Bancor Simulator

Algorithm Notations About Bancor Simulator

Kenny

Sept, 2017

At Shanghai

1 Algorithms in Python Classes

Here, we illustrate algorithms of codes implemented in our simulator, which mainly consists of four classes – *Smart Token*, *Customer*, *Bancor Market* and *Classic Market*. And we will introduce respectively.

1.1 Smart Token

Algorithm 1 Algorithm Notation of Smart Token Class, purchasing() and destroying() in smartToken.py

Input: n: the tokens' number customer wants to buy or sell
Output: n': the tokens' number customer receive – when buying, returns smart tokens' number while when selling, returns reserve tokens'.

if Customers buy smart tokens **then**
  Let $n' := \text{round}(S \times \left[\left(\frac{B+n}{B}\right)^{\text{CRR}} - 1\right])$,
  where S denotes supply, B means reserve balance and CRR corresponds to the constant reserve ratio.
  Update the supply for smart token class, let $S := S + n'$.
  Update the reserve balance for smart token class, let $B := B + n$.
  Update the smart token's price p, let $p := \frac{B}{S \times \text{CRR}}$.
  return n'
else
  Let $n' := \text{round}(B \times \left[1 - \left(\frac{B-n}{B}\right)^{\frac{1}{\text{CRR}}}\right])$,
  where S denotes supply, B means reserve balance and CRR corresponds to the constant reserve ratio.
  Update the supply for Smart Token class, let $S := S - n$.
  Update the reserve balance for Smart Token class, let $B := B - n'$.
  Update the smart token's price p, let $p := \frac{B}{S \times \text{CRR}}$.
  return n'
end if

The Smart Token class implements the key idea of Bancor protocol, including the calculation of smart token's price and the number of issued or destroyed smart token's number. In fact, the whitepaper of Bancor protocol [**?**] hides the calculating formula for specifying how many smart tokens or reserve tokens should be issued in purchasing or selling. And we find these calculating methods from Bancor protocol's source code [**?**], which we list below:

$$I_s =< S \times \left[\left(\frac{B + \Delta_p}{B}\right)^{\text{CRR}} - 1\right] >,$$

$$I_r =< B \times \left[1 - \left(\frac{B - \Delta_s}{B}\right)^{\frac{1}{\text{CRR}}}\right] >,$$

where I_s and I_r denote the number of issued smart tokens and reserve tokens, B means the reserve balance, CRR represents the constant reserve ratio, Δ_p and Δ_d correspond to the reserve tokens' number buyer uses to purchase smart tokens and the smart token number seller uses to sell respectively, and symbol

<> means the round function, which indicates that the number of issued tokens must be integer according to the Bancor protocol.

Here in Algorithm 1, we show algorithm of Smart Token class.

1.2 Customer

The Customer class simulates customer's behavior, in which customers can know the smart token's price in market, launch their orders in market. Here we assume that in every time slot, customers in the market will change their valuations of smart token. If their valuations are larger than smart token's current price, they would be likely to buy smart tokens, yet be eager to sell if valuations are smaller.

Here, we present the algorithm for customer to change his valuation and in the meanwhile launch his order in Algorithm 2.

Algorithm 2 Valuation Altering and Order Launching Algorithm in Customer class, changeValuation() in customers.py

Input: v': new valuation

if The customer's order has been finished by market **then**
  Change the valuation v of smart token in Customer class, let $v := v'$.
  Tell market to cancel the old transaction order: cancelOrder().
  Get the current smart token's price p_c by asking market, let p_c :=market.getCurrentPrice().
  if $v > p_c$ and customer's reserve balance $b_r > 0$ **then**
    Launch a new buy transaction order to market, call market.buy().
  else
    if $v < p_c$ and customer's smart token balance $b_t > 0$ **then**
      Launch a new sell transaction order to market, call market.sell().
    else
      Do nothing and return.
    end if
  end if
else
  Do nothing
end if

1.3 Bancor Market

The Bancor Market class simulates the market which handles customers' requests by Bancor protocol. In the beginning of every time slot, Bancor market updates the current price of smart token which will be broadcasted within customers and help customers to make valuations.

Here in Algorithm 3, we show how Bancor Market deals with the customers' requests. And in simulating, to simplify the model, we assume that at one time, one customer can only push one order to market.

Algorithm 3 Pocess the Transaction Orders in Bancor Market Class, BancorMarket.buy() and BancorMarket.sell() in market.py

Input: $\mathbb{C}$: the customer who wants to cancel the transaction order, a_t: the amount of transaction value.

Verify whether the a_t is legal, otherwise throw error – check whether it is integer, and larger than 0, and etc.
if $\mathbb{C}$ is launching a buy order **then**
  if $\mathbb{C}$'s valuation is larger or equal to smart token's current price **then**
    Call smart token class to buy a_t smart token by Algorithm 1, get recerived smart tokens' number as a_r.
    Add $\mathbb{C}$'s smart tokens' number, call $\mathbb{C}$.changeTokenBalance(a_r).
    Decrease $\mathbb{C}$'s reserve tokens' number, call $\mathbb{C}$.changeReserveBalance(-a_t).
  else
    Plus the canceled transaction number by 1 and do nothing else, since this order cannot be processed by Bancor Market.
  end if
else
  if $\mathbb{C}$'s valuation is smaller or equal to smart token's current price **then**
    Call smart token class to sell a_t smart token by Algorithm 1, get recerived reserve tokens' number as a'_r.
    Add $\mathbb{C}$'s reserve tokens' number, call $\mathbb{C}$.changeReserveBalance(a'_r).
    Decrease $\mathbb{C}$'s smart tokens' number, call $\mathbb{C}$.changeTokenBalance(-a_t).
  else
    Plus the canceled transaction number by 1 and do nothing else, since this order cannot be processed by Bancor Market.
  end if
end if

1.4 Classic Market

The Classic Market class simulates the market running under classic market laws. Here, we show how Classic Market processes customers' transaction orders in Algorithm 4 and 5, which, compared with the Bancor market, is more complicated since when market managing transaction orders, it needs to search for whether there are matched orders in *orderlist*, which is a list maintained in Classic Market.

Algorithm 4 Pocess the Transaction Orders in Classic Market, ClassicMarket.buy() and ClassicMarket.sell() in market.py

Input: $\mathbb{C}$: the customer who launches the transaction order, a_t: the amount of transaction value.

Verify whether the a_t is legal, otherwise throw error – check whether it is integer, and larger than 0, and etc.
if $\mathbb{C}$ is launching a buy order **then**
    Make a new buy order (combining $\mathbb{C}$ and a_t into a python list), which will be transfered to orderlist maintained by market class.
    Call updateOrderList() function, and transfer the new buy order to this function.
else
    Make a new sell order (combining $\mathbb{C}$ and a_t into a python list), which will be transfered to orderlist maintained by market class.
    Call updateOrderList() function, and transfer the new sell order to this function.
end if

2 Algorithm Notations in Main Function

In this part, we talk about how to simulate the whole market trading processes, which are achieved in main-Bancor.py and main-Classic.py for Bancor Market and Classic Market respectively. The parameters which are supposed to be initialized is listed in Table 1.

Table 1. Parameters for Simulating Experiments

Parameters	Definations
N_c	Number of customers who make deals in market
T	Time interval between market crazes, measured by count of time slots
R	Valuation bouncing range parameter of market craze
σ_0	Variable in Gaussian function for generating customer number's distribution

(1) The Algorithm for Bancor Market Simulating: The algorithm of Bancor market's simulation is presented in Algorithm 6. In real codes, we use the pseudo-random seeds from 1 to 10 to ensure that our experiments can be reproduced.

Algorithm 5 Orderlist Updating Function in Classic Market, ClassicMarket.updateOrderList() in market.py

Input: $\mathbb{O}$: new order which contained parameters such as customer $\mathbb{C}$, transaction amount a_t and flag f which decide whether this order is to buy or to sell.
Temporal Parameters: L_s: a temporary list which contains all matched sell orders, L_b: a temporary list which contains all matched buy orders.

if the order list maintained in classic market class L is empty **then**
  Append $\mathbb{O}$ to L and return, since there is no matched order in L.
end if
if $\mathbb{O}.f$ is BUY **then**
  for all other orders $\mathbb{O}_l$ in L **do**
    if $\mathbb{O}_l.f$ is SELL and $\mathbb{O}_l.\mathbb{C}$ has lower valuation than $\mathbb{O}.\mathbb{C}$'s **then**
      Add $\mathbb{O}_l$ into L_s, which is initialized to be empty.
    end if
  end for
  if L_s is empty **then**
    Append $\mathbb{O}$ to L and return, since there is no matched order.
  else
    Sort L_s by sellers' valuation from low to high, as goods with low price are preferred.
    for all orders $\mathbb{O}_{l_s}$ in sorted L_s **do**
      if $\mathbb{O}.a_t$ is smaller than maximum amount $a_{\max}$ $\mathbb{O}_{l_s}$ could offer **then**
        Update $\mathbb{O}.\mathbb{C}$'s information, by decreasing a_t reserve tokens and decreasing corresponding amount of smart tokens, i.e., $<\frac{a_t}{v_s}>$, where v_s is $\mathbb{O}_{l_s}.\mathbb{C}$'s valuation.
        Update $\mathbb{O}_{l_s}$'s information, decrease $<\frac{a_t}{v_s}>$ smart tokens it provides to sell.
        Update $\mathbb{O}_{l_s}.\mathbb{C}$'s information, by adding a_t reserve tokens and decreasing $<\frac{a_t}{v_s}>$ reserve tokens.
        Break the loop.
      else
        Update $\mathbb{O}.\mathbb{C}$'s information, by decreasing $a_{\max}$ reserve tokens and adding $<\frac{a_{\max}}{v_s}>$ smart tokens.
        Update $\mathbb{O}_{l_s}.\mathbb{C}$'s information, by adding $a_{\max}$ reserve tokens and decreasing $<\frac{a_{\max}}{v_s}>$ smart tokens.
        Pop the $\mathbb{O}_{l_s}$ from L_s, and remove this order from L.
        Update $\mathbb{O}$'s information by decreasing $a_{\max}$ reserve tokens from its transaction value, i.e., $a_t := a_t - a_{\max}$.
      end if
    end for
    if $\mathbb{O}.a_t$ is still larger than 0 **then**
      Append $\mathbb{O}$ to L and return.
    end if
  end if
else
  .....
  {Do same things for new sell order.
  Create L_b but sort is from high to low, since high purchasing price is preferred by seller.}
end if

Algorithm 6 Algorithm Notations for Bancor Maket's Simulation, main-Bancor.py

Initialize parameters: N_t, N_c, T, R and σ_0.
Initialize a smart token, such as $\mathbb{S}$=Smartcoin(*name*='KennyCoin', *reservetoken-Name*='ETH', *CRR*=0.2, *Price*=1, *IssueNum*=initIssue).
Initialize a Bancor Market which uses smart token we have just initialized, such as $\mathbb{M}$=BancorMarket(*smartToken*=$\mathbb{S}$)
Initialize customers with N_c number, such as Customer(*smartToken*=$\mathbb{S}$, *market*=$\mathbb{M}$, *tokenBalance*$\approx$200, *reserveBalance*$\approx$200)
for every time slot t in N_t **do**
    Sychronize the $\mathbb{M}$ by t. {Ask market to generate the current market price}
    if t is not the multiple of T **then**
        Generate valuation array, which makes number of customers Gaussian-like distributed by current market price p as μ, and σ_0 as σ in Gaussian function.
        Use valuation array generated to call changeValuation() for every customer – as shown in Algorithm 2, by which customers actually also launch orders.
    else
        Simulate market craze, and generate valuation array by randomly select mean valuation in range $(\frac{1}{R}p, Rp)$ as μ, and σ_0 as σ in Gaussian function.
        Use valuation array generated to call changeValuation() for every customer – as shown in Algorithm 2.
    end if
    Collect simulating data for future analysis.
end for

(2) The Algorithm for Classic Market Simulating: The algorithm of Bancor market's simulation is presented in Algorithm 7, which is almost similar with the Algorithm 6 with the experimental principle of controlling differences. Also, what should be noticed is that in classic market, there is no smart tokens be issued or destroyed in transactions. Therefore, customers actually generate valuations based on a fixed constant.

Algorithm 7 Algorithm Notations for Bancor Maket's Simulation, main-Classic.py

Initialize parameters: N_t, N_c, T, R and σ_0.
Initialize a smart token, such as $\mathbb{S}$=Smartcoin(*name*='KennyCoin', *reservetokenName*='ETH', *CRR*=0.2, *Price*=1, *IssueNum*=initIssue).
Initialize a Bancor Market which uses smart token we have just initialized, such as $\mathbb{M}$=ClassicMarket(*smartToken*=$\mathbb{S}$)
Initialize customers with N_c number, such as Customer(*smartToken*=$\mathbb{S}$, *market*=$\mathbb{M}$, *tokenBalance*≈200, *reserveBalance*≈200)
for every time slot t in N_t **do**
    Sychronize the $\mathbb{M}$ by t. {Ask market to generate the current market price}
    if t is not the multiple of T **then**
        Generate valuation array, which makes number of customers Gaussian-like distributed by current market price p as μ, and σ_0 as σ in Gaussian function.
        Use valuation array generated to call changeValuation() for every customer – as shown in Algorithm 2, by which customers actually also launch orders.
    else
        Simulate market craze, and generate valuation array by randomly select mean valuation in range $(\frac{1}{R}p, Rp)$ as μ, and σ_0 as σ in Gaussian function.
        Use valuation array generated to call changeValuation() for every customer – as shown in Algorithm 2.
    end if
    Collect simulating data for future analysis.
end for