ABCE Documentation

Release 0.3.1alpha

Davoud Taghawi-Nejad

June 28, 2013

CONTENTS

ABCE is a Python Agent-Based Complete Economy Platform, written by Davoud Taghawi-Nejad. The impatient reader can jump directly to the 'walk through', which explains how to set up a simulation. In the walk through you learn how to set up an agent and how to trade with other agents. The households and firms classes allow you to produce with different production functions and consume with according production functions.

CHAPTER

ONE

INTRODUCTION

ABCE is a Python based modeling platform for economic simulations. For simulations of trade, production and consumption, ABCE comes with standard functions that implement these kinds of interactions and actions. The modeler only has to implement the logic and decisions of an agent; ABCE takes care of all exchange of goods and production and consumption.

One special feature of ABCE is that goods have the physical properties of goods in reality. In other words if agent A gives a good to agent B, then - unlike information - agent B receives the good and agent B does not have the good anymore. That means that agents can trade, produce or consume a good. The ownership and transformations (production or consumption) of goods are automatically handled by the platform.

ABCE has been designed for economic problems where spacial representation is not important or the spacial position of the agents is (largely) fixed. Therefore instead of representing the simulation graphically, ABCE collects data of the simulation and outputs it ready to use for R and Excel.

ABCE models are programmed in standard Python, stock functions of agents are inherited from archetype classes (Agent, Firm or Household). The only not-so-standard Python is that agents are executed in parallel by the Simulation class (in start.py).

ABCE allows the modeler to program agents as ordinary Python class-objects, but run the simulation on a multi-core/processor computer. It takes no effort or intervention from the modeller to run the simulation on a multi-core processor production, consumption, trade, communication and similar functions are automatically handled by the platform. The modeler only needs to instruct ABCE, which automatically executes the specific functions.

ABCE is a scheduler [1] and a set of agent classes. According to the schedule the simulation class calls - each sub-round - agents to execute some actions. Each agent executes these actions using some of the build-in functions, such as trade, production and consumption of ABCE. The agents can use the full set of commands of the Python general purpose language.

The audience of ABCE are economists that want to model agent-based models of trade and production. It is especially geared towards simulations that are similar to standard economic models like general or partial equilibrium models [2]. What is more ABCE is especially designed to make writing the simulation and the execution fast. Therefore models can be developed in an interlinked process of running and rewriting the simulation.

ABCE uses Python - a language that is especially beginner friendly, but also easy to learn for people who already know object oriented programming languages such as Java, C++ or even MATLAB. Python allows simple, but fully functional, programming for economists. What is more Python is readable even for non Python programmers.

[1] the Simulation class

[2] with out the equilibrium of course

CHAPTER

TWO

DESIGN

ABCE's first design goal is that, code can be rapidly written, to enable a modeler to quickly write down code and quickly explore different alternatives of a model.

Execution speed is a secondary concern to the goal of rapid development. Execution speed is achieved by making use of multiple-cores/processors.

Secondly, the modeler can concentrate on programming the behavior of the agents and the specification of goods, production and consumption function. The functions for economic simulations such as production, consumption, trade, communication are provided and automatically performed by the platform.

Python has also been chosen as a programming language, because of it's rich environment of standard libraries. Python for example comes with a stock representation of agents in a spacial world, which allow the modeler to model a spatial model although ABCE was not designed for spatial models.

Python is a language that lends itself to writing of code fast, because it has low overhead. In Python variables do not have to be declared, garbage does not have to be collected and classes have no boiler-plate code.

Python, is slower than Java or C, but its reputation for slow speed is usually exaggerated. Various packages for numerical calculations and optimization such as numpy and scipy offer the C like speed to numerical problems. Contrary to the common belief Python is not an interpreted language. Python is compiled to bytecode and than executed. What is more Python in combination with ZeroMQ allows us to parallelize the code and gain significant speed advantage over single-threaded code, that does not make use of the speed advantage of multi-core or multi-processor computers.

ABCE 0.3.1 supports Python 2.7, Python 3 support is planned for the 0.5 release.

One of the mayor impediments of speed is that the GIL - global interpreter lock - of Python forces us to use the processing module instead of the threading module. Using multi-threading would allow the usage of ZeroMQ's 'inproc' socket instead of the slower 'ipc' or 'tpc' sockets. Once a GIL free version compatible with ZeroMQ is available this speed break can readily be removed as the processing and the threading module have the same API.

In the current 3.0.1 version simulations are not entirely deterministic, the order of messages depends on which agent is called first. The call order of the agents by the virtual parallelization is not necessarily consistent. In other words, if your system has less cores than agents, not all agents do actually run in parallel. The parallelization is achieved by randomizing the message order. This leads to some agents sending the messages (or goods) faster than others, which determines the order of the messages before the randomization. The randomization in turn is not deterministic. In the 4.0.0 the randomization will not depend on the message order, but on the message id, which is a deterministic combination of the agent's name, id and message's sequence number.

CHAPTER

THREE

DIFFERENCES TO OTHER AGENT-BASED MODELING PLATFORMS

We identified several survey articles as well as a quite complete overview of agent-based modeling software on Wikipedia. [?], [?] [?], [?], [?], [?]. The articles 'Tools of the Trade' by Madey and Nikolai [?] and 'Survey of Agent Based Modelling and Simulation Tools' by Allan [?] attempt to give a complete overview of agent-based modelling platforms/frameworks. The Madey and Nikolai paper categorizes the abm-platforms according to several categories. (Programming Language, Type of License, Operating System and Domain). According to this article, there is only one software platform which aims at the specific domain of economics: JASA. But JASA is a modeling platform that aims specifically at auctions. Wikipedia [?] lists JAMEL as an economic platform, but JAMEL a is closed source and an non-programming platform. The 'Survey of Agent Based Modelling and Simulation Tools' by Allan [?] draws our attention to LSD, which, as it states, is rather a system dynamic, than an agent-based modeling platform. We conclude that there is a market for a domain specific language for economics.

While the formerly mentioned modeling platforms aim to give a complete overview, 'Evaluation of free Java - libraries for social scientific agent based simulation' [?] by Tobias and Hoffman chooses to concentrate on a smaller number of simulation packages. Tobias and Hoffman discuss: RePast, Swarm, Quicksilver, and VSEit. We will follow this approach and concentrate on a subset of ABM models. First as economics is a subset of social science we dismiss all platforms that are not explicitly targeted at social science. The list of social science platforms according to [?] Madey and Nikolai is: AgentSheets, LSD, FAMOJA, MAML, MAS-SOC, MIMOSE, NetLogo, Repast SimBioSys, StarLogo, StarLogoT, StarLogo TNG, Sugarscape, VSEit NetLogo and Moduleco. We dismiss some of these frameworks/platforms:

AgentSheets, because it is closed source and not 'programable'

LSD, because it is a system dynamics rather than an agent-based modeling environment

MAML, because it does not use a standard programming language, but his its own.

MAS-SOC, because we could not find it in the Internet and its documentation according to [?] is sparse.

MIMOSE, is an interesting language, but we will not analyze as it is based on a completely different programming paradigm, functional programming, as opposed to object-oriented programming.

SimBioSys, because it has according to Allan [?] and our research a sparse documentation.

StarLogo, StarLogoT, StarLogo TNG, because they have been superseded by NetLogo

Moduleco, because it has according to Allan [?] and our research a sparse documentation. Further, it appears not to be updated since roughly 2001

We will concentrate on the most widely used ABM frameworks/platforms: MASON, NetLogo, Repast.

3.1 General differences to other agent-based modeling platforms

First of all ABCE is domain specific, that enables it to provide the basic functions such as production, consumption, trade and communication as fully automated stock methods. Because any kind of agent interaction (communication and exchange of goods) is handled automatically ABCE, it can run the agents (virtually) parallel and run simulations on multi-core/processor systems without any intervention by the modeler.

The second biggest difference between ABCE and other platforms is that ABCE introduces the physical good as an ontological object in the simulation. Goods can be exchanged and transformed. ABCE handles these processes automatically, so that for the model a physical good behaves like a physical good and not like a message. That means that if a good is transfered between two agents the first agent does not have this good anymore, and the second agent has it now. Once, for example, two agents decide to trade a good ABCE makes sure that the transaction is cleared between the two agents.

Thirdly, ABCE is just a scheduler that schedules the actions of the agents and a python base class that enables the agent to produce, consume, trade and communicate. A model written in ABCE, therefore is standard Python code and the modeler can make use of the complete Python language and the Python language environment. This is a particular useful feature because Python comes with about 30.000 [1] publicly available packages, that could be used in ABCE. Particularly useful packages are:

pybrain a neural network package

numpy a package for numerical computation

scipy a package for numerical optimization and statistical functions

sympy a package for symbolic manipulation

turtle a package for spacial representation ala NetLogo

Fourth, many frameworks such as FLAME, NetLogo, StarLogo, Ascape and SugerScape and, in a more limited sense, Repast are designed with spacial representation in mind. For ABCE a spacial representation was explicitly not a design goal. However, since agents in ABCE are ordinary Python objects, they can use python modules such as python-turtle and therefore gain a spacial representation much like NetLogo. This does by no means mean that ABCE could not be a good choice for a problem where the spacial position plays a role. If for example the model has different transport costs or other properties according to the geographical position of the agents, but the agent's do not move or the movement does not have to be represented graphically, ABCE could still be a reasonable choice.

3.2 Physical Goods

Physical goods are at the heart of almost every economic model. The core feature and main difference to other ABM platforms is the implementation of physical goods. In contrast to information or messages, sharing a good means having less of it. In other words if agent A gives a good to agent B then agent A does not have this good anymore. On of the major strength of ABCE is that this is automatically handled.

In ABCE goods can be created, destroyed, traded, given or changed through production and consumption. All these functions are implemented in ABCE and can be inherited by an agent as a method. These functions are automatically handled by ABCE upon decision from the modeler.

Every agent in ABCE must inherit from the abce.Agent class. This gives the agent a couple of stock methods: create, destroy, trade and give. Create and destroy create or destroy a good immediately. Because trade and give involve a form of interaction between the agents they run over several sub-rounds. Selling of a good for example works like this:

- **Sub-round 1. The first agent offers the goods.** The good is automatically subtracted from the agents possessions, to avoid double selling.

[1] https://pypi.python.org/

- **Sub-round 2. The counter agent receives the offer. The agent can**
    1. accept: the goods are added to the counter part's possessions. Money is subtracted.
    2. reject (or equivalently ignore): Nothing happens in this sub-round
    3. partially accept the offer: The partial amount of goods is added to the counter part's possessions. Money is subtracted.
- **Sub-round 3. In case of**
    1. acceptance, the money is credited
    2. rejection the original good is re-credited
    3. partial acceptance the money is credited and the unsold part of the good is re-credited.

3.3 Difference to MASON

Masons is a single-threaded discrete event platform that is intended for simulations of social, biological and economical systems. [**?**]. Mason is a platform that was explicitly designed with the goal of running it on large platforms. MASON distributes a large number of single threaded simulations over deferent computers or processors. ABCE on the other hand is multi-threaded it allows to run agents in parallel. A single run of a simulation in MASON is therefore not faster on a computing cluster than on a potent single-processor computer. ABCE on the other hand uses the full capacity of multi-core/processor systems for a single simulation run. The fast execution of a model in ABCE allow a different software development process, modelers can 'try' their models while they are developing and adjust the code until it works as desired. The different nature of both platforms make it necessary to implement a different event scheduling system.

Mason is a discrete event platform. Events can be scheduled by the agents. ABCE on the other hand is scheduled - it has global list of sub-rounds that establish the sequence of actions in every round. Each of these sub-rounds lets a number of agents execute the same actions in parallel.

This, however does not mean that the order of actions is fixed. It is possible that one sub-round leads to different actions according to the internal logic of the agent. An implementation of a discrete event scheduler like in MASON is on the TODO list for ABCE.

MASON, like Repast Java is based on Java, while ABCE is based on Python, the advantages have been discussed before.

MASON comes with a visualization layer. ABCE, does not have this feature for several reasons. First spacial models are not an explicit design goal of ABCE. Second for models that have a spacial representation the Python's standard turtle library can be used and ad Netlogo like functionality. Thirdly ABCE has the ability of detailed statistical output, which can be readily visualized with standard software, such as R and Excel.

One form of spacial representation in MASON are networks. Networks in ABCE must be created by hand: in ABCE directed links of an agent are simply a Python-list with the id-numbers of the agents that are connected to it.

3.4 Difference to NetLogo

Netlogo is a multi-agent programming language, which is part of the Lisp language family. Netlogo is interpreted. [**?**] Python on the other hand is a compiled [2] general programming language. Consequently it is faster than NetLogo.

NetLogo has the unique feature to integrate the interface with with the programming language. ABCE goes a different direction rather than integrating the interface it forgoes the interface completely and everything is written in Python (even the configuration files), parameters are specified in excel-sheets (.csv).

[2] Python contrary to the common believe is compiled to bytecode similar to Java's compilation to bytecode.

Netlogo's most prominent feature are the turtle agents. To have turtle agents in ABCE, Python's turtle library has to be used. The graphical representation of models is therefore not part of ABCE, but of Python itself, but needs to be included by the modeler.

One difference between Netlogo and ABCE is that it has the concept of the observer agent, while in ABCE the simulation is controlled by the simulation process.

3.5 Difference Repast

Repast is a modeling environment for social science. It was originally conceived as a Java recoding of SWARM. [**?**] [**?**] Repast comes in several flavors: Java, .Net, and a Python like programming language. Repast has been superseded by Repast Symphony which maintains all functionality, but is limited to Java. Symphony has a point and click interface for simple models. :raw-tex:cite{NORTH2005a}

Repast does allow static and dynamic scheduling. [**?**]. ABCE, does not (yet) allow for dynamic scheduling. In ABCE, the order of actions - or in ABCE language order of sub-rounds - is fixed and is repeated for every round. This however is not as restrictive as it sounds, because in any sub-round an agent could freely decide what he does.

The advantage of the somehow more limited implementation of ABCE is ease of use. While in Repast it is necessary to subclass the scheduler in ABCE it is sufficient to specify the schedule and pass it the Simulation class.

Repast is vast, it contains 210 classes in 9 packages :raw-tex'cite{Collier}'. ABCE, thanks to its limited scope and Python, has only 6 classes visible to the modeler in a single package.

CHAPTER

FOUR

PARALLEL EXECUTION OF AGENTS AND MESSAGING WITH ZEROMQ

4.1 Concurrency

Agents run in parallel, time consistency is achieved by stepping through the simulation sub-round by sub-round. Consistency in communication is achieved by allowing the communication only between the sub-round. During a sub-round messages (or transfer of goods) are collected and between the sub-round they are transferred to the recipient. The message order is randomized.

4.2 Internal Control of the schedule and communication

There are two command processes. One organize the scheduling of the action and the other one the time consistency of the communication between agents.

A simulation contains a number of rounds comprised of sub-rounds. Every sub-round the following sequence of events occurs:

The simulation process manages the schedule, it sends the sub-round corresponding to be executed to the agents and simultaneously sends the number of agents that have received the order to execute this sub-round to the communication agent. The agents then execute the sub-round, send messages for other agents to the communication process; when they are finished they signal this to the communication process. Once all agents have signaled that they are finished the communication process then distributes the messages to the recipients. When the agents communicate that they have received the messages, the communication process signals the completion of a sub-round to simulation. The sub-round is concluded.

Agents also send messages to the logging process, messages to the logging are not synchronized, but time stamped.

Message passing and synchronization are managed by a third party library ZeroMQ [**?**]. Every agent has four communication sockets: one to receive commands, one to receive messages, one to send data to the database and the last one to send messages and signal the completion of the sub-round and reception of messages. It is therefore imperative to have sockets that are very small. What is more potentially millions of messages are exchanged both in the execution of the program and as messages representing the interaction between agents. Messaging speed is therefore extremely important. ZeroMQ is a very small and efficient transport protocol. [**?**]. Further it might become part of the Linux core.

CHAPTER

FIVE

DOWNLOAD AND INSTALLATION

5.1 Installation of stable version Ubuntu

1. Download stable version form: https://dl.dropboxusercontent.com/u/3655123/abce-0.3.tar.gz
2. If pip is not installed in terminal:

   ```
   sudo apt-get install python-pip, python-scipy, python-numpy
   ```

3. If R has not been installed automatically install it:

   Install R from www.CRAN.org, at least version 2.15

4. In terminal:

   ```
   sudo pip install abce-0.3.1.tar.gz
   ```

5. Download templats and examples from: https://dl.dropboxusercontent.com/u/3655123/abce_templates-0.3.zip
6. unzip abce_templates-0.3.zip

5.2 Installation of stable version Windows

1. Install Python2.7 (preferably 32bit)
2. Next, set the system's PATH variable to include directories

   that include Python components and packages we'll add later. To do this: - Right-click Computer and select Properties. - In the dialog box, select Advanced System Settings. - In the next dialog, select Environment Variables. - In the User Variables section, edit the PATH statement to include this:

   ```
   C:\Python27;C:\Python27\Lib\site-packages\;C:\Python27\Scripts\;
   ```

2. Install Setuptools from http://pypi.python.org/pypi/setuptools#downloads
3. install pip:

   easy_install pip

4. Install R form www.cran.org
5. Download stable version form: https://dl.dropboxusercontent.com/u/3655123/abce-0.3.tar.gz
6. install ABCE:

   pip install abce-0.3.1.tar.gz

In case of problems reinstall python http://www.anthonydebarros.com/2011/10/15/setting-up-python-in-windows-7/

7. Download templats and examples from: https://dl.dropboxusercontent.com/u/3655123/abce_templates-0.3.zip
8. unzip abce_templates-0.3.zip

5.3 Installation of development version

The installation has two parts. Installing the necessary software packages. Retrieving ABCE with git or as a zip file.

Alternative 1 as a zip (EASY):

1. download the zip file from: https://github.com/DavoudTaghawiNejad/abce
2. extract zip file

Alternative 2 via git [1] in terminal (RECOMMENDED):

```
[register with git and automatize a secure connection with your computer]
sudo apt-get install git
mkdir abce
cd abce
git init
git pull git@github.com:DavoudTaghawiNejad/abce.git
```

Optional for development you can install sphinx and sphinx-apidoc, the system that created this documentation. sphinx-apidoc currently needs the newest version of sphinx.

[1] Git is a a version controll system. It is higly recommended to use it to make your development process more efficient and less error prone. http://gitimmersion.com/

CHAPTER

SIX

WALK THROUGH / TUTORIAL

This tutorial is a step by step guide to create an agent-based model with ABCE. In the following two text boxes, the two concepts that make ABCE special are explained: The explicit modeling of tradeable goods and the optional concept of a physically closed economy.

Objects the other ontological object of agent-based models.

Objects have a special stance in agent-based modeling:

- objects can be recovered (resources)
- exchanged (trade)
- transformed (production)
- consumed (transformed :-))
- destroyed (not really) and time depreciated

ABCE, takes care of trade, production / transformation and consumption of goods automatically. Good categories can also be made to perish or regrow.

Services or labor We can model services and labor as goods that perish and that are replenished every round. This would amount to a worker that can sell one unit of labor every round, that disappears if not used.

Closed economy When we impose that goods can only be transformed. The economy is physically closed (the economy is stock and flow consistent). When the markets are in a complete network our economy is complete. Think "general" in equilibrium economics.

Caveats: If agents horde without taking their stock into account it's like destruction.

With this basic understanding you can now start writing your own model.

To create a model you basically have to follow three steps:

1. Specify endowments that replenish every round and goods / services that perish
2. Specify the order of actions
3. Write the agents with their actions

There is of course a little bit of administrative work you have to do:

1. import agents in the model
2. specify parameters

6.1 Have a look on the *abce/examples/* folder

It is instructive to look at a simple example, for example the 2x2 economy. Then you can make a working copy of the template or a copy of an example.

6.2 Make a working copy

copy abce/example to your_model_path:

```
cd your_model_path
cp path.to/abce/template/* .
```

6.3 start.py

6.3.1 Overview

In start.py the simulation, thus the parameters, objects, agents and time line are set up. Further it is declared, what is observed and written to the database. [1]

```
from Firm import Firm
from Household import Household
```

Here the Agent class Firm is imported from the file Firm.py. Likewise the Household class.

ABCE, reads the model parameter from a spreed sheet, every line is one simulation:

```
for parameters in simulation.read_parameters('simulation_parameters.csv'):

   ...
```

With the parameters ABCE loops over the intended line, to create the simulation and then runs the simulation. (after that it reads the next line an loops again). The variable parameters contain all parameters from 'simulation_parameters.csv'. See *simulation_parameters and agent_parameters* for details.

To set up a new model, you create a class a that will comprise your model:

```
s = Simulation(parameters)

...
```

After this the order of actions, agents and objects are added.

```
action_list = [
('Household', 'offer_capital'),

...

('Household', 'consumption')
]
s.add_action_list(action_list)
```

This establishes the order of the simulation. It can also be read from file `abce.Simulation.add_action_list_from_file()`

In order to add an agent which was imported before we simply build these agents:

```
s.build_agents(Firm, 'number_of_firms')
s.build_agents(Household, 10)
```

[1] from __future__ import division, instructs python to handle division always as a floating point division. Use this in all your python code. If you do not use this `3 / 2 = 1` instead of `3 / 2 = 1.5` (floor division).

The number of firms to be built is read from the column in simulation_parameters.csv called number_of_firms. The number of households on the other side is fixed at 10.

Or you can create panel data for a group of agents:

```
s.panel_db('Firm', command='after_sales_before_consumption')
s.panel_db('Household')  # at the beginning
...

s.run()
```

6.3.2 The order of actions: The order of actions within a round

Every agents-based model is characterized by the order of which the actions are executed. In ABCE, there are rounds, every round is composed of sub-rounds, in which all agents, a group of agents or a single agent, act in parallel. In the code below you see a typical sub-round.

You have to declare an action_list, that is made of tuples telling ABCE which agent or agent group, should execute which method:

```
action_list = [
repeat([
    ('Household', 'offer_capital'),
    ('Firm', 'buy_capital'),
],
repetitions=10),
('Household', 'search_work'),
('Firm', 'hire_labor'),
('Firm', 'production'),
'after_sales_before_consumption',
('Household', 'consumption')
]
s.add_action_list(action_list)
```

The first tuple for example tells all Household agents to execute the method "offer_capital". The 'after_sales_before_consumption' is a database command. see `abce.panel_db()`.

The repeat function allows repeating actions within the brackets a determinate amount of times.

Interactions happen between sub-rounds. An agent, sends a message in one round. The receiving agent, receives the message the following sub-round. A trade is finished in three rounds: (1) an agent sends an offer the good is blocked, so it can not be sold twice (2) the other agent accepts or rejects it. (3) If accepted, the good is automatically delivered. If the trade was rejected: the blocked good is unblocked.

6.3.3 The goods

A normal good can be traded and used for production or consumption. The only thing you have to do is create the amount of goods for every agent with `abce.Agent.create()` in the agent's __init__ method.

If an agent receives an endowment every round this can be automatically handled, with `abce.Simulation.declare_round_endowment()`. For example the following command gives, at the beginning of every round, to whom who possess one unit of 'field' 100 units of 'corn':

```
s.declare_round_endowment('field', 100, 'corn')
```

You can also declare goods that last only one round and then automatically perish. `abce.Simulation.declare_perishable()`

```
s.declare_perishable('corn')
```

This example declares 'corn' perishable and every round the agent gets 100 units of of 'corn' for every unit of field he possesses. If the corn is not consumed, it automatically disappears at the end of the round.

One important remark, for a logically consistent **macro-model** it is best to not create any goods during the simulation, but only in `abce.Agent.__init__()`. During the simulation the only new goods should be created by declare_round_endowment. In this way the economy is physically closed. An exception is, of course, money.

6.4 The agents

Agents are modeled in a separate file. In the template directory, you will find three agents: agent.py, firm.py and household.py.

At the beginning of each agent you will again find:

```
from __future__ import division
```

An agent has to import the `abce` module and some helpers:

```
import abce
from abcetools import is_zero, is_positive, is_negative, NotEnoughGoods
```

This imports the base classes: abce, Household and Firm.

An agent is a class and must at least inherit `abce.Agent`. `abce.Trade` - `messaging.Messaging` and `database.Database` are automatically inherited:

```
class Agent(abce.Agent):
```

To create an agent that can also consume:

```
class Household(abce.Agent, abce.Household):
```

You see our Household agent inherits from `abce.Agent`, which is compulsory and `abce.Household`. Household on the other hand are a set of methods that are unique for Household agents. (there is also a Firm class)

6.4.1 The __init__ method

```
def __init__(self, simulation_parameters, agent_parameters, _pass_to_engine):
    abce.__init__(self, *_pass_to_engine)
    self.create('labor_endowment', 1)
    self.create('capital_endowment', 1)
    self.set_cobb_douglas_utility_function({"MLK": 0.300, "BRD": 0.700})
    self.prices = {}
    self.prices['labor'] = 1
    self.number_of_firms = simulation_parameters['number_of_firms']
    self.renter = random.randint(0, 100)
    self.last_utility = None
```

The __init__ method is the method that is called when the agents are created (by the `abce.Simulation.build_agents()` or `abce.Simulation.build_agents_from_file()` method.) In this method agents can access the simulation_parameters from the 'simulation_parameters.csv'.

If the agents are built using `abce.Simulation.build_agents_from_file()`. The agents can access the parameters in their row, in 'agents_parameters.csv', by agent_parameters in the __init__ function.

Line 2 is compulsory to pass the parameters to the abce.

With self.create the agent creates the good 'labor_endowment'. Any good can be created. Generally speaking. The __init__ method is the only place where it is consistent to create a good. (except for money, if you simulate a naive central bank).

This agent class inherited `abce.Household.set_cobb_douglas_utility_function()` from `abce.Household`. With `abce.Household.set_cobb_douglas_utility_function()` you can create a cobb-douglas function. Other functional forms are also available.

self.prices is a dictionary, created by the modeler, that saves prices for specific goods. Here the price for labor is set to 1.

In order to let the agent remember a simulation_parameter it has to be saved in the self domain the agent. [2]

There is a random number assigned to self.renter and self.last_utility is initialized with None. It is often necessary to initialize variable in the __init__ method to avoid errors in the first round.

6.4.2 The action methods and a consuming Household

All the other methods of the agent are executed when the corresponding sub-round is called from the Simulation set up in start.py. [3]

For example when in the action list *('household', 'eat')* is called the eat method is executed of each household agent is executed:

```
class Agent(abce.Agent, abce.Household)
    def __init__(self):
        self.set_cobb_douglas_utility_function({'cookies': 0.9', 'bread': 0.1})
        self.create('cookies', 1)
        self.create('bread', 5)

...
def eat(self):
    utility = self.consume_everything()
    self.log('utility', {'a': utility})
```

In the above example we see how a utility function is declared and how the agent consumes. The utility is logged and can be retrieved see *retrieval of the simulation results*

6.4.3 Firms and Production functions

Firms do two things they produce (transform) and trade. The following code shows you how to declare a technology and produce bread from labor and yeast.

```
class Agent(abce.Agent, abce.Household):
def init(self):
   set_cobb_douglas('BRD', 1.890, {"yeast": 0.333, "LAB": 0.667})
    ..
def production(self):
    self.produce_use_everything()
```

More details in `abce.Firm`. `abce.FirmMultiTechnologies` offers a more advanced interface for firms with complicated technologies.

[2] (self.number_of_firms = simulation_parameters['number_of_firms'])

[3] With the exception of methods, whose names start with a '_' underscore.underscoring methods that the agent uses only internally can speed up your code.

6.4.4 Trade

ABCE handles trade fully automatically. That means, that goods are automatically exchanged, double selling of a good is avoided by subtracting a good from the possessions when it is offered for sale. The modeler has only to decide when the agent offers a trade and sets the criteria to accept the trade:

```
# Agent 1
def selling(self):
    offerid = self.sell(buyer, 'BRD', 1, 2.5)
    self.checkorders.append(offerid)

# Agent 2
def buying(self):
    offers = self.get_offers('cookies')
    for offer in offers:
       try:
          self.accept(offer)
       except NotEnoughGoods:
          self.reject(offer)
```

You can find a detailed explanation how trade works in `abce.Trade`

6.4.5 Data production

There are three different ways of observing your agents:

Trade Logging

ABCE by default logs all trade and creates a SAM or IO matrix.

Manual in agent logging

An agent can log a variable, `abce.Agent.possessions()`, `abce.Agent.possessions_all()` and most other methods such as `abce.Firm.produce()` with `abce.Agent.log()` or a change in a variable with `log_change()`:

```
self.log('possessions', self.possesions_all())
self.log('custom', {'price_setting': 5: 'production_value': 12})
prod = self.production_use_everything()
self.log('current_production', prod)
```

Panel Data

`panel_data()` creates panel data for all agents in a specific agent group at a specific point in every round. It is set in start.py:

```
s.panel_data('Household', command='aftersalesbeforeconsumption')
```

The command has to be inserted in the action_list.

Retrieving the logged data

the results are stored in a subfolder of the ./results/ folder.

The tables are stored as '.csv' files which can be opened with excel and libreoffice. Further you can import the files with R:

1. change to the subfolder of ./results/ that contains your simulation results
2. start R
3. *load('database.R')*

CHAPTER

SEVEN

EXAMPLES

ABCE comes with two example projects 'one_household_one_firm', describes a one sector economy with one representative firm and household, '2sectors', describes a two sector economy with one representative household

7.1 One sector model

One household one firm is a minimalistic example of a 'macro-economy'. It is 'macro' in the sense that the complete circular flow of the economy is represented. Every round the following sub-rounds are executed:

> **household:** sell_labor
>
> **firm:** buy_labor
>
> **firm:** production
>
> **firm:** sell_goods
>
> **household:** buy_goods
>
> **household:** consumption

After the firms' production and the acquisition of goods by the household a statistical panel of the firms' and the households' possessions, respectively, is written to the database.

The economy has two goods a representative 'GOOD' good and 'labor' as well as money. 'labor', which is a service that is represented as a good that perishes every round when it is not used. Further the endowment is of the labor good that is replenished every round for every agent that has an 'adult'. 'Adults' are handled like possessions of the household agent.

The household has a degenerate Cobb-Douglas utility function and the firm has a degenerate Cobb-Douglas production function:

```
utility = GOOD ^ 1
```

```
GOOD = labor ^ 1
```

The firms own an initial amount of money of 1 and the household has one adult, which supplies one unit of (perishable) labor every round.

First the household sells his unit of labor. The firm buys this unit and uses all available labor for production. The complete production is offered to the household, which in turn buys everything it can afford. The good is consumed and the resulting utility logged to the database.

7.2 Two sector model

The two sector model is similar to the one sector model. It has two firms and showcases ABCE's ability to control the creation of agents from an excel sheet.

There are two firms. One firm manufactures an intermediary good. The other firm produces the final good. Both firms are implemented with the same good. The type a firm develops is based on the excel sheet.

The two respective firms production functions are:

```
intermediate_good = labor ^ 1

consumption_good = intermediate_good ^ 1 * labor ^ 1
```

The only difference is that, when firms sell their products the intermedate good firm sells to the final good firm and the final good firm, in the same sub-round sells to the household.

In start.py we can see that the firms that are build are build from an excel sheet:

> w.build_agents_from_file(Firm, parameters_file='agents_parameters.csv') w.build_agents_from_file(Household)

And here the excel sheet:

> agent_class number sector firm 1 intermediate_good firm 1 consumption_good household 1 0 household 1 1

The advantage of this is that the parameters can be used in the agent. The line *self.sector = agent_parameters['sector']* reads the sector column and assigns it to the self.sector variable. The file simulation parameters is read - line by line - into the variable simulation_parameters. It can be used in start.py and in the agents with simulation_parameters['columnlabel'].

CHAPTER

EIGHT

UNIT TESTING

One of the major problem of doing science with simulations is that results found could be a mere result of a mistake in the software implementation. This problem is even stronger when emergent phenomena are expected. The first hedge against this problem is of course carefully checking the code. ABCE and Pythons brevity and readability are certainly helping this. However structured testing procedures create more robust software.

Currently all trade and exchange related as well as endowment and data logging facilities are unit tested. It is planned to extend unit testing to production and consumption, so that by version 0.4 all functions of the agents will be fully unit tested.

The modeler can run the unit testing facilities on his own system and therefore assert that on his own system the code runs correctly.

Unit testing is the testing of the testable part of a the software code. [**?**]. As in ABCE the most crucial functions are the exchange of goods or information, the smallest testable unit is often a combination of two actions [**?**]. For example making an offer and then by a second agent accepting or rejecting it. The interaction and concurrent nature of ABCE simulation make it unpractical to use the standard unit testing procedures of Python.

[**?**] argue that unit-testing is economical. In the analysis of three projects they find that unit-testing finds errors in the code and argue that its cost is often exaggerated. We can therefore conclude that unit-testing is necessary and a cost efficient way of ensuring the correctness of the results of the simulation. For the modeler this is an additional incentive to use ABCE, if he implemented the simulation as a stand alone program he would either have to forgo the testing of the agent's functions or write his own unit-testing facilities.

CHAPTER

NINE

AGENTS

The `abce.agent.Agent` class is the basic class for creating your agents. It automatically handles the possession of goods of an agent. In order to produce/transforme goods you also need to subclass the `abceagent.Firm` [1] or to create a consumer the `abce.agent.Household`.

For detailed documentation on:

Trading: see `abce.agent.Trade`

Logging and data creation: see `abce.agent.Database` and *Retrieval of the simulation results*

Messaging between agents: see `abce.agent.Messaging`.

exception `abce.tools.`**`NotEnoughGoods`**(*_agent_name*, *good*, *amount_missing*)

Methods raise this exception when the agent has less goods than needed

These functions (self.produce, self.offer, self.sell, self.buy) should be encapsulated by a try except block:

```
try:
   self.produce(...)
except NotEnoughGoods:
   alternative_statements()
```

class `abce.`**`Agent`**(*idn*, *group*, *_addresses*, *trade_logging*)

Bases: `abce.database.Database`, `abce.logger.Logger`, `abce.trade.Trade`, `abce.messaging.Messaging`, `multiprocessing.process.Process`

Every agent has to inherit this class. It connects the agent to the simulation and to other agent. The `abceagent.Trade`, `abceagent.Database` and `abceagent.Messaging` classes are included. You can enhance an agent, by also inheriting from `abceagent.Firm`.:class:*abceagent.FirmMultiTechnologies* or `abceagent.Household`.

For example:

```
class Household(abceagent.Agent, abceagent.Household):
    def __init__(self, simulation_parameters, agent_parameters, _pass_to_engine):
    abceagent.Agent.__init__(self, *_pass_to_engine)
```

`aesof_eval`(*column_name*)

evaluates an expression' in your excel file. see instruction of pythons eval commmand

`aesof_exec`(*column_name*)

executes a command in your excel file. see instruction of pythons exec commmand

`create`(*good*, *quantity*)

creates quantity of the good out of nothing

[1] or `abce.agent.FirmMultiTechnologies` for simulations with complex technologies.

Use create with care, as long as you use it only for labor and natural resources your model is macroeconomally complete.

Args: 'good': is the name of the good quantity: number

destroy(*good*, *quantity*)
destroys quantity of the good,

Args:

```
'good': is the name of the good
quantity: number
```

Raises:

```
NotEnoughGoods: when goods are insufficient
```

destroy_all(*good*)
destroys all of the good, returns how much

Args:

```
'good': is the name of the good
```

possession(*good*)
returns how much of good an agent possesses.

Returns: A number.

possession does not return a dictionary for self.log(...), you can use self.possessions([...]) (plural) with self.log.

Example:

if self.possession('money') < 1: self.financial_crisis = True

if not(is_positive(self.possession('money')): self.bancrupcy = True

possessions(*list_of_goods*)
returns a dictionary of goods and the corresponding amount an agent owns

Argument: A list with good names. Can be a list with a single element.

Returns: A dictionary, that can be used with self.log(..)

Examples:

```
self.log('buget', self.possesions(['money']))

self.log('goods', self.possesions(['gold', 'wood', 'grass']))

have = self.possessions(['gold', 'wood', 'grass']))
for good in have:
    if have[good] > 5:
        rich = True
```

possessions_all()
returns all possessions

possessions_filter(*goods=None*, *but=None*, *match=None*, *beginswith=None*, *endswith=None*)
returns a subset of the goods an agent owns, all arguments can be combined.

Args:

goods (list, optional): a list of goods to return

but(list, optional): all goods but the list of goods here.

match(string, optional TODO): goods that match pattern

beginswith(string, optional): all goods that begin with string

endswith(string, optional) all goods that end with string

is(string, optional TODO)

'resources': return only goods that are endowments

'perishable': return only goods that are perishable

'resources+perishable': goods that are both

'produced_by_resources': goods which can be produced by resources

Example:

```
self.consume(self.possessions_filter(but=['money']))
# This is redundant if money is not in the utility function
```

run()

CHAPTER TEN

TRADE

class `abce.`**`Trade`**

Agents can trade with each other. The clearing of the trade is taken care of fully by ABCE. Selling a good works in the following way:

1. An agent sends an offer. `sell()`

   The good offered is blocked and self.possession(...) does not account for it.

2. **Next subround:** An agent receives the offer `get_offers()`, and can `accept()`, `reject()` or partially accept it. `accept_partial()`

   The good is credited and the price is deducted from the agent's possesions.

3. **Next subround:**

   - in case of acceptance *the money is automatically credited.*
   - in case of partial acceptance *the money is credited and part of the blocked good is unblocked.*
   - in case of rejection *the good is unblocked.*

Analogously for buying. (`buy()`)

Example:

```
# Agent 1
def sales(self):
    self.remember_trade = self.sell('Household', 0, 'cookies', quantity=5, price=self.price)

# Agent 2
def receive_sale(self):
    oo = self.get_offers('cookies')
    for offer in oo:
        if offer['price'] < 0.3:
            try:
                self.accept(offer)
            except NotEnoughGoods:
                self.accept_partial(offer, self.possession('money') / offer['price'])
        else:
            self.reject(offer)

# Agent 1, subround 3
def learning(self):
    offer = self.info(self.remember_trade)
    if offer['status'] == 'reject':
        self.price *= .9
```

```
elif offer['status'] = 'partial':
    self.price *= offer['final_quantity'] / offer['quantity']
```

Quotes on the other hand allow you to ask a trade partner to send you a not committed price quote. The modeller has to implement a response mechanism. For convenience `accept_quote()` and `accept_quote_partial()`, send a committed offer that it is the uncommitted price quote.

accept (*offer*)
The offer is accepted and cleared

Args:

```
offer: the offer the other party made
(offer not quote!)
```

Return: Returns a dictionary with the good's quantity and the amount paid.

accept_max_possible (*offer*)
TODO The offer is partly accepted and cleared

Args: offer: the offer the other party made (offer not quote!)

Return: Returns a dictionary with the good's quantity and the amount paid.

accept_partial (*offer*, *quantity*)
TODO The offer is partly accepted and cleared

Args: offer: the offer the other party made (offer not quote!)

Return: Returns a dictionary with the good's quantity and the amount paid.

accept_partial_max_possible (*offer*, *quantity*)
TODO The offer is partly accepted and cleared

Args: offer: the offer the other party made (offer not quote!)

Return: Returns a dictionary with the good's quantity and the amount paid.

accept_quote (*quote*)
makes a commited buy or sell out of the counterparties quote

Args:: quote: buy or sell quote that is accepted

accept_quote_partial (*quote*, *quantity*)
makes a commited buy or sell out of the counterparties quote

Args:: quote: buy or sell quote that is accepted quantity: the quantity that is offered/requested it should be less than propsed in the quote, but this is not enforced.

buy (*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)
commits to sell the quantity of good at price

The goods are not in haves or self.count(). When the offer is rejected it is automatically re-credited. When the offer is accepted the money amount is credited. (partial acceptance accordingly)

Args: receiver_group: an agent name NEVER a group or 'all'! (it is an error but with a confusing warning) 'good': name of the good quantity: maximum units disposed to buy at this price price: price per unit

buy_max_possible (*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)
Same as buy but if money is insufficient, it executes the deal with a lower amount of goods using all available money.

get_offers(*good*, *descending=False*)

returns all offers of the 'good' ordered by price.

Offers that are not accepted in the same subround (def block) are automatically rejected. However you can also manualy reject.

Args:

good: the good which should be retrieved

descending(bool,default=False): False for descending True for ascending by price

Returns: A list of offers ordered by price

Example:

```
offers = get_offers('books')
for offer in offers:
    if offer['price'] < 50:
        self.accept(offer)
    elif offer['price'] < 100:
        self.accept_partial(offer, 1)
    else:
        self.reject(offer)  # optional
```

get_offers_all(*descending=False*)

returns all offers in a dictionary, with goods as key. The in each goods-category the goods are ordered by price. The order can be reversed by setting descending=True

Offers that are not accepted in the same subround (def block) are automatically rejected. However you can also manualy reject.

Args:

```
descending(optional): is a bool. False for descending True for
                      ascending by price
```

Example2:

```
oo = get_offers_all(descending=False)
for good_category in oo:
   print('The cheapest good of category' + good_category
   + ' is ' + good_category[0])
#sorted list of beer prices and seller
for offer in oo['beer']:
   print(offer['price'], offer['sender'])
```

Lists can only efficiently pop the last item. Therefore it is more efficient to order backwards and buy the last good first:

```
def buy_input_good(self):
   offers = self.get_offers_all(descending=True)
   while offers:
       if offers[good][-1]['quantity'] == self.prices_for_which_buy[good]:
           self.accept(offers[good].pop())
```

get_quotes(*good*, *descending=False*)

self.get_quotes() returns all new quotes and removes them. The order is randomized.

Args:

good: the good which should be retrieved

descending(bool,default=False): False for descending True for ascending by price

Returns: list of quotes ordered by price

Example:

```
quotes = self.get_quotes()
```

get_quotes_all(*descending=False*)
self.get_quotes_all() returns a dictionary with all now new quotes ordered by the good type and removes them. The order is randomized.

Args:

descending(bool,default=False): False for descending True for ascending by price

Returns: dictionary of list of quotes ordered by price. The dictionary itself is ordered by price.

Example:

```
quotes = self.get_quotes()
```

give(*receiver_group*, *receiver_idn*, *good*, *quantity*)
gives a good to another agent

Args:

receiver_group: Group name of the receiver

receiver_idn: id number of the receiver

good: the good to be transfered

quantity: amount to be transfered

Raises:

AssertionError, when good smaller than 0.

Return: Dictionary, with the transfer, which can be used by self.log(...).

Example:

```
self.log('taxes', self.give('money': 0.05 * self.possession('money'))
```

info(*offer_idn*)
lets you access all fields of a **given** offer. This allows you to check whether an offer was accepted, partially accepted or rejected and retrieve the quantity actually traded.

If in your first round the value you are testing is not set, set the variable to *None*. None in the first round returns an empty trade with quantity = 0 and price = 1. The status in accepted.

Example:

```
class Example:
    def __init__(self):
        self.last_offer = None
        self.price = 5

    def selling(self):
        if self.info(self.last_offer)['status'] == 'accept':
            self.price += 1
        self.last_offer = self.sell('Household', 1, 'cookies', 5, self.price)
```

Returns a dictionary; Fields:

['status']:

'accepted': trade fully accepted

'partial': ['final_quantity'] and self.offer_partial_status_percentage(...) for the quantities actually accepted

'rejected': trade rejected

'pending': offer has not yet answered, and is not older than one round.

'perished': the **perishable** good was not accepted by the end of the round and therefore perished.

['quantity']: the quantity of the original quote.

['final_quantity']:

This returns the actual quantity bought or sold. (Equal to quantity if the offer was accepted fully)

Raises:

KeyError: If the offer was answered more than one round ago.

Example Pending:

```
status = self.info(self.offer_idn)['status']
if status == 'pending':
    print('offer has not yet been answered')
```

Example:

```
from pybrain.rl.learners.valuebased import ActionValueTable
from pybrain.rl.agents import LearningAgent
from pybrain.rl.learners import Q

def __init__(self):
    controller = ActionValueTable(dimState=1, numActions=1)
    learner = Q()
    rl_price = LearningAgent(controller, learner)
    self.car_cost = 500

def sales(self):
    price = reinforcement_learner.getAction():
    self.offer = self.sell('Household', 1, 'car', 1, price)

def learn(self):
    reinforcement_learner.integrateObservation([self.info(self.offer)])
    reinforcement_learner.giveReward([self.info(self.offer) * price - self.car_cost])
```

partial_status_percentage(*offer_idn*)

returns the percentage of a partial accept

Args:

offer_idn: on offer as returned by self.sell(...) ord self.buy(...)

Returns: A value between [0, 1]

Raises: KeyError, when no answer has been given or received more than one round before

peak_offers(*good*, *descending=False*)

returns a peak on all offers of the 'good' ordered by price. Peaked offers can not be accepted or rejected, but they do not expire. Args:

good: the good which should be retrieved

descending(bool,default=False): False for descending True for ascending by price

Returns: A list of offers ordered by price

Example:

```
offers = get_offers('books')
for offer in offers:
    if offer['price'] < 50:
        self.accept(offer)
    elif offer['price'] < 100:
        self.accept_partial(offer, 1)
    else:
        self.reject(offer)  # optional
```

quote_buy(*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)

quotes a price to buy quantity of 'good' a receiver

price (money) per unit offers a deal without checking or committing resources

Args:

receiver_group: agent group name of the agent

receiver_idn: the agent's id number

'good': name of the good

quantity: maximum units disposed to buy at this price

price: price per unit

quote_sell(*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)

quotes a price to sell quantity of 'good' to a receiver

price (money) per unit offers a deal without checking or committing resources

Args:

receiver_group: agent group name of the agent

receiver_idn: the agent's id number

'good': name of the good

quantity: maximum units disposed to sell at this price

price: price per unit

reject(*offer*)

The offer is rejected

Args: offer: the offer the other party made (offer not quote!)

retract(*offer_idn*)

The agent who made a buy or sell offer can retract it

The offer an agent made is deleted at the end of the sub-round and the committed good reappears in the haves. However if another agent accepts in the same round the trade will be cleared and not retracted.

Args: offer: the offer he made with buy or sell (offer not quote!)

sell(*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)
commits to sell the quantity of good at price

The goods are not in haves or self.count(). When the offer is rejected it is automatically re-credited. When the offer is accepted the money amount is credited. (partial acceptance accordingly)

Args: receiver_group: an agent name NEVER a group or 'all'!!! (its an error but with a confusing warning) 'good': name of the good quantity: maximum units disposed to buy at this price price: price per unit

Returns: A reference to the offer. The offer and the offer status can be accessed with *self.info(offer_reference)*.

Example:

```
def subround_1(self):
    self.offer = self.sell('household', 1, 'cookies', quantity=5, price=0.1)

def subround_2(self):
    offer = self.info(self.offer)
    if offer['status'] == 'partial':
        print(offer['final_quantity'] , 'cookies have be bougth')
    elif:
        offer['status'] == 'accepted':
        print('Cookie monster bougth them all')
    elif:
        offer['status'] == 'rejected':
        print('On diet')
```

sell_max_possible(*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)
Same as sell but if the possession of good is smaller than the number, it executes the deal with a lower amount of goods using everything available of this good.

CHAPTER

ELEVEN

FIRM AND PRODUCTION

The Firm class gives an Agent the ability to set production functions and produce.

class `abce.`**`Firm`**

Bases: `abce.firmmultitechnologies.FirmMultiTechnologies`

The firm class allows you to declare a production function for a firm. `set_leontief()`, `set_production_function()` `set_cobb_douglas()`, `set_production_function_fast()` (FirmMultiTechnologies, allows you to declare several) With `produce()` and `produce_use_everything()` you can produce using the according production function. You have several auxiliary functions for example to predict the production. When you multiply `predict_produce()` with the price vector you get the profitability of the production.

`predict_produce_output_simple`(*input_goods*)

Calculates the output of a production (but does not produce)

Predicts the production of produce(production_function, input_goods) see also: Predict_produce(.) as it returns a vector

Args:

{'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good used for the production.

Example:

```
print(A.predict_output_produce(two_cars))
>>> {'car': 2}
```

`predict_produce_simple`(*input_goods*)

Returns a vector with input (negative) and output (positive) goods

Predicts the production of produce(production_function, input_goods) and the use of input goods. net_value(.) uses a price_vector (dictionary) to calculate the net value of this production.

Args:

{'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good used for the production.

Example:

```
prices = {'car': 50000, 'tire': 100, 'metal': 10, 'plastic':  0.5}
value_one_car = net_value(predict_produce(one_car), prices)
value_two_cars = net_value(predict_produce(two_cars), prices)
if value_one_car > value_two_cars:
    A.produce(one_car)
else:
    A.produce(two_cars)
```

produce(*input_goods*)

Produces output goods given the specified amount of inputs.

Transforms the Agent's goods specified in input goods according to a given production_function to output goods. Automatically changes the agent's belonging. Raises an exception, when the agent does not have sufficient resources.

Args:

{'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good used for the production.

Raises:

NotEnoughGoods: This is raised when the goods are insufficient.

Example:

```
self.set_cobb_douglas_production_function('car' ..)
car = {'tire': 4, 'metal': 2000, 'plastic':  40}
try:
    self.produce(car)
except NotEnoughGoods:
    print('today no cars')
```

produce_use_everything()

Produces output goods from all input goods.

Example:

```
self.produce_use_everything()
```

set_cobb_douglas(*output*, *multiplier*, *exponents*)

sets the firm to use a Cobb-Douglas production function.

A production function is a production process that produces the given input goods according to the formula to the output good.

Args: 'output': Name of the output good multiplier: Cobb-Douglas multiplier {'input1': exponent1, 'input2': exponent2 ...}: dictionary containing good names 'input' and corresponding exponents

Example:

```
self.set_cobb_douglas('plastic', 0.000001, {'oil' : 10, 'labor' : 1})
self.produce({'oil' : 20, 'labor' : 1})
```

set_leontief(*output*, *utilization_quantities*, *multiplier=1*, *isinteger='int'*)

sets the firm to use a Leontief production function.

A production function is a production process that produces the given input given according to the formula to the output good.

Warning, when you produce with a Leontief production_function all goods you put in the produce(...) function are used up. Regardless whether it is an efficient or wasteful bundle

Args: 'output': Name of the output good {'input1': utilization_quantity1, 'input2': utilization_quantity2 ...}: dictionary containing good names 'input' and corresponding exponents multiplier: multiplier isinteger='int' or isinteger='': When 'int' produces only integer amounts of the good. When '', produces floating amounts.

Example:

```
self.create_leontief('car', {'tire' : 4, 'metal' : 1000, 'plastic' : 20}, 1)
two_cars = {'tire': 8, 'metal': 2000, 'plastic':  40}
self.produce(two_cars)
```

set_production_function (*formula*, *typy='from_formula'*)

sets the firm to use a Cobb-Douglas production function from a formula.

A production function is a production process that produces the given input given input goods according to the formula to the output goods. Production_functions are than used to produce, predict_vector_produce and predict_output_produce.

create_production_function_fast is faster but more complicated

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are separated by a ;

Example:

```
formula = 'golf_ball = (ball) * (paint / 2); waste = 0.1 * paint'
self.set_production_function(formula)
self.produce({'ball' : 1, 'paint' : 2}
```

//exponential is ** not ^

set_production_function_fast (*formula*, *output_goods*, *input_goods*, *typ='from_formula'*)

sets the firm to use a Cobb-Douglas production function from a formula, with given outputs

A production function is a production process that produces the given input given according to the formula to the output goods. Production_functions are than used to produce, predict_vector_produce and predict_output_produce.

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are separated by a ; [output]: list of all output goods (left hand sides of the equations)

Example:

```
formula = 'golf_ball = (ball) * (paint / 2); waste = 0.1 * paint'
self.production_function_fast(formula, 'golf', ['waste'])
self.produce(self, {'ball' : 1, 'paint' : 2}
```

//exponential is ** not ^

sufficient_goods (*input_goods*)

checks whether the agent has all the goods in the vector input

CHAPTER

TWELVE

HOUSEHOLD AND CONSUMPTION

The Household class extends the agent by giving him utility functions and the ability to consume goods.

class `abce.`**`Household`**

`consume`(*input_goods*)

consumes input_goods returns utility according to the agent's consumption function

A utility_function, has to be set before see py:meth:~*abceagent.Household.set_ utility_function*, py:meth:~*abceagent.Household.set_cobb_douglas_utility_function* or

Args:

{'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good consumed.

Raises: NotEnoughGoods: This is raised when the goods are insufficient.

Returns: A the utility a number. To log it see example.

Example:

```
self.consumption_set = {'car': 1, 'ball': 2000, 'bike':  2}
self.consumption_set = {'car': 0, 'ball': 2500, 'bike':  20}
try:
    utility = self.consume(utility_function, self.consumption_set)
except NotEnoughGoods:
    utility = self.consume(utility_function, self.smaller_consumption_set)
self.log('utility': {'u': utility})
```

`consume_everything`()

consumes everything that is in the utility function returns utility according consumption

A utility_function, has to be set before see py:meth:~*abceagent.Household.set_ utility_function*, py:meth:~*abceagent.Household.set_cobb_douglas_utility_function*

Returns: A the utility a number. To log it see example.

Example:

```
utility = self.consume_everything()
self.log('utility': {'u': utility})
```

`predict_utility`(*input_goods*)

Predicts the utility of a vecor of input goods

Predicts the utility of consume_with_utility(utility_function, input_goods)

Args:

```
{'input_good1': amount1, 'input_good2': amount2 ...}: dictionary
containing the amount of input good used for the production.
```

Returns:

```
utility: Number
```

Example:

```
print(A.predict_utility(self._utility_function, {'ball': 2, 'paint': 1}))
```

`set_cobb_douglas_utility_function`(*exponents*)

creates a Cobb-Douglas utility function

Utility_functions are than used as an argument in consume_with_utility, predict_utility and predict_utility_and_consumption.

Args: {'input1': exponent1, 'input2': exponent2 ...}: dictionary containing good names 'input' and correstponding exponents

Returns: A utility_function that can be used in consume_with_utility etc.

Example: self._utility_function = self.create_cobb_douglas({'bread' : 10, 'milk' : 1}) self.produce(self.plastic_utility_function, {'bread' : 20, 'milk' : 1})

`set_utility_function`(*formula*, *typ='from_formula'*)

creates a utility function from formula

Utility_functions are then used as an argument in consume_with_utility, predict_utility and predict_utility_and_consumption.

create_utility_function_fast is faster but more complicated utility_function

Args: "formula": equation or set of equations that describe the utility function. (string) needs to start with 'utility = ...'

Returns: A utility_function

Example: formula = 'utility = ball + paint' self._utility_function = self.create_utility_function(formula) self.consume_with_utility(self._utility_function, {'ball' : 1, 'paint' : 2})

//exponential is ** not ^

`set_utility_function_fast`(*formula*, *input_goods*, *typ='from_formula'*)

creates a utility function from formula

Utility_functions are then used as an argument in consume_with_utility, predict_utility and predict_utility_and_consumption.

create_utility_function_fast is faster but more complicated

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are separated by a ; [output]: list of all output goods (left hand sides of the equations)

Returns: A utility_function that can be used in produce etc.

Example: formula = 'utility = ball + paint'

self._utility_function = self.create_utility_function(formula, ['ball', 'paint']) self.consume_with_utility(self._utility_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

utility_function()

the utility function should be created with: set_cobb_douglas_utility_function, set_utility_function or set_utility_function_fast

CHAPTER

THIRTEEN

MESSAGING

This is the agent's facility to send and receive messages. Messages can either be sent to an individual with `messaging.Messaging.message()` or to a group with `messaging.Messaging.message_to_group()`. The receiving agent can either get all messages with `messaging.Messaging.get_messages_all()` or messages with a specific topic with `messaging.Messaging.get_messages()`.

class `messaging.`**`Messaging`**

`get_messages`(*topic='m'*)

self.messages() returns all new messages send with `message()` (topic='m'). The order is randomized. self.messages(topic) returns all messages with a topic.

A message is a string with the message. You can also retrieve the sender by *message.sender_group* and *message.sender_idn* and view the topic with 'message.topic'. (see example)

If you are sending a float or an integer you need to access the message content with *message.content* instead of only *message*.

! if you want to recieve a **float** or an **int**, you must msg.content

Returns a message object:

msg.content: returns the message content string, int, float, ...

msg: returns also the message content, but only as a string

sender_group: returns the group name of the sender

sender_idn: returns the id of the sender

topic: returns the topic

Example:

```
... agent_01 ...
self.messages('firm_01', 'potential_buyers', 'hello message')

... firm_01 - one subround later ...
potential_buyers = get_messages('potential_buyers')
for msg in potential_buyers:
   print('message: ', msg)
   print('message: ', msg.content)
   print('group name: ', msg.sender_group)
   print('sender id: ', msg.sender_idn)
   print('topic: ', msg.topic)
```

get_messages_all()

returns all messages irregardless of the topic, in a dictionary by topic

A message is a string with the message. You can also retrieve the sender by *message.sender_group* and *message.sender_idn* and view the topic with 'message.topic'. (see example)

If you are sending a float or an integer you need to access the message content with *message.content* instead of only *message*.

get_messages_biased(*topic='m'*)

like self.messages(topic), but the order is not properly randomized, but it is faster. use whenever you are sure that the way you process messages is not affected by the order

message(*receiver_group*, *receiver_idn*, *topic*, *content*)

sends a message to agent. Agents receive it at the beginning of next round with `get_messages()` or `get_messages_all()`.

See: message_to_group for messages to multiple agents

Args:

```
receiver_group: agent, agent_group or 'all'
topic: string, with which this message can be received
content: string, dictionary or class, that is send.
```

Example:

```
... household_01 ...
self.message('firm', 01, 'quote_sell', {'good':'BRD', 'quantity': 5})

... firm_01 - one subround later ...
requests = self.get_messages('quote_sell')
for req in requests:
    self.sell(req.sender, req.good, reg.quantity, self.price[req.good])
```

Example2:

```
self.message('firm', 01, 'm', "hello my message")
```

message_to_group(*receiver_group*, *topic*, *content*)

sends a message to agent, agent_group or 'all'. Agents receive it at the beginning of next round with `get_messages()` or `get_messages_all()`.

Args:

```
receiver_group: agent, agent_group or 'all'
topic: string, with which this message can be received
content: string, dictionary or class, that is send.
```

Example:

```
... household_01 ...
self.message('firm_01', 'quote_sell', {'good':'BRD', 'quantity': 5})

... firm_01 - one subround later ...
requests = self.get_messages('quote_sell')
for req in requests:
    self.sell(req.sender, req.good, reg.quantity, self.price[req.good])
```

Example2:

```
self.message('firm_01', 'm', "hello my message")
```

CHAPTER

FOURTEEN

OBSERVING AGENTS

There are three different ways of observing your agents:

Trade Logging ABCE by default logs all trade and creates a SAM or IO matrix.

Manual in agent logging An agent is instructed to log a variable with `log()` or a change in a variable with `log_change()`.

Panel Data `panel_data()` creates panel data for all agents in a specific agent group at a specific point in every round. It is set in start.py

How to retrieve the Simulation results is explained in *Retrieval of the simulation results*

14.1 Trade Logging

By default ABCE logs all trade and creates a social accounting matrix or input output matrix. Because the creation of the trade log is very time consuming you can change the default behavior in world_parameter.csv. In the column 'trade_logging' you can choose 'individual', 'group' or 'off'. (Without the apostrophes!).

14.2 Manual logging

All functions except the trade related functions can be logged. The following code logs the production function and the change of the production from last year:

```
output = self.produce(self.inputs)
self.log('production', output)
self.log_change('production', output)
```

Log logs dictionaries. To log your own variable:

```
self.log('price', {'input': 0.8, 'output': 1})
```

Further you can write the change of a varibale between a start and an end point with: `observe_begin()` and `observe_end()`.

class `database.`**`Database`**
The database class

`log`(*action_name*, *data_to_log*)
With log you can write the models data. Log can save variable states and and the working of individual functions such as production, consumption, give, but not trade(as its handled automatically).

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Example:

```
self.log('profit', {'': profit})

... different method ...

self.log('employment_and_rent', {'employment': self.possession('LAB'),
                                  'rent': self.possession('CAP'), 'composite': self.composite

for i in range(self.num_households):
    self.log('give%i' % i, self.give('Household', i, 'money', payout / self.num_households))
```

See also:

`log_nested()`: handles nested dictianaries

`log_change()`: loges the change from last round

`observe_begin():`

`log_change`(*action_name*, *data_to_log*)

This command logs the change in the variable from the round before. Important, use only once with the same action_name.

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Examples:

```
self.log_change('profit', {'money': self.possession('money')]})
self.log_change('inputs', {'money': self.possessions(['money', 'gold', 'CAP', 'LAB')]})
```

`log_dict`(*action_name*, *data_to_log*)

same as the log function, only that it supports nested dictionaries see: `log()`.

`log_value`(*name*, *value*)

logs a value, with a name

Args:

'name'(string): the name of the value/variable

value(int/float): the variable = value to log

`observe_begin`(*action_name*, *data_to_observe*)

observe_begin and observe_end, observe the change of a variable. observe_begin(...), takes a list of variables to be observed. observe_end(...) writes the change in this variables into the log file

you can use nested observe_begin / observe_end combinations

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Example:

```
self.log('production', {'composite': self.composite,
                        self.sector: self.final_product[self.sector]})

... different method ...

self.log('employment_and_rent', {'employment': self.possession('LAB'),
                                'rent': self.possession('CAP')})
```

observe_end(*action_name*, *data_to_observe*)

This command puts in a database called log, whatever values you want values need to be delivered as a dictionary:

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Example:

```
self.log('production', {'composite': self.composite,
                        self.sector: self.final_product[self.sector]})

... different method ...

self.log('employment_and_rent', {'employment': self.possession('LAB'),
                                'rent':self.possession('CAP')})
```

14.3 Panel Data

Simulation.**panel_data**(*group*, *variables='goods'*, *typ='FLOAT'*, *command='round_end'*)

Ponel_data writes variables of a group of agents into the database, by default the db write is at the end of the round. You can also specify a command and insert the command you choose in the action_list. If you choose a custom command, you can declare a method that returns the variable you want to track. This function in the class of the agent must have the same name as the command.

You can use the same command for several groups, that report at the same time.

Args:

group: can be either a group or 'all' for all agents

variables (optional): default='goods' monitors all the goods the agent owns you can insert any variable your agent possesses. For self.knows_latin you insert 'knows_latin'. If your agent has self.technology you can use 'technology['formula']' In this case you must set the type to CHAR(50) with the typ='CHAR(50)' parameter.

typ: the type of the sql variable (FLOAT, INT, CHAR(length)) command

Example in start.py:

```
w.panel_data(group='Firm', command='after_production')

or

w.panel_data(group=firm)
```

Optional in the agent:

```
class Firm(AgentEngine):

...
def after_production(self):
    track = {}
    track['t'] = 'yes'
    for key in self.prices:
        track['p_' + key] = self.prices[key]
    track.update(self.product[key])
    return track
```

CHAPTER

FIFTEEN

FIRMMULTITECHNOLOGIES

The FirmMultiTechnologies class allows you to set up firm agents with complex or several production functions. While the simple Firm automatically handles one technology, FirmMultiTechnologies allows you to manage several technologies manually.

The create_* functions allow you to create a technology and assign it to a variable. `abce.FirmMultiTechnologies.produce()` and similar methods use this variable to produce with the according technology.

class `abce.`**`FirmMultiTechnologies`**

`create_cobb_douglas`(*output*, *multiplier*, *exponents*)

creates a Cobb-Douglas production function

A production function is a production process that produces the given input goods according to the formula to the output good. Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

Args: 'output': Name of the output good multiplier: Cobb-Douglas multiplier {'input1': exponent1, 'input2': exponent2 ...}: dictionary containing good names 'input' and corresponding exponents

Returns: A production_function that can be used in produce etc.

Example: self.plastic_production_function = self.create_cobb_douglas('plastic', {'oil' : 10, 'labor' : 1}, 0.000001) self.produce(self.plastic_production_function, {'oil' : 20, 'labor' : 1})

`create_leontief`(*output*, *utilization_quantities*, *isinteger=''*)

creates a Leontief production function

A production function is a production process that produces the given input goods according to the formula to the output good. Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

Warning, when you produce with a Leontief production_function all goods you put in the produce(...) function are used up. Regardless whether it is an efficient or wasteful bundle

Args:

'output': Name of the output good

utilization_quantities: a dictionary containing good names and corresponding exponents

isinteger='int' or isinteger='': When 'int' produce only integer amounts of the good. When '', produces floating amounts. (default)

Returns: A production_function that can be used in produce etc.

Example: self.car_technology = self.create_leontief('car', {'tire' : 4, 'metal' : 1000, 'plastic' : 20}, 1) two_cars = {'tire': 8, 'metal': 2000, 'plastic': 40} self.produce(self.car_technology, two_cars)

`create_production_function` (*formula*, *typ='from_formula'*)

creates a production function from formula

A production function is a production process that produces the given input goods according to the formula to the output goods. Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

create_production_function_fast is faster but more complicated

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are separated by a ;

Returns: A production_function that can be used in produce etc.

Example: formula = 'golf_ball = (ball) * (paint / 2); waste = 0.1 * paint' self.production_function = self.create_production_function(formula) self.produce(self.production_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

`create_production_function_fast` (*formula*, *output_goods*, *input_goods*, *typ='from_formula'*)

creates a production function from formula, with given outputs

A production function is a production process that produces the given input goods according to the formula to the output goods. Production_functions are then used as an argument in produce, predict_vector_produce and predict_output_produce.

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are separated by a ; [output]: list of all output goods (left hand sides of the equations)

Returns: A production_function that can be used in produce etc.

Example: formula = 'golf_ball = (ball) * (paint / 2); waste = 0.1 * paint' self.production_function = self.create_production_function(formula, 'golf', ['waste', 'paint']) self.produce(self.production_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

`net_value` (*goods_vector*, *price_vector*)

Calculates the net_value of a goods_vector given a price_vector

> goods_vectors are vector, where the input goods are negative and the output goods are positive. When we multiply every good with its according price we can calculate the net_value of the corresponding production. goods_vectors are produced by predict_produce(.)

Args: goods_vector: a dictionary with goods and quantities e.G. {'car': 1, 'metal': -1200, 'tire': -4, 'plastic': -21} price_vector: a dictionary with goods and prices (see example)

Example:

```
prices = {'car': 50000, 'tire': 100, 'metal': 10, 'plastic':  0.5}
value_one_car = net_value(predict_produce(car_production_function, one_car), prices)
value_two_cars = net_value(predict_produce(car_production_function, two_cars), prices)
if value_one_car > value_two_cars:
   produce(car_production_function, one_car)
else:
   produce(car_production_function, two_cars)
```

predict_produce (*production_function*, *input_goods*)
Returns a vector with input (negative) and output (positive) goods

> Predicts the production of produce(production_function, input_goods) and the use of input goods. net_value(.) uses a price_vector (dictionary) to calculate the net value of this production.

Args: production_function: A production_function produced with create_production_function, create_cobb_douglas or create_leontief {'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good used for the production.

Example:

```
prices = {'car': 50000, 'tire': 100, 'metal': 10, 'plastic':  0.5}
value_one_car = net_value(predict_produce(car_production_function, one_car), prices)
value_two_cars = net_value(predict_produce(car_production_function, two_cars), prices)
if value_one_car > value_two_cars:
   A.produce(car_production_function, one_car)
else:
   A.produce(car_production_function, two_cars)
```

predict_produce_output (*production_function*, *input_goods*)
Predicts the output of a certain input vector and for a given production function

> Predicts the production of produce(production_function, input_goods) see also: Predict_produce(.) as it returns a calculatable vector

Args:

```
production_function: A production_function produced with
create_production_function, create_cobb_douglas or create_leontief
{'input_good1': amount1, 'input_good2': amount2 ...}: dictionary
containing the amount of input good used for the production.
```

Returns:

```
A dictionary of the predicted output
```

Example:

```
print(A.predict_output_produce(car_production_function, two_cars))
>>> {'car': 2}
```

produce (*production_function*, *input_goods*)
Produces output goods given the specified amount of inputs.

Transforms the Agent's goods specified in input goods according to a given production_function to output goods. Automatically changes the agent's belonging. Raises an exception, when the agent does not have sufficient resources.

Args:

production_function: A production_function produced with py:meth:~*abceagent.FirmMultiTechnologies..create_prod*
py:meth:~*abceagent.FirmMultiTechnologies..create_cobb_douglas* or
py:meth:~*abceagent.FirmMultiTechnologies..create_leontief*

input goods {dictionary}: dictionary containing the amount of input good used for the production.

Raises: NotEnoughGoods: This is raised when the goods are insufficient.

Example:

```
car = {'tire': 4, 'metal': 2000, 'plastic':  40}
bike = {'tire': 2, 'metal': 400, 'plastic':  20}
try:
    self.produce(car_production_function, car)
except NotEnoughGoods:
    A.produce(bike_production_function, bike)
```

produce_use_everything(*production_function*)

Produces output goods from all input goods, used in this production_function, the agent owns.

Args:

```
production_function: A production_function produced with
py:meth:`~abceagent.FirmMultiTechnologies.create_production_function`, py:meth:`~abceagent.F
py:meth:`~abceagent.FirmMultiTechnologies.create_leontief`
```

Example:

```
self.produce_use_everything(car_production_function)
```

sufficient_goods(*input_goods*)

checks whether the agent has all the goods in the vector input

CHAPTER

SIXTEEN

THE SIMULATION IN START.PY

The best way to start creating a simulation is by copying the start.py file and other files from ‘abce/template’.

To see how to create a simulation read *Walk through / Tutorial*. In this module you will find the explanation for the command.

This is a minimal template for a start.py:

```
from __future__ import division  # makes / division work correct in python !
from agent import Agent
from abce import *


for parameters in read_parameters('simulation_parameters.csv'):
    s = Simulation(parameters)
    action_list = [
    ('all', 'one'),
    ('all', 'two'),
    ('all', 'three'),
    ]
    s.add_action_list(action_list)
    s.build_agents(Agent, 2)
    s.run()
```

class `abce.`**`Simulation`**(*simulation_parameters*)

This class in which the simulation is run. It takes the simulation_parameters to set up the simulation. Actions and agents have to be added. databases and resource declarations can be added. Then runs the simulation.

Usually the parameters are specified in a tab separated csv file. The first line are column headers.

Args:

```
simulation_parameters: a dictionary with all parameters. "name" and
"num_rounds" are mandatory.
```

Example::

for simulation_parameters in read_parameters(‘simulation_parameters.csv’): action_list = [(‘household’, ‘recieve_connections’), (‘household’, ‘offer_capital’), (‘firm’, ‘buy_capital’), (‘firm’, ‘production’), (‘household’, ‘buy_product’) ‘after_sales_before_consumption’ (‘household’, ‘consume’)] w = Simulation(simulation_parameters) w.add_action_list(action_list) w.build_agents(Firm, ‘firm’, ‘num_firms’) w.build_agents(Household, ‘household’, ‘num_households’)

w.declare_round_endowment(resource=’labor_endowment’, productivity=1, product=’labor’) w.declare_round_endowment(resource=’capital_endowment’, productivity=1, product=’capital’) w.panel_data('firm', command='after_sales_before_consumption')

w.run()

add_action_list (*action_list*)

add an *action_list*, which is a list of either:

- tuples of an goup_names and and action
- a single command string for panel_data or follow_agent
- [tuples of an agent name and and action, currently not unit tested!]

Example:

```
action_list = [
   repeat([
           ('Firm', 'sell'),
           ('Household', 'buy')
       ],
       repetitions=10
   ),
   ('Household_03', 'dance')
   'panel_data_end_of_round_befor consumption',
   ('Household', 'consume'),
   ]
w.add_action_list(action_list)
```

add_action_list_from_file (*parameter*)

The action list can also be declared in the simulation_parameters.csv file. Which allows you to run a batch of simulations with different orders. In simulation_parameters.csv there must be a column with which contains the a declaration of the action lists:

num_rounds	num_agents	action_list	endow-ment
1000,	10,	[('firm', 'sell'), ('household', 'buy')],	(5,5)
1000,	10,	[('firm', 'buf'), ('household', 'sell')],	(5,5)
1000,	10,	[('firm', 'sell'), ('household', 'calculate_net_wealth'), ('household', 'buy')],	(5,5)

The command:

```
self.add_action_list_from_file('parameters['action_list'])
```

Args:

```
parameter
    a string that contains the action_list. The string can be read
    from the simulation_parameters file: parameters['action_list'], where action_list
    is the column header in simulation_parameters.
```

ask_agent (*group*, *idn*, *command*)

This is only relevant when you derive your custom world/swarm not in start.py applying a method to a single agent

Args:

```
agent_name as string or using agent_name('group_name', number)
method: as string
```

ask_each_agent_in (*group_name*, *command*)

This is only relevant when you derive your custom world/swarm not in start.py applying a method to a group of agents group_name, method.

Args:

```
agent_group: using group_address('group_name', number)
method: as string
```

build_agents (*AgentClass*, *number=None*, *group_name=None*, *agents_parameters=None*)

This method creates agents, the first parameter is the agent class. "num_agent_class" (e.G. "num_firm") should be difined in simulation_parameters.csv. Alternatively you can also specify number = 1.s

Args:

```
AgentClass: is the name of the AgentClass that you imported
number (optional): number of agents to be created. or the colum name
of the row in simulation_parameters.csv that contains this number. If not
specified the column name is assumed to be 'num_' + agent_name
(all lowercase). For example num_firm, if the class is called
Firm or name = Firm.
[group_name (optional): to give the group a different name than the
class_name. (do not use this if you have not a specific reason]
```

Example:

```
w.build_agents(Firm, number='num_firms')
# 'num_firms' is a column in simulation_parameters.csv
w.build_agents(Bank, 1)
w.build_agents(CentralBank, number=1)
```

build_agents_from_file (*AgentClass*, *parameters_file=None*, *multiply=1*, *delimiter='t'*, *quotechar=""*)

This command builds agents of the class AgentClass from an csv file. This way you can build agents and give every single one different parameters.

The file must be tab separated. The first line contains the column headers. The first column "agent_class" specifies the agent_class. The second column "number" (optional) allows you to create more than one agent of this type. The other columns are parameters that you can access in own_parameters the __init__ function of the agent.

Agent created from a csv-file:

```
class Agent(AgentEngine):
   def __init__(self, simulation_parameter, own_parameters, _pass_to_engine):
       AgentEngine.__init__(self, *_pass_to_engine)
       self.size = own_parameters['firm_size']
```

declare_aesof (*aesof_file='aesof.csv'*)

AESOF lets you determine agents behaviour from an comma sepertated sheet.

First row must be column headers. There must be one column header 'round' and a column header name. A name can be a goup are and individal (goup_id e.G. firm_01) it can also be 'all' for all agents. Every round, the agents self.aesof parameters get updated, if a row with the corresponding round and agent name exists.

Therefore an agent can access the parameters *self.aesof[column_name]* for the current round. (or the precedent one when there was no update) parameter is set. You can use it in your source code. It is persistent until the next round for which a corresponding row exists.

You can alse put commands or call methods in the excel file. For example:
self.aesof_exec(column_name).

Alternatively you can declare a variable according to a function: *willingness_to_pay = self.aesof_eval(column_name)*.

There is a big difference between *self.aesof_exec* and *self.aesof_eval*. exec is only executed in rounds that have corresponding rows in aesof.csv. *self.aesof_eval* is persistent every round the expression of the row corresponding to the current round round or the last declared round is executed.

Args:

aesof_file(optional): name of the csv_file. Default is the group name plus 'aesof.csv'.

declare_perishable (*good*, *command='perish_at_the_round_end'*)
This good only lasts one round and then disappears. For example labor, if the labor is not used today today's labor is lost. In combination with resource this is useful to model labor or capital.

In the example below a worker has an endowment of labor and capital. Every round he can sell his labor service and rent his capital. If he does not the labor service for this round and the rent is lost.

Args:

```
good: the good that perishes
[command: In order to perish at another point in time you can choose
a commmand and insert that command in the action list.

Example::

    w.declare_round_endowment(resource='LAB_endowment', productivity=1000, product='LAB')
    w.declare_round_endowment(resource='CAP_endowment', productivity=1000, product='CAP')
    w.declare_perishable(good='LAB')
    w.declare_perishable(good='CAP')
```

declare_round_endowment (*resource*, *productivity*, *product*, *command='default_resource'*, *group='all'*)
Every round the agent gets 'productivity' units of good 'product' for every 'resource' he possesses.

By default the this happens at the beginning of the round. You can change this. Insert the command string you chose it self.action_list. One command can be associated with several resources.

Round endowments can be goup specific, that means that when somebody except this group holds them they do not produce. The default is 'all'. Restricting this to a group could have small speed gains.

panel_data (*group*, *variables='goods'*, *typ='FLOAT'*, *command='round_end'*)
Ponel_data writes variables of a group of agents into the database, by default the db write is at the end of the round. You can also specify a command and insert the command you choose in the action_list. If you choose a custom command, you can declare a method that returns the variable you want to track. This function in the class of the agent must have the same name as the command.

You can use the same command for several groups, that report at the same time.

Args:

group: can be either a group or 'all' for all agents

variables (optional): default='goods' monitors all the goods the agent owns you can insert any variable your agent possesses. For self.knows_latin you insert 'knows_latin'. If your agent has self.technology you can use 'technology['formula']' In this case you must set the type to CHAR(50) with the typ='CHAR(50)' parameter.

typ: the type of the sql variable (FLOAT, INT, CHAR(length)) command

Example in start.py:

```
w.panel_data(group='Firm', command='after_production')

or

w.panel_data(group=firm)
```

Optional in the agent:

```
class Firm(AgentEngine):

...
def after_production(self):
    track = {}
    track['t'] = 'yes'
    for key in self.prices:
        track['p_' + key] = self.prices[key]
    track.update(self.product[key])
    return track
```

run()
This runs the simulation

abce.**read_parameters**(*parameters_file='simulation_parameters.csv'*, *delimiter='\t'*, *quotechar='"'*)
reads a parameter file line by line and gives a list. Where each line contains all parameters for a particular run of the simulation.

Args:

parameters_file (optional): filename of the csv file. (default:*simulation_parameters.csv*)

delimiter (optional): delimiter of the csv file. (default: tabs)

quotechar (optional): for single entries that contain the delimiter. (default: ") See python csv lib http://docs.python.org/library/csv.html

This code reads the file and runs a simulation for every line:

```
for parameter in read_parameters('simulation_parameters.csv'):
   w = Simulation(parameter)
   w.build_agents(Agent, 'agent', 'num_agents')
   w.run()
```

class abce.**repeat**(*action_list*, *repetitions*)
Repeats the contained list of actions several times

Args:

action_list: action_list that is repeated

repetitions: the number of times that an actionlist is repeated or the name of the corresponding parameter in simulation_parameters.csv

Example with number of repetitions in simulation_parameters.csv:

```
action_list = [
    repeat([
            ('firm', 'sell'),
            ('household', 'buy')
        ],
        repetitions=parameter['number_of_trade_repetitions']
    ),
    ('household_03', 'dance')
    'panel_data_end_of_round_before consumption',
    ('household', 'consume'),
    ]
s.add_action_list(action_list)
```

class abce.**repeat_while**(*action_list*, *repetitions=None*)

NOT IMPLEMENTED Repeats the contained list of actions until all agents_risponed done. Optional a maximum can be set.

Args:

```
action_list: action_list that is repeated
repetitions: the number of times that an actionlist is repeated or the name of
the corresponding parameter in simulation_parameters.csv
```

CHAPTER

SEVENTEEN

SIMULATION_PARAMETERS AND AGENT_PARAMETERS

In the file *simulation_parameters.csv*, parameters that either govern the simulation or are accessible to all agents, can be specified. The file *agent_parameters.csv* is used to create agents from a file with `abce.Simulation.build_agents_from_file()`. The agents created can access the agent_parameters for their agent group. For every simulation all agents are specified in the same file. Even though, they are build separately.

We will first expose the compulsory columns in simulation_parameters.csv and agent_parameters respectively and then show how agents can access the parameters.

There are a few conventions: - The files must be tab separated - First row has column headings - All lower case - *num_* indicates number of

17.1 simulation_parameters.csv

compulsory fields:

name: name of the simulation

num_rounds: number of rounds of this simulation

trade_logging: Can be set to *group* (fast) or *individual* (slow, default) or *off*)

optional fields:

random_seed: random seed 0 or missing chooses a random_seed at random

17.2 agent_parameters.csv

This table it self does not create the agents. Rather `abce.Simulation.build_agents_from_file()` creates the agent. build_agents_from_file, searches for the line(s) with the agent_class, specified. It then creates the number of agents of this class, specified in the number column. There can be several lines with the same agent class, in this case for each line the the number of agents are created. These agents get the *agent_parameters* specified in the particular line.

compulsory fields:

agent_class: name of the agent's agent class.

number: number of agents for this class.

17.3 Accessing parameters in agents

Agents can only access the parameters in the __init__ method. You can store parameter separately or store all of them:

storing single parametern:

```
class Firm(abceagent.Agent, abceagent.Firm):
   def __init__(self, simulation_parameters, agent_parameters, _pass_to_engine):
       abceagent.Agent.__init__(self, *_pass_to_engine)

       self.sector = agents_pameters['sector']
       # saves the value or string from the column sector, for this class of agents

       self.gravity = simulation_parameters['gravity']
       # saves the simulation parameter gravity

       self.num_households = simulation_parameters['num_household']
       # saves the auto generated number of agents of the type household
```

As you can see above there is an **autogenerated simulation parameter 'num_agent_class'** for every agent class that is created.

You can also store all parameters as dictionary, but that is not recommended:

```
def __init__(self, simulation_parameters, agent_parameters, _pass_to_engine):
    abceagent.Agent.__init__(self, *_pass_to_engine)
    self.simulation_parameters = simulation_parameters
    self.agent_parameters = agent_parameters
    print(simulation_parameters)
```

CHAPTER

EIGHTEEN

RETRIEVAL OF THE SIMULATION RESULTS

Agents can log their internal states and the simulation can create panel data. `abce.database`.

the results are stored in a subfolder of the ./results/ folder.

The tables are stored as '.csv' files which can be opened with excel and libreoffice. Further you can import the files with R, which also gives you a social accounting matrix:

1. **start a in the subfolder of ./results/ that contains your simulation** results
2. start R
3. *load('database.R')*

The same data is also as a sqlite database 'database.db' available. It can be opened by 'sqlitebrowser' in ubuntu.

CHAPTER

NINETEEN

TOOLS AND FLOATING POINT ISSUES: ABCETOOLS MODULE

This file contains functions to compare floating point variables to 0. All variables in this simulation as in every computer programm are floating point variables. Floating point variables are not exact. Therefore var_a == var_b has no meaning. Further a variable that is var_c = 9.999999999999966e-30 is for our purpose equal to zero, but var_c == 0 would lead to False. `is_zero()`, `is_positive()` and `is_negative()` work around this problem by defining float epsilon and determine whether the variable is sufficiently close to zero or not.

This file also defines the `tools.NotEnoughGoods`

exception `tools.`**`NotEnoughGoods`**(*_agent_name*, *good*, *amount_missing*)

Methods raise this exception when the agent has less goods than needed

These functions (self.produce, self.offer, self.sell, self.buy) should be encapsulated by a try except block:

```
try:
   self.produce(...)
except NotEnoughGoods:
   alternative_statements()
```

`tools.`**`agent_name`**(*group_name*, *idn*)

Given a group name and a id-number it returns the agent_name of the individual agent with the number idn. A message sent to the agent_name, will be received by this individual agent

`tools.`**`is_negative`**(*x*)

see is positive

`tools.`**`is_positive`**(*x*)

checks whether a number is positive and sufficiently different from zero. All variables in ABCE are floating point numbers. Due to the workings of floating point arithmetic. If x is 1.0*e^-100 so really close to 0, x > 0 will be true, eventhough it is very very small; is_zero will be true.

`tools.`**`is_zero`**(*x*)

checks whether a number is sufficiently close to zero. All variables in ABCE are floating point numbers. Due to the workings of floating point arithmetic. If x is 1.0*e^-100 so really close to 0, x == 0 will be false; is_zero will be true.

CHAPTER

TWENTY

FILES IN THIS PACKAGE

abce/abce: the actual ABCE engine, you have to import the modules in this directory

abce/template: a template from which you can start writing your own simulation including start.py, agents and parameter files

abce/examples: a series of example simulations

abce/unitest: Unit testing is a way to ensure that your software works as specified. The unit test simulation ensures, that all functions of the simulation and the agent's base classes work correctly.

abce/doc: The documentation source code.

USING AMAZON ELASTIC CLOUD SERVER

1. Create an amazon ec2 account
2. Launch an ubuntu instance
3. Open two bash / command lines. On for your local computer one to login to the server
4. In the management console find your server address

(replace the addesses below with ubuntu@your_server_string.compute.amazonaws.com)

Log on the amazon server:

```
ssh -i ./ec2/ec2.pem ubuntu@ec2-79-125-32-99.eu-west-1.compute.amazonaws.com
```

on amazon ec2 ubuntu server:

```
mkdir cce
mkdir abce
mkdir abce/lib
```

on local computer:

```
SPATH="ubuntu@ec2-79-125-32-99.eu-west-1.compute.amazonaws.com"
scp -r -i ./ec2/ec2.pem ./abce/lib/*.py  $SPATH:~/abce/lib/
scp -r -i ./ec2/ec2.pem ./cce/*.py  $SPATH:~/cce/
scp -r -i ./ec2/ec2.pem ./cce/*.csv  $SPATH:~/cce/
```

To shut down your ubuntu instance from the command line:

```
sudo shutdown -P now
```

CHAPTER TWENTYTWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX