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, using NetLogo

LUIS R. IZQUIERDO, SEGISMUNDO S. IZQUIERDO, AND WILLIAM H. SANDHOLM

Agent-Based Evolutionary Game Dynamics by Luis R. Izquierdo, Segismundo S. Izquierdo & William H. Sandholm is licensed under a Creative Commons Attribution 4.0 International License, except where otherwise noted.

Cover picture by Dino Reichmuth.

This book was produced using Pressbooks.com, and PDF rendering was done by PrinceXML.

Contents

Preface vii

0. Introduction

0.1. Introduction to evolutionary game theory 2
0.2. Introduction to agent-based modeling 14
0.3. Introduction to NetLogo 21
0.4. The fundamentals of NetLogo 26

1. Our first agent-based evolutionary model

1.0. Our very first model 45
1.1. Extension to any number of strategies 57
1.2. Noise and initial conditions 67
1.3. Interactivity and efficiency 76
1.4. Analysis of these models 88

2. Spatial interactions on a grid

2.0. Spatial chaos in the Prisoner's Dilemma 100

3. Games on networks

4. Revision protocols and general payoff functions

5. Endogenous networks

6. Multipopulation games

7. Solving the mean dynamics at runtime

References 114

Preface

1. What is this book about?

The objective of this book is to help you learn to *implement* and *analyze* evolutionary models of social interactions in finite populations. The following paragraphs explain why these two skills –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 consistent and completely specified. A computer implementation of a model is necessarily *consistent* 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. This implies that the audience may be understanding something slightly different from what is meant to be communicated, and results may be driven by assumptions that are not made explicitly. By contrast, computers need all assumptions to be spelt out, and this requirement makes the process of scientific modelling 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 chain analyses, Monte Carlo simulations, mean dynamics, stochastic stability analyses and diffusion 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. How to use this book

This book has been written and formatted following a hands-on approach. To make the most of it, we encourage you to have NetLogo open, and to code the models as you read the book. 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 sections, you will master the art of implementing and analyzing agent-based evolutionary dynamics.

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

- If you are interested in learning to analyze finite-population evolutionary models, and in their relation with other models in Evolutionary Game Theory, but **you do not wish to program**, then you should skip all the sections preceded by the label CODE.
- 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 section titled "Analysis of these models", which you will find at the end of most chapters.

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.

0. INTRODUCTION

0.1. Introduction to evolutionary game theory

Evolutionary Game Theory (EGT) is a branch of a more general discipline called game theory. Therefore, to understand what EGT is about, 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?

Interactive 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,

evolutionary models where payoffs represent Darwinian fitness.

		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. 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 one side of the road, one text messaging app, and one restaurant in town that is preferred, if only slightly. 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.

---

preferences in game theory are assumed to account for anything, i.e. they may include altruistic 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).

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, this equilibrium is worse than any of the other

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.

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 of 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 (⅔ , ⅓), 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** = (⅔ , ⅓) 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,

THEN:

- the frequency of strategies with greater payoffs will tend to increase (and this change in strategy

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.

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 revision protocol, 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 section, let us revisit our coordination example (with payoff matrix shown in Fig. 1) in a population context. We will analyze two revision protocols that lead to different results: imitative pairwise-difference protocol and best experienced payoff protocol.

- Under the *imitative pairwise-difference protocol* (Helbing (1992), Hofbauer (1995), Schlag (1998)), a revising agent looks at another individual at random and imitates 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 protocol 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 protocol in a coordination game.

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

9. To prove this statement, note that the mean dynamic of this revision protocol is the well-known replicator dynamic (Taylor and Jonker, 1978)
10. See exercise 4 in section 1.0

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=94

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

- Under the *best experienced payoff protocol* (Osborne and Rubinstein (1998), Sethi (2000), Sandholm et al. (2017), Izquierdo et al. (2018b)), 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 protocol in large populations will tend to approach the state where every agent plays action B for any initial condition (Fig. 3).[11]

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

The following video shows some NetLogo simulations that illustrate these dynamics. We will learn to implement this model in this book.[12]

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=94

Simulation runs of the best experienced payoff protocol in coordination game [[1 0][0 2]].

The example above shows that different protocols can lead to very different dynamics in non-trivial ways. Both protocols 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 protocols will almost certainly lead the population to the state where everyone plays A, while the other protocol 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.

Evolutionary Game Theory and Engineering

Many engineering infrastructures are becoming increasingly complex to manage due to their large-scale distributed

11. This statement is a direct application of proposition 4.5 in Izquierdo et al. (2018b)
12. See exercise 5 in section 1.0

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 revision protocols 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 protocol is more likely to lead to the most efficient outcome than the imitative pairwise-difference protocol.

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 revision protocol.

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 revision protocols 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.

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).

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).

- 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), Sandholm (2009).
    - Advanced: Hofbauer and Sigmund (1988), Weibull (1995), Samuelson (1997), Sandholm (2010).
    - 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).

0.2. Introduction to agent-based modeling

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.

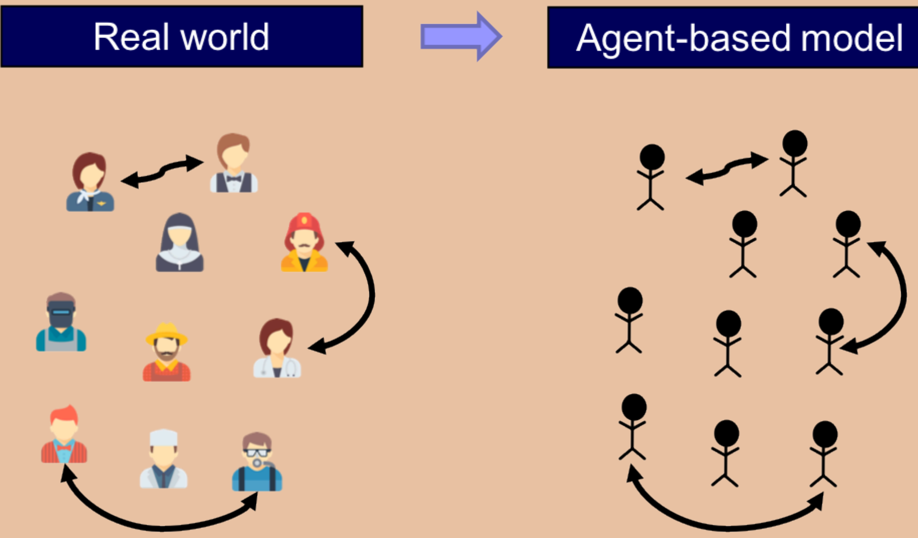


Figure 1. In an agent-based model, the individual units of the real-world system to be modeled and their interactions are explicitly and individually represented in the model.

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.

1. These three videos by Bruce Edmonds and Michael Price and Uri Wilensky nicely describe what ABM is about.

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 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

2. The extent to which *repeated* interactions are *explicitly* represented in traditional game theoretical models is not so clear.

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 they 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 revision protocol)
- 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).[3][4] 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 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]

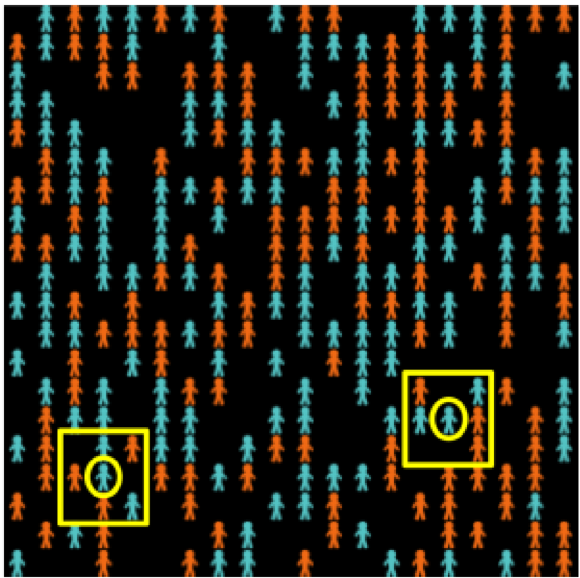


Figure 2. Grid of Schelling-Sakoda model (20×20), with 133 blue agents, 133 orange agents. The agents in yellow circles have 2 out of 5 neighbors of the same color.

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.

---

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).
5. Izquierdo et al. (2009, appendix B) analyze this model as a Markov chain.
6. Cells on a side have five neighbors and cells at a corner have three neighbors.

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.

Individually-owned variables and instructions

In this model, agent's individually-owned variables are:

- color, which can be either blue or orange,
- (xcor, ycor), which determine the agent's location on the grid, and
- happy?, which indicates whether the agent is happy or not.

Agents are able to run the following instructions:

- to move, to change the agent's location to a random empty cell, and
- to update-happiness, to update the agent's individually-owned variable happy?.

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.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=40

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.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=40

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.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=40

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). The 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 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

7. See Aydinonat (2007).

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?

The following books are all excellent introductions to scientific agent-based modeling, and all of them make use of NetLogo: Gilbert (2007), Janssen (2010)[8], Railsback and Grimm (2011) and Wilensky and Rand (2015). 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.

8. Free online book

0.3. Introduction to NetLogo

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 (Wilensky, 1999) 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-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

---

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.
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.

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)

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.

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-Users yahoo 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.

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 before moving to the next chapter. 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 6.0.2.[3]
- Go through the three tutorials in the NetLogo user manual, i.e.
    - Tutorial #1: Models
    - 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 section 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.

3. Please, make sure you download version 6.0.2 or greater, since NetLogo syntax changed significantly in version 6.0.

0.4. The fundamentals of NetLogo

This section provides a succinct overview of the fundamentals of NetLogo. It is strongly based on the excellent NetLogo user manual, version 6.0.2 (Wilensky, 2017). By no means do we claim originality on the content of this section; 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 section 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).

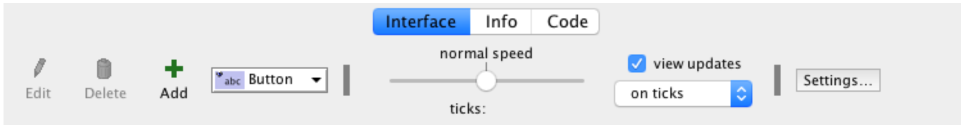


Figure 1. Top bar of the NetLogo Interface tab, where you can select the tab you want to see.

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

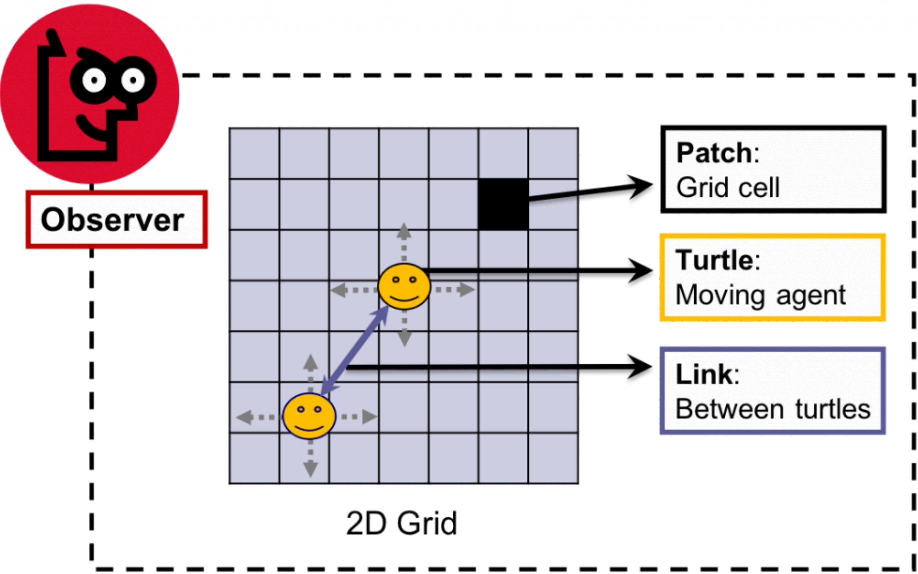


Figure 2. The NetLogo world is made up of turtles, patches, links and the observer.

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 are 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.

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

Here are some more complex things you can do:

```
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).

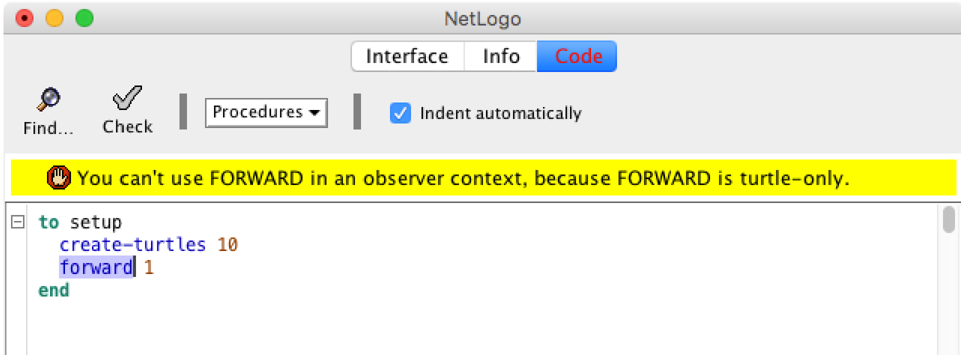


Figure 3. Inconsistency error.

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 section, we present some simple code that implements the Schelling-Sakoda model described in section 0.2 “Introduction to agent-based modeling”. The code we show here is simpler than the one used for the videos in section 0.2, 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).

2. Both implementations lead to exactly the same dynamics.

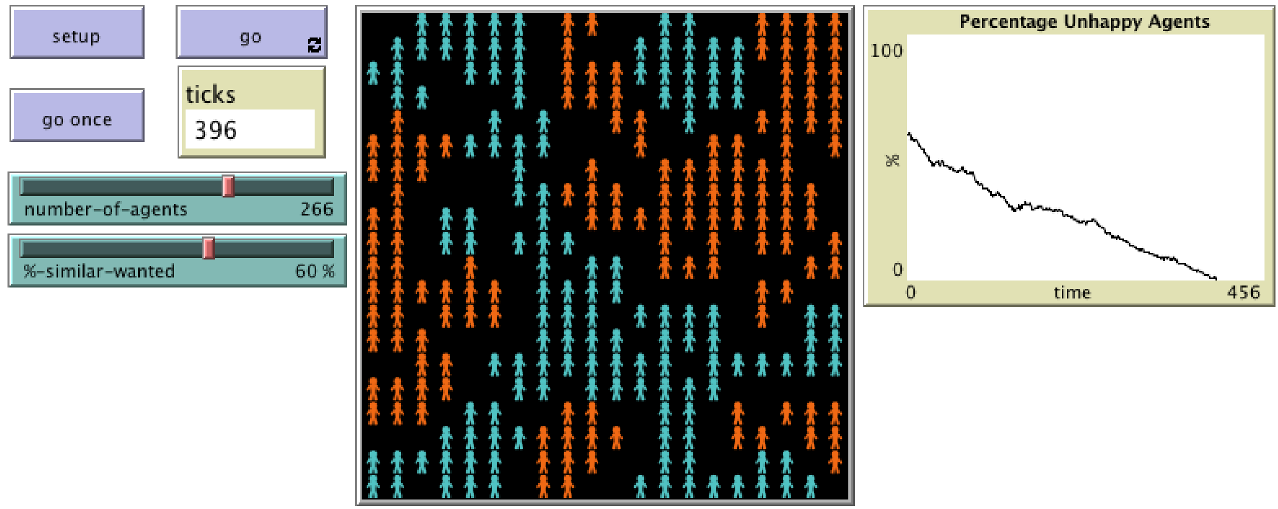


Figure 4. Interface of a simple version of Schelling-Sakoda model.

The code that goes in the code tab is shown below. You can download the whole model here 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:

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.

```
inspect turtle 21
```

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
```

1. OUR FIRST AGENT-BASED EVOLUTIONARY MODEL

1.0. Our very first model

1. Goal

The goal of this section 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,
- an assignment rule, which determines how revision opportunities are assigned to agents, and
- a revision protocol, 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 revision protocol 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 are repeatedly matched to 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 revision 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.

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

CODE 4. Interface design

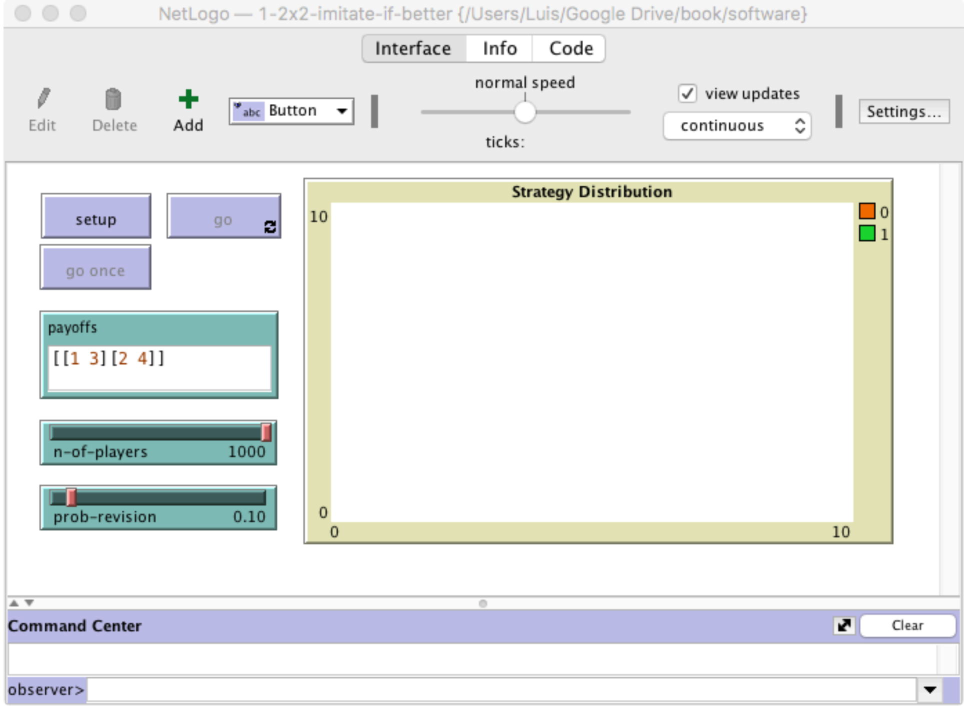


Figure 2. Interface design.

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.
  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

4.1. 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
]
```

4.2. 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:[1]

1. 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.

```
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:

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

where `AGENTSET` should be a random subset o 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
```

4.3. Go procedure

The procedure to go contains all the instructions that will be executed in every tick. In this particular model, these instructions include *a)* asking all players to interact with another (randomly selected) player to obtain a payoff and *b)* 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 play and to update-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. Following this modular design, the procedure to go is particularly easy to code and understand:

```
ask players [play]
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*.

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
```

4.4 Other procedures

to play

Importantly, note that the procedure to play *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. 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 play.

to update-strategy

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, adopt the observed-agent's strategy.

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

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

To compare the payoffs and, if appropriate, adopt the observed-agent's strategy, we can write:

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

This concludes the definition of the procedure to update-strategy and, actually, of all the code in the Code tab, which by now should look as shown below.

4.5. Complete code in the Code tab

```
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
  ask players [play]
```

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

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

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

4.6. 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.

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 revision protocol 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.[2] The video below shows some representative runs.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=18

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.

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 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.

2. 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.

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.

CODE Exercise 4. Reimplement the procedure to update-strategy so the revising agent uses the imitative pairwise-difference protocol that we saw in section 0.1.

CODE Exercise 5. Reimplement the procedure to update-strategy so the revising agent uses the best experienced payoff protocol that we saw in section 0.1.

1.1. Extension to any number of strategies

1. Goal

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

2. Motivation. Rock, paper, scissors

The model we will develop in this section 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 ($i, j \in \{0, 1, \ldots, n\}$). 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 revision 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.

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

CODE 4. Interface design

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

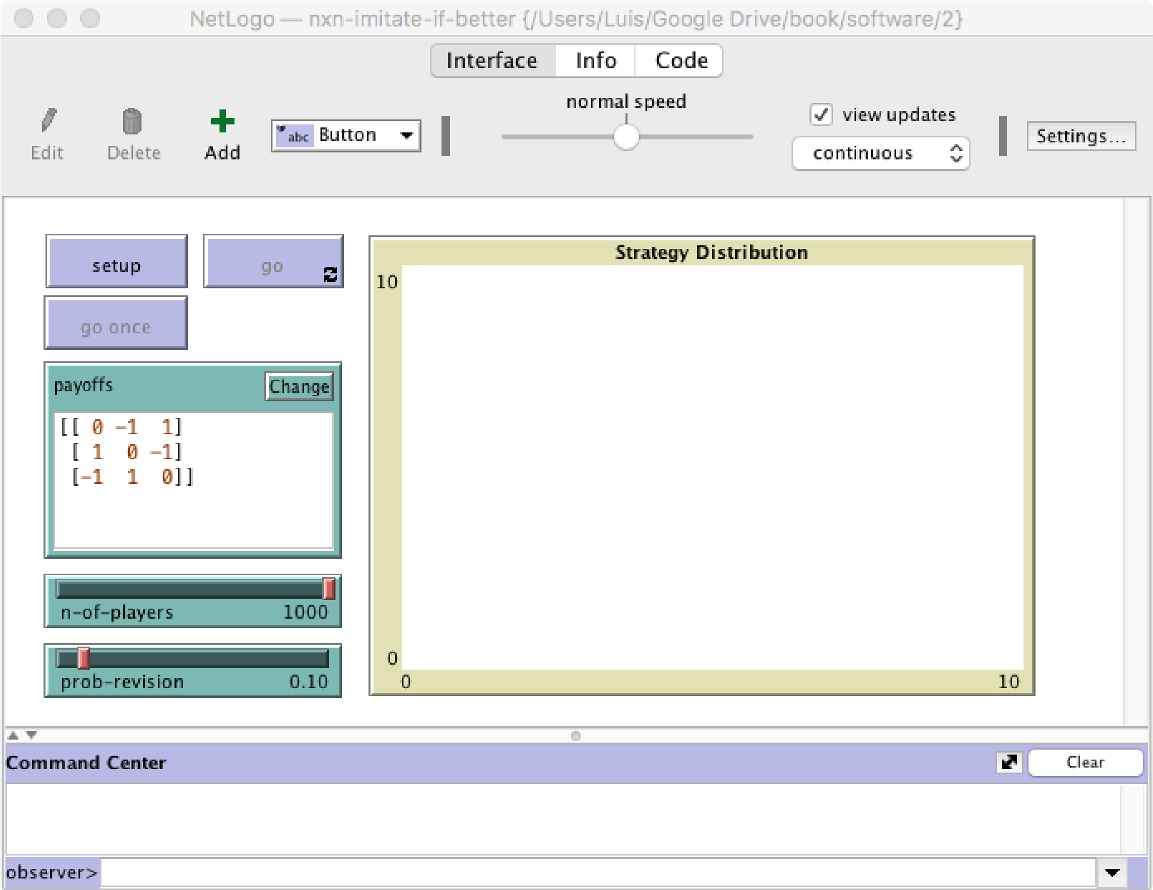


Figure 1. Interface design

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

4.1. 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, then:

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

4.2. 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]
  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).

```
to setup-players
 create-players n-of-players [
   set payoff 0
   set strategy (random n-of-strategies)
 ]
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 $i \in \{0, 1, ..., (\text{n-of-strategies} - 1)\}$, 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 2 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.

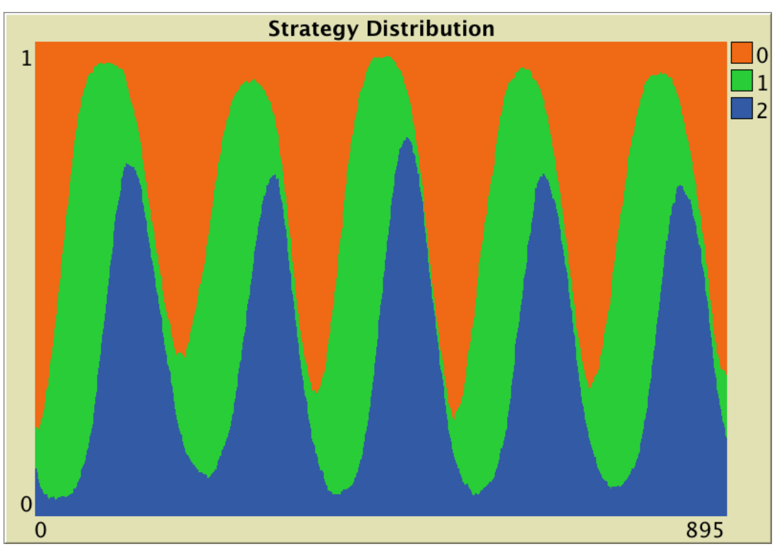


Figure 2. Example of stacked bar chart showing the strategy distribution as ticks go by

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
```

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

```
  tick
  update-graph
end
```

4.4. Other procedures

Note that there is no need to modify the code of to play or to update-strategy.

4.5. Complete code in the Code tab

The Code tab is ready!

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

breed [players player]

players-own [
  strategy
  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)
  ]
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 [play]
  ask players [
    if (random-float 1 < prob-revision) [update-strategy]
  ]
  tick
  update-graph
end

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

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

4.6. 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.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=98

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.

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?

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

CODE 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`.

CODE Exercise 5. Reimplement the procedure to update-strategy 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.

CODE Exercise 6. Reimplement the procedure to update-strategy 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 *sample best response revision protocol* (Sandholm (2001), Kosfeld et al. (2002), Oyama et al. (2015)). Resolve ties as you wish.

1.2. Noise and initial conditions

1. Goal

Our goal is to extend the model we have created in the previous section 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 dynamical behavior dramatically, even creating new attractors. This is important because dynamic characteristics 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. Rock, paper, scissors

In the previous section we saw that simulations of the Rock-Paper-Scissors game under the "imitate if better" revision protocol 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"** protocol:

   > 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 possible strategies at the end of every tick.

CODE 4. Interface design

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

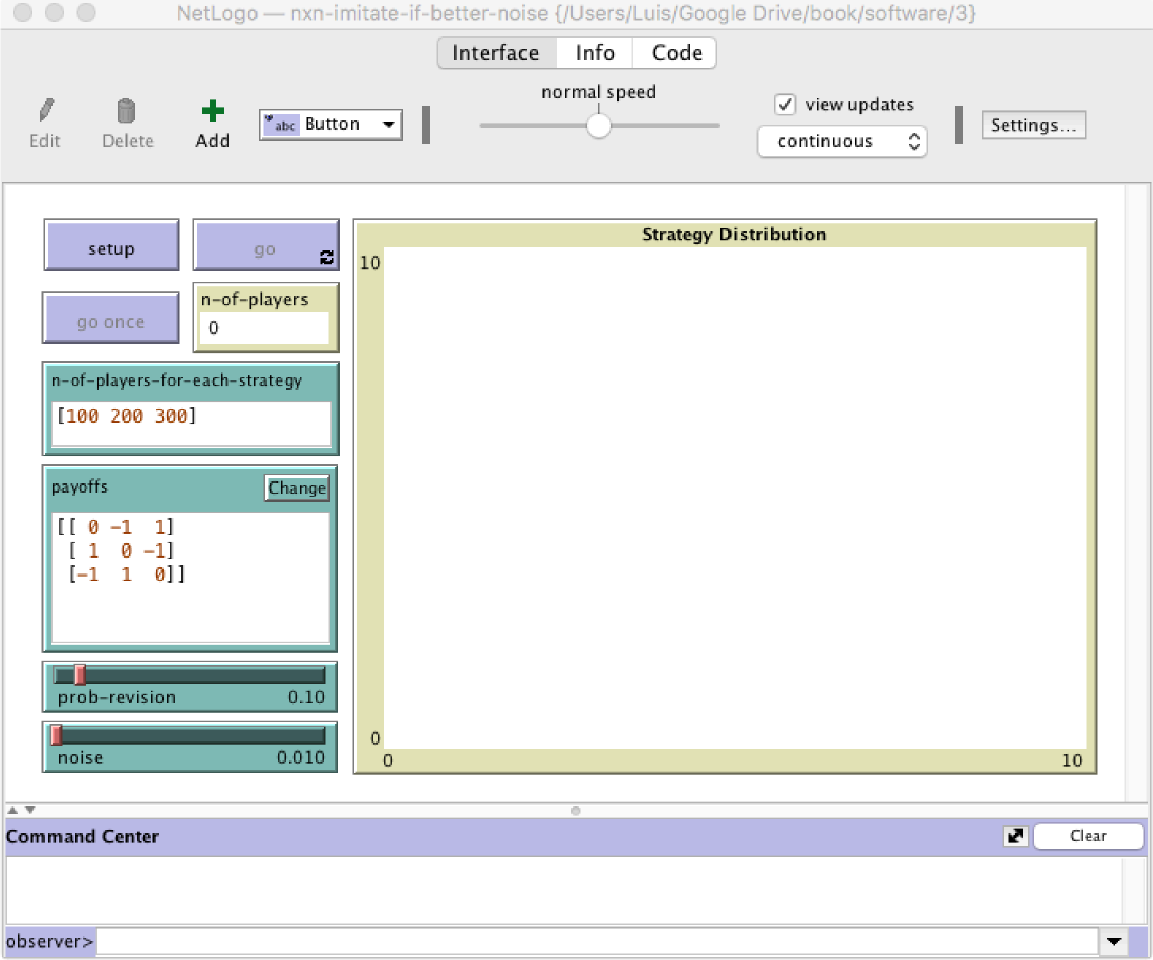


Figure 1. Interface design

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.

CODE 5. Code

4.1. 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.

4.2. 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, implementing short procedures with specific tasks and meaningful names, makes our life easy when we extend 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 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.

4.3. 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. At present, the code of this procedure looks as follows:

```
to update-strategy
  let observed-player one-of other players
  if ([payoff] of observed-player) > payoff [
    set strategy ([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 could be as follows:

```
to update-strategy
  ifelse random-float 1 < noise
  ;; the condition is true with probability noise
    [ ;; code to be executed if there is noise
```

```
      set strategy (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 ([strategy] of observed-player)
      ]
    ]
end
```

4.4. 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
  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 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 [play]
  ask players [
    if (random-float 1 < prob-revision) [update-strategy]
  ]
  tick
  update-graph
end

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

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

```
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" revision protocol? To do that, let us use the same setting as in the previous section, 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.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=103

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 states.

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).

7. Exercises

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

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?

CODE 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?

CODE 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?

1.3. Interactivity and efficiency

CODE 1. Goal

Our goal in this section 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 section, 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 section, we will modify the code of our model slightly to make it run significantly faster.

Sometimes there is a trade-off between interactivity and efficiency. As a matter of fact, making the model more interactive generally implies some loss of efficiency. Nonetheless, oftentimes 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. Rock, paper, scissors

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 Rock-Paper-Scissors game under the “imitate if better” revision protocol with noise? In this section 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 section 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).

CODE 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]
```

And parameter *noise* is used only in procedure to update-strategy, 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.

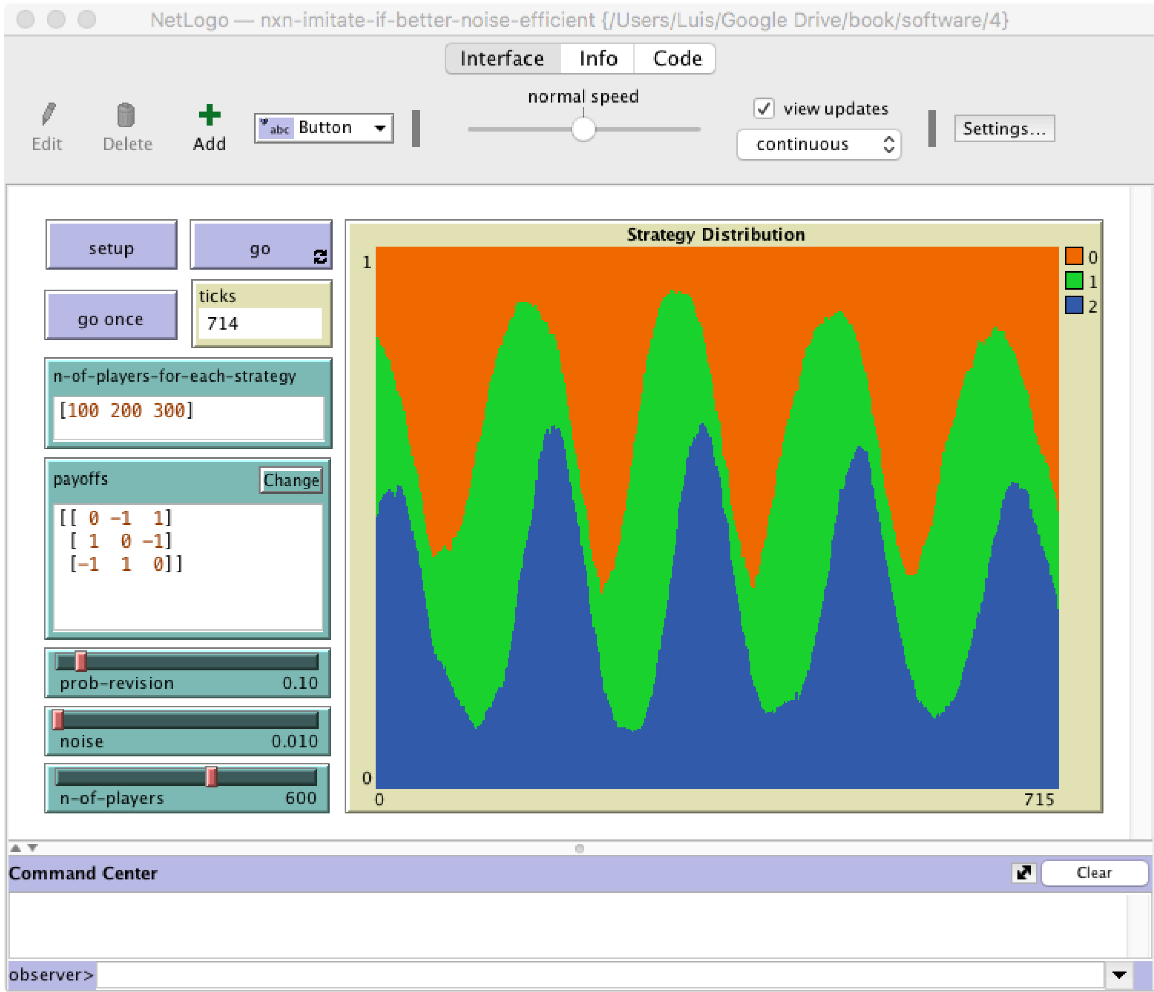


Figure 1. Interface design

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 [play]
  ask players [
    if (random-float 1 < prob-revision) [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!

CODE 5. Efficiency

Naturally, to make a model run faster, one can always untick the "view updates" box on the Interface tab.[1] This is a must in models that do not make use of the view, like the ones we are programming in this chapter, since it implies a significant speed-up at no cost. But beyond this simple piece of advice, in general, how can we know whether our model can run faster? A good first 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.

1. This action is equivalent to pushing the speed slider to its rightmost position and can also be done via code using primitive no-display

- 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.

4.1. Example of computations that we conduct but do not use

Can you identify computations that we perform in the current implementation but are not actually needed (i.e. the model would behave in the same way without carrying them out)?

Note that in this model we make all agents play 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 play. One way of implementing this efficiency improvement would be to modify the code of procedures to go and to update-strategy as follows:

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

to update-strategy
  let observed-player one-of other players

  play                        ;; <== New line
  ask observed-player [play]  ;; <== New line

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

These changes will make simulations with low *prob-revision* run much faster.

4.2. 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 a computation 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 play:

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

And also every time an agent revises her strategy in to update-strategy:

```
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
  payoff
  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-agents).

```
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-agents, 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 code

```
other players
```

we should write other-players instead. These changes will make simulations with many players run faster.

4.3. 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
```

A 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 well documented here. 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.

```
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
```

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. An example of the output printed by `show-profiler-report` follows:

```
BEGIN PROFILING DUMP
Sorted by Exclusive Time
Name                          Calls Incl T(ms) Excl T(ms) Excl/calls
PLAY                         119130   2804.330   2804.330      0.024
UPDATE-STRATEGY               60131   4441.429   1637.099      0.027
UPDATE-GRAPH                   1000    231.718    231.718      0.232
GO                             1000   4823.320    147.693      0.148
UPDATE-N-OF-PLAYERS            1000      2.480      2.480      0.002

Sorted by Inclusive Time
GO                             1000   4823.320    147.693      0.148
UPDATE-STRATEGY               60131   4441.429   1637.099      0.027
PLAY                         119130   2804.330   2804.330      0.024
UPDATE-GRAPH                   1000    231.718    231.718      0.232
UPDATE-N-OF-PLAYERS            1000      2.480      2.480      0.002

Sorted by Number of Calls
PLAY                         119130   2804.330   2804.330      0.024
UPDATE-STRATEGY               60131   4441.429   1637.099      0.027
GO                             1000   4823.320    147.693      0.148
UPDATE-GRAPH                   1000    231.718    231.718      0.232
UPDATE-N-OF-PLAYERS            1000      2.480      2.480      0.002
END PROFILING DUMP
```

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

- Simulations spend most of the time executing procedure to play (2804.330 ms) and procedure to update-strategy (1637.099 ms).
- The procedure that is called the greatest number of times is to play, which is called 119130 times. This makes sense, since there were 600 agents in this simulation, *prob-revision* was 0.1, a revision requires a play by the agent and by the opponent he observes, and we ran the model 1000 ticks ($600 \times 0.1 \times 2 \times 1000 = 120000$).
- Our implementation to allow the user to modify the number of agents at runtime hardly takes any computing time (just 2.480 ms).

4.4. 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.

CODE 6. Complete code in the Code tab

```
globals [
```

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

breed [players player]

players-own [
  strategy
  payoff
  other-players
]

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 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]
  ]
  tick
  update-graph
end

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

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

6. 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 Rock-Paper-Scissors game under the "imitate if better" revision protocol with noise? To do that, let us use the same setting as in the previous sections (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.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=1061

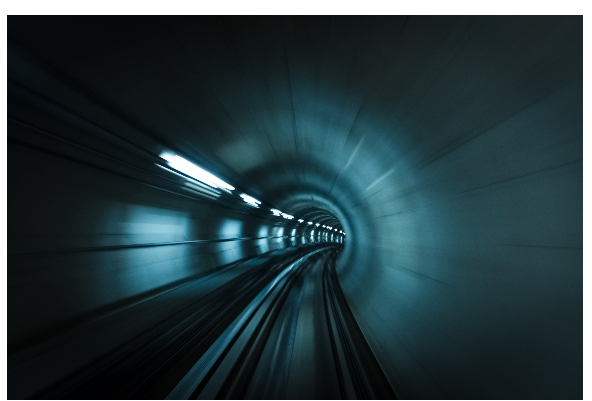
As you can see, when the number of agents is small, the population consistently follows cycles of large amplitude among the three strategies. The cycles are so wide that sometimes one or even two strategies go extinct for a while. In stark contrast, when the population is large, the cycles get much smaller and the population tends to linger around the state where each strategy is used by approximately a third of the population.[2]

7. Exercises

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

2. 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 1.2.2.

CODE Exercise 1. In this section we have improved both the interactivity and the efficiency of our model. Can you quantify how much faster the current version of the code runs compared to the previous one? For the sake of concreteness, use 1000-tick simulations with 600 agents and *prob-revision* 0.1.



Picture by Romain Peli

CODE Exercise 2. In this section we have reduced the number of times procedure to play is called (as long as *prob-revision* is less than 0.5). To illustrate this, compare the number of times this procedure is called in a 1000-tick simulation with 600 agents and *prob-revision* 0.1, before and after our efficiency improvement. Can you compute the number of times procedure to play is called in the general case?

CODE Exercise 3. In this section 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.

CODE Exercise 4. In this section we have reduced the number of times procedure to play is called (as long as *prob-revision* is less than 0.5). However, it is still possible that some players will execute procedure to play more than once in the same tick, specially if *prob-revision* is high. Can you think of a way to reduce the number of calls to procedure to play even further?

1.4. 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 a) running many simulations –i.e. sampling the model many times– and, with the data thus obtained, b) trying to infer general patterns and properties of the model. In terms of types of reasoning, this approach involves logical deduction first (when the data is generated by the computer following the algorithmic rules that define the model) and induction later (when general patterns are inferred from the generated data).
- 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 by no means incompatible; they are complementary in the sense that they can provide fundamentally different insights on the same model and, furthermore, there are plenty of synergies to be exploited by using the two approaches together (Izquierdo et al., 2013).

Here we provide several references to material that is helpful to analyze the agent-based models we have developed in this chapter 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 [[3 0][0 2]] and played by 1000 agents who revise their strategy with probability 0.01 in every tick (following the imitate-if-better rule without noise). 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 0-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.

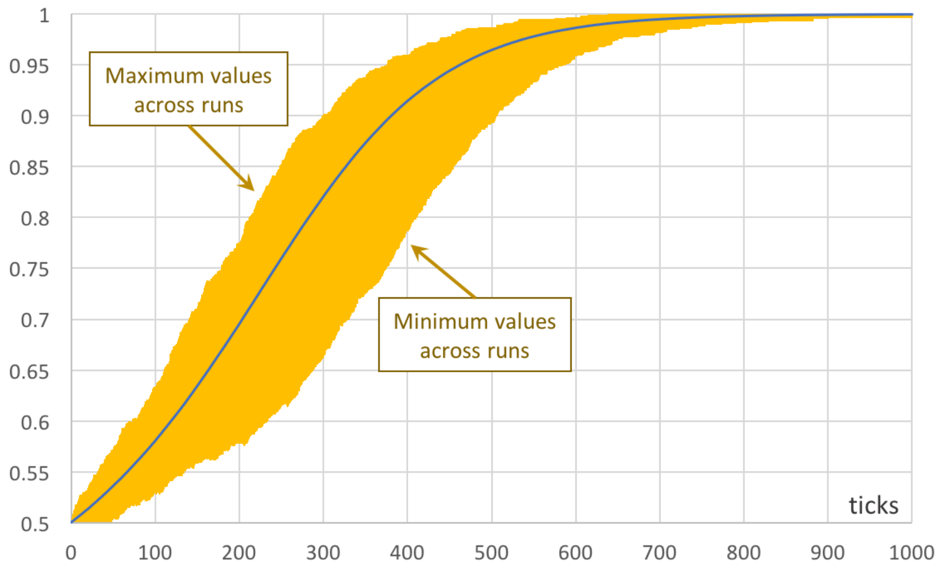


Figure 1. Average proportion of 0-strategists in an experiment of 1000 simulation runs. Orange error bars show the minimum and maximum values observed across the 1000 runs. Payoffs: [[3 0][0 2]]; prob-revision: 0.01; noise 0; initial conditions [500 500].

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.

Experiment

Experiment name coordination-game

Vary variables as follows (note brackets and quotation marks):

```
["payoffs" "[[3 0][0 2]]"]
["n-of-players-for-each-strategy" "[500 500]"]
["prob-revision" 0.01]
["n-of-players" 1000]
["noise" 0]
```

Either list values to use, for example:
["my-slider" 1 2 7 8]
or specify start, increment, and end, for example:
["my-slider" [0 1 10]] (note additional brackets)
to go from 0, 1 at a time, to 10.
You may also vary max-pxcor, min-pxcor, max-pycor, min-pycor, random-seed.

Repetitions 1000

run each combination this many times

☑ Run combinations in sequential order

For example, having ["var" 1 2 3] with 2 repetitions, the experiments' "var" values will be:
sequential order: 1, 1, 2, 2, 3, 3
alternating order: 1, 2, 3, 1, 2, 3

Measure runs using these reporters:

```
count players with [strategy = 0] / n-of-players
```

one reporter per line; you may not split a reporter
across multiple lines

☑ Measure runs at every step

if unchecked, runs are measured only when they are over

Setup commands: setup

Go commands: go

Stop condition:
the run stops if this reporter becomes true

Final commands:
run at the end of each run

Time limit 1000

stop after this many steps (0 = no limit)

Cancel OK

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 0-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 0-strategists in an experiment of 1000 simulation runs. Orange error bars show the standard error. Payoffs: [[3 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.

The models developed in this chapter can be seen as time-homogeneous Markov chains on the finite

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.

space of possible strategy distributions. This means that the number of agents that are following 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 matrix of the Markov chain and operating with it. The example in section 2 above is a case in point. In that coordination game, the number of possible states is just $\binom{N+s-1}{s-1} = \binom{1000+2-1}{2-1} = 1001$, and transition probabilities are not hard to compute (with the help of a computer).

However, most often a full Markov analysis is unfeasible due to one or more of the following –somewhat related– reasons:

- The number of states is too large.[3]
- The transition probabilities are hard to derive.
- The model has too many parameters.

In these common cases where we cannot operate directly with the transition matrix of the Markov chain, can we still say something about the model using mathematical approaches? Certainly so! Even when a full analysis of the Markov chain is not viable, there are often many insights to be gained by focusing on specific aspects of the model. Below we summarize some of the mathematical approaches that can often be conducted.

3.1. Partition of the state space into its closed communicating classes (and all the rest)

In many cases where the transition matrix cannot be easily derived or it is unfeasible to operate with it, one can still partition the state space as the union of all *closed* communicating classes plus another class containing all the states that belong to non-closed communicating classes.[4] This partition can uncover many properties of the Markov chain even without knowing the exact values of its transition matrix, and these properties can yield neat insights about the dynamics of the associated model. You can find examples of application of this technique to several well-known agent-based models in the social simulation literature in Izquierdo et al. (2009).

As an illustration, consider the coordination game example in section 2. The partition of the state space of this model without noise reveals that simulations will necessarily end up in one of two

---

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 in our model is $\binom{1000+4-1}{4-1} = 167668501.$
4. This partition is unique (see decomposition theorem by Chung (1960)).

possible absorbing states: the state where everyone chooses strategy 0, or the state where everyone chooses strategy 1. The probability of reaching one absorbing state or the other depends on initial conditions and can be numerically approximated to any degree of accuracy. In stark contrast, if there is noise, then it is possible to go from any state to any other state, so the whole state space constitutes one single communicating class (and we say that the Markov chain is irreducible). This implies that the probability of finding the system in each of its states in the long run is strictly positive and independent of the initial conditions. It also implies that the limiting distribution coincides with the occupancy distribution. Again, this distribution can be numerically approximated to any degree of accuracy.

3.2. 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 process over finite time spans. The mean dynamic of a Markov chain is derived by taking the limit when the population size goes to infinity, so it characterizes the dynamics of the model over moderate time spans, as long as the population is large enough. For an intuitive introduction to the mean dynamics, see Izquierdo and Izquierdo (2013), who also provide a detailed analysis of the Hawk-Dove game with the imitate-if-better rule. For a concise and rigorous presentation of the most relevant results on Markov chains and on their convergence to the mean dynamics in the context of evolutionary game theory, see Sandholm (2010, chapter 10).[5]

As an illustration of the usefulness of the mean dynamic to approximate transient dynamics, consider the coordination game example in section 2. The mean dynamic for this model reads:

$$\dot{x} = x(x-1)\left(x^2 - 3x + 1\right)$$

where x stands for the fraction of 0-strategists.[6] The solution of this ordinary differential equation with initial condition $x_0 = 0.5$ is shown in figure 4 below. It is clear that the mean dynamic provides a remarkably good approximation to the average transient dynamics plotted in figures 1 and 3.

5. For details, see Izquierdo and Izquierdo (2013).
6. One unit of clock time is defined in such a way that each player expects to receive one revision opportunity per unit of clock time. In this particular example, since *prob-revision* = 0.01, one unit of clock time corresponds to 100 ticks (i.e. 1 / *prob-revision*).

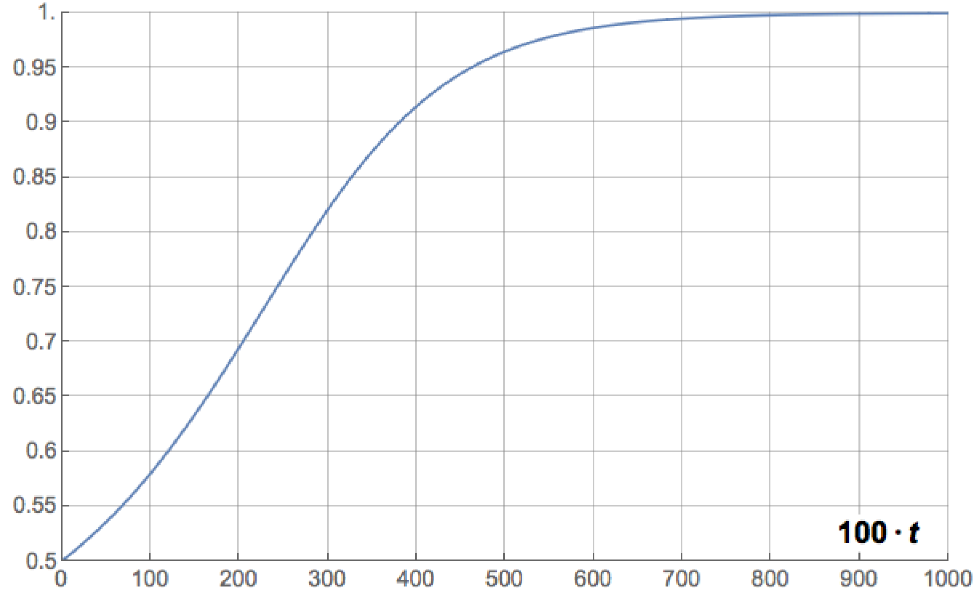


Figure 4. Trajectory of the mean dynamic of the example in section 2, showing the proportion of 0-strategists as a function of time (rescaled to match figures 1 and 3).

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 5 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}][t/100]
  , {x0, 0, 1, 0.01}]
 ], {t, 0, 1000}]
```

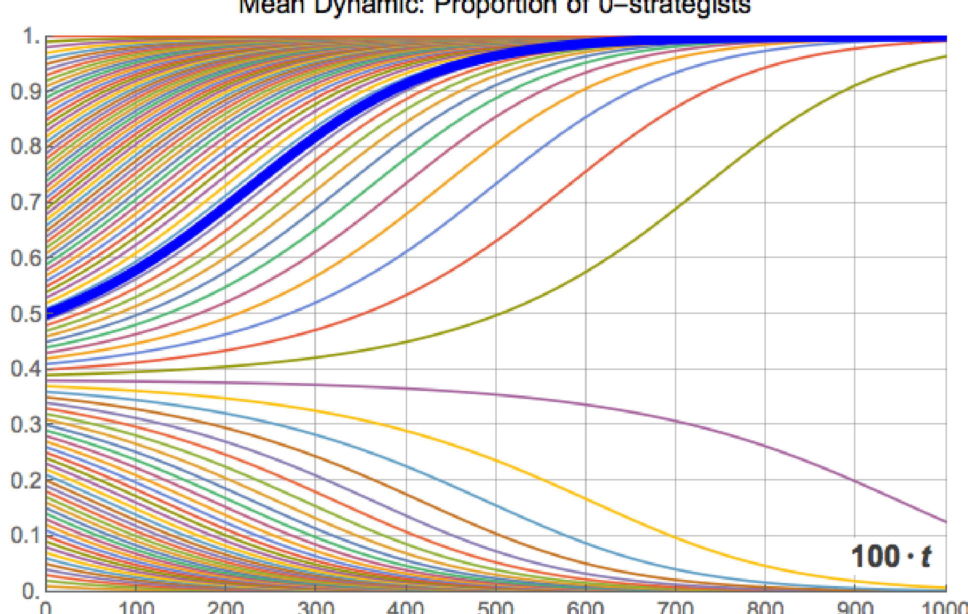


Figure 5. Trajectories of the mean dynamic of the example in section 2, showing the proportion of 0-strategists as a function of time (rescaled to match figures 1 and 3) for different initial conditions.

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. To solve the mean dynamic of the imitate-if-better rule in 3-strategy games, you may want to use this demonstration.

3.3. 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. Thus, at such equilibria, the expected flows of agents switching between different strategies cancel one another out but, naturally, agents keep revising and changing strategies, potentially in a stochastic fashion.

As an example, consider a Hawk-Dove game with payoffs [[0 3][1 2]] and the imitate-if-better revision rule without noise. The mean dynamic of this model is:

$$\dot{x} = x(1 - x)(1 - 2x)$$

where x stands for the fraction of 0-strategists, i.e. "Hawk" agents.[7] 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

7. For details, see Izquierdo and Izquierdo (2013).

“Dove”, and stay around there for long. Figure 6 below shows several trajectories for different initial conditions.

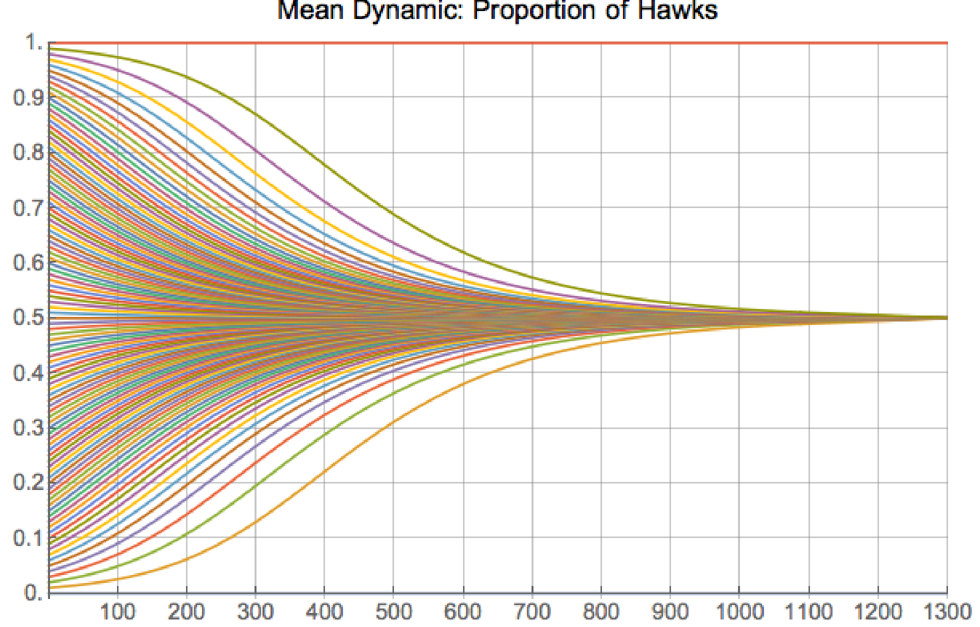


Figure 6. Trajectories of the mean dynamic of an imitate-if-better Hawk-Dove game, showing the proportion of “Hawks” as a function of time for different initial conditions.

Naturally, simulations do not get stuck in the half-and-half state, since agents keep revising their strategy in a stochastic fashion (see figure 7). 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 revision protocol and in the matching process imply that the strategy distribution is in perpetual change. In the simulation shown in figure 7, 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 $\frac{1}{\sqrt{10}}$.[8]

8. A good approximation for these standard deviations over long time spans when the number of agents is large can be exactly computed (see Izquierdo et al. (2018a, example 3.1)).

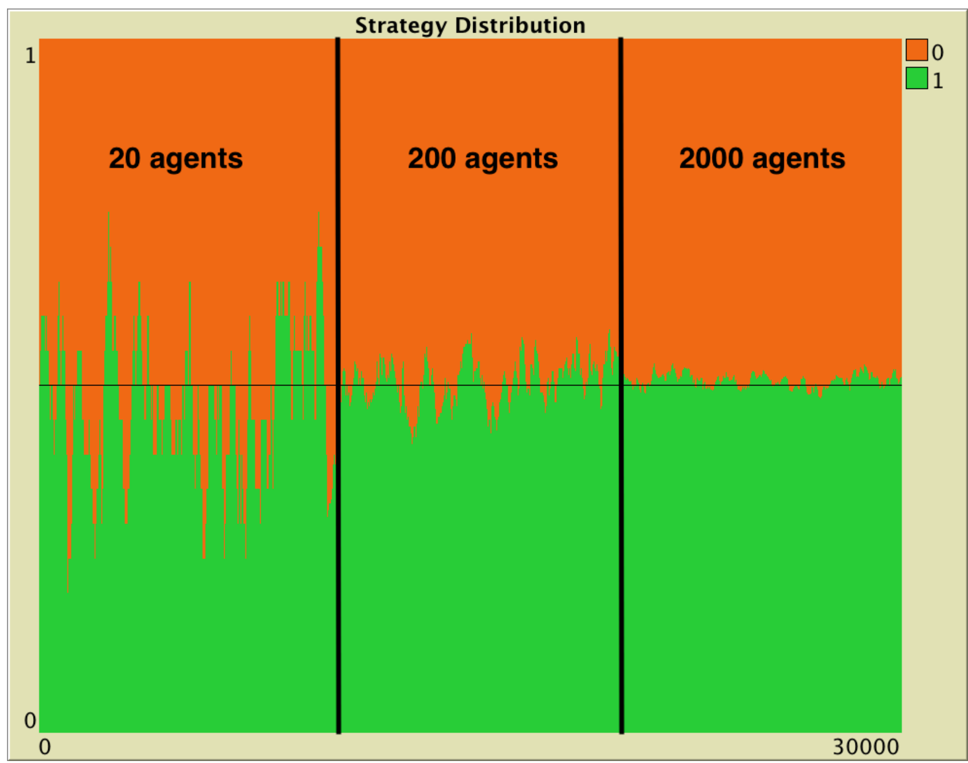


Figure 7. A simulation run of an imitate-if-better Hawk-Dove game, set up with 20 agents during the first 10000 ticks, then 200 agents during the following 10000 ticks, and finally 2000 agents during the last 10000 ticks. Payoffs: [[0 3][1 2]]; prob-revision: 0.01; noise 0; initial conditions [10 10].

3.4. Stochastic stability analyses

In the last model we have implemented in this unit, recall that when *noise* is strictly positive, it is possible to go from any state to any other state, so 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 dynamics of the Markov chain are often concentrated in just a few states. Stochastic stability analyses are devoted to identifying such states, which are often called *stochastically stable states*, and are a subset of the absorbing states of the process without noise.

To learn about this type of analysis, the following references are particularly useful: Vega-Redondo (2003, section 12.6), Fudenberg and Imhof (2008), Sandholm (2010, chapters 11 and 12) and Wallace and Young (2015).

To illustrate the applicability of stochastic stability analyses, consider our imitate-if-better model where agents play a Prisoner's Dilemma with payoffs [[2 4][1 3]] and strictly positive *noise*. It can be proved that the only stochastically stable state in this model is the state where everyone defects.[9]

9. The proof can be conducted using the concepts and theorems put forward by Ellison (2000). Note that the radius of the state

This means that, as *noise* tends to 0, the infinite-horizon dynamics of the model will concentrate on that single state.

4. Exercises

Exercise 1. Derive the mean dynamic of a Prisoner's Dilemma game for the imitate-if-better protocol.

Exercise 2. Derive the mean dynamic of the coordination game discussed in section 0.1 (with payoffs [[1 0][0 2]]) for the imitative pairwise-difference protocol.

Exercise 3. Derive the mean dynamic of the coordination game discussed in section 0.1 (with payoffs [[1 0][0 2]]) for the best experienced payoff protocol.

where everyone defects 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 cooperates to the state where everyone defects).

2. SPATIAL INTERACTIONS ON A GRID

2.0. Spatial chaos in the Prisoner's Dilemma

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]

- 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. (2007 a,b).

More generally, populations where some pairs of agents are more likely to interact with each other than others are called *structured* populations. If, on the other hand, all members of a population are equally likely to interact with each other, the population is said to be *well-mixed*. Generally, the dynamics of an evolutionary process in a well-mixed population can be very different from that in a structured population. In social dilemmas in particular, population structure plays a crucial role (Gotts et al. (2003), Hauert (2002, 2006), Roca et al. (2009a, 2009b)).[2]

2. Motivation. Cooperation in spatial settings

In the previous chapter, we saw that if agents play the Prisoner's Dilemma in a well-mixed population, 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 section 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 dimensions, can be set by the user. Each patch has eight neighboring patches (i.e. the eight cells

1. Note that in most evolutionary models there are two types of neighborhoods for each individual agent A:
2. Christoph Hauert has an excellent collection of interactive tutorials on this topic at his site EvoLudo (Hauert 2018).

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 *symultaneously* revise their strategy according to the following rule:

> 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.

CODE 4. Interface design

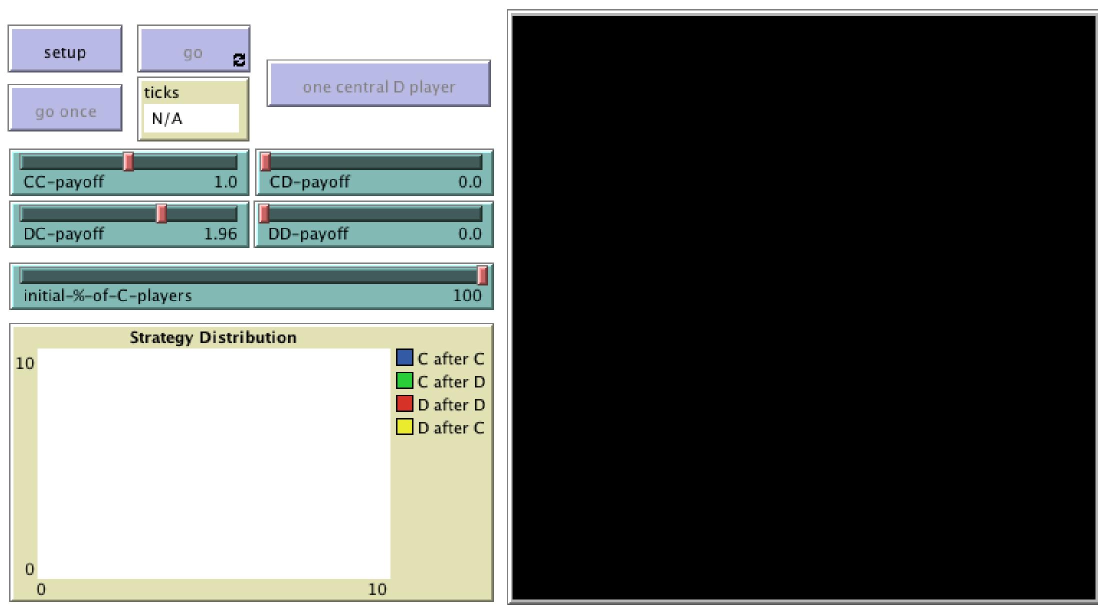


Figure 1. Interface design.

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 the previous chapter.

Note that the interface in figure 1 has an extra button labeled one central D player. 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

4.1. 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 was a C-player or a D-player before it revised and updated its strategy. This is needed because we want to model synchronous updating, i.e. we want all agents to change their strategy at the same time. To do this, we need to keep in memory the strategy of every agent before the first agent switches strategy. For this purpose, we may use the boolean variable C-player-last?.

- 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-last?
  payoff
  my-nbrs-and-me
  n-of-my-nbrs-and-me
]
```

4.2. 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,[3] 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.

3. 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 patches [
  set payoff 0
  set C-player? false
  set C-player-last? 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, 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-last? true
   ]
   ```

4. Color patches according to the four possible combinations of values of C-player? and C-player-last?. 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?
     [
       ifelse C-player-last?
       [set pcolor blue]
       [set pcolor lime]
     ]
     [
       ifelse C-player-last?
       [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?
     [ifelse-value C-player-last? [blue] [lime]]
     [ifelse-value C-player-last? [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-last?
  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-last? 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-last? true
    ]
end

to go

end

to update-color
  set pcolor
    ifelse-value C-player?
     [ifelse-value C-player-last? [blue] [lime]]
     [ifelse-value C-player-last? [yellow] [red]]
end
```

4.3. 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 store the current value of C-player? (i.e. the player's current strategy) in the variable C-player-last?. In this way, the variable C-player-last? will keep the strategy with which the current payoff has been obtained, and we can update the value of C-player? without losing that information, which will be required by neighboring players when they update their strategy.
3. To update its strategy (i.e. the value of C-player?). To keep our code nice and modular, we will do this in a separate new procedure called to update-strategy.
4. To update its colour according to their new C-player? and C-player-last? values.

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:[4]

```
to go
  ask patches [play]
```

4. The command `tick` could just as well be included at the very end of procedure to go. We tend to like using `tick` to separate the main logic of the round from the instructions that deal with plotting and visualization, but this is just a matter of preference.

```
  ask patches [set C-player-last? C-player?]
  ask patches [update-strategy]
  tick
  ask patches [update-color]
end
```

4.4 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 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:

```
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

In this procedure, which will be run by individual patches, we want the patch to adopt 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-last?. With this, we conclude the code of procedure to update-strategy.

```
to update-strategy
  set C-player? [C-player-last?] of one-of my-nbrs-and-me with-max [payoff]
end
```

4.5. Complete code in the Code tab

The Code tab is ready!

```
patches-own [
  C-player?
  C-player-last?
  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-last? 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-last? true
  ]
```

```
end

to go
  ask patches [play]
  ask patches [set C-player-last? C-player?]
  ask patches [update-strategy]
  tick
  ask patches [update-color]
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
  set C-player? [C-player-last?] of one-of (my-nbrs-and-me with-max [payoff])
end

to update-color
  set pcolor
    ifelse-value C-player?
     [ifelse-value C-player-last? [blue] [lime]]
     [ifelse-value C-player-last? [yellow] [red]]
end
```

4.6. Code in the plots

We will use the following color code for the patches:

- Blue if it is occupied by a C-player and in the previous period it was also occupied by a C-player.
- Green if it is occupied by a C-player and in the previous period it was occupied by a D-player.
- Red if it is occupied by a D-player and in the previous period it was occupied by a D-player.
- Yellow if it is occupied by a D-player and in the previous period it was occupied by a C-player.

To complete the Interface tab, edit the graph and create the pens as in the image below:

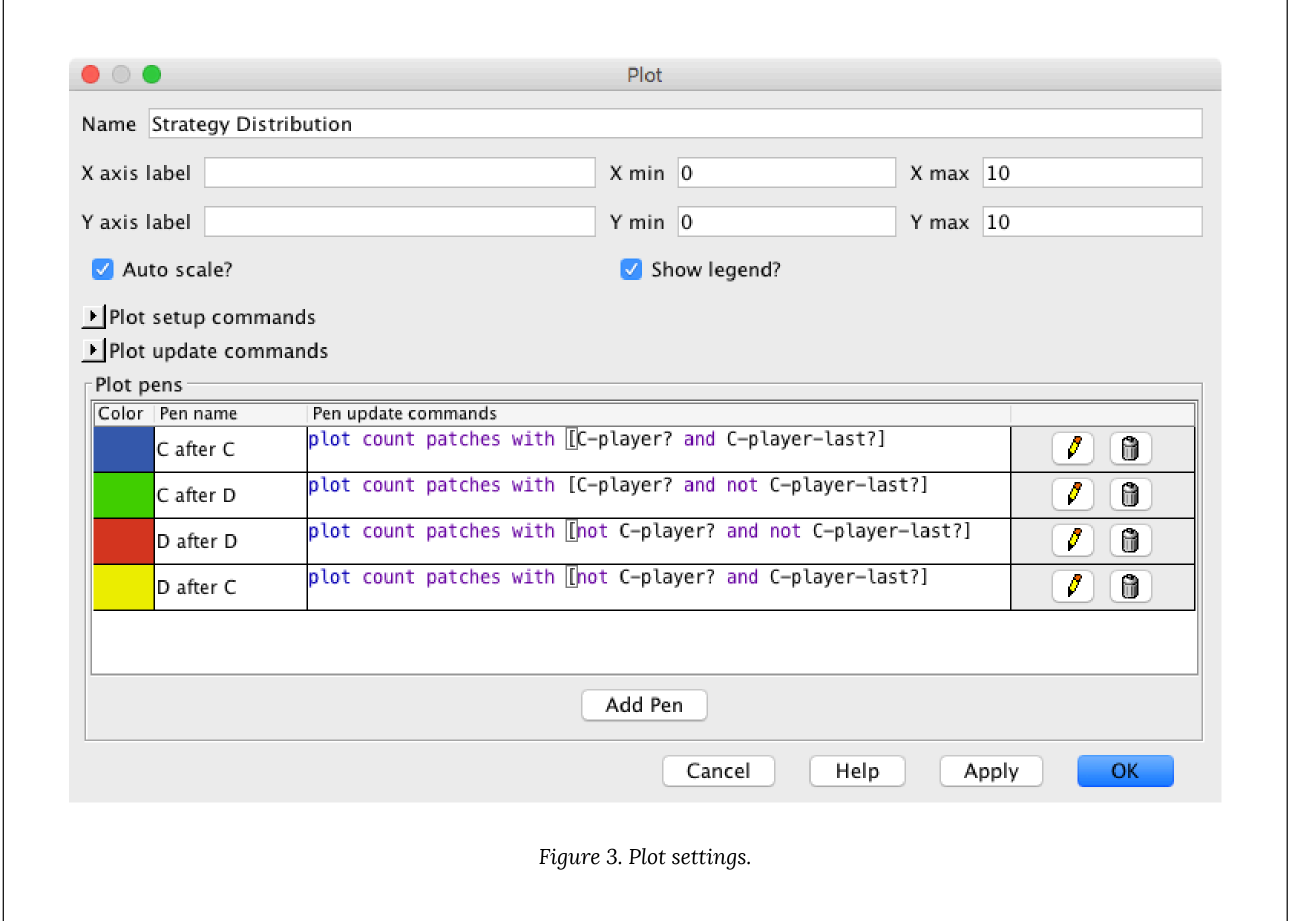


Figure 3. Plot settings.

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.[5] An illustration of the sort of patterns that this model generates is shown in the video below.

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=108

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).

5. Some authors make *CD-payoff* = *DD-payoff*, so they can parameterize the Prisoner's Dilemma with just one parameter, i.e. *DC-payoff*. Note, however, that this reduces the range of possibilities that can be studied.

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. 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:

A video element has been excluded from this version of the text. You can watch it online here: https://wisc.pb.unizin.org/agent-based-evolutionary-game-dynamics/?p=108

7. Exercises

You can use the following link to download the complete NetLogo model: 2×2-imitate-best-nbr.

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.

CODE Exercise 2. Create a button to turn the central patch into a D-player. You may want to label it one central D player. 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).

CODE 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).

CODE 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).

CODE 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, ɸ, of cheating by switching to defection.

You may wish to rerun the sample run above with a small value for ɸ. You may also want to replicate the experiment shown in Mukherji et al. (1996, fig. 1).

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

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

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., 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

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.

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*, pages 15–32. Springer Berlin Heidelberg. https://doi.org/10.1080/00222500590921283

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.

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.

Gilbert, N. (2007). *Agent-Based Models*, volume 153 of *Quantitative Applications in the Social Sciences*. Sage Publications, London.

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.

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/

Head, B. (2018). NetLogo Python extension. *Software*. https://github.com/qiemem/PythonExtension.

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

Hofbauer, J. (1995). Imitation dynamics for games. Unpublished manuscript, University of Vienna.

Hofbauer, J. and Sigmund, K. (1988). *Theory of Evolution and Dynamical Systems*. Cambridge University Press, Cambridge.

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

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., and Sandholm, W. H. (2018a). *An introduction to ABED: Agent-based simulation of evolutionary game dynamics*.

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, 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

Izquierdo, S. S., Sandholm, W. H., and Izquierdo, L. R. (2018b). Best experienced payoff dynamics. Unpublished manuscript, Universidad de Valladolid, University of Wisconsin, and Universidad de Burgos.

Janssen, M.A. (2010). *Introduction to Agent-Based Modeling*. openABM. https://www.openabm.org/book/introduction-agent-based-modeling

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

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

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

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)*.

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

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

Mukherji, A., Rajan, V., and Slagle, J. R. (1996). Robustness of cooperation. *Nature*, 379:125–126. http://dx.doi.org/10.1038/379125b0

Myerson, R. B. (1997). *Game theory: Analysis of Conflict*. Harvard University Press.

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.

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., Bonhoeer, S., and May, R. M. (1994). More spatial games. *International Journal of Bifurcation and Chaos*, 4:33–56. https://doi.org/10.1142/S0218127494000046

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

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. (2011). *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton University Press, Princeton, NJ. 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

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

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. (2009). Evolutionary game theory. In Meyers, R. A., editor, *Encyclopedia of Complexity and Systems Science*, pages 3176–3205. Springer, Heidelberg.

Sandholm, W. H. (2010). *Population Games and Evolutionary Dynamics*. MIT Press, Cambridge.

Sandholm, W. H., Izquierdo, S. S., and Izquierdo, L. R. (2017). Best experienced payoff dynamics and cooperation in the Centipede game. Unpublished manuscript, University of Wisconsin, Universidad de Valladolid, and Universidad de Burgos.

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

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

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

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

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

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.

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

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. (2017). *The NetLogo User Manual*. Version 6.0.2. http://ccl.northwestern.edu/netlogo/6.0.2/docs/

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

Wolfram Research, Inc. (2019). Mathematica. *Software*. Champaign, Illinois. https://www.wolfram.com

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.