EWaGI Documentation V1.1

Philipp Harting Hankui Wang

April 30, 2024

Contents

1 Model **5**
1.1 Continuous space and grid initialization . . . 5
1.2 Demographic setup . . . 5
1.3 Agent Initialization . . . 6
1.4 ROW setup . . . 8

2 Firms **9**
2.1 chooseGoodsSupplier1() . . . 9
2.2 rdPlanning() . . . 10
2.3 financialPlanning() . . . 10
2.4 creditApplication() . . . 11
2.5 chooseCreditOffers() . . . 11
2.6 financialStatus() . . . 12
2.7 interestInstallmentPayments() . . . 12
2.8 debtRestructuring() . . . 13
2.9 replanningProduction() . . . 14
2.10 inputGoodsOrderRound1() . . . 14
2.11 inputGoodsDeliveryRound1() . . . 15
2.12 chooseGoodsSupplier2() . . . 15
2.13 inputGoodsOrderRound2() . . . 16
2.14 inputGoodsDeliveryRound2() . . . 16
2.15 payWages() . . . 17
2.16 processInnovation() . . . 17
2.17 productInnovation() . . . 18
2.18 specificSkillUpdate() . . . 18
2.19 productivityAdjustment() . . . 19
2.20 updateBalanceSheet() . . . 19
2.21 InputItem . . . 20
2.22 pricing . . . 20
2.23 computePlannedOutput . . . 21
2.24 computeInputDemandForProductionPlan . . . 21

2.25 labormarketActivities . . . 22
2.26 labormarketActivities2 . . . 23
2.27 production . . . 23
2.28 capitalStockAdjustment . . . 23
2.29 outputDelivery . . . 24
2.30 accounting . . . 24
2.31 Considerations . . . 25

3 Household **26**
3.1 setupIteration() . . . 26
3.2 determineConsumptionBudget() . . . 26
3.3 shoppingActivity() . . . 27
3.4 laborMarketActivities2() . . . 28
3.5 consumption() . . . 29
3.6 collectDividends() . . . 29
3.7 computeIncome() . . . 29
3.8 activateNewWorkContract() . . . 30

4 Bank **31**
4.1 resetting . . . 31
4.2 updateMonetaryVariables . . . 31
4.3 updateBankBalanceSheet . . . 32
4.4 determineCreditSupply . . . 32
4.5 collectApplicationsMakeOffers . . . 33
4.6 payoutCredits . . . 34
4.7 executeTransactions1 . . . 34
4.8 executeTransactions2 . . . 35
4.9 collectInterests . . . 35
4.10 payoutInterests . . . 36
4.11 profitAccounting . . . 36
4.12 executePendingTransactions() . . . 37
4.13 creditDefaultProcedure() . . . 37
4.14 BalanceSheetBank.java . . . 38

5 Central Bank **39**
5.1 profitAccounting() . . . 39
5.2 updateCBBalanceSheet() . . . 39
5.3 updateBalanceSheet() . . . 39
5.4 Credit.java . . . 40
5.5 CreditOffer.java . . . 40
5.6 CreditApplication.java . . . 40
5.7 BalanceSheetCentralBank.java . . . 40

6 Statistic Agency **40**
6.1 computeFirmStatistics() . . . 40

7 Clearing House **41**

8 Sector **42**
8.1 laborMarketMatching() . . . 42
8.2 cleanUp() . . . 42
8.3 clearOrderBooks1() . . . 42
8.4 allocateOrders1() . . . 43
8.5 computeStatistics() . . . 43
8.6 ServiceOrder.java . . . 44
8.7 ServiceInventory.java . . . 44
8.8 GoodsSector.java . . . 44
8.9 GoodsOrder.java . . . 44
8.10 GoodsInventory.java . . . 44
8.11 EnergySector.java . . . 44
8.12 EnergyOrder.java . . . 44
8.13 EnergyContract.java . . . 44
8.14 Inventory.java . . . 44

9 Mall **44**
9.1 GoodsMall.java . . . 44
9.2 ServiceMall.java . . . 44
9.3 EnergyMall.java . . . 44

10 Labor Market **44**
10.1 JobSearch.java . . . 44
10.2 WorkContract.java . . . 44
10.3 VacancyComparator.java . . . 44
10.4 Vacancy.java . . . 45
10.5 TechnicalCoefficient.java . . . 45
10.6 Region.java . . . 45

11 Rest of World **45**
11.1 exportRequestsGoods() . . . 45
11.2 receiveExports() . . . 46
11.3 adjustImports() . . . 46
11.4 deliverImports() . . . 47
11.5 PortfolioItem.java . . . 48
11.6 Plant.java . . . 48
11.7 PaymentTransaction.java . . . 48
11.8 PaymentAccount.java . . . 48
11.9 Order.java . . . 48
11.10 ImportGoodsFirm.java . . . 48
11.11 ImportFirmService.java . . . 48
11.12 ImportFirmEnergy.java . . . 48
11.13 ImportFirm.java . . . 48

11.14ForeignBank.java 48
11.15BalanceSheet.java 48

12 Calculations **48**

13 ConsumptionCoefficient.java **49**
13.1 ConsumptionCoefficientService.java 49
13.2 ConsumptionCoefficientGoods.java 49
13.3 ConsumptionCoefficientEnergy.java 49

14 Apeendix **49**
14.1 File lists 49
14.2 Q&A 54
14.3 Attributes 54

1 Model

This model simulates an economic system with various agents, including firms, banks, households, and a clearing house. Here's a breakdown of the main components and functionalities of the model code: Model.java.

1.1 Continuous space and grid initialization

The model sets up a continuous space and a grid for agent interactions. The space is defined as a torus (wrap-around borders) with a specified number of positions.
ContinuousSpace <Object> space = spaceFactory.createContinuousSpace(...) creates a continuous space object using a continuous space factory (*spaceFactory*). The *createContinuousSpace(...)* method is called on the factory to instantiate the space. This method typically takes several parameters to configure the space:
- *space* is the name given to the continuous space, a string identifier that helps to distinguish this space from others within the simulation context.
- *context* is the container for all agents, objects, and spatial environments.
- *new RandomCartesianAdder<Object>()* specifies the method used for adding objects to the space. A *RandomCartesianAdder* randomly places objects within the space.
- *new repast.simphony.space.continuous.WrapAroundBorders()* defines the boundary conditions of the continuous space. A torus (wrap-around) means that objects that move off one edge of the space reappear on the opposite edge.
- *numPositions* determines the size of the continuous space along one dimension, indicating that the space will have *numPositions* positions along that dimension.
- *1* indicates that the continuous space is one-dimensional.
Grid<Object> grid = gridFactory.createGrid("grid", context, ...) creates a grid object using the grid factory (*gridFactory*). The *createGrid(...)* method is called on the factory to instantiate the grid. This method typically takes several parameters to configure the grid.
- *new GridBuilderParameters<Object>(...)* specifies the parameters for building the grid.
- *new WrapAroundBorders()* defines the boundary conditions of the grid as a torus.
- *new SimpleGridAdder<Object>()* adds objects to the grid without any specific pattern/rule.
- *true* indicates that the grid has wrap-around borders.
- *numPositions* determines the size of the grid along one dimension.
- *1* specifies the grid is one-dimensional.

1.2 Demographic setup

This section defines parameters for the number of sectors, goods sectors, service sectors, energy sectors, households, firms per sector, banks, and more.
- *int numSectors = 12* represents the total number of sectors in the model.
- *int numGoodsSectors = 8* represents the number of goods sectors.
- *int serviceSectors = 3* represents the number of service sectors.
- *int numEnergySectors = 1* represents the number of energy sectors.
- *int numHouseholds = 10000* represents the total number of households in the model.
- *int numFirmsPerSector = 10* represents the number of firms per sector.

- *int numBanks = 2* represents the total number of banks in the model.
- *double employmentShare = 1.0* represents the share of employment in the total population.
- *double totalSizeLaborForce = employmentShare*numHouseholds;* calculates the total size of the labor force based on the employment share and the total number of households.
- *double totalLaborIncome = 0* represents the total labor income in the model.
- *double totalLaborForceData = 0* represents the total labor force data in the model. - *int ID = 0* is used as an identifier for agents and entities in the model.
- *ArrayList<Sector> tempSectorList = new ArrayList<Sector>()* is used to temporarily store sectors during initialization.
- *ArrayList<Bank> tempBankList = new ArrayList<Bank>()* is used to temporarily store banks during initialization.
- *ArrayList<ConsumptionCoefficient> consumptionCoefficientList = new ArrayList <ConsumptionCoefficient>()* is used to store consumption coefficients for different sectors.

1.3 Agent Initialization

The central bank, foreign bank, and local banks are initialized, each with associated attributes and functionalities. Clearinghouse is created for financial operations. Sectors (goods, energy, and service) are created along with malls and firms within each sector.
1. Central Bank Initialization
- *CentralBank aCentralBank = new CentralBank(ID)* creates an instance of the *CentralBank* class with the given ID.
- *int centralBankID = ID* stores the ID of the central bank for future reference.
-*context.add(aCentralBank)* adds the central bank to the simulation context.
-*ID++*: increments the ID counter for the next entity.
2. Foreign Bank Initialization
-*ForeignBank aForeignBank = new ForeignBank(ID)* creates an instance of the *ForeignBank* class with the given ID.
-*context.add(aForeignBank)* adds the foreign bank to the simulation context.
3. Local Banks Initialization
Iterates over a loop to create and add local banks to the simulation context.For each iteration
-*Bank aBank = new Bank(ID, centralBankID, aForeignBank.ID)* creates an instance of the *Bank* class with the given ID, central bank ID, and foreign bank ID.
-*context.add(aBank)* adds the local bank to the simulation context.
-*tempBankList.add(aBank)* adds the local bank to a temporary list for future reference.
-*aCentralBank.bankReserveList.add(aBank.centralBankReserves)* and *aCentralBank.bankStandingFacilitiesList.add(aBank.standingFacility)* add the bank's reserve and standing facility to the central bank's lists.
4. Clearing House Initialization
Assigning a bank randomly from the list of local banks (*tempBankList*) to serve as the central clearing house and adding it to the simulation context:
-*int bankIndex = (int)(Math.random() * ((numBanks-1) + 1))* generates a random index within the range of available local banks in the *tempBankList*.
-*int bankID = tempBankList.get(bankIndex).ID* retrieves the ID of the randomly selected bank from the *tempBankList*.

-*ClearingHouse aClearingHouse = new ClearingHouse(ID, bankID)* creates an instance of the -*ClearingHouse* class with the given ID and the ID of the bank chosen to serve as the clearing house.
-*context.add(aClearingHouse)* adds the clearing house to the simulation context.
-*tempBankList.get(bankIndex).paymentAccountList.add(aClearingHouse.paymentAccount)* adds the payment account associated with the clearing house to the payment account list of the randomly selected bank.
-*tempBankList.get(bankIndex).centralBankReserves.accountBalance += aClearingHouse.paymentAccount.accountBalance* updates the central bank reserves of the randomly selected bank by adding the balance of the clearing house's payment account.
5. Sector Initialization
The loop iterates over each sector from 1 to *numSectors*. This loop initializes all the sectors required for the simulation.
- Sectors 1 to 7 and sector 9 are identified as industrial sectors
- Sector 8 is designated as the energy sector.
- Sectors 10 to 12 are classified as service sectors.
- For each sector type, the corresponding sector object is instantiated (*GoodsSector*, *EnergySector*, or *ServiceSector*).
- Properties such as space, grid, sector ID, total size of labor force, and number of firms per sector are passed to the sector constructor.
- Total labor costs and employment data are updated based on the characteristics of the newly created sector.
6. Mall and Firm Initialization
- A mall (*GoodsMall*, *EnergyMall*, or *ServiceMall*) is added to the sector's mall list (*mallList*). Each sector has one mall.
- Firms (*GoodsFirm*, *EnergyFirm*, or *ServiceFirm*) are created and added to the sector.
- The number of firms per sector is determined by *numFirmsPerSector*.
- Each firm is associated with a bank (*tempBankID*) and the clearing house (*aClearingHouse.ID*). - Inventories (*GoodsInventory*, *EnergyInventory*, or *ServiceInventory*) are created for each firm and added to the corresponding mall's inventory list.
- The initial output of each firm is added to the sector's initial output.
7. Household initialization
This process ensures that households are created, assigned to sectors based on labor force distribution, associated with banks, and initialized with relevant attributes for consumption modeling within the simulation.
- Loop through Households. Iterates over each household, from 0 to *numHouseholds*.
- Assign Workers to Sectoral Labor Force

- Workers are assigned to labor forces based on the distribution of labor forces in the sectors.
- *cumLabForce* keeps track of the cumulative labor force as households are assigned to sectors.
- The loop iterates through the sectors (*tempSectorList*) until the cumulative labor force surpasses the labor force of the current sector. Then, it moves to the next sector.

- Assign Bank. Each household is assigned a bank randomly selected from the available banks (*tempBankList*).The household's payment account is added to the selected bank's payment account list, and its account balance is updated accordingly.
- Reading in the consumption coefficients of households from "consumption.csv". This ensures that the simulation has access to consumption coefficients for each sector(e.g. how much of the consumption budget is spent on this product category)
- Creating Consumption Coefficient Objects and added to the *consumptionCoefficientList*.
- Set Household Attributes

- The household's *consumptionCoefficientList* is set to the list of consumption coefficients obtained earlier.
- The household's *disposableIncome* is calculated based on the total labor income divided by the total labor force data.
- The *propensityToConsume* attribute of the household is set. This line of code likely sets a specific value, it should be made more generic.
- The household's *currentWorkingSector* is set to the sector to which it's assigned.

1.4 ROW setup

- An instance of the Rest of World entity (*ROW*) is created with a unique identifier (*ID*) and associated with the Foreign Bank (*aForeignBank*).
- The *nominalExportVolumeTarget* attribute of the RoW is set based on a calculation involving total labor income, total labor force data, and the number of households.
- The RoW's payment account is added to the payment account list of the Foreign Bank (*aForeignBank*), likely for handling transactions involving exports and imports.
- An instance of the Statistic Agency is created with a unique identifier (*ID*), possibly for tracking and analyzing simulation data.
- The *totalOutputID* and *numHouseholds* are passed to the Statistic Agency constructor.

2 Firms

The Firm class models a firm agent in an agent-based simulation. @ScheduledMethod annotation scheduled events summarized below:
1 Price setting → 2 Computes planned output → 3 Choose goods input supplier → 4 R&D planning; → 5 Choose service input supplier → 6 Choose energy supplier; → 7 Determine the input demand given the planned output; → 8 Financial planning; → 9 Apply for bank loans; → 10 Firm checks credit offers → 11 Check financial status after external financing; → 12 Firm pays interest and debt installments on its loans → 13 Debt restructuring → 14 Replanning production → 15 Labor market activities → 16 Firms enter the goods malls → 17 Firms enter the service malls → 18 Firms enter the enery malls → 19 Firm received input goods → 20 Firm received input service → 21 Firm received input energy → 22 if the firms has been rationed in the first round; choose new suppliers → 23 Order new inputs in case of rationing → 24 Firm receiveds second round of input deliveries → 25 Production of output → 26 Pay wages to all workers → 27 Adjust capital stock → 28 determines wage offer → 29 Carry out process innovations → 30 Product innovations → 31 Adjustment of specific skills of workers → 32 Adjust the different productivities → 33 Accounting.

The following part gives a detailed explanations methods used in Firm.java, GoodsFirm.java.

2.1 chooseGoodsSupplier1()

This method is responsible for selecting goods input suppliers for a given list of input goods requirements. *chooseServiceSupplier1()*and *chooseEnergySupplier1()* also have the samilar logics. The following bullets give an overview of the algorithm.
1. Initialization: Retrieve the current simulation context and then iterate over each input goods requirement in the list.
2. Supplier Selection: If the tick is equal to 1 or a random probability check (*probChangeSupplierGoods*) succeeds, indicating the need for a supplier change or selection:
- Initialize a list (*potentialSupplierList*) to store potential suppliers.
- Calculate the denominator for the logit model (*denLogit*) by summing the exponential of the product of price intensity and quality intensity for each potential supplier.
- Iterate over all goods sectors in the context to find potential suppliers.
- Calculate the probability of choosing each potential supplier based on their price and quality, using the logit model.
- Select a supplier based on the calculated probabilities and update the current supplier list and input goods requirement with the chosen supplier.
3. Probability Check: Determines whether to change suppliers based on a predefined probability (*probChangeSupplierGoods*). Uses a logit model to probabilistically select suppliers based on their prices and qualities.
4. Current Supplier List: Maintains current suppliers for each input goods requirement.
5. Potential Supplier List: Stores potential suppliers for each input goods requirement.

2.2 rdPlanning()

This method is responsible for planning research and development (R&D) expenditures based on a percentage of revenues, which is referred to as the R&D intensity.
- Calculation of Planned R&D Expenditure: The planned R&D expenditure is calculated by multiplying the revenue generated by the entity with the R&D intensity percentage.
- Calculation of Planned R&D Output: The planned R&D output is then calculated based on the expenditure and the unit costs. If the unit costs are greater than zero, the planned R&D output is computed as the planned R&D expenditure divided by the unit costs. This ensures that R&D expenditure translates into a tangible output in terms of units produced or developed.
- Integration with Overall Planned Output: The planned R&D output is added to the overall planned output. This implies that the R&D output contributes to the entity's total production or development goals.
By setting planned R&D expenditures based on revenue the model can simulate how real companies tend to be more in to invest in R&D when they are doing well financially and to cut R&D spending when finances are tighter. The rdintensity parameter provides control the aggressiveness of R&D investments in all companies.

2.3 financialPlanning()

This method computes various financial metrics required for the firm's financial planning, including production costs, financial obligations, and external financing needs. This method ensures that the firm maintains sufficient financial resources to meet its operational and financial commitments.

- *totalFinancialNeedsProduction* computes the total financial needs for production, including expected labor costs, investment expenditures, intermediate costs, energy costs, and service costs.
- *internalFinancialResources* calculates the internal financial resources available, typically stored in the firm's payment account.
- *externalFinancialObtained* initializes the external financial resources obtained, initially set to 0.
- *plannedDebtInstallmentPayments* computes the planned debt installment payments by summing up the installment payments of all existing credits.
- *plannedInterestPayments* calculates the planned interest payments by summing up the interest payments of all existing credits.
- *totalfinancialNeedsFinancialObligations* calculates the total financial needs for financial obligations, including tax payments, debt installment payments, and interest payments.
- *totalFinancialNeeds* computes the total financial needs by summing up the financial needs for production and financial obligations.

- *externalFinancialNeeds* determines the external financial needs by subtracting the internal financial resources from the total financial needs. If the result is negative, it sets external financial needs to 0, indicating that there's no deficit.

2.4 creditApplication()

This method simulates the firm's process of applying for credit to cover its external financing needs. In this implementation, the firm applies for credit only at its home bank.

- *creditOfferList.clear()* clears any existing credit offer list.
- *if(externalFinancialNeeds>0)* checks if external financing is needed. If the firm requires external financing (i.e., if *externalFinancialNeeds* is greater than 0), it proceeds with the credit application process.
- *Iterable<Bank> Banks = context.getObjects(Bank.class)* retrieves all bank objects from the simulation context.
  - Checks if the bank's ID matches the firm's home bank ID (i.e., the bank where the firm holds its payment account).
  - If a match is found, creates a credit application at the home bank by adding it to the bank's *creditApplicationList*. The credit application includes the firm's ID, the bank's ID, the amount of external financial needs, and information from the firm's balance sheet (equity and total debt).

2.5 chooseCreditOffers()

This method simulates the firm's process of choosing credit offers from various banks and accepting the most suitable offer based on the requested credit amount and interest rates.

- *CreditOffer bestOffer = null* initializes a variable to hold the best credit offer.
- *CreditOffer rationedOffer = null* initializes a variable to hold the credit offer that accommodates partial financing.
- *CreditOffer acceptedOffer = nul* initializes a variable to hold the accepted credit offer.
- Checks if there are any credit offers in the *creditOfferList*.
- Iterates through each credit offer in the list:
  - If the offered credit is almost equal to the requested credit (within a small tolerance), selects the offer with the lowest interest rate as the best offer.
  - If the offered credit is less than the requested credit, selects the offer that offers the highest amount of credit (partial financing) as the rationed offer.
  - If there is a best offer (i.e., full financing is available), it is chosen as the accepted offer. Otherwise, the rationed offer is chosen.

- If an accepted offer is found:
  - Retrieves the simulation context.
  - Retrieves all bank objects from the context.
  - Iterates through each bank:
    * Checks if the bank's ID matches the ID of the bank that provided the accepted offer.
    * If a match is found, adds the accepted offer to the bank's 'acceptedCreditApplications' list.

2.6 financialStatus()

This method assesses the firm's financial condition based on its available financial resources and obligations, determining whether it is financially sound, constrained, or facing a crisis.

- Calculates the total internal financial resources by adding the internal resources (payment account balance) with the external financial resources obtained from credit offers.
- Computes the total financial needs, which include financial obligations (such as taxes, debt installments, and interest payments) and production-related financial needs (such as labor costs, investment expenditures, etc.).
- Evaluates the firm's financial status based on the comparison between internal financial resources and total financial needs:
  - If the internal financial resources are greater than or equal to the total financial needs, the firm remains financially sound, and flags for financial crisis, bankruptcy, and financial constraints are set to false.
  - If the internal financial resources are less than the total financial needs but greater than the financial obligations, the firm is financially constrained, indicating that it can cover financial obligations but not the full production costs.
  - If the internal financial resources are insufficient to cover all financial obligations, the firm is in a financial crisis, and flags for financial crisis, bankruptcy, and financial constraints are set accordingly.

2.7 interestInstallmentPayments()

This method facilitates the timely payment of debt installments and interest on loans obtained by the firm, ensuring compliance with financial obligations and maintaining financial stability.

- Checks if the firm is not in a financial crisis. If it is, no payments are made.
- Initializes variables for debt installment payments and interest payments.

- Retrieves the context to access bank objects.
- Iterates through the list of credits obtained by the firm.
- For each credit, if it has just been granted (not yet utilized), marks it accordingly.
- For each utilized credit, finds the corresponding bank and updates the debt level by subtracting the installment payment.
- Deducts the installment payment and interest payment from the firm's internal financial resources.
- Adds payment transactions for debt installments and interest to the firm's pending outgoing transactions list.
- Adjusts the central bank reserves accordingly.

2.8 debtRestructuring()

This method handles the restructuring of debts when the firm faces a financial crisis, allowing temporary relief from debt obligations to prevent bankruptcy and facilitate recovery.

- Checks if the firm is in a financial crisis.
- Initializes the debt restructuring counter to the debt restructuring period defined.
- Resets debt installment payments and interest payments.
- Retrieves the context to access bank objects.
- Iterates through the list of credits obtained by the firm.
- For each credit:
  - If it has not been marked for restructuring before, marks it as being restructured.
  - If it has been restructured before, flags the firm as bankrupt and exits the loop.
- If the firm is not bankrupt, proceeds with debt restructuring:
  - Increases the debt level by adding the accrued interest payments to the current debt level.
  - Sets interest payments and installment payments to zero.
  - If the debt restructuring period expires, marks the credit as restructured and resets the restructuring flag.
  - Decrements the debt restructuring counter.
- If the firm is bankrupt, flags the credits as defaulted and initiates the credit default procedure with the respective banks.

2.9 replanningProduction()

This method dynamically adjusts the production planning based on the financial condition of the firm, ensuring that the planned output remains within the firm's financial capabilities during periods of financial constraint or crisis.

- Checks if the firm is financially constrained or in a financial crisis.
- Cancels all R&D plans by setting the planned R&D output to 0.
- Initiates an iterative process to adjust the planned output until available resources are sufficient to pay all obligations and the adjusted output.
- Computes the initial financial needs for production.
- Iteratively decrements the planned output by a certain percentage (*PARoutputAdjustment*) each time, and recomputes the input demand for the production plan.
- Checks if the updated financial needs for production exceed the internal financial resources and if the planned output is above a threshold (5% of the initial planned output). If so, continues the adjustment process.

2.10 inputGoodsOrderRound1()

This method facilitates the process of firms obtaining the inputs needed for production by placing orders with the goods malls, ensuring that the necessary goods are available for production. The *inputServiceOrderRound1()* and *inputEnergyOrderRound1()* follow the same logics.

- Retrieves the context to access other objects in the simulation.
- Iterates over the list of input goods requirements.
- Retrieves all sectors in the context.
- Matches the sector ID of each input goods requirement with the sector ID of the goods sector.
- Iterates over the firm inventories in the mall associated with the goods sector.
- Checks if the supplier firm ID of the input goods requirement matches the firm ID of any inventory in the mall.
- If a match is found, creates a goods order for the input goods requirement, indicating the total demand and setting the order status to *true*.
- Places the order in the order book of the corresponding firm inventory in the mall.

2.11 inputGoodsDeliveryRound1()

This method handles the reception and allocation of input goods for production processes, ensuring that the firm's production activities are properly supported with the necessary inputs. The *inputServicesDeliveryRound1()* and *inputEnergyDeliveryRound1()* follow the same logics.

- Iterates over the list of input goods requirements.
- Sets the received quantity of each input goods requirement based on the accepted quantity from the order.
- Records the transaction of purchasing input goods in the payment account with appropriate details such as IDs and transaction type (*salesRevenue*).
- Checks if the received quantity is less than 95% of the total demand and marks the input goods requirement as rationed if so.
- Adjusts the allocation of received input goods based on whether they are used directly in production or for investment purposes.
- Updates various financial and production-related metrics such as intermediate goods stock, intermediate input expenditures, capital investment, and quality capital investments.

2.12 chooseGoodsSupplier2()

This method ensures that if the firm has been rationed in the first round for certain input goods, it tries to choose new suppliers for those goods in the subsequent round to fulfill its requirements. The *chooseEnergySupplier2()* and *chooseEnergySupplier2()* follows the same logics.

- Iterates over the list of input goods requirements.
- Clears the list of potential suppliers for each input goods requirement.
- If an input goods requirement has been rationed in the first round, it proceeds to choose a new supplier for that requirement.
- Calculates the denominator for the logit model, which is the sum of exponentials of the product of intensity factors and quality attributes of potential suppliers.
- Iterates over all sectors to find potential suppliers that have inventory stock available.
- Calculates the probability of choosing each potential supplier based on the logit model.
- Randomly selects a potential supplier based on the computed probabilities and updates the current supplier list and the supplier of the input goods requirement accordingly.

2.13 inputGoodsOrderRound2()

This method ensures that if the firm has been rationed for certain input goods in the first round, it tries to order new inputs for those goods in the subsequent round to fulfill its requirements. Additionally, it mentions a TODO comment indicating that the financing of the second round needs to be checked as well, suggesting a consideration for future implementation or refinement. The *inputServiceOrderRound2()* and *inputEnergyOrderRound2()* follows the same logics.

- Iterates over the list of input goods requirements.
- Checks if an input goods requirement has been rationed in the previous round.
- If an input goods requirement has been rationed, it proceeds to order new inputs for that requirement in the second round.
- Iterates over all sectors to find the sector corresponding to the input goods requirement.
- Creates a new order for the supplier of the rationed input goods requirement with the total demand quantity and marks it as true.
- Adds the new order to the order book of the supplier.

2.14 inputGoodsDeliveryRound2()

This method ensures that if the firm has been rationed for certain input goods in the first round, it processes the second round of input deliveries and updates relevant financial and inventory information accordingly. The *inputServiceDeliveryRound2()* and *inputEnergyDeliveryRound2()* follow the same logics.

- Iterates over the list of input goods requirements.
- Checks if an input goods requirement has been rationed in the previous round.
- If an input goods requirement has been rationed, it proceeds to receive the second round of input deliveries for that requirement.
- Retrieves information about the received quantity and the corresponding order.
- Records the received quantity of input goods and updates the intermediate goods stock accordingly.
- Records the expenditure for the received input goods and updates the capital investment if applicable.
- Calculates the value and price of intermediate goods based on the received quantities and expenditures.

2.15 payWages()

This method ensures that all workers are paid their wages according to their respective contracts, contributing to the overall management of the firm's finances and human resources.

- It initializes the variable 'laborCosts' to zero. This variable likely keeps track of the total labor costs incurred by the firm during each execution of the method.
- It iterates over the list of 'workContractList', which presumably contains all active work contracts for the firm's employees.
- For each work contract in the list:
  - It retrieves the details of the contract, such as the worker's ID, wage, and associated bank account.
  - It initiates a payment transaction to transfer the worker's wage from the firm's payment account to the worker's bank account. This is achieved by creating a new *PaymentTransaction* object and adding it to the list of pending outgoing transactions (*pendingOutgoingTransactionsList*) in the firm's payment account.
  - It adds the worker's wage to the total labor costs (*laborCosts*), incrementally accumulating the total wage expenses incurred by the firm.

2.16 processInnovation()

This method simulates the occurrence of process innovations within the firm, considering factors such as sector knowledge stock, the balance between product and process innovations, and the efficiency of the innovation process.

- It begins by generating a random number ('random') between 0 and 1, which is used to determine whether a process innovation will occur.
- The probability of a process innovation ('rdProbabilityProcess') is calculated based on various factors:
  - *ownSector.avgKnowledgeStock*: Average knowledge stock in the firm's sector.
  - *shareProductInnovation*: Share of product innovations in the firm's overall innovation efforts.
  - *rdKnowledgeStock*: Knowledge stock related to research and development activities.
  - *processInnoEfficiency*: Efficiency factor influencing the probability of process innovation.
- If the random number falls below the calculated probability (*rdProbabilityProcess*), a process innovation is initiated.
- Inside the innovation process, a random draw from a Gaussian distribution (*randomInnovationDraw*) is performed, representing the magnitude or impact of the innovation.

- If the drawn value is positive (indicating a positive impact), the cumulative count of process innovations (*cumulatedProcessInnovations*) is incremented by this value.

2.17 productInnovation()

This method simulates the occurrence of product innovations within the firm, affecting both product quality and productivity. The probability and impact of these innovations are influenced by various factors, including sector knowledge stock and the efficiency of the innovation process.

- It starts by generating a random number (*random*) between 0 and 1, which determines whether a product innovation will occur.
- The probability of a product innovation (*rdProbabilityProduct*) is calculated based on factors similar to those in the *processInnovation()* method, such as the firm's sector's average knowledge stock, the balance between product and process innovations, and the efficiency of the innovation process.
- If the random number falls below the calculated probability (*rdProbabilityProduct*), a product innovation is initiated.
- Inside the innovation process, a random draw from a Gaussian distribution (*randomInnovationDraw*) is performed to determine the magnitude or impact of the innovation.
- If the drawn value is positive (indicating a positive impact), the quality of the firm's products is increased by this value (*quality = quality + randomInnovationDraw*). Additionally, a portion of the innovation's impact is allocated to productivity improvement (*productivityEffectProductInnovation*), which is further subjected to an adjustment factor (*productivityEffectProductInnovationAdjustment*).
- If the random number exceeds the probability threshold, the productivity improvement from the previous round of product innovation is decreased (*productivityEffectProductInnovation = (1 - productivityEffectProductInnovationAdjustment) * productivityEffectProductInnovation*).

2.18 specificSkillUpdate()

This method simulates the continuous improvement or adjustment of workers' specific skills based on technological advancements. It ensures that workers' skills align with the technological requirements of the firm, thereby enhancing overall labor productivity.

- At each execution, the method updates the labor productivity of workers based on specific skill adjustments.
- The adjustment factor ('specificSkillAdjustment') is multiplied by the maximum of 0 and the difference between the technology level and the current labor productivity. This ensures that the adjustment is positive and proportional to the difference between the technology level and the current labor productivity.

- The result of this computation is added to the labor productivity, effectively updating it.

2.19 productivityAdjustment()

This method simulates the adjustment of various productivities within the firm, taking into account technological advancements, labor productivity, and the effect of product innovation on productivity. It also models the depreciation of research and development knowledge over time.

- It calculates the technology level based on the capital productivity and the cumulative process innovations (*cumulatedProcessInnovations*). The capital productivity is multiplied by (*1 + cumulatedProcessInnovations*) to incorporate the effect of process innovations on technology.
- The method computes the total factor productivity (*totalFactorProductivity*) as the minimum of the technology level and the labor productivity adjusted by the productivity effect of product innovation (*productivityEffectProductInnovation*). This ensures that total factor productivity is limited by the minimum of technology and the labor productivity adjusted for product innovation.
- Lastly, the method updates the research and development (R&D) knowledge stock by applying knowledge depreciation (*knowledgeDepreciation*).

2.20 updateBalanceSheet()

function updates the balance sheet of the firm. Here's what it does:

- It assigns the current values of various assets to the balance sheet.
- *capitalStockValue*: Represents the value of the capital stock owned by the firm.
- *cash*: Represents the amount of cash available in the firm's payment account.
- *financialCapitalValue*: Represents the value of financial capital assets (not updated in this function).
- *inventoryFinalProductValue*: Represents the value of final products held in inventory.
- *inventoryIntermediateProductsValue*: Represents the value of intermediate products held in inventory, including services (*serviceLevelValue*) and energy (*energyLevelValue*).
- After assigning asset values, it computes the equity of the firm using the *computeEquity()* method of the balance sheet.

The *chooseServiceProvider()* and *chooseEnergyProvider()* functions appear to be placeholder functions that return *null*. These functions are likely placeholders for choosing service providers and energy providers, respectively, based on some criteria from the provided inventories (*firmInventories*). However, they do not currently contain any logic and simply

return *null*. Depending on your requirements, you may need to implement logic within these functions to choose appropriate service and energy providers based on certain criteria.

2.21 InputItem

This class seems to encapsulate the information related to an input item needed by a firm or process, including its demand, supplier information, and order details. However, the definition of the 'Inventory' and 'Order' classes, as well as their functionalities, are not provided in the code snippet.

- *sectorID*: An integer representing the sector ID to which this input item belongs.
- *type*: An integer representing the type of input item.
- *rationed*: A boolean indicating whether this input item has been rationed or not.
- *demandDirect*: A double representing the direct demand for this input item.
- *demandCapital*: A double representing the demand for this input item for capital purposes.
- *totalDemand*: A double representing the total demand for this input item.
- *receivedQuantity*: A double representing the quantity of this input item received.
- *supplier*: An object of type 'Inventory' representing the supplier from which this input item is obtained.
- *order*: An object of type 'Order' representing the order for this input item.
- The constructor initializes the *sectorID*, *type*, *rationed*, *demandDirect*, *demandCapital*, *totalDemand*, and *receivedQuantity* attributes with the provided values. It sets *rationed* to *false* initially as there is no rationing.

2.22 pricing

- **Output Rationing Check**:
  - Calculate the difference between planned output (plannedOutput) and actual output (output), storing it in deltaOutput.
  - If the delta output is greater than 5% of the planned output, set the outputRationed flag to true, indicating output rationing.
- **Markup Adjustment**:
  - If the tick is greater than 1:
    - If the target markup (targetMarkUp) is not set and the current markup (markUp) is positive, set the target markup to the current markup.

    - Adjust the target markup based on the change in market shares (diffMarketShares) and the output rationing status.
    - Ensure that the target markup does not fall below 1.0.

- **Price Calculation**:
  - Initialize the price to zero.
  - Iterate over each inventory item in the mall and calculate its price based on average unit costs (avgUnitCosts) and the target markup.
  - Calculate the total price and the sum of sales quantities.
  - If there are sales, compute the average price. Otherwise, set the price to 1.0.

2.23 computePlannedOutput

- **Demand Calculation**: For each goods inventory item in the mall (mallInventoryList), calculate the expected demand based on the primary demand. If the tick is less than 2, set the expected demand and refill level to initial values. Otherwise, if there is excess demand and output rationing is not enabled, add a fraction (5%) of the excess demand to the expected demand.
- **Refill Level Calculation**: Compute the refill level for each inventory item as the expected demand multiplied by a factor (inventoryBufferPCT) representing the inventory buffer percentage.
- **Planned Delivery Calculation**:Determine the planned delivery for each inventory item by subtracting the current inventory level from the refill level. Replace negative values with zero.
- **Output Smoothing**:
  - Accumulate the planned deliveries for all inventory items to calculate the planned output before smoothing.
  - Calculate the planned output production by applying output smoothing (a smoothing factor) to the planned output before smoothing.
- **Final Adjustment**:
  - Update the planned output by adding the planned output production.
  - Adjust the planned delivery for each inventory item proportionally based on the ratio of the total planned output before smoothing to the total planned output.

2.24 computeInputDemandForProductionPlan

- **Capital Demand**: Calculate the capital required for production based on the planned output and sector-specific capital coefficient. If the current capital stock is less than the required amount, determine the additional capital demand.

- **Intermediate Input Demand**:Compute the demand for intermediate goods based on the planned output, sector-specific technical coefficients, and current inventory levels.
- **Service Demand**: Determine the demand for services using similar calculations as for intermediate goods.
- **Energy Demand**: Calculate the demand for energy based on planned output, sector-specific energy coefficients, and current energy levels.
- **Labor Demand**: Compute the demand for labor considering planned output, total factor productivity, and labor coefficient. Determine redundancies and vacancies accordingly.
- **Supplier Selection**:Choose an energy provider based on a probabilistic selection process considering supplier prices and intensity factors.

2.25 labormarketActivities

- Check if there are any redundancies in the labor market (‘redundancies‘ variable).
- If there are redundancies:
  - Set ‘unmatchedVacancies‘ to 0.
  - Shuffle the list of work contracts (‘workContractList‘).
  - Iterate:
    - Cancel the first work contract.
    - Remove the first work contract from the list.
    - Decrement the ‘redundancies‘ count.
  - Continue iterating while there are still redundancies and at least two work contracts remaining in the list.
- Check if there are any vacancies in the labor market (‘vacancies‘ variable).
- If there are vacancies:
  - Set ‘unmatchedVacancies‘ to the number of vacancies.
  - Iterate over the range of vacancies:
    - Create a new vacancy (‘Vacancy‘ object) with the current ID, wage, and reference to the current labor market.
    - Add the new vacancy to the ‘vacancyList‘ of the labor market’s own sector.

2.26 labormarketActivities2

- `laborStock = 0;`: Initializes the `laborStock` variable to 0. This variable likely represents the total number of active workers in the labor market.
- `for(int i =0; i < workContractList.size(); i++) { ... }`: Iterates over the list of work contracts in the `workContractList`.
  - `if(workContractList.get(i).canceled) { ... }`: Checks if the current work contract is canceled. If so, it executes the following block of code:
    - `workContractList.remove(i);`: Removes the canceled work contract from the list.
    - `i-`: Decrements the loop counter to account for the removed element and avoid skipping the next work contract.
  - `else { ... }`: If the work contract is not canceled, it increments the `laborStock` variable to count it as an active worker.

This method essentially updates the `laborStock` by iterating over the list of work contracts, removing any canceled contracts, and counting the remaining active workers. It ensures that the `laborStock` reflects the current state of the labor market by accounting for canceled work contracts.

2.27 production

- **Calculation of Potential Output:** Calculates the potential output based on total factor productivity and the minimum of labor, intermediate, capital, service, and energy inputs.
- **Output Calculation for Different Inputs:** Calculates the output for each input (intermediate, labor, capital, service, energy) based on total factor productivity and respective coefficients.
- **Adjustment of Potential Output:** Adjusts the the Potential Output based on intermediate input/service input/energy input/internal inventory stock.
- **Output Determination:** Output is the determined by the minimum of based on the planned output and the potential output.

2.28 capitalStockAdjustment

- **Depreciation**:
  - If there are real capital investments, adjust the quality of capital investments.
  - Calculate the amount of depreciated capital based on the depreciation rate and the output produced.
  - Update the capital stock by subtracting the depreciated capital.

- **Capital Productivity**: Update the capital productivity considering the existing capital stock, quality of capital investments, and output produced.
- **Capital Stock Update**: Add the real capital investments to the capital stock.
- **Capital Stock Value and Costs**: Compute the value of the capital stock and the associated costs based on the calculated capital price and depreciated capital.

2.29 outputDelivery

- **Output Gap Calculation**: Calculate the output gap as the difference between actual output and planned output.
- **Output Allocation**:
  - If the output gap is negative, allocate output to the market and research and development (RD) based on planned output production.
  - If the output gap is non-negative, allocate output to RD based on planned RD output, and the remaining to the market.
- **Update Knowledge Stock and Inventory**:
  - Update the RD knowledge stock with output allocated to RD.
  - Adjust internal inventory and deliver goods to mall inventories based on planned delivery.
- **Sector-Specific Output Information**: Print sector-specific output information for the firm if the sector ID matches a specific value.

2.30 accounting

- **Revenue and Sales Calculation**: Initialize revenue and sales-related variables. Iterate through mall inventories to calculate revenue, sales to firms and households, total sold quantities, total market sales, and inventory value.
- **Market Share Calculation**: Calculate market share based on total sold quantities and total market sales.
- **Costs Computation**: Calculate intermediate input, energy, and service costs. Compute total costs and unit costs.
- **Research and Development Investment**: Determine RD investment based on unit costs and output allocated to RD.
- **Average Market Share and Unit Costs**: Update average market share and unit costs using smoothing parameters.
- **Earnings and Dividends**:

  - Compute earnings considering revenue, inventory value change, interest income, and total costs.
  - Update balance sheet and distribute dividends if equity allows.

- **Equity Update**:Update equity and cash balance in the balance sheet.
- **Service Provider Selection**: Choose a service provider based on price and intensity choice parameters.

2.31 Considerations

- Efficiency: Depending on the size of the simulation and the number of potential suppliers, the efficiency of this method may need optimization.
- Model Calibration: The parameters (probChangeSupplierGoods, intensityChoicePriceGoods, intensityChoiceQualityGoods) should be carefully calibrated to reflect realistic decision-making behavior within the simulation.
- Logging: Add logging statements to track supplier selection and changes for debugging and analysis purposes.
- Parameter Tuning: Experiment with different parameter values to observe their effects on supplier selection behavior and overall simulation outcomes.
- Heterogeneity: Consider introducing variability in supplier's characteristics to make supplier selection more realistic.Goods/Service/Energy firm Heterogeneity

3 Household

This class represents a household agent in the simulation, which interacts with various sectors, firms, and markets to allocate its budget, consume goods and services, and participate in the labor market.

3.1 setupIteration()

This method is responsible for preparing the household for each new iteration of the simulation. It initializes various attributes and data structures, resets consumption-related values, and clears payment transactions to ensure a clean start for the household in each iteration. At the start of each iteration:

- Retrieves the current tick count from the simulation environment.
- If it's the first tick (tick count equals 1):
  - Adds the stocks of goods firms, service firms, and energy firms to the household's asset portfolio.
  - Inserts the household's asset portfolio into the appropriate clearing house.
- Resets demand and received quantities for each consumption coefficient.
- Resets the dividend income to zero.
- Clears pending outgoing and received payment transactions.

3.2 determineConsumptionBudget()

This method calculates the household's consumption budget based on its disposable income and a predetermined propensity to consume. It sets the consumptionBudget attribute accordingly.

- Calculates the consumption budget using the formula: consumptionBudget = disposableIncome * propensityToConsume.
- Updates the consumptionBudget attribute with the calculated value.

budgetAllocation()
This method allocates the consumption budget to each consumption coefficient in the household's list. It calculates the budget for each coefficient based on its coefficient value and the overall consumption budget.

- Iterates through each consumption coefficient in the consumptionCoefficientList.
- Calculates the budget for each coefficient using the formula: coefficientBudget = coefficient * consumptionBudget.
- Updates the budget attribute of each consumption coefficient with the calculated value.

3.3 shoppingActivity()

This method represents the shopping activity of the household. It iterates through each consumption coefficient in the household's list and selects suppliers for goods, services, and energy based on certain criteria.

- Retrieving Simulation Context: The method starts by retrieving the simulation context using *ContextUtils.getContext(this)*. This context provides access to various objects and information within the simulation environment.
- Iterating Through Consumption Coefficients: The method iterates through each consumption coefficient in the *consumptionCoefficientList*. These coefficients represent the household's preferences and demands for different types of goods, services, and energy.
- Selecting Suppliers: For each consumption coefficient, the method identifies the corresponding sector (Goods, Services, or Energy) to determine where to purchase the item. It then selects a supplier (firm) for the item based on certain conditions:
  - If it's the first tick of the simulation (*tick == 1*), or if there's a probability of changing suppliers (*probChangeSupplier*), or if the current supplier is out of stock.
  - It calculates a probability distribution based on supplier prices and quality, favoring suppliers with lower prices and higher quality.
  - It randomly selects a supplier based on the calculated probabilities.
- Placing Orders: Once a supplier is selected, the method places an order with the chosen supplier: For goods, it creates a goods order. For services, it creates a service order. For energy, it creates an energy order. These orders include information such as the household ID, supplier ID, quantity to purchase, and whether the order is urgent.
- Adding Orders to Supplier's Order Book: After creating the order, the method adds it to the selected supplier's order book. This ensures that the supplier processes the order and fulfills the household's demand.
- Repeat for Each Consumption Coefficient:The process repeats for each consumption coefficient in the list, ensuring that all types of goods, services, and energy needs are addressed.
- Simulation Execution: This method is scheduled to execute at regular intervals during the simulation (*interval = 1*). It ensures that the household's shopping behavior is simulated continuously throughout the simulation runtime.

By simulating the household's shopping activity in this way, the model can capture realistic consumer behavior, including factors such as price sensitivity, quality preference, and supplier availability. This contributes to a more comprehensive understanding of the dynamics within the simulated economy.

3.4 laborMarketActivities2()

This method plays a crucial role in simulating the labor market dynamics within the model, contributing to the realistic representation of household behaviors and employment outcomes.

- Initialization: The method starts by setting the ‘onTheJobSearch‘ flag to ‘false‘, indicating that the household is not actively searching for a job at the beginning of the simulation iteration.
- Work Contract Status Check: It checks if the household has a current work contract (‘workContract‘) and if it has been canceled. If the contract is canceled, it sets ‘workContract‘ to ‘null‘ and updates the ‘employed‘ status to ‘false‘, indicating that the household is currently unemployed.
- Determining Job Search Behavior for Unemployed Households:
  - If the household is unemployed (‘!employed‘), it evaluates whether it should start searching for a new job based on a random probability (‘probChangeSectorUnemployed‘).
  - It calculates a probability distribution for selecting a new sector to search for a job. This calculation considers the number of open positions in each sector and applies a logistic function with the intensity of choice determined by the parameter ‘intensityChoiceSectorUnemployed‘.
  - Based on the calculated probabilities, it selects a sector for job search.
- On-the-Job Search for Employed Households:
  - If the household is already employed, it checks if it should engage in on-the-job search based on a random probability (‘onTheJobSearchRate‘). This represents the probability of an employed individual actively seeking better job opportunities while still employed.
  - If the condition is met, it sets the ‘onTheJobSearch‘ flag to ‘true‘.
- Initiating Job Search:
  - If the household is either unemployed or engaging in on-the-job search, it initiates a new job search process.
  - It creates a ‘JobSearch‘ object with relevant parameters such as household ID, general skill level, reservation wage, and initial search effort.
  - The created ‘JobSearch‘ object is added to the job search list of the sector where the household is currently seeking employment.
- Simulation Execution:
  - This method is scheduled to execute at regular intervals during the simulation.

- It captures the dynamic behavior of households in the labor market, including transitions between employment and unemployment, decisions related to job search, and engagement in on-the-job search activities.
- The priority assigned to this method determines its execution order relative to other scheduled methods in the simulation environment.

3.5 consumption()

This method is responsible for managing household consumption activities. Within the method, a loop iterates over each item in the *consumptionCoefficientList*. For each item in the list:

- It updates the *receivedQuantity* attribute based on the quantity of the item that was accepted or acquired.
- It updates the *consumptionExpenditures* variable by adding the invoice amount of the item. This represents the total expenditures incurred by the household due to consumption.
- It adds a new *PaymentTransaction* to the *pendingOutgoingTransactionsList* of the household's *paymentAccount*. This transaction likely represents a payment made to the supplier firm for the consumed goods or services.
- The *PaymentTransaction* includes relevant information such as the household ID , the firm ID of the supplier, bank IDs, transaction amount, and a label indicating the nature of the transaction (e.g., *salesRevenue*).

3.6 collectDividends()

This method simulates the passive income earned by the household through dividend payments from its investment portfolio. Within the method, a loop iterates over each item in the *assetPortfolio*, which represents various financial assets held by the household. For each asset in the portfolio:

- It calculates the dividend income associated with the asset. This is typically determined by multiplying the number of shares of the asset (*assetPortfolio.get(i).numShares*) by the dividend per share (*assetPortfolio.get(i).stock.dividendPerShare*).
- The calculated dividend income for each asset is then added to the total *dividendIncome* variable. This variable likely accumulates the total dividends received by the household over the simulation period.

3.7 computeIncome()

This method simulates the process of calculating the total income of a household in the simulation, considering various sources such as wages, dividends, and interest payments.

- It retrieves all instances of *StatisticAgency* from the simulation context. These agencies likely provide information about average wages in different sectors or regions.
- It checks if the household has a valid *workContract*. If it does, the *laborIncome* is set to the wage specified in the work contract (*workContract.wage*).
- If the household does not have a work contract, it iterates over all *StatisticAgency* instances to calculate an average wage. The *laborIncome* is set to 60% of this average wage, indicating that the household earns a percentage of the average wage in the absence of a specific work contract.
- The *interestIncome* is obtained from the *paymentAccount.interestPayment*. This represents any interest earned on savings or investments held in the payment account. - Finally, the *disposableIncome* is computed by summing up the *laborIncome*, *dividendIncome*, and *interestIncome*.

3.8 activateNewWorkContract()

This method manages the process of activating a new work contract for a household in the simulation, updating its employment status and work contract details accordingly.

- The method begins by checking if the household already has an existing work contract.
- If an existing work contract is found, it sets the *canceled* attribute of the current work contract to *true*. This indicates that the old contract has been canceled.
- After canceling the old contract (if applicable), the method assigns the new work contract (*newWorkContract*) to the household's *workContract* attribute.
- It also sets the *employed* attribute to *true*, indicating that the household is now employed under the new work contract.

4 Bank

4.1 resetting

This method plays a crucial role in the simulation by ensuring that the bank starts each step with a clean and consistent state, ready to process new data and transactions. It helps maintain the accuracy and reliability of the simulation results by preventing the accumulation of outdated information and ensuring a fresh start for each iteration.

- Clearing Data Structures: The method clears specific data structures within the bank to ensure a clean slate for the next iteration of the simulation. This includes clearing the ‘creditApplicationList‘ and ‘acceptedCreditApplications‘ lists.
- Resetting Pending Transactions: It clears the pending outgoing and received transactions lists associated with the ‘centralBankReserves‘ payment account. This ensures that any pending transactions from the previous simulation step are handled appropriately, preventing them from carrying over to the next step.
- Maintaining Simulation Consistency: By resetting relevant attributes and data structures, the method helps maintain the consistency and integrity of the simulation. It ensures that the bank starts each simulation step with a fresh state, free from any residual data or transactions from previous steps.
- Preventing Data Accumulation: Clearing data structures prevents the accumulation of outdated or irrelevant data, which could potentially impact the accuracy and performance of the simulation over time. It allows the simulation to focus on current and relevant information for decision-making and analysis.
- Initialization for Next Step: The method prepares the bank for processing new information, transactions, and events in the subsequent simulation steps. It sets the stage for the bank to receive and handle new data effectively, without interference or confusion from previous iterations.

4.2 updateMonetaryVariables

This method is responsible for updating various monetary variables based on information obtained from the simulation context, particularly from the ‘CentralBank‘ objects.

- Accessing Central Bank Information: It retrieves all instances of ‘CentralBank‘ objects from the simulation context.
- Matching Bank ID: Within the loop iterating over central banks, the method finds the central bank that matches the ‘centralBankID‘ attribute of the current bank instance.
- Updating Interest Rates: Once the matching central bank is found, the method updates various interest rates and ratios based on the central bank’s parameters. These updates include:

  - Setting the ‘interestRate‘ and ‘interestRateMonthly‘ attributes of the ‘standingFacility‘ credit based on the central bank’s ‘interestBaseRate‘.
  - Calculating the ‘interestMarkDownReserves‘ and ‘interestRateOnDeposits‘ based on the central bank’s parameters and applying them to the bank’s reserves and deposits.
  - Updating the ‘interestRate‘ and ‘interestRateMonthly‘ attributes of the ‘centralBankReserves‘ payment account based on the central bank’s parameters.

- Updating Capital Ratios: The method also updates the ‘capitalAdequacyRatio‘ and ‘reserveRequirementRatio‘ attributes of the bank based on the corresponding parameters from the central bank.

- Ensuring Consistency: By updating these monetary variables based on central bank parameters, the method ensures that the bank’s financial operations and decisions align with the broader monetary policies set by the central bank. This helps maintain consistency and realism in the simulation by reflecting real-world monetary dynamics.

4.3 updateBankBalanceSheet

It hasn’t been set.

4.4 determineCreditSupply

This method enables the bank to evaluate its credit supply capacity based on regulatory constraints, existing credit risks, and available liquidity.

- Compute Risk Budget:
  - Risk Assessment Loop: The method iterates through the existing credits (creditList) to assess the risk associated with each credit using the riskAssessment method of the Credit class.
  - Calculate Risk-Weighted Assets: It calculates the total risk-weighted assets by summing up the exposures at risk of all existing credits.

- Calculate Risk Exposure Budget:
  - Compute Alpha: It calculates the coefficient alpha based on the Capital Adequacy Ratio (CAR). If the CAR is greater than 0, alpha is computed as the inverse of CAR; otherwise, it is set to 0 to avoid division by zero.
  - Compute Risk Budget: The method then computes the risk exposure budget using the formula alpha * balanceSheet.equity - riskWeightedAssets. This budget represents the bank’s available capital for new lending activities after accounting for existing credit risks.

- Calculate Excess Liquidity: It computes the excess liquidity by subtracting the reserves required by regulatory standards (reserveRequirementRatio) from the bank's cash reserves. Excess liquidity indicates the funds available for lending beyond regulatory requirements.

4.5 collectApplicationsMakeOffers

This method facilitates the assessment of credit applications, determines the total credit demand, and generates credit offers based on available liquidity and risk exposure budget constraints. It plays a pivotal role in the bank's decision-making process regarding credit issuance to firms.

- Total Credit Demand Calculation:
  - The method initializes a variable 'totalCreditDemand' to track the cumulative credit demand from all credit applications.
  - It also initializes temporary variables 'tempExcessLiquidity' and 'tempRiskExposureBudget' to store the current excess liquidity and risk exposure budget, respectively.
- Iterate Through Credit Applications:
  - The method iterates through each credit application in the 'creditApplicationList'.
  - For each application, it calculates the total credit demand by summing up the credit amounts requested by all applicants.
- Offer Generation:
  - Before making an offer, the method checks if there's enough liquidity ('tempExcessLiquidity') and risk exposure budget ('tempRiskExposureBudget') available.
  - If both liquidity and risk budget constraints are met, it proceeds to generate a credit offer for the applicant.
  - It selects a firm associated with the credit application and calculates the interest rate offered for the credit. The interest rate calculation involves factors such as the standing facility's base interest rate ('standingFacility.interestRate'), a risk premium ('lambdaB * creditApplicationList.get(i).riskOfDefault'), and a random component ('epsilon') for variation.
- Credit Offer Creation:
  - Upon determining the interest rate, the method creates a 'CreditOffer' object containing details such as the bank ID, firm ID, interest rate, credit amount (limited by available liquidity), and repayment period.
  - It reduces the temporary excess liquidity and risk exposure budget by the amount offered in the credit, ensuring that these constraints are updated for subsequent credit offers.

- Offer Limitations: It's important to note that this version of the method only allows firms to apply for credit from their home bank. Therefore, credit offers are generated only if the applying firm matches the bank's ID associated with the credit application.

4.6 payoutCredits

This method manages the payout process for accepted credit applications, ensures the proper recording of transactions, and updates the bank's financial position accordingly. It plays a crucial role in facilitating the flow of credit to firms within the simulation context while maintaining the bank's financial stability and profitability.

- Credit Payout Iteration: The method iterates through each accepted credit application in the 'acceptedCreditApplications' list. For each accepted credit application, it proceeds to initiate the payout process.
- Credit Payout Process:
  - For each accepted credit application, the method identifies the associated firm by matching its ID with the firms in the simulation context.
  - Once the firm is identified, a new 'Credit' object is created representing the credit provided to the firm. This credit includes details such as the firm ID, bank ID, credit amount, interest rate, installment period, and the firm's balance sheet.
  - The newly created credit is added to the firm's 'creditList', representing the credit obtained by the firm.
  - The external financial obligation of the firm is updated to reflect the initial debt level of the credit.
  - The excess liquidity and risk exposure budget of the bank are adjusted based on the credit payout. If the excess liquidity becomes negative, the bank may utilize its standing facility to cover the shortfall.
- Transaction Recording: The method records the transaction by adding a payment transaction to the bank's 'pendingOutgoingTransactionsList'. This transaction represents the payout made to the firm as credit. If the excess liquidity of the bank becomes negative due to the payout, the method adjusts the bank's balance by transferring funds from the standing facility to cover the shortfall.
- Profit and Loss Considerations: The method updates the bank's profit and loss statement by considering the interest received from the credited loans and any interest paid to the central bank or depositors.

4.7 executeTransactions1

This method ensures the timely execution of pending transactions, facilitates the transfer of funds between accounts, and maintains accurate records of financial transactions within the bank's ecosystem. The

4.8 executeTransactions2

has the same logics.

- Transaction Execution: The method iterates through the list of payment accounts managed by the bank. For each payment account, it examines the list of pending outgoing transactions. It identifies the recipient of each transaction based on the receiver bank ID and receiver ID specified in the transaction.
  - If the recipient is the bank itself or another bank within the simulation context, the method executes the transaction by transferring funds between accounts.
  - If the recipient is a foreign bank, it locates the corresponding foreign bank object within the simulation context and executes the transaction accordingly.
- Transaction Recording: Upon successful execution of a transaction, the method updates the account balances of both the sender and receiver. It also adds the transaction to the receiver's list of received transactions for record-keeping purposes.
- Central Bank Reserves Adjustment: For transactions involving the central bank reserves, the method adjusts the central bank's balance sheet accordingly.
  - If the transaction involves transferring funds to the central bank, it increases the central bank's cash reserves.
  - If the transaction involves receiving funds from the central bank, it decreases the central bank's cash reserves.
- Foreign Bank Transaction Handling: When executing transactions with foreign banks, the method locates the corresponding foreign bank object within the simulation context. It transfers funds between the bank's payment account and the payment account of the foreign bank.
- Handling of Pending Transactions: After executing each transaction, the method removes it from the list of pending outgoing transactions to avoid duplicate processing.

4.9 collectInterests

This method accurately computes the total interest income generated by the bank's credit activities. This information is crucial for assessing the bank's financial performance and profitability.

- Iterate Through Credit List: The method iterates through the list of credits maintained by the bank. For each credit, it retrieves the interest payment associated with it.
- Calculate Total Interest: As it iterates through the credit list, the method accumulates the interest payments received from each credit. This cumulative sum represents the total interest income generated by the bank from its outstanding loans.
- Update Interest Received Variable: The total interest amount computed in the previous step is assigned to the 'interestReceivedCredits' variable. This variable holds the total interest income received by the bank from its credit operations.

4.10 payoutInterests

This method ensures that interest payments are accurately calculated and distributed to depositors and the central bank, maintaining the bank's financial stability and compliance with its obligations.

- Calculate Interest Payments for Deposits: - For each deposit account held by the bank, the method calculates the interest payment based on the account balance and the interest rate on deposits. The interest payment for each deposit account is added to the 'interestPaidOnDeposits' variable, representing the total interest paid to depositors.

- Distribute Interest Payments: The method deducts the interest payments from each deposit account's balance. If the bank's excess liquidity is insufficient to cover the interest payments, it uses the standing facility to borrow from the central bank to meet its obligations.

- Update Central Bank Reserves: The interest payment owed to the central bank, if any, is calculated based on the standing facility's current debt level and interest rate. The central bank reserves account balance is adjusted accordingly to reflect the interest payment received.

- Adjust Excess Liquidity: If the excess liquidity after interest payments becomes negative, indicating a shortfall, the standing facility's current debt level is increased to cover the deficit. The central bank reserves account balance is adjusted to reflect this borrowing.

- Update Standing Facility Interest Payment: The interest payment owed on the standing facility is computed based on its current debt level and interest rate. This interest payment is added to the 'interestPaidToCB' variable, representing the total interest paid to the central bank.

4.11 profitAccounting

This method enables the bank to accurately assess its financial performance and determine its profits after accounting for various income and expense components.

- Calculate Interest Income: The method computes the total interest received by the bank from its credit portfolio. This includes interest payments from all outstanding loans provided by the bank.

- Calculate Interest Expenses: It determines the total interest paid by the bank on its deposits. This encompasses the interest payments made to depositors based on their account balances and the prevailing interest rate.

- Update Central Bank Interest Payment: The interest payment owed to the central bank, representing the interest on the standing facility borrowed from the central bank, is determined.

- Update Central Bank Reserves: The interest payment received from the central bank is added to the bank's central bank reserves account balance.
- Compute Profits: The method calculates the bank's profits by subtracting its total interest expenses (including interest paid to depositors and the central bank) from its total interest income. Additionally, any interests received from the central bank are added to the profits.
- Update Central Bank Balance Sheet: Finally, the method updates the central bank's balance sheet to reflect the interest payment received by the bank.

4.12 executePendingTransactions()

This method ensures that all financial transfers are completed accurately and promptly, reflecting the bank's current financial state and maintaining consistency with its balance sheet and transaction records.

- Processing Outgoing Transactions: The method iterates through each payment account in the bank's payment account list. For each payment account, it iterates through its list of pending outgoing transactions. It checks the receiver of each transaction:
  - If the receiver is the bank itself (internal transaction), it updates the account balances accordingly.
  - If the receiver is another bank, it identifies the receiver bank and updates the respective payment account balances accordingly.
  - If the receiver is a foreign bank, it locates the foreign bank and updates the account balance of the receiver's payment account.
- Processing Central Bank Reserves Transactions: Similarly, the method iterates through each pending transaction in the central bank reserves' pending outgoing transactions list. For each transaction, it checks if the receiver is the central bank itself. If the receiver is the central bank, it updates the central bank's balance sheet by adding the transaction amount to its cash reserves.

4.13 creditDefaultProcedure()

This method is responsible for handling the scenario when a credit defaults. It takes a 'Credit' object representing the defaulted credit as input. When a credit defaults, the method is invoked to handle the consequences. It may involve various actions depending on the system's design and requirements. Some common actions include:

- Updating the credit status to reflect the default.
- Adjusting the bank's balance sheet to account for the loss.
- Initiating debt recovery or restructuring procedures.

- Notifying relevant stakeholders about the default.
- Implementing risk mitigation measures to prevent future defaults.
- The specific steps taken in the credit default procedure can vary based on factors such as the severity of the default, the terms of the credit agreement, regulatory requirements, and risk management policies.

Handling credit defaults is crucial for maintaining the stability and integrity of the banking system. Effective credit default procedures help mitigate losses, protect stakeholders' interests, and ensure the bank's financial health. By implementing appropriate procedures, banks can minimize the impact of defaults on their operations and reputation, thereby fostering trust and confidence among depositors, investors, and regulators.

4.14 BalanceSheetBank.java

5 Central Bank

The class encapsulates functionality related to managing reserves, offering standing credit facilities, and maintaining the central bank's balance sheet. The balance sheet is updated periodically to reflect changes in reserves and standing facilities. Profit accounting ensures that the central bank's financial records are up-to-date and accurate. The central bank's actions, such as setting interest rates and regulating reserve requirements, can have significant implications for the broader economy, making these functionalities crucial in a simulation context.

5.1 profitAccounting()

Upon invocation, the profitAccounting() method calls the updateBalanceSheet() method. The primary purpose of this method is to ensure that the central bank's balance sheet is accurate and up-to-date. By calling updateBalanceSheet(), the central bank recalculates its assets, liabilities, and equity based on the current state of its reserves and standing facilities.

5.2 updateCBBalanceSheet()

Internally, the updateCBBalanceSheet() method delegates the task of updating the balance sheet to the updateBalanceSheet() method. This ensures code modularity and reusability, as both methods share the same functionality of updating the balance sheet.

5.3 updateBalanceSheet()

This method is responsible for recalculating the central bank's balance sheet based on its current financial state. The primary purpose of this method is to ensure that the central bank's balance sheet accurately reflects its financial position at any given time. It calculates the total assets, total liabilities, and equity of the central bank based on the information stored in its data structures, such as 'bankReserveList' and 'bankStandingFacilitiesList'.

- Calculation of Assets: The method iterates over the 'bankReserveList' and 'bankStandingFacilitiesList' to calculate the total assets of the central bank. It sums up the balances in the reserve accounts and the current debt levels of standing facilities to determine the total asset value.

- Calculation of Liabilities: For liabilities, the method computes the total reserves held by other banks in the 'bankReserveList'. It does not explicitly calculate other liabilities, such as central bank liabilities to governments or other institutions, which might be present in a real-world scenario but are not represented in this simplified simulation.

- Calculation of Equity: Once the total assets and liabilities are determined, the method calculates the central bank's equity by subtracting total liabilities from total assets. This computation adheres to the basic accounting principle that equity represents the residual interest in the assets of an entity after deducting liabilities.

5.4 Credit.java

5.5 CreditOffer.java

5.6 CreditApplication.java

5.7 BalanceSheetCentralBank.java

6 Statistic Agency

This class represents an entity responsible for gathering and computing various statistical data related to firms within a simulation environment. The class provides a set of getter methods for accessing specific statistical attributes. These methods allow external components to retrieve statistical data from the 'StatisticAgency' object. The 'StatisticAgency' class serves as a central component for collecting and processing statistical information related to firms within the simulation. It helps in monitoring the performance of various sectors, tracking economic indicators such as output and employment, and analyzing trends over time.

The class contains numerous attributes representing different statistical measures, such as total output, total inventory, unemployment rate, output by sectors, planned output, sales to households and firms, etc.

6.1 computeFirmStatistics()

This provides a comprehensive analysis of firm activities and economic indicators within the simulation, facilitating the monitoring and evaluation of the simulated economy's performance.

- Iteration through Firms:The method iterates through all the firms in the simulation, including 'ServiceFirm', 'GoodsFirm', and 'EnergyFirm'.
- Accumulation of Statistics: Total output, planned output, total imports, total exports, total inventory, total employment, average wage, and price index for domestic firms are accumulated by summing up the corresponding values for each firm.
- Sector-specific Statistics: Output statistics are computed separately for different sectors, such as goods, services, and energy sectors.
- Calculation of Unemployment Rate: The unemployment rate is calculated based on the total employment and the number of households in the simulation environment.
- Printing Statistics: After computing the statistics, the method prints out the total output, planned output, and total employment.
- Getter Methods: Getter methods are provided to retrieve specific statistics, such as the unemployment rate, total output, sales to firms, etc.

7 Clearing House

It hasn't been set.

8 Sector

8.1 laborMarketMatching()

This method simulates the process of matching job vacancies with job seekers based on wage offers and reservation wages, facilitating the employment of workers by firms within the sector.

- Shuffling Lists: Before matching, the method shuffles two lists: ‘laborMarket.vacancyList‘ and ‘laborMarket.jobSearchList‘. This shuffling ensures randomness in the matching process.
- Sorting Vacancy List: Next, the method sorts the ‘laborMarket.vacancyList‘ by wages in descending order. This sorting allows higher-wage offers to be processed first, increasing the likelihood of matching with potential employees.
- Matching Process: The method iterates through each vacancy in the ‘laborMarket.vacancyList‘ and each job seeker in the ‘laborMarket.jobSearchList‘. For each vacancy, it checks if the wage offer exceeds the reservation wage of the job seeker.
- Matching Criteria: If the wage offer is higher than the job seeker’s reservation wage, a ‘WorkContract‘ object is created, representing an agreement between the employer (firm) and the employee (household) at the offered wage. The job seeker is activated with the new work contract, and the vacancy is filled.
- Updating Lists: After a successful match, the matched vacancy and job seeker are removed from their respective lists. This prevents them from being considered for future matches.
- Open Positions: At the end of the matching process, the ‘laborMarket.openPositions‘ variable is updated to reflect the number of remaining vacancies in the labor market.

8.2 cleanUp()

The primary purpose of the cleanUp() method is to reset the labor market by removing all entries from the job search list and vacancy list.
The jobSearchList and vacancyList of the laborMarket object are cleared using the clear() method. This ensures that any remaining entries from previous iterations are removed, and the labor market starts fresh for the next iteration of the simulation.

8.3 clearOrderBooks1()

This method ensures that the order books of firms within the service sector are reset at regular intervals to maintain the accuracy of market interactions and simulations. The primary purpose of this method is to reset the order books of all firms operating within the malls of the service sector.
The method iterates through each mall (‘ServiceMall‘) in the ‘mallList‘. For each mall, it

iterates through the inventories of the firms (‘ServiceFirm‘) within that mall. It then clears the order book of each firm by calling the ‘clear()‘ method on the order book.
Clearing the order books ensures that orders from previous simulation ticks are removed, allowing firms to start fresh in each tick. This process prevents accumulation of outdated orders and ensures that firms respond to current market conditions accurately.

8.4 allocateOrders1()

This method simulates the allocation of orders to firms within the service sector, contributing to the dynamics of market interactions and economic activities. The main purpose of this method is to allocate orders to firms operating within the malls of the service sector. The method iterates through each mall (‘ServiceMall‘) in the ‘mallList‘. For each mall, it initializes the mall by calling the ‘setup()‘ method. Then, it allocates orders to the firms within that mall by calling the ‘allocateOrders()‘ method on the mall. The ‘allocateOrders()‘ method is responsible for distributing orders among the firms based on certain criteria or algorithms.
Allocating orders to firms simulates the process of customers placing orders for goods or services. This step is crucial for driving the economic activity within the simulation by ensuring that firms receive orders and produce accordingly.

8.5 computeStatistics()

This method in the ‘ServiceSector‘ class is responsible for calculating various statistics related to the performance and characteristics of firms within the service sector. The main purpose of this method is to compute statistics such as total sales, price index, and average knowledge stock for firms within the service sector.
It initializes variables to store total sales, price index, total knowledge stock, and average knowledge stock. Then, it iterates through each mall (‘ServiceMall‘) in the ‘mallList‘. For each mall, it iterates through the firm inventories within that mall and calculates the total sales and accumulates the sold quantities for each firm. It calculates the price index by dividing the sum of sold quantities by the total sales. It counts the number of firms in the sector and accumulates their research and development (R&D) knowledge stock. Finally, it calculates the average knowledge stock by dividing the total knowledge stock by the number of firms.
Computing statistics helps in analyzing the performance and characteristics of firms within the service sector. It provides insights into market trends, such as total sales and price movements, which can be valuable for decision-making and policy formulation. Calculating average knowledge stock reflects the innovation and technological advancement level within the sector.

8.6 ServiceOrder.java

8.7 ServiceInventory.java

8.8 GoodsSector.java

clear Order Books1;allocate Orders 1;clear Order Books2; allocate Orders 2;compute Statistics

8.9 GoodsOrder.java

8.10 GoodsInventory.java

8.11 EnergySector.java

clear Order Books 1; allocate Orders 1;clear Order Books 2; allocate Orders 2 compute Statistics

8.12 EnergyOrder.java

8.13 EnergyContract.java

8.14 Inventory.java

9 Mall

It hasn't been set.

9.1 GoodsMall.java

9.2 ServiceMall.java

9.3 EnergyMall.java

10 Labor Market

10.1 JobSearch.java

It hasn't been set.

10.2 WorkContract.java

It hasn't been set.

10.3 VacancyComparator.java

It hasn't been set.

10.4 Vacancy.java

It hasn't been set.

10.5 TechnicalCoefficient.java

10.6 Region.java

11 Rest of World

This class simulates the interaction between the home country and rest of the world, handling exports, imports, and financial transactions. It organizes sectors into profiles and manages the flow of goods and services between them, adjusting import levels based on demand and delivering imported goods to appropriate entities. These attributes store essential information about the region's identity, financial transactions, export and import profiles, and volumes of trade. They are used throughout the class methods to manage the simulation's behavior and outcomes.

budgetAllocation() This method is to distribute budgets among different sector export profiles based on their share of the nominal export volume target. This allocation ensures that each sector receives an appropriate budget for exporting goods and services.
The method iterates over each sector export profile stored in the sectorExportProfiles list. For each sector export profile, it calculates the budget by multiplying the share of the nominal export volume target by the total nominal export volume target. The calculated budget is then assigned to the budget attribute of the sector export profile.

11.1 exportRequestsGoods()

This method orchestrates the export process for goods sectors by determining export quantities, selecting inventory items for export based on certain criteria, and generating export orders for further processing. *exportRequestsService()* and *exportRequestsEnergy()* follow the similar logics.

- Export Profile Iteration: It iterates over each sector's export profiles stored in the 'sectorExportProfiles' list.
- Sector Identification: Within each iteration, it identifies the corresponding goods sector by matching the sector ID from the export profile with the sector ID in the simulation.
- Export Calculations: For each goods sector, the method calculates the export quantities and generates export orders for eligible goods inventories. It uses a logit function to calculate the probability of selecting each inventory item for export, based on its price and the intensity of supplier choice ('intensityChoiceSupplier').
- Order Generation: Export orders are created for selected inventory items, specifying the quantity to be exported and other relevant details.

- Order Placement: The generated export orders are added to the respective inventory's order book for further processing.

11.2 receiveExports()

This method effectively manages the reception of exports from various sectors, ensuring that the received quantities are correctly recorded and accounted for. It updates the nominal export volume and total exports, providing crucial information for analyzing the region's trade dynamics and financial transactions.
Before processing the exports, the method first clears the 'totalExports' variable to prepare for recalculating the total exports. The method then iterates over each sector's export profiles stored in the 'sectorExportProfiles' list. For each export profile, it processes exports from different types of firms (goods, services, energy). For goods, services, and energy exports:

- It iterates over the list of firm exports within each sector's export profile.
- For each firm export, it records the received quantity based on the accepted quantity from the firm's export order.
- It updates the nominal export volume based on the received quantity and the invoice amount from the export order.
- It adds a payment transaction to the region's payment account for the sales revenue generated from the exports.
- It updates the 'totalExports' variable by adding the received quantity.

11.3 adjustImports()

This method ensures that the import inventory levels and prices are adjusted dynamically to respond to changes in demand and market conditions within the simulation.

- Iterating Over Import Profiles: The method iterates over each import profile stored in the 'sectorImportProfiles' attribute.
- Adjusting Goods Imports: For each import profile associated with goods sectors ('importFirmListGoods'), the method adjusts the inventory levels and prices of imported goods based on the total excess demand. If there's excess demand, the refill level of the inventory is increased by 5% of the total excess demand. Additionally, if the average price of imported goods is available, it updates the price of goods. Service and energy imports follow the same logics.
- Updating Inventory Levels: After adjusting the refill levels and prices, the method updates the inventory levels of imported goods, services, and energy based on the adjusted quantities.

11.4 deliverImports()

This method ensures that the imported goods and services are appropriately delivered and accounted for in the region's inventory, contributing to the functioning of the simulation's trade dynamics.

- Clear Total Imports: At the beginning of the method, the 'totalImports' attribute is set to 0. This variable will be used to keep track of the total quantity of imported goods and services during the current iteration.
- Iterate Through Sector Import Profiles: The method iterates through each sector's import profiles stored in the 'sectorImportProfiles' list.
- Process Import Profiles: For each sector's import profile, the method further iterates through the list of import firms (separated by goods, services, and energy).
- Calculate Quantity to Deliver: For each import firm, the method calculates the quantity of goods or services to deliver. It computes the difference between the refill level and the current inventory stock.
- Update Total Imports: The quantity calculated in the previous step is added to the 'totalImports' variable to keep track of the total imported quantity.
- Update Inventory Stock: The calculated quantity is then added to the inventory stock of the import firm.
- Update Inventory Quality: The method also updates the quality of the inventory stock based on the quality attribute associated with the import firm.

There are some nested classes:

- **SectorExportProfile**: Represents export profiles for each sector, containing details like sector ID, share, budget, and lists of exported items.
- **SectorImportProfile**: Represents import profiles for each sector, containing details like sector ID, price, quantity, and lists of importing firms.
- **ExportItemGoods**, **ExportItemService**, **ExportItemEnergy**: Represent export items for goods, services, and energy respectively, containing details like firm ID, bank ID, share, demand, received quantity, and order.

11.5 PortfolioItem.java

11.6 Plant.java

11.7 PaymentTransaction.java

11.8 PaymentAccount.java

11.9 Order.java

11.10 ImportGoodsFirm.java

11.11 ImportFirmService.java

11.12 ImportFirmEnergy.java

11.13 ImportFirm.java

11.14 ForeignBank.java

11.15 BalanceSheet.java

execute Transactions 1;execute Transactions 2

12 Calculations

unemploymentRate = Math.max(0,1 - totalEmployment/numHouseholds); numHouseholds=10000; this.totalEmployment += obj.laborStock; laborStock = 0; for(int i =0; i < workContractList.size(); i++) if(workContractList.get(i).canceled) workContractList.remove(i); i–; else laborStock +=1; laborStock = ownSector.laborForce / ownSector.numFirms laborForce = employmentShare*totalSizeLaborForce; totalSizeLaborForce = employmentShare*numHouseholds employmentShare = 1.0;

$$\text{potentialOutput} = \text{TFP} \times \min \left(\begin{array}{l} \text{ownSector.laborCoefficient} \times \text{laborStock}, \\ \text{ownSector.capitalCoefficient} \times \text{capitalStock}, \\ \text{ownSector.serviceCoefficient} \times \text{serviceLevel}, \\ \text{ownSector.intermediateInputCoefficient} \times \text{intermediateGoodsStock}, \\ \text{ownSector.energyCoefficient} \times \text{energyLevel} \end{array} \right)$$

It takes into account various factors such as labor, capital, service level, intermediate goods stock, and energy level, each weighted by their respective coefficients in the ownSector object. The min function ensures that the potential output is limited by the factor that has the smallest contribution, which is a common approach in production modeling to reflect the concept of limiting factors.

13 ConsumptionCoefficient.java

13.1 ConsumptionCoefficientService.java

13.2 ConsumptionCoefficientGoods.java

13.3 ConsumptionCoefficientEnergy.java

14 Apeendix

Here are the all the appendix information of this project.

14.1 File lists

Files	Description	Notes
BalanceSheet.java	0.02021	0.02912
BalanceSheetBank.java	0.02021	0.02912
BalanceSheetCentralBank.java	0.02021	0.02912
Bank.java	0.02021	0.02912
cap_stock.csv	0.02021	0.02912
CentralBank.java	0.02021	0.02912
ClearingHouse.java	0.02021	0.02912
consumption.csv	0.02021	0.02912
ConsumptionCoefficient.java	0.02021	0.02912
ConsumptionCoefficientEnergy.java	0.02021	0.02912
ConsumptionCoefficientGoods.java	0.02021	0.02912
ConsumptionCoefficientService.java	0.02021	0.02912
Credit.java	0.02021	0.02912
CreditApplication.java	0.02021	0.02912
CreditOffer.java	0.02021	0.02912
empl.csv	0.02021	0.02912
empl_shares.csv	0.02021	0.02912
EnergyContract.java	0.02021	0.02912
EnergyFirm.java	0.02021	0.02912
EnergyInventory.java	0.02021	0.02912
EnergyMall.java	0.02021	0.02912
EnergyOrder.java	0.02021	0.02912

Files	Description	Notes
EnergySector.java	0.02021	0.02912
energy_coefficient.csv	0.02021	0.02912
exports.csv	0.02021	0.02912
filenames.txt	0.02021	0.02912
Firm.java	0.02021	0.02912
ForeignBank.java	0.02021	0.02912
GoodsFirm.java	0.02021	0.02912
GoodsInventory.java	0.02021	0.02912
GoodsMall.java	0.02021	0.02912
GoodsOrder.java	0.02021	0.02912
GoodsSector.java	0.02021	0.02912
Household.java	0.02021	0.02912
ImportFirm.java	0.02021	0.02912
ImportFirmEnergy.java	0.02021	0.02912
ImportFirmService.java	0.02021	0.02912
ImportGoodsFirm.java	0.02021	0.02912
imports.csv	0.02021	0.02912
input_coefficents.csv	0.02021	0.02912
intermediate_coefficient.csv	0.02021	0.02912
Inventory.java	0.02021	0.02912
investments.csv	0.02021	0.02912

Files	Description	Notes
JobSearch.java	0.02021	0.02912
LaborMarket.java	0.02021	0.02912
Mall.java	0.02021	0.02912
Model.java	0.02021	0.02912
Order.java	0.02021	0.02912
PaymentAccount.java	0.02021	0.02912
PaymentTransaction.java	0.02021	0.02912
Plant.java	0.02021	0.02912
PortfolioItem.java	0.02021	0.02912
Region.java	0.02021	0.02912
ROW.java	0.02021	0.02912
Sector.java	0.02021	0.02912
ServiceContract.java	0.02021	0.02912
ServiceFirm.java	0.02021	0.02912
erviceInventory.java	0.02021	0.02912
ServiceMall.java	0.02021	0.02912
ServiceOrder.java	0.02021	0.02912
ServiceSector.java	0.02021	0.02912
SpecificSkill.java	0.02021	0.02912
StatisticAgency.java	0.02021	0.02912
Stock.java	0.02021	0.02912

Files	Description	Notes
TechnicalCoefficient.java	0.02021	0.02912
Vacancy.java	0.02021	0.02912
VacancyComparator.java	0.02021	0.02912
workContract.java	0.02021	0.02912

14.2 Q&A

1. Infinite loop firstPlannedOutput -313.59936521946497
plannedOutput 2.5E-323
plannedRDExpenditure -950.0737957485776
revenue -31669.12652495259
salesToFirms 0.0
salesToHouseholds -12552.522078027177
plannedOutput < 0
salesToHouseholds < 0

I found in GoodsMall.java
firmInventories.get(i).salesToHouseholds += firmInventories.get(i).orderBook.get(j).acceptedQuantity;
firmInventories.get(i).orderBook.get(j).acceptedQuantity = Math.min(firmInventories.get(i).orderBook.get

inventoryStock -130712.13898964223
this.mallInventoryList.get(0).inventoryStock = 1.0*initialOutput;
But
initialOutput 1433.2000795105666

intermediateGoodsStock -2217.0982054786696
receivedQuantity -2522.6927900784726
acceptedQuantity -2522.6927900784726
rationingQuota -0.016903661365253564
internalFinancialResources -17697.609537370798
valueInventory 32132.01711416454
but in GoodsFirm.java, should the valueInventory be a negative value as well
valueInventory = this.internalInventoy.inventoryStock*price;
inventoryStock -130712.13898964223
price 1.0

14.3 Attributes

- *Constructor* method to create new Firm instance.
- *ID, sector, clearingHouseID* etc. to uniquely identify the firm.
- *space* and *grid* attributes identify firm's location.
- *supplierList* and *potentialSupplierList* model network of input suppliers.
- *contractLists* track ongoing contracts for inputs, services, labor etc..
- *workContractList* stores workforce contracts.
- *creditOfferList* and *creditList* track credit relationships.
- *plantList* contains the firm's production facilities.
- *inputGoodsRequirements* and *inputServiceRequirements* define needed inputs.
- *Technology* and *productivity* attributes model production capabilities.
- *outputIntermediate*, *outputLabor* etc track outputs.

- *previousOutputs* and *plannedOutput* track production plans.
- *stock* tracks firm's equity shares.
- *paymentAccount* records cash balance.
- *balanceSheet* records assets and liabilities.
- *revenues*, *costs* and *profit* variables track financials.
- *interests*, *dividends* etc. model financial flows.
- *creditCosts*, *debtRestructuring* etc. model credit and bankruptcy.

- *ID*: An integer representing the ID of the household.
- *tick*: An integer representing the current tick in the simulation.
- *bankID*: An integer representing the ID of the bank associated with the household.
- *clearingHouseID*: An integer representing the ID of the clearing house associated with the household.
- Various attributes representing financial and economic parameters such as *disposableIncome*, *consumptionBudget*, *propensityToConsume*, etc.
- *employed*: A boolean indicating whether the household is currently employed or not.
- *onTheJobSearch*: A boolean indicating whether the household is actively searching for a job.
- *generalSkillLevel*: An integer representing the general skill level of the household.
- *workContract*: An object of type *WorkContract* representing the current work contract of the household.
- *currentWorkingSector*: An object representing the sector in which the household is currently employed.
- *paymentAccount*: An object of type *PaymentAccount* representing the payment account of the household.
- *consumptionCoefficientList*: A list of objects representing consumption coefficients for different sectors.
- *potentialSupplierList*: A list of potential suppliers for household consumption.
- *assetPortfolio*: A list representing the asset portfolio of the household.
- Constructor: Initializes the household with the provided ID, bank ID, and clearing house ID. Initializes various lists and objects.
- *space* and *grid*: These attributes represent the spatial location of the bank within the simulation environment. The 'space' is a continuous space, while the 'grid' is a grid-based space.

- *ID*: An integer representing the unique identifier of the bank.
- *centralBankID*: An integer representing the unique identifier of the central bank associated with this bank.
- *foreignBankID*: An integer representing the unique identifier of the foreign bank associated with this bank.
- *capitalAdequacyRatio*: A double value representing the capital adequacy ratio of the bank, which is a measure of its financial health and stability.
- *reserveRequirementRatio*: A double value representing the reserve requirement ratio, indicating the portion of deposits that the bank must hold in reserve.
- *riskExposureBudget*: A double value representing the budget allocated for managing the bank's risk exposure.
- *riskWeightedAssets*: A double value representing the total value of the bank's assets weighted by their respective risk factors.
- *excessLiquidity*: A double value representing the excess liquidity available to the bank, which is the difference between its cash reserves and the required reserve amount.
- *totalCreditDemand*: A double value representing the total demand for credit from borrowers.
- *profits*: A double value representing the profits earned by the bank.
- *interestPaidOnDeposits*: A double value representing the total interest paid on deposits held by the bank.
- *interestReceivedCredits*: A double value representing the total interest received on credits extended by the bank.
- *interestPaidToCB*: A double value representing the interest paid by the bank to the central bank.
- *interestsReceivedFromCB*: A double value representing the interest received by the bank from the central bank.
- *initEquity*: A double value representing the initial equity or capital of the bank.
- *paymentAccountList*: An ArrayList of PaymentAccount objects representing the payment accounts associated with the bank.
- *creditList*: An ArrayList of Credit objects representing the credits extended by the bank.
- *creditApplicationList*: An ArrayList of CreditApplication objects representing the credit applications received by the bank.

- *acceptedCreditApplications*: An ArrayList of CreditOffer objects representing the credit applications accepted by the bank.
- *balanceSheet*: An instance of the BalanceSheetBank class representing the balance sheet of the bank.
- Various other attributes related to interest rates, lambda values, and repayment periods, which are used in financial calculations and operations within the bank.
- *ID*: An integer identifier for the central bank.
- *balanceSheet*: An instance of the 'BalanceSheetCentralBank' class representing the balance sheet of the central bank.
- *bankReserveList*: An ArrayList containing PaymentAccount objects representing reserves held by commercial banks at the central bank.
- *bankStandingFacilitiesList*: An ArrayList containing Credit objects representing standing credit facilities offered by the central bank to commercial banks.
- *interestBaseRate*: Base interest rate set by the central bank.
- *capitalAdequacyRatio*: Capital adequacy ratio, a regulatory measure representing the ratio of a bank's capital to its risk-weighted assets.
- *reserveRequirementRatio*: Reserve requirement ratio, indicating the portion of customer deposits that banks must hold in reserve.
- *interestMarkDownbankReserves*: A markdown applied to the interest rate on bank reserves.
- *technicalCoefficients*: An 'ArrayList' to store technical coefficients derived from input-output tables.
- *sectorID*: An integer representing the ID of the sector.
- *type*: An integer representing the type of sector (1 for goods, 2 for services, 3 for energy).
- *numFirms*: An integer representing the number of firms in the sector.
- *capitalCoefficent*: A double representing the coefficient of capital in the production function.
- *laborCoefficient*: A double representing the coefficient of labor in the production function.
- *intermediateInputCoeffient*: A double representing the coefficient of intermediate inputs in the production function.

- *energyCoefficient*: A double representing the coefficient of energy in the production function.
- *serviceCoefficient*: A double representing the coefficient of services in the production function.
- *employmentShare*: A double representing the share of employment in the sector.
- *employmentData*: A double representing employment data specific to the sector.
- *laborForce*: A double representing the total labor force in the sector.
- *wage*: A double representing the average wage in the sector.
- *totalLaborCosts*: A double representing the total labor costs in the sector.
- *initialOutput*: A double representing the initial output of the sector.
- *importQuota*: A double representing the import quota of the sector.
- *priceIndex*: A double representing the price index of the sector.
- *avgKnowledgeStock*: A double representing the average knowledge stock in the sector.
- *laborMarket*: An instance of the 'LaborMarket' class representing the labor market for the sector.
- *foreignBankID*: Identifies the foreign bank associated with the region.
- *intensityChoiceSupplier*: Indicates the intensity of supplier choice using a greedy algorithm.
- *paymentAccount*: Represents the payment account associated with the region.
- *sectorExportProfiles*: ArrayList storing export profiles for different sectors.
- *sectorImportProfiles*: ArrayList storing import profiles for different sectors.
- *nominalExpotVolumeTarget*: Target nominal export volume.
- *nominalExpotVolume*: Actual nominal export volume.
- *totalImports*: Total imported goods and services.
- *totalExports*: Total exported goods and services.

References