iZiSwap: Building Decentralized Exchange with Discretized Concentrated Liquidity and Limit Order

Jimmy Yin
iZUMi Finance
Jimmy@izumi.finance

Mac Ren
iZUMi Finance
MacRen000@gmail.com

ABSTRACT

This paper presents iZiSwap, a decentralized exchange protocol with the backbone of a new-designed discretized concentrated liquidity mechanism as the automated market maker (AMM) and the efficient support of limit orders. From the liquidity perspective, iZiSwap unifies the reusable concentrated liquidity from liquidity providers and the one-off liquidity from limit orders. It is more friendly to a trader to enjoy both the advantages of minimal price slippage due to the combined concentrated liquidity and the flexibility of limit order to seize desired transaction opportunity. The proposed protocol is efficient enough to run in a network situation similar to Ethereum and some implementation details under the Ethereum Virtual Machine environment are provided.

KEYWORDS

blockchain, decentralized finance, decentralized exchange, concentrated liquidity, order book

1 INTRODUCTION

Decentralized exchanges(DEX) are gradually becoming mainstream. Backed by the technique of blockchain, DEXs effectively solve the problem of middleman trust and are more robust to avoid the failure of a single point. As one of the famous representatives, Uniswap surpassed 500 billion dollars in cumulative trading volume recently in October 2021, since it launched in November 2018.

Submitted, November 2021, arXiv version

The rise of DEXs is inseparable from the development of automatic market maker(AMM) mechanism, which essentially describes a price curve which is dynamic as the reserves change. The reserves are provided by *liquidity providers* and act like the counterparty for the active traders, which allow digital assets to be traded in a permissionless and automatic way. Among various AMM designs, the *constant product formula* $x \cdot y = k$, as shown in Fig. 1, adopted by Uniswap V2[1], Pancake, SushiSwap, etc., is the most famous one, where x and y are reserves for the two tokens. During the swap procedure, the liquidity k is kept unchanged. When a trader tries to swap Δx he will receive Δy calculated from $(x + \Delta x)(y + \Delta y) = k$ without considering the transaction fees.

Although clean and neat, the constant formula suffers from the drawback of low capital utilization efficiency. Many improvements were proposed from different perspectives then to adapt to different application scenarios, such as Curve [3] that keeps price constant in the middle for the stable coin situation. In March 2021, Uniswap V3 proposed concentrated liquidity provision mechanism, which allows liquidity providers to place their liquidity in approximate any price range instead of the whole price space. The concentrated liquidity mechanism has significantly improved the capital utilization efficiency from the liquidity providers' perspective and gives minimal price slippage for the traders, bring AMM into a new era. In just two months, Uniswap V3 occupied about 50% of the whole market share.

From a trader's perspective, he can only swap tokens with current market price following the Uniswap V3 protocol. Such swapping procedure is similar to the market order mode in the order book protocol, which is often adopted and is closer to centralized exchanges. However, a market maker trader can place limit orders

waiting to be fulfilled with desired prices with the order book protocol, which is similar to the liquidity providers in AMM except that the liquidity is one-off. In general, the order book protocol is computational inefficient under the blockchain environment since the native implementation needs to loop over the potentially large order book. DEXs with support of limit orders, such as DyDx, usually seek blockchain with cheap computing power, such as Layer 2 solutions of Ethereum. The lacking of limit order in AMM such as Uniswap V3 makes it easy to lose fleeting trading opportunities.

In this paper, we present iZiSwap, a decentralized exchange protocol with the backbone of concentrated liquidity as AMM and the efficient support of limit orders with several significant features:

- *Discrete Concentrated Liquidity:* Instead of using constant product formula, the liquidity can only be placed on discrete ticks with different prices, which is more similar to the order book protocol. In each price tick, the liquidity acts following the *constant sum formula*. The design allows liquidity providers to concentrate their liquidity in their desired price range with a more clear price/liquidity relationship.
- *Grouped Limit Order:* A trader can place limit orders in certain price ticks. They are one-off liquidity that the token will not be swapped back once the price crosses the target. Limit orders on the same price tick are grouped together to improve efficiency. A trader needs to claim the swapped tokens if the order is fulfilled. If only a part of the limit orders on a price tick are fulfilled, the claim procedure follows "First claim first get" rule.

One of the main challenges is that the whole protocol needs to be implemented in the environment with limited computation and storage power, such as in the Ethereum network. The proposed iZiSwap protocol is efficient enough to run in the network situation similar to Ethereum and a full implementation under the Ethereum Virtual Machine environment and can be found in this Github repo[1].

1.1 Outline

Section 2 introduces the concentrated liquidity mechanism in the pioneering work Uniswap V3 and the motivations to our work. Section 3 introduces the design and key features of the iZiSwap protocol. In section 4, we provide some important details of the implementation of the protocol in the Ethereum Virtual Machine(EVM) environment. Finally, we conclude in section 5.

2 BACKGROUND AND MOTIVATIONS

Without further explanation, we consider one trading pair which consists of two kinds of tokens X, Y. We take X as the base token and Y as the quote one, that is, the term "price" is measure by how much Y can we get if we try to swap one unit of X. For example, when X is ETH and Y is USDC, the currently the price p means $1\ ETH = p\ USDC$.

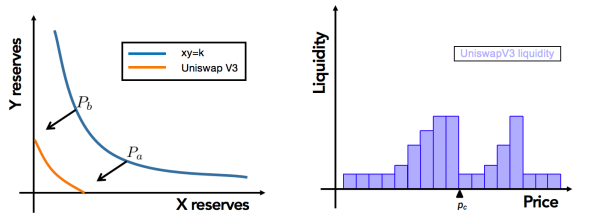
In the context of AMMs, including Uniswap V3, a protocol maintains a pool with reserves of token X and Y. Denote the reserves of X, Y as x and y respectively, and (x, y) is able to determine a unique price p. Hence, the protocol describes a price curve as the reserves change, where the instantaneous *price function* $p(x) = -dy/dx \geq 0$ represents how many Y tokens can we get with dx X tokens when the reserve of token X is x. Note that there is a minus sign since the tokens' amount of change is the opposite.

Alternatively, the protocol can essentially be described by a reserve curve as the price p changes. Specifically, suppose the price moves from current price p_c, with corresponding reserve (x_c, y_c), to p_b and $p_b > p_c$, the amount of change satisfies $\Delta y|_c^b = \int_{p_c}^{p_b} x'(p)p\,dp \geq 0$ and $\Delta x|_c^b = -\int_{p_a}^{p_c} x'(p)dp \leq 0$, i.e., the reserve (x_c, y_c) changes to $(x_c + \Delta x|_c^b, y_c + \Delta y|_c^b)$, where $x'(p)$ is the reserve density depending on the protocol.

People who place tokens into the reserves are called *liquidity providers*, they are the passive counterparty to the traders, who try to swap one token for the other.

2.1 Concentrated Liquidity in Uniswap V3

Uniswap v3 introduces the concept of concentrated liquidity, where instead of putting liquidity on the price range of $[0, \infty)$, liquidity providers can provide their liquidity in a custom finite range $[p_a, p_b]$. When the price crosses the boundary of the range, one of the reserves will be depleted.



(a) The relation between the reserves. (b) The liquidity distribution of price.

Figure 1: Illustration of the Uniswap V3 dynamics from different perspectives.

The key point for introducing concentrated liquidity in Uniswap V3 is to partition the whole price space and manage the liquidity in each piece separately. Specifically, the whole price space $[0, \infty)$ is partitioned by discrete *ticks* into *bins*, following the below definition.

Definition 2.1 (ticks & bins). When the pool is initialized with a price p_0, and a base step-size d, e.g. 0.0001, the i-th, tick is a price value that satisfies:

$$p_i = p_0 \cdot (1 + d)^i, \quad i \in (-\infty, \infty).$$

The i-th bin b_i is the price range with two adjacent ticks p_i and p_{i+1} as boundaries:

$$b_i = [p_i, p_{i+1}], \quad i \in (-\infty, \infty).$$

For each bin b_i, suppose the real reserves of X, Y are x_i and y_i respectively. The relationship between x_i and y_i is similar to the constant product formula except that they are added by "virtual" reserves $L_i/\sqrt{p_{i+1}}$ and $L_i\sqrt{p_i}$:

$$(x_i + L_i/\sqrt{p_{i+1}})(y_i + L_i\sqrt{p_i}) = L_i^2, \tag{1}$$

[1] https://github.com/izumiFinance/izumi-swap-core

where L_i is the *liquidity* , which can be defined alternatively by $L_i = dy/d\sqrt{P_i}$ and $P_i = -dy_i/dx_i$. Roughly speaking, the liquidity describes the price slippage rate, the larger liquidity, the lower price slippage and the better user experience. Notice that L_i and P_i are defined locally on b_i and P_i can be treated as the price P restricted in $[p_i, p_{i+1}]$. With L_i and P_i in hand, it is convenient to calculate x_i and y_i by solving

$$L_i = \sqrt{x_i y_i}, \quad P_i = y_i/x_i \tag{2}$$

A nice property for the bins b_i is that they can be merged sometimes. For a merged bin, i.e., a union of a set of continuous bins $b_u = \cup\{b_k, b_{k+1}, ..., b_{k+l}\}$, with price interval $b_u = [p_a, p_b]$, suppose the liquidity in each bin is the same L and the whole reserve $x = \sum_{i=k}^{k+l} x_i$, $y_{i=k}^{k+l} = \sum y_i$. The relationship between x and y is the merged version of Eq. 1:

$$(x + L/\sqrt{p_b})(y + L\sqrt{p_a}) = L^2. \tag{3}$$

In the implementation of the swapping procedure, this property is useful to fast locate the price P and the calculate the reserves x, y when the price P crosses more than one bins.

As shown in Fig. 1.a, the reserve curve in a certain bin can be obtained by shifting the $xy = k$ curve to intercept the axes geometrically. The global reserve curve is the splicing of the curves of all bins. From the perspective of liquidity distribution in terms of price, as shown in Fig. 1.b, it is obvious that the liquidity can be arbitrary shape at the granularity of bin. Notice that the liquidity is constant within a bin as price changes, which means the liquidity is uniformly distributed in a sense.

2.2 Motivations

The Uniswap V3 protocol is subtly designed. For liquidity providers, the protocol improves the capital utilization efficiency and for the traders, they can enjoy a lower price slippage rate.

However, the operating space for the traders is limited. If a trade wants to swap token X for Y, he can only accept the current price and operate in real time. This is similar to the *market order* in the order book protocol. If he wants to swap X for Y for a desired target price, i.e., *limit order*, he needs to wait until the price in the Uniswap pool reached it and then consumes the liquidity around. Otherwise, he needs to change his role from a trader to a liquidity provider and place liquidity around the target price. However, the provided liquidity will not withdraw automatically. When the price first crosses the target and then goes back and he does not withdraw the liquidity in time, he will lose the transaction opportunity.

From the perspective of liquidity, a limit order can be treated as a one-off liquidity placing on a single target price point, which can also increase the depth of liquidity. Therefore how to effectively make the two modes compatible in one system, under the blockchain environment, is a meaningful question.

3 IZISWAP PROTOCOL DESIGN

In this section, we introduce the design and key features for the iZiSwap protocol. The main goal is to incorporate the limit order into the concentrated liquidity system. We first introduce the discrete concentrated liquidity and then show how to use it to support limit order. The key design consideration is to meet user needs while being able to be implemented with as little time and space complexity as possible to meet the blockchain conditions such as Ethereum.

3.1 From Continuous to Discrete

We discuss from the liquidity and price perspective. The whole price space $(0, \infty)$ is again separated by discrete *ticks* p_i defined in Def. 2.1. For a liquidity provider, we limit that the reserve tokens X and Y, provided by liquidity providers and limit orders, can only be placed on the ticks p_i, i.e., boundaries on bins b_i.

Compared to the continuous case in Uniswap V3, in which the price changes continuously as the reserves change, price in iZiSwap can only choose discrete values. There are several reasons encouraging us to choose the discrete liquidity distribution.

- A limit order is essentially *discrete*, which specifies a target price instead of a price range. However, if we follow the liquidity defined in Uniswap V3, the liquidity measurement is not prefectly compatible for the discrete case. Recall that $L = dy/d\sqrt{p}$ actually defines a density over the price p, and in each bin b_i, the liquidity is uniformly distributed. If we incorporate the limit order into the system, we can not directly use L to measure the liquidity[2]. Although we can define a point mass in the price tick to view the liquidity in the integral form, the liquidity loses its intuitiveness and formal unity.
- Since the whole price range is already separated into small bins, the price variation in each bin is almost negligible, e.g. the practical value for d in Def. 2.1 is 1.0001. Replacing the continued price with a single one can avoid the vagueness in a small range. From the user experience perspective, a trader usually uses the price as the first information to judge the current market information. A discrete price tick is stable locally, which helps him make more accurate judgments. Although in practice, using the property mentioned in Sec.2.1, some bins can be grouped together to a large range to realize fast locate price and calculate the amount of swapped token. This can be realized for the discrete case as well.

Suppose the current price for the X/Y pair is p_c, where c is a feasible index. For the price tick $p_i < p_c$, there is only token Y on it, meaning that the token Y can be swapped to token X at the price p_i if p_c reaches it. This is similar to the *bid price* in the order book protocol. Symmetrically, for the price tick $p_i > p_c$, there is only token X on it and the token X can be swapped to token Y at that price, similar to the *ask price*. When $p_i = p_c$, token X and Y exist at the same time, which serve as the passive counterparty waiting for the traders to swap.

The swapping procedure is natural from the trader's view. In the most general case, if he wants to swap token X to Y of amount x, the protocol will first check if the reserves of Y on price tick p_c is enough. If enough, the trader will receive $x \cdot p_c$ amount token Y, otherwise, the current price will move left to the tick p_{c-1} and continue swapping the remaining with price p_{c-1}, until x is depleted. In section 4, we will introduce some skills to accelerate the swapping procedure.

[2]Not too rigorously, $d\sqrt{p} = 0$ if dy is small enough. This is similar to density and mass in probability theory.

The input tokens x are processed based on whether the liquidity from liquidity providers or limit orders is consumed.

3.2 Concentrated Liquidity

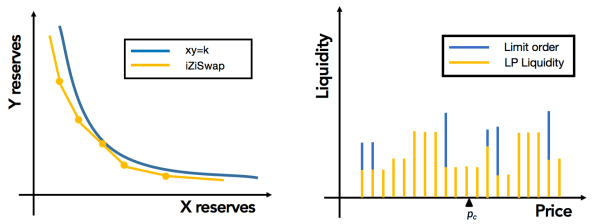
The first type of liquidity comes from liquidity providers. We first formally introduce the concept of *liquidity*, which is used to measure the trading depth as the price changes. It is an unified indicator to shielding the differences brought by the different size relationship between p_i and p_c. Besides, the liquidity is the key to manage several consecutive price ticks as a whole to improve efficiency, as we will show in the following. We define the *liquidity* in iZiSwap as follows:

Definition 3.1 (liquidity). Given a price tick p, the liquidity L on the tick is

$$L = x\sqrt{p} + y/\sqrt{p}, \tag{4}$$

where x and y are the amounts of token X and Y on that price tick respectively.

Obviously, the liquidity is defined locally. For different price ticks, the liquidity is separately managed and a liquidity provider can place all his tokens on a subset of the price ticks, so called *concentrated liquidity*. Liquidity on p_i are in the form of token Y for $p_i < p_c$ and token X for price tick $p_i > p_c$. For $p_i = p_c$, token X and Y may exist simultaneously. The pick of liquidity satisfies symmetry in a sense and satisfies some desired properties.



(a) The relation between the reserves. (b) The liquidity distribution of price.

Figure 2: Illustration of the iZiSwap dynamics from different perspectives.

- *Concentricity*: The required amount of token decreases as the price tick p moves away from the current price p_c for the same liquidity. Which means, if a liquidity provider wants to place the same liquidity on a set of price ticks, more tokens are concentrated around p_c. This is useful when combined with consecutive price ticks operations, as introduced in section 4, when a liquidity provider selects a relatively large set based on the prediction of the future price to keep the liquidity active.
- *Invariability*: The liquidity keeps invariant when swapping at the price tick, which means the value does not depends on the shares of token X and Y or the current price p_c, meeting the requirement to shield the differences caused by the these values. This is easily verified by plugging $\Delta y = -\Delta x \cdot p$ into $(x + \Delta x)\sqrt{p} + (y + \Delta y)/\sqrt{p}$. We call the swapping procedure that keeps the liquidity L invariant *constant sum formula* . With discrete price, the constant sum formula can also be suitable for the AMM model.

Fig. 2.a shows the reserve curve of iZiSwap. Compared with the constant product formula $xy = k$, or Uniswap V3 which is piecewise constant product, the curve is piecewise linear, where the slope is a constant when the swapping occurs on a certain price tick. The yellow lines in Fig. 2.b show the liquidity distribution in terms of price, which is similar to the order book if rotating 90 degree counterclockwise. Similar to Uniswap V3, the liquidity distribution can also be almost arbitrary at the granularity of the price tick interval delta d.

Discussion: comparison with Uniswap V3.
The constant sum formula is the discretization of the dynamics

$$Ldp = -\sqrt{p}dx, \quad Ldp = 1/\sqrt{p}dy, \tag{5}$$

where $L(p), x(p)$ and $y(p)$ are functions of reserves with respect of price p.

To verify this, we start from defining a liquidity density $L'(p)$ and token density $x'(p), y'(p)$ over the price space $[p_a, p_b]$ with current price p_c, which satisfy for all $p_i, p_j, p_a \le p_i \le p_c \le p_j \le p_b$:

$$\int_{p_i}^{p_j} L'(p)dp = \int_{p_c}^{p_i} y'/\sqrt{p}dp + \int_{p_j}^{p_c} x'\sqrt{p}dp. \tag{6}$$

Suppose now the current price p_c moves to $p_c + dp$, where dp is a small value. Before swapping, the amount of token X contained in $[p_c, p_c + dp]$ is $\int_{p_c}^{p_c+dp} x'(p)dp = x'(p_c)dp$. After swapping, the token X changes to token Y and we have $x'(p_c)dp = -dx$. By definition, we have

$$L'|_{p_c}dp = \int_{p_c}^{p_c+dp} L'(p)dp = \int_{p_c}^{p_c+dp} x'(p)\sqrt{p}dp = x'|_{p_c}\sqrt{p_c}dp,$$

which means

$$L'|_{p_c}dp = -\sqrt{p_c}dx. \tag{7}$$

Sinct the swapping procedure makes L invariant, we have

$$\int_{p_c}^{p_c+dp} x'(p)\sqrt{p}dp = \int_{p_c}^{p_c+dp} y'(p)/\sqrt{p}dp, \tag{8}$$

which implies $\frac{dy}{dx}|_{p_c} = p_c$. Plugging into Eq. 7, we have $L'|_{p_c}dp = 1/\sqrt{p_c}dy$.

The discretization can be obtained by choosing $L' = \sum \delta(p - p_i)$, where p_i are some special points containing liquidity.

Recall that in Uniswap V3, we have

$$L^u d\frac{1}{\sqrt{p}} = dx, \quad L^u d\sqrt{p} = dy, \tag{9}$$

where L^u is a constant. Essentially, both the constant sum formula and the Uniswap V3 defines a uniform liquidity density over the price space, however, with different symmetry, i.e., $\{d\frac{1}{\sqrt{p}}, d\sqrt{p}\}$ in Uniswap V3 and $\{\sqrt{p}dp, \frac{1}{\sqrt{p}}dp\}$ in constant sum formula. We leave the further exploration of the differences and connections as future work.

3.3 Limit Order

A limit order can be regarded as a one-off liquidity that comes from a trader instead of a liquidity provider. Based on the size relationship between the current price p_c and target price p, a trader should place different tokens, corresponding to buy or sell token X. When $p > p_c$, the trader sells token X for Y, and p is the *ask* price. Otherwise, when $p < p_c$, the trader buys token X with Y, and p is the *bid* price.

The liquidity definition is compatible for the limit order and the corresponding liquidity can be calculated by Def. 3.1. The liquidity can be added directly into the liquidity on that tick. Overall, the liquidity distribution of price is illustrated in Fig.2.b. When swapping, in order to accelerate the swapping procedure, all limit orders in one price tick p are grouped as a whole to avoid looping over potentially a large amount of small orders. After swapping, if the liquidity made by limit orders is used, the amount of related input tokens can be calculated in $O(1)$ time and then removed from the liquidity reserve and cached. *The traders need to claim the cached tokens swapped by the limit order.* Then the computational costs are amortized by the limit order providers.

To ensure fairness and correctness, some requirements must be satisfied. We list them here and the details for implementation can be found in the next section.

- *(General) Chronologically correct:* Once the price crosses a point, the current cached tokens waiting for claim on that point have nothing to do with the future orders. Specifically, when a trader places a limit order at time t and target price p, if the price crosses p at $t_1 > t$, he can claim the swapped tokens anytime after t_1 at price p. This prevents the arbitrage behavior that after the price first crosses through and then crosses back the target, a trader places a limit order with the old target price and declares that the previous swapped tokens fill his order.
- *First claim first get:* Once the current price crosses the target price p, all the limit orders placed before are fulfilled. The swapped tokens are locked to the corresponding orders and can be claimed by the trader at any time after the moment. However, when the current price reaches the target price tick but not crosses it, not all the limit orders are fulfilled. In this situation, we follow the "first come first served" rule. Everyone who places an order before can view the fulfilled part as if it first fulfilled his order, he can claim his part if no one beats him to it.

4 IMPLEMENTATION

We list some details for the implementation of the iZiSwap protocol under the Ethereum Virtual Environment. The full implementation can be found in the Github repo 1. The pioneering work Uniwap V3 efficiently implements some algorithms and data structure to support, including but not limited to, tick math and transaction fee management, we refer readers to [2] for more details.

5 RANGE SET LIQUIDITY

For a liquidity provider, liquidity is managed at the granularity of the *range set* in iZiSwap. The liquidity on adjacent ranges can be managed together to improve efficiency, especially when calculating the amount of tokens in the swapping procedure. A range set is composed of a series of adjacent price ticks. It is customary to use an interval $[p_a, p_b)$ that is closed at left and opened at right to represent a range set with price ticks $\{p_a, p_{a+1}, ...p_{b-1}\}$ in it. The range set $[p_a, p_b)$ and liquidity, i.e., tokens, on them form a *Liquidity*. In iZiSwap, liquidity on each price tick $p \in [p_a, p_b)$ must be a *same value* L . A *Liquidity* can then be represented by $Liq(a, b, L)$ and created by the *mint()* API, where $leftPt$, $rightPt$ and $liquidDelta$ correspond to a, b and L.

```
function mint(
    int24 leftPt,
    int24 rightPt,
    uint128 liquidDelta
)
```

When minting $Liq(a, b, L)$, the key step is to calculate the required amounts x and y for token X and Y. Suppose the current price is p_c, x and y can be calculated by the relationship between a, b and c.

- $p_c \geq p_b$: A liquidity provider only needs to provide token Y with amount:

$$y = \sum_{i=a}^{b-1} L\sqrt{p_i}. \tag{10}$$

  Since $p_i = (1+d)^i$, y is the sum of a geometric sequence which can be calculated in $O(1)$ with

$$y = L(\sqrt{1+d}^{b} - \sqrt{1+d}^{a})/(\sqrt{1+d} - 1). \tag{11}$$

- $p_c < p_a$: A liquidity provider only needs to provide token X with amount:

$$\begin{aligned} x &= \sum_{i=a}^{b-1} L_i/\sqrt{p_i} \\ &= L(\sqrt{1+d}^{-b} - \sqrt{1+d}^{-a})/(\sqrt{1+d}^{-1} - 1) \end{aligned} \tag{12}$$

- $p_a \leq p_c < p_b$: For the price tick p_c, both token X and token Y are feasible theoretically. Here we only require token Y in our implementation. Hence a liquidity provider needs to provide token X in range set $[p_a, p_c)$ while token Y in $[p_c, p_b)$. The amount can be calculated by Eq. 11 and Eq. 12 respectively.

We set $d = 0.0001$ in our implementation. Besides, we require that both a and b can be divisible by *tickSpacing* , where the latter equals to 10 or 30 depending on the property of trading pairs. As in Uniswap V3, this allows that the current price p_c efficiently jumps more than one tick at once in the swapping procedure and prevents the degenerate condition.

5.1 Remove Liquidity

Suppose a liquidity provider has a *Liquidity* $Liq(a, b, L)$, he can remove some liquidity $L' \leq L$ with API *burn()*.

```
function burn(
    int24 leftPt,
    int24 rightPt,
    uint128 liquidDelta
)
```

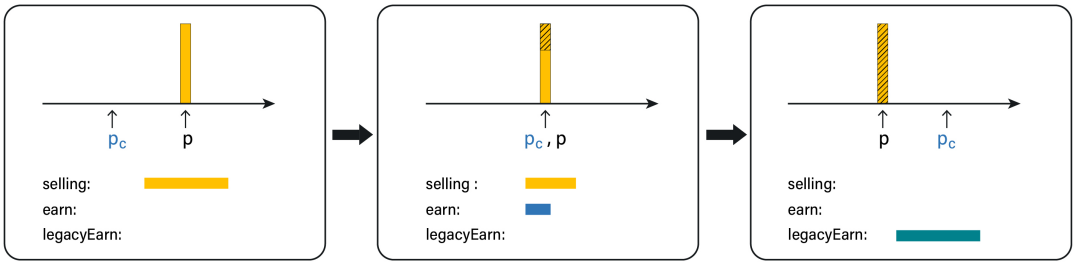


Figure 3: Illustration of some value of *PointOrder* structure as the current price p_c pass over the target price. When the grouped limit orders are partially filled, the earned tokens are stored in *earn*. Once the orders are all filled, it will stored in *legacyEarn*. Note that the current price $p_c > p$ is not necessary.

When removing liquidity, we first need to collect the transaction fees generated between the last claim and current time. We refer readers to [2] for details on how to efficiently maintain the related fees of a *Liquidity*. Then we need to calculate the amount x and y corresponding to the liquidity L', which follows from the same formula in the mint procedure, i.e., Eq. 11, 12, except one difference. When the range set $[p_a, p_b)$ covers the current price p_c, since there may be transactions before, the reserve for token Y on p_c may not sufficient for the amount required. In this case, we first pay token Y and they pay token X for the remaining part.

5.2 Limit Order

Similar to liquidity, a trader can place limit order at the price ticks that are multiple of *tickSpacing*. Specifically, a trader can place a limit order that

- buys X at any $p_k \in (0, p_c]$ and $k \ mod \ tickSpace = 0$.
- sells X at any $p_k \in [p_c, +\infty)$ and $k \ mod \ tickSpace = 0$.

The limit order will not be fulfilled with the liquidity provided by LP at once, otherwise a trader could directly swap to avoid paying more transaction fees.

5.3 Data Structure of Limit Order

There are two data structures related to limit order. For each price tick, there is a *PointOrder* structure and for each limit order , there is a *UserEarn* structure.

```
struct PointOrder {
    uint128 sellingX;
    uint256 accEarnX;
    uint128 sellingY;
    uint256 accEarnY;
    uint128 earnX;
    uint128 earnY;
    uint128 legacyEarnX;
    uint128 legacyEarnY;
    uint256 legacyAccEarnX;
    uint256 legacyAccEarnY;
}
```

We call a limit order *legacy* if, at a time before now and after the placement of it, all limit orders with same direction (sell X or sell Y) and same point, i.e., the grouped one, had been all fulfilled during one swap transaction. Distinguish the *legacy* orders from *unlegacy* ones are crucial to meet the "Chronologically correct" property. The key observation is that the "legacy" property is backward compatible in the time dimension, which means if a limit order becomes *legacy* at some time, it will be *legacy* forever. If we store the earned token in a separate space, the amount is enough for supporting all *legacy* order up to now to claim them.

The value *sellingX* represents the remaining amount of X that are selling at that tick currently. *accEarnY* represents the accumulated amount of Y obtained by selling X till now. *earnY* represents the *unlegacy*, i.e., the case that the grouped limit order is not fully fulfilled, amount of Y obtained by selling X and not be claimed currently. The *claim* procedure will be explained in the next section. *legacyEarnY* represents the *legacy* amount of Y obtained by selling X and not be claimed currently. The value *legacyEarnY* is updated by adding value of *earnY* when all X at this point are sold out after a swap, i.e., when *sellingX* value becomes 0. And note that when *legacyEarnY* is updated, the value of *earnY* will be set to 0. Fig. 3 illustrate the changing process when the update condition triggers. And the update condition is shown in Proc. 2. The value *legacyAccEarnY* is updated by latest *accEarnY* value when *legacyEarnY* is updated.

sellingY, *accEarnX*, *earnX*, *legacyEarnX*, *legacyAccEarnX* are defined symmetrically for Y.

```
struct UserEarn {
    uint256 lastAccEarn;
    uint256 sellingRemain;
    uint256 earn;
    uint256 legacyEarn;
}
```

UserEarn describes the information of swapped, or earned, tokens for a certain limit order. *lastAccEarn* records the value of *accEarn* in the corresponding *PointOrder* when the trader last time update the information. Notice that if the limit order sells X, *accEarn* records *PointOrder.accEarnY*. Otherwise, *accEarn* records *PointOrder.accEarnX*. *sellingRemain* represents how many token

X or Y are not sold out in the order till now. *Earn* and *unlegacyEarn* represent after several times of transactions and updating operation, how many *unlegacy* or *legacy* token X or Y that the trader has claimed but not withdraw, see explanation below.

5.4 Claim Swapped Tokens

The claim action means a trader *confirms* the swapped tokens. Note that the claimed tokens still lie in the contract, recorded by *earn* and *unlegacyEarn* above, and can then be withdrawn by the trader with an extra *withdraw* action. Suppose a trader places a limit order at p that sells X for Y. At any time, usually after several transactions at tick p, the trade can update the information in $UserEarn$ for the limit order with API *decLimOrderWithX()* or *decLimOrderWithY()* by setting $delta = 0$. The update procedure finishes the claim action and the pseudo code is shown in Alg. 1. The pseudo code for claiming the limit order that sells Y for X is symmetrical.

```
function decLimOrderWithX(
    int24 pt,
    uint128 deltaX
)
```

Algorithm 1 Update UserEarn

Let ue denotes $UserEarn$ for the order;
Let po denotes $PointOrder$ for the order;
if $po.legacyAccEarnY > ue.accEarnY$ **then**
  Let $e = ue.sellingRemain \cdot (1+d)^p$;
  $ue.legacyEarn \mathrel{+}= e$
  $ue.sellingRemain = 0$
  $po.legacyEarnY \mathrel{-}= e$
  $ue.lastAccEarn = po.accEarnY$
else
  Let $e1 = po.accEarnY - ue.lastAccEarn$;
  Let $e2 = ue.sellingRemain \cdot (1+d)^p$;
  Let $earn = min\{e1, e2, po.earnY\}$;
  $ue.earn \mathrel{+}= earn$;
  $ue.sellingRemain \mathrel{-}= earn/(1+d)^p$;
  $ue.lastAccEarn = po.accEarnY$;
  $po.earnY \mathrel{-}= earn$
end if

The *earn* and *legacyEarn* will not auto-update unless the trader actively updates it, and the update procedure for *earn* satisfies the "First claim first get" rule in the partial fulfilled case. The values $PointOrder.accEarnY$ $PointOrder.earnY$ $PointOrder.legacyEarnY$ and $PointOrder.legacyAccEarnY$ will be updated when a swap happens. It is worth noting that, for *unlegacy* part of unclaimed earnings, i.e., maintained in *earn*, the above pseudo code can not guarantee "Chronologically correct" property, which requires that limit orders placed later can not claim the swapped tokens fulfilled before by the others. However, once all the limit orders on the point are fulfilled, we will mark all fulfilled limit orders as legacy in $O(1)$ time via simply update *legacyAccEarnX* or *legacyAccEarnY* value as latest *accEarn* value on the corresponding point, and move all earnings by those limit orders to *legacy* part, as shown in Proc. 2. Since then, any limit order placed after that swap could not claim any earnings from limit orders placed before that swap.

5.5 Swap Procedure

In this section we show the procedure when a trader tries to swap tokens. We focus on the calculation about the amount of swapped tokens in the swapping procedure. We take the case that a trader tries to swap y for x for example and the other case can be obtained symmetrically. We first list different cases during the swapping process and then show the overall algorithm. The related API is $swapY2X()$, where *highPt* is the highest acceptable price of X and *data* is related to the token transfer procedure. Fig. 4 roughly shows the general procedure.

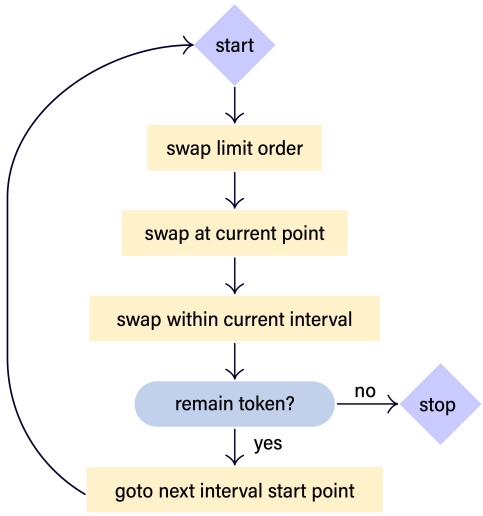


Figure 4: Illustration of the overall swap procedure. In an inner loop, it first fills the limit order and the liquidity on current point, and then jumps within an interval.

```
function swapY2X(
    address recipient,
    uint128 amount,
    int24 highPt,
    bytes calldata data
)
```

5.6 Case 1: Swap on an Limit Order

When a trader swaps using the liquidity of a limit order on tick p, the swapped token will follow from the price p. The contract will update the information in the corresponding *PointOrder*, i.e. *earnY*, *accEarnY* and *sellingX*. The value *sellingX* will constrain the amount of token X can be obtained in the next time. *earnY* and *accEarnY* will be used in the claiming action, as shown before. Proc. 2 shows the pseudo code for this case.

5.7 Case 2: Swap on a tick with liquidity

Suppose the trader swaps on tick p_a. The swapping procedure keeps the liquidity at least invariant and the swapped tokens also follows

Procedure 2 Swap On an Limit Order

procedure SwapOnLimOrder()
// p is the target price of the limit order.
// po is the related *PointOrder*.
// $totCostY, totAcquireX, y$ are references to record the amount of swapped and input token. These input parameters are similar in the other procedures
input: p, po, $totCostY$, $totAcquireX$, y;
$costY_{Lim} = min\{y, po.sellingX * p\}$;
$acquireX_{Lim} = min\{po.sellingX, y/p\}$;
y −= $costY_{Lim}$;
$po.sellingX$ −= $acquireX_{Lim}$;
$po.accEarnY$ += $costY_{Lim}$;
$po.earnY$ += $costY_{Lim}$;
$totCostY$ += $costY_{Lim}$;
$totAcquireX$ += $acquireX_{Lim}$;
if $po.sellingX = 0$ **then**
$po.legacyAccEarnY = po.accEarnY$
$po.legacyEarnY$ += $po.earnY$
$po.earnY = 0$
end if
end procedure

from the price p_a. After swapping, x and y should be non-negative. Proc. 4 shows the procedure.

Procedure 3 Swap On a Tick With Liquidity

procedure SwapTickL()
// a is the index of the price tick.
// L_a is the liquidity at p_a.
input: $a, L_a, totCostY, totAcquireX, y$;
output: $totCostY, totAcquireX, y$;
Let $currX_a$ denotes the amount of token X in tick p_a;
Let $currY_a$ denotes the amount of token Y in tick p_a;
Let $acquireX = min\{y/p_a, currX_a\}$;
Let $costY = acquireX * p_a$;
Let y −= $costY$;
Let $totCostY$ −= $costY$;
Let $totAcquireX$ += $acquireX$;
end procedure

5.8 Case 3: Swap Across a Range Set with Same Liquidity

In this case, liquidity at each tick $pin[p_a, p_b)$ is the same L and p_a is the current price. Obviously, liquidity on $[p_{a+1}, p_b)$ are all in the form of token X. There are two cases about the liquidity on tick p_a.

If all liquidity at p_a are X, then so is $[p_a, p_b)$. In this case, the trader pushes the price from p_a to p_b and the consumed amount y and swapped amount x can be calculated by 11 and 12.

If there are some token Y at p_a, this happens because of the transactions happen before. In this case, the trader will first consume all the x as in *Case*2 and the remaining liquidity in $[p_{a+1}, p_b)$ reduces to the above case. The pseudo code is shown in Proc. 4.

Procedure 4 Swap Across a Range Set with Same Liquidity

procedure SwapRangeL()
input: $a, b, L_a, d, totCostY, totAcquireX, y$;
output: $totCostY, totAcquireX, y$;
let $costY = L_a(\sqrt{1+d}^{b} - \sqrt{1+d}^{a})/(\sqrt{1+d} - 1)$;
let $acquireX = L_a(\sqrt{1+d}^{-b} - \sqrt{1+d}^{-a})/(\sqrt{1+d}^{-1} - 1)$;
let y −= $costY$;
let $totCostY$ −= $costY$;
let $totAcquireX$ += $acquireX$;
end procedure

5.9 Case 4: Stop in a Range Set

Different from Case 3, the trader can only buy a part of token X in $[p_a, p_b)$. In this case, we should first calculate how many ticks the trader can cover. Specifically, we need to find the largest t that all the liquidity in $[p_a, p_t)$, are swapped out. For simplicity, suppose that there are only token X in tick p_a. We have:

Procedure 5 Swap in $[p_a, p_b)$ And Check Whether Finished

procedure SwapInRange()
input: $a, b, L_a, d, totCostY, totAcquireX, y$;
output: $totCostY, totAcquireX, y, finished$;
$SwapTickL(a, L_a, totCostY, totAcquireX, y)$;
if $y == 0$ **then**
finished = true;
RETURN;
end if
Let $Y^b_{a+1} = L_a(\sqrt{1+d}^{b} - \sqrt{1+d}^{a})/(\sqrt{1+d} - 1)$;
if $y >= Y^b_{a+1}$ **then**
$SwapRangeL(a+1, b, L_a, d, totCostY, totAcquireX, y)$;
Let $a = b$;
if $y == 0$ **then**
finished = true;
RETURN;
end if
else
Let $t = \lfloor log_{\sqrt{1+d}}[\frac{y}{L_a}(\sqrt{1+d} - 1) + \sqrt{1+d}^{a}]\rfloor$;
$SwapRangeL(a+1, t, L_a, d, totCostY, totAcquireX, y)$;
Let $a = t$;
if $y == 0$ **then**
finished = true;
RETURN;
end if
$SwapTickL(t, L_a, totCostY, totAcquireX, y)$;
finished = true;
RETURN;
end if
finished = false;
end procedure

$$y >= L(\sqrt{1+d}^{t} - \sqrt{1+d}^{a})/(\sqrt{1+d} - 1), \tag{13}$$

where y is the amount the trader pays, solving the inequality we have:

$$t = \lfloor log_{\sqrt{1+d}}[\frac{y}{L}(\sqrt{1+d}-1) + \sqrt{1+d}^{a}]\rfloor \quad (14)$$

After calculating t, the trader first buy all token X in $[p_a, p_t)$. For the remaining Y, since t is the largest value satisfying 13, liquidity at tick p_t is enough to consume all the remaining Y, which follows the Case 2. Obviously, when this case happens, the whole swapping procedure has finished. The pseudo code for the procedure was shown in Proc. 5.

5.10 Overall Swap Procedure

The limit order and *Liquidity* divide the how price space $[p_c, \infty)$ into small range sets with different amount of liquidity on them. The whole swapping procedure will loop over the range sets with the above procedures and the overall pseudo code is shown in ALg. 6.

Algorithm 6 Swap from token Y to X

```
// y and HighestPoint corresponds to amount and highPt
input: y, HighestPoint;
Let p_a be the current price tick of X.
Let totAcquireX = 0;
Let totCostY = 0;
while y > 0 or a < HighestPoint do
    Let po be the PointOrder of p_a.
    if po.sellingX > 0 then
        SwapOnLimOrder(a, po, totCostY, totAcquireX, y);
    end if
    if y == 0 then
        break;
    end if
    Let p_b be the right nearest tick of a limit order or a boundary
    of a range set.
    Let b = min{b, HighestPoint};
    // Now the liquidity in [p_a, p_b) must be the same.
    Let L_a denotes liquidity of tick a;
    Let finished =
        SwapInRange(a, L_a, d, totCostY, totAcquireX, y);
    if finished == true then
        break;
    end if
end while;
Update the current price for X with p_a.
Transfer tokens according to toCostY and toAcquireX.
```

6 CONCLUSION

In this paper, we propose a novel Decentralized Exchange protocol iZiSwap. The protocol unifies the recyclable liquidity provided by the liquidity providers and the one-off liquidity provided by the traders' limit orders, making the trading depth deeper than any one alone. From the trader's perspective, a trader can enjoy both the minimal price slippage due to the depth and the flexibility of limit order to seize desired transaction opportunities. The proposed protocol is efficient enough to run in the network situation similar to Ethereum and a full implementation under the Ethereum Virtual Machine environment is provided.

REFERENCES

[1] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2020. Uniswap v2 core. *URl: https://uniswap.org/whitepaper. pdf* (2020).
[2] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. 2021. Uniswap v3 Core. (2021).
[3] Michael Egorov. 2019. StableSwap - efficient mechanism for Stablecoin liquidity. (2019). https://curve.fi/files/stableswap-paper.pdf