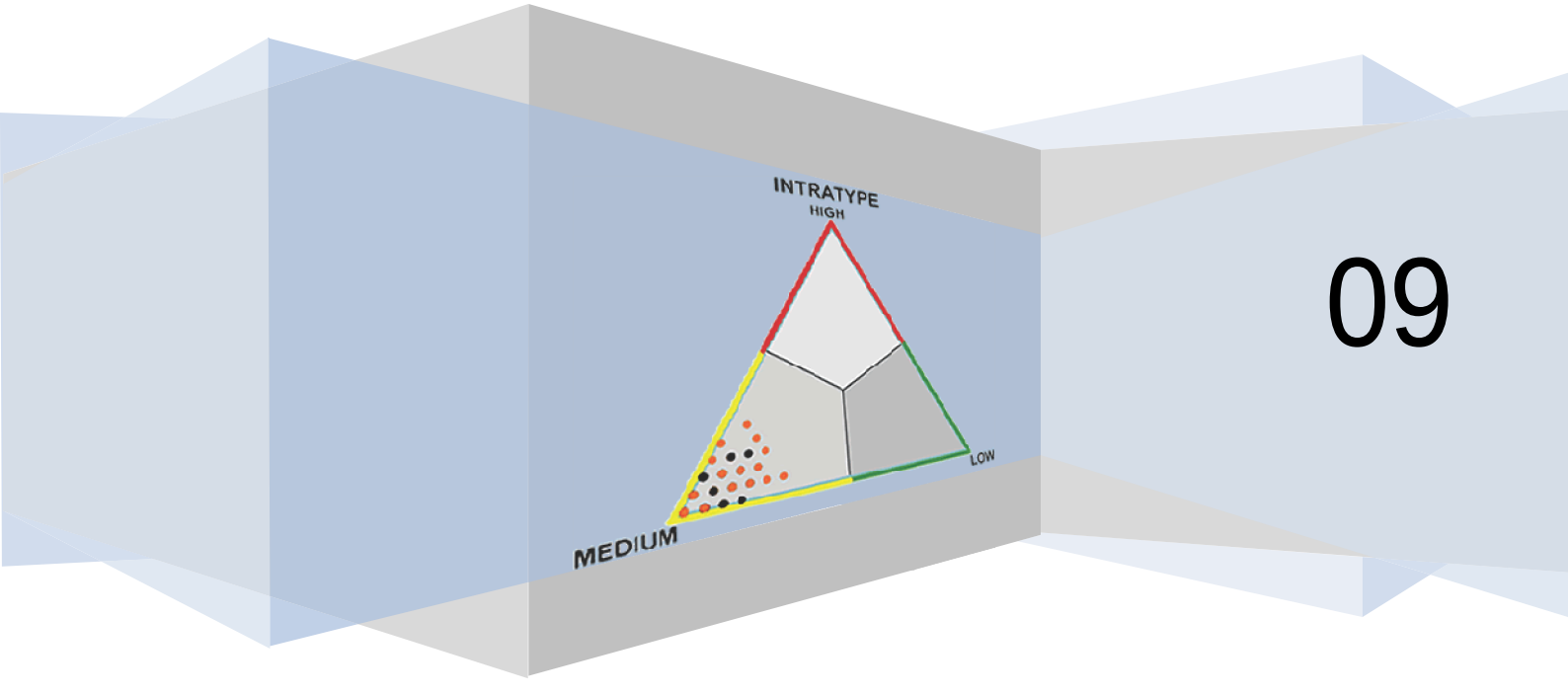
BARGAINING MODEL

D. J. Poza; F. A. Villafáñez



09

ACKNOWLEDGEMENTS

Authors acknowledge the financial support from the Spanish Ministry of Science (Grant TIN2008-06464-C02-02) and the INSISOC members for fruitful discussions and intense debates.

1. INTRODUCTION 3

2. THE MODEL 3

3. ORIGINAL MODEL AND NEW FUNCTIONALITIES 4

3.1. Payoff Matrix 4
3.2. Memory Type 4
3.3. Decision Rule 5

4. SETTING THE PARAMETERS BEFORE STARTING THE SIMULATION 7

4.1. Parameters to Control the Simulation 7
4.2. Parameters to Change the Contour Conditions 8
4.3. Parameters that change the behaviour of the system 9

5. HOW TO RUN THE MODEL 10

6. STOP CONDITIONS AND INTERESTING SCENARIOS 12

6.1. One-Agent Type 12
6.2. Two-Agent Types (THE TAG MODEL) 12

FUTURE EXTENSIONS 14

REFERENCES 14

APPENDIX I. HOW TO PLACE THE AGENTS ON THE SIMPLEX 15

APPENDIX II. HOW TO BUILD THE DECISION BORDERS 21

APPENDIX III. SOURCE CODE 28

1. INTRODUCTION

This application is a replication of a model developed by Robert Axtell, Joshua M. Epstein and H. Peyton Young [1]. The paper in which the model is described (The Emergence of Classes in a Multi-Agent Bargaining Model) was initially published as a Working Paper of the Brookings Institution, 'Center on Social and Economic Dynamics', working paper No. 9, in February 2000.

In this model[1], two players randomly paired demand some portion of the same pie, and the reward that each player gets depends on the portion demanded by her opponent. They can demand three possible portions: low, medium or high. As long as the sum of the two demands is equal or less than 100 percent of the pie, each player gets what she demands; otherwise each gets nothing.

2. THE MODEL

There is a population of **n** agents that are randomly paired to play. Each agent has a memory in which she retains the decision taken by her opponents in **m** previous games. The agent uses the information stored in her memory to demand the portion of the pie that maximizes her benefit (with probability 1-ε) and randomly (with probably ε, where ε represents the noise level in the system).

The decision rule is quite simple. Each player takes a decision computing in her memory how often each option has been chosen by her opponents in previous matches. Then, the player chooses the best decision according to a "maximizing" decision rule. We consider the original decision rule used in this model, and compare the results with a more simple-satisficing one. Both are explained later on this document[2].

Afterwards, the two agents store the decision taken by their opponents in their memories.

Then, two different players are randomly chosen to start a new match. This process continues until all the agents in the population have played.

At this point, the current iteration has finished (i.e. all the agents have played once). This process is repeated successively until either the maximum number of iterations[3] is reached or the system reaches a particular scenario.

Agents are represented on a simplex determined by the memory state of the agents. The more demands of 'low' an agent keeps in his memory, the closer to the bottom-right vertex it is plotted. Equivalently, if a player's memory contains a considerable amount of 'high' decisions, it is placed near the top vertex. Finally, if most of the elements in an agent's memory are 'medium', it is plotted close to the bottom-left vertex.

The simplex is split into three different regions[4], separated by three 'decision borders'. The top region is dominated by frequent demands of 'high' in previous matches; on the right region, 'low' is the dominant element in the agents' memories; whereas agents on the left region have often found that their opponents demand M.

This application has two versions:

- 1-agent type bargaining model.
- 2-agent types bargaining model (tag model).

In the first version, all the agents have the same tag, whereas in the second version, half of the population has one tag (colour) and the other half has another tag (another colour). This lets the agents be distinguishable from one another and makes the system evolve to different scenarios in which segregation can emerge.

In the 1-agent type model, the agents have only one memory set. This is why there is only one simplex in the application. However, in the 2-agent types model, we need two simplexes, as each agent has two

[1] Reader can access the applets at http://www.insisoc.org/bargaining_model_no_tags.html and http://www.insisoc.org/bargaining_model_tag_model.html

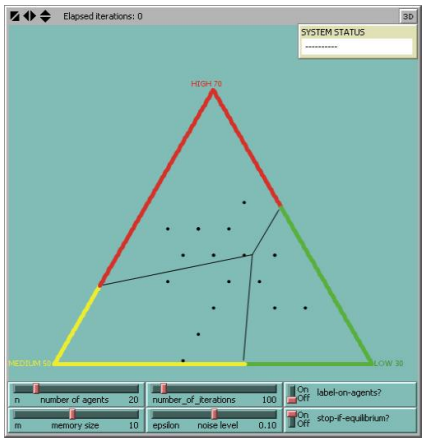
[2] Although there was only one decision rule in the original model, we introduced the possibility of choosing a new decision rule in our replication.

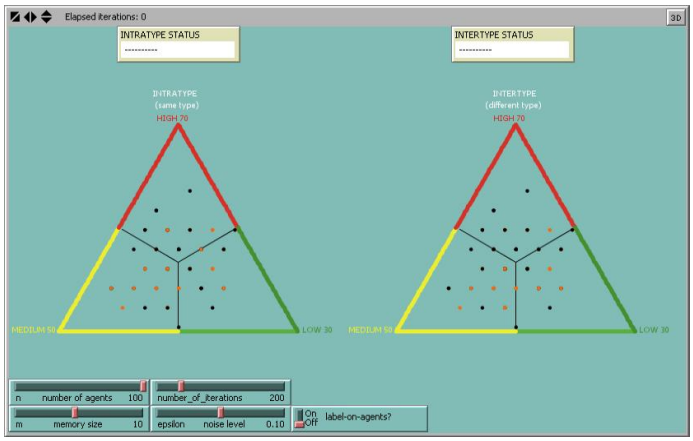
[3] The maximum number of iterations is chosen by the user before the simulation is started.

[4] The layout of the decision borders depends on the selected decision rule.

memory sets: intratype-memory (where she records the matches against players with the same tag) and intertype-memory (where she retains the matches against players with different tag).

The following figures show the interface of the application for the 1-agent type model and the 2-agent types model, respectively:





3. ORIGINAL MODEL AND NEW FUNCTIONALITIES

Although our work is a replication, we have added some new features to the original model to study its structural robustness [2]. To this aim, we added the possibility of changing some of its fundamental assumptions:

- Payoff matrix
- Memory type
- Decision rule

3.1. Payoff Matrix

The payoff matrix represents the different combinations of rewards for each player depending on the decision taken by her opponent:

	H	M	L
H	0,0	0,0	**90,10**
M	0,0	**50,50**	50,10
L	**10,90**	10,50	10,10

	H	M	L
H	0,0	0,0	**80,20**
M	0,0	**50,50**	50,20
L	**20,80**	20,50	20,20

	H	M	L
H	0,0	0,0	**70,30**
M	0,0	**50,50**	50,30
L	**30,70**	30,50	30,30

	H	M	L
H	0,0	0,0	**60,40**
M	0,0	**50,50**	50,40
L	**40,60**	40,50	40,40

In the original model, the values assigned to low, medium and high were fixed: 30% of the pie for L; 50% of the pie for M and 70% of the pie for H. However, in our replication, we added the possibility of changing these rewards by choosing a different payoff matrix (i.e. we added different combinations of payoffs).

3.2. Memory Type

The user can choose up to three different memory types. Notice that the original model operates with 'Standard memory'. In any case, the initial values of the memory are chosen at random. The main characteristics of each memory type are described below:

- **Standard memory:** This memory configuration was used in the original model. All the elements in the memory set have the same weight (i.e. the recent memories have no more importance than the older ones). When the system is started, the agents' memories are initialized with m random values of low, medium and high. When an agent is paired to play with another agent, she stores in her memory the decision taken by her opponent and 'forgets' the oldest element in her memory so that the memory size is kept.
- **Progressive memory:** As it occurred in the standard memory, all the elements in the memory set have the same weight (i.e. the recent memories have no more importance than the older ones). However, all the agents are initialized with a memory-size of 1. When an agent is paired to play with another agent, she adds the decision taken by her opponent in her memory, but does not delete the oldest element stored in her memory until the memory reaches a size of m. This means that the memory size is i ($1 \leq i < m$) during the first m iterations and m thereafter.
- **Endorsed memory:** As it occurred in the standard memory, when an agent is paired to play with another agent, she stores in her memory the decision taken by her opponent and 'forgets' the oldest element in her memory, so that the memory size is kept. When the system is started, the agents' memories are initialized with m random values of low, medium and high. However, in this memory configuration, the recent memories have more importance than the older ones. To this aim, a weight is assigned to each memory position. These weights follow an arithmetic progression with common difference 'd'. The value for 'd' is chosen by the user in the interface screen.

3.3. Decision Rule

There are two decision rules available for the agents:

- Demand the option that maximizes the expected benefit (original decision rule).
- Choose the best reply against the opponent's most frequent demand (new feature added to the replication).

The differences between these decision rules are explained below:

- **DEMAND THE OPTION THAT MAXIMIZES THE EXPECTED BENEFIT:**

  Before taking a decision, the agent calculates the probability that her opponent chooses L, M or H. She assumes that the probabilities that her opponent chooses L, M or H are the following:

  - P('opponent chooses L') = sum of elements equal to L in her memory / memory size
  - P('opponent chooses M') = sum of elements equal to M in her memory / memory size
  - P('opponent chooses H') = sum of elements equal to H in her memory / memory size

  where P('opponent chooses x') is the probability that her opponent chooses x.

  Afterwards, she calculates the expected benefit of choosing L, M or H according to the selected payoff matrix (i.e. combinations of values for L, M and H):

  - B(L) = L · P(opponent chooses L) + L·P(opponent chooses M) + L·P(opponent chooses H)
  - B(M) = M · P(opponent chooses L) + M·P(opponent chooses M) + 0·P(opponent chooses H)
  - B(H) = H · P(opponent chooses L) + 0·P(opponent chooses M) + 0·P(opponent chooses H)

    where: B(x) is the agent's expected payoff if she demands x.

  Finally, she chooses the maximum of the three expressions above, as it maximizes her expected benefit.

- **CHOOSE THE BEST REPLY AGAINST THE OPPONENT'S MOST FREQUENT DEMAND**

  In this case, the agent also calculates the probability that her opponent chooses L, M or H in the same way as it did in the original decision rule:

  - P(opponent chooses L) = sum of elements equal to L in her memory / memory size
  - P(opponent chooses M) = sum of elements equal to M in her memory / memory size
  - P(opponent chooses H) = sum of elements equal to H in her memory / memory size

  However, in this case, the agent calculates which of the three options above is the most likely, and then, she chooses the best reply against the expected opponent's decision.

  This is to say:

  - If the agent expects that his opponent will choose L, she decides to choose H (as it is the best reply against L).
  - If the agent expects that his opponent will choose M, she decides to choose M (as it is the best reply against M).
  - If the agent expects that his opponent will choose H, she decides to choose L (as it is the best reply against H).

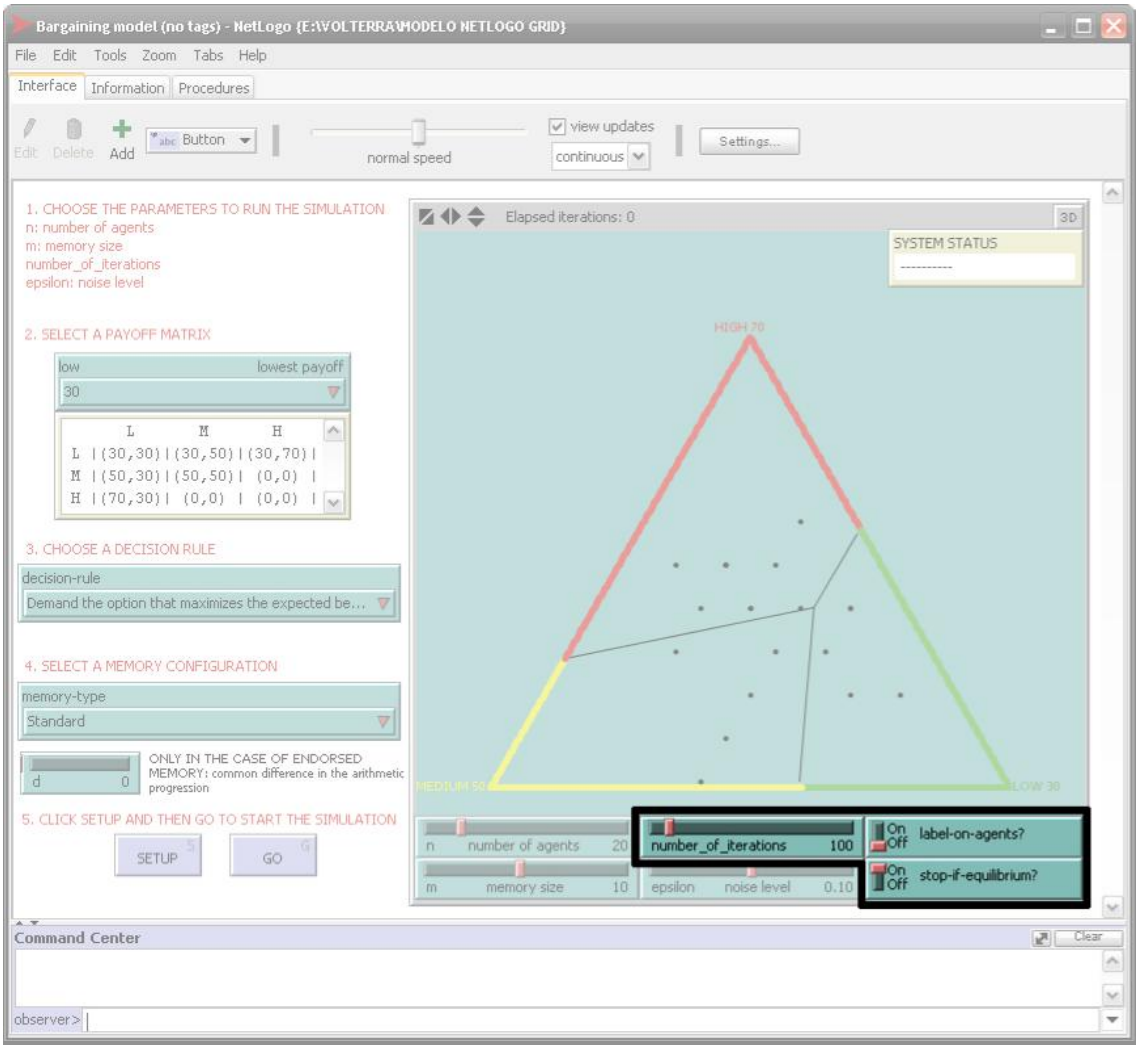
4. SETTING THE PARAMETERS BEFORE STARTING THE SIMULATION

The user can adjust several parameters in the interface of the application. Some of these parameters control the simulation; others are useful to change the contour conditions and some can be used to change the behaviour of the system.

4.1. Parameters to Control the Simulation

The user can change some of the parameters that control the simulation:

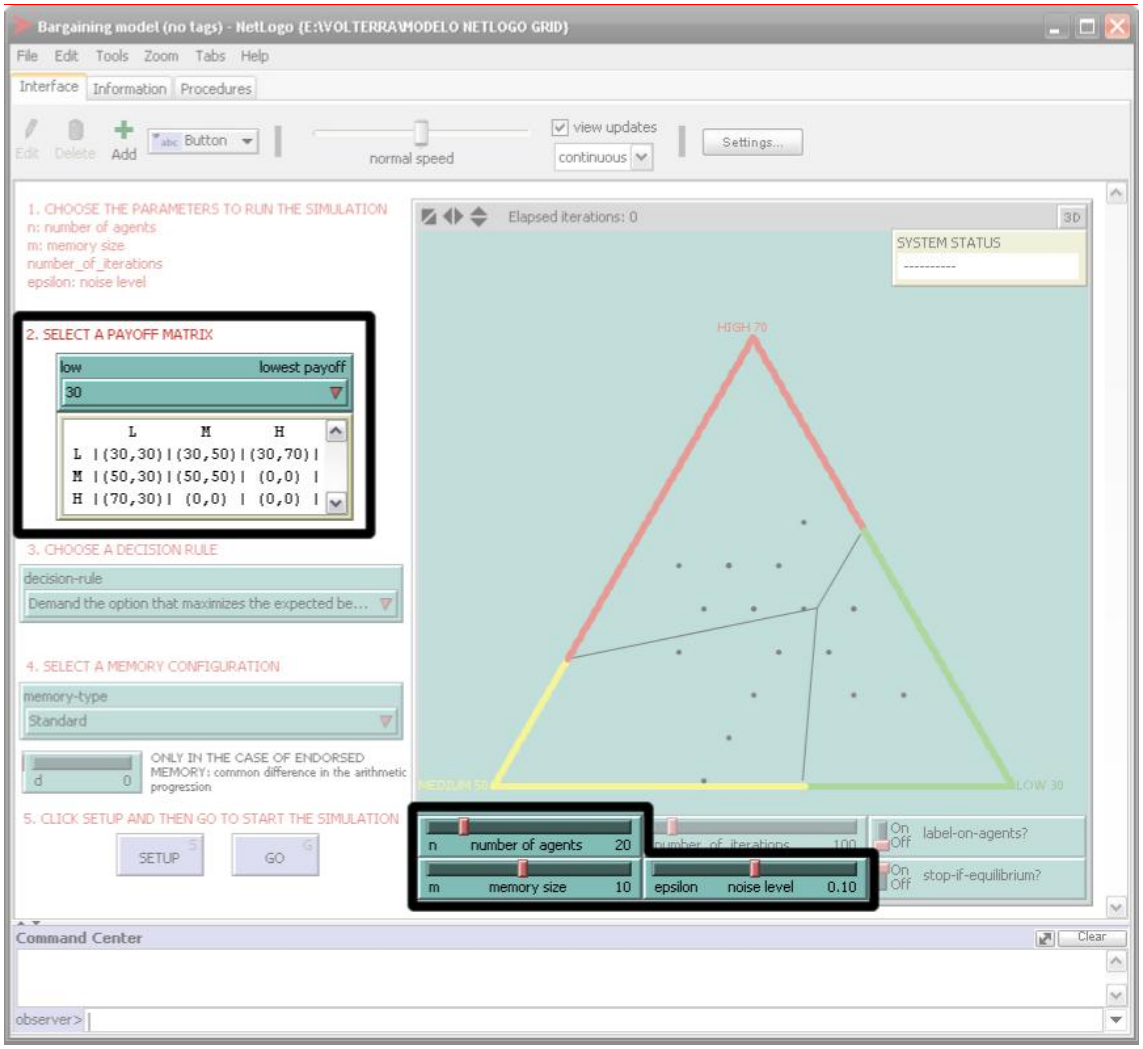
- **number_of_iterations:** This is the number of matches that each agent plays during the simulation. We say that an iteration is completed when all the agents have played once.
- **label_on_agents?:** When this switch is activated, each agent displays her id. This is especially useful in the tag model, where an agent is placed in a different position in the intratype simplex and the intertype simplex.
- **stop_if_equilibrium?** If this switch is activated, the simulation stops when the system reaches a particular scenario[5] (fractious state or equitable equilibrium). If it is switched off, the simulation continues until the specified number_of_iterations is reached.



[5] These particular scenarios are described later on this document

4.2. Parameters to Change the Contour Conditions

- **n:** number of agents.
- **m:** agent's memory size.
- **epsilon:** Noise level in the system. This parameter represents the probability that the agent takes a decision at random instead of using the specified decision rule.
- **low** (lowest payoff): The user selects the value assigned to low (i.e. the lowest reward) and the application generates the corresponding payoff matrix: The value assigned to medium is always 50 and the value assigned to H is 100 – low. The payoff matrix is shown in the interface screen. Notice that the decision borders change according to the selected payoff matrix[6].

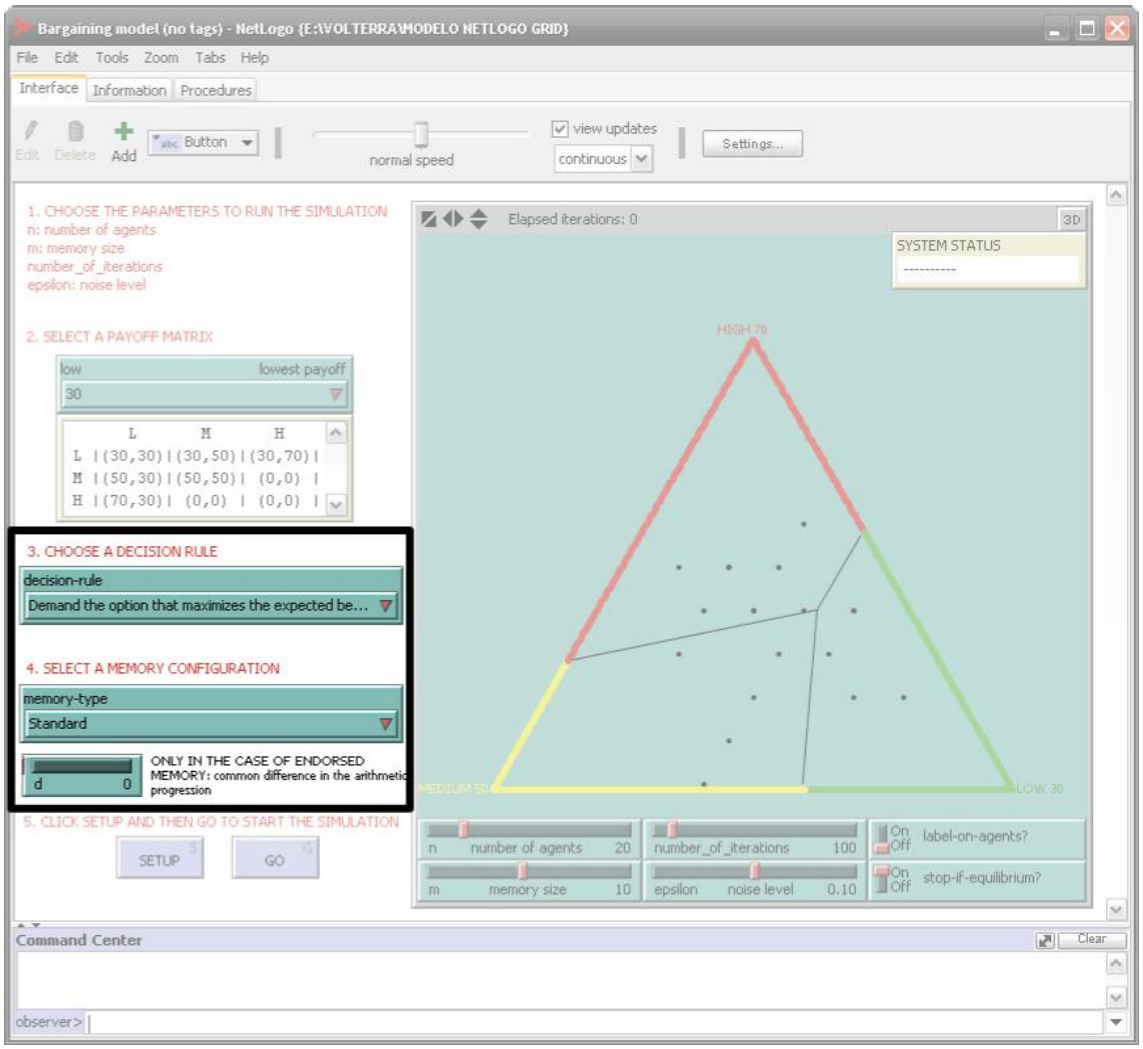


[6] Only if the selected decision rule is 'Demand the option that maximizes the expected benefit (original decision rule).

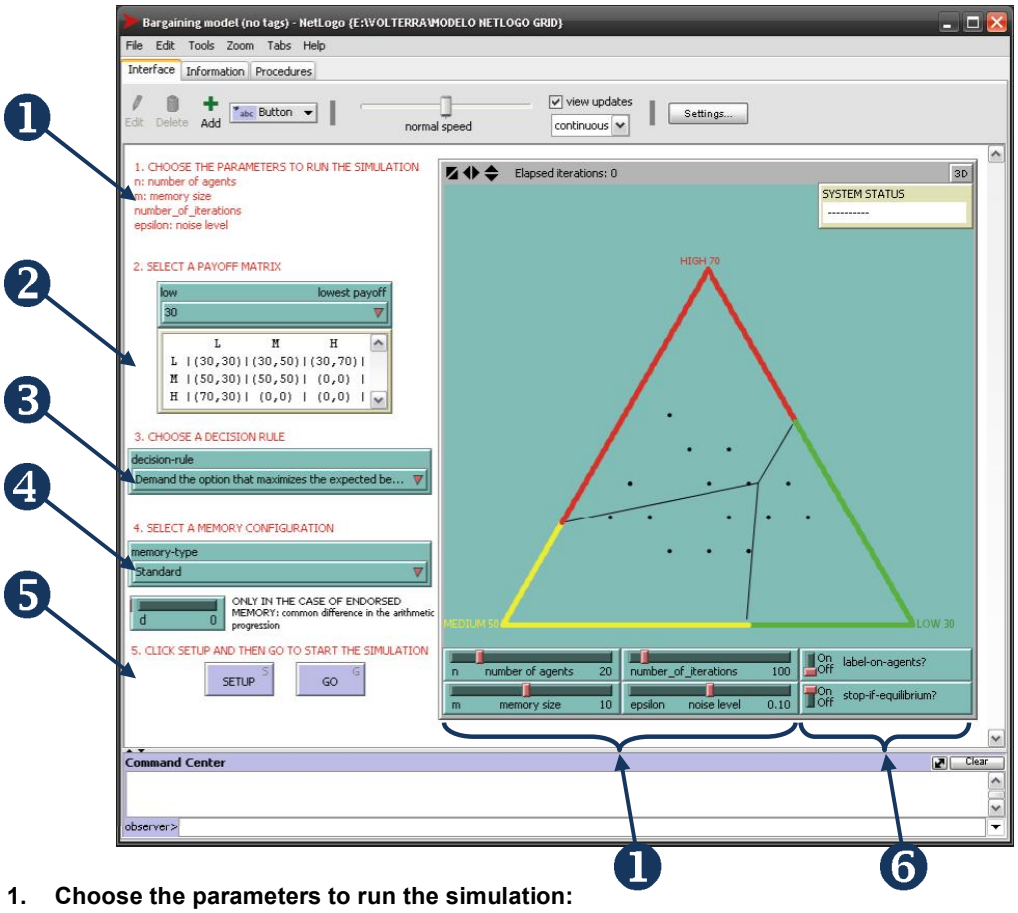
4.3. Parameters that change the behaviour of the system

- **decision-rule**: This button lets the user choose the decision rule that the agents use. There are two possible options:
    - Demand the option that maximizes the expected benefit. This decision rule is used in the original model.
    - Choose the best reply against the opponent's most frequent demand:

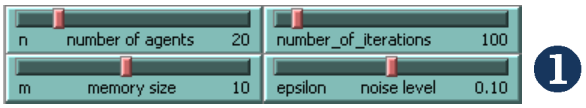
- **memory-type**: This button allows to choose among three possible memory configurations:
    - Standard memory
    - Progressive memory
    - Endorsed memory (in this case we have to select the common difference of the arithmetic progression).



5. HOW TO RUN THE MODEL

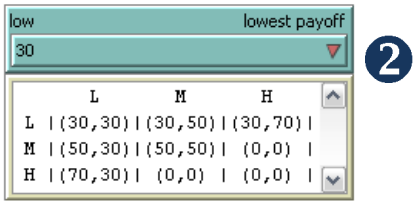


1. **Choose the parameters to run the simulation:**
   - n: number of agents
   - m: memory size
   - number of iterations
   - epsilon: noise level

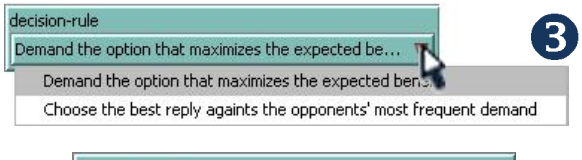


2. **Select a payoff matrix**

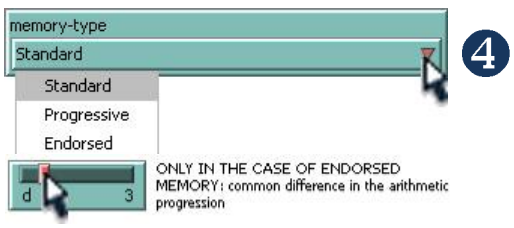
   Choose the value for low. The corresponding payoff matrix will be displayed as soon as you press the setup button.



3. **Choose a decision rule:**
   - Demand the option that maximizes the expected benefit.
   - Choose the best reply against the opponent's most frequent demand.



4. **Choose a memory configuration:**
   - Standard
   - Progressive
   - Endorsed (in this case select the common difference of the arithmetic progression).

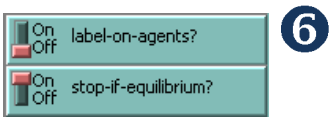


5. **Click setup and then go.**

   Use the SETUP button to initialize the system,

   and then use the GO button to begin the execution.

6. **In addition, you can:**
   - Display the agents' id: use the 'label-on-agents?' slider.
   - Decide whether to stop or not the simulation when a concrete scenario is reached.



6. STOP CONDITIONS AND INTERESTING SCENARIOS

6.1. One-Agent Type

We distinguish two interesting scenarios [3]:

- **EQUITABLE EQUILIBRIUM:** In this state, all the agents have found frequent demands of M in the past, and they assume that M is the best response. Because all the agents demand M, all the pie is shared out among the players, which means that the system has reached an efficient state.

  We consider that the system has reached an equitable equilibrium when *all the agents have, at least, (1-ε)·m elements equal to M in their memories.*

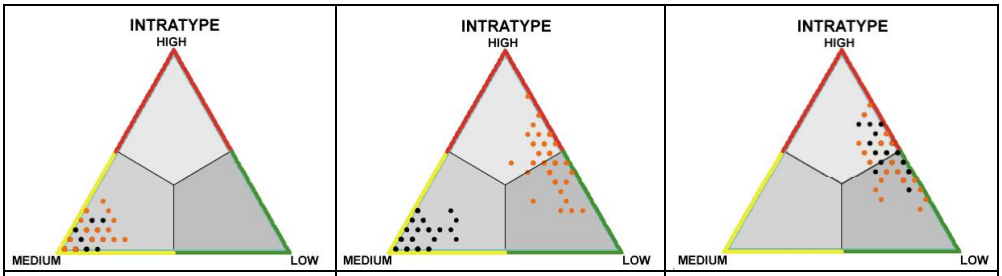
- **FRACTIOUS STATE:** In this case, all the agents are aggressive or passive (most of them select L or H; M is hardly chosen) and no equilibrium is reached.

  We consider that the system has reached a fractious state when *all the agents have, at most, ε ·m elements equal to M in their memories.*

6.2. Two-Agent Types (THE TAG MODEL)

We distinguish up to five different evolutions in the system [3]:

- **INTRATYPE MATCHES (matches between agents with the same tag)**



INTRATYPE EQUITABLE EQUILIBRIUM	INTRATYPE SEGREGATION	INTRATYPE FRACTIOUS STATE

  - **INTRATYPE EQUITABLE EQUILIBRIUM:** All the agents (no matter their tag) reach an equitable equilibrium. The stop condition for this scenario is the following:

    all the agents have, at least, (1-ε)·m elements equal to M in their memories

  - **INTRATYPE SEGREGATION:** The agents of one tag (colour) reach a fractious state and the agents with the other tag (colour) reach a fractious state. The stop condition for this situation is explained below:

    all the dark agents have, at least, (1-ε)·m elements equal to M in their memories and all the light agents have, at most, ε ·m elements equal to M in their memories

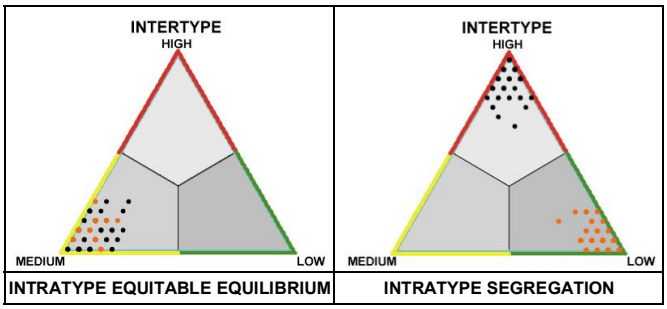
    or

    all the light agents have, at least, (1-ε)·m elements equal to M in their memories and all the dark agents have, at most, ε ·m elements equal to M in their memories

  - **INTRATYPE FRACTIOUS STATE:** All the agents (no matter the colour) reach a fractious state. The stop condition for this is the following:

    all the agents have, at most, ε ·m elements equal to M in their memories.

- **INTERTYPE MATCHES: (matches between agents with different tag)**



INTRATYPE EQUITABLE EQUILIBRIUM	INTRATYPE SEGREGATION

  - **INTERTYPE EQUITABLE EQUILIBRIUM:** All the agents (no matter their tag) reach an equitable equilibrium. The stop condition for this scenario is the following:

    all the agents have, at least, (1- ε)·m elements equal to M in their memories

  - **INTERTYPE SEGREGATION:** All the agents with one tag (colour) demand H and all the agents with the other tag (colour) demand L. The stop condition is the following:

    all the dark agents have, at least, (1- ε)·m elements equal to H in their memories and all the light agents have, at least, (1- ε)·m elements equal to L in their memories

    or

    all the dark agents have, at least, (1- ε)·m elements equal to L in their memories and all the light agents have, at least, (1- ε)·m elements equal to H in their memories.

FUTURE EXTENSIONS

We are currently working in playing the game in a 2D grid and with different social networks topologies, to study how the segregation can affect/be affected when agents are not randomly paired. A preview of this application is available at http://social-simulation.blogspot.com/2009_04_01_archive.html

REFERENCES

1. Axtell, R.; Epstein, J.M. and Young, P.(2000). The Emergence of Classes in a Multi-Agent Bargaining Model. Brookings Institution - Working Papers No.9. February 2000.

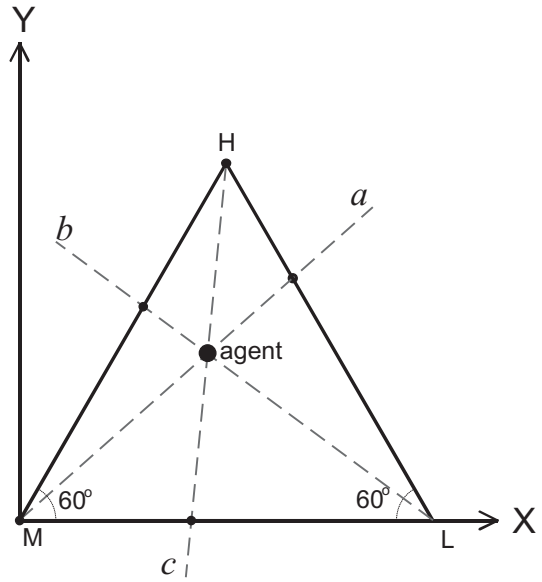
2. Poza, D., Villafáñez, F., and Pajares J. (2009). "Impact of tag recognition in economic decisions". In Hernández, Posada and López-Paredes (Eds): Artificial Economics. The Generative Method in Economics, LNEMS Vol. 631. Springer.

3. Poza, D., Villafáñez, F., Pajares J., López-Paredes, A. And Hernández, C. (2009). "New insights on the Emergence of Classes Model". Proceedings ESSA2009 Conference. Surrey.

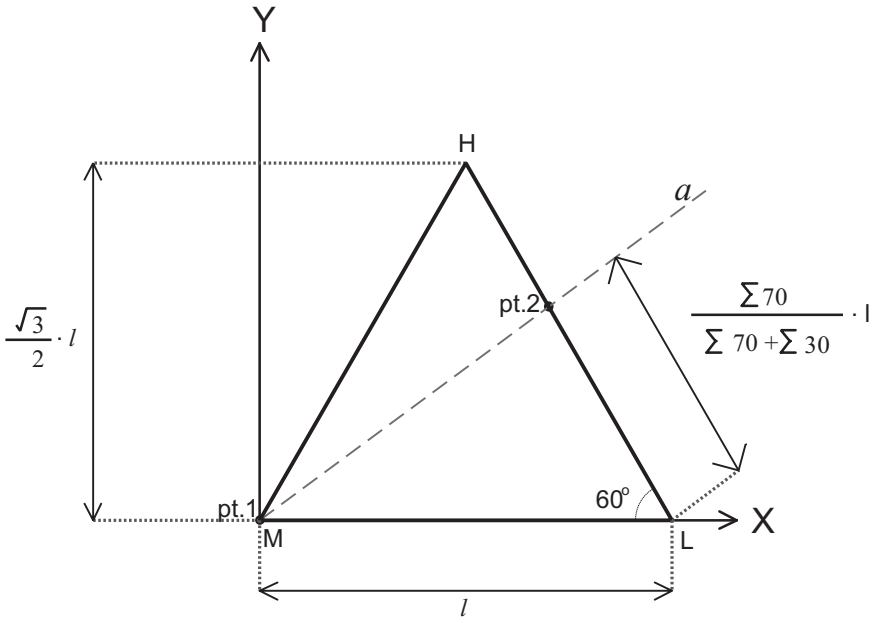
APPENDIX I. HOW TO PLACE THE AGENTS ON THE SIMPLEX

The aim of this appendix is to show the steps we have followed to obtain the coordinates of an agent as a function of the number of elements equal to 30, 50 and 70 stored in her memory.

Let $\Sigma 30$ be the sum of the number of elements equal to 30 in the memory vector.
Let $\Sigma 50$ be the sum of the number of elements equal to 50 in the memory vector.
Let $\Sigma 70$ be the sum of the number of elements equal to 70 in the memory vector.
Let *l* be the length of the side of the triangle.



We know the following information about the line *a:*



-It passes through the point 1: $(0,0)$

-It passes through the point 2: $\left(l - \frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\cos(60^\circ)\cdot l, \frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\sin(60^\circ)\cdot l\right)$

The equation of a generic line is:

$$x = a + b\cdot y \qquad (1)$$

Since the point 1 belongs to the line ***a***, it fulfils its equation. If we replace the value of the point 1 in the equation (1), then we get $a = 0$. If we replace the obtained value of ***a*** in the equation (1), we obtain the following equation for the line ***a***:

$$x = b\cdot y \qquad (2)$$

In addition, the point 2 also fulfils the equation 2, as it belongs to that line. If we replace the value of the point 2 in the equation (2) we obtain the following expression:

$$l - \frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\cos(60^\circ)\cdot l = b\cdot\frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\sin(60^\circ)\cdot l \qquad (3)$$

If we isolate ***b*** in the expression above, we obtain the following value for b:

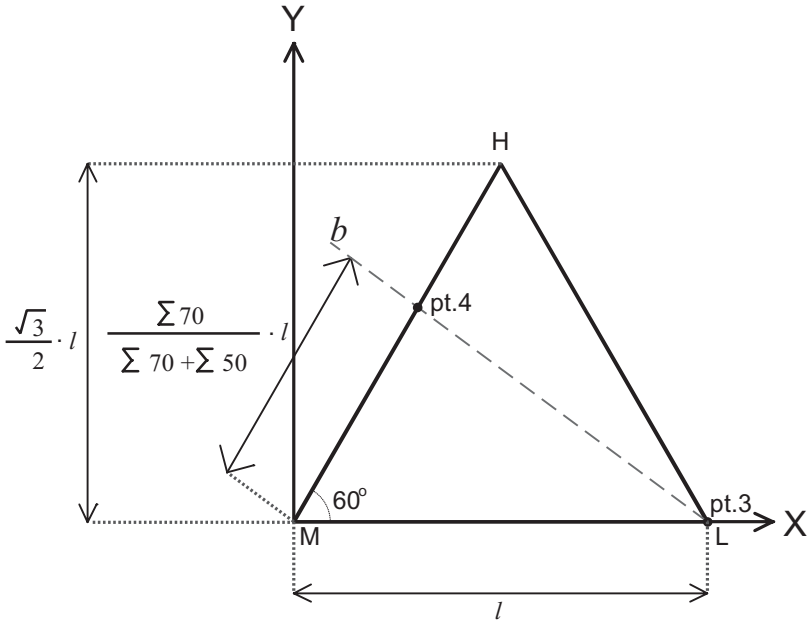
$$b = \frac{1}{\frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\sin(60^\circ)} - \frac{\cos(60^\circ)}{\sin(60^\circ)} = \frac{1}{\frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\sin(60^\circ)} - \frac{1}{\tan(60^\circ)} \qquad (4)$$

Finally, in we replace the value of b (4) in the expression (2), we obtain the equation of the line ***a***:

$$x = \left[\frac{1}{\frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\sin(60^\circ)} - \frac{1}{\tan(60^\circ)}\right]\cdot y \qquad (5)$$

Equation of the line *a*

We know the following information about the line *b:*



- It passes through the point 3: $(l,0)$
- It passes through the point 4:
  $\left(\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\cos(60^\circ)\cdot l, \frac{\Sigma 70}{\Sigma 50+\Sigma 70}\sin(60^\circ)\cdot l\right)$

Let us use the generic equation of a line (1). The point 3 belongs to the line *b*, this is why it fulfils its equation. Therefore, if we replace its value in the equation (1) we obtain the following value for a:

$$a = l \qquad (6)$$

The point 4 also belongs to the line b. Consequently, it fulfils its equation. If we replace the value of the point 4 in the equation (1), we get the following expression:

$$\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\cos(60^\circ)\cdot l = a + b\cdot\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\sin(60^\circ)\cdot l \qquad (7)$$

If we replace the value of *a* (6) in the expression (7), then we can isolate the value of *b*:

$$b = \frac{\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\cos(60^\circ)\cdot l - l}{\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\sin(60^\circ)\cdot l} = \frac{\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\cos(60^\circ)\cdot -1}{\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\sin(60^\circ)} = \frac{1}{\tan 60^\circ} - \frac{1}{\frac{\Sigma 70}{\Sigma 50+\Sigma 70}\sin(60^\circ)} \qquad (8)$$

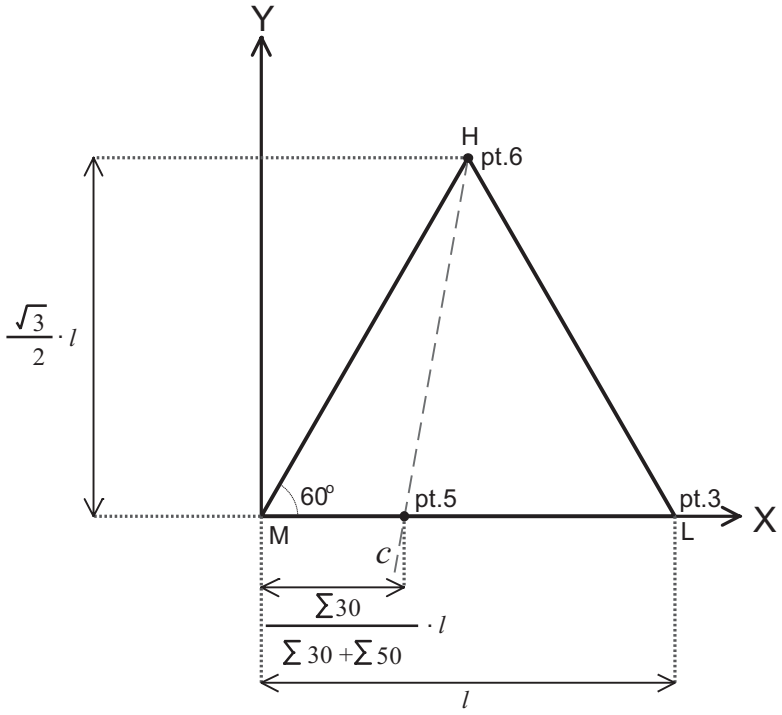
Let us replace the obtained values of a (6) and b (8) in the generic expression of

a line (1) and we obtain the equation of the line *b*:

$$x = l + \frac{1}{\tan 60^{o}} - \frac{1}{\frac{\Sigma 70}{\Sigma 50 + \Sigma 70}\sin(60^{o})} \cdot y \qquad (9)$$

Equation of the line *b*

We know the following information about the line c:



- It passes through the point 5: $\left(0, \frac{\Sigma 30}{\Sigma 30 + \Sigma 50} \cdot l\right)$ (10)
- It passes through the point 6: $\left(\frac{l}{2}, \frac{\sqrt{3}}{2} \cdot l\right)$ (11)

Since the point 5 belongs to the line *c*, it fulfils its equation. If we replace the value of the point 5 in the equation (1), we obtain the following:

$$0 = a + b \cdot \frac{\Sigma 30}{\Sigma 30 + \Sigma 50} \cdot l \qquad (12)$$

Moreover, the point 6 also fulfils the equation of the line *c*, as it belongs to that line. Let us replace the coordinates of the point 6 in the equation (1):

$$\frac{l}{2} = a + b \cdot \frac{\sqrt{3}}{2} \cdot l \qquad (13)$$

The equations (12) and (13) make a system of two equations and two variables. If we solve this system, we obtain the following values for *a* and *b*:

$$a = \frac{1}{2} \cdot \frac{\frac{\Sigma 30}{\Sigma 30 + \Sigma 50}}{\frac{\Sigma 30}{\Sigma 30 + \Sigma 50} - \frac{\sqrt{3}}{2}} \cdot l \qquad (14)$$

$$b = -\frac{1}{2} \cdot \frac{1}{\frac{\Sigma 30}{\Sigma 30 + \Sigma 50} - \frac{\sqrt{3}}{2}} \qquad (15)$$

Finally, we replace the values *a* and *b* in the generic expression of a line (1) and obtain the equation of the line *c*:

$$x = \frac{1}{2} \cdot \frac{\frac{\Sigma 30}{\Sigma 30 + \Sigma 50}}{\frac{\Sigma 30}{\Sigma 30 + \Sigma 50} - \frac{\sqrt{3}}{2}} \cdot l + -\frac{1}{2} \cdot \frac{1}{\frac{\Sigma 30}{\Sigma 30 + \Sigma 50} - \frac{\sqrt{3}}{2}} \cdot y \qquad (16)$$

Equation of the line *c*

Let us consider a player with $\Sigma 30$ elements equal to 30 in her memory; $\Sigma 50$ elements equal to 50 and $\Sigma 70$ elements equal to 70. Its position in the simplex will be determined by the intersection of the three lines (*a*, *b* and *c*) which equations we have just calculated.

First, let us calculate the intersection of the lines *a* and *b.* To this aim, we set the expression (5) equal to (9) and get the following coordinates:

$$x = \frac{\left[\frac{1}{\left[\frac{\Sigma 70}{\Sigma 30 + \Sigma 70}\right] \cdot \sin 60^{\circ}} - \frac{1}{\tan 60^{\circ}}\right]}{\left[\frac{1}{\frac{\Sigma 70}{\Sigma 30 + \Sigma 70}} + \frac{1}{\frac{\Sigma 70}{\Sigma 50 + \Sigma 70}}\right] \cdot \frac{1}{\sin 60^{\circ}} - \frac{2}{\tan 60^{\circ}}} \cdot l \qquad (17)$$

$$y = \frac{1}{\left[\frac{1}{\frac{\Sigma 70}{\Sigma 30 + \Sigma 70}} + \frac{1}{\frac{\Sigma 70}{\Sigma 50 + \Sigma 70}}\right] \cdot \frac{1}{\sin 60^{\circ}} - \frac{2}{\tan 60^{\circ}}} \cdot l \qquad (18)$$

In the case that $\Sigma 70 = 0$ (there are no elements equal to 70 in the agent's memory), the equations (17) and (18) are no longer valid as the denominator becomes zero.

In this case, we know that the agent will be placed across the horizontal side of the simplex (segment M – L). Therefore, the coordinates of this player and the point 5 will match. This is to say:

$$x = \frac{\Sigma 30}{\Sigma 30 + \Sigma 50} \cdot l \qquad (19)$$

$$y = 0 \qquad (20)$$

Important notice: In the original model, the values for 'low' (L), 'medium' (M) and 'high' (H) were fixed. All the calculated expressions match the original model: 'low' is 30%; 'medium' is 50% and 'high' is 70%. We can generalize the calculated expressions for other combination of rewards by replacing the following terms:

$$\Sigma 30 \leftarrow\rightarrow \Sigma L$$
$$\Sigma 50 \leftarrow\rightarrow \Sigma M$$
$$\Sigma 70 \leftarrow\rightarrow \Sigma H$$

APPENDIX II. HOW TO BUILD THE DECISION BORDERS

The layout of the decision borders depends on the decision rule:

1. CHOOSE THE BEST REPLY AGAINST THE OPPONENTS' MOST FREQUENT DEMAND:

In the equilibrium point, an agent has the same tendency to demand L, M or H, as the number of elements equal to L, M and H in her memory match. This is to say:

$$\Sigma 30 = \Sigma 50 = \Sigma 70 \qquad (21)$$

If we take into account the expression (21) in the expressions (17) and (18), we obtain the coordinates of the equilibrium point:

$$x = \frac{1}{2} \cdot l \qquad (22)$$

$$y = \frac{\sqrt{3}}{6} \cdot l \qquad (23)$$

- **Calculate the coordinates of the point of the simplex in which there is no tendency to choose L and there is the same tendency to choose M or H:**

  This case emerges when the agent's memory fulfils the following conditions:

$$\Sigma 30 = 0 \qquad (24)$$
$$\Sigma 50 = \Sigma 70 \qquad (25)$$

  If we replace the values of the expressions (24) and (25) in the equations (17) and (18), we obtain the coordinates of the point that we are searching:

$$x = \frac{1}{4} \cdot l \qquad (26)$$

$$y = \frac{\sqrt{3}}{4} \cdot l \qquad (27)$$

- **Calculate the coordinates of the point of the simplex in which there is no tendency to choose M and there is the same tendency to choose L or H:**

  In this case, the agent's memory fulfils the following condition:

$$\Sigma 50 = 0 \quad (28)$$
$$\Sigma 30 = \Sigma 70 \quad (29)$$

  If we replace the values of the expressions (28) and (29) in the equations (17) and (18), we obtain the coordinates of the point that we are searching:

$$x = \frac{3}{4} \cdot l \quad (30)$$

$$y = \frac{\sqrt{3}}{4} \cdot l \quad (31)$$

- **Calculate the coordinates of the point of the simplex in which there is no tendency to choose H and there is the same tendency to choose L or M:**

  In this case, the agent's memory fulfils the following condition:

$$\Sigma 70 = 0 \quad (32)$$
$$\Sigma 30 = \Sigma 50 \quad (33)$$

  If we replace the values of the expressions (32) and (33) in the equations (17) and (18), we obtain the coordinates of the point that we are searching:

$$x = \frac{1}{2} \cdot l \quad (34)$$

$$y = 0 \quad (35)$$

2. DEMAND THE OPTION THAT MAXIMIZES THE EXPECTED BENEFIT

In the equilibrium point, an agent will have the same tendency to choose L, M or H when the benefits of choosing L, M and H match.

In this case, the benefit of choosing 30, 50 and 70 will be the following:

$$B(30) = 30 \quad (36)$$
$$B(50) = 50 \cdot (\Sigma 30 + \Sigma 50) / m \quad (37)$$
$$B(70) = 70 \cdot \Sigma 30 / m \quad (38)$$

At this point, the aim is to calculate the values of $\Sigma 30$, $\Sigma 50$ and $\Sigma 70$ that make the expressions (36), (37) and (38) match. Once these values are obtained, we will replace them in the equations (17) y (18) and obtain the coordinates of the equilibrium point.

To make this process simpler, we will call:

$$A = \Sigma 30 / m \quad (39)$$

$$B = \Sigma 50 / m \quad (40)$$

$$C = \Sigma 70 / m \quad (41)$$

Notice that the sum of the expressions (39), (40) and (41) is 1:

$$A + B + C = \Sigma 30 / m + \Sigma 50 / m + \Sigma 70 / m = m / m = 1 \quad (42)$$

Following this notation, the equations (36), (37) y (38) will turn into the following expressions:

$$B(30) = 30 \quad (43)$$

$$B(50) = 50 \cdot (A + B) \quad (44)$$

$$B(70) = 70 \cdot A \quad (45)$$

In the equilibrium point, the expected benefits of choosing 30, 50 and 70 match. Therefore, in this point, the expressions (43), (44) and (45) also match:

$$30 = 50(A + B) = 70A \quad (46)$$

If we set (43) equal to (45):

$$30 = 70 \cdot A$$

If we isolate A, we obtain the following:

$$A = \frac{3}{7} \quad (47)$$

Now, let us set the expression (43) equal to (44):

$$30 = 50 \cdot (A + B)$$

If we isolate B in the expression above and replace the value of A (46), we

obtain the following:

$$B = \frac{3}{5} - \frac{3}{7} = \frac{6}{35} \tag{48}$$

Finally, we obtain the value of C. To this aim, we take into account the equation (42)

$$C = 1 - A - B \tag{49}$$

Let us replace the obtained values for A and B (47) and (48) respectively in the equation (49). We obtain the following:

$$C = 1 - \frac{3}{5} - \frac{6}{35} = \frac{3}{7} \tag{50}$$

If we replace (39), (40) and (41) in the expressions (17) and (18), we obtain the coordinates of a point as a function of *A*, *B* and *C*:

$$x = \frac{\left[\frac{1}{\left[\frac{C}{A+C}\right]\cdot\sin 60^{o}} - \frac{1}{\tan 60^{o}}\right]}{\left[\frac{1}{\frac{C}{A+C}} + \frac{1}{\frac{C}{B+C}}\right]\cdot\frac{1}{\sin 60^{o}} - \frac{2}{\tan 60^{o}}}\cdot l \tag{51}$$

$$y = \frac{1}{\left[\frac{1}{\frac{C}{A+C}} + \frac{1}{\frac{C}{B+C}}\right]\cdot\frac{1}{\sin 60^{o}} - \frac{2}{\tan 60^{o}}}\cdot l \tag{52}$$

If we replace the obtained values for *A* (47), *B* (48) and *C* (50) in the equations (51) and (52), we obtain the equilibrium, point in which the benefit is the same for the three possible options (30, 50 and 70):

- **Calculate the coordinates of the point of the simplex in which the benefit of choosing L equals the benefit of choosing H:**

If the benefit of choosing 30 and 70 match, this point fulfils the following condition:

$$B(30) = B(70) \quad (53)$$

If we replace (43) and (45) in the expression (53), we obtain the equation below:

$$70 \cdot A = 30 \quad (54)$$

Moreover, because this point is on the right side of the simplex, there are no elements equal to 50 in the agent's memory:

$$\Sigma 50 = 0 \quad (55)$$

If we replace the value of (40) in (55) we get:

(56)

Therefore, to calculate this point, we have a system of three equations (42), (54) and (56) and three variables (A, B and C):

$$A + B + C = 1$$

$$B = 0$$

$$70 \cdot A = 30$$

If we isolate the three variables, we obtain the following values:

$$A = \frac{3}{7} \qquad B = 0 \qquad C = 1 - \frac{3}{7} \quad (57)$$

To obtain the coordinates of this point, we replace (57) in (17) and (18).

- **Calculate the coordinates of the point of the simplex in which the benefit of choosing M equals the benefit of choosing H:**

If the benefit of choosing 50 and 70 match, this point fulfils the following condition:

$$B(50) = B(70) \quad (58)$$

If we replace (44) and (45) in the expression (58), we obtain the equation below:

$$50\cdot(A+B)=70\cdot A \quad (59)$$

Moreover, because this point is on the left side of the simplex, there are no elements equal to 70 in the agent's memory:

$$\Sigma 70=0 \quad (60)$$

If we replace the value of (41) in (60) we get:

$$C=0 \quad (61)$$

Therefore, to calculate this point, we have a system of three equations (42), (59) and (61) and three variables (A, B and C):

$$A+B+C=1$$

$$C=0$$

$$50\cdot(A+B)=70\cdot A$$

If we solve the system, we obtain the following values:

$$A=\frac{5}{7} \qquad B=1-\frac{5}{7} \qquad C=0 \quad (62)$$

To obtain the coordinates of this point, we replace (62) in (17) and (18).

- **Calculate the coordinates of the point of the simplex in which the benefit of choosing M equals the benefit of choosing L:**

If the benefit of choosing 50 and 30 match, this point fulfils the following condition:

$$B(50)=B(30) \quad (63)$$

If we replace (43) and (44) in the expression (63), we obtain the equation below:

$$50\cdot(A+B)=L \quad (64)$$

Moreover, because this point is on the horizontal side of the simplex, there are no elements equal to 30 in the agent's memory:

$$\Sigma 30 = 0 \qquad (65)$$

If we replace the value of (39) in (65) we get:

$$A = 0 \qquad (66)$$

Therefore, to calculate this point, we have a system of three equations (42), (54) and (66) and three variables (A, B and C):

$$A + B + C = 1$$

$$A = 0$$

$$50 \cdot (A + B) = L$$

If we solve the system, we obtain the following values:

$$A = 0 \qquad B = \frac{3}{5} \qquad C = 1 - \frac{3}{5} \qquad (67)$$

To obtain the coordinates of this point, we replace (67) in (17) and (18).

Important notice: To make the analysis simpler, we considered that the simplex is an equilateral triangle which left vertex is set on the origin of coordinates.

However, in the application, the simplex is centred on the origin of coordinates. Therefore, we need to shift the coordinates (in the x and y axis) in the application as follows:

$$x' = x - \frac{1}{2} \cdot l$$

$$y' = y - \frac{\sqrt{3}}{4} \cdot l$$

APPENDIX III. SOURCE CODE

1-AGENT TYPE BARGAINING MODEL.

```
breed [agents agent]

agents-own [memory]

globals
[
   options                              ;; list that contains the following values - according to the chosen payoff matrix:
               [low medium high]
   medium                               ;; numeric value assigned to the 'medium' demand
   high                                 ;; numeric value assigned to the 'high' demand
   simplex-side-length                    ;; length of the simplex side
   agents-list                           ;; list that contains all the agents in the population
   elapsed-iterations                     ;; number of elapsed iterations since the simulation was started
   equitable-equilibrium-stop-condition      ;; conditions for equitable-equilibrium
   fractious-state-stop-condition           ;; conditions for fractious state
   equitable-equilibrium?                  ;; boolean variable that will become true if and only if the system has reached
               an equitable equilibrium IN THE CURRENT ITERATION
   equitable-equilibrium-at-least-once?      ;; boolean variable that will become true if and only if the system has reached
               an equitable equilibrium AT LEAST ONCE DURING THE SIMULATION
   fractious-state-at-least-once?           ;; boolean variable that will become true if and only if the system has reached
               a fractious state IN THE CURRENT ITERATION
   fractious-state?                       ;; boolean variable that will become true if and only if the system has reached
               an equitable equilibrium AT LEAST ONCE DURING THE SIMULATION
   system-status                         ;; this variable contains the message shown in the system-status monitor, in the
               interface scrren.
   endorsed-memory-weights                 ;; this list contains the weights assigned to each term in the endorsed memory
               set
]

to setup

   print ""                             ;; leave a black line in the command center
   clear-all                            ;; reset variables

   ;; DEFINE THE REWARDS ACCORDING TO THE SELECTED PAYOFF MATRIX (the value for the lowest reward -low- is chosen in the
               interface screen)
   set high 100 - low                    ;; set the value assigned to high
   set medium 50                        ;; set the value assigned to medium
   set options (list low medium high)       ;; create a list which contains the values assigned to low, medium and high
               according to the selected payoff matrix

   ;; PRINT THE PAYOFF MATRIX ACCORDING TO THE SELECTED VALUE FOR LOW
   output-print "      L      M      H    "
   output-type " L |(" output-type low output-type "," output-type low output-type ")|(" output-type low output-type ","
               output-type medium output-type ")|(" output-type low output-type "," output-type high output-print ")|"
   output-type " M |(" output-type medium output-type "," output-type low output-type ")|(" output-type medium output-type
               "," output-type medium output-print ")| (0,0) |"
   output-type " H |(" output-type high output-type "," output-type low output-print ")| (0,0) | (0,0) |"

   ;; CREATE THE SIMPLEX
   set simplex-side-length 150             ;; set the simplex size
   create-simplex                        ;; call the procedure 'create-simplex'

   ;; CREATE A POPULATION OF n AGENTS
   create-agents n
   [
     set shape "dot"                                          ;; set the agent's shape
     set size 4                                              ;; set the agent's size
     set color black                                          ;; set the agent's colour
     if label-on-agents? [set label who]                          ;; create a label with the id of each agent
     if memory-type = "Standard" [set memory n-values m [one-of options]]    ;; create an m-size memory with random values
               for each agent
     if memory-type = "Endorsed" [set memory n-values m [one-of options]]    ;; create an m-size memory with random values
               for each agent
     if memory-type = "Progressive" [set memory n-values 1 [one-of options]]  ;; initialize the memory with just one value
               choosen at random
     set xcor place-agents-xcor (memory)                          ;; place the agents in the simplex according to
               their memory status (x-coordinate)
     set ycor place-agents-ycor (memory)                          ;; place the agents in the simplex according to
```

```
               their memory status (y-coordinate)
   ]

   ;; SET THE INITIAL VALUE OF SOME OF THE GLOBALS VARIABLES
   set system-status "----------"                          ;; this message is shown in the system-status monitor until an
               equitable equilibrium or a fractious state is reached
   set equitable-equilibrium? false                        ;; at first, there is not an equitable equilibrium in the system
   set fractious-state? false                              ;; at first, there is not a fractious state in the system
   set equitable-equilibrium-at-least-once? false             ;; at first, there is not an equitable equilibrium in the
               system
   set fractious-state-at-least-once? false                  ;; at first, there is not a fractious state in the system

   ;; CREATE A LIST WITH ALL THE AGENTS IN THE POPULATION
   set agents-list sort agents                             ;; turns the agentset into a list

   ;; CREATE A LIST OF WEIGHTS FOR THE ENDORSED MEMORY
   if memory-type = "Endorsed"
   [
      let weights []        ;; list of weights
      let current-weight 1   ;; lowest weight
      let i 0              ;; counter (loop)
      while [i < m]
      [
         set weights fput current-weight weights  ;; add the current weight to the list of weights
         set current-weight current-weight + d    ;; increase the weight for the next element
         set i i + 1                          ;; increase the loop counter
      ]
      set endorsed-memory-weights weights         ;; update the global variable
   ]

   if memory-type != "Endorsed" [set d 0]          ;; if the memory is not 'endorsed', 'd' (difference common in the arithmetic
               progression which is followed by the weights of each memory position makes no sense here
   if decision-rule = "Choose the best reply againts the opponents' most frequent demand" and memory-type = "Endorsed"
               [beep user-message "This decision rule is not compatible with an endorsed memory" ]  ;; displays a warning
               in a pop-up window
end


to go
   set elapsed-iterations 0                                ;; initialize the elapsed-iterations counter
   while [elapsed-iterations < number_of_iterations]           ;; we run as many iterations as the value of the variable
               number_of_iterations
   [
      let players agents-list                              ;; retrieve the original list of agents
      repeat (length agents-list) / 2                      ;; the number of matches in each iteration is half the number
               of agents
      [
        ;; 1. TAKE TWO PLAYERS AT RANDOM FROM THE LIST OF AGENTS
        let player1 one-of players                          ;; choose one agent from the list of agents at random
        set players remove player1 players                  ;; delete this agent so that it can't be chosen again during
               this iteration
        let player2 one-of players                          ;; choose one agent from the list of agents at random
        set players remove player2 players                  ;; delete this agent so that it can't be chosen again during
               this iteration

       ;; 2. EACH AGENT TAKES A DECISION TAKING INTO ACCOUNT HIS MEMORIES ABOUT PREVIOUS MATCHES
        let player1-decision take-decision ([memory] of player1) ;; player 1 takes his decision
        let player2-decision take-decision ([memory] of player2) ;; player 2 takes his decision

        ;; 3. EACH AGENT STORES THE OPPONENT'S DECISION IN HIS MEMORY
        ask player1 [set memory fput player2-decision memory]    ;; player 1 stores player 2's decision in his memory
        ask player2 [set memory fput player1-decision memory]    ;; player 2 stores player 1's decision in his memory

        if memory-type = "Standard" or memory-type = "Endorsed" [ask player1 [set memory but-last memory]]          ;;
               Standard and endorsed memory: player 1 deletes his oldest value in his memory.
        if memory-type = "Progressive" and elapsed-iterations >= (m - 1) [ask player1 [set memory but-last memory]]   ;;
               Progressive memory: player 1 deletes his oldest value in his memory when the m-size memory is reached

        if memory-type = "Standard" or memory-type = "Endorsed" [ask player2 [set memory but-last memory]]          ;;
               Standard and endorsed memory: player 2 deletes his oldest value in his memory.
        if memory-type = "Progressive" and elapsed-iterations >= (m - 1) [ask player2 [set memory but-last memory]]   ;;
               Progressive memory: player 2 deletes his oldest value in his memory when the m-size memory is reached


        ;; 4. UPDATE THE POSITION OF THE TWO AGENTS IN THE SIMPLEX ACCORDING TO THE NEW MEMORY STATUS
        ask player1 [setxy place-agents-xcor (memory) place-agents-ycor (memory)]   ;; place the agents in the simplex according
```

```
                to their memory status (x-coordinate)
          ask player2 [setxy place-agents-xcor (memory) place-agents-ycor (memory)]   ;; place the agents in the simplex according
                to their memory status (y-coordinate)
        ]
      set elapsed-iterations elapsed-iterations + 1          ;; the current iteration has finished
      tick                                              ;; increase the tick counter (number of elapsed iterations)

      check-stop-conditions                               ;; call the procedure 'check-stop-conditions'

      if equitable-equilibrium? = true and stop-if-equilibrium? [print "* * * SIMULATION FINISHED * * *" stop] ;; if the
                'stop-if-equilibrium?' swith is on, and the system has reached an equitable equilibrium in this iteration,
                the simulation stops now.

      if fractious-state? = true and stop-if-equilibrium? [print "* * * SIMULATION FINISHED * * *" stop]      ;; if the
                'fractious-state?' swith is on, and the system has reached a fractious state in this itaration, the simulation
                stops now.

      if elapsed-iterations = number_of_iterations          ;; when the maximum number of iterations is reached, a message
                is shown and the simulation finishes.
      [
        print "* * * SIMULATION FINISHED: The system has reached the maximum number of iterations. * * *"
        if equitable-equilibrium-at-least-once? = false and fractious-state-at-least-once? = false [set system-status
                "SIMULATION FINISHED" print "* * * No equilibrium was reached in the system. * * *"]
        stop
      ]

    ]

end

to-report take-decision [agents_memory]

  let random-number random-float 1  ;; create a random number
  ifelse random-number < epsilon

  ;; if the random number is lower than epsilon, the decision is taken randomly
  [
    report one-of options
  ]

  ;; if the random number is lower than epsilon, the decision rule is 'rational'
  [
     if decision-rule = "Demand the option that maximizes the expected benefit"
     [
        let probability-opponent-demands-L (length filter [? = low] agents_memory) / m      ;; count the number of appearances
                of L in the agent's memory
        let probability-opponent-demands-M (length filter [? = medium] agents_memory) / m   ;; count the number of appearances
                of M in the agent's memory
        let probability-opponent-demands-H (length filter [? = high] agents_memory) / m     ;; count the number of appearances
                of H in the agent's memory

        let reward-L low     ;; set the reward assigned to low
        let reward-M medium   ;; set the reward assigned to medium
        let reward-H high    ;; set the reward assigned to high

        ;; the agent calculates the expected benefit if he chooses L, M or H
        let expected-benefit-L reward-L * probability-opponent-demands-L + reward-L * probability-opponent-demands-M +
                reward-L * probability-opponent-demands-H
        let expected-benefit-M reward-M * probability-opponent-demands-L + reward-M * probability-opponent-demands-M +
                0 * probability-opponent-demands-H
        let  expected-benefit-H reward-H * probability-opponent-demands-L + 0 * probability-opponent-demands-M + 0 *
                probability-opponent-demands-H

        ;; calculate the best option
        let possible-demands (list expected-benefit-L expected-benefit-M expected-benefit-H) ;; the list possible-demands
                contains: [benefit if the agent chooses L , benefit if the agent chooses M , benefit if the agent chooses
                H]
        let best-option max possible-demands ;; the best option is the highest value in the list possible-demands

        ;; PARTICULAR CASE: TWO OR THREE OPTIONS RESULT IN THE SAME EXPECTED BENEFIT
        if length filter [? = best-option] possible-demands > 1   ;; if two or three options produce the same benefit...
        [
          if item 0 possible-demands = item 1 possible-demands and item 1 possible-demands = item 2 possible-demands [report
                one-of options] ;; benefit (low) = benefit (medium) = benefit (high) -> choose low, medium or high at random
          if item 0 possible-demands = item 1 possible-demands [report one-of list low medium]      ;; benefit (low) = benefit
                (medium) -> choose low or medium at random
```

```
        if item 0 possible-demands = item 2 possible-demands [report one-of list low high]       ;; benefit (low) = benefit
              (high) -> choose low or high at random
        if item 1 possible-demands = item 2 possible-demands [report one-of list medium high]    ;; benefit (medium) =
              benefit (high) -> choose medium or high at random
    ]

    ;; GENERAL CASE: ONLY ONE OF THE THREE DEMANDS MAXIMIZE THE EXPECTED BENEFIT
    if item 0 possible-demands = best-option [report low]     ;; if the first element in the list of possible demands
            is the best option, then the agent chooses low
    if item 1 possible-demands = best-option [report medium]  ;; if the second element in the list of possible demands
            is the best option, then the agent chooses medium
    if item 2 possible-demands = best-option [report high]    ;; if the third element in the list of possible demands
            is the best option, then the agent chooses high
]

if decision-rule = "Demand the option that maximizes the expected benefit" and memory-type = "Endorsed"
[
    let i 0                                  ;; set a counter for the while loop: 0<=i<memory-size
    let L 0                                  ;; L is the number of appearances of low endorsed by the position that
            they appear in the memory
    let Me 0                                 ;; Me is the number of appearances of medium endorsed by the position
            that they appear in the memory
    let H 0                                  ;; H is the number of appearances of high endorsed by the position that
            they appear in the memory
    let highest-weight 1 + (m - 1) * d           ;; the highest weight is the last term in an arithmetic progression
            with common difference 'd'
    let current-weight highest-weight            ;; the value of current-weight decreases as we move towards the oldest
            values stored in the memory
    while [i < length agents_memory]
    [
       if item i agents_memory = low [set L L + current-weight]      ;; the value of L is increased if the current
            memory position is low; current-weight decreases as we move towards the oldest values stored in the memory
       if item i agents_memory = medium [set Me Me + current-weight]  ;; the value of Me is increased if the current
            memory position is low or medium; current-weight decreases as we move towards the oldest values stored in
            the memory
       if item i agents_memory = high [set H H + current-weight]     ;; the value of L is increased if the current
            memory position is low; current-weight decreases as we move towards the oldest values stored in the memory
       set i i + 1                                  ;; increase the array position
       set current-weight current-weight - d            ;; substract the common difference from the current-weight
            to obtain a new weight for the next iteration
    ]
    let S (2 + (m - 1) * d) / 2 * m

    let probability-opponent-demands-L L / S     ;; count the number of appearances of L in the agent's memory
    let probability-opponent-demands-M Me / S    ;; count the number of appearances of Me in the agent's memory
    let probability-opponent-demands-H H / S     ;; count the number of appearances of H in the agent's memory

    let reward-L low     ;; set the reward assigned to low
    let reward-M medium   ;; set the reward assigned to medium
    let reward-H high    ;; set the reward assigned to high

    ;; the agent calculates the expected benefit if he chooses L, M or H
    let expected-benefit-L reward-L * probability-opponent-demands-L + reward-L * probability-opponent-demands-M +
            reward-L * probability-opponent-demands-H
    let expected-benefit-M reward-M * probability-opponent-demands-L + reward-M * probability-opponent-demands-M +
            0 * probability-opponent-demands-H
    let  expected-benefit-H reward-H * probability-opponent-demands-L + 0 * probability-opponent-demands-M + 0 *
            probability-opponent-demands-H

    ;; calculate the best option
    let possible-demands (list expected-benefit-L expected-benefit-M expected-benefit-H) ;; the list possible-demands
            contains: [benefit if the agent chooses L , benefit if the agent chooses M , benefit if the agent chooses
            H]
    let best-option max possible-demands ;; the best option is the highest value in the list possible-demands

    ;; PARTICULAR CASE: TWO OR THREE OPTIONS RESULT IN THE SAME EXPECTED BENEFIT
    if length filter [? = best-option] possible-demands > 1   ;; if two or three options produce the same benefit...
      [
        if item 0 possible-demands = item 1 possible-demands and item 1 possible-demands = item 2 possible-demands [report
            one-of options] ;; benefit (low) = benefit (medium) = benefit (high) -> choose low, medium or high at random
        if item 0 possible-demands = item 1 possible-demands [report one-of list low medium]      ;; benefit (low) =
            benefit (medium) -> choose low or medium at random
        if item 0 possible-demands = item 2 possible-demands [report one-of list low high]        ;; benefit (low) =
            benefit (high) -> choose low or high at random
        if item 1 possible-demands = item 2 possible-demands [report one-of list medium high]     ;; benefit (medium)
            = benefit (high) -> choose medium or high at random
```

```
          ]

        ;; GENERAL CASE: ONLY ONE OF THE THREE DEMANDS MAXIMIZE THE EXPECTED BENEFIT
        if item 0 possible-demands = best-option [report low]      ;; if the first element in the list of possible demands
                is the best option, then the agent chooses low
        if item 1 possible-demands = best-option [report medium]   ;; if the second element in the list of possible demands
                is the best option, then the agent chooses medium
        if item 2 possible-demands = best-option [report high]     ;; if the third element in the list of possible demands
                is the best option, then the agent chooses high
      ]

    if decision-rule = "Choose the best reply againts the opponents' most frequent demand"
    [
        let probability-opponent-demands-L (length filter [? = low] agents_memory) / m      ;; count the number of appearances
                of L in the agent's memory
        let probability-opponent-demands-M (length filter [? = medium] agents_memory) / m   ;; count the number of appearances
                of M in the agent's memory
        let probability-opponent-demands-H (length filter [? = high] agents_memory) / m     ;; count the number of appearances
                of H in the agent's memory

        ;; estimate the option that the opponent is likely to take
        let       possible-demands       (list       probability-opponent-demands-L       probability-opponent-demands-M
                probability-opponent-demands-H) ;; the list possible-demands contains: [probability that the opponent chooses
                L , probability that the opponent chooses M , probability that the opponent choosess H]
        let opponent-most-likely-demand max possible-demands ;; the opponent's most likely option is the highest value
                in the list possible-demands

        ;; PARTICULAR CASE: THERE IS A TIE IN THE NUMBER OF APPEARANCES OF low, medium or high
        if length filter [? = opponent-most-likely-demand] possible-demands > 1   ;; there is a tie in the opponents' most
                frequent demands in the previous matches
        [
           if item 0 possible-demands = item 1 possible-demands and item 1 possible-demands = item 2 possible-demands [report
               one-of options] ;; frequency (low) = frequency (medium) = frequency (high) -> choose low, medium or high
               at random
           if item 0 possible-demands = item 1 possible-demands [report one-of list low medium]      ;; frequency (low)
               = frequency (medium) -> choose low or medium at random
           if item 0 possible-demands = item 2 possible-demands [report one-of list low high]        ;; frequency (low)
               = frequency (high) -> choose low or high at random
           if item 1 possible-demands = item 2 possible-demands [report one-of list medium high]     ;; frequency (medium)
               = frequency (high) -> choose medium or high at random
        ]

        ;; GENERAL CASE: ONLY ONE OF THE THREE DEMANDS IS THE MOST FREQUENT
        if item 0 possible-demands = opponent-most-likely-demand [report high]      ;; if the first element in the list
                of possible demands is the best option, then the agent chooses low
        if item 1 possible-demands = opponent-most-likely-demand [report medium]    ;; if the second element in the list
                of possible demands is the best option, then the agent chooses medium
        if item 2 possible-demands = opponent-most-likely-demand [report low]       ;; if the third element in the list of
                possible demands is the best option, then the agent chooses high
     ]
  ]

end

to-report place-agents-xcor [agents_memory]

  let appearances-of-L length filter [? = low] agents_memory      ;; count the number of appearances of L in the agent's
                memory
  let appearances-of-M length filter [? = medium] agents_memory   ;; count the number of appearances of M in the agent's
                memory
  let appearances-of-H length filter [? = high] agents_memory     ;; count the number of appearances of H in the agent's
                memory

  ;; GENERAL CASE (there is one or more appearances of H in the memory vector)
  ifelse appearances-of-H > 0
  [
    let prop_H_L (appearances-of-H / (appearances-of-L + appearances-of-H))  ;; #H / (#L + #H)
    let prop_H_M (appearances-of-H / (appearances-of-M + appearances-of-H))  ;; #H / (#M + #H)
    let x-coordinate (1 / (((1 / prop_H_L) + (1 / prop_H_M))*(1 / sin 60) - (2 / tan 60))) * (((1 / prop_H_L)*(1 / sin 60))-
                (1 / tan 60)) * simplex-side-length
    let shifted-x-coordinate x-coordinate - simplex-side-length / 2         ;; we need to shift this as the triangle is centered
                in (0,0)
    report shifted-x-coordinate
  ]

  ;; PARTICULAR CASE (if there are no H's in the memory vector, a determination appears in the general expression. We deal
```

```
                with this case separately)

  [
    let prop_L_M (appearances-of-L / (appearances-of-L + appearances-of-M))  ;; #L / (#L + #M)
    let x-coordinate prop_L_M * simplex-side-length
    let shifted-x-coordinate x-coordinate - simplex-side-length / 2         ;; we need to shift this as the triangle is centered
                in (0,0)
    report shifted-x-coordinate
  ]
end


to-report place-agents-ycor [agents_memory]

  let appearances-of-L length filter [? = low] agents_memory     ;; count the number of appearances of L in the agent's
                memory
  let appearances-of-M length filter [? = medium] agents_memory   ;; count the number of appearances of M in the agent's
                memory
  let appearances-of-H length filter [? = high] agents_memory     ;; count the number of appearances of H in the agent's
                memory

   ;; GENERAL CASE (there is one or more appearances of H in the memory vector)
  ifelse appearances-of-H > 0
  [
    let prop_H_L (appearances-of-H / (appearances-of-L + appearances-of-H))     ;; #H / (#L + #H)
    let prop_H_M (appearances-of-H / (appearances-of-M + appearances-of-H))     ;; #H / (#M + #H)
    let y-coordinate (1 / (((1 / prop_H_L) + (1 / prop_H_M))*(1 / sin 60) - (2 / tan 60))) * simplex-side-length
    let shifted-y-coordinate y-coordinate - sqrt(3) / 4 * simplex-side-length   ;; we need to shift this as the triangle
                is centered in (0,0)
    report shifted-y-coordinate
  ]

  ;; PARTICULAR CASE (if there are no H's in the memory vector, a determination appears in the general expression. We deal
                with this case separately)
  [
    let y-coordinate 0
    let shifted-y-coordinate y-coordinate - sqrt(3) / 4 * simplex-side-length  ;; we need to shift this as the triangle is
                centered in (0,0)
    report shifted-y-coordinate
  ]

end


to create-simplex
  let M_xcor (- simplex-side-length / 2)                    ;; x-coordinate of the left vertex of the simplex (M).
  let M_ycor (- sqrt(3) * simplex-side-length / 4)            ;; y-coordinate of the left vertex of the simplex (M).
  let L_xcor (simplex-side-length / 2)                      ;; x-coordinate of the right vertex of the simplex (L).
  let L_ycor (- sqrt(3) * simplex-side-length / 4)            ;; y-coordinate of the right vertex of the simplex (L).
  let H_xcor 0                                          ;; x-coordinate of the top vertex of the simplex (H).
  let H_ycor (sqrt(3) / 4 * simplex-side-length)              ;; y-coordinate of the top vertex of the simplex (H).

 let background-color [130 188 183]                       ;; define the background color
 ask patches [set pcolor background-color]                  ;; set the background color

 let surrounding-simplex-side-length simplex-side-length + 6    ;; set the size of the surrounding simplex.

 if decision-rule = "Demand the option that maximizes the expected benefit"
    [
      ;; LEFT SIDE OF THE TRIANGLE: calculate the proportions of M and H in the memory vector so that the expected benefits
                of choosing M and choosing H match when there are no appearances of L in the memory vector (le = left equilibrium)
      let Ale (medium / high)
      let Ble (1 -( medium / high))

      ;; RIGHT SIDE OF THE TRIANGLE: calculate the proportions of L and H in the memory vector so that the expected benefits
                of choosing L and choosing H match when there are no appearances of M in the memory vector (re = right equilibrium)
      let Cre (1 -( low / high))
      let Apede (low / high)

      ;; LOWER SIDE OF THE TRIANGLE: calculate the proportions of L and M in the memory vector so that the expected benefits
                of choosing L and choosing M match when there are no appearances of H in the memory vector (lwe = lower
                equilibrium)
      let Clwe (1 -( low / medium))
      let Blwe (low / medium)

      ;; calculate the proportions of L, M and H in the memory vector so that the expected benefit of choosing L, choosing
```

```
            M and choosing H match
    let Aep low / high                  ;; value of A in the equilibrium point (ep)
    let Bep low / medium - low / high     ;; value of B in the equilibrium point (ep)
    let Cep 1 - low / medium             ;; value of C in the equilibrium point (ep)

    let ep-x-coordinate-numerator 1 / ((Cep / (Aep + Cep)) * sin 60) - 1 / tan 60
    let ep-x-coordinate-denominator ((1 / (Cep / (Aep + Cep)) + 1 / (Cep / (Bep + Cep))) * 1 / sin 60 ) - 2 / tan 60
    let ep-x-coordinate ep-x-coordinate-numerator / ep-x-coordinate-denominator * simplex-side-length
    let shifted-ep-x-coordinate ep-x-coordinate - simplex-side-length / 2

    let ep-y-coordinate (1 / (((1 / (Cep / (Aep + Cep))) + (1 / (Cep / (Bep + Cep)))) * (1 / sin 60) - 2 / tan 60 )) *
            simplex-side-length
    let shifted-ep-y-coordinate ep-y-coordinate - sqrt(3) / 4 * simplex-side-length

    ;; DRAW THE DECISION BORDERS
    create-turtles 1 [
                    ;;    We    only    draw    the    decision    borders    if    the    memory    type    is    not
                          endorsed.                                                                                ;\\
                    ifelse memory-type = "Endorsed"
                    [
                       ;; OUTER TRIANGLE
                       set size 0
                       setxy (surrounding-simplex-side-length / 2)(- sqrt(3) / 4 * (surrounding-simplex-side-length
                             - (surrounding-simplex-side-length - simplex-side-length) / 4))
                       set color black set pen-size 5 pd
                       set  heading  270  fd  surrounding-simplex-side-length      ;;  move  towards  the  left  vertex
                             (M)                                                          ;\\
                       set  heading  30  fd  surrounding-simplex-side-length        ;;  move  towards  the  top  vertex
                             (H)                                                          ;\\
                       set  heading  150  fd  surrounding-simplex-side-length      ;;  move  towards  the  right  vertex
                             (L)                                                          ;\\
                    ]
                    [
                       ;; OUTER TRIANGLE
                       set size 0
                       setxy (surrounding-simplex-side-length / 2)(- sqrt(3) / 4 * (surrounding-simplex-side-length
                             - (surrounding-simplex-side-length - simplex-side-length) / 4))
                       set color green set pen-size 5 pd
                       set  heading  270  fd  (surrounding-simplex-side-length  *  Clwe)  set  color  yellow  fd
                             (surrounding-simplex-side-length  *  Blwe)        ;;  move  towards  the  left  vertex
                             (M)                                                          ;\\
                       set  heading  30  fd  (surrounding-simplex-side-length  *  Ble)  set  color  red  fd
                             (surrounding-simplex-side-length * Ale)             ;;  move  towards  the  top  vertex
                             (H)                                                          ;\\
                       set  heading  150  fd  (surrounding-simplex-side-length  *  Apede)  set  color  green  fd
                             (surrounding-simplex-side-length  *  Cre)  pu     ;;  move  towards  the  right  vertex
                             (L)                                                          ;\\


                       ;; DECISION BORDERS
                       ;; right border
                       setxy shifted-ep-x-coordinate shifted-ep-y-coordinate
                       set color black set pen-size 1 pd move-to patch (simplex-side-length / 2 - Cre * simplex-side-length
                             * cos 60) (Cre * simplex-side-length * sin 60 - sqrt(3) / 4 * simplex-side-length) pu
                       ;; left border
                       setxy shifted-ep-x-coordinate shifted-ep-y-coordinate
                       set  color  black  set  pen-size  1  pd  move-to  patch  (Ble  *  simplex-side-length  *  cos  60  -
                             simplex-side-length  /  2)  (Ble  *  simplex-side-length  *  sin  60  -  sqrt(3)  /  4  *
                             simplex-side-length) pu
                       ;; lower border
                       setxy shifted-ep-x-coordinate shifted-ep-y-coordinate
                       set color black set pen-size 1 pd move-to patch ( simplex-side-length / 2 - Clwe * simplex-side-length)
                             (- sqrt(3) / 4 * simplex-side-length) pu
                    ]
                    die
                ]

    ;; ADD A LABEL TO THE SIMPLEX VERTICES
      ask patch (M_xcor - 5) (M_ycor - 2) [set plabel-color yellow set plabel "MEDIUM 50"]
      ask patch (L_xcor + 18)(L_ycor - 2) [set Plabel-color 54 set plabel WORD "LOW " low]
      ask patch (H_xcor + 4) (H_ycor + 6) [set Plabel-color RED set plabel WORD "HIGH " high]
]

if decision-rule = "Choose the best reply againts the opponents' most frequent demand"
[
     let ep-x-coordinate simplex-side-length / 2
```

```
      let shifted-ep-x-coordinate ep-x-coordinate - simplex-side-length / 2
      let ep-y-coordinate (sqrt (3) / 6) * simplex-side-length
      let shifted-ep-y-coordinate ep-y-coordinate - (sqrt (3) / 4) * simplex-side-length
      let coordenada_equilibrio_y (sqrt (3) / 6 - sqrt (3) / 4) * simplex-side-length
      ;; DRAW THE DECISION BORDERS
      create-turtles 1 [
                                                                                                  ;\\

                  ;; OUTER TRIANGLE
                  set size 0
                  setxy (surrounding-simplex-side-length / 2)(- sqrt(3) / 4 * (surrounding-simplex-side-length -
                              (surrounding-simplex-side-length - simplex-side-length) / 4))
                  set color green set pen-size 5 pd
                  set heading 270 fd (surrounding-simplex-side-length * 1 / 2) set color YELLOW fd
                              (surrounding-simplex-side-length * 1 / 2)
                  set heading 30 fd (surrounding-simplex-side-length * 1 / 2) set color RED fd
                              (surrounding-simplex-side-length * 1 / 2)
                  set heading 150 fd (surrounding-simplex-side-length * 1 / 2) set color 54 fd
                              (surrounding-simplex-side-length * 1 / 2) pu
                  ;; DECISION BORDERS
                  set color black
                  set pen-size 1
                  ;; right border
                  setxy shifted-ep-x-coordinate shifted-ep-y-coordinate
                  pd move-to patch (simplex-side-length / 4) 0 pu
                  ;; left border
                  setxy shifted-ep-x-coordinate shifted-ep-y-coordinate
                  pd move-to patch (- simplex-side-length / 4) 0 pu
                  ;; lower border
                  setxy shifted-ep-x-coordinate shifted-ep-y-coordinate
                  pd move-to patch 0 (- sqrt (3) / 4 * simplex-side-length)
                  die
                ]
      ;; ADD A LABEL TO THE SIMPLEX VERTICES
      ask patch (M_xcor - 5) (M_ycor - 2) [set plabel-color yellow set plabel "MEDIUM 50"]
      ask patch (L_xcor + 18)(L_ycor - 2) [set Plabel-color 54 set plabel WORD "LOW " low]
      ask patch (H_xcor + 4) (H_ycor + 6) [set Plabel-color RED set plabel WORD "HIGH " high]
  ]

end

to check-stop-conditions

  if memory-type != "Endorsed"
     [
       set equitable-equilibrium-stop-condition all? agents [length filter [? = medium] memory >= (1 - epsilon) * m]
       set fractious-state-stop-condition all? agents [length filter [? = medium] memory <= epsilon * m]
     ]

  if memory-type = "Endorsed"
     [
       let S (2 + (m - 1) * d) / 2 * m
       set equitable-equilibrium-stop-condition all? agents [check-endorsed-memory (memory) >= (1 - epsilon) * S ]
       set fractious-state-stop-condition all? agents [check-endorsed-memory (memory) <= epsilon * S ]
     ]

  ;; IF THIS IS THE FIRST TIME THE SYSTEM HAS REACHED AN EQUITABLE EQUILIBRIUM
  if equitable-equilibrium-stop-condition = true and equitable-equilibrium? = false
    [
     print (word "* * * The system has reached an equitable equilibrium in the iteration number " elapsed-iterations " *
                * *")
     set system-status "EQUITABLE EQUILIBRIUM"       ;; display "EQUITABLE EQUILIBRIUM" in the system-status monitor
     set equitable-equilibrium? true               ;; now, the system is in equitable equilibrium
     set equitable-equilibrium-at-least-once? true    ;; the system has reached an equitable equilibrium at least once
    ]

  ;; IF THIS IS THE FIRST TIME THE SYSTEM HAS REACHED A FRACTIOUS STATE
  if fractious-state-stop-condition and fractious-state? = false
    [
     print (word "* * * The system has reached a fractious state in the iteration number " elapsed-iterations " * * *")
     set system-status "FRACTIOUS STATE"            ;; display "FRACTIOUS STATE" in the system-status monitor
     set fractious-state? true                   ;; now, the system is in fractious state
     set fractious-state-at-least-once? true         ;; the system has reached a fractious state at least once
    ]

  ;; IF THE SYSTEM LEAVES THE EQUITABLE EQUILIBRIUM...
```

```
  if equitable-equilibrium? = true and not equitable-equilibrium-stop-condition
    [
     print (word "* * * The system has left the equitable equilibrium in the iteration number " elapsed-iterations " * * *")
     set equitable-equilibrium? false

    ]

  ;; IF THE SYSTEM LEAVES THE FRACTIOUS STATE...
  if fractious-state? = true and not fractious-state-stop-condition
    [
     print (word "* * * The system has left the fractious state in the iteration number " elapsed-iterations " * * *")
     set fractious-state? false
    ]
end

to-report check-endorsed-memory  [agents-memory]
  let q 0
  (foreach agents-memory endorsed-memory-weights [if ?1 = medium [set q q + ?2]])
  report q    ;; returns the number of elements equal to medium in the agent's memory (each appearance of medium in the memory set is multiplied by the weight of the position in which it appears)
end
```

2-AGENT TYPES BARGAINING MODEL (TAG MODEL)

```
breed [agents agent]
breed [images image]

agents-own [id tag intertype-memory intratype-memory]
images-own [id tag intertype-memory intratype-memory]

globals
[
   options                                ;; list that contains the following values - according to the chosen payoff matrix: [low medium high]
   medium                                 ;; numeric value assigned to the 'medium' demand
   high                                   ;; numeric value assigned to the 'high' demand
   simplex-side-length                    ;; length of the simplex side
   agents-list                            ;; list that contains all the agents in the population
   elapsed-iterations                     ;; number of elapsed iterations since the simulation was started

   intratype-equitable-equilibrium-stop-condition   ;; conditions for equitable equilibrium in the case of intratype matches
   intratype-fractious-state-stop-condition         ;; conditions for fractious state in the case of intratype matches
   intratype-segregation-stop-condition             ;; conditions for equitable equilibrium for the agents with one colour and fractious state for the agents with another colour

   intertype-equitable-equilibrium-stop-condition   ;; conditions for equitable equilibrium in the case of intertype matches
   intertype-segregation-stop-condition             ;; the agents with one colour choose H and the agents with the other colour choose L

   intratype-simulation-finished?           ;; boolean variable that will be true if the simulation is finished in the intratype simplex
   intertype-simulation-finished?           ;; boolean variable that will be true if the simulation is finished in the intertype simplex

   intratype-status                         ;; this variable contains the message shown in the intratype-status monitor, in the screen interface.
   intertype-status                         ;; this variable contains the message shown in the intertype-status monitor, in the screen interface.

   endorsed-memory-weights                  ;; this list contains the weights assigned to each term in the endorsed memory set
   shift                                    ;; distance from each symplex to the origin
]

to setup

   print ""                                 ;; leave a black line in the command center
   clear-all                                ;; reset variables

   ;; DEFINE THE REWARDS ACCORDING TO THE SELECTED PAYOFF MATRIX (the value for the lowest reward -low- is chosen in the interface screen)
```

```
set high 100 - low                    ;; set the value assigned to high
set medium 50                         ;; set the value assigned to medium
set options (list low medium high)       ;; create a list which contains the values assigned to low, medium and high
                                             according to the selected payoff matrix

;; PRINT THE PAYOFF MATRIX ACCORDING TO THE SELECTED VALUE FOR LOW
output-print "      L      M      H    "
output-type " L |(" output-type low output-type "," output-type low output-type ")|(" output-type low output-type ","
            output-type medium output-type ")|(" output-type low output-type "," output-type high output-print ")|"
output-type " M |(" output-type medium output-type "," output-type low output-type ")|(" output-type medium output-type
            "," output-type medium output-print ")| (0,0) |"
output-type " H |(" output-type high output-type "," output-type low output-print ")| (0,0) | (0,0) |"

;; CREATE THE SIMPLEX
set simplex-side-length 100             ;; set the simplex size
set shift 75                          ;; distance from each symplex to the origin
create-simplex                        ;; call the procedure 'create-simplex'

;; CREATE A POPULATION OF n AGENTS
create-agents n
[
  set id who                                                  ;; attach an id to each agent to monitor its position
            in both simplexes
  set shape "dot"                                              ;; set the agent's shape
  set size 4                                                  ;; set the agent's size
  set color black                                              ;; set the agent's colour
  if label-on-agents? [set label who]                             ;; create a label with the id of each agent
  if memory-type = "Standard" [set intratype-memory n-values m [one-of options] set intertype-memory intratype-memory]    ;;
            create an m-size memory with random values for each agent
  if memory-type = "Endorsed" [set intratype-memory n-values m [one-of options] set intertype-memory intratype-memory]    ;;
            create an m-size memory with random values for each agent
  if memory-type = "Progressive" [set intratype-memory n-values 1 [one-of options] set intertype-memory intratype-memory] ;;
            initialize the memory with just one value choosen at random
  set xcor place-agents-xcor (intratype-memory) - shift                    ;; place the agents in the simplex according
            to their memory status (x-coordinate)
  set ycor place-agents-ycor (intratype-memory)                            ;; place the agents in the simplex according
            to their memory status (y-coordinate)
]

;; (n/2) AGENTS TURN ORANGE
ask n-of (n / 2) agents [set color orange]

;; ATTACH A LABEL (COLOUR) TO EACH PLAYER
ask agents [set tag [color] of self]

;; CREATE n 'images' and place them on the right simplex (intertype)
ask agents [hatch-images 1 [set shape "dot" set xcor place-agents-xcor (intertype-memory) + shift set ycor place-agents-ycor
            (intertype-memory)]]


;; SET THE INITIAL VALUE OF SOME OF THE GLOBALS VARIABLES
set intertype-status "----------"                    ;; this message is shown in the intertype-status monitor until
            an equitable equilibrium or a fractious state is reached
set intratype-status "----------"                    ;; this message is shown in the intratype-status monitor until
            an equitable equilibrium or a fractious state is reached

set intratype-simulation-finished? false               ;; one of the conditions to stop the simulation
set intertype-simulation-finished? false               ;; one of the conditions to stop the simulation

;; CREATE A LIST WITH ALL THE AGENTS IN THE POPULATION
set agents-list sort agents                          ;; turns the agentset into a list

;; CREATE A LIST OF WEIGHTS FOR THE ENDORSED MEMORY
if memory-type = "Endorsed"
[
   let weights []        ;; list of weights
   let current-weight 1   ;; lowest weight
   let i 0             ;; counter (loop)
   while [i < m]
   [
      set weights fput current-weight weights  ;; add the current weight to the list of weights
      set current-weight current-weight + d    ;; increase the weight for the next element
      set i i + 1                         ;; increase the loop counter
   ]
   set endorsed-memory-weights weights        ;; update the global variable
]
```

```
   if memory-type != "Endorsed" [set d 0]          ;; if the memory is not 'endorsed', 'd' (difference common in the arithmetic
               progression which is followed by the weights of each memory position makes no sense here
   if decision-rule = "Choose the best reply againts the opponents' most frequent demand" and memory-type = "Endorsed"
               [beep user-message "This decision rule is not compatible with an endorsed memory" ]  ;; displays a warning
               in a pop-up window
end


to go
   set elapsed-iterations 0                                ;; initialize the elapsed-iterations counter
   while [elapsed-iterations < number_of_iterations]           ;; we run as many iterations as the value of the variable
               number_of_iterations
   [
      let players agents-list                              ;; retrieve the original list of agents
      repeat (length agents-list) / 2                      ;; the number of matches in each iteration is half the number
               of agents
      [
        ;; 1. TAKE TWO PLAYERS AT RANDOM FROM THE LIST OF AGENTS
        let player1 one-of players                          ;; choose one agent from the list of agents at random
        set players remove player1 players                  ;; delete this agent so that it can't be chosen again during
               this iteration
        let player2 one-of players                          ;; choose one agent from the list of agents at random
        set players remove player2 players                  ;; delete this agent so that it can't be chosen again during
               this iteration

       ;; 2. EACH AGENT TAKES A DECISION TAKING INTO ACCOUNT HIS MEMORIES ABOUT PREVIOUS MATCHES
           let player1-decision 0  ;; define variable
           let player2-decision 0  ;; define variable

           ifelse [tag] of player1 = [tag] of player2
           [
             ;; 2.1 INTRATYPE MATCHES (agent 1's tag = agent 2's tag)
                 set player1-decision take-decision ([intratype-memory] of player1) ;; player 1 takes his decision
                 set player2-decision take-decision ([intratype-memory] of player2) ;; player 2 takes his decision

           ]
             ;; 2.2 INTERTYPE MATCHES (agent 1's tag != agent 2's tag)
           [
                 set player1-decision take-decision ([intertype-memory] of player1) ;; player 1 takes his decision
                 set player2-decision take-decision ([intertype-memory] of player2) ;; player 2 takes his decision
           ]

       ;; 3. EACH AGENT STORES THE OPPONENT'S DECISION IN HIS MEMORY
           ifelse [tag] of player1 = [tag] of player2
           [
             ;; 3.1 INTRATYPE MATCHES (agent 1's tag = agent 2's tag)
                ask player1 [set intratype-memory fput player2-decision intratype-memory]     ;; player 1 stores player
                            2's decision in his intratype-memory
                ask player2 [set intratype-memory fput player1-decision intratype-memory]     ;; player 2 stores player
                            1's decision in his intratype-memory

                if memory-type = "Standard" or memory-type = "Endorsed" [ask player1 [set intratype-memory but-last
                            intratype-memory]]            ;; Standard and endorsed memory: player 1 deletes his oldest
                            value in his memory.
                if memory-type = "Progressive" and length [intratype-memory] of player1 > m [ask player1 [set intratype-memory
                            but-last intratype-memory]]    ;; Progressive memory: player 1 deletes his oldest value in
                            his memory when the m-size memory is reached

                if memory-type = "Standard" or memory-type = "Endorsed" [ask player2 [set intratype-memory but-last
                            intratype-memory]]            ;; Standard and endorsed memory: player 2 deletes his oldest
                            value in his memory.
                if memory-type = "Progressive" and length [intratype-memory] of player2 > m [ask player2 [set intratype-memory
                            but-last intratype-memory]]    ;; Progressive memory: player 2 deletes his oldest value in
                            his memory when the m-size memory is reached
           ]
           [
             ;; 3.2 INTERTYPE MATCHES (agent 1's tag != agent 2's tag)
                ask player1 [set intertype-memory fput player2-decision intertype-memory]     ;; player 1 stores player
                     2's decision in his intertype-memory
                ask player2 [set intertype-memory fput player1-decision intertype-memory]     ;; player 2 stores player
                     1's decision in his intertype-memory

                if memory-type = "Standard" or memory-type = "Endorsed" [ask player1 [set intertype-memory but-last
                     intertype-memory]]           ;; Standard and endorsed memory: player 1 deletes his oldest value
                     in his memory.
```

```
                    if memory-type = "Progressive" and length [intertype-memory] of player1 > m [ask player1 [set intertype-memory
                          but-last intertype-memory]]    ;; Progressive memory: player 1 deletes his oldest value in his memory
                          when the m-size memory is reached

                    if memory-type = "Standard" or memory-type = "Endorsed" [ask player2 [set intertype-memory but-last
                          intertype-memory]]           ;; Standard and endorsed memory: player 2 deletes his oldest value
                          in his memory.
                    if memory-type = "Progressive" and length [intertype-memory] of player2 > m [ask player2 [set intertype-memory
                          but-last intertype-memory]]    ;; Progressive memory: player 2 deletes his oldest value in his memory
                          when the m-size memory is reached
               ]

            ;; 4. UPDATE THE PLAYERS' IMAGES (RIGHT SIMPLEX)
               ask images with [id = [id] of player1] [set intertype-memory [intertype-memory] of player1 set intratype-memory
                    [intratype-memory] of player1]
               ask images with [id = [id] of player2] [set intertype-memory [intertype-memory] of player2 set intratype-memory
                    [intratype-memory] of player2]

            ;; 5. UPDATE THE POSITION OF THE TWO AGENTS IN THE SIMPLEX ACCORDING TO THE NEW MEMORY STATUS
              ifelse [tag] of player1 = [tag] of player2
              [
                ;; 3.1 INTRATYPE MATCHES (agent 1's tag = agent 2's tag): The left simplex (intratype) changes; the right simplex
                    (intertype) doesn't.
                ask player1 [setxy place-agents-xcor (intratype-memory) - shift place-agents-ycor (intratype-memory)]   ;; place
                    the agents in the simplex according to their memory status (x-coordinate)
                ask player2 [setxy place-agents-xcor (intratype-memory) - shift place-agents-ycor (intratype-memory)]   ;; place
                    the agents in the simplex according to their memory status (y-coordinate)
              ]
              [
                ;; 3.2 INTERTYPE MATCHES (agent 1's tag != agent 2's tag): The right simplex (intertype) changes; the left simplex
                    (intratype) doesn't.
                ask images with [id = [id] of player1] [setxy place-agents-xcor (intertype-memory) + shift place-agents-ycor
                    (intertype-memory)]   ;; place the agents in the simplex according to their memory status (x-coordinate)
                ask images with [id = [id] of player2] [setxy place-agents-xcor (intertype-memory) + shift place-agents-ycor
                    (intertype-memory)]   ;; place the agents in the simplex according to their memory status (y-coordinate)
              ]
          ]


        set elapsed-iterations elapsed-iterations + 1          ;; the current iteration has finished
        tick                                                   ;; increase the tick counter (number of elapsed iterations)

        check-stop-conditions                                  ;; call the procedure 'check-stop-conditions'

        if intratype-simulation-finished? and intertype-simulation-finished? [print "* * * SIMULATION FINISHED * * *" stop] ;;
                    if the system has reached one of the five scenarios in this iteration, the simulation stops now.

        if elapsed-iterations = number_of_iterations           ;; when the maximum number of iterations is reached, a message
                    is shown and the simulation finishes.
        [
          print "* * * SIMULATION FINISHED: The system has reached the maximum number of iterations. * * *"
          ;;if equitable-equilibrium-at-least-once? = false and fractious-state-at-least-once? = false [set system-status
                    "SIMULATION FINISHED" print "* * * No equilibrium was reached in the system. * * *"]
          stop
        ]

      ]

end

to-report take-decision [agents_memory]

  let random-number random-float 1  ;; create a random number
  ifelse random-number < epsilon

  ;; if the random number is lower than epsilon, the decision is taken randomly
  [
    report one-of options
  ]

  ;; if the random number is lower than epsilon, the decision rule is 'rational'
  [
     if decision-rule = "Demand the option that maximizes the expected benefit"
     [
        let probability-opponent-demands-L (length filter [? = low] agents_memory) / m      ;; count the number of appearances
                    of L in the agent's memory
```

```
        let probability-opponent-demands-M (length filter [? = medium] agents_memory) / m   ;; count the number of appearances
                of M in the agent's memory
        let probability-opponent-demands-H (length filter [? = high] agents_memory) / m     ;; count the number of appearances
                of H in the agent's memory

        let reward-L low     ;; set the reward assigned to low
        let reward-M medium   ;; set the reward assigned to medium
        let reward-H high    ;; set the reward assigned to high

        ;; the agent calculates the expected benefit if he chooses L, M or H
        let expected-benefit-L reward-L * probability-opponent-demands-L + reward-L * probability-opponent-demands-M +
                reward-L * probability-opponent-demands-H
        let expected-benefit-M reward-M * probability-opponent-demands-L + reward-M * probability-opponent-demands-M +
                0 * probability-opponent-demands-H
        let  expected-benefit-H reward-H * probability-opponent-demands-L + 0 * probability-opponent-demands-M + 0 *
                probability-opponent-demands-H

        ;; calculate the best option
        let possible-demands (list expected-benefit-L expected-benefit-M expected-benefit-H) ;; the list possible-demands
                contains: [benefit if the agent chooses L , benefit if the agent chooses M , benefit if the agent chooses
                H]
        let best-option max possible-demands ;; the best option is the highest value in the list possible-demands

        ;; PARTICULAR CASE: TWO OR THREE OPTIONS RESULT IN THE SAME EXPECTED BENEFIT
        if length filter [? = best-option] possible-demands > 1   ;; if two or three options produce the same benefit...
        [
          if item 0 possible-demands = item 1 possible-demands and item 1 possible-demands = item 2 possible-demands [report
                one-of options] ;; benefit (low) = benefit (medium) = benefit (high) -> choose low, medium or high at random
          if item 0 possible-demands = item 1 possible-demands [report one-of list low medium]      ;; benefit (low) = benefit
                (medium) -> choose low or medium at random
          if item 0 possible-demands = item 2 possible-demands [report one-of list low high]        ;; benefit (low) = benefit
                (high) -> choose low or high at random
          if item 1 possible-demands = item 2 possible-demands [report one-of list medium high]     ;; benefit (medium) =
                benefit (high) -> choose medium or high at random
        ]

        ;; GENERAL CASE: ONLY ONE OF THE THREE DEMANDS MAXIMIZE THE EXPECTED BENEFIT
        if item 0 possible-demands = best-option [report low]      ;; if the first element in the list of possible demands
                is the best option, then the agent chooses low
        if item 1 possible-demands = best-option [report medium]   ;; if the second element in the list of possible demands
                is the best option, then the agent chooses medium
        if item 2 possible-demands = best-option [report high]     ;; if the third element in the list of possible demands
                is the best option, then the agent chooses high
    ]

    if decision-rule = "Demand the option that maximizes the expected benefit" and memory-type = "Endorsed"
    [
        let i 0                                    ;; set a counter for the while loop: 0<=i<memory-size
        let L 0                                    ;; L is the number of appearances of low endorsed by the position that
                they appear in the memory
        let Me 0                                   ;; Me is the number of appearances of medium endorsed by the position
                that they appear in the memory
        let H 0                                    ;; H is the number of appearances of high endorsed by the position that
                they appear in the memory
        let highest-weight 1 + (m - 1) * d             ;; the highest weight is the last term in an arithmetic progression
                with common difference 'd'
        let current-weight highest-weight              ;; the value of current-weight decreases as we move towards the oldest
                values stored in the memory
        while [i < length agents_memory]
        [
           if item i agents_memory = low [set L L + current-weight]      ;; the value of L is increased if the current
                memory position is low; current-weight decreases as we move towards the oldest values stored in the memory
           if item i agents_memory = medium [set Me Me + current-weight]  ;; the value of Me is increased if the current
                memory position is low or medium; current-weight decreases as we move towards the oldest values stored in
                the memory
           if item i agents_memory = high [set H H + current-weight]     ;; the value of L is increased if the current
                memory position is low; current-weight decreases as we move towards the oldest values stored in the memory
           set i i + 1                                    ;; increase the array position
           set current-weight current-weight - d              ;; substract the common difference from the current-weight
                to obtain a new weight for the next iteration
        ]
        let S (2 + (m - 1) * d) / 2 * m

        let probability-opponent-demands-L L / S     ;; count the number of appearances of L in the agent's memory
        let probability-opponent-demands-M Me / S    ;; count the number of appearances of Me in the agent's memory
        let probability-opponent-demands-H H / S     ;; count the number of appearances of H in the agent's memory
```

```
    let reward-L low     ;; set the reward assigned to low
    let reward-M medium   ;; set the reward assigned to medium
    let reward-H high    ;; set the reward assigned to high

    ;; the agent calculates the expected benefit if he chooses L, M or H
    let expected-benefit-L reward-L * probability-opponent-demands-L + reward-L * probability-opponent-demands-M +
           reward-L * probability-opponent-demands-H
    let expected-benefit-M reward-M * probability-opponent-demands-L + reward-M * probability-opponent-demands-M +
           0 * probability-opponent-demands-H
    let  expected-benefit-H reward-H * probability-opponent-demands-L + 0 * probability-opponent-demands-M + 0 *
           probability-opponent-demands-H

    ;; calculate the best option
    let possible-demands (list expected-benefit-L expected-benefit-M expected-benefit-H) ;; the list possible-demands
           contains: [benefit if the agent chooses L , benefit if the agent chooses M , benefit if the agent chooses
           H]
    let best-option max possible-demands ;; the best option is the highest value in the list possible-demands

    ;; PARTICULAR CASE: TWO OR THREE OPTIONS RESULT IN THE SAME EXPECTED BENEFIT
    if length filter [? = best-option] possible-demands > 1   ;; if two or three options produce the same benefit...
      [
        if item 0 possible-demands = item 1 possible-demands and item 1 possible-demands = item 2 possible-demands [report
            one-of options] ;; benefit (low) = benefit (medium) = benefit (high) -> choose low, medium or high at random
        if item 0 possible-demands = item 1 possible-demands [report one-of list low medium]      ;; benefit (low) =
            benefit (medium) -> choose low or medium at random
        if item 0 possible-demands = item 2 possible-demands [report one-of list low high]       ;; benefit (low) =
            benefit (high) -> choose low or high at random
        if item 1 possible-demands = item 2 possible-demands [report one-of list medium high]    ;; benefit (medium)
            = benefit (high) -> choose medium or high at random
      ]

    ;; GENERAL CASE: ONLY ONE OF THE THREE DEMANDS MAXIMIZE THE EXPECTED BENEFIT
    if item 0 possible-demands = best-option [report low]      ;; if the first element in the list of possible demands
           is the best option, then the agent chooses low
    if item 1 possible-demands = best-option [report medium]   ;; if the second element in the list of possible demands
           is the best option, then the agent chooses medium
    if item 2 possible-demands = best-option [report high]     ;; if the third element in the list of possible demands
           is the best option, then the agent chooses high
  ]

if decision-rule = "Choose the best reply againts the opponents' most frequent demand"
[
    let probability-opponent-demands-L (length filter [? = low] agents_memory) / m      ;; count the number of appearances
           of L in the agent's memory
    let probability-opponent-demands-M (length filter [? = medium] agents_memory) / m   ;; count the number of appearances
           of M in the agent's memory
    let probability-opponent-demands-H (length filter [? = high] agents_memory) / m     ;; count the number of appearances
           of H in the agent's memory

    ;; estimate the option that the opponent is likely to take
    let       possible-demands       (list       probability-opponent-demands-L       probability-opponent-demands-M
           probability-opponent-demands-H) ;; the list possible-demands contains: [probability that the opponent chooses
           L , probability that the opponent chooses M , probability that the opponent choosess H]
    let opponent-most-likely-demand max possible-demands ;; the opponent's most likely option is the highest value
           in the list possible-demands

    ;; PARTICULAR CASE: THERE IS A TIE IN THE NUMBER OF APPEARANCES OF low, medium or high
    if length filter [? = opponent-most-likely-demand] possible-demands > 1   ;; there is a tie in the opponents' most
           frequent demands in the previous matches
    [
       if item 0 possible-demands = item 1 possible-demands and item 1 possible-demands = item 2 possible-demands [report
           one-of options] ;; frequency (low) = frequency (medium) = frequency (high) -> choose low, medium or high
           at random
       if item 0 possible-demands = item 1 possible-demands [report one-of list low medium]      ;; frequency (low)
           = frequency (medium) -> choose low or medium at random
       if item 0 possible-demands = item 2 possible-demands [report one-of list low high]       ;; frequency (low)
           = frequency (high) -> choose low or high at random
       if item 1 possible-demands = item 2 possible-demands [report one-of list medium high]    ;; frequency (medium)
           = frequency (high) -> choose medium or high at random
    ]

    ;; GENERAL CASE: ONLY ONE OF THE THREE DEMANDS IS THE MOST FREQUENT
    if item 0 possible-demands = opponent-most-likely-demand [report high]     ;; if the first element in the list
           of possible demands is the best option, then the agent chooses low
    if item 1 possible-demands = opponent-most-likely-demand [report medium]   ;; if the second element in the list
```

```
                of possible demands is the best option, then the agent chooses medium
          if item 2 possible-demands = opponent-most-likely-demand [report low]      ;; if the third element in the list of
                possible demands is the best option, then the agent chooses high
        ]
  ]

end

to-report place-agents-xcor [agents_memory]

  let appearances-of-L length filter [? = low] agents_memory      ;; count the number of appearances of L in the agent's
                                                                        memory
  let appearances-of-M length filter [? = medium] agents_memory   ;; count the number of appearances of M in the agent's
                                                                        memory
  let appearances-of-H length filter [? = high] agents_memory     ;; count the number of appearances of H in the agent's
                                                                        memory

  ;; GENERAL CASE (there is one or more appearances of H in the memory vector)
  ifelse appearances-of-H > 0
  [
    let prop_H_L (appearances-of-H / (appearances-of-L + appearances-of-H))  ;; #H / (#L + #H)
    let prop_H_M (appearances-of-H / (appearances-of-M + appearances-of-H))  ;; #H / (#M + #H)
    let x-coordinate (1 / (((1 / prop_H_L) + (1 / prop_H_M))*(1 / sin 60) - (2 / tan 60))) * (((1 / prop_H_L)*(1 / sin 60))-
                (1 / tan 60)) * simplex-side-length
    let shifted-x-coordinate x-coordinate - simplex-side-length / 2         ;; we need to shift this as the triangle is centered
                in (0,0)
    report shifted-x-coordinate
  ]

  ;; PARTICULAR CASE (if there are no H's in the memory vector, a determination appears in the general expression. We deal
                with this case separately)

  [
    let prop_L_M (appearances-of-L / (appearances-of-L + appearances-of-M))  ;; #L / (#L + #M)
    let x-coordinate prop_L_M * simplex-side-length
    let shifted-x-coordinate x-coordinate - simplex-side-length / 2         ;; we need to shift this as the triangle is centered
                in (0,0)
    report shifted-x-coordinate
  ]
end


to-report place-agents-ycor [agents_memory]

  let appearances-of-L length filter [? = low] agents_memory     ;; count the number of appearances of L in the agent's
                                                                        memory
  let appearances-of-M length filter [? = medium] agents_memory  ;; count the number of appearances of M in the agent's
                                                                        memory
  let appearances-of-H length filter [? = high] agents_memory    ;; count the number of appearances of H in the agent's
                                                                        memory

   ;; GENERAL CASE (there is one or more appearances of H in the memory vector)
  ifelse appearances-of-H > 0
  [
    let prop_H_L (appearances-of-H / (appearances-of-L + appearances-of-H))    ;; #H / (#L + #H)
    let prop_H_M (appearances-of-H / (appearances-of-M + appearances-of-H))    ;; #H / (#M + #H)
    let y-coordinate (1 / (((1 / prop_H_L) + (1 / prop_H_M))*(1 / sin 60) - (2 / tan 60))) * simplex-side-length
    let shifted-y-coordinate y-coordinate - sqrt(3) / 4 * simplex-side-length  ;; we need to shift this as the triangle
                is centered in (0,0)
    report shifted-y-coordinate
  ]

  ;; PARTICULAR CASE (if there are no H's in the memory vector, a determination appears in the general expression. We deal
                with this case separately)
  [
    let y-coordinate 0
    let shifted-y-coordinate y-coordinate - sqrt(3) / 4 * simplex-side-length  ;; we need to shift this as the triangle is
                centered in (0,0)
    report shifted-y-coordinate
  ]

end


to create-simplex
  ;; INTRATYPE SIMPLEX (LEFT TRIANGLE)
```

```
let M1_xcor (- simplex-side-length / 2 - shift)              ;; x-coordinate of the left vertex of the left simplex (M).
let M1_ycor (- sqrt(3) * simplex-side-length / 4)            ;; y-coordinate of the left vertex of the left simplex (M).
let L1_xcor (simplex-side-length / 2 - shift)               ;; x-coordinate of the right vertex of the left simplex (L).
let L1_ycor (- sqrt(3) * simplex-side-length / 4)            ;; y-coordinate of the right vertex of the left simplex (L).
let H1_xcor 0 - shift                                        ;; x-coordinate of the top vertex of the left simplex (H).
let H1_ycor (sqrt(3) / 4 * simplex-side-length)              ;; y-coordinate of the top vertex of the left simplex (H).

;; INTERTYPE SIMPLEX (RIGHT TRIANGLE)
let M2_xcor (- simplex-side-length / 2 + shift)              ;; x-coordinate of the left vertex of the right simplex (M).
let M2_ycor (- sqrt(3) * simplex-side-length / 4)            ;; y-coordinate of the left vertex of the right simplex (M).
let L2_xcor (simplex-side-length / 2 + shift)               ;; x-coordinate of the right vertex of the right simplex (L).
let L2_ycor (- sqrt(3) * simplex-side-length / 4)            ;; y-coordinate of the right vertex of the right simplex
             (L).
let H2_xcor 0 + shift                                      ;; x-coordinate of the top vertex of the right simplex (H).
let H2_ycor (sqrt(3) / 4 * simplex-side-length)              ;; y-coordinate of the top vertex of the right simplex (H).

;; ADD A LABEL TO EACH SIMPLEX
ask patch (H1_xcor + 8)(H1_ycor + 17)[set plabel-color white set plabel "INTRATYPE"]            ;; LEFT SIMPLEX
ask patch (H1_xcor + 9)(H1_ycor + 12)[set plabel-color white set plabel "(same type)"]          ;; LEFT SIMPLEX
ask patch (H2_xcor + 9)(H2_ycor + 17)[set plabel-color white set plabel "INTERTYPE"]            ;; RIGHT SIMPLEX
ask patch (H2_xcor + 11)(H2_ycor + 12)[set plabel-color white set plabel "(different type)"]     ;; RIGHT SIMPLEX

let background-color [130 188 183]                        ;; define the background color
ask patches [set pcolor background-color]                  ;; set the background color

let surrounding-simplex-side-length simplex-side-length + 6   ;; set the size of the surrounding simplex.

if decision-rule = "Demand the option that maximizes the expected benefit"
  [
   ;; LEFT SIDE OF THE TRIANGLE: calculate the proportions of M and H in the memory vector so that the expected benefits
            of choosing M and choosing H match when there are no appearances of L in the memory vector (le = left equilibrium)
   let Ale (medium / high)
   let Ble (1 -( medium / high))

   ;; RIGHT SIDE OF THE TRIANGLE: calculate the proportions of L and H in the memory vector so that the expected benefits
            of choosing L and choosing H match when there are no appearances of M in the memory vector (re = right equilibrium)
   let Cre (1 -( low / high))
   let Apede (low / high)

   ;; LOWER SIDE OF THE TRIANGLE: calculate the proportions of L and M in the memory vector so that the expected benefits
            of choosing L and choosing M match when there are no appearances of H in the memory vector (lwe = lower
            equilibrium)
   let Clwe (1 -( low / medium))
   let Blwe (low / medium)

   ;; calculate the proportions of L, M and H in the memory vector so that the expected benefit of choosing L, choosing
            M and choosing H match
   let Aep low / high                  ;; value of A in the equilibrium point (ep)
   let Bep low / medium - low / high     ;; value of B in the equilibrium point (ep)
   let Cep 1 - low / medium             ;; value of C in the equilibrium point (ep)

   let ep-x-coordinate-numerator 1 / ((Cep / (Aep + Cep)) * sin 60) - 1 / tan 60
   let ep-x-coordinate-denominator ((1 / (Cep / (Aep + Cep)) + 1 / (Cep / (Bep + Cep))) * 1 / sin 60 ) - 2 / tan 60
   let ep-x-coordinate ep-x-coordinate-numerator / ep-x-coordinate-denominator * simplex-side-length
   let shifted-ep-x1-coordinate ep-x-coordinate - simplex-side-length / 2 - shift  ;; left triangle
   let shifted-ep-x2-coordinate ep-x-coordinate - simplex-side-length / 2 + shift  ;; right triangle

   let ep-y-coordinate (1 / (((1 / (Cep / (Aep + Cep))) + (1 / (Cep / (Bep + Cep)))) * (1 / sin 60) - 2 / tan 60 )) *
            simplex-side-length
   let shifted-ep-y-coordinate ep-y-coordinate - sqrt(3) / 4 * simplex-side-length

   ;; DRAW THE DECISION BORDERS
   create-turtles 1 [
                    ;;    We    only    draw    the    decision    borders    if    the    memory    type    is    not
            endorsed.                                                                                    ;\\
                   ifelse memory-type = "Endorsed"
                   [
                      ;; LEFT SIMPLEX OUTER TRIANGLE
                      set size 0
                      setxy    (surrounding-simplex-side-length    /    2    -    shift)(-    sqrt(3)    /    4    *
            (surrounding-simplex-side-length - (surrounding-simplex-side-length - simplex-side-length) / 4))
                      set color black set pen-size 5 pd
                      set heading 270 fd surrounding-simplex-side-length    ;; move towards the left vertex (M)
                      set heading 30 fd surrounding-simplex-side-length     ;;move towards the top vertex (H)
                      set heading 150 fd surrounding-simplex-side-length    ;; move towards the right vertex (L)
                      pu
```

```
                ;; RIGHT SIMPLEX OUTER TRIANGLE

                set size 0
                setxy (surrounding-simplex-side-length / 2 + shift)(- sqrt(3) / 4 *
(surrounding-simplex-side-length - (surrounding-simplex-side-length - simplex-side-length) / 4))
                pd
                set color black set pen-size 5 pd
                set heading 270 fd surrounding-simplex-side-length    ;; move towards the left vertex (M)
                set heading 30 fd surrounding-simplex-side-length     ;; move towards the top vertex (H)
                set heading 150 fd surrounding-simplex-side-length    ;; move towards the right vertex (L)
            ]
            [
                ;; LEFT SIMPLEX OUTER TRIANGLE
                set size 0
                setxy (surrounding-simplex-side-length / 2 - shift)(- sqrt(3) / 4 *
                      (surrounding-simplex-side-length - (surrounding-simplex-side-length -
                      simplex-side-length) / 4))
                set color green set pen-size 5 pd
                set heading 270 fd (surrounding-simplex-side-length * Clwe) set color yellow fd
                      (surrounding-simplex-side-length * Blwe)        ;; move towards the left vertex
                      (M)                                                ;\\
                set heading 30 fd (surrounding-simplex-side-length * Ble) set color red fd
                      (surrounding-simplex-side-length * Ale)             ;; move towards the top vertex
                      (H)                                                   ;\\
                set heading 150 fd (surrounding-simplex-side-length * Apede) set color green fd
                      (surrounding-simplex-side-length * Cre) pu    ;; move towards the right vertex
                      (L)                                                ;\\
                pu

                ;; RIGHT SIMPLEX OUTER TRIANGLE
                set size 0
                setxy (surrounding-simplex-side-length / 2 + shift)(- sqrt(3) / 4 *
                      (surrounding-simplex-side-length - (surrounding-simplex-side-length -
                      simplex-side-length) / 4))
                pd
                set color green set pen-size 5 pd
                set heading 270 fd (surrounding-simplex-side-length * Clwe) set color yellow fd
                      (surrounding-simplex-side-length * Blwe)        ;; move towards the left vertex
                      (M)                                                ;\\
                set heading 30 fd (surrounding-simplex-side-length * Ble) set color red fd
                      (surrounding-simplex-side-length * Ale)             ;; move towards the top vertex
                      (H)                                                   ;\\
                set heading 150 fd (surrounding-simplex-side-length * Apede) set color green fd
                      (surrounding-simplex-side-length * Cre) pu    ;; move towards the right vertex
                      (L)                                                ;\\

                ;; LEFT SIMPLEX DECISION BORDERS
                ;; right border
                setxy shifted-ep-x1-coordinate shifted-ep-y-coordinate
                set color black set pen-size 1 pd move-to patch (simplex-side-length / 2 - Cre * simplex-side-length
                      * cos 60 - shift) (Cre * simplex-side-length * sin 60 - sqrt(3) / 4 * simplex-side-length)
                      pu
                ;; left border
                setxy shifted-ep-x1-coordinate shifted-ep-y-coordinate
                set color black set pen-size 1 pd move-to patch (Ble * simplex-side-length * cos 60 -
                      simplex-side-length / 2 - shift) (Ble * simplex-side-length * sin 60 - sqrt(3) / 4 *
                      simplex-side-length) pu
                ;; lower border
                setxy shifted-ep-x1-coordinate shifted-ep-y-coordinate
                set color black set pen-size 1 pd move-to patch ( simplex-side-length / 2 - Clwe * simplex-side-length
                      - shift) (- sqrt(3) / 4 * simplex-side-length) pu

                ;; RIGHT SIMPLEX DECISION BORDERS
                ;; right border
                setxy shifted-ep-x2-coordinate shifted-ep-y-coordinate
                set color black set pen-size 1 pd move-to patch (simplex-side-length / 2 - Cre * simplex-side-length
                      * cos 60 + shift) (Cre * simplex-side-length * sin 60 - sqrt(3) / 4 * simplex-side-length)
                      pu
                ;; left border
                setxy shifted-ep-x2-coordinate shifted-ep-y-coordinate
                set color black set pen-size 1 pd move-to patch (Ble * simplex-side-length * cos 60 -
                      simplex-side-length / 2 + shift) (Ble * simplex-side-length * sin 60 - sqrt(3) / 4 *
                      simplex-side-length) pu
                ;; lower border
                setxy shifted-ep-x2-coordinate shifted-ep-y-coordinate
                set color black set pen-size 1 pd move-to patch ( simplex-side-length / 2 - Clwe * simplex-side-length
```

```
                                + shift) (- sqrt(3) / 4 * simplex-side-length) pu

                        ]
                        die
                ]

    ;; ADD A LABEL TO THE SIMPLEX VERTICES
      ask patch (M1_xcor - 5) (M1_ycor - 2) [set plabel-color yellow set plabel "MEDIUM 50"]
      ask patch (L1_xcor + 18)(L1_ycor - 2) [set Plabel-color 54 set plabel WORD "LOW " low]
      ask patch (H1_xcor + 4) (H1_ycor + 6) [set Plabel-color RED set plabel WORD "HIGH " high]

      ask patch (M2_xcor - 5) (M2_ycor - 2) [set plabel-color yellow set plabel "MEDIUM 50"]
      ask patch (L2_xcor + 18)(L2_ycor - 2) [set Plabel-color 54 set plabel WORD "LOW " low]
      ask patch (H2_xcor + 4) (H2_ycor + 6) [set Plabel-color RED set plabel WORD "HIGH " high]
]

if decision-rule = "Choose the best reply againts the opponents' most frequent demand"
[
      let ep-x-coordinate simplex-side-length / 2
      let shifted-ep-x-coordinate ep-x-coordinate - simplex-side-length / 2
      let ep-y-coordinate (sqrt (3) / 6) * simplex-side-length
      let shifted-ep-y-coordinate ep-y-coordinate - (sqrt (3) / 4) * simplex-side-length

      ;; DRAW THE DECISION BORDERS
      create-turtles 1 [
                                                                                                  ;\\

                    ;; LEFT SIMPLEX OUTER TRIANGLE
                    set size 0
                    setxy (surrounding-simplex-side-length / 2 - shift)(- sqrt(3) / 4 * (surrounding-simplex-side-length
                          - (surrounding-simplex-side-length - simplex-side-length) / 4))
                    set color green set pen-size 5 pd
                    set heading 270 fd (surrounding-simplex-side-length * 1 / 2) set color YELLOW fd
                          (surrounding-simplex-side-length * 1 / 2)
                    set heading 30 fd (surrounding-simplex-side-length * 1 / 2) set color RED fd
                          (surrounding-simplex-side-length * 1 / 2)
                    set heading 150 fd (surrounding-simplex-side-length * 1 / 2) set color 54 fd
                          (surrounding-simplex-side-length * 1 / 2) pu
                    ;; DECISION BORDERS
                    set color black
                    set pen-size 1
                    ;; right border
                    setxy shifted-ep-x-coordinate - shift shifted-ep-y-coordinate
                    pd move-to patch (simplex-side-length / 4 - shift) 0 pu
                    ;; left border
                    setxy shifted-ep-x-coordinate - shift shifted-ep-y-coordinate
                    pd move-to patch (- simplex-side-length / 4 - shift) 0 pu
                    ;; lower border
                    setxy shifted-ep-x-coordinate - shift shifted-ep-y-coordinate
                    pd move-to patch (0 - shift) (- sqrt (3) / 4 * simplex-side-length)
                    pu

                    ;; RIGHT SIMPLEX OUTER TRIANGLE
                    setxy (surrounding-simplex-side-length / 2 + shift)(- sqrt(3) / 4 * (surrounding-simplex-side-length
                          - (surrounding-simplex-side-length - simplex-side-length) / 4))
                    set color green set pen-size 5 pd
                    set heading 270 fd (surrounding-simplex-side-length * 1 / 2) set color YELLOW fd
                          (surrounding-simplex-side-length * 1 / 2)
                    set heading 30 fd (surrounding-simplex-side-length * 1 / 2) set color RED fd
                          (surrounding-simplex-side-length * 1 / 2)
                    set heading 150 fd (surrounding-simplex-side-length * 1 / 2) set color 54 fd
                          (surrounding-simplex-side-length * 1 / 2) pu
                    ;; DECISION BORDERS
                    set color black
                    set pen-size 1
                    ;; right border
                    setxy shifted-ep-x-coordinate + shift shifted-ep-y-coordinate
                    pd move-to patch (simplex-side-length / 4 + shift) 0 pu
                    ;; left border
                    setxy shifted-ep-x-coordinate + shift shifted-ep-y-coordinate
                    pd move-to patch (- simplex-side-length / 4 + shift) 0 pu
                    ;; lower border
                    setxy shifted-ep-x-coordinate + shift shifted-ep-y-coordinate
                    pd move-to patch (0 + shift) (- sqrt (3) / 4 * simplex-side-length)
                    die
                ]
```

```
      ;; ADD A LABEL TO THE SIMPLEX VERTICES
      ask patch (M1_xcor - 5) (M1_ycor - 2) [set plabel-color yellow set plabel "MEDIUM 50"]
      ask patch (L1_xcor + 18)(L1_ycor - 2) [set Plabel-color 54 set plabel WORD "LOW " low]
      ask patch (H1_xcor + 4) (H1_ycor + 6) [set Plabel-color RED set plabel WORD "HIGH " high]

      ask patch (M2_xcor - 5) (M2_ycor - 2) [set plabel-color yellow set plabel "MEDIUM 50"]
      ask patch (L2_xcor + 18)(L2_ycor - 2) [set Plabel-color 54 set plabel WORD "LOW " low]
      ask patch (H2_xcor + 4) (H2_ycor + 6) [set Plabel-color RED set plabel WORD "HIGH " high]
 ]

end

to check-stop-conditions

 if memory-type != "Endorsed"
    [
     ;; THREE POSSIBLE SCENARIOS FOR INTRATYPE MATCHES
     set intratype-equitable-equilibrium-stop-condition all? agents [length filter [? = medium] intratype-memory >= (1
              - epsilon) * m]
     set intratype-fractious-state-stop-condition all? agents [length filter [? = medium] intratype-memory <= epsilon
              * m]
     set intratype-segregation-stop-condition (all? agents with [tag = black] [length filter [? = medium] intratype-memory
              <= epsilon * m] and all? agents with [tag = orange] [length filter [? = medium] intratype-memory >= (1 -
              epsilon) * m]) or
              (all? agents with [tag = orange] [length filter [? = medium] intratype-memory <= epsilon * m] and all? agents
              with [tag = black] [length filter [? = medium] intratype-memory >= (1 - epsilon) * m])

     ;; TWO POSSIBLE SCENARIOS FOR INTERTYPE MATCHES
     set intertype-equitable-equilibrium-stop-condition all? agents [length filter [? = medium] intertype-memory >= (1
              - epsilon) * m]
     set intertype-segregation-stop-condition (all? agents with [tag = black] [length filter [? = high] intertype-memory
              >= (1 - epsilon) * m] and all? agents with [tag = orange] [length filter [? = low] intertype-memory >= (1
              - epsilon) * m]) or
                                    (all? agents with [tag = orange] [length filter [? = high] intertype-memory >=
              (1 - epsilon) * m] and all? agents with [tag = black] [length filter [? = low] intertype-memory >= (1 -
              epsilon) * m])
    ]

 if memory-type = "Endorsed"
    [

     let S (2 + (m - 1) * d) / 2 * m
     ;; THREE POSSIBLE SCENARIOS FOR INTRATYPE MATCHES
     set intratype-equitable-equilibrium-stop-condition all? agents [check-endorsed-memory (intratype-memory) >= (1 -
              epsilon) * S ]
     set intratype-fractious-state-stop-condition all? agents [check-endorsed-memory (intratype-memory) <= epsilon * S ]
     set intratype-segregation-stop-condition (all? agents with [tag = black] [check-endorsed-memory (intratype-memory)
              <= epsilon * S ] and all? agents [check-endorsed-memory (intratype-memory) >= (1 - epsilon) * S ]) or
                                    (all? agents with [tag = orange] [check-endorsed-memory (intratype-memory) <=
              epsilon * S ] and all? agents with [tag = black] [check-endorsed-memory (intratype-memory) >= (1 - epsilon)
              * S ])

     ;; TWO POSSIBLE SCENARIOS FOR INTERTYPE MATCHES

     set intertype-equitable-equilibrium-stop-condition all? agents [check-endorsed-memory (intertype-memory) >= (1 -
              epsilon) * S ]
     set intertype-segregation-stop-condition (all? agents with [tag = black] [check-endorsed-memory-H (intertype-memory)
              >= (1 - epsilon) * S] and all? agents with [tag = orange] [check-endorsed-memory-L (intertype-memory) >=
              (1 - epsilon) * S]) or
                                    (all? agents with [tag = orange] [check-endorsed-memory-H (intertype-memory) >=
              (1 - epsilon) * S] and all? agents with [tag = black] [check-endorsed-memory-L (intertype-memory) >= (1
              - epsilon) * S])

    ]

 ;; 1. INTRATYPE MATCHES

    ;; 1.1. THE SYSTEM HAS REACHED AN EQUITABLE EQUILIBRIUM (intratype matches)
    if intratype-equitable-equilibrium-stop-condition = true and intratype-simulation-finished? = false
      [
       print (word "* * * INTRATYPE MATCHES: The system has reached an equitable equilibrium in the iteration number "
              elapsed-iterations " * * *")
       set intratype-status "EQUITABLE EQUILIBRIUM"       ;; display "EQUITABLE EQUILIBRIUM" in the system-status monitor
       set intratype-simulation-finished? true           ;; the simulation for intratype matches is now finished.
      ]
```

```
    ;; 1.2. THE SYSTEM HAS REACHED A FRACTIOUS STATE (intratype matches)
    if intratype-fractious-state-stop-condition = true and intratype-simulation-finished? = false
      [
       print (word "* * * INTRATYPE MATCHES: The system has reached a fractious state in the iteration number " elapsed-iterations
              " * * *")
       set intratype-status "FRACTIOUS STATE"            ;; display "FRACTIOUS STATE" in the system-status monitor
       set intratype-simulation-finished? true           ;; the simulation for intratype matches is now finished.
      ]

    ;; 1.3. INTRATYPE-SEGREGATION: AGENTS WITH ONE COLOUR -> EQUITABLE EQUILIBRIUM AND AGENTS WITH THE OTHER COLOUR ->
              FRACTIOUS STATE (intratype matches)
    if intratype-segregation-stop-condition = true and intratype-simulation-finished? = false
      [
       print (word "* * * INTRATYPE MATCHES: Segregation has emerged in the iteration number " elapsed-iterations " * * *")
       set intratype-status "SEGREGATION"               ;; display "SEGREGATION" in the system-status monitor
       set intratype-simulation-finished? true          ;; the simulation for intratype matches is now finished.
      ]

   ;; 2. INTERTYPE MATCHES

    ;; 2.1. THE SYSTEM HAS REACHED AN EQUITABLE EQUILIBRIUM (intertype matches)
    if intertype-equitable-equilibrium-stop-condition = true and intertype-simulation-finished? = false
      [
       print (word "* * * INTERTYPE MATCHES: The system has reached an equitable equilibrium in the iteration number "
              elapsed-iterations " * * *")
       set intertype-status "EQUITABLE EQUILIBRIUM"      ;; display "EQUITABLE EQUILIBRIUM" in the system-status monitor
       set intertype-simulation-finished? true           ;; the simulation for intratype matches is now finished.
      ]

    ;; 2.2. INTERTYPE-SEGREGATION: AGENTS WITH ONE COLOUR -> high AND AGENTS WITH THE OTHER COLOUR -> low (intertype matches)
    if intertype-segregation-stop-condition = true and intertype-simulation-finished? = false
      [
       print (word "* * * INTERTYPE MATCHES: Segregation has emerged in the iteration number " elapsed-iterations " * * *")
       set intertype-status "SEGREGATION"               ;; display "SEGREGATION" in the system-status monitor
       set intertype-simulation-finished? true          ;; the simulation for intratype matches is now finished.
      ]
end

to-report check-endorsed-memory  [agents-memory]
  let q 0
  (foreach agents-memory endorsed-memory-weights [if ?1 = medium [set q q + ?2]])
  report q    ;; returns the number of elements equal to medium in the agent's memory (each appearance of medium in the
              memory set is multiplied by the weight of the position in which it appears)
end

to-report check-endorsed-memory-H [agents-memory]
  let q 0
  (foreach agents-memory endorsed-memory-weights [if ?1 = high [set q q + ?2]])
  report q    ;; returns the number of elements equal to high in the agent's memory (each appearance of high in the memory
              set is multiplied by the weight of the position in which it appears)
end

to-report check-endorsed-memory-L [agents-memory]
  let q 0
  (foreach agents-memory endorsed-memory-weights [if ?1 = low [set q q + ?2]])
  report q    ;; returns the number of elements equal to low in the agent's memory (each appearance of low in the memory
              set is multiplied by the weight of the position in which it appears)
end
```