CoopSim -
A Computer Simulation of the Evolution of Cooperation
User's Manual

Eckhart Arnold

October 14, 2006

Contents

1 Introduction

CoopSim is a computer simulation of the reiterated prisoner's dilemma. The reiterated prisoner's dilemma is a model for many (but not for all) cooperation dilemmas discussed in social sciences. It is also useful as a model for the evolution of altruistic behaviour in biology. The computer program *CoopSim* has been written for use in an undergraduate course on the "Evolution of Cooperation". Its purpose is mainly educational.

The reiterated prisoner's dilemma and its simulation on the computer is discussed in a very understandable form in the book "The Evolution of Cooperation" by Robert Axelrod. Therefore, it will not be explained here any more (see section "Further Reading" for some recommendable books on the topic). A basic knowledge of what the *prisoner's dilemma* is and what it has got to do with altruism and cooperation is presupposed in the following. The program *CoopSim* is largely based on the description of Axelrod's book. However, the whole program has been written from scratch without taking recourse to Axelrod's original Fortran program. I did so, because I wanted the program code to be readable enough so that students with some

programming knowledge might be encouraged to extend the program and to implement strategies of their own. Also, I wanted the simulation to have a nice user interface so that I could do "life" simulations under different boundary conditions in class.

2   Aknowledgements

CoopSim is based on the description of a computer tournament in Robert Axelrod's book "The Evolution of Cooperation". The nomenclature ("computer tournament", "ecological simulation") has been taken from this book. Some of the strategies built into *CoopSim* are (sometimes only loosely) based on strategies with the same name described in Axelrods book.

I would also like to thank the following people for contributing strategies of their own: Alex Mainzer, Björn van den Bruck, Christian Erlen, Stefan Pennartz, Sven Sommer, Paul Boehm.

Finally, I would like to thank the initiators, makers and contributers to the *Python* programming language (www.python.org `www.python.org`) and the *wxWidgets* GUI-Toolkit (www.wxwidgets.org `www.wxwidgets.org`; www.wxpython.org `www.wxpython.org`). *Python* and *wxWidgets* are open source software packages without which the making of *CoopSim* would not have been possible in this form.

3   License and Disclaimer

```
This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

The full text of the GPL License is contained in the file
"LICENSE.GPL" in the distribution directory of this program.
If for some reason this file should be missing, you can still get
the full text of the GPL License under
```

```
  http://www.gnu.org/copyleft/gpl.html
```

```
or write to
```

```
  Free Software Foundation, Inc.
  59 Temple Place - Suite 330
  Boston, MA 02111-1307 USA
```

4 Installation

4.1 System Requirements

Hardware Requirements:

- AMD Athlon or Pentium III System, 1 Ghz or above (otherwise *Coop-Sim* might run pretty slow)

- 128 MB or more of memory

Software Requirements:

- Linux or Windows 98/XP Operating System

- Python 2.3 or above (`www.python.org`)

- wxPython 2.4 or above (`www.wxpython.org`)

4.2 Installing CoopSim

To install and successfully run *CoopSim* you need to have Python version 2.3 or above and wxPython version 2.4 or above installed on your System. You can download the installation packages from the websites mentioned above. *CoopSim* runs under Windows, Linux and potentially under MacOS as well, but it has only been tested under Windows and Linux. In order to install *CoopSim* you only need to unpack the zip archive "CoopSim.zip" anywhere on your hard disk.

4.3 Running "CoopSim"

To run *CoopSim*, you have to start the executable python file "CoopSim.py" in the main installation directory either by double clicking or by changing to the CoopSim directory and entering "python CoopSim.py" at the command line.

5 First Steps - A guided tour through *Coop-Sim*

5.1 Starting a predefined simulation

After successfully starting CoopSim you should see an application Window like this:
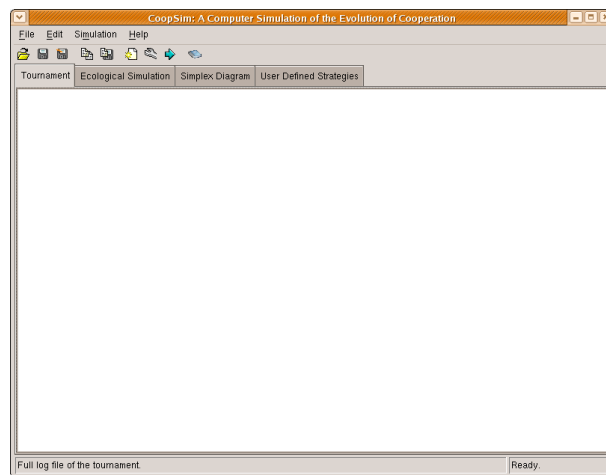
In the beginning the screen is empty, of course. You will notice four notebook pages named: *Tournament*, *Ecological Simulation*, *Simplex Diagram* and *User Defined Strategies*. The purpose of these pages will be explained later. First, we will just start the simulation *Simple Example*. This simulation is preselected when the program starts up (you can see which simulation is active and which simulations are available in the *Simulation* menu), so it is just enough to click the *Continue Simulation* button to start the simulation. This is the button with the blue arrow pointing to the right. It is the second button from the left in the button row under the menu. After a short time of calculating the result of the tournament should appear on the *Tournament* page:
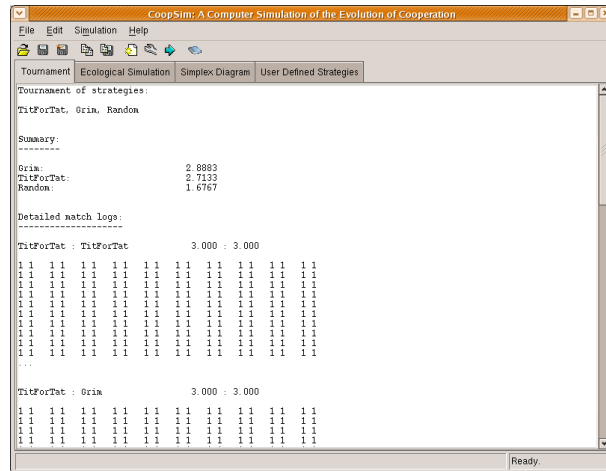
The tournament page displays the following information:

1. The strategies that took part in the tournament.

2. Then ranking of the strategies.

3. The outcome of each single match of a pair of strategies as well as the first and last 50 moves of the players, where **0** indicates a defection and a **1** a cooperative move.

4. The ranking of the strategies in the ecological simulation after a certain number of generations.

You can scroll the contents of the text window using the slider on the right side in order to see all the information. As you can see from the tournament log, the simulation *Simple Example* is a tournament of three strategies only: GRIM, TITFORTAT and RANDOM. Among these three strategies GRIM emerges as the clear winner.

Now, let us have a look at the other notebook pages. Select the page *Ecological Simulation* in order to view the graph of the ecological simulation. The graph shows the population dynamics of the strategies in the tournament, assuming that the fitness of a strategy is determined by its score in the tournament. It should look like this:
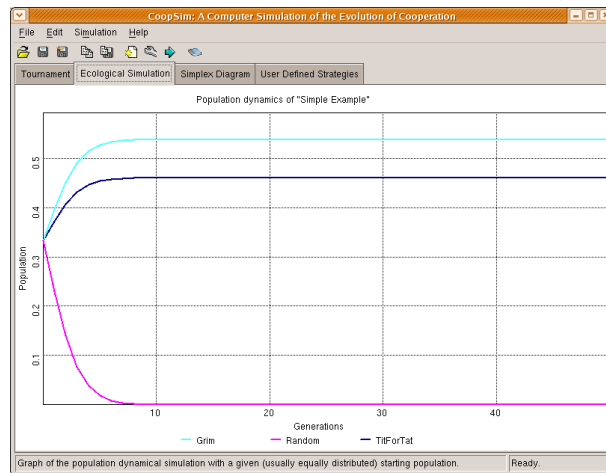
If you wonder how the development is going to continue after the 50th generation (you probably don't when the graph is as simple as in this case, but sometimes it takes more generations until a clear result crystalizes out), you can click on the blue arrow button again to let the simulation continue. You can do this as often as you like. (To go back to the first 50 generations, just restart the simulation from the *Simulation* menu.)

Now suppose you would like to save the graph (maybe, because you are just about to write a paper on cooperation in the reiterated prisoner's dilemma, for which some graphical illustrations might be useful). You can do so by selecting *Save Page As* from the *Edit* menu or by clicking the *Save Page* button in the toolbar (somewhere in the middle, not to be confused with the *Save* button which saves the whole setup of simulations!). A dialog box will appear where you can the select a directory and enter a filename for the graph:
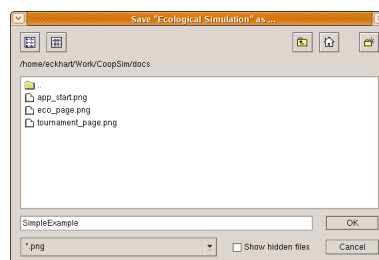
This does not only work with the *Ecological Simulation* page but with all other pages as well. So, whenever you want to save some content you are seeing on the screen, just select *Save Page As* in the *Edit* menu and you will be prompted to save the content of the currently selected page. Alternatively, you can select *Copy Page* in the *Edit* menu (or click the *Copy Page* button

in the toolbar) to copy the content of the selected page to the clipboard. You can then easily insert the content into another application, say a word processor, by pressing *Crtl-V* within this application.

Apart from the graph of the ecological simulation, which usually starts with a uniformly distributed population, you may also want to know how the three strategies fare in the ecological simulation when they are given different population shares in the beginning. For three strategies the population dynamics can be visualized as a simplex diagram. (If there are more than three strategies in the simulation then the three strongest strategies are depicted.) On a simplex diagram each point within the simplex represents a certain population distribution. The visualization of the population dynamics looks very similar to a vector field in physics. But, bear in mind that population dynamics is a discrete process, while vector fields in physics are usually continuous. (For a more comprehensive explanation of simplex diagrams, look into the literature on evolutionary game theory, as for example Maynard-Smith's "Evolution and the Theory of Games"). If you select the *Simplex Diagram* page you should see a picture like this:
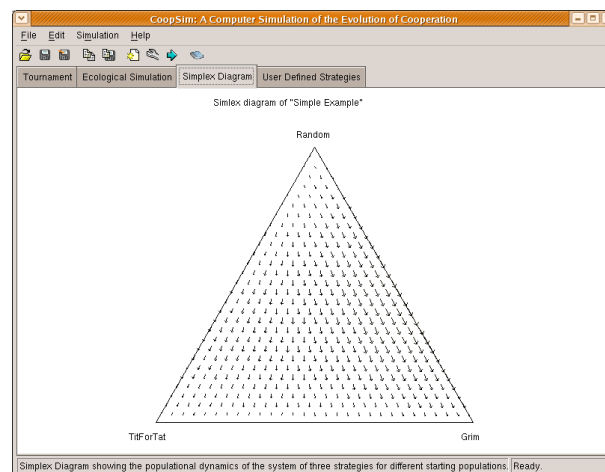

The arrows indicate the direction of the "field" (i.e. the direction the population drifts to at a certain point). The length of the arrows indicates the strength of the drift. Big arrows mean a strong drift while small arrows indicate an only slow population drift.

5.2 Defining a new simulation

Now, you might think, what would happen if we add another strategy to our tournament? Nothing is easier than that! In order to do so, we have to create a new simulation setup. This can be done by selecting *New Simulation* in the

Simulation menu or by clicking the *New Simulation* button in the toolbar (the button before the one with the wrench symbol). Selecting *New Simulation* opens a dialog where you can define a new simulation setup based on the current simulation setup. The dialog looks like this:
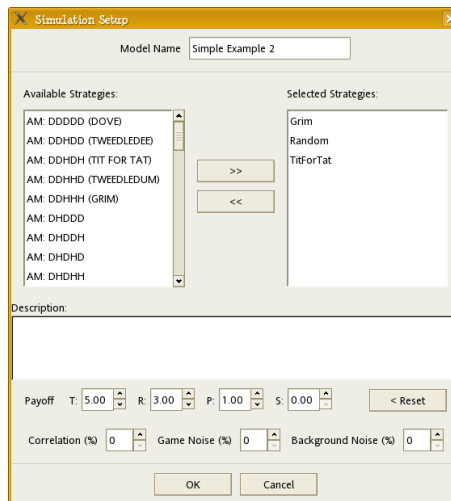
The dialog allows you to select the strategies you want to let take part in the tournament. You can see the list of available strategies on the left hand side. On the right hand side you can see the strategies that will play in the tournament.

Let us try to add the strategy TESTER to our tournament. In order to do so, select TESTER with a mouse click in the list box on the left hand side. (Hint: You may have to scroll the list down a little bit, before TESTER appears.) Notice that when you select a strategy, an explanation of the strategy appears in the text box below. When you have selected TESTER, click on the button ">>" in the middle of the dialog box. This should transfer TESTER from the list of available strategies to the list of selected strategies.
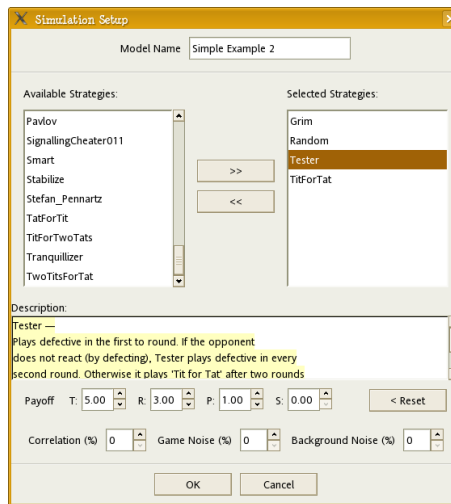
Finally you should enter a name for your new simulation setup in the text
entry widget at the top of the dialog box. Take "New Example", if you do
not know what to enter. Now click *O.K.* to see what happens. Well, since
now there are four strategies in the tournament, there won't be a simplex
diagram any more. Select the page *Ecological Simulation* in order to see what
has changed. Obviously, this time TITFORTAT has come out at the top of
the strategies. Although TESTER has not been very successful itself, it has
changed the overall results dramatically.

5.3 Programming a new strategy

The last step of this introductory walkthrough will show you how to add your
own strategies to *CoopSim*, if you know the basics of the *Python* programming
language. For the sake of brevity, I will only explain how to activate a custom
strategy that has already been coded but deactivated by comment signs. In
order to program your own strategy you have to change to the *User Defined*
Strategies page in the main window. The page contains a simple text editor,
where you can enter python program code:

As you can see, there is already quite a bit of program code there. This program code has been inserted for explanatory purposes. Every line starts with a "#", which is a token for the python interpreter to treat the line as a comment and not to execute it. Try to locate the class definition of *class LesserTFT*. Reactivate this strategy by removing the comment sign in front of the class definition as well as the following single space character (this is important) so that the statement "class LesserTFT(Strategy):" starts in the very first column. Do the same for the following lines up to the line where the class LesserTFT is instantiated. This is line that reads "lesser_tft = LesserTFT()". Remember that indentation has syntactical significance in *Python*, so beware of inadvertently changing the indentation while removing the comment signs and the following space character.

Now select *Edit Simulation* from the *Simulation* menu (or click on the wrench symbol in the toolbar). If no errors have occurred in the program code

you will now find a new strategy *LesserTFT* among the available strategies in the simulation setup dialog box. If you receive an error message, you may have forgotten to delete a single trailing space after the comment sign "#" in one or more lines.

6   Comprehensive Overview

This Chapter of the manual gives a comprehensive overview over all menu commands of the *CoopSim* application.

6.1   File Menu

6.1.1   New

Resets the application to its startup state. This means that all simulation setups and custom strategies will be deleted. Usually there is not much need to select *New* at all, except when you get lost.

6.1.2   Open

Opens a previously saved state, including all simulation setups as well as any user defined strategies or setups. All previous simulation setups will be lost. (There is - at the moment - no way of merging different simulation setups.)

6.1.3   Save

Saves the full application state in a file, including all simulation setups and and any program code that has been entered in the *User Defined Strategies* page.

Warning: The save files may not be interchangeable between different versions (including subversions) of CoopSim! Loading a save file from a different version of CoopSim may cause CoopSim to crash!

If you want to save program code from the *User Defined Strategies* page, the best idea is to copy and paste it to a text editor and save it as a text file.

6.1.4   Save As

Promts for a filename and then saves the full application state in a file, including all simulation setups and and any program code that has been entered on the *User Defined Strategies* page.

Warning: The save files may not be interchangeable between different versions (including subversions) of CoopSim! Loading a save file from a different version of CoopSim may cause CoopSim to crash!

If you want to save program code from the *User Defined Strategies* page, the best idea is to copy and paste it to a text editor and save it as a text file.

6.1.5 Exit

Quits the application.

6.2 Edit Menu

As you may have noticed, there is no "Paste" entry in the *Edit* menu. If you want to add some program code from another application, say a python editor, to your own custom strategies on the *User Defined Strategies* page, you can access the usual *cut*, *copy* and *paste* functions on this page via keyboard with *crtl-x*, *crtl-c*, *crtl-v* respectively.

6.2.1 Copy Page

Copies the content of the currently visible page to the clipboard. This can either be an image (pages *Ecological Simulation* and *Simplex Diagram*) or HTML file (page *Tournament*) or plain ASCII text (page *User Defined Strategies*).

6.2.2 Save Page As

Saves the content of the currently visible page to a file. This can either be an image (pages *Ecological Simulation* and *Simplex Diagram*) or plain ASCII text (pages *Tournament* and *User Defined Strategies*). Currently the only supported file format for graphical images is the *png* format.

6.3 Simulation Menu

6.3.1 New Simulation

Invokes the setup dialog for setting up a new simulation. In the setup-dialog the set of strategies that take part in the tournament can be selected, and the payoff parameters T (tempation of cheating), R (reward for cooperation), P (punishment for mutual non cooperation), S (sucker's payoff) and the parameters *Noise*, *Correlation*, *Background noise* can be adjusted. The latter three of these parameters can be assigned percentage values from 0 to 100

percent. *Noise* specifies the in game noise that is a random probability with which the move of a player is turned into its opposite. *Correlation* determines how often players meet with players of their own kind. It ranges from totally random (= 0 %) to the closest possible correlation when only players of the same type meet (= 100 %). (Try a tournament between DOVE and HAWK and change this parameter in steps of 10 % from 0 % to 50 %. What can you observe?) *Background noise* specifies an evolutionary background noise meaning that reproduction does depend on the fitness plus/minus a certain random factor.

6.3.2   Edit Simulation

Same as *New Simulation*, only that the current simulation setup is manipulated instead of creating a new simulation setup. Simulation setups that have been programmed on the *User Defined Strategies* page can not be edited. You will have to create a new simulation setup instead. In this case the parameters *population* and *mutators* will be reset to their default values in order to avoid incongruous setup data. (Setup data would become incongruous if it specifies the mutation of a strategy to another strategy that has manually been removed from the setup).

6.3.3   The *Simulation Setups*

Following in the *Edit* menu is a list of simulation setups. Whenever you define a new simulation setup it appears in this list. The currently selected simulation setup is marked. You can run or restart a simulation by selecting it from this menu.

6.3.4   Remove Models

Opens a dialog that allows you to remove setups that you do not need any more from the *Simulation* menu.

6.4   Help Menu

6.4.1   Help

Shows this manual as HTML text in a browser window.

6.4.2   License

Shows the license agreement for this software. This software is open source under the GNU Public License.

6.4.3 About

Tells you who wrote this program.

7 Advanced Topics

7.1 Technical notes on the dynamical model

Population dynamics crucially depends on two factors: How fitness is transformed into a reproduction rate (assuming that the transformation is always monotonous, this still leaves open quite a number of possibilities), and whether or when species die out. The model used in *CoopSim* uses population shares rather than an integer number of individuals for each species (strategy). This means that species never die out, even though their population share might become arbitrarily small. The reproduction rate is determined by the score a strategy gains in the tournament divided by the average score. Thus the reproduction rate does depend on the payoff parameters. Both assumptions are to a certain degree arbitrary.

Evolutionary noise (which can be adjusted via the *Background noise* parameter in the simulation setup dialog) is modeled as random disturbance on the reproduction rate (not on the population share).

7.2 Programming user strategies

7.2.1 Preface

There are two ways of adding user strategies to *CoopSim*: Either by entering them on the *User Defined Strategies* page when the application is running or by writing them directly to the *Strategies.py* module of the program code. Since *CoopSim* is open source software and thus comes with the complete source code you can easily do so. Except for experimenting, the latter method of adding your own strategies directly to the program code is probably the better one, because entering strategies on the *User Defined Strategies* page is more error prone and can quickly get frustrating. Also, changing the module *Strategies.py* can be done with a real python editor with syntax highlighting, class browser and other comforts. On the other hand, you should only change *Strategies.py* if you really know what you are doing, or otherwise *CoopSim* might not run any more.

7.2.2 The *Strategy* class

Any strategy in the game must be derived from *class Strategies.Strategy*. This class defines the two methods **firstMove**(self) and **nextMove**(self, myMoves, opMoves). Both methods must return either 0 or 1, where 0 means "defection" and 1 stands for "cooperation". *firstMove* does not take any parameters except "self", while *nextMove* gets the list of its own moves during the previous rounds of the current match as well as a list of its opponents previous moves. Both lists are list of zeros and ones. Implementing your own user strategies is pretty easy now: Simply derive a class from *class Strategies.Strategy* and define the methods *firstMove* and *nextMove*, nothing else is needed, neither a constructor, nor is it necessary to worry about a name for the strategy, because the class name is automatically used as the strategies' name. All state saving variables of your class must be reset in method *firstmove*! This is necessary, because the same strategy object is used in all matches of the tournament. Here is a simple example of a custom strategy class:

```
class LesserTFT(Strategy):
    """Retailiate only when not having retailiated in
    the last round already.
    """
    def firstMove(self):
        return 1  # start friendly
    def nextMove(self, myMoves, opMoves):
        if opMoves[-1] == 0 and myMoves[-1] != 0:
            return 0  # retailiate
        else:
            return 1  # cooperate


# Do not forget to instantiate your class, otherwise
# it will not appear among the available strategies!

lesser_tft = LesserTFT()
```

7.3 Defining user setups

There are many more parameters that determine a simulation than can be edited through the simulation setup dialog. To change these parameters, thereby gaining access to a much wider range of possible simulation scenarios, it is necessary to define the simulation setup manually on the *User Defined Strategies* page.

All parameters that determine a simulation's behaviour are defined in an object of *class Simulation.SimSetup.* To define a simulation setup, it suffices to instantiate this class with the needed parameters. Here is an explanation of the parameters the constructor of the SimSetup class takes:

```
name          = string: name of the model
strategyList  = list of Strategy objects: the list of the strategies
population    = tuple: population share for each strategy
correlation   = float [0.0-1.0]: correlation factor
gameNoise     = float [0.0-1.0]: in game noise
noise         = float [0.0-1.0]: evolutionary background noise
iterations    = int:  number of iterations for one match
samples       = int:  number of sample matches to take (only useful for
                  randomizing strategies)
payoff        = tuple of floats:  payoff tuple (T, R, P, S)
demes         = DemeDescriptor: defines the deme structure of the
                  population or 'None' if there is only one deme
mutators      = list of Mutator objects: description of possible
                   mutation (or degeneration resp.) of strategies during
the course of the evolutionary development.
cachedPM      = cached payoff matrix
cachedLog     = cached tournament log string
```

The most interesting use for programmed setups is setups with *mutators.* Mutators describe a genetic drift of certain strategies into another type of strategy. For example one could imagine a certain percentage of TITFOR-TAT players degenerating into DOVE every generation, because they weren't really able to understand the principle of TITFORTAT. The assumption of degeneration has great consequences for the topics of evolutionary stability and the like. Here is an example of a simulation setup with a non uniform population consisting of the strategies GRIM, DOVE and TESTER, where GRIM degenerates to DOVE at a rate of one percent per generation. (Try this one out and let the ecological simulation continue for at least 400 generations. What can you observe?)

```
custom_setup = SimSetup(name         = "Grim => Dove, Tester",
                        strategyList = [Grim(), Dove(), Tester()],
                        population   = (0.8, 0.01, 0.19),
                        mutators     = [Mutator(0,1,0.01)])
```

8   Further Reading

For those who would like to learn more about the game theoretical foundation of *CoopSim*, here are a few good books on this topic:

Axelrod, Robert (1984): Die Evolution der Kooperation, Oldenbourg, Mnchen (5 Aufl. 2000; engl. Original 1984).

Axelrod, Robert (1997): The Complexity of Cooperation. Agent-Based Models of Competition and Collaboration, Princeton University Press, Princeton.

Binmore, Ken / Samuelson, Larry (1992): Evolutionary Stability in Repeated Games Played by finite Automata, in: Journal of Economic Theory 57 (2/1992), 278-305.

Binmore, Ken (1994): Game Theory and the Social Contract I. Playing Fair, MIT Press, Cambridge (Massachusetts), London (England) (4. Nachdruck 2000).

Binmore, Ken (1998): Game Theory and the Social Contract II. Just Playing, MIT Press, Cambridge (Massachusetts), London (England).

Maynard Smith, John (1982): Evolution and the Theory of Games, Cambridge Univ. Press, Cambridge (8. Aufl. 2000).

Schuessler, Rudolf (1990): Kooperation unter Egoisten: vier Dilemmata, R.Oldenbourg Verlag, Muenchen (2.Aufl. 1997)

After having occupied myself with computer simulations of this type for a longer time, my own oppinion about the scientific value and particularly the explanatory power of this brand of simulations has become very critical. If curious, read my paper on:
http://www.eckhartarnold.de/papers/simulations_2006/Simulations_preconference.html