with the visualization of the network:

    1. One button named relax-network, which runs the procedure to relax-network indefinitely.

    2. One button named drag-and-drop, which runs the procedure to drag-and-drop indefinitely.

    In the Code tab, write the procedures to relax-network and to drag-and-drop, without including any code inside for now.

    ```
    to relax-network
      ;; empty for now
    end
    ```

```
to drag-and-drop
  ;; empty for now
end
```

Then, create the buttons. Since these buttons deal only with visual aspects of the model, you may want to use the primitive `with-local-randomness`, which guarantees that this piece of code does not interfere with the generation of pseudorandom numbers for the rest of the model. Also, do not forget to tick the "Forever" option. When pressed, these buttons will make their respective procedures run repeatedly until the button is pressed again. We will understand later why we want this.
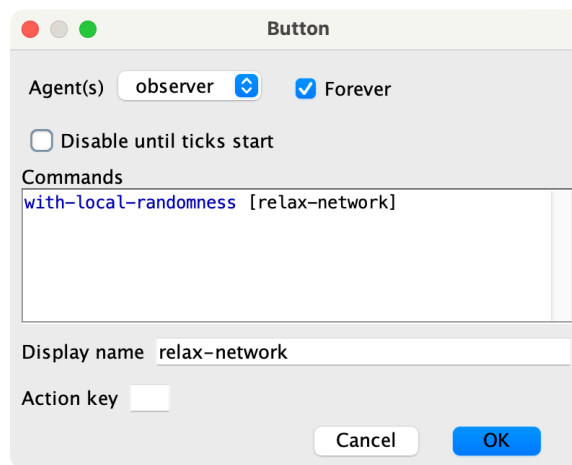


*Figure 4. Settings for relax-network button. Do not forget to tick on "Forever"*

- Finally, let us create a slider to let the user choose the probability *prob-link* with which each link should be created in the random network.

Create a slider for global variable *prob-link*. You can choose limit values 0 (as the minimum) and 1 (as the maximum), and an increment of 0.01.

# 5. Code

## 5.1. Skeleton of the code

Figure 5 below provides a schematic view of the code. You can find the legend for code skeletons in Appendix A-2.
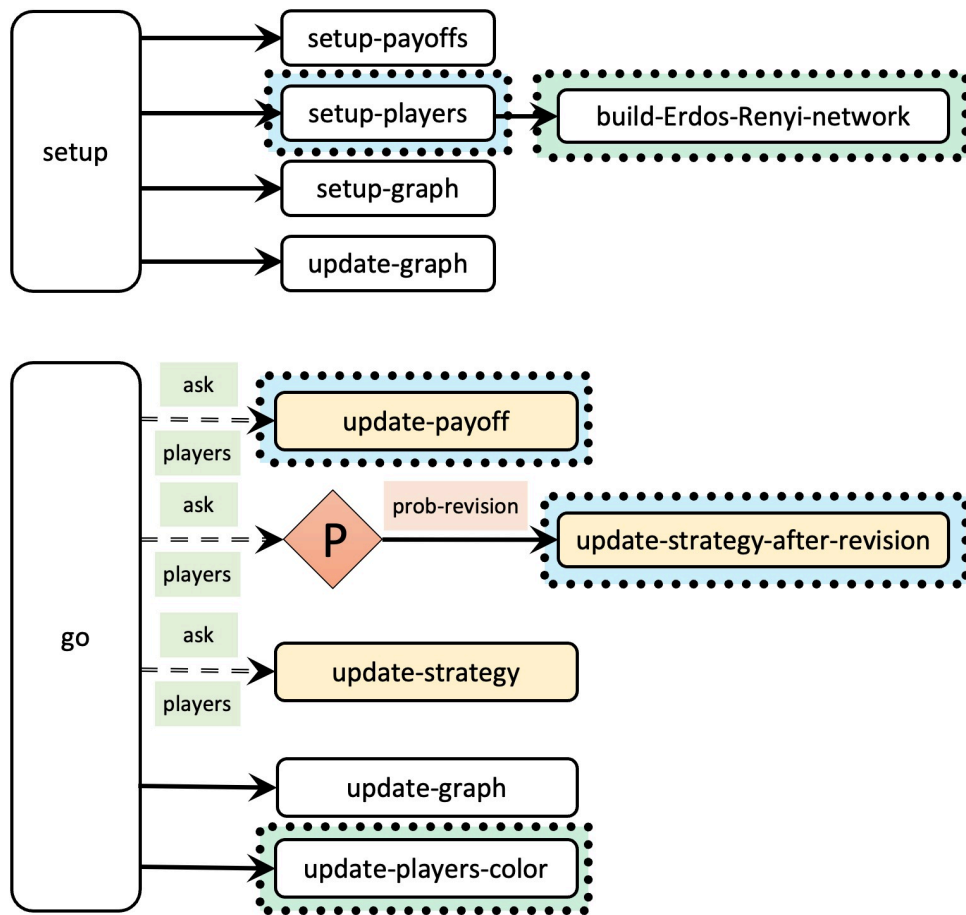
*Figure 5. Skeleton of the code. The dashed green rectangles highlight the main additions in the code. The dashed blue rectangles highlight the main modifications in the code*

Note that the only differences between the skeleton of our current code and the skeleton of the new model we are creating now (fig. 5) are that:

- we are going to call a new procedure named to build-Erdos-Renyi-network when setting up the players (to build the random network),

- we are going to modify procedure to update-payoff so agents obtain their payoff by selecting **one of her link-neighbors** at random (rather than one other agent in the population at random),

- we are going to modify procedure to update-strategy-after-revision so revising agents look at **one (randomly selected) link-neighbor** (rather than at any other agent in the population), and

- we are going to call a new procedure named to update-players-color at the end of procedure to go, to color the players according to their strategy in the 2D view.

Let's make it happen!!

# 5.2. Extensions, global variables and individually-owned variables

## Extensions

To implement models with networks in NetLogo, there is a wonderful extension that will make our life much easier: the nw extension. It is highly recommended that you read its short documentation now. Then, to load the nw extension, write the following line at the very top of the code:

```
extensions [nw]
```

## Global variables

There is no need to add or remove any global variables in our code. The current global variables are:

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
]
```

## Individually-owned variables

There is no need to add or remove any individually-owned variables in our code. The current individually-owned variables are:

```
players-own [
  strategy
  strategy-after-revision
  payoff
]
```

# 5.3. Setup procedures

In this model we want to embed the players in a G(*n-of-players*, *prob-link*) Erdős–Rényi random network. Thus, we should create a procedure to do that. We can call it to build-Erdos-Renyi-network. Primitive `nw:generate-random` from the nw extension builds the network for us. Thus, the code for this procedure is particularly easy:

```
to build-Erdos-Renyi-network
  nw:generate-random players links n-of-players prob-link
end
```

To create the network, we have to specify the breed of turtles and the breed of links that will be used. Note that the command `nw:generate-random` will actually create new turtles and new links, from the breed that we specify. We want `players` to be the breed of turtles in our network and, since we have not defined a specific breed of links, we can just use the primitive `links`. To generate the network, we also have to specify the number of `players` we want to create, so we will have to make sure that variable n-of-players has the appropriate value at the time of calling procedure to build-Erdos-Renyi-network. Finally, *prob-link* is a parameter set by the user, so we do not have to worry about that.

Since procedure to build-Erdos-Renyi-network actually creates the players, we should call it from procedure to setup-players, which we can modify as follows:

```
to setup-players
  let initial-distribution
      read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]

  ;; we have to compute the number of players
  ;; before running procedure build-Erdos-Renyi-network
  set n-of-players sum initial-distribution

  ;; the following line is just for aesthetics
  set-default-shape players "circle"

  ;; now we build the network
  build-Erdos-Renyi-network
  ;; now we have created the players and the links

  ;; the following line is just for aesthetics
  ask players [fd 15]

  ;; the following lines ensure that
  ;; we set the initial distribution of strategies
  ;; according to initial-distribution
  ask players [set strategy -1]
  let i 0
  foreach initial-distribution [ j ->
    ;; note that, below, we do not create the
    ;; players anymore (since they already exist)
    ask n-of j players with [strategy = -1] [
      set payoff 0
      set strategy i
      set strategy-after-revision strategy
    ]
    set i (i + 1)
  ]

  set n-of-players count players
end
```

At this point, we can run our code and check that we have successfully created the network!
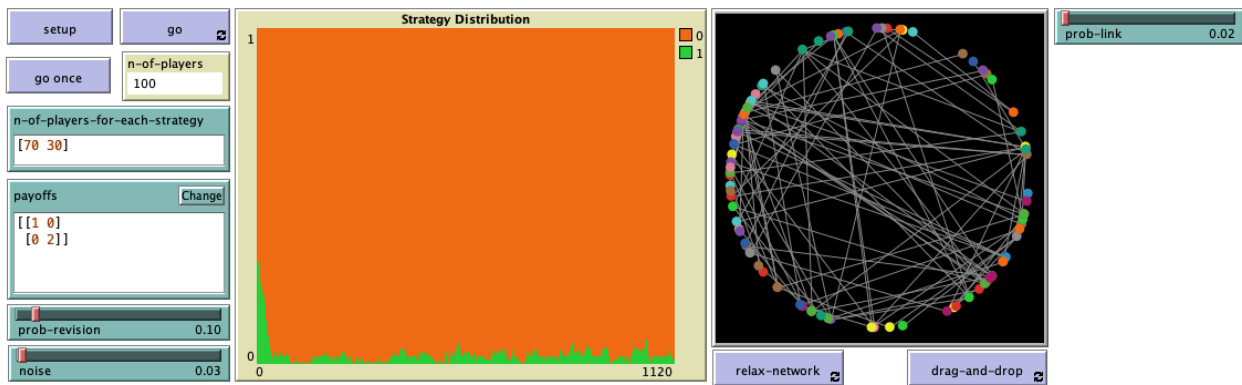
*Figure 6. Snapshot of a simulation run with the current code*

Even though we have successfully created the network, players can still observe and play with any other player in the population… but we will fix that in no time!

## 5.4. Go and other main procedures

In our model, we want players to interact only with their link-neighbors. In NetLogo, there is actually a primitive called `link-neighbors`, which we can use once we have created the network. Thus, in our code we should replace

```
other players
```

with the primitive `link-neighbors` at the places where we ask players to interact with other players. One such place is at procedure to update-payoff:

```
to update-payoff
  ;; let mate one-of other players   <== deleted line
  let mate one-of link-neighbors  ;; <== added line
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end
```

And the other place is at procedure to update-strategy-after-revision:

```
to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      ;; let observed player one-of other players   <== deleted line
      let observed-player one-of link-neighbors ;; <== added line

      if ([payoff] of observed-player) > payoff [
        set strategy-after-revision
            ([strategy] of observed-player)
      ]
    ]
end
```

At this point, our code should not contain any syntactic errors. However, if you run it, you will get an error. Do you understand why? Can you fix it? The box below contains the solution. We hide it to give you a chance to experience the great satisfaction that comes when you accomplish something challenging.

Fix

Well done! Indeed, we sometimes ask players who have no neighbors to select one. We can easily fix this mistake using primitive `any?` as follows:

```
to update-payoff
  if any? link-neighbors [    ;; <== added line
    let mate one-of link-neighbors
    set payoff
        item ([strategy] of mate) (item strategy payoff-matrix)
  ]                            ;; <== added line
end
```

```
to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      if any? link-neighbors [  ;; <== added line
        let observed-player one-of link-neighbors

        if ([payoff] of observed-player) > payoff [
          set strategy-after-revision
              ([strategy] of observed-player)
        ]
      ]                              ;; <== added line
    ]
end
```

Now our program runs as desired. We just have to improve the visualization.

# 5.5 Other procedures

to update-players-color

We would like to color players according to their strategy. To keep our code modular, we will implement a separate procedure named to update-players-color for that. Can you code it?

```
to update-players-color
  ask players [set color 25 + 40 * strategy]
end
```

We should use the same colors for the strategies here as in the strategy plot, i.e. the colors that we set in procedure to setup-graph for the strategies.

Once we have procedure to update-players-color implemented, we should call it both at the end of procedure to setup-players and of procedure to go. With this, we can run the model and see every individual player's strategy (see fig. 7).
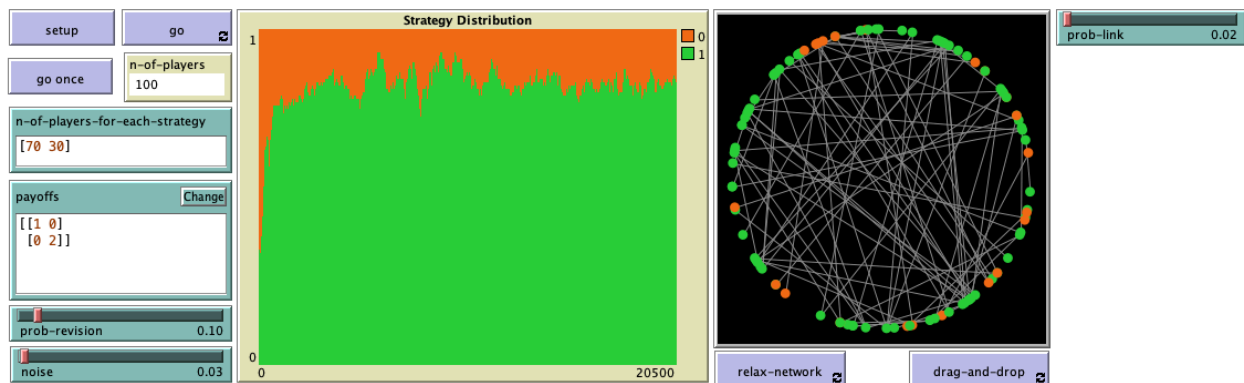


*Figure 7. Snapshot of a simulation run with the current code*

## Visualization of the network: to relax-network and to drag-and-drop

Here we give the implementation of two procedures that will allow the user to visualize and inspect the network more easily. These are included in our code just for visualization purposes, and do not affect the behavior of the players.

Procedure to relax-network distributes the players on the world so they are not too close to each other. It is based on NetLogo's primitive `layout-spring`. The following video shows its beautiful workings:

> *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=341#video-341-2*

Procedure to drag-and-drop lets the user select one player with the mouse and move it around the world.

The credit for the code of these two procedures should go to Wilenski (2005a, 2005b) and his wonderful team.

```
;;;;;;;;;;;;;;;
;;; Layout ;;;
;;;;;;;;;;;;;;;

;; Procedures taken from Wilensky's (2005a) NetLogo Preferential
;; Attachment model
;; http://ccl.northwestern.edu/netlogo/models/PreferentialAttachment
;; and Wilensky's (2005b) Mouse Drag One Example
;; http://ccl.northwestern.edu/netlogo/models/MouseDragOneExample
to relax-network
  ;; the number 3 here is arbitrary; more repetitions slows down the
  ;; model, but too few gives poor layouts
  repeat 3 [
    ;; the more players we have to fit into
    ;; the same amount of space, the smaller
    ;; the inputs to layout-spring we'll need to use
    let factor sqrt count players
    ;; numbers here are arbitrarily chosen for pleasing appearance
    layout-spring players links
                  (1 / factor) (7 / factor) (3 / factor)
    display  ;; for smooth animation
  ]
  ;; don't bump the edges of the world
  let x-offset max [xcor] of players + min [xcor] of players
  let y-offset max [ycor] of players + min [ycor] of players
  ;; big jumps look funny, so only adjust a little each time
  set x-offset limit-magnitude x-offset 0.1
  set y-offset limit-magnitude y-offset 0.1
  ask players [ setxy (xcor - x-offset / 2) (ycor - y-offset / 2) ]
end

to-report limit-magnitude [number limit]
  if number > limit [ report limit ]
  if number < (- limit) [ report (- limit) ]
  report number
end

to drag-and-drop
  if mouse-down? [
    let candidate min-one-of players
        [distancexy mouse-xcor mouse-ycor]
    if [distancexy mouse-xcor mouse-ycor] of candidate < 1 [
    ;; The WATCH primitive puts a "halo" around the watched turtle.
      watch candidate
      while [mouse-down?] [
        ;; If we don't force the view to update, the user won't
        ;; be able to see the turtle moving around.
        display
        ;; The SUBJECT primitive reports the turtle being watched.
        ask subject [ setxy mouse-xcor mouse-ycor ]
```

```
      ]
      ;; Undoes the effects of WATCH.
      reset-perspective
    ]
  ]
end
```

## 5.6. Complete code in the Code tab

```
extensions [nw]

globals [
  payoff-matrix
  n-of-strategies
  n-of-players
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]

to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end

to setup-players
  let initial-distribution
      read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
```

```
      payoffs
    )
  ]

  set n-of-players sum initial-distribution
  set-default-shape players "circle"

  build-Erdos-Renyi-network

  ask players [fd 15]

  ask players [set strategy -1]
  let i 0
  foreach initial-distribution [ j ->
    ask n-of j players with [strategy = -1] [
      set payoff 0
      set strategy i
      set strategy-after-revision strategy
    ]
    set i (i + 1)
  ]

  set n-of-players count players
  update-players-color
end

to build-Erdos-Renyi-network
  nw:generate-random players links n-of-players prob-link
end

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

to go
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]

  tick

  update-graph
  update-players-color
end
```

```
to update-payoff
  if any? link-neighbors [
    let mate one-of link-neighbors
    set payoff
        item ([strategy] of mate) (item strategy payoff-matrix)
  ]
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      if any? link-neighbors [
        let observed-player one-of link-neighbors
        if ([payoff] of observed-player) > payoff [
          set strategy-after-revision
              ([strategy] of observed-player)
        ]
      ]
    ]
end

to update-strategy
  set strategy strategy-after-revision
end

to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers

  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end

to update-players-color
  ask players [set color 25 + 40 * strategy]
end

;;;;;;;;;;;;;;
;;; Layout ;;;
;;;;;;;;;;;;;;

;; Procedures taken from Wilensky's (2005a) NetLogo Preferential
;; Attachment model
```

```
;; http://ccl.northwestern.edu/netlogo/models/PreferentialAttachment
;; and Wilensky's (2005b) Mouse Drag One Example
;; http://ccl.northwestern.edu/netlogo/models/MouseDragOneExample

to relax-network
  ;; the number 3 here is arbitrary; more repetitions slows down the
  ;; model, but too few gives poor layouts
  repeat 3 [
    ;; the more players we have to fit into
    ;; the same amount of space, the smaller
    ;; the inputs to layout-spring we'll need to use
    let factor sqrt count players
    ;; numbers here are arbitrarily chosen for pleasing appearance
    layout-spring players links
                 (1 / factor) (7 / factor) (3 / factor)
    display  ;; for smooth animation
  ]
  ;; don't bump the edges of the world
  let x-offset max [xcor] of players + min [xcor] of players
  let y-offset max [ycor] of players + min [ycor] of players
  ;; big jumps look funny, so only adjust a little each time
  set x-offset limit-magnitude x-offset 0.1
  set y-offset limit-magnitude y-offset 0.1
  ask players [ setxy (xcor - x-offset / 2) (ycor - y-offset / 2) ]
end

to-report limit-magnitude [number limit]
  if number > limit [ report limit ]
  if number < (- limit) [ report (- limit) ]
  report number
end

to drag-and-drop
  if mouse-down? [
    let candidate min-one-of players
        [distancexy mouse-xcor mouse-ycor]
    if [distancexy mouse-xcor mouse-ycor] of candidate < 1 [
    ;; The WATCH primitive puts a "halo" around the watched turtle.
      watch candidate
      while [mouse-down?] [
        ;; If we don't force the view to update, the user won't
        ;; be able to see the turtle moving around.
        display
        ;; The SUBJECT primitive reports the turtle being watched.
        ask subject [ setxy mouse-xcor mouse-ycor ]
      ]
      ;; Undoes the effects of WATCH.
      reset-perspective
    ]
  ]
end
```

# 6. Sample runs

Now that we have the model, we can investigate the question we posed at the motivation section above. If you run the model, you will see that, when embedded on a sparse random network, most players manage to eventually coordinate on the efficient strategy, achieving a much higher payoff than in the setting where they could interact with the whole population. To speed up your simulations, you can untick the "view updates" square at the Interface tab, and also make sure that none of the two network visualization buttons are down. The video below shows some representative runs.

> 🎥 *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=341#video-341-3*

This is a clear example of how network structure can impact evolutionary dynamics. The dynamics when players can only interact with a few neighbors are completely different from the dynamics when they can interact with the whole population.

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-imitate-if-better-rd-nw.nlogo.

Exercise 1. Consider the single-optimum coordination game [[1 0][0 2]]. We have seen that a population of 100 agents embedded on a G(*n-of-players* = 100, *prob-link* = 0.02) Erdős–Rényi random network, using the *imitate if better* rule with *noise* = 0.03, *prob-revision* = 0.1, and starting with 70 A-strategists and 30 B-strategists, will most likely approach the efficient state where most players are choosing strategy B, and spend most of the time around it. Please, check this observation by running several simulations. (You can easily do that by leaving the button go pressed down and clicking the setup button every time you want to start again.)

*Photo by NASA on Unsplash*

In this exercise, we ask you to assess the impact of *prob-link* (which determines the network density) on the proportion of players who choose strategy B. In particular, we would like to see a scatter plot with *prob-link* on the horizontal axis and the average proportion of B-strategists at tick 5000, across several runs, on the vertical axis.

**Exercise 2.** In exercise 1 above, how would changing the payoff matrix to [[1 0][0 20]] affect your answer?

**Exercise 3.** Using the same parameter values as in exercise 1 above, with *prob-link* = 0.04, the average proportion of B-strategists at tick 5000 is approximately 0.61. Does that mean that if you run a simulation and look at it at tick 5000, you can expect to see about 61% of agents choosing strategy B and the other 39% using strategy A?

**Exercise 4.** Make the necessary changes in the code so players follow the *imitate the best neighbor* decision rule, which reads as follows:

> Consider the set of all your neighbors plus yourself; then adopt the strategy of one of the agents in this set who has obtained the greatest payoff. If there is more than one agent with the greatest payoff, choose one of them at random to imitate.

Hint to implement the *imitate the best neighbor* decision rule

You may want to revisit chapter III-1 of the book.

**Exercise 5.** Repeat the experiment you conducted in exercise 1 above, now with the imitate the best neighbor rule, instead of the imitate if better rule adapted for networks. Before you see the results, make a guess about what you will observe.

**Exercise 6.** Can you implement procedure to build-Erdos-Renyi-network without using the nw extension?

Hint to implement procedure to build-Erdos-Renyi-network without using the nw extension

You will have to use primitive `create-links-with`. Also, players' `who` number may be useful to make sure that you only consider each pair of players once.

## IV-2. Different types of networks

# 1. Goal

Our goal here is to extend the model we have created in the previous chapter to study different types of networks.

# 2. Motivation. Assessing the significance of network structure

The model we will develop in this chapter will allow us to explore the importance of network structure on evolutionary game dynamics. Consider, for instance, the 2-player 2-strategy single-optimum coordination game of the previous chapter:

<div align="center">

**Player 2**

|  | Player 2 chooses **A** | Player 2 chooses **B** |
|---|---|---|
| Player 1 chooses **A** | **1** , **1** | **0** , **0** |
| Player 1 chooses **B** | **0** , **0** | **2** , **2** |

</div>

(with **Player 1** labelling the rows)

In the previous chapter we saw that, in this game, under certain conditions:[1]

- an *unstructured* population of 100 agents (i.e., complete network) will most likely approach the inefficient state where all agents choose strategy A and spend most of the time around there (see fig. 1),[2] but,

---

1. Conditions were that agents use the *imitate if better* rule with *noise* = 0.03, *prob-revision* = 0.1, and initial strategy distribution is 70 A-strategists and 30 B-strategists.
2. This statement refers to finite-time horizons, i.e., what Binmore et al. (1995, p. 10) call the long run.
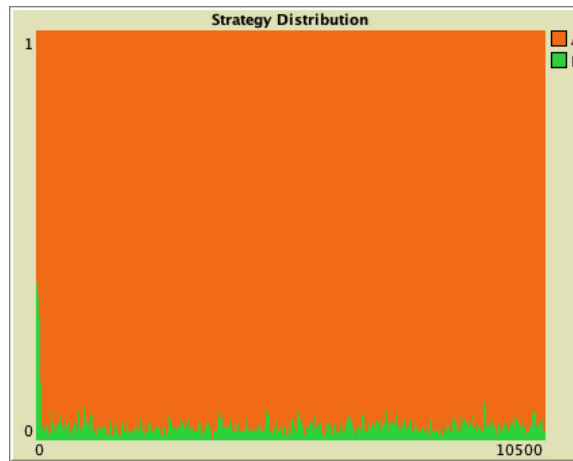
*Figure 1. Simulation in an unstructured population (complete network)*

- in stark contrast, if we embed that population on a G(*n-of-players* = 100, *prob-link* = 0.02) Erdős–Rényi random network, then agents will most likely approach the state where all agents choose strategy B and spend most of the time close to it (see fig. 2).
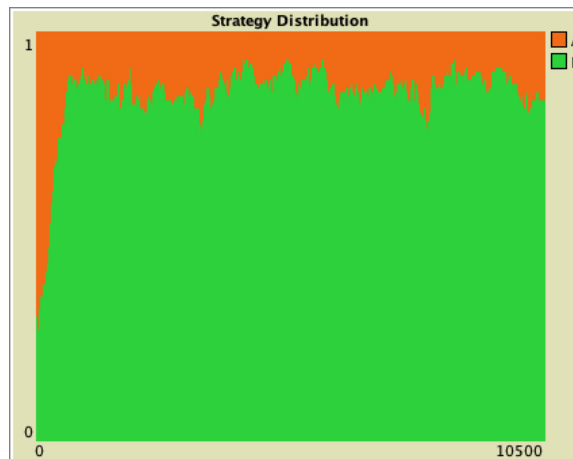


*Figure 2. Simulation in an Erdős–Rényi random network*

What is different in these two networks? For a start, note that the degree (i.e. number of link-neighbors) of players in the unstructured population model is 99 (i.e. complete network), while the *expected* degree of players in the G(100,0.02) random network is just $0.02 \cdot 99 = 1.98 \approx 2$. Thus, an interesting question is: will agents approach the efficient state in any network where they have an average degree of about 2? or is there something special about the random network?

To explore this question, we should embed the population on other networks with average degree about 2, but with different structure. The following figure shows other types of networks where players have about two link-neighbors on average.
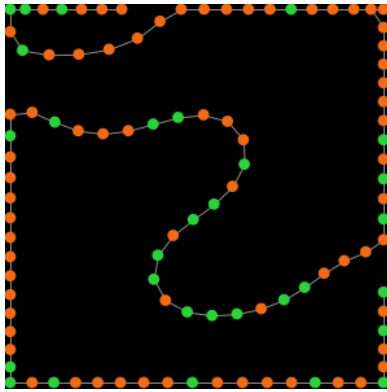
Path network
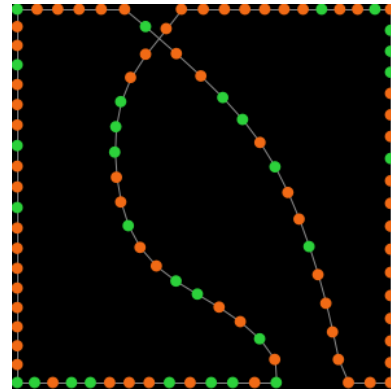

Ring network
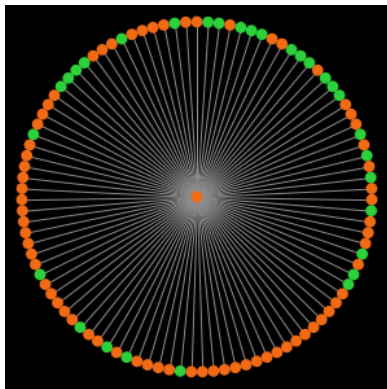

Star network
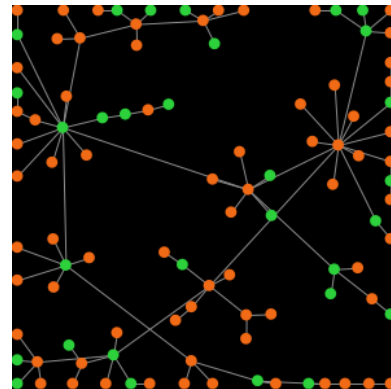

Preferential attachment network
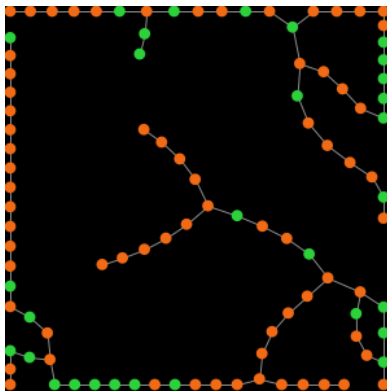

Watts–Strogatz small-world network with
probability of rewiring = 0.1
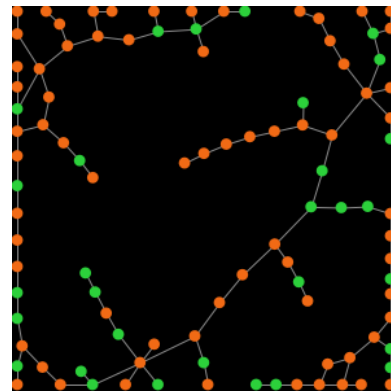

Watts–Strogatz small-world network with
probability of rewiring = 0.5

*Figure 3. Different types of networks with average degree about 2. The average degree is exactly 2 for the ring and the small world networks, and it is (2 · 99 / 100) = 1.98 for the path, star and preferential attachment network*

For each of the networks shown above, do you think that we will get similar results as with the random network? The average degree is about the same in all of them, but the network structure is very different in each case.

Let us build a model to explore this question!

# 3. Description of the model

The only functionality we are going to add to the program implemented in the previous chapter is the possibility of using different network-generating algorithms to create the network. Thus, we refer to the previous chapter to read the basic description of the model. The only information we should add is the following:

Agents are embedded on a network which is created using a network model determined by parameter *network-model*. This parameter is implemented as a chooser, with 8 possible values:

- **"Erdos-Renyi"**. The network is created following the G(*n-of-players*, *prob-link*) Erdős–Rényi random network model (Erdös and Rényi (1959)).

- **"Watts-Strogatz-small-world"**. The network is created using the Watts–Strogatz model (Watts and Strogatz (1998)). This model has two parameters:

    ◦ *avg-degree-small-world*, which determines the average degree of the network, and

    ◦ *prob-rewiring*, which determines the probability of rewiring.

    Informally, the algorithm works as follows: initially, nodes are placed in a circle and each node is linked to its closest *avg-degree-small-world* *spatial* neighbors (considering both sides). This forms a regular[3] ring lattice, where every node is linked to exactly *avg-degree-small-world* other nodes. Then, starting at any one node, consider each of her original links that go clockwise and, with probability *prob-rewiring*, rewire its end at random. Then go to the next node clockwise, and repeat until all nodes have been considered. Self-links (i.e., loops) and duplicated links (i.e., more than one link between two nodes) are not allowed. See example with *avg-degree-small-world* = 2 and *prob-rewiring* = 0.1.

- **"preferential-attachment"**. The network is created following the Barabási–Albert model (Barabási and Albert (1999)). This model has one parameter, i.e. *min-degree*, which determines the minimum degree that a node can have. Informally, the network is created starting from a complete network of *min-degree* nodes, and then sequentially adding new nodes. Each new node comes with *min-degree* additional links to the network, which the new node will use to link to existing nodes with probability proportional to the existing nodes' degree. See example with *min-degree* = 1.

---

3. A regular network is a network where every node has the same degree.

- **"ring"**. The network created is a ring, i.e. a network where each node is linked with exactly two other nodes. (See example.)

- **"star"**. The network created is a star, i.e. a network where one node is linked with every other node, and there are no more links. (See example.)

- **"wheel"**. The network created is a wheel, i.e. a ring network with one extra node that is linked with every other node. (See example.)

- **"grid-4-nbrs"**. The network created is a square grid network. In this case, the program will compute the largest integer no greater than the square root of the number of players, and build a square grid network with that many players in each row and column. (See example.)

- **"path"**. The network created is a path, i.e. a ring network where one link has been removed. (See example.)

The network is created once at the beginning of the simulation and it is kept fixed for the whole simulation. Players can only interact with their link-neighbors in the network.
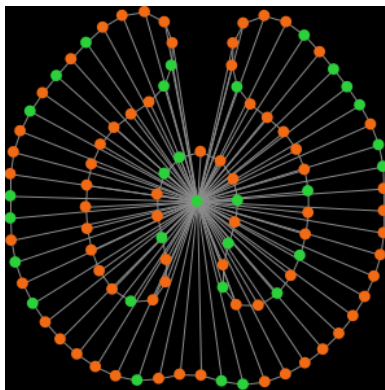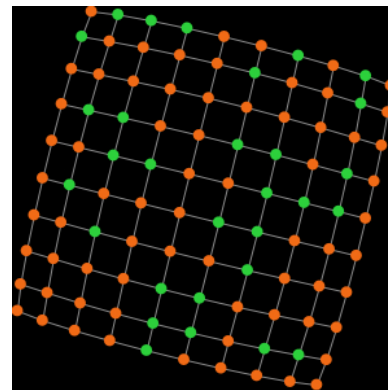


A *wheel network*                                A *square grid network*

Figure 4. A wheel network and a square grid network, each with 100 nodes

# 🔖 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).
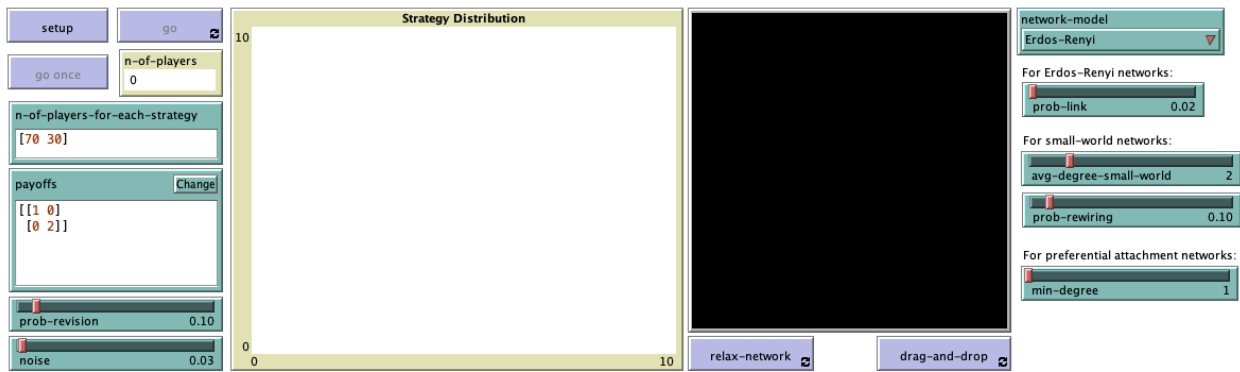
*Figure 5. Interface design*

The new interface (see figure 5 above) includes one new chooser and three new sliders at the right side of the interface, where we are placing every parameter related to networks. To be precise, we have to add:

- One chooser for new parameter *network-model* (with possible values "Erdos-Renyi", "Watts-Strogatz-small-world", "preferential-attachment", "ring", "star", "wheel", "grid-4-nbrs" and "path").

- Two sliders for the Watts-Strogatz small world networks: one for parameter *avg-degree-small-world* (with *minimum* = 0, *maximum* = 20, and *increment* = 2), and another one for parameter *prob-rewiring* (with *minimum* = 0, *maximum* = 1, and *increment* = 0.01).

- One slider for the preferential-attachment networks, for parameter *min-degree* (with *minimum* = 1, *maximum* = 5, and *increment* = 1).

You may want to add some notes above the sliders, as in figure 5, to let the user know the network model for which each parameter is relevant.

# 5. Code

## 5.1. Skeleton of the code

Since we only have to modify how the network is created, and this is something that is conducted in procedure to setup-players, we will only have to modify that procedure. Nonetheless, to make our code modular and elegant, we will create a new procedure named to build-network where the network will be created, and some other procedures which will run the different network-generating algorithms (see fig. 6).
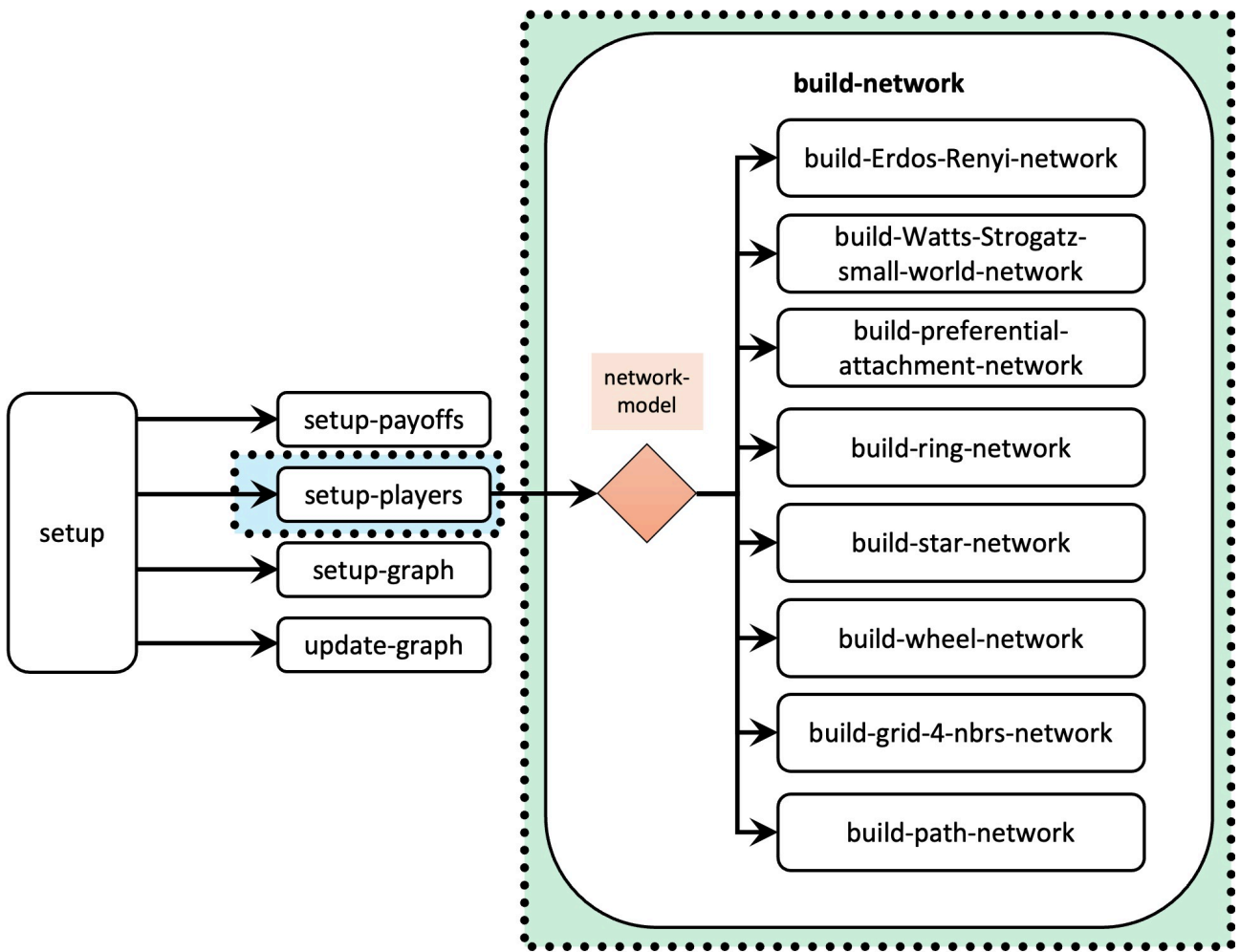
*Figure 6. Skeleton of procedure to setup. The dashed green rectangle highlights the main addition in the code. The dashed blue rectangle highlights the main modification in the code*

## 5.2. Procedures to create networks

In the same way that we programmed a procedure to build networks according to the Erdős–Rényi random network model (i.e., to build-Erdos-Renyi-network), we will have to create new procedures for the other network-generating algorithms. To do this, the nw extension will be invaluable.

All procedures we will program to generate networks will be very short (two lines long, at the most) so, in the following sections, rather than providing you with the code, we will just give you the name of the key command you will have to use. We advise you to read the documentation of the key command and try to implement the procedure by yourself. It is not going to be easy, but we know you are ready for the challenge and, if you try this, you will become an even better programmer.

Also, please, do test your code before looking at the solution, and try to fix it. Your first attempts at the code will most likely contain several errors. Being able to understand and fix errors is one of the most important skills we need to develop. Errors are great opportunities to learn. Embrace them as an integral part of the learning process, and enjoy fixing them!

### to build-Watts-Strogatz-small-world-network

Have a look at the documentation of command `nw:generate-watts-strogatz` and try to

implement this procedure. Recall that you will have to use parameters *avg-degree-small-world* and *prob-rewiring*.

> ### Implementation of procedure to build-Watts-Strogatz-small-world-network
>
> ```
> to build-Watts-Strogatz-small-world-network
>   nw:generate-watts-strogatz players links n-of-players
>       (avg-degree-small-world / 2) prob-rewiring
> end
> ```

## to build-preferential-attachment-network

Have a look at the documentation of command `nw:generate-preferential-attachment` and try to implement this procedure. Recall that you will have to use parameter *min-degree*.

> ### Implementation of procedure to build-preferential-attachment-network
>
> ```
> to build-preferential-attachment-network
>   nw:generate-preferential-attachment players links n-of-players
>       min-degree
> end
> ```

## to build-ring-network

Have a look at the documentation of command `nw:generate-ring` and try to implement this procedure.

> ### Implementation of procedure to build-ring-network
>
> ```
> to build-ring-network
>   nw:generate-ring players links n-of-players
> end
> ```

## to build-star-network

Have a look at the documentation of command `nw:generate-star` and try to implement this procedure.

Implementation of procedure to build-star-network

```
to build-star-network
  nw:generate-star players links n-of-players
end
```

## to build-grid-4-nbrs-network

Have a look at the documentation of command `nw:generate-lattice-2d` and try to implement this procedure. Recall that we want to compute the largest integer no greater than the square root of the number of players, and build a square grid network with that many players in each row and column.

Implementation of procedure to build-grid-4-nbrs-network

```
to build-grid-4-nbrs-network
  let players-per-line (floor sqrt n-of-players)
  nw:generate-lattice-2d players links
      players-per-line players-per-line false
end
```

## to build-wheel-network

Have a look at the documentation of command `nw:generate-wheel` and try to implement this procedure.

Implementation of procedure to build-wheel-network

```
to build-wheel-network
  nw:generate-wheel players links n-of-players
end
```

## to build-path-network

There is no command to create a path network in the nw extension, but you can easily create it departing from a ring network. Give it a try!

Implementation of procedure to build-path-network

```
to build-path-network
  build-ring-network
  ask one-of links [die]
end
```

# 5.3. Procedure **to build-network**

Our next challenge is to create procedure to build-network, which will run all the commands related to the creation of the network. Currently, some of these commands are in procedure to setup-players, so we should move them to the new procedure. In to build-network we should also run the network-generating algorithm selected by the user in chooser *network-model*. Taking all this into account, we could program the new procedure as follows:

```
to build-network
  set-default-shape players "circle"
  ;; the line above comes from setup-players
  ;; and it should be included before we
  ;; create the network (which creates the players)

  (ifelse
   network-model = "Erdos-Renyi"
     [build-Erdos-Renyi-network]
   network-model = "Watts-Strogatz-small-world"
     [build-Watts-Strogatz-small-world-network]
   network-model = "preferential-attachment"
     [build-preferential-attachment-network]
   network-model = "ring"
     [build-ring-network]
   network-model = "star"
```

```
        [build-star-network]
    network-model = "grid-4-nbrs"
        [build-grid-4-nbrs-network]
    network-model = "wheel"
        [build-wheel-network]
    network-model = "path"
        [build-path-network]
    )

    ask players [fd 15]
    ;; the line above comes from setup-players
    ;; and it should be included after we
    ;; have created the players
end
```

The `ifelse` block of code above can be replaced by one simple line using command `run` (a primitive that can take a string containing the name of a command as an input, and it runs the command). Can you implement that line?

Implementation using `run`

```
to build-network
  set-default-shape players "circle"
  run (word "build-" network-model "-network")
  ask players [fd 15]
end
```

## 5.4. Procedure to setup-players

Once we have implemented procedure to build-network, we should call it from procedure to setup-players and clean up a little bit. The new implementation of to setup-players could look as follows:

```
to setup-players
  let initial-distribution
      read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
```

```
        length payoff-matrix "):\n"
        payoffs
      )
    ]


  set n-of-players sum initial-distribution


  ;; the tasks below take place in
  ;; to build-network now

  ;; set-default-shape players "circle"   <== deleted line
  ;; build-Erdos-Renyi-network            <== deleted line
  ;; ask players [fd 15]                   <== deleted line


  build-network    ;; <== new line


  ask players [set strategy -1]
  let i 0
  foreach initial-distribution [ j ->
    ask n-of j players with [strategy = -1] [
      set payoff 0
      set strategy i
      set strategy-after-revision strategy
    ]
    set i (i + 1)
  ]


  set n-of-players count players
  update-players-color
end
```

## 5.5. Other procedures

Note that there is no need to modify the code of any other procedure.

## 5.6. Final fixes

The bulk of the code is done, but there are still a couple of minor issues we should deal with. We present them below as instructions that will throw an error. Your goal is to fix the problem.

> Run the model with 3 agents embedded on a wheel network

If you try to do this, you will get an error because you need at least 4 agents to create a wheel. To fix this problem, you could modify procedure to setup-players to make sure that there are at least 3 agents, and issue a message if not:

```
to setup-players
  let initial-distribution
      read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]

  set n-of-players sum initial-distribution
  ifelse n-of-players < 4                              ;<==
    [ user-message "There should be at least 4 players" ]  ;<==
    [                                                  ;<==
      build-network

      ask players [set strategy -1]
      let i 0
      foreach initial-distribution [ j ->
        ask up-to-n-of j players with [strategy = -1] [
          set payoff 0
          set strategy i
          set strategy-after-revision strategy
        ]
        set i (i + 1)
      ]

      set n-of-players count players
      update-players-color
    ]                                                  ;<==
end
```

Create a grid-4-nbrs with initial distribution [20 15]

If you try to do this, you will get the error "Requested 15 random agents from a set of only 5 agents". This is because procedure to build-grid-4-nbrs-network creates a network of $\left\lfloor \sqrt{20 + 15} \right\rfloor^2 = 25$ agents. Then, at the foreach loop in procedure to setup-players, initially, 20 random agents out of the 25 with strategy = -1 will be assigned strategy 0. In the second

step of the foreach loop, the code asks 15 agents with strategy = -1 to set their strategy to 1, but there are only 5 agents with strategy -1. Thus the error. To fix it, you can use reporter up-to-n-of instead of n-of.

## 5.7. Complete code in the Code tab

The Code tab is ready!

```
extensions [nw]

globals [
  payoff-matrix
  n-of-strategies
  n-of-players
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]

;;;;;;;;;;;;;
;;; SETUP ;;;
;;;;;;;;;;;;;

to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end

to setup-players
  let initial-distribution
      read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
```

```
        length initial-distribution "):\n"
        n-of-players-for-each-strategy
        "\nshould be equal to the number of rows\n"
        "in the payoff matrix (i.e. "
        length payoff-matrix "):\n"
        payoffs
      )
    ]

  set n-of-players sum initial-distribution
  ifelse n-of-players < 4
    [ user-message "There should be at least 4 players" ]
    [
      build-network

      ask players [set strategy -1]
      let i 0
      foreach initial-distribution [ j ->
        ask up-to-n-of j players with [strategy = -1] [
          set payoff 0
          set strategy i
          set strategy-after-revision strategy
        ]
        set i (i + 1)
      ]

      set n-of-players count players
      update-players-color
    ]
end

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; NETWORK CONSTRUCTION ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to build-network
  set-default-shape players "circle"
  run (word "build-" network-model "-network")
  ask players [fd 15]
end

to build-Erdos-Renyi-network
  nw:generate-random players links n-of-players
    prob-link
```

```
  end

  to build-Watts-Strogatz-small-world-network
   nw:generate-watts-strogatz players links n-of-players
       (avg-degree-small-world / 2) prob-rewiring
  end

  to build-preferential-attachment-network
    nw:generate-preferential-attachment players links n-of-players
        min-degree
  end

  to build-ring-network
    nw:generate-ring players links n-of-players
  end

  to build-star-network
    nw:generate-star players links n-of-players
  end

  to build-grid-4-nbrs-network
    let players-per-line (floor sqrt n-of-players)
    nw:generate-lattice-2d players links
        players-per-line players-per-line false
  end

  to build-wheel-network
    nw:generate-wheel players links n-of-players
  end

  to build-path-network
    build-ring-network
    ask one-of links [die]
  end

  ;;;;;;;;;;
  ;;; GO ;;;
  ;;;;;;;;;;

  to go
    ask players [update-payoff]
    ask players [
      if (random-float 1 < prob-revision) [
        update-strategy-after-revision
      ]
    ]
    ask players [update-strategy]

    tick
    update-graph
    update-players-color
  end
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;
;;; UPDATE PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;

to update-payoff
  if any? link-neighbors [
    let mate one-of link-neighbors
    set payoff
        item ([strategy] of mate) (item strategy payoff-matrix)
  ]
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      if any? link-neighbors [
        let observed-player one-of link-neighbors
        if ([payoff] of observed-player) > payoff [
          set strategy-after-revision
              ([strategy] of observed-player)
        ]
      ]
    ]
end

to update-strategy
  set strategy strategy-after-revision
end

to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers

  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end

to update-players-color
  ask players [set color 25 + 40 * strategy]
end


;;;;;;;;;;;;;;
;;; LAYOUT ;;;
;;;;;;;;;;;;;;
```

```
;; Procedures taken from Wilensky's (2005a) NetLogo Preferential
;; Attachment model
;; http://ccl.northwestern.edu/netlogo/models/PreferentialAttachment
;; and Wilensky's (2005b) Mouse Drag One Example
;; http://ccl.northwestern.edu/netlogo/models/MouseDragOneExample
to relax-network
  ;; the number 3 here is arbitrary; more repetitions slows down the
  ;; model, but too few gives poor layouts
  repeat 3 [
    ;; the more players we have to fit into
    ;; the same amount of space, the smaller
    ;; the inputs to layout-spring we'll need to use
    let factor sqrt count players
    ;; numbers here are arbitrarily chosen for pleasing appearance
    layout-spring players links
                  (1 / factor) (7 / factor) (3 / factor)
    display  ;; for smooth animation
  ]
  ;; don't bump the edges of the world
  let x-offset max [xcor] of players + min [xcor] of players
  let y-offset max [ycor] of players + min [ycor] of players
  ;; big jumps look funny, so only adjust a little each time
  set x-offset limit-magnitude x-offset 0.1
  set y-offset limit-magnitude y-offset 0.1
  ask players [ setxy (xcor - x-offset / 2) (ycor - y-offset / 2) ]
end

to-report limit-magnitude [number limit]
  if number > limit [ report limit ]
  if number < (- limit) [ report (- limit) ]
  report number
end

to drag-and-drop
  if mouse-down? [
    let candidate min-one-of players
        [distancexy mouse-xcor mouse-ycor]
    if [distancexy mouse-xcor mouse-ycor] of candidate < 1 [
    ;; The WATCH primitive puts a "halo" around the watched turtle.
      watch candidate
      while [mouse-down?] [
        ;; If we don't force the view to update, the user won't
        ;; be able to see the turtle moving around.
        display
        ;; The SUBJECT primitive reports the turtle being watched.
        ask subject [ setxy mouse-xcor mouse-ycor ]
      ]
      ;; Undoes the effects of WATCH.
      reset-perspective
    ]
  ]
end
```

# 6. Sample runs

Now that we have the model, we can investigate the question we posed at the motivation section above. To obtain robust statistics, we can conduct an experiment where we run 1000 runs for different types of networks with average degree about 2, and for each type of network we gather the average percentage of players with strategy B at tick 5000 (across the 1000 runs).

The baseline settings will be the same as in the previous chapter (i.e.: *noise* = 0.03, *prob-revision* = 0.1, and initial strategy distribution [70 30]), and we will consider the following network-generating algorithms and parameter values:

- "Erdos-Renyi", with *prob-link* = 0.02
- "Watts-Strogatz-small-world", with *avg-degree-small-world* = 2 and five different values for *prob-rewiring*: {0, 0.25, 0.5, 0.75, 1}
- "preferential-attachment" with *min-degree* = 1
- "ring"
- "star"
- "path"

Try to set up this experiment by yourself. You can find our setup below.

Experiment setup

We have set up the following two experiments in BehaviorSpace:

1. One for all the runs that do not use the network model "Watts-Strogatz-small-world":

*Experiment for all the runs that do not use network model*
*"Watts-Strogatz-small-world"*

The full setting of variables for this experiment is:

```
["network-model" "Erdos-Renyi" "preferential-attachment" "ring" "star" "path"]
["prob-link" 0.02]
["min-degree" 1]
["payoffs" "[[1 0]\n [0 2]]"]
["n-of-players-for-each-strategy" "[70 30]"]
["prob-revision" 0.1]
["noise" 0.03]
["avg-degree-small-world" 2]
["prob-rewiring" 0]
```

2. And a second experiment for all the runs that use the network model "Watts-Strogatz-small-world":

*Experiment for all the runs that use network model "Watts-Strogatz-small-world"*

The full setting of variables for this experiment is:

```
["network-model" "Watts-Strogatz-small-world"]
["avg-degree-small-world" 2]
["prob-rewiring" [0 0.25 1]]
["payoffs" "[[1 0]\n [0 2]]"]
["prob-revision" 0.1 ]
["noise" 0.03]
["n-of-players-for-each-strategy" "[70 30]"]
["prob-link" 0.02]
["min-degree" 1]
```

We obtained the data in table format and, with the help of a pivot table (within an Excel spreadsheet), we easily created the following table:

| Network model | Average percentage of B-strategists | Std. error of the average[4] |
|---|---|---|
| Erdos-Renyi, with *prob-link* = 0.02 | 87.18% | 0.17% |
| Watts-Strogatz-small-world, with *avg-degree-small-world* = 2 and | | |
| —> *prob-rewiring* = 0 | 95.56% | 0.10% |
| —> *prob-rewiring* = 0.25 | 94.92% | 0.12% |
| —> *prob-rewiring* = 0.50 | 94.73% | 0.13% |
| —> *prob-rewiring* = 0.75 | 94.39% | 0.16% |
| —> *prob-rewiring* = 1.00 | 94.31% | 0.15% |
| preferential-attachment with *min-degree* = 1 | 93.95% | 0.23% |
| ring | 95.65% | 0.09% |
| star | 50.33% | 1.50% |
| path | 95.48% | 0.10% |

Looking at the table, we can see that under most network models, agents are able to coordinate on the efficient strategy regardless of the network structure, but there is one exception: the star network. See the following video:

> ▣ *One or more interactive elements has been excluded from this version of the text. You can view them online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=343#video-343-1*

Under this configuration, the population spends most of the time near one of the two monomorphic states (where all players are choosing the same strategy), with frequent switches between the two regimes. Thus, the 50.33% average does not denote that at tick 5000 we can expect to see half the population using strategy A (strategy 0 in the program) and the other half strategy B (strategy 1 in the program). What we can observe is most agents choosing strategy A with probability close to 50% and most agents choosing strategy B with probability close to 50%. This bimodal distribution explains the high standard error of the average, compared with the other networks.

---

4. The standard error of the average equals the standard deviation of the sample divided by the square root of the sample size (1000 in our case).

Thus, in this particular case,[5] it seems that, for most networks with average degree about 2, we can expect most agents to coordinate on efficient strategy B, but there is at least one network (e.g. the star network) for which this statement is not true.

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-imitate-if-better-networks.nlogo.

Exercise 1. In this chapter we have seen that if players are embedded on a star network, simulations with nxn-imitate-if-better-networks.nlogo with low (but strictly positive) *noise* will spend most of the time near one of the two monomorphic states (where all players are choosing the same strategy), with frequent switches between the two regimes. Furthermore, the long-run fraction of agents that use strategy B seems to be 50% Can you explain why this is the case?

Hint: When does the central player change strategy? When do the players at the periphery change strategy?
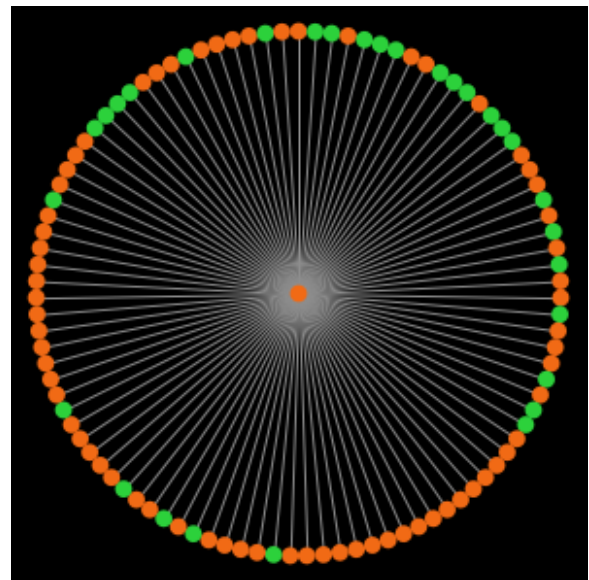


*A star network with 100 players*

Exercise 2. In this chapter we have implemented several network-generating algorithms. Some of these can generate a greater number of different networks than others. Can you fill in the following table assuming there are *N* **distinguishable** nodes? (i.e., a path network 1-2-3 is different from a path network 1-3-2). Assume *N* > 4.

---

5. We are assuming that agents use the *imitate if better* rule with *noise* = 0.03, *prob-revision* = 0.1, and initial strategy distribution is 70 A-strategists and 30 B-strategists.

| Network model | Number of different networks that can appear | Expected degree | Do all generated networks have the same average degree? | Do all nodes of all generated networks have the same degree? |
|---|---|---|---|---|
| Erdős–Rényi, with *prob-link* = $p > 0$ | | | | |
| ring | | | | |
| star | | | | |
| wheel | | | | |
| path | | | | |

To fill in the last three columns, consider the following notes:

**Expected degree** $(\mathrm{E}(d))$ is the expected degree of a node before the specific network is created. Expected degree is a property of the network model.

**Average degree** $(\mathrm{avg}(d))$ of a specific (already created) network is the average of every node's degree. In general, a network model may produce different networks, with potentially different average degrees. However, in some network models, all specific networks that are generated with that model have the same average degree; naturally, in those cases, the average degree is the same as the expected degree, but it is useful to think about whether every network generated by a certain network model has the same average degree or not.

Finally, in some network models, all nodes of all specific networks that are generated with that model have the same **degree** $(d)$. It is also useful to think about whether this is the case for each of the network models. Naturally, in those cases, the degree of every node is the same as the average degree of the specific network and the same as the expected degree of the network model.

**Exercise 3.** Try to answer the following questions:

1. It is possible that the Erdős–Rényi network model produces a ring network?
2. It is possible that the Erdős–Rényi network model produces a star network?
3. It is possible that the Erdős–Rényi network model produces a wheel network?
4. It is possible that the Erdős–Rényi network model produces a path network?
5. It is possible that the Watts-Strogatz network model produces a ring network?
6. It is possible that the Watts-Strogatz network model produces a star network?
7. It is possible that the Watts-Strogatz network model produces a wheel network?
8. It is possible that the Watts-Strogatz network model produces a path network?
9. Assume the number of nodes is *N*. Is the Erdős–Rényi network model with *prob-link* = 2/(*N*-1) the same as the Watts-Strogatz network model with *avg-degree-small-world* = 2 and *prob-rewiring* = 1?

**Exercise 4.** Can you implement procedure to build-star-network without using the nw extension?

> Hint to implement procedure to build-star-network without using the nw extension
>
> You will have to use primitive `create-links-with`.

**Exercise 5.** Can you implement procedure to build-ring-network without using the nw extension?

> Hint to implement procedure to build-ring-network without using the nw extension
>
> You may want to build a list with all the players using `sort` and then use primitive `foreach` over two lists. The following sketch may be of help:
> ```
> 1 ~ 2 ~ 3 ~ 4 ~ ... ~ (n-1) ~   n
> |   |   |   |           |       |
> n ~ 1 ~ 2 ~ 3 ~ ... ~ (n-2) ~ (n-1)
> ```
> You can create the second list from the first one using `fput`, `last` and `but-last`.

**Exercise 6.** Can you implement procedure to build-wheel-network without using the nw extension?

> Hint to implement procedure to build-wheel-network without using the nw extension
>
> You may want to start building a ring network.

# IV-3. Implementing network metrics

## 1. Goal

Our goal in this chapter is to include some network metrics in our model. These metrics will give us information about the structure of the generated network. In particular, we will compute:

- The network density, which is the number of links present in the network divided by the total number of links that could exist.

- The size of the largest component. A component is a maximal set of connected nodes, i.e. a maximal group of nodes such that there is a path from each node to every other node.

- The average local clustering coefficient. The local clustering coefficient of a node is the number of existing links between its neighbors divided by the total number of links that could possibly exist between them.[1] In a social network of friendships, this metric would measure the extent to which your friends are friends among themselves.

- The degree distribution, which shows the number of nodes that have degree $k$ (with $k$ = 0, 1, 2, 3...). As an example, figure 1 below shows the degree distribution of various networks with 100 nodes.[2]
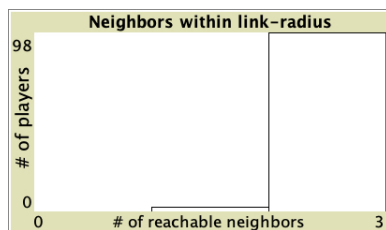


*Path network (2 at 1; 98 at 2)*
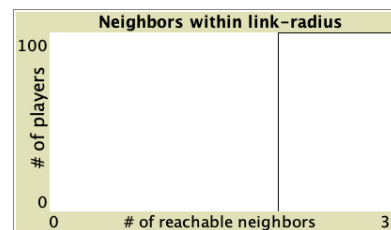


*Ring network (100 at 2)*
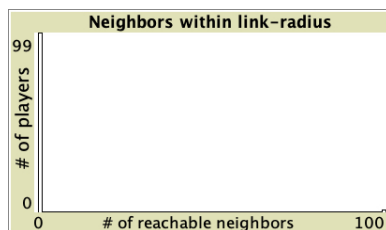


*Star network (99 at 1; 1 at 99)*
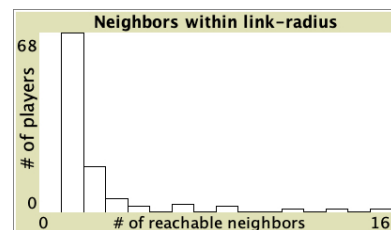


*Preferential attachment network*

---

1. By default, the local clustering coefficient is not defined for nodes with less than 2 neighbors. Here, we assume that the coefficient is 0 in those cases.

2. In these histograms, note that the bin for $x$ neighbors goes from $x$ to ($x$+1). For instance, in the distribution on the top left of figure 1, which corresponds to the path network, there are no nodes with degree 0, two nodes with degree 1, and 98 nodes with degree 2.
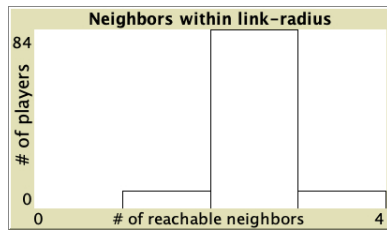
*Watts-Strogatz small-world network with average degree 2 and probability of rewiring = 0.1*
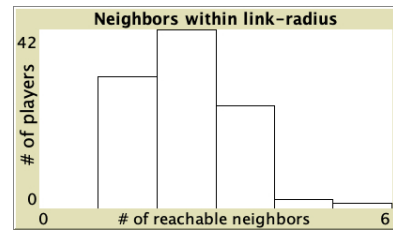


*Watts-Strogatz small-world network with average degree 2 and probability of rewiring = 0.5*
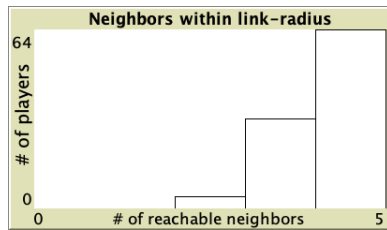


*10×10 square grid network (4 at 2; 32 at 3; 64 at 4)*
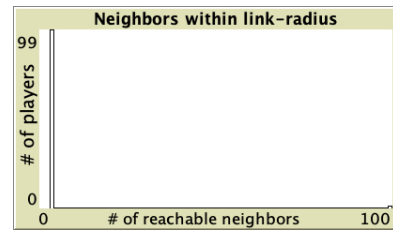


*Wheel network (99 at 3; 1 at 99)*

*Figure 1. Degree distribution of different types of networks with 100 nodes*

Note that the degree of a node is the number of nodes that are at (geodesic) distance 1; in our model we will also compute the number of nodes that are within distances greater than 1.

# 2. Motivation. Reassessing the significance of network structure

Let us revisit the 2-player 2-strategy single-optimum coordination game of the previous chapter:

| | | **Player 2** | |
|---|---|---|---|
| | | Player 2 chooses **A** | Player 2 chooses **B** |
| **Player 1** | Player 1 chooses **A** | **1 , 1** | **0 , 0** |
| | Player 1 chooses **B** | **0 , 0** | **2 , 2** |

In the previous chapter we saw that, in this game, under certain conditions,[3] a population of 100 agents embedded on a network with average degree about 2, will most likely approach the state where all agents choose strategy B and spend most of the time close to it, **regardless of the network structure**. The star network was an exception, but the result seems to hold for most network structures. Network structure did not seem to play a significant role in networks with very high

---

3. Conditions were that agents use the *imitate if better* rule with *noise* = 0.03, *prob-revision* = 0.1, and initial strategy distribution is 70 A-strategists and 30 B-strategists.

density either; in those networks, given our conditions, agents tend to approach the inefficient state and spend most of the time around there.

However, for moderately low densities (e.g. average degree about 10), network structure clearly plays a role, as you can see in the figures below. Figures 2 and 3 show two networks with average degree about 10, but completely different structure. Figure 2 shows a random network generated with the Erdős–Rényi model, and figure 3 shows a ring lattice. In the random network, most simulation runs approach the inefficient state and stay around it, while in the ring lattice, most simulations approach and stay around the efficient state.
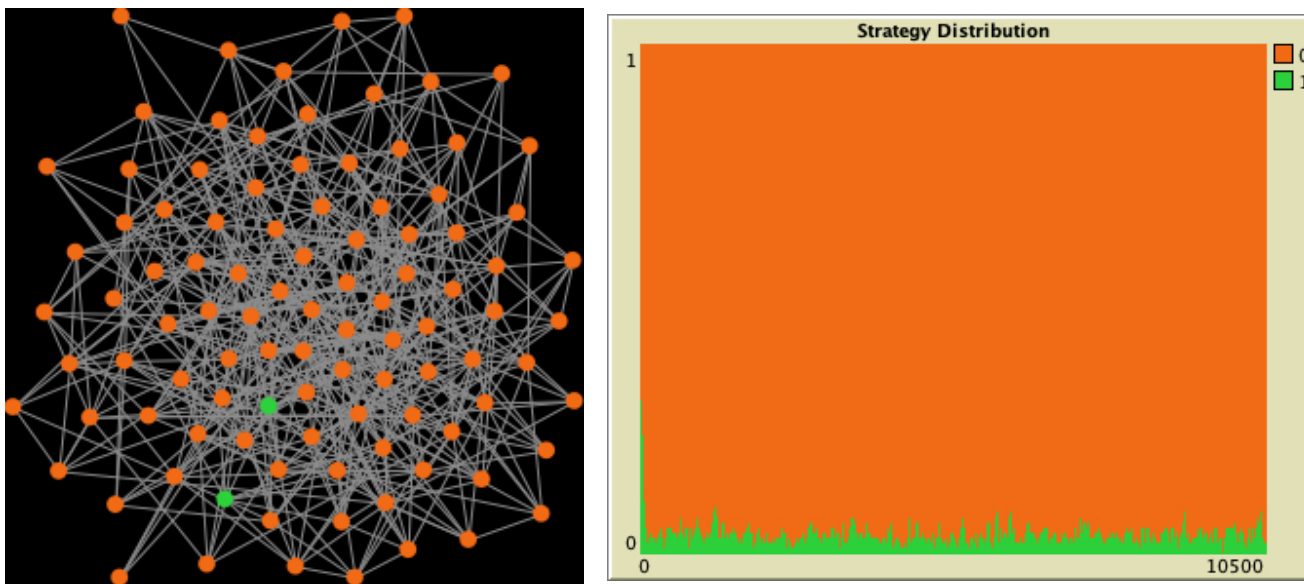


*Figure 2. On the left, an Erdős–Rényi random network with average degree about 10 (prob-link = 0.1). On the right, a representative run of our model in that network*
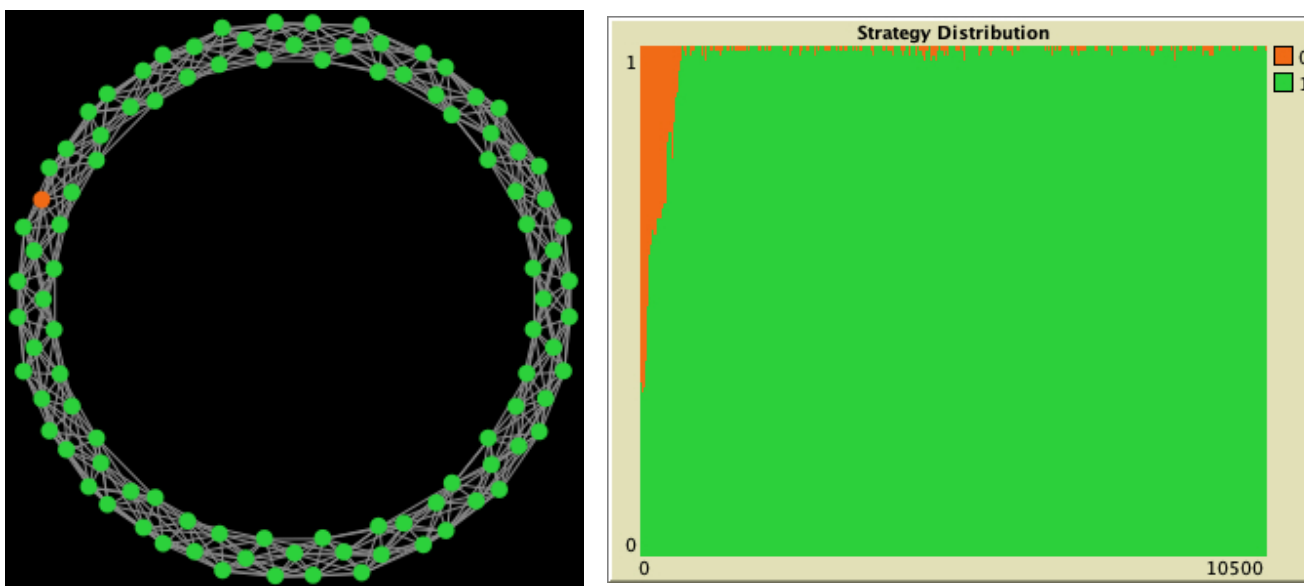


*Figure 3. On the left, a ring lattice where all players have degree 10, generated using the Watts–Strogatz model with prob-rewiring = 0. On the right, a representative run of our model in that network*

For those cases where the average degree is certainly not enough to predict whether the population

will approach the efficient state or not, we would like to explore whether there is any other property of the network that may help us predict the most likely outcome of the simulation. In particular, we hypothesize that the average local clustering coefficient may be useful, since this is a metric that is very different in the two networks shown in figures 2 and 3. For sufficiently large networks, the local clustering coefficient in Erdős–Rényi random networks is about *prob-link* for every node (so it is about 0.1 in the random network above),[4] while it is exactly $\frac{3\,(k-2)}{4\,(k-1)}$ for all nodes in ring lattices of degree $k$ (so it is exactly 2/3 ≈ 0.67 in the ring lattice above). So, does clustering help to approach the efficient state in our model?

Let us extend our model to explore this question!

# 3. Description of the model

We will not make any modification on the formal model our program implements. Thus, we refer you to the previous chapter to read the description of the model. The only paragraph we should add (about the program itself) is the following:

> The program computes the following metrics for the generated network:
>
> - The network density
>
> - The size of the largest component.
>
> - The average local clustering coefficient.
>
> - The histogram for the number of players within distance *link-radius* (which is a new parameter in the model). If *link-radius* = 1, this histogram shows the degree distribution. If *link-radius* = 2, the histogram gives information about how many players can be reached through 2 or fewer links. In general, this histogram shows the number of players that have *k* other players within distance *link-radius* (with *k* = 0, 1, 2, 3...).
>
> - The average number of (other) players within distance *link-radius*.

# 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).

---

4. In Erdős–Rényi random networks, the probability that *any* two nodes are neighbors equals *prob-link*.
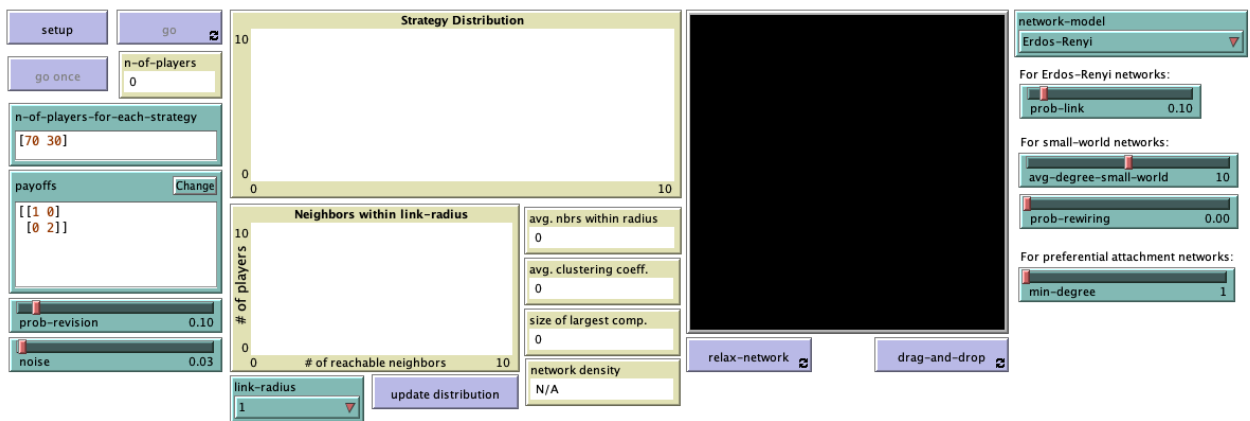
*Figure 4. Interface design*

The new interface (see figure 4 above) includes a new plot, a new chooser, a new button and four new monitors, all of them placed below the "Strategy Distribution" plot. To be precise, we have to add:

- One plot for the histogram. The horizontal axis should show the number of reachable neighbors (i.e., neighbors within distance *link-radius*) and the vertical axis should show how many players have the corresponding number of reachable neighbors.

Let us create a plot with the following setup:
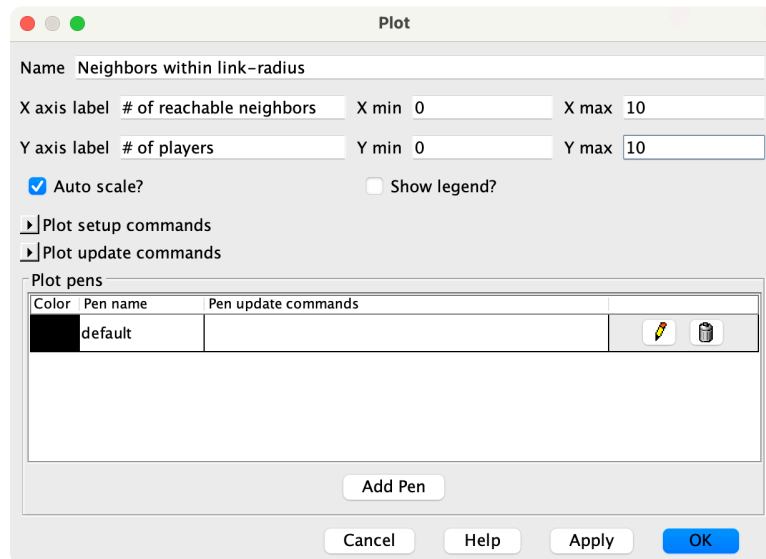


*Settings for plot "Neighbors within link-radius"*

And we have to make sure that we set up the default pen to draw bars, by clicking on the pencil symbol and setting the mode to "Bar":
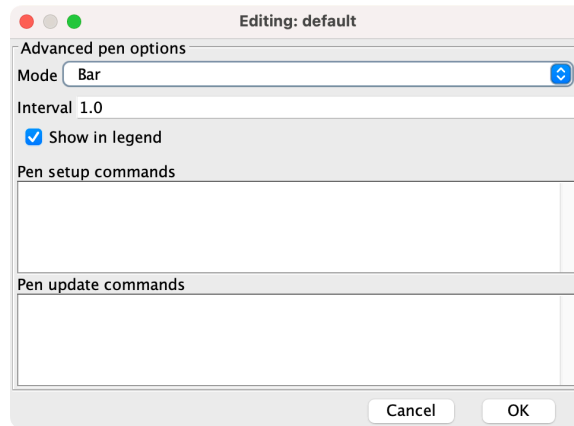
*Settings for the pencil in plot "Neighbors within link–radius"*

- One chooser for new parameter *link-radius*, with possible values 1, 2, 3, 4, 5, 10, 20 and "Infinity".

Let us create a chooser with the following setup:
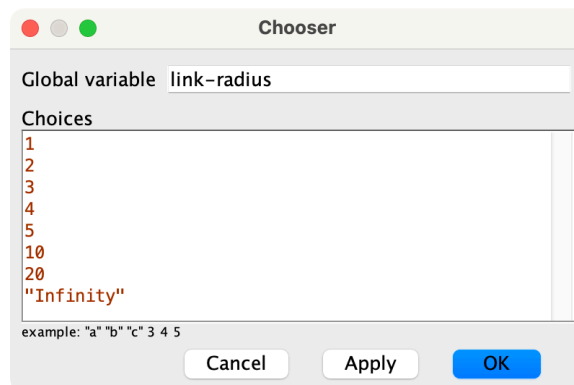


*Settings for the chooser of global variable link–radius*

- One button, to update the histogram.

In the Code tab, let us write the procedure to plot-accessibility, without including any code inside for now. This procedure will be in charge of updating the histogram.

```
to plot-accessibility
  ;; empty for now
end
```

Now we can create the button to run plot-accessibility. The user will have to click on this button after changing the value of parameter *link-radius*, so the new distribution is computed. Since this button only deals with informational aspects of the model, you may want to use the primitive `with-local-randomness`, which guarantees that this piece of code does not interfere with the generation of pseudorandom numbers for the rest of the model.

<div style="border:1px solid;">

**Button**

Agent(s) [ observer ⌄ ]   ☐ Forever

☐ Disable until ticks start

Commands
`with-local-randomness [plot-accessibility]`

Display name [ update distribution ]

Action key [ ]

[ Cancel ]   [ **OK** ]

</div>
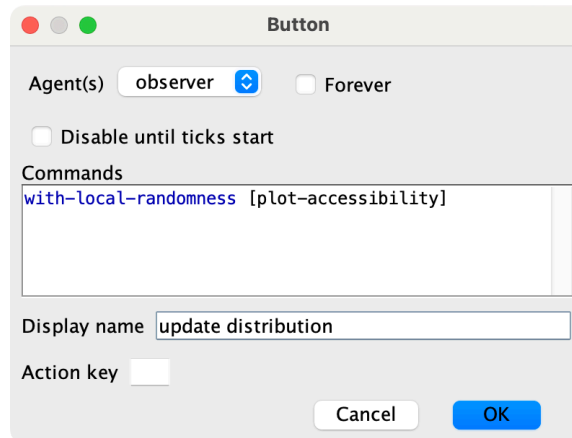
*Settings for the button to update plot "Neighbors within link-radius"*

- Four monitors

1. Let us create a monitor to show the average number of neighbors within distance *link-radius*. We will store this value in a new global variable named avg-nbrs-within-radius. Thus, before creating the monitor, please add this new global variable in the Code tab.

<div style="border:1px solid;">

**Monitor**

Reporter
`avg-nbrs-within-radius`

Display Name [ avg. nbrs within radius ]

Decimal places [ 3 ]   Font Size [ 11 ]
full precision is 17

[ Cancel ]   [ **OK** ]

</div>
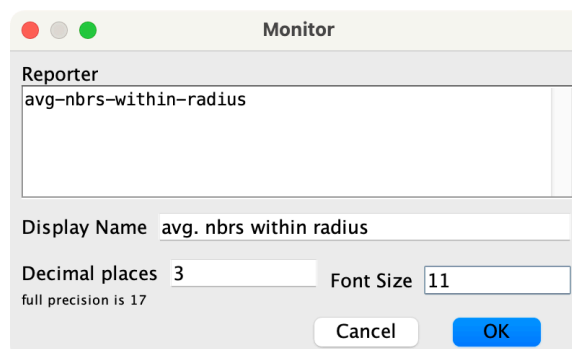
*Settings for the monitor of global variable avg-nbrs-within-radius*

2. Let us create a monitor to show the average local clustering coefficient. We

will store this value in a new global variable named avg-clustering-coefficient. Thus, before creating the monitor, please add this new global variable in the Code tab.
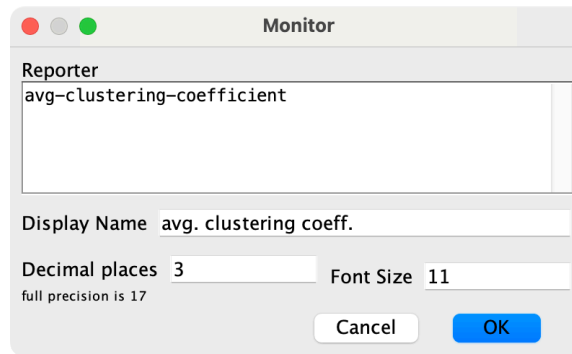


*Settings for the monitor of global variable avg-clustering-coefficient*

3. Let us create a monitor to show the size of the largest component. We will store this value in a new global variable named size-of-largest-component. Thus, before creating the monitor, please add this new global variable in the Code tab.
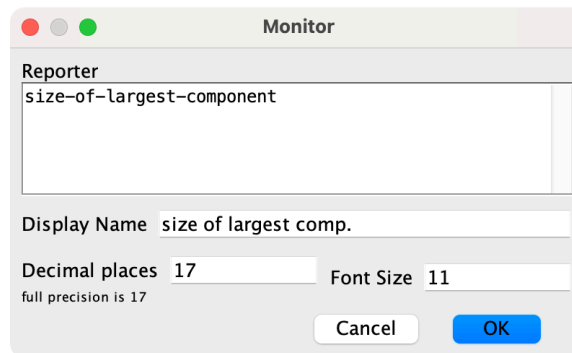


*Settings for the monitor of global variable size-of-largest-component*

4. And finally, let us create a monitor to show the network density. The maximum number of undirected links in a $N$-node simple network is $\binom{N}{2}$, so the density of a simple undirected network with $l$ links is $\frac{l}{\binom{N}{2}} = \frac{2\,l}{N(N-1)}$.[5] Since the formula to compute the density is very simple, we can write it up directly in the monitor window.

---

5. A simple network is one without self-links and with no more than one link between any pair of nodes.

Settings for the monitor of the network density
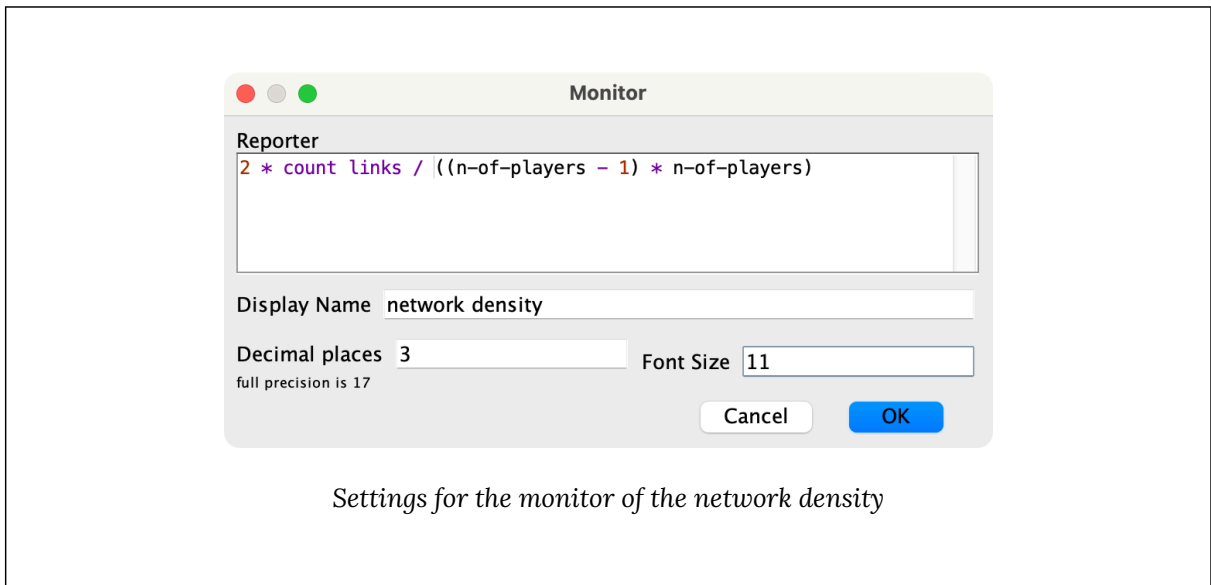
# ▶ 5. Code

## 5.1. Skeleton of the code

To keep our code beautiful and tidy, we will gather all the computations of the network metrics in a new procedure named to compute-network-metrics. It makes sense to run this new procedure soon after the network has been created, in procedure to setup (see fig. 5), since the network does not change over the course of the simulation. Procedure to compute-network-metrics will call three new procedures:

- to plot-accessibility, which will compute and show the histogram for the number of players within distance *link-radius*, and will also set the value of avg-nbrs-within-radius.

- to compute-avg-clustering-coefficient, which will compute the value of avg-clustering-coefficient.

- to compute-size-of-largest-component, which will compute the value of size-of-largest-component.
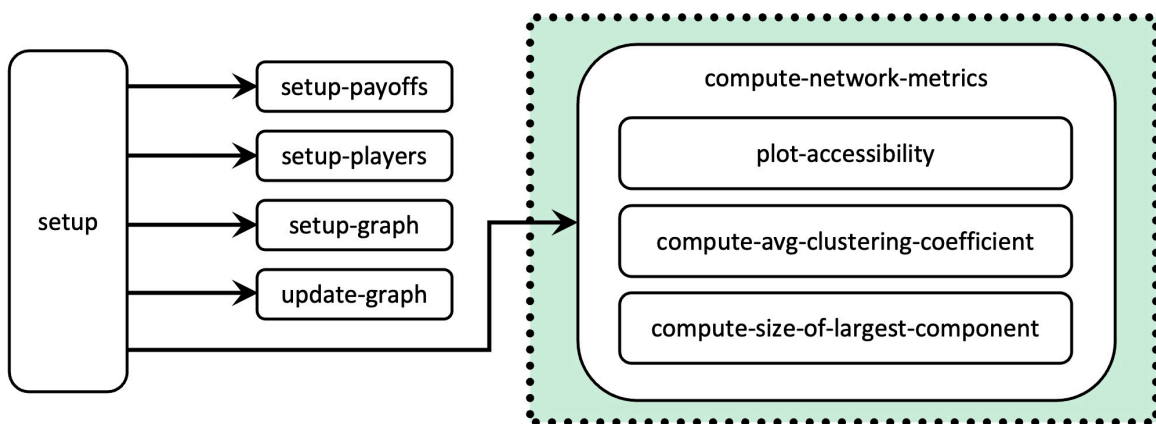


Figure 5. Skeleton of procedure to setup

## 5.2. Global variables and individually-owned variables

### Global variables

If you have already added global variables avg-nbrs-within-radius, avg-clustering-coefficient, and size-of-largest-component, then there is no need to add or remove any other global variables in our code. The current global variables are:

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players

  avg-nbrs-within-radius       ;; <== new line
  avg-clustering-coefficient  ;; <== new line
  size-of-largest-component   ;; <== new line
]
```

### Individually-owned variables

There is no need to add or remove any individually-owned variables in our code.

## 5.3. Procedure to compute-network-metrics

Procedure to compute-network-metrics is just a container that gathers all the procedures in charge of computing the different network metrics (see fig. 5). We use this container to keep our code nice and modular. Its implementation is particularly simple:

```
to compute-network-metrics
  plot-accessibility
  compute-avg-clustering-coefficient
  compute-size-of-largest-component
end
```

As shown in fig. 5, we should call this new procedure at the end of setup:

```
to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
  compute-network-metrics  ;; <== new line
end
```

## 5.4. Procedures to compute network metrics

In this section we implement the three procedures that compute the different network metrics.

### to plot-accessibility

This procedure should compute and plot the accessibility histogram (i.e. the "Neighbors within link-radius" histogram). To do this, the reporter `nw:turtles-in-radius` is very useful, since it returns the set of neighbors that are within a certain (geodesic) distance of the calling player in the network. Thus, we just have to count the number of neighbors in this set for every player, after having set the required radius to *link-radius*:

```
to plot-accessibility

  let n-of-nbrs-of-each-player
    [(count nw:turtles-in-radius link-radius) - 1] of players
      ;; - 1 because nw:turtles-in-radius includes the calling player

end
```

Note, however, that parameter *link-radius* may take the value "Infinity", to see the number of players that players can reach through any number of links. To compute this, note that the greatest possible distance between any two players connected in the network is the number of players in the network minus 1. Thus, if *link-radius* equals "Infinity", we may call reporter `nw:turtles-in-radius` with the value of (n-of-players – 1). To implement this elegantly, we define a local variable named steps, as follows.

```
to plot-accessibility
  let steps link-radius
  if link-radius = "Infinity" [set steps (n-of-players - 1)]

  let n-of-nbrs-of-each-player
    [(count nw:turtles-in-radius steps) - 1] of players
      ;; - 1 because nw:turtles-in-radius includes the calling player
end
```

Now that we have the list of the number of reachable neighbors for each player, we can plot the histogram using the `histogram` command. Note, however, that there is an issue we have to be careful about when using `histogram`. In the produced histogram, the height of the bin that goes from *x* to (*x*+1) will be the frequency of all the values in the range [*x*, *x*+1), which in this case is just the integer value *x*. Thus, to include a bin for the maximum value in the list, we should set the upper limit of the horizontal range of the plot to the **maximum value of the list plus 1**. If we do not add this 1, we will not see the bin corresponding to the maximum number in the list.

Finally, in this procedure we should also compute the mean of the distribution and store it in global variable avg-nbrs-within-radius.

```
to plot-accessibility
  let steps link-radius
  if link-radius = "Infinity" [set steps (n-of-players - 1)]
  let n-of-nbrs-of-each-player
    [(count nw:turtles-in-radius steps) - 1] of players


  let max-n-of-nbrs-of-each-player max n-of-nbrs-of-each-player
  set-current-plot "Neighbors within link-radius"
  set-plot-x-range 0 (max-n-of-nbrs-of-each-player + 1)
                        ;; + 1 to make room for the width of the last bar
  histogram n-of-nbrs-of-each-player
  set avg-nbrs-within-radius mean n-of-nbrs-of-each-player
end
```

## to compute-avg-clustering-coefficient

Have a look at the documentation of reporter `nw:clustering-coefficient` and try to implement this procedure.

Implementation of procedure to compute-avg-clustering-coefficient

```
to compute-avg-clustering-coefficient
  set avg-clustering-coefficient
      mean [ nw:clustering-coefficient ] of players
end
```

## to compute-size-of-largest-component

Have a look at the documentation of reporter `nw:weak-component-clusters` and try to implement this procedure.

Implementation of procedure to compute-size-of-largest-component

```
to compute-size-of-largest-component
  set size-of-largest-component max map count nw:weak-component-clusters
end
```

## 5.5. Other procedures

Note that there is no need to modify the code of any other procedure.

## 5.6. Complete code in the Code tab

We have finished our model!

```
extensions [nw]

globals [
  payoff-matrix
  n-of-strategies
  n-of-players

  avg-nbrs-within-radius
  avg-clustering-coefficient
  size-of-largest-component
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]

;;;;;;;;;;;;;;
;;; SETUP ;;;
;;;;;;;;;;;;;;

to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
  compute-network-metrics
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
end

to setup-players
  let initial-distribution
      read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
```

```
        "n-of-players-for-each-strategy (i.e. "
        length initial-distribution "):\n"
        n-of-players-for-each-strategy
        "\nshould be equal to the number of rows\n"
        "in the payoff matrix (i.e. "
        length payoff-matrix "):\n"
        payoffs
      )
    ]

  set n-of-players sum initial-distribution
  ifelse n-of-players < 4
    [ user-message "There should be at least 4 players" ]
    [
      build-network

      ask players [set strategy -1]
      let i 0
      foreach initial-distribution [ j ->
        ask up-to-n-of j players with [strategy = -1] [
          set payoff 0
          set strategy i
          set strategy-after-revision strategy
        ]
        set i (i + 1)
      ]

      set n-of-players count players
      update-players-color
    ]
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; NETWORK CONSTRUCTION ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;

to build-network
  set-default-shape players "circle"
  run (word "build-" network-model "-network")
  ask players [fd 15]
end

to build-Erdos-Renyi-network
  nw:generate-random players links n-of-players
     prob-link
end

to build-Watts-Strogatz-small-world-network
 nw:generate-watts-strogatz players links n-of-players
    (avg-degree-small-world / 2) prob-rewiring
end

to build-preferential-attachment-network
```

```
    nw:generate-preferential-attachment players links n-of-players
        min-degree
  end

  to build-ring-network
    nw:generate-ring players links n-of-players
  end

  to build-star-network
    nw:generate-star players links n-of-players
  end

  to build-grid-4-nbrs-network
    let players-per-line (floor sqrt n-of-players)
    nw:generate-lattice-2d players links
        players-per-line players-per-line false
  end

  to build-wheel-network
    nw:generate-wheel players links n-of-players
  end

  to build-path-network
    build-ring-network
    ask one-of links [die]
  end

  ;;;;;;;;;;
  ;;; GO ;;;
  ;;;;;;;;;;

  to go
    ask players [update-payoff]
    ask players [
      if (random-float 1 < prob-revision) [
        update-strategy-after-revision
      ]
    ]
    ask players [update-strategy]

    tick
    update-graph
    update-players-color
  end

  ;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;; UPDATE PROCEDURES ;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;

  to update-payoff
    if any? link-neighbors [
      let mate one-of link-neighbors
      set payoff
```

```
          item ([strategy] of mate) (item strategy payoff-matrix)
  ]
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      if any? link-neighbors [
        let observed-player one-of link-neighbors
        if ([payoff] of observed-player) > payoff [
          set strategy-after-revision
              ([strategy] of observed-player)
        ]
      ]
    ]
end

to update-strategy
  set strategy strategy-after-revision
end

;;;;;;;;;;;;;
;;; PLOTS ;;;
;;;;;;;;;;;;;

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers

  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end

to update-players-color
  ask players [set color 25 + 40 * strategy]
```

```
  end

;;;;;;;;;;;;;;;;;;;;;;;;
;;; NETWORK METRICS ;;;
;;;;;;;;;;;;;;;;;;;;;;;;

to compute-network-metrics
  plot-accessibility
  compute-avg-clustering-coefficient
  compute-size-of-largest-component
end

to plot-accessibility
  let steps link-radius
  if link-radius = "Infinity" [set steps (n-of-players - 1)]
  let n-of-nbrs-of-each-player
    [(count nw:turtles-in-radius steps) - 1] of players

  let max-n-of-nbrs-of-each-player max n-of-nbrs-of-each-player
  set-current-plot "Neighbors within link-radius"
  set-plot-x-range 0 (max-n-of-nbrs-of-each-player + 1)
                         ;; + 1 to make room for the width of the last bar
  histogram n-of-nbrs-of-each-player
  set avg-nbrs-within-radius mean n-of-nbrs-of-each-player
end

to compute-avg-clustering-coefficient
  set avg-clustering-coefficient
      mean [ nw:clustering-coefficient ] of players
end

to compute-size-of-largest-component
  set size-of-largest-component max map count nw:weak-component-clusters
end

;;;;;;;;;;;;;;;
;;; LAYOUT ;;;
;;;;;;;;;;;;;;;

;; Procedures taken from Wilensky's (2005a) NetLogo Preferential
;; Attachment model
;; http://ccl.northwestern.edu/netlogo/models/PreferentialAttachment
;; and Wilensky's (2005b) Mouse Drag One Example
;; http://ccl.northwestern.edu/netlogo/models/MouseDragOneExample
to relax-network
  ;; the number 3 here is arbitrary; more repetitions slows down the
  ;; model, but too few gives poor layouts
  repeat 3 [
    ;; the more players we have to fit into
    ;; the same amount of space, the smaller
    ;; the inputs to layout-spring we'll need to use
    let factor sqrt count players
    ;; numbers here are arbitrarily chosen for pleasing appearance
```

```
    layout-spring players links
                (1 / factor) (7 / factor) (3 / factor)
    display  ;; for smooth animation
  ]
  ;; don't bump the edges of the world
  let x-offset max [xcor] of players + min [xcor] of players
  let y-offset max [ycor] of players + min [ycor] of players
  ;; big jumps look funny, so only adjust a little each time
  set x-offset limit-magnitude x-offset 0.1
  set y-offset limit-magnitude y-offset 0.1
  ask players [ setxy (xcor - x-offset / 2) (ycor - y-offset / 2) ]
end

to-report limit-magnitude [number limit]
  if number > limit [ report limit ]
  if number < (- limit) [ report (- limit) ]
  report number
end

to drag-and-drop
  if mouse-down? [
    let candidate min-one-of players
        [distancexy mouse-xcor mouse-ycor]
    if [distancexy mouse-xcor mouse-ycor] of candidate < 1 [
    ;; The WATCH primitive puts a "halo" around the watched turtle.
      watch candidate
      while [mouse-down?] [
        ;; If we don't force the view to update, the user won't
        ;; be able to see the turtle moving around.
        display
        ;; The SUBJECT primitive reports the turtle being watched.
        ask subject [ setxy mouse-xcor mouse-ycor ]
      ]
      ;; Undoes the effects of WATCH.
      reset-perspective
    ]
  ]
end
```

# 6. Sample runs

Now that we have the model, we can investigate the question we posed at the motivation section above, i.e.: is the (average local) clustering coefficient of a network useful to predict the likelihood of approaching the efficient state?

To investigate this question, the Watts–Strogatz model is very convenient, since it allows us to create networks with a fixed average degree, but with different clustering, simply by modifying the value of *prob-rewiring*. Figure 6 below shows several box plots of the clustering coefficient for different networks of average degree 10, obtained by running the Watts–Strogatz model with values of *prob-*

*rewiring* equal to 0, 0.1, 0.2, …, 1. As you can see in figure 6, the greater the value of *prob-rewiring*, the lower the clustering.
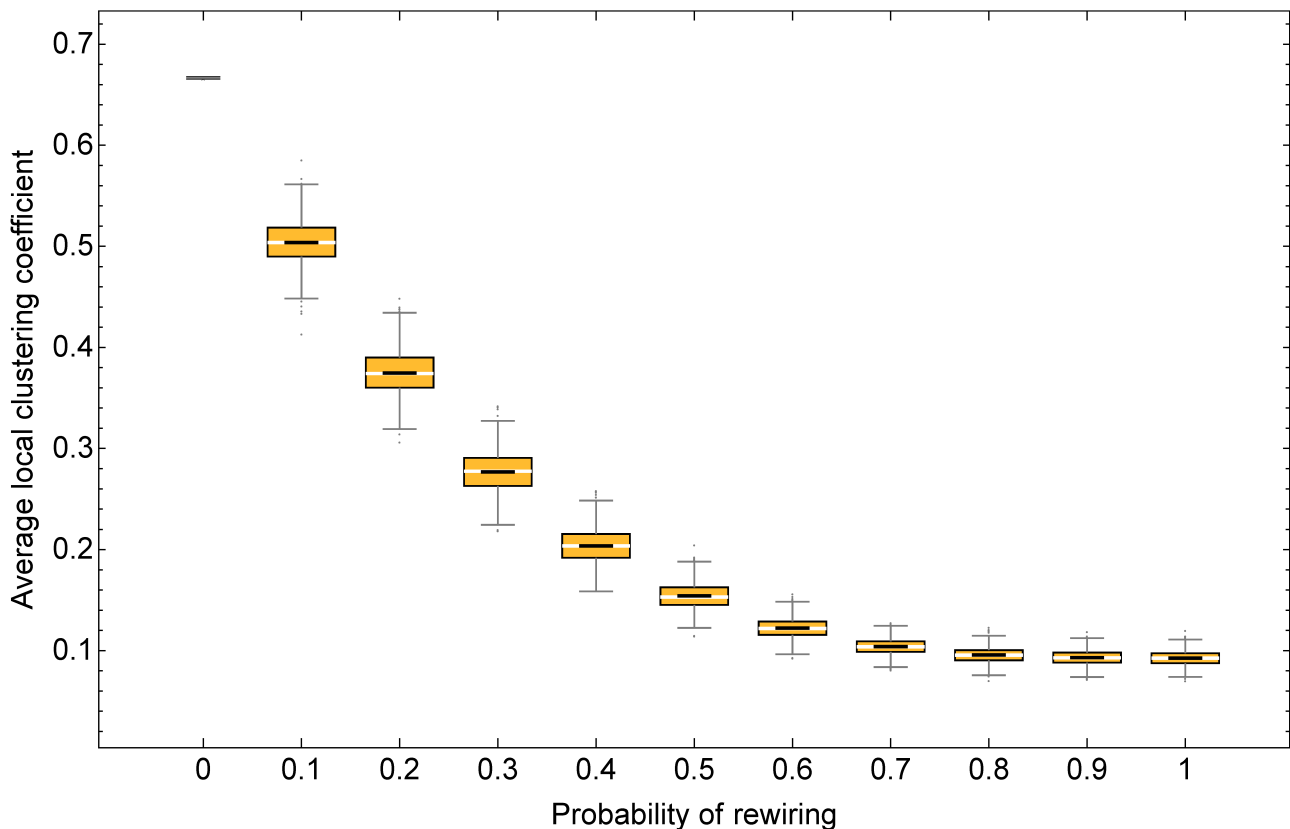


*Figure 6. Box plots of the (average local) clustering coefficient for different networks obtained with the Watts–Strogatz model, using avg-degree-small-world = 10 and different values of prob-rewiring. Each box plot summarizes a set of 1000 networks created with a certain value of prob-rewiring. The white line marks the median of the sample distribution, while the black line marks the mean*

Our goal now is to investigate whether clustering favors approaching the efficient state in networks of average degree 10 under our baseline conditions (i.e.: *noise* = 0.03, *prob-revision* = 0.1, and initial strategy distribution [70 30]). For that, we can run a computational experiment on Watts–Strogatz networks like the ones shown in figure 6 and, for each run, record both the average local clustering coefficient and the percentage of B-strategists by tick 5000.

Try to set up this experiment by yourself. You can find our setup below.

Experiment setup

We have set up the following experiment in BehaviorSpace:

*Experiment to study the effect of clustering*

The full setting of variables for this experiment is:

```
["network-model" "Watts-Strogatz-small-world"]
["avg-degree-small-world" 10]
["prob-rewiring" [0 0.1 1]]
["payoffs" "[[1 0]\n [0 2]]"]
["n-of-players-for-each-strategy" "[70 30]"]
["prob-revision" 0.1]
["noise" 0.03]
["prob-link" 0.1]
["min-degree" 5]
["link-radius" 1]
```

We ran this experiment and, looking at the data, it was clear that –at tick 5000– most simulations were at one of two possible regimes: an inefficient one, where very few players (≤ 5%) were B-strategists, and an efficient regime, where most players (≥ 95%) were B-strategists. Thus, we decided to compute the fraction of runs in each of these two regimes, and also the fraction of those runs in

none of the two regimes.[6] These fractions are shown in the following table, for different values of the (rounded) average local clustering coefficient, together with the standard errors (in brackets):[7]

| Rounded average local clustering coefficient | Total number of runs | Percentage of runs where the percentage of B-strategists is no more than 5% (inefficient regime) | Percentage of runs where the percentage of B-strategists is in the range (5%, 95%) | Percentage of runs where the percentage of B-strategists is no less than 95% (efficient regime) |
|---|---|---|---|---|
| 0.1 | 5380 | 84.14% (0.50%) | 1.64% (0.17%) | 14.22% (0.48%) |
| 0.2 | 1710 | 79.36% (0.98%) | 1.81% (0.32%) | 18.83% (0.95%) |
| 0.3 | 1037 | 69.33% (1.43%) | 2.12% (0.45%) | 28.54% (1.40%) |
| 0.4 | 879 | 53.58% (1.68%) | 2.84% (0.56%) | 43.57% (1.67%) |
| 0.5 | 980 | 26.73% (1.41%) | 1.53% (0.39%) | 71.73% (1.44%) |
| 0.6 | 14 | 14.29% (9.71%) | 0.00% (0.00%) | 85.71% (9.71%) |
| 0.7 | 1000 | 7.00% (0.81%) | 1.90% (0.43%) | 91.10% (0.90%) |

*Table 1. Results of a computational experiment of our model run on Watts–Strogatz networks of avg-degree-small-world = 10 and values of prob-rewiring equal to 0, 0.1, 0.2, ..., 1. Baseline conditions: noise = 0.03, prob-revision = 0.1, and initial strategy distribution [70 30]*

The main insights from the data shown in table 1 above are summarized in figure 7 below.

---

6. This is something that can be easily done in an Excel spreadsheet, by defining a column for each regime, such that the value of the corresponding row equals 1 if the run is in the regime and 0 otherwise. The average of this column is the fraction of runs at the regime.

7. The frequency of the event "the population is at a certain regime" calculated over $n$ simulation runs can be seen as the mean of a sample of $n$ i.i.d. Bernouilli random variables where success denotes that the event occurred and failure denotes that it did not. Thus, the frequency $f$ is the maximum likelihood (unbiased) estimator of the exact probability with which the event occurs. The standard error of the calculated frequency $f$ is the standard deviation of the sample divided by the square root of the sample size. In this particular case, the formula reads:

$$\text{Std. error } (f, n) = (f\,(1-f)\,/\,(n-1))^{1/2}$$
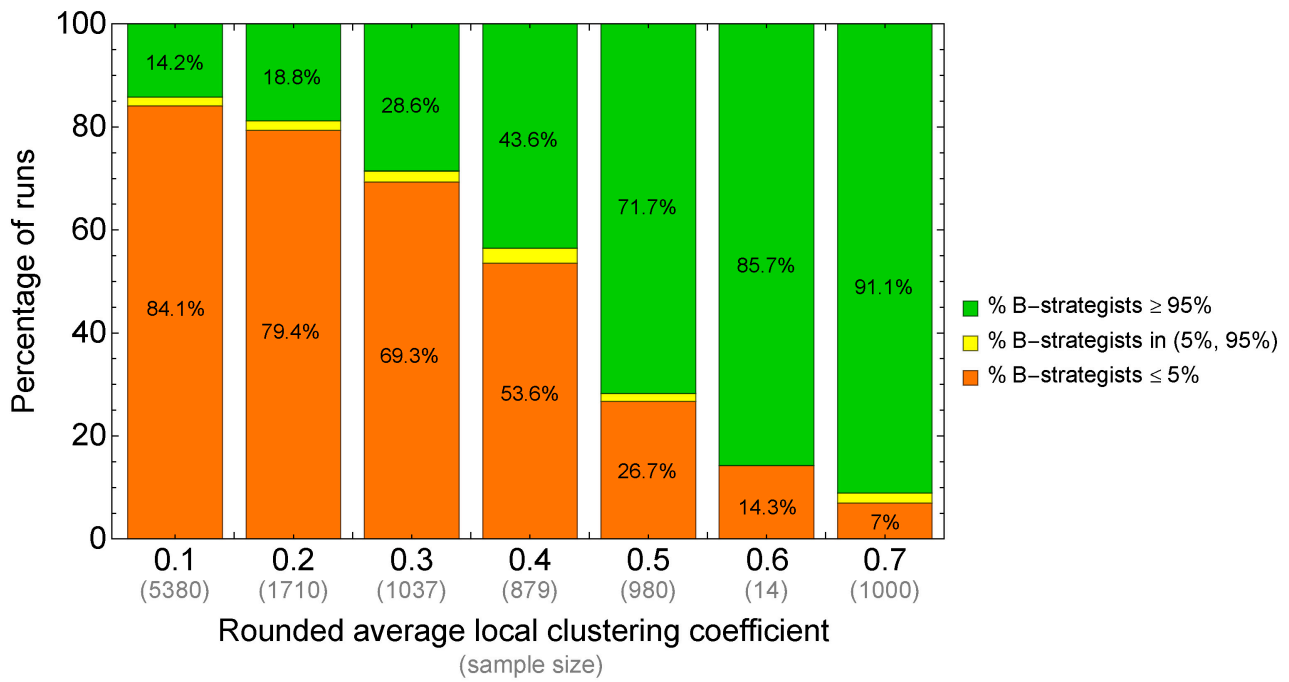
*Figure 7. Summary of the results reported in table 1*

Looking at figure 7 it is clear that, as suspected, a higher clustering coefficient seems to increase the likelihood of approaching the efficient regime. However, note that this experiment has only explored networks obtained with the Watts–Strogatz model, so we do not really know whether this relation holds for other network models.

To investigate whether similar results hold for other networks, we run the same experiment as before, but with Erdős–Rényi networks and preferential attachment networks. The clustering coefficient obtained in 1000 networks created with each of these models are shown in figure 8 below:
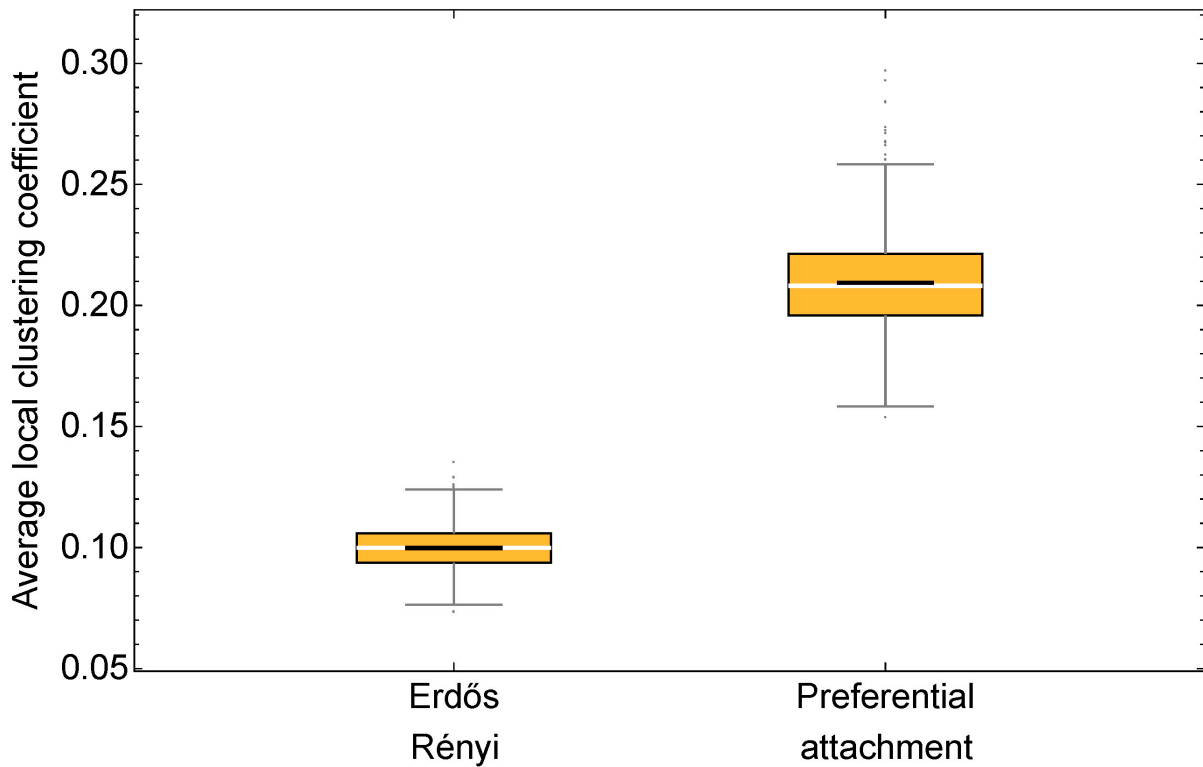
*Figure 8. Box plots of the (average local) clustering coefficient for 1000 networks with 100 nodes. On the left, 1000 networks created with the Erdős–Rényi model, using prob-link = 0.1. On the right, 1000 preferential-attachment networks created with the Barabási–Albert model, using min-degree = 5. In each box plot, the white line marks the median of the sample distribution, while the black line marks the mean*

The rounded average local clustering coefficient of every Erdős–Rényi network in figure 8 is 0.1, while this value is 0.2 for 970 of the 1000 preferential-attachment networks and 0.3 for the other 30. Let us now see the proportion of runs at each of the regimes (standard errors in brackets):

| Network model | Rounded average local clustering coefficient | Total number of runs | Percentage of runs where the percentage of B-strategists is no more than 5% (inefficient regime) | Percentage of runs where the percentage of B-strategists is in the range (5%, 95%) | Percentage of runs where the percentage of B-strategists is no less than 95% (efficient regime) |
|---|---|---|---|---|---|
| Erdős–Rényi | 0.1 | 1000 | 82.20% (1.21%) | 1.50% (0.38%) | 16.30% (1.17%) |
| Preferential attachment | 0.2 | 970 | 81.96% (1.24%) | 2.37% (0.49%) | 15.67% (1.17%) |
| Preferential attachment | 0.3 | 30 | 73.33% (8.21%) | 0.00% (0.00%) | 26.67% (8.21%) |

*Table 2. Results of a computational experiment of our model run on 1000 Erdős–Rényi networks with prob-link = 0.1 and 1000 preferential attachment networks with min-degree = 5. Baseline conditions: noise = 0.03, prob-revision = 0.1, and initial strategy distribution [70 30]*

The main insights from the data shown in table 2 above are summarized in figure 9 below.
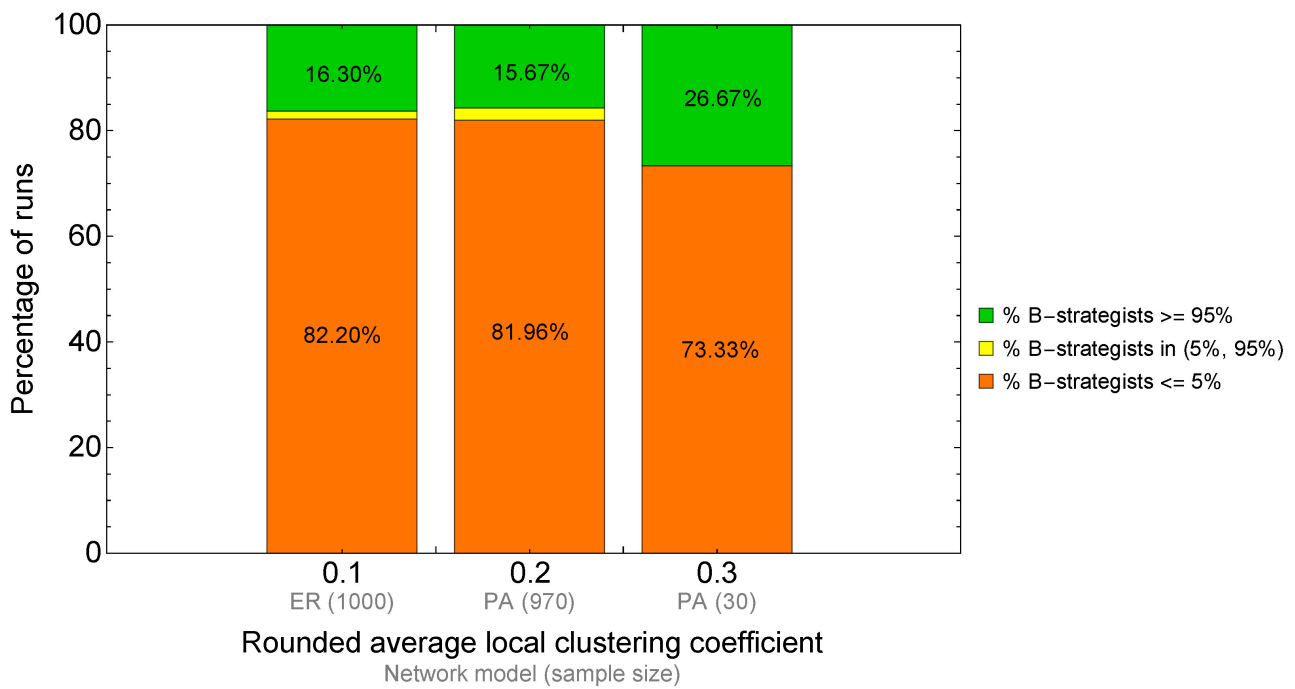
*Figure 9. Summary of the results reported in table 2*

Comparing figure 7 and figure 9, it seems that the results obtained for Watts–Strogatz networks apply reasonably well to Erdős–Rényi networks and preferential attachment networks too. However, it is important not to draw conclusions beyond these network models. It is not difficult to design networks where the effect of clustering differs significantly from that observed in the network models above. The following two examples illustrate this observation.

Let us start with a network with low clustering but with a high proportion of runs at the efficient regime by tick 5000. Consider the following method to build a (regular) undirected network: place the nodes in a circle, and link every node to its 10 closest *spatial* neighbors, *after having ignored the four closest ones* (i.e. ignore the two closest nodes at each side). The links of node 0 in this network are illustrated in figure 10. This network is similar to a ring lattice, but there is a gap of two nodes at either side of every node. Thus, we call this network a gap-2 ring lattice of degree 10.
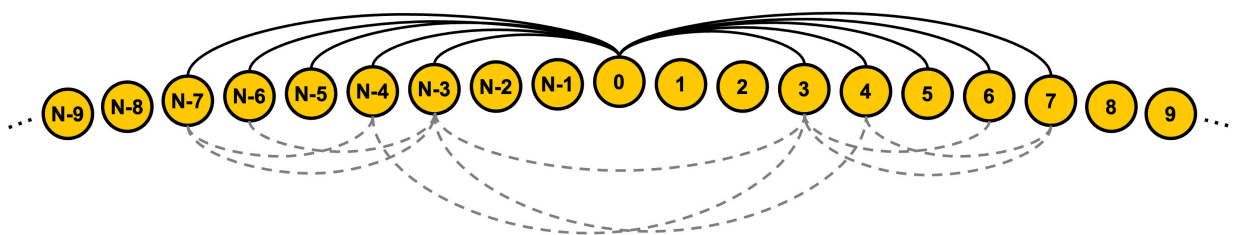


*Figure 10. Sketch of a gap-2 ring lattice of degree 10. In black, the 10 links of node 0. In grey, the 9 links that exist between the neighbors of node 0*

The local clustering coefficient of every node in this network is $\frac{9}{\binom{10}{2}} = 0.2$. Thus, if this was a Watts–Strogatz network or a preferential attachment network, we would expect that approximately 80% of the runs would be at the inefficient regime by tick 5000 (see figure 7 and figure 9). However, we have run a 1000-run experiment to assess this, and the actual proportion of runs at the inefficient

regime by tick 5000 in this type of network is only approximately 35%. It is more likely to reach the efficient regime (about 63% of the runs do it), even though the clustering coefficient is only 0.2.

Our second example illustrates the other extreme: a network with high clustering coefficient but with a low proportion of runs at the efficient regime by tick 5000. For this, we built a complete network with 27 nodes, and then we added 73 nodes; each of the new 73 nodes links to two random nodes within the set of the 27 original nodes (see figure 11).
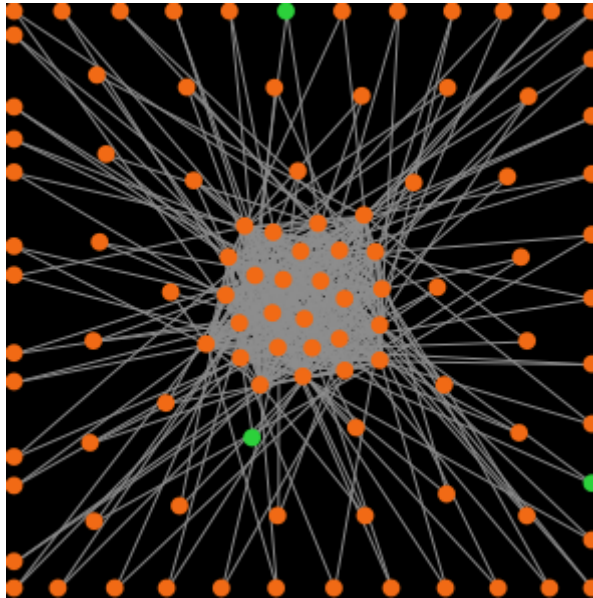


*Figure 11. A complete network with 27 nodes, to which we add 73 extra nodes, each with two neighbors in the original complete network*

The average degree of this network is $\frac{2\left(\binom{27}{2}+2*73\right)}{100}=9.94$, and the average local clustering coefficient was greater than 0.9 in all the networks created for our 1000-run experiment. With such a high clustering coefficient, one would expect that more than 90% of the runs would approach the efficient regime (see figure 7). However, in our 1000-run experiment, only 23.20% of the runs were at the efficient regime by tick 5000 (and 74.70% were at the inefficient regime).

To conclude, it is important to realize that each network model implies a probability distribution over a set of networks, and the mass of this probability distribution is often concentrated on a very small proportion of networks. To put things in perspective, note that the set of all possible undirected networks with $N$ distinguishable nodes is $2^{\binom{N}{2}}$. For $N=24$, this number is already greater than the number of protons in the observable universe (i.e. the Eddington number). When we create networks in our computers –using any model–, we are inevitably exploring a tiny proportion of all the possible networks that exist, and most often this proportion will not be representative of all the possible networks. Thus, one has to be very cautious when formulating statements about the effect of any network metric on any dynamic, if the statements are only based on simulations.

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-imitate-if-better-networks-metrics.nlogo.

**Exercise 1.** Consider a ring lattice (i.e. the network generated with the Watts–Strogatz model using *prob-rewiring* = 0) where nodes have (even) degree $k$. Assume that the number of nodes is greater than $\frac{3\,k}{2}$. Can you prove that every node's local clustering coefficient is exactly $\frac{3\,(k-2)}{4\,(k-1)}$?

**Exercise 2.** Can you implement a procedure to build ring lattices with arbitrary even *degree* without using the nw extension?
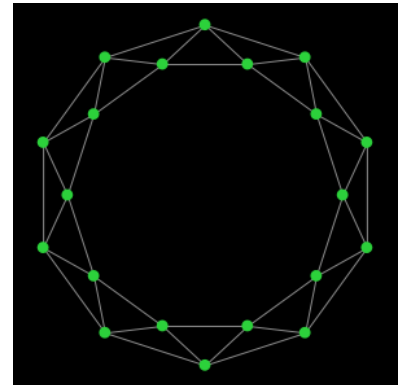


*A ring lattice with 20 nodes and degree 4*

> Hint to implement a procedure to build ring lattices without using the nw extension

```
create-link-with player ((who + i) mod n-of-players)
```

**Exercise 3.** In this chapter we defined a type of network that we named gap-2 ring lattice of degree 10 (see figure 10). Can you implement a procedure to build networks of this type with arbitrary even *degree* and arbitrary *gap*?

**Exercise 4.** Using the code you have created in exercise 3, run a BehaviorSpace experiment to fill in the following table for gap-*x* ring lattices of 100 nodes with degree 10:

| Gap | Local clustering coefficient of every node | Percentage of runs where the percentage of B-strategists is no more than 5% (inefficient regime) | Percentage of runs where the percentage of B-strategists is in the range (5%, 95%) | Percentage of runs where the percentage of B-strategists is no less than 95% (efficient regime) |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |

Exercise 5. In this chapter we built a complete network with 27 nodes. Then, we added 73 extra nodes and linked each of these new nodes to two random nodes within the set of the 27 original nodes (see figure 11). Can you implement a procedure to build this network?

Exercise 6. Using the code you have created in exercise 5, replicate the 1000-run BehaviorSpace experiment that allowed us to say that in that network "only 23.20% of the runs were at the efficient regime by tick 5000 (and 74.70% were at the inefficient regime)".

# IV-4. Other ways of computing payoffs and other decision rules

# 1. Goal

Our goal in this chapter is to extend the model we have created in the previous chapter by adding two features that can have a major impact on the dynamics of evolutionary models run on networks:

- The possibility to model different ways of computing payoffs. In our current model, agents obtain their payoffs by playing with **one** of their neighbors, chosen at random. This means that every agent obtains just one payoff, regardless of their network degree. In this chapter we will allow for the possibility that agents play with **all** their neighbors. When agents play with all their neighbors, they may consider the **average** payoff or, alternatively, the **total** (accumulated) payoff.

- The possibility to model other decision rules besides the imitate if better rule. In particular, we will implement all the decision rules we implemented in chapter III-4 for games played on grids.

# 2. Motivation. Cooperation on scale-free networks

In a series of highly-cited papers, Santos, Pacheco and colleagues (Santos and Pacheco (2005, 2006); Santos et al. (2006a, 2006b)) showed that scale-free networks can greatly promote cooperation in the Prisoner's Dilemma (figure 1).
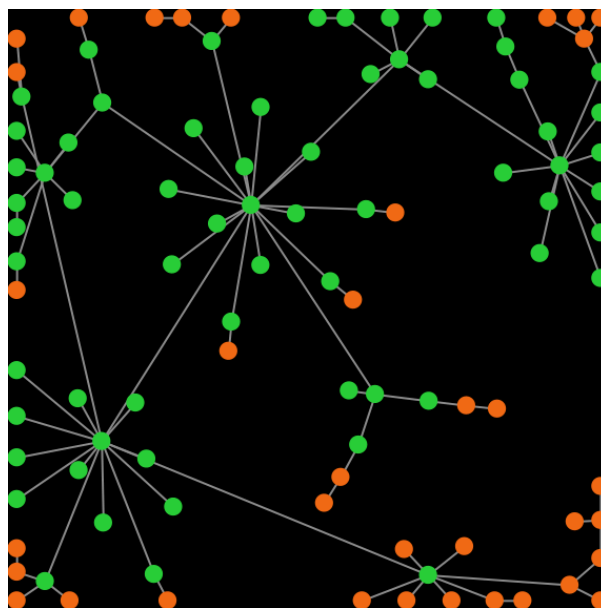


*Fig. 1. A preferential attachment network where hubs are cooperating (green)*

In their model, agents play with all their neighbors and their payoff is the total (accumulated) payoff

over all their interactions in the tick. In this chapter we are going to extend our model so we can replicate some of their results and check their robustness to different factors. Let's do it!

# 3. Description of the model

The model we are going to develop here is a generalization of the model implemented in the previous chapter. In particular, we are going to add the following three parameters:

- *play-with*. This parameter is used to determine how agents' payoffs are computed in each tick. It will be implemented with a chooser, with three possible values:

  - "*one-random-nbr*": agents play with one of their neighbors chosen at random.
  - "*all-nbrs-AVG-payoff*": agents play with all their neighbors and use the average payoff.
  - "*all-nbrs-TOTAL-payoff*": agents play with all their neighbors and use the total (accumulated) payoff.

- *decision-rule*. This parameter determines the decision rule that agents will follow to update their strategies, just like in the model we developed in chapter III-4. Note that all decision rules use the agents' payoffs, and these are computed following the method prescribed by parameter *play-with*. Parameter *decision-rule* will be implemented with a chooser, with six possible values:

  - "*best-neighbor*". This is the imitate the best neighbor rule we implemented in chapter III-4.
  - "*imitate-if-better*". This is the imitate if better rule adapted to networks.
  - "*imitative-pairwise-difference*". This is the imitative pairwise-difference rule implemented in chapter III-4.
  - "*imitative-positive-proportional-m*". This is the imitative-positive-proportional-m rule implemented in chapter III-4.
  - "*Fermi-m*". This is the Fermi-m rule implemented in chapter III-4.
  - "*Santos-Pacheco*". To facilitate the replication of previous results in the literature, we will also implement a variant of the *imitative pairwise-difference* rule initially proposed by Santos and Pacheco (2005).[1] Under this rule, the revising agent $i$ looks at one of her neighbors $j$ at random and copies her strategy with probability $\frac{\pi_j - \pi_i}{\Delta \max(k_i, k_j)}$, where $\pi_z$ denotes agent $z$'s payoff, $\Delta$ is the maximum payoff difference in the payoff matrix, and $k_z$ denotes agent $z$'s degree.

- *m*. This is the parameter that controls the intensity of selection in decision rules *imitative-positive-proportional-m* and *Fermi-m* (see description in chapter III-4).

Everything else stays as described in the previous chapter.

---

1. This variant has been used in several papers, such as Santos and Pacheco (2006), Santos et al. (2006a, 2006b), Gómez–Gardeñes et al. (2007) and Poncela et al. (2007).

# 4. Interface design

We depart from the model we developed in the previous chapter (so if you want to preserve it, now is a good time to duplicate it).
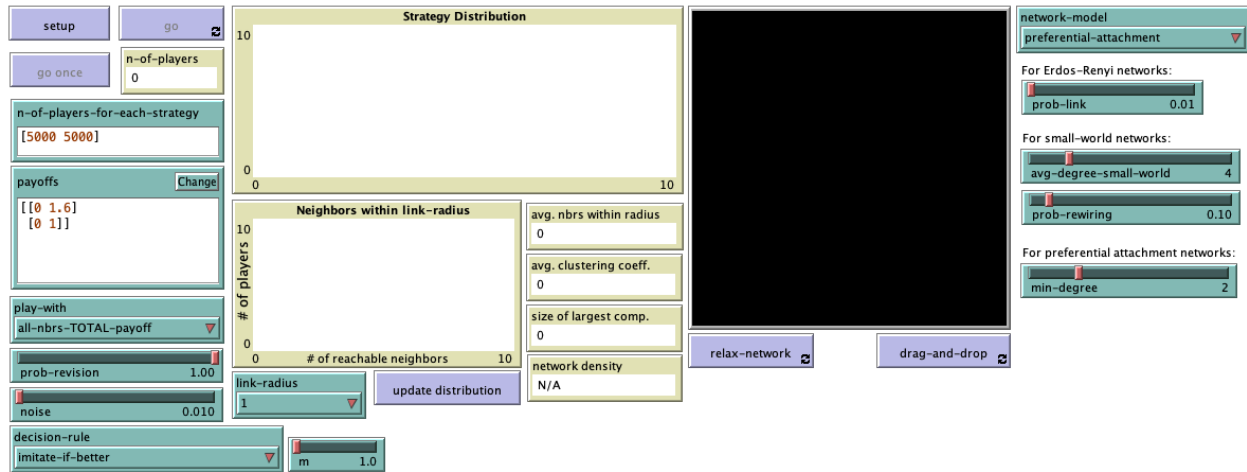


*Figure 2. Interface design*

The new interface (see figure 2 above) includes:

- One chooser for new parameter *play-with* (with possible values "*one-random-nbr*", "*all-nbrs-AVG-payoff*" and "*all-nbrs-TOTAL-payoff*").

- One chooser for new parameter *decision-rule* (with possible values "*best-neighbor*", "*imitate-if-better*", "*imitative-pairwise-difference*", "*imitative-positive-proportional-m*", "*Fermi-m*" and "*Santos-Pacheco*").

- A slider for parameter *m* (with *minimum* = 0 and *increment* = 0.1).

# 5. Code

## 5.1. Skeleton of the code

The main changes we will make to our model will take place in procedures to update-payoff and to update-strategy-after-revision. We will also modify other procedures such as to go, to setup-payoffs, and to setup-players, but those changes will be less significant. The skeleton of procedure to go is shown in figure 3.
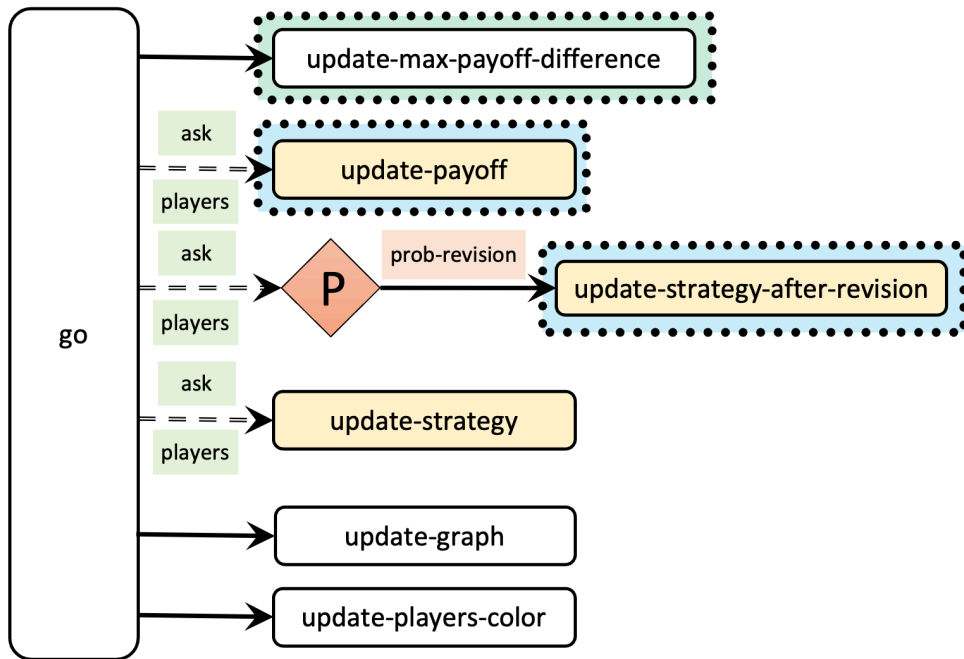
*Figure 3. Skeleton of procedure to go. The dashed green rectangle highlights the main addition in the code. The dashed blue rectangles highlight the main modifications in the code*

## 5.2. Extension I. Different ways of computing payoffs

To implement different ways of computing payoffs, we should modify to update-payoff, so it runs a different procedure depending on the value of parameter *play-with*, as shown in figure 4.
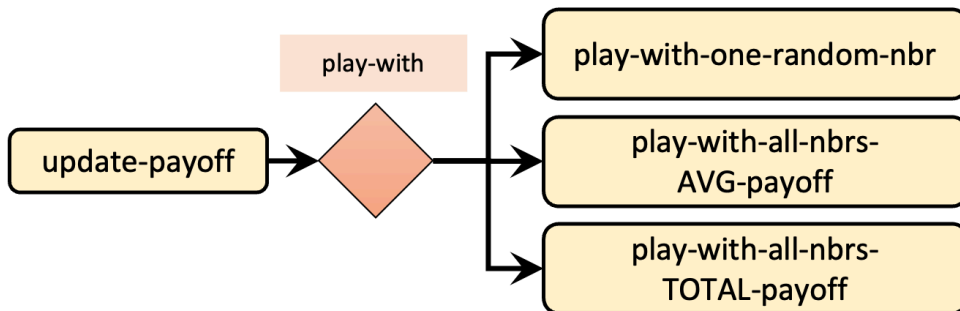


*Figure 4. Calls to other procedures from procedure to update–payoff*

Given the names we have chosen for the new procedures to be implemented (to play-with-one-random-nbr, to play-with-all-nbrs-TOTAL-payoff and to play-with-all-nbrs-AVG-payoff), the new code for procedure to update-payoff is particularly simple:

```
to update-payoff
  if any? link-neighbors [
    run (word "play-with-" play-with)
  ]
end
```

To fully understand the code above, note that if the value of parameter *play-with* is "*one-random-nbr*", then the code:

```
(word "play-with-" play-with)
```

will report the string "play-with-one-random-nbr".

Therefore, the code:

```
run (word "play-with-" play-with)
```

will run procedure to play-with-one-random-nbr, as desired.

Now we can implement the new payoff-computing procedures. The first one, to play-with-one-random-nbr, is not particularly difficult having seen the previous implementation of to update-payoff. Can you try to do it?

> ### Implementation of procedure to play-with-one-random-nbr
>
> Yes, well done!
>
> ```
> to play-with-one-random-nbr
>     let mate one-of link-neighbors
>     set payoff item ([strategy] of mate) (item strategy payoff-matrix)
> end
> ```

Let us now implement to play-with-all-nbrs-TOTAL-payoff. For this one, looking at the implementation of procedure to play in the model we developed in chapter III-4 will be very useful. Note, however, that in our new model we are not going to allow agents to play with themselves, so the variable named my-coplayers will have to be replaced by a Netlogo primitive. Can you give it a try?

> ### Implementation of procedure to play-with-all-nbrs-TOTAL-payoff
>
> Awesome!
>
> ```
> to play-with-all-nbrs-TOTAL-payoff
>   let n-of-coplayers-with-strategy-? n-values n-of-strategies [ i ->
>     count link-neighbors with [strategy = i] ]
>   let my-payoffs (item strategy payoff-matrix)
>   set payoff sum (map * my-payoffs n-of-coplayers-with-strategy-?)
> end
> ```

Finally, let us implement to play-with-all-nbrs-AVG-payoff. To do this, it is very tempting to start by copying and pasting the code of procedure to play-with-all-nbrs-TOTAL-payoff but please, don't do it! Duplicating code is not good practice because it makes the process of detecting and fixing errors much harder, and leads to code that is difficult to read and maintain. Can you implement procedure to play-with-all-nbrs-AVG-payoff without rewriting any of the code written in procedure to play-with-all-nbrs-TOTAL-payoff?

Implementation of procedure to play-with-all-nbrs-AVG-payoff

If you did this correctly, you do deserve a medal right now!

```
to play-with-all-nbrs-AVG-payoff
  play-with-all-nbrs-TOTAL-payoff
  set payoff (payoff / count link-neighbors)
end
```

With this, we have finished the implementation of the first extension to the model.

## 5.3. Extension II. Different decision rules

The implementation of the decision rule takes place in procedure to update-strategy-after-revision, so we will have to modify its code. Just like we did in chapter III-4, we will make the implementation of decision rules elegant and modular by coding a different procedure for each decision rule and by dealing with noise in a unified way. Figure 5 shows the skeleton of our new procedure to update-strategy-after-revision.
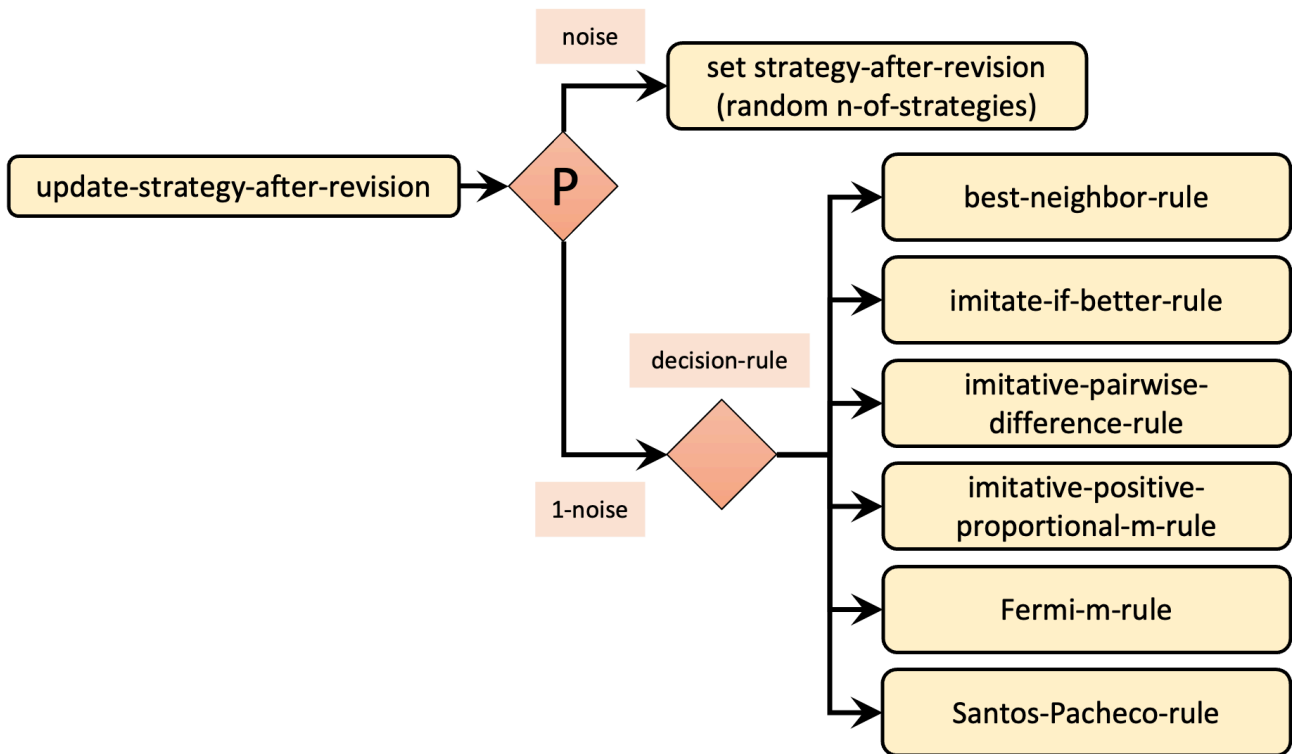
*Figure 5. Calls to other procedures from procedure to update–strategy–after–revision*

Given the skeleton shown in figure 5 and bearing in mind how we use NetLogo primitive `run` in our current implementation of procedure to update-payoff, can you venture a simple implementation for our new procedure to update-strategy-after-revision?

Implementation of procedure to update-strategy-after-revision

```
to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [ if any? link-neighbors [run (word decision-rule "-rule")] ]
end
```

Now we just have to implement the procedures for each of the six decision rules. This should be relatively easy given that we have implemented very similar rules in chapter III-4, but we have to keep in mind that:

- now the agents' payoffs may come in three different flavors, depending on the value of parameter *play-with*, and

- the set of neighbors in our network model is the set of `link-neighbors`.

Let us start with procedure to best-neighbor-rule. It will be useful to look at our implementation of

this rule for games played on grids. Just like we did in that model, you may want to define a new individually-owned variable named my-nbrs-and-me (and set its value at procedure to setup-players).

Implementation of procedure to best-neighbor-rule

We start by defining the new players-own variable my-nbrs-and-me.

```
players-own [
  strategy
  strategy-after-revision
  payoff
  my-nbrs-and-me   ;; <== new line
]
```

Then, we set its value at procedure to setup-players.

```
to setup-players
...
  ifelse n-of-players < 4
    [ user-message "There should be at least 4 players" ]
    [
      build-network
      ask players [set strategy -1]
      let i 0
      foreach initial-distribution [ j ->
        ask up-to-n-of j players with [strategy = -1] [
          set payoff 0
          set strategy i
          set strategy-after-revision strategy
          set my-nbrs-and-me                 ;; <== new line
              (turtle-set link-neighbors self) ;; <== new line
        ]
        set i (i + 1)
      ]
      set n-of-players count players
      update-players-color
    ]
end
```

Once this is done, we can just copy our implementation of this rule for games played on grids.

```
to best-neighbor-rule
  set strategy-after-revision
      [strategy] of one-of my-nbrs-and-me with-max [payoff]
```

```
    end
```

Note that this code will work correctly no matter how payoffs are computed.

To implement procedure to imitate-if-better-rule, our implementation of procedure to imitate-if-better-all-nbrs-rule for games played on grids will be useful, but we have to keep in mind that the computation of agents' payoffs is now taken care at procedure to update-payoff, so players-own variable payoff always contains the right value of the payoff (i.e., computed according to the value of parameter *play-with*).

Implementation of procedure to imitate-if-better-rule

```
to imitate-if-better-rule
  let observed-player one-of link-neighbors
  if ([payoff] of observed-player) > payoff [
    set strategy-after-revision ([strategy] of observed-player)
  ]
end
```

The implementation of procedure to imitative-pairwise-difference-rule is similar to the one we did for games played on grids but we have to bear in mind that the payoff in our model could now come from playing just once or from playing several times. Thus, we will have to think carefully about how to set the value of the maximum payoff difference.

Implementation of procedure to imitative-pairwise-difference-rule

A possible implementation of this procedure is as follows:

```
to imitative-pairwise-difference-rule
  let observed-player one-of link-neighbors

  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - payoff)
```

```
    set strategy-after-revision ifelse-value
      (random-float 1 < (payoff-diff / max-payoff-difference))
        [ [strategy] of observed-player ]
        [ strategy ]
      ;; If your strategy is the better, payoff-diff is negative,
      ;; so you are going to stick with it.
      ;; If it's not, you switch with probability
      ;; (payoff-diff / max-payoff-difference)
  end
```

Recall that the value of max-payoff-difference has to be set so the quotient (payoff-diff / max-payoff-difference) does not exceed 1, i.e. max-payoff-difference ≥ payoff-diff. The problem is that the maximum value of payoff-diff now depends on how payoffs are computed.

Let max-payoff-difference-in-matrix be the maximum payoff difference in the payoff matrix. If agents play once (*play-with* = "*one-random-nbr*") or their payoff is an average (*play-with* = "*all-nbrs-AVG-payoff*"), then payoff-diff ≤ max-payoff-difference-in-matrix. However, if agents use the total payoff (*play-with* = "*all-nbrs-TOTAL-payoff*"), then the maximum value of payoff-diff can well be greater than max-payoff-difference-in-matrix. In that case, the maximum value of payoff-diff could be up to max-payoff-difference-in-matrix * max-n-of-nbrs-of-each-player, where max-n-of-nbrs-of-each-player is the maximum degree in the network.

Given all this, we should define global variables max-payoff-difference-in-matrix, max-payoff-difference and max-n-of-nbrs-of-each-player, and set their values accordingly.

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
  avg-nbrs-within-radius
  avg-clustering-coefficient
  size-of-largest-component

  max-payoff-difference-in-matrix ;; <== new line
  max-payoff-difference           ;; <== new line
  max-n-of-nbrs-of-each-player    ;; <== new line
]
```

Now we should set the value of these new global variables. Let us start with the easiest one: max-n-of-nbrs-of-each-player. This value is already computed at procedure to plot-accessibility. The only change we have to make is to replace `let` with `set`, since this variable is now global (because we want to read it from anywhere in the code).

```
to plot-accessibility
  let steps link-radius
  if link-radius = "Infinity" [set steps (n-of-players - 1)]
  let n-of-nbrs-of-each-player
    [(count nw:turtles-in-radius steps) - 1] of players


  ;; modified line below (let -> set)
  set max-n-of-nbrs-of-each-player max n-of-nbrs-of-each-player
  set-current-plot "Neighbors within link-radius"
  set-plot-x-range 0 (max-n-of-nbrs-of-each-player + 1)
                   ;; + 1 to make room for the width of the last bar
  histogram n-of-nbrs-of-each-player


  set avg-nbrs-within-radius mean n-of-nbrs-of-each-player
end
```

The best place to set the value of max-payoff-difference-in-matrix, is at procedure to update-payoff.

```
to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix


  ;; new lines below
  set max-payoff-difference-in-matrix
      (max-of-matrix payoff-matrix) - (min-of-matrix payoff-matrix)
end
```

Note that we have to include the definition of two new procedures to compute the minimum and the maximum value of a matrix:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; SUPPORTING PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;;; Matrices ;;;
;;;;;;;;;;;;;;;;;
to-report max-of-matrix [matrix]
  report max reduce sentence matrix
end
to-report min-of-matrix [matrix]
  report min reduce sentence matrix
end
```

Finally, let us set the value of max-payoff-difference in a new procedure created for that purpose.

```
to update-max-payoff-difference
  set max-payoff-difference
    max-payoff-difference-in-matrix *
      ifelse-value play-with = "all-nbrs-TOTAL-payoff"
        [max-n-of-nbrs-of-each-player][1]
end
```

Once this procedure has been implemented, we can call it from to go to make sure that the value of max-payoff-difference is updated if the user changes the way payoffs are computed at runtime.

```
to go
  update-max-payoff-difference  ;; <== new line

  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]

  tick

  update-graph
  update-players-color
end
```

To implement procedure to imitative-positive-proportional-m-rule, we can use the code we wrote for games played on grids, but before we should include the extension rnd at the beginning of our code

```
extensions [nw rnd]
```

Implementation of procedure to imitative-positive-proportional-m-rule

```
to imitative-positive-proportional-m-rule
  let chosen-nbr rnd:weighted-one-of my-nbrs-and-me [ payoff ^ m ]
  set strategy-after-revision [strategy] of chosen-nbr
end
```

To avoid errors when payoffs are negative, we will include procedure to check-payoffs-are-non-negative, just like we did in the model for games played on grids.

```
to check-payoffs-are-non-negative
  if min reduce sentence payoff-matrix < 0 [
    user-message (word
      "Since you are using decision-rule =\n"
      "imitative-positive-proportional-m,\n"
      "all elements in the payoff matrix\n"
      payoffs
      "\nshould be non-negative numbers.")
  ]
end
```

An appropriate place to call this procedure would be at the end of procedure to setup-payoffs, which would then be as follows:

```
to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix

  ;; new lines below
  set max-payoff-difference
      (max-of-matrix payoff-matrix) - (min-of-matrix payoff-matrix)
  if decision-rule = "imitative-positive-proportional-m"
      [ check-payoffs-are-non-negative ]
end
```

Feel free to implement procedure to Fermi-m-rule by yourself, looking at our previous implementation.

Implementation of procedure to Fermi-m-rule

A possible implementation of this procedure is as follows:

```
to Fermi-m-rule
  let observed-player one-of link-neighbors

  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - payoff)
```

```
    set strategy-after-revision ifelse-value
      (random-float 1 < (1 / (1 + exp (- m * payoff-diff))))
        [ [strategy] of observed-player ]
        [ strategy ]
  end
```

Finally, looking at the description of "Santos-Pacheco" rule, please try to implement procedure to Santos-Pacheco-rule.

Implementation of procedure to Santos-Pacheco-rule

A possible implementation of this procedure is as follows:

```
to Santos-Pacheco-rule
  let observed-player one-of link-neighbors

  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - (payoff))

  let degree count link-neighbors
  let degree-of-observed-player
      [count link-neighbors] of observed-player

  set strategy-after-revision ifelse-value
    (random-float 1 < (payoff-diff / (max-payoff-difference-in-matrix *
      max (list degree degree-of-observed-player))))
        [ [strategy] of observed-player ]
        [ strategy ]
  end
```

With this, we have finished the implementation of the second extension to the model.

# 5.4. Complete code in the Code tab

We have finished our model!

```
extensions [nw rnd]

globals [
  payoff-matrix
  n-of-strategies
  n-of-players

  avg-nbrs-within-radius
  avg-clustering-coefficient
  size-of-largest-component

  max-payoff-difference-in-matrix ;; <== new line
  max-payoff-difference           ;; <== new line
  max-n-of-nbrs-of-each-player    ;; <== new line
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff

  my-nbrs-and-me ;; <== new line
]

;;;;;;;;;;;;;;
;;; SETUP ;;;
;;;;;;;;;;;;;;

to setup
  clear-all
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-graph
  compute-network-metrics
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix

  ;; new lines below
  set max-payoff-difference-in-matrix
      (max-of-matrix payoff-matrix) - (min-of-matrix payoff-matrix)
  if decision-rule = "imitative-positive-proportional-m"
      [ check-payoffs-are-non-negative ]
```

```
    end

to setup-players
  let initial-distribution
      read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]

  set n-of-players sum initial-distribution
  ifelse n-of-players < 4
    [ user-message "There should be at least 4 players" ]
    [
      build-network

      ask players [set strategy -1]
      let i 0
      foreach initial-distribution [ j ->
        ask up-to-n-of j players with [strategy = -1] [
          set payoff 0
          set strategy i
          set strategy-after-revision strategy
          set my-nbrs-and-me                      ;; <== new line
              (turtle-set link-neighbors self) ;; <== new line
        ]
        set i (i + 1)
      ]

      set n-of-players count players
      update-players-color
    ]
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; NETWORK CONSTRUCTION ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to build-network
  set-default-shape players "circle"
  run (word "build-" network-model "-network")
  ask players [fd 15]
end

to build-Erdos-Renyi-network
```

```
    nw:generate-random players links n-of-players prob-link
end

to build-Watts-Strogatz-small-world-network
 nw:generate-watts-strogatz players links n-of-players
     (avg-degree-small-world / 2) prob-rewiring
end

to build-preferential-attachment-network
  nw:generate-preferential-attachment players links n-of-players
     min-degree
end

to build-ring-network
  nw:generate-ring players links n-of-players
end

to build-star-network
  nw:generate-star players links n-of-players
end

to build-grid-4-nbrs-network
  let players-per-line (floor sqrt n-of-players)
  nw:generate-lattice-2d players links
     players-per-line players-per-line false
end

to build-wheel-network
  nw:generate-wheel players links n-of-players
end

to build-path-network
  build-ring-network
  ask one-of links [die]
end

;;;;;;;;;;
;;; GO ;;;
;;;;;;;;;;

to go
  update-max-payoff-difference    ;; <== new line
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]

  tick

  update-graph
```

```
    update-players-color
end


;;;;;;;;;;;;;;;;;;;;;;;;;
;;; UPDATE PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;

to update-payoff
  if any? link-neighbors [
    run (word "play-with-" play-with)
  ]
end


to play-with-one-random-nbr
    let mate one-of link-neighbors
    set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end


to play-with-all-nbrs-TOTAL-payoff
  let n-of-coplayers-with-strategy-? n-values n-of-strategies [ i ->
    count link-neighbors with [strategy = i] ]
  let my-payoffs (item strategy payoff-matrix)
  set payoff sum (map * my-payoffs n-of-coplayers-with-strategy-?)
end


to play-with-all-nbrs-AVG-payoff
  play-with-all-nbrs-TOTAL-payoff
  set payoff (payoff / count link-neighbors)
end


to update-max-payoff-difference
  set max-payoff-difference
    max-payoff-difference-in-matrix *
      ifelse-value play-with = "all-nbrs-TOTAL-payoff"
        [max-n-of-nbrs-of-each-player][1]
end

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [ if any? link-neighbors [run (word decision-rule "-rule")] ]
end


to best-neighbor-rule
  set strategy-after-revision [strategy] of
      one-of my-nbrs-and-me with-max [payoff]
end


to imitate-if-better-rule
  let observed-player one-of link-neighbors
  if ([payoff] of observed-player) > payoff [
    set strategy-after-revision ([strategy] of observed-player)
  ]
```

```
    end

to imitative-pairwise-difference-rule
  let observed-player one-of link-neighbors


  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - payoff)


  set strategy-after-revision ifelse-value
    (random-float 1 < (payoff-diff / max-payoff-difference))
      [ [strategy] of observed-player ]
      [ strategy ]
    ;; If your strategy is the better, payoff-diff is negative,
    ;; so you are going to stick with it.
    ;; If it's not, you switch with probability
    ;; (payoff-diff / max-payoff-difference)
end

to imitative-positive-proportional-m-rule
  let chosen-nbr rnd:weighted-one-of my-nbrs-and-me [ payoff ^ m ]
  set strategy-after-revision [strategy] of chosen-nbr
end

to Fermi-m-rule
  let observed-player one-of link-neighbors


  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - payoff)


  set strategy-after-revision ifelse-value
    (random-float 1 < (1 / (1 + exp (- m * payoff-diff))))
      [ [strategy] of observed-player ]
      [ strategy ]
end

to Santos-Pacheco-rule
  let observed-player one-of link-neighbors


  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - (payoff))


  let degree count link-neighbors
  let degree-of-observed-player
      [count link-neighbors] of observed-player


  set strategy-after-revision ifelse-value
    (random-float 1 < (payoff-diff / (max-payoff-difference-in-matrix *
      max (list degree degree-of-observed-player))))
        [ [strategy] of observed-player ]
        [ strategy ]
end
```

```
to update-strategy
  set strategy strategy-after-revision
end


;;;;;;;;;;;;;
;;; PLOTS ;;;
;;;;;;;;;;;;;

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end


to update-graph
  let strategy-numbers (range n-of-strategies)
  let strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers

  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end

to update-players-color
  ask players [set color 25 + 40 * strategy]
end


;;;;;;;;;;;;;;;;;;;;;;;
;;; NETWORK METRICS ;;;
;;;;;;;;;;;;;;;;;;;;;;;

to compute-network-metrics
  plot-accessibility
  compute-avg-clustering-coefficient
  compute-size-of-largest-component
end

to plot-accessibility
  let steps link-radius
  if link-radius = "Infinity" [set steps (n-of-players - 1)]
  let n-of-nbrs-of-each-player
    [(count nw:turtles-in-radius steps) - 1] of players
```

```
  ;; modified line below (let -> set)
  set max-n-of-nbrs-of-each-player max n-of-nbrs-of-each-player
  set-current-plot "Neighbors within link-radius"
  set-plot-x-range 0 (max-n-of-nbrs-of-each-player + 1)
                    ;; + 1 to make room for the width of the last bar
  histogram n-of-nbrs-of-each-player

  set avg-nbrs-within-radius mean n-of-nbrs-of-each-player
end


to compute-avg-clustering-coefficient
  set avg-clustering-coefficient
      mean [ nw:clustering-coefficient ] of players
end


to compute-size-of-largest-component
  set size-of-largest-component max map count nw:weak-component-clusters
end


;;;;;;;;;;;;;;;
;;; LAYOUT ;;;
;;;;;;;;;;;;;;;

;; Procedures taken from Wilensky's (2005a) NetLogo Preferential
;; Attachment model
;; http://ccl.northwestern.edu/netlogo/models/PreferentialAttachment
;; and Wilensky's (2005b) Mouse Drag One Example
;; http://ccl.northwestern.edu/netlogo/models/MouseDragOneExample
to relax-network
  ;; the number 3 here is arbitrary; more repetitions slows down the
  ;; model, but too few gives poor layouts
  repeat 3 [
    ;; the more players we have to fit into
    ;; the same amount of space, the smaller
    ;; the inputs to layout-spring we'll need to use
    let factor sqrt count players
    ;; numbers here are arbitrarily chosen for pleasing appearance
    layout-spring players links
                  (1 / factor) (7 / factor) (3 / factor)
    display  ;; for smooth animation
  ]
  ;; don't bump the edges of the world
  let x-offset max [xcor] of players + min [xcor] of players
  let y-offset max [ycor] of players + min [ycor] of players
  ;; big jumps look funny, so only adjust a little each time
  set x-offset limit-magnitude x-offset 0.1
  set y-offset limit-magnitude y-offset 0.1
  ask players [ setxy (xcor - x-offset / 2) (ycor - y-offset / 2) ]
end


to-report limit-magnitude [number limit]
  if number > limit [ report limit ]
  if number < (- limit) [ report (- limit) ]
```

```
    report number
  end

to drag-and-drop
  if mouse-down? [
    let candidate min-one-of players
        [distancexy mouse-xcor mouse-ycor]
    if [distancexy mouse-xcor mouse-ycor] of candidate < 1 [
    ;; The WATCH primitive puts a "halo" around the watched turtle.
      watch candidate
      while [mouse-down?] [
        ;; If we don't force the view to update, the user won't
        ;; be able to see the turtle moving around.
        display
        ;; The SUBJECT primitive reports the turtle being watched.
        ask subject [ setxy mouse-xcor mouse-ycor ]
      ]
      ;; Undoes the effects of WATCH.
      reset-perspective
    ]
  ]
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; SUPPORTING PROCEDURES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;

to check-payoffs-are-non-negative
  if min reduce sentence payoff-matrix < 0 [
    user-message (word
      "Since you are using decision-rule =\n"
      "imitative-positive-proportional-m,\n"
      "all elements in the payoff matrix\n"
      payoffs
      "\nshould be non-negative numbers.")
  ]
end

;;;;;;;;;;;;;;;;;
;;; Matrices ;;;
;;;;;;;;;;;;;;;;;

to-report max-of-matrix [matrix]
  report max reduce sentence matrix
end

to-report min-of-matrix [matrix]
  report min reduce sentence matrix
end
```

# 6. Sample runs

## 6.1. Cooperation on preferential-attachment networks

Now that we have our model ready, we can replicate some of the results put forward by Santos and Pacheco (2005, 2006) and Santos et al. (2006a, 2006b).[2] These authors observed that scale-free networks (especially preferential-attachment networks created following the Barabási–Albert model) can promote cooperation in social dilemmas. Let us see this with our new model.

Figure 6 below shows a representative simulation run of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], using total payoffs and *Santos-Pacheco* decision rule. Strategy 0 (orange) corresponds to defection and strategy 1 (green) corresponds to cooperation. The network is preferential attachment with min-degree 2 (average degree ≈ 4). As you can see, the whole population reaches very high levels of cooperation (green) after a few thousand ticks. This is our baseline setting.



*Figure 6. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], using total payoffs and Santos–Pacheco decision rule. The network is preferential attachment with min–degree 2 (average degree ≈ 4)*

The whole simulation setting for figure 6 (i.e., baseline setting) is included below, in case you want to run it for yourself. We chose 10 000 agents to match the original experiments, but feel free to use a smaller population (e.g. 1000 agents) so the model runs faster. Results will be very similar.

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[0 1.1875]\n [-0.1875 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "Santos-Pacheco"]
["play-with" "all-nbrs-TOTAL-payoff"]
```

---

2. The algorithm these authors use to generate scale-free networks is slightly different from the one implemented in NetLogo. They start with an empty network, while we start with a complete network. This is a very small difference that is immaterial for our purposes.

```
["prob-revision" 1]
["noise" 0]
```

The cooperation obtained for the preferential attachment network in our baseline setting vanishes if the model is run on more homogeneous networks where all agents have a similar number of link-neighbors. To see this, let us run a few simulations on Watts-Strogatz networks. In contrast with preferential attachment networks, Watts-Strogatz networks have degree distributions that tend to be clustered around the mean. As you can see in Table 1 below, defection prevails on these networks.

| Rewiring probability | Representative simulation | Degree Distribution |
|:---:|:---:|:---:|
| 0 |  |  |
| 0.25 |  |  |
| 0.50 |  |  |
| 0.75 |  |  |
| 1 |  |  |

*Table 1. Representative simulations of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], using total payoffs and Santos-Pacheco decision rule. The networks are Watts-Strogatz with average degree 4 and different rewiring probabilities*

The parameter values used for the simulations shown in Table 1 are included below.

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[0 1.1875]\n [-0.1875 1]]"]
["network-model" "Watts-Strogatz-small-world"]
["avg-degree-small-world" 4]
["prob-rewiring" [0 0.25 0.5 0.75 1]]
["decision-rule" "Santos-Pacheco"]
["play-with" "all-nbrs-TOTAL-payoff"]
["prob-revision" 1]
["noise" 0]
```

How does cooperation emerge in Barabási–Albert scale-free networks? The mechanism, which is explained by Santos and Pacheco (2006), Szabó and Fáth (2007, section 6.8), and Gómez-Gardeñes et al. (2007), relies on using total (accumulated) payoffs.

> *Finally, we would like to point out that the results for the evolution of cooperation are affected if, instead of using the accumulated payoff for each agent, its fitness is associated with the accumulated payoff divided by the number of interactions each agent engages during his life-cycle.* Santos and Pacheco (2006, p. 732)

If agents use average (rather than total) payoffs, cooperation is severely undermined (see e.g. Wu et al. (2005), Wu et al. (2007), Tomassini et al. (2007), Szolnoki et al. (2008), Antonioni and Tomassini (2012), and Maciejewski et al. (2014)).[3] In the next section, we run some simulations to illustrate this.

## 6.2. Robustness of cooperation on scale-free preferential-attachment networks

In this final section, we run a few simulations to check the robustness of the emergence of cooperation on preferential-attachment networks.

### The crucial importance of using total (accumulated) payoffs

First, we corroborate that, indeed, if agents use average payoffs rather than total payoffs, cooperation levels drop. In our model, this is best checked by using the *imitative pairwise-difference* decision rule. When using average payoffs, the only difference between the *imitative pairwise-difference* decision rule and *Santos-Pacheco* rule is that the latter divides the normalized payoff difference by $\max(k_i, k_j)$, and this seems unnecessary since the normalized payoff difference is already in the range [0,1]. In any case, the results using *Santos-Pacheco* rule are qualitatively the same, so feel free to use that rule to run the experiments below, if you like.

---

3. Tomassini et al. (2007), Szolnoki et al. (2008) and Antonioni and Tomassini (2012) develop models to study the transition from the extreme setting of total payoffs to the opposite extreme setting of average payoffs in a continuous way.

Figure 7 below shows a representative simulation run with the same baseline setting as in figure 6, but using average rather than total payoffs. As you can see, the level of cooperation quickly drops to very low values.



*Figure 7. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], using average payoffs and the imitative pairwise-difference decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4)*

The whole simulation setting for figure 7 is included below, in case you want to run it for yourself.

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[0 1.1875]\n [-0.1875 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "imitative-pairwise-difference"]
["play-with" "all-nbrs-AVG-payoff"]
["prob-revision" 1]
["noise" 0]
```

The results for the case where agents play with one random neighbor are even clearer (see figure 8). The whole population defects after only a few ticks.



*Figure 8. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], obtain their payoff by playing with one random neighbor each tick, and follow the imitative pairwise-difference decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4)*

The whole simulation setting for figure 8 is included below, in case you want to run it for yourself.

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[0 1.1875]\n [-0.1875 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "imitative-pairwise-difference"]
["play-with" "one-random-nbr"]
["prob-revision" 1]
["noise" 0]
```

## Random noise

It turns out that the emergence of cooperation in preferential-attachment networks is also quite sensitive to random noise. Figure 9 below shows a representative simulation with the same baseline setting as in figure 6, but adding a little bit of noise (*noise* = 0.01). Adding this little noise decreases the level of cooperation from nearly 100% to about 50%.



*Figure 9. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], using total payoffs and Santos-Pacheco decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4). Noise is 0.01*

The whole simulation setting for figure 9 is included below, in case you want to run it for yourself.

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[0 1.1875]\n [-0.1875 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "Santos-Pacheco"]
["play-with" "all-nbrs-TOTAL-payoff"]
["prob-revision" 1]
["noise" 0.01]
```

## Payoff values

As we saw in Part III, local interactions in structured populations (as opposed to global interactions in well-mixed populations) can often promote cooperation, but only to some extent, i.e. within a limited

range of payoff values. In general, one can make cooperation vanish by modifying the payoffs in a way that favors defectors.

As an example, figure 10 below shows a representative simulation run with the same baseline setting as in figure 6, but replacing *Temptation* = 1.1875 with *Temptation* = 1.375 and *Suckers* = -0.1875 with *Suckers* = -0.375. This small change in payoff values consistently leads to full defection in a few hundred ticks.
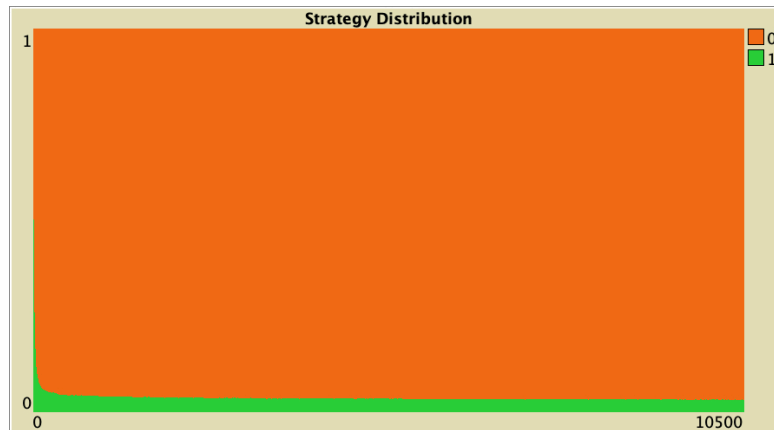


*Figure 10. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.375] [-0.375 1]], using total payoffs and Santos-Pacheco decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4)*

The whole simulation setting for figure 10 is included below.

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[0 1.375]\n [-0.375 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "Santos-Pacheco"]
["play-with" "all-nbrs-TOTAL-payoff"]
["prob-revision" 1]
["noise" 0]
```

## Payoff translations or interaction costs

When all agents play the game the same number of times or when they use average payoffs, many decision rules are invariant to positive affine transformations of payoffs. In other words, one can multiply all payoffs by a positive constant and/or add any constant to all payoffs, and the dynamics will not be affected.

However, in networks where agents use total payoffs and they have different numbers of neighbors, this is often no longer the case. Decision rule *Santos-Pacheco* in preferential-attachment networks is a case in point (Tomassini et al., 2007).

As an example, figure 11 below shows a representative simulation run with the same baseline setting as in figure 6, but where we have subtracted 0.25 from all payoffs. This 0.25 can be interpreted as

a cost for every interaction where agents are involved (Masuda, 2007). As you can see, this small subtraction to all payoff values significantly decreases the level of cooperation.



*Figure 11. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[-0.25 0.9375] [-0.4375 0.75]], using total payoffs and Santos-Pacheco decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4)*

The whole simulation setting for figure 11 is included below.

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[-0.25 0.9375]\n [-0.4375 0.75]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "Santos-Pacheco"]
["play-with" "all-nbrs-TOTAL-payoff"]
["prob-revision" 1]
["noise" 0]
```

## Initial conditions

Note that the state where everyone defects is absorbing. Also, we can think of nearby states such that if we start a simulation there, the system will evolve to the state of full defection. States with one single cooperator are clear examples. This suggests that, in general, the state of full defection will have some basin of attraction. In other words, if we choose initial conditions close enough to the state of full defection, it is likely that the simulation will end up with the whole population defecting.

In our baseline setting, it turns out that the basin of attraction of the state of full defection is quite sizable. As an example, Figure 12 below shows a representative simulation with the same baseline setting as in figure 6, but with initial conditions [9000 1000]. Most simulation runs with these initial conditions end up with no cooperation at all.
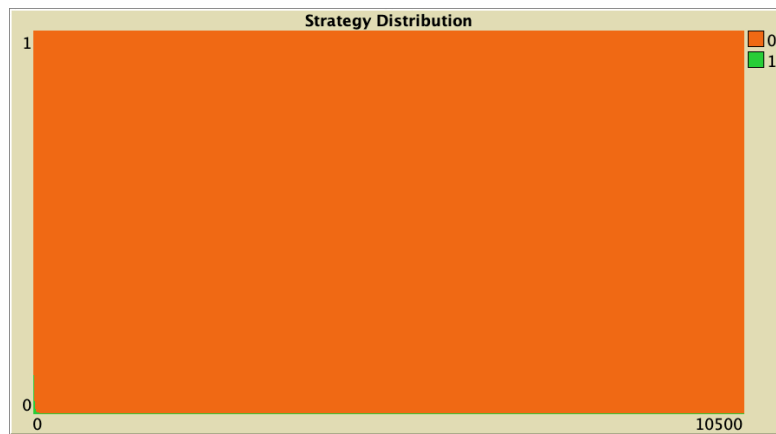
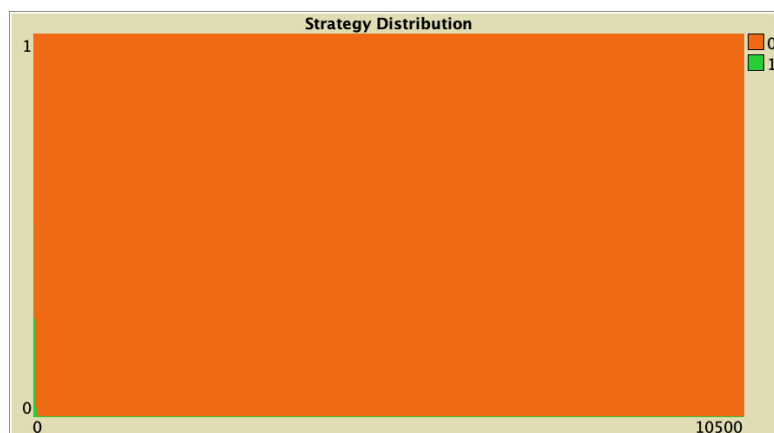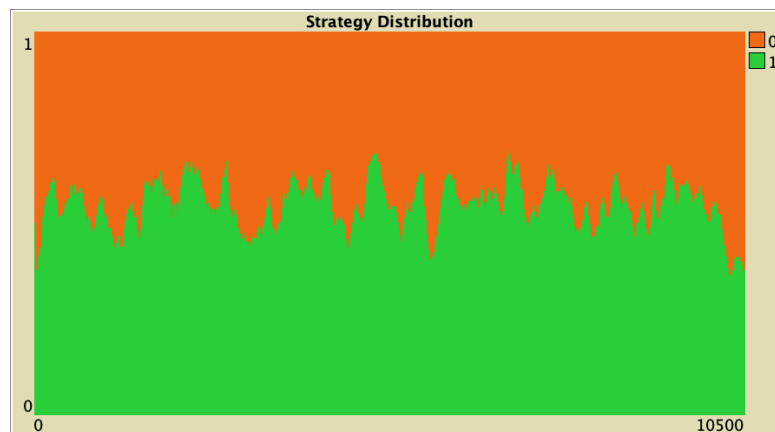*Figure 12. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], using total payoffs and Santos-Pacheco decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4). Initial conditions as [9000 1000]*

The whole simulation setting for figure 12 is included below, in case you want to run it for yourself.

```
["n-of-players-for-each-strategy" "[9000 1000]"]
["payoffs" "[[0 1.1875]\n [-0.1875 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "Santos-Pacheco"]
["play-with" "all-nbrs-TOTAL-payoff"]
["prob-revision" 1]
["noise" 0]
```

We also get very low cooperation levels if we start from initial conditions a bit further away from the state of full defection. As an example, Figure 13 below shows a representative simulation with the same baseline setting as in figure 6, but with initial conditions [8000 2000].
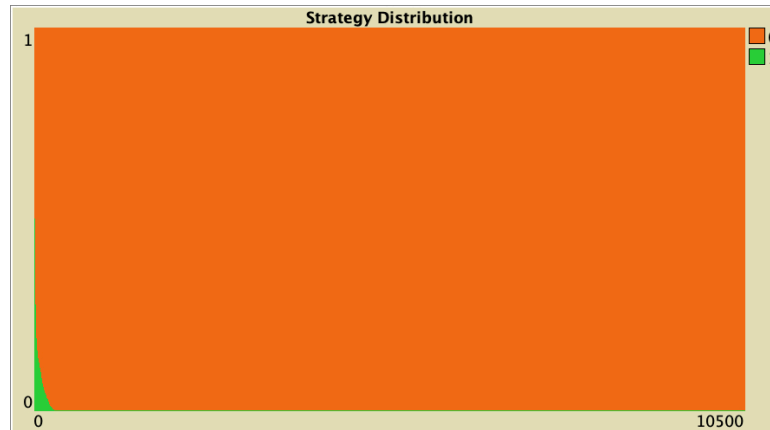


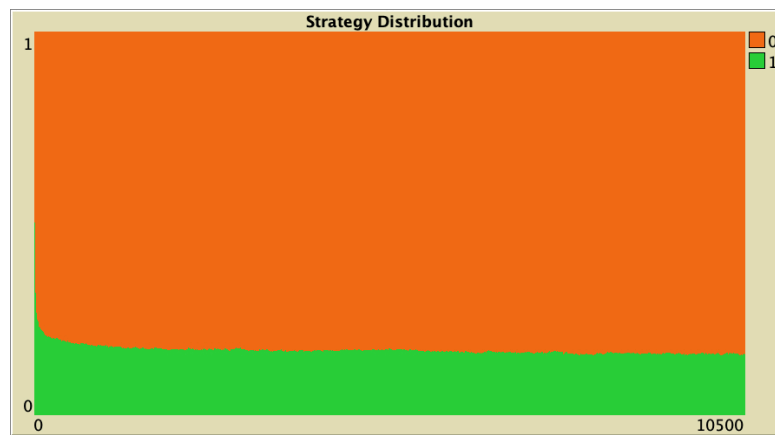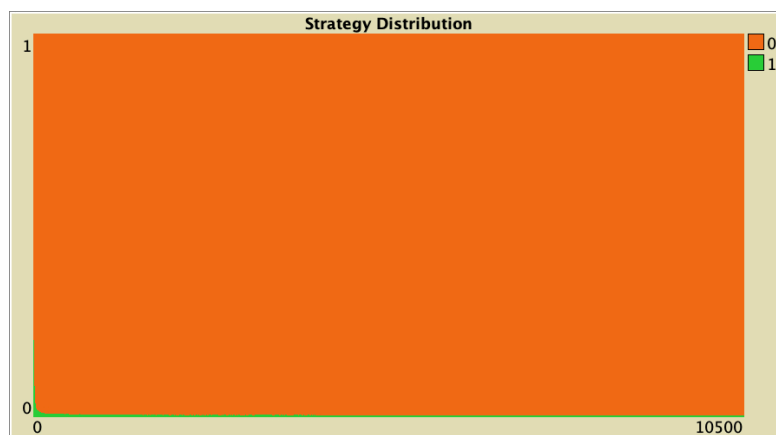*Figure 13. Representative simulation of a population of 10 000 agents who play the Prisoner's Dilemma with payoffs [[0 1.1875] [-0.1875 1]], using total payoffs and Santos-Pacheco decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4). Initial conditions as [8000 2000]*

The whole simulation setting for figure 13 is included below, in case you want to run it for yourself.

```
["n-of-players-for-each-strategy" "[8000 2000]"]
["payoffs" "[[0 1.1875]\n [-0.1875 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "Santos-Pacheco"]
["play-with" "all-nbrs-TOTAL-payoff"]
["prob-revision" 1]
["noise" 0]
```

## Final thoughts

We are approaching the end of our journey together, and by now we hope that the following words we wrote in the preface of this book start to make some sense to you:

> *To use a scientific model rigorously, it is important to be fully aware of all the assumptions embedded in it, and also of the various alternative assumptions that could have been chosen. If we don't understand all the details of a model, we run the risk of over-extrapolating its scope and of drawing unsound conclusions.*

In this chapter we have seen –once again– that the dynamics of evolutionary models can often be significantly affected by a multitude of factors that may not seem very relevant at first sight. This implies that in most cases it is extremely hard, if not impossible, to derive general simple rules about evolutionary dynamics on networks. For good or for bad, this is the way our universe seems to work, and there is nothing we can do to change that.

On a more positive note, this reflection highlights –once again– the importance of the skills you have learned. You are now able to design, implement and analyze models that can help you derive sound conclusions. The scope of these conclusions will often be more limited than what we would have wished for, but there is not much we can do about that. What is definitely under our control is to make sure that our conclusions are sound and rigorous, and by following this book you have taken a huge step in that direction.

# 7. Exercises

You can use the following link to download the complete NetLogo model: nxn-games-on-networks.nlogo.

**Exercise 1.** How can we parameterize our model to replicate the results shown in figure 2 of Santos and Pacheco (2005)?

**Exercise 2.** How can we parameterize our model to replicate the results shown in figure 6 of Santos and Pacheco (2006)?

**Exercise 3.** How can we parameterize our model to replicate the results shown in figure 5 of Tomassini et al. (2007)?

**Exercise 4.** In our model, all *revising* agents update their strategy synchronously. What changes would you have to make in the code so revising agents within the tick update their strategies sequentially (in random order), rather than simultaneously?
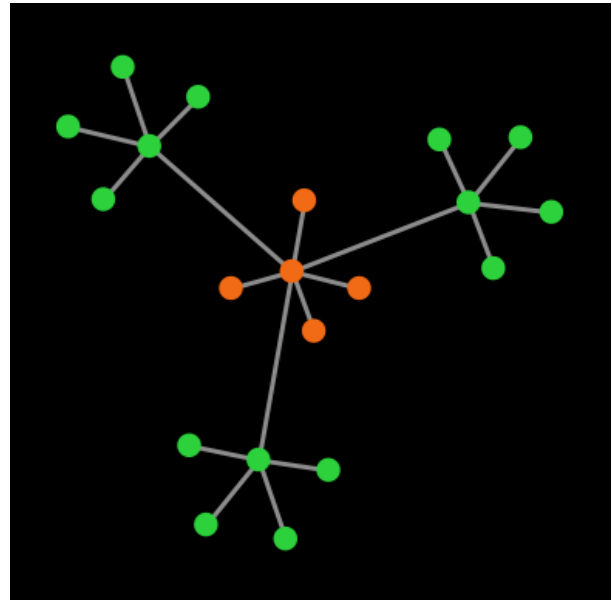


*A beautiful network*

Hint to implement asynchronous strategy updating

It is possible to do this by making a couple of minor changes in procedure to go, without touching the rest of the code.

**Exercise 5.** How can we parameterize the model developed in exercise 4 above to replicate the results shown in figure 2 of Tomassini et al. (2007)?

**Exercise 6.** For our "sample runs", we have used unconventional payoff values, such as 1.1875, -0.1875, 1.375 or -0.375. Can you guess why did we not use better looking numbers such as 1.2, -0.2, 1.4 and -0.4 instead?

Hint

Write all these decimal numbers in binary. If you use a finite number of bits, once you have converted all these decimal numbers to finite bit strings, convert the binary representations back into decimal base.

# IV-5. Analysis of these models

## 1. Introduction

In Part II, we learned to implement and analyze models in *well-mixed populations*. These are models where every agent has the same probability of interacting with every other agent. In such models, the state of the system can be defined by the distribution of strategies in the population; in other words, the strategy distribution contains all the information we need to (probabilistically) predict the evolution of the system as accurately as it is possible. Because of this, we saw that there are various mathematical techniques that we can use to analyze and characterize the dynamics of these models impressively well (see chapter II-5). For well mixed populations, the usefulness of mathematical analysis cannot be overstated.

In Part III, we learned to implement and analyze models on *spatial grids*. In these models, the strategy distribution is not enough to predict the evolution of the system anymore because each agent has its own set of neighbors (see chapter III-5). Nonetheless, in grid models there is still some symmetry in the sense that all agents have the same number of neighbors (except, maybe, at the boundaries). Because of this, for some of these models, we could still derive some analytical approximations such as the pair approximation for regular networks. These approximations can be useful because they are sometimes able to predict overall trends, but in general there is no guarantee that they will work well. For this reason, the relative usefulness of the computer simulation approach as a tool for exploration and analysis (vs the mathematical approach) is significantly greater in spatial models than in well-mixed populations.

Finally, here in Part IV, we have learned to implement models where agents are embedded on arbitrary *networks*. In these models, each agent has its own set of neighbors and, in general, there is no symmetry whatsoever, so the state of the system must be defined at the individual level. Because of this, the mathematical approach loses much of its usefulness and we are bound to resort to computer simulation as the main tool for exploration and analysis.

> *The challenge is high, because heterogeneity in both agent types and connectivity structure breaks down the symmetry of agents, and thus requires a dramatic change of perspective in the description of the system from the aggregate level to the agent level. The resulting huge increase in the relevant system variables makes most standard analytical techniques, operating with differential equations, fixed points, etc., largely inapplicable. What remains is agent based modeling, meaning extensive numerical simulations and analytical techniques going beyond the traditional mean-field level.* Szabó and Fáth (2007, p. 102)

In this chapter, we are going to illustrate the importance of four best practices that we should try to follow when using the computer simulation approach:

- Avoid errors
- Use informative metrics
- Report meaningful statistics
- Derive sound conclusions

Let us see each of these in turn!

# 2. Avoid errors

## 2.1. Introduction

Following Galán et al. (2009, par. 3.3), we consider that an error has occurred whenever there is a mismatch between what the model does and what we, developers, think the model does. Naturally, we should always try to avoid errors.

The most common errors are those that we inadvertently introduce in our code, i.e., situations where we are telling the computer to do something different from what we want it to do. We have seen several examples of this type of coding error in this book. For example, in our very first model (chapter II-1), we initially discussed a natural but faulty implementation of procedure to go, where some strategies could be imitated based on payoffs that had not been obtained with those strategies (see section "A naive implementation"). To fix this error we defined a new agents' variable named strategy-after-revision.

In section 5.7 of that same chapter, we saw another error. At first, we did not properly initialize the agents' variable strategy-after-revision, and that led to dynamics that were not the ones we intended to implement. You can find other examples of errors (and their fix) in chapter IV-1, where our code could ask players with no neighbors to select one of them, and in chapter IV-2, where our code could try to select 15 agents out of a set of only 5.

Galán et al. (2009) discuss several activities aimed at detecting and avoiding errors in agent-based models, including reimplementing the model in different programming languages and testing the model in extreme cases that are perfectly understood (Gilbert and Terna, 2000). Examples of extreme cases that are often easy to understand include setting the probability of revision equal to 0, setting all payoffs to 0, or setting the noise to 1.

Here we would like to discuss a subtle type of error that is often neglected, but which can have undesirable consequences that may be difficult to detect and understand, i.e., floating-point errors.

## 2.2. Floating-point errors

Like most programming languages, NetLogo represents numbers internally in base 2 (using 64-bit strings) and adheres to the IEEE standard for floating-point arithmetic (IEEE 754).[1] This means, in particular, that any number that is not exactly representable in base 2 with a finite bitstring cannot be exactly stored in NetLogo. Some examples of decimal numbers that cannot be represented exactly with a finite bitstring are 0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8, and 0.9. As a matter of fact, only 63 of the 999 999 numbers of the form $0.000001 \cdot i$ ($i$ = 1, 2... 999 999) are exactly representable in IEEE 754 single or double precision. Thus, whenever we input one of these numbers in our model, we are inevitably introducing some rounding error. This error will generally be very small, but when dealing with discontinuous functions, small errors can have big consequences.

**Example 1**. As an example, consider the imitate if better decision rule, which dictates that an agent A will copy the strategy of an observed agent B if and only if the observed agent B has a greater payoff. Now imagine that agents obtain a payoff by playing three times, and agent A's payoff is (0.3 + 0 + 0) while agent B's payoff is (0.1 + 0.1 + 0.1). It turns out that the IEEE 754 floating-point result of the sum (0.1 + 0.1 + 0.1) is slightly greater than the floating-point result of the sum (0.3 + 0 + 0). You can check this by typing the following lines in the command center of NetLogo:[2]

```
show (0.1 + 0.1 + 0.1)
show (0.1 + 0.1 + 0.1) > (0.3 + 0 + 0)
```

Therefore, in this particular example, agent A would copy agent B's strategy, but it should not copy it according to real arithmetic. Let us emphasize that this is not a problem specific to NetLogo. You would get the same result in any IEEE 754-compliant platform.

**Example 2**. To see floating-point errors in action in one of our models, you can run our last model with payoffs [[0 1.2] [-0.2 1]] and, after a few ticks, check the agents' payoffs by typing the following line in the command center:

```
show sort remove-duplicates [payoff] of players
```

When we do this, we get numbers such as -0.2000000000000018, 0.19999999999999996, 0.3999999999999999, 1.599999999999999, or 1.7999999999999998 (and also numbers very close to these, such as 1.6 and 1.8). By contrast, if you run the same model with the payoffs we used in the sample runs ([[0 1.1875] [-0.1875 1]]), you will not find any floating-point errors. This is the reason we chose these payoff values.

**Example 3**. Let us finish with a full-fledged example. Consider the Prisoner's dilemma [[0 1.2] [-0.4 1]] played on the network shown in figure 1 below, using total (accumulated) payoffs. Green nodes are cooperators and red nodes are defectors.

---

1. See Math section in NetLogo programming guide.
2. You may want to try this out in other programming languages such as Python, R, Javascript, Java, C, C++ ...
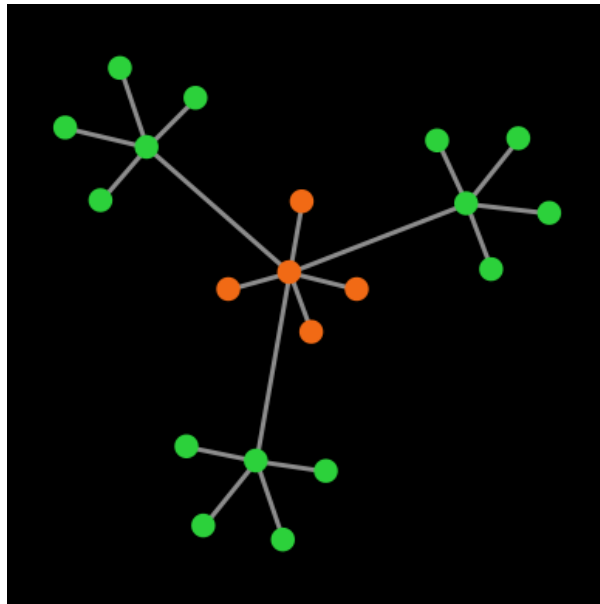
*Figure 1. A beautiful network, with cooperators in green and defectors in red*

The payoffs for the different agents are:

| Type of agent | Payoff computation | Payoff value (exact arithmetic) | Payoff value (floating-point arithmetic) |
|---|---|---|---|
| Cooperating hubs | 4 * 1 + 1 * (-0.4) | 3.6 | 3.6 [3] |
| Defecting hub | 4 * 0 + 3 * 1.2 | 3.6 | 3.5999999999999996 |
| Cooperating leaves | 1 * 1 | 1 | 1 |
| Defecting leaves | 1 * 0 | 0 | 0 |

Looking at the payoffs for the different agents above, we know that under decision rules imitate the best neighbor and imitate if better, the defecting hub will imitate one of the cooperating hubs (since 3.6 > 3.5999999999999996), and then the defecting leaves will follow suit. Thus, under these two decision rules, using IEEE 754 floating-point arithmetic, the population will always evolve to a state of full cooperation.

However, under real arithmetic, the end state of these simulations is very different. If agents use the imitate if better rule, the population will stay in the initial state forever; and if agents use the imitate the best neighbor rule, the simulation will end up in one of a set of four possible end states (allowing for symmetries). Table 1 below shows the probability of ending up in each of the possible different end states, assuming *prob-revision* = 1.

---

3. Number 3.6 is not exactly representable in binary, but since the result of 4 * 1 + 1 * (-0.4) is the closest floating-point number to 3.6, NetLogo outputs 3.6.
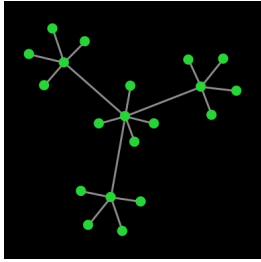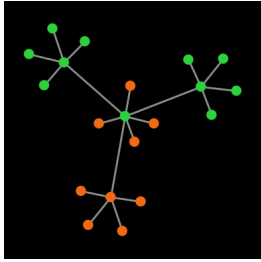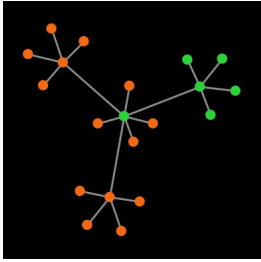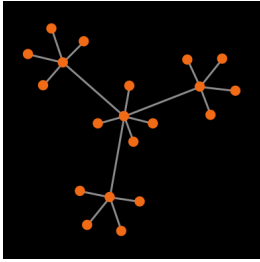
| End state |  |  |  |  |
|---|---|---|---|---|
| Number of cooperators | 20 | 11 | 6 | 0 |
| Probability | 9.68 % | 38.71 % | 38.71 % | 12.90 % |

*Table 1. Possible end states and their probabilities for a population of agents playing the Prisoner's dilemma with payoffs [[0 1.2] [–0.4 1]], using total (accumulated) payoffs and the imitate the best neighbor rule. Initial conditions are shown in figure 1 and prob-revision = 1. States that differ on a rotation of the network are lumped together*

It is clear that the results obtained with floating-point arithmetic are completely different from those obtained with real arithmetic. A simple way to avoid floating-point errors in this particular example would be to multiply all payoffs by 10. This is a good practice in general: if we have the freedom to do so, we should try to parameterize models using numbers that are exactly representable in base 2. For instance, if we want to explore different values of a parameter that ranges from 0 to 1, we should do it in steps of 0.125, rather than in steps of 0.1.

Nonetheless, it is also important to note that not all models are equally susceptible to floating-point errors, by any means. Decision rules such as the imitative pairwise-difference rule, the Fermi-m rule or the Santos-Pacheco rule are continuous in the sense that small changes in payoffs will lead to small changes in agents' behavior. Since floating-point errors in these models are often very small in magnitude, their impact on agents' behavior will also be small. Therefore, continuous models are not usually severely affected by floating-point errors.

To learn more about the potential impact of floating-point errors on agent-based models, see Polhill et al. (2006) and Izquierdo and Polhill (2006).

# 3. Use informative metrics

## 3.1. Metrics

We obtain information from simulation models by using metrics. Metrics here are just measurements of some aspect of the dynamics, such as the proportion of players using a certain strategy:

```
count players with [strategy = 1] / count players
```

These metrics could refer to one particular tick or to several ticks. For instance, it is common to measure the average proportion of players using a certain strategy over a certain period of time, since averages tend to have lower variability than single observations at a particular tick. This can be easily done in NetLogo as follows:

```
globals [
  ...
  first-tick-of-report
  last-tick-of-report
  cumulative-x1
]

to setup
  ...
  setup-reporters
end

to go
  ...
  tick
  update-reporters
  ...
end

;;;;;;;;;;;;;;;;;;;;;;;
;;;    Reporters     ;;;
;;;;;;;;;;;;;;;;;;;;;;;

to setup-reporters
  set first-tick-of-report 10001
  set last-tick-of-report  11000
  set cumulative-x1 0
end

to update-reporters
  if (ticks >= first-tick-of-report and ticks <= last-tick-of-report) [
    let x1 count players with [strategy = 1] / count players
    set cumulative-x1 (cumulative-x1 + x1)
  ]
end

to-report avg-x1
  report cumulative-x1 / (last-tick-of-report - first-tick-of-report + 1)
end
```

The code additions above ensure that, after tick last-tick-of-report, variable cumulative-x1 will contain the sum of all the values of x1 from first-tick-of-report up until last-tick-of-report, both inclusive. So, if called after tick last-tick-of-report, reporter avg-x1 will return the average proportion of players using strategy 1 in the interval [first-tick-of-report , last-tick-of-report]. We can then use reporter avg-x1 as a metric in BehaviorSpace.

## 3.2. Stability of metrics

Naturally, our metrics should measure something that is informative for our purposes. Here we comment on one aspect of metrics that is specially relevant in dynamic models: their stability. Most often, we wish to report a metric when its value has stabilized. There is nothing wrong in reporting the value of a metric when it is not yet stable, but the metric is usually more informative if we know that the reported value will not change much in future time-steps. How do we know if a metric has stabilized? One way is to compute the metric at a later tick in the same simulation and compare the two values. If the second value is systematically greater or smaller than the first value, then this is a strong indication that the metric has not yet stabilized. Let us see this with an example.

Consider the last model we have implemented (nxn-games-on-networks.nlogo), parameterized as follows:

```
["n-of-players-for-each-strategy" "[5000 5000]"]
["payoffs" "[[0 1.1]\n [0 1]]"]
["network-model" "preferential-attachment"]
["min-degree" 2]
["decision-rule" "imitative-pairwise-difference"]
["play-with" "all-nbrs-AVG-payoff"]
["prob-revision" 1]
["noise" 0]
```

This is a setting that corresponds to value b = 1.1 in figure 6 of Santos and Pacheco (2006) for the Barabási-Albert network. The value reported in that figure is the average proportion of cooperators over the period that goes from tick 10 001 to tick 11 000, and it is slightly greater than 20%.[4] To see whether this 1000-tick average is reasonably stable after 10 000 ticks, we can compute the same average starting at a later tick and compare the two averages for each simulation run. We can do this using the following code, which is not particularly elegant or efficient, but it will do the job:

```
globals [
  ...

  ;; for first metric
  first-tick-of-report
  last-tick-of-report
  cumulative-x1

  ;; for second metric
  first-tick-of-report-2
  last-tick-of-report-2
  cumulative-x1-2
```

---

4. See also figure 5 (top left) in Tomassini et al. (2007). Tomassini et al. (2007) note that 10 000 ticks may not be enough for the metric to stabilize and run their simulations for much longer (21 000 ticks). They report a value above 30%, which is significantly higher than the one reported in Santos and Pacheco (2006).

```
  ]

to setup
  ...
  setup-reporters
end

to go
  ...
  tick
  update-reporters
  ...
end

;;;;;;;;;;;;;;;;;;;;;;;
;;;    Reporters    ;;;
;;;;;;;;;;;;;;;;;;;;;;;

to setup-reporters
  ;; for first metric
  set first-tick-of-report 10001
  set last-tick-of-report  11000
  set cumulative-x1 0

  ;; for second metric
  set first-tick-of-report-2 11001
  set last-tick-of-report-2  12000
  set cumulative-x1-2 0
end

to update-reporters
  let x1 count players with [strategy = 1] / count players

  ;; for first metric
  if (ticks >= first-tick-of-report and ticks <= last-tick-of-report) [
    set cumulative-x1 (cumulative-x1 + x1)
  ]

  ;; for second metric
  if (ticks >= first-tick-of-report-2 and ticks <= last-tick-of-report-2) [
    set cumulative-x1-2 (cumulative-x1-2 + x1)
  ]
end

;; for first metric
to-report avg-x1
  report cumulative-x1 / (last-tick-of-report - first-tick-of-report + 1)
end

;; for second metric
to-report avg-x1-2
report cumulative-x1-2 / (last-tick-of-report-2 - first-tick-of-report-2 + 1)
end
```

We have run 100 simulations and obtained that the second average (avg-x2) was greater than the first one (avg-x1) in 91 runs, which suggests that the value is still increasing by tick 10 000. If the metric was stable by tick 10 000, we would expect the second average to be greater in approximately 50 runs (or fewer, if ties are likely). If the metric was stable, we would observe the result we have obtained (91/100 runs), or an even more extreme result, with probability less than $10^{-17}$.[5] Thus, this computational experiment provides very strong evidence against the hypothesis that the metric is stable by tick 10 000.

Figure 2 below shows a representative run of this setting, with a vertical black line at tick 10 000. It is clear that the proportion of cooperators (in green) has not stabilized by tick 10 000. In this run, the average proportion of cooperators is 23.6% over the period [10 001 , 11 000], but significantly higher afterwards (e.g., 33.9% over the period [110 001 , 111 000]).
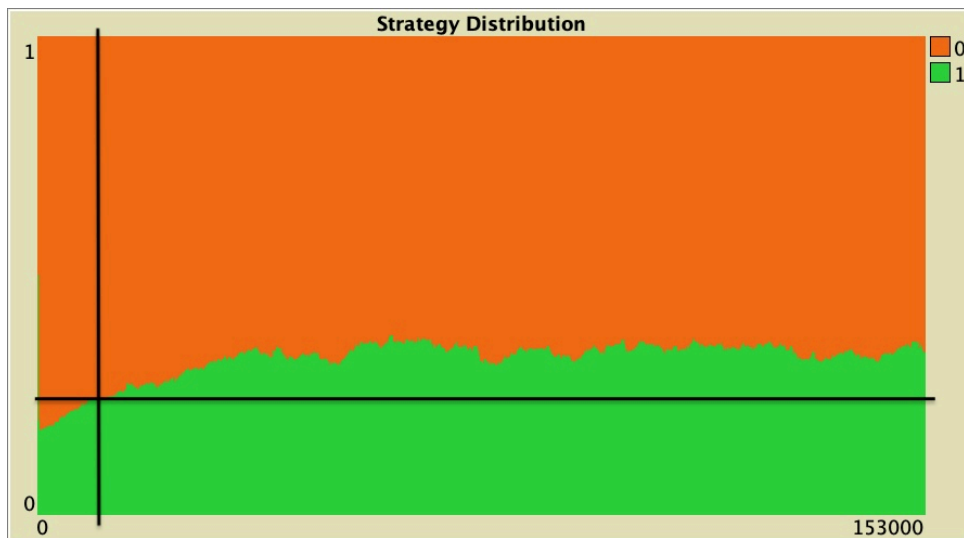


*Figure 2. Representative simulation of a population of 10 000 agents who play the (weak) Prisoner's Dilemma with payoffs [[0 1.1] [0 1]], using average payoffs and the imitative pairwise-difference decision rule. The network is preferential attachment with min-degree 2 (average degree ≈ 4)*

As a conclusion, we recommend to check the stability of reported metrics, as we have done here. It is a reasonably simple test that will make the metrics we report more informative.

# 4. Report meaningful statistics

When we run a stochastic model, we are usually interested in finding out not only how the model *can* behave, but also how it *usually* behaves. For this, we run many simulation runs and compute different statistics on the metrics obtained from each run. A statistic is any quantity computed from values

---

5. For simplicity, we have computed the $p$-value ignoring ties (or, equivalently, assuming they are counted as success or failure with equal probability), using a simple two-tailed binomial test. In Mathematica:

```
2 * CDF[BinomialDistribution[100, 0.5], 100 - 91]
```

in a sample. Here the sample consists of the values of a certain metric obtained from each of the simulation runs in the experiment. An example of a statistic is the average.

One point we would like to comment here is that we should try to report meaningful statistics. Some statistics can be deceiving in certain cases, and we should try to avoid them. For instance, reporting only the average of a bimodal distribution is not ideal, since the average masks the fact that the distribution is bimodal and may be easily misinterpreted.

We saw an example of a bimodal distribution in chapter IV-2. The 1000 simulations we ran there on a star network yielded an average percentage of B-strategists by tick 5000 equal to 50.33%. Looking at that average, one could (mistakenly) infer that, at tick 5000, roughly half the population uses strategy A and the other half uses strategy B. But actually, the truth is that the distribution is bimodal: in nearly half of the runs almost all agents were choosing strategy A, and in nearly half of the runs almost all agents were choosing strategy B (see figure 3 below).
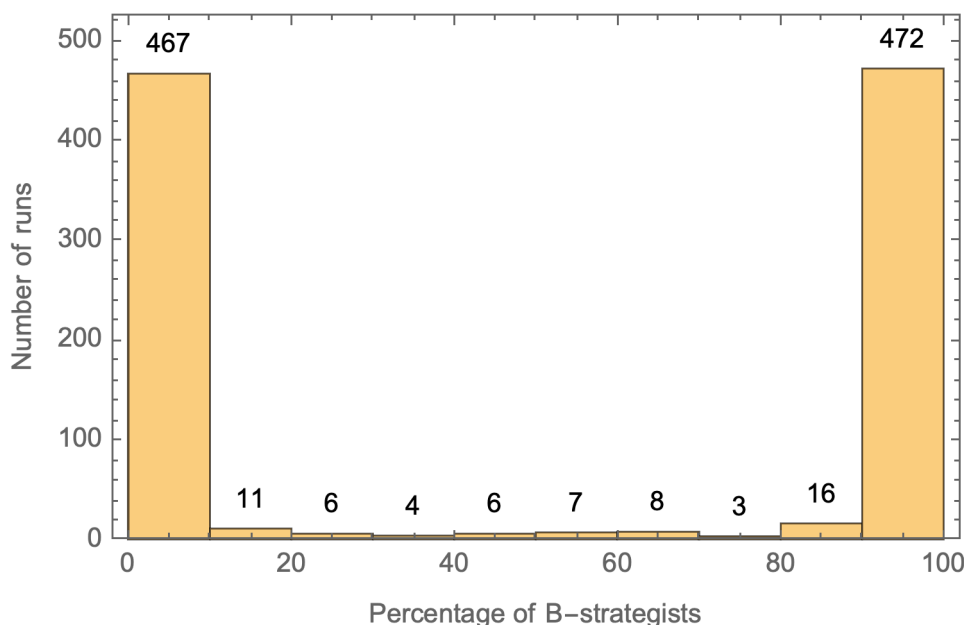


*Figure 3. Histogram showing the number of runs with different percentages of B-strategists in the experiment run on a star network in the sample runs of chapter IV-2*

Multimodal distributions are very common in evolutionary models: these models tend to spend most of the time in one out of a small set of possible regimes. We saw another example of this feature in the sample runs of the previous chapter. The simulations we ran there spent most of the time at one of two possible regimes: one where the proportion of B-strategists was very high ($\geq 95\%$) and another one where the proportion of B-strategists was very low ($\leq 5\%$). Reporting only the average of B-strategists across runs would have been misleading in that case. It is more meaningful to report the percentage of runs that were at each of the two regimes, as we did in figure 7 and in figure 9.

# 5. Derive sound conclusions

Our goal when we analyze any model is to fully understand *why* we are observing what we are observing. In other words, we would like to identify the features or assumptions in our model that are responsible for the emergence of the phenomena we observe. To check whether a certain assumption is responsible for a particular phenomenon, we must be able to replace the assumption with an alternative one. Thus the importance of implementing and exploring various alternative assumptions (e.g. different decision rules, different ways of computing payoffs, different network-generating algorithms, etc.) to make sure that our conclusions are sound. If we don't do this, we run the risk of over-extrapolating the scope of our model and even drawing unsound conclusions.

One feature that is often easy to implement and can provide important insights is *noise*. We have seen that the dynamics of many models can change significantly when we introduce small amounts of stochasticity in the agents' behavior or in the way they revise their strategies (e.g., *sequentially* or *simultaneously*). We believe it is important to assess the robustness of any observed phenomenon to low levels of stochasticity since, after all, our models are imperfect abstractions of real-world systems that are usually significantly less deterministic and well-behaved.

# 6. Final thoughts

We realize that, after reading this chapter, you may feel overwhelmed by the amount of things that can go wrong when trying to draw sound conclusions using an agent-based model. You're right: many things can go wrong. We hope that this book has opened your eyes in this respect but, more importantly, we hope that you have learned a wide range of useful techniques to avoid falling into the many traps that lie out there for the rigorous agent-based modeler. If you have managed to read this book until here, we are confident that you are now well prepared to implement and analyze your own agent-based models and to rigorously use these models to advance Science. It is also our hope that you will enjoy this process as much as we have enjoyed writing this book.

# 7. Exercises

**Exercise 1.** Implement a procedure named to setup-players-special to create the network shown in figure 1.

Hint to implement procedure to setup-players-special

You may find useful to use primitives `hatch` and `myself`, as in the following line:

```
hatch 4 [create-link-with myself]
```

**Exercise 2.** Can you analytically derive the probabilities of reaching each of the end states shown in Table 1?

**Exercise 3.** Once you have implemented procedure to setup-players-special in exercise 1 above, design a computational experiment to estimate the exact probabilities shown in Table 1.
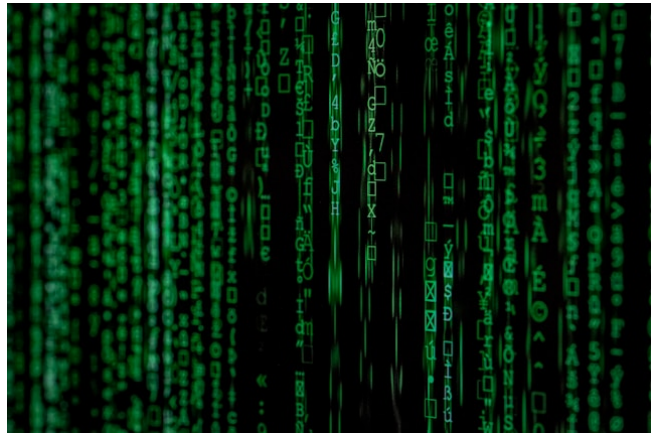


*Photo by Markus Spiske on Unsplash*

**Exercise 4.** Imagine that you suspect that your model spends a significant amount of time in a regime where the proportion of players using strategy 1 is in the interval [0.3, 0.4]. Implement a reporter that returns 1 if the simulation is in this regime and 0 otherwise.

**Exercise 5.** Generalize the reporter you have implemented in Exercise 4 above so the endpoints of the interval (0.3 and 0.4 in exercise 4) are inputs that can be chosen by the user, and there are also two additional inputs to control whether each of the interval endpoints is included or excluded.

**Exercise 6.** Once you have implemented the general reporter in Exercise 5, write the metrics you would have to include in BehaviorSpace to check whether the simulation is in each of the following regimes:

- Proportion of players using strategy 1 is in the interval [0.3 , 0.4]
- Proportion of players using strategy 1 is in the interval (0.2 , 0.3]
- Proportion of players using strategy 1 is in the interval [0.6 , 0.8)
- Proportion of players using strategy 1 is in the interval [0.9 , 1]

# PART V. AGENT-BASED MODELS vs ODE MODELS

# V-1. Introduction

Many models in Evolutionary Game Theory are described as systems of Ordinary Differential Equations (ODEs). The most famous example is the Replicator Dynamics (Taylor and Jonker, 1978), which reads:

$$\dot{x}_i = x_i(\pi_i - \pi)$$

where:

- $x_i$ is the fraction of $i$-strategists in the population.
- $\dot{x}_i = \frac{dx_i}{dt}$ describes how the fraction of $i$-strategists $x_i$ changes in time.
- $\pi_i$ is the expected payoff of strategy $i$. That is, if $\pi_{ij}$ is the payoff that an $i$-strategist obtains against a $j$-strategist, then $\pi_i = \sum_j x_j \pi_{ij}$. This is also the average payoff that an $i$-strategist would obtain if he played with the whole population.
- $\pi = \sum_j x_j \pi_j$ is the average payoff in the population.

ODE models are very different from the agent-based models we have considered in this book. Our goal in this Part V is to clarify the relationship between these two kinds of models. We will see that most ODE models in Evolutionary Game Theory can be seen as the mean dynamic of an agent-based model where agents follow a certain decision rule in a well-mixed population.[1] This implies that those ODE models provide a good deterministic approximation to the dynamics of the corresponding agent-based model over finite time spans *when the number of agents is sufficiently large* (Benaïm & Weibull, 2003; Sandholm, 2010a, chapter 10; Roth & Sandholm, 2013). For this reason, ODE models in Evolutionary Game Theory are often called infinite-population models (because they describe dynamics of populations whose size tends to infinity), while agent-based models are sometimes called finite-population models.[2]

It is also clear that the ODE models represent a higher level of abstraction than the agent-based models, in the sense that the variables in the ODE models are population-level aggregates, while the agent-based models are defined at the individual level (fig. 1).

---

1. The term *well-mixed population* refers to a population where all individuals are equally likely to interact with each other.
2. Another reason is that the fraction of $i$-strategists $x_i$ in ODE models changes continuously in the interval [0,1], while in a finite population of $N$ individuals, this fraction would have to be a multiple of $1/N$. See footnote 31 in Alexander (2023).
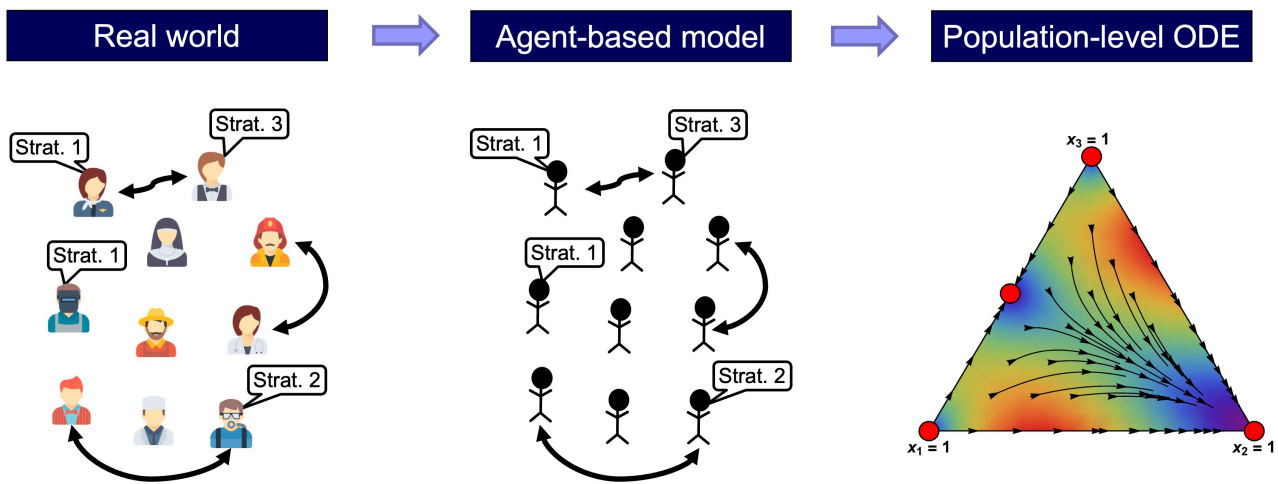
Figure. 1. *Agent–based models are described at the individual level, while ODE models are described at the population level*

In fact, in many cases, there is a wide range of different agent-based models which share the same mean dynamic. For instance, consider the replicator dynamics, which is often derived as the infinite-population limit of a certain model of biological evolution (see derivations in e.g. Weibull (1995, section 3.1.1), Vega-Redondo (2003, section 10.3.1) or Alexander (2023, section 3.2.1)). In the following chapters we will see that the replicator dynamics is also the mean dynamic of the following disparate agent-based models:[3]

- A model where agents in a well-mixed population follow the imitative pairwise-difference rule using expected payoffs (Helbing,1992; Schlag, 1998; Sandholm, 2010a, example 5.4.2; Sandholm, 2010b, example 1).

- A model where agents in a well-mixed population play the game just once with a random agent and follow the imitative pairwise-difference rule (Izquierdo et al., 2019, example A.2).

- A model where agents in a well-mixed population follow the so-called imitative linear-attraction rule using expected payoffs (Hofbauer, 1995a; Sandholm, 2010a, example 5.4.4; Sandholm, 2010b, example 1).

- A model where agents in a well-mixed population play the game just once with a random agent and follow the so-called imitative linear-attraction rule (Izquierdo et al., 2019, remark A.3).

- A model where agents in a well-mixed population follow the so-called imitative linear-dissatisfaction rule using expected payoffs (Weibull, 1995, section 4.4.1; Björnerstedt and Weibull, 1996; Sandholm, 2010a, example 5.4.3; Sandholm, 2010b, example 1).

- A model where agents in a well-mixed population play the game just once with a random agent and follow the so-called imitative linear-dissatisfaction rule (Izquierdo et al., 2019, remark A.3).

In the next chapter, we extend the (well-mixed population) model we developed in Part II by including

---

3. Some of these models lead to the replicator dynamics *up to a speed factor*, i.e., a constant may appear multiplying the whole right-hand side of the equation of the replicator dynamics. This constant can be interpreted as a change of time scale.

different decision rules that have been studied in the literature and different ways of computing payoffs. Then, in chapter V-3, we derive the mean dynamic of each of the possible parameterizations of this extended agent-based model. We will see that some parameterizations that generate different stochastic dynamics share the same mean dynamic. In this way, we hope that the (many-to-one) relationship between agent-based models and ODEs will be perfectly clear.

# V-2. A rather general model for games played in well-mixed populations

## 1. Goal

The goal of this chapter is to extend the well-mixed population model we developed in chapter II-3 (nxn-imitate-if-better-noise) by adding two features that will make our program significantly more general:

- The possibility to use **expected payoffs**. In all the models we developed in Part II, agents obtain their payoffs by playing with **one** agent, chosen at random. In this chapter we will allow agents to use expected payoffs. The expected payoff of strategy $i$ is also the average payoff that an $i$-strategist would obtain if she played with the whole population.

- The possibility to model other decision rules besides the imitate if better rule.

## 2. Motivation

Once we have implemented different decision rules and the option to use expected payoffs, we will be able to model many different agent-based evolutionary dynamics. In the next chapter, we will develop the mean dynamic of each of these agent-based evolutionary dynamics, and we will see that they correspond to ODE models that have been studied in the literature.

## 3. Description of the model

We depart from the program we implemented in chapter II-3 (nxn-imitate-if-better-noise).[1] You can find the full description of this model in chapter II-3. Our extension will include the following three additional parameters:

- *payoff-to-use*. This parameter specifies the type of payoff that agents use in each tick. It will be implemented with a chooser, with two possible values:

    - "*play-with-one-rd-agent*": agents play with another agent chosen at random.

    - "*use-strategy-expected-payoff*": agents use their strategy's expected payoff.

- *decision-rule*. This parameter determines the decision rule that agents follow to update their strategies, just like in the models we developed in chapter III-4 and chapter IV-4. Note that

---

1. In chapter II-4 we were able to implement a more efficient version of this model, but the speed boost came at the expense of making our code less readable. Here we want to focus on code readability, so we start with the most natural implementation of the model, i.e. nxn-imitate-if-better-noise. Nonetheless, we can add the primitive `no-display` in procedure to setup, right after `clear-all`, to improve the efficiency of the model at hardly any cost.

all decision rules use the agents' payoffs, and these are computed following the method prescribed by parameter *payoff-to-use*. Parameter *decision-rule* will be implemented with a chooser, with seven possible values:

- "*imitate-if-better*". This is the imitate if better rule already implemented in the current model.

- "*imitative-pairwise-difference*". This is the imitative pairwise-difference rule we saw in chapter I-2. Under this rule, the revising agent looks at another individual at random and considers imitating her strategy only if her payoff is greater than the reviser's payoff; in that case, he switches with probability proportional to the payoff difference. In order to turn the payoff difference into a meaningful probability, we divide the payoff difference by the maximum possible payoff minus the minimum possible payoff.

- "*imitative-linear-attraction*". Under this rule, the revising agent selects another individual at random and imitates her strategy with probability equal to the difference between the observed individual's payoff and the minimum possible payoff, divided by the maximum possible payoff minus the minimum possible payoff. (The revising agent's payoff is ignored.)

- "*imitative-linear-dissatisfaction*". Under this rule, the revising agent selects another agent at random and imitates her strategy with probability equal to the difference between the maximum possible payoff and his own payoff, divided by the maximum possible payoff minus the minimum possible payoff. (The observed agent's payoff is ignored.)

The four decision rules above are **imitative**, i.e. they all start by selecting one agent and consider imitating her strategy. In contrast, the following decision rules are **direct** (Sandholm, 2010a).[2] Under direct decision rules, revising agents choose candidate strategies directly without looking at any agent, so a strategy's popularity does not directly influence the probability with which it is considered. In the following *direct* decision rules, if *payoff-to-use* = "*play-with-one-rd-agent*", each strategy is tested against one random individual, potentially different in each test.

- "*direct-best*". The revising agent selects the strategy with the greatest payoff. Ties are resolved uniformly at random.

- "*direct-pairwise-difference*". The revising agent randomly selects another strategy, and considers adopting it only if its payoff is greater; in that case, he switches with probability proportional to the payoff difference. In order to turn the difference in payoffs into a meaningful probability, we divide the payoff difference by the maximum possible payoff minus the minimum possible payoff.

- "*direct-positive-proportional-m*". The revising agent selects one strategy at random, with probabilities proportional to payoffs raised to parameter *m*, and adopts it. Parameter *m* controls the intensity of selection (see below). We do not allow for negative payoffs when using this rule.

---

2. There are decision rules that are neither imitative nor direct. See, e.g., Lahkar and Sandholm (2008).

- *m*. This is the parameter that controls the intensity of selection in decision rule *direct-positive-proportional-m*.

Everything else stays as described in chapter II-3.

# 4. Extension I. Implementation of different ways of computing payoffs

In this first extension, we just want to implement the option to use expected payoffs.

## 4.1. Skeleton of the code

The main procedure we have to modify is to update-payoff, which is where payoffs are updated. Figure 1 below shows the skeleton of the new implementation of procedure to update-payoff.
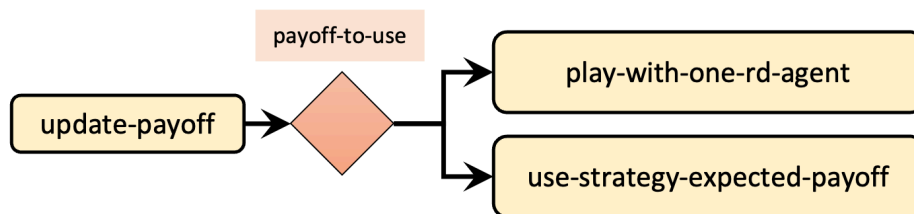


*Figure 1. Skeleton of procedure to update–payoff*

## 4.2. Interface design

We depart from the model we created in chapter II-3 (nxn-imitate-if-better-noise), so if you want to preserve it, now is a good time to duplicate it. The current interface looks as shown in figure II-3-1. The only change we have to make to the interface is to add parameter *payoff-to-use*. We should add it as a chooser with possible values "*play-with-one-rd-agent*" and "*use-strategy-expected-payoff*". The new interface should look as figure 2 below.
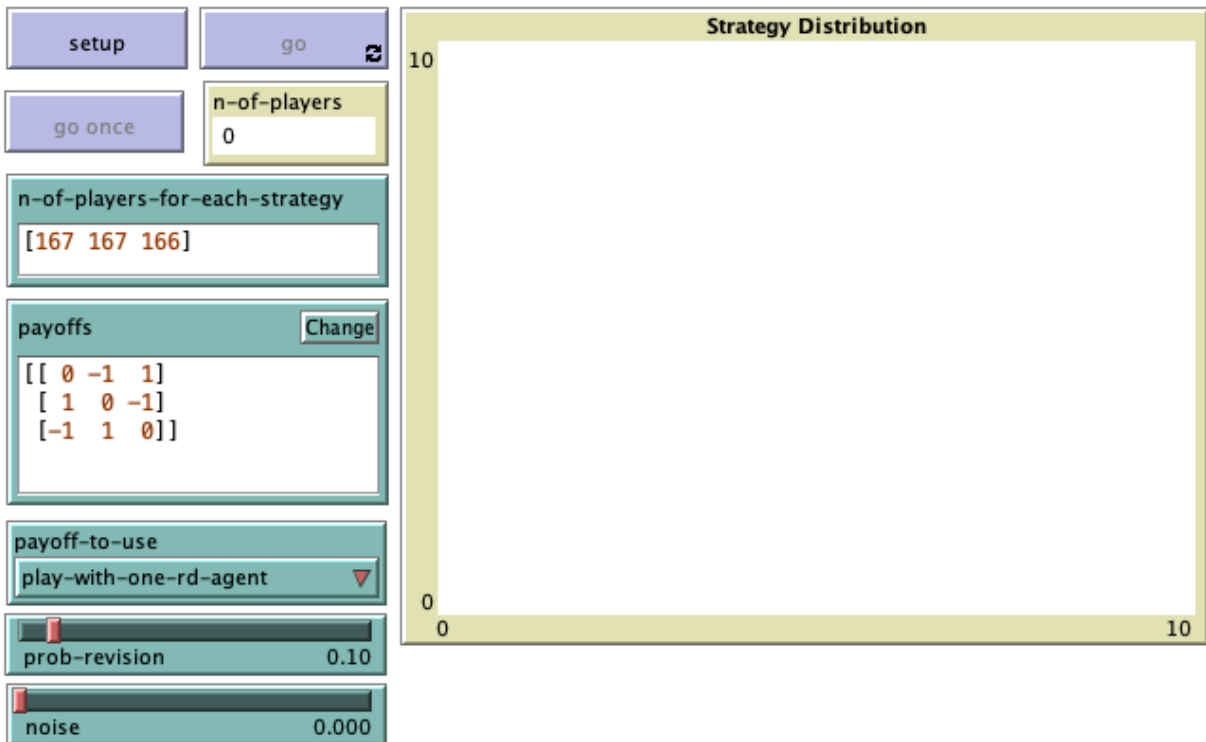
*Figure 2. Interface design for the first extension*

## 4.3. Code

### 4.3.1. Implementation of to update-payoff

The way payoffs are computed is determined by parameter *payoff-to-use*:

- If *payoff-to-use* = "*play-with-one-rd-agent*", agents play with another agent chosen at random.

- If *payoff-to-use* = "*use-strategy-expected-payoff*", agents use their strategy's expected payoff.

A nice and modular way of implementing procedure to update-payoff is to create two new procedures, i.e., to play-with-one-rd-agent and to use-strategy-expected-payoff and call one or the other depending on the value of *payoff-to-use* (see skeleton in figure 1). Thus, the new code of to update-payoff could look as follows:

```
to update-payoff
  ifelse payoff-to-use = "play-with-one-rd-agent"
    [play-with-one-rd-agent]
    [use-strategy-expected-payoff]
end
```

Given that we have chosen the names of the two new procedures to match the possible values of parameter *payoff-to-use*, we can also use run (a primitive that can take a string containing the name of a command as an input, and it runs the command) as follows:

```
to update-payoff
  run payoff-to-use
end
```

Our following step is to implement the two new procedures.

## 4.3.2. Implementation of to play-with-one-rd-agent

Note that in our baseline model nxn-imitate-if-better-noise, agents obtained a payoff by playing with another agent chosen at random. Thus, for the implementation of to play-with-one-rd-agent, we can just use the code that we previously had in procedure to update-payoff:

```
to play-with-one-rd-agent
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end
```

## 4.3.3. Implementation of to use-strategy-expected-payoff

This new procedure should assign the appropriate expected payoff to the agent who runs it. Note that we do not want every agent to compute the expected payoff of their strategy. This would be inefficient because there are usually many more agents than strategies. Instead, we should compute the expected payoff of each strategy just once in each tick, and have these payoffs ready for agents to read.

We will implement a procedure to compute expected payoffs in the next section, but for now, let us assume that these expected payoffs are available to agents in a global variable called strategy-expected-payoffs. This variable will contain a list with the expected payoff of each strategy, in order. Assuming that, the code for procedure to use-strategy-expected-payoff is particularly simple:

```
to use-strategy-expected-payoff
  set payoff item strategy strategy-expected-payoffs
end
```

## 4.3.4. Implementation of to update-strategy-expected-payoffs

Our goal now is to code the procedure in charge of computing the expected payoff for each strategy and of storing them in a global variable named strategy-expected-payoffs. Let us call this new procedure to update-strategy-expected-payoffs, and work through the logic together.

It is often easier to start with a concrete example and then try to generalize. Let us start by thinking how to compute the expected payoff of strategy 0. To do this, we need the payoffs that strategy 0 can obtain (e.g., [0 -1 1] in figure 2) and we need the frequencies of each strategy in the population (e.g. [0.2 0.3 0.5]). Then, we would only have to multiply the two lists element by element ( [ 0*0.2 -1*0.3 1*0.5 ] ) and sum all the elements of the resulting list (0*0.2 + -1*0.3 + 1*0.5).

Our current code already computes the frequencies of each strategy in procedure to update-graph, so we already have the code for that:

```
let strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
    ] strategy-numbers
```

Assuming we have the strategy frequencies stored in strategy-frequencies, to compute the expected payoff of strategy 0 in our example we would only have to do:

```
sum (map * [ 0 -1 1 ] strategy-frequencies)
```

Now is the time to generalize. To compute the expected payoff of each of the strategies, we just have to apply the code snippet above to the list of payoffs of each strategy. Note that the payoff matrix contains precisely the lists of payoffs for each strategy, so we can use map as follows:

```
set strategy-expected-payoffs map [ list-of-payoffs ->
    sum (map * list-of-payoffs strategy-frequencies)
    ] payoff-matrix
```

As mentioned before, since we want agents to be able to read the strategy expected payoffs, we should define variable strategy-expected-payoffs as global. Also, there is no point in computing the strategies frequencies more than once in each tick, so we should also define strategy-frequencies as a global variable, and update it only in one place. And finally, now we will be using the strategy numbers at two places (procedures to update-strategy-expected-payoffs and to update-graph), so we could also define strategy-numbers as a global variable:

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players

  strategy-numbers          ;; <= new global variable
  strategy-frequencies      ;; <= new global variable
  strategy-expected-payoffs ;; <= new global variable
]
```

With all this, the code for procedure to update-strategy-expected-payoffs would be:

```
to update-strategy-expected-payoffs
  set strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers
  set strategy-expected-payoffs map [ list-of-payoffs ->
      sum (map * list-of-payoffs strategy-frequencies) ] payoff-matrix
end
```

Since we are updating strategy-frequencies in this procedure, and the variable is now global, we can remove that computation from procedure to update-graph. Similarly, since variable strategy-numbers

is now global and its value does not change over the course of a simulation run, we should move its computation from to update-graph to a setup procedure, such as to setup-payoffs:

```
to update-graph
; let strategy-numbers (range n-of-strategies)
; let strategy-frequencies map [ n ->
;     count players with [strategy = n] / n-of-players
;    ] strategy-numbers

set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
end
```

```
to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
  set strategy-numbers range n-of-strategies   ;; <= new line
end
```

Now, we should make sure that we call procedure to update-strategy-expected-payoffs (where strategy-frequencies is updated) before running procedure to update-graph, where these frequencies are used. We do that at procedures to setup and to go. We also take this opportunity to improve the efficiency of the model by including the primitive `no-display` in procedure to setup, right after `clear-all`.

```
to setup
  clear-all
  no-display ;; <= new line to improve efficiency
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-strategy-expected-payoffs ;; <= new line
  update-graph
end
```

```
to go
  ask players [update-payoff]

  ask players [
    if (random-float 1 < prob-revision) [
```

```
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick

  update-strategy-expected-payoffs ;; <= new line
  update-graph
end
```

Finally, note that, since procedure to update-strategy-expected-payoffs is called from to go, we know that variable strategy-expected-payoffs is updated in every tick, as required.

There is a subtle final issue here. We have to make sure that, given a certain population state (i.e. once agents have updated their strategies), variable strategy-expected-payoffs is updated **before** agents update their payoffs in procedure to update-payoff. Note that this is so in our current code, even at the first tick (since procedure to update-strategy-expected-payoffs is called from to setup too). In other words, whenever agents update their payoffs, variable strategy-expected-payoffs contains the updated expected payoffs, i.e., the ones corresponding to the current population state.

Table 1 below shows a sketch of the order in which the main procedures are executed in a simulation run. The lines highlighted in yellow indicate when the population state changes. Note that, in between highlighted lines, procedure to update-strategy-expected-payoffs is executed **before** agents update their payoffs in procedure to update-payoff.

| | | | |
|---|---|---|---|
| ↓ | | ↓ ........ | ;; <= various setup procedures |
| ↓ setup | | ↓ **update-strategy-expected-payoffs** | |
| ↓ | | ↓ update-graph | |
| ↓ | | ↓ **ask players [update-payoff]** | |
| ↓ | | ↓ ........ | ;; <= agents update strategy-after-revision |
| ↓ | go | ↓ ask players [update-strategy] | ;; **<= change of population state** |
| ↓ | | ↓ tick | |
| ↓ | | ↓ **update-strategy-expected-payoffs** | |
| ↓ | | ↓ ........ | ;; <= update-graph |
| ↓ | | ↓ **ask players [update-payoff]** | |
| ↓ | | ↓ ........ | ;; <= agents update strategy-after-revision |
| ↓ | go | ↓ ask players [update-strategy] | ;; **<= change of population state** |
| ↓ | | ↓ tick | |
| ↓ | | ↓ **update-strategy-expected-payoffs** | |
| ↓ | | ↓ ........ | ;; <= update-graph |
| ↓ | | ↓ **ask players [update-payoff]** | |
| ↓ | | ↓ ........ | ;; <= agents update strategy-after-revision |
| ↓ | go | ↓ ask players [update-strategy] | ;; **<= change of population state** |
| ↓ | | ↓ tick | |
| ↓ | | ↓ update-strategy-expected-payoffs | |
| ↓ | | ↓ ........ | ;; <= update-graph |

*Table 1. Sketch of the order in which the main procedures are executed in a simulation run*

## 4.3.5. Final checks

We can conduct a few simple checks to gain confidence in our code. For instance, to see the different payoffs that 0-strategists obtain, we can include the following line in procedure to go, right after agents update their payoffs:

```
show remove-duplicates [payoff] of players with [strategy = 0]
```

```
to go
  ask players [update-payoff]
  show remove-duplicates [payoff] of players with [strategy = 0]

  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
```

```
    tick
    update-strategy-expected-payoffs
    update-graph
 end
```

With this code in place, please, try to answer the four questions below:

1. Can you find out what we should see in the command center if we click on setup and then on go once with the setting shown in figure 2?

Since *payoff-to-use* = "*play-with-one-rd-agent*", agents play with another agent chosen at random. Thus, most likely, some 0-strategists will meet another 0-strategist (and obtain a payoff of 0), some will meet a 1-strategist (and obtain a payoff of -1), and some will meet a 2-strategist (and obtain a payoff of 1). Therefore, almost for certain we will see a list containing these three payoffs [0 -1 1], in any order.

2. And if we change *payoff-to-use* to = "*use-strategy-expected-payoff*"?

The payoff for all 0-strategists is now the expected payoff of strategy 0, so we should only see one number in the list. At the initial state, this number should be:
(0*(167/500) + -1*(167/500) + 1*(166/500)) = -1/500 = -0.002
We actually see [-0.0020000000000000018] due to floating-point errors.

3. What should we observe if we remove **with** [strategy = 0] from the line we added in procedure to go, still using *payoff-to-use* to = "*use-strategy-expected-payoff*"?

We should see a list containing the three expected payoffs, in any order. At the initial state, these payoffs are:
( 0*(167/500) + -1*(167/500) + 1*(166/500)) = -1/500 = -0.002
( 1*(167/500) + 0*(167/500) + -1*(166/500)) = 1/500 = 0.002
(-1*(167/500) + 1*(167/500) + 0*(166/500)) = 0

We actually see [0 0.0020000000000000018 -0.0020000000000000018] due to floating-point errors.

## 4.4. Complete code of Extension I in the Code tab

Well done! We have now finished the first extension! You can use the following link to download the complete NetLogo model: nxn-imitate-if-better-noise-payoff-to-use.nlogo.

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
  strategy-numbers         ;; <= new global variable
  strategy-frequencies     ;; <= new global variable
  strategy-expected-payoffs ;; <= new global variable
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]

;;;;;;;;;;;;;;
;;; SETUP ;;;
;;;;;;;;;;;;;;

to setup
  clear-all
  no-display ;; <= new line to improve efficiency
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-strategy-expected-payoffs ;; <= new line
  update-graph
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
  set strategy-numbers range n-of-strategies  ;; <= new line
end

to setup-players
```

```
    let initial-distribution read-from-string n-of-players-for-each-strategy
    if length initial-distribution != length payoff-matrix [
      user-message (word "The number of items in\n"
        "n-of-players-for-each-strategy (i.e. "
        length initial-distribution "):\n"
        n-of-players-for-each-strategy
        "\nshould be equal to the number of rows\n"
        "in the payoff matrix (i.e. "
        length payoff-matrix "):\n"
        payoffs
      )
    ]

    let i 0
    foreach initial-distribution [ j ->
      create-players j [
        set payoff 0
        set strategy i
        set strategy-after-revision strategy
      ]
      set i (i + 1)
    ]

    set n-of-players count players
end

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

;;;;;;;;;;
;;; GO ;;;
;;;;;;;;;;

to go
  ask players [update-payoff]
  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick
  update-strategy-expected-payoffs ;; <= new line
  update-graph
end

;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; UPDATE PAYOFFS ;;;
;;;;;;;;;;;;;;;;;;;;;;

to update-strategy-expected-payoffs
  set strategy-frequencies map [ n ->
        count players with [strategy = n] / n-of-players
      ] strategy-numbers
  set strategy-expected-payoffs map [ list-of-payoffs ->
      sum (map * list-of-payoffs strategy-frequencies) ] payoff-matrix
end

to update-payoff
  run payoff-to-use
end

to play-with-one-rd-agent
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end

to use-strategy-expected-payoff
  set payoff item strategy strategy-expected-payoffs
end


;;;;;;;;;;;;;;;;;;;;;;;;
;;; UPDATE STRATEGIES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [
      let observed-player one-of other players
      if ([payoff] of observed-player) > payoff [
        set strategy-after-revision ([strategy] of observed-player)
      ]
    ]
end

to update-strategy
  set strategy strategy-after-revision
end


;;;;;;;;;;;;;;;;;;;;
;;; UPDATE GRAPH ;;;
;;;;;;;;;;;;;;;;;;;;

to update-graph
  ; let strategy-numbers (range n-of-strategies)
  ; let strategy-frequencies map [ n ->
  ;      count players with [strategy = n] / n-of-players
  ;    ] strategy-numbers
```

```
    set-current-plot "Strategy Distribution"
    let bar 1
    foreach strategy-numbers [ n ->
      set-current-plot-pen (word n)
      plotxy ticks bar
      set bar (bar - (item n strategy-frequencies))
    ]
    set-plot-y-range 0 1
  end
```

# 5. Extension II. Implementation of different decision rules

In our second extension, we will implement the following procedures, one for each decision rule:

- to imitate-if-better-rule
- to imitative-pairwise-difference-rule
- to imitative-linear-attraction-rule
- to imitative-linear-dissatisfaction-rule
- to direct-best-rule
- to direct-pairwise-difference-rule
- to direct-positive-proportional-m-rule

Note that we have chosen the name of the procedures to match the possible values of parameter *decision-rule* (plus the suffix "-rule").

## 5.1. Skeleton of the code

The main procedure we have to modify is to update-strategy-after-revision, which is where the different decision rules are called from. Figure 3 below shows the skeleton of the new implementation of procedure to update-strategy-after-revision.
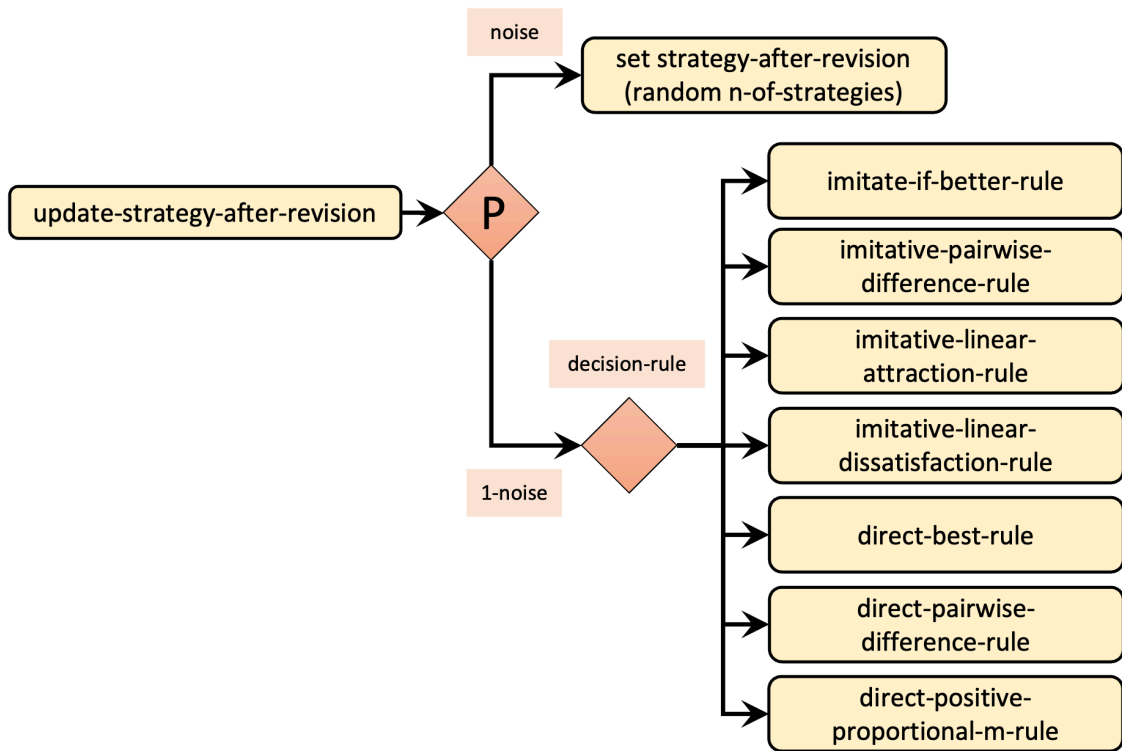
*Figure 3. Skeleton of procedure* to update–strategy–after–revision

## 5.2. Interface design

The current interface looks as shown in figure 2. The only change we have to make to the interface is to add the following parameters:

- *decision-rule*. We can add it as a chooser with possible values "*imitate-if-better*", "*imitative-pairwise-difference*", "*imitative-linear-attraction*", "*imitative-linear-dissatisfaction*", "*direct-best*", "*direct-pairwise-difference*", and "*direct-positive-proportional-m*".

- *m*. We can add it as a slider (with *minimum* = 0 and *increment* = 0.1).

The new interface should look as figure 4 below.
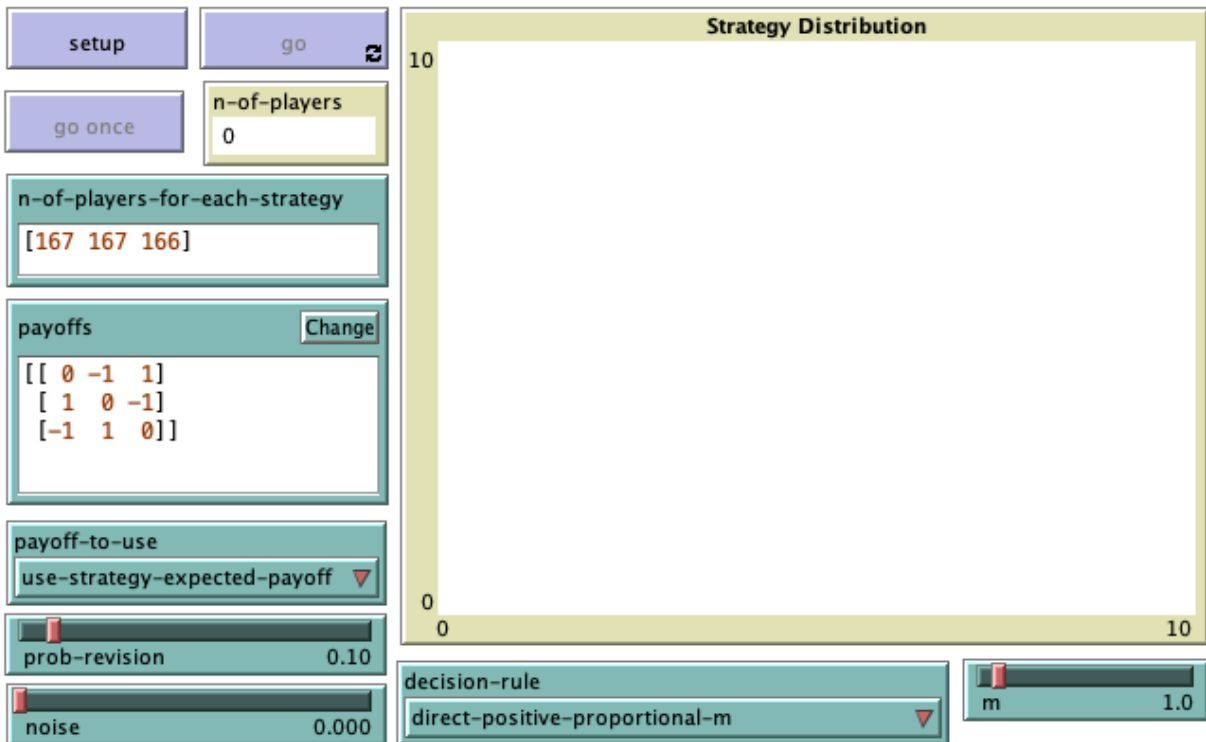
*Figure 4. Interface design for the first extension*

## 5.3. Code

### 5.3.1. Global variables

For some decision rules, agents will need access to the minimum and the maximum possible payoffs, and to the maximum possible payoff difference. For this reason, it makes sense to define these as global variables.

```
globals [
  payoff-matrix
  n-of-strategies
  n-of-players
  strategy-numbers
  strategy-frequencies
  strategy-expected-payoffs

  min-of-payoff-matrix  ;; <= new global variable
  max-of-payoff-matrix  ;; <= new global variable
  max-payoff-difference ;; <= new global variable
]
```

Since the value of these global variables will not change over the course of a simulation run, we should set them at a setup procedure. The most appropriate place is at procedure to setup-payoffs.

```
to setup-payoffs
  set payoff-matrix read-from-string payoffs
```

```
   set n-of-strategies length payoff-matrix
   set strategy-numbers range n-of-strategies

   ;; new lines below
   set min-of-payoff-matrix min reduce sentence payoff-matrix
   set max-of-payoff-matrix max reduce sentence payoff-matrix
   set max-payoff-difference (max-of-payoff-matrix - min-of-payoff-matrix)
 end
```

## 5.3.2. Implementation of to update-strategy-after-revision

Let us start with the implementation of procedure to update-strategy-after-revision. Given the names we have chosen for the procedures for the different decision rules (see skeleton in Figure 3), can you venture a simple implementation for our new procedure to update-strategy-after-revision using primitive run?

Implementation of procedure to update-strategy-after-revision

```
to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [ run (word decision-rule "-rule") ]
  end
```

Now we just have to implement the procedures for each of the seven decision rules. If you have managed to follow this book up until here, you are most likely ready to implement them by yourself!

### 5.3.3. Imitative decision rules

*Implementation of to imitate-if-better-rule*

This is the decision rule we had implemented in our baseline model nxn-imitate-if-better-noise, so we can just copy the code.

Implementation of procedure to imitate-if-better-rule

```
to imitate-if-better-rule
  let observed-player one-of other players
```

```
    if ([payoff] of observed-player) > payoff [
      set strategy-after-revision ([strategy] of observed-player)
    ]
  end
```

*Implementation of to imitative-pairwise-difference-rule*

To implement this rule, the code we wrote in chapter III-4 will be very useful.

Implementation of procedure to imitative-pairwise-difference-rule

```
to imitative-pairwise-difference-rule
  let observed-player one-of other players

  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - payoff)

  set strategy-after-revision ifelse-value
    (random-float 1 < (payoff-diff / max-payoff-difference))
      [ [strategy] of observed-player ]
      [ strategy ]
    ;; If your strategy is the better, payoff-diff is negative,
    ;; so you are going to stick with it.
    ;; If it's not, you switch with probability
    ;; (payoff-diff / max-payoff-difference)
  end
```

*Implementation of to imitative-linear-attraction-rule*

Can you try to implement this rule by yourself?

Implementation of procedure to imitative-linear-attraction-rule

```
to imitative-linear-attraction-rule
  let observed-player one-of players
```

```
    set strategy-after-revision ifelse-value
      (random-float 1 <
            ([payoff] of observed-player - min-of-payoff-matrix) /
            ( max-of-payoff-matrix - min-of-payoff-matrix) )
        [ [strategy] of observed-player ]
        [ strategy ]
  end
```

*Implementation of to imitative-linear-dissatisfaction-rule*

The code for this rule is similar to the previous one. Can you implement it?

Implementation of procedure to imitative-linear-dissatisfaction-rule

```
  to imitative-linear-dissatisfaction-rule
    let observed-player one-of players

    set strategy-after-revision ifelse-value
      (random-float 1 <
            (max-of-payoff-matrix - payoff) /
            (max-of-payoff-matrix - min-of-payoff-matrix) )
        [ [strategy] of observed-player ]
        [ strategy ]
  end
```

## 5.3.4. Direct decision rules

In this section, we implement three direct decisions rules: "*direct-best*", "*direct-pairwise-difference*", and "*direct-positive-proportional-m*". In *direct* decision rules, agents consider different candidate strategies, and choose one of them based on their assigned payoffs. Thus, the first thing we should do is to implement a procedure that assigns payoffs to strategies.

*Assigning payoffs to strategies*

The payoff assigned to each strategy is determined by parameter *payoff-to-use*:

- If *payoff-to-use* = "*play-with-one-rd-agent*", the payoff assigned to strategy $i$ will be the payoff resulting from trying out strategy $i$ against one agent chosen at random from the population. Importantly, according to the description of the model, every time a strategy is tested by a reviser, it should be tested against a newly (randomly) drawn agent.

- If *payoff-to-use* = "*use-strategy-expected-payoff*", the payoff assigned to strategy $i$ will be strategy $i'$s expected payoff.

Our goal now is to implement a reporter named to-report payoff-for-strategy [$s$], which takes a strategy number $s$ as an input, and returns the appropriate payoff for strategy $s$, taking into account the current value of parameter *payoff-to-use*. Bearing in mind that strategies' expected payoffs are stored in global variable strategy-expected-payoffs, can you implement to-report payoff-for-strategy?

### Implementation of reporter to-report payoff-for-strategy

```
to-report payoff-for-strategy [s]
  report ifelse-value payoff-to-use = "play-with-one-rd-agent"
    [item ([strategy] of one-of other players) (item s payoff-matrix)]
    [item s strategy-expected-payoffs]
end
```

*Implementation of to direct-best-rule*

To implement this rule, it is useful to create a list of two-item lists (i.e., a list of pairs) that we can call pairs-strategy-payoff. In this list, there will be one pair for each strategy. The first item in each pair will be the strategy number and the second item will be the payoff assigned to that strategy:

```
let pairs-strategy-payoff (map [
    [s] -> list s (payoff-for-strategy s)
  ] strategy-numbers)
```

With this list in place, you can use primitive `sort-by` to select one of the pairs with the highest payoff. Remember that ties should be resolved at random. Admittedly, implementing this procedure is not easy, so if you manage to do it, you can certainly give yourself a pat on the back!

### Implementation of procedure to direct-best-rule

```
to direct-best-rule
  let pairs-strategy-payoff (map [
    [s] -> list s (payoff-for-strategy s)
  ] strategy-numbers)
```

```
    ;; the following line ensures that ties
    ;; are resolved (uniformly) at random
    set pairs-strategy-payoff shuffle pairs-strategy-payoff

    let sorted-list sort-by [ [l1 l2] -> last l1 > last l2 ]
        pairs-strategy-payoff
    set strategy-after-revision (first (first sorted-list))
  end
```

*Implementation of to direct-pairwise-difference-rule*

To implement this rule, the code of procedure to imitative-pairwise-difference-rule and primitive `remove-item` can be useful.

Implementation of procedure to direct-pairwise-difference-rule

```
to direct-pairwise-difference-rule
  let candidate-strategy one-of (remove-item strategy strategy-numbers)

  ;; compute difference in payoffs
  let payoff-diff ((payoff-for-strategy candidate-strategy) - payoff)

  set strategy-after-revision ifelse-value
    (random-float 1 < (payoff-diff / max-payoff-difference))
      [ candidate-strategy ]
      [ strategy ]
    ;; If your strategy is the better, payoff-diff is negative,
    ;; so you are going to stick with it.
    ;; If it's not, you switch with probability
    ;; (payoff-diff / max-payoff-difference)
end
```

*Implementation of to direct-positive-proportional-m-rule*

To implement this rule, it is useful to have a look at the code of procedure to direct-best-rule, load the `rnd` extension, and use primitive `rnd:weighted-one-of-list`.

Implementation of procedure to direct-positive-proportional-m-rule

```
to direct-positive-proportional-m-rule
  let pairs-strategy-payoff (map [
    [s] -> list s ((payoff-for-strategy s) ^ m)
  ] strategy-numbers)

  set strategy-after-revision first
      rnd:weighted-one-of-list pairs-strategy-payoff [ [p] -> last p ]
end
```

To avoid errors when payoffs are negative, we will include procedure to check-payoffs-are-non-negative, just like we did in chapter III-4 and chapter chapter IV-4.

```
to check-payoffs-are-non-negative
  if min-of-payoff-matrix < 0 [
    user-message (word
      "Since you are using decision-rule =\n"
      "imitative-positive-proportional-m,\n"
      "all elements in the payoff matrix\n"
      payoffs
      "\nshould be non-negative numbers.")
  ]
end
```

An appropriate place to call this procedure would be at the end of procedure to setup-payoffs, which would then be as follows:

```
to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
  set strategy-numbers range n-of-strategies

  ;; new lines below
  set min-of-payoff-matrix min reduce sentence payoff-matrix
  set max-of-payoff-matrix max reduce sentence payoff-matrix
  set max-payoff-difference (max-of-payoff-matrix - min-of-payoff-matrix)

  if decision-rule = "direct-positive-proportional-m"
      [ check-payoffs-are-non-negative ]
end
```

## 5.4. Complete code of Extension II in the Code tab

Well done! We have now finished the second extension!

```
extensions [rnd]

globals [
  payoff-matrix
  n-of-strategies
  n-of-players

  strategy-numbers
  strategy-frequencies
  strategy-expected-payoffs

  min-of-payoff-matrix   ;; <= new global variable
  max-of-payoff-matrix   ;; <= new global variable
  max-payoff-difference  ;; <= new global variable
]

breed [players player]

players-own [
  strategy
  strategy-after-revision
  payoff
]

;;;;;;;;;;;;;;
;;; SETUP ;;;
;;;;;;;;;;;;;;

to setup
  clear-all
  no-display
  setup-payoffs
  setup-players
  setup-graph
  reset-ticks
  update-strategy-expected-payoffs
  update-graph
end

to setup-payoffs
  set payoff-matrix read-from-string payoffs
  set n-of-strategies length payoff-matrix
  set strategy-numbers range n-of-strategies

  ;; new lines below
  set min-of-payoff-matrix min reduce sentence payoff-matrix
  set max-of-payoff-matrix max reduce sentence payoff-matrix
  set max-payoff-difference (max-of-payoff-matrix - min-of-payoff-matrix)
```

```
    if decision-rule = "direct-positive-proportional-m"
      [ check-payoffs-are-non-negative ]
end

to setup-players
  let initial-distribution read-from-string n-of-players-for-each-strategy

  if length initial-distribution != length payoff-matrix [
    user-message (word "The number of items in\n"
      "n-of-players-for-each-strategy (i.e. "
      length initial-distribution "):\n"
      n-of-players-for-each-strategy
      "\nshould be equal to the number of rows\n"
      "in the payoff matrix (i.e. "
      length payoff-matrix "):\n"
      payoffs
    )
  ]

  let i 0
  foreach initial-distribution [ j ->
    create-players j [
      set payoff 0
      set strategy i
      set strategy-after-revision strategy
    ]
    set i (i + 1)
  ]

  set n-of-players count players
end

to setup-graph
  set-current-plot "Strategy Distribution"
  foreach (range n-of-strategies) [ i ->
    create-temporary-plot-pen (word i)
    set-plot-pen-mode 1
    set-plot-pen-color 25 + 40 * i
  ]
end

;;;;;;;;;;
;;; GO ;;;
;;;;;;;;;;

to go
  ask players [update-payoff]

  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
```

```
  ask players [update-strategy]

  tick
  update-strategy-expected-payoffs
  update-graph
end


;;;;;;;;;;;;;;;;;;;;;;;
;;; UPDATE PAYOFFS ;;;
;;;;;;;;;;;;;;;;;;;;;;;

to update-strategy-expected-payoffs
  set strategy-frequencies map [ n ->
      count players with [strategy = n] / n-of-players
    ] strategy-numbers
  set strategy-expected-payoffs map [ list-of-payoffs ->
      sum (map * list-of-payoffs strategy-frequencies) ] payoff-matrix
end

to update-payoff
  run payoff-to-use
end

to play-with-one-rd-agent
  let mate one-of other players
  set payoff item ([strategy] of mate) (item strategy payoff-matrix)
end

to use-strategy-expected-payoff
  set payoff item strategy strategy-expected-payoffs
end


;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; UPDATE STRATEGIES ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;

to update-strategy-after-revision
  ifelse random-float 1 < noise
    [ set strategy-after-revision (random n-of-strategies) ]
    [ run (word decision-rule "-rule") ]
end

to update-strategy
  set strategy strategy-after-revision
end


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; imitative decision rules ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to imitate-if-better-rule
  let observed-player one-of other players
```

```
    if ([payoff] of observed-player) > payoff [
      set strategy-after-revision ([strategy] of observed-player)
    ]
end

to imitative-pairwise-difference-rule
  let observed-player one-of other players

  ;; compute difference in payoffs
  let payoff-diff ([payoff] of observed-player - payoff)

  set strategy-after-revision ifelse-value
    (random-float 1 < (payoff-diff / max-payoff-difference))
      [ [strategy] of observed-player ]
      [ strategy ]
    ;; If your strategy is the better, payoff-diff is negative,
    ;; so you are going to stick with it.
    ;; If it's not, you switch with probability
    ;; (payoff-diff / max-payoff-difference)
end

to imitative-linear-attraction-rule
  let observed-player one-of players

  set strategy-after-revision ifelse-value
    (random-float 1 <
          ([payoff] of observed-player - min-of-payoff-matrix) /
          ( max-of-payoff-matrix - min-of-payoff-matrix) )
      [ [strategy] of observed-player ]
      [ strategy ]
end

to imitative-linear-dissatisfaction-rule
  let observed-player one-of players

  set strategy-after-revision ifelse-value
    (random-float 1 <
          (max-of-payoff-matrix - payoff) /
          (max-of-payoff-matrix - min-of-payoff-matrix) )
      [ [strategy] of observed-player ]
      [ strategy ]
end


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; direct decision rules ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to-report payoff-for-strategy [s]
  report ifelse-value payoff-to-use = "play-with-one-rd-agent"
    [item ([strategy] of one-of other players) (item s payoff-matrix)]
    [item s strategy-expected-payoffs]
end
```

```
to direct-best-rule
  let pairs-strategy-payoff (map [
    [s] -> list s (payoff-for-strategy s)
  ] strategy-numbers)

  ;; the following line ensures that ties
  ;; are resolved (uniformly) at random
  set pairs-strategy-payoff shuffle pairs-strategy-payoff

  let sorted-list sort-by [ [l1 l2] -> last l1 > last l2 ]
      pairs-strategy-payoff
  set strategy-after-revision (first (first sorted-list))
end

to direct-pairwise-difference-rule
  let candidate-strategy one-of (remove-item strategy strategy-numbers)

  ;; compute difference in payoffs
  let payoff-diff ((payoff-for-strategy candidate-strategy) - payoff)

  set strategy-after-revision ifelse-value
    (random-float 1 < (payoff-diff / max-payoff-difference))
      [ candidate-strategy ]
      [ strategy ]
    ;; If your strategy is the better, payoff-diff is negative,
    ;; so you are going to stick with it.
    ;; If it's not, you switch with probability
    ;; (payoff-diff / max-payoff-difference)
end

to direct-positive-proportional-m-rule
  let pairs-strategy-payoff (map [
    [s] -> list s ((payoff-for-strategy s) ^ m)
  ] strategy-numbers)

  set strategy-after-revision first
      rnd:weighted-one-of-list pairs-strategy-payoff [ [p] -> last p ]
end

;;;;;;;;;;;;;;;;;;;;;
;;; UPDATE GRAPH ;;;
;;;;;;;;;;;;;;;;;;;;;

to update-graph
  set-current-plot "Strategy Distribution"
  let bar 1
  foreach strategy-numbers [ n ->
    set-current-plot-pen (word n)
    plotxy ticks bar
    set bar (bar - (item n strategy-frequencies))
  ]
  set-plot-y-range 0 1
```

```
   end


   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
   ;;; SUPPORTING PROCEDURES ;;;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;


   to check-payoffs-are-non-negative
     if min-of-payoff-matrix < 0 [
       user-message (word
         "Since you are using decision-rule =\n"
         "imitative-positive-proportional-m,\n"
         "all elements in the payoff matrix\n"
         payoffs
         "\nshould be non-negative numbers.")
     ]
   end
```

# 6. Exercises

You can use the following link to download the complete NetLogo model: nxn-games-in-well-mixed-populations.nlogo.



**Exercise 1.** Implement a decision rule that you can call "*imitate-the-best*". This is the imitate the best neighbor rule adapted to well-mixed populations. Under this rule, the revising agent copies the strategy of the individual with the greatest payoff. Resolve ties uniformly at random.

**Exercise 2.** Implement a decision rule that you can call "*imitative-positive-proportional-m*". This is the imitative-positive-proportional-m rule implemented in chapter III-4, adapted to well-mixed populations. Under this decision rule, the revising agent selects one individual at random, with probabilities proportional to payoffs raised to parameter *m*, and copies her strategy. Please do not allow for negative payoffs when using this rule.

**Exercise 3.** Implement a decision rule that you can call "*imitative-logit-m*". Under this decision rule, the revising agent selects one individual at random, with probabilities proportional to $e^{m\pi_i}$ (where $\pi_i$ denotes agent $i$'s payoff), and copies her strategy.

**Exercise 4.** Implement a decision rule that you can call "*Fermi-m*". This is the Fermi-m rule

implemented in chapter III-4, adapted to well-mixed populations. Under this rule, the revising agent $i$ looks at another individual $j$ at random and copies her strategy with probability $\frac{e^{m\pi_j}}{e^{m\pi_i}+e^{m\pi_j}}$, where $\pi_z$ denotes agent $z$'s payoff and $m \geq 0$.

**Exercise 5.** Implement a decision rule that you can call "*direct-logit-m*". Under this rule, the revising agent selects one strategy at random, with probabilities proportional to $e^{m\pi_i}$ (where $\pi_i$ denotes strategy $i$'s payoff), and adopts it.

**Exercise 6.** In this model, we run procedure to update-strategy-expected-payoffs at the end of to go, as shown below:

```
to go
  ask players [update-payoff]

  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick

  update-strategy-expected-payoffs ;; <= current position
  update-graph
end
```

Since procedure to update-strategy-expected-payoffs updates the expected payoffs, and these are used by agents in procedure to update-payoffs, it may seem more natural to run procedure to update-strategy-expected-payoffs just before asking players to update their payoff, as shown below:

```
to go
  update-strategy-expected-payoffs ;; <= new position
  ask players [update-payoff]

  ask players [
    if (random-float 1 < prob-revision) [
      update-strategy-after-revision
    ]
  ]
  ask players [update-strategy]
  tick

  update-graph
end
```

What would change if we did this?

You may compare the two alternatives in a setting where agents play Rock-Paper-Scissors (as in figure 2), *payoff-to-use* = "*use-strategy-expected-payoff*", *prob-revision* = 1, *noise* = 0, and *decision-rule* = "*direct-best*".

## V-3. Mean Dynamics

# 1. Introduction

Every dynamic that can be run with the NetLogo model we have implemented in the previous chapter (nxn-games-in-well-mixed-populations.nlogo) can be usefully seen as a time-homogeneous Markov chain on the space of possible strategy distributions. This means that the fraction of agents that are using each strategy, i.e., the population state, contains all the information we need to –probabilistically– predict the evolution of the agent-based dynamic as accurately as it is possible.

Let $x_i$ denote the fraction of agents using strategy $i$. In games with $n$ strategies, population states $x = (x_1, ..., x_n)$ are elements of the simplex $\Lambda = \{x \in \mathbb{R}^n_+ \colon \sum_{i=1}^n x_i = 1\}$. More specifically, if there are $N$ agents, population states are elements of the finite grid $\mathbb{X}^N = \{x \in \Lambda \colon Nx_i \in \mathbb{Z} \text{ for all } i\}$.

A fully parameterized model defines a discrete-time Markov chain $\{X_k^N\}$ on state space $\mathbb{X}^N$, where $k$ is the index for the ticks. Our goal in this chapter is to approximate the transient dynamics of this (stochastic) process $\{X_k^N\}$ with its corresponding (deterministic) mean dynamic.

In the next section, we explain how to derive the mean dynamic in general, and in section 3 we present the mean dynamic for every possible parameterization of NetLogo model nxn-games-in-well-mixed-populations.nlogo. Then, in section 4 we show how we can numerically solve the mean dynamic using the Euler method within NetLogo. In section 5, we present some representative simulations together with their corresponding mean dynamics solved at runtime within NetLogo. In this way, we will be able to appreciate how useful the mean dynamic can be. We conclude the chapter (and the book) emphasizing how sensitive agent-based evolutionary dynamics can be to seemingly unimportant details.

# 2. The mean dynamic

The mean dynamic (Benaïm & Weibull, 2003; Sandholm, 2010a, chapter 10; Roth & Sandholm, 2013) is a system of ODEs that provides a good deterministic approximation to the dynamics of the stochastic evolutionary process $\{X_k^N\}$

- over finite time spans,
- when the number of agents is sufficiently large, and
- when the revision probability is sufficiently low.

To derive the mean dynamic, we consider the behavior of the process $\{X_k^N\}$ over the next $dt$ time units, departing from population state $x$. For convenience, we define one unit of clock time as 1/*prob-revision* ticks, i.e., the number of ticks over which every agent is expected to receive exactly

one revision opportunity.[1] Thus, over the time interval $dt$, the number of agents who are expected to receive a revision opportunity is $Ndt$.

Of the $Ndt$ agents who revise their strategies over the time interval $dt$, $x_j Ndt$ are expected to be $j$-strategists. Let $p_{ji}^N(x)$ be the conditional probability that a reviser using strategy $j$ adopts strategy $i$ (so $p_{jj}$ denotes the probability that a reviser using strategy $j$ keeps using strategy $j$ after revision). Using this notation, the expected number of $j$-revisers who adopt strategy $i$ over the time interval $dt$ is $x_j \, p_{ji}^N(x) \, Ndt$. Hence, the expected change in the number of agents that are using strategy $i$ over the time interval $dt$ equals:

- the number of revisers who adopt strategy $i$, i.e. the **inflow**, which is $\sum_j x_j \, p_{ji}^N(x) \, Ndt$,

- minus the number of $i$-revisers, i.e. the **outflow**, which is $x_i \, Ndt$.

Therefore, the expected change in the fraction of agents using strategy $i$, i.e. the expected change in variable $x_i$ at population state $x$, is:

$$dx_i = \tfrac{1}{N} \left( \sum_j x_j \, p_{ji}^N(x) \, Ndt - x_i \, Ndt \right) = \left( \sum_j x_j \, p_{ji}^N(x) - x_i \right) dt$$

Note that conditional probabilities $p_{ji}^N(x)$ may depend on $N$. This does not represent a problem as long as this dependency vanishes as $N$ gets large. In that case, to deal with that dependency, we take the limit of $p_{ji}^N(x)$ as $N$ goes to infinity since, after all, the mean dynamic approximation is only valid for large $N$. Thus, defining

$$p_{ji}^\infty(x) = \lim_{N \to \infty} p_{ji}^N(x)$$

we arrive at the mean dynamic equation for strategy $i$:

$$\tfrac{d}{dt} x_i = \dot{x}_i = \sum_j x_j \, p_{ji}^\infty(x) - x_i \tag{1}$$

An equivalent formulation that is sometimes more convenient to use is the following:

$$\dot{x}_i = \sum_{j \neq i} x_j \, p_{ji}^\infty(x) - x_i \sum_{j \neq i} p_{ij}^\infty(x) \tag{2}$$

The two formulations above only differ in how they treat $i$-revisers who keep strategy $i$. In the first formulation (1), these agents are included both in the inflow and in the outflow, while in the second formulation (2) they are excluded from both flows. In any case, it is clear that all we need to derive the mean dynamic of a model is the conditional probabilities $p_{ji}^\infty(x)$.

Finally, introducing noise in the mean dynamic is not difficult. If *noise* $= \mu$, i.e., if revisers choose a random strategy with probability $\mu$, and with probability $(1 - \mu)$ they adopt strategies with conditional probabilities $p_{ji}^N$, the first formulation of the general mean dynamic (1) would be:

---

1. This choice of time scale is convenient, but any other definition for one unit of clock time would also be fine.

$$\dot{x}_i = (1 - \mu)\left(\sum_j x_j\, p_{ji}^\infty(x) - x_i\right) + \mu\left(\tfrac{1}{n} - x_i\right)$$

And the second formulation would be adapted as follows:

$$\dot{x}_i = (1 - \mu)\left(\sum_{j \neq i} x_j\, p_{ji}^\infty(x) - x_i \sum_{j \neq i} p_{ij}^\infty(x)\right) + \mu\left(\tfrac{1}{n} - x_i\right)$$

# 3. Derivation of the mean dynamic for different stochastic processes

In this section we derive the mean dynamic for every possible parameterization of the NetLogo model we developed in the previous chapter (nxn-games-in-well-mixed-populations.nlogo). In particular, we analyze every possible combination of values for parameters *decision-rule* and *payoff-to-use*. For each combination of these two parameter values, we provide a formula for the mean dynamic that is valid for any game (*payoffs*) and any initial condition (*n-of-players-for-each-strategy*). The impact of *noise* can be easily taken into account as explained above. The mean dynamic will be a good approximation for the transient dynamics of the corresponding agent-based model when *prob-revision* is low and the population size $N$ is large.

## 3.1. Imitate if better

> Description: The revising agent looks at another individual at random and imitates her strategy if and only if the observed agent's payoff is greater than the reviser's payoff.

Let us consider the two possible values of parameter *payoff-to-use*.

> *payoff-to-use* = "*play-with-one-rd-agent*"

The probability that a reviser is a $j$-player with payoff $\pi_{jl}$ is $x_j\, x_l$. The probability that the observed agent is an $i$-player with payoff $\pi_{ik}$ is $x_i\, x_k$.

The probability that a $j$-player with payoff $\pi_{jl}$ observes an $i$-player with payoff $\pi_{ik}$ is $x_j\, x_l\, x_i\, x_k$.

Considering that the revising $j$-player adopts the strategy of the $i$-player if $\pi_{ik} > \pi_{jl}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \sum_{j \neq i} x_j\, p_{ji}^\infty(x) = x_i \sum_{j \neq i} x_j \sum_k x_k \left(\sum_{l:\pi_{ik} > \pi_{jl}} x_l\right)$$

and the total $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^\infty(x) = x_i \sum_{j \neq i} x_j \sum_k x_k \left(\sum_{l:\pi_{ik} < \pi_{jl}} x_l\right)$$

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x) = x_i \sum_{j \neq i} x_j \sum_k x_k \left(\sum_{l:\pi_{ik} > \pi_{jl}} x_l - \sum_{l:\pi_{ik} < \pi_{jl}} x_l\right)$$

This rule has been studied by Izquierdo and Izquierdo (2013) and Loginov (2021). Loginov (2021) calls this rule "imitate-the-better-realization".

*payoff-to-use* = "*use-strategy-expected-payoff*"

The probability that a reviser is a $j$-player (whose payoff is $\pi_j$) who observes an $i$-player (whose payoff is $\pi_i$) is $x_j\,x_i$.

Considering that the revising $j$-player adopts the strategy of the $i$-player iff $\pi_i > \pi_j$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \sum_{j \neq i} x_j\, p_{ji}^\infty(x) = \sum_{j:\pi_j < \pi_i} x_j x_i$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^\infty(x) = x_i \sum_{j:\pi_j > \pi_i} x_j$$

The mean dynamic is then

$$
\begin{aligned}
\dot{x}_i &= Inflow_i(x) - Outflow_i(x) \\
&= \sum_{j:\pi_j < \pi_i} x_j x_i - x_i \sum_{j:\pi_j > \pi_i} x_j \\
&= x_i \left( \sum_{j:\pi_j < \pi_i} x_j - \sum_{j:\pi_j > \pi_i} x_j \right)
\end{aligned}
$$

## 3.2. Imitative pairwise-difference

Description: The revising agent looks at another individual at random and considers imitating her strategy only if her payoff is greater than the reviser's payoff; in that case he switches with probability proportional to the payoff difference. In order to turn the difference in payoffs into a meaningful probability, we divide the payoff difference by the maximum possible payoff ($M$) minus the minimum possible payoff ($m$).

*payoff-to-use* = "*play-with-one-rd-agent*"

The probability that a reviser is a $j$-player with payoff $\pi_{jl}$ is $x_j\,x_l$. The probability that the observed agent is an $i$-player with payoff $\pi_{ik}$ is $x_i\,x_k$. The probability that a $j$-player with payoff $\pi_{jl}$ observes an $i$-player with payoff $\pi_{ik}$ is $x_j\,x_l\,x_i\,x_k$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{[\pi_{ik} - \pi_{jl}]_+}{M-m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is[2]

$$Inflow_i(x) = \sum_{j \neq i} x_j \, p_{ji}^{\infty}(x) = \sum_{j \neq i} x_j x_i \sum_k x_k \sum_l x_l \frac{[\pi_{ik} - \pi_{jl}]_+}{M-m}$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^{\infty}(x) = x_i \sum_{j \neq i} x_j \sum_k x_k \sum_l x_l \frac{[\pi_{jl} - \pi_{ik}]_+}{M-m}$$

The mean dynamic is then

$$\begin{aligned}
\dot{x}_i &= Inflow_i(x) - Outflow_i(x) \\
&= \frac{x_i}{M-m} \sum_{j \neq i} x_j \sum_k x_k \sum_l x_l \left([\pi_{ik} - \pi_{jl}]_+ - [\pi_{jl} - \pi_{ik}]_+\right) \\
&= \frac{x_i}{M-m} \sum_{j \neq i} x_j \sum_k x_k \sum_l x_l (\pi_{ik} - \pi_{jl}) \\
&= \frac{x_i}{M-m} (\pi_i - \pi)
\end{aligned}$$

This ODE is the replicator dynamics with a constant speed factor $\frac{1}{M-m}$. The orbits of the mean dynamic do not change with the speed factor. However, if we are interested in the relationship between the solution of the mean dynamic at a given time $t$ and the stochastic process after a number of ticks, then the speed factor must be considered.

> *payoff-to-use* = "*use-strategy-expected-payoff*"

The probability that a revising $j$-player (with payoff $\pi_j$) observes an $i$-player (with payoff $\pi_i$) is $x_j \, x_i$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{[\pi_i - \pi_j]_+}{M-m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \sum_{j \neq i} x_j \, p_{ji}^{\infty}(x) = \sum_{j \neq i} x_j x_i \frac{[\pi_i - \pi_j]_+}{M-m}$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^{\infty}(x) = x_i \sum_{j \neq i} x_j \frac{[\pi_j - \pi_i]_+}{M-m}$$

The mean dynamic is then

---

2. $[z]_+ = \max(0, z)$.

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \frac{x_i}{M-m} \sum_{j \neq i} x_j \left( [\pi_i - \pi_j]_+ - [\pi_j - \pi_i]_+ \right)$$
$$= \frac{x_i}{M-m} \sum_{j \neq i} x_j (\pi_i - \pi_j)$$
$$= \frac{x_i}{M-m} (\pi_i - \pi)$$

This ODE is the replicator dynamics with constant speed factor $\frac{1}{M-m}$.

## 3.3. Imitative linear attraction

Description: The revising agent selects another individual at random and imitates her strategy with probability equal to the difference between the observed individual's payoff and the minimum possible payoff ($m$), divided by the maximum possible payoff ($M$) minus the minimum possible payoff. (The revising agent's payoff is ignored.)

*payoff-to-use* = "*play-with-one-rd-agent*"

The probability that a revising $j$-player observes an $i$-player with payoff $\pi_{ik}$ is $x_j \, x_i \, x_k$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{\pi_{ik}-m}{M-m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \sum_{j \neq i} x_j \, p_{ji}^{\infty}(x) = \sum_{j \neq i} x_j x_i \sum_k x_k \frac{\pi_{ik}-m}{M-m}$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^{\infty}(x) = x_i \sum_{j \neq i} x_j \sum_k x_k \frac{\pi_{jk}-m}{M-m}$$

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \frac{x_i}{M-m} \sum_j x_j \sum_k x_k (\pi_{ik} - \pi_{jk})$$
$$= \frac{x_i}{M-m} (\pi_i - \pi)$$

This ODE is the replicator dynamics with constant speed factor $\frac{1}{M-m}$.

*payoff-to-use* = "*use-strategy-expected-payoff*"

The probability that a revising $j$-player observes an $i$-player (whose payoff is $\pi_i$) is $x_j \, x_i$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{\pi_i - m}{M - m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \sum_{j \neq i} x_j\, p_{ji}^{\infty}(x) = \sum_{j \neq i} x_j x_i \frac{\pi_i - m}{M - m}$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^{\infty}(x) = x_i \sum_{j \neq i} x_j \frac{\pi_j - m}{M - m}$$

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \frac{x_i}{M - m} \sum_j x_j (\pi_i - \pi_j)$$
$$= \frac{x_i}{M - m} (\pi_i - \pi)$$

This ODE is the replicator dynamics with constant speed factor $\frac{1}{M - m}$.

## 3.4. Imitative linear dissatisfaction

Description: The revising agent selects another agent at random and imitates her strategy with probability equal to the difference between the maximum possible payoff ($M$) and his own payoff, divided by the maximum possible payoff minus the minimum possible payoff ($m$). (The observed agent's payoff is ignored.)

*payoff-to-use* = "*play-with-one-rd-agent*"

The probability that a revising $j$-player with payoff $\pi_{jk}$ observes an $i$-player is $x_j\, x_k\, x_i$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{M - \pi_{jk}}{M - m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \sum_{j \neq i} x_j\, p_{ji}^{\infty}(x) = x_i \sum_{j \neq i} x_j \sum_k x_k \frac{M - \pi_{jk}}{M - m}$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^{\infty}(x) = x_i \sum_{j \neq i} x_j \sum_k x_k \frac{M - \pi_{ik}}{M - m}$$

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \frac{x_i}{M - m} \sum_j x_j \sum_k x_k (\pi_{ik} - \pi_{jk})$$
$$= \frac{x_i}{M - m} (\pi_i - \pi)$$

Again, this ODE is the replicator dynamics with constant speed factor $\frac{1}{M-m}$.

The probability that a revising $j$-player (whose payoff is $\pi_j$) observes an $i$-player is $x_j\, x_i$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{M-\pi_j}{M-m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \sum_{j \neq i} x_j\, p_{ji}^{\infty}(x) = x_i \sum_{j \neq i} x_j \frac{M-\pi_j}{M-m}$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \sum_{j \neq i} p_{ij}^{\infty}(x) = x_i \sum_{j \neq i} x_j \frac{M-\pi_i}{M-m}$$

The mean dynamic is then

$$\begin{aligned}\dot{x}_i &= Inflow_i(x) - Outflow_i(x) \\ &= \frac{x_i}{M-m} \sum_j x_j(\pi_i - \pi_j) \\ &= \frac{x_i}{M-m}(\pi_i - \pi)\end{aligned}$$

Again, this ODE is the replicator dynamics with constant speed factor $\frac{1}{M-m}$.

## 3.5. Direct best

Description: The revising agent selects the strategy with the greatest payoff. Ties are resolved uniformly at random.

Under this process, a revising agents tests each of the $n$ possible strategies once (with each play of each strategy being against a newly drawn opponent) and selects one of the strategies that obtained the greatest experienced payoff in the test. Ties are resolved uniformly at random, with equal probability for each of the maximum-payoff strategies.

Note that which strategy is selected depends on what strategy is sampled (i.e., which strategy is used by the sampled co-player) when testing each of the $n$ strategies, and we will need some notation to refer to the strategy $s_i$ of the agent that is sampled when testing strategy $i$. Consider the following notation:

- Sample list $s$. Considering a sample of $n$ agents (one for each tested strategy), let a sample list $s = (s_1, ..., s_n)$ be a list whose $i$-th component $s_i$ indicates the strategy of the agent that was sampled when testing strategy $i$. Let $S^n$ be the set of all such lists of strategies of length

$n$. For instance, $S^{n=2} = \{(1,1),(1,2),(2,1),(2,2)\}$, with e.g. sample list (2,1) indicating that when testing strategy 1 (first position) the sampled agent (co-player) played strategy 2 (element in first position), and when testing strategy 2 (second position) the sampled agent played strategy 1 (element in position 2).

- $P_x(s)$: probability of obtaining sample list $s$ at state $x$. This probability is $P_x(s) = \prod_{j=1}^{n} x_{(s_j)}$.

- Experienced payoff vector $\pi(s)$. This is the vector of experienced payoffs obtained by testing each of the $n$ strategies, when the strategies of the sampled co-players are those in $s$. The $i$-th component of this payoff vector, $\pi_i(s)$, is the payoff to an agent testing strategy $i$ whose co-player uses strategy $s_i$.

- Best experienced payoff indicator for strategy $i$, $\mathrm{I}_i(\pi(s))$. Given an experienced payoff vector $\pi(s)$, this function returns the value 1 if strategy $i$ obtains the maximum payoff in $\pi(s)$, i.e., if $\pi_i(s) = \max_h \pi_h(s)$, and the value 0 otherwise.

With the previous notation, adding the probabilities of all the events that lead a revising agent to choose strategy $i$, we have

$$Inflow_i(x) = \sum_{s \in S^n} P_x(s) \frac{\mathrm{I}_i(\pi(s))}{\sum_j \mathrm{I}_j(\pi(s))}$$

$Outflow_i(x)$ is given by the fraction of revisers that are $i$-players, i.e., $Outflow_i(x) = x_i$.

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \sum_{s \in S^n} P_x(s) \frac{\mathrm{I}_i(\pi(s))}{\sum_j \mathrm{I}_j(\pi(s))} - x_i$$

This is a *best-experienced-payoff dynamic* analyzed by Osborne and Rubinstein (1998), Sethi (2000, 2021) and Sandholm et al. (2019, 2020).

*payoff-to-use* = "*use-strategy-expected-payoff*"

Under this process, a revising agent adopts one of the strategies with the greatest expected payoff in the population. This is a best-response dynamic. The greatest expected payoff obtained by (one or several) strategies at state $x$ is $M(x) = \max_j \pi_j$. Let $BR(x)$ be the set of stategies that obtain the greatest expected payoff $M(x)$ at state $x$ and let $\#BR(x)$ be the cardinality of that set, i.e., the number of strategies that obtain the maximum expected payoff at state $x$.

The probability that a reviser adopts strategy $i$ is then

$$Inflow_i(x) = \begin{cases} \frac{1}{\#BR(x)} & i \in BR(x) \\ 0 & \text{otherwise} \end{cases}$$

$Outflow_i(x)$ is given by the fraction of revisers that are $i$-players, i.e., $Outflow_i(x) = x_i$. Thus, the mean dynamic is

$$\dot{x}_i = \begin{cases} \frac{1}{\#BR(x)} - x_i & i \in BR(x) \\ -x_i & \text{otherwise} \end{cases}$$

Gilboa and Matsui (1991), Hofbauer (1995b) and Kandori and Rob (1995) are pioneering references on best-response dynamics.

## 3.6. Direct pairwise-difference

Description: The revising agent randomly selects another strategy, and considers adopting it only if the payoff associated with the selected strategy is greater than the agent's current strategy's payoff; in that case, he switches with probability proportional to the payoff difference. In order to turn the difference in payoffs into a meaningful probability, we divide the payoff difference by the maximum possible payoff $M$ minus the minimum possible payoff $m$.

*payoff-to-use* = *"play-with-one-rd-agent"*

The probability that the revising agent is a $j$-player with payoff $\pi_{jk}$ and he selects candidate strategy $i$, with a payoff $\pi_{il}$ obtained for strategy $i$, is $x_j \, x_k \, \frac{1}{n-1} \, x_l$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{[\pi_{il} - \pi_{jk}]_+}{M-m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \frac{1}{(n-1)(M-m)} \sum_{j \neq i, k, l} x_j x_k x_l [\pi_{il} - \pi_{jk}]_+$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \frac{1}{(n-1)(M-m)} \sum_{j \neq i, k, l} x_k x_l [\pi_{jk} - \pi_{il}]_+$$

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$

$$= \frac{1}{(n-1)(M-m)} \sum_{j \neq i} \left( x_j \sum_{k,l} x_k \, x_l [\pi_{il} - \pi_{jk}]_+ - x_i \sum_{k,l} x_k \, x_l [\pi_{jk} - \pi_{il}]_+ \right)$$

*payoff-to-use* = *"use-strategy-expected-payoff"*

The probability that the revising agent is a $j$-player (with payoff $\pi_j$) and he selects candidate strategy $i$ (with payoff $\pi_i$) is $x_j \frac{1}{n-1}$.

Considering that the revising $j$-player adopts the strategy of the $i$-player with probability $\frac{[\pi_i - \pi_j]_+}{M - m}$, the total $Inflow_i$ of $j$-players who switch to strategy $i$ (adding for every $j \neq i$) is

$$Inflow_i(x) = \frac{1}{(n-1)(M-m)} \sum_{j \neq i} x_j [\pi_i - \pi_j]_+$$

and the $Outflow_i$ of $i$-strategists who switch to some other strategy $j \neq i$ is

$$Outflow_i(x) = x_i \frac{1}{(n-1)(M-m)} \sum_{j \neq i} [\pi_j - \pi_i]_+$$

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \frac{1}{(n-1)(M-m)} \sum_j \left( x_j [\pi_i - \pi_j]_+ - x_i [\pi_j - \pi_i]_+ \right)$$

This is the Smith dynamic (Smith, 1984; Sandholm, 2010b) with constant speed factor $\frac{1}{(n-1)(M-m)}$.

## 3.7. Direct positive proportional

Description: The revising agent selects one strategy at random, with probabilities proportional to payoffs raised to parameter *m*, and adopts this strategy. Payoffs are assumed to be non-negative.

*payoff-to-use* = "*play-with-one-rd-agent*"

Under this process, using the notation introduced for decision rule "*direct-best*", the probability that a reviser selects strategy $i$ is

$$Inflow_i(x) = \sum_{s \in S^n} P_x(s) \frac{\pi_{is_i}^m}{\sum_j \pi_{js_j}^m}$$

$Outflow_i(x)$ is given by the fraction of revisers that are $i$-players, i.e., $Outflow_i(x) = x_i$.

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \sum_{s \in S^n} P_x(s) \frac{\pi_{is_i}^m}{\sum_j \pi_{js_j}^m} - x_i$$

*payoff-to-use* = "*use-strategy-expected-payoff*"

Under this process, the probability that a reviser selects strategy $i$ is

$$Inflow_i(x) = \frac{\pi_i^m}{\sum_j \pi_j^m}$$

$Outflow_i(x)$ is given by the fraction of revisers that are $i$-players, i.e., $Outflow_i(x) = x_i$.

The mean dynamic is then

$$\dot{x}_i = Inflow_i(x) - Outflow_i(x)$$
$$= \frac{\pi_i^m}{\sum_j \pi_j^m} - x_i$$

# 4. Running an agent-based model and solving its mean dynamic at runtime

In this section, we explain the Euler method to solve ODEs and we show how to implement it in NetLogo. For illustration purposes we use the 1-2 coordination game, which is the 2-player 2-strategy single-optimum coordination game that we introduced in chapter I-2 and analyzed in Part IV. Its payoff matrix is shown in Figure 1 below:

|  | | Player 2 | |
| --- | --- | --- | --- |
|  | | Player 2 chooses **A** | Player 2 chooses **B** |
| Player 1 | Player 1 chooses **A** | **1 , 1** | **0 , 0** |
|  | Player 1 chooses **B** | **0 , 0** | **2 , 2** |

*Figure 1. Payoffs for the 1–2 coordination game*

## 4.1. The Euler method to numerically solve ODEs

Since there are only two strategies in the 1-2 coordination game, the mean dynamic can be expressed using one single variable. Let $x$ denote the fraction of B-strategists.

To solve the mean dynamic, we use the Euler method, which is one of the simplest numerical procedures for solving ODEs with an initial condition. Consider the ODE $\dot{x} = f(x)$ with initial condition $x(t = 0)$. Our goal is to compute a series of values $\{x_k\}_{k=0,1,\ldots,n}$ which will approximate the solution trajectory $x(t)$ that departs from initial value $x(t = 0)$. Note that in this section $x_k$ denotes a value in the series $\{x_k\}_{k=0,1,\ldots,n}$ (not the fraction of $k$-strategists).

We start by setting $x_0 = x(t = 0)$, i.e. the first point in our series is the initial condition. To compute the following values $\{x_k\}_{k=1,\ldots,n}$ we proceed recursively as follows:

$$x_{k+1} = x_k + h\, f(x_k)$$

where $h$ is a step size chosen by us.

Geometrically, note that $f(x_k)$ can be interpreted as the slope of the tangent line to the solution curve $x(t)$ at $x = x_k$. So basically, departing from $x_k$, we are approximating the value of the solution

trajectory after $h$ time units using the tangent line at $x_k$, which is the best linear approximation to the curve $x(t)$ at that point (see Figure 2 below).
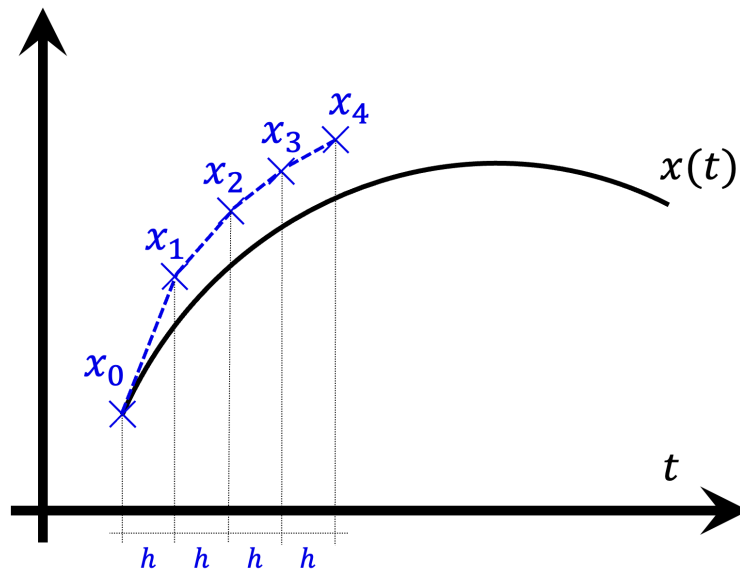


*Figure 2. Sketch of the Euler method*

In this way, by using the Euler method, we are approximating the solution curve $x(t)$ with initial value $x(t = 0)$ by the series of line segments that connect the consecutive points $\{x_k\}_{k=0,1,...,n}$. The value $x_i$ is the approximation to the solution curve $x(t)$ with initial value $x(t = 0)$ after $i\,h$ time units.

How should we choose the value of step size $h$? In general, the lower the value of $h$, the better the approximation. Here, since we are going to run a discrete-time simulation and solve its mean dynamic at the same time, a natural candidate is to choose the time that corresponds to one tick in the NetLogo model. And how long is that? Recall that, in section 2 above, we derived the mean dynamic defining one unit of clock time as 1/*prob-revision* ticks. Therefore, one tick corresponds to *prob-revision* time units, so we choose $h =$*prob-revision*.

## 4.2. Solving an ODE numerically within NetLogo

As an example, consider the agent-based dynamic with *decision-rule* = "*direct-positive-proportional-m*", *m* = 1, and *payoff-to-use* = "*play-with-one-rd-agent*". The mean dynamic for this setting in the 1-2 coordination game (Fig. 1) is:

$$\dot{x} = x\,(1 - x)/6$$

where $x$ is the number of B-strategists.

Therefore, using the Euler method we compute the value $x_{k+1}$ from $x_k$ as follows:

$$x_{k+1} = x_k + h\,x_k\,(1 - x_k)/6$$

In NetLogo, we can define a global variable x and we can update its value every tick as follows:

```
set x x + prob-revision * x * (1 - x) / 6
```

For simplicity, here we have discussed a model where the mean dynamic has one single independent variable, but the Euler method can be used (and implemented within NetLogo) with any number of variables.[3]

## 4.3. The influence of population size

Figure 3 below shows some representative simulations of the parameterization considered in the previous section for different population sizes. The numerical solution of the mean dynamic $(\dot{x} = x\,(1-x)/6)$ is shown as a black line in every plot. All these graphs have been created with NetLogo model nxn-games-in-well-mixed-populations-md.nlogo, which numerically solves the mean dynamic for any setting in the 1-2 coordination game, at runtime.
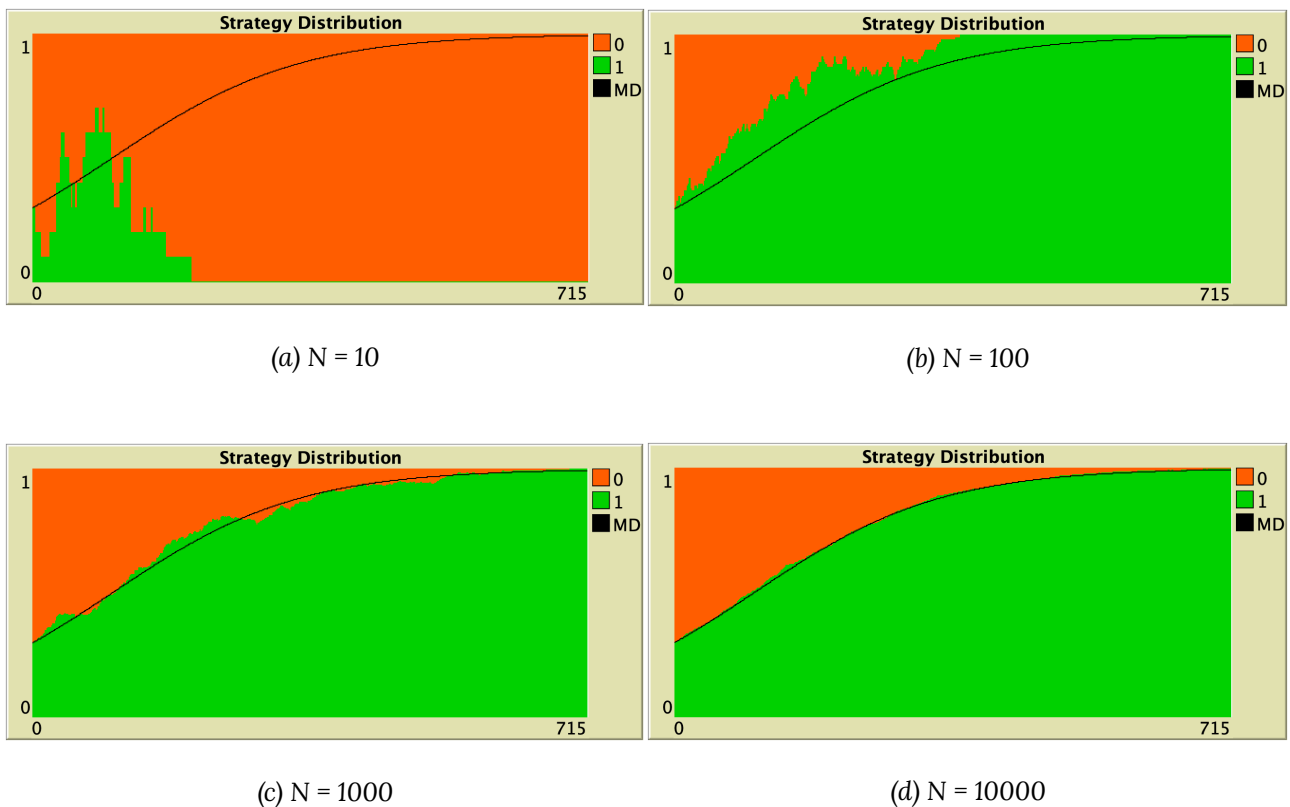


*(a)* N = 10

*(b)* N = 100

*(c)* N = 1000

*(d)* N = 10000

*Figure 3. Representative simulation runs executed with NetLogo model nxn-games-in-well-mixed-populations-md.nlogo, for different population sizes $N$. The black line shows the numerical solution of the mean dynamic, obtained using the Euler method. Settings are: decision-rule = "direct-positive-proportional-m", m = 1, payoff-to-use = "play-with-one-rd-agent", noise = 0 and prob-revision = 0.05. The game is the 1-2 coordination game. Initially, 70% of the agents are A-strategists (orange) and 30% are B-strategists (green)*
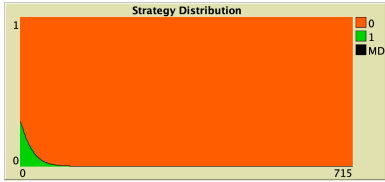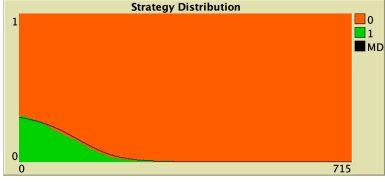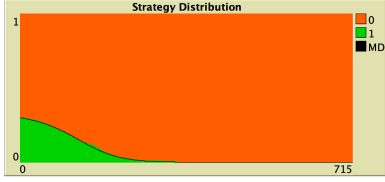
As you can see in Figure 3, the greater the population size $N$, the better the mean dynamic

---

3. Izquierdo et al. (2014) use the Euler method to numerically solve a mean dynamic with 35 independent variables within NetLogo.

approximates the stochastic dynamics of our agent-based model. As $N$ grows, the stochastic fluctuations of the agent-based model (in strategy proportions) diminish, and simulations tend to get closer and closer to the –deterministic– solution of the mean dynamic.

# 5. Representative simulations together with their mean dynamics

In this section we explore the influence of parameters *decision-rule* and *payoff-to-use* in our model. Table 1 below shows representative simulations for different values of these parameters in the 1-2 coordination game, including the numerical solution of the corresponding mean dynamic as a black line. All these graphs can be replicated using NetLogo model nxn-games-in-well-mixed-populations-md.nlogo. The equation displayed under each plot is the corresponding mean dynamic (where $x$ denotes the fraction of B-strategists, which is shown in green in the plots).

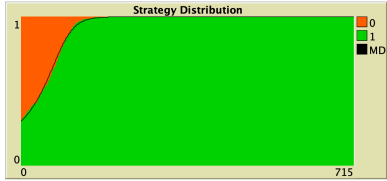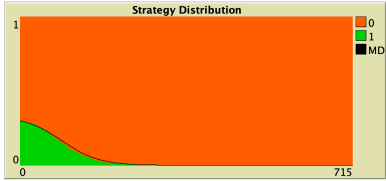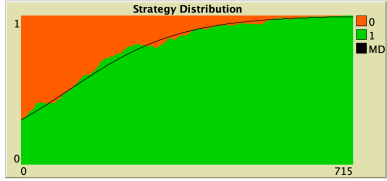| decision-rule | payoff-to-use | |
| --- | --- | --- |
| | *play-with-one-rd-agent* | *use-strategy-expected-payoff* |
| **Imitate if better** |  $\dot{x} = x\,(x-1)\,(x^2 - 3x + 1)$ |  $\dot{x} = x\,(1-x)\begin{cases} -1 & x < \frac{1}{3} \\ 0 & x = \frac{1}{3} \\ 1 & x > \frac{1}{3} \end{cases}$ |
| **Imitative pairwise-difference** |  $\dot{x} = x\,(4x - 3x^2 - 1)/2$ |  $\dot{x} = x\,(4x - 3x^2 - 1)/2$ |
| **Direct best** |  $\dot{x} = x(1-x)/2$ |  $\dot{x} = \begin{cases} -x & x < \frac{1}{3} \\ \frac{1}{2} - x & x = \frac{1}{3} \\ 1 - x & x > \frac{1}{3} \end{cases}$ |

| decision-rule | payoff-to-use | |
| --- | --- | --- |
| | *play-with-one-rd-agent* | *use-strategy-expected-payoff* |
| **Direct pairwise-difference** |  $$\dot{x} = (1 - x)\,x^2$$ |  $$\dot{x} = \tfrac{1}{2}\,(3x - 1)\begin{cases} x & x < \tfrac{1}{3} \\ 1 - x & x \geq \tfrac{1}{3} \end{cases}$$ |
| **Direct positive proportional** *m* = 1 |  $$\dot{x} = \frac{x\,(1-x)\,(2^m - 1)}{2\,(2^m + 1)}$$ |  $$\dot{x} = (1 - x) - \frac{(1-x)^m}{(1-x)^m + 2^m\,x^m}$$ |
| **Direct positive proportional** *m* = 10 |  $$\dot{x} = \frac{x\,(1-x)\,(2^m - 1)}{2\,(2^m + 1)}$$ |  $$\dot{x} = (1 - x) - \frac{(1-x)^m}{(1-x)^m + 2^m\,x^m}$$ |
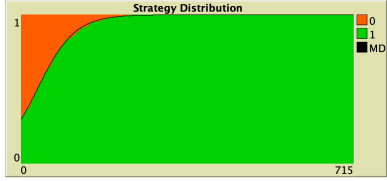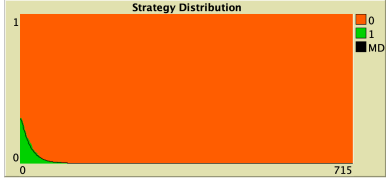
*Table 1. Representative simulation runs executed with NetLogo model nxn-games-in-well-mixed-populations-md.nlogo, for different decision rules and different ways of computing payoffs. Initial conditions are [700 300], i.e. 700 A-strategists (in orange) and 300 B-strategist (in green), prob-revision = 0.05 and there is no noise. The equation of the mean dynamic (where $x$ denotes the fraction of B-strategists) is included under each plot. The black line shows the numerical solution of the mean dynamic, obtained using the Euler method*

The results shown in Table 1 allow us to draw a number of observations. The most obvious one is that the mean dynamic provides a very good approximation for the dynamics of all parameterizations shown in the table. The population size in all simulations shown in Table 1 is $N = 1000$. The approximation would become even better for greater population sizes (and would degrade for lower population sizes).

Looking at Table 1, it is also clear that we can make the population go to one monomorphic state or the other simply by changing the value of parameter *decision-rule* and, in some cases, the value of *payoff-to-use*. These two parameters affect the dynamics in ways that do not seem easy to understand without the help of the mean dynamic.

Finally, in terms of mean dynamics, note that only one of the rows in Table 1 corresponds to the

replicator dynamics (i.e., the row for *decision-rule* = "*imitative-pairwise-difference*").[4] The other decision rules in Table 1 have mean dynamics that are different from the replicator dynamics; and, in many cases, these other dynamics lead to the opposite end of the state space. For this reason, it seems clear that we should not make general statements about evolutionary dynamics if such statements are based only on the behavior of the replicator dynamics (or any other particular model, for that matter). There are many other evolutionary dynamics that lead to results completely different from those obtained with the replicator dynamics.

To be clear, in our view, all the decision rules in Table 1 could be defended as sensible models of evolution in the sense that, under all of them, agents preferentially switch to strategies that provide greater payoffs. Nonetheless, they lead to completely different dynamics. It is true that the replicator dynamics appears as the mean dynamic of many different agent-based models (see chapter V-1), but it is also true that there are a myriad of other sensible models of evolution with properties completely different from those of the replicator dynamics.

# 6. Details matter

In previous chapters of this book, we learned that evolutionary dynamics on grids (and, more generally, on networks) can be significantly affected by small, seemingly unimportant details. In this final chapter, we have seen that this sensitivity to apparently innocuous details can also be present in the simpler setting of well-mixed populations. In general, evolutionary game dynamics in well-mixed populations can also depend on factors that may seem insignificant at first sight. Because of this, if we had to summarize this book in one sentence, this sentence could well be: "Details matter"

This sensitivity to small details suggests that a simple general theory on evolutionary game dynamics cannot be derived. It also highlights the usefulness of the skills you have acquired with this book. We hope that you have learned new ways to explore and tame the complexity and the beauty of evolutionary game dynamics, using both computer simulation and mathematical analysis.

---

4. The replicator dynamics is also obtained if *decision-rule* = "*imitative-linear-attraction*" and if *decision-rule* = "*imitative-linear-dissatisfaction*".

# 7. Exercises



**Exercise 1.** Derive the mean dynamic for a decision rule that you can call "*imitate-the-best*" using expected payoffs (Hofbauer, 1995a; Vega-Redondo, 1997). Under this rule, the revising agent copies the strategy of the individual with the greatest expected payoff. Resolve ties uniformly at random.

**Exercise 2.** Derive the mean dynamic for a decision rule that you can call "*imitative-positive-proportional*" assuming agents obtain their payoff by playing with one random agent, i.e., *payoff-to-use* = "*play-with-one-rd-agent*". Under this decision rule, the revising agent selects one individual at random, with probabilities proportional to payoffs, and copies her strategy. Negative payoffs are not allowed when using this rule.

**Exercise 3.** Derive the mean dynamic for a decision rule that you can call "*imitative-logit-m*" using expected payoffs (Weibull, 1995, example 4.5). Under this decision rule, the revising agent selects one individual at random, with probabilities proportional to $e^{m\pi_i}$ (where $\pi_i$ denotes agent $i$'s expected payoff), and copies her strategy.

**Exercise 4.** Derive the mean dynamic for a decision rule that you can call "*Fermi-m*", assuming agents obtain their payoff by playing with one random agent, i.e., *payoff-to-use* = "*play-with-one-rd-agent*" (Woelfing and Traulsen, 2009). Under this rule, the revising agent $i$ looks at another individual $j$ at random and copies her strategy with probability $\frac{e^{m\pi_j}}{e^{m\pi_i}+e^{m\pi_j}}$, where $\pi_z$ denotes agent $z$'s payoff and $m \geq 0$.

**Exercise 5.** Derive the mean dynamic for a decision rule that you can call "*direct-logit-m*" using expected payoffs (Blume, 1997; Fudenberg and Levine, 1998; Hofbauer and Sandholm, 2007). Under this rule, the revising agent selects one strategy at random, with probabilities proportional to $e^{m\pi_i}$ (where $\pi_i$ denotes strategy $i$'s expected payoff), and adopts it.

**Exercise 6.** Derive the mean dynamic for a decision rule where the revising agent looks at a random agent in the population and adopts her strategy.

# APPENDICES

# 1. Introduction

In this appendix we will learn that there are different ways of implementing any particular model, and we will discuss how to choose the most appropriate implementation for a given purpose. To this end, we consider three different algorithms agents may use as a decision rule which lead to the same (probabilistic) output, i.e., they lead to the same conditional probabilities $p_{ij}$ that a reviser using strategy $i$ switches to strategy $j$. Therefore, these three algorithms lead to exactly the same dynamics. Nonetheless, the three algorithms differ in efficiency, code readability, and on how realistic the story behind them may sound.

# 2. Three possible stories behind the same behavior

In this section we present three different algorithms agents may use as a decision rule which lead to the same conditional probabilities $p_{ij} = \frac{x_j \pi_j}{\sum_k x_k \pi_k}$, where $x_j$ is the fraction of $j$-strategists and $\pi_j$ is the average payoff of $j$-strategists.

## 2.1. Vose's alias method

The following decision rule uses NetLogo primitive `rnd:weighted-one-of`, which returns one agent selected stochastically. The probability of each agent being picked is proportional to her payoff. Once the agent is selected, the reviser copies her strategy.

```
to Vose-alias-imitate-positive-proportional-rule
  ;; assumes that all payoffs are non-negative
  let chosen-player rnd:weighted-one-of players [payoff]
  set next-strategy [strategy] of chosen-player
end
```

Primitive `rnd:weighted-one-of` in NetLogo 6.4.0 uses Keith Schwarz's (2011) implementation of Vose's (1991) alias method. This algorithm is very efficient (especially when several draws are required), but we would find it hard to sell as an actual procedure that human agents may use to select a strategy.

## 2.2. Roulette wheel

The following decision rule implements the so-called "roulette wheel selection algorithm" to select one agent with probability proportional to payoffs. Let $N$ be the number of agents and let $\gamma_i$ be player $i$'s payoff. The algorithm starts by assigning one line segment of length $\gamma_i$ to each agent $i$.

These segments are then placed consecutively to build a larger segment of length equal to the sum of all players' payoffs $\sum_{i=1}^{N} \gamma_i$ (Figure 1).
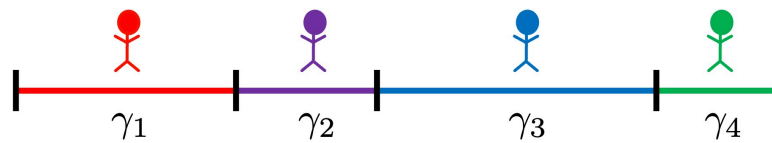


*Figure 1. Sketch of the roulette wheel selection algorithm for a population of four agents*

We then obtain a random number between 0 and $\sum_{i=1}^{N} \gamma_i$ (i.e. a random location in the big segment), and we select the agent who owns that part of the big segment. Once the agent is selected, the reviser copies her strategy.[1] The following decision rule is an inefficient but quite readable implementation of this algorithm in NetLogo.

```
to roulette-wheel-imitate-positive-proportional-rule
  ;; assumes that all payoffs are non-negative

  ;; returns the strategy of one agent selected
  ;; with probability proportional to payoffs,
  ;; using an inneficient implementation of
  ;; the roulette wheel algorithm

  let list-of-agents sort players
  let list-of-payoffs map [a -> [payoff] of a] list-of-agents

  if (sum list-of-payoffs = 0) [
    ;; applies when all players have payoff = 0
    set list-of-payoffs n-values n-of-players [1]
  ]

  let cumulative-payoffs [0]
  foreach list-of-payoffs [p ->
    set cumulative-payoffs lput (p + last cumulative-payoffs)
                                cumulative-payoffs
  ]
    ;; cumulative-payoffs is (n-of-players + 1) items long,
    ;; its first value is 0 and its last value is the sum of all payoffs.
    ;; You can also use reduce instead of foreach for a
    ;; more efficient but less readable implementation

  let rd-number random-float (last cumulative-payoffs)
    ;; take a random number between 0 and
    ;; the sum of all payoffs

  let index length filter [p -> rd-number >= p]
                (butfirst cumulative-payoffs)
```

---

1. It is also possible to implement this algorithm using one segment for each strategy, rather than one segment for each agent.

```
    ;; see in which player's segment the random number is located.
    ;; The efficiency of this process can be greatly improved
    ;; by using e.g. binary search

  let chosen-player (item index list-of-agents)
  set next-strategy [strategy] of chosen-player
end
```

## 2.3. Repeated sampling

In this section we provide an implementation of an algorithm that Sandholm (2010a, example 5.4.5) calls "repeated sampling".[2] The algorithm works as follows:

1. The reviser sets a random aspiration level $\gamma^*$ in the interval $[0, M]$, where $M = \max_i \gamma_i$ is the maximum payoff in the population of agents.[3]

2. The reviser selects one agent at random. Let $o$ denote this selected agent and let $\gamma_o$ be her payoff.

3. If $\gamma_o \geq \gamma^*$, i.e., if the observed agent's payoff is greater than or equal to the reviser's aspiration, then the reviser copies the observed agent's strategy. Otherwise the reviser goes back to step 1.

Below you can find an implementation of this algorithm in Netlogo.

```
to repeated-sampling-imitate-positive-proportional-rule
  ;; assumes that all payoffs are non-negative
  loop [
    let my-random-aspiration random-float max-payoff
    let one-at-random one-of players
    if [payoff] of one-at-random >= my-random-aspiration [
      set next-strategy [strategy] of one-at-random
      stop
    ]
  ]
end
```

Note that, in contrast with the two previous algorithms, in this case the reviser will not generally use the payoffs of every player. The algorithm does require that the reviser *has access* to all the payoffs and, in the worst-case scenario, the algorithm will use them; but, in general, the reviser will most likely use just a few of those payoffs.

---

2. See also Lipowski and Lipowska (2012).

3. $M$ can be any number greater than or equal to $\max_i \gamma_i$, but the algorithm runs fastest if $M = \max_i \gamma_i$. Thus, there is no problem in using the maximum possible payoff that an agent can obtain, even if no agent has actually obtained that payoff.

# 3. Which algorithm should we use?

The three decision rules above provide exactly the same conditional probabilities $p_{ij} = \frac{x_j \pi_j}{\sum_k x_k \pi_k}$. This means that if we only observed the actions of the agents, it would be impossible to tell which of the three algorithms the agents are using. The three decision rules lead to exactly the same dynamics. For this reason, we could say that they are:

- three different micro-foundations for the same model,

- three different decision rules that give the same output, or

- in the context of population games, three different implementations of the same revision protocol (Sandholm, 2010a).

Personally, if we had to motivate the decision rule, we would probably use the story behind repeated sampling. This decision rule seems the most natural of the three as a decision-making algorithm and, in general, it requires less information than the other two.

If we wanted to prioritize code readability, we would choose NetLogo primitive `rnd:weighted-one-of`, as we did in the implementation of procedure to imitative-positive-proportional-m-rule in chapters III-4 and IV-4.

And finally, to run several simulations, knowing that the three decision rules lead to exactly the same dynamics, we would choose the most efficient one. This would imply ruling out our naive implementation of the "roulette wheel selection algorithm". The relative efficiency of the other two decision rules depends on various factors such as the number of agents, the distribution of payoffs, and the number of times we want to execute the algorithm for a fixed distribution of payoffs.[4] In our models, it turns out that repeated sampling is often the most efficient decision rule too. For details on the efficiency and the intuition behind these three algorithms (and some others), see Keith Schwarz's (2011) excellent web page "Darts, Dice, and Coins: Sampling from a Discrete Distribution".

---

4. For many draws, we should use NetLogo primitive `rnd:weighted-n-of`.

# A-2. Legend for code skeletons

This book includes many code skeletons that provide a schematic bird's-eye view of the code. You can find the first code skeleton in chapter II-1. The legend for all the code skeletons is shown in Figure 1 below.



*Figure 1. Legend for code skeletons*
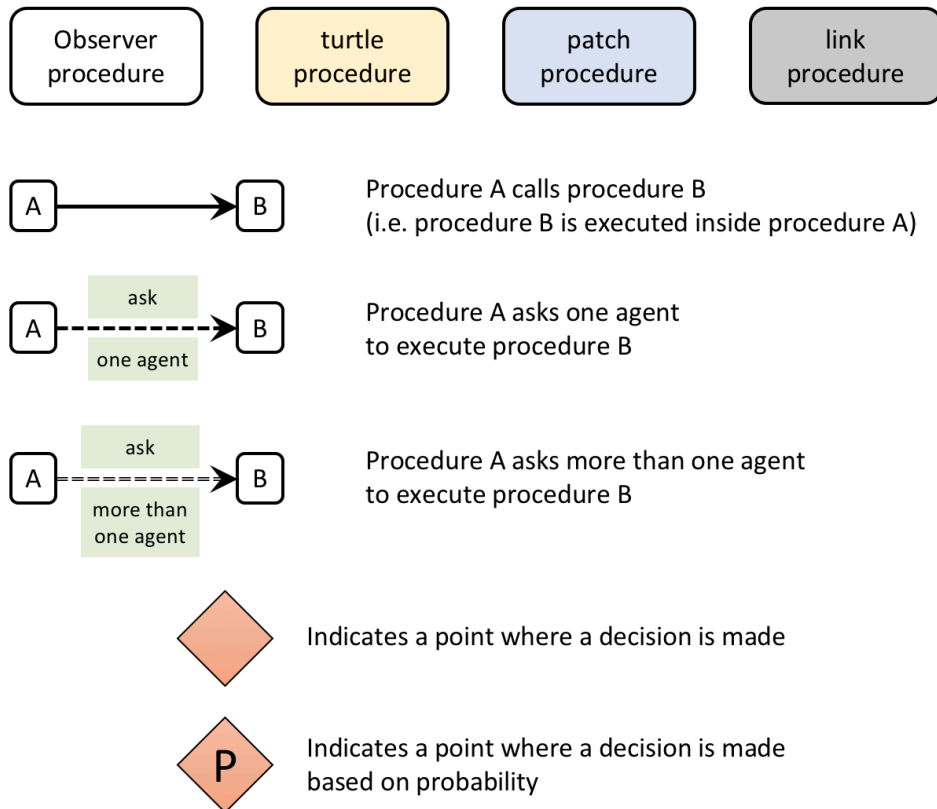
In some skeletons, we use dashed rectangles to highlight certain parts of the code. In those cases:

- A green dashed rectangle indicates that the code has been added. See Figure 2 in chapter II-4.
- A red dashed rectangle indicates that the code has been removed. See Figure 3 in chapter II-4.
- A blue dashed rectangle indicates that the code has been modified. See Figure 4 in chapter II-4.

# A-3. Models implemented in this book

## In Part I. Introduction

### I-3. Introduction to agent-based modeling
- schelling-sakoda.nlogo

### I-5. The fundamentals of NetLogo
- schelling-sakoda-simple.nlogo

## In Part II. Our first agent-based evolutionary model

### II-1. Our very first model
- 2×2-imitate-if-better.nlogo

### II-2. Extension to any number of strategies
- nxn-imitate-if-better.nlogo

### II-3. Noise and initial conditions
- nxn-imitate-if-better-noise.nlogo

### II-4. Interactivity and efficiency
- nxn-imitate-if-better-noise-interactive-profiler.nlogo
- nxn-imitate-if-better-noise-efficient-but-more-than-once-profiler.nlogo
- nxn-imitate-if-better-noise-efficient-played-profiler.nlogo
- nxn-imitate-if-better-noise-efficient-tick-I-played-last-profiler.nlogo
- nxn-imitate-if-better-noise-efficient-tick-I-played-last-and-other-players-profiler.nlogo
- nxn-imitate-if-better-noise-efficient.nlogo (after exercise 5)

## In Part III. Spatial interactions on a grid

### III-1. Spatial chaos in the Prisoner's Dilemma
- 2×2-imitate-best-nbr.nlogo

### III-2. Robustness and fragility
- 2×2-imitate-best-nbr-extended.nlogo

### III-3. Extension to any number of strategies

- nxn-imitate-best-nbr.nlogo

### III-4. Other types of neighborhoods and other decision rules

- nxn-imitate-best-nbr-extended.nlogo

# In Part IV. Games on networks

### IV-1. The nxn game on a random network

- nxn-imitate-if-better-rd-nw.nlogo

### IV-2. Different types of network

- nxn-imitate-if-better-networks.nlogo

### IV-3. Implementing network metrics

- nxn-imitate-if-better-networks-metrics.nlogo

### IV-4. Other ways of computing payoffs and other decision rules

- nxn-games-on-networks.nlogo

# In Part V. Agent-based models vs ODE models

### V-2. A rather general model for games played in well-mixed populations

- nxn-imitate-if-better-noise-payoff-to-use.nlogo

- nxn-games-in-well-mixed-populations.nlogo

### V-3. Mean Dynamics

- nxn-games-in-well-mixed-populations-md.nlogo

# References

Abar, S., Theodoropoulos, G. K., Lemarinier, P., and O'Hare, G. M. (2017). Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24 (Supplement C):13–33. https://doi.org/10.1016/j.cosrev.2017.03.001

Adami, C., Schossau, J., and Hintze, A. (2016). Evolutionary game theory using agent-based methods. *Physics of Life Reviews*, 19:1–26. https://doi.org/10.1016/j.plrev.2016.08.015

Alexander, J. M. (2023). *Evolutionary Game Theory*. Elements in Decision Theory and Philosophy. Cambridge University Press. https://doi.org/10.1017/9781108582063

Antonioni, A. and Tomassini, M. (2012). Cooperation on social networks and its robustness. *Advances in Complex Systems*, 15(supp01):1250046. https://doi.org/10.1142/S0219525912500464

Aydinonat, N. E. (2007). Models, conjectures and exploration: an analysis of Schelling's checkerboard model of residential segregation. *Journal of Economic Methodology*, 14(4):429–454. https://doi.org/10.1080/13501780701718680

Bakshy, E. and Wilensky, U. (2007). NetLogo-Mathematica link. *Software*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/mathematica.html

Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512. https://doi.org/10.1126/science.286.5439.509

Benaïm, M. and Weibull, J.W. (2003). Deterministic approximation of stochastic evolution in games. *Econometrica*, 71:873–903. https://doi.org/10.1111/1468-0262.00429

Berto, F. and Tagliabue, J. (2023). Cellular Automata. In Zalta, E. N. and Nodelman, U., editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2023 edition. https://plato.stanford.edu/archives/win2023/entries/cellular-automata/

Bhattacharjee, K., Naskar, N., Roy, S., and Das, S. (2020). A survey of cellular automata: types, dynamics, non-uniformity and applications. *Natural Computing*, 19(2):433–461. https://doi.org/10.1007/s11047-018-9696-8

Biggs, M. B. and Papin, J. A. (2013). Novel multiscale modeling tool applied to pseudomonas aeruginosa biofilm formation. *PLOS ONE*, 8(10). https://doi.org/10.1371/journal.pone.0078011

Binmore, K. (2007). *Playing for Real: A Text on Game Theory*. Oxford University Press.

Binmore, K. (2011). *Rational Decisions*. Princeton University Press.

Binmore, K. (2013). Sexual drift. Biological Theory, 8(2):201–208. https://doi.org/10.1007/s13752-013-0103-5

Binmore, K. and Samuelson, L. (1994). An economist's perspective on the evolution of norms. *Journal of Institutional and Theoretical Economics*, 150(1):45–63. http://www.jstor.org/stable/40753015

Binmore, K., Samuelson, L., and Vaughan, R. (1995). Musical chairs: Modeling noisy evolution. *Games and Economic Behavior*, 11:1–35. Erratum, 21 (1997), 325. https://doi.org/10.1006/game.1995.1039

Binmore, K. and Shaked, A. (2010). Experimental economics: Where next?. *Journal of Economic Behavior and Organization*, 73(1):87–100. https://doi.org/10.1016/j.jebo.2008.10.019

Björnerstedt, J. and Weibull, J.W. (1996). Nash equilibrium and evolution by imitation. In Arrow, K. J. et al., editors, *The Rational Foundations of Economic Behavior*, pages 155–181. St. Martin's Press, New York.

Blume, L. E. (1997). Population games. In Arthur, W. B., Durlauf, S. N., and Lane, D. A., editors, *The Economy as an Evolving Complex System II*, pages 425–460. Addison-Wesley, Reading, MA. https://doi.org/10.1201/9780429496639

Boyd, R. and Richerson, P. J. (1985). *Culture and the Evolutionary Process*. University of Chicago Press.

Chung, K. L. (1960). *Markov Chains with Stationary Transition Probabilities*. Springer Berlin Heidelberg. http://dx.doi.org/10.1007/978-3-642-49686-8

Colman, A. M. (1995). *Game Theory and its Applications in the Social and Biological Sciences*. Routledge, 2nd edition.

Cook, M. (2004). Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40. https://www.complex-systems.com/abstracts/v15_i01_a01/

Cornforth, D., Green, D. G., and Newth, D. (2005). Ordered asynchronous processes in multi-agent systems. *Physica D: Nonlinear Phenomena*, 204(1):70–82. https://doi.org/10.1016/j.physd.2005.04.005

Darwin, C. R. (1859). *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*. John Murray, London.

Dawes, R. M. (1980). Social dilemmas. *Annual Review of Psychology*, 31(1):169–193. https://doi.org/10.1146/annurev.ps.31.020180.001125

Dixit, A. K. and Nalebuff, B. J. (2008). *The Art of Strategy: A Game Theorist's Guide to Success in Business and Life*. W. W. Norton & Company.

Edmonds, B. (2001). The use of models – making MABS more informative. In Moss, S. and Davidsson, P., editors, *Multi-Agent-Based Simulation: Second International Workshop, MABS 2000 Boston, MA, USA*, 15–32. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-44561-7_2

Edmonds, B. and Hales, D. (2005). Computational simulation as theoretical experiment. *The Journal of Mathematical Sociology*, 29(3):209–232. https://doi.org/10.1080/00222500590921283

Ellison, G. (2000). Basins of attraction, long run equilibria, and the speed of step-by-step evolution. *Review of Economic Studies*, 67:17–45. https://doi.org/10.1111/1467-937X.00119

Epstein, J. M. and Axtell, R. (1996). *Growing Artificial Societies*. Brookings Institution Press/MIT Press, Washington/Cambridge.

Erdös, P. and Rényi, A. (1959). On random graphs I. *Publicationes Mathematicae*, 6(3–4):290–297. https://doi.org/10.5486%2FPMD.1959.6.3-4.12

Foster, D. P. and Young, H. P. (1990). Stochastic evolutionary game dynamics. *Theoretical Population Biology*, 38:219–232. Corrigendum, 51 (1997), 77-78. https://doi.org/10.1016/0040-5809(90)90011-J

Fu, F., Nowak, M. A., and Hauert, C. (2010). Invasion and expansion of cooperators in lattice populations: Prisoner's dilemma vs. snowdrift games. *Journal of Theoretical Biology*, 266(3):358–366. https://doi.org/10.1016/j.jtbi.2010.06.042

Fudenberg, D. and Imhof, L. A. (2008). Monotone imitation dynamics in large populations. *Journal of Economic Theory*, 140:229–245. https://doi.org/10.1016/j.jet.2007.08.002

Fudenberg, D. and Levine, D. K. (1998). *The Theory of Learning in Games*. MIT Press, Cambridge.

Fudenberg, D. and Tirole, J. (1991). *Game Theory*. MIT Press, Cambridge.

Galán, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., del Olmo, R., López-Paredes,A., and Edmonds, B. (2009). Errors and artefacts in agent-based modelling. *Journal of Artificial Societies and Social Simulation*, 12(1):1. https://www.jasss.org/12/1/1.html

García, J. and van Veelen, M. (2016). In and out of equilibrium I: Evolution of strategies in repeated games with discounting. *Journal of Economic Theory*, 161:161–189. http://dx.doi.org/10.1016/j.jet.2015.11.007

García, J. and van Veelen, M. (2018). No strategy can win in the repeated prisoner's dilemma: Linking game theory and computer simulations. *Frontiers in Robotics and AI*, 5:102. https://doi.org/10.3389/frobt.2018.00102

Gilbert, N. (2007). *Agent-Based Models*, volume 153 of *Quantitative Applications in the Social Sciences*. Sage Publications, London.

Gilbert, N. and Terna, P. (2000). How to build and use agent-based models in social science. *Mind & Society*, 1(1):57–72. https://doi.org/10.1007/BF02512229

Gilboa, I. and Matsui, A. (1991). Social stability and equilibrium. *Econometrica*, 59:859–867. https://doi.org/10.2307/2938230

Gintis, H. (2009). *Game Theory Evolving Game Theory Evolving: A Problem-Centered Introduction to Modeling Strategic Interaction*. Princeton University Press, 2nd edition.

Gintis, H. (2013). Markov models of social dynamics: Theory and applications. *ACM Trans. Intell. Syst. Technol.*, 4(3), Article 53. http://dx.doi.org/10.1145/2483669.2483686

Gintis, H. (2014). *The Bounds of Reason: Game Theory and the Unification of the Behavioral Sciences*. Princeton University Press, revised edition.

Gómez-Gardeñes, J., Campillo, M., Floría, L. M., and Moreno, Y. (2007). Dynamical organization of cooperation in complex topologies. *Phys. Rev. Lett.*, 98:108103. http://dx.doi.org/10.1103/PhysRevLett.98.108103

Gotts, N., Polhill, J., and Law, A. (2003). Agent-based simulation in the study of social dilemmas. *Artificial Intelligence Review*, 19(1):3–92. https://doi.org/10.1023/A:1022120928602

Hamill, L. and Gilbert, N. (2016). *Agent-based Modelling in Economics*. John Wiley & Sons, Ltd. http://dx.doi.org/10.1002/9781118945520

Hamilton, W. D. (1967). Extraordinary sex ratios. *Science*, 156:477–488. http://dx.doi.org/10.1126/science.156.3774.477

Harsanyi, J. C. (1967). Games with Incomplete Information Played by "Bayesian" Players. Part I. The Basic Model. *Management Science*, 14(3):159–182. https://doi.org/10.1287/mnsc.14.3.159

Harsanyi, J. C. (1968a). Games with Incomplete Information Played by "Bayesian" Players. Part II. Bayesian Equilibrium Points. *Management Science*, 14(5):320-334. https://doi.org/10.1287/mnsc.14.5.320

Harsanyi, J. C. (1968b). Games with Incomplete Information Played by "Bayesian" Players. Part III. The Basic Probability Distribution of the Game. *Management Science*, 14(7):486-502. https://doi.org/10.1287/mnsc.14.7.486

Hauert, C. (2002). Effects of space in 2×2 games. *International Journal of Bifurcation and Chaos*, 12(07):1531–1548. https://doi.org/10.1142/S0218127402005273

Hauert, C. (2006). Spatial effects in social dilemmas. *Journal of Theoretical Biology*, 240(4):627–636. https://doi.org/10.1016/j.jtbi.2005.10.024

Hauert, C. (2018). *EvoLudo: Interactive tutorials in evolutionary games*. https://wiki.evoludo.org/

Hauert, C. and Doebeli, M. (2004). Spatial structure often inhibits the evolution of cooperation in the snowdrift game. *Nature*, 428:643–646. http://dx.doi.org/10.1038/nature02360

Hauert, C. and Miękisz, J. (2018). Effects of sampling interaction partners and competitors in evolutionary games. *Phys. Rev. E*, 98:052301. https://doi.org/10.1103/PhysRevE.98.052301

Hauert, C. and Szabó, G. (2005). Game theory and physics. *American Journal of Physics*, 73(5):405–414. https://doi.org/10.1119/1.1848514

Head, B. (2018). NetLogo Python extension. *Software*. https://github.com/NetLogo/Python-Extension.

Hegselmann, R. (2017). Thomas C. Schelling and James M. Sakoda: The intellectual, technical, and social history of a model. *Journal of Artificial Societies and Social Simulation*, 20(3):15. https://doi.org/10.18564/jasss.3511

Helbing, D. (1992). A mathematical model for behavioral changes by pair interactions. In Haag, G., Mueller, U., and Troitzsch, K. G., editors, *Economic Evolution and Demographic Change: Formal Models in Social Sciences*, pages 330–348. Springer, Berlin. https://doi.org/10.1007/978-3-642-48808-5_18

Hilbe, C. and Traulsen, A. (2016). Only the combination of mathematics and agent-based simulations can leverage the full potential of evolutionary modeling: Comment on "evolutionary game theory using agent-

based methods" by C. Adami, J. Schossau and A. Hintze. *Physics of Life Reviews*, 19:29–31. https://doi.org/10.1016/j.plrev.2016.10.004

Hindersin, L., Wu, B., Traulsen, A., and García, J. (2019). Computation and simulation of evolutionary game dynamics in finite populations. *Scientific Reports*, 9(1):6946. https://doi.org/10.1038/s41598-019-43102-z

Hofbauer, J. (1995a). Imitation dynamics for games. Unpublished manuscript, University of Vienna.

Hofbauer, J. (1995b). Stability for the best response dynamics. Unpublished manuscript, University of Vienna.

Hofbauer, J. and Sandholm, W. H. (2007). Evolution in games with randomly disturbed payoffs. *Journal of Economic Theory*, 132:47–69. https://doi.org/10.1016/j.jet.2005.05.011

Hofbauer, J. and Sigmund, K. (1998). *Evolutionary Games and Population Dynamics*. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9781139173179

Hofbauer, J. and Sigmund, K. (2003). Evolutionary game dynamics. *Bulletin of the American Mathematical Society*, 40(4):479–519. http://dx.doi.org/10.1090/S0273-0979-03-00988-1

Holt, C. A. and Roth, A. E. (2004). The Nash equilibrium: A perspective. *Proceedings of the National Academy of Sciences*, 101(12):3999–4002. http://dx.doi.org/10.1073/pnas.0308738101

Huberman, B. A. and Glance, N. S. (1993). Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences*, 90:7716–7718. https://doi.org/10.1073/pnas.90.16.7716

Isaac, A. G. (2008). Simulating evolutionary games: a python-based introduction. *Journal of Artificial Societies and Social Simulation*, 11(3):8. http://jasss.soc.surrey.ac.uk/11/3/8.html

Izquierdo, L. R., Izquierdo, S. S., Galán, J. M., and Santos, J. I. (2009). Techniques to understand computer simulations: Markov chain analysis. *Journal of Artificial Societies and Social Simulation*, 12(1):6. http://jasss.soc.surrey.ac.uk/12/1/6.html

Izquierdo, L. R., Izquierdo, S. S., Galán, J. M., and Santos, J. I. (2013). Combining mathematical and simulation approaches to understand the dynamics of computer models. In Edmonds, B. and Meyer, R., editors, *Simulating Social Complexity: A Handbook*, chapter 11, pages 235–271. Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-540-93813-2_11. Second edition (2017) available at: https://doi.org/10.1007/978-3-319-66948-9_13

Izquierdo, L. R., Izquierdo, S. S., Galán, J. M., Santos, J. I., and Sandholm, W. H. (2022). Schelling-Sakoda model of spatial segregation. Software available at https://luis-r-izquierdo.github.io/schelling-sakoda-refuting-machine/. https://doi.org/10.5281/zenodo.7065864

Izquierdo, L. R., Izquierdo, S. S., and Hauert, C. (2024). Pair approximation for 2×2 symmetric games on regular networks. Software available at https://github.com/luis-r-izquierdo/pair-approximation. https://doi.org/10.5281/zenodo.10975163

Izquierdo, L. R., Izquierdo, S. S., and Rodríguez, J. (2022). Fast and scalable global convergence in single-optimum decentralized coordination problems. *IEEE Transactions on Control of Network Systems*, 9(4):1937–1948. https://doi.org/10.1109/TCNS.2022.3181545

Izquierdo, L. R., Izquierdo, S. S., and Sandholm, W. H. (2019). An introduction to ABED: Agent-based simulation of evolutionary game dynamics. *Games and Economic Behavior*, 118:434–462. https://doi.org/10.1016/j.geb.2019.09.014

Izquierdo, L. R., Izquierdo, S. S., and Vega-Redondo, F. (2012). Learning and evolutionary game theory. In Seel, N. M., editor, *Encyclopedia of the Sciences of Learning*, pages 1782–1788. Springer US, Boston, MA. https://doi.org/10.1007/978-1-4419-1428-6_576

Izquierdo, L. R., Izquierdo, S. S., and Vega-Redondo, F. (2014). Leave and let leave: A sufficient condition to explain the evolutionary emergence of cooperation. *Journal of Economic Dynamics and Control*, 46:91–113. https://doi.org/10.1016/j.jedc.2014.06.007

Izquierdo, L. R. and Polhill, J. G. (2006). Is your model susceptible to floating-point errors? *Journal of Artificial Societies and Social Simulation*, 9(4):4. http://jasss.soc.surrey.ac.uk/9/4/4.html

Izquierdo, S. S. and Izquierdo, L. R. (2013). Stochastic approximation to understand simple simulation models. *Journal of Statistical Physics*, 151(1):254–276. http://dx.doi.org/10.1007/s10955-012-0654-z

Janssen, M.A. (2020). *Introduction to Agent-Based Modeling: With applications to social, ecological and social-ecological systems*. https://intro2abm.com

Janssen, J. and Manca, R. (2006). *Applied semi-markov processes*. Springer-Verlag, New York. http://dx.doi.org/10.1007/0-387-29548-8

Jaxa-Rozen, M. and Kwakkel, J. H. (2018). PyNetlogo: Linking NetLogo with Python. *Journal of Artificial Societies and Social Simulation*, 21(2):4. https://dx.doi.org/10.18564/jasss.3668

Kandori, M. and Rob, R. (1995). Evolution of equilibria in the long run: A general theory and applications. *Journal of Economic Theory*, 65:383–414. https://doi.org/10.1006/jeth.1995.1014

Karr, A. F. (1990). Markov processes. In Heyman, D. P. and Sobel, M. J. (eds.), *Stochastic Models*, volume 2 of *Handbooks in Operations Research and Management Science*, chapter 2, pages 95–123. Elsevier. https://doi.org/10.1016/S0927-0507(05)80166-5

Killingback, T. and Doebeli, M. (1996). Spatial evolutionary game theory: Hawks and doves revisited. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 263(1374):1135–1144. https://doi.org/10.1098/rspb.1996.0166

Kosfeld, M., Droste, E., and Voorneveld, M. (2002). A myopic adjustment process leading to best reply matching. *Journal of Economic Theory*, 40:270–298. https://doi.org/10.1016/S0899-8256(02)00007-6

Kravari, K. and Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11. https://doi.org/10.18564/jasss.2661

Kulkarni, V. G. (1995). *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, Ltd., London, UK.

Kulkarni, V. G. (1999). *Modeling, Analysis, Design, and Control of Stochastic Systems*. Springer New York, NY. https://doi.org/10.1007/978-1-4757-3098-2

Lahkar, R. and Sandholm, W. H. (2008). The projection dynamic and the geometry of population games. *Games*

*and Economic Behavior*, 64(2):565–590. Special Issue in Honor of Michael B. Maschler. https://doi.org/10.1016/j.geb.2008.02.002

Lipowski, A. and Lipowska, D. (2012). Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6):2193–2196. https://doi.org/10.1016/j.physa.2011.12.004

Loginov, G. (2022). Ordinal Imitative Dynamics. *International Journal of Game Theory*, 51:391–412. https://doi.org/10.1007/s00182-021-00797-7

Lytinen, S. L. and Railsback, S. F. (2012). The evolution of agent-based simulation platforms: A review of NetLogo 5.0 and ReLogo. In *Proceedings of the fourth international symposium on agent-based modeling and simulation (21st European Meeting on Cybernetics and Systems Research)*.

Maciejewski,W., Fu, F., and Hauert, C. (2014). Evolutionary game dynamics in populations with heterogenous structures. *PLOS Computational Biology*, 10(4):1–16. https://doi.org/10.1371/journal.pcbi.1003567

Marden, J. R. and Shamma, J. S. (2015). Game theory and distributed control. In Young, H.P. and Zamir, S., editors, *Handbook of Game Theory with Economic Applications*, volume 4, chapter 16, pages 861–899. Elsevier, Amsterdam. https://doi.org/10.1016/B978-0-444-53766-9.00016-1

Masuda, N. (2007). Participation costs dismiss the advantage of heterogeneous networks in evolution of cooperation. P*roceedings of the Royal Society B: Biological Sciences*, 274(1620):1815–1821. https://doi.org/10.1098/rspb.2007.0294

Maynard Smith, J. (1982). *Evolution and the Theory of Games*. Cambridge University Press, Cambridge.

Maynard Smith, J. and Price, G. R. (1973). The logic of animal conflict. *Nature*, 246:15–18. https://doi.org/10.1038/246015a0

Mitchell, M. (1998). Computation in cellular automata: A selected review. In Gramß, T., Bornholdt, S., Groß, M., Mitchell, M., and Pellizzari, T., editors, *Non-Standard Computation: Molecular Computation – Cellular Automata – Evolutionary Algorithms – Quantum Computers*, chapter 4, pages 95–140. John Wiley & Sons, Ltd. https://doi.org/10.1002/3527602968.ch4

Morita, S. (2008). Extended Pair Approximation of Evolutionary Game on Complex Networks. *Progress of Theoretical Physics*, 119(1):29–38. https://doi.org/10.1143/PTP.119.29

Mukherji, A., Rajan, V., and Slagle, J. R. (1996). Robustness of cooperation. *Nature*, 379(6561):125–126. https://dx.doi.org/10.1038/379125b0

Myerson, R. B. (1997). *Game theory: Analysis of Conflict*. Harvard University Press.

Nakamaru, M., Matsuda, H., and Iwasa, Y. (1997). The evolution of cooperation in a lattice-structured population. *Journal of Theoretical Biology*, 184(1):65–81. https://doi.org/10.1006/jtbi.1996.0243

Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36:48–49. https://doi.org/10.1073/pnas.36.1.48

Nelson, R. R. and Winter, S. G. (1982). *An Evolutionary Theory of Economic Change*. Harvard University Press.

Newth, D. and Cornforth, D. (2009). Asynchronous spatial evolutionary games. *Biosystems*, 95(2):120–129. https://doi.org/10.1016/j.biosystems.2008.09.003

Newton, J. (2018). Evolutionary game theory: A renaissance. *Games*, 9(2), 31. https://doi.org/10.3390/g9020031

Nikolai, C. and Madey, G. (2009). Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2):2. http://jasss.soc.surrey.ac.uk/12/2/2.html

Norris, J. R. (1997). *Markov Chains*. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511810633

Nowak, M. A., Bonhoeffer, S., and May, R. M. (1994a). More spatial games. *International Journal of Bifurcation and Chaos*, 4:33–56. https://doi.org/10.1142/S0218127494000046

Nowak, M. A., Bonhoeffer, S., and May, R. M. (1994b). Spatial games and the maintenance of cooperation. *Proceedings of the National Academy of Sciences*, 91:4877–4881. https://doi.org/10.1073/pnas.91.11.4877

Nowak, M. A., Bonhoeffer, S., and May, R. M. (1996). Robustness of cooperation. *Nature*, 379(6561):126–126. https://doi.org/10.1038/379126a0

Nowak, M. A. and May, R. M. (1992). Evolutionary games and spatial chaos. *Nature*, 359:826–829. http://dx.doi.org/10.1038/359826a0

Nowak, M. A. and May, R. M. (1993). The spatial dilemmas of evolution. *International Journal of Bifurcation and Chaos*, 3:35–78. https://doi.org/10.1142/S0218127493000040

Ohtsuki, H., Nowak, M. A., and Pacheco, J. M. (2007a). Breaking the symmetry between interaction and replacement in evolutionary dynamics on graphs. *Phys. Rev. Lett.*, 98:108106. https://doi.org/10.1103/PhysRevLett.98.108106

Ohtsuki, H., Pacheco, J. M., and Nowak, M. A. (2007b). Evolutionary graph theory: Breaking the symmetry between interaction and replacement. *Journal of Theoretical Biology*, 246(4):681–694. https://doi.org/10.1016/j.jtbi.2007.01.024

Osborne, M. (2004). *An Introduction to Game Theory*. Oxford University Press, Oxford.

Osborne, M. J. and Rubinstein, A. (1998). Games with procedurally rational players. *American Economic Review*, 88:834–847. https://www.jstor.org/stable/117008

Oyama, D., Sandholm, W. H., and Tercieux, O. (2015). Sampling best response dynamics and deterministic equilibrium selection. *Theoretical Economics*, 10:243–281. https://doi.org/10.3982/TE1405

Perc, M. and Szolnoki, A. (2010). Coevolutionary games—a mini review. *Biosystems*, 99(2):109–125. https://doi.org/10.1016/j.biosystems.2009.10.003

Polhill, J. G., Izquierdo, L. R., and Gotts, N. M. (2006). What every agent-based modeller should know about floating point arithmetic. *Environmental Modelling & Software*, 21(3):283–309. https://doi.org/10.1016/j.envsoft.2004.10.011

Poncela, J., Gómez-Gardeñes, J., Floría, L. M., and Moreno, Y. (2007). Robustness of cooperation in the

evolutionary prisoner's dilemma on complex networks. *New Journal of Physics*, 9(6):184. https://doi.org/10.1088/1367-2630/9/6/184

Probst, D. (1999). Book review of "Evolutionary Game Theory" by Jörgen W. Weibull. *Journal of Artificial Societies and Social Simulation*, 2(1). http://jasss.soc.surrey.ac.uk/2/1/review3.html

Python Software Foundation (2019). Python. *Software*. http://www.python.org.

Quijano, N., Ocampo-Martinez, C., Barreiro-Gomez, J., Obando, G., Pantoja, A., and Mojica-Nava, E. (2017). The role of population games and evolutionary dynamics in distributed control systems: The advantages of evolutionary game theory. *IEEE Control Systems Magazine*, 37(1):70–97. https://doi.org/10.1109/MCS.2016.2621479

R Core Team (2019). R: A Language and Environment for Statistical Computing. *Software*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org.

Railsback, S., Ayllón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C., and Thiele, J. (2017). Improving execution speed of models implemented in netlogo. *Journal of Artificial Societies and Social Simulation*, 20(1):3. https://doi.org/10.18564/jasss.3282

Railsback, S. F. and Grimm, V. (2019). *Agent-Based and Individual-Based Modeling: A Practical Introduction*, Second edition. Princeton University Press, Princeton, NJ. http://www.railsback-grimm-abm-book.com. First edition (2011) at http://www.jstor.org/stable/j.ctt7sns7

Railsback, S. F., Lytinen, S. L., and Jackson, S. K. (2006). Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609–623. https://doi.org/10.1177/0037549706073695

Rand, D. A. (1999). Correlation Equations and Pair Approximations for Spatial Ecologies. In *Advanced Ecological Theory*, J. McGlade (Ed.), chapter 4, pages 100–142. John Wiley & Sons, Ltd. https://doi.org/10.1002/9781444311501.ch4

Roca, C. P., Cuesta, J. A., and Sánchez, A. (2009a). Evolutionary game theory: Temporal and spatial effects beyond replicator dynamics. *Physics of Life Reviews*, 6(4):208 – 249. https://doi.org/10.1016/j.plrev.2009.08.001

Roca, C. P., Cuesta, J. A., and Sánchez, A. (2009b). Effect of spatial structure on the evolution of cooperation. *Phys. Rev. E*, 80:046106. https://doi.org/10.1103/PhysRevE.80.046106

Roth, G. and Sandholm, W. H. (2013). Stochastic approximations with constant step size and differential inclusions. *SIAM Journal on Control and Optimization*, 51:525–555. https://doi.org/10.1137/110844192

Sakoda, J. M. (1949). *Minidoka: An Analysis of Changing Patterns of Social Behavior*. PhD thesis, University of California.

Sakoda, J. M. (1971). The checkerboard model of social interaction. *The Journal of Mathematical Sociology*, 1(1):119–132. https://doi.org/10.1080/0022250X.1971.9989791

Samuelson, L. (1997). *Evolutionary Games and Equilibrium Selection*. MIT Press, Cambridge.

Sandholm, W. H. (2001). Almost global convergence to *p*-dominant equilibrium. *International Journal of Game Theory*, 30:107–116. https://doi.org/10.1007/s001820100067

Sandholm, W. H. (2003). Evolution and equilibrium under inexact information. *Games and Economic Behavior*, 44:343–378. https://doi.org/10.1016/S0899-8256(03)00026-5

Sandholm, W. H. (2007). Simple formulas for stationary distributions and stochastically stable states. *Games and Economic Behavior*, 59:154–162. https://doi.org/10.1016/j.geb.2006.07.001

Sandholm, W. H. (2009). Evolutionary game theory. In Meyers, R. A., editor, *Encyclopedia of Complexity and Systems Science*, pages 3176–3205. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-27737-5_188-3

Sandholm, W. H. (2010a). *Population Games and Evolutionary Dynamics*. MIT Press, Cambridge.

Sandholm, W. H. (2010b). Pairwise comparison dynamics and evolutionary foundations for Nash equilibrium. *Games*, 1:3–17. https://doi.org/10.3390/g1010003

Sandholm, W. H., Izquierdo, S. S., and Izquierdo, L. R. (2019). Best experienced payoff dynamics and cooperation in the Centipede game. *Theoretical Economics*, 14: 1347–1385. https://doi.org/10.3982/TE3565

Sandholm, W. H., Izquierdo, S. S., and Izquierdo, L. R. (2020). Stability for best experienced payoff dynamics. *Journal of Economic Theory*, 185:104957. https://doi.org/10.1016/j.jet.2019.104957

Sandholm, W. H. and Staudigl, M. (2018). Sample path large deviations for stochastic evolutionary game dynamics. *Mathematics of Operations Research*, 43(4):1348–1377. https://doi.org/10.1287/moor.2017.0908

Santos, F. C., Rodrigues, J. F., and Pacheco, J. M. (2006a). Graph topology plays a determinant role in the evolution of cooperation. *Proceedings of the Royal Society B: Biological Sciences*, 273(1582):51–55. https://doi.org/10.1098/rspb.2005.3272

Santos, F. C. and Pacheco, J. M. (2005). Scale-free networks provide a unifying framework for the emergence of cooperation. *Phys. Rev. Lett.*, 95:098104. https://doi.org/10.1103/PhysRevLett.95.098104

Santos, F. C. and Pacheco, J. M. (2006). A new route to the evolution of cooperation. *Journal of Evolutionary Biology*, 19(3):726–733. https://doi.org/10.1111/j.1420-9101.2005.01063.x

Santos, F. C., Pacheco, J. M., and Lenaerts, T. (2006b). Evolutionary dynamics of social dilemmas in structured heterogeneous populations. *Proceedings of the National Academy of Sciences*, 103(9):3490–3494. https://doi.org/10.1073/pnas.0508201103

Sayama, H. (2015). *Introduction to the Modeling and Analysis of Complex Systems*. Milne Open Textbooks. https://milneopentextbooks.org/introduction-to-the-modeling-and-analysis-of-complex-systems/

Schelling, T. C. (1969). Models of segregation. *The American Economic Review*, 59(2):488–493. http://www.jstor.org/stable/1823701

Schelling, T. C. (1971). Dynamic models of segregation. *The Journal of Mathematical Sociology*, 1(2):143–186. https://doi.org/10.1080/0022250X.1971.9989794

Schelling, T. C. (1978). *Micromotives and Macrobehavior*. Norton, New York.

Schlag, K. H. (1998). Why imitate, and if so, how? A boundedly rational approach to multi-armed bandits. *Journal of Economic Theory*, 78:130–156. https://doi.org/10.1006/jeth.1997.2347

Schwarz, K. (2011). Darts, dice, and coins: Sampling from a discrete distribution. https://www.keithschwarz.com/darts-dice-coins/

Selten, R. (1965). Spieltheoretische Behandlung eines Oligopolmodells mit Nachfrageträgheit. *Zeitschrift für die gesamte Staatswissenschaft / Journal of Institutional and Theoretical Economics*, 121(2):301–324. http://www.jstor.org/stable/40748884

Selten, R. (1975). Reexamination of the perfectness concept for equilibrium points in extensive games. *International Journal of Game Theory*, 4(1):25–55. https://doi.org/10.1007/BF01766400

Seri, R. (2016). Analytical approaches to agent-based models. In Secci, D. and Neumann, M., editors, *Agent-Based Simulation of Organizational Behavior*, chapter 13, pages 265–286. Springer International Publishing. https://doi.org/10.1007/978-3-319-18153-0_13

Sethi, R. (2000). Stability of equilibria in games with procedurally rational players. *Games and Economic Behavior*, 32:85–104. https://doi.org/10.1006/game.1999.0753

Sethi, R. (2021). Stable sampling in repeated games. *Journal of Economic Theory*, 197:105343. https://doi.org/10.1016/j.jet.2021.105343

Sigmund, K. (1983). *Games of Life: Explorations in Ecology, Evolution, and Behaviour*. Oxford University Press.

Sklar, E. (2007). NetLogo, a multi-agent simulation environment. *Artificial Life*, 13(3):303–311. https://doi.org/10.1162/artl.2007.13.3.303

Skyrms, B. (2001). The Stag Hunt. *Proceedings and Addresses of the American Philosophical Association*, 75(2):31–41. https://doi.org/10.2307/3218711

Smaldino, P. E. (2023). *Modeling Social Behavior: Mathematical and Agent-Based Models of Social Dynamics and Cultural Evolution*. Princeton University Press.

Smith, M. J. (1984). The stability of a dynamic model of traffic assignment —an application of a method of Lyapunov. *Transportation Science*, 18:245–252. https://doi.org/10.1287/trsc.18.3.245

Szabó, G. and Fáth, G. (2007). Evolutionary games on graphs. *Physics Reports*, 446:97–216. https://doi.org/10.1016/j.physrep.2007.04.004

Szabó, G. and Tőke, C. (1998). Evolutionary prisoner's dilemma game on a square lattice. *Phys. Rev. E*, 58:69–73. https://doi.org/10.1103/PhysRevE.58.69

Szolnoki, A., Perc, M., and Danku, Z. (2008). Towards effective payoffs in the prisoner's dilemma game on scale-free networks. *Physica A: Statistical Mechanics and its Applications*, 387(8):2075–2082. https://doi.org/10.1016/j.physa.2007.11.021

Taylor, P. D. and Jonker, L. (1978). Evolutionarily stable strategies and game dynamics. *Mathematical Biosciences*, 40:145–156. https://doi.org/10.1016/0025-5564(78)90077-9

The MathWorks, Inc. (2019). Matlab. *Software*. Natick, Massachusetts. https://mathworks.com

Thiele, J. C. (2014). R marries NetLogo: Introduction to the RNetLogo package. *Journal of Statistical Software*, 58(2):1–41. http://dx.doi.org/10.18637/jss.v058.i02

Thiele, J. C. and Grimm, V. (2010). NetLogo meets R: Linking agent-based models with a toolbox for their analysis. *Environmental Modelling & Software*, 25(8):972–974. https://doi.org/10.1016/j.envsoft.2010.02.008

Thiele, J. C., Kurth, W., and Grimm, V. (2012a). Agent-based modelling: Tools for linking netlogo and R. *Journal of Artificial Societies and Social Simulation*, 15(3):8. http://dx.doi.org/10.18564/jasss.2018

Thiele, J. C., Kurth, W., and Grimm, V. (2012b). RNetLogo: An R package for running and exploring individual-based models implemented in NetLogo. *Methods in Ecology and Evolution*, 3(3):480–483. http://dx.doi.org/10.1111/j.2041-210X.2011.00180.x

Thiele, J. C., Kurth, W., and Grimm, V. (2014). Facilitating parameter estimation and sensitivity analysis of agent-based models: A cookbook using netlogo and R. *Journal of Artificial Societies and Social Simulation*, 17(3):11. http://dx.doi.org/10.18564/jasss.2503

Thomas, B. (1984). Evolutionary stability: States and strategies. *Theoretical Population Biology*, 26:49–67. https://doi.org/10.1016/0040-5809(84)90023-6

Tomassini, M., Pestelacci, E., and Luthi, L. (2007). Social dilemmas and cooperation in complex networks. *International Journal of Modern Physics C*, 18(07):1173–1185. https://doi.org/10.1142/S0129183107011212

Traulsen, A. and Hauert, C. (2009). Stochastic evolutionary game dynamics. In Schuster, H. G., editor, *Reviews of Nonlinear Dynamics and Complexity*, volume 2, pages 25–61. Wiley, New York. https://doi.org/10.1002/9783527628001.ch2

van Baalen, M. (2000). Pair Approximations for Different Spatial Geometries. In Dieckmann, U., Law, R., and Metz, J. A. J., editors, *The Geometry of Ecological Interactions: Simplifying Spatial Complexity*, pages 359–387. Cambridge Studies in Adaptive Dynamics. Cambridge University Press. https://doi.org/10.1017/CBO9780511525537.023

Vega-Redondo, F. (1997). The evolution of Walrasian behavior. *Econometrica*, 65:375–384. https://doi.org/10.2307/2171898

Vega-Redondo, F. (2003). *Economics and the Theory of Games*. Cambridge University Press, Cambridge, UK.

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Prentice-Hall, Princeton.

Vose, M. (1991). A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering*, 17(9):972–975. https://doi.org/10.1109/32.92917

Wallace, C. and Young, H. P. (2015). Stochastic evolutionary game dynamics. In Young, H.P. and Zamir, S., editors, *Handbook of Game Theory with Economic Applications*, volume 4, chapter 6, pages 327–380. Elsevier, Amsterdam. https://doi.org/10.1016/B978-0-444-53766-9.00006-9

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442. https://doi.org/10.1038/30918

Weibull, J. W. (1995). *Evolutionary Game Theory*. MIT Press, Cambridge.

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

Wilensky, U. (2023). *The NetLogo User Manual*. Version 6.4.0. https://ccl.northwestern.edu/netlogo/6.4.0/docs/

Wilensky, U. (2005a). *NetLogo Preferential Attachment model*. http://ccl.northwestern.edu/netlogo/models/ PreferentialAttachment. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wilensky, U. (2005b). *Mouse Drag One Example*. http://ccl.northwestern.edu/netlogo/models/ MouseDragOneExample. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wilensky, U. and Rand, W. (2015). *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. The MIT Press. https://www.jstor.org/stable/j.ctt17kk851

Wilensky, U. and Shargel, B. (2002). Behaviorspace. *Software*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/ behaviorspace.html

Wilensky, U. and Stroup, W. (1999). Hubnet. *Software*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/hubnet.html

Woelfing, B. and Traulsen, A. (2009). Stochastic sampling of interaction partners versus deterministic payoff assignment. *Journal of Theoretical Biology*, 257(4):689–695. https://doi.org/10.1016/j.jtbi.2008.12.025

Wolfram, S. (1983). Statistical mechanics of cellular automata. *Reviews of modern physics*, 55(3):601. https://doi.org/10.1103/RevModPhys.55.601

Wolfram, S. (2002). *A new kind of science*. Wolfram-Media. https://www.wolframscience.com/nks/

Wolfram Research, Inc. (2019). Mathematica. *Software*. Champaign, Illinois. https://www.wolfram.com

Wu, Z.-X., Guan, J.-Y., Xu, X.-J., and Wang, Y.-H. (2007). Evolutionary prisoner's dilemma game on Barabási–Albert scale-free networks. *Physica A: Statistical Mechanics and its Applications*, 379(2):672–680. https://doi.org/10.1016/j.physa.2007.02.085

Wu, Z.-X., Xu, X.-J., and Wang, Y.-H. (2005). Does the scale-free topology favor the emergence of cooperation?. arXiv:physics/0508220 https://doi.org/10.48550/arXiv.physics/0508220

Young, H. P. (1998). *Individual Strategy and Social Structure*. Princeton University Press, Princeton.

Young, H. P. (2004). *Strategic Learning and Its Limits*. Oxford University Press, Oxford.