Planting a Tree: Agent-based Model Project

Gabriel Vincenzi

Abstract

The paper serves as a companion manual for the Matlab project: `ABMPlanting` for the course in Computational Financial Lab at Ca' Foscari University of Venice. The code aims to create a controlled environment where agents can thrive following different rules of movement, resources and behavior to study the impact of different parameters on each agent's final characteristics, on the corresponding final environment and distribution of wealth.

1. Introduction to the Code

The following code has been created to study the interactions between agents in a simple environment which grows in complexity as time passes. The agents have initial different parameters creating an heterogeneous starting population and they have the possibility of choosing to improve their environment or deplete its resources, they furthermore can update own probability of choosing regarding their surrounding environment following a Bayesian updating set-up. The model creates a controlled framework that allows to study the implications of different parameter changes and their cross-effects from the environmental and the wealth distribution perspective. The model concludes that, for example, given these set of possible moves and parameters, a population that starts with a smaller resource usage coefficient, so it is more prone to leave resources to future agents, on one hand changes the environment increasing the resources, as one could imagine, and on the other hand creates a population distribution that is, in mean behavior, more right skewed but wealthier in levels than a population that depletes all its resources while having a much more egalitarian distribution.

The folder `FinalProject` contains the functions and classes that allow to run smoothly the Matlab application `ABMPlanting.mlapp`. In particular we could divide them in families: the init functions, used to initialize the agents, the environment and update it (`initcircularsugar.m`, `initagents.m`, `updates.m`), then the display functions, which allow to visualize the agent's location in the environment and the distribution of wealth (`dispagentloc.m`, `dispwealth.m`) and the util functions that allow the agents to observe, move and behave given the environment (`observe.m`, `locationcheck.m`, `observenew.m`, `moveagent.m`).

1.1 Agent Class

The file `Agent.m` contains the individual unit of the model. It is a class object with multiple properties and methods resembling a simple behavior in an equally simple environment. The agent has own `metabolism` and `vision` that are random proportions of the maximum level possible (here we could have used a random selection from integers but would have been more time consuming

and less efficient), then from the vision property another one is created (`visionGrid`), that constitutes the baseline for future comparison between the present environment and the future one. Furthermore the agent has informations about its own `position`, the whole `wealthHistory` initialized at zero but afterwards overwritten and on a condition called `active` that represents if the agent is alive or not, initialized at one. The agent is also able to update its own choice probabilities given the parameters `alpha` and `beta` of a Beta distribution, following a Bayesian approach of updating starting from a neutral `probPlanting = 0.5`. The remaining functions allow the agent to update the wealth history (`updateWealth`), to get the last wealth value (`getLastWealth`) and to update own probabilities (`updatePlantingProb`) given the outcomes of its surrounding environmemt.

1.2 Main Program

The main App function of `ABMPlanting` starts after the user inserted all the relevant informations to build the model and pressed the "start model" button, such as: the number of agents, the grid size, the maximum parameters for vision and metabolism which regulate the maximum observable cells and the amount of resources to consume in order to stay alive, the maximum amount of resources in the peaks of the environment, an accumulation rule which governs the replenishing of resources, a steepness parameter to have higher or lower valleys or peaks in the environment, a parameter for resource usage that governs the rule of a "planting a tree" model, and the number of runs. Before the loop begins the three main objects are initialized: the environmental grid (`s`), the agent's grid (`grid`) and the `agents` object containing all the agents of the model. Then the loop starts and goes as follows:

1. Display on the plot of the environment the agents' position (`dispagentloc`) and in another graph represent the wealth distribution (`dispwealth`), then update the environment following the accumulation rule (`updates`).

2. Select randomly an agent from the `agents` object that is still alive (`active = 1`) and start the inner loop initializing a temporary set of variables, that will contain informations of the until-now best choice for each agent, at the current positions and resource levels.

3. For each `vision` level of the agent search via the function `observe` in the available grid space for better free locations than the present one, while storing the informations of the surrounding environment in the property `visionGrid` of the agent. If the algorithm found a better place, the agent moves to it.

4. Observe the new surrounding environment and create a new object that stores these informations in the same way as before.

5. Given the previous and present informations the agent will decide to "plant a tree" if the model was build with this functionality. The choice starts as a random one, from a $Beta(1,1)$ which is equal to say $Ber(0.5)$ or a $Bin(0.5,1)$, so a random choice, but after observing the difference between past and present surroundings the agent will update its beliefs following a Bayesian updating rule based on the number of spaces with higher `maxcapacity` than before.

2. Families of Functions

2.1 Init Functions

In this cluster of functions we find the ones that initialize agents, environment and update the latter. The function `initsugarscape` accepts a parameter for the `size` of the environment grid, one for the maximum sugar in the latter, `maxsugar`, even if after the calculations this number is smaller than the one given, but serves as an initial upper bound and is not significant for the individual behaviors, and a parameter `beta` that governs the steepness of each resource peak, higher the parameter and steeper the distribution of resources. After initializing the environment (`s`) as a structure object and found the center we can create the whole environment with two radial peaks that then fall to create a valley, all calculated via a fine-tuned function to fill each cell with its maximum capacity, then the current level is initialized as the maximum one.

The function `updates` instead accepts the environment grid `s`, the `size` value and the parameter `alpha` that represents the pace at which resources grow back. The choice here was to grow resources as a constant proportion of the maximum capacity until the latter is reached.

While function `initagents` instead works on the initialization of a set of `Agent` class objects. It creates different agents following the number of agents variable (`numAgents`) and the primitives of the model such as maximum vision and metabolism, putting them in different and non-overlapping positions on a grid. At each random position an agent is created filling the properties of the `Agent` class, in particular giving an initial wealth equal to the amount of current resources present on the grid position and starting the probabilities from a $Beta(1, 1)$ that is equal to a $Ber(0.5)$. The randomness and non-overlapping characteristics are determined by the creation of all possible positions in the grid and then indexing them by taking a random permutation of length equal to the number of agents.

2.2 Display Functions

The next functions are used to display both the agents' positions and the changing environment, and the wealth distribution as time passes. The function `dispagentloc` takes as inputs the environment, the grid and their size to create a map given a predetermined color palette that resembles earth and grass where we can see each agents' position from the function `spy()` that displays a binary grid. The color bar is limited to 10 in order to appreciate the possible heterogeneity at lower wealth levels according to different planting behaviors.

In addition the `dispwealth` function displays the last wealth values of each active agent as a frequency histogram given the informations about the last wealth value.

2.3 Utils Functions

These are the core functions of the model, which allow the agents to move and behave following simple rules that create complex aggregate responses. The function `observe` checks the level of sugar in each direction based on vision (`k`) and returns the coordinates of the best location for the agent to move given the starting inputs of the function which comprehend: the current agent,

the current vision level `k`, both environmental grids `s` and `grid` with its size and the temporary variables that represent the cell and position of a cell in the environment initialized outside the function as the current positions (`temps`, `tempi`, `tempj`). In it we define a cell object `directions` which is composed by a cell for each cardinal direction, each of the latter in turn is composed by values that represent in order: the maximum grid cells that an agent can observe, the limiting size of the map, row and column directions if the vision surpasses the borders, row and column directions if vision does not surpass the borders. This is all done because, in the eyes of agents, the map is a Torus, in this sense it is never-ending, an agent can move from north to south and from east to west and vice-versa passing through the borders. After a long stream of trials and errors I found something similar but overwhelmingly slow and with a high number of agents the whole code took a lot to complete each iterations, so I preferred to use a script found on a page of Stack Overflow which is much faster. In order to master its meaning I then used variations of it in other parts of the Matlab App. So for each direction the algorithm searches for a grid cell that follows the rule inside the function `locationcheck` that is explained below. The same function is then used to create a first iteration of the `visionGrid` property of each agent by storing the informations of the present maximum capacity of the surrounding grid cells following the same ratio of the search code before.

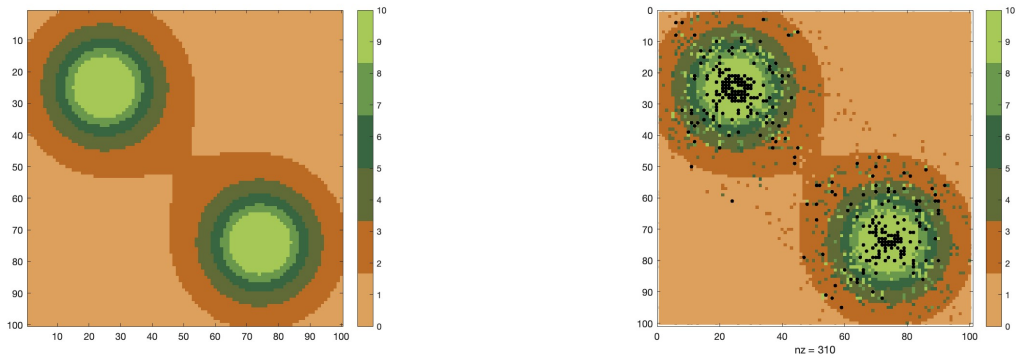
To continue, the function `locationcheck` checks first of all if the currently observed location is free, then if it has more current resources than the one in which the agent is located, if so the temporary variables become this newly discovered best choice. It follows that on next iteration the choice is going to be between the last highest resourceful cell and the newly observed one.

While the function `observenew` is a simplified version of `observe`, which creates a new grid that stores informations about the current environment where the agent has moved to, in particular the max capacity of the surrounding cells.

The function `moveagent` instead allows each agent to effectively locate themselves to the best location found via the `observe` function or to stay in the current cell, it updates current wealth of each agent and the current available resources of the environment following different according to the input `planting` that changes completely the behavior of agents. This function accepts as inputs the agent, environmental grids, newly found highest resourceful cell indexed by the temporary variables, then a boolean for the so called "planting" variant of the model and a parameter (`resourceusage`) that governs the rule of accumulation and depletion of resources. If the agent has found a better location to move, its internal position and the one on the grid are updated, then is time for the wealth and environment update. If the user chose to build a model with `planting = true`, then a random extraction from a $Bin(p, 1)$ where p is the internal probability of planting a tree. If the extracted value is a 1, coded as the boolean *true*, the agent receives not all the current level of resources in the cell, but a proportion given by the `resourceusage` parameter and its wealth is decreased by the amount of resourses that it needs to survive (`metabolism`. The remaining resources are then used to increase the max capacity of its cell as if it decided to leave a better environment than the one it founded by "planting a tree". So the current level is fully used and goes to zero. Otherwise, the agent receives as wealth all the current resources depleting fully the environment but causing the decrease in the max capacity

of the latter, in a sense an egoistic behavior. If instead the user built a model with `planting = false`, a standard rule is implemented and if the agent moves it gains as wealth all the current level or resources without influencing the environment. If the agent stays in the same position we exploit the fact that with `temps = s` and the rules are the same as before but we assume that, given that it is going to stay in the same position, it would not deplete and decrease the resources of its cell, so it is surely going to "plant a tree". In conclusion the function checks if the agent's wealth (from the `lastWealth` property) is negative, in the latter case the agent dies, freeing the grid, this translates in the parameter `active = 0`.

3. Figures



(a) Starting environment

(b) Environment after 16 iterations

Figure 1: Environment change with a `resourceusage = 0.2`