COMPUTATIONAL ECONOMICS

Revised Edition

Agent-Based Model with Trade in MATLAB

David A. Kendrick
Ruben P. Mercado
Hans M. Amman

1 Introduction

In Chapter 17, called Agent Based Model in MATLAB, we introduced a simple version of the famous Sugarscaspe model developed by Epstein and Axtell. In that version we had a group of agents who lived on a grid that contained in its cells different quantities of a resource called sugar. Each agent had a metabolism that determined its minimum sugar needs for survival, and a range of vision to see the sugar content of the cells in its neighborhood. Its neighborhood was defined by cells in the north, south, east, and west directions up to the limit determined by its range of vision. In each period, each agent searched within its neighborhood for the free cell with the highest quantity of sugar, moved to it, added the quantity of sugar found to its sugar stock, and consumed what was necessary to satisfy its metabolic need. If the quantity of sugar was insufficient, it died. The quantities of sugar in the Sugarscape regenerated autonomously after each period. As extensions of this simple version of the Sugarscape model, in this Chapter we will cover two more complex versions, also following the developments of Epstein and Axtell.

First, we present a model with trade. To do this, we introduce a new resource in the Sugarscape, called spice. Thus, each cell in the Sugarscape will contain some quantity of sugar and some of spice. Each agent will be able to exchange resources with other agents located within its neighborhood. The exchange will take place through a bilateral sequential bargaining process, as a result of which the prices and quantities exchanged will be determined.

Second, we present a model that makes the model with trade more complex. In this new model, a process of cultural influence will take place between agents affecting their preferences in each period of the simulation. Thus, agents preferences will change endogenously, and this will affect the prices and quantities exchanged in each period of the simulation.[1]

2 The Model with Trade

In this model each agent has minimum required metabolisms for sugar and for spice, and a range of vision. It can move, in each period, only within its neighborhood. We assume that each agent values its holdings of sugar and spice according to a Cobb-Douglass welfare function W of the form:

$$W\left(w_{su},\, w_{sp}\right) = w_{su}^{m_{su}/m_T} \quad w_{sp}^{m_{sp}/m_T}$$

[1] See Esptein, J. and Axtell, R.: 1996, Growing Artificial Societies: Social Science from the bottom up, The MIT Press, Cambridge, Massachusetts, USA. The models are in Chapter IV.

where w_{su} is the quantity of sugar in the agents stock, w_{sp} is its stock of spice, m_{su} is its metabolic need for sugar and m_{sp} its metabolic need for spice, and where $m_T = m_{su} + m_{sp}$. The arguments of the welfare function are accumulated stocks, so welfare is state dependent. This is a useful way of representing a situation in which agents do not consume all their stocks within a period. The agent moves within its neighborhood looking for the free cell which, given the quantities of sugar and spice it contains, and given the agent's stocks of sugar and spice, yields the highest welfare.

Once the agent located and moved to the best welfare improving cell, it searches within its neighborhood for other agents to trade resources with. The exchange takes place through a bilateral sequential bargaining process, and which is carried out as long as each exchange increases both agents welfare. To characterize this process, we need to define how an agent's valuation of each resource is determined, the direction of the exchange between any two agents, and the determination of the exchange price and the traded quantities.

Following standard microeconomic theory, an agent's valuation of each resource will result from the marginal rate of substitution (MRS) of one good for another. Thus, given the welfare function W, the MRS of spice per 1 unit of sugar is:

$$MRS = \frac{\frac{\partial W(w_{su}, w_{sp})}{\partial w_{su}}}{\frac{\partial W(w_{su}, w_{sp})}{\partial w_{sp}}} = \frac{\frac{w_{sp}}{m_{sp}}}{\frac{w_{su}}{m_{su}}}$$

The direction of the exchange between each pair of agents will follow this rule: spice goes from the agent with a higher MRS to the agent with a lower MRS, while sugar moves in the opposite way. There are many possible rules for the bargaining process to determine the exchange price. Following Epstein and Axtell, given any two agents a and b, the exchange price p will be determined as the geometric mean of their MRSs:

$$p = \sqrt[2]{MRS_a \quad MRS_b}$$

This rule is useful to moderate the effect of two agents having MRS far away from each other. Also following Epstein and Axtell, the quantities exchanged are determined by the following rule:

if p $>$ 1 then p units of spice are exchanged for 1 of sugar

if p $<$ 1 then 1/p units of sugar are exchanged for 1 unit of spice

The process of determination of prices and quantities repeats itself if it improves both agents' welfare, and if their MRSs do not crossover[2] or until they equalize.

[2] Trade takes place in discrete quantities. Thus, repeated trade may never lead to the equal-

In summary, the trade process between two neighboring agents begins if they have different MRSs. In such a case, they try to make an exchange that makes both of them better off. To do so, they engage in a bargaining process, determining the price and quantities exchanged. Trade occurs if it does not cause the agents' MRSs to cross over. After trading, the agents update their MRSs, and bargaining starts anew. The sequential bargaining process ends when agents' MRS equalize or cross over.

This trade process contrasts against the way in which prices and quantities are determined in standard general equilibrium theory: since in this theory agents are supposed to be fully rational, and fully informed, they trade, in one shot, at the equilibrium prices and in the quantities that equalize their MRSs. While in this model the implicit assumption is that agents are boundedly rational, thus they move in small incremental steps towards the equalization of their MRSs, or towards a point close to it. Thus, prices and quantities display a transitional dynamic of disequilibrium. Also in general equilibrium theory an agent can engage in trade with any other agent, while in this model it can only trade with agents within its neighborhood.

3 The Model with Trade in MATLAB

The structure of the program of this more complex version of the Sugarscape model is similar to the one of the simpler model presented in Chapter 17, with some modifications in the MATLAB functions and the addition of two new functions: seeT and neighborT. The MATLAB representation of the model consists of a main program named sugarscapeT, and a number of functions whose details we present later. The description of the code for the main program follows.

```
%Initialize model parameters
nruns = 50;
size = 50; \%even number
metsugar = 5;
metspice = 5;
vision = 5; \%set always smaller than size
maxsugar = 30;
maxspice = 30;
initwup = 50;
```

ization of agents' MRSs. This may cause an infinite loop in which two trading agents alternate the roles of being buyers and sellers of the same resource during the sequential bargaining process. To avoid this, the condition that MRSs do not cross over one another is imposed.

```
initwlow = 25;

 %Initialize sugarscape and display
[s, ssugar, sspice] = initsugarscapeT(size, maxsugar, maxspice);

%Initialize agents population
a\_str = initagentsT(size, vision, metsugar, metspice, initwup, initwlow);
```

The program begins by initializing the number of runs, the size of the grid of the Sugarscape, the maximum metabolism for sugar and spice and for the vision range of the agents, and the maximum levels of sugar and spice in the Sugarscape. It also initializes the upper and lower levels of the initial endowment of wealth of the population of agents. Later it calls the initsugarscapeT function to generate the initial distribution of sugar and spice in the Sugarscape, retuning the matrices ssugar and sspice -which respectively contain the distribution of sugar and spice- and the s matrix containing a kind of index of the joint distribution of both resources for display purposes. Then the main program calls the initagentsT function, returning the MATLAB data structure a_str containing the initial random distribution of the population of agents and their attributes. Then the main loop of the program begins.

```
%Main loop (runs)
for runs = 1:nruns;
 iprv = 0;  %Set initial index value for price vector prv
 %Display agents locations
 dispagentlocT(a\_str, size, nruns, runs);
 %Select agents in a random order
 for i = randperm(size);
  for j= randperm(size);
         {\dots}{\dots}
  end  \% for j loop
 end  \% for i loop
  %Compute average price for one run and store in vector to plot later

 avprv = geomean(prv);
 avprruns(runs) = avprv;
 axesx(runs) = runs;
end  \% for runs
```

The main loop runs from 1 to nruns, the number of runs. The index variable iprv will be used to move within the price vector generated later by the

program, and it is initialized to zero. The dispagentlocT function is called to display the location of the population of agents on the grid of the Sugarscape at each run. The statements:

```
for i = randperm(size);
for j= randperm(size);
```

begin two loops that select each agent of the population in a random way using the MATLAB function randperm to randomly select each location index. Finally, at the end of each run, the average price for the run -avprv- is computed as the geometric mean of the prices stored in the price vector prv, and stored in the avprruns vector to later be plotted against an axis containing the runs numbers. Following Epstein and Axtell, we use the prices geometric mean to mitigate the influence of possible outliers.

Next we will see what happens inside the two loops that select each agent of the population in a random way. The corresponding code follows.

```
 if (a\_str(i,j).active == 1) \%is there an angent on this location?

\%Agent explores sugarscape in random directions and selects best location
tempssugar = ssugar(i,j);
tempsspice = sspice(i,j);
tempi = i;
tempj = j;
for k = a\_str(i,j).vision : -1 : 1;
[tempssugar, tempsspice, tempi, tempj, move] = ...
seeW(i,j,k,a\_str,ssugar,sspice,size,tempssugar,tempsspice,tempi,tempj);
end
\%Agent moves to best location, updates sugar stock and eats sugar
a\_str = moveagentT(a\_str, i, j, ...
tempssugar, tempsspice, tempi, tempj, move);
\%Agent explores sugarscape in random directions and trades with neighbors
for k = a\_str(i,j).vision : -1 : 1;
[avpav, a\_str] = seeT(i,j,k,a\_str,size);
\%Store average price of agent trades with its neighbors
if avpav $\mathrm{>}$ 0
iprv = iprv + 1;
prv(iprv) = avpav;
end
end
end  \% for if active agent
```

As we mentioned above, and as we will see in detail when presenting the initagentsT function, the attributes of each agent are stored in a MATLAB data structure named a_str. One of the attributes is if the agent is active (represented by the number 1) or not (represented by 0). Thus, the statement

```
if (a_str(i,j).active == 1)
```

is used to check if there is an active agent in the (i,j) location of the grid. If so, the program continues with the settings of three temporary variables. The variables tempsugar and tempspice contain the level of sugar and spice in the agent's current location, while tempi and tempj contain the location's coordinates. Then follows a loop that goes from the agent's maximum level of vision to 1, in decrements of one unit. At each pass of this loop, the function seeW is called. This function sees around the agent's neighborhood in the north, south, east and west directions, from the farthest position the agent can see to its immediate surroundings, and detects the cell whose resources yield the highest welfare for the agent. It returns in the variables tempsugar and tempspice - the levels of sugar and spice that yield the highest welfare for the agent- and the corresponding cell location coordinates in the variables tempi and tempj respectively. It also returns the move variable, that signals if the agent should move or not to a new location. Finally, once the loop is completed, the function moveagentT is called to move the agent to the new location and to update its stocks of wealth, returning the updated data structure a_str.

Then begins a loop similar to the one just seen. At each pass of this loop, the function seeT is called, to see around the agent's neighborhood to locate other agents and to trade with them, returning avpav, the average price of the agent trades with its neighbors, and the updated data structure a_str. At each pass of the loop, after increasing in one unit the price vector index iprv, if avpav is greater than zero its value is stored in the price vector prv.

Finally, once the main loop of the program is over, a figure is created to plot the evolution of the average price along of the runs.

```
figure
scatter(axesx,avprruns);
axis square;
```

3.1 InitsugarscapeT and InitsugarscapeTb

The initsugarscapeT and initsugarscapeTb functions initialize the levels of sugar and spice in the Sugarscape, creating a four-peak Sugarscape, with peaks of

sugar in the south west and the north east, and peaks of spice in the north west and the south east. The procedure is similar to the one used in Chapter 17, modified to include the spice resource. The first statement of the initsugarscapeT function is:

```
function [s, ssugar, sspice] = initsugarscapeT(size, maxsugar, maxspice)
```

Then, the function creates a matrix named s1 containing a Sugarscape with a distribution of sugar descending from a south west peak with the following code.

```
x = -ceil(0.75*size) : size-ceil(0.75*size)-1;
y = -ceil(0.25*size) : size-ceil(0.25*size)-1;
maxresource = maxsugar;
stemp = initsugarscapeTb(size, maxresource, x, y);
s1 = stemp;
```

The x and y vectors correspond to the "shifted" row and column indexes of the Sugarscape grid, where the elements with a value equal to 0 will define the (x,y) location of the peak. Thus, in this case, the peak will be located on a row that is 75% down and 25% to the right of the grid, that is, on the south west region. Details on the construction and interpretation of the x and y vectors are in Chapter 17. After setting the coordinates of the peak, the maximum level of sugar maxsugar -defined in the initial conditions of the main program- is assigned to the maxresource variable. And then the initsugarscapeTb function is called.

```
function stemp = initsugarscapeTb(size, maxresource, x, y)
for i = 1:size;
    for j = 1:size;
        if (x(i) == 0 \&\& y(j) == 0)
            stemp(i,j) = maxresource;
        else
            stemp(i,j) = maxresource / (abs(x(i)) + abs(y(j)));
        end
    end
end
```

As explained in detail in Chapter 17, the code in this function generates a Sugarscape with a decreasing resource distribution from the peak down. In this case, it creates a matrix named stemp, containing a sugarscape with a peak in

the coordinates corresponding to the (x,y) location with value equal to 0. The remaining cells of the grid contain quantities of the resource given by the ratio between the maximum quantity and the sum of the absolute value of the elements of the x and y vectors. Thus, as we get farther away from the peak, the resulting quantity of the resource becomes smaller. Finally, in the initsugarscapeT function, the stemp matrix returned by the initsugarscapeTb function is assigned to the s1 matrix.

The code of the initsugarscapeT function continues with the creation of the s2 matrix, which contains a sugar distribution with a north east peak. This just requires changing the peak coordinates and assigning the stemp matrix to the s2 matrix.

```
x = -ceil(0.25*size) : size-ceil(0.25*size)-1;
y = -ceil(0.75*size) : size-ceil(0.75*size)-1;
maxresource = maxsugar;
stemp = initsugarscapeTb(size, maxresource, x, y);
s2 = stemp;
```

To generate a two-peak sugar Sugarscape, which will be contained in the ssugar matrix, matrices s1 and s2 are added.

```
ssugar = s1 + s2;
```

To generate a matrix named s3 with a distribution of spice with one south east peak, we proceed as we did with the s1 and s2 matrices, just changing the coordinates of the peak to:

```
x = -ceil(0.75*size) : size-ceil(0.75*size)-1;
y = -ceil(0.75*size) : size-ceil(0.75*size)-1;
```

and assigning the maxspice variable to maxresource, before calling the initsugarscapeTb function. To generate a matrix named s4 with a distribution of spice with a north west peak, we do as with the s3 matrix, just changing the coordinates of the peak to:

```
x = -ceil(0.25*size) : size-ceil(0.25*size)-1;
y = -ceil(0.25*size) : size-ceil(0.25*size)-1;
```

Finally, to generate a two-peak spice Sugarscape, which will be contained in the sspice matrix, we just add the s3 and s4 matrices.

```
sspice = s3 + s4;
```

Now, just for the sake of generating a figure representing the four-peak sugar and spice Sugarscape, where the yellow color represents sugar and the blue color spice; where the more intense the color the higher the amount of the resource; and where the green color represents different mixtures of sugar and spice in each cell, we proceed as follows. First, we create the s matrix, as the difference between the ssugar and the sspice matrices. Thus, where sugar predominates, the difference will be positive, while where spice predominates it will be negative.

```
s = ssugar - sspice;
```

Finally, we create a square figure representing the Sugarscape, and display the s matrix using the imagesc MATLAB function, choosing the palette of colors named parula from the image colormap. In this way, cells with relatively high positive numbers -i.e. with more sugar than spice- will be shown as scales of yellow, those with relatively high negative ones -i.e. with more spice than sugar- will be shown as scales of blue, and those where the absolute value of the numbers in the cells are relatively small will be shown in scales of green.

```
figure;
colormap(parula);
imagesc(s);
axis square;
```


3.2 InitagentsT

The initagentsT function initializes the population of agents. To do so, it creates a data structure named a_str containing all agents' attributes: if it is an active agent, its metabolism, its vision, and its initial wealth of sugar and spice. The rand MATLAB function generates random numbers between 0 and 1 with a uniform distribution. The statement

```
if (rand <  0.3)
```

tells us that approximately one third of the grid of the Sugarscape will be occupied by active agents. For each of them, all of its attributes are generated in a random way, taking values between their minimum and maximum levels as defined in the initial conditions of the main program.

```
%InitagentsT
%Initialize agents population
```

```
function a_str = initagentsT(size, vision, metsugar, metspice, initwup, initwlow)
for i = 1:size;
    for j = 1:size;
        if (rand < 0.3)
            a_str(i,j).active = 1; % put an agent on this location
            a_str(i,j).metsugar = ceil(rand * metsugar);
            a_str(i,j).metspice = ceil(rand * metspice);
            a_str(i,j).vision = ceil(rand * vision);
            a_str(i,j).wealthsugar = rand * (initwup-initwlow) + initwlow;
            a_str(i,j).wealthspice = rand * (initwup-initwlow) + initwlow;
        else
            a_str(i,j).active = 0; % keep this location empty
            a_str(i,j).metsugar = 0;
            a_str(i,j).metspice = 0;
            a_str(i,j).vision = 0;
            a_str(i,j).wealthsugar = 0;
            a_str(i,j).wealthspice = 0;

        end
    end
end
```


3.3 DispagentlocT

The dispagentlocT displays the location on the Sugarscape grid of each active agent at each run of the simulation.

```
% DispagentlocT
% Transform field "agent" from data structure into matrix and display agents locati
function dispagentlocT(a_str, size, nruns, runs)
a = zeros(size); av = zeros(size); amsugar = zeros(size); amspice = zeros(size);

for i = 1:size;
    for j = 1:size;
        if (a_str(i,j).active == 1)
         a(i,j) = a_str(i,j).active;
            av(i,j) = a_str(i,j).vision;
            amsugar(i,j) = a_str(i,j).metsugar;
            amspice(i,j) = a_str(i,j).metspice;
            awsu(i,j) = a_str(i,j).wealthsugar;
            awsp(i,j) = a_str(i,j).wealthspice;
```

```
        end
    end
end

if (runs==1 || runs==5 || runs==10 || runs==50 ||runs==100 || runs==nruns)
    figure(runs)
    spy(a)
    title(['run:  ', num2str(runs)])
    axis square;
end
avgvision = sum(sum(av))/sum(sum(a));
avgmsugar = sum(sum(amsugar))/sum(sum(a));
avgmspice = sum(sum(amspice))/sum(sum(a));
```

The function transforms each field from the data structure a_str into several matrices, to perform different operations with each one. The a matrix contains the locations of all the active agents. The other matrices contain information on metabolism, vision range and wealth. The if sentence selects specific runs to generate a figure for each of them displaying the location of the agents on the grid of the Sugarscape. The spy(a) sentence displays the active agents -the nonzero elements of the a matrix- while the next two sentences define the title of the corresponding figure and specify that it is a square plot. Finally, the last three sentences compute the average vision, average sugar metabolism and average spice metabolism of the population of agents in each run. If we eliminate the semicolon at the end of each of these three sentences, their variables values will be display, for each run, in the MATLAB command window.

3.4 SeeW

The seeW function explores the neighborhood an agent can see according to its level of vision in four directions - north, south, east and west - each direction selected in a random order. The workings of this function are explained in detail in Chapter 17. The only difference here is in the variables that are received and returned by the function, and in the function that is called: neighborW, instead of the neighbor function as in Chapter 17.

```
% SeeW
% Explore neighborhood looking for best location to move

function [tempssugar, tempsspice, tempi, tempj, move] = ...
    seeW(i,j,k,a_str,ssugar,sspice,size,tempssugar,tempsspice,tempi,tempj);
```

```
south = [i+k  size  i+k-size  j  i+k  j];
north = [k-i  -1  i-k+size  j  i-k  j];
east  = [j+k  size  i  j+k-size  i  j+k];
west  = [k-j  -1  i  j-k+size  i  j-k];

c{1} = south;  c{2} = north;  c{3} = east;  c{4} = west;

for m = randperm(4);

 if (c{m}(1) > c{m}(2))
        u = c{m}(3);
        v = c{m}(4);
        [tempssugar, tempsspice, tempi, tempj, move] = ...
            neighborWK(u,v,a_str,ssugar,sspice,tempssugar,tempsspice,tempi,tempj,i
 else
        u = c{m}(5);
        v = c{m}(6);
        [tempssugar, tempsspice, tempi, tempj, move] = ...
            neighborWK(u,v,a_str,ssugar,sspice,tempssugar,tempsspice,tempi,tempj,i
    end
end
```


3.5 NeighborW

The neighborW function checks if the available quantities of sugar and spice in a location of the agent's neighborhood are welfare improving.

```
% NeighborW
% Check if available sugar and spice in location are welfare improving

function [tempssugar, tempsspice, tempi, tempj, move] = ...
    neighborW(u,v,a_str,ssugar,sspice,tempssugar,tempsspice,tempi,tempj,i,j)

% Initialize welfare variables at zero and move variable to not, just to have them
welfareij = 0;
welfareuv = 0;
move = 'not';

% Compute and compare welfare at current location and potential new free location
if (a_str(u,v).active == 0)
```

```
    % Rename some variables to shorten notation
    wsu = a_str(i,j).wealthsugar;
    wsp = a_str(i,j).wealthspice;
    msu = a_str(i,j).metsugar;
    msp = a_str(i,j).metspice;
    mtot = msp + msu;

    % Compute welfare
    welfareij = (wsu + ssugar(i,j))^(msu/mtot) * (wsp + sspice(i,j))^(msp/mtot) ;
    welfareuv = (wsu + ssugar(u,v))^(msu/mtot) * (wsp + sspice(u,v))^(msp/mtot) ;

    if (welfareuv >= welfareij)
        move = 'yes';
        tempssugar = ssugar(u,v);
        tempsspice = sspice(u,v);
        tempi = u;
        tempj = v;
    else
        move = 'not';
    end
end
```

The initial if statement checks if the (u,v) location is not occupied by an agent. Then, after renaming some variables to shorten notation, the function computes welfareij -the level of welfare in the current agent location (i,j)- and wlefareuv -the welfare level the agent would achieve if moving to the (u,v) location. Finally, if welfare at the (u,v) location is higher, the function sets the move variable to 'yes' and stores the (u,v) coordinates and the corresponding levels of sugar and spice in temporary variables, that will be used to move the agent to the new and better location.

3.6 MoveagentT

The moveagentT function moves the agent to the best location found within its neighborhood and updates its levels of wealth.

```
function a_str = moveagentT(a_str, i, j, tempssugar, ...
   tempsspice, tempi, tempj, move)
```

If the move variable is equal to 'yes', the a_str corresponding to the agent is associated to the u and v coordinates of the new location, which are contained

in the tempi and tempj variables respectively. Then, the old location is set as unoccupied, setting all the a_str(i,j) values to zero. The agent's wealth levels are updated, adding to its previous stocks of wealth the quantities of sugar and spice found in the new location, and subtracting the quantities consumed according to its metabolic needs. If the resulting levels of sugar or spice are lower than zero, the agent dies, and its location is set to unoccupied.

```
if move == 'yes'

    a_str(tempi,tempj) = a_str(i,j);

    % Set old location to unoccupied
    a_str(i,j).active = 0;
    a_str(i,j).vision = 0;
    a_str(i,j).metsugar = 0;
    a_str(i,j).metspice = 0;
    a_str(i,j).wealthsugar = 0;
    a_str(i,j).wealthspice = 0;

    % Update wealth at new location
    a_str(tempi,tempj).wealthsugar = ...
        a_str(tempi,tempj).wealthsugar + tempssugar - a_str(tempi,tempj).metsugar;
    a_str(tempi,tempj).wealthspice = ...
        a_str(tempi,tempj).wealthspice + tempsspice - a_str(tempi,tempj).metspice;

    % If wealth is less than zero set location to unoccupied
    if (a_str(tempi,tempj).wealthsugar <= 0) || (a_str(tempi,tempj).wealthspice <=
        a_str(tempi,tempj).active = 0;
        a_str(tempi,tempj).vision = 0;
        a_str(tempi,tempj).metsugar = 0;
        a_str(tempi,tempj).metspice = 0;
        a_str(tempi,tempj).wealthsugar = 0;
        a_str(tempi,tempj).wealthspice = 0;
    end
```

If the move variable is not equal to 'yes', the agent stays in its current location and, as above, its wealth levels are updated, and if the resulting levels of sugar or spice are lower than zero its location is set to unoccupied.

```
else
```

```
    % Agent stays in position and updates wealth
    a_str(i,j).wealthsugar = a_str(i,j).wealthsugar + tempssugar - a_str(i,j).metsu
    a_str(i,j).wealthspice = a_str(i,j).wealthspice + tempsspice - a_str(i,j).metsp

    % If wealth is less than zero set location to unoccupied
    if (a_str(i,j).wealthsugar <= 0) || (a_str(i,j).wealthspice <= 0)
        a_str(i,j).active = 0;
        a_str(i,j).vision = 0;
        a_str(tempi,tempj).metsugar = 0;
        a_str(tempi,tempj).metspice = 0;
        a_str(tempi,tempj).wealthsugar = 0;
        a_str(tempi,tempj).wealthspice = 0;
    end
end
```

3.7 SeeT

The seeT function is essentially the same as the seeW function presented above, with some additions and minor modifications. It explores the neighborhood an agent can see according to its level of vision in four directions - north, south, east and west - each direction selected in a random order. However, at variance with the seeW function which explored the neighborhood searching for the most welfare improving cell, the seeT function explores the neighborhood searching for agents to trade with. At the beginning of the function, the first element of the price vector of the agent's trades named pav, that vector index named ipav, and the average price of the agent's trades avpav are all set to zero. Later, the neighborT function is called. Finally, if the aptv variable is greater than zero, the function stores the price of one trade between the agent and the neighbor it is trading with. To do so, it increases ipav by one unit, assigns the value of the aptv variable to the corresponding element in the pav vector, and computes and stores the geomentric mean of the pav vector into avpav. As we did in the main program, we use the geometric mean of prices to mitigate the influence of possible outliers.


```
 SeeT
% Explore neighborhood looking for agents to trade with

function [avpav, a_str] = seeT(i,j,k,a_str,size)

% Define pav price vector of agent's trades and initialize vector index
```

```
% and average price of agent's trades
pav(1)= 0;
ipav = 0;
avpav = 0;

south = [i+k  size  i+k-size  j  i+k  j];
north = [k-i  -1  i-k+size  j  i-k  j];
east  = [j+k  size  i  j+k-size  i  j+k];
west  = [k-j  -1  i  j-k+size  i  j-k];

c{1} = south;  c{2} = north;  c{3} = east;  c{4} = west;

for m = randperm(4)
 if (c{m}(1) > c{m}(2))
        u = c{m}(3);
        v = c{m}(4);
        [aptv, a_str] = neighborT(u,v,a_str,i,j);
 else
        u = c{m}(5);
        v = c{m}(6);
        [aptv, a_str] = neighborT(u,v,a_str,i,j);
    end

    % Store price of one trade between agent and one neighbor
    if aptv > 0
        ipav = ipav + 1;
        pav(ipav) = aptv;
        avpav = geomean(pav);
    end
end
```

3.8 NeighborT

The neighborT function computes the sequential bargaining process between an agent named a -located in cell (i,j)- and its neighbor named b -located in cell (u,v)- to obtain and return in the variable aptv the geometric mean of the successive bargaining prices agreed upon by the two trading agents.

```
function [aptv, a_str] = neighborT(u,v,a_str,i,j)
```

There are three main features of the workings of this function to take into account. First, as we explained in Section 2 above, trade between any two agents

will take place as long as their MRSs do not cross over; only if it improves the welfare of both agents; and only if does not imply that an agent will be left with a level of sugar or spice lower than the necessary to survive. Thus, every step of the sequential bargaining process will be taken only after checking that those conditions are satisfied. Second, what we just said implies that we will work with two types of variables associated with old and new values. The old variables will have associated the prefixes "old" or "o", and the new variables will have associated the prefixes "new" or "n". The old variables will refer to the current values of them for each agent participating in the exchange. The new variables will refer to the values that those variables will take in the case that trade is effectively carried on. We have to do this to ensure that the values resulting from the exchange process do not violate any of the conditions listed above. Only when those conditions are satisfied, the values of the new variables will substitute the ones of the old variables. Otherwise, the last trade step in the sequential bargaining process will not be effectively carried out, and the old variables' values will be left unchanged and the neighborT function will be terminated. Third, to signal if a condition is satisfied so that the neighborT function flow can continue, we will use some variables defined as "signals", whose values will be "yes" or "not".

The function begins by setting the aptv variable to 0 to have a value for the output function variable aptv in case there is no trade in this function and no price is set, which would lead the program to crash.


```
aptv = 0;
```

Then it sets the first element of price trade vector ptv to 0 in case the is no trade in this function and thus the ptv vector is left undefined.

```
ptv = 0;
```

It then checks that the (u,v) cell in the neighborhood of agent a is occupied by an agent. If so, the workings of the function begin. Otherwise, nothing is done since in that location there is no agent to trade with.

```
if (a_str(u,v).active == 1)
```

Then some variables containing information on wealth and metabolism of agents a and b are renamed to shorten notation.

```
% Rename some variables to shorten notation
 owsua = a_str(i,j).wealthsugar;
 owspa = a_str(i,j).wealthspice;
```

```
owsub = a_str(u,v).wealthsugar;
owspb = a_str(u,v).wealthspice;

msua = a_str(i,j).metsugar;
mspa = a_str(i,j).metspice;
msub = a_str(u,v).metsugar;
mspb = a_str(u,v).metspice;

mtota = mspa + msua;
mtotb = mspb + msub;
```

Then the code shown below follows.

```
% Check that old wealth levels are positive
if (owspa > 0 && owsua > 0 && owspb > 0 && owsub > 0)

    signalloop = 'yes'; % Assign initianl value to signalloop

    % Compute old welfare
    waold = owsua^(msua/mtota) * owspa^(mspa/mtota) ;
    wbold = owsub^(msub/mtotb) * owspb^(mspb/mtotb) ;

    % Compute old mrs
    omrsa = (owspa / mspa) / (owsua / msua);
    omrsb = (owspb / mspb) / (owsub / msub);

    % Set "direction" of mrs
        if omrsa > omrsb
            signalmrsold = '>';
        end
        if omrsa < omrsb
            signalmrsold = '<';
        end
        if omrsa == omrsb
            signalloop = 'not'; % Exit function
        end

    iptv = 0; % set ptv vector index to 0

% If agents old wealth levels are not positive exit loop
else signalloop = 'not'; % Exit function
```

The if statement checks if agent a old levels of wealth in the form of sugar and spice -owspa and owsua- and those of agent b -owspb and owsub- are positive. If so, the signalloop variable is turned to the 'yes' value, which means that the loop that later implements the sequential bargaining process will be carried on. Otherwise, it is turned to the 'not' value, which will lead to exit the neighborT function. Then the old levels of welfare and the old MRSs for each agent are computed. A new signal variable, named signalmrsold, is created and assigned the value $>$ if the old MRS of agent a -omrsa- is greater than that of agent b, or the value $<$ otherwise. In the case that both agents MRSs are equal there is no incentive to trade, so that the signalloop variable is set to 'not', which will lead to exit the neighborT function. The statement:

```
signaltrade = 'not';
```

creates and assigns the initial value 'not' to a variable named signaltrade. This variable value will be checked at the end of the execution of the neighborT function to determine if there was trade between agents a and b. Then the loop that implements the sequential bargaining process begins. Since this is a relatively long code, we show first the pseudocode.

```
while signalloop == 'yes'

   compute trade price and quantities
   check that the resulting agents' wealth levels would be positive
   (otherwise set signalloop == 'not' and exit the function)
   check that trade would improve both agents' welfare
   (otherwise set signalloop == 'not' and exit the function)
   check that the resulting MRSs would not cross over
   (otherwise set signalloop == 'not' and exit the function)
   carry on trade
   end
```


As we can see, the most important basic operations within the loop are the computation of prices and quantities to be traded, the checking of the conditions to carry on the trade, and the actual implementation of the trade only if all those conditions are met. Let's begin to present the code of the trades loop.

```
% Begin trades loop
while signalloop == 'yes'

    % Compute trade price
    p = (omrsa * omrsb)^(1/2);
```

```
% Compute trade quantities as a function of price
if p > 1
qsu = 1;   qsp = p;
end

if p < 1
qsu = 1/p; qsp = 1;
end
```

The loop is a while loop performed as long as the signalloop variable is in the 'yes' state. The price p is computed as the geometric mean of the product of the MRSs of agents a and b. Then qsu and qsp, the quantities of sugar and spice, are computed following the rule presented above in Section 2, which stated that:

```
if p > 1 then p units of spice are exchanged for 1 of sugar
if p < 1 then 1/p units of sugar are exchanged for 1 unit of spice
```

Also in Section 2 we presented the direction of trade rule that states that spice goes from the agent with a higher MRS to the agent with a lower MRS, while sugar moves in the opposite way. Thus, the next part of the code shown below sets the quantities of the changes in the wealth of the agents, according to the directions of trade, that is, depending on the relative magnitudes of their MRSs. If the signalmrsold variable is in the '>' state, that means that the MRS of agent a is greater than the one of agent b. Thus the quantity of sugar to be traded will go from agent b to agent a. To represent this, the value of the variable qsu is assigned to the intermediate variable qsua, and the value of minus qsu to the intermediate variable qsub. At the same time, the quantity of spice to be traded will go from agent a to agent b. To represent this, the value of the variable qsp is assigned to the intermediate variable qspb, and the value of minus qsp to the intermediate variable qsua. If the signalmrsold variable is in the '<' state, that means that the MRS of agent a is greater than the one of agent b, and the procedure is the reverse of the one just presented.

```
% Set quantities of changes in wealth according to directions of
        % trade
        if signalmrsold == '>'
            qsua = qsu; qspa = -qsp; qsub = -qsu; qspb = qsp;
        end
```

```
if signalmrsold == '<'
    qsua = -qsu; qspa = qsp; qsub = qsu; qspb = -qsp;
end
```

Next the agents' new wealth levels that would result if trade is actually made are computed, adding to their old wealth levels the corresponding traded quantities.

```
% Compute new wealth levels if trade is carried out
        nwsua = owsua + qsua;
        nwspa = owspa + qspa;
        nwsub = owsub + qsub;
        nwspb = owspb + qspb;
```

The next part of the code checks the conditions for trade and, if they are satisfied, implements the trade. It first checks if the new wealth levels that would result from the trade are all positive. If they are not, the signalloop variable is set to 'not', which means that we exit the trade loop. It then computes the new welfare levels that would result from the trade, and checks that they are higher than the old ones, that is, than the pre trade levels. If they are not, the signalloop variable is set to 'not' to exit the loop. It then computes the new MRSs that would result from the trade, checks the relative magnitudes of the agents' MRSs to establish the direction of trade, and finally by comparing the signalmrsold variable against the signalmrsnew variable determines if the direction of trade remains the same. If they are not equal, that means that the MRSs have crossed over, then trade should not be actually made, setting the signalloop variable to 'not' to exit the trade loop. If they are equal, the variable signaltrade is set to 'yes', to signal that trade will actually take place. Thus trade is made, updating the agents' old wealth levels and old MRSs with the new ones, increasing by one unit the variable iptv -the index of the price trade vector ptv-, storing the trade price p in ptv, and storing the geometric mean of the ptv in the average price trade vector variable named aptv. Finally, the last end sentence closes the while loop.

```
        % Check that new wealth levels are positive
        if (nwsua > 0 && nwspa > 0 && nwsub > 0 && nwspb > 0)

                % Compute new welfare if trade is carried out
                wanew = nwsua ^ (msua/mtota) * nwspa ^ (mspa/mtota);
                wbnew = nwsub ^ (msub/mtotb) * nwspb ^ (mspb/mtotb);
```

```
% Check that trade improves welfare of both agentes
if (wanew > waold) && (wbnew > wbold)

    % Compute new mrs if trade is carried out
    nmrsa = (nwspa / mspa) / (nwsua / msua);
    nmrsb = (nwspb / mspb) / (nwsub / msub);

    % Set "direction" of mrs
    if nmrsa > nmrsb signalmrsnew = '>';
    end
    if nmrsa < nmrsb signalmrsnew = '<';
    end
    if nmrsa == nmrsb signalloop = 'not'; % Exit loop
    end

    % Check that mrs do not cross over
    if signalmrsold == signalmrsnew

        signaltrade = 'yes';

        % Carry out trade and update wealth leveles
        owsua = nwsua;
        owspa = nwspa;
        owsub = nwsub;
        owspb = nwspb;

        % Update old mrs with new mrs
        omrsa = nmrsa;
        omrsb = nmrsb;

        % Store price in price vector
        iptv = iptv + 1;
        ptv(iptv) = p;
        aptv = geomean(ptv);

    % Agents mrs cross over thus exit loop
    else signalloop = 'not';
    end

% Agents welfare does not improve thus exit loop
```

```
                else signalloop = 'not';
                end

        % Agents wealth is not positive thus exit loop
        else signalloop = 'not';
        end

    end % End trades loop
```

Once the trades loop is finished, the function checks if there was at least one trade between agents a and b, that is, if the signaltrade variable is the 'yes' state. If so, it updates the wealth levels of agents a and b in the data structure a_str with the new wealth levels.

```
    if signaltrade == 'yes'
        % Update wealth levels of agents a and b in data structure if there was at
        % least one trade between them
        a_str(i,j).wealthspice = nwsua;
        a_str(u,v).wealthspice = nwspa;
        a_str(i,j).wealthsugar = nwsub;
        a_str(u,v).wealthsugar = nwspb;

    end

end  % End if for active agent
```

Finally, the last sentence of the neighborT function is an end sentence closes the if statement that checked if there was an active agent to trade with in the (u,v) location.

4 Results for the Model with Trade

In this section we present the results of a simulation of the model with trade for 400 periods, with an initial population of 708 agents randomly distributed on a 50x50 grid. Their metabolic needs of sugar and spice are randomly distributed between 1 and 5 units and vision ranges also randomly distributed between 1 and 5. The maximum levels of sugar and spice in the Sugarscape are set as equal to 30, and between 25 and 50 units of initial endowments of sugar and spice are randomly distributed among the agents. Figure 1 shows the distribution of sugar and spice in the Sugarscape. Cells with more sugar than spice are in scales

Figure 1: Four-peak Sugarscape

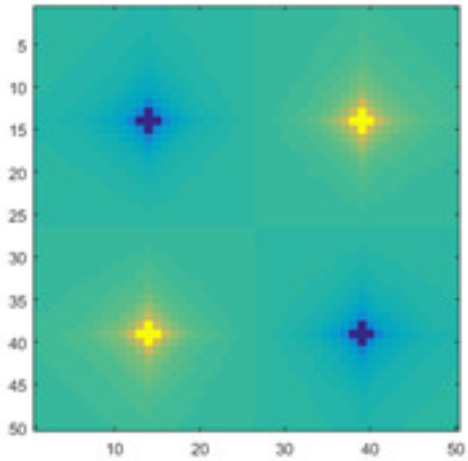
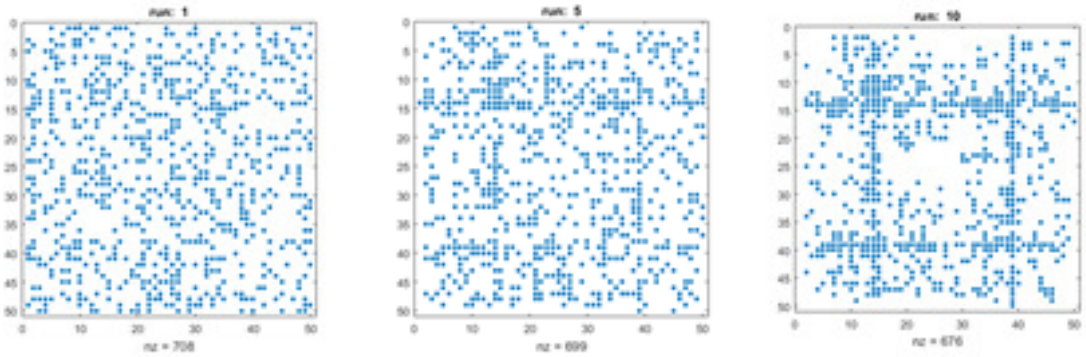


Figure 2: Evolution of the Agents Population for Selected Runs

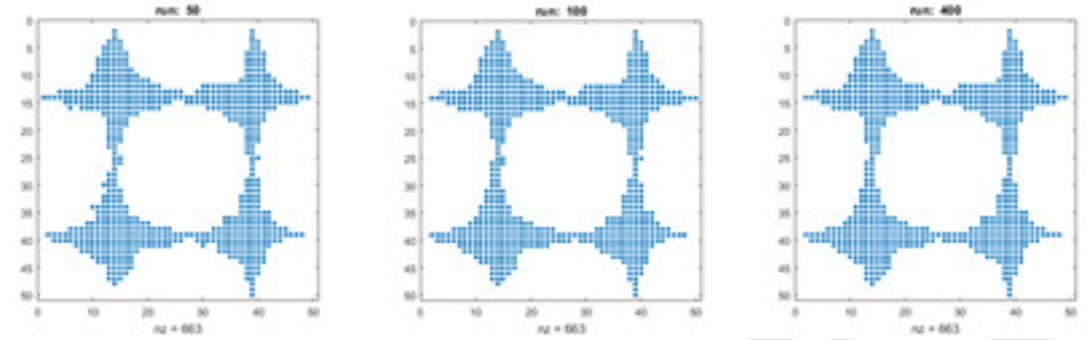


of yellow, those with more spice than sugar are in scales of blue, and those with more or less similar quantities of both resources are in scales of green. We can see that the Sugarscape contains four peaks, two with sugar and two with spice.

Figure 2 shows the evolution of the population of agents along some selected runs, where nz means non zero agents, that is, the number of agents alive in that period. We can see that the population decreases until it stabilizes in 663 agents, and also that agents tend to group on and around the peaks of resources.

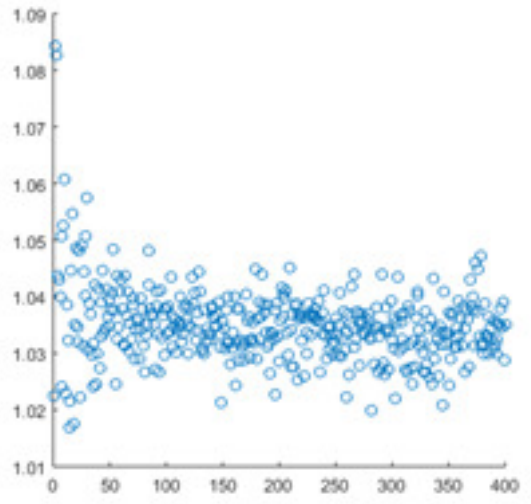
Figure 3 shows the evolution of the average price in each period. To moderate the influence of outliers, this average price is computed as the geometric mean of all prices within the period. Remember that at each period, each agent engages in a sequential bilateral bargaining process with its neighbors, and that

Figure 3: Evolution of the Average Price



at each step of this process a trade price is agreed upon. Thus, each period average price results from a large number of bilateral prices. We can see that prices are randomly distributed around a value near 1. Notice that since in the simulation there is approximately the same quantity of sugar and spice in the *Sugarscape*, the symmetry of initial stocks of sugar and spice and the symmetry of preferences implicit in the form of the welfare function W yield that the general equilibrium price corresponding to this economy would be approximately equal to 1. Thus, even though in this model we are dealing with boundedly rational agents, each restricted to a limited neighborhood to engage in trades, the average dynamics of the model leads to a statistical equilibrium close to the equivalent general equilibrium. However, notice also that that averages result from many prices that, within each period, may take values far away from 1.

Figure 3: Evolution of the Average Exchange Price



5 The Model with Trade and Cultural Influence

To model de process of cultural influence, we will assume that just as there are units of biological information in genes, there are units of cultural information called memes or cultural tags. And just as genes form a chromosome, we will assume that tag strings form cultural chains. Each agent will have a particular tag string, and the process of cultural influence will consist of the change in some tags in tag strings.

Thus, in this new model, we will assume that agents have a cultural attribute, in addition to other attributes such as metabolism and vision. The cultural attribute will be represented by a tag string in which each tag represents a cultural trait such as the kind of food it likes, the type of clothing it wears, etc. Wi will also assume that there are two different cultures, one named red (R) and the other named green (G), and that the tag string for an agent's cultural attribute contains 25 tags. So, a string like the following:

GGGGGGGGGGGGGGGGGGGGGGGGG

means that all the cultural traits of the agent belong to the green culture, while a chain like the following one:

GGGGRRRGGGGGRRRGRRRGRGRRR

indicates that the agent's culture is a mixture of the green and red cultures. We will also assume that cultural interaction between any pair of agents occurs during trade, that is, during the bilateral bargaining process. More specifically, we will assume that during trade between agents a and b, agent a exercises a cultural influence over b: when they meet to trade, one of the tags in agent's b tag string is randomly selected and replaced by the corresponding tag from agent's a tag string.

Assume now that f is the fraction of tags equal to R in an agents tag string. If more than half of the tags in an agent's tag string are red -i.e. if f is greater than 0.5- we will say that the agent belongs to the red culture. Otherwise it will belong to the green culture. Assume also that an agents preferences depend on the cultural group it belongs to. To introduce this feature in our model, we modify the agents welfare function:

$$W\left(w_{su},\, w_{sp}\right) = w_{su}^{m_{su}/m_T} \quad w_{sp}^{m_{sp}/m_T}$$

to this new function:

$$W\left(w_{su},\, w_{sp}\right) = w_{su}^{\left(\frac{m_{su}}{m_T}\right) f} \quad w_{sp}^{\left(\frac{sp}{m_T}\right)(1-f)}$$

where m_T is now:

$$m_T = m_{su}\, f + m_{sp}\,(1-f)$$

Thus, the MRS of spice per 1 unit of sugar becomes:

$$MRS = \frac{\frac{\partial W(w_{su},w_{sp})}{\partial w_{su}}}{\frac{\partial W(w_{su},w_{sp})}{\partial w_{sp}}} = \frac{\frac{w_{sp}}{m_{sp}\,(1-f)}}{\frac{w_{su}}{m_{su}}\, f}$$

Since f will change during the model's simulation due to the process of cultural influence, the welfare function representing agents preferences will also change, so that preferences become endogenous in this model of trade with cultural influence. And this will affect the determination and the dynamic of prices and quantities.[3]

6 The Model with Trade and Cultural Influence in MATLAB

The MATLAB representation of this model is essentially the same as the one of the model with trade only, with some minor additions and changes in the main program and in some functions. The initsugarscapeT, seeW, and moveagentT functions are exactly the same as in the previous model. The major change happens in the neighborT function, which is renamed as neighborTK function, and which will be the function where the cultural influence process takes place, together with the trade process.

The MATLAB representation of the model consists of a main program named sugarscapeTK. The changes in the code for this program w.r.t the code of the main program of the trade only program called sugarscapeT are as follows. In the initial conditions, we add the sentences:

```
stl = 25; \%odd number
groupR = zeros(1,nruns);
```

[3] Notice that in this new model an agent's metabolic needs can be eclipsed by cultural forces, such as in the cases of f near zero or near one, leading to neglect the needs for sugar o spice respectively, so that the agent may die from lack of any of those resources.

The first one defines the string tag length with an odd number, so that the $f > 0.5$ rule to assign agents to the red group, that is, when there are more R tags than G tags in the agent's tag string, is unambiguous. And the second one initializes to zero the vector containing the number of agents belonging to the red cultural group.

The function that initializes the agents' population is now called initagentsTK instead of initagentsT, and the only differences are that in the function call we add the stl variable:

```
%InitagentsTK
%Initialize agents population
function a_str = initagentsTK(size, vision, metsugar, metspice, initwup, initwlow,
```

and in the body of the initagentsTK function we add the following sentences:

```
% Create a vector with cultural tag string
ve = ones(1,stl);
for n = 1:stl
    if rand > 0.5
        ve(n) = 'R';
    else ve(n) = 'G';
    end
end
a_str(i,j).tagstring = ve;

%Initialize the cultural group to which de agent belongs to
fR = sum(ve(:) == 'R')/stl;
if fR > (1/2)
    a_str(i,j).group = 'R';
else
    a_str(i,j).group = 'G';
end
a_str(i,j).fR = fR;
```

The first part creates a vector with a cultural tag string, and the second part initializes the cultural group to which the agent belongs to. The first sentence defines and initializes the ve vector. Then the for loop randomly assigns the values R or G to each vector's element, and finally assigns vector ve to the tagstring attribute of the a_str data structure. The last sentence initializes to zero the group attribute of the a_str data structure. The fR contains the proportion of red tags in the agent tag string, i.e. the sum of red tags divided by the

tag string length stl. Then, if fR is greater than 0.5, the agent's group attribute is set as equal to 'R', otherwise is set as equal to 'G'. Finally, the value of fR is assigned to the fR attribute of the agent.

The function that displays the agents' locations at each run is called dispagentlocTK instead of dispagentT, and the only differences are that in the function call we add the groupR variable, and we also add the groupR variable and the a_str data structure in the function's return.

```
[groupR, a_str] = dispagentlocTK(a\_str, size, nruns, runs, groupR);
```

In the body of the dispagentlocTK function, in the definition of the attributes of the agent in the data structure a_str, we add the sentence:

```
groups(i,j) = a_str(i,j).group;
```

and at the end of the function we add, to compute average statistics for the R group:

```
% Compute average statistics for group R
groupR(runs) = sum(groups(:) == 'R') /( sum(groups(:) == 'R') + sum(groups(:) == '(
```

The function neighborWK is the same as neighborW, except that it contains the sentences:

```
f = a_str(i,j).fR;
mtot = msu*f + msp*(1-f);
```


They assign to the f variable the value of the fR attribute of the agent, i.e. the fraction of red tags in its tag string, and define the mtot variable according to the modified welfare functions corresponding to this model, as seen in Section 13 above. Thus, the welfare functions are as follows:

```
% Compute welfare
  welfareij = (wsu + ssugar(i,j))^((msu/mtot)*f) * ...
      (wsp + sspice(i,j))^((msp/mtot)*(1-f)) ;
  welfareuv = (wsu + ssugar(u,v))^((msu/mtot)*f) * ...
      (wsp + sspice(u,v))^((msp/mtot)*(1-f)) ;
```

The function seeTK is the same as seeT, except that in the function call it also passes the variable stl:

```
function [avpav, a_str] = see2(i,j,k,a_str,size,stl)
```

and in the main body of the seeTK function the neighborTK function is called, also passing stl to this function.

Finally, at the end of the main program, we add the following sentences to generate a plot of the evolution of the red cultural group.

```
figure;
scatter(axesx,groupR);
axis square;
```

6.1 NeighborTK

As we said above, the neighborTK function is where the cultural influence process takes place, together with the trade process. The first part of this function contains the code for the cultural influence process, while the second part contains the code for the trade process, which is almost the same as the code in the neighborT function of the trade only model. The neighborTK function begins with the statement:

```
function [aptv, a_str] = neighborTK(u,v,a_str,i,j,stl)
```

then the code of the cultural influence process follows.


```
% Check that cell (u,v) in the neighborhood of agent a is occupied by an agent
if (a_str(u,v).active == 1)

    % Define and assign values to temporary vectors
    veca = ones(1,stl);
    vecb = ones(1,stl);
    veca = a_str(i,j).tagstring;
    vecb = a_str(u,v).tagstring;

    % Randomly select a tag position in the cultural tag string
    n = ceil(rand * stl);

    % Set cultural tag in position n of agent b tag string as equal to the one of a
        vecb(n) = veca(n);
        a_str(u,v).tagstring = vecb;

    % Assign to fa the fraction of agent a tags equal to R
    fa = a_str(i,j).fR;
```

```
    % Update the  fraction of agent b tags equal to R
    fb = sum(vecb(:) == 'R')/ stl;
    a_str(u,v).fR = fb;

    % Update the cultural group of the b agent
    if fb > (1/2)
        a_str(u,v).group = 'R';
    else
        a_str(u,v).group = 'G';
    end

    %Avoid fa or fb being equal to 0 or 1
        if fa == 1
            fa = 0.999;
        end

        if fb == 1
            fb = 0.999;
        end

        if fa == 0
            fa = 0.001;
        end

        if fb == 0
            fb = 0.001;
        end

end % End if for active agent
```

After checking if there is an active agent b -located in the (u,v) coordinates- in the neighborhood of the a agent -located in the (i,j) coordinates-, the vectors veca and vecb are defined as vectors with ones, to later assign them the tagstrings corresponding to agent a and b respectively. Then a variable named n is created, to randomly select a tag position in the tagstring of the agents. This is done by assigning to n the ceiling -i.e. the nearest higher integer number- of the product of the rand MATLAB function times the stl variable, which contains the tagstring length. Then the n cell of the tagstring of the b agent vector is set as equal to the n cell of the tagstring of the a agent vector, and the resulting tagstring of the b agent is assigned to its tagstring attribute in its data structure

a_str.

The fa and fb variables contain the proportions of red tags in the tagstrings of agents a and b. If fb is greater than 0.5, the group attribute of agent b is set as equal to R -the agent belongs to the red group-, otherwise is set as equal to G -the agent belongs to the green group. Finally, if fa or fb are equal to 0 or 1, they are set as equal to 0.001 or 0.999, otherwise the formulas to compute the MRSs in the trade process part of the function will contain divisions by zero.

As we said above, the second part of the neighborTK function contains the code for the trade process, which is almost the same as the code in the neighborT function of the trade only model. There are just minor changes related to the fa and fb variables. To introduce fa and fb in the welfare functions and in the MRSs equations, we substitute the sentences to define the intermediate variables mtota and mtotb by the following ones:

```
mtota = msua*fa + mspa*(1-fa);
mtotb = msub*fb + mspb*(1-fb);
```

the sentences to compute the old welfare and the old MRSs by the following ones:

```
 % Compute old welfare
waold = owsua^((msua/mtota)*fa) * owspa^((mspa/mtota)*(1-fa)) ;
wbold = owsub^((msub/mtotb)*fb) * owspb^((mspb/mtotb)*(1-fb)) ;

% Compute old mrs
omrsa = (owspa / (mspa * (1-fa))) / (owsua / (msua * fa));
omrsb = (owspb / (mspb * (1-fb))) / (owsub / (msub * fb));
```

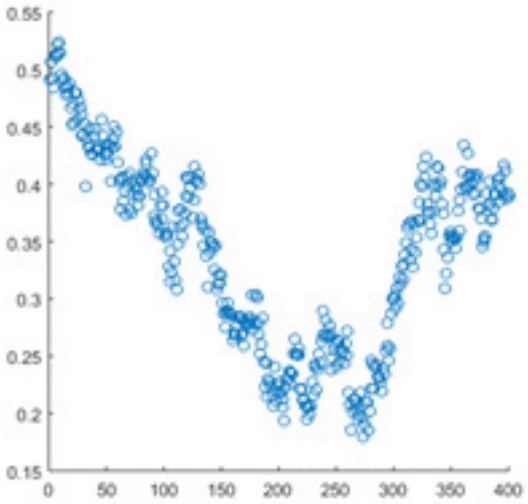
and the sentences to compute the new welfare and the new MRSs by the following ones:

```
% Compute new welfare if trade is carried out
      wanew = nwsua ^ ((msua/mtota)*fa) * nwspa ^ ((mspa/mtota)*(1-fa));
      wbnew = nwsub ^ ((msub/mtotb)*fb) * nwspb ^ ((mspb/mtotb)*(1-fb));

          % Check that trade improves welfare of both agentes
          if (wanew > waold) && (wbnew > wbold)

          % Compute new mrs if trade is carried out
          nmrsa = (nwspa / (mspa * (1-fa))) / (nwsua / (msua* fa));
          nmrsb = (nwspb / (mspb * (1-fb))) / (nwsub / (msub * fb));
```

Figure 4: Evolution of the proportion of agents in the red cultural group



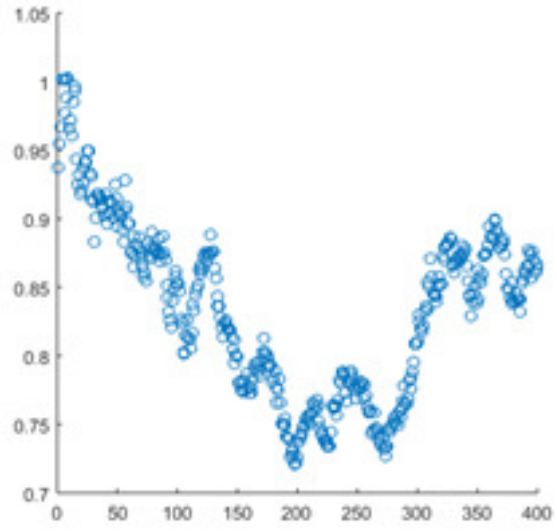
7 Results for the Model with Trade and Cultural Influence

Figure 4 shows the process of cultural evolution throughout a simulation of the model, with the same parameters as those used in the simulation of the trade only model whose results we presented in Section 12, and with a tagstring of lengh equal to 25. We can observe that the simulation begins with a similar proportion of agents belonging to the red and green culture, since f is approximately equal to 0.5, then the proportion of agents belonging to the red culture decreases irregularly, then increases.[4]

Figure 5 shows the evolution of the average exchange price. We can observe that in this experiment that price follows a more random path than in the experiment performed with the trade only model in Section 12. This is due to the fact that in this model agents' preferences are constantly changing as a result of the process of cultural influence.

[4] in a population engaged in a process of cultural influence (tag flipping) and cultural grouping (tag majority) such as the one we are dealing with is not uncommon, after a very large number of runs, to observe a convergence to a monochromatic state, or the change from a near monochromatic state to another.

Figure 5: Evolution of the average exchange price



7.1 Experiments

Fort the model with trade only, you may want to try different rules to determine the price in the bilateral bargaining process, instead of the geometric mean of the MRSs of the agents. For instance, you may try the mean of their MRSs, or a random value within the interval defined by the MRSs of both agents, and see if it affects the price dynamics. Also, you may increase the agents' range of vision and see if it has an impact on the price dispersion. You may also use the mean instead of the geometric mean to compute the average exchange prices within each period of the simulation, and see the magnitude of outliers in the price dynamics.

For the model with trade and cultural influence, you may change the cultural string length and see the impact on the formation of cultural groups and on the price dynamics. You may try different cultural influence rules, flipping more than one the neighbor's tags, or reversing roles between agent and neighbor, making the neighbor to flip the agents tags. You may explore the convergence to a monochromatic state comparing simulations with very large and different number of runs –e.g. 500, 1000, 3000.