COMPUTATIONAL ECONOMICS

Revised Edition

Agent-Based Sugerscape Model in MATLAB

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

1 introduction

Agent-based computational economics is one of the newer fields. Agent-based models simulate the behavior of multiple heterogeneous agents interacting in a variety of ways. While the modeling of economic agents has a long tradition, agent-based modeling departs from it in a number of ways. For example, when modeling a market economy, the standard neoclassical competitive general equilibrium approach usually assumes that agents have fixed preferences, perfect and complete information, and no reproductive behavior, as well as that trade is organized by a central auctioneer, which given all agents' preferences and endowments, computes the set of equilibrium prices. Thus, agents are price-takers and do not engage in trade at prices other than those given by the central auctioneer. Moreover, space, which is geography, is usually an absent dimension in that approach. In contrast, agent-based models allow agents to display a number of more realistic characteristics and behaviors, that is, changing preferences, bounded rationality and memory, imperfect and incomplete information, and local trade; agents may interact with neighbors in a geographically defined space and prices emerge from these decentralized interactions.

In this chapter we introduce a famous agent-based model known as the Sugarscape model, developed by Joshua M. Epstein and Robert Axtell (1996). This is a model designed to simulate a variety of social phenomena, such as population dynamics, migration, interaction with the environment, trade, group formation, combat, and transmission of culture. We show how to represent and simulate the simplest version of this model in **MATLAB**. To do this, the knowledge of basic **MATLAB** operations and data types—vectors and matrices, with the addition of data-type named structures and cell arrays, which we explain later—suffice. However, more sophisticated simulations may require the use of object-oriented programming techniques, also available in MATLAB (see ***MATLAB*** *Classes and Objects* in the Programming and Data Types section of the **MATLAB** help navigator) as well as in lower-level object-oriented programming languages such as C++, C#, and Java.

2 The Sugarscape Model

2.1 Introduction to the Model

The version of the classic Sugarscape model that we use in this chapter can be thought of as two major cities located near one another, like Dallas and Fort Worth in Texas or Minneapolis and St. Paul in Minnesota. There is an original distribution of stores of a certain type in this terrain; for example, coffee houses such as Starbucks or perhaps mailing and business services stores such as UPS Stores. The franchise owners at each location work with varying degrees of efficiency and thus have different costs, so they require different levels of revenue in order to continue to make a profit. Their profit each period is added to their accumulated wealth; however, if this wealth goes to zero the franchise is shut down. The surviving franchise owners look around each period for a nearby location that would be more favorable, and they move the store if they find a higher-revenue location. However, some of the franchise owners scout further away from their present location than others.

More formally, the Sugarscape model consists of two main elements: a terrain where events unfold named *sugarscape*, which contains the spatial distribution of a generalized resource named *sugar*, which can be thought of as the customer potential or revenue level at that location. The agents have metabolism levels and must eat to survive. This metabolism may be thought of as the cost of running the business in each period. Thus the difference between the sugar that the agents obtain at their location in each period and their metabolism level is like the profit of the enterprise in each period. This profit is accumulated as wealth from period to period; however, if the wealth level goes to zero in any period the agent dies, that is, goes out of business. Thus, the agents are characterized by a set of fixed states (genetic characteristics such as metabolism and length of vision) and variable states (such as location and wealth), and they move around the sugarscape following simple rules of behavior.

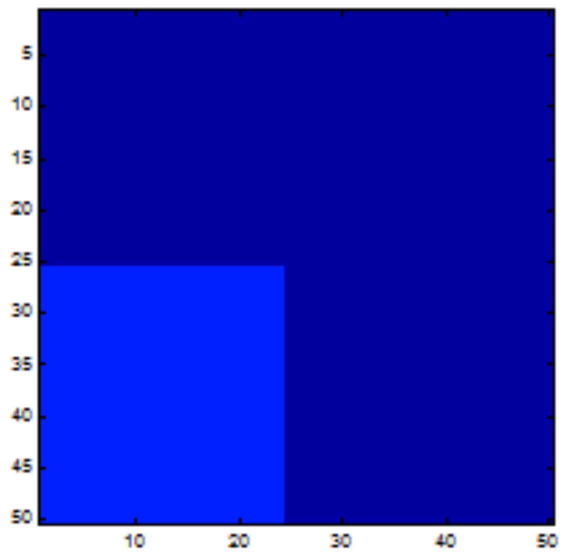
The sugarscape is represented by a two-dimensional coordinate grid or lattice. At every point of the grid given by the coordinates (x,y) there is a sugar level. Thus, we can easily represent the sugarscape in **MATLAB** by means of a matrix. For example, if we want to create and display a 50×50 sugarscape with a level of sugar equal to four units in the southwest quadrant and a level of two units elsewhere, we can do it with the following statements:

```
for i = 1: 50;
    for j = 1: 50;
        if (i > 25 & j < 25)
            s(i,j) = 4;
        else
            s(i,j) = 2;
        end
    end
end
```

```
image(s);
```

In the foregoing statements image(s) is a MATLAB function that displays the array s. Figure 14.1 shows the result, where the lighter region corresponds to the value four and the darker region corresponds to the value two.

Figure 1: Sugarscape with two levels of sugar



To represent agents, we can use another data type available in **MATLAB** called a structure, which is an array with data containers named *fields*. These fields can contain any kind of data. For example, let us assume that every agent is characterized by two states: active, which signals if the agent is alive or not with values equal to 1 and 0, respectively, and metabolism, which is the amount of sugar each agent has to eat per time period to survive. The statements

```
a_str.active = 1;
a_str.metabolism = 4;
```

create the simple 1×1 structure a_str containing two fields. If we use the statements

```
a_str(2).active = 1;
a_str(2).metabolism = 3;
```

then a_str becomes a 1×2 array with two fields. Let us assume that we want to create and display a random population of agents—say all those for whom the corresponding value from a [0,1] uniform distribution is lower than 0.2—on a 50×50 grid. We also assume that there can only be one agent at each location. We can achieve this using the following statements, with which we can create a structure with 2500 elements, each with two fields:

```
for i = 1:50;
    for j = 1:50;
        if (rand < 0.2)
            a_str(i,j).active = 1; % put an agent on this location
            a_str(i,j).metabolism = 3;
        else
            a_str(i,j).active = 0; % keep this location empty
            a_str(i,j).metabolism = 0;
        end
    end
end
```

If we want to display the location of every agent on the grid, we can do it with the following statements, where we transfer the elements of the field active into the a matrix and the **MATLAB** function spy(a) displays all the nonzero elements in matrix a:

```
for i = 1:50;
    for j = 1:50;
        a(i,j) = a{\_}str(i,j).active;
    end
end
spy(a);
```

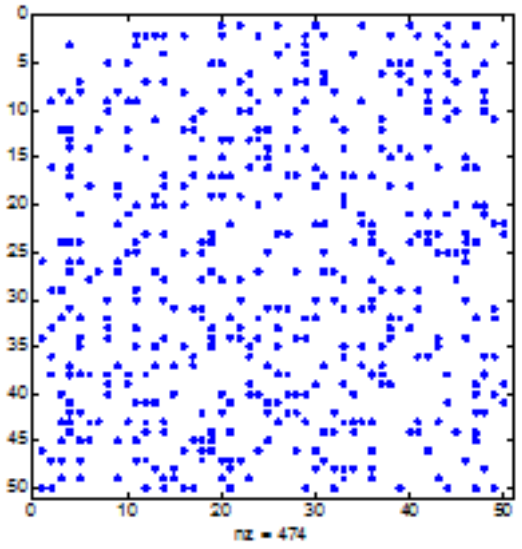
The result, with a number of agents equal to 474, is shown in Figure 14.2, where nz means the number of nonzero elements.

Now that we have introduced the basic building blocks of the Sugarscape model and its **MATLAB** representation, we can move on to a more detailed presentation.

2.2 Characterizing the Model

In this section we present a more complex topography for the sugarscape and also more complex agent characteristics. We also define rules that govern the

Figure 2: Agents' location.



autonomous growth of sugar in the sugarscape and the movement of the agents on it.

We assume that the sugarscape is characterized by two mountains of sugar, one in the southeast portion of the grid and the other in the northwest, and that these two mountains are symmetric. Thus, for a 50×50 grid, we assume that one peak of the sugarscape is approximately on the (0.75 * 50, 0.25 * 50) coordinate, while the other is on the (0.25 * 50, 0.75 * 50) coordinate. From the peaks down, the level of sugar at each location follows decreasing paths.

We also specify a very simple grow-back rule for the sugarscape:

Sugarscape rule G_∞: Grow back to full capacity immediately.

Thus, at each run of the model, the level of sugar grows back to its initial height. The symbol G_∞ is a fancy way of specifying how rapidly the amount of sugar (revenue) grows back in each time period. Epstein and Axtell (1996) use a variety of such rules.

We also assume that the sugarscape is what in geometry is known as a torus or, in more familiar terms, that it corresponds to the surface of a donut. This means, for example, that an agent moving to the south on column 6, after reaching row 50 appears on the sugarscape from the north in the coordinate (1,6), and an agent moving to the east on row 6, after reaching column 50 appears on the sugarscape from the west on the coordinate (6,1). Analogous patterns are followed by agents moving north or west.

Turning now to the agents, we assume that each agent has four characteristics, two of them fixed and the other two variable. The fixed characteristics are metabolism—the amount of sugar the agent has to consume in each time period to stay alive—and vision—the number of sites in the sugarscape each agent can see. We assume that agents can see only in four directions: north, south, east, and west, so they cannot see in a diagonal line. The level of vision is the maximum number of sites each agent can see in a given direction. Metabolism and vision are genetic characteristics randomly distributed among agents.

The variable characteristics of agents are location on the sugarscape and wealth, with the latter understood as the agents' stocks of sugar. We assume that agents are randomly born around the sugarscape at the beginning of the simulation. Each agent starts its life with a level of wealth equal to the level of sugar in the sugarscape location were it was born.

We specify a rule that governs the behavior of each agent on the sugarscape:

Agent movement rule M:
@bl:• Look out as far as vision permits in the four principal directions and identify the unoccupied site(s) having the most sugar.
• If the greatest sugar value appears on multiple sites then select the nearest one.
• Move to this site.
• Collect all the sugar at this new position.
@normal:Once sugar is collected, the agent's wealth is incremented by the sugar collected and decremented by its metabolic rate. An agent lives forever unless its wealth falls below its metabolic rate, in which case it dies and is removed from the sugarscape. In principle, all agents should apply this rule simultaneously. However, since the simulation is run on a serial computer, only one agent is active at any given instant. In this case, the recommendation is to randomize agents' order of movement, and we do so in the **MATLAB** code. We also randomize the first step of the rule, that is, the order in which each agent searches the four directions.

Having presented the building blocks of the simplest version of the Sugarscape model, we now turn to its **MATLAB** representation.

3 The Sugarscape Model in MATLAB

The **MATLAB** representation consists of a main program named sugarscape1.m and a number of functions, all of which are available from the book web site. The code of the main program follows:

```
%Sugarscape1
%sugarscape: (two-peak sugarscape, rule: Ginf)
%agents: (moving sequence: random, view: four directions, rule: M)

%Initialize model parameters
nruns = 6;
size = 50; %even number
metabolismv = 4;
visionv = 6; %set always smaller than size
maxsugar = 20;

%Initialize sugarscape and display
s = initsugarscape(nruns, size, maxsugar);

%Initialize agents population
a_str = initagents(size, s, visionv, metabolismv);


%Main loop (runs)
for runs = 1:nruns;

    %Display agents locations
 dispagentloc(a_str, size, nruns, runs);

 %Select agents in a random order and move around the sugarscape following rule M
    for i = randperm(size);
        for j = randperm(size);

            if (a_str(i,j).active == 1) %is there an angent on this location?

                %Agent explores sugarscape in random directions and selects best lo
                temps = s(i,j);
                tempi = i;
                tempj = j;
                for k = a_str(i,j).vision : -1 : 1;
                    [temps, tempi, tempj] = see(i,j,k,a_str,s,size,temps,tempi,temp
                end

                %Agent moves to best location, updates sugar stock and eats sugar
                a_str = moveagent(a_str, s, i, j, temps, tempi, tempj);
            end         % if
        end             % for j
    end                 % for i
end                     % for runs
```

The program begins with the initialization of the model parameters: the number of runs, the size of the sugarscape, the maximum value of the metabolism and vision of the agents, and the maximum level of sugar in the sugarscape. Then follows a call to the function initsugarscape, which returns a matrix s containing the sugar levels in the sugarscape. Next a call to the function initagents returns the data structure a_str that contains the agents' population.

Then follows the main loop of the program corresponding to the number of runs—each run represents a time period—of the simulation. At each pass of the loop, the locations of the agents on the sugarscape are displayed as a way of visualizing their movements. This is achieved by calling the function dispagentloc.

Then each agent, in a random order, explores the sugarscape, selects the best location, updates its wealth, and eats sugar to survive. This section of the program begins with the following statements:

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

The randperm(n) function performs a random permutation of the elements of the set (1, 2,. . . , n). Thus the randperm(size) **MATLAB** function creates a vector with a number of elements equal to size and performs a random permutation of those elements. Thus, once the two for loops—one for i and the other for j—are completed, the whole population of agents will have moved but in a random order. The conditional

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

checks if there is an active agent in the (i,j) location being examined, where a 1 in the field active of the agent data structure denotes that there is an agent, whereas a 0 denotes the opposite. If there is an active agent in the location, the program proceeds to apply the agent's rule of movement, whereas if that is not the case it proceeds to examine another location looking for an active agent. The agent's rule of movement is implemented with the following statements:

```
% Agent explores sugarscape in random directions and
% selects best location

temps = s(i,j);
tempi = i;
tempj = j;

for k = a_str(i,j).vision : -1 : 1;
    [temps, tempi, tempj] =
    see(i,j,k,a{\_}str,s,size,temps,tempi,tempj);
end

% Agent moves to best location, updates sugar stock and
% eats sugar

a_str = moveagent(a_str, s, i, j, temps, tempi, tempj);
```

The statements begin with the setting of three temporary variables: temps contains the level of sugar 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 see 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 returns the maximum level of sugar in the variable temps and its location coordinates in the variables tempi and tempj, respectively. Finally, once the loop is completed, the function moveagent is called to move the agent to the new location and update its stock of wealth.

From this overview of the main program we turn next to descriptions of the functions.

4 Functions

4.1 Initsugarscape

The initsugarscape function initializes the level of sugar at each location of the sugarscape. To better understand the procedure, we begin with simpler examples. Suppose that we want to generate an 11 $\times$ 11sugarscape s1 with a single mountain with a peak in the center. The corresponding statements are shown in the following, where i and j are the matrix coordinates varying from 1 to 11. The vectors x and y are two identical eleven-element vectors containing the values [–5 –4 –3 –2 –1 0 1 2 3 4 5]:

```
% Generate sugarscape with one peak in the center
x = -5:5;
y = -5:5;
maxsugar = 20;

for i = 1:11;
        for j = 1:11;
                if (x(i) == 0 {\&} y(j) == 0)
                        s1(i,j) = maxsugar;
                else
                        s1(i,j) = maxsugar / (abs(x(i)) + abs(y(j)));
                end
        end
end
```

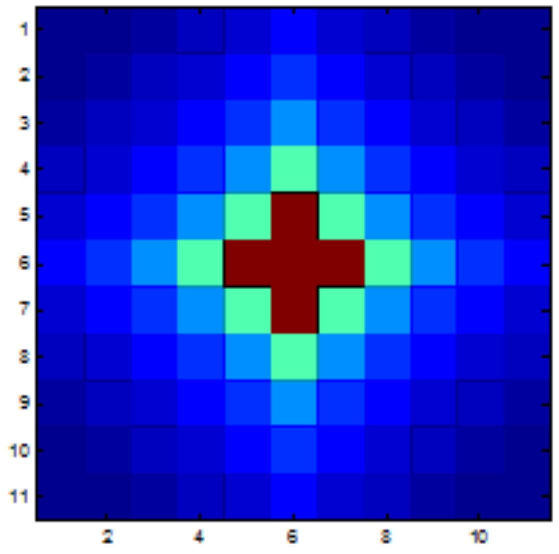
The value of each element in the s1 matrix is computed by dividing the given maximum level of sugar by the sum of the absolute value of the corresponding elements in the x and y vectors:

```
s1(i,j) = maxsugar / (abs(x(i)) + abs(y(j)));
```

where abs is the absolute value. The peak of the mountain is where the corresponding elements of the x and y vectors equal zero. Thus, the value of s1(6,6), which is located at the center of the sugarscape, is equal to maxsugar—making a minor adjustment to avoid the division by zero. The value on the corners—that is, s1(1,1)—is equal to (maxsugar/10). All the other values would be, in descending order, between maxsugar and (maxsugar/10), as shown in Figure 14.3.

Now, if we want to generate a sugarscape with a peak in the southeast instead of the center, the values of x and y should be shifted to

Figure 3: Sugar scape with center peak.



x = [–9 –8 –7 –6 –5 –4 –3 –2 –1 0 1]

and

y = [–3 –2 –1 0 1 2 3 4 5 6 7].

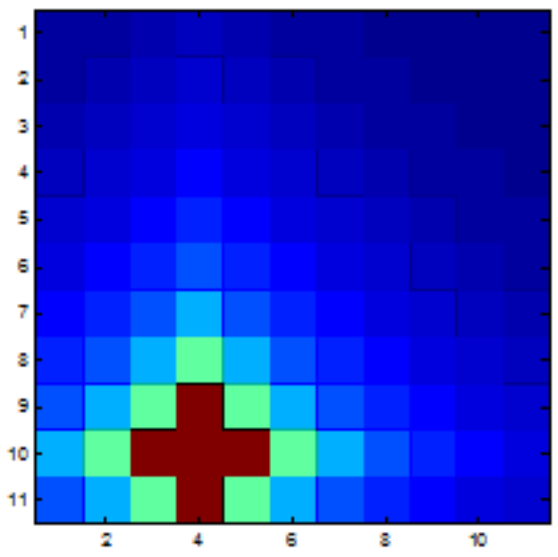
In this case, the peak of the sugarscape is in the s1(10,4) location, as shown in Figure 14.4.

The initsugarscape function initializes the level of sugar at each location of the sugarscape. This particular function generates a topography characterized by two mountains of sugar, one in the southwest portion of the grid and the other in the northeast. These two mountains are symmetric. From the peaks down, the level of sugar follows decreasing paths. The function code is available in file initsugarscape.m.

This function begins by generating a sugarscape s1 containing a single peak in the southwest. To do so, the *Generate sugarscape with one southwest peak* section of the function, reproduced below, applies a similar procedure to the one just described:

```
%Generate sugarscape with one south west peak
x = -ceil(0.75*size) : size-ceil(0.75*size)-1;
y = -ceil(0.25*size) : size-ceil(0.25*size)-1;
```

Figure 4: Sugar scape with a southwest peak.



```
for i = 1:size;
        for j = 1:size;
                if (x(i) == 0 {\&} y(j) == 0)
                        s1(i,j) = maxsugar;
                else
                        s1(i,j) = maxsugar / (abs(x(i)) + abs(y(j)));
                end
        end
end
```

For example, for a value of size equal to fifty, it begins by generating a fifty-element vector x. The statement

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

is used to create a fifty-element vector of integers as follows. The values in the vector begin at minus the ceiling of the product (0.75 * 50), that is, the next integer above 37.5, namely 38. They end at the value (50 – 38 –1), that is, 11. So x is a fifty-element vector with the values

$$[-38, -37, \cdots, -1, 0, 1, \cdots, 10, 11]$$

Thus, the value zero is in the thirty-ninth position of the x vector. In a similar way the vector y, which goes from –13 to 36, is generated with the value zero in the fourteenth position. After this is done, each element of the sugarscape matrix s1 is generated. The result is a sugarscape with a peak in the s1(39,14)location, that is, in the southwest corner of the array.

Once the first mountain is generated, a symmetric one is obtained by transposing the matrix s1 with the statement

```
s2 = s1';
```

Then, the statement

```
s = s1 + s2;
```

generates the two-peak sugarscape. The two statements

```
maxrow = max(s);
max(maxrow)
```

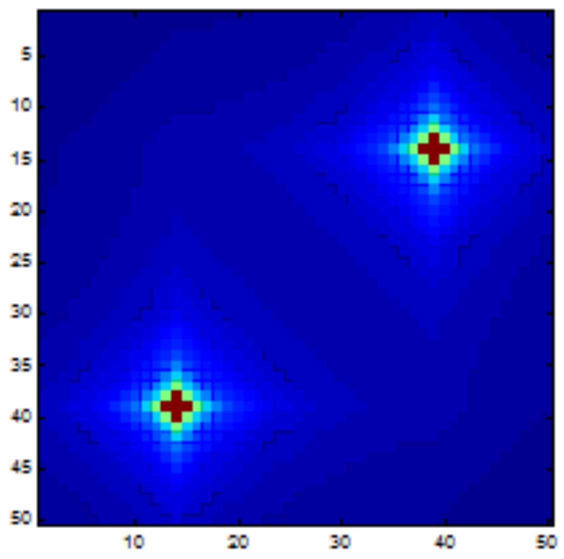
compute the row containing the maximum value in the matrix s and print the maximum value in this row. This may seem redundant, since we set the parameter maxsugar at the beginning of the program. That value is indeed the maximum for the peaks in s1 and s2. But the peaks in s are a bit higher since we add to each original peak the value of the corresponding cell in the symmetric matrix, which is a low value given its distance from the peak.

The final statements display the image of the sugarscape shown in Figure 14.5:

```
figure(1);
imagesc(s);
axis square;
```

The statement figure(1)generates a figure where an image is displayed, and imagesc(s) scales the data in matrix s to the full range of colors and displays the corresponding image of the sugarscape matrix s. Finally, the statement axis square makes the image square. Th
e result is two centers of economic activity, as shown in Figure 14.5.
Next we turn from the code for the sugarscape to the code for the agents.

Figure 5: Two-peak sugarscape.



4.2 Initagents

The function initagents generates a random initial population of agents, which has the following code:

```
%Initagentsage
%Initialize agents population including age range
function a_str = initagentsage(size, s, visionv, metabolismv);
for i = 1:size;
    for j = 1:size;

        if (rand < 0.2)
            a_str(i,j).active = 1; %put an agent on this location
            a_str(i,j).metabolism = ceil(rand * metabolismv);
            a_str(i,j).vision = ceil(rand * visionv);
            a_str(i,j).wealth = s(i,j);

        else
            a_str(i,j).active = 0; %keep this location empty
            a_str(i,j).metabolism = 0;
            a_str(i,j).vision = 0;
            a_str(i,j).wealth = 0;
        end
    end
end
```

The information about agents is stored in the data structure a_str with four fields. The field active contains a 1 or 0 depending of the situation of the agent in a specific location (active, i.e., alive; or inactive, i.e., dead). A location with an inactive agent is treated in the main program and other functions as empty. If the values generated by the uniform distribution **MATLAB** function rand are below 0.2, an agent is born.

The fields metabolism and vision contain the corresponding integers randomly distributed between 1 and the maximum level of each characteristic. The MATLAB function ceil is used to round out the randomly created vision and metabolism variables up to the next integer. The field wealth is initialized as equal to the amount of sugar in the location of the sugarscape where the agent was born.

4.3 Dispagentloc (Display Agent Location)

This simple function transforms the field agent from the agent's data structure into a matrix named a and displays agents' locations, since MATLAB does not allow one to display that field directly. The code of the function is as follows:

```
%Dispagentloc
%Transform field "agent" from data structure into matrix and display agents locati
function dispagentloc(a_str, size, nruns, runs);
a = zeros(size); av = zeros(size); am = 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;
            am(i,j) = a_str(i,j).metabolism;
        end
    end
end

figure(2);
subplot(ceil(sqrt(nruns)),ceil(sqrt(nruns)),runs), spy(a);
```

```
axis square;

avgvision = sum(sum(av))/sum(sum(a))
avgmetabolism = sum(sum(am))/sum(sum(a)
```

The statement figure(2)tells MATLAB to display a second figure with the agent's locations—remember that a first figure was created previously to display the sugarscape.

Consider next the line of code

```
subplot(ceil(sqrt(nruns)),ceil(sqrt(nruns)),runs), spy(s);
```

and note that this one line contains two separate **MATLAB** statements, that is, the function calls

```
subplot()
```

and

```
spy()
```

The call to subplot divides the window into a number of panes and the call to spy plots the active pane. These statements allow us to display multiple images in a single figure, such as the images of agents' locations in successive runs of the program. The **MATLAB** function

```
subplot(m,n,p);
```

creates an axis in the pth pane of a figure divided into an m $\times$ n matrix of rectangular panes. For example, if we set the number of runs parameter in the main program equal to eight, then the statement

```
subplot(ceil(sqrt(nruns)),ceil(sqrt(nruns)),nruns), spy(s);
```

where ceil(sqrt(nruns)is the ceiling (i.e., the integer above), the square root of the number of runs divides the figure (window) into a matrix with three rows and three columns of panes to accommodate the images of the agent's locations in successive runs.

4.4 See and Neighbor

The see and neighbor functions explore 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. Remember that the location coordinates of the agent are given by (i,j) and that the agent's level of vision is equal to k. For each integer between k and 1—that is, going from the outermost part of the neighborhood to its center—the function checks the level of sugar in each of the four directions. Every time the level of sugar in a location being examined is greater than the level in the agent's location, the level and coordinates of the higher value are stored in the temporary variables temps, tempi, and tempj, respectively. Thus, at the end of the exploration, these variables contain the highest level of sugar found and its location.

Imagine that we begin by exploring the neighborhood to the south for a level of vision equal to k and from the location (i,j). We want to examine the location (i+k,j). If (i+k <= size), where size is the dimension of the sugarscape, there is no problem. However, if (i+k > size), we have to remember that in Section 14.3.2 we defined the sugarscape as a torus, so in this case the location to be examined is (i+k-size, j). For example, if we start from the location (48,2) with k = 6, then the location to be examined is (4,2). Thus, to summarize, we can write the following pseudocode, where neighbor is a function that checks the level of sugar in the location (u,v):

```
if (i + k $>$ size)
    u = i + k - size;
    v = j;
    neighbor(u,v);
else
    u = i + k;
    v = j;
    neighbor(u,v);
end
```

Now, in the case in which we want to examine the neighborhood to the north, the code should be

```
if (i - k < 1) % or equivalently if(k - i > -1)
    u = i - k + size;
    v = j;
    neighbor(u,v);
else
```

```
    u = i - k;
    v = j;
    neighbor(u,v);
end
```

Analogous codes can be written for the neighborhoods to the east and west. However, we want to write a general code that encompasses all four cases, that is, something of the form

```
if (c{m}(1) > c{m}(2))
       u = c{m}(3);
       v = c{m}(4);
       [temps, tempi, tempj] = neighbor(u,v,a_str,s,temps,tempi,tempj)
else
    u = c{m}(5);
    v = c{m}(6);
    [temps, tempi, tempj] = neighbor(u,v,a_str,s,temps,tempi,tempj);
end
```

To do so, we proceed as follows. We define the following four vectors, each with six elements:

```
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];
```

Next we make use of a **MATLAB** object named cell array. A cell array is an array whose elements are also arrays. For our case, think of it as a matrix whose elements are vectors instead of numbers. The following statements create a cell array of dimension 1×4 whose elements are the vectors south, north, east, and west. Note that the indices of a cell array are shown within braces:

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

Now, for example, if we want to access the third element of the north vector, we can do it using a double indexing notation such as

```
c{\{}2{\}}(3);
```

Then, a general code to explore the neighborhood of an agent, selecting four directions of search in a random manner, can be written as:

```
for m = randperm(4);
    if (c{m}(1) > c{m}(2))
        u = c{m}(3);
        v = c{m}(4);
        [temps, tempi, tempj] = neighbor(u,v,a_str,s,temps,tempi,tempj);
    else
        u = c{m}(5);
        v = c{m}(6);
        [temps, tempi, tempj] = neighbor(u,v,a_str,s,temps,tempi,tempj);
    end
end
```

To check this go through the south and then the north cases and you should get the same results as those previously shown.

We turn now to explain the workings of the neighbor function, which is a very simple one. As can be seen in the foregoing code this function receives as inputs, among other arguments, the variables temps, tempi, and tempj and returns the same variables as outputs. Remember that temps contains the level of sugar in a given location and tempi and tempj contain the coordinates of the location. The code of the neighbor function is as follows:

```
%Neighbor
function [temps, tempi, tempj] = neighbor(u,v,a_str,s,temps,tempi,tempj);
if (a_str(u,v).active == 0)
    if (s(u,v) >= temps)
        temps = s(u,v);
        tempi = u;
        tempj = v;
    end
end
```

Thus, the function first checks whether the (u,v) location is free so that an agent can move there. If that is the case, it checks to see whether or not the level of sugar in the (u,v) location of the sugarscape is greater than or equal to the one previously found and stored in the variable temps. If so, it puts the new level found in the temps variable and its corresponding (u,v) coordinates in the variables tempi and tempj.

To conclude this section, we reproduce below the entire code of the see function:

```
% See
function [temps, tempi, tempj] = see(i,j,k,a_str,s,size,temps,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);
        [temps, tempi, tempj] = neighbor(u,v,a_str,s,temps,tempi,tempj);
 else
        u = c{m}(5);
        v = c{m}(6);
        [temps, tempi, tempj] = neighbor(u,v,a_str,s,temps,tempi,tempj);
 end

end
```

4.5 Moveagent

Once the neighborhood of the agent has been examined, it is time to move the agent to the best location found, update its wealth, and let it eat sugar. This is the job of the moveagent function that follows:

```
%Moveagent
function a_str = moveagent(a_str, s, i, j, temps, tempi, tempj);

if (temps > s(i,j))
%Agent moves to best location and updates wealth
    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).metabolism = 0;
    a_str(i,j).wealth = 0;
    % Update wealth at new location
    a_str(tempi,tempj).wealth = a_str(tempi,tempj).wealth + temps - a_str(tempi,ter
    % If wealth is less than zero set location to unoccupied
    if (a_str(tempi,tempj).wealth <= 0)
        a_str(tempi,tempj).active = 0;
        a_str(tempi,tempj).vision = 0;
        a_str(tempi,tempj).metabolism = 0;
        a_str(tempi,tempj).wealth = 0;
    end
else
%Agent stays in position and updates wealth
    a_str(i,j).wealth = a_str(i,j).wealth + temps - a_str(i,j).metabolism;
    if (a_str(i,j).wealth <= 0)
        a_str(i,j).active = 0;
        a_str(i,j).vision = 0;
        a_str(i,j).metabolism = 0;
        a_str(i,j).wealth = 0;
    end
end
```

If a new and better location than the one previously occupied by the agent is found, that is, if the statement

```
if (temps > s(i,j))
```

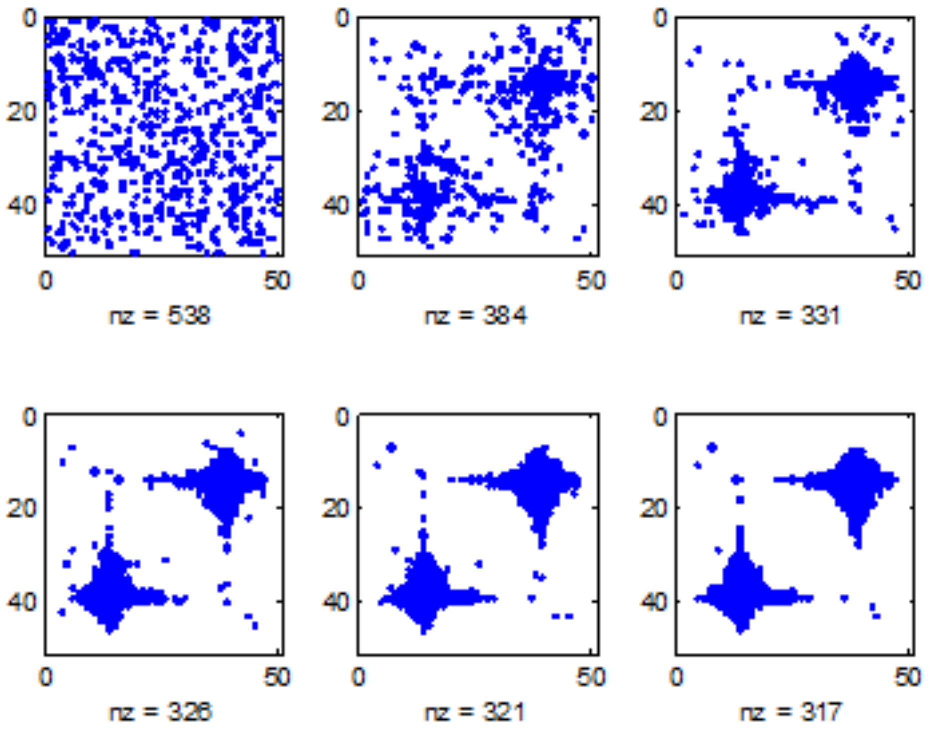
is true, the agent moves to the new location whose coordinates are stored in the variables tempi and tempj. The old location is set to unoccupied, and the agent's wealth is updated adding the amount of sugar found in the new location to its previous wealth and subtracting the sugar to be consumed according to its metabolic rate. If the resulting level of wealth is less than or equal to zero then the agent dies and all its fields are set to zero.

In the case that no better location is found, the agent stays in place, updates its wealth, and eats its sugar. Again, if the resulting level of wealth is less than or equal to zero, the agent dies.

5 Results

We are now ready to analyze the behavior of the population of agents in the sugarscape given the topography, the grow-back rule G_∞, and the agents' rule of movement M. The agents' locations for six successive runs for a maximum vision of 6 and a maximum metabolism equal to 4 are shown in Figure 14.6. The order of graphs corresponding to the successive runs goes from left to right and then down to the next row.

Figure 6: Agents' location for six runs

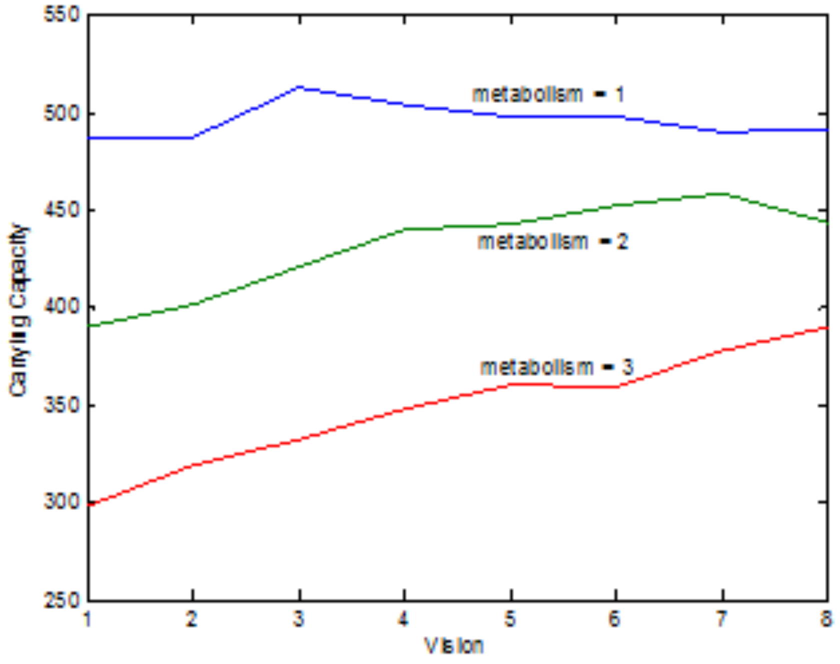


We can observe that in the first run there is a total population of 538 agents (nz means nonzero elements) randomly distributed on the sugarscape. As one would expect, during each run some agents die and others move toward the peaks of the sugarscape.

For this experiment, the average metabolism of the population goes from 3.5 in the first run to 2 in the sixth run while the average vision goes from 3.5 to 3.8. Thus, as one should expect, lower metabolism and higher vision increase the chances of survival. We can see also that the population tends to reach a stable size and spatial configuration.

Figure 14.7 shows the carrying capacity of the sugarscape—that is, what population size the sugarscape can support—as a function of the maximum level of vision and metabolism of the agents. For each level of vision and metabolism, the average value of ten simulations of six runs each is presented. We can observe how a larger vision and a lower metabolism tend to increase the carrying capacity of the sugarscape.

Figure 7: Carrying capacity



6 Experiments

A simple experiment would be to add moving cost proportional to the distance moved. This tends to slow down the convergence to the hilltop locations.

The Sugarscape model can also be extended in a number of ways so that many experiments of increasing complexity can be performed. A first step in that direction would be to replace the Sugarscape rule G_∞ used earlier with the following:

Sugarscape grow-back rule G_1: At each lattice position, sugar grows back at a rate of α units per time interval up to the capacity at that position.

To introduce this rule, you may want to start by transforming the sugarscape matrix s into a structure with two fields, one containing the capacity and the other the current level of sugar. Then you can check how different grow-back rules affect the results.

You might also try to work with agents with finite lives, where their maximum age is a random integer drawn from a given interval [a,b]. Then, you might introduce an agent replacement rule such as the following:

Agent replacement rule $R_{[a,b]}$: When an agent dies it is replaced by an agent of age 0 having random genetic attributes, random position on the sugarscape, random initial endowment, and a maximum age randomly selected from the range [a,b].

Epstein and Axtell (1996) present a number of rules for pollution formation, agent mating, agent inheritance, trade, credit, and so on, that can be implemented in the Sugarscape model. To learn about the specifics of these rules refer to their book.

7 Further Reading

For a comprehensive presentation of the Sugarscape model see Epstein and Axtell (1996). See also the web page of the Sugarscape model at the Brookings Institution at `www.brook.edu/es/dynamics/sugarscape/default.htm`. For an approach to estimating agent-based models see Gilli and Winker (2003).

For a survey on agent-based modeling and an application to finance see LeBaron (2006), which appeared in Judd and Tesfatsion (2006), containing many state-of-the-art papers on agent-based modeling. For a comprehensive site with resources on agent-based computational economics, see the web site developed by Leigh Tesfatsion at `www.econ.iastate.edu/tesfatsi/ace.htm`. For a review of agent-based modeling as an approach to economic theory, see Tesfatsion (2006).

References

Epstein, J. and Axtell, R.: 1996, *Growing Artificial Societies: Social Science from the Bottom Up*, MIT Press, Cambridge, Massachusetts, USA.

Gilli, M. and Winker, P.: 2003, A global optimization heuristic for estimating agent based models, *Computational Statistics and Data Analysis* **42**, 299–312.

Judd, K. L. and Tesfatsion, L.: 2006, *Agent-Based Computational Economics*, Vol. 2 of *Handbook of Computational Economics*, North Holland, Amsterdam, the Netherlands.

LeBaron, B.: 2006, Agent based computational finance, *in* K. Judd and L. Tesfatsion (eds), *Handbook of Computational Economics*, Vol. 2 of *Handbook in Economics*, North Holland, Amsterdam, the Netherlands, pp. 1187–1232.

Tesfatsion, L.: 2006, Ace: A constructive approach to economic theory, *in* K. L. Judd and L. Tesfatsion (eds), *Handbook of Computational Economics*, Vol. 2 of *Handbook in Economics*, North Holland, Amsterdam, the Netherlands, pp. 1187–1232.