Agent-Based Computational Economics Documentation

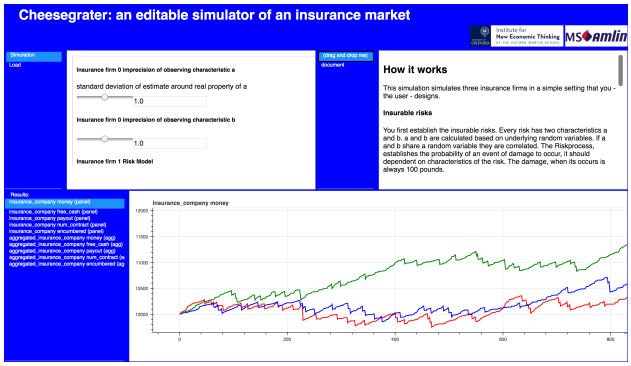
Release 0.9.3b0

Davoud Taghawi-Nejad

Oct 07, 2021

Contents

abcEconomics is a Python based modeling platform for economic simulations. abcEconomics comes with standard functions to simulations of trade, production and consumption. The modeler can concentrate on implementing the logic and decisions of an agents; abcEconomics takes care of all exchange of goods and production and consumption.



In abcEconomics goods have the physical properties of goods in reality in the sense that 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. The ownership and transformations (production or consumption) of goods are automatically handled by the platform.

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

abcEconomics 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 system. The speed advantages of using abcEconomics with multi-processes enabled. abcEconomics are typically only observed for 10000 agents and more. Below, it might be slower than pure python implementation. abcEconomics supports pypy3, which is approximately 30 times faster than CPython.

abcEconomics provides two more additions to the Python language. First of all, agent groups can be executed simultaneously. Secondly agents can interact with each other sending messages (and goods).

The audience of abcEconomics are economists that want to model agent-based models of trade and production.

abcEconomics does support an accounting framework for financial simulations. abcFinance can be downloaded here.

abcEconomics runs on macOS, Windows, and Linux. abcEconomics runs 30x faster on pypy!

CHAPTER 1

Introduction

1.1 Design

abcEconomics'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 or using pypy.

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 especially beginner friendly, but also easy to learn for people who already know object oriented programming languages such as Java, C++ or even MATLAB. abcEconomics 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.

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. Pypy even provides a just in time complier Python is compiled to bytecode and than executed. abcEconomics 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.

abcEconomics 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.

1.1.1 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. [Serenko2002], [Allan2010] [Societies2009], [Tobias2004], [Railsback2006], [abmcomparison-Wikipedia2013]. The articles 'Tools of the Trade' by Madey and Nikolai [Societies2009] and 'Survey of Agent Based Modelling and Simulation Tools' by Allan [Allan2010] 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 [abmcomparisonWikipedia2013] 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 [Allan2010] 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' [Tobias2004] 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 [Societies2009] 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 [Allan2010] 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 [Allan2010] and our research a sparse documentation.

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

Moduleco, because it has according to Allan [Allan2010] 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.

1.1.2 General differences to other agent-based modeling platforms

First of all abcEconomics 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 abcEconomics, 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 abcEconomics and other platforms is that abcEconomics introduces the physical good as an ontological object in the simulation. Goods can be exchanged and transformed. abcEconomics 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 abcEconomics makes sure that the transaction is cleared between the two agents.

Thirdly, abcEconomics 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 abcEconomics, 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[2] publicly available packages, that could be used in abcEconomics. 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 abcEconomics a spacial representation is possible, but not a design goal. However, since agents in abcEconomics 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 abcEconomics could not be a good choice for a problem where the spacial position plays a role. If for example the model has different transport costs or other properties according to the geographical position of the agents, but the agent's do not move or the movement does not have to be represented graphically, abcEconomics could still be a good choice.

Difference to MASON

Masons is a single-threaded discrete event platform that is intended for simulations of social, biological and economical systems. [Luke]. 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. abcEconomics 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. abcEconomics 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 abcEconomics 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. abcEconomics 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 abcEconomics 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. [Tisue2004] Python on the other hand is a compiled[3] general programming language. Consequently it is faster than NetLogo.

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

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

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

One difference between Netlogo and abcEconomics is that it has the concept of the observer agent, while in abcEconomics 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. [Collier] [NORTH2005] 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. [NORTH2005a]

Repast does allow static and dynamic scheduling. [Collier]. abcEconomics, does not (yet) allow for dynamic scheduling. In abcEconomics, the order of actions - or in abcEconomics 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 abcEconomics is ease of use. While in Repast it is necessary to subclass the scheduler in abcEconomics it is sufficient to specify the schedule and pass it the Simulation class.

Repast is vast, it contains 210 classes in 9 packages [Collier]. abcEconomics, thanks to its limited scope and Python, has only 6 classes visible to the modeler in a single package.

1.1.3 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 abcEconomics is that this is automatically handled.

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

Every agent in abcEconomics must inherit from the abcEconomics.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.

1.2 Download and Installation

abcEconomics works exclusively with python 3!

1.2.1 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 pip3 install abcEconomics
```

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

1.2.2 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 pip3 install abcEconomics
```

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 pip3 install abcEconomics
```

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

[2] If this fails `sudo apt-add-repository universe` and `sudo apt-get update`

[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/

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 abcEconomics

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

1.2.3 Installation Windows

abcEconomics 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 abcEconomics
```

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

1.2.4 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

1.2.5 If you have any problems with the installation

Mail to: DavoudTaghawiNejad@gmail.com

1.3 Interactive jupyter notebook Tutorial

1.3.1 You will learn how to:

1.3.2 Create Simulation & Agents

1.3.3 Run a simulation and advance time

1.3.4 Give Goods & Trade

1.3.5 Capturing Data and Pandas

1.3.6 Communication between Simulation and Agents

1.4 Walk through

In order to learn using abcEconomics we will now dissect and explain a simple abcEconomics 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

abcEconomics, 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.

1.4.1 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
```

(continues on next page)

(continued from previous page)

```
    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 abcEconomics import Simulation
from firm import Firm
from household import Household


def main():
    simulation = Simulation()

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

    for r in range(100):
        simulation.advance_round(r)
        households.refresh_services(service='labor' , derived_from='labor_
↪endowment', units=1)
        households.sell_labor()
        firms.buy_labor()
        firms.production()
        (households + firms).panel_log(goods=['money', 'GOOD'])
        households.panel_log(variables=['current_utility'])
        firms.sell_goods()
        households.buy_goods()
        households.consumption()

    simulation.finalize()

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

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

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 abcEconomics, 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 'perish' every round. These properties have to be refreshed at the end of every round:

```
for round in range(1000):
    simulation.advance_round(round)
    # ...
    households.refresh_services(service='labor' , derived_from='labor_endowment',␣
↪units=1)
```

In this example, the refresh_services removes the existing 'labor' goods and regenerates 1 unit of labor from scratch from every unit of labor_endowment

One important remark, for a logically consistent **macro-model** it is best to not create any goods during the simulation, but only in *abcEconomics.Agent.init()*. During the simulation the only new goods should be created by `abcEconomics.Goods.refresh_services()`. In this way the economy is physically closed.

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

households.agg_log('household', goods=['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 abcEconomics 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)* (`abcEconomics.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.

1.4.2 The agents

The Household agent

```
import abcEconomics


class Household(abcEconomics.Agent, abcEconomics.Household):
    def init(self):
        """ 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.labor_endowment = 1
        self.utility_function = self.create_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
        """
```

(continues on next page)

(continued from previous page)

```
        self.current_utility = self.consume(self.utility_function, ['GOOD'])
        self.log('HH', self.current_utility)
```

The Firm agent

```
import abcEconomics


class Firm(abcEconomics.Agent, abcEconomics.Firm):
    def init(self):
        """ 1. Gets an initial amount of money
        2. create a cobb_douglas function: GOOD = 1 * labor ** 1.
        """
        self.create('money', 1)
        self.inputs = {"labor": 1}
        self.output = "GOOD"
        self.pf = self.create_cobb_douglas(self.output, 1, self.inputs)

    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(self.pf, self.inputs)

    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["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 *abcEconomics* module and the *`abcEconomics.NotEnoughGoods`* exception

```
import abcEconomics
from abcEconomics import NotEnoughGoods
```

This imports the module abcEconomics 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 *`abcEconomics.Agent`*. It automatically inherits `abcEconomics.Trade` - `abcEconomics.Messenger` and `abcEconomics.Logger`

```
class Agent(abcEconomics.Agent):
```

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

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

To create an agent that can produce:

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

You see our Household agent inherits from *abcEconomics.Agent*, which is compulsory and abcEconomics.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 abcEconomics's init method, which is called when the agents are created

```
def init(self, parameters, agent_parameters):
    self.labor_endownment = 1
    self.utility_function = self.create_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 abcEconomics.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 abcEconomics.Household.create_cobb_douglas_utility_function() from abcEconomics.Household. With abcEconomics.Household.create_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 abcEconomics's methods with the agents methods.** For example if one would call the consumption(self) method below consume(self), abcEconomics's consume function would not work anymore.

```
class Household(abcEconomics.Agent, abcEconomics.Household):
    def init(self, simulation_parameters, agent_parameters):
        self.labor_endowment = 1
```

(continues on next page)

(continued from previous page)

```
        self.utility_function = self.create_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(abcEconomics.Agent, abcEconomics.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 `abcEconomics.Firm`. `abcEconomics.FirmMultiTechnologies` offers a more advanced interface for firms with layered production functions.

Trade

abcEconomics 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)
```

(continues on next page)

(continued from previous page)

```
            except NotEnoughGoods:
                self.accept(offer, self['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['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)` (`abcEconomics.Trade.reject()`).

You can find a detailed explanation how trade works in `abcEconomics.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 accessible as csv files in the `simulation.path` directory

Manual in agent logging

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

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.

1.5 Tutorial for Plant Modeling

1. Lets write the 1st agent:

    a. Create a file chpplant.py import abcEconomics and create a Plant class.

```
import abcEconomics


class CHPPlant(abcEconomics.Agent, abcEconomics.Firm):
```

b. In `def init(self):` (not __init__!) we need to create some initial goods

```
class CHPPlant(...):

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

c. 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)`. So there add the following lines, `class CHPPLant(...):`

```
class CHPPlant(...):

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

        def production_function(biogas, water):
            electricity = biogas ** 0.25 * water ** 0.5
            steam = min(biogas, water)
            biogas = 0
            water = 0
            return locals()

        self.production_function = production_function
```

The `def production_function(biogas, water):` returns the production result as a dictionary. (try `print(production_function(10, 10))`). Each key is a good that is produced or what remains of a good after the production process. If goods are used up they must be set to 0. 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.

The `production_function` function is local function in the `init` method. Make sure the `return locals()` is part of the `def production_function(...):` not of the `def init(self):` method.

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

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

e. also add:

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

3. Create a file `start.py` to run this incomplete simulation.

   a. Import abcEconomics and the plant:

   ```
   import abcEconomics
   from chpplant import CHPPlant
   ```

   b. Create a simulation instance:

   ```
   simulation = abcEconomics.Simulation()
   ```

   c. Build an a plant

   ```
   chpplant = 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`. Therefore its name is the tuple ('chpplant', 0)

   1. Loop over the simulation:

   ```
   for r in range(100):
       simulation.advance_round(r)
       chpplant.production()
       chpplant.panel_log(goods=['electricity', 'biogas', 'water',
   ↪'steam'], variables=[])
       chpplant.refill()

   simulation.finalize()
   ```

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

   simulation.advance_round(r) sets the time r. Lastly `simulation.finalize()` tells the simulation that the loop is done. Otherwise the program hangs at the end.

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 the production function to

```
def production_function(steam, electricity):
    biogas = min(electricity, steam)
    water = min(electricity, steam)
    electricity = 0
    steam = 0
    return locals()
```

   d. Given the new technology, we need to feed different goods into our machines. Replace the production step

```
def production(self):
    self.produce(self.production_function, {'steam': self['steam'],
↪'electricity': self['electricity']})
```

   self['steam'], looks up the amount of steam the company owns. self.not_reserved['steam'], would look up the amount of steam a company owns minus all steam that is offered to be sold to a different company.

   e. 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. Add another method to the ADPlant class.

```
def selling(self):
    amount_biogas = self['biogas]
    amount_water = self['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.

   f. In CHPPlant respond to this offer, by adding the following method.

```
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['money']):
        self.accept(water_offer)
        self.accept(biogas_offer)
    else:
        quantity_allocationg_half_my_money = self['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['money']))
```

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

   g. reversely in CHPPlant:

```
def selling(self):
    amount_electricity = self['electricity']
    amount_steam = self['steam']
```

(continues on next page)

(continued from previous page)

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

h. 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['money']):
        self.accept(el_offer)
        self.accept(steam_offer)
    else:
        quantity_allocationg_half_my_money = self['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['money']))
```

8. let's modify start.py

   b. in `start.py` import thu ADPlant:

```
from adplant import ADPlant
```

   and

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

   c. 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()
```

9. now it should run again.

1.6 Examples

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

1.6.1 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				Extended GUI				
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							

1.6.2 Models

CCE

This is the most complete example featuring an agent-based model of climate change tax policies for the United States. It is databased 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 abcEconomics'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.

1.7 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. abcEconomics 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 abcEconomics 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 abcEconomics 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 abcEconomics, 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

2.1 The simulation in start.py

The best way to start creating a simulation is by copying the start.py file and other files from 'abcEconomics/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 abcEconomics import *


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

Note two things are important: there must be a

`finalize()` at the end otherwise the simulation blocks at the end. Furthermore, every round needs to be announced using simulation.advance_round(time), where time is any representation of time.

class abcEconomics.**Simulation**(*name='abcEconomics', random_seed=None, trade_logging='off', processes=1, dbplugin=None, dbpluginargs=[], path='auto', multiprocessing_database=False*)

Bases: `object`

This is the class in which the simulation is run. Actions and agents have to be added. Databases and resource declarations can be added. Then run 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 runs in parallel. Each process hosts a share of the agents. By default, if this parameter is not specified, *processes* 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. **For easy debugging set processes to 1, this way only one agent runs at a time and only one error message is displayed**

check_unchecked_msgs: check every round that all messages have been received with get_massages or get_offers.

path: path for database use None to omit directory creation.

dbplugin, dbpluginargs: database plugin, see *Database Plugins*

Example:

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

Example for a simulation:

```
num_firms = 5
num_households = 2000

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

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 time in range(100):
    self.time = time
    endowment.refresh_services('labor', derived_from='labor_endowment', units=5)
    households.recieve_connections()
    households.offer_capital()
    firms.buy_capital()
    firms.production()
    if time == 250:
        centralbank.intervention()
    households.buy_product()
    all.after_sales_before_consumption()
    households.consume()

w.finalize()
```

advance_round(*time*)

build_agents (*AgentClass*, *group_name*, *number=None*, *agent_parameters=None*, ***parameters*)

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.

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

any other parameters: are directly passed to the agent

Example:

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

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

create_agent (*AgentClass*, *group_name*, *simulation_parameters=None*, *agent_parameters=None*)

create_agents (*AgentClass*, *group_name*, *simulation_parameters=None*, *agent_parameters=None*, *number=1*)

delete_agent (**ang*)

delete_agents (*group*, *ids*)

This deletes a group of agents. The model has to make sure that other agents are notified of the death of agents in order to stop them from corresponding with this agent. Note that if you create new agents after deleting agents the ID's of the deleted agents are reused.

Args:

group: group of the agent

ids: a list of ids of the agents to be deleted in that group

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()
```

time
Set and get time for simulation and all agents

2.2 Agents

The *abcEconomics.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 abcEconomics.Firm or to create a consumer the abcEconomics.Household.

For detailed documentation on:

Trading, see *Trader*

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

Messaging between agents, see Messenger.

class abcEconomics.**Agent**(*id*, *agent_parameters*, *simulation_parameters*, *name=None*)

Bases: abcEconomics.logger.logger.Logger, *abcEconomics.agents.trader.Trader*, abcEconomics.agents.messenger.Messenger, abcEconomics.agents.goods.Goods

Every agent has to inherit this class. It connects the agent to the simulation and to other agent. The abcEconomics.Trade, abcEconomics.Logger and abcEconomics.Messenger classes are included. An agent can also inheriting from abcEconomics.Firm, abcEconomics.FirmMultiTechnologies or abcEconomics.Household classes.

Every method can return parameters to the simulation.

For example:

```
class Household(abcEconomics.Agent, abcEconomics.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['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())
```

group = None
self.group returns the agents group or type READ ONLY!

id = None
self.name returns the agents name, which is the group name and the id

init()
This method is called when the agents are build. It can be overwritten by the user, to initialize the agents. Parameters are the parameters given to `abcEconomics.Simulation.build_agents()`.

Example:

```
class Student(abcEconomics.Agent):
    def init(self, rounds, age, lazy, school_size):
        self.rounds = rounds
        self.age = age
        self.lazy = lazy
        self.school_size = school_size

    def say(self):
        print('I am', self.age ' years old and go to a school
        that is ', self.school_size')


def main():
    sim = Simulation()
    students = sim.build_agents(Student, 'student',
                                agent_parameters=[{'age': 12, lazy: True},
                                                  {'age': 12, lazy: True},
                                                  {'age': 13, lazy: False},
                                                  {'age': 14, lazy: True}],
                                rounds=50,
                                school_size=990)
```

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'

2.3 Groups

class abcEconomics.**Group**(*sim*, *scheduler*, *names*, *agent_arguments=None*)
Bases: `object`

A group of agents. Groups of agents inherit the actions of the agents class they are created by. When a group is called with an agent action all agents execute this actions simultaneously. e.G. `banks.buy_stocks()`, then all banks buy stocks simultaneously.

agents groups are created like this:

```
sim = Simulation()

Agents = sim.build_agents(AgentClass, 'group_name', number=100, param1=param1,␣
↪param2=param2)
Agents = sim.build_agents(AgentClass, 'group_name',
                          param1=param1, param2=param2,
                          agent_parameters=[dict(ap=ap1_agentA, ap=ap2_agentA),
                                            dict(ap=ap1_agentB, ap=ap2_agentB),
                                            dict(ap=ap1_agentC, ap=ap2_agentC)])
```

Agent groups can be combined using the + sign:

```
financial_institutions = banks + hedgefunds
...
financial_institutions.buy_stocks()
```

or:

```
(banks + hedgefunds).buy_stocks()
```

Simultaneous execution means that all agents act on the same information set and influence each other only after this action.

individual agents in a group are addressable, you can also get subgroups (only from non combined groups):

banks[5].buy_stocks()

(banks[6,4] + hedgefunds[7,9]).buy_stocks()

agents actions can also be combined:

```
buying_stuff = banks.buy_stocks & hedgefunds.buy_feraries
buy_stocks()
```

or:

```
(banks.buy_stocks & hedgefunds.buy_feraries)()
```

agg_log (*variables=[], goods=[], func={}, len=[]*)

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

Args:

goods (list, optional): a list of all goods 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(goods=['money', 'input'],
                  variables=['production_target', 'gross_revenue'])
    households.buying()
```

by_name (*name*)

Return a group of a single agents by its name

by_names (*names*)

Return a callable group of agents from a list of names.group

Example:

```
banks.by_names(['UBS', 'RBS', "DKB"]).give_loans()
```

create_agents (*Agent, number=1, agent_parameters=None, **common_parameters*)

Create new agents to this group. Works only for non-combined groups

Args:

Agent: The class used to initialize the agents

agent_parameters: List of dictionaries of agent_parameters

number: number of agents to create if agent_parameters is not set

any keyword parameter: parameters directly passed to `agent.init` methood

Returns: The id of the new agent

`delete_agents`(*names*)

Remove an agents from a group, by specifying their id.

Args:

ids: list of ids of the agent

Example:

```
students.delete_agents([1, 5, 15])
```

`panel_log`(*variables=[], goods=[], func={}, len=[]*)

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

Args:

goods (list, optional): a list of all goods 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(goods=['money', 'input'],
                    variables=['production_target', 'gross_revenue'])
    households.buying()
```

2.4 Physical goods and services

2.4.1 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
- `self['money']` returns the quantity an agent possesses

2.4.2 Services

Services are like goods, but the need to be declared as services in the simulation `abcEconomics.__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. `simulation.declare_service('adults', 8, 'labor')`.

2.5 Trader

class abcEconomics.agents.trader.**Trader**(*id*, *agent_parameters*, *simulation_parameters*)

Bases: `object`

Agents can trade with each other. The clearing of the trade is taken care of fully by abcEconomics. 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['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
```

(continues on next page)

(continued from previous page)

```
    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['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 `abcEconomics.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 `abcEconomics.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]:
```

(continues on next page)

(continued from previous page)

```
        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

`abcEconomics.agents.trader.Offer`(*sender*, *receiver*, *good*, *quantity*, *price*, *currency*, *sell*, *status*, *final_quantity*, *id*, *made*, *status_round*)

This is an offer container that is send to the other agent. You can access the offer container both at the receiver as well as at the sender, if you have saved the offer. (e.G. self.offer = self.sell(...))

it has the following properties:

sender: this is the name of the sender

receiver: This is the name of the receiver

currency: The other good against which the good is traded.

good: the good offered or demanded

quantity: the quantity offered or demanded

price: the suggested transaction price

sell: this can have the values False for buy; True for sell

status:

'new': has been created, but not answered

'accepted': trade fully accepted

'rejected': trade rejected

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

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

final_quantity: If the offer has been answerd this returns the actual quantity bought or sold. (Equal to quantity if the offer was accepted fully)

id: a unique identifier

2.6 Messaging

2.7 Firm and production

class `abcEconomics.agents.Firm`

Bases: `object`

With `self.produce` a firm produces a good using production functions. For example the following farm has a cobb-douglas production function:

class Farm(abcEconomics.Agent, abcEconomics.Firm):

def init(self):

self.production_function = create_cobb_douglas({'land': 0.7, 'capital': 0.1, 'labor': 0.2})

def firming(self):

self.produce(self.production_function, {{'land': self['land'], 'capital': self['capital'], 'labor': 2}})

Production functions can be auto generated with:

- py:meth:~*abcEconomics.Firm.create_cobb_douglas* or
- py:meth:~*abcEconomics.Firm.create_ces* or

- py:meth:~*abcEconomics.Firm.create_leontief*

or specified by hand:

A production function looks like this:

```
def production_function(wheels, steel, stearing_wheels, machines):
    result = {'car': min(wheels / 4, steel / 10, stearing_wheels),
              'wheels': 0,
              'steel': 0,
              'steering_wheels': 0,
              'machine': machine * 0.9}
    return result
```

Or more readably like this:

def production_function(wheels, steel, stearing_wheels, machines): car = min(wheels / 4, steel / 10, stearing_wheels) wheels = 0 steel = 0 stearing_wheels = 0 machine = machine * 0.9 return locals()

This production function, produces one car for every four wheels, 10 tonnes of steel and one stearing_wheel, it also requires one machine. Wheels, steel and stearing_wheels are completely used. The plant is not used and the machine depreciates by 10%.production.

A production function can also produce multiple goods. The last line `return locals()`, can not be omitted. It returns all variables you define in this function as a dictionary.

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

produce (*production_function*, *input_goods*, *results=False*)

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:~*abcEconomics.Firm.create_production_functio* py:meth:~*abcEconomics.Firm.create_cobb_douglas* or py:meth:~*abcEconomics.Firm.create_leontief*

input goods dictionary or list: dictionary containing the amount of input good used for the production or a list of all goods that get completely used.

results: If True returns a dictionary with the used and produced goods.

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)

self.produce(car_production_function, ['tire', 'metal', 'plastic'])  #␣
↪produces using all goods
```

2.8 Household and consumption

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

class abcEconomics.agents.**Household**

Bases: object

consume(*utility_function*, *input_goods*)

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

A utility_function, has to be set before see py:meth:*~abcEconomics.Household.create_cobb_douglas_utility_function* or manually; see example.

Args:

utility_function: A function that takes goods as parameters and returns a utility or returns (utility, left_over_dict). Where left_over_dict is a dictionary of all goods that are not completely consumed

input goods dictionary or list: dictionary containing the amount of input good used consumed or a list of all goods that get completely consumed.

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

Returns: The utility as a number. To log it see example.

Example:

```
def utility_function(car, cookies, bike):
    utility = car ** 0.5 * cookies ** 0.2 * bike ** 0.3
    cookies = 0  # cookies are consumed, while the other goods are not␣
↪consumed
    return utility, locals()


def utility_function(cake, cookies, bonbons):  # all goods get completely␣
↪consumed
    utility = cake ** 0.5 * cookies ** 0.2 * bonbons ** 0.3
    return utility

self.consumption_set = {'car': 1, 'cookies': 2000, 'bike':  2}
self.consume_everything = ['car', 'cookies', 'bike']
try:
```

(continues on next page)

(continued from previous page)

```
    utility = self.consume(utility_function, self.consumption_set)
except NotEnoughGoods:
    utility = self.consume(utility_function, self.consume_everything)
self.log('utility': {'u': utility})
```

create_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})

2.9 Observing agents and logging

There are different ways of observing your agents:

Trade Logging: abcEconomics 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*

2.9.1 Trade Logging

By default abcEconomics 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!).

2.9.2 Manual logging

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

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

Log logs dictionaries. To log your own variable:

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

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

2.9.3 Panel Data

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

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

Args:

goods (list, optional): a list of all goods 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(goods=['money', 'input'],
                variables=['production_target', 'gross_revenue'])
    households.buying()
```

2.9.4 Aggregate Data

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

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

Args:

goods (list, optional): a list of all goods 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(goods=['money', 'input'],
                variables=['production_target', 'gross_revenue'])
    households.buying()
```

2.10 Retrieval of the simulation results

Agents can log their internal states and the simulation can create panel data. `abcEconomics.logger`.

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 = abcEconomics.Simulation(...)
...
simulation.run()

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

2.11 NotEnoughGoods Exception

exception abcEconomics.**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

3.1 Quote

3.2 Spatial and Netlogo like Models

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

3.2.1 Netlogo like models

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

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

On github

A wrapper file to start the graphical representation and the simulation

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

(continues on next page)

(continued from previous page)

```
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],
                           "abcEconomics and MESA integrated",
                           {'num_agents': 1000, 'x_size': x_size, 'y_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 abcEconomics and mesa.
The model and scheduler specification are taken care of in
abcEconomics 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)
```

(continues on next page)

(continued from previous page)

```
"""
import abcEconomics as 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(x)
    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):
        super().__init__(name='abcEconomics and MESA integrated',
                         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,
                                        grid=self.grid)
        # abcEconomics 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)]
        self.r = 0

    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.advance_round(self.r)
        self.agents.move()
        self.agents.give_money()
        self.wealths = self.agents.report_wealth()
        # agents report there wealth in a list self.wealth
        self.datacollector.collect(self)
        # collects the data
        self.r += 1


if __name__ == '__main__':
    """ If you run model.py the simulation is executed without graphical
    representation """
```

(continues on next page)

(continued from previous page)

```
    money_model = MoneyModel(1000, 20, 50)
    for r in range(100):
        print(r)
        money_model.step()
```

A simple agent

```
import abcEconomics as 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, grid):
        self.grid = 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.name, good='money', quantity=1)
            except abce.NotEnoughGoods:
                pass

    def report_wealth(self):
        return self['money']
```

3.3 Create Plugins

abcEconomics has three plugin so far: abcFinance, abcLogistics, abcCython. If you want to author your own plugin - its dead simple. All you have to do is write a class that inherits from Agent in agent.py. This class can overwrite:

```
def __init__(self, id, group, trade_logging, database, random_seed, num_managers,
             agent_parameters, simulation_parameters,
```

(continues on next page)

(continued from previous page)

```
              check_unchecked_msgs, start_round=None):
def _begin_subround(self):
def _end_subround(self):
def _advance_round(self, time):
```

For example like this:

```
class UselessAgent(abcEconomics.Agent):
    def __init__(self, id, group, trade_logging, database, random_seed, num_managers,
                 agent_parameters, simulation_parameters,
                 check_unchecked_msgs, start_round=None):
        super().__init__(id, group, trade_logging,
                         database, random_seed, num_managers, agent_parameters,
                         simulation_parameters, check_unchecked_msgs,
                         start_round):
        print("Here i begin")

    def _begin_subround(self):
        super()._begin_subround()
        print('subround begins')

    def _end_subround(self):
        super()._end_subround()
        print('subround finishes')

    def _advance_round(self, time):
        super()._advance_round(time)
        print('Super I made it to the next round')

    def ability(self):
        print("its %r o'clock" % self.time)
        print("the simulation called my ability")
```

Do not overwrite the init(parameters, simulation_parameters) method

3.4 Database Plugins

In order to write custom logging functions, create a class with your custom logging:

```
class CustomLogging:
    def __init__(self, dbname, tablename, arg3):
        self.db = dataset.connect('sqlite:///factbook.db')
        self.table = self.db[tablename]


    def write_everything(self, name, data):
        self.table.insert(dict(name=name, data=data))

    def close(self):
         self.db.commit()
```

The close method is called when the simulation in ended with simulation.finalize().

The CustomLogging class must be given to the simulation, in will be initialized with the dbpluginargs argument list:

```
sim = Simulation(name='mysim', dbplugin=CustomLogging, dbpluginargs=['somedb.db',
↪'sometable', 'arg3')
```

The agents can execute your custom logging function like this:

```
self.custom_log('write_everything', name='joe', data=5)
```

CHAPTER 4

Frequently asked Questions

4.1 How to share public information?

Agents can return information via a return statement at the end of a method. The returned variables are returned to start.py as a list of the values. It is often useful to include the agents name e.G. `return (self.name, info)`

The returned information can than be passed as arguments for another method:

```
for r in range(100):
    simulation.advance_round(r)
    agents.do_something()
    info = agents.return_info()
    agents.receive_public_information(info=info)
```

Currently only named function parameters are supported.

4.2 How to share a global state?

A shared global state, breaks multiprocessing, so if you want to run the simulation on multiple cores see 'How to share public information'. In single processing mode, you can give each agent a dictionary as a parameter. All information in this dictionary is shared.

4.3 How to access other agent's information?

Once again this breaks multiprocessing. But you can return an agent's self and give it as a parameter to other agents.

4.4 How to make abcEconomics fast?

There is several ways:

1. Use pypy3 instead of CPython, it can be downloaded here: https://pypy.org/download.html. With pypy3 you can run the same code as with CPython, but about 30 times faster.
2. If you use scipy pypy3 might not work. Use numba instead. http://numba.pydata.org
3. Run the simulation with a different number of processes. With very simple agents and many messages one is optimal, with compute intensive agents number of physical processors minus one is usually most efficient. But experimenting even with more processes than physical processors might be worth it.
4. Use kernprof to find which agent's method is slowest. https://github.com/rkern/line_profiler

4.5 How to load agent-parameters from a csv / excel / sql file?

The list of parameters can be passed as agent_parameters and is passed to init, as keyword arguments:

```
with open('emirati.csv', 'r') as f:
    emirati_file = csv.DictReader(f)
    emiratis_data = list(emirati_file)


emiratis = sim.build_agents(Emirati, 'emirati', agent_parameters=emiratis_data)
```

Note that list(file) is necessary.

4.6 Troubleshooting

Can't find my problem: ask a question on https://stackoverflow.com, tagging the question as abcEconomics and send an email to davoud@taghawi-nejad.de. If its a bug or an enhancement open an issue on github: https://github.com/DavoudTaghawiNejad/abcEconomics/issues

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

P

R

S

T