Supplementary Appendix for "Inequality, Aspirations, and Social Comparisons"

Jonathan Bendor
Stanford University

Daniel Diermeier
Northwestern University

Michael M. Ting
Columbia University

June 2014

This document contains two egodicity results and their proofs. It also contains R code for the simulation program used in our paper.

1 Ergodicity Results

This part of the supplementary appendix demonstrates that the computational model in "Inequality, Aspirations, and Social Comparisons" represents an ergodic process: it has a unique invariant distribution over the state space (action-propensities and aspirations) and it converges to that distribution from any initial probability distribution over the state space.

Establishing ergodicity is relatively straightforward for finite-state Markov chains with stationary transition probabilities. Therefore, before getting into the nuts and bolts of the proof of our first result it is helpful to examine how our general set-up—in particular, A1-A3—and our specific computational model relates to these three basic properties. (For the reader's convenience we reproduce A1-A3 here.)

A1 (positive feedback): If i used action α_i in t and if $\pi_{i,t} \geq a_{i,t}$ then $p_{i,t+1}(\alpha_i) \geq p_{i,t}(\alpha_i)$; this conclusion holds strictly if $p_{i,t}(\alpha_i) < 1$ and $\pi_{i,t} > a_{i,t}$.

A2 (negative feedback): If i used action α_i in t and if $\pi_{i,t} < a_{i,t}$ then $p_{i,t+1}(\alpha_i) \leq p_{i,t}(\alpha_i)$; this conclusion holds strictly if $p_{i,t}(\alpha_i) > 0$.

A3 (aspiration adjustment):

(i) There are $r \geq 1$ $n \times n$ social influence matrices, labeled Λ^q $(1 \leq q \leq r)$. The probabilities $\rho^q(o)$ of selecting Λ^q are fixed and independent over time but may depend on o, the realized vector of payoffs.

(ii) For any realized social influence matrix Λ^q, the weights $\Lambda^q_{i,j}$ are non-negative and on each row i $(i = 1, \ldots, n)$ sum to $1 - \lambda^q_i$, where $\lambda^q_i \in (0, 1)$ is the weight on $a_{i,t}$ in Λ^q.

(iii) For any realized social influence matrix Λ^q, $a_{i,t+1} = \lambda^q_i a_{i,t} + \sum_{j=1}^{n} \Lambda^q_{i,j} \pi_{j,t}$.

Several observations are in order. First, it is clear by inspection of A3 that the adjustment of aspirations is Markovian. By themselves, however, A1 and A2 do not imply that propensity-adjustment has this property. But the computational model uses the standard Bush-Mosteller rule, which is Markovian. Hence we will directly assume in the proposition itself that propensity-adjustment has the Markov property, and we can do this without losing touch with the computational model.

Second, because the computational model is implemented on a finite-state machine (a computer), the simulations are produced by a stochastic process that has finitely many states (values of propensities and aspirations).[1] Hence assuming a Markov process with

[1] The computational model must use a discrete approximation of the weighted average rule. In the context of a proof of ergodicity, this approximation amounts to ensuring that the coarseness of the state space does

finitely many states is consistent with the computational model, and with A1-A3 as well.[2]

Third, A1-A3 neither imply nor preclude the possibility that the process has stationary transition probabilities. Hence we will explicitly assume stationarity in the proposition itself.

Finally, the text of section V refers to two properties that are vital for proving ergodicity that are restricted to the computational model; unlike A1-A3 we do not assume them for the general analytical model presented in section II. These two properties are trembles in propensity-adjustment and inertia regarding both aspiration- and propensity-adjustment. Together, these properties help to ensure that all nontransient states communicate, which guarantees that there is a unique invariant distribution, and aperiodicity, which implies that the process converges to that distribution. Because these two properties are so important we state them here formally. (Because the symbol 'A4' is already used in the text we call these assumptions A5 and A6.)

A5: With probability $\epsilon \in (0,1)$ an agent trembles so that his action-propensities are selected according to a probability distribution with full support over all feasible propensity vectors.

A6: With probability $\omega_p \in (0,1)$ an agent does not adjust his action-propensities. With probability $\omega_a \in (0,1)$ an agent does not adjust her aspirations. These events are independent of each other and over time.

In the following result we assume that a unique social influence matrix governs social comparisons. This is purely to prevent the proof from becoming too cluttered with notation. It is straightforward to extend the result to allow for multiple social influence matrices, provided that they are evoked by a stationary Markov process that is independent of the agents' actions.

Note that Proposition 1-A, unlike the simulations, is not restricted to two-player games.

Proposition 1-A: Suppose A1-A3 and A5-A6 hold. Propensity-adjustment is Markovian and stationary, there are finitely many propensity values and aspiration levels, and a single matrix Λ governs social comparisons. If either (1) or (2) holds then the process is ergodic:

(1) the stage game has a pair of outcomes, o_1 and o_2, such that $\min_i\{\pi_i(o_1)\} > \max_i\{\pi_i(o_2)\}$;

not cause the process to bog down. For example, if $\pi_{i,t} > a_{i,t}$ and given a continuum of aspiration-levels the weighted average rule would yield an $a_{i,t+1}$ that is strictly larger than $a_{i,t}$, then this must continue to hold under the discrete approximation.

[2]For convenience we assume, for all i, that the set of feasible propensity values for agent i includes the extreme propensities—zero and one—for all of i feasible actions. Similarly, we assume for all i that the set of feasible aspiration levels for agent i includes all of i's feasible payoffs.

(2) the stage game is constant-sum and $a_{i,t+1} = \lambda_i a_{i,t} + (1 - \lambda_i)\frac{\sum_{j=1}^{n} \pi_{j,t}}{n}$ for $i = 1, \ldots, n$.

Because Proposition 1-A uses the standard assumption of stationary transition probabilities it does not cover the last simulation in the paper (see Figures 10 and 11). The process in this simulation is nonstationary because it presumes magnitude-sensitive inertia, which depends on an agent's aspiration level, a parameter that varies over time. Fortunately, one can prove that certain special kinds of nonstationary Markov chains are ergodic.

Proposition 2-A: Suppose A1-A6 hold. Propensity-adjustment is Markovian, there are finitely many propensity values and aspiration levels, and a single matrix Λ governs social comparisons. If the stage game is constant-sum and $a_{i,t+1} = \lambda_i a_{i,t} + (1 - \lambda_i)\frac{\sum_{j=1}^{n} \pi_{j,t}}{n}$ for $i = 1, \ldots, n$ then the process is ergodic.

We now provide proofs of both propositions.

Proof of Proposition 1-A

A standard result in the theory of finite Markov chains is that if a stationary finite chain is irreducible and aperiodic then it is ergodic (Feller 1950, p.393-394). Further, since transient states can be disregarded in the limit, if the nontransient states form an irreducible and aperiodic set then the chain must be ergodic. Given the assumptions described above, our Markov chain must be aperiodic, since agents are inertial with positive probability at every state. Hence it only remains to show that the nontransient states form an irreducible set. To show this, the following lemma is useful.

Lemma 1-A: The nontransient states of any finite Markov chain form an irreducible set if the chain has a state which is accessible from all states.

Proof: Call such a state s^*. Since s^* is accessible from all states it must belong to every closed set of states. (A set of states C is "closed" if once the process enters C then it must stay there forever.) By Theorem 3 of Feller (vol. 1, chapter 15, p. 392), any Markov chain can be partitioned, in a unique way, into nonoverlapping sets $T, C_1, C_2, \ldots$, where T is composed of the chain's transient states and the C_k are closed sets. Since these closed sets are disjoint and s^* belongs to each of them, our chain must have only one closed set, and in a finite chain with a unique closed set the nontransient states (the elements of the unique closed set) form an irreducible set. QED.

We now return to the proof of the main result. We first prove case (1): the stage game has a pair of outcomes, o_1 and o_2, such that $\min_i\{\pi_i(o_1)\} > \max_i\{\pi_i(o_2)\}$.

We exploit Lemma 1-A by identifying a state, s^*, that is accessible from all other states. This, together with the aperiodicity provided by inertia, ensures ergodicity. For s^* we nominate the vector of pure propensities corresponding to o_1 and the vector of aspirations that corresponds (given the unique social influence matrix Λ) to the payoffs that occur in o_1.

Suppose that in date t the process is in some arbitrary state s. Because each agent trembles with positive probability to a totally mixed-propensity vector, every outcome can be reached with positive probability in $t+1$. In particular, outcome o_2 can be reached. For agents who are going to get negative feedback (payoffs below aspirations) in o_2 in $t+1$, freeze their propensities at their totally-mixed vector while letting their aspirations adjust. The other agents can be allowed to adjust their propensities as their aspirations adjust. Keep repeating this combination of inertia and noninertia until all aspirations reach the stable aspiration profile associated with the pure propensity-vector corresponding to o_2. (This must happen since there are finitely many aspiration levels.)

At this point use the propensity-tremble to shift the process to o_1, in say date $t+r$. Since the stable aspirations of o_2 are just convex combinations of the payoffs produced in that outcome and since $\min_i\{\pi_i(o_1)\} > \max_i\{\pi_i(o_2)\}$, all agents must be satisfied by the payoffs they get from o_1 in $t+r$. Let player i's propensity to play $\alpha_i(o_1)$ rise in $t+r$, for all i. If i's aspiration in $t+r+1$ would exceed $\pi_i(o_1)$ then freeze i's aspiration at its level of $t+r$; otherwise it may adjust as well. Proceed in this way until all propensities reach the pure values associated with o_1. For those who are getting negative feedback, freeze their propensities at the pure level, while letting their aspirations rise to the stable values for o_1. For agents getting positive feedback, propensities may be frozen or not. Since each agent has finitely many aspiration values, this process must reach the profile of stable aspirations for o_1. Since propensities were either frozen at 1.0 (for agents getting negative feedback) or were maintained there by positive feedback, the process reaches the combination of propensity values and aspiration levels that defines s^*. QED.

Case (2): the stage game is constant-sum and $a_{i,t+1} = \lambda_i a_{i,t} + (1-\lambda_i)\frac{\sum_{j=1}^{n}\pi_{j,t}}{n}$ for $i = 1, \ldots, n$.

We know from the proof of Remark 1 that $a_{i,t}$ converges to $\frac{b}{n}$ as $t \to \infty$ when the properties of Case (2) hold.[3] It follows that in every nontransient state all aspirations equal $\frac{b}{n}$. Regarding propensities, A5, the assumption about trembles, implies that the process can go from any set of propensity-vectors to any other set of propensity-vectors in one step with

[3]Strictly speaking, this convergence holds when aspirations can take on a continuum of values. Since here we are assuming that there are finitely many possible aspiration levels, we use the approximation-assumption discussed in footnote 1. Thus, the coarseness of the grid of state-space values does not prevent the a_t's from reaching $\frac{b}{n}$.

positive probability. Consequently, all nontransient states communicate. Hence the set of nontransient states is irreducible, which ensures that there is a unique invariant distribution. This, together with the aperiodicity produced by inertia, yields ergodicity. QED

Proof of Proposition 2-A

The proof is in two steps. The first step is to point out that although the sequence of probability transition matrices, $P_1, P_2, \ldots$, depends on time (i.e., the process is nonstationary), it converges to a stationary P_∞ as $t \to \infty$. The argument is the same as that of part (i) of Remark 1. (The generalization here to propensity-adjustment rules other than the bang-bang Bush-Mosteller has no effect on this argument: the nonstationarity of the transition matrices arises only because of the nonstationarity of the a_t's, and these must converge to $\frac{b}{n}$ as $t \to \infty$.)

The second step is to show that P_∞ represents an ergodic process. Here we once again use A5: the process can go from any set of propensity-vectors to any other set of vectors in one step with positive probability. This, together with the convergence of aspirations to $\frac{b}{n}$, implies that all nontransient states communicate. Finally, A4 ensures that the process is inertial wpp in every state. QED

2 Simulation Programs

The simulation program is written in R and consists of four files, as follows:

1. `BDT-games-psrm.R` — the main program; contains most simulation parameters.
2. `templateMgr-BDT-140616.R` — contains parameters of the games and behavioral models.
3. `gameClassDefs-140616.R` — a general framework for playing normal form games with adaptively rational agents.
4. `repeatGamePlot.R` — generates histogram plots for each outcome.

To run the simulations, execute the file `BDT-games-psrm.R`. The program requires the `multicore` package, and program assumes that files are located in the directory `~/Dropbox/Collective Action/software`; this setting may be changed in the `BDT-games-psrm.R` file. This file also allows the user to change the number of simulations and the number of periods to be run.

The following subsections contain the code, which can also be downloaded from: `http://www.columbia.edu/~mmt2033/BDT-games.zip`

2.1 File 'BDT-games-psrm.R'

```
rm(list=ls())

# Start the clock!
processingTime <- proc.time()

setwd('~/Dropbox/Collective Action/software')
source("gameClassDefs-140616.R")
source("repeatGamePlot.R")
source("templateMgr-BDT-140616.R")

library(multicore)

# choose agents and game payoffs from template file 'templateMgr-BDT-140616.R'
# to add new agent or game types, edit templateMgr-BDT.R and run:
updateGameTemplates(getwd())
load("gameTemplates-BDT.Rimage")

# setwd('~/Dropbox/Collective Action/software')
set.seed(2045635090, kind="Mersenne-Twister")

# function to run one game session
# reports fraction of time that each player cooperated
runGame <- function(iter, payoffList, nPlays, agentClass, agentProto1, agentProto2) {
    p1 <- new(agentClass, agentProto1)
    p2 <- new(agentClass, agentProto2)
    game1 <- new("normalGame", payoffs=payoffList, players=list(p1,p2))
    repeatPlay(game1, nPlays)
}

nSim <- as.integer(1000)  # change number of simulations
nPlay <- as.integer(1000) # change number of periods
abarType <- "bm"

# the following are payoff matrices from 'templateMgr-BDT-140616.R'.

BDT.V1 <- gameTemplate$BDT.V6a
BDT.V2 <- gameTemplate$BDT.V6c
BDT.V3 <- gameTemplate$BDT.V3
BDT.V4 <- gameTemplate$BDT.V7c
BDT.V5 <- gameTemplate$BDT.V8a

for (gameName in c("BDT.V5")) {
    pL <- get(gameName)
    pM <- matrix(data=unlist(pL), nrow=(length(unlist(pL)) / length(pL)),ncol=length(pL))
```

```
    # to change behavior rule, replace instances of agentTemplate$xxx with relevant model
    # from 'templateMgr-BDT-140616.R'

    # asocial version

    cat(paste("Running Sims:",gameName,", asocial...\n",sep=" "))
    aC <- "abarAgent" # switch to "abarAgent" to use non-reference group player types,
      "socialAgent" for social types
    ap1 <- createAgentProto(type=aC, abar=abarType, id = as.integer(1), nActions=as.integer(2),
      paramsList=agentTemplate$bmPropInertiaSensAsocial)
    ap2 <- createAgentProto(type=aC, abar=abarType, id = as.integer(2), nActions=as.integer(2),
      paramsList=agentTemplate$bmPropInertiaSensAsocial)
    allOutcomes <- do.call(rbind, mclapply(1:nSim, FUN=runGame, payoffList=pL, nPlays=nPlay,
      agentClass=aC, agentProto1=ap1, agentProto2=ap2))
    rgPlotWithStates(allOutcomes, dims=c(2,2), filename=paste(gameName,"-asocial.pdf",sep=""))
    rm(ap1, ap2, allOutcomes)

    # social version

    cat(paste("Running Sims:",gameName,", social...\n",sep=" "))
    aC <- "socialAgent"
    ap1 <- createAgentProto(type=aC, abar=abarType, id = as.integer(1), nActions=as.integer(2),
      paramsList=agentTemplate$bmPropInertiaSens, refGrp=as.integer(2))
    ap2 <- createAgentProto(type=aC, abar=abarType, id = as.integer(2), nActions=as.integer(2),
      paramsList=agentTemplate$bmPropInertiaSens, refGrp=as.integer(1))
    allOutcomes <- do.call(rbind, mclapply(1:nSim, FUN=runGame, payoffList=pL, nPlays=nPlay,
      agentClass=aC, agentProto1=ap1, agentProto2=ap2))
    rgPlotWithPayoffs(allOutcomes, dims=c(2,2), payoffs=pM, filename=paste(gameName,
      "-social.pdf",sep=""))
    rm(ap1, ap2, allOutcomes)
}

# Stop the clock
proc.time() - processingTime
```

2.2 File 'templateMgr-BDT-140616.R'

```
# generate game and player templates to save to disk
# note - to run, the game and agent classes must be defined

updateGameTemplates <- function(saveDir) {
    setwd(saveDir)
    agentTemplate <- list()
    gameTemplate <- list()

    agentTemplate$bmAlpha01 <-list(aspiration=1.5,
        prevAspWgt=0.7,
        refWgt=0.15,
        refGrp=as.integer(2),
        extraParms=list(alphaFunction=function(x){return(0.1)},
            propInertiaFct=function(x){return(0.01)},
```

```
        aspInertiaProb=0.01),
        trembleProb=0.01)

agentTemplate$bmAlpha01asocial <-list(aspiration=1.5,
    prevAspWgt=0.7,
    extraParms=list(alphaFunction=function(x){return(0.1)},
        propInertiaFct=function(x){return(0.01)},
        aspInertiaProb=0.01),
    trembleProb=0.01)

agentTemplate$bmMagSens <- list(aspiration=1.5,
    prevAspWgt=0.7,
    refWgt=0.15,
    refGrp=as.integer(2),
    extraParms=list(alphaFunction=function(x){return(1 - exp(-x^2))},
        propInertiaFct=function(x){return(0.01)},
        aspInertiaProb=0.01),
    trembleProb=0.01)

agentTemplate$bmMagSensAsocial <- list(aspiration=1.5,
    prevAspWgt=0.7,
    extraParms=list(alphaFunction=function(x){return(1 - exp(-x^2))},
        propInertiaFct=function(x){return(0.01)},
        aspInertiaProb=0.01),
    trembleProb=0.01)

agentTemplate$bmPropInertiaSens <-list(aspiration=1.5,
    prevAspWgt=0.7,
    refWgt=0.15,
    refGrp=as.integer(2),
    extraParms=list(alphaFunction=function(x){return(0.1)},
        propInertiaFct=function(x){return(exp(-x^2))},
        aspInertiaProb=0.01),
    trembleProb=0.01)

agentTemplate$bmPropInertiaSensAsocial <-list(aspiration=1.5,
    prevAspWgt=0.7,
     extraParms=list(alphaFunction=function(x){return(0.1)},
        propInertiaFct=function(x){return(exp(-x^2))},
        aspInertiaProb=0.01),
    trembleProb=0.01)

payoffMat1 <- matrix(data=c(8.01,1,0,2), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(8,0,1,2.01), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V1 <- payoffList

payoffMat1 <- matrix(data=c(100.01,1,0,2), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(100,0,1,2.01), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V1a <- payoffList
```

```
payoffMat1 <- matrix(data=c(8.01,1,0,2), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(8,0,1,2), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V2 <- payoffList

payoffMat1 <- matrix(data=c(8,1,0,2), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(8,0,1,2), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V2a <- payoffList

payoffMat1 <- matrix(data=c(1,3,2,0), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(1,2,3,0), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V3 <- payoffList

payoffMat1 <- matrix(data=c(2,1,1,0), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(2,1,1,0), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V4 <- payoffList

payoffMat1 <- matrix(data=c(2,1,2,0), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(2,2,1,0), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V5 <- payoffList

payoffMat1 <- matrix(data=c(3,1,0,2), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(3,0,1,2), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V6a <- payoffList

payoffMat1 <- matrix(data=c(3.001,1,0,2.001), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(3,0,1,2), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V6b <- payoffList

payoffMat1 <- matrix(data=c(3.01,1,0,2.01), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(3,0,1,2), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V6c <- payoffList

payoffMat1 <- matrix(data=c(1,3,2,0), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(1,2,3,0), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V7a <- payoffList

payoffMat1 <- matrix(data=c(1.001,3,2,0.001), nrow=2,ncol=2)
payoffMat2 <- matrix(data=c(1,2,3,0), nrow=2, ncol=2)
payoffList <- list(p1=payoffMat1, p2=payoffMat2)
gameTemplate$BDT.V7b <- payoffList
```

```
    payoffMat1 <- matrix(data=c(1.01,3,2,0.01), nrow=2,ncol=2)
    payoffMat2 <- matrix(data=c(1,2,3,0), nrow=2, ncol=2)
    payoffList <- list(p1=payoffMat1, p2=payoffMat2)
    gameTemplate$BDT.V7c <- payoffList

    payoffMat1 <- matrix(data=c(0.01,-100,-100,10), nrow=2,ncol=2)
    payoffMat2 <- matrix(data=c(-0.01,100,100,-10), nrow=2, ncol=2)
    payoffList <- list(p1=payoffMat1, p2=payoffMat2)
    gameTemplate$BDT.V8a <- payoffList

    payoffMat1 <- matrix(data=c(0.01,-3,-3,2), nrow=2,ncol=2)
    payoffMat2 <- matrix(data=c(-0.01,3,3,-2), nrow=2, ncol=2)
    payoffList <- list(p1=payoffMat1, p2=payoffMat2)
    gameTemplate$BDT.V8b <- payoffList

    payoffMat1 <- matrix(data=c(1,-3,-3,2), nrow=2,ncol=2)
    payoffMat2 <- matrix(data=c(-1,3,3,-2), nrow=2, ncol=2)
    payoffList <- list(p1=payoffMat1, p2=payoffMat2)
    gameTemplate$BDT.V8c <- payoffList

    save(agentTemplate, gameTemplate, file="gameTemplates-BDT.Rimage")
}

createAgentProto <- function(type, id, nActions, abar=NULL, paramsList=NULL, refGrp=NULL) {

    if (type == "Agent") {
        agent <- new("Agent", id=as.integer(id), nActions=nActions)
    } else if (type == "abarAgent") {
        if (abar == "satisficing") {
            a <- paramsList$aspiration
            pw <- paramsList$prevAspWgt
            ep <- paramsList$extraParms
            tp <- paramsList$trembleProb
            agent <- new("abarAgent", id=as.integer(id), trembleProb=tp,
              propUpdateFunction=propUpdateSatisfice, nActions=nActions, aspiration=a,
              prevAspWgt=pw,extraParms=ep)
        } else if (abar == "bm") {
            a <- paramsList$aspiration
            pw <- paramsList$prevAspWgt
            ep <- paramsList$extraParms
            tp <- paramsList$trembleProb
            agent <- new("abarAgent", id=as.integer(id), trembleProb=tp,
              propUpdateFunction=propUpdateBM, nActions=nActions, aspiration=a,
              prevAspWgt=pw,extraParms=ep)
          }
        } else if (type == "socialAgent") {
            if (abar == "satisficing") {
                a <- paramsList$aspiration
                pw <- paramsList$prevAspWgt
                ep <- paramsList$extraParms
                rw <- paramsList$refWgt
```

```
                tp <- paramsList$trembleProb
                agent <- new("socialAgent", id=as.integer(id), trembleProb=tp,
                          propUpdateFunction=propUpdateSatisfice, nActions=nActions,
                          aspiration=a,prevAspWgt=pw,refWgt=rw,refGrp=as.integer(refGrp),
                          extraParms=ep)
            } else if (abar == "bm") {
                a <- paramsList$aspiration
                pw <- paramsList$prevAspWgt
                ep <- paramsList$extraParms
                rw <- paramsList$refWgt
                tp <- paramsList$trembleProb
                agent <- new("socialAgent", id=as.integer(id), trembleProb=tp,
                propUpdateFunction=propUpdateBM, nActions=nActions, aspiration=a,
                prevAspWgt=pw,refWgt=rw,refGrp=as.integer(refGrp),extraParms=ep)
              }
      }
    return(agent)
}
```

2.3 File 'gameClassDefs-140616.R'

```
# GENERIC method definitions - GAME
setGeneric("play", function(object, ...) standardGeneric("play"))
setGeneric("repeatPlay", function(object, ...) standardGeneric("repeatPlay"))
setGeneric("repeatPlayCorr", function(object, ...) standardGeneric("repeatPlayCorr"))
setGeneric("reportPayoffs", function(object, ...) standardGeneric("reportPayoffs"))
setGeneric("updatePlayers", function(object, ...) standardGeneric("updatePlayers"))
setGeneric("getOutcomeNumber", function(object, ...) standardGeneric("getOutcomeNumber"))

# GENERIC method definitions - AGENT
setGeneric("getAction", function(object, ...) standardGeneric("getAction"))
setGeneric("getRelevantPayoffs", function(object, payoffs, ...)
  standardGeneric("getRelevantPayoffs"))
setGeneric("setAspiration", function(object, ...) standardGeneric("setAspiration"))
setGeneric("setPropensity", function(object, ...) standardGeneric("setPropensity"))

# UTILITY functions to add / drop players from a game
attachPlayer <- function(game, player, id) {
    if (id <= length(game@players)){
        player@id <- as.integer(id)
        game@players[[id]] <- player
        #cat(paste("Player",id,"added successfully.","\n",sep=" "))
    }
    return(game)
}

dropPlayers <- function(game) {
    for (j in 1:length(game@players)) {
        game@players[[j]] <- NULL
    }
    return(game)
```

```
}

# CLASS Game - basic class that other game types extend
setClass("Game", representation(players="list", lastPayoffs="numeric", lastActions="integer"))

setMethod("initialize", "Game", function(.Object, ...) {
    .Object <- callNextMethod(.Object, ...)
    nPlayers <- length(.Object@players)
    .Object@lastPayoffs <- vector(mode="numeric", length=nPlayers)
    .Object@lastActions <- vector(mode="integer", length=nPlayers)
    return(.Object)
})

setMethod("updatePlayers", "Game", function(object) {
    for (i in 1:length(object@players)) {
        object@players[[i]] <- getRelevantPayoffs(object@players[[i]], object@lastPayoffs)
        object@players[[i]] <- setPropensity(object@players[[i]])
        object@players[[i]] <- setAspiration(object@players[[i]])
    }
    return(object)
})

# CLASS normalGame - represent payoffs by matrix of dimension (players * actions) for every player
setClass("normalGame",representation(payoffs="list",  nOutcomes="numeric"), contains="Game")

setMethod("initialize", "normalGame", function(.Object, payoffs, ...) {
    .Object <- callNextMethod(.Object, ...)
    .Object@payoffs <- payoffs
    .Object@nOutcomes <- prod(dim(.Object@payoffs[[1]]))
    .Object
})

setMethod("play", "normalGame", function(object) {
    # get all actions from players
    action <- vector(mode="integer", length=length(object@players))
    for (i in 1:length(object@players)) {
        action[i] <- getAction(object@players[[i]])
        object@players[[i]]@lastAction <- as.integer(action[i])
    }
    # plug into payoff matrices
    payoff <- vector(mode="numeric", length=length(object@players))
    for (i in 1:length(payoff)) {
        payoffMatrixI <- object@payoffs[[i]]
        payoff[i] <- payoffMatrixI[matrix(data=action, nrow=1, ncol=length(action))]
    }
    object@lastPayoffs <- payoff
    object@lastActions <- action

    # report payoffs to players, update their aspirations, update action propensities
    object <- updatePlayers(object)
    return(object)
```

```
})

setMethod("getOutcomeNumber", "normalGame", function(object) {
    actions <- object@lastActions
    cells <- array(data=1:object@nOutcomes, dim=dim(object@payoffs[[1]]))
    outcomeNumber <- cells[matrix(data=actions, nrow=1, ncol=length(actions))]
})

setMethod("repeatPlay","normalGame", function(object, reps) {
    #cat("Playing Round: ")
    outcomeVec <- vector(mode="integer", length=reps)
    for (i in 1:reps) {
        #cat(paste(i,"...",sep=""))
        object <- play(object)
        outcomeVec[i] <- as.integer(getOutcomeNumber(object))
    }
    return(outcomeVec)
})

# CLASS publicGoodGame - simplified interface for multiplayer games where only number of
# players cooperating matters
setClass("publicGoodGame", representation(payoffs="numeric", cooperators="integer"),
  contains="Game")

setMethod("initialize", "publicGoodGame", function(.Object, payoffs, ...) {
    .Object <- callNextMethod(.Object, ...)
    .Object@payoffs <- payoffs
    .Object@cooperators <- as.integer(1:length(.Object@players))
    .Object
})

setMethod("play", "publicGoodGame", function(object) {
    # get all actions from players
    action <- vector(mode="integer", length=length(object@players))
    for (i in 1:length(object@players)) {
        action[i] <- getAction(object@players[[i]])
        object@players[[i]]@lastAction <- as.integer(action[i])
    }
    # compute payoffs: everybody gets the payoff from the cooperators, minus a cost if they
    # personally cooperated
    payoff <- vector(mode="numeric", length=length(object@players))
    for (i in 1:length(payoff)) {
        payoff[i] <- object@payoffs[1] * sum(action) + object@payoffs[2] *
          as.integer(action[i] == 1)
    }
    object@lastPayoffs <- payoff
    object@lastActions <- action
    object@cooperators <- as.integer(action == 1)

    # report payoffs to players, update their aspirations, update action propensities
    object <- updatePlayers(object)
```

```
    return(object)
})

setMethod("repeatPlay","publicGoodGame", function(object, reps) {
    #cat("Playing Round: ")
    cooperatorMatrix <- matrix(data=0, ncol=length(object@players), nrow=reps)
    for (i in 1:reps) {
        #cat(paste(i,"...",sep=""))
        object <- play(object)
        cooperatorMatrix[i,] <- object@cooperators
    }
    cooperatorVec <- apply(cooperatorMatrix, MARGIN=2, FUN=mean)
    return(cooperatorVec)
})

setMethod("repeatPlayCorr", "publicGoodGame", function(object, reps) {
    corrMatrix <- diag(x=reps, ncol=length(object@players), nrow=length(object@players))
    for (i in 1:reps) {
        object <- play(object)
        for (j in 1:(length(object@players)-1)) {
            for (k in (j+1):length(object@players)) {
                corrMatrix[j,k] <- corrMatrix[j,k] + as.integer(object@lastActions[j] ==
                  object@lastActions[k]) - as.integer(object@lastActions[j] !=
                  object@lastActions[k])
            }
        }
    }
    corrMatrix <- corrMatrix / reps
    return(corrMatrix)
})

# CLASS agent

setClass("Agent", representation(id="integer", nActions="integer", propensity="numeric",
  lastAction="integer", lastPayoff="numeric", trembleProb="numeric"))

setMethod("initialize", "Agent", function(.Object, ...) {
    .Object <- callNextMethod(.Object, ...)
    if(is.null(.Object@trembleProb)) .Object@trembleProb <- 0
    .Object@propensity <- rep(1/.Object@nActions, times=.Object@nActions) # neutral start
    .Object@lastAction <- as.integer(1)
    .Object@lastPayoff <- 0
    .Object
})

setMethod("getRelevantPayoffs", "Agent", function(object, payoffs) {
    object@lastPayoff <- payoffs[object@id]
    return(object)
})

setMethod("getAction", "Agent", function(object) {
```

```
    x <- runif(1)
    action <- as.integer(1)
    for(j in 1:(length(object@propensity)-1)) {
        if (x >= sum(object@propensity[1:j])) action <- action + 1
    }
    as.integer(action)
})

#setMethod("getAction", "Agent", function(object) {
#    x <- runif(1)
#    action <- as.integer(1)
#    for(j in 1:(length(object@propensity)-1)) {
#        if (x >= sum(object@propensity[1:j])) action <- action + 1
#    }
#    if (object@trembleProb > 0){
#        tremble <- runif(1)
#        if(tremble < object@trembleProb) {
#            action <- sample.int(object@nActions, size=1)
#        }
#    }
#    as.integer(action)
#})

# CLASS abarAgent - implements general form of aspiration based adjustment rules
setClass("abarAgent", representation(aspiration="numeric", prevAspWgt="numeric",
  propUpdateFunction="function",extraParms="list"), contains="Agent")
setMethod("initialize", "abarAgent", function(.Object, ...) {
    .Object <- callNextMethod(.Object, ...)
})
setValidity("abarAgent", function(object) {
    object@prevAspWgt <= 1 && object@prevAspWgt >= 0
})

setMethod("setAspiration", "abarAgent", function(object) {
    # agent only cares about own payoff
    # inertia for aspirations
    rnd <- runif(1)
    if (rnd > object@extraParms$aspInertiaProb) {
        object@aspiration <- object@prevAspWgt * object@aspiration +
          (1 - object@prevAspWgt) * object@lastPayoff
    }
    return(object)
})

setMethod("setPropensity", "abarAgent", function(object) {
    # inertia for propensities
    x <- abs(object@lastPayoff - object@aspiration)
    propInertiaProb <- object@extraParms$propInertiaFct(x)
    rnd <- runif(1)
    if (rnd > propInertiaProb) {
        if(object@lastPayoff >= object@aspiration) {
```

```
            object@propensity <- object@propUpdateFunction(object, success=TRUE)
        } else {
            object@propensity <- object@propUpdateFunction(object, success=FALSE)
        }
    }
    rnd <- runif(1)
    if (rnd < object@trembleProb){
        object@propensity <- tremble(object@propensity)
    }
    return(object)
})

# Tremble function - modifies propensity vector when random draw indicates tremble
# in this implementation, draw from mvt normal density (of dim nActions - 1)
tremble <- function(propensity) {
    sigma = 0.01
    nActions <- length(propensity)
    basis <- do.call("cbind", getBasis(nActions))
    rnddraw <- matrix(data=rnorm((nActions-1), mean=0, sd=sigma), nrow=(nActions-1), ncol=1)
    newpropensity <- as.numeric(propensity + basis %*% rnddraw)
    if(!all(pmin(pmax(newpropensity, 1e-09),1-1e-09) == newpropensity)) {
        # ensure that we are still on the strictly positive unit simplex
        # accomplish this by just tossing and redrawing, inefficient but shouldn't be too
        # bad most of the time
        newpropensity <- tremble(propensity)
    }
    return(newpropensity)
}

getBasis <- function(n) {
    #returns remaining n-1 orthonormal basis vectors, taking the first to be (1,1,1,...,1)
    eye <- diag(n)
    basis <- vector(mode="list", n)
    basis[[1]] <- rep(1, n)
    for (i in 2:n) {
        vI <- as.vector(eye[i,])
        basis[[i]] <- vI
        for (j in 1:(i-1)) {
            basis[[i]] <- basis[[i]] - (sum(vI * basis[[j]]) / sum(basis[[j]]^2)) * basis[[j]]
        }
        norm <- sqrt(sum(basis[[i]]^2))
        basis[[i]] <- basis[[i]] / norm
    }
    basis[[1]] <- NULL
    return(basis)
}

# Propensity update functions - instantiate the abarAgent class with one of these to use
# various ABAR heuristics
# note that different functions will expect different stuff to be in the "extraParms" slot
# of the abarAgent object - be sure to fill these appropriately!
```

```
# FUNCTION propUpdateSatisfice - implements satisficing heuristic
propUpdateSatisfice <- function(object, success) {
    lastAction <- object@lastAction
    prop <- object@propensity
    nActions <- length(prop)
    if(success) {
        if (lastAction > 1) prop[1:(lastAction - 1)] <- 0
        prop[lastAction] <- 1
        if (lastAction < nActions) prop[(lastAction + 1):nActions] <- 0
    } else {
        x <- abs(object@lastPayoff - object@aspiration)
        swProb <- object@extraParms$switchFunction(x)
        if (lastAction > 1) prop[1:(lastAction - 1)] <- (swProb / (nActions-1))
        prop[lastAction] <- 1 - swProb
        if (lastAction < nActions) prop[(lastAction + 1):nActions] <- (swProb / (nActions-1))
    }
    return(prop)
}

# FUNCTION propUpdateBM - implements bush-mosteller heuristic
propUpdateBM <- function(object, success) {
    lastAction <- object@lastAction
    prop <- object@propensity
    x <- abs(object@lastPayoff - object@aspiration)
    alpha <- object@extraParms$alphaFunction(x)
    nActions <- length(prop)
    if(success) {
        if (lastAction > 1) prop[1:(lastAction - 1)] <- prop[1:(lastAction - 1)] -
          (alpha * (1 - prop[lastAction]) / (nActions - 1))
        if (lastAction < nActions) prop[(lastAction + 1):nActions] <-
          prop[(lastAction + 1):nActions] - (alpha * (1 - prop[lastAction]) / (nActions - 1))
        prop[lastAction] <- prop[lastAction] + alpha * (1 - prop[lastAction])
    } else {
        if (lastAction > 1) prop[1:(lastAction - 1)] <- prop[1:(lastAction - 1)] +
          (alpha * prop[lastAction] / (nActions-1))
        if (lastAction < nActions) prop[(lastAction + 1):nActions] <-
          prop[(lastAction + 1):nActions] + (alpha * prop[lastAction] / (nActions-1))
        prop[lastAction] <- prop[lastAction] * (1 - alpha)
    }
    return(prop)
}

# CLASS socialAgent - inherits from abarAgent, implements reference-group-based
# aspiration adjustment

setClass("socialAgent", representation(refGrp="integer", refGrpLastPayoff="numeric",
  refWgt="numeric"), contains="abarAgent")
setMethod("initialize", "socialAgent", function(.Object, ...) {
    .Object <- callNextMethod(.Object, ...)
})
```

```
setValidity("socialAgent", function(object) {
    object@refWgt + object@prevAspWgt <= 1
})

setMethod("setAspiration", "socialAgent", function(object) {
    # consider the payoffs of the reference group also
    # inertia for aspirations
    rnd <- runif(1)
    if (rnd > object@extraParms$aspInertiaProb) {
         object@aspiration <- object@prevAspWgt * object@aspiration + object@refWgt *
           object@refGrpLastPayoff + (1 - object@prevAspWgt - object@refWgt) * object@lastPayoff
    }
    return(object)
})

setMethod("getRelevantPayoffs", "socialAgent", function(object, payoffs) {
    object@lastPayoff <- payoffs[object@id]
    object@refGrpLastPayoff <- mean(payoffs[object@refGrp])
    return(object)
})
```

2.4 File 'repeatGamePlot.R'

```
rgPlot <- function(outcomeMatrix, dims, file="X") {
    if (file == "X") {X11()}
    else png(filename=file)
    nOutcomes <- prod(dims)
    m <- array(1:nOutcomes, dim=dims)
    layout(m)
    layout.show(nOutcomes)
    for (j in 1:prod(dims)) {
        thisOutcome <- matrix(data=as.integer(outcomeMatrix==j),
          nrow=dim(outcomeMatrix)[1], ncol=dim(outcomeMatrix)[2])
        thisOutcomeMean <- apply(thisOutcome, MARGIN=1,FUN=mean)
        hist(thisOutcomeMean,10, col="blue", main=paste("Outcome: ", j, sep=""),
          xlab=paste("Mean Proportion = ", mean(thisOutcomeMean), sep=""), ylab="",xlim=c(0, 1))
    }
    if (file == "X") {}
    else dev.off()
}

rgPlotWithPayoffs <- function(outcomeMatrix, dims, payoffs, filename="X") {
    if (filename == "X") {X11()}
    else pdf(file=filename)
    nOutcomes <- prod(dims)
    m <- array(1:nOutcomes, dim=dims)
    layout(m)
    #layout.show(nOutcomes)
    for (j in 1:prod(dims)) {
        thisOutcome <- matrix(data=as.integer(outcomeMatrix==j), nrow=dim(outcomeMatrix)[1],
          ncol=dim(outcomeMatrix)[2])
```

```
        thisOutcomeMean <- apply(thisOutcome, MARGIN=1,FUN=mean)
        histTitle1 <- switch(j, "(C,C)", "(D,C)", "(C,D)", "(D,D)")
        histTitle2 <- paste(payoffs[j,], collapse=",")
        histTitle <- paste(histTitle1, ": (", histTitle2, ")", sep="")
        hist(thisOutcomeMean,10, col="blue", main=histTitle, xlab=paste("Mean Proportion = ",
          mean(thisOutcomeMean), sep=""), ylab="",xlim=c(0, 1))
    }
    if (filename == "X") {}
    else dev.off()
}

rgPlotWithStates <- function(outcomeMatrix, dims, filename="X") {
    if (filename == "X") {X11()}
    else pdf(file=filename)
    nOutcomes <- prod(dims)
    m <- array(1:nOutcomes, dim=dims)
    layout(m)
    #layout.show(nOutcomes)
    for (j in 1:prod(dims)) {
        thisOutcome <- matrix(data=as.integer(outcomeMatrix==j), nrow=dim(outcomeMatrix)[1],
          ncol=dim(outcomeMatrix)[2])
        thisOutcomeMean <- apply(thisOutcome, MARGIN=1,FUN=mean)
        histTitle <- switch(j, "(C,C)", "(D,C)", "(C,D)", "(D,D)")
        hist(thisOutcomeMean,(0:20)/20, col="blue", main=histTitle,
          xlab=paste("Mean Proportion = ", mean(thisOutcomeMean), sep=""), ylab="",xlim=c(0, 1))
    }
    if (filename == "X") {}
    else dev.off()
}

rgPlotPublicGoods <- function(coopMatrix, dims, file="X") {
    if (file == "X") {X11()}
    else png(filename=file)
    nPlayers <- prod(dims)
    m <- array(1:nPlayers, dim=dims)
    layout(m)
    layout.show(nPlayers)
    for (j in 1:nPlayers) {
        histTitle <- paste("Player ", j, sep="")
        hist(coopMatrix[,j],10, col="blue", main=histTitle, xlab=paste("Mean Proportion = ",
          mean(coopMatrix[,j]), sep=""), ylab="",xlim=c(0, 1))
    }
    if (file == "X") {}
    else dev.off()
}
```

3 References

Feller, William. 1950. *An Introduction to Probability Theory and Its Applications, Vol. 1.* New York: John Wiley and Sons.