Black Rhino

Example 1: Simple exchange model with no market clearing

Documentation

Running:

**python black\_rhino.py**

Folders & Files:

**black\_rhino.py** – the main file calling the example to be run. It has hardcoded arguments, this can be changed to arguments taken from the console by commenting line 40 and uncommenting line 41. In this version we have args = ['./black\_rhino.py', "environments/", "test\_all\_methods", "log/"] which point to the environment config directory, the identifier of the simulation, and the log directory. This script initializes the environment, reads its config file, initializes the runner, and does the update. All of this is done x number of times, where x is the num\_simulations parameter from the config file described below.

**abm\_template**/ – folder contacting the current (at the time of creating the example) abm\_template package, i.e. the abstract classes upon which the source code is based, providing super-classes and super-methods which are inherited

**agents/** – config files for all agents, separately for banks (banks/), firms (firms/), and households (households/). Config files are xml files with parameters and transactions in them, as necessary. See the example below, for one of the banks in the example. It is essential that identifier is in the config files, and that identifiers are unique (not only among certain kinds of agents, but globally across agents, otherwise it will raise errors). The starting wrapper <bank></bank> can in principle be anything, but it’s good practice to keep them with the same names as the agent it belongs to. Parameters are within <parameter></parameter>, and require a type (static or variable), a name (string), and a value (usually a float or int, sometimes can be a string). If you require transactions to be read from the config files, to start the simulations, these are within <transaction></transaction> and require type, asset, from, to, amount, interest, maturity, and time\_of\_default. The first four are strings, the rest consists of floats or integers. It is important to note that from and to should be legal agent identifiers.

<bank identifier='bank\_test\_config\_id'>

<parameter type='static' name='interest\_rate\_loans' value='0.06'></parameter>

<parameter type='static' name='interest\_rate\_deposits' value='0.02'></parameter>

</bank>

**Environments/** – config file for the simulation, an xml file, see the example below. Config is essential to the work of the simulator. Identifier should be provided. Parameters are wrapped inside <parameter></parameter> and require type, name and value, similar to the agent configs above. There is the number of simulations and number of sweeps per simulation, as well as the number of particular agents, and the directories which store config files for agents.

<environment identifier='tests\_for\_all\_methods'>

<parameter type='static' name='num\_sweeps' value='100'></parameter>

<parameter type='static' name='num\_simulations' value='1'></parameter>

<parameter type='static' name='num\_banks' value='1'></parameter>

<parameter type='static' name='num\_firms' value='1'></parameter>

<parameter type='static' name='num\_households' value='1'></parameter>

<parameter type='static' name='bank\_directory' value='agents/banks/'></parameter>

<parameter type='static' name='firm\_directory' value='agents/firms/'></parameter>

<parameter type='static' name='household\_directory' value='agents/households/'></parameter>

</environment>

**log/** – a folder in which log file from the simulation will be written

**src/** – source files for the simulator:

**bank.py** – a class for banks (inherited from BaseAgent from abm\_template), one of the three kinds of agents in this example (banks, firms & households). It’s characterised by an identifier (should be a unique string), parameters (dictionary), state\_variables (a dictionary, not used in this example), and accounts (for storing transactions). In this particular example, banks are price setters, and contain parameters for interest rates on loans and deposits (parameters["interest\_rate\_loans"] & parameters["interest\_rate\_deposits"]). The variables can be changed with get/set standard functions as described in the file. There are functions for printing the bank (\_\_str\_\_) and the parameters. The parameters and transactions can be read through get\_parameters\_from\_file and get\_transactions\_from\_file. Accounts can be cleared or purged with the provided functions (clear\_accounts & purge\_accounts). There is a function for adding a transaction (add\_transaction), and two functions for getting the total amount or the number of instances of a certain kind of transaction in the bank’s accounts (get\_account & get\_account\_num\_transactions). A function check\_consistency checks whether assets and liabilities balance each other out. Implemented \_\_getattr\_\_ function means that parameters don’t need to be extracted from the dictionaries by hand (e.g. bank.parameters[“name\_of\_parameter”]), but can be called as an attribute directly (e.g. bank.name\_of\_parameter). This means that variable names need to be unique among parameters and state\_variables.

**firm.py** – a class for firms (inherited from BaseAgent from abm\_template), one of the three kinds of agents in this example (banks, firms & households). It’s characterised by an identifier (should be a unique string), parameters (dictionary), state\_variables (a dictionary, not used in this example), and accounts (for storing transactions). In this particular example, firms are producers, and contain parameter for productivity in the production function (parameters["productivity"]). The variables can be changed with get/set standard functions as described in the file. There are functions for printing the bank (\_\_str\_\_) and the parameters. The parameters and transactions can be read through get\_parameters\_from\_file and get\_transactions\_from\_file. Accounts can be cleared or purged with the provided functions (clear\_accounts & purge\_accounts). There is a function for adding a transaction (add\_transaction), and two functions for getting the total amount or the number of instances of a certain kind of transaction in the firm’s accounts (get\_account & get\_account\_num\_transactions). Implemented \_\_getattr\_\_ function means that parameters don’t need to be extracted from the dictionaries by hand (e.g. firm.parameters[“name\_of\_parameter”]), but can be called as an attribute directly (e.g. firm.name\_of\_parameter). This means that variable names need to be unique among parameters and state\_variables.

**household.py** – a class for households (inherited from BaseAgent from abm\_template), one of the three kinds of agents in this example (banks, firms & households). It’s characterised by an identifier (should be a unique string), parameters (dictionary), state\_variables (a dictionary, not used in this example), and accounts (for storing transactions). In this particular example, households are consumers, savers, and workforce, and contain parameters for the propensity to save, and for the number of workhours available to sell per each simulation step (parameters["propensity\_to\_save"] & parameters["labour"]). The variables can be changed with get/set standard functions as described in the file. There are functions for printing the bank (\_\_str\_\_) and the parameters. The parameters and transactions can be read through get\_parameters\_from\_file and get\_transactions\_from\_file. Accounts can be cleared or purged with the provided functions(clear\_accounts & purge\_accounts). There is a function for adding a transaction (add\_transaction), and two functions for getting the total amount or the number of instances of a certain kind of transaction in the household’s accounts (get\_account & get\_account\_num\_transactions). A function check\_consistency checks whether assets and liabilities balance each other out. Implemented \_\_getattr\_\_ function means that parameters don’t need to be extracted from the dictionaries by hand (e.g. household.parameters[“name\_of\_parameter”]), but can be called as an attribute directly (e.g. household.name\_of\_parameter). This means that variable names need to be unique among parameters and state\_variables.

**environment.py** – a class for the environment (inherited from BaseConfig from abm\_template), that is the global parameters of the simulation. It’s characterised by an identifier (should be a unique string), static\_parameters (dictionary), variable\_parameters (a dictionary, not used in this example), and accounts (for storing transactions). In particular we have parameters for the number of simulations to run (num\_simulations), number of sweeps (steps) per simulation (num\_sweeps), number of particular agents (num\_banks, num\_households, num\_firms), and the directories in which configs for specific agent types are stored (bank\_directory, firm\_directory, household\_directory). Environment also has lists containing agents: banks, firms, and households. There are standard get/set functions for these variables. There are standard functions for printing the environment, the parameters, and writing the config file (\_\_str\_\_, print\_parameters, write\_environment\_file). Function read\_xml\_config\_file reads the xml file with parameters, as described above. When the environment is initialized, it zeroes out all the variables, reads the config file from the supplied directory, and creates all the agents from their respective directories, and then reads the transactions from their configs as well (this utilizes initialize\_banks\_from\_files, initialize\_firms\_from\_files, initialize\_households\_from\_files, read\_transactions\_for\_banks, read\_transactions\_for\_firms, read\_transactions\_for\_households. Implemented get\_agent\_by\_id function returns an object with the agent with specified identifier. This function requires all agents to have unique identifiers (it is recommended their identifiers are prefixed with the type of the agent, e.g. bank\_id) Implemented \_\_getattr\_\_ function means that parameters don’t need to be extracted from the dictionaries by hand (e.g. environment.static\_parameters[“name\_of\_parameter”]), but can be called as an attribute directly (e.g. environment.name\_of\_parameter). This means that variable names need to be unique among parameters and state\_variables.

**transaction.py** – a class for transactions (inherited from BaseTransaction from abm\_template), which end up in accounts of the above-mentioned agents, a transaction has a type (type\_) which describes what it is (“deposits”, “loans”, etc.), asset (for investments, describes asset class), from (from\_) & to (these point to the agents which are the two sides of the transaction, e.g. for a deposit from\_ is a household and to is a bank), amount (value of the particular transaction in monetary terms), interest (interest accruing on the transactions every step in the simulations, 0 for non-accruing transactions), maturity (time in steps to maturity of the transactions, -1 for perpetuities), time\_of\_default (control variable checking for defaulted transactions). Note that not all of these are useful for this minimal example. There are functions to print and write the transactions, as well as this\_transactions function which assigns the above attributes to self.

**runner.py** – a class for handling the running of the model (inherited from BaseRunner from abm\_template), initializes the updater and runs the updating loop in the updater x number of times where x is num\_sweeps parameter from the environment config file. In our example it also writes the sweep number, the banks, and the lists of agents with their memory allocations at each sweep, for testing purposes.

**updater.py** – a class containing the model (inherited from BaseModel from abm\_template), it controls the updating of the agents and the environment. In this example the updating consists of one function (do\_update), which calls the other functions, in order:

* accrue\_interests – accrues interests on all transaction within the system; that is adds the amount to them determined by their interest rate parameter;
* endow\_labour – adds a transaction to the books of households with labour “manhours” equal to their endowment per simulation step as specified in the config;
* sell\_labour – the households sell their labour transactions to firms, and get cash in return;
* produce – firms turn bought manhours into goods according to a simple Leontief production function based only on labour and it’s productivity;
* consume – firms sell goods to the households and get cash in return;
* make\_deposits – households turn cash into deposits at the bank at this step;
* remove\_labour – removes all transactions from the households with labour, as labour cannot be stored overnight, and new labour endowment is given out at the start of each simulation step;
* remove\_goods – we also remove goods from the books of the agents, as goods are perishable in this simple model.

**helper.py** – helper class not used in the example, it’s used for testing and contains functions for initializing agents without config files for testing purposes.

Documentation created by Paweł Fiedor ([Pawel.Fiedor@uct.ac.za](mailto:Pawel.Fiedor@uct.ac.za)) on the 20th of April 2016 in Cape Town.