Hash and Beans Economy (ABM)

December 7, 2016

Contents

1 Introduction

This is the documentation of a project that originated from my attendance at the lecture Complexity and Behavioral Macroeconomics at Ruhr Universität Bochum held by the chair for macroeconomics. Based on the model outlined in Leigh Tesfatsion (2006), I want to crate variants (simplifications and extensions of this simple two sector economy) in order to examine, what a model needs to feature in order to explain a given stylized macroeconomic fact. Sometimes this question can be answered a priori: A philips curve for example can only emerge in a model featuring a labor market. To put it differently, if the computer program does not feature a labor market, then one can not hope to find an inverse relation between inflation and unemployment, as unemployment simply can not be defined in this model. But certainly one can think of constellations where answering the above question becomes much more difficult. In order to keep track of the different variants of the model I created the Git Hub Repository der-spaete-jo/HashBeansEconomy. The more fundamental but less interesting purpose is for me to get familiar with agent-based modeling (ABM).

ABM papers often focus on the economic interpretation of some computer program modeling an economy, while the actual coding remains obscure. On the other hand writing a standard documentation for a computer program is not sufficient to capture an agent-based economic model. Therefore in this documentation both extremes are combined. You will find mathematical equations describing the agents behavior as well as the economic interpretation for those equations. But you will also find detailed descriptions of all classes with their attributes and methods. The crunchpoint of course lies in finding a common ground for these two (different) views on the same problem.

This documentation is structured as follows: Section 2 covers all preliminaries, including the theoretical background but also concrete programs that are needed in order to run the model on a computer. There is also a small subsection devoted to the role of Git Hub for this project, as it is strongly connected to the purpose of this project. In section 3, firstly the Hash and Beans Economy as constructed in Leigh Tesfatsion (2005) will be revisited (with a major simplification concerning firm pricing behavior). Afterwards the realization of this model as a computer program will be discussed. Section 4 then goes into the details of all the points on the ToDo-list, which the reader may find in the appendix. A sample run of the model is analyzed and interpreted in section 5.

2 Preliminaries

2.1 Agent-based Computational Economics

Standard macroeconomic models are comprised of a system of equations together with a condition for this system to be in steady-state (equilibrium). Each equation describes either the decision of an agent (a sector) or the coordination between two agents (sectors). In a steady-state, the endogenous variables (de-

scribed by the equations) do not change. Given that the equilibrium exists and is unique, one can analyze the model using comparative statics. But as shown by Kirman (199?), in an aggregate context uniqueness of equilibria requires implausible assumptions concerning agents preferences. This problem was addressed (but not solved) by the representative agent framework: A whole sector of the economy is modeled as if it was only one agent (the mean consumer, the mean firm, ...). Now standard assumptions allow to deduce existence and uniqueness of equilibria, but they are again implausible because a sector does not behave like one individual.

2.2 Leight Tesfatsion: A Constructive Approach

2.3 Dependencies

2.4 Git Hub

There exists a Git Hub repository for this project: der-spaete-jo/HashBeansEconomy. It is organized into different branches that belong to one of the following categories:
1) The master branch in its outstanding uniqueness constitutes his own type of branches. It contains those files that are needed in every variant of the model.
2) Branches that have names starting with evolve_ contain newer versions of certain files from any branch. Those branches only exist temporarily: If the newer version of the file has been tested and workes, the branch will be merged into another branch, to update it with the changes. Example: evolve_OutputHandler.
3) All other branches represent variants of the model and contain all the files this model variant is comprised of. Example: adaptive_pricing.

3 Examining the Hash and Beans Economy

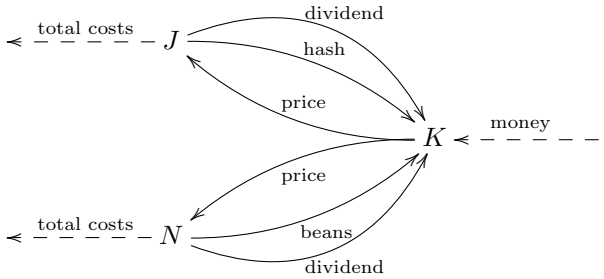
3.1 The Economic Model

This discussion corresponds to pages 15–17 of Tesfatsion (2006)

3.1.1 The Framework

The Hash and Beans Economy (HBE) is a system involving three types of agents: consumers, hash firms and bean firms. The agents of each type form a sector, denoted by K, J and N respectively. Those sectors are connected via the interaction between agents: consumers buy hash from hash firms, and hash firms receive the price for hash in return. Moreover if a hash firm earned a positive profit it declares a dividend. The following diagram 3.1.1 shows all

sectors and their connections (arrows between sectors).

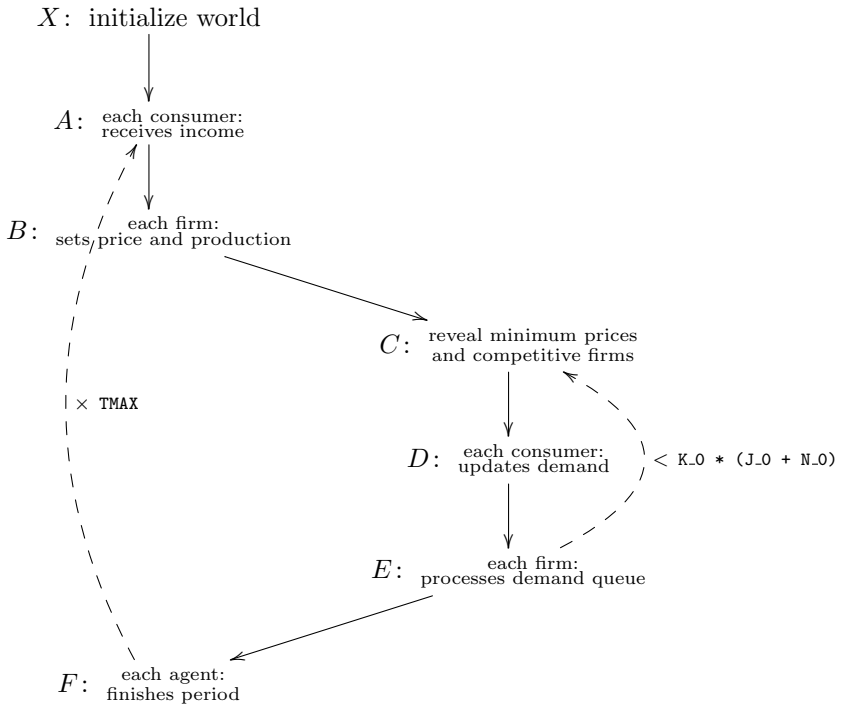


A pair of arrows in reverse directions ranging between two sectors, one titled with "price" and the other titled with the name of some good, corresponds to a market. The HBE is comprised of two markets: a hash market $J \rightleftarrows K$ and a bean market $N \rightleftarrows K$. An arrow without a "partner" going in the different direction, might be a subsidy or a tax. In this case it is a dividend which is distributed among consumers as if each would hold an equal share of equity of all firms. Some arrows are dashed to indicate that they leave the or start outside the model world, i.e. a sector A has a connection to some sector B, although B is not part of the model economy.[1] In the HBE consumers are endowed with a random amount of money each period in addition to the dividend they receive from profitable firms. Firms however have to pay costs for producing goods (depending on their production level, their size measured in capacity units and 5 parameters) and this money is "stolen" from the economy. Moreover firms trade any amount of capacities at a constant price, but the sector which supplies the capacities is not modeled.

Sector K is populated by homogeneous agents seeking at first to fulfill their subsistence needs for hash and beans and then to maximize utility.

[1] Note that dashed arrows do not correspond to initial parameters in a 1-to-1 fashion. They are indeed given exogenously, but most initial parameters are set once, where arrows in the diagram correspond to a repeated flow of goods or money (like in a market). This is also the reason why the money endowment of each consumer is a dashed arrow: There is some really benevolent sector out there subsidizing consumers each period, but this sector is not featured in the HBE. In fact there are different ways to turn this dashed arrow into a normal (endogenous) arrow, e.g. a labor market or a government sector.

3.1.2 The Sequence of Events



3.2 The Implementation

The program is organized in so called python modules. Those are basically files with the file extension .py. More precisely the program adaptive_pricing is split into the two files simplest_two_sector_economy.py and OutputHandler.py.[2] In the latter file only one class called Historian is defined. This class provides all methods needed to capture the data which is generated by the model economy.

A model economy in the sense of a computer program is very similar to the code that implements a computer game. Concerning the program code of a game one would expect to find at least three classes (four if there is something like a gaming board involved[3]). One of them (`Rules`) laying out the rules and implementing methods to check the move of a player with respect to those rules. Another class (`Player`) defines player objects, that need to have names, maybe an accumulated score, surely a method that enables the player to make a move (`makeMove()`). Last but not least a class that rules the entire game (`GameController`): instantiating all the players and then starting the game loop. This game loop is a method of the class GameController and it defines

[2]Looking at the purpose of this project it might be favorable to further split the first file, s.t. one can evolve and update single agents.

[3]five if one or both of the players are real humans: then there is need for some `InputOutputHandler`-class.

the sequence of events, which is of course different for each game, but a call like `player_object.makeMove()` will surely be involved in this game loop.

In order to stay in this analogy, we will forget about the OutputHandler respectively the Historian class and focus on the classes defined in simplest_two_sector_economy.

3.2.1 The simplest_two_sector_economy API

The attributes of all the classes can be assigned to the categories: parameters, intern calculations, choice variables and code organization. Parameters are attributes, that are set by the constructor of the class and generally do not change during the economy loop. Attributes of the category intern calculations are containers to save intermediate results of calculations. Most of them do have economic interpretations but are not considered to be interesting for a detailed analysis (e.g. stocks in a perishable context). Otherwise the attribute would be considered a choice variable, which is also the result of a calculation performed by the agent and is of high interest concerning a macroeconomic analysis (e.g. price offers by the firms). The last category pools attributes with no meaningful economic interpretation. They exist to enable certain structures or links between classes or just to enable unambiguous identification of objects.
A getter method (setter method) is a method, which only returns the (assigns a new) value of (to) a certain attribute. For most of the attributes of a given class there exists both a getter and a setter method, but as they always do the same thing they will be missing in the discussion of the different classes. Note however that their naming follows a rule: if the attribute is named `self.part1_part2`, then methods are named `getPart1Part2()` resp. `setPart1Part2()`. Methods that change the value of one or more attributes, in a more sophisticated way than just setting it to a new value like a setter method, will be called `updateSomething()`.[4]
All classes implement a `__str__` method just to get meaningful console output and a `__repr__` method, which may be useful later[5].

`class simplest_two_sector_economy.BeanFirm(self, Money_0, Cap_0, f, F, S, R, m, d, p_B)`

> Represents a firm agent in the bean sector. Implements no new methods on top of the normal firm methods. Strictly speaking one could leave this subclass away and save the missing information (about the product and the price for capacities) in attributes of the firm class. But in this way the coding reflects the economic reality of two distinct sectors.
> **Inherits from:** `Firm`
> **Attributes:** `self.product`, `self.Cap_unit_price`

[4] Actually I have to admit that I did not follow those rules close enough.

[5] `c.repr()` returns a string containing all attribute values (initial parameters) that were used to instantiate the agent `c`. This information can be used to create a new agent which is equivalent to `c` w.r.t initial parameters. See Focus: Death and Market Imperfections for a motivation.

`class simplest_two_sector_economy.Consumer(self, h_sn, b_sn, a, Endow)`

Represents a consumer agent
Inherits from: -
Attributes
Program organization: `name`, `historian`
Parameters: `h_sn`, `b_sn`, `a`, `Endow`
Choice variables: `h_d`, `b_d`
Intern calculations: `Exp`, `Inc`, `Sav`, `h`, `b`
Methods
`updateShoppingList(self, min_p_h, min_p_b)`

Takes a pair of prices for hash and beans (floats) and solves the household problem given the utility function the consumer is endowed with and the budget constraint derived from the pair of prices and the residual income. Actually it does this for any pair of prices but in the code it is only called on the currently lowest prices in the two markets. The method returns nothing but sets both `self.h_d` and `self.b_d` to a new (optimal) value.

`makePurchase(self, product, quantity, price)`

Takes a string containing the name of a product, and two floats. Returns nothing but updates `self.h` resp. `self.b` and adds to `self.Exp` the product of `quantity` times `price`.

`finishPeriod(self)`

This method updates the savings of the consumer. Then it reports the values of `self.Exp`,`self.Sav`, `self.h` and `self.b` (`self.Inc` is reported right at the beginning of the period) so that they can be stored for later analysis (the consumer however forgets those values). Afterwards the amount of hash and bean owned by the consumer and his expenditures are reset to 0. Income and savings are overridden in the next period, so no need to reset them.

`class simplest_two_sector_economy.EconomyController(t_max, K_0, J_0, N_0, p_H, p_B, init_consumer_data, init_firm_data)`

Abstract class that instantiates the world and rules its evolution. All attributes can be considered to belong to program organization, since the class itself has no interpretation in the real world.
Inherits from: -
Attributes: `t_max`, `K`, `J`, `N`, `K_0`, `J_0`, `N_0`, `p_H`, `p_B`, `init_consumer_data`, `init_hashfirm_data`, `init_beanfirm_data`, `Div_per_capita`, `consumer_id`, `hash_firm_id`, `bean_firm_id`, `historian`, `period`
Methods
`initializeAgents(self)`

Invoked by the constructor. Instantiates `K_0` Consumer-, `J_0` HashFirm- and `N_0` BeanFirm-objects and invokes `self.registerAgent(agent)` for each of them which are saved in lists `K`, `J` and `N` respectively.

`registerAgent(self, agent)`

Takes a Consumer-, HashFirm- or BeanFirm-object and performs different steps of wiring the object with the world. The agent gets a name (more a serial number), it is saved in a list `K`, `J` and `N` respectively and it gets a connection to the OutputHandler.

`unregisterAgent(self, agent)`

Takes a Consumer-, HashFirm- or BeanFirm-object and deletes it after cutting all wires of this agent with the rest of the world[6]. In this simple setting, there is only one object referencing any given agent: the `EconomyController`. Therefore it suffices to remove the agent from the respective list `self.K`, `N` or `J`.

`run(self)`

This is the core piece of the whole code. It implements the sequence of events by invoking the right methods at the right point in time. Staying in the notation of the diagram 3.1.2:
A: `Consumer.updateInc(T, self.Div_per_capita)`
B: `Firm.updateSupplyOffer()`
C–E: `self.priceDiscoveryProcess(hash_supply, bean_supply, solvent_consumers, stocked_firms, T, k)`
F: `Agent.finishPeriod()`.

`priceDiscoveryProcess(self, hash_supply, bean_supply, solvent_consumers, stocked_firms, T, k)`

C: calculate the minimum of all offered unit prices and form a list containing all firms supplying at this price. Both is done right in place.
D: `Consumer.updateShoppingList(min_p_h, min_p_b)`
E: `Firm.processDemandQueue()`

`class simplest_two_sector_economy.Firm(self, Money_0, Cap_0, f, F, S, R, m, d)`

[6] It is important for the consistency of the model, that all wires are cut. Otherwise the agent would still be queried by the `EconomyController` to do stuff, although it is dead/bankrupt. It gains further importance, if one considers a setting where the model should be ran many times with a high number of agents, thus efficiency (how much time and memory space is needed to perform a given task) matters. Under the condition that no other object is referencing the given agent anymore ("all wires are cut", which is equivalent to "reference count = 1"), the `del` keyword reduces the reference count for the agent to zero. Then pythons routine to remove objects (the garbage collector) will remove it and - this is the point - free the memory space, that was used by the agent.

Top level class used as base for hash and bean firm.
Inherits from: -
Attributes
Program organization: `name`, `historian`
Parameters: `Money_0`, `Cap_0`, `f`, `F`, `S`, `R`, `m`, `d`
Choice variables: `production`, `unit_price`
Intern calculations: `Exp`, `Inc`, `Sav`, `h`, `b`
Methods
`updateSupplyOffer(self)`

calculates values for `self.unit_price` and `self.production` based on very simple rule discussed above. Initializes `self.stock`, i.e. sets its value to `self.production`. Reports a bunch of data.

`processDemandQueue(self)`

rationing/no rationing cite Tesfatsion

`funcTCosts(self, Q)`

returns the total cost of producing `Q` units of the good.

`allocateProfit(self)`

This implements the profit allocation method. It is an exact translation of Tesfatsion mathematical definition[7] to python.

`finishPeriod(self)`

save Cap for next period, calculate profit, report values, allocate profits via a call of `self.allocateProfit()`, reset profit and inventories (`self.profit=self.stock=0`) and update FCosts and NetWorth.[8]

`class simplest_two_sector_economy.HashFirm(self, Money_0, Cap_0, f, F, S, R, m, d, p_H)`

Represents a firm agent in the hash sector. Implements no new methods on top of the normal firm methods.
Inherits from: `Firm`
Attributes: `self.product`, `self.Cap_unit_price`

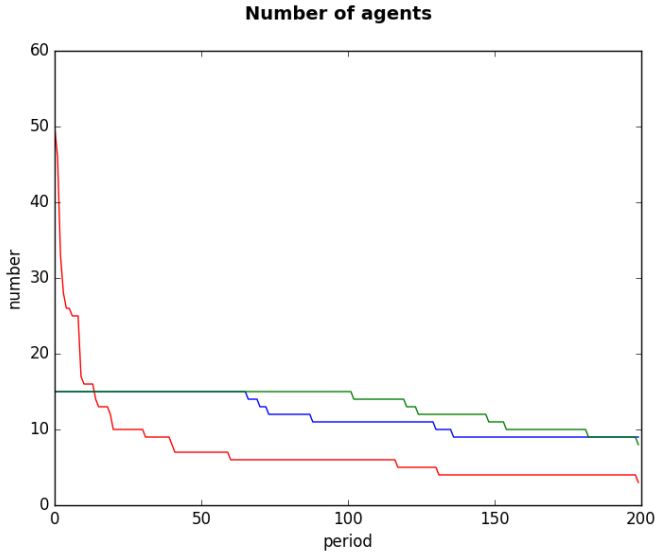
`function simplest_two_sector_economy.createAgent(self, data)`

under construction... (a function that allows you to instantiate an agent based on a `__repr__` string).

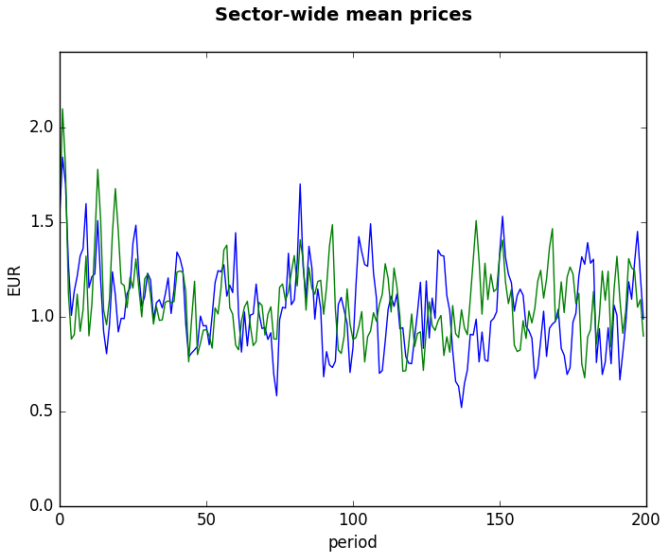
[7] See Appendix A.3 in Tesfation(2006)
[8] This method includes the activity flow for firms as defined in Appendix A.2 (Tesfatsion, 2006).

Besides the definition of the above classes, there are only two lines of code in the file, namely the instantiation of one `EconomyController` object and the call to the `run()`-method of this object.

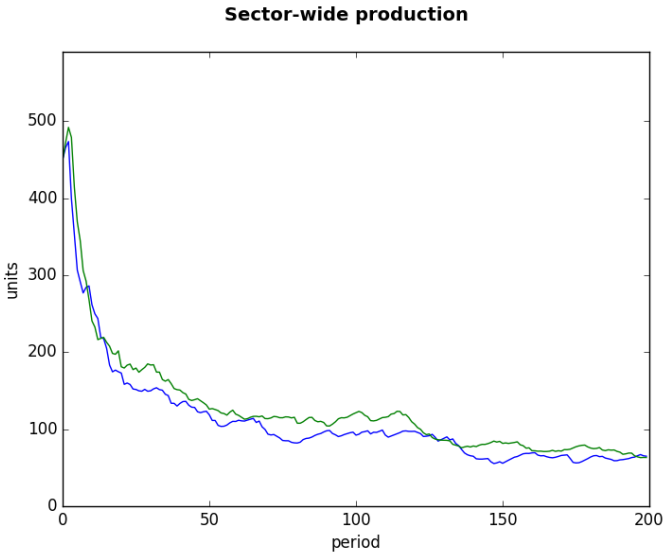
4 Sample Run: HashBeansEconomy20161124181546



shows the evolution of the number of consumers (red), hash firms (blue) and bean firms (green).



Mean price evolution in the hash market (blue) and the bean market (green). This is just the arithmetic average over all prices of all firms in one sector, they are not weighted e.g. with the firms share in total production.



Production time series in the hash market (blue) and the bean market (green).

5 Focus

5.1 Fixed!

5.1.1 Keep track of the data, generated by the program

There are some methods of classes that report values. This means that they call `self.historian.reportValue(agent_reporting,...)`, where `self.historian` is the instance of the `OutputHandler`-class generated by the `EconomyController`-object at the beginning of the program. This implies that both `Consumer`- and `Firm`-objects could in theory call all methods of the `OutputHandler`. In particular they could get the values of certain variables in the periods past. But in the end the programmer decides if agents exploit this error. Concerning the simplest_hash_and_bean_economy agents only call the method `reportValue(...)`, so they don't try to access information that they should not have. But (and this is the point here) in theory one could build a model economy using the `OutputHandler` and give agents full information about periods past.
Design and idea how the historian class gathers/collects data and how this data is saved. (Workflow? how to read output?)
The OutputHandler API

5.2 Extend?

5.2.1 Measuring Demand

Consumers change their demand in each price discovery round, therefore summing up the demand of all consumers for each period is not well-defined. Restricting to the problem of aggregate hash demand, let $E_k = \{(h^d_{k,t,s})|1 \leq t \leq$

TMAX, $1 \leq s \leq R(t)\}$ be the set containing consumer k's demand for hash in each price discovery round of each period, where $R(t)$ is the number of price discovery rounds that were played in period t. Furthermore let

$$\Sigma_{t_0} : E_1 \times ... \times E_K \to \mathbb{R}^+$$
$$(h^d_{1,t_0,s}, ..., h^d_{K,t_0,s}) \mapsto h^d_{1,t_0,s} + ... + h^d_{K,t_0,s}$$

be the function that sends a K-Tuple containing the hash demand in period t_0 round s of all K consumers to their sum. Forgetting about the s, this would be the naive aggregator function. Since in this economy it might (actually very likely) happen that some consumers already have their demand satisfied and therefore left the market place ($h^d_{k,t_0,s} = 0$ for all $s \geq \bar{s}$, for some k), while others still try to buy something ($h^d_{\tilde{k},t_0,s} = 0$ for some $s \geq \bar{s}$, for some $\tilde{k}$), it is obvious that the choice of s matters. More precisely s should be equal for all components of a hash demand vector, in order to get a meaningful sum of the components. So we are looking for a function Σ_{t_0,s_0}, but there is no natural choice for s_0. If one would choose $s_0 = 1$ for example, one would only get the quantity demanded at the lowest price offered by the firms. The point is that the function Σ_{t_0} takes different values for different choices of s, therefore without such a choice Σ_{t_0} is not well-defined.
But maybe there is a very easy fix. In a model where demand changes only once each period, the function Σ_{t_0} would called once for each $t_0 \in [1, \texttt{TMAX}]$. Now we could call Σ_{t_0,s_0} $R(t)$-times for each $t \in [1, \texttt{TMAX}]$. Holding t fix this could be interpreted as a demand curve in the following way: The aggregate demand in period t $D_t(p) = \Sigma_{t,\tilde{s}}$, if $\tilde{s}$ is the price discovery round, where p was the currently minimum unit price available in the market. But this is not 100% since in round $\tilde{s}$ some consumers might already have left the price discovery process although they would have a positive demand (in a gedankenexperiment where they have their whole income available). Note also that this aggregate demand function is only defined for those prices that actually ocure as minimum prices during the price discovery process. So I don't think that this is the fix for measuring aggregate demand.

5.2.2 Price Volatility, Firm Learning

The HBEcon is actually a simplification of the model in the paper, since firms don't learn. Instead they a the simple pricing rule from Axelrod (2008?). Because of my choices for the parameters involved in price setting, firms over-react (See mean price evolution... it's a mess).

5.2.3 Death and Market Imperfections

The model allows for agents to die: If a consumer does not manage to fulfill his substantial need for hash and beans through the market interactions, he will die at the end of the period. If a firms net worth becomes negative, the firm will be considered bankrupt and leaves the market. On the other hand, there is

no way, that new consumers are born and enter the market or, that new firms are founded and enter the market. Focus for the moment on the consumers (form a management point of view, it is the death of consumers that matters). Assuming that for every run of the model there is some lower bound $d \geq \underline{d} > 0$ for the death rate $d_t = G_t/K_t$ (where G_t is the number of deaths in period t and K_t is the number of consumers in period t), one can say that this model has the intrinsic property to converge to an equilibrium. This equilibrium could be called the trivial or zero equilibrium as it is characterized by the number of consumers being 0. Soon after that, also the number of firms will be zero, and then also the mean prices and production levels are zero, and so on. While this is indeed a very stable equilibrium, it is not really interesting, nothing special happens in this equilibrium. Even worse: As soon as the model reached this equilibrium it will stop to explain things.So the question is how to cope with this property of the model: What influences the death rate d_t? Can one manage $G_t = 0 \ \forall t \geq \bar{t}$?

Market imperfections essentially are/lead to (?) excess demand. Note that there is a direct link between excess demand X_t in period t and the number of deaths, namely $X_t = 0 \Rightarrow G_t = 0$, in words: if there is no excess demand then all consumers will survive. No thats not true: if Consumer k has not enough income to buy his substantial need, he will not be count as excess demand.

There are three possible reasons for G_t to be greater than zero in any given period t:

1) Firms have not produced enough hash (or beans) to supply all substantial needs: $P_t^H < \sum_{k=1}^{K} h_{sn,i}$, where P_t^H is aggregate production of hash in t and $h_{sn,i}$ is consumer i-th substantial need for hash. (classical excess demand situation)

2) Firms have produced enough, but the income of some consumer is not sufficient to buy an amount of hash (and beans) equal to his substantial needs. Suppose consumer k was rationed in the price discovery rounds 1 to s_0 (i.e. he may have recieved some amounts of hash and beans, but not yet enough). If in round $s_0 + 1$ prices for hash and beans are such, that consumer k leaves the price discovery process, he will die.

In case firms have produced enough and income of all consumers is sufficient[9], then there is no possibility that a consumer dies this period. This is due to the construction of the price discovery process. It will stop if either all firms are stocked out or no consumer is solvent anymore[10]. Since it is assumed that firms have produced enough the loop will not break because of stocked out firms. Therefore the loop breaks because of insolvent consumers. But consumers are assumed to have sufficient income, therefore the loop comes to an end because

[9] A consumer is said to have sufficient income if given the highest possible price in both the hash and the bean market, his money would still buy him an amount of hash and beans greater or equal to his substantial needs

[10] A consumer leaves the group of solvent consumers, if (a) he has was not rationed or (b) given the currently lowest prices for hash and beans, his income is not sufficient to fulfill his substantial needs. See the while-loop in `EconomyController.run()` and the evolution of `solvent_consumers` in `EconomyController.priceDiscoveryProcess()`

all consumers have experienced non-rationing once in each sector.

5.2.4 Labor Market

Consumers get endogenously determined income
Use the same market mechanism as in the goods markets, but with roles changed?
Not very realistic

5.2.5 Capacities Market

This is a ranging question!
communist organized distribution of farmland vs auctions vs Pacht vs ...
With a market for capacities, which is capital of course, the model would change fundamentally, because one normally does not finance investments with cash flow. It is questionable if the model could ever explain long run growth processes or capital concentration problems with this dying population.

5.2.6 Hash Firms Entering the Market for Beans

cattle needs to be fed
this leads to more power for the hash firms or the bean firms? actually a microeconomic question
Take the markets for meat and soy. A huge percentage of soy goes into the production of feed for cattle and the rest is consumed by humans. Since soy production heavily relies on burning down the jungle something needs to change here: If humans would consume soy directly rather than feed cattle with soy and then eat the meat, the need for agricultural land could be reduced. Hash firms entering the market for beans along with a more sophisticated market for capacities (= agricultural land) could provide the framework to analyze systems of taxes/subsidies that encourage sustainable use of agricultural land.

5.2.7 Vegetarians...?

Change the utility function to be non-differentiating w.r.t. the fulfillment of some substancial need... but one could say that hash is more effective in breaking the threshold. I think of something like $U(h,b) = \text{sign}(e_H h + e_B b - t) \cdot \text{Cobb-Douglas}(h,b)$, where e_i is some effectiveness measure, t is the substancial need for protein and $\text{Cobb-Douglas}(h,b) = h^\alpha b^{1-\alpha}$, for some $\alpha \in [0,1]$. A (voluntary) vegetarian would then be a consumer with $\alpha = 0$.

5.2.8 Price Index

Which ways are there to get to a reasonable definition of a consumer price index?
Which information is needed?
Idea 1:
For each consumer save the (weighted) mean price, concerning his purchases of

hash and beans: $\bar{p}^k = \sum_{q=1}^{Q^k} \frac{x_q}{h+b} \cdot p_q$, where $q = 1 \dots Q^k$ indexes the purchases[11] made by consumer k in period t, h and b are the total units of hash resp. beans purchased in period t, x_q is the amount purchased in the q-th transaction measured in units (independent of the product purchased) and p_q is the unit price in the q-th transaction. To be more precisely we can add more scripts to the variables:

$$\bar{p}^{k,t} = \sum_{q=1}^{Q^{k,t}} \frac{x_q^{k,t}}{h^{k,t} + b^{k,t}} \cdot p_q^t = \frac{1}{h^{k,t} + b^{k,t}} \sum_{q=1}^{Q^{k,t}} x_q^{k,t} \cdot p_q^t$$

and receive the per unit mean price consumer k faces in period t, which is of course dependent on the very preferences of consumer k, since the preferences influence $x_q^{k,t}$. The number $\bar{p}^{k,t}$ can be interpreted as follows: if you have to guess at which price consumer k was buying an arbitrarily chosen unit of hash or beans from all his purchases in period t, then guess $\bar{p}^{k,t}$; it will very likely be false, but it will be the best guess. Note that by construction: $\bar{p}^{k,t} \cdot (h^{k,t} + b^{k,t}) =$ `self.Exp`. We arrive at a consumer price index simply by calculating the arithmetic average over all individual per unit mean prices:

$$P_t = \frac{1}{K_t} \sum_{k=1}^{K_t} \bar{p}^{k,t} \tag{1}$$

5.2.9 Measure the Stock and Flow inconsistency

As discussed in previous sections, this model is not stock and flow consistent.

6 Appendix

[11] Each purchase corresponds to a call to `makePurchase(...)`

6.1 Translation table: docs/paper - code

Tesfatsion	Code	init_params	interpretation
Initial economy data			
$TMax$	`t_max`	`TMAX`	How many periods?
$J(0)$, $N(0)$, $K(0)$		`J_0`, `N_0`, `K_0`	Initial number of consumers, hash- and Bean-firms respectively
ρ_H, ρ_B		`CAPACITIES_UNIT_PRICE_HASH` resp. `..._BEAN`	Exogenously given prices for capacities
Initial firm data			
$\text{Money}_{H_j}(0)$, $\text{Money}_{B_n}(0)$		INIT_MONEY	start money of firms
$\text{Cap}_{H_j}(0)$, $\text{Cap}_{B_n}(0)$		INIT_CAP	initial capacities
$\bar{h}$		SUBSTANCIAL_NEED_HASH	min amount of hash consumers must buy each period
$\bar{b}$		SUBSTANCIAL_NEED_BEAN	min amount of bean a consumer needs to consume each period
Initial consumer data			