APPEARANCES CAN BE DECEIVING: LESSONS LEARNED RE-IMPLEMENTING AXELROD’S “EVOLUTIONARY APPROACH TO NORMS”

By

José Manuel Galán

&

Luis R. Izquierdo
# CONTENTS

1. Introduction ..................................................................................................................................... 3

2. The Conceptual Model ....................................................................................................................... 3
  2.1. Axelrod’s norms game .................................................................................................................... 3
  2.2. The Metanorms Model ................................................................................................................ 5

3. Model Implementation: the program .................................................................................................. 8
  3.1. Installation .................................................................................................................................... 8
  3.2. Program Parameters ..................................................................................................................... 8
  3.3. Repast Parameters ...................................................................................................................... 17
  3.4. How to run simulations .............................................................................................................. 18
  3.5. GUI simulation ............................................................................................................................ 19
  3.6. Batch simulation ........................................................................................................................ 23

4. Where to find everything .................................................................................................................. 26

5. The authors ....................................................................................................................................... 27
  José M. Galán. ...................................................................................................................................... 27
  Luis R. Izquierdo. .............................................................................................................................. 27

6. List of figures and tables .................................................................................................................. 28
1. Introduction

This document outlines the usage of a computer program created to replicate and extend Prof. Axelrod’s agent-based model described in the paper “An evolutionary approach to norms”\(^1\), published in *American Political Science Review* 80, nº4, December 1996: 1095-1111. The code is written in Java 2, and is known to work using RePast-2.2 libraries\(^2\) and JDK 1.4.2_06-b03 distribution on PCs running Windows 2000, NT and XP.

2. The Conceptual Model

In the original paper Axelrod explores two different models:

1. Axelrod’s norms game
2. Axelrod’s metanorms game

A previous important point in this model is the definition of norm. Following Axelrod’s definition, we understand that a norm exists in a given social setting to the extent that individuals usually act in a certain way and are often punished when seen not to be acting in this way.

2.1. Axelrod’s norms game

The Norms game is played by 20 agents who have to make two decisions:

1. Agents have to decide whether to cooperate or defect. A defecting agent gets a *Temptation* payoff \((T = 3)\) and inflicts each of the other agents a *Hurt* payoff \((H = -1)\). If, on the other hand, the agent cooperates, no one’s payoff is altered.
2. The opportunity to defect given to an agent comes with a known chance of being seen by *each* of the other agents, called \(S\). This probability of being observed is drawn from a uniform distribution between 0 and 1 every time a certain agent is given the opportunity to defect. For each observed defection, agents have to decide whether to punish the defector or not. Punishers incur an *Enforcement* cost \((E = -2)\) every time they *punish* \((P = -9)\) a defector.

The strategy of an agent is defined by its propensity to defect (*Boldness*), and its propensity to punish agents they have observed defecting (*Vengefulness*). Agents defect when given the opportunity if their *Boldness* is higher than the probability of being observed \((S)\); and they punish observed defectors with probability *Vengefulness*. In this model, each of these propensities is implemented as a 3-bit string denoting eight evenly-distributed values from 0 to 1 \((0/7, 1/7, \ldots, 7/7)\). The actual values for each agent’s strategy are determined randomly at the beginning of each simulation run.

---


A round in this model is completed when every agent has been given exactly one opportunity to defect, and also the opportunity to observe (and maybe punish) any given defection that has taken place.

**AXELROD’S MODELS: The Norms model**

**Figure 1.** Sketch of the Norms Game

Four rounds constitute a generation. At the beginning of every generation the agents’ payoffs are initialised; at the end of every generation the payoff obtained by every agent in the four rounds is computed and two evolutionary forces come into play:

1. Agents with a payoff exceeding the population average by at least one standard deviation are replicated twice; agents who are at least one standard deviation below the population average are eliminated; and the rest of the agents are replicated once. The number of agents is kept constant, but Axelrod does not specify exactly how.\(^3\)
2. Whenever a bitstring is replicated, every bit has a certain probability to be flipped (*MutationRate = 0.01*).

---

\(^3\) See [here](#) our implementation of this method.
2.2. The Metanorms Model

Having concluded that the norm to cooperate collapses in the previous model, Axelrod investigates the role of metanorms as a way of enforcing norms. The metanorm dictates that one must punish those who do not follow the norm (i.e. those who do not punish observed defectors). However, someone who does not punish an observed defector might not be caught. In the Metanorms game, the chance of being seen not punishing a defection (given that the defection has been seen) by each of the other 18 agents (excluding the defector) is the same as the chance of seeing such defection. Similarly, the propensity to punish those who do not comply with the norm (meta-punish) is the same as the propensity to punish defectors. As far as payoffs are concerned, meta-punishers incur a Meta-enforcement cost ($ME = -2$) every time they Meta-punish ($MP = -9$) someone who has not punished an observed defector.
AXELROD’S MODELS: The MetaNorms model

The Norms model

20-player PD

Vengefulness j

j does not punish j

k sees j

k does not see j

j punishes i

k meta-punishes j

MEncoragement = -2
MPunishment = -9

Each k ≠ i, j

k does not punish j

j does not punish i

i defects

j cooperates

Temptation = 3

Hurt = -1

S

Figure 3. Sketch of the MetaNorms Game
Figure 4. UML activity diagram of the method \texttt{metaNorms(Number, Agent, Agent)} of the object model. This method is called in the UML activity diagram shown in figure 2. The condition \texttt{metaNormsActive} is \texttt{false} in the Norms model and \texttt{true} in the Metanorms model.
3. Model Implementation: the program

3.1. Installation

Our implementation of Axelrod’s model (called RAEN for Reimplementation of Axelrod’s Evolutionary Approach to Norms) is written in Java. Assuming you have correctly installed an appropriate version of Java\(^4\) in your platform, installation of the model only involves unpacking the zip file RAEN-standAlone.zip. You then should be able to execute the file RAEN-standAlone.jar. This .jar file includes the classes we have implemented and other classes which are necessary to run our model (all of which are distributed as part of the package RePast-2.2, and they are all open source). If you already have RePast-2.2 installed in your computer, you can download just the file RAEN-jar.zip which contains the jar file RAEN.jar with our classes only.

The source code is also available in RAEN-src.zip, so you can download the .java files if you prefer to make your own .class files to run the model.

3.2. Program Parameters

In this model we have tried to minimize the use of floating point numbers\(^5\) but unfortunately we have had to use some doubles.

ActivateMetanormsModel. This parameter is represented by a boolean variable. It represents each of the two types of model that can be run in the program:

1. Axelrod’s norms game, if ActivateMetanormsModel is false. In the GUI mode ActivateMetanormsModel is false if the check box is not activated.

   ![ActivateMetanormsModel](image)

   Figure 5. Checkbox deactivated. Running the Norms game

2. Axelrod’s metanorms game if ActivateMetanormsModel is true (if its check box is activated).

   ![ActivateMetanormsModel](image)

   Figure 6. Checkbox activated. Running the Metanorms game

ParamOutputFile. If ParamOutputFile is true (if its check box is activated) the program creates a .csv\(^6\) file with the relevant parameters after each execution. Only the parameters used in the run are printed to file. For instance, if the selected selection mechanism is STANDARD DEVIATION, the parameter ConstantForRouletteWheel will

---

\(^4\) You can obtain a free version in http://java.sun.com/. We have tested everything with JDK 1.4.2_06-b03 and JDK 1.5.0-b64.

\(^5\) To read more about the topic, see http://www.macaulay.ac.uk/fearlus/floating-point/index.html.

\(^6\) Comma separated values. It can be opened with a word processor or a spreadsheet.
not be included in the file because it is associated with the \textit{ROULETTE\_WHEEL} selection mechanism.

The file is created in the folder where the program is, and its name includes the word “input”, the date, and the time of the beginning of the run.

If \textit{ParamOutputFile} is false, the program runs the model but the parameters of the execution are not stored.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{param_output_file.png}
\caption{Example of the param output file}
\end{figure}

\textbf{DataOutputFile}. If this parameter is true, the population averages of boldness, vengefulness and metavengefulness (if it is different from the vengefulness) are stored in a .txt file while the simulation is running. As in the previous case, the file is created in the folder where the program is, and its name includes the word “output”, the date, and the time of the beginning of the run.

If the \textit{ParamInputFile} and \textit{DataOutputFile} parameters are both activated, the .csv and .txt files will contain exactly the same date and time.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{data_output_file.png}
\caption{Examples of the data output file opened with a word processor and some output files}
\end{figure}
**OutputStep.** This parameter is related with the *DataOutputFile* parameter. It determines the frequency of data saving to the data file. For instance, if the value of the parameter is 2, the population averages are recorded every two generations. If *DataOutputFile* is false (not activated), the *OutputStep* has no influence in the program.

**NumAgents.** This parameter represents the number of players in the game. It must be a positive integer.

**ArrayDimension.** The characteristics of the agents in Axelrod’s original model were represented by a 3-bit string to allow eight different levels. In our program the parameter *ArrayDimension* determines the number of bits representing each of the characteristics of the agents. A higher value of the parameter involves more different levels of boldness, vengefulness or metavengefulness available for the players.

![3-bit string and its meaning](image)

**NumberOfGenerations.** This parameter determines the number of generations to run. It partially determines the length of the simulation. We can consider that a single simulation is divided into four steps (See Figure 10). The “NumberOfGenerations” parameter sets the number of times that a new population of players is created according to their relative scores.

![Dynamics of the simulation](image)

**Rounds.** This parameter represents the number of rounds that the game is played in each generation. (See the first step of figure 10)
**MutationRate_per_bit.** This parameter is a double between 0 and 1. It is the probability that each of the bits representing an agent’s strategy (boldness, vengefulness and maybe metaVengefulness) will be flipped every time an agent is replicated.

**MutationRate_per_generation.** This parameter is the expected value of mutations (bits flipped) per generation. It is linearly related with *MutationRate_per_bit*. Setting one of them completely determines the other. If we fix the *MutationRate_per_bit* parameter, then *MutationRate_per_generation* is calculated with the following code:

```java
if (metaVengefulnessEqualToVengefulness) {
    mutationRate_per_generation =
    mutationRate_per_bit * (numAgents * (arrayDimension * 2));
} else {
    mutationRate_per_generation =
    mutationRate_per_bit * (numAgents * (arrayDimension * 3));
}
```

Consequently if we set up the *MutationRate_per_generation* then the *MutationRate_per_generation* is:

```java
if (metaVengefulnessEqualToVengefulness) {
    mutationRate_per_bit =
    mutationRate_per_generation / (numAgents * arrayDimension * 2);
} else {
    mutationRate_per_bit =
    mutationRate_per_generation / (numAgents * arrayDimension * 3);
}
```

**Defection_T.** This parameter is a double that determines the payoff initially obtained by a defecting agent (\(T = Temptation\) to defect).

**HurtByOthers_H.** If an agent \(i\) defects, it obtains *Defection_T* payoff, but each of the others are hurt \((H)\), getting a payoff with the value of the parameter *HurtByOthers_H*. Please note that this parameter has typically a negative value. It is also implemented as a double.

**Punishment_P.** If agent \(i\) defects, some of the other agents may see the defection, and those who do may choose to punish \((P)\) the defector. A defector receives a *Punishment_P* payoff every time it is punished. Like all the payoff parameters in the program it is represented with a double, and since it represents a punishment it usually has a negative value.

**EnforcementCost_E.** If an agent \(j\) decides to punish an agent \(i\) who has defected, agent \(i\) obtains the *Punishment_P* payoff but because the act of punishing is typically somewhat costly, the punisher (agent \(j\)), has to pay an *enforcement cost* \((E)\). The enforcement cost is represented by a double in the *EnforcementCost_E* parameter and it is typically a negative number.
The parameters Defection\_T, HurtByOthers\_H, Punishment\_P and EnforcementCost\_E completely determine the payoff matrix in Axelrod’s norm game.

**Table 1.** Payoff matrix in the Axelrod’s norms game

<table>
<thead>
<tr>
<th>Event</th>
<th>Payoff per Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defection</td>
<td>T = 3</td>
</tr>
<tr>
<td>Punishment</td>
<td>P = -9</td>
</tr>
<tr>
<td>Hurt by Others</td>
<td>H = -1</td>
</tr>
<tr>
<td>Enforcement Cost</td>
<td>E = -2</td>
</tr>
</tbody>
</table>

**PunishmentAgainstNonpunishment\_PNP.** This parameter has influence only in the metanorms model (activateMetanormsModel must be true). In the metanorms game, if someone defects (agent i), and another agent (agent j) sees that defection but does not punish it, then the other players (excluding agent i and agent j) have a chance to see and to punish agent j. If a non-punisher (agent j) is punished, it receives a PunishmentAgainstNonpunishment\_PNP payoff. Like all the payoff parameters in the program it is represented with a double, and since it represents a punishment it usually has a negative value.

**EnforcementCostAgainstNonpunishment\_ENP.** This parameter is only relevant in the metanorms game. Following the same nomenclature that in the preceding paragraph, when an agent k has seen an agent j that has not punished a defector i, and k decides to punish j, the agent punished j receives a PunishmentAgainstNonpunishment\_PNP payoff, but agent k has to pay an enforcement cost against the non punishment (ENP). This cost is represented by a double in the EnforcementCostAgainstNonpunishment\_ENP parameter and it is typically a negative number.

Together with the parameters Defection\_T, HurtByOthers\_H, Punishment\_P and EnforcementCost\_E, the parameters PunishmentAgainstNonpunishment\_PNP and EnforcementCostAgainstNonpunishment\_ENP completely specify the payoff matrix of the metanorms game.

**Table 2.** Payoff matrix in the Axelrod’s metanorms game

<table>
<thead>
<tr>
<th>Event</th>
<th>Payoff per Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defection</td>
<td>T = 3</td>
</tr>
<tr>
<td>Punishment</td>
<td>P = -9</td>
</tr>
<tr>
<td>Hurt by Others</td>
<td>H = -1</td>
</tr>
<tr>
<td>Enforcement Cost</td>
<td>E = -2</td>
</tr>
<tr>
<td>Meta Punishment</td>
<td>MP = -9</td>
</tr>
<tr>
<td>Meta Enforcement Cost</td>
<td>ME = -2</td>
</tr>
</tbody>
</table>

**Initialization.** In the current version of the implementation there are two different methods available to initialise the characteristics of the population:

1. The first one is RANDOM. This is the method used in Axelrod’s original implementation. It consists in generating the initial boldness, vengefulness and metavengefulness of each agent randomly with a uniform distribution among the
values set. If this initialization method is selected, the parameters initialBoldnessAverage, initialVengefulnessAverage and initialMetaVengefulnessAverage can not be modified, and their probes show their expected values.

2. The second initialization method is CUSTOM. This method is deterministic. It allows setting up a population with the specific values preferred by a user. The only constraint is that the initial characteristics of the agents are the same for every agent. In that case the parameters initialBoldnessAverage, initialVengefulnessAverage and initialMetaVengefulnessAverage determine the specified values of the initial population. Please note that the simulator (in the discrete version) does not allow setting up a non integer value in this parameter. The simulator also checks if the parameter metaVengefulnessEqualToVengefulness is activated, and if so, it ensures that both parameters are equal. If the user deactivates metaVengefulnessEqualToVengefulness, then it is possible to set different values for the two parameters.

**InitialBoldnessAverage.** This parameter has two different interpretations (see Initialization parameter):

1. If initialization has the value RANDOM, InitialBoldnessAverage shows the expected average value of the initial boldness of the population, and its value can not be modified.
2. If initialization has the value CUSTOM, InitialBoldnessAverage allows customizing the boldness of all the agents in the initial population (All the agents with the value determined by the user, but all with the same value).

**InitialVengefulnessAverage.** Analogous to InitialBoldnessAverage.

---

7 Please note that since it is a stochastic process, the real average values of the initial population may not coincide with the expected average values.
**InitialMetaVengefulnessAverage.** This parameter has three different interpretations (see Initialization parameter):

1. If initialization is RANDOM, the parameter *InitialMetaVengefulnessAverage* shows the expected average value of the initial metavengefulness of the population, and its value can not be modified.
2. If initialization is CUSTOM and the parameter *metaVengefulnessEqualToVengefulness* is true, then *InitialMetaVengefulnessAverage* will have the same value as *InitialVengefulnessAverage*.
3. If initialization is CUSTOM and the parameter *MetaVengefulnessEqualToVengefulness* false, then *InitialMetaVengefulnessAverage* and *InitialVengefulnessAverage* parameters are decoupled and the user can set up the metaVengefulness of all the agents in the initial population (All the agents with the value determined by the user, but all with the same value).

**Selection.** In Axelrod’s models one of the evolutionary forces is the selection method (see point 3 of Figure 13). In Axelrod’s original implementation the selection mechanism was as follows: agents with a payoff exceeding the population average by at least one standard deviation are replicated twice; agents who are at least one standard deviation below the population average are eliminated; and the rest of the agents are replicated once. The number of agents is kept constant, but Axelrod did not specify exactly how.

---

See [here](#) our implementation.

---

**Figure 13.** Dynamics of the simulation

In our program there are three selection mechanisms available:
1. **STANDARD DEVIATION**. This method is a generalisation of the one implemented by Axelrod. Under this selection mechanism agents with a payoff equal to or greater than the population average plus $\text{StandardDeviationForDoubleReproduction}$ standard deviations are replicated twice; agents who are below the population average plus $\text{StandardDeviationForNoReproduction}$ standard deviations are eliminated; and the rest of the agents are replicated once.\(^9\)

![Axelrod's selection mechanism](image)

**Figure 14.** Axelrod’s selection mechanism

The method used in our implementation to keep the number of agents constant is the following: After the selection mechanism has been executed, we may find that the number of offspring is not the same as the desired number of agents ($\text{NumAgents}$), so our solution is:

1. What is the number of offspring agents?
2. Is that number the same as the desired number?
   a. Yes. Great!!! The algorithm has finished.
   b. No, there are fewer offspring than $\text{NumAgents}$. The algorithm randomly selects one offspring and duplicates it. Check if we have the same number: if yes, finished, if no go back to 2.b
   c. No, there are more offspring than $\text{NumAgents}$. The algorithm randomly selects one offspring and eliminates it. Check if we have the same number: if yes, finished, if no go back to 2.c

2. **RANDOM_TOURNAMENT.** This method involves selecting two agents from the population at random and replicating the one with the higher payoff for the next generation. If case of tie, one of them is selected at random. This process is repeated as many times as necessary to keep the number of agents constant ($\text{NumAgents}$).

\(^9\)A particular case of this method is Average selection ($\text{StandardDeviationForDoubleReproduction} = 0$ & $\text{StandardDeviationForNoReproduction} = 0$); using these parameters, agents with a payoff greater than or equal to the population average are replicated twice and agents who are below the population average are eliminated.

\(^{10}\)Note that this value is typically a negative real number.
3. **ROULETTE_WHEEL.** This method involves calculating every agent’s fitness, which is equal to their payoff minus the minimum payoff obtained in the generation plus a parameter called \textit{ConstantForRouletteWheel}$.^{11}$ Agents are then given a probability of being replicated (in each of the $\textit{NumAgents}$ replications) that is directly proportionate to their fitness. If every agent obtained exactly the same score, we execute a random tournament for that generation.

**Figure 15.** Random tournament selection mechanism

**Figure 16.** Roulette wheel selection mechanism

**StandardDeviationForDoubleReproduction.** This parameter determines the number of standard deviations plus the population average payoff that an agent needs to obtain to be replicated twice in the selection method of \textit{STANDARD\_DEVIATION}$^{12}$.

**StandardDeviationForNoReproduction.** That parameter determines the number of standard deviations plus the population average payoff that an agent needs to obtain to be replicated at least once in the selection method of \textit{STANDARD\_DEVIATION}$^{13}$.

$^{11}$ Typically the constant has the value 0.

$^{12}$ See selection parameter.

$^{13}$ See selection parameter and note that this value should be a negative real number according to its definition.
ConstantForRouletteWheel. This parameter is only used when the selection mechanism is ROULETTE_WHEEL. Such method involves calculating every agent’s fitness, which is equal to their payoff minus the minimum payoff obtained in the generation plus $\text{ConstantForRouletteWheel}$ (double).

\[
\text{Fitness}[i] = \text{Score}[i] + \text{minScoreInThePopulation} + \text{ConstantForRouletteWheel}
\]

MetaVengefulnessEqualToVengefulness. This parameter is represented by a boolean variable. If it is true (its check box is activated), the metanorms game includes Axelrod’s hypothesis that an agent’s propensity to punish a defection (vengefulness) is the same as its propensity to punish to a non-punisher (metavengefulness). If false, the variables vengefulness and metaVengefulness will not be necessarily equal.

3.3. Repast Parameters

CellDepth. No influence in the model.

CellHeight. No influence in the model.

CellWidth. No influence in the model.

PauseAt. You can set the simulation to pause at a specific iteration (e.g., at 500 on its way to 2000). The user might use this feature to take a snapshot of the middle point of a simulation. This feature can be turned off by placing a -1 in the PauseAt parameter.

RandomSeed. Each simulation begins with the selection of a random seed that is used to initialize variables. If you wish to exactly duplicate a sequence of pseudo-random numbers, you will need to use the same random seed (as well as the rest of the parameters).
3.4. How to run simulations

The program can be run from the command line as a typical java application, even if you do not have RePast-2.2 installed in your computer\textsuperscript{14}. If that is the case, you will have to download the file RAEN-standAlone.zip. What you will always need is the Java Virtual Machine\textsuperscript{15}.

After unzipping RAEN-standAlone.zip or RAEN-jar.zip you will see a .jar file and a parameter file. In this section we will refer to the .jar file as RAEN-standAlone.jar and to the parameter file as Experiment.pf (if you are not using the stand-Alone version the .jar file will be named RAEN.jar).

Basically the model has two different modes of execution:

1. Graphic User Interface (GUI) Mode. It enables the user to observe the results and to interact with the model on real-time, but execution is noticeably slower than in batch mode.
2. Batch Mode. It enables the user to run the model sequentially as many times as desired, exploring the parameter space and storing data from each run. Execution is significantly faster than in the GUI mode.

In the usage synopsis below, square brackets [ ] are used to denote an optional element and angle brackets <> to describe some value the user should provide.

```
java -jar RAEN-standAlone.jar [-help] [-b <ParameterFile>]
```

The command-line options are:

- -help or -h to see the help message
- -b to run the model in batch mode (GUI is the default mode).
- <NameOfParameterFile> is the name of the parameter file. If batch mode is active, this parameter is required.

For instance, the following command can be used to execute the model in GUI mode.

```
java -jar path_to_the_model/RAEN-standAlone.jar
```

\textsuperscript{14} See \url{http://repast.sourceforge.net} for a complete description of how to execute RePast models in different ways.

\textsuperscript{15} You can obtain a free version in \url{http://java.sun.com/}. We have tested everything with JDK 1.4.2_06-b03 and JDK 1.5.0-b64.
The current version of the program does not allow the user to change the default parameters of the GUI mode using a parameter file. If you want to change the default parameters and run the simulation in GUI mode you have to change the parameters in the GUI window.

3.5. GUI simulation

Simulation controls

GUI mode is typically used for exploratory runs of the model and to generate pictures for diagrams. When the model is first started in GUI mode, two windows appear: the RePast window to control the execution, and the model settings window to configure your simulation. The RePast window allows the user to start, stop, pause, setup, and exit a simulation using the toolbar. The table below lists the toolbar buttons and their functions.

![RePast simulation controls](image)

**Figure 19.** RePast simulation controls

1. **Start:** The start button starts the simulation when it is paused or has not yet been started.
2. **Step:** The step button performs a single iteration of the schedule after the simulation has been paused or has not yet been started.
3. **Initialize:** The initialize button executes the initializing code only.
4. **Stop:** The stop button stops the simulation.
5. **Pause:** The pause button causes the simulation to pause.
6. **Setup:** The setup button "sets up" the simulation by executing the programmers’ defined setup code.
7. **Load Model:** The load model button pops up a dialog allowing the user to specify a model to load.
8. **View Settings:** The view settings button will display the various model settings panel if it is hidden or destroyed.
9. **Exit:** The exit button will shut the simulation down and exit.

To run a simulation you can click the start, step or initialize buttons. If you click the initialize button, you should then click step or start. If you want to stop the simulation and run it again, you can click stop and then the setup button.

The other window is the settings window. The initial parameters will appear in the settings window on the tab entitled "Parameters" into which the user can see the default
starting parameters and enter new ones\textsuperscript{16} (do not forget to press return after each entry!!!).

\textbf{Figure 20.} Model settings window

Also in the settings window are the Custom Actions tab and the Repast Actions tab. The Custom Actions tab does not contain any information in this specific model.

The Repast Actions tab is shown in Figure 21:

\textsuperscript{16} Please note that some of the parameters are related to each other, so the modification of one parameter can cause another one to change.
• The first two buttons allow the user to make a movie or take a snapshot. Clicking on one of these buttons will bring up a dialog box containing further information and choices.
• The third button will bring up a dialog box for creating a sequence chart. It is possible to create new sequence charts, apart of the default ones, to observe the evolution of the accessible variables that we are interested.
• The "In Alpha Order" check box controls the order of the parameters in the parameters tab. If it is checked the parameters will appear in alphabetical order. Otherwise they will appear in the order they are defined in the model.
• The "Set RngSeed in Defaults" checkbox determines whether the current random number seed will be included as a default value either when written to a file or set as the current default.
• "Update Probes" determines whether any probed objects will have their displayed properties updated in real-time or not.
• "Show custom charts" determines whether or not any custom charts created for this model using the "create / edit charts" button are automatically created and displayed when this model is run.
• The "Set As Default" button will set the current parameters as the default until RePast is exited. This means that whenever the setup button is pressed these new defaults will be used rather the initial parameters.
• The "write parameters" button will write the current parameters to a file so that can be loaded later.
• The "about" button displays a typical ‘about’ window giving some further information about RePast.

Displaying a simulation can be one of the slowest parts of a simulation. Repast will not update the display if the display window is minimized, so you can increase the speed of the simulation by minimizing the windows you are not interested in at a certain time. When the window is displayed again it will update its content.
Simulation outputs

There are two kinds of outputs in a GUI model, the file outputs and the graphical outputs. The file outputs are configured with the checkboxes of the `ParamOutputFile` and `DataOutputFile` parameters\(^\text{17}\).

The graphical outputs are two windows with similar information:

1. The first one is a graph with the temporal evolution of the agents’ average properties: average boldness, average vengefulness and, if appropriate, average metavengefulness\(^\text{18}\).

![Temporal evolution of agent properties](image)

**Figure 22.** Temporal evolution of agent properties graph

2. A second plot shows dynamically the evolution of the properties in a boldness average (X axis) - vengefulness average (Y axis) map. This map is intended to observe easily the proportion of time that the system spends in states of norm collapse and in states where the norm is established, and how the evolutionary transition from the former to the latter occurs.

- Norm Collapse: We say that the norm has collapsed when the simulation is in states where the average Boldness is at least 6/7 and the average Vengefulness is no more than 1/7 (see fig. 23).
- Norm Establishment: We say that the norm has been established when the simulation is in states where the average Boldness is no more than 2/7 and the average Vengefulness is at least 5/7 (see fig. 23).

\(^{17}\) A description of these parameters is available in the parameter chapter.

\(^{18}\) If vengefulness and metavengefulness are not equal.
3.6. Batch simulation

As we have seen in the previous sections, to run the model in batch mode, you should use the –b flag and provide the name of a suitable parameter file in the command line. A typical command to run the model in batch mode would like this:

```java -jar path/RAEAN-standAlone.jar -b params.pf```

It is important to say that in order to run a batch simulation it is necessary to specify all the parameters in the file\(^\text{19}\).

In a batch model, the parameter file can also define a parameter space and describe how the model should explore that space. A parameter file has the following format:

```java
runs: x
Parameter {
    value_definition
}
```

where x is an integer and Parameter is the name of some model parameter accessible through get and set methods (in our case, all specified parameters in this guide). runs specifies the number of runs to execute for the current parameter value. The value_definition is composed of one or more keywords and some values, as described below.

The multi-keyword value definitions:

- "start:" the starting numerical value of the parameter.

\(^{19}\) In each of the files provided there is an example
- "end:" the ending numerical value of the parameter.
- "incr:" the amount to increment the current value of the parameter.

Single keyword value definitions:

- "set:" defines a single numerical value as a constant for the entire collection of batch runs.
- "set_list:" defines a space-separated list of numerical values. A batch simulation will iterate through the list.
- "set_boolean:" defines a boolean value as a constant for the entire batch simulation. Allowed values are "true" and "false" (without the quotes).
- "set_string:" defines a string value as a constant for the entire batch simulation. The string value must not contain any white space.
- "set_boolean_list:" same as set_list but a list of boolean values (true or false).
- "set_string_list:" same as set_list but a list of string values.

Some examples:

```plaintext
runs: 10
Defection_T {
  start: 10
  end: 30
  incr: 10
}
```

This means start with a `defection_T` parameter with a value of 10 and run the simulation 10 times using this value. Increment the `defection_T` value by 10 and run the simulation 10 times with a `defection_T` value of 20 (start 10 + incr 10). Increment the `defection_T` value by another 10, and run another 10 times with the `defection_T` value of 30 (start 10 + incr 10 + incr 10). Incrementing the current value at this point would result in a value greater than the ending value and so the simulation ends.

More than one parameter can be specified, so for example:

```plaintext
runs: 10
NumAgents {
  start: 10
  end: 30
  incr: 10
}
Defection_T {
  start: 3
  end: 9
  incr: 1
}
```

20 Although the examples have been adapted to the current work, the basis of this explanation has been taken from [http://repast.sourceforge.net/](http://repast.sourceforge.net/) in the section How to Use Parameters and Parameter Files.

21 RePast would run 10 simulations with `numAgents=10 & defection_T=3`, then 10 with `numAgents=20 & defection_T=4` and then 10 with `numAgents=30 & defection_T=5`
Where both \textit{numAgents} and \textit{defection\_T} are incremented as described above. If using more than one parameter it is important to synchronize them, as whenever any parameter's current value is greater than its end value, the simulation will exit.

Parameters can also be nested. For example,

\begin{verbatim}
runs: 1
NumAgents {
    start: 10
    end: 30
    incr: 10
    {
        runs: 10
        Defection\_T {
            start: 3
            end: 9
            incr: 1
        }
    }
}
\end{verbatim}

This example means starting with a \textit{numAgents} value of 10 run the simulation 10 times with a \textit{defection\_T} of 3. Increment \textit{defection\_T} by 1 and run the simulation 10 times, continue until the value of \textit{defection\_T} is greater than 9. At this point, increment \textit{numAgents} by 10 and run the simulation 10 times with a \textit{defection\_T} of 3. Increment \textit{defection\_T} by 1 and run the simulation 10 times. This continues until the value of \textit{numAgents} is greater than 30. Multiple levels of nesting are possible.

Setting constants:

\begin{verbatim}
runs: 1
NumAgents {
    start: 10
    end: 30
    incr: 10
    {
        runs: 10
        Defection\_T {
            start: 3
            end: 9
            incr: 1
        }
    }
}
\end{verbatim}

\begin{verbatim}
RngSeed {
    set: 1
}
\end{verbatim}
**RngSeed** is a parameter of every model and can be manipulated like any other parameter. And here it is set to one and this value will remain constant over all the individual batch runs.

List parameters:

```plaintext
runs: 1
NumAgents {
  start: 10
  end: 30
  incr: 10
}
  runs: 10
  Defection_T {
    set_list: 1.2 3 10 12 84
  }
}
RngSeed {
  set: 1
}
```

This is the same as above except that *defection_T* will be incremented via the list. So first run with *defection_T* as 1.2, do this for 10 runs. Then set *defection_T* to 3 and run with this value for 10 times. Continue until the end of the list, then increment *numAgents* and start at the beginning of the *defection_T* list, and so on until the *numAgents* parameter is greater than 30.

Parameter files can contain comments delimited by the standard c/c++/java comment markers: `'//', '/*...*/'

### 4. Where to find everything

The website [www.insisoc.org/metanorms/](http://www.insisoc.org/metanorms/) has been created to give support to everyone interested in the reimplementation of this model. From there you can download:

- The source code ([www.insisoc.org/metanorms/RAEAN-src.zip](http://www.insisoc.org/metanorms/RAEAN-src.zip)). The source code is released under the [GNU General Public Licence](http://www.insisoc.org/metanorms/RAEAN-src.zip). Clicking the model source download link will be taken as an assertion that you agree to abide by the terms of this licence.
- The user guide ([www.insisoc.org/metanorms/RAEAN-userGuide.pdf](http://www.insisoc.org/metanorms/RAEAN-userGuide.pdf)).
- JAR file [www.insisoc.org/metanorms/RAEAN-jar.zip](http://www.insisoc.org/metanorms/RAEAN-jar.zip)
5. The authors

José M. Galán. José Manuel Galán is trained in Industrial Engineering. He works as assistant lecturer in the University of Burgos (Spain). He is also doing his PhD, supervised by Adolfo López-Paredes of the INSISOC Group of the University of Valladolid and by Ricardo Del Olmo of the University of Burgos. He is interested in water management, traffic control, replication, agent-based modelling, game theory and the use of models in general.

University of Burgos
Escuela Politécnica Superior
Avenida de Cantabria, s/n. Ed.A1
09006 Burgos
Spain

Email: jmgalan@ubu.es
Web: http://www.insisoc.org

Luis R. Izquierdo. Luis R. Izquierdo is trained in Industrial Engineering, Business and Economics. He has been working with Nick Gotts and Gary Polhill at the Macaulay Institute since July 2002, extending the FEARLUS land use modelling system. He is also doing a part-time PhD, supervised by Nick Gotts and Bruce Edmonds. The title of his PhD thesis is 'Assessing Factors that Promote Cooperation in Common-Pool Resource Dilemmas'. He is interested in social dilemmas, agent-based modelling, game theory and the use of models in general.

The Macaulay Institute
Craigiebuckler
Aberdeen AB15 8QH
Scotland
UK

Email: lizquierdo@macaulay.ac.uk
Web: http://www.macaulay.ac.uk/fearlus/
6. List of figures and tables

Figure 1. Sketch of the Norms Game ................................................................. 4
Figure 2. UML Activity diagram of one round in Axelrod’s models. The UML diagram of method `metaNorms(Number, Agent, Agent)` is provided in figure 4................................................................. 5
Figure 3. Sketch of the MetaNorms Game ......................................................... 6
Figure 4. UML activity diagram of the method `metaNorms(Number, Agent, Agent)` of the object model. This method is called in the UML activity diagram shown in figure 2. The condition `metaNormsActive` is false in the Norms model and true in the Metanorms model.......................... 7
Figure 5. Checkbox deactivated. Running the Norms game ............................. 8
Figure 6. Checkbox activated. Running the Metanorms game............................ 8
Figure 7. Example of the param output file....................................................... 9
Figure 8. Examples of the data output file opened with a word processor and some output files ................................................................................................. 9
Figure 9. 3-bit string and its meaning ............................................................... 10
Figure 10. Dynamics of the simulation ............................................................. 10
Figure 11. RAMDOM strategy in initialization parameter ............................... 13
Figure 12. CUSTOM strategy in initialization parameter ................................. 13
Figure 13. Dynamics of the simulation ............................................................. 14
Figure 14. Axelrod’s selection mechanism ....................................................... 15
Figure 15. Random tournament selection mechanism ..................................... 16
Figure 16. Roulette wheel selection mechanism ............................................. 16
Figure 17. Average selection, when `StandardDeviationForDoubleReproduction = 0` & `StandardDeviationForNoReproduction = 0` .................................................. 17
Figure 18. RePast parameters ........................................................................ 18
Figure 19. RePast simulation controls ............................................................ 19
Figure 20. Model settings window .................................................................. 20
Figure 21. RePast Action tab .......................................................................... 21
Figure 22. Temporal evolution of agent properties graph ............................... 22
Figure 23. Sequence graph ............................................................................ 23