Agent-Based Computational Economics Documentation

Release 0.8.2alpha0

Davoud Taghawi-Nejad

Sep 05, 2017

Contents

1 Introduction **3**
- 1.1 Introduction . . . 3
- 1.2 Download and Installation . . . 7
- 1.3 Interactive jupyter / IPython notebook Tutorial . . . 9
- 1.4 Creating the Agent . . . 9
- 1.5 Giving a Good . . . 10
- 1.6 Trade . . . 10
- 1.7 Lets capture data . . . 11
- 1.8 Using statistical software . . . 13
- 1.9 Walk through . . . 14
- 1.10 Tutorial for Plant Modeling . . . 23
- 1.11 Examples . . . 26
- 1.12 unit testing . . . 30

2 Simulation Programming **31**
- 2.1 The simulation in start.py . . . 31
- 2.2 Agents . . . 35
- 2.3 Physical goods and services . . . 38
- 2.4 Trade . . . 38
- 2.5 Messaging . . . 43
- 2.6 Firm and production . . . 44
- 2.7 Household and consumption . . . 49
- 2.8 Observing agents and logging . . . 51
- 2.9 Retrieval of the simulation results . . . 54
- 2.10 NotEnoughGoods Exception . . . 54

3 Advanced **55**
- 3.1 Contracting . . . 55
- 3.2 FirmMultiTechnologies . . . 58
- 3.3 Quote . . . 62
- 3.4 Spatial and Netlogo like Models . . . 64

4 Graphical User Interface and Results **69**
- 4.1 Graphical User Interface . . . 69
- 4.2 Files in this package . . . 71
- 4.3 Deploying an ABCE simulation on-line . . . 71

5 Indices and tables 75

ABCE is a Python Agent-Based Computational Economy Platform, written by Davoud Taghawi-Nejad. The impatient reader can jump directly to the 'Interactive jupyter / IPython notebook Tutorial', which explains how to set up a simulation. In the walk through you will learn how to set up an agent and how to trade with other agents. The Household and Firm classes allow to produce with different production functions and consume with utility functions. But models don't have to use neoclassical assumptions.

ABCE runs on macOS, Windows, and Linux. ABCE runs 10x faster on pypy!

CHAPTER 1

Introduction

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 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 modeler 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. The speed advantages of ABCE are typically only observed for 10000 agents and more. Below, it might be between half as fast to equally fast to a pure python implementation.

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].

[1] the Simulation class

[2] with out the equilibrium of course

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. ABCE uses C++, to handle background tasks to increase speed. Python allows simple, but fully functional, programming for economists. What is more Python is readable even for non Python programmers.

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 and using C++ for background tasks.

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.

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. ABCE allows 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 0.6 supports Python 3.

For the simulated problem all agents are executed in parallel. This is achieved by randomizing the arrival of messages and orders between sub-rounds. For example if in one sub-round all agents make offers and in the next sub-round all agents receive and answer the offers, the order in which the agents receive is random, as if the agent's in the round before would make offers in a random order.

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 it is it's 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.

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[4] 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 SugarScape and, in a more limited sense, Repast are designed with spacial representation in mind. For ABCE a spacial representation is possible, but 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

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

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 good choice.

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.
- **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.

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.

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

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[5] general programming language. Consequently it is faster than NetLogo.

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.

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.

Download and Installation

ABCE works exclusively with python 3!

Installation Ubuntu

1. If python3 and pip not installed in terminal[2]

   ```
   sudo apt-get install python3
   sudo apt-get install python3-pip
   ```

2. In terminal:

   ```
   sudo python3 -m pip install abce
   ```

3. download and unzip the zip file with examples and the template from: https://github.com/AB-CE/examples

4. Optional for a 10 fold speed increase:

   Install pypy3 from https://pypy.org/download.html

5. Install pypy3 additionally:

   sudo pypy3 -m pip install abce

[5] Python contrary to the common believe is compiled to bytecode similar to Java's compilation to bytecode.
[2] If this fails `sudo apt-add-repository universe` and `sudo apt-get update`

6. For pypy execute models with `pypy3 start.py` instead of `python3 start.py`

Installation Mac

1. If you are on OSX Yosemite, download and install: `Command line Tools (OS X 10.10) for XCODE 6.4` from https://developer.apple.com/downloads/
2. If pip not installed in terminal:

```
sudo python3 -m easy_install pip
```

3. In terminal:

```
sudo python3 -m pip install abce
```

4. If you are on El Capitain, OSX will ask you to install cc - xcode "Command Line Developer Tools", click accept.[1]
5. If XCODE was installed type again in terminal:

```
sudo python3 -m pip install abce
```

6. download and unzip the zip file with examples and the template from: https://github.com/AB-CE/examples
7. Optional for a 10 fold speed increase:

   Install pypy3 from https://pypy.org/download.html
8. Install pypy3 additionally:

   sudo pypy3 -m pip install abce
9. For pypy execute models with `pypy3 start.py` instead of `python3 start.py`

Installation Windows

ABCE works best with anaconda python 3.5 follow the instructions blow.

1. Install the **python3.5** anaconda distribution from https://continuum.io/downloads
3. anaconda prompt or in the command line (cmd) type:

```
pip install abce
```

3. download and unzip the zip file with examples and the template from: https://github.com/AB-CE/examples

Known Issues

- When you run an IDE such as spyder sometimes the website blocks. In

order to avoid that, modify the 'Run Setting' and choose 'Execute in external System Terminal'.

- When the simulation blocks, there is probably a simulation.finalize() command

missing after the simulation loop

[1] xcode 7 works only on OSX El Capitan. You need to either upgrade or if you want to avoid updating download xcode 6.4 from here: https://developer.apple.com/downloads/

If you have any problems with the installation

Mail to: DavoudTaghawiNejad@gmail.com

Interactive jupyter / IPython notebook Tutorial

This tutorial works on jupyter notebook. Please download examples from here: https://github.com/AB-CE/examples and load the jupyter_tutorial by changing in the `examples/jupyter_tutorial` folder and typing `jupyter notebook jupyter_tutorial.ipynb`. Then you can execute the notebook cell by cell and observe ABCE at work.

Creating the Agent

We create a simple agent and intialize it. An agent is an object that has propeties and does things. In python that is a class. We create an agent that has a name, knows the world size and can say his name.

The agent needs to inherit from abce.Agents. *The ''__init__'' function should not be overwritten, instead create an ''init'' function*. The init function will be called during initialisation. The agents gets two parameters. All agents get the parameters we will specify in the next round. From agent_parameters the agents get only agent_parameters[id], where id is the number of the agent. Agents have for now only one function. They can say their name.

From the abstract agent, we create four concrete agents.

`agents` allows you to call each agents in the agents group. Each agent in this group has a group name, which is 'agent' and an id. Further we give each of the agents a parameter a family_name. Run the simulation and let all agents say their name:

```
Round0
hello I am fred my id 0 and my group is 'agent', it is the 0 round
hello I am astaire my id 1 and my group is 'agent', it is the 0 round
hello I am altair my id 2 and my group is 'agent', it is the 0 round
hello I am deurich my id 3 and my group is 'agent', it is the 0 round
Round1
hello I am fred my id 0 and my group is 'agent', it is the 1 round
hello I am astaire my id 1 and my group is 'agent', it is the 1 round
hello I am altair my id 2 and my group is 'agent', it is the 1 round
hello I am deurich my id 3 and my group is 'agent', it is the 1 round
Round2
hello I am fred my id 0 and my group is 'agent', it is the 2 round
hello I am astaire my id 1 and my group is 'agent', it is the 2 round
hello I am altair my id 2 and my group is 'agent', it is the 2 round
hello I am deurich my id 3 and my group is 'agent', it is the 2 round
Round3
hello I am fred my id 0 and my group is 'agent', it is the 3 round
hello I am astaire my id 1 and my group is 'agent', it is the 3 round
hello I am altair my id 2 and my group is 'agent', it is the 3 round
hello I am deurich my id 3 and my group is 'agent', it is the 3 round
Round4
hello I am fred my id 0 and my group is 'agent', it is the 4 round
hello I am astaire my id 1 and my group is 'agent', it is the 4 round
hello I am altair my id 2 and my group is 'agent', it is the 4 round
hello I am deurich my id 3 and my group is 'agent', it is the 4 round
```

It is necessary to tell the simulation when a new round starts and what time it is simulation.advance_round(r) does that. The parameter can be any representation of time.

Giving a Good

ABCE provide goods. Goods are things that can be given, sold or transformed. We create 5 agents, the first one has a ball the agents pass the ball around.

self.create, creates an object. self.possession, checks how much of one object an agent has. self.give, gives an object to another agent, specied by its group name and its id.

When agent_parameters is not specified the numer of agents to be created needs to be spezified

```
Round0
....
Round1
....
Round2
..*..
Round3
...*.
Round4
....*
Round5
....
Round6
....
```

Trade

Well in every school yard we have a drug dealer.

The new kids, approach a random drug dealer and offer him 10 bucks.

Drug dealer look at all the sell offers they get and decide to sell only to those kids that are willing to give them at least 10 dollars.

builds 1 drug dealer and one customer.

Groups of agents can be merged to 'super' groups. We will print the amount of drugs and money all kids have for each of the two kids

```
Round0
Customer offers 10 dollar:
    drug_dealer{'money': 0, 'drugs': 1.0}
    customer{'money': 90.0}
Drug Dealer accepts or rejects the offer:
    drug_dealer{'money': 0, 'drugs': 1.0}
    customer{'money': 100.0}

Round1
Customer offers 10 dollar:
    drug_dealer{'money': 0, 'drugs': 1.0}
    customer{'money': 90.0}
Drug Dealer accepts or rejects the offer:
```

```
    drug_dealer{'money': 0, 'drugs': 1.0}
    customer{'money': 100.0}
```

When looking at round one one can see that after the customer offered 10 dollars, the 10 dollars are not available to him util the deal has either been accepted or rejected. After the drug dealer accepts the offer in the 0 round. The money is transfered to the drug dealer and the drugs to the customer.

In round 1, where the drug dealer runs out of drugs the 10 dollars go back to the customer.

Lets capture data

There are three ways of capturing data. `aggregate` and `panel` collect data from a specified group at a specified point of time. This has the advantage that there is no logging code in the agent class. `self.log('name', value)` saves a value under a certain name.

It is specified which agents group collects which variables and possessions.

Every round the groups need to be instructed to collect the according data. **simulation.finalize()** must be called after the simulation, to write the data! Otherwise the program hangs. Never forget to put **simulation.finalize()** otherwise the program will just block()

```
Round0
Round1
Round2
Round3
Round4
Round5
Round6
Round7
Round8
Round9
Round10
Round11
Round12
Round13
Round14
Round15
Round16
Round17
Round18
Round19
Round20
Round21

Round22time only simulation  94.36

Round23
Round24
Round25
Round26
Round27
Round28
Round29
Round30
Round31
Round32
```

```
Round33
Round34
Round35
Round36
Round37
Round38
Round39
Round40
Round41
Round42
Round43
Round44
Round45
Round46
Round47
Round48
Round49
Round50
Round51
Round52
Round53
Round54
Round55
Round56
Round57
Round58
Round59
Round60
Round61
Round62
Round63
Round64
Round65
Round66
Round67
Round68
Round69
Round70
Round71
Round72
Round73
Round74
Round75
Round76
Round77
Round78
Round79
Round80
Round81
Round82
Round83
Round84
Round85
Round86
Round87
Round88
Round89
Round90
```

```
Round91
Round92
Round93
Round94
Round95
Round96
Round97
Round98
Round99

time only simulation   1.72
time with data and network  94.42
{
    "name": "ipythonsimulation",
    "random_seed": 1504536068.345761,
    "num_kids": 5
}
time with data and network   2.09
{
    "name": "gatherdata",
    "random_seed": 1504536161.0385568
}
```

We can find the directory of the simulation data by using the `simulation.path` property

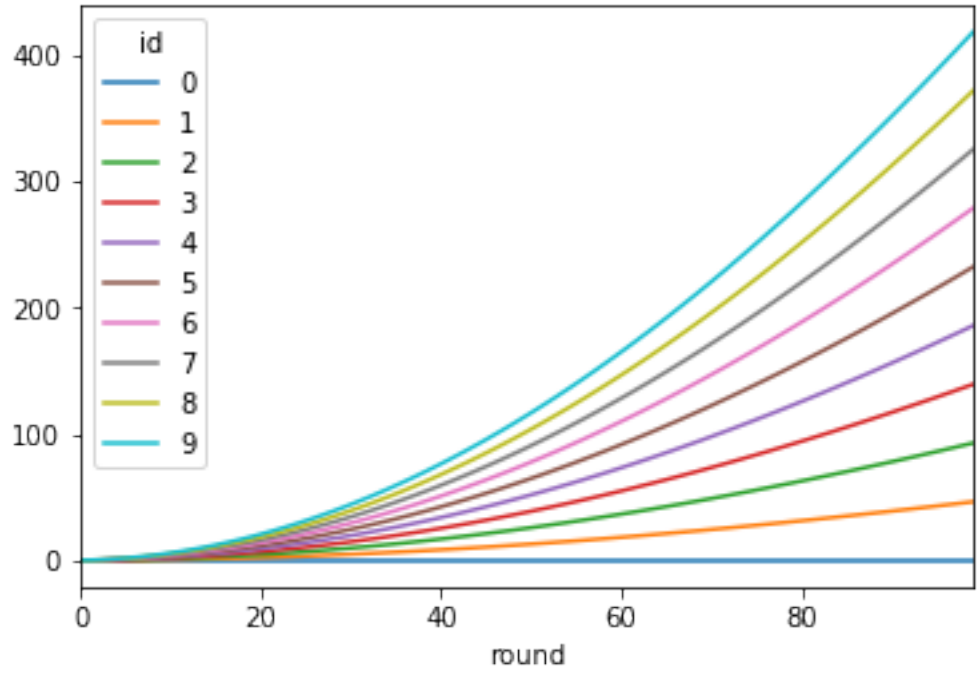
```
/Users/taghawi/Dropbox/workspace/abce_examples/examples/jupyter_tutorial/result/
↪gatherdata_2017-09-04_11-42
```

In that directory are the data files and a describtion.txt

```
['aggregate_datadealer.csv',
 'aggregated_datadealer.csv',
 'description.txt',
 'panel_datadealer.csv']
```

Using statistical software

```
<matplotlib.axes._subplots.AxesSubplot at 0x1104a5710>
```



When running a simulation with python from a start.py simulation.graphs() displays all recorded data. You can also use the @gui decorator to ship abce as an interactive web-app.

Walk through

In order to learn using ABCE we will now dissect and explain a simple ABCE model. Additional to this walk through you should also have a look on the examples in

<https://github.com/AB-CE/examples>(https://github.com/AB-CE/examples),

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
- 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.

start.py

```
""" 1. Build a Simulation
    2. Build one Household and one Firm follow_agent
    3. For every labor_endowment an agent has he gets one trade or usable labor
    per round. If it is not used at the end of the round it disappears.
    4. Firms' and Households' possessions are monitored to the points marked in
    timeline.
"""

from abce import Simulation, gui
from firm import Firm
from household import Household


def main():
    simulation = Simulation()

    simulation.declare_round_endowment(resource='labor_endowment', units=1, product='labor')
    simulation.declare_perishable(good='labor')

    firms = simulation.build_agents(Firm, 'firm', 1)
    households = simulation.build_agents(Household, 'household', 1)

    for r in range(100):
        simulation.advance_round(r)
        households.sell_labor()
        firms.buy_labor()
        firms.production()
        (households + firms).panel_log(possessions=['money', 'GOOD'])
        households.panel_log(variables=['current_utility'])
        firms.sell_goods()
        households.buy_goods()
        households.consumption()

    simulation.graphs()

if __name__ == '__main__':
    main()
```

It is of utter most importance to end with either simulation.graphs() or simulation.finalize()

A simulation with GUI

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.

```
from abce import Simulation, gui
from firm import Firm
from household import Household
```

Here the Agent class Firm is imported from the file firm.py. Likewise the Household class. Further the Simulation base class and the graphical user interface (gui) are imported

Parameters are specified as a python dictionary

```
parameters = {'name': '2x2',
              'random_seed': None,
              'rounds': 10,
              'slider': 100.0,
              'Checkbox': True,
              'Textbox': 'type here',
              'integer_slider': 100,
              'limited_slider': (20, 25, 50)}


@gui(parameters)
def main(parameters):
    . . .

if __name__ == '__main__':
    main(parameters)
```

The main function is generating and executing the simulation. When the main function is preceded with `@gui(simulation_parameters)` The graphical user interface is started in your browser the simulation_parameters are used as default values. If no browser window open you have to go manually to the address "http://127.0.0.1:8000/". The graphical user interface starts the simulation.

During development its often more practical run the simulation without graphical user interface (GUI). In order to switch of the GUI comment out the `#@gui(simulation_parameters)`. In order show graphs at the end of the simulation add `simulation.graphs()` after `simulation.run`, as it is done in start.py above.

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

```
simulation = Simulation(name="ABCE")

...
```

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 a group or several groups of agents act in parallel. In the code below you see a typical sub-round. Therefore after declaring the `Simulation` the order of actions, agents and objects are added.

```
for round in range(1000):
    simulation.advance_round(round)
    households.sell_labor()
    firms.buy_labor()
    firms.production()
    (households + firms).panel_log(...)
    firms.sell_goods()
    households.buy_goods()
    households.consumption()
```

This establishes the order of the simulation. Make sure you do not overwrite internal abilities/properties of the agents. Such as 'sell', 'buy' or 'consume'.

A more complex example could be:

```
for week in range(52):
    for day in ['mo', 'tu', 'we', 'th', 'fr']:
    simulation.advance_round((week, day))
    if day = 'mo':
        households.sell_labor()
        firms.buy_labor()
    firms.production()
    (households + firms).panel()
    for i in range(10):
        firms.sell_goods()
        households.buy_goods()
    households.consumption()
    if week == 26:
        government.policy_change()
```

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 at the beginning of the sub-round. If the trade was rejected: the blocked good is automatically unblocked.

Special goods and services

Now we will establish properties of special goods. A normal good can just be created or produced by an agent; it can also be destroyed, transformed or consumed by an agent. Some goods 'replenish' every round. And some goods 'perish' every round. These properties have to be declared:

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.

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

simulation.declare_round_endowment(resource='labor_endowment',
                                   units=1,
                                   product='labor'
    )
```

declare_round_endowment, establishes that at the beginning of every round, every agent that possesses x units of a resource, gets x*units units of the product. Every owner of x fields gets 100*x units of corn. Every owner of labor_endowment gets one unit of labor for each unit of labor_endowment he owns. An agent has to create the field or labor_endowment by `self.create('field', 5)`, for labor_endowment respectively.

```
simulation.declare_perishable('corn')
simulation.declare_perishable(good='labor')
```

declare_perishable, establishes that every unit of the specified good that is not used by the end of the round ceases to exist.

Declaring a good as replenishing and perishable is ABCE's way of treating services. In this example every household has some units of labor that can be used in the particular period. `abce.Simulation.declare_service()` is a synthetic way of declaring a good as a service.

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 `abce.Simulation.declare_round_endowment()`. In this way the economy is physically closed.

```
firms.panel_log(possessions=['good1', 'good2') # a list of firm possessions to track␣
↪here

households.agg_log('household', possessions=['good1', 'good2'],
                   variables=['utility']) #  a list of household variables to track␣
↪here
```

The possessions good1 and good2 are tracked, the agent's variable `self.utility` is tracked. There are several ways in ABCE to log data. Note that the variable names a strings.

Alternative to this you can also log within the agents by simply using *self.log('text', variable)* (`abce.Database.log()`) Or self.log('text', {'var1': var1, 'var2': var2}). Using one log command with a dictionary is faster than using several seperate log commands.

Having established special goods and logging, we create the agents:

```
simulation.build_agents(Firm, 'firm', number=simulation_parameters['number_of_firms'],
↪ parameters=simulation_parameters)
simulation.build_agents(Household, 'household', number=10, parameters=simulation_
↪parameters)
```

- Firm is the class of the agent, that you have imported
- 'firm' is the group_name of the agent
- number is the number of agents that are created
- parameters is a dictionary of parameters that the agent receives in the `init` function (which is discussed later)

```
simulation.build_agents(Plant, 'plant',
                        parameters=simulation_parameters,
                        agent_parameters=[{'type':'coal' 'watt': 20000},
                                          {'type':'electric' 'watt': 99}
                                          {'type':'water' 'watt': 100234}])
```

This builds three Plant agents. The first plant gets the first dictionary as a agent_parameter {'type':'coal' 'watt': 20000}. The second agent, gets the second dictionary and so on.

The agents

The Household agent

```
import abce


class Household(abce.Agent, abce.Household, abce.Trade):
    def init(self, simulation_parameters, agent_parameters):
        """ 1. labor_endowment, which produces, because of simulation.declare_
↪resource(...)
        in start.py one unit of labor per month
        2. Sets the utility function to utility = consumption of good "GOOD"
        """
        self.create('labor_endowment', 1)
        self.set_cobb_douglas_utility_function({"GOOD": 1})
        self.current_utility = 0

    def sell_labor(self):
```

```
        """ offers one unit of labor to firm 0, for the price of 1 "money" """
        self.sell('firm', 0,
                  good="labor",
                  quantity=1,
                  price=1)

    def buy_goods(self):
        """ receives the offers and accepts them one by one """
        oo = self.get_offers("GOOD")
        for offer in oo:
            self.accept(offer)

    def consumption(self):
        """ consumes_everything and logs the aggregate utility. current_utility
        """
        self.current_utility = self.consume_everything()
        self.log('HH', self.current_utility)
```

The Firm agent

```
import abce


class Firm(abce.Agent, abce.Firm, abce.Trade):
    def init(self, simulation_parameters, agent_parameters):
        """ 1. Gets an initial amount of money
        2. create a cobb_douglas function: GOOD = 1 * labor ** 1.
        """
        self.create('money', 1)
        self.set_cobb_douglas("GOOD", 1, {"labor": 1})

    def buy_labor(self):
        """ receives all labor offers and accepts them one by one """
        oo = self.get_offers("labor")
        for offer in oo:
            self.accept(offer)

    def production(self):
        """ uses all labor that is available and produces
        according to the set cobb_douglas function """
        self.produce_use_everything()

    def sell_goods(self):
        """ offers one unit of labor to firm 0, for the price of 1 "money" """
        self.sell('household', 0,
                  good="GOOD",
                  quantity=self.possession("GOOD"),
                  price=1)
```

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

At the beginning of each agent you will find

An agent has to import the *abce* module and the `abce.NotEnoughGoods` exception

```
import abce
from abce import NotEnoughGoods
```

This imports the module abce in order to use the base classes Household and Firm. And the NotEnoughGoods exception that allows us the handle situation in which the agent has insufficient resources.

An agent is a class and must at least inherit *abce.Agent*. It automatically inherits *abce.Trade* - `abce.Messaging` and `abce.Database`

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

To create an agent that has can create a consumption function and consume

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

To create an agent that can produce:

```
class Firm(abce.Agent, abce.Firm)
```

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. The Firm class accordingly

The init method

When an agent is created it's init function is called and the simulation parameters as well as the agent_parameters are given to him

DO NOT OVERWRITE THE __init__ method. Instead use ABCE's init method, which is called when the agents are created

```
def init(self, parameters, agent_parameters):
    self.create('labor_endowment', 1)
    self.set_cobb_douglas_utility_function({"MLK": 0.300, "BRD": 0.700})
    self.type = agent_parameters['type']
    self.watt = agent_parameters['watt']
    self.number_of_firms = parameters['number_of_firms']
```

The init method is the method that is called when the agents are created (by the `abce.Simulation.build_agents()`). When the agents were build, a parameter dictionary and a list of agent parameters were given. These can now be accessed in `init` via the `parameters` and `agents_parameters` variable. Each agent gets only one element of the `agents_parameters` list.

With self.create the agent creates the good 'labor_endowment'. Any good can be created. Generally speaking. In order to have a physically consistent economy goods should only be created in the init method. The good money is used in transactions.

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.

In order to let the agent remember a parameter it has to be saved in the self domain of the agent.

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 action_list in the Simulation in start.py.

For example when in the action list *('household', 'consumption')* is called the consumption method is executed of each household agent is executed. **It is important not to overwrite abce's methods with the agents methods.** For example if one would call the `consumption(self)` method below `consume(self)`, abce's consume function would not work anymore.

```
class Household(abce.Agent, abce.Household):
    def init(self, simulation_parameters, agent_parameters):
        self.create('labor_endowment', 1)
        self.set_cobb_douglas_utility_function({"GOOD": 1})
        self.current_utility = 0

    . . .

    def consumption(self):
        """ consumes_everything and logs the aggregate utility. current_utility
        """
        self.current_utility = self.consume_everything()
        self.log('HH', self.current_utility)
```

In the above example we see how a (degenerate) utility function is declared and how the agent consumes. The dictionary assigns an exponent for each good, for example a consumption function that has .5 for both exponents would be {'good1': 0.5, 'good2': 0.5}.

In the method *consumption*, which has to be called form the action_list in the Simulation, everything is consumed an the utility from the consumption is calculated and logged. The utility is logged and can be retrieved see retrieval of the simulation results

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.Firm):
    def init(self):
       set_cobb_douglas('bread', 1.890, {"yeast": 0.333, "labor": 0.667})
        ...

    def production(self):
        self.produce_use_everything()
```

More details in `abce.Firm`. `abce.FirmMultiTechnologies` offers a more advanced interface for firms with layered production functions.

Trade

ABCE clears trade 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):
    offer = self.sell(buyer, 2, 'BRD', price=1, quantity=2.5)
    self.checkorders.append(offer)  # optional
```

```
# Agent 2
def buying(self):
    offers = self.get_offers('cookies')
    for offer in offers:
        if offer.price < 0.5
            try:
                self.accept(offer)
            except NotEnoughGoods:
                self.accept(offer, self.possession('money') / offer.price)
```

```
# Agent 1
def check_trade(self):
    print(self.checkorders[0])
```

Agent 1 sends a selling offer to Agent 2, which is the agent with the id `2` from the `buyer` group (`buyer_2`) Agent 2 receives all offers, he accepts all offers with a price smaller that 0.5. If he has insufficient funds to accept an offer an NotEnoughGoods exception is thrown. If a NotEnoughGoods exception is thrown the except block `self.accept(offer, self.possession('money') / offer.price)` is executed, which leads to a partial accept. Only as many goods as the agent can afford are accepted. If a polled offer is not accepted its automatically rejected. It can also be explicitly rejected with `self.reject(offer)` (*abce.Trade.reject()*).

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

Data production

There are three different ways of observing your agents:

Trade Logging

when you specify `Simulation(..., trade_logging='individual')` all trades are recorded and a SAM or IO matrix is created. This matrices are currently not display in the GUI, but accessible as csv files in the `simulation.path` directory

Manual in agent logging

An agent can log a variable, *abce.Agent.possession()*, *abce.Agent.possessions()* and most other methods such as `abce.Firm.produce()` with `abce.Database.log()`:

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

Retrieving the logged data

If the GUI is switched off there must be a `abce.Simulation.graphs()` after `abce.Simulation.run()`. Otherwise no graphs are displayed. If no browser window open you have to go manually to the address "http://127.0.0.1:8000/"

The results are stored in a subfolder of the ./results/ folder. `simulation.path` gives you the path to that folder.

The tables are stored as '.csv' files which can be opened with excel.

Tutorial for Plant Modeling

In this tutorial we will implement an economy that has two plants. These plants have products and by products, products and by products are traded.

1. **Prepare the template**

   (a) Copy start.py household.py and firm.py from the template directory of abce to a new directory (ZIP can be downloaded from here https://github.com/AB-CE/examples).

   (b) rename firm.py to chpplant.py

2. Lets write the 1st agent:

   (a) Open chpplant.py change the Firm `class Firm(abce.Agent, abce.Firm):` to `class CHPPlant(abce.Agent, abce.Firm):`

   (b) Now we need to specify production functions. There are standard production functions like cobb-douglas and leontief already implemented, but our plants get more complicated production functions. We define the production function a firm uses in `def init(self)`. (not __init__ !) So there add the following lines, `class CHPPLant(...):`

```
class Firm(abce.Agent, abce.Firm):
 def init(self, simulation_parameters, agent_parameters):
     def production_function(goods):
         output = {'electricity': goods['biogas'] ** 0.25 * goods[
↪'water'] ** 0.5,
                   'steam': min(goods['biogas'], goods['water'])}
         return output

     use = {'biogas': 1, 'water': 1}  # inputs that are used for␣
↪production are consumed completely

     self.set_production_function_many_goods(production_function, use)
```

   The `def production_function(goods):` returns the production result as a dictionary. Each key is a good that is produced. The value is a formula that sets how the amount of this product is calculated. For example the function above creates electricity and steam. Electricity is produced by a cobb-douglas production function. While steam is the minimum between the amount of water and fuel used.

   `use = {'biogas': 1, 'water': 1}`, specifies how much percent of each good is used up in the process.

   `self.set_production_function(production_function, use)` sets the agents production function to the functions we just created.

   (a) in `def init(self):` we need to create some initial goods

```
...
self.create('biogas', 100)
self.create('water', 100)
```

   (b) In order to produce create a production method in `class CHPPlant(...):` insert the following code right after the `def init(self):` method:

```
def production(self):
    self.produce({'biogas' : 100, 'water' : 100})
```

   (c) also add:

```
def refill(self):
    self.create('biogas', 100)
    self.create('water', 100)
```

3. We will now modify `start.py` to run this incomplete simulation.

   (a) replace `from firm import Firm` and `household import Household` with `from chpplant import CHPPLant`. This imports your agent in start.py.

   (b) delete `simulation.declare_round_endowment(...)` delete `simulation.declare_perishable(...)` delete `simulation.build_agents(Household, 'household',...)`

   (c) change `simulation.build_agents(Firm, 'firm',...)` to

   ```
   simulation.build_agents(CHPPlant, 'chpplant ', number=1)
   ```

   With this we create 1 agent of type CHPPLANT, it's group name will be `chpplant` and its number `0`.

   (d) change:

   ```
   simulation.panel('household', possessions=['good1', 'good2']
                                 variables=['utility'])
   ```

   to:

   ```
   simulation.panel('chpplant', possessions=['electricity', 'biogas',
   ↪'water', 'steam'], variables=[])
   ```

   panel and all other declarations must be before the agents are build.

   (a) change:

   ```
   for r in range(100):
       simulation.advance_round(r)
       chpplant.production()
       chpplant.refill()
       chpplant.panel()
   ```

   This will tell the simulation that in every round, the firms execute the `production` method we specified in CHPPLant. Then it refills the input goods. Lastly, it creates a snapshot of the possessions of chpplant as will be specified in (e).

4. To run your simulation, the best is to use the terminal and in the directory of your simulation type `python start.py`. In SPYDER make sure that BEFORE you run the simulation for the first time you modify the 'Run Setting' and choose 'Execute in external System Terminal'. If you the simulation in the IDE without making this changes the GUI might block.

5. Lets modify the agent so he is ready for trade

   (a) now delete the refill function in CHPPlant, both in the agent and in the actionlist delete `chpplant.refill()`

   (b) let's simplify the production method in CHPPlant to

   ```
   def production(self):
       self.produce_use_everything()
   ```

(c) in `init` we create money with `self.create('money', 1000)`

7. Now let's create a second agent ADPlant.

(a) copy chpplant.py to applant.py and

(b) in adplant.py change the class name to ADPlant

(c) ADPlant will produce biogas and water out of steam and electricity. In order to achieve this forget about thermodynamics and change:

```
    output = {'electricity': goods['biogas'] ** 0.25 * goods['water'] ** 0.5,
              'steam': min(goods['biogas'], goods['water'])}
    return output

use = {'biogas': 1, 'water': 1}  # inputs that are used for production are␣
↪consumed completely
```

to

```
    output = {'biogas':  min(goods['electricity'], goods['steam']),
              'water': min(goods['electricity'], goods['steam'])}
    return output

use = {'electricity': 1, 'steam': 1}
```

(d) ADPlant will sell everything it produces to CHPPlant. We know that the group name of chpplant is 'chpplant and its id number (id) is 0:

```
def selling(self):
    amount_biogas = self.possession('biogas')
    amount_water = self.possession('water')
    self.sell('chpplant', 0, good='water', quantity=amount_water, price=1)
    self.sell('chpplant', 0, good='biogas', quantity=amount_biogas, price=1)
```

This makes a sell offer to chpplant.

(e) In CHPPlant respond to this offer

```
def buying(self):
    water_offer = self.get_offers('water')[0]
    biogas_offer = self.get_offers('biogas')[0]

    if (water_offer.price * water_offer.quantity
        + biogas_offer.price * biogas_offer.quantity < self.possession('money
↪')):
        self.accept(water_offer)
        self.accept(biogas_offer)
    else:
        quantity_allocationg_half_my_money = self.possession('money') / water_
↪offer.price
        self.accept(water_offer, min(water_offer.quantity, quantity_
↪allocationg_half_my_money))
        self.accept(biogas_offer, min(biogas_offer, self.possession('money')))
```

This accepts both offers if it can afford it, if the plant can't, it allocates half of the money for either good.

(f) reversely in CHPPlant:

```
def selling(self):
    amount_electricity = self.possession('electricity')
    amount_steam = self.possession('steam')
    self.sell('adplant', 0, good='electricity', quantity=amount_electricity,␣
↪price=1)
    self.sell('adplant', 0, good='steam', quantity=amount_steam, price=1)
```

(g) and in ADPlant:

```
def buying(self):
    el_offer = self.get_offers('electricity')[0]
    steam_offer = self.get_offers('steam')[0]

    if (el_offer.price * el_offer.quantity
        + steam_offer.price * steam_offer.quantity < self.possession('money
↪')):
        self.accept(el_offer)
        self.accept(steam_offer)
    else:
        quantity_allocationg_half_my_money = self.possession('money') / el_
↪offer.price
        self.accept(el_offer, min(el_offer.quantity, quantity_allocationg_
↪half_my_money))
        self.accept(steam_offer, min(steam_offer, self.possession('money')))
```

8. let's modify start.py

(a) in `start.py` add

```
from adplant import ADPlant
```

and

```
simulation.build_agents(ADPlant, 'adplant', number=1)
```

(b) change the action list to:

```
for r in range(100):
    simulation.advance_round(r)
    (chpplant + adplant).production()
    (chpplant + adplant).selling()
    (chpplant + adplant).buying()
    chpplant.panel()
```

2. now it should run again.

Examples

ABCE's examples can be downloaded from here: https://github.com/AB-CE/examples

Concepts used in examples

Example	jupyter	pandas	logging	Trade	multi-core	create agents	delete agents	graphical user interface	endowment	perishable	mesa graphical spacial	contracts
jupyter_tutorial	X	X	X	X								
50000_firms					X							
create_agents delete_agent						X	X					
one_household_one_firm								X				
									X	X		
pid_controller				X								
mesa_example sugarscape											X	
CCE		X		trade logging				GUI	Extended			
cheesegrater insurance								X				X
2sectors												
Model of Car market												

Example	production function	utility function	arbitrary time intervals	multi-core	create agents	delete agents	graphical user interface	endowment	perishable	mesa graphical spacial
jupyter_tutorial										
50000_firms				X						
create_agents delete_agent					X	X				
one_household_one_firm							simple			
								X	X	
pid_controller										
mesa_example sugarscape										X
CCE	X	X					X			
cheesegrater insurance							X			
2sectors	X	X								
Model of Car market										
Calendar			X							

Models

CCE

This is the most complete example featuring an agent-based model of climate change tax policies for the United States. It includes a GUI, is databased and and uses production and utility functions.

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.

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 intermediate 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'].

50000 agents example

This is a sheer speed demonstration, that lets 50000 agents trade.

PID controllers

PID controller are a simple algorithm for firms to set prices and quantities. PID controller, work like a steward of a ship. He steers to where he wants to go and after each action corrects the direction based on how the ship changed it's direction,

pid_controller analytical

A simulation of the first Model of Ernesto Carrella's paper: Sticky Prices Microfoundations in a Agent Based Supply Chain Section 4 Firms and Production

Here we have one firm and one market agent. The market agent has the demand function q = 102 - p. The PID controller uses an analytical model of the optimization problem.

Simple Seller Example

A simulation of the first Model of Ernesto Carrella's paper: Zero-Knowledge Traders, journal of artificial societies and social simulation, December 2013

This is a partial 'equilibrium' model. A firm has a fixed production of 4 it offers this to a fixed population of 10 household. The household willingness to pay is household id * 10 (10, 20, 30 ... 90). The firms sets the prices using a PID controller.

Fully PID controlled

A simulation of the first Model of Ernesto Carrella's paper: Sticky Prices Microfoundations in a Agent Based Supply Chain Section 4 Firms and Production

Here we have one firm and one market agent. The market agent has the demand function q = 102 - p. The PID controller has no other knowledge then the reaction of the market in terms of demand.

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, production utility and data logging facilities are unit tested. It is planned to extend unit testing to quotes, so that by version 1.0 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 2

Simulation Programming

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' in https://github.com/AB-CE/examples.

To see how to create a simulation read ipython_tutorial.

This is a minimal template for a start.py:

```
from agent import Agent
from abce import *


simulation = Simulation(name='ABCE')
agents = simulation.build_agents(Agent, 'agent', 2)
for time in range(100):
    simulation.advance_round(time)
    agents.one()
    agents.two()
    agents.three()
simulation.graphs()
```

Note two things are important: there must be either a

`graphs()` or a `finalize()` at the end otherwise the simulation blocks at the end. Further, every round needs to be announced using simulation.advance_round(time). Where time is any representation of time.

class abce.**Simulation**(*name='abce'*, *random_seed=None*, *trade_logging='off'*, *processes=1*)

Bases: object

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

Args:

name: name of the simulation

random_seed (optional): a random seed that controls the random number of the simulation

trade_logging: Whether trades are logged,trade_logging can be 'group' (fast) or 'individual' (slow) or 'off'

processes (optional): The number of processes that run in parallel. Each process hosts a share of the agents. Default is all your logical processor cores times two, using hyper-threading when available. For easy debugging set processes to one and the simulation is executed without parallelization. Sometimes it is advisable to decrease the number of processes to the number of logical or even physical processor cores on your computer. 'None' for all processor cores times 2. **For easy debugging set processes to 1, this way only one agent runs at a time and only one error message is displayed**

Example:

```
simulation = Simulation(name='ABCE',
                        trade_logging='individual',
                        processes=None)
```

Example for a simulation:

```
num_firms = 5
num_households = 2000

w = Simulation(name='ABCE',
               trade_logging='individual',
               processes=None)

w.declare_round_endowment(resource='labor_endowment',
                          productivity=1,
                          product='labor')

w.panel('firm', command='after_sales_before_consumption')

firms = w.build_agents(Firm, 'firm', num_firms)
households = w.build_agents(Household, 'household', num_households)

all = firms + households

for r in range(100):
    self.advance_round(r)
    households.recieve_connections()
    households.offer_capital()
    firms.buy_capital()
    firms.production()
    if r == 250:
        centralbank.intervention()
    households.buy_product()
    all.after_sales_before_consumption()
    households.consume()

w.finalize()
w.graphs()
```

add_and_delete_agents(*round*)

advance_round(*time*)

build_agents(*AgentClass*, *group_name*, *number=None*, *parameters={}*, *agent_parameters=None*)
This method creates agents.

Args:

AgentClass: is the name of the AgentClass that you imported

group_name: the name of the group, as it will be used in the action list and transactions. Should generally be lowercase of the AgentClass.

number: number of agents to be created.

parameters: a dictionary of parameters

agent_parameters: a list of dictionaries, where each agent gets one dictionary. The number of agents is the length of the list

Example:

```
firms = simulation.build_agents(Firm, 'firm',
    number=simulation_parameters['num_firms'])
banks = simulation.build_agents(Bank, 'bank',
                                parameters=simulation_parameters,
                                agent_parameters=[{'name': 'UBS'},
                                {'name': 'amex'},{'name': 'chase'})

centralbanks = simulation.build_agents(CentralBank, 'centralbank',
                                       number=1,
                                       parameters={'rounds':
                                                   num_rounds})
```

declare_expiring(*good*, *duration*)

This type of good lasts for several rounds, but eventually expires. For example computers would last for several years and than become obsolete.

Args:

good: the good, which expires

duration: the duration before the good expires

declare_perishable(*good*)

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

Example::

    w.declare_perishable(good='LAB')
    w.declare_perishable(good='CAP')
```

declare_round_endowment(*resource*, *units*, *product*)

At the beginning of very round the agent gets 'units' units of good 'product' for every 'resource' he possesses.

Round endowments are group specific, that means that when somebody except the specified group holds them they do not produce.

Args:

```
resource:
    The good that you have to hold to get the other

units:
    the multiplier to get the produced good

product:
    the good that is produced if you hold the first good

groups:
    a list of agent groups, which gain the second good,
    if they hold the first one
```

Example:

```
A farmer gets a ton of harvest for every acre:

w.declare_round_endowment(resource='land',
                          units=1000,
                          product='wheat')
```

declare_service(*human_or_other_resource*, *units*, *service*)

When the agent holds the human_or_other_resource, he gets 'units' of service every round the service can be used only with in this round.

Args:

```
human_or_other_resource:
    the good that needs to be in possessions to create the other
    good 'self.create('adult', 2)'
units:
    how many units of the service is available
service:
    the service that is created
groups:
    a list of agent groups that can create the service
```

Example:

```
For example if a household has two adult family members, it gets
16 hours of work

w.declare_service('adult', 8, 'work')
```

finalize()

simulation.finalize() must be run after each simulation. It will write all data to disk

Example:

```
simulation = Simulation(...)
...
for r in range(100):
    simulation.advance_round(r)
    agents.do_something()
    ...

simulation.finalize()
```

graphs()
after the simulation is run, graphs() shows graphs of all data collected in the simulation. Shows the same output as the @gui decorator shows.

Example:

```
simulation = Simulation(...)
for r in range(100):
    simulation.advance_round(r)
    agents.do_something()
    ...

simulation.graphs()
```

path = None
the path variable contains the path to the simulation outcomes it can be used to generate your own graphs as all resulting csv files are there.

time = None
Returns the current time set with simulation.advance_round(time)

Agents

The *abce.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 `abce.Firm` or to create a consumer the `abce.Household`.

For detailed documentation on:

Trading, see *Trade*

Logging and data creation, see *Observing agents and logging*.

Messaging between agents, see *Messaging*.

class `abce.`**Agent**(*id*, *group*, *trade_logging*, *database*, *random_seed*, *num_managers*, *start_round=None*)
Bases: `abce.database.Database`, `abce.trade.Trade`, `abce.messaging.Messaging`

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

Every method can return parameters to the simulation.

For example:

```
class Household(abce.Agent, abce.Household):
    def init(self, simulation_parameters, agent_parameters):
        self.num_firms = simulation_parameters['num_firms']
        self.type = agent_parameters['type']
        ...

    def selling(self):
        for i in range(self.num_firms):
            self.sell('firm', i, 'good', quantity=1, price=1)

    ...
    def return_quantity_of_good(self):
        return possession('good')
```

```
...

simulation = Simulation()
households = Simulation.build_agents(household, 'household',
                                     parameters={...},
                                     agent_parameters=[{'type': 'a'},
                                                       {'type': 'b'}])
for r in range(10):
    simulation.advance_round(r)
    households.selling()
    print(households.return_quantity_of_good())
```

create(*good*, *quantity*)
creates quantity of the good out of nothing

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

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

create_agent(*AgentClass*, *group_name*, *parameters=None*, *agent_parameters=None*)
create a new agent.

Args:

AgentClass: the class of agent to create. (can be the same class as the creating agent)

'group_name': the name of the group the agent should belong to

parameters: a dictionary of parameters

agent_parameters: a dictionary of parameters

Example:

```
self.create_agent(BeerFirm, 'beerfirm',
                  parameters=self.parameters,
                  agent_parameters={'creation': self.round + 1})
```

create_timestructured(*good*, *quantity*)
creates quantity of the time structured good out of nothing. For example:

```
self.creat_timestructured('capital', [10,20,30])
```

Creates capital. 10 units are 2 years old 20 units are 1 year old and 30 units are new.

It can alse be used with a quantity instead of an array. In this case the amount is equally split on the years.:

```
self.create_timestructured('capital', 60)
```

In this case 20 units are 2 years old 20 units are 1 year old and 20 units are new.

Args:

'good': is the name of the good

quantity: an arry or number

delete_agent(*group_name*, *id*, *quite=True*)
This deletes an agent, an agent can delete itself. There are two ways of deleting an agent. By default, quite

is set to True, all future messages to this agent are deleted. If quite is set to False agents are completely deleted. This makes the simulation faster, but if messages are send to this agents the simulation stops.

Args:

group_name: group name of the agent

id: the id of the agent to be deleted

quite: whether the agent deletes incomming messages.

destroy(*good*, *quantity=None*)

destroys quantity of the good. If quantity is omitted destroys all

Args:

```
'good':
    is the name of the good
quantity (optional):
    number
```

Raises:

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

group = None

self.group returns the agents group or type READ ONLY!

id = None

self.id returns the agents id READ ONLY

init(*parameters*, *agent_parameters*)

This method is called when the agents are build. It can be overwritten by the user, to initialize the agents. parameters and agent_parameters are the parameters given in `abce.Simulation.build_agents()`

name = None

self.name returns the agents name, which is the group name and the id

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()

returns all possessions

round = None

self.round is depreciated

time = None

self.time, contains the time set with simulation.advance_round(time) you can set time to anything you want an integer or (12, 30, 21, 09, 1979) or 'monday'

Physical goods and services

Goods

An agent can access a good with `self['cookies']` or `self['money']`.

- `self.create(money, 15)` creates money
- `self.destroy(money, 10)` destroys money
- goods can be given, taken, sold and bought
- :code:‘self[’money’] returns the quantity an agent possesses

Services

Services are like goods, but the need to be declared as services in the simulation `abce.__init__.service()`. In this function one declares a good that creates the other good and how much. For example if one has `self['adults'] = 2`, one could get 16 hours of labor every day. :code:‘simulation.declare_service(‘adults’, 8, ‘labor’).

Trade

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

Bases: `object`

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 shows the decreased amount.

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

   The good is credited and the price is deducted from the agent’s possessions.

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(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 = 'accepted':
        self.price *= offer.final_quantity / offer.quantity
```

Example:

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

# Agent 2
def receive_sale(self):
    oo = self.get_offers('cookies')
    for offer in oo:
        if ((offer.currency == 'dollars' and offer.price < 0.3 * exchange_rate)
            or (offer.currency == 'euros' and dollars'offer.price < 0.3)):

            try:
                self.accept(offer)
            except NotEnoughGoods:
                self.accept(offer, self.possession('money') / offer.price)
        else:
            self.reject(offer)
```

If we did not implement a barter class, but one can use this class as a barter class,

accept (*offer*, *quantity=-999*, *epsilon=1e-11*)
The buy or sell offer is accepted and cleared. If no quantity is given the offer is fully accepted; If a quantity is given the offer is partial accepted.

Args:

offer: the offer the other party made

quantity: quantity to accept. If not given all is accepted

epsilon (optional): if you have floating point errors, a quantity or prices is a fraction of number to high or low. You can increase the floating point tolerance. See troubleshooting – floating point problems

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

buy (*receiver*, *good*, *quantity*, *price*, *currency='money'*, *epsilon=1e-11*)
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: The name of the receiving agent a tuple (group, id). e.G. ('firm', 15)

'good': name of the good

quantity: maximum units disposed to buy at this price

price: price per unit

currency: is the currency of this transaction (defaults to 'money')

epsilon (optional): if you have floating point errors, a quantity or prices is a fraction of number to high or low. You can increase the floating point tolerance. See troubleshooting – floating point problems

get_buy_offers(*good*, *sorted=True*, *descending=False*, *shuffled=True*)

get_buy_offers_all(*descending=False*, *sorted=True*)

get_offers(*good*, *sorted=True*, *descending=False*, *shuffled=True*)
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 manually reject.

peek_offers can be used to look at the offers without them being rejected automatically

Args:

good: the good which should be retrieved

sorted(bool, default=True): Whether offers are sorted by price. Faster if False.

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

shuffled(bool, default=True): whether the order of messages is randomized or correlated with the ID of the agent. Setting this to False speeds up the simulation considerably, but introduces a bias.

Returns: A list of `abce.trade.Offer` 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(offer, 1)
    else:
        self.reject(offer)   # optional
```

get_offers_all(*descending=False*, *sorted=True*)
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 manually reject.

Args:

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

sorted(default=True): Whether offers are sorted by price. Faster if False.

Returns:

a dictionary with good types as keys and list of `abce.trade.Offer` as values

Example:

```
oo = get_offers_all(descending=False)
for good_category in oo:
   print('The cheapest good of category' + good_category
   + ' is ' + good_category[0])
   for offer in oo[good_category]:
       if offer.price < 0.5:
           self.accept(offer)

for offer in oo.beer:
   print(offer.price, offer.sender_group, offer.sender_id)
```

get_sell_offers(*good*, *sorted=True*, *descending=False*, *shuffled=True*)

get_sell_offers_all(*descending=False*, *sorted=True*)

give(*receiver*, *good*, *quantity*, *epsilon=1e-11*)
gives a good to another agent

Args:

receiver: The name of the receiving agent a tuple (group, id). e.G. ('firm', 15)

good: the good to be transfered

quantity: amount to be transfered

epsilon (optional): if you have floating point errors, a quantity or prices is a fraction of number to high or low. You can increase the floating point tolerance. See troubleshooting – floating point problems

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'))
```

peak_buy_offers(*good*, *sorted=True*, *descending=False*, *shuffled=True*)

peak_offers(*good*, *sorted=True*, *descending=False*, *shuffled=True*)
returns a peak on all offers of the 'good' ordered by price. Peaked offers can not be accepted or rejected and 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(offer, 1)
    else:
        self.reject(offer)  # optional
```

peak_sell_offers (*good*, *sorted=True*, *descending=False*, *shuffled=True*)

reject (*offer*)

Rejects and offer, if the offer is subsequently accepted in the same subround it is accepted'. Peaked offers can not be rejected.

Args:

offer: the offer to be rejected

sell (*receiver*, *good*, *quantity*, *price*, *currency='money'*, *epsilon=1e-11*)

commits to sell the quantity of good at price

The good is not available for the agent. 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: group of the receiving agent

receiver_id: number of the receiving agent

'good': name of the good

quantity: maximum units disposed to buy at this price

price: price per unit

currency: is the currency of this transaction (defaults to 'money')

epsilon (optional): if you have floating point errors, a quantity or prices is a fraction of number to high or low. You can increase the floating point tolerance. See troubleshooting – floating point problems

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 == 'accepted':
        print(offer.final_quantity , 'cookies have be bougth')
    else:
        offer.status == 'rejected':
        print('On diet')
```

take (*receiver*, *good*, *quantity*, *epsilon=1e-11*)

take a good from another agent. The other agent has to accept. using self.accept()

Args:

receiver_group: group of the receiving agent

receiver_id: number of the receiving agent

good: the good to be taken

quantity: the quantity to be taken

epsilon (optional): if you have floating point errors, a quantity or prices is a fraction of number to high or low. You can increase the floating point tolerance. See troubleshooting – floating point problems

abce.trade.**Offer**(*sender_group*, *sender_id*, *receiver_group*, *receiver_id*, *good*, *quantity*, *price*, *currency*, *sell*, *status*, *final_quantity*, *id*, *made*, *status_round*)

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 abce.messaging.**Messaging**

Bases: object

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_id* 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_id: 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_id)
    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_id* 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*.

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

send(*receiver*, *topic*, *content*)
sends a message to agent. Agents receive it at the beginning of next round with `get_messages()` or `get_messages_all()`.

Args:

```
receiver:
    The name of the receiving agent a tuple (group, id).
    e.G. ('firm', 15)

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")
```

Firm and production

class abce.agents.**Firm**
Bases: abce.agents.firmmultitechnologies.FirmMultiTechnologies

The firm class allows you to declare a production function for a firm. `abce.Firm.set_leontief()`, `abce.Firm.set_production_function()` `abce.Firm.set_cobb_douglas()`, `abce.Firm.set_production_function_fast()` (FirmMultiTechnologies, allows you to declare several) With `abce.Firm.produce()` and `abce.Firm.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 `abce.Firm.predict_produce()` with the price vector you get the profitability of the production.

If you want to create a firm with more than one production technology, you, should use the `abce.FirmMultiTechnologies` class.

net_value(*produced_goods*, *used_goods*, *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:

produced_goods: a dictionary with goods and quantities

used_goods: 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_net_value(*input_goods*, *price_vector*)

Predicts the net value of a production, given a price vector

Args:

production_function: a production function

input_goods: the goods to be used in the simulated production

price_vector: vector of prices for input and output goods

Example:

```
input_goods = {'wheels': 4, 'chassi': 1}
price_vector = {'wheels': 10, 'chassi': 100, 'car':1000}
self.predict_net_value(self.consumption_good_production_function, input_goods,
↪ price_vector)

>>> 860
```

predict_produce_input(*input_goods*)

Returns a vector with input of goods

Predicts the use of input goods, for a production.

Args:

production_function: A production_function produced with create_production_function, create_cobb_douglas or create_leontief

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

Example:

```
print(A.predict_produce_input(car_production_function, two_cars))

>>> {'wheels': 4, 'chassi': 1}
```

predict_produce_output(*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)

Args:

```
production_function:
    A production_function produced with
    create_production_function, create_cobb_douglas or create_leontief
input_goods {'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_produce_output(car_production_function, two_cars))
>>> {'car': 2}
```

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_ces (*output*, *gamma*, *multiplier=1*, *shares=None*)
creates a CES production function

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

$Q = F \cdot [\sum_{i=1}^{n} a_i X_i^{\gamma}]^{\frac{1}{\gamma}}$

Production_functions are than used by in produce, predict_vector_produce and predict_output_produce.

Args:

'output': Name of the output good

gamma: elasticity of substitution $= s = \frac{1}{1-\gamma}$

multiplier: CES multiplier F

shares: a_i = Share parameter of input i, $\sum_{i=1}^{n} a_i = 1$ when share_parameters is not specified all inputs are weighted equally and the number of inputs is flexible. Share parameters are an array with good names as keys and the shares as values.

Example:

```
def init(self):
    self.set_ces('stuff', gamma=0.5, multiplier=1, shares={'labor': 0.25,
↪'stone':0.25, 'wood':0.5})

...


def producing(self):
    self.produce({'stone' : 20, 'labor' : 1, 'wood': 12})
```

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*)
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*, *output*, *use*)
Creates the firm's production functions from a formula.

A production function is a production process that produces a given output good from several input goods. Once you have set a production function you can use `abce.Firm.produce()` to produce.

If you want to produce more than one output good try `abce.Firm.set_production_function_many_goods()`

Args:

formula: this is a method, that takes the possession dictionary as an input and returns a float.

output: the name of the good 'string'

use: a dictionary of how much percent of each good is used up in the process

Example:

```
def init(self):
    ...
    def production_function(goods):
        return goods['a'] ** 0.25 * goods['b'] ** 0.5 * goods['c'] ** 0.25

    use = {'a': 1, 'b': 0.1, 'c': 0}

    self.set_production_function(production_function, output='cookies'␣
↪use=use)

def production(self):
    self.produce({'a' : 1, 'b' : 2}
```

set_production_function_many_goods (*formula*, *use*)
creates a production function that produces many goods

A production function is a production process that produces the several output goods according to the formula to the output goods and uses up some or all of the input goods. Production_functions are than used by produce, predict_vector_produce and predict_output_produce.

create_production_function_fast is faster but more complicated

Args:

formula: this is a method, that takes a goods dictionary as an input and returns a dictionary with the newly created goods.

use: a dictionary of how much percent of each good is used up in the process

Returns:

A production_function that can be used in produce etc.

Example:

```
def init(self):
    ...
    def production_function(goods)
        output = {'soft_rubber': goods['a'] ** 0.25 * goods['b'] ** 0.5 *␣
↪goods['c'] ** 0.25,
```

```
                    'hard_rubber': goods['a'] ** 0.1 * goods['b'] ** 0.2 *␣
↪goods['c'] ** 0.01,
                    'waste': goods['b'] / 2}
        return output

    use = {'a': 1, 'b': 0.1, 'c': 0}

    self.set_production_function_many_goods(production_function, use)

def production(self):
    self.produce({'a' : 1, 'b' : 2, 'c': 5})
```

sufficient_goods(*input_goods*)
checks whether the agent has all the goods in the vector input

Household and consumption

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

class abce.agents.**Household**
Bases: object

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: The utility as 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(self.consumption_set)
except NotEnoughGoods:
    utility = self.consume(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})
```

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

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*, *use*)
creates a utility function from a formula

The formula is a function that takes a dictionary of goods as an argument and returns a floating point number, the utility. use is a dictionary containing the percentage use of the goods used in the consumption. Goods can be fully used (=1) for example food, partially used e.G. a car. And not used at all (=0) for example a house.

Args:

formula: a function that takes a dictionary of goods and computes the utility as a floating number.

use: a dictionary that specifies for every good, how much it depreciates in percent.

Example:

```
self init(self):
    ...
    def utility_function(goods):
        return goods['house'] ** 0.2 * good['food'] ** 0.6 + good['car'] ** 0.
↪2

    {'house': 0, 'food': 1, 'car': 0.05}

    self.set_utility_function(utility_function, use)
```

Observing agents and logging

There are 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()`.

Aggregate Data: `aggregate()` save agents possessions and variable aggregated over a group

Panel Data: `panel()` 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*

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!).

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 variable between a start and an end point with: `observe_begin()` and `observe_end()`.

class abce.database.**Database**
Bases: object

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). Sending a dictionary instead of several using several log statements with a single variable is faster.

Args:

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

data_to_log: a variable or a dictionary with data to log in the the database

Example:

```
self.log('profit', profit)

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

self.log(self.produce_use_everything())
```

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')]})
```

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')})
```

Panel Data

`Group.`**`panel_log`**(*variables=[]*, *possessions=[]*, *func={}*, *len=[]*)

panel_log(.) writes a panel of variables and possessions of a group of agents into the database, so that it is displayed in the gui.

Args:

possessions (list, optional): a list of all possessions you want to track as 'strings'

variables (list, optional): a list of all variables you want to track as 'strings'

func (dict, optional): accepts lambda functions that execute functions. e.G. `func = lambda self: self.old_money - self.new_money`

len (list, optional): records the length of the list or dictionary with that name.

Example in start.py:

```
for round in simulation.next_round():
    firms.produce_and_sell()
    firms.panel_log(possessions=['money', 'input'],
                variables=['production_target', 'gross_revenue'])
    households.buying()
```

Aggregate Data

`Group.`**`agg_log`**(*variables=[]*, *possessions=[]*, *func={}*, *len=[]*)

agg_log(.) writes a aggregate data of variables and possessions of a group of agents into the database, so that it is displayed in the gui.

Args:

possessions (list, optional): a list of all possessions you want to track as 'strings'

variables (list, optional): a list of all variables you want to track as 'strings'

func (dict, optional): accepts lambda functions that execute functions. e.G. `func = lambda self: self.old_money - self.new_money`

len (list, optional): records the length of the list or dictionary with that name.

Example in start.py:

```
for round in simulation.next_round():
    firms.produce_and_sell()
    firms.agg_log(possessions=['money', 'input'],
                  variables=['production_target', 'gross_revenue'])
    households.buying()
```

Network logging

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 exact path is in simulation.path. So if you want to post-process your data, you can write a function that changes in to the simulation.path directory and manipulates the CSV files there. The tables are stored as '.csv' files which can be opened with excel.

The same data is also as a sqlite3 database 'database.db' available. It can be opened by 'sqlitebrowser' in ubuntu.

Example:

```
In start.py

simulation = abce.Simulation(...)
...
simulation.run()

os.chdir(simulation.path)
firms = pandas.read_csv('aggregate_firm.csv')
...
```

NotEnoughGoods Exception

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

Bases: `Exception`

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()
```

CHAPTER 3

Advanced

Contracting

Warning: Contracting is experimental and the API is not stable yet

class abce.**Contracting**
Bases: object

This is a class, that allows you to create contracts. For example a work contract. One agent commits to deliver a good or service for a set amount of time.

For example you have a firm and a worker class. 'Labor' is set as a service meaning that it lasts not longer than one round and the worker how has an adult gets one unit of labor every round see: abce.declare_service(). The firm offers a work contract, the worker responds. Every round the worker delivers the labor and the firm pays.:

```
class Firm(abce.Agent, abce.Contract)
    def request_offer(self):
        if self.round % 10 == 0:
            self.given_contract = self.request_contract('contractbuyer', 0,
                                                        good='labor',
                                                        quantity=5,
                                                        price=10,
                                                        duration=10 - 1)

    def deliver_or_pay(self):
        self.pay_contract('labor')

class Worker(abce.Agent, abce.Contract):
    def init(self):
        self.create('adult', 1)

    def accept_offer(self):
```

```
        contracts = self.get_contract_requests('labor')
        for contract in contracts:
            if contract.price < 5:
                self.accepted_contract = self.accept_contract(contract)

    def deliver_or_pay(self):
        self.deliver('labor')
```

Firms and workers can check, whether they have been paid/provided with labor using the `is_paid()` and `is_delivered()` methods.

The worker can also initiate the transaction by requesting a contract with `make_contract_offer()`.

A contract has the following fields:

sender_group:

sender_id:

deliver_group:

deliver_id:

pay_group:

pay_id:

good:

quantity:

price:

end_date:

makerequest: 'm' for make_contract_offer and 'r' for request_contract

id: unique number of contract

accept_contract (*contract*, *quantity=None*)

Accepts the contract. The contract is completely accepted, when the quantity is not given. Or partially when quantity is set.

Args:

contract: the contract in question, received with get_contract_requests or get_contract_offers

quantity (optional): the quantity that is accepted. Defaults to all.

calculate_assetvalue (*prices={}*, *parameters={}*, *value_functions={}*)

calculate_liablityvalue (*prices={}*, *parameters={}*, *value_functions={}*)

calculate_netvalue (*prices={}*, *parameters={}*, *value_functions={}*)

calculate_valued_assets (*prices={}*, *parameters={}*, *value_functions={}*)

calculate_valued_liablities (*prices={}*, *parameters={}*, *value_functions={}*)

contracts_to_deliver (*good*)

contracts_to_deliver_all ()

contracts_to_receive (*good*)

contracts_to_receive_all ()

deliver_contract(*contract*)
delivers on a contract

end_contract(*contract*)

get_contract_offers(*good*, *descending=False*)
Returns all contract offers and removes them. The contract are ordered by price (ascending), when tied they are randomized.

Args:

good: good that underlies the contract

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

Returns: list of contract offers ordered by price

offer_good_contract(*receiver_group*, *receiver_id*, *good*, *quantity*, *price*, *duration*)
This method offers a contract to provide a good or service to the receiver. For a given time at a given price.

Args:

receiver_group: group to receive the good

receiver_id: group to receive the good

good: the good or service that should be provided

quantity: the quantity that should be provided

price: the price of the good or service

duration: the length of the contract, if duration is None or not set, the contract has no end date.

Example:

```
self.given_contract = self.make_contract_offer('firm', 1, 'labor', quantity=8,
↪ price=10, duration=10 - 1)
```

pay_contract(*contract*)
delivers on a contract

request_good_contract(*receiver_group*, *receiver_id*, *good*, *quantity*, *price*, *duration*)
This method requests a contract to provide a good or service to the sender. For a given time at a given price. For example a job advertisement.

Args:

receiver_group: group of the receiver

receiver_id: id of the receiver

good: the good or service that should be provided

quantity: the quantity that should be provided

price: the price of the good or service

duration: the length of the contract, if duration is None or not set, the contract has no end date.

was_delivered_last_round(*contract*)

was_delivered_this_round(*contract*)

was_paid_last_round(*contract*)

was_paid_this_round(*contract*)

FirmMultiTechnologies

class abce.agents.**FirmMultiTechnologies**
Bases: object

create_ces(*output*, *gamma*, *multiplier=1*, *shares=None*)
creates a CES production function

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

$Q = F \cdot [\sum_{i=1}^{n} a_i X_i^{\gamma}]^{\frac{1}{\gamma}}$

Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

Args:

'output': Name of the output good

gamma: elasticity of substitution $= s = \frac{1}{1-\gamma}$

multiplier: CES multiplier F

shares: a_i = Share parameter of input i, $\sum_{i=1}^{n} a_i = 1$ when share_parameters is not specified all inputs are weighted equally and the number of inputs is flexible.

Returns:

A production_function that can be used in produce etc.

Example:

```
self.stuff_production_function = self.create_ces('stuff', gamma=0.5,␣
↪multiplier=1, shares={'labor': 0.25, 'stone':0.25, 'wood':0.5})
self.produce(self.stuff_production_function, {'stone' : 20, 'labor' : 1, 'wood
↪': 12})
```

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 Cobb-Douglas 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:

def init(self): self.plastic_production_function = self.create_cobb_douglas('plastic', {'oil' : 10, 'labor' : 1}, 0.000001)

...

def producing(self): self.produce(self.plastic_production_function, {'oil' : 20, 'labor' : 1})

create_leontief (*output*, *utilization_quantities*)
creates a Leontief production function

A production function is a production process that produces the given input goods according to the Leontief 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: dictionary of multipliers it min(good1 * a, good2 * b, good3 * c...)

{'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.car_production_function = self.create_leontief('car', {'wheel' : 4, 'chassi' : 1}) self.produce(self.car_production_function, {'wheel' : 20, 'chassi' : 5})

create_production_function_many_goods (*formula*, *use*)
creates a production function that produces many goods

A production function is a production process that produces several output goods from several input goods. It does so according to the formula given. Input goods are usually partially or completely used up. Production_functions can then ben used as an argument in `abce.FirmMultiTechnologies.produce()`, `abce.FirmMultiTechnologies._predict_produce_output()` `abce.FirmMultiTechnologies._predict_produce_input()`

create_production_function_fast is faster but more complicated

Args:

formula: this is a method, that takes a goods dictionary as an input and returns a dictionary with the newly created goods.

use: a dictionary of how much percent of each good is used up in the process

Returns:

A production_function that can be used in produce etc.

Example:

```
def init(self):
    ...
    def production_function(goods)
        output = {'soft_rubber': goods['a'] ** 0.25 * goods['b'] ** 0.5 *␣
↪goods['c'] ** 0.25,
                  'hard_rubber': goods['a'] ** 0.1 * goods['b'] ** 0.2 *␣
↪goods['c'] ** 0.01,
                  'waste': goods['b'] / 2}
        return output

    use = {'a': 1, 'b': 0.1, 'c': 0}

    self.production_function = self.create_production_function(production_
↪function, use)
```

```
def production(self):
    self.produce(self.production_function, {'a' : 1, 'b' : 2, 'c': 5})
```

create_production_function_one_good(*formula*, *output*, *use*)
creates a production function, that produces one good

A production function is a production process that produces the given input goods according to the formula to the output goods and uses up some or all of the input 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: this is a method, that takes the possession dictionary as an input and returns a float.

output: the name of the good 'string'

use: a dictionary of how much percent of each good is used up in the process

Returns:

A production_function that can be used in produce etc.

Example:

```
def init(self):
    ...
    def production_function(goods)
        return goods['a'] ** 0.25 * goods['b'] ** 0.5 * goods['c'] ** 0.25

    use = {'a': 1, 'b': 0.1, 'c': 0}

    self.production_function = self.create_production_function(production_
↪function, use)

def production(self):
    self.produce(self.production_function, {'a' : 1, 'b' : 2}
```

net_value(*produced_goods*, *used_goods*, *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:

produced_goods: a dictionary with goods and quantities

used_goods: 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_net_value (*production_function*, *input_goods*, *price_vector*)

Predicts the net value of a production, given a price vector

Args:

production_function: a production function

input_goods: the goods to be used in the simulated production

price_vector: vector of prices for input and output goods

Example:

```
input_goods = {'wheels': 4, 'chassi': 1}
price_vector = {'wheels': 10, 'chassi': 100, 'car':1000}
self.predict_net_value(self.consumption_good_production_function, input_goods,
↪ price_vector)

>>> 860
```

predict_produce_input (*production_function*, *input_goods*)

Returns a vector with input of goods

Predicts the use of input goods, for a production.

Args:

production_function: A production_function produced with create_production_function, create_cobb_douglas or create_leontief

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

Example:

```
print(A._predict_produce_input(car_production_function, two_cars))

>>> {'wheels': 4, 'chassi': 1}
```

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)

Args:

```
production_function:
    A production_function produced with
    create_production_function, create_cobb_douglas or create_leontief
input_goods {'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_produce_output(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.FirmMultiTechnologies.create_cobb_douglas` or
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

Quote

class abce.quote.**Quote**

Bases: object

Quotes as opposed to trades are uncommitted offers. They can be made even if they agent can not fullfill them. With `accept_quote()` and `accept_quote_partial()`, the receiver of a quote can transform them into a trade.

accept_quote(*quote*)
makes a commited buy or sell out of the counterparties quote. For example, if you receive a buy quote you can accept it and a sell offer is send to the offering party.

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.

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()
```

quote_buy(*receiver*, *good=None*, *quantity=None*, *price=None*)
quotes a price to buy quantity of 'good' a receiver. Use None, if you do not want to specify a value.

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

Args:

receiver_group: agent group name of the agent

receiver_id: 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*, *good=None*, *quantity=None*, *price=None*)
quotes a price to sell quantity of 'good' to a receiver. Use None, if you do not want to specify a value.

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

Args:

receiver_group: agent group name of the agent

receiver_id: the agent's id number

'good': name of the good

quantity: maximum units disposed to sell at this price

price: price per unit

abce.quote.**Quotation**(*sender_group*, *sender_id*, *receiver_group*, *receiver_id*, *good*, *quantity*, *price*, *buysell*, *id*)

Spatial and Netlogo like Models

ABCE deliberately does not provide spatial representation, instead it integrates with other packages that specialize in spatial representation.

Netlogo like models

For Netlogo like models in Python, we recommend using ABCE together with MESA

A simple example shows how to build a spatial model in ABCE using MESA:

On github

A wrapper file to start the graphical representation and the simulation

```
""" This is a simple demonstration model how to integrate ABCE and mesa.
The model and scheduler specification are taken care of in
ABCE instead of Mesa.

Based on
https://github.com/projectmesa/mesa/tree/master/examples/boltzmann_wealth_model.

For further reading, see
[Dragulescu, A and Yakovenko, V. Statistical Mechanics of Money, Income, and Wealth:␣
↪A Short Survey. November, 2002](http://arxiv.org/pdf/cond-mat/0211175v1.pdf)
"""
from model import MoneyModel
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer
from mesa.visualization.modules import ChartModule


def agent_portrayal(agent):
    """ This function returns a big red circle, when an agent is wealthy and a
    small gray circle when he is not """
    portrayal = {"Shape": "circle",
                 "Filled": "true",
                 "r": 0.5}

    if agent.report_wealth() > 0:
        portrayal["Color"] = "red"
        portrayal["Layer"] = 0
    else:
        portrayal["Color"] = "grey"
        portrayal["Layer"] = 1
```

```
        portrayal["r"] = 0.2
    return portrayal


def main(x_size, y_size):
    """ This function sets up a canvas to graphically represent the model 'MoneyModel'
    and a chart, than it runs the server and runs the model in model.py in the␣
↪browser """
    grid = CanvasGrid(agent_portrayal, x_size, y_size, 500, 500)

    chart = ChartModule([{"Label": "Gini",
                          "Color": "Black"}],
                        data_collector_name='datacollector')
    # the simulation uses a class DataCollector, that collects the data and
    # relays it from self.datacollector to the webpage

    server = ModularServer(MoneyModel,
                           [grid, chart],
                           "ABCE and MESA integrated",
                           x_size * y_size, x_size, y_size)
    server.port = 8534  # change this number if address is in use
    server.launch()


if __name__ == '__main__':
    main(25, 25)
```

A file with the simulation itself, that can be executed also without the GUI

```
""" This is a simple demonstration model how to integrate ABCE and mesa.
The model and scheduler specification are taken care of in
ABCE instead of Mesa.

Based on
https://github.com/projectmesa/mesa/tree/master/examples/boltzmann_wealth_model.

For further reading, see
[Dragulescu, A and Yakovenko, V. Statistical Mechanics of Money, Income, and Wealth:␣
↪A Short Survey. November, 2002](http://arxiv.org/pdf/cond-mat/0211175v1.pdf)
"""
import abce
from mesa.space import MultiGrid
from mesa.datacollection import DataCollector
from moneyagent import MoneyAgent


def compute_gini(model):
    """ calculates the index of wealth distribution form a list of numbers """
    agent_wealths = model.wealths
    x = sorted(agent_wealths)
    N = len(agent_wealths)
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B


class MoneyModel(abce.Simulation):  # The actual simulation must inherit from␣
↪Simulation
```

```
    """ The actual simulation. In order to interoperate with MESA the simulation
    needs to be encapsulated in a class. __init__ sets the simulation up. The step
    function runs one round of the simulation. """

    def __init__(self, num_agents, x_size, y_size):
        abce.Simulation.__init__(self,
                                 name='ABCE and MESA integrated',
                                 rounds=300,
                                 processes=1)
        # initialization of the base class. MESA integration requires
        # single processing
        self.grid = MultiGrid(x_size, y_size, True)
        self.agents = self.build_agents(MoneyAgent, 'MoneyAgent', num_agents,
                                         parameters={'grid': self.grid})
        # ABCE agents must inherit the MESA grid
        self.running = True
        # MESA requires this
        self.datacollector = DataCollector(
            model_reporters={"Gini": compute_gini})
        # The data collector collects a certain aggregate value so the graphical
        # components can access them

        self.wealths = [0 for _ in range(num_agents)]

    def step(self):
        """ In every step the agent's methods are executed, every set the round
        counter needs to be increased by self.next_round() """
        self.next_round()
        self.agents.do('move')
        self.agents.do('give_money')
        self.wealths = self.agents.do('report_wealth')
        # agents report there wealth in a list self.wealth
        self.datacollector.collect(self)
        # collects the data


if __name__ == '__main__':
    """ If you run model.py the simulation is executed without graphical
    representation """
    money_model = MoneyModel(1000, 20, 50)
    for r in range(100):
        print(r)
        money_model.step()
```

A simple agent

```
import abce
import random


class MoneyAgent(abce.Agent):
    """ agents move randomly on a grid and give_money to another agent in the same␣
↪cell """

    def init(self, parameters, agent_parameters):
        self.grid = parameters["grid"]
```

```
        """ the grid on which agents live must be imported """
        x = random.randrange(self.grid.width)
        y = random.randrange(self.grid.height)
        self.pos = (x, y)
        self.grid.place_agent(self, (x, y))
        self.create('money', random.randrange(2, 10))

    def move(self):
        """ moves randomly """
        possible_steps = self.grid.get_neighborhood(self.pos,
                                                    moore=True,
                                                    include_center=False)
        new_position = random.choice(possible_steps)
        self.grid.move_agent(self, new_position)

    def give_money(self):
        """ If the agent has wealth he gives it to cellmates """
        cellmates = self.grid.get_cell_list_contents([self.pos])
        if len(cellmates) > 1:
            other = random.choice(cellmates)
            try:
                self.give(other.group, other.id, good='money', quantity=1)
            except abce.NotEnoughGoods:
                pass

    def report_wealth(self):
        return self.possession('money')
```

CHAPTER 4

Graphical User Interface and Results

Graphical User Interface

`python -m abce.show` shows the simulation results in ./result/*

`abce.gui.`**`gui`**(*parameter_mask*, *names=None*, *header=None*, *story=None*, *title='Agent-Based Computational Economics'*, *texts=None*, *pages=None*, *histograms=None*, *serve=False*, *runtime='browser-X'*, *truncate_rounds=0*, *hostname='0.0.0.0'*, *port=80*, *pypy=None*)

gui is a decorator that can be used to add a graphical user interface to your simulation.

Args:

parameter_mask: a dictionary with the parameter name as key and an example value as value. Instead of the example value you can also put a tuple: (min, default, max)

parameters can be:

- **float:** {'exponent': (0.0, 0.5, 1.1)}
- **int:** {'num_firms': (0, 100, 100000)}
- dict or list, which should be strings of a dict or a list (see example):

  {'list_to_edit': "['brd', 'mlk', 'add']"}
- **a list of options:** {'several_options': ['opt_1', 'opt_2', 'opt_3']}
- **a string:** {'name': '2x2'}

names (optional): a dictionary with the parameter name as key and an alternative text to be displayed instead.

title: a string with the name of the simulation.

header: html string for a bar on the top

story: a dictionary with text to be displayed alongeside the graphs. Key must be the graphs title, value can be text or html.

pages: A dictinoary with title as key and links to external websites as values, which are displayed on the right hand side.

truncate_rounds: Does not display the initial x rounds, in the result graphs

runtime: webbrowser to start the simulation in, can be 'xui' or python's webbrowser module's webbrowser string.

histograms: specifies in which round histograms are generated. If it is not specified rounds from the menu is used. Alternatively you can create 'histogram' parameter in parameter_mask.

serve: If you run this on your local machine serve must be False. If used as a web server must be True

hostname: Hostname if serve is active, defaults to '0.0.0.0'

port: Port if serve is active, defaults to 80

pypy: Name of the pypy interpreter to run ABCE super fast. e.G. 'pypy' or 'pypy3'. The mainfile needs to be run with cpython e.G.: `python3 start.py`

Example:

```
parameter_mask = {'name': 'name',
                  'random_seed': None,
                  'rounds': 40,
                  'num_firms': (0, 100, 100000),
                  'num_households': (0, 100, 100000),
                  'exponent': (0.0, 0.5, 1.1),
                  'several_options': ['opt_1', 'opt_2', 'opt_3']
                  'list_to_edit': "['brd', 'mlk', 'add']",
                  'dictionary_to_edit': "{'v1': 1, 'v2': 2}"}

names = {'num_firms': 'Number of Firms'}

@gui(parameter_mask, names,
     title="Agent-Based Computational Economics",
     serve=False)
def main(simulation_parameters):
    parameter_list = eval(simulation_parameters['list_to_edit'])
    simulation = Simulation()
    firms = simulation.build_agents(Firm,
        simulation_parameters['num_firms'])
    households = simulation.build_agents(Household,
        simulation_parameters['num_households'])

    for r in range(simulation_parameters['rounds']):
        simulation.advance_round(r)
        firms.work()
        households.buy()

if __name__ == '__main__':
    main(simulation_parameters)
```

`Simulation.`**`graphs`**`()`

after the simulation is run, graphs() shows graphs of all data collected in the simulation. Shows the same output as the @gui decorator shows.

Example:

```
simulation = Simulation(...)
for r in range(100):
    simulation.advance_round(r)
    agents.do_something()
    ...

simulation.graphs()
```

It is possible to add text frames between the simulation results. In order to add a text frame to the simulation output, simply add a text file `.txt` or `.html` in either your source code directory or if generated during the simulation in the `simulation.path` directory (where the `.csv` are). The first line of the text file is the title the rest the text. The text can be formatted as html, the first line is separate from the rest. In the simulation results all frames are displayed in alphabetical order, the title in the first line of the text-file determines the position where the text frame is displayed. In order to display a text before all graphs, you have to put leading space (' ') in front of the title.

Files in this package

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.

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

Deploying an ABCE simulation on-line

Prepare your simulation to be displayed on the web

In order for your simulation to be able to be run on the web it must be running in the web browser. for this you need to add @gui(...) before the main function. Further @gui needs to be switched to serve:

```
...

title = "Computational Complete Economy Model on Climate Gas Reduction"
text = """ This simulation simulates climate change
"""

parameters = OrderedDict({'wage_stickiness': (0, 0.5, 1.0),
                                    'price_stickiness': (0, 0.5, 1.0),
                                    'network_weight_stickiness': (0, 0.5, 1.0),
                                    'carbon_tax': (0, 50.0, 80.0),
                                    'tax_change_time': 100,
                                    'rounds': 200})

@gui(parameters, text=text, title=title, serve=True)
def main(simulation_parameters):
    ...

    simulation = Simulation(processes=1)
```

```
    simulation.run()

    #simulation.graphs() This must be commented out or deleted
    simulation.finalize()


if __name__ == '__main__':
    main(parameters)
```

It is important to note that the `main()` function is not called, when start.py is imported! `if __name__ == '__main__':`, means that it is not called when start.py is imported. you can also simply delete the call of `main()`.

`@gui` is the part that generates the web application and runs it. `serve` must be set to True in `@gui(simulation_parameters, text=text, title=title, serve=True)`.

The easiest way to get your code to the server is via github. For this follow the following instructions. Push the simulation to github. If you are unsure what git and github is, refer to this **'gitimmersion.com<http://gitimmersion.com/>'_**. If your code is not yet a git repository change in your directory:

```
git init
git add --all
git commit -m"initial commit"
```

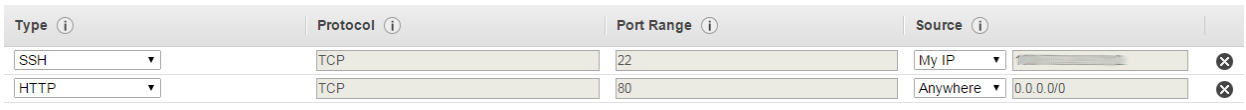
Go to github sign up and create a new repository. It will than display you instruction how to push an existing repository from the command line your, they will look like this:

```
git remote add origin https://github.com/your_name/myproject.git
git push -u origin master
```

Deploy you ABCE simulation on amazon ec2 or your own Ubuntu server

create an amazon ec2 instance following **'Amazon's tutorial here<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-launch-instance_linux.html>'_**

make sure that in step 7b, configure the security groups, such that you have a HTTP access. This setting allows access to port 80 (HTTP) from anywhere, and ssh access only from your IP address.



then from the console ssh into your account

```
ssh -i amazoninstanceweb2py.pem ubuntu@ec2-54-174-70-207.compute-1.amazonaws.com
```

Install the server software and ABCE requirements:

```
sudo python3 -m pip install abce
```

copy or clone your ABCE simulation into the ~/myproject directory the easiest way is to use a git repository, but you can also use scp:

```
git clone https://github.com/your_name/myproject.git
```

start simulation with nohup:

```
cd myproject

nohup sudo python3 start.py &
tail -f nohup.out
```

The last line displays the logging messages.

If something does not work delete all files and directories that have root as user. (find them with `ll`)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

A

abce (module), 31
abce.agent (module), 35
abce.agents.household (module), 49
abce.messaging (module), 43
abce.show (module), 69
accept() (abce.Trade method), 39
accept_contract() (abce.Contracting method), 56
accept_quote() (abce.quote.Quote method), 62
accept_quote_partial() (abce.quote.Quote method), 63
add_and_delete_agents() (abce.Simulation method), 32
advance_round() (abce.Simulation method), 32
Agent (class in abce), 35
agg_log() (abce.group.Group method), 53

B

build_agents() (abce.Simulation method), 32
buy() (abce.Trade method), 39

C

calculate_assetvalue() (abce.Contracting method), 56
calculate_liablityvalue() (abce.Contracting method), 56
calculate_netvalue() (abce.Contracting method), 56
calculate_valued_assets() (abce.Contracting method), 56
calculate_valued_liablities() (abce.Contracting method), 56
consume() (abce.agents.Household method), 49
consume_everything() (abce.agents.Household method), 49
Contracting (class in abce), 55
contracts_to_deliver() (abce.Contracting method), 56
contracts_to_deliver_all() (abce.Contracting method), 56
contracts_to_receive() (abce.Contracting method), 56
contracts_to_receive_all() (abce.Contracting method), 56
create() (abce.Agent method), 36
create_agent() (abce.Agent method), 36
create_ces() (abce.agents.FirmMultiTechnologies method), 58
create_cobb_douglas() (abce.agents.FirmMultiTechnologies method), 58
create_leontief() (abce.agents.FirmMultiTechnologies method), 59
create_production_function_many_goods() (abce.agents.FirmMultiTechnologies method), 59
create_production_function_one_good() (abce.agents.FirmMultiTechnologies method), 60
create_timestructured() (abce.Agent method), 36

D

Database (class in abce.database), 51
declare_expiring() (abce.Simulation method), 33
declare_perishable() (abce.Simulation method), 33
declare_round_endowment() (abce.Simulation method), 33
declare_service() (abce.Simulation method), 34
delete_agent() (abce.Agent method), 36
deliver_contract() (abce.Contracting method), 56
destroy() (abce.Agent method), 37

E

end_contract() (abce.Contracting method), 57

F

finalize() (abce.Simulation method), 34
Firm (class in abce.agents), 44
FirmMultiTechnologies (class in abce.agents), 58

G

get_buy_offers() (abce.Trade method), 40
get_buy_offers_all() (abce.Trade method), 40
get_contract_offers() (abce.Contracting method), 57
get_messages() (abce.messaging.Messaging method), 43
get_messages_all() (abce.messaging.Messaging method), 44
get_offers() (abce.Trade method), 40

get_offers_all() (abce.Trade method), 40
get_quotes() (abce.quote.Quote method), 63
get_quotes_all() (abce.quote.Quote method), 63
get_sell_offers() (abce.Trade method), 41
get_sell_offers_all() (abce.Trade method), 41
get_utility_function() (abce.agents.Household method), 49
give() (abce.Trade method), 41
graphs() (abce.Simulation method), 34, 70
group (abce.Agent attribute), 37
gui() (in module abce.gui), 69

H

Household (class in abce.agents), 49

I

id (abce.Agent attribute), 37
init() (abce.Agent method), 37

L

log() (abce.database.Database method), 51
log_change() (abce.database.Database method), 52

M

message() (abce.messaging.Messaging method), 44
Messaging (class in abce.messaging), 43

N

name (abce.Agent attribute), 37
net_value() (abce.agents.Firm method), 45
net_value() (abce.agents.FirmMultiTechnologies method), 60
NotEnoughGoods, 54

O

observe_begin() (abce.database.Database method), 52
observe_end() (abce.database.Database method), 52
Offer() (in module abce.trade), 43
offer_good_contract() (abce.Contracting method), 57

P

panel_log() (abce.group.Group method), 53
path (abce.Simulation attribute), 35
pay_contract() (abce.Contracting method), 57
peak_buy_offers() (abce.Trade method), 41
peak_offers() (abce.Trade method), 41
peak_sell_offers() (abce.Trade method), 42
possession() (abce.Agent method), 37
possessions() (abce.Agent method), 37
predict_net_value() (abce.agents.Firm method), 45
predict_net_value() (abce.agents.FirmMultiTechnologies method), 61
predict_produce_input() (abce.agents.Firm method), 45
predict_produce_input() (abce.agents.FirmMultiTechnologies method), 61
predict_produce_output() (abce.agents.Firm method), 46
predict_produce_output() (abce.agents.FirmMultiTechnologies method), 61
predict_utility() (abce.agents.Household method), 50
produce() (abce.agents.Firm method), 46
produce() (abce.agents.FirmMultiTechnologies method), 62
produce_use_everything() (abce.agents.Firm method), 46
produce_use_everything() (abce.agents.FirmMultiTechnologies method), 62

Q

Quotation() (in module abce.quote), 64
Quote (class in abce.quote), 62
quote_buy() (abce.quote.Quote method), 63
quote_sell() (abce.quote.Quote method), 63

R

reject() (abce.Trade method), 42
request_good_contract() (abce.Contracting method), 57
round (abce.Agent attribute), 37

S

sell() (abce.Trade method), 42
send() (abce.messaging.Messaging method), 44
set_ces() (abce.agents.Firm method), 46
set_cobb_douglas() (abce.agents.Firm method), 47
set_cobb_douglas_utility_function() (abce.agents.Household method), 50
set_leontief() (abce.agents.Firm method), 47
set_production_function() (abce.agents.Firm method), 48
set_production_function_many_goods() (abce.agents.Firm method), 48
set_utility_function() (abce.agents.Household method), 50
Simulation (class in abce), 31
sufficient_goods() (abce.agents.Firm method), 49
sufficient_goods() (abce.agents.FirmMultiTechnologies method), 62

T

take() (abce.Trade method), 42
time (abce.Agent attribute), 37
time (abce.Simulation attribute), 35
Trade (class in abce), 38

W

was_delivered_last_round() (abce.Contracting method), 57

was_delivered_this_round() (abce.Contracting method), 57
was_paid_last_round() (abce.Contracting method), 57
was_paid_this_round() (abce.Contracting method), 57