Files:

* Banks
* CapitalMarket
* CentralBank
* CFirms
* ConsumptionMarket
* CreditMarket
* DepositsMarket
* Generator
* Government
* Households
* KFirms
* LaborMarket
* Main
* Parameters
* Statistics
* TimeIteration

import parameters

import random

import numpy as np

import matplotlib.pyplot as plt

plt.style.use('ggplot')

import pandas as pd

class household:

pass

class consumptionfirm:

pass

class capitalfirm:

pass

class commercialbank:

pass

class centralbank:

pass

class government:

pass

# The number of iterations and data points desired are set.

global n\_iterations; n\_iterations = 1000

global n\_data\_points; n\_data\_points = 100

def initialize():

global hhs; hhs = []

global nhhs; nhhs = parameters.param\_dict['Number of households']

for i in range(nhhs):

hh = household()

hh.ID = "H" + str(i + 1)

hh.deposits = parameters.param\_dict['Initial household deposits']/nhhs

hh.interest\_on\_deposits = hh.deposits\*parameters.param\_dict['Initial interest rate on deposits']

hh.employer = None

hh.old\_supplier = None

hh.bank\_deposits = None

hh.wage = parameters.param\_dict['Initial wages']

hh.last\_price = parameters.param\_dict['Initial price for C goods']

hh.exp\_price = None

hh.net\_disposable\_income = parameters.param\_dict['Initial net income']/nhhs

hh.desired\_consumption = parameters.param\_dict['Initial consumption by households']/nhhs

hhs.append(hh)

global cfirms; cfirms = []

global ncfirms; ncfirms = parameters.param\_dict['Number of consumption firms']

for i in range(ncfirms):

firm = consumptionfirm()

firm.ID = "CF" + str(i + 1)

firm.employees = []

firm.supplier\_list = None

firm.status = None

firm.mark\_up = parameters.param\_dict['Initial mark-up on ULC for C firms']

firm.old\_supplier = None

firm.old\_bank = None

firm.bank\_deposits = None

firm.deposits = parameters.param\_dict['Initial deposits for C sector']/ncfirms

firm.interest\_on\_deposits = firm.deposits\*parameters.param\_dict['Initial interest rate on deposits']

firm.sales = parameters.param\_dict['Initial output of C sector']/ncfirms

firm.exp\_sales = None

firm.inventories = parameters.param\_dict['Initial inventories for C sector']/ncfirms

firm.inventories\_past = parameters.param\_dict['Initial inventories for C sector']/ncfirms

firm.output = parameters.param\_dict['Initial output of C sector']/ncfirms

firm.price = parameters.param\_dict['Initial price for C goods']

firm.unit\_cost\_present = parameters.param\_dict['Initial unit costs for C sector']

firm.exp\_OCF = None

firm.OCF = parameters.param\_dict['Initial OFC for C sector']/ncfirms

firm.profit\_rate = parameters.param\_dict['Target profit rate (Investment function)']

firm.exp\_profit\_post\_tax = None

firm.profit\_post\_tax = (parameters.param\_dict['Initial pre-tax profits for C sector'] - parameters.param\_dict['Initial taxes paid by C sector'])/ncfirms

firm.exp\_average\_wage = None

firm.capital\_sheet = [] #[number\_of\_k,age,price,value,supplier]

for i in range(parameters.param\_dict['Capital goods duration']):

firm.capital\_sheet.append([parameters.param\_dict['Initial stock of K']/(ncfirms\*parameters.param\_dict['Capital goods duration']),

i,

parameters.param\_dict['Initial price for K goods']/((1 + parameters.param\_dict['Nominal rate of growth in the SS'])\*\*i),

(parameters.param\_dict['Initial stock of K']\*parameters.param\_dict['Initial price for K goods']\*(parameters.param\_dict['Capital goods duration'] - i))/(ncfirms\*(parameters.param\_dict['Capital goods duration']\*\*2)\*((1 + parameters.param\_dict['Nominal rate of growth in the SS'])\*\*i)),

None])

firm.capital\_sheet.reverse()

firm.capital\_stock = 0

for element in firm.capital\_sheet:

firm.capital\_stock += element[0]

firm.loan\_granted = 0

firm.loan\_sheet = [] #[value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

for i in range(parameters.param\_dict['Loans duration']):

firm.loan\_sheet.append([parameters.param\_dict['Initial Loan granted a t=0 to C sector']/(ncfirms\*((1 + parameters.param\_dict['Nominal rate of growth in the SS'])\*\*i)),

i,

parameters.param\_dict['Initial interest rate on loans'],

(parameters.param\_dict['Initial Loan granted a t=0 to C sector']\*(parameters.param\_dict['Loans duration'] - i))/(ncfirms\*parameters.param\_dict['Loans duration']\*((1 + parameters.param\_dict['Nominal rate of growth in the SS'])\*\*i)),

None,

None])

firm.loan\_sheet.reverse()

cfirms.append(firm)

global kfirms; kfirms = []

global nkfirms; nkfirms = parameters.param\_dict['Number of capital firms']

for i in range(nkfirms):

firm = capitalfirm()

firm.ID = "KF" + str(i + 1)

firm.employees = []

firm.status = None

firm.mark\_up = parameters.param\_dict['Initial mark-up on ULC for K firms']

firm.old\_bank = None

firm.bank\_deposits = None

firm.deposits = parameters.param\_dict['Initial deposits for K sector']/nkfirms

firm.interest\_on\_deposits = firm.deposits\*parameters.param\_dict['Initial interest rate on deposits']

firm.sales = parameters.param\_dict['Initial output of K sector']/nkfirms

firm.exp\_sales = None

firm.inventories = parameters.param\_dict['Initial inventories for K sector']/nkfirms

firm.inventories\_past = parameters.param\_dict['Initial inventories for K sector']/nkfirms

firm.output = parameters.param\_dict['Initial output of K sector']/nkfirms

firm.price = parameters.param\_dict['Initial price for K goods']

firm.unit\_cost\_present = parameters.param\_dict['Initial unit costs for K sector']

firm.OCF = parameters.param\_dict['Initial OFC for K sector']/nkfirms

firm.exp\_profit\_post\_tax = None

firm.profit\_post\_tax = (parameters.param\_dict['Initial pre-tax profits for K sector'] - parameters.param\_dict['Initial taxes paid by K sector'])/nkfirms

firm.exp\_average\_wage = None

firm.loan\_granted = 0

firm.loan\_sheet = [] #[value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

for i in range(parameters.param\_dict['Loans duration']):

firm.loan\_sheet.append([parameters.param\_dict['Initial Loan granted a t=0 to K sector']/(nkfirms\*((1 + parameters.param\_dict['Nominal rate of growth in the SS'])\*\*i)),

i,

parameters.param\_dict['Initial interest rate on loans'],

(parameters.param\_dict['Initial Loan granted a t=0 to K sector']\*(parameters.param\_dict['Loans duration'] - i))/(nkfirms\*parameters.param\_dict['Loans duration']\*((1 + parameters.param\_dict['Nominal rate of growth in the SS'])\*\*i)),

None,

None])

firm.loan\_sheet.reverse()

kfirms.append(firm)

global banks; banks = []

global nbanks; nbanks = parameters.param\_dict['Number of banks']

for i in range(nbanks):

bank = commercialbank()

bank.ID = "B" + str(i + 1)

bank.i\_loans = parameters.param\_dict['Initial interest rate on loans']

bank.i\_deposits = parameters.param\_dict['Initial interest rate on deposits']

bank.reserves = 0

bank.loan\_number = 0

bank.bonds = parameters.param\_dict['Initial bonds held by banks']/(nbanks\*parameters.param\_dict['Bonds price'])

bank.net\_worth = parameters.param\_dict['Initial net wealth for banks']/nbanks

bank.cash\_advances = 0

bank.loan\_sheet = []

bank.deposits\_list = []

banks.append(bank)

global cb

cb = centralbank()

cb.ID = "CB"

cb.bonds = parameters.param\_dict['Initial bonds held by CB']/parameters.param\_dict['Bonds price']

cb.cash\_advances = 0

global gov

gov = government()

gov.ID = "G"

gov.employees = []

gov.bonds = parameters.param\_dict['Initial stocks of bonds in the economy']/parameters.param\_dict['Bonds price']

gov.reserves = 0

global FNmean; FNmean = parameters.param\_dict['Folded Normal Distribution Mean']

global FNsd; FNsd = parameters.param\_dict['Folded Normal Distribution Variance']

global max\_u\_rate; max\_u\_rate = parameters.param\_dict['Initial unemployment']

global i\_cash\_advances; i\_cash\_advances = parameters.param\_dict['CB interest rates on advances']

global i\_bonds; i\_bonds = parameters.param\_dict['Bonds interest rate']

global x\_capital; x\_capital = parameters.param\_dict['Number of potential partners on capital goods market']

global x\_credit; x\_credit = parameters.param\_dict['Number of potential partners on credit market']

global x\_labour; x\_labour = parameters.param\_dict['Number of potential partners on labour market (for each vacant job)']

global x\_consumption; x\_consumption = parameters.param\_dict['Number of potential partners on consumption goods market']

global x\_deposits; x\_deposits = parameters.param\_dict['Number of potential partners on deposits market']

global e\_capital; e\_capital = parameters.param\_dict['Intensity of choice in capital market']

global e\_credit; e\_credit = parameters.param\_dict['Intensity of choice in credit market']

global e\_consumption; e\_consumption = parameters.param\_dict['Intensity of choice in consumption market']

global e\_deposits; e\_deposits = parameters.param\_dict['Intensity of choice in deposits market']

global phi; phi = parameters.param\_dict['Adaptive expectations parameter']

global tau\_hh; tau\_hh = parameters.param\_dict['Income tax rate']

global tau\_pi; tau\_pi = parameters.param\_dict['Profit tax rate']

global omega; omega = parameters.param\_dict['Dole (share of average wages)']

global alpha\_1; alpha\_1 = parameters.param\_dict['Propensity to consume out of income']

global alpha\_2; alpha\_2 = parameters.param\_dict['Propensity to consume out of wealth']

global buffer; buffer = parameters.param\_dict['Firms inventories target share']

global period; period = 0

global max\_capital\_age; max\_capital\_age = parameters.param\_dict['Capital goods duration']

global max\_loan\_age; max\_loan\_age = parameters.param\_dict['Loans duration']

global capital\_productivity; capital\_productivity = parameters.param\_dict['Productivity of K']

global capital\_labour\_ratio; capital\_labour\_ratio = parameters.param\_dict['Capital/labour ratio']

global labour\_productivity; labour\_productivity = parameters.param\_dict['Productivity of labour in K sector']

global capacity\_utilization\_normal; capacity\_utilization\_normal = parameters.param\_dict['Target capacity utilization (Investment function)']

global profit\_rate\_normal; profit\_rate\_normal = parameters.param\_dict['Target profit rate (Investment function)']

global gamma\_1; gamma\_1 = parameters.param\_dict['Profit rate weight (Investment function)']

global gamma\_2; gamma\_2 = parameters.param\_dict['Capacity utilization rate weight (Investment function)']

global div\_firm; div\_firm = parameters.param\_dict['Firms profits share distributed as dividends']

global div\_bank; div\_bank = parameters.param\_dict['Banks profit share distributed as dividends']

global sigma; sigma = parameters.param\_dict['Firms precautionary deposits as share of WB']

global varsigma\_c; varsigma\_c = parameters.param\_dict['Banks risk aversion towards C firms']

global varsigma\_k; varsigma\_k = parameters.param\_dict['Banks risk aversion towards K firms']

global iota; iota = parameters.param\_dict['Haircut on defaulted firms capital value']

global epsilon; epsilon = 0.001

global turnover; turnover = parameters.param\_dict['Labour turnover ratio']

global min\_liquidity\_ratio; min\_liquidity\_ratio = 0.08

global min\_capital\_ratio; min\_capital\_ratio = 0.06

global min\_employees; min\_employees = 1

global min\_capital\_units; min\_capital\_units = 20

global bankruptcy\_buffer; bankruptcy\_buffer = 0.20

global price\_bonds; price\_bonds = parameters.param\_dict['Bonds price']

global Y\_cons\_nominal; Y\_cons\_nominal = []

global CPI; CPI = []

global Y\_cons\_real; Y\_cons\_real = []

global unemployment; unemployment = []

global GINI\_income\_pre; GINI\_income\_pre = []

global GINI\_income\_post; GINI\_income\_post = []

global GINI\_wealth\_pre; GINI\_wealth\_pre = []

global GINI\_wealth\_post; GINI\_wealth\_post = []

global D\_consumption; D\_consumption = []

global Planned\_capital\_purchase; Planned\_capital\_purchase = []

global Capital\_sales; Capital\_sales = []

global K\_inventories; K\_inventories = []

global Available\_deposits; Available\_deposits = []

global Average\_wages; Average\_wages = []

global data; data = {}

for firm in cfirms:

data[str(firm.ID)+'\_status'] = [firm.status]

data[str(firm.ID)+'\_sales'] = [firm.sales]

data[str(firm.ID)+'\_inventories'] = [firm.inventories]

data[str(firm.ID)+'\_buffer'] = [firm.inventories/firm.sales]

data[str(firm.ID)+'\_price'] = [firm.price]

data[str(firm.ID)+'\_mark\_up'] = [firm.mark\_up]

data[str(firm.ID)+'\_ULC'] = [None]

data[str(firm.ID)+'\_ULC\_expected'] = [None]

data[str(firm.ID)+'\_number\_employees'] = [len(firm.employees)]

data[str(firm.ID)+'\_output\_desired'] = [None]

data[str(firm.ID)+'\_output\_planned'] = [None]

data[str(firm.ID)+'\_output'] = [firm.output]

data[str(firm.ID)+'\_capital\_stock'] = [firm.capital\_stock]

data[str(firm.ID)+'\_growth\_rate\_desired'] = [None]

data[str(firm.ID)+'\_profit\_rate'] = [firm.profit\_rate]

data[str(firm.ID)+'\_uc\_rate'] = [None]

data[str(firm.ID)+'\_capital\_demand'] = [None]

data[str(firm.ID)+'\_capital\_bought'] = [None]

data[str(firm.ID)+'\_loan\_granted'] = [firm.loan\_granted]

data[str(firm.ID)+'\_total\_loan\_repayment'] = [None]

data[str(firm.ID)+'\_deposits'] = [firm.deposits]

data[str(firm.ID)+'\_OCF'] = [firm.OCF]

data[str(firm.ID)+'\_%L\_of\_costs'] = [None]

data[str(firm.ID)+'\_agg\_wages'] = [None]

def expectation(actual, exp = None):

global phi

if exp == None:

exp = actual

return phi\*exp + (1-phi)\*actual

def initial\_network():

global hhs; global nhhs

global cfirms; global ncfirms

global kfirms; global nkfirms

global banks; global nbanks; global max\_loan\_age

global gov

for firm in cfirms:

# Consumption firms hire workers.

random.shuffle(hhs)

i = 0

# There is a maximum amount of workers that can be hired.

while len(firm.employees) < parameters.param\_dict['Consumption firms initial workers']/ncfirms:

hh = hhs[i]

# If the household is unemployed, the firm hires it.

if hh.employer == None:

hh.employer = firm

firm.employees.append(hh)

i += 1

if i == len(hhs):

break

# Each household is assigned an old supplier.

random.shuffle(hhs)

i = 0

for hh in hhs:

if hh.old\_supplier == None:

hh.old\_supplier = firm

i += 1

if i == int(nhhs/ncfirms):

break

# If a household has no old supplier assigned, then it chooses one at random.

for hh in hhs:

if hh.old\_supplier == None:

random.shuffle(cfirms)

hh.old\_supplier = cfirms[0]

for firm in kfirms:

# Consumption firms hire workers.

random.shuffle(hhs)

i = 0

# There is a maximum amount of workers that can be hired.

while len(firm.employees) < parameters.param\_dict['Capital firms initial workers']/nkfirms:

hh = hhs[i]

# If the household is unemployed, the firm hires it.

if hh.employer == None:

hh.employer = firm

firm.employees.append(hh)

i += 1

if i == len(hhs):

break

# Each firm is assigned an old capital supplier.

random.shuffle(cfirms)

i = 0

for cfirm in cfirms:

if cfirm.old\_supplier == None:

cfirm.old\_supplier = firm

i += 1

if i == int(ncfirms/nkfirms):

break

# If a firm has no old supplier assigned, then it chooses one at random.

for cfirm in cfirms:

if cfirm.old\_supplier == None:

random.shuffle(kfirms)

cfirm.old\_supplier = kfirms[0]

for bank in banks:

# Each household is assigned a deposits bank.

random.shuffle(hhs)

i = 0

for hh in hhs:

if hh.bank\_deposits == None:

hh.bank\_deposits = bank

bank.deposits\_list.append(hh)

i += 1

if i == int(nhhs/nbanks):

break

# Each consumption firm is assigned a deposits bank.

random.shuffle(cfirms)

i = 0

for firm in cfirms:

if firm.bank\_deposits == None:

firm.bank\_deposits = bank

bank.deposits\_list.append(firm)

i += 1

if i == int(ncfirms/nbanks):

break

# Each capital firm is assigned a deposits bank.

random.shuffle(kfirms)

i = 0

for firm in kfirms:

if firm.bank\_deposits == None:

firm.bank\_deposits = bank

bank.deposits\_list.append(firm)

i += 1

if i == int(nkfirms/nbanks):

break

#[value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

for i in range(max\_loan\_age):

random.shuffle(cfirms)

j = 0

for firm in cfirms:

for element in firm.loan\_sheet:

if element[1] == i and element[4] == None:

bank.loan\_number += 1

element[4] = bank.ID + "\_" + str(bank.loan\_number)

ele = element[:]

element[5] = bank

ele[5] = firm

if element[1] == 0:

firm.old\_bank = bank

bank.loan\_sheet.append(ele)

j += 1

break

if j == int(ncfirms/nbanks):

break

random.shuffle(kfirms)

j = 0

for firm in kfirms:

for element in firm.loan\_sheet:

if element[1] == i and element[4] == None:

bank.loan\_number += 1

element[4] = bank.ID + "\_" + str(bank.loan\_number)

ele = element[:]

element[5] = bank

ele[5] = firm

if element[1] == 0:

firm.old\_bank = bank

bank.loan\_sheet.append(ele)

j += 1

break

if j == int(nkfirms/nbanks):

break

for hh in hhs:

if hh.bank\_deposits == None:

random.shuffle(banks)

bank = banks[0]

hh.bank\_deposits = bank

bank.deposits\_list.append(hh)

for firm in cfirms:

if firm.bank\_deposits == None:

random.shuffle(banks)

bank = banks[0]

firm.bank\_deposits = bank

bank.deposits\_list.append(firm)

for element in firm.loan\_sheet:

if element[4] == None:

random.shuffle(banks)

bank = banks[0]

bank.loan\_number += 1

element[4] = bank.ID + "\_" + str(bank.loan\_number)

ele = element[:]

element[5] = bank

ele[5] = firm

if element[1] == 0:

firm.old\_bank = bank

bank.loan\_sheet.append(ele)

for firm in kfirms:

if firm.bank\_deposits == None:

random.shuffle(banks)

bank = banks[0]

firm.bank\_deposits = bank

bank.deposits\_list.append(firm)

for element in firm.loan\_sheet:

if element[4] == None:

random.shuffle(banks)

bank = banks[0]

bank.loan\_number += 1

element[4] = bank.ID + "\_" + str(bank.loan\_number)

ele = element[:]

element[5] = bank

ele[5] = firm

if element[1] == 0:

firm.old\_bank = bank

bank.loan\_sheet.append(ele)

global price\_bonds

for bank in banks:

deposits = 0

for agent in bank.deposits\_list:

deposits += agent.deposits

total\_loan = 0

for element in bank.loan\_sheet:

total\_loan += element[3]

bank.reserves = bank.net\_worth - bank.bonds\*price\_bonds - total\_loan + deposits + bank.cash\_advances

bank.liquidity\_ratio = bank.reserves/deposits

bank.capital\_ratio = bank.net\_worth/total\_loan

random.shuffle(hhs)

i = 0

for hh in hhs:

if hh.employer == None:

hh.employer = gov

gov.employees.append(hh)

i += 1

if i == parameters.param\_dict['Number of public servants (constant)']:

gov.n\_employees = parameters.param\_dict['Number of public servants (constant)']

break

global unem; unem = []

for hh in hhs:

if hh.employer == None:

hh.periods\_unem\_list = [1,1,1,1]

hh.periods\_unem = 4

unem.append(hh)

else:

hh.periods\_unem\_list = [0,0,0,0]

hh.periods\_unem = 0

global u\_rate

u\_rate = len(unem)/nhhs

def plan():

global cfirms

global kfirms

global buffer; global max\_capital\_age; global capital\_productivity; global capital\_labour\_ratio; global FNmean; global FNsd

global gamma\_1; global gamma\_2; global profit\_rate\_normal; global capacity\_utilization\_normal

global labour\_productivity; global min\_capital\_units

global data

# Each firm plans its production level, price, labour and capital demand accordingly.

# Consumption firms' planning:

for firm in cfirms:

# If the firm has enough capital:

if firm.status != 'bankrupt type 2' and firm.capital\_stock > 0:

# If the firm intended to sell some units in the past iteration, the expected sales are updated. If the firm did not

# participate or intended to participate in the market, the expected sales are not updated.

if firm.sales + firm.inventories > 0:

# The expected sales (units of good) for the next period is calculated.

# Eq. 3.1

firm.exp\_sales = expectation(firm.sales,firm.exp\_sales)

# The firm's desired output is calculated such that it is enough to satisfy the expected demand

# (the expected sales), including a buffer or a precautionary amount of goods against unexpected

# demand swings.

# Eq.3.2

firm.output\_desired = max(0,firm.exp\_sales\*(1 + buffer) - firm.inventories)

# The firm calculates the total stock of non-expired capital it has. To do this, it updates its

# list of capital batches, eliminating those that have expired. The firm also plans ahead to replace the capital units that will

# become obsolete.

obsolete\_capital = 0

i = 0

while True:

try:

element = firm.capital\_sheet[i]

except IndexError:

break

else:

element[1] += 1

if element[1] > max\_capital\_age:

firm.capital\_sheet.remove(element)

else:

if element[1] == max\_capital\_age:

obsolete\_capital += element[0]

i += 1

# The total tock of capital units is computed.

total\_capital\_stock = 0

for element in firm.capital\_sheet:

total\_capital\_stock += element[0]

# The labour requirement is computed given a desired capital utilization rate.

# Eq. 3.5

firm.capacity\_utilization\_desired = min(1,firm.output\_desired/(total\_capital\_stock\*capital\_productivity))

# The desired amount of labour is calculated.

# Eq. 3.4

firm.n\_employees\_desired = round(firm.capacity\_utilization\_desired\*total\_capital\_stock/capital\_labour\_ratio)

# The planned output is calculated:

firm.planned\_output = min(firm.output\_desired, capital\_productivity\*capital\_labour\_ratio\*firm.n\_employees\_desired)

# To set its price, the firm must calculated the (expected) average wage.

# Eq. 3.6

agg\_wages = 0

if len(firm.employees) > 0:

for employee in firm.employees:

agg\_wages += employee.wage

firm.average\_wage = agg\_wages/len(firm.employees)

if firm.n\_employees\_desired <= len(firm.employees):

firm.exp\_average\_wage = firm.average\_wage

else:

firm.exp\_average\_wage = expectation(firm.average\_wage,firm.exp\_average\_wage)

# The price is also set with adaptive mark-up, which is updated according to the firm's sales performance.

# Eq. 3.7

if firm.sales > 0:

if firm.inventories/firm.sales > buffer:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 + abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

if firm.planned\_output > 0 and firm.exp\_average\_wage > 0 and firm.n\_employees\_desired > 0:

# The price is set as a mark-up over the expected unit variable cost.

# Eq. 3.6

firm.expected\_ULC = firm.exp\_average\_wage\*firm.n\_employees\_desired/firm.planned\_output

data[str(firm.ID)+'\_ULC\_expected'].append(firm.expected\_ULC)

firm.price = (1 + firm.mark\_up)\*firm.exp\_average\_wage\*firm.n\_employees\_desired/firm.planned\_output

elif firm.planned\_output + firm.inventories > 0:

# If the firm requires no output (no employees), it will only change its past price by an amount according to the firm's sales performance.

firm.price = ((1 + firm.mark\_up)/(1 + firm.mark\_up/var))\*firm.price

assert firm.price > 0

# The investment requirement is calculated according to a desired growth rate and in order to replace the (future) obsolete units of capital.

# Eq. 3.9

firm.growth\_rate\_desired = gamma\_1\*((firm.profit\_rate - profit\_rate\_normal)/profit\_rate\_normal) + gamma\_2\*((firm.capacity\_utilization\_desired - capacity\_utilization\_normal)/capacity\_utilization\_normal)

# The firm will only demand a non negative quantity of capital units.

firm.capital\_demand = max(0,firm.growth\_rate\_desired\*(total\_capital\_stock) + obsolete\_capital)

# If the firm is not bankrupt but does not have enough capital.

elif firm.status != 'bankrupt type 2' and firm.capital\_stock == 0:

# If the firm intended to sell some units in the past iteration, the expected sales are updated. If the firm did not

# participate or intended to participate in the market, the expected sales are not updated.

if firm.sales + firm.inventories > 0:

# The expected sales (units of good) for the next period is calculated.

# Eq. 3.1

firm.exp\_sales = expectation(firm.sales,firm.exp\_sales)

firm.output\_desired = 0

firm.planned\_output = 0

# The firm's capital demand is then defined as the minimum capital units needed.

firm.capital\_demand = min\_capital\_units

data[str(firm.ID)+'\_ULC\_expected'].append(None)

# If the inventories are positive, the price changes.

if firm.inventories > 0:

# The price is also set with adaptive mark-up, which is updated according to the firm's sales performance.

# Eq. 3.7

if firm.sales > 0:

if firm.inventories/firm.sales > buffer:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 + abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

# The price is updated.

firm.price = ((1 + firm.mark\_up)/(1 + firm.mark\_up/var))\*firm.price

assert firm.price > 0

# If the firm is bankrupt and does not have enough capital.

else:

# If the firm intended to sell some units in the past iteration, the expected sales are updated. If the firm did not

# participate or intended to participate in the market, the expected sales are not updated.

if firm.sales + firm.inventories > 0:

# The expected sales (units of good) for the next period is calculated.

# Eq. 3.1

firm.exp\_sales = expectation(firm.sales,firm.exp\_sales)

firm.output\_desired = 0

firm.planned\_output = 0

total\_capital\_stock = 0

obsolete\_capital = 0

# The firm calculates the total stock of non-expired capital it has. To do this, it updates its

# list of capital batches, eliminating those that have expired. The firm also plans ahead to replace the capital units that will

# become obsolete.

i = 0

while True:

try:

element = firm.capital\_sheet[i]

except IndexError:

break

else:

element[1] += 1

if element[1] > max\_capital\_age:

firm.capital\_sheet.remove(element)

else:

if element[1] == max\_capital\_age:

obsolete\_capital += element[0]

i += 1

for element in firm.capital\_sheet:

total\_capital\_stock += element[0]

# The firm's capital demand is then defined as what is needed to replace the obsolete capital and have at least the minimum capital units needed.

firm.capital\_demand = max(0,min\_capital\_units + obsolete\_capital - total\_capital\_stock)

data[str(firm.ID)+'\_ULC\_expected'].append(None)

# If the inventories are positive, the price changes.

if firm.inventories > 0:

# The price is also set with adaptive mark-up, which is updated according to the firm's sales performance.

# Eq. 3.7

if firm.sales > 0:

if firm.inventories/firm.sales > buffer:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 + abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

# The price is updated.

firm.price = ((1 + firm.mark\_up)/(1 + firm.mark\_up/var))\*firm.price

assert firm.price > 0

if firm.status != 'bankrupt type 2':

data[str(firm.ID)+'\_profit\_rate'].append(firm.profit\_rate)

data[str(firm.ID)+'\_growth\_rate\_desired'].append(firm.growth\_rate\_desired)

else:

data[str(firm.ID)+'\_profit\_rate'].append(None)

data[str(firm.ID)+'\_growth\_rate\_desired'].append(None)

# Capital firms' planning:

for firm in kfirms:

# The expected sales (units of good) for the next period is calculated.

# Eq. 3.1

firm.exp\_sales = expectation(firm.sales,firm.exp\_sales)

# The firm's desired output is calculated such that it is enough to satisfy the expected demand

# (the expected sales), including a buffer or a precautionary amount of goods against unexpected

# demand swings.

# Eq.3.2

firm.output\_desired = max(0,firm.exp\_sales\*(1 + buffer) - firm.inventories)

# The desired amount of labour is calculated.

# Eq. 3.3

firm.n\_employees\_desired = round(firm.output\_desired/labour\_productivity)

# To set its price, the firm must calculated the (expected) average wage.

# Eq. 3.6

agg\_wages = 0

if len(firm.employees) > 0:

for employee in firm.employees:

agg\_wages += employee.wage

firm.average\_wage = agg\_wages/len(firm.employees)

if firm.n\_employees\_desired <= len(firm.employees):

firm.exp\_average\_wage = firm.average\_wage

else:

firm.exp\_average\_wage = expectation(firm.average\_wage,firm.exp\_average\_wage)

# The price is also set with adaptive mark-up, which is updated according to the firm's sales performance.

# Eq. 3.7

if firm.sales > 0:

if firm.inventories/firm.sales > buffer:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 + abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

else:

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

firm.mark\_up = firm.mark\_up\*var

if firm.output\_desired > 0 and firm.exp\_average\_wage > 0 and firm.n\_employees\_desired > 0:

# The price is set as a mark-up over the expected unit variable cost.

# Eq. 3.6

firm.price = (1 + firm.mark\_up)\*firm.exp\_average\_wage\*firm.n\_employees\_desired/firm.output\_desired

else:

# If the firm requires no output (no employees), it will only change its past price by an amount according to the firm's sales performance.

firm.price = ((1 + firm.mark\_up)/(1 + firm.mark\_up/var))\*firm.price

assert firm.price > 0

# The preliminary inventories is the amount of planned supply, with which consumption firms

# and capital firms set their relationships in the first capital market interaction.

firm.preliminary\_inventories = firm.output\_desired + firm.inventories

def capital\_market\_1():

global cfirms

global kfirms

global x\_capital; global e\_capital; global div\_firm; global sigma; global tau\_pi; global max\_loan\_age

global Planned\_capital\_purchase; PCP = 0

# For it to be 'fair', consumption firms are chosen in a random order for choosing their

# capital supplier.

random.shuffle(cfirms)

for firm in cfirms:

# Only firms with a positive capital demand will participate in the market.

if firm.capital\_demand > 0:

# A list of suppliers is created:

firm.supplier\_list = []

# A list of potential suppliers (different to the firm's old supplier) is created.

opt = []

random.shuffle(kfirms)

x = x\_capital

for kfirm in kfirms:

if kfirm != firm.old\_supplier:

opt.append(kfirm)

if x == len(opt):

break

# A list is created to sort the potential suppliers according to their price.

options = []

for kfirm in opt:

options.append([kfirm.price,kfirm])

options.sort(reverse = True)

# The supplier with the lowest price is chosen.

new\_supplier\_list = options.pop()

new\_supplier = new\_supplier\_list[1]

# The supplier chosen must have a non negative quantity of inventories available.

while new\_supplier.preliminary\_inventories <= 0:

try:

new\_supplier\_list = options.pop()

except IndexError:

# If non of the potential supplier have a non negative quantity of inventories available,

# the firm will have no supplier.

new\_supplier = None

break

else:

new\_supplier = new\_supplier\_list[1]

# If the firm's old supplier has available (preliminary) inventories, the firm will consider it as a potential supplier.

if firm.old\_supplier.preliminary\_inventories > 0:

# If there are other potential suppliers, the firm will have to decide what firm will be its supplier.

if new\_supplier != None:

differential = (new\_supplier.price - firm.old\_supplier.price)/firm.old\_supplier.price

# If the potential supplier with the lowest price has a price lower than that of the firm's old supplier,

# the firm will decide to switch or not.

if differential < 0:

# The probability of switching is computed.

p\_switch = 1 - np.e\*\*(e\_capital\*differential)

# The firm decides to switch: if it chooses to do so, switch = 1, if not, switch = 0.

switch = np.random.choice([0,1], p = [1-p\_switch,p\_switch])

if switch == 1:

firm.supplier\_list.append([new\_supplier])

else:

firm.supplier\_list.append([firm.old\_supplier])

# If the old supplier has the lowest price, available, the firm will choose it as its capital supplier.

else:

firm.supplier\_list.append([firm.old\_supplier])

# If the old supplier is the only potential supplier, the firm will choose it as its capital supplier.

else:

firm.supplier\_list.append([firm.old\_supplier])

# If the old supplier has no invetories available, the firm will choose the potential supplier with the lowest

# price, in the case that it exists.

else:

if new\_supplier != None:

firm.supplier\_list.append([new\_supplier])

else:

firm.supplier\_list = None

if firm.supplier\_list != None:

# If the chosen supplier has not enough inventories to satisfy the firm's demand, the firm

# will choose other addictional suppliers if it can:

if firm.supplier\_list[0][0].preliminary\_inventories < firm.capital\_demand:

# The firm registers how many capital units the chosen supplier will provide.

firm.supplier\_list[0].append(firm.supplier\_list[0][0].preliminary\_inventories)

# The chosen supplier is left with no inventories.

firm.supplier\_list[0][0].preliminary\_inventories = 0

residual\_demand = firm.capital\_demand - firm.supplier\_list[0][0].preliminary\_inventories

# If the firm's old supplier was not chosen and has non negative available inventories, the firm will

# consider it as a potential supplier.

if firm.old\_supplier.preliminary\_inventories > 0 and firm.old\_supplier != firm.supplier\_list[0][0]:

options.append([firm.old\_supplier.price,firm.old\_supplier])

options.sort(reverse = True)

# The firm will choose the potential suppliers with the lowest prices until it satisfies its capital demand

# or until it runs out of options.

while residual\_demand > 0:

try:

new\_supplier\_list = options.pop()

except IndexError:

# If the firm runs out of options.

break

else:

new\_supplier = new\_supplier\_list[1]

# If the new supplier satifies the firm's demand.

if new\_supplier.preliminary\_inventories >= residual\_demand:

# The firm registers how many capital units the chosen supplier will provide.

firm.supplier\_list.append([new\_supplier,residual\_demand])

new\_supplier.preliminary\_inventories -= residual\_demand

residual\_demand = 0

# If the new supplier does not satisfy the firm's demand but has non negative inventories.

elif new\_supplier.preliminary\_inventories > 0:

# The firm registers how many capital units the chosen supplier will provide.

firm.supplier\_list.append([new\_supplier,new\_supplier.preliminary\_inventories])

# The chosen supplier is left with no inventories.

new\_supplier.preliminary\_inventories = 0

residual\_demand -= new\_supplier.preliminary\_inventories

# If the chosen supplier has enough inventories to satisfy the firm's demand, it will be the only supplier.

else:

firm.supplier\_list[0][0].preliminary\_inventories -= firm.capital\_demand

# The firm registers how many capital units the chosen supplier will provide.

firm.supplier\_list[0].append(firm.capital\_demand)

for firm in cfirms:

planned\_capital\_expense = 0

# Only firms with a positive capital demand will participate in the market.

if firm.capital\_demand > 0:

# If the firm was able to find and choose a capital supplier, the planned capital expense will be positive.

if firm.supplier\_list != None:

# The firm sets its chosen capital supplier to be its old supplier.

firm.old\_supplier = firm.supplier\_list[0][0]

# The firm calculates its planned capital expense to calculate its loan requirement.

for element in firm.supplier\_list:

planned\_capital\_expense += element[1]\*element[0].price

PCP += element[1]

# The firm calculates its expected revenue.

if firm.status != 'bankrupt type 2' and firm.capital\_stock > 0:

firm.exp\_revenue = min(firm.planned\_output + firm.inventories,firm.exp\_sales)\*firm.price

else:

firm.exp\_revenue = min(firm.inventories,firm.exp\_sales)\*firm.price

# The firm calculates its expected loan and interest repayment (not including the new loan).

loan\_interest\_repayment = 0

interest\_on\_loans = 0

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if period == 1:

if element[1] > 0:

loan\_interest\_repayment += (element[0]/max\_loan\_age + element[2]\*element[3])

interest\_on\_loans += element[2]\*element[3]

else:

assert element[1] > 0

loan\_interest\_repayment += (element[0]/max\_loan\_age + element[2]\*element[3])

interest\_on\_loans += element[2]\*element[3]

# The firm calculates its capital amortization.

amortization = 0

for element in firm.capital\_sheet:

assert element[1] > 0

amortization = element[0]\*element[2]/max\_capital\_age

# The firm calculates its expected profits and dividend disbursement to calculate its loan requirement.

if firm.planned\_output > 0:

exp\_unit\_cost = (firm.exp\_average\_wage\*firm.n\_employees\_desired + amortization)/firm.planned\_output

else:

exp\_unit\_cost = firm.unit\_cost\_present

firm.exp\_profit\_pre\_tax = firm.exp\_revenue + firm.interest\_on\_deposits + firm.inventories\*exp\_unit\_cost - firm.inventories\_past\*firm.unit\_cost\_present - firm.exp\_average\_wage\*firm.n\_employees\_desired - interest\_on\_loans - amortization

firm.exp\_tax\_payment = max(0,tau\_pi\*firm.exp\_profit\_pre\_tax)

firm.exp\_profit\_post\_tax = firm.exp\_profit\_pre\_tax - firm.exp\_tax\_payment

firm.exp\_dividend\_disbursement = max(0,div\_firm\*firm.exp\_profit\_post\_tax)

firm.exp\_OCF = firm.exp\_revenue + firm.interest\_on\_deposits - firm.exp\_average\_wage\*firm.n\_employees\_desired - loan\_interest\_repayment - firm.exp\_tax\_payment

# The firm calculates a non negative loan requirement to finance its investment.

# Eq. 3.11

firm.loan\_requirement = max(0, planned\_capital\_expense + firm.exp\_dividend\_disbursement + sigma\*firm.exp\_average\_wage\*firm.n\_employees\_desired - firm.exp\_OCF)

Planned\_capital\_purchase.append(PCP)

for firm in kfirms:

# The firm calculates its expected revenue.

firm.exp\_revenue = min(firm.output\_desired + firm.inventories,firm.exp\_sales)\*firm.price

# The firm calculates its expected loan and interest repayment (not including the new loan).

loan\_interest\_repayment = 0

interest\_on\_loans = 0

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if period == 1:

if element[1] > 0:

loan\_interest\_repayment += (element[0]/max\_loan\_age + element[2]\*element[3])

interest\_on\_loans += element[2]\*element[3]

else:

assert element[1] > 0

loan\_interest\_repayment += (element[0]/max\_loan\_age + element[2]\*element[3])

interest\_on\_loans += element[2]\*element[3]

# The firm calculates its expected profits and dividend disbursement to calculate its loan requirement.

if firm.output\_desired > 0:

exp\_unit\_cost = firm.exp\_average\_wage\*firm.n\_employees\_desired/firm.output\_desired

else:

exp\_unit\_cost = firm.unit\_cost\_present

firm.exp\_profit\_pre\_tax = firm.exp\_revenue + firm.interest\_on\_deposits + firm.inventories\*exp\_unit\_cost - firm.inventories\_past\*firm.unit\_cost\_present - firm.exp\_average\_wage\*firm.n\_employees\_desired - interest\_on\_loans

firm.exp\_tax\_payment = max(0,tau\_pi\*firm.exp\_profit\_pre\_tax)

firm.exp\_profit\_post\_tax = firm.exp\_profit\_pre\_tax - firm.exp\_tax\_payment

firm.exp\_dividend\_disbursement = max(0,div\_firm\*firm.exp\_profit\_post\_tax)

firm.exp\_OCF = firm.exp\_revenue + firm.interest\_on\_deposits - firm.exp\_average\_wage\*firm.n\_employees\_desired - loan\_interest\_repayment - firm.exp\_tax\_payment

# The firm calculates a non negative loan requirement.

firm.loan\_requirement = max(0, firm.exp\_dividend\_disbursement + sigma\*firm.exp\_average\_wage\*firm.n\_employees\_desired - firm.exp\_OCF)

def capital\_market\_2():

global cfirms

global kfirms

global sigma

global Capital\_sales; CS = 0

global Available\_deposits; AD = 0

# Accounting variables for revenues and sales are set to zero.

for firm in kfirms:

firm.revenue = 0

firm.sales = 0

# For the interaction to be 'fair', consumption firms will be selected at random to interact with suppliers.

random.shuffle(cfirms)

for firm in cfirms:

firm.capital\_bought = 0

if firm.supplier\_list != None:

available\_deposits = max(0, firm.deposits + firm.exp\_OCF)

AD += available\_deposits

for element in firm.supplier\_list:

# The real transaction takes place. The transaction is constrained by the available inventories of the supplier and the firm's available deposits.

units = min(element[1], available\_deposits/element[0].price, element[0].inventories)

# The transaction is valid only if there are units to be transacted.

if units > 0:

# The supplier's inventories decrease by the units sold.

element[0].inventories -= units

# The units sold are registered.

element[0].sales += units

CS += units

# The units bought are registered [number\_of\_k,age,price,value,supplier].

firm.capital\_sheet.append([units,0,element[0].price,units\*element[0].price,element[0]])

firm.capital\_bought += units

# The monetary transaction takes place.

# The firm's deposits (and firm's bank's reserves) decrease by the amount of expenditure.

firm.deposits -= units\*element[0].price

firm.bank\_deposits.reserves -= units\*element[0].price

# Available deposits are reduced by the amount of expenditure.

available\_deposits -= units\*element[0].price

# The supplier's deposits (and the supplier's bank' reserves) are increased by the amount of expenditure.

element[0].deposits += units\*element[0].price

element[0].bank\_deposits.reserves += units\*element[0].price

# The supplier registers its revenue.

element[0].revenue += units\*element[0].price

# If there are no available deposits left, the interaction is ended.

if available\_deposits == 0:

break

# The firm's supplier list is set to default.

firm.supplier\_list = None

Capital\_sales.append(CS)

Available\_deposits.append(AD)

def loanevaluation(firm,bank,amount):

global cfirms

global kfirms

global max\_loan\_age; global varsigma\_c; global iota; global varsigma\_k

# The first tranche of payment associated to the hypothetical loan is calculated.

debt\_service = (bank.i\_loans + (1/max\_loan\_age))\*amount

# The probability of default for the first period after the loan is granted is calculated.

# Eq. 3.13

if firm in cfirms:

p\_default = 1/(1 + np.e\*\*min(709,((firm.OCF - varsigma\_c\*debt\_service)/debt\_service)))

# The value of revenue after fire sale of capital is calculated.

discounted\_value = 0

for element in firm.capital\_sheet:

discounted\_value += (1-iota)\*element[3]

# The probability of default for the first period after the loan is granted is calculated.

# Eq. 3.13

if firm in kfirms:

p\_default = 1/(1 + np.e\*\*min(709,((firm.OCF - varsigma\_k\*debt\_service)/debt\_service)))

# Capital firms have no stock of capital (as collateral).

discounted\_value = 0

# The amount of total outstanding debt the firm currently has is calculated.

total\_outstanding\_debt = 0

if len(firm.loan\_sheet) > 0:

for element in firm.loan\_sheet:

total\_outstanding\_debt += element[3]

# The portion of the fire sales that will be distributed to the bank in the case of a default is calculated.

delta = min(1, discounted\_value/(total\_outstanding\_debt + amount))

# The expected return on the loan is then calculated.

exp\_return = p\_default\*(delta - 1)\*amount

for n in range(max\_loan\_age - 1):

interest\_payment = 0

for i in range(n + 1):

interest\_payment += bank.i\_loans\*amount\*(1 - (i/max\_loan\_age))

exp\_return += ((1 - p\_default)\*\*(n + 1))\*p\_default\*(interest\_payment + (delta - 1)\*(1 - ((n + 1)/max\_loan\_age))\*amount)

final\_payment = 0

for n in range(max\_loan\_age):

final\_payment += bank.i\_loans\*amount\*(1 - (n/max\_loan\_age))

exp\_return += ((1 - p\_default)\*\*(max\_loan\_age))\*final\_payment

# The expected return for the loan is returned.

return exp\_return

def credit\_market():

global cfirms

global kfirms

global banks

global min\_capital\_ratio; global price\_bonds ;global FNmean; global FNsd; global x\_credit; global e\_credit; global epsilon

# Banks update their interest rate on loans. To do this, the average interest rate and

# average capital ratio are calculated.

agg\_capital\_ratio = 0

agg\_i\_loans = 0

n\_banks = 0

for bank in banks:

if bank.capital\_ratio != 'no loans':

agg\_capital\_ratio += bank.capital\_ratio

n\_banks += 1

agg\_i\_loans += bank.i\_loans

if n\_banks > 0:

average\_capital\_ratio = agg\_capital\_ratio / n\_banks

else:

average\_capital\_ratio = None

average\_i\_loans = agg\_i\_loans / len(banks)

# Each bank updates its capital ratio.

for bank in banks:

# The bank calculates the total outstanding loans it has as assets.

total\_loan = 0

for element in bank.loan\_sheet:

total\_loan += element[3]

# The bank calculates the total deposits it has as liabilities.

deposits = 0

for agent in bank.deposits\_list:

deposits += agent.deposits

# The bank's net worth is calculated.

bank.net\_worth = bank.reserves + total\_loan + bank.bonds\*price\_bonds - deposits - bank.cash\_advances

# The bank's capital ratio is calculated.

if total\_loan > 0:

bank.capital\_ratio = bank.net\_worth/total\_loan

# In the case of there not being any loans as assets, the bank will lower its interest rate.

# For this, the bank is marked with the flag 'no loans'.

else:

bank.capital\_ratio = 'no loans'

# The bank has to comply with a minimum capital ratio. So, if the average capital ratio of the

# banking sector is higher than the minimum, it will decide to update its interest rate

# according to this average.

if average\_capital\_ratio != None:

if average\_capital\_ratio >= min\_capital\_ratio:

# If the bank has no loans as assets, it will try to attract borrowers by setting a lower interest rate.

if bank.capital\_ratio == 'no loans':

# The interest rate is set as an stochastic (positive) mark-up below the average interest rate of the sector.

# Eq. 3.12

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

bank.i\_loans = average\_i\_loans\*var

# The bank will decide to set an interest rate above the average if it is less capitalized

# than desired, this is, if it has a lower capital ratio than the average.

elif bank.capital\_ratio < average\_capital\_ratio:

# The interest rate is set as an stochastic mark-up over the average interest rate of the sector.

# Eq. 3.12

var = 1 + abs(np.random.normal(FNmean,FNsd))

bank.i\_loans = average\_i\_loans\*var

# The bank will decide to set an interest rate below the average if it is more (or equally) capitalized

# than desired, this is, if it has a higher (or equal) capital ratio than the average.

else:

# The interest rate is set as an stochastic (positive) mark-up below the average interest rate of the sector.

# Eq. 3.12

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

bank.i\_loans = average\_i\_loans\*var

# If the average capital ratio of the sector is lower than the legal minimum, the bank will decide to update its interest

# rate according to this minimum.

else:

# If the bank has no loans as assets, it will try to attract borrowers by setting a lower interest rate.

if bank.capital\_ratio == 'no loans':

# The interest rate is set as an stochastic (positive) mark-up below the average interest rate of the sector.

# Eq. 3.12

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

bank.i\_loans = average\_i\_loans\*var

# The bank will decide to set an interest rate above the average if it is less capitalized

# than desired, this is, if it has a lower capital ratio than the minimum.

elif bank.capital\_ratio < min\_capital\_ratio:

# The interest rate is set as an stochastic mark-up over the average interest rate of the sector.

# Eq. 3.12

var = 1 + abs(np.random.normal(FNmean,FNsd))

bank.i\_loans = average\_i\_loans\*var

# The bank will decide to set an interest rate below the average if it is more (or equally) capitalized

# than desired, this is, if it has a higher (or equal) capital ratio than the average.

else:

# The interest rate is set as an stochastic (positive) mark-up below the average interest rate of the sector.

# Eq. 3.12

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

bank.i\_loans = average\_i\_loans\*var

else:

# The interest rate is set as an stochastic (positive) mark-up below the average interest rate of the sector.

# Eq. 3.12

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

bank.i\_loans = average\_i\_loans\*var

# A list with firms with a positive loan requirement is created.

demand = []

for firm in cfirms:

firm.loan\_granted = 0

if firm.loan\_requirement > 0:

demand.append(firm)

for firm in kfirms:

firm.loan\_granted = 0

if firm.loan\_requirement > 0:

demand.append(firm)

# For the interaction to be 'fair', firms will be chosen in random order to interact with the banks.

random.shuffle(demand)

for firm in demand:

# A list of potential credit suppliers (different to the firm's old bank) is created.

opt = []

random.shuffle(banks)

x = x\_credit

for bank in banks:

if bank != firm.old\_bank:

opt.append(bank)

if x == len(opt):

break

# A list of the potential suppliers that are willing to accept the loan requirement is created.

accepted = []

amount = firm.loan\_requirement

for bank in opt:

if loanevaluation(firm, bank, amount) >= 0:

accepted.append([bank.i\_loans,bank])

# If the firm's old credit supplier (old bank) is willing to supply the credit, the firm will

# consider it as a potential supplier.

if loanevaluation(firm, firm.old\_bank, amount) >= 0:

# If at least another bank is also willing to supply the firm's credit requirement, the firm

# will have to decide which new supplier to choose.

if len(accepted) > 0:

# The new supplier will be either the firm's old supplier or the bank with the lowest

# interest rate.

accepted.sort(reverse = True)

new\_bank\_list = accepted.pop()

new\_bank = new\_bank\_list[1]

differential = (new\_bank.i\_loans - firm.old\_bank.i\_loans)/firm.old\_bank.i\_loans

# If the potential supplier with the lowest interest rate has an interest rate lower than that of the firm's old bank,

# the firm will decide whether to switch or not.

if differential < 0:

# The probability of switching is computed.

p\_switch = 1 - np.e\*\*(e\_credit\*differential)

# The firm decides to switch: if it chooses to do so, switch = 1, if not, switch = 0.

switch = np.random.choice([0,1], p = [1-p\_switch,p\_switch])

if switch == 0:

new\_bank = firm.old\_bank

else:

new\_bank = firm.old\_bank

else:

new\_bank = firm.old\_bank

# The amount granted as a loan will be equal to the firm's credit requirement.

new\_loan = firm.loan\_requirement

# The transaction is made:

# The bank and the firm register the new loan [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank] or [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,firm].

new\_bank.loan\_number += 1

loan\_ID = new\_bank.ID + "\_" + str(new\_bank.loan\_number)

firm.loan\_sheet.append([new\_loan,0,new\_bank.i\_loans,new\_loan,loan\_ID,new\_bank])

new\_bank.loan\_sheet.append([new\_loan,0,new\_bank.i\_loans,new\_loan,loan\_ID,firm])

# The firm's deposits increase by the amount granted.

firm.deposits += new\_loan

# The loan granted is registered.

firm.loan\_granted += new\_loan

# The firm's deposits bank increases its reserves by the amount granted.

firm.bank\_deposits.reserves += new\_loan

# The new credit supplier's reserves decrease by the amount granted.

new\_bank.reserves -= new\_loan

else:

# Out of the banks that are willing to supply the firm's credit requirement, the firm will choose the one with the lowest

# interest rate.

if len(accepted) > 0:

accepted.sort(reverse = True)

new\_bank\_list = accepted.pop()

new\_bank = new\_bank\_list[1]

# The amount granted as a loan will be equal to the firm's credit requirement.

new\_loan = firm.loan\_requirement

# The transaction is made:

# The bank and the firm register the new loan [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank] or [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,firm].

new\_bank.loan\_number += 1

loan\_ID = new\_bank.ID + "\_" + str(new\_bank.loan\_number)

firm.loan\_sheet.append([new\_loan,0,new\_bank.i\_loans,new\_loan,loan\_ID,new\_bank])

new\_bank.loan\_sheet.append([new\_loan,0,new\_bank.i\_loans,new\_loan,loan\_ID,firm])

# The firm's deposits increase by the amount granted.

firm.deposits += new\_loan

# The loan granted is registered.

firm.loan\_granted += new\_loan

# The firm's deposits bank increases its reserves by the amount granted.

firm.bank\_deposits.reserves += new\_loan

# The new credit supplier's reserves decrease by the amount granted.

new\_bank.reserves -= new\_loan

# If none of the banks are willing to supply the firm's credit requirement, the firm will ask for loans that the banks are

# willing to supply, starting with the bank with the lowest interest rate.

else:

opt.append(firm.old\_bank)

# A list of the potential banks with their associated interest rate is created.

options = []

for bank in opt:

options.append([bank.i\_loans,bank])

options.sort(reverse = False)

# The residual is the amount of the credit requirement that has not been granted yet.

residual = firm.loan\_requirement

for element in options:

bank = element[1]

# If the bank is willing to supply the residual of the credit demand

try:

loanevaluation(firm, bank, residual)

except OverflowError:

pass

else:

if residual > 0 and loanevaluation(firm, bank, residual) >= 0:

new\_loan = residual

residual = 0

# The transaction is made:

# The bank and the firm register the new loan [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank] or [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,firm].

bank.loan\_number += 1

loan\_ID = bank.ID + "\_" + str(bank.loan\_number)

firm.loan\_sheet.append([new\_loan,0,bank.i\_loans,new\_loan,loan\_ID,bank])

bank.loan\_sheet.append([new\_loan,0,bank.i\_loans,new\_loan,loan\_ID,firm])

# The firm's deposits increase by the amount granted.

firm.deposits += new\_loan

# The loan granted is registered.

firm.loan\_granted += new\_loan

# The firm's deposits bank increases its reserves by the amount granted.

firm.bank\_deposits.reserves += new\_loan

# The new credit supplier's reserves decrease by the amount granted.

bank.reserves -= new\_loan

break

# The maximum loan thet the bank is willing to supply is calculated with bisection search.

else:

high = residual

low = 0

while True:

amount = (high + low)/2

try:

eps = loanevaluation(firm, bank, amount)

except ZeroDivisionError:

break

else:

if eps > epsilon:

low = amount

elif eps < 0:

high = amount

else:

if amount > 0:

residual -= amount

new\_loan = amount

# The transaction is made:

# The bank and the firm register the new loan [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank] or [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,firm].

bank.loan\_number += 1

loan\_ID = bank.ID + "\_" + str(bank.loan\_number)

firm.loan\_sheet.append([new\_loan,0,bank.i\_loans,new\_loan,loan\_ID,bank])

bank.loan\_sheet.append([new\_loan,0,bank.i\_loans,new\_loan,loan\_ID,firm])

# The firm's deposits increase by the amount granted.

firm.deposits += new\_loan

# The loan granted is registered.

firm.loan\_granted += new\_loan

# The firm's deposits bank increases its reserves by the amount granted.

firm.bank\_deposits.reserves += new\_loan

# The new credit supplier's reserves decrease by the amount granted.

bank.reserves -= new\_loan

break

# If the amount that the bank is willing to give is non-positive, no transaction is made.

else:

break

if residual <= 0:

break

def labour\_market():

global hhs

global cfirms

global kfirms

global gov

global unem

global u\_rate; global nhhs; global FNmean; global FNsd; global max\_u\_rate

global turnover; global x\_labour

# Each household updates its wage.

for hh in hhs:

# If the household is currently unemployed and has been so in more than two out of the four most recent iterations,

# it will lower its reservation wage.

# Eq. 15

if hh.employer == None and hh.periods\_unem > 2:

# The wage is lowered by an stochastic (positive) fraction.

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

hh.wage = hh.wage\*var

# If the household is currently employed, has been so in at least two out of the four most recent iterations

# and the unemployment rate is low enough, it will lower its reservation wage.

# Eq. 15

if hh.employer != None and hh.periods\_unem <= 2 and u\_rate <= max\_u\_rate:

# The wage is increased by an stochastic fraction.

var = 1 + abs(np.random.normal(FNmean,FNsd))

hh.wage = hh.wage\*var

# If the wage is, indeed, increased, the household will become unemployed.

if var > 1:

# The household is removed from its employer's list of employees.

hh.employer.employees.remove(hh)

# The household no longer has an employer.

hh.employer = None

unem.append(hh)

# A list with all consumption firms and

demand = []

for firm in cfirms:

# Only firms with enough capital can hire labour.

if firm.status != 'bankrupt type 2':

demand.append(firm)

for firm in kfirms:

demand.append(firm)

# For the interactions to be 'fair', firms will be chosen in a random order to interact with unemployed households.

random.shuffle(demand)

# Each firm plans its labour demand.

for firm in demand:

# A random turnover takes place.

n\_turnover = round(turnover\*len(firm.employees))

random.shuffle(firm.employees)

for i in range(n\_turnover):

hh = firm.employees.pop()

# The household becomes unemployed.

hh.employer = None

unem.append(hh)

# The firm will set its labour requirement.

# In the case of there being employees in excess, it will ramdomly fire some of the employees.

if len(firm.employees) > firm.n\_employees\_desired:

random.shuffle(firm.employees)

while len(firm.employees) > firm.n\_employees\_desired:

hh = firm.employees.pop()

# The household becomes unemployed.

hh.employer = None

unem.append(hh)

# The firm has no positive labour requirement.

firm.labour\_demand = 0

# In the case of there being exactly the amount of employees desired, the firm

# will not have a positive labour requirement.

elif len(firm.employees) == firm.n\_employees\_desired:

firm.labour\_demand = 0

# In the case of there being not enough employees, the firm will have a

# positive labour requirement.

else:

firm.labour\_demand = firm.n\_employees\_desired - len(firm.employees)

# Market interaction between firms and unemployed households:

for firm in demand:

if firm.labour\_demand > 0:

# For each employee desired, the firm can select from a subset of unemployed households.

for j in range(firm.labour\_demand):

options = []

random.shuffle(unem)

for i in range(min(len(unem),x\_labour)):

options.append([unem[i].wage,unem[i]])

# The firm hires the household with the lowest reservation wage.

if len(options) > 0:

options.sort(reverse = True)

hhL = options.pop()

hh = hhL[1]

# The household is hired.

firm.employees.append(hh)

# The household becomes employed.

hh.employer = firm

unem.remove(hh)

# The government sets its labour demand.

random.shuffle(gov.employees)

n\_turnover = round(turnover\*len(gov.employees))

for i in range(n\_turnover):

hh = gov.employees.pop()

# The household becomes unemployed.

hh.employer = None

unem.append(hh)

# If there are public servants in excess, the government will randomly fire them.

while len(gov.employees) > gov.n\_employees:

hh = gov.employees.pop()

# The household becomes unemployed.

hh.employer = None

unem.append(hh)

# If there are not enough public servants, the government will have a positive labour demand.

if len(gov.employees) < gov.n\_employees:

labour\_demand = gov.n\_employees - len(gov.employees)

# For each employee desired, the government can select from a subset of unemployed households.

for j in range(labour\_demand):

options = []

random.shuffle(unem)

for i in range(min(len(unem),x\_labour)):

options.append([unem[i].wage,unem[i]])

# The government hires the household with the lowest reservation wage.

if len(options) > 0:

options.sort(reverse = True)

hhL = options.pop()

hh = hhL[1]

# The household is hired.

gov.employees.append(hh)

# The household becomes employed.

hh.employer = gov

unem.remove(hh)

# Households update their labour status.

for hh in hhs:

# Unemployed households will register another period being unemployed.

if hh.employer == None:

hh.periods\_unem\_list.pop(0)

hh.periods\_unem\_list.append(1)

# Employed households will register another period being employed.

else:

hh.periods\_unem\_list.pop(0)

hh.periods\_unem\_list.append(0)

# The amount of periods unemployed (out of the last four iterations) is updated.

hh.periods\_unem = sum(hh.periods\_unem\_list)

def production():

global cfirms

global kfirms

global capital\_productivity; global capital\_labour\_ratio; global max\_capital\_age; global labour\_productivity

global K\_inventories; k\_inv = 0

for firm in cfirms:

# Only firms with enough capital can produce.

if firm.status != 'bankrupt type 2' and firm.capital\_stock > 0:

# The firm produces according to a labour-constrained function.

firm.output = min(firm.planned\_output, capital\_productivity\*capital\_labour\_ratio\*len(firm.employees))

# Firms that are bankrupt and don't have enough capital, do not produce.

else:

firm.output = 0

# The firm registers its past inventories.

firm.inventories\_past = firm.inventories

# The firm registers its new total inventories.

firm.inventories = firm.output + firm.inventories\_past

# The value of capital units is updated (the past capital value is calculated for profit rate calculation)

firm.past\_capital\_value = 0

for element in firm.capital\_sheet: #[number\_of\_k,age,price,value,supplier]

firm.past\_capital\_value += element[3]

element[3] = element[0]\*element[2]\*(1 - (element[1]/max\_capital\_age))

for firm in kfirms:

# The firm produces according to a labour-constrained function.

firm.output = min(firm.output\_desired, len(firm.employees)\*labour\_productivity)

# The firm registers its past inventories.

firm.inventories\_past = firm.inventories

# The firm registers its new total inventories.

firm.inventories = firm.output + firm.inventories\_past

k\_inv += firm.inventories

K\_inventories.append(k\_inv)

def consumption\_market():

global hhs

global cfirms

global x\_consumption; global e\_consumption; global Y\_cons; global Y\_cons\_r

# The nominal consumption output is set to zero.

Y\_cons = 0

# The real consumption output is set to zero.

Y\_cons\_r = 0

# The accounting variables for revenues and sales are set to zero.

for firm in cfirms:

firm.revenue = 0

firm.sales = 0

# For the interaction to be fair, households will be chosen at random to interact with firms.

random.shuffle(hhs)

for hh in hhs:

# A list of potential suppliers is created.

opt = []

random.shuffle(cfirms)

x = x\_consumption

for firm in cfirms:

if firm != hh.old\_supplier:

opt.append(firm)

if x == len(opt):

break

# The price charged by each potential supplier is registered.

options = []

for firm in opt:

options.append([firm.price,firm])

# The supplier with the lowest price and available inventories is chosen as an option.

options.sort(reverse = True)

q = 0

while q == 0:

new\_supplier\_list = options.pop()

new\_supplier = new\_supplier\_list[1]

q = new\_supplier.inventories

if len(options) == 0:

if q == 0:

new\_supplier = None

break

# If the household's last supplier has available inventories, it consider it as a potential supplier.

if hh.old\_supplier.inventories > 0:

# If there are other options apart from the old supplier.

if new\_supplier != None:

differential = (new\_supplier.price - hh.old\_supplier.price)/hh.old\_supplier.price

# If the potential supplier charges a lower price than the old supplier, the household must decide to switch or not.

if differential < 0:

p\_switch = 1 - np.e\*\*(e\_consumption\*differential)

switch = np.random.choice([0,1], p = [1-p\_switch,p\_switch])

# The supplier is registered.

if switch == 1:

supplier = new\_supplier

else:

# The supplier is registered.

supplier = hh.old\_supplier

else:

# The supplier is registered.

supplier = hh.old\_supplier

else:

# The supplier is registered.

supplier = hh.old\_supplier

# If the old supplier has no inventories left.

else:

# If there exists a potential supplier.

if new\_supplier != None:

# The supplier and its price are registered.

supplier = new\_supplier

else:

# There is no supplier.

supplier = None

# If a supplier exists, a transaction will be done.

if supplier != None:

# The last price is registered (for consumption planning in next iteration).

hh.last\_price = supplier.price

consumption = min(hh.desired\_consumption, hh.deposits/supplier.price, supplier.inventories)

# The transaction is made.

# The supplier's inventories decrease by the amount sold.

supplier.inventories -= consumption

# The sale is registered.

supplier.sales += consumption

# The household's deposits (and household's bank's reserves) decrease by the expenditure.

hh.deposits -= consumption\*supplier.price

hh.bank\_deposits.reserves -= consumption\*supplier.price

# The supplier's deposits (and supplier's bank's reserves) increase by the expenditure.

supplier.deposits += consumption\*supplier.price

supplier.bank\_deposits.reserves += consumption\*supplier.price

# The revenue is registered.

supplier.revenue += consumption\*supplier.price

# The purchase is registered.

Y\_cons += consumption\*supplier.price

Y\_cons\_r += consumption

# If the household's demand is not satisfied, the household may look for additional suppliers.

residual = hh.desired\_consumption - consumption

# If the household's demand is not satified and it can afford to buy more goods.

if residual > 0 and hh.deposits > 0:

# If the household's old supplier is not this iteration's supplier, then the household will consider

# it as a potential supplier if it has available inventories left.

if hh.old\_supplier != supplier and hh.old\_supplier.inventories > 0:

options.append([hh.old\_supplier.price,hh.old\_supplier])

# The potential suppliers are sorted according to their price.

options.sort(reverse = False)

for firm\_list in options:

firm = firm\_list[1]

consumption = min(residual, hh.deposits/firm.price, firm.inventories)

if consumption > 0:

residual -= consumption

# The transaction is made.

# The supplier's inventories decrease by the amount sold.

firm.inventories -= consumption

# The sale is registered.

firm.sales += consumption

# The household's deposits (and household's bank's reserves) decrease by the expenditure.

hh.deposits -= consumption\*firm.price

hh.bank\_deposits.reserves -= consumption\*firm.price

# The supplier's deposits (and supplier's bank's reserves) increase by the expenditure.

firm.deposits += consumption\*firm.price

firm.bank\_deposits.reserves += consumption\*firm.price

# The revenue is registered.

firm.revenue += consumption\*firm.price

# The purchase is registered.

Y\_cons += consumption\*firm.price

Y\_cons\_r += consumption

if hh.deposits == 0 or residual == 0:

break

# The 'new' old supplier is registered.

hh.old\_supplier = supplier

def deposits\_payment():

global banks

global hhs

# Each household's income is set to zero.

for hh in hhs:

hh.income = 0

for bank in banks:

for agent in bank.deposits\_list:

agent.deposits += agent.interest\_on\_deposits

if agent in hhs:

agent.income += agent.interest\_on\_deposits

def deposits\_market():

global banks

global hhs

global cfirms

global kfirms

global average\_liquidity\_ratio

global FNmean; global FNsd; global x\_deposits; global e\_deposits

# The sector's average liquidity ratio and interest rate on deposits are calculated.

agg\_liquidity\_ratio = 0

agg\_i\_deposits = 0

n\_banks = 0

for bank in banks:

agg\_i\_deposits += bank.i\_deposits

if bank.liquidity\_ratio != 'no deposits':

agg\_liquidity\_ratio += bank.liquidity\_ratio

n\_banks += 1

average\_liquidity\_ratio = agg\_liquidity\_ratio/n\_banks

average\_i\_deposits = agg\_i\_deposits/len(banks)

# Each bank updates its liquidity ratio.

for bank in banks:

# The bank calculates the total deposits it has as liabilities.

deposits = 0

for agent in bank.deposits\_list:

deposits += agent.deposits

# The bank's liquidity ratio is calculated.

if deposits > 0:

bank.liquidity\_ratio = bank.reserves/deposits

# In the case of there not being any deposits as liabilities, the bank will increase its interest rate.

# For this, the bank is marked with the flag 'no deposits'.

else:

bank.liquidity\_ratio = 'no deposits'

# If the bank has no deposits as liabilities, it will try to attract depositors by setting a higher interest rate.

if bank.liquidity\_ratio == 'no deposits':

# The interest rate is set as an stochastic (positive) mark-up over the average interest rate of the sector.

# Eq. 3.14

var = 1 + abs(np.random.normal(FNmean,FNsd))

bank.i\_deposits = average\_i\_deposits\*var

# The bank will decide to set an interest rate above the average if it is has less liquidity

# than desired, this is, if it has a lower liquidity ratio than the average.

elif bank.liquidity\_ratio < average\_liquidity\_ratio:

# The interest rate is set as an stochastic mark-up over the average interest rate of the sector.

# Eq. 3.14

var = 1 + abs(np.random.normal(FNmean,FNsd))

bank.i\_deposits = average\_i\_deposits\*var

# The bank will decide to set an interest rate below the average if it has more (or equal) liquidity

# than desired, this is, if it has a higher (or equal) liquidity ratio than the average.

else:

# The interest rate is set as an stochastic (positive) mark-up below the average interest rate of the sector.

# Eq. 3.14

var = 1 - abs(np.random.normal(FNmean,FNsd))

while var <= 0:

var = 1 - abs(np.random.normal(FNmean,FNsd))

bank.i\_deposits = average\_i\_deposits\*var

# A list of demand agents is created.

demand = []

for hh in hhs:

demand.append(hh)

for firm in cfirms:

demand.append(firm)

for firm in kfirms:

demand.append(firm)

# For the interaction to be 'fair', the demand agents will be selected at random to interact with the banks.

random.shuffle(demand)

for agent in demand:

# A list of potential banks is created.

opt = []

random.shuffle(banks)

x = x\_deposits

for bank in banks:

if bank != agent.bank\_deposits:

opt.append(bank)

if x == len(opt):

break

# The potential banks are ordered according to their interest rate.

options = []

for bank in opt:

options.append([bank.i\_deposits,bank])

options.sort(reverse = False)

# The bank with the highest interest rate is selected.

new\_bank\_list = options.pop()

new\_bank = new\_bank\_list[1]

# The agent must decide to switch or not if the potential bank has a higher interest rate.

differential = (new\_bank.i\_deposits - agent.bank\_deposits.i\_deposits)/agent.bank\_deposits.i\_deposits

if differential > 0:

# The probability of switching is calculated.

p\_switch = 1 - np.e\*\*(-1\*e\_deposits\*differential)

# The agent decides to switch or not.

switch = np.random.choice([0,1], p = [1-p\_switch,p\_switch])

# If the agent decides to switch, a transfer of deposits is made.

if switch == 1:

# The agent is removed from its old bank's deposits list.

agent.bank\_deposits.deposits\_list.remove(agent)

# The old bank's deposits decrease by the amount transfered.

agent.bank\_deposits.reserves -= agent.deposits

# The new bank is set as the agent's bank.

agent.bank\_deposits = new\_bank

# The agent is added to its new bank's deposits list.

agent.bank\_deposits.deposits\_list.append(agent)

# The new bank's deposits increase by the amount transfered.

agent.bank\_deposits.reserves += agent.deposits

# The agent's interest payment is calculated.

agent.interest\_on\_deposits = agent.bank\_deposits.i\_deposits\*agent.deposits

def cfirm\_acc():

global cfirms

global hhs

global banks

global gov

global unem

global max\_capital\_age; global tau\_pi; global div\_firm; global max\_loan\_age; global iota; global min\_capital\_units; global bankruptcy\_buffer

global data

# Each bank's interest payment on loans is set to zero.

for bank in banks:

bank.interest\_payment = 0

# The government's tax, bond, cash advances and dole payments are set to zero.

gov.tax\_payment = 0

gov.wage\_payment = 0

gov.bond\_payment = 0

gov.cash\_advances\_payment = 0

gov.dole\_payment = 0

# Firm accounting.

for firm in cfirms:

# The firm calculates its wage disbursement (for accounting).

agg\_wages = 0

for employee in firm.employees:

agg\_wages += employee.wage

# The firm calculates its capital amortization:

amortization = 0

for element in firm.capital\_sheet: # [number\_of\_k,age,price,value,supplier]

if element[1] > 0:

amortization = element[0]\*element[2]/max\_capital\_age

# The firm calculates its interest payment on loans (for accounting).

interests\_on\_loans = 0

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if element[1] > 0:

interests\_on\_loans += element[2]\*element[3]

# The firm updates and calculates its unit costs and nominal variations of inventories.

firm.unit\_cost\_past = firm.unit\_cost\_present

# Only firms that have produced can update their unit costs.

if firm.output > 0:

firm.unit\_cost\_present = (agg\_wages + amortization)/firm.output

firm.unit\_labour\_cost = agg\_wages/firm.output

else:

firm.unit\_labour\_cost = None

nominal\_variation\_inventories = firm.inventories\*firm.unit\_cost\_present - firm.inventories\_past\*firm.unit\_cost\_past

# The firm calculates its profits before tax payment.

# Eq. 3.8

firm.profit\_pre\_tax = firm.revenue + firm.interest\_on\_deposits + nominal\_variation\_inventories - agg\_wages - amortization - interests\_on\_loans

# The firm calculates its tax payment (for accounting).

firm.tax = max(0, firm.profit\_pre\_tax\*tau\_pi)

# The firm calculates its profits after tax payment (for accounting).

firm.profit\_post\_tax = firm.profit\_pre\_tax - firm.tax

# The firm calculates its dividend disbursement.

firm.dividend\_disbursement = max(0, div\_firm\*firm.profit\_post\_tax)

# The firm calculates its loan repayment (for accounting):

loan\_repayment = 0

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if element[1] > 0:

loan\_repayment += element[0]/max\_loan\_age

# The firm calculates its Operating Cash Flow.

firm.OCF = firm.revenue + firm.interest\_on\_deposits - agg\_wages - interests\_on\_loans - loan\_repayment - firm.tax

data[str(firm.ID)+'\_total\_loan\_repayment'].append(interests\_on\_loans + loan\_repayment)

if firm.past\_capital\_value > 0:

# The firm updates its profit rate.

firm.profit\_rate = firm.OCF/firm.past\_capital\_value

total\_cost = agg\_wages + interests\_on\_loans + loan\_repayment

if total\_cost > 0:

data[str(firm.ID)+'\_%L\_of\_costs'].append(agg\_wages/total\_cost)

else:

data[str(firm.ID)+'\_%L\_of\_costs'].append(None)

data[str(firm.ID)+'\_agg\_wages'].append(agg\_wages)

# If deposits are enough to pay wages.

if firm.deposits >= agg\_wages or agg\_wages == 0:

# Wages are paid.

for employee in firm.employees:

# The firm's deposits (and firm's bank's reserves) decrease by the wage paid.

firm.deposits -= employee.wage

firm.bank\_deposits.reserves -= employee.wage

# The employee's deposits (and employee's bank's reserves) increse by the wage paid.

employee.deposits += employee.wage

employee.bank\_deposits.reserves += employee.wage

# The employee registers the wage payment as income.

employee.income += employee.wage

# If deposits are not enough to pay all wages.

else:

# The firm's status is updated.

firm.status = 'bankrupt'

amount\_per\_unit = firm.deposits/agg\_wages

for employee in firm.employees:

# The firm's deposits (and firm's bank's reserves) decrease by the amount paid, which is proportional to the employee's wage.

amount = employee.wage\*amount\_per\_unit

firm.deposits -= amount

firm.bank\_deposits.reserves -= amount

# The employee's deposits (and employee's bank's reserves) increse by the amount paid.

employee.deposits += amount

employee.bank\_deposits.reserves += amount

# The employee registers the wage payment as income.

employee.income += amount

# The employee becomes unemployed

employee.employer = None

employee.periods\_unem\_list[3] = 1

employee.periods\_unem = sum(employee.periods\_unem\_list)

unem.append(employee)

# All employees are fired

firm.employees = []

# If deposits are enough to pay loan and interest repayments.

if firm.deposits >= interests\_on\_loans + loan\_repayment and firm.status != 'bankrupt':

# Loan responsibilities are paid.

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if element[1] > 0:

# The firm's deposits (and firm's bank's reserves) decrease by the amount paid.

firm.deposits -= (element[2]\*element[3] + element[0]/max\_loan\_age)

firm.bank\_deposits.reserves -= (element[2]\*element[3] + element[0]/max\_loan\_age)

# The lender's reserves increase by the amount paid.

element[5].reserves += (element[2]\*element[3] + element[0]/max\_loan\_age)

element[5].interest\_payment += element[2]\*element[3]

# The outstanding debt in loan sheets are updated:

element[3] -= element[0]/max\_loan\_age

for ele in element[5].loan\_sheet:

# The bank recognizes the loan.

if ele[4] == element[4]:

# The outstanding loan is updated.

ele[3] = element[3]

# The age of the loan (in the bank's loan sheet) is updated.

ele[1] += 1

# If the loan reaches maturity, the loan is removed from the bank's loan sheet.

if ele[3] == 0:

element[5].loan\_sheet.remove(ele)

break

# The age of the loan (in the firm's loan sheet) is updated.

i = 0

while True:

try:

element = firm.loan\_sheet[i]

except IndexError:

break

else:

element[1] += 1

# If the loan reaches maturity, the loan is removed from the firm's loan sheet.

if element[3] == 0:

firm.loan\_sheet.remove(element)

else:

i += 1

# If deposits are not enough to pay loan and interest repayments.

else:

# The firm's status is updated.

firm.status = 'bankrupt'

for employee in firm.employees:

# The employee becomes unemployed

employee.employer = None

employee.periods\_unem\_list[3] = 1

employee.periods\_unem = sum(employee.periods\_unem\_list)

unem.append(employee)

# All employees are fired

firm.employees = []

payment\_list = []

for element in firm.loan\_sheet:

if element[1] > 0:

payment\_list.append([element[2]\*element[3] + element[0]/max\_loan\_age, element])

payment\_list.sort(reverse = False)

for p\_list in payment\_list:

if firm.deposits >= p\_list[0]:

# The firm's deposits (and the firm's bank's reserves) decrease by the amount paid.

firm.deposits -= p\_list[0]

firm.bank\_deposits.reserves -= p\_list[0]

# The lender's reserves increase by the amount paid.

p\_list[1][5].reserves += p\_list[0]

p\_list[1][5].interest\_payment += p\_list[1][2]\*p\_list[1][3]

# The outstanding debt in loan sheets are updated:

p\_list[1][3] -= element[0]/max\_loan\_age

for ele in p\_list[1][5].loan\_sheet:

# The bank recognizes the loan.

if ele[4] == p\_list[1][4]:

# The outstanding loan is updated.

ele[3] = p\_list[1][3]

# The age of the loan (in the bank's loan sheet) is updated.

ele[1] += 1

# If the loan reaches maturity, the loan is removed from the bank's loan sheet.

if ele[3] == 0:

p\_list[1][5].loan\_sheet.remove(ele)

break

# The age of the loan (in the firm's loan sheet) is updated.

i = 0

while True:

try:

element = firm.loan\_sheet[i]

except IndexError:

break

else:

if element[1] == 0:

for ele in element[5].loan\_sheet:

if ele[4] == element[4]:

# The age of the loan (in the bank's loan sheet) is updated.

ele[1] += 1

element[1] += 1

# If the loan reaches maturity, the loan is removed from the firm's loan sheet.

if element[3] == 0:

firm.loan\_sheet.remove(element)

else:

i += 1

# The status of the firm, if not bankrupt, is updated:

if firm.status != 'bankrupt':

firm.status = None

# If deposits are enough to pay for taxes, taxes are paid.

if firm.deposits >= firm.tax and firm.status != 'bankrupt':

# The firm's deposits (and the firm's bank's reserves) decrease by the amount paid.

firm.deposits -= firm.tax

firm.bank\_deposits.reserves -= firm.tax

# The government's reserves increase by the amount paid.

gov.reserves += firm.tax

gov.tax\_payment += firm.tax

# If deposits are enough to pay the dividend disbursement.

if firm.deposits >= firm.dividend\_disbursement and firm.status != 'bankrupt':

# The firm's deposits (and the firm's bank's reserves) decrease by the amount paid.

firm.deposits -= firm.dividend\_disbursement

firm.bank\_deposits.reserves -= firm.dividend\_disbursement

# For disbursement, the aggregate wealth of households is calculated.

agg\_hh\_wealth = 0

for hh in hhs:

agg\_hh\_wealth += hh.deposits

for hh in hhs:

# The household's deposits (and household's bank's reserves) increase by the amount paid, which is registered as income.

amount = firm.dividend\_disbursement\*hh.deposits/agg\_hh\_wealth

hh.bank\_deposits.reserves += amount

hh.income += amount

hh.deposits += amount

# Procedure for bankrupt firm.

if firm.status == 'bankrupt':

while True:

try:

element = firm.capital\_sheet.pop() # [number\_of\_k,age,price,value,supplier]

except IndexError:

break

else:

# The aggregate outstanding debt is calculated for fire sale.

agg\_outstanding\_debt = 0

for loan in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

agg\_outstanding\_debt += loan[3]

# If no outstanding debt is left to pay, the loop ends.

if agg\_outstanding\_debt == 0:

break

# If some outstanding debt is left to pay.

else:

# The aggregate wealth of households is calculated for fire sale.

agg\_hh\_wealth = 0

for hh in hhs:

agg\_hh\_wealth += hh.deposits

# The fire sale of the capital batch is made.

for hh in hhs:

amount = min(hh.deposits, iota\*element[3]\*hh.deposits/agg\_hh\_wealth)

# The household's deposits (and household's bank's reserves) decrease by the amount paid.

hh.deposits -= amount

hh.bank\_deposits.reserves -= amount

# The firm's deposits (and firm's bank's reserves) increase by the amount paid.

firm.deposits += amount

firm.bank\_deposits.reserves += amount

# The loan payment is made.

amount\_per\_unit = firm.deposits/agg\_outstanding\_debt

for loan in firm.loan\_sheet:

amount = min(loan[3], loan[3]\*amount\_per\_unit)

# The firm updates the outstanding debt.

loan[3] -= amount

# The firm's deposits (and firm's bank's reserves) decrease by the amount paid.

firm.deposits -= amount

firm.bank\_deposits.reserves -= amount

# The lender's reserves increase by the amount paid.

loan[5].reserves += amount

for ele in loan[5].loan\_sheet:

# The lender recognizes the loan.

if ele[4] == loan[4]:

# The lender updates the outstading loan.

ele[3] = loan[3]

# If the loan is repaid, it is removed from the lender's loan sheet.

if ele[3] == 0:

loan[5].loan\_sheet.remove(ele)

break

# The firm eliminates all loan responsibilities

for loan in firm.loan\_sheet:

for ele in loan[5].loan\_sheet:

# The lender recognizes the loan.

if ele[4] == loan[4]:

# The loan is removed from the lender's loan sheet.

loan[5].loan\_sheet.remove(ele)

break

# All loans are removed from the firm's loan sheet.

firm.loan\_sheet = []

# The total amount of non-obsolete capital stock is calculated

capital\_stock = 0

for element in firm.capital\_sheet:

if element[1] < max\_capital\_age:

capital\_stock += element[0]

# If there are enough capital units.

if capital\_stock >= min\_capital\_units:

firm.status = 'bankrupt type 1'

print('bankrupt type 1'+';'+str(period))

# If there are not enough capital units.

else:

firm.status = 'bankrupt type 2'

print('bankrupt type 2'+';'+str(period))

# The total amount of non-obsolete capital stock is calculated.

capital\_stock = 0

for element in firm.capital\_sheet:

if element[1] < max\_capital\_age - 1:

capital\_stock += element[0]

capital\_required = min\_capital\_units - capital\_stock

# The firm determines the expected OCF.

firm.exp\_OCF = expectation(firm.OCF, firm.exp\_OCF)

resources\_required = firm.old\_supplier.price\*(1 + bankruptcy\_buffer)\*capital\_required - firm.exp\_OCF

agg\_hh\_wealth = 0

for hh in hhs:

agg\_hh\_wealth += hh.deposits

# The firm collects the resources required.

for hh in hhs:

amount = min(hh.deposits, resources\_required\*hh.deposits/agg\_hh\_wealth)

# The household's deposits (and household's bank's reserves) decrease by the amount paid.

hh.deposits -= amount

hh.bank\_deposits.reserves -= amount

# The firm's deposits (and firm's bank's reserves) increase by the amount paid.

firm.deposits += amount

firm.bank\_deposits.reserves += amount

firm.capital\_stock = 0

for element in firm.capital\_sheet:

if element[1] < max\_capital\_age:

firm.capital\_stock += element[0]

for firm in cfirms:

data[str(firm.ID)+'\_status'].append(firm.status)

data[str(firm.ID)+'\_output\_desired'].append(firm.output\_desired)

data[str(firm.ID)+'\_output\_planned'].append(firm.planned\_output)

data[str(firm.ID)+'\_capital\_stock'].append(firm.capital\_stock)

data[str(firm.ID)+'\_sales'].append(firm.sales)

data[str(firm.ID)+'\_inventories'].append(firm.inventories)

if firm.sales > 0:

data[str(firm.ID)+'\_buffer'].append(firm.inventories/firm.sales)

else:

data[str(firm.ID)+'\_buffer'].append(None)

data[str(firm.ID)+'\_price'].append(firm.price)

data[str(firm.ID)+'\_output'].append(firm.output)

data[str(firm.ID)+'\_capital\_demand'].append(firm.capital\_demand)

data[str(firm.ID)+'\_number\_employees'].append(len(firm.employees))

data[str(firm.ID)+'\_OCF'].append(firm.OCF)

data[str(firm.ID)+'\_loan\_granted'].append(firm.loan\_granted)

data[str(firm.ID)+'\_capital\_bought'].append(firm.capital\_bought)

data[str(firm.ID)+'\_deposits'].append(firm.deposits)

data[str(firm.ID)+'\_mark\_up'].append(firm.mark\_up)

data[str(firm.ID)+'\_ULC'].append(firm.unit\_labour\_cost)

data[str(firm.ID)+'\_uc\_rate'].append(firm.capacity\_utilization\_desired)

def kfirm\_acc():

global kfirms

global hhs

global gov

global unem

global max\_capital\_age; global tau\_pi; global div\_firm; global max\_loan\_age

# Firm accounting.

for firm in kfirms:

# The firm calculates its wage disbursement (for accounting).

agg\_wages = 0

for employee in firm.employees:

agg\_wages += employee.wage

# The firm calculates its interest payment on loans (for accounting).

interests\_on\_loans = 0

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if element[1] > 0:

interests\_on\_loans += element[2]\*element[3]

# The firm updates and calculates its unit costs and nominal variations of inventories.

firm.unit\_cost\_past = firm.unit\_cost\_present

# Only firms that have produced can update their unit costs.

if firm.output > 0:

firm.unit\_cost\_present = agg\_wages/firm.output

nominal\_variation\_inventories = firm.inventories\*firm.unit\_cost\_present - firm.inventories\_past\*firm.unit\_cost\_past

# The firm calculates its profits before tax payment.

# Eq. 3.8

firm.profit\_pre\_tax = firm.revenue + firm.interest\_on\_deposits + nominal\_variation\_inventories - agg\_wages - interests\_on\_loans

# The firm calculates its tax payment (for accounting).

firm.tax = max(0, firm.profit\_pre\_tax\*tau\_pi)

# The firm calculates its profits after tax payment (for accounting).

firm.profit\_post\_tax = firm.profit\_pre\_tax - firm.tax

# The firm calculates its dividend disbursement.

firm.dividend\_disbursement = max(0, div\_firm\*firm.profit\_post\_tax)

# The firm calculates its loan repayment (for accounting):

loan\_repayment = 0

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if element[1] > 0:

loan\_repayment += element[0]/max\_loan\_age

# The firm calculates its Operating Cash Flow.

firm.OCF = firm.revenue + firm.interest\_on\_deposits - agg\_wages - interests\_on\_loans - loan\_repayment - firm.tax

# If deposits are enough to pay wages.

if firm.deposits >= agg\_wages or agg\_wages == 0:

# Wages are paid.

for employee in firm.employees:

# The firm's deposits (and firm's bank's reserves) decrease by the wage paid.

firm.deposits -= employee.wage

firm.bank\_deposits.reserves -= employee.wage

# The employee's deposits (and employee's bank's reserves) increse by the wage paid.

employee.deposits += employee.wage

employee.bank\_deposits.reserves += employee.wage

# The employee registers the wage payment as income.

employee.income += employee.wage

# If deposits are not enough to pay all wages.

else:

# The firm's status is updated.

firm.status = 'bankrupt'

print('bankrupt'+';'+str(period))

amount\_per\_unit = firm.deposits/agg\_wages

for employee in firm.employees:

# The firm's deposits (and firm's bank's reserves) decrease by the amount paid, which is proportional to the employee's wage.

amount = employee.wage\*amount\_per\_unit

firm.deposits -= amount

firm.bank\_deposits.reserves -= amount

# The employee's deposits (and employee's bank's reserves) increse by the amount paid.

employee.deposits += amount

employee.bank\_deposits.reserves += amount

# The employee registers the wage payment as income.

employee.income += amount

# The employee becomes unemployed

employee.employer = None

employee.periods\_unem\_list[3] = 1

employee.periods\_unem = sum(employee.periods\_unem\_list)

unem.append(employee)

# All employees are fired

firm.employees = []

# If deposits are enough to pay loan and interest repayments.

if firm.deposits >= interests\_on\_loans + loan\_repayment and firm.status != 'bankrupt':

# Loan responsibilities are paid.

for element in firm.loan\_sheet: # [value\_of\_loan,age,interest\_rate,outstanding\_debt,loan\_ID,bank]

if element[1] > 0:

# The firm's deposits (and firm's bank's reserves) decrease by the amount paid.

firm.deposits -= (element[2]\*element[3] + element[0]/max\_loan\_age)

firm.bank\_deposits.reserves -= (element[2]\*element[3] + element[0]/max\_loan\_age)

# The lender's reserves increase by the amount paid.

element[5].reserves += (element[2]\*element[3] + element[0]/max\_loan\_age)

element[5].interest\_payment += element[2]\*element[3]

# The outstanding debt in loan sheets are updated:

element[3] -= element[0]/max\_loan\_age

for ele in element[5].loan\_sheet:

# The bank recognizes the loan.

if ele[4] == element[4]:

# The outstanding loan is updated.

ele[3] = element[3]

# The age of the loan (in the bank's loan sheet) is updated.

ele[1] += 1

# If the loan reaches maturity, the loan is removed from the bank's loan sheet.

if ele[3] == 0:

element[5].loan\_sheet.remove(ele)

break

# The age of the loan (in the firm's loan sheet) is updated.

i = 0

while True:

try:

element = firm.loan\_sheet[i]

except IndexError:

break

else:

element[1] += 1

# If the loan reaches maturity, the loan is removed from the firm's loan sheet.

if element[3] == 0:

firm.loan\_sheet.remove(element)

else:

i += 1

# If deposits are not enough to pay loan and interest repayments.

else:

# The firm's status is updated.

firm.status = 'bankrupt'

for employee in firm.employees:

# The employee becomes unemployed

employee.employer = None

employee.periods\_unem\_list[3] = 1

employee.periods\_unem = sum(employee.periods\_unem\_list)

unem.append(employee)

# All employees are fired

firm.employees = []

payment\_list = []

for element in firm.loan\_sheet:

if element[1] > 0:

payment\_list.append([element[2]\*element[3] + element[0]/max\_loan\_age, element])

payment\_list.sort(reverse = False)

for p\_list in payment\_list:

if firm.deposits >= p\_list[0]:

# The firm's deposits (and the firm's bank's reserves) decrease by the amount paid.

firm.deposits -= p\_list[0]

firm.bank\_deposits.reserves -= p\_list[0]

# The lender's reserves increase by the amount paid.

p\_list[1][5].reserves += p\_list[0]

p\_list[1][5].interest\_payment += p\_list[1][2]\*p\_list[1][3]

# The outstanding debt in loan sheets are updated:

p\_list[1][3] -= element[0]/max\_loan\_age

for ele in p\_list[1][5].loan\_sheet:

# The bank recognizes the loan.

if ele[4] == p\_list[1][4]:

# The outstanding loan is updated.

ele[3] = p\_list[1][3]

# The age of the loan (in the bank's loan sheet) is updated.

ele[1] += 1

# If the loan reaches maturity, the loan is removed from the bank's loan sheet.

if ele[3] == 0:

p\_list[1][5].loan\_sheet.remove(ele)

break

# The firm eliminates all loan responsibilities

for loan in firm.loan\_sheet:

for ele in loan[5].loan\_sheet:

# The lender recognizes the loan.

if ele[4] == loan[4]:

# The loan is removed from the lender's loan sheet.

loan[5].loan\_sheet.remove(ele)

break

# All loans are removed from the firm's loan sheet.

firm.loan\_sheet = []

# The status of the firm, if not bankrupt, is updated:

if firm.status != 'bankrupt':

firm.status = None

# If deposits are enough to pay for taxes, taxes are paid.

if firm.deposits >= firm.tax and firm.status != 'bankrupt':

# The firm's deposits (and the firm's bank's reserves) decrease by the amount paid.

firm.deposits -= firm.tax

firm.bank\_deposits.reserves -= firm.tax

# The government's reserves increase by the amount paid.

gov.reserves += firm.tax

gov.tax\_payment += firm.tax

# If deposits are enough to pay the dividend disbursement.

if firm.deposits >= firm.dividend\_disbursement and firm.status != 'bankrupt':

# The firm's deposits (and the firm's bank's reserves) decrease by the amount paid.

firm.deposits -= firm.dividend\_disbursement

firm.bank\_deposits.reserves -= firm.dividend\_disbursement

# For disbursement, the aggregate wealth of households is calculated.

agg\_hh\_wealth = 0

for hh in hhs:

agg\_hh\_wealth += hh.deposits

for hh in hhs:

# The household's deposits (and household's bank's reserves) increase by the amount paid, which is registered as income.

amount = firm.dividend\_disbursement\*hh.deposits/agg\_hh\_wealth

hh.bank\_deposits.reserves += amount

hh.income += amount

hh.deposits += amount

def bank\_acc():

global banks

global hhs

global gov

global cb

global i\_cash\_advances; global i\_bonds; global tau\_pi; global div\_bank; global min\_capital\_ratio; global price\_bonds

for bank in banks:

# The bank calculates its interest responsibilities.

interest\_on\_deposits = 0

for agent in bank.deposits\_list:

interest\_on\_deposits += agent.interest\_on\_deposits

# The bank calculates its profits before tax payment.

bank.profit\_pre\_tax = bank.interest\_payment + i\_bonds\*bank.bonds\*price\_bonds - interest\_on\_deposits - i\_cash\_advances\*bank.cash\_advances

# The bank calculates its tax payment (for accounting).

bank.tax = max(0, bank.profit\_pre\_tax\*tau\_pi)

# The bank calculates its profits after tax payment (for accounting).

bank.profit\_post\_tax = bank.profit\_pre\_tax - bank.tax

# The firm calculates its dividend disbursement.

bank.dividend\_disbursement = max(0, div\_bank\*bank.profit\_post\_tax)

# The bank receives its bond repayment.

amount = price\_bonds\*bank.bonds\*(1 + i\_bonds)

gov.reserves -= amount

gov.bond\_payment += price\_bonds\*bank.bonds\*i\_bonds

bank.reserves += amount

# The governmnet updates the amount of bonds.

gov.bonds -= bank.bonds

# The bank's bonds are set to 0.

bank.bonds = 0

# The bank pays the cash advance repayment.

gov.reserves += bank.cash\_advances\*i\_cash\_advances

gov.cash\_advances\_payment += bank.cash\_advances\*i\_cash\_advances

bank.reserves -= bank.cash\_advances\*(1 + i\_cash\_advances)

# The central bank updates the amount of cash advances.

cb.cash\_advances -= bank.cash\_advances

# The bank's cash advances are set to 0.

bank.cash\_advances = 0

# The bank's reserves decrease by the tax paid.

bank.reserves -= bank.tax

# The gov's reserves increase by the tax paid and is registered for government's accounting.

gov.reserves += bank.tax

gov.tax\_payment += bank.tax

# The bank's reserves decrease by the amount disbursed.

bank.reserves -= bank.dividend\_disbursement

# For disbursement, the aggregate wealth of households is calculated.

agg\_hh\_wealth = 0

for hh in hhs:

agg\_hh\_wealth += hh.deposits

for hh in hhs:

# The household's deposits (and household's bank's reserves) increase by the amount paid, which is registered as income.

amount = bank.dividend\_disbursement\*hh.deposits/agg\_hh\_wealth

hh.deposits += amount

hh.bank\_deposits.reserves += amount

# The bank calculates the total outstanding loans it has as assets.

total\_loan = 0

for element in bank.loan\_sheet:

total\_loan += element[3]

# The bank calculates the total deposits it has as liabilities.

deposits = 0

for agent in bank.deposits\_list:

deposits += agent.deposits

# The bank's net worth is calculated.

bank.net\_worth = bank.reserves + total\_loan + bank.bonds\*price\_bonds - deposits - bank.cash\_advances

# The bank's capital ratio is calculated.

if total\_loan > 0:

bank.capital\_ratio = bank.net\_worth/total\_loan

# In the case of there not being any loans as assets, the bank will lower its interest rate.

# For this, the bank is marked with the flag 'no loans'.

else:

bank.capital\_ratio = 'no loans'

if bank.capital\_ratio != 'no loans' and bank.capital\_ratio < 0:

# The resources needed to restore the capital ratio are calculated.

change = min\_capital\_ratio\*total\_loan - bank.net\_worth

# The total deposits are calculated.

total\_deposits = 0

for agent in bank.deposits\_list:

total\_deposits += agent.deposits

# Depositors' deposits decrease.

for agent in bank.deposits\_list:

amount= change\*agent.deposits/total\_deposits

agent.deposits -= amount

def hh\_acc():

global hhs

global gov

global unem

global tau\_hh; global omega; global alpha\_1; global alpha\_2

global D\_consumption

global Average\_wages

d\_cons = 0

# For dole calculation, the average wage of employed households is calculated.

agg\_wages\_employed = 0

total\_employed = 0

for hh in hhs:

if hh.employer != None:

agg\_wages\_employed += hh.wage

total\_employed += 1

average\_wage = agg\_wages\_employed/total\_employed

Average\_wages.append(average\_wage)

# The dole is calculated.

dole = omega\*average\_wage

# Household accounting: the household sets its net disposable income.

for hh in hhs:

# Public servants receive their wage.

if hh.employer == gov:

# The household's deposits (and household's bank's reserves) increase by the wage paid, which is registered as income.

hh.income += hh.wage

hh.deposits += hh.wage

hh.bank\_deposits.reserves += hh.wage

# The government's reserves decrease by the wage paid, and it is registered for government's accounting.

gov.reserves -= hh.wage

gov.wage\_payment += hh.wage

# The tax paid by the household is calculated.

hh.tax = tau\_hh\*hh.income

# The household's wealth before tax payment is registered.

hh.wealth\_pre = hh.deposits

# The tax is paid: the household's deposits (and household's bank's reserves) decrease by the tax paid.

amount = min(hh.deposits, hh.tax)

hh.deposits -= amount

hh.bank\_deposits.reserves -= amount

# The government's reserves increase by the tax paid and it is registered for government's accounting.

gov.reserves += amount

gov.tax\_payment += amount

# The net disposable income is calculated.

hh.net\_income = hh.income - amount

# If the household is unemployed, it receives a tax-exempt dole from the government.

if hh.employer == None:

# The household's deposits (and household's bank's reserves) increase by the dole received, which is registered as net income.

hh.net\_income += dole

hh.deposits += dole

hh.bank\_deposits.reserves += dole

# The government's reserves decrease by the dole given.

gov.reserves -= dole

gov.dole\_payment += dole

# The household plans its future consumption.

# The household updates its price expectation.

hh.exp\_price = expectation(hh.last\_price, hh.exp\_price)

# The household calculates its desired consumption.

hh.desired\_consumption = (1/hh.exp\_price)\*(alpha\_1\*hh.net\_income + alpha\_2\*hh.deposits)

d\_cons += hh.desired\_consumption\*parameters.param\_dict['Initial price for C goods']

D\_consumption.append(d\_cons)

def gov\_acc():

global gov

global cb

global price\_bonds

# All government bonds reach maturity. The government repays the central bank (as the interests are given to the government their ignored)

gov.reserves -= cb.bonds\*price\_bonds

gov.bonds -= cb.bonds

cb.bonds = 0

# New bonds are emitted.

gov.bonds = max(0, gov.wage\_payment + gov.dole\_payment + gov.bond\_payment - (gov.tax\_payment + gov.cash\_advances\_payment))/price\_bonds

def bonds\_and\_cash\_market():

global banks

global gov

global cb

global banks

global average\_liquidity\_ratio; global min\_liquidity\_ratio; global price\_bonds

# Each bank updates its liquidity ratio.

for bank in banks:

# The bank calculates the total deposits it has as liabilities.

deposits = 0

for agent in bank.deposits\_list:

deposits += agent.deposits

# The bank's liquidity ratio is calculated.

if deposits > 0:

bank.liquidity\_ratio = bank.reserves/deposits

# In the case of there not being any deposits as liabilities, the bank will increase its interest rate.

# For this, the bank is marked with the flag 'no deposits'.

else:

bank.liquidity\_ratio = 'no deposits'

if bank.liquidity\_ratio != 'no deposits':

# If the average liquidity ratio of the sector is above the legal minimum, the banks will decide according to

# the average liquidity ratio.

if average\_liquidity\_ratio >= min\_liquidity\_ratio:

# The desired amount of reserves is calculated.

desired\_reserves = deposits\*average\_liquidity\_ratio

# If the bank has more reserves than needed, it will invest in government bonds.

if bank.reserves >= desired\_reserves:

bank.bond\_demand = (bank.reserves - desired\_reserves)/price\_bonds

bank.cash\_demand = 0

# If the bank has less reserves than needed, it will ask for a cash advance.

else:

bank.bond\_demand = 0

bank.cash\_demand = desired\_reserves - bank.reserves

# If the average liquidity ratio of the sector is below the legal minimum, the banks will decide according to

# the minimum liquidity ratio.

else:

# The desired amount of reserves is calculated.

desired\_reserves = deposits\*min\_liquidity\_ratio

# If the bank has more reserves than needed, it will invest in government bonds.

if bank.reserves >= desired\_reserves:

bank.bond\_demand = (bank.reserves - desired\_reserves)/price\_bonds

bank.cash\_demand = 0

# If the bank has less reserves than needed, it will ask for a cash advance.

else:

bank.bond\_demand = 0

bank.cash\_demand = desired\_reserves - bank.reserves

# For the interaction to be 'fair', the banks will be selected at random to interact with the government or the central bank.

random.shuffle(banks)

# A list of banks that demand bonds is created.

bond\_demand = []

# A list of banks that demand cash advances is created.

cash\_demand = []

for bank in banks:

if bank.liquidity\_ratio != 'no deposits':

if bank.bond\_demand > 0:

bond\_demand.append(bank)

if bank.cash\_demand > 0:

cash\_demand.append(bank)

# Each bank that demands bonds interacts with the government.

bonds\_remaining = gov.bonds

for bank in bond\_demand:

# The amount of bonds bought is constrained by the quantity of bonds remaining.

bank.bonds = min(bonds\_remaining, bank.bond\_demand)

# The transaction is made.

# The bank's reserves decrease by the amount paid.

bank.reserves -= bank.bonds\*price\_bonds

# The government's reserves increase by the amount paid.

gov.reserves += bank.bonds\*price\_bonds

# The amount of available bonds is updated.

bonds\_remaining -= bank.bonds

# If not all bonds have been bought by comercial banks, the central bank buys the remaining bonds.

if bonds\_remaining > 0:

cb.bonds = bonds\_remaining

# The government's reserves increase by the amount paid.

gov.reserves += cb.bonds\*price\_bonds

# Each bank that demands cash advances interacts with the central bank.

for bank in cash\_demand:

# The amount resources given as cash advance is not restricted.

bank.cash\_advances = bank.cash\_demand

# The transaction is made.

# The bank's reserves increase by the amount given.

bank.reserves += bank.cash\_advances

# The central bank updates the amount of cash advances given.

cb.cash\_advances += bank.cash\_advances

def statistics():

global Y\_cons\_nominal

global CPI

global Y\_cons\_real

global unemployment

global GINI\_income\_pre

global GINI\_income\_post

global GINI\_wealth\_pre

global GINI\_wealth\_post

global Y\_cons; global Y\_cons\_r; global u\_rate; global unem; global nhhs; global hhs

# The nominal consumption output is added to the list.

Y\_cons\_nominal.append(Y\_cons)

# The average price for consumption goods is calculated.

if Y\_cons\_r > 0:

average\_price\_cons = Y\_cons/Y\_cons\_r

else:

average\_price\_cons = None

if average\_price\_cons != None:

# The CPI is calculated.

cpi = (average\_price\_cons/parameters.param\_dict['Initial price for C goods'])\*100

else:

cpi = None

# The CPI is added to the list.

CPI.append(cpi)

# The real consumption output is calculated.

y\_cons\_real = Y\_cons\_r\*parameters.param\_dict['Initial price for C goods']

# The real consumption output is added to the list.

Y\_cons\_real.append(y\_cons\_real)

# The economy's unemployment rate is updated.

u\_rate = len(unem)/nhhs

# The unemployment rate is added to the list.

unemployment.append(u\_rate)

# GINI coefficients:

# For income before taxes and subsidies:

agg\_income\_pre = 0

hh\_list = []

for hh in hhs:

agg\_income\_pre += hh.income

hh\_list.append(hh.income)

hh\_list.sort()

average = agg\_income\_pre/nhhs

sum\_1 = 0

SUM\_1 = 0

SUM\_2 = 0

k = 0

for amount in hh\_list:

k += 1

sum\_1 += amount

SUM\_1 += (k\*average - sum\_1)

SUM\_2 += (k\*average)

# The GINI coefficient is calculated.

GINI = SUM\_1/SUM\_2

# The GINI coefficient is added to the list.

GINI\_income\_pre.append(GINI)

# For income after taxes and subsidies:

agg\_income\_post = 0

hh\_list = []

for hh in hhs:

agg\_income\_post += hh.net\_income

hh\_list.append(hh.net\_income)

hh\_list.sort()

average = agg\_income\_post/nhhs

sum\_1 = 0

SUM\_1 = 0

SUM\_2 = 0

k = 0

for amount in hh\_list:

k += 1

sum\_1 += amount

SUM\_1 += (k\*average - sum\_1)

SUM\_2 += (k\*average)

# The GINI coefficient is calculated.

GINI = SUM\_1/SUM\_2

# The GINI coefficient is added to the list.

GINI\_income\_post.append(GINI)

# For wealth before taxes and subsidies:

agg\_wealth\_pre = 0

hh\_list = []

for hh in hhs:

agg\_wealth\_pre += hh.wealth\_pre

hh\_list.append(hh.wealth\_pre)

hh\_list.sort()

average = agg\_wealth\_pre/nhhs

sum\_1 = 0

SUM\_1 = 0

SUM\_2 = 0

k = 0

for amount in hh\_list:

k += 1

sum\_1 += amount

SUM\_1 += (k\*average - sum\_1)

SUM\_2 += (k\*average)

# The GINI coefficient is calculated.

GINI = SUM\_1/SUM\_2

# The GINI coefficient is added to the list.

GINI\_wealth\_pre.append(GINI)

# For wealth after taxes and subsidies:

agg\_wealth\_post = 0

hh\_list = []

for hh in hhs:

agg\_wealth\_post += hh.deposits

hh\_list.append(hh.deposits)

hh\_list.sort()

average = agg\_wealth\_post/nhhs

sum\_1 = 0

SUM\_1 = 0

SUM\_2 = 0

k = 0

for amount in hh\_list:

k += 1

sum\_1 += amount

SUM\_1 += (k\*average - sum\_1)

SUM\_2 += (k\*average)

# The GINI coefficient is calculated.

GINI = SUM\_1/SUM\_2

# The GINI coefficient is added to the list.

GINI\_wealth\_post.append(GINI)

def plot\_general():

global Y\_cons\_nominal

global CPI

global Y\_cons\_real

global unemployment

global GINI\_income\_pre

global GINI\_income\_post

global GINI\_wealth\_pre

global GINI\_wealth\_post

global n\_data\_points

plt.plot(Y\_cons\_nominal)

plt.xlabel('Time')

plt.ylabel('Nominal Consumption Output')

plt.title('Evolution of nominal consumption output', )

plt.show

plt.plot(Y\_cons\_real)

plt.xlabel('Time')

plt.ylabel('Real Consumption Output')

plt.title('Evolution of real consumption output', )

plt.show

plt.plot(CPI)

plt.xlabel('Time')

plt.ylabel('Consumer Price Index (%)')

plt.title('Evolution of consumer price index', )

plt.show

plt.plot(unemployment)

plt.xlabel('Time')

plt.ylabel('Unemployment rate')

plt.title('Evolution of unemployment rate', )

plt.show

plt.plot(GINI\_income\_post)

plt.xlabel('Time')

plt.ylabel('Income Gini Coefficient')

plt.title('Evolution of income gini coefficient', )

plt.show

plt.plot(GINI\_wealth\_post)

plt.xlabel('Time')

plt.ylabel('Wealth Gini Coefficient')

plt.title('Evolution of wealth gini coefficient', )

plt.show

def export\_data():

global cfirms; global data; global l

l = []

for firm in cfirms:

l.append(len(data[str(firm.ID)+'\_status']))

l.append(len(data[str(firm.ID)+'\_output\_desired']))

l.append(len(data[str(firm.ID)+'\_capital\_stock']))

l.append(len(data[str(firm.ID)+'\_sales']))

l.append(len(data[str(firm.ID)+'\_inventories']))

l.append(len(data[str(firm.ID)+'\_buffer']))

l.append(len(data[str(firm.ID)+'\_price']))

l.append(len(data[str(firm.ID)+'\_output']))

l.append(len(data[str(firm.ID)+'\_capital\_demand']))

l.append(len(data[str(firm.ID)+'\_number\_employees']))

l.append(len(data[str(firm.ID)+'\_OCF']))

l.append(len(data[str(firm.ID)+'\_profit\_rate']))

l.append(len(data[str(firm.ID)+'\_loan\_granted']))

l.append(len(data[str(firm.ID)+'\_growth\_rate\_desired']))

l.append(len(data[str(firm.ID)+'\_capital\_bought']))

l.append(len(data[str(firm.ID)+'\_deposits']))

l.append(len(data[str(firm.ID)+'\_mark\_up']))

l.append(len(data[str(firm.ID)+'\_total\_loan\_repayment']))

l.append(len(data[str(firm.ID)+'\_ULC']))

l.append(len(data[str(firm.ID)+'\_ULC\_expected']))

l.append(len(data[str(firm.ID)+'\_output\_planned']))

l.append(len(data[str(firm.ID)+'\_uc\_rate']))

l.append(len(data[str(firm.ID)+'\_%L\_of\_costs']))

l.append(len(data[str(firm.ID)+'\_agg\_wages']))

n = max(l)

for firm in cfirms:

if len(data[str(firm.ID)+'\_status']) < n:

m = n - len(data[str(firm.ID)+'\_status'])

for i in range(m):

data[str(firm.ID)+'\_status'].append(None)

if len(data[str(firm.ID)+'\_output\_desired']) < n:

m = n - len(data[str(firm.ID)+'\_output\_desired'])

for i in range(m):

data[str(firm.ID)+'\_output\_desired'].append(None)

if len(data[str(firm.ID)+'\_capital\_stock']) < n:

m = n - len(data[str(firm.ID)+'\_capital\_stock'])

for i in range(m):

data[str(firm.ID)+'\_capital\_stock'].append(None)

if len(data[str(firm.ID)+'\_sales']) < n:

m = n - len(data[str(firm.ID)+'\_sales'])

for i in range(m):

data[str(firm.ID)+'\_sales'].append(None)

if len(data[str(firm.ID)+'\_inventories']) < n:

m = n - len(data[str(firm.ID)+'\_inventories'])

for i in range(m):

data[str(firm.ID)+'\_inventories'].append(None)

if len(data[str(firm.ID)+'\_buffer']) < n:

m = n - len(data[str(firm.ID)+'\_buffer'])

for i in range(m):

data[str(firm.ID)+'\_buffer'].append(None)

if len(data[str(firm.ID)+'\_price']) < n:

m = n - len(data[str(firm.ID)+'\_price'])

for i in range(m):

data[str(firm.ID)+'\_price'].append(None)

if len(data[str(firm.ID)+'\_output']) < n:

m = n - len(data[str(firm.ID)+'\_output'])

for i in range(m):

data[str(firm.ID)+'\_output'].append(None)

if len(data[str(firm.ID)+'\_capital\_demand']) < n:

m = n - len(data[str(firm.ID)+'\_capital\_demand'])

for i in range(m):

data[str(firm.ID)+'\_capital\_demand'].append(None)

if len(data[str(firm.ID)+'\_number\_employees']) < n:

m = n - len(data[str(firm.ID)+'\_number\_employees'])

for i in range(m):

data[str(firm.ID)+'\_number\_employees'].append(None)

if len(data[str(firm.ID)+'\_OCF']) < n:

m = n - len(data[str(firm.ID)+'\_OCF'])

for i in range(m):

data[str(firm.ID)+'\_OCF'].append(None)

if len(data[str(firm.ID)+'\_profit\_rate']) < n:

m = n - len(data[str(firm.ID)+'\_profit\_rate'])

for i in range(m):

data[str(firm.ID)+'\_profit\_rate'].append(None)

if len(data[str(firm.ID)+'\_loan\_granted']) < n:

m = n - len(data[str(firm.ID)+'\_loan\_granted'])

for i in range(m):

data[str(firm.ID)+'\_loan\_granted'].append(None)

if len(data[str(firm.ID)+'\_growth\_rate\_desired']) < n:

m = n - len(data[str(firm.ID)+'\_growth\_rate\_desired'])

for i in range(m):

data[str(firm.ID)+'\_growth\_rate\_desired'].append(None)

if len(data[str(firm.ID)+'\_capital\_bought']) < n:

m = n - len(data[str(firm.ID)+'\_capital\_bought'])

for i in range(m):

data[str(firm.ID)+'\_capital\_bought'].append(None)

if len(data[str(firm.ID)+'\_deposits']) < n:

m = n - len(data[str(firm.ID)+'\_deposits'])

for i in range(m):

data[str(firm.ID)+'\_deposits'].append(None)

if len(data[str(firm.ID)+'\_mark\_up']) < n:

m = n - len(data[str(firm.ID)+'\_mark\_up'])

for i in range(m):

data[str(firm.ID)+'\_mark\_up'].append(None)

if len(data[str(firm.ID)+'\_total\_loan\_repayment']) < n:

m = n - len(data[str(firm.ID)+'\_total\_loan\_repayment'])

for i in range(m):

data[str(firm.ID)+'\_total\_loan\_repayment'].append(None)

if len(data[str(firm.ID)+'\_ULC']) < n:

m = n - len(data[str(firm.ID)+'\_ULC'])

for i in range(m):

data[str(firm.ID)+'\_ULC'].append(None)

if len(data[str(firm.ID)+'\_ULC\_expected']) < n:

m = n - len(data[str(firm.ID)+'\_ULC\_expected'])

for i in range(m):

data[str(firm.ID)+'\_ULC\_expected'].append(None)

if len(data[str(firm.ID)+'\_output\_planned']) < n:

m = n - len(data[str(firm.ID)+'\_output\_planned'])

for i in range(m):

data[str(firm.ID)+'\_output\_planned'].append(None)

if len(data[str(firm.ID)+'\_uc\_rate']) < n:

m = n - len(data[str(firm.ID)+'\_uc\_rate'])

for i in range(m):

data[str(firm.ID)+'\_uc\_rate'].append(None)

if len(data[str(firm.ID)+'\_%L\_of\_costs']) < n:

m = n - len(data[str(firm.ID)+'\_%L\_of\_costs'])

for i in range(m):

data[str(firm.ID)+'\_%L\_of\_costs'].append(None)

if len(data[str(firm.ID)+'\_agg\_wages']) < n:

m = n - len(data[str(firm.ID)+'\_agg\_wages'])

for i in range(m):

data[str(firm.ID)+'\_agg\_wages'].append(None)

data\_frame = pd.DataFrame(data)

data\_frame.to\_csv('cfirms\_data.csv')

def run():

global n\_iterations

global period

period = 0

initialize()

initial\_network()

while period <= n\_iterations:

period += 1

plan()

capital\_market\_1()

credit\_market()

labour\_market()

production()

capital\_market\_2()

consumption\_market()

deposits\_payment()

cfirm\_acc()

kfirm\_acc()

bank\_acc()

hh\_acc()

gov\_acc()

deposits\_market()

bonds\_and\_cash\_market()

statistics()

print(period)

#plot\_general()

run()

export\_data()