**README for William Cutler’s PHIL 2001 Final Project Code**

**Reputation in the Ultimatum Game**

**Introduction:**

This README describes the inner workings of my computer simulation of Reputation scores in the Ultimatum Bargaining game. It is written in Python version 3.8.0 and I use the Python IDE “Pyzo”, downloadable here: <https://pyzo.org/index.html>.

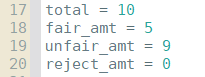
This document is split into two main parts, “Quickstart” and “Advanced”, based on the programming/Python experience of the user. For those with less Python experience, this README will explain how to change critical variables that apply to the entire simulation, plug in variables to guide the simulation, and interpret the output of the main simulation function. The second part will go into some detail about the inner workings of important functions and processes and describe the ways in which the core game structure can be changed.

**Quick-Start:**

The main variables that can be manipulated can be found in the first 20 lines of code as shown in the following pictures.



* reputation\_bias : A real number ranging from 0 to 1 inclusive. This determines the probability that an agent, when in the proposer role, will look to the responder’s previous history in the face of an unfair offer to guide whether the proposal will be fair or unfair. Otherwise, the proposer simply follows his default strategy. At a value of 0, it is as if the reputation does not exist and resembles the standard ultimatum game. This is the main variable manipulated in the final project.
* convergence\_threshold : A real number ranging from 0 to 1 inclusive. This determines the proportion of the whole population a strategy must reach to be considered “convergent”.
* tremble\_threshold : A real number ranging from 0 to 1 inclusive. This is the probability that a player will make a “mistake” and act in the opposite manner he intended, and is currently applied to both the Proposer and the Responder.



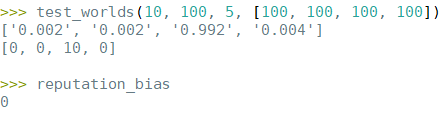
These values can be manipulated to alter the payoff structure of the game.

* total : The total utility to be divided (only applied to unfair offers)
* fair\_amt : The utility given to both players if a fair offer is proposed (is automatically accepted)
* unfair\_amt : The utility the proposer receives if his unfair offer is accepted. Responder receives (total – unfair\_amt)
* reject\_amt : The utility both players receive if the offer is rejected

To begin operating tests, run the program in the IDE, and in the shell/IDLE where the program is run, call the “test\_worlds” function. It takes in the following parameters, in order:

test\_worlds(Number of tests, Generations, Games, Population[Easy Riders, Fairmen, Gamesmen, Hypocrites])

1. Number of tests : A natural number that dictates the number of tests that will be run
2. Generations : A natural number that dictates the number of generations each test will contain
3. Games : A natural number that dictates the number of games played before the next generations
4. Population : An array of 4 Natural numbers separated by commas, representing the initial populations of each strategy at the beginning of each test



This is an example of a test\_worlds call with a reputation\_bias of 0, and with 10 tests, 100 generations, 5 games per generation, and initial populations of 100 of each strategy.

The first line of output is an array representing the average final proportions of each strategy (Easy Riders, Fairmen, Gamesmen, and Hypocrites) [e, f, g, h]

The second line of output is the count of the strategies that are convergent according to the convergent\_threshold global variable.

**Advanced:**

This section will not lay out the inputs and outputs of each function and their descriptions as that can be found commented in the code itself. Rather, it will discuss the implementation of certain non-trivial processes and places in the program for modification.

**Discrete Replicator Dynamics:**

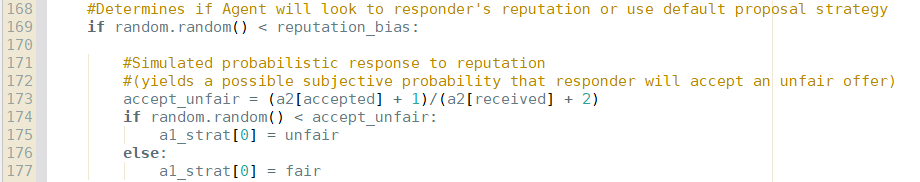


The simulation uses this equation, where the proportion of a strategy in the next generation is equal to its current proportion multiplied by the ratio of the strategy’s fitness to the average fitness. I implement a modified version using numbers of strategies rather than proportions, achieved by multiplying both sides of the equation by the total population. Then, the number of agents employing strategy ‘x’ multiplied by the average fitness of strategy ‘x’ is equal to the total fitness/utility gained across all agents using that strategy. This number is measured by iterating through every agent of a given strategy and adding its utility to a count. The total utility of a strategy is divided by the average utility of the entire population, yielding the number of agents in the next generation.

**Probabilistic Reputation Implementation:**

One “reject” and one “accept” labeled ball are placed in a bin, plus another “accept” ball for every offer the responder previously accepted and another “reject” ball for rejections (number of unfair offers received minus number accepted). One ball is chosen at random, and the proposer makes the offer best suited to the responder acting according to the chosen ball. This probabilistic response is summarized by , as the probability that the proposer will make an unfair offer, given that he is looking to the responder’s reputation.

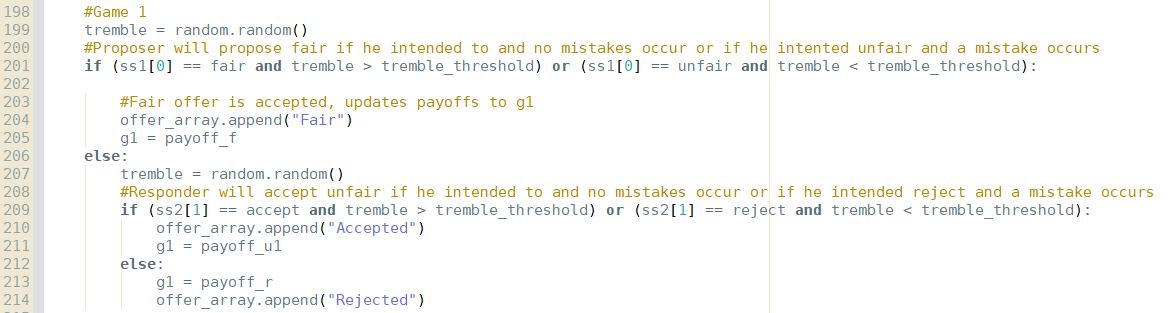
This is the Python implementation of it, found within the “agent\_play” function.



The fraction summary occurs on line 173, only after it has been determined that the proposer will look at the responder’s reputation.

**Trembles:**

Currently, the trembles apply equally to both proposer and responder. This however, can be changed, but adding a second tremble variable at the top of the program with the other global variables and applying it to the conditionals present in the “play\_game” function.



The first conditional handles trembles for the proposer, and for the purposes of Game 1, it could be reverted back to the standard, no-tremble version by simply changing the conditional on line 201 to “if(ss1[0] == fair):”. Similarly, the conditional on line 209 deals with trembles for the responder, and can be reverted similarly by changing it to “if (ss2[1] == accept): “

**Game structure:**

The structure of the extended form game is governed solely by the play\_game function. Given a list of strategies for Player 1 and a list of strategies for Player 2, it traverses down the hypothetical extended form tree by selecting the appropriate strategy from each player at each decision point. Therefore, with more conditionals, more strategies can be added and the tree structure can be changed, perhaps to include three different offers, or perhaps to allow the responder to make a counter-offer with a diminished “cake”. No matter what the structural modification intended is, it will most easily occur in the play\_game function.

**Contact Info (for any questions, or just to chat):**

William Cutler - cutler.wi@husky.neu.edu