DutchX Documentation

https://gitter.im/gnosis/DutchX

June 2018

Contents

1 Specifications **3**
1.1 Introduction . . . 3
1.2 Basic mechanisms . . . 3
1.2.1 Exchange . . . 3
1.2.2 Auction . . . 3
1.2.3 Claiming . . . 4
1.3 Advanced mechanisms . . . 5
1.3.1 Adding a token pair . . . 5
1.3.2 Scheduling . . . 5
1.3.3 Fees & MGN tokens . . . 5
1.3.4 Price oracle . . . 6

2 Math in DutchX **8**
2.1 Notation . . . 8
2.2 Non-core functions . . . 8
2.2.1 getPriceInPastAuction . . . 8
2.2.2 getPriceOfTokenInLastAuction . . . 9
2.2.3 getCurrentAuctionPrice . . . 9
2.2.4 getFeeRatio . . . 10
2.2.5 settleFee . . . 10
2.2.6 scheduleNextAuction . . . 11
2.2.7 clearAuction . . . 11
2.3 Core functions . . . 11
2.3.1 postSellOrder . . . 11
2.3.2 claimSellerFunds . . . 12
2.3.3 postBuyOrder . . . 12
2.3.4 claimBuyerFunds . . . 13

3 States **15**
3.1 Introduction . . . 15
3.1.1 Special case: “zero” auction . . . 16
3.1.2 Special case: adding a token pair . . . 16
3.2 postSellOrder . . . 16
3.3 claimSellerFunds . . . 17

3.4 postBuyOrder 17
3.5 claimBuyerFunds 17

Chapter 1

Specifications

1.1 Introduction

The Dutch Exchange is a fully decentralized exchange based on the Dutch auction principle, featuring a price mechanism that starts with a high price which falls monotonically. Eventually, every successful buyer pays the same price once the auction closes, ensuring orders don't have to be cancelled when markets fluctuate. This mechanism allows for a more streamlined trading experience and, most importantly, eliminates the bottlenecks of decentralized order book exchanges such as front-running and scaling difficulties.

1.2 Basic mechanisms

1.2.1 Exchange

The exchange runs as a set of token pairs, which do not have order: an ETH-GNO token pair is a GNO-ETH.

Each pair consists of two opposite/symmetrical auctions – order does matter for an auction.

Opposite auctions always begin at the same time (but may close at different times). "Closing" and "clearing" auctions are synonyms for the purposes of our exchange.

1.2.2 Auction

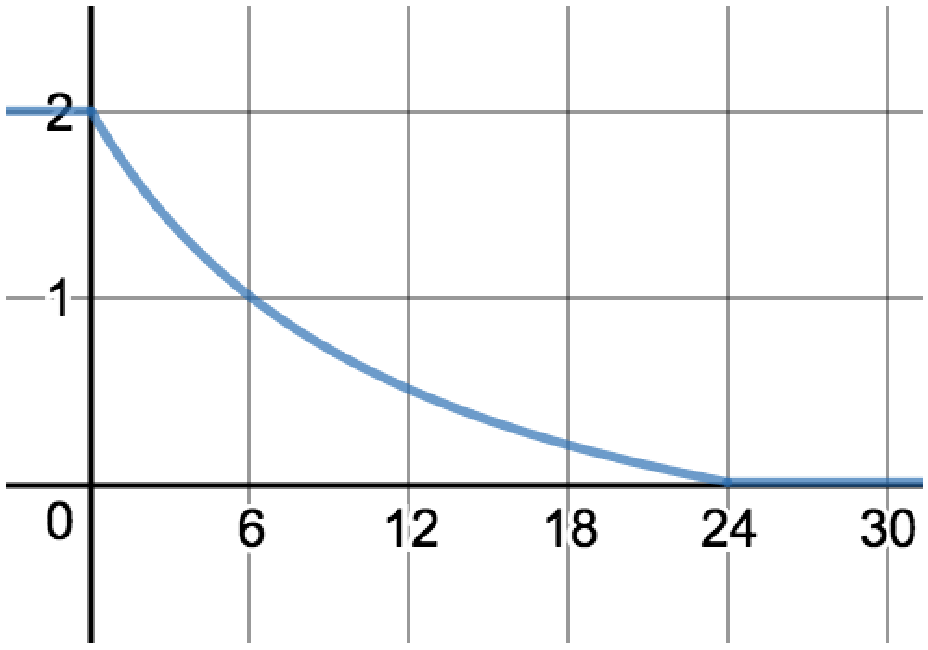
Each auction follows the same mechanism: sell orders are accepted *before* an auction begins. We call the auctioned amount of sell tokens the *sell volume*. Consequently, an auction always has a constant sell volume while it's running.

Buy orders are accepted only *during* an auction. The amount of buy tokens is called the *buy volume*, and an auction closes when the sell volume * current price is smaller than or equal to the buy volume. In fact, the amount processed by the final buy order should be such that the sell volume * current price is *equal* to the buy volume (up to rounding errors).

Current price $P(t)$ at any point in time is a function of time. For t in hours and $t \in [0, 24]$, it is given by *the* reciprocal function that satisfies:

$$\begin{aligned} P(0) &= 2x, \\ P(6) &= x, \\ P(24) &= 0, \end{aligned}$$

and 0 for $t > 24$, where x is a weighted average of the closing prices of the two last auctions, weighted by their trading volume. Consequently, the price function is until 24 hours, a monotonically decreasing function. For $t \in [0, 24]$ and $x = 1$, here is a graph of the price function and here's an image:



1.2.3 Claiming

At any point during the auction, a buyer may claim his/her intermediate funds. As long as the auction clears in a later Ethereum block, the price will drop after that, so he/she is guaranteed to get more tokens when claiming again in the future. A buyer's funds are given by his/her balance divided by the price (current or closing, depending on time of claim).

Sellers may only claim their funds (in buy tokens) *after* an auction has cleared. A seller's funds are given by his/her balance * price.

Buyers and sellers may also claim their remaining funds at any point in the future.

1.3 Advanced mechanisms

1.3.1 Adding a token pair

Token pairs are added by providing 5 variables: addresses of 2 distinct tokens, sell funding for both tokens and what we call an *initial closing price.* For pairs that don't include ETH, we require there to exist ETH-token pairs for both tokens.

For ETH-token pairs, the sell funding of ETH must be at least $10,000. For pairs that don't include ETH, the combined value of sell funding of both pairs must be at least $10,000.

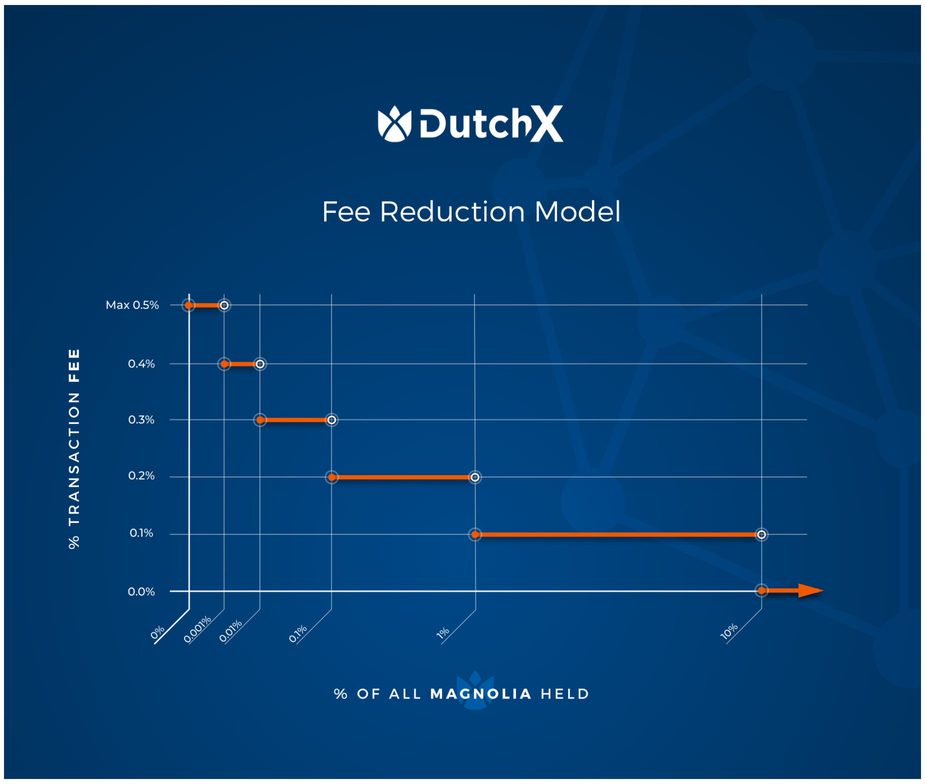
1.3.2 Scheduling

When a token pair is added, the first auction pair is scheduled to begin in 6 hours.

When both auctions in a token pair close, we check if either has received at least $1000 worth of sell funding. If not, the token pair goes into a paused state until that condition is satisfied. The next auction pair is then scheduled to begin in 10 minutes.

1.3.3 Fees & MGN tokens

All buy and sell orders are subject to fees. Let $F(b, t)$ be the fee function, where b is a user's balance of MGN and t is the total supply of MGN. F is a step function that follows the following graph:



A user may pay up to one half of the fee with OWL tokens. In that case we use the conversion 1 OWL = 1 USD.

Sell funding supplied with adding a new token pair also incurs fees.

The last buy order does not have to pay fees.

Finally, every trader receives 1 MGN for each 1 ETH of claimed funds. This conversion is done using the price oracle at the time of the auction.

1.3.4 Price oracle

We provide a reliable price oracle for ERC-20 tokens to ETH. This function computes an average of the closing prices of the two opposite auctions ETH-token and token-ETH. This average is weighted by the trading volume of those two auctions, in the non-ETH token.[1]

We start this process at the current auctionIndex. I.e. if one auction had closed, the price oracle would output only its closing price.

This function is used to determine the passing of thresholds in adding a token pair and scheduling next auction pair, together with an external ETH-

[1]The reader might wonder why we weigh in the non-ETH token rather than by ETH. This is for computational reasons - the math turns out to be much simpler this way, eliminating rounding errors caused by deadline with overflow issues.

USD oracle, as well as in settling fees (calculating how many OWL's must be spent) and issuing MGN.

A generalised version (one where neither token has to be ETH) is also used in determining the current price of all auctions on the Dutch Exchange.

Chapter 2

Math in DutchX

This is an attempt to prove math in DutchX code meets the specifications laid out in Chapter 1. In particular, we want to show the following things: rounding errors behave as expected, overflow and underflow never happen and there isn't a problem with the asynchronous nature of blockchain.

2.1 Notation

We use AB to denote auction with sellToken A and buyToken B. We may abbreviate ETH and Token auctions as ET or TE.

We use $V_S(AB), V_B(AB)$ for sellVolume, buyVolume (respectively) of auction AB.

We use $P_c(AB, i)$ for closingPrice of auction AB with index i. If index is obvious, we use $P_c(AB)$.

2.2 Non-core functions

We'll start by discussing "non-core functions".

2.2.1 getPriceInPastAuction

(token1, token2, auctionIndex) $\rightarrow$ fraction

If both tokens are equal, it ouputs 1/1. If they're not, it loops back, starting from auctionIndex, considering the two closingPrices for each auctionIndex. Once both terms of either closingPrice are > 0, the loop ends.

This loop will always end, because for both auctions, closingPrices[0] had num, den > 0 (see addTokenPair).

We can have 2 cases:

1. One num $= 0$, the other > 0. In that case output will be the closingPrice of the larger one.

2. Both num > 0. In that case the ouput is a weighted average of both closingPrices, weighted by the trading volume. We get:

$$p = \frac{V_B(ET)}{V_B(ET) + V_S(TE)} * \frac{1}{P_c(ET)} + \frac{V_S(TE)}{V_B(ET) + V_S(TE)} * P_c(TE)$$

$$p = \frac{V_B(ET) * \frac{V_S ET}{V_B(ET)} + V_S(TE) * \frac{V_B(TE)}{V_S(TE)}}{V_B(ET) + V_S(TE)}$$

$$p = \frac{V_S(ET) + V_B(TE)}{V_B(ET) + V_S(TE)}$$

NO DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

2.2.2 getPriceOfTokenInLastAuction

(token) $\rightarrow$ fraction

Does two things:

1. Gets latest auction index of token-ETH (if token = ETH, it will output 0).
2. calls getPriceInPastAuction with those tokens and that index

NO DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

2.2.3 getCurrentAuctionPrice

(sellToken, buyToken, auctionIndex) $\rightarrow$ fraction

Should disambiguate based on auctionIndex and state:

1. If auction has closed, output closingPrice.
2. If auction has not begun yet, output 0/0.
3. If auction is running, call getPriceInPastAuction and use price function to output price.
4. If the output is smaller than $\frac{\text{buyVolume}}{\text{sellVolume}}$, we want to output that. In buy order, we check whether this is true. Hence the only time this can be true is when the lsat buy order hadn't cleared the auction, but the price dropped sufficiently later. Since buyers may claim intermediate amounts, this prevents them from claiming more than they will be allowed once the auction clears.

NO DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

2.2.4 getFeeRatio

(user) $\rightarrow$ fraction

Should do two things:

1. Get totalMGN and user's balanceOfMGN.
2. Should output fee ratio based on fee function.

NO DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

2.2.5 settleFee

(primaryToken, secondaryToken, auctionIndex, user, amount) $\rightarrow$ amountAfterFee

(Input vars aren't called sellToken, buyToken, because they are unrelated to sellToken and buyToken of the function they're called in.)

Should do several things:

1. Get feeRatio.
2. Multiply amount by fee ratio to get fee.
3. If fee > 0:
    (a) Find USD value of fee (using priceOracle and ETHUSDPrice).
    (b) Allow user to spend up to half of that with OWL tokens.
4. If fee $= 0$, output amount. Otherwise, output amount $-$ fee.

YES DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

Division

1. To calculate fee.
2. To calculate feeInETH.
3. To calculate amountOfOWLBurned.
4. Adjust fee.

Results:

1. fee will be slightly smaller.
2. feeInUSD will be slightly smaller.
3. amount of OWL a user can burn will be slightly smaller.
4. fee will by adjusted by a slightly smaller amount (will be larger than should be per how many OWL's user burned). This doesn't break anything, because the same number is added to extraTokens as is subtracted from amount to get amountAfterFee.

2.2.6 scheduleNextAuction

(sellToken, buyToken) $\rightarrow$ null

Should:

1. Fetch price oracles of both tokens.
2. Find sellVolumesCurrent of both tokens in USD.
3. If either is $\geq$ the threshold, set auctionStart in 10 minutes and set isWaiting to false. Otherwise, set isWaiting to true.

NO DIVISION, NO OVERFLOW, NO ASYNC ISSUES when called from either clearAuction or postSellOrder.

2.2.7 clearAuction

(sellToken, buyToken, auctionIndex, sellVolume) $\rightarrow$ null

Called from: postBuyOrder.

This function can do many things:

1. Save $\frac{\text{buyVolume}}{\text{sellVolume}}$ as closingPrice. Should always do so.
2. clear auctionPair: should do so if if opposite is a 0 auction OR price reached 0 OR opposite auction closed. This does several things:
    - If opposite auction is non-zero and hasn't cleared yet, save its closing price
    - Sets sellVolumeCurrent to sellVolumeNext for both auctions
    - Resets sellVolumeNext and buyVolumes for both auctions
    - Increments auctionIndex and calls scheduleNextAuction.

NO DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

2.3 Core functions

2.3.1 postSellOrder

(sellToken, buyToken, auctionIndex, amount) $\rightarrow$ null

Should:

1. Adjust amount by user's balance.
2. Require amount and latestAuctionIndex > 0.
3. If user specified auctionIndex 0, adjust is to the correct one. Otherwise, require it is the correct one.
4. Settle fee and calculate amountAfterFee.

5. Adjust user's balances by amount and sellerBalances and sellVolumes by amountAfterFee.

6. If we're in threshold–waiting period, call scheduleNextAuction.

NO DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

2.3.2 claimSellerFunds

(sellToken, buyToken, user, auctionIndex) $\rightarrow$ (returned, MGNIssued)
Should:

1. Require sellerBalance > 0.

2. Require auction to have closed AND not be a zero auction (this is done by requiring den > 0).

3. Calculate returned amount.

4. If both tokens are approved, calculate ETH value of returned and issue that many MGN.

5. Reset sellerBalances and update user's balances.

YES DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

Division

1. To calculate returned.

2. To calculate MGNIssued.

Results:

1. returned will be slightly smaller.

2. amount of MGN issued will be slightly smaller.

2.3.3 postBuyOrder

(sellToken, buyToken, auctionIndex, amount) $\rightarrow$ null
Should:

1. Make sure auction hasn't closed and is running.

2. Make sure auctionIndex is correct.

3. Adjust amount by user's balances (call the result a).

4. Find current price and use it to calculate outstandingVolume, call that V_O.

5. Since $a \geq 0$, we have just a few cases:
   (a) $V_O \leq 0$: should merely clear auction.
   (b) $0 < V_O \leq a$: set $a = V_O$, lower user's balances and increase buyerBalances and buyVolumes by a. Clear auction.
   (c) $0 < a < V_O$: settle fee and find amountAfterFee. Lower user's balances by a and increase buyerBalances and buyVolumes by amountAfterFee.

YES DIVISION, NO OVERFLOW, NO ASYNC ISSUES

Division

1. Calculate outstandingVolume

Results:

1. outstandingVolume will be slightly smaller, so closingPrice will be smaller. Since all calculations use tokens' decimal places (usually 18), the difference will be absolutely marginal to have significance.

2.3.4 claimBuyerFunds

(sellToken, buyToken, user, auctionIndex) $\rightarrow$ (returned, MGNIssued)
Should:

1. Make sure particular auction has ever run.
2. (a) Auction is still running: Compute unclaimed funds by dividing by current price and subtracting claimedAmounts. Add that amount to claimedAmount and user's balances.
   (b) Auction has closed - Computer unclaimed funds by dividing by closing price and subtracting claimedAmounts. Add extraTokens based on user's proportion of buyerBalance to buyVolume. If both tokens are approved, find ETH value and issue MGN. Reset buyerBalances and claimedAmounts and update user's balances.

YES DIVISION, NO OVERFLOW, NO ASYNC ISSUES.

Division

1. calculate unclaimedBuyerFunds
2. calculate tokensExtra
3. calculate MGNIssued

Results:

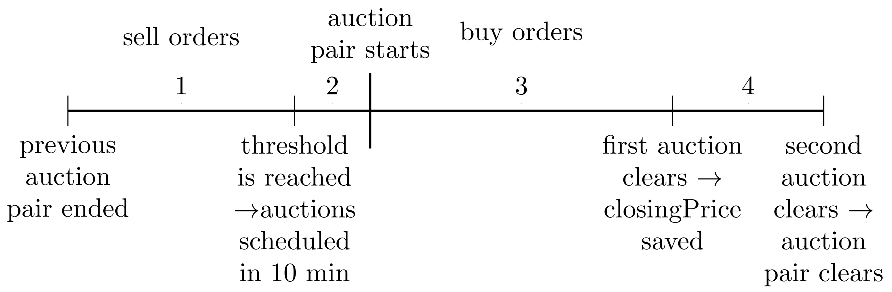
1. unclaimedBuyerFunds will be slightly smaller
2. extra tokens will be slightly smaller
3. MGNIssued will be slightly smaller

Chapter 3

States

3.1 Introduction

Each token pair consists of two opposite auctions. That auction pair has two variables in common: latest auction index and auction start. Here is a diagram illustrating the life of an auction during one auction index:


sell orders
auction pair starts
buy orders
1
2
3
4
previous auction pair ended
threshold is reached →auctions scheduled in 10 min
first auction clears → closingPrice saved
second auction clears → auction pair clears

During (1) and (2), only sell orders are accepted. Since auctionIndex has already been incremented, sell orders must use latestAuctionIndex and they go into sellVolumesCurrent. In (1), isWaiting is true, in all other states, it's false.

Once the threshold has been reached, auctionStart is set to now +10 minutes.

When that time is reached, both auctions begin. Sell orders for the current auction are no longer accepted (they must use latestAuctionIndex +1 and they go into sellVolumesNext).

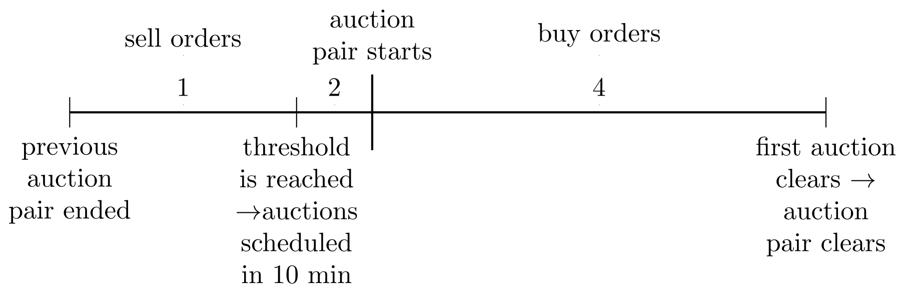
Once the first auction clears, the closingPrice is saved.

When the second auction clears, the entire auction pair clears (see Section 2.2.7, clearAuction).

3.1.1 Special case: “zero” auction

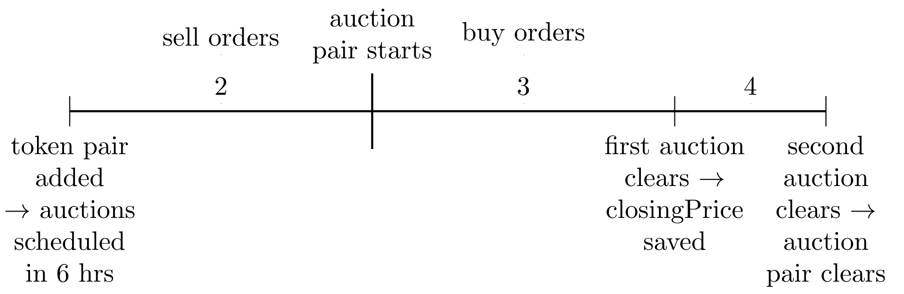
This case covers the situation when one auction is a 0 auction. A “zero” auction is one with sellVolume 0 (and consequently buyVolume 0).

In this case, the 0 auction is cleared from the start.


sell orders
auction pair starts
buy orders
1
2
4
previous auction pair ended
threshold is reached →auctions scheduled in 10 min
first auction clears → auction pair clears

3.1.2 Special case: adding a token pair

This case covers the situation when a token pair is added. The threshold is reached by definition, so period (1) isn't present.


sell orders
auction pair starts
buy orders
2
3
4
token pair added → auctions scheduled in 6 hrs
first auction clears → closingPrice saved
second auction clears → auction pair clears

3.2 postSellOrder

This function is complicated only in one regard: it must handle two different cases. These are disambiguated based on current auctionStart and isWaiting:

If we're in periods (1) or (2) in the diagrams, auctionIndex should be latestAuctionIndex and the volume should go to sellVolumesCurrent. This case is found if isWaiting is true (period (1)) OR if auctionStart > now (period (2)). Also, in period (1), we call scheduleNextAuction to check whether next auction can be scheduled.

If we're in periods 3 or 4, auctionIndex should be latestAuctionIndex +1, because we are posting into *next* auction (and volume should go to sellVolumesNext). This happens when the first case doesn't happen.

3.3 claimSellerFunds

ClaimSellerFunds just checks den > 0, which happens to handle all cases that we want to revert:

1. Particular auction hasn't started.
2. Particular auction hasn't closed.
3. It is a 0 auction.

If price reached 0, *returned* will be 0, so no balances will be added:

```
if (returned > 0) {
    balances[buyToken][user] += returned;
}
```

3.4 postBuyOrder

Several requirements must be met:

1. auctionIndex == latestAuctionIndex
2. isWaiting is false
3. auctionStart $\leq$ now
4. den == 0

and we can see that these requirements place are necessary and sufficient conditions.

3.5 claimBuyerFunds

We use the value of den to determine the state. If den $= 0$, auction is running. Otherwise, the auction has closed.