Poolshark

alphak3y

Abstract
Poolshark is a noncustodial directional automated market maker implemented for the Ethereum Virtual Machine. In comparison to its predecessors, it allows liquidity providers to adopt directional strategies using a closed-form solution for liquidity position tracking.

Keywords
Directional Automated Market Maker — Directional Liquidity — Cover Liquidity — Limit Liquidity

*__Corresponding author__: alphak3y@protonmail.com January 10th, 2023

Contents

Introduction

Automated market makers (AMMs) are defined as software that prices liquidity using a pre-defined algorithm. In the context of decentralized finance, $x * y = k$ is often associated with constant-function market makers (CFMMs). This pricing mechanism is commonplace due to its well-defined blockspace usage. For a pool in which two tokens are paired with one another, the formula maintains that the total value of the first token in the pool must always equal the total value of the second token. In order to provide efficient liquidity swapping and stable transaction costs, tokens provided to an AMM use a strategy called merging to form all liquidity provider positions into a single liquidity pool.

With the introduction of range-bound liquidity[1], liquidity providers could define a custom price range over which their liquidity will trade while still being efficiently aggregated inside a liquidity pool. This results in increased capital efficiency for liquidity providers with the trade-off of making more liquidity available near market price.

Directional automated market makers (DAMMs) are an extension of automated market makers wherein liquidity in the pool is non–recyclable and discrete liquidity curves exist for each trading direction. The intention of using such an AMM is to shift focus from liquidity providers capturing fees to liquidity providers capturing volatility. The former has been proven to work on stable pairs and produce consistent revenue while the latter is inherently useful when price action for a pair is non-stable.

Directional automated market makers in short only capture one side of a pair over some bound range, allowing them to fill more similarly to a limit order than predecessor AMMs. As of current, AMMs in decentralized finance have only offered buy-and-sell liquidity positions, whereas Poolshark provides buy-only and sell-only liquidity positions.

Motivation

Bidirectional liquidity here is defined as a smart contract allowing liquidity to be traded from either

$$token_0 => token_1 \ \vee \ token_1 => token_0$$

This mechanism is commonly associated with the term AMM as of this writing. The way that such a smart contract trades assets using $x * y = k$ math is based both on market demand as well as supply from liquidity providers. Here a mean-reversion trading strategy is applied, meaning we assume there is some stable trend over the timeframe in which liquidity is provided.

Within bidirectional liquidity, traders execute liquidity swaps and set the pool price by shifting the pool reserves. It is worth noting the value of this smart contract design is to enable continuous liquidity on-chain and collect trading fees for doing so.

This style of AMM can become unfavorable as price divergence between pool assets increases. This is due to the reduced liquid market value of the deposited assets, with the liquidity provider taking the losing side when there is a large directional move. This phenomenon is commonly referred to as *impermanent loss*, however for sake of simplicity, it is easier to define this impermanent loss as

$$ASP_{market} - ASP_{LP position}$$

where the ASP (i.e. average selling price) difference between the liquid market value and the ASP of the LP (i.e. liquidity provider) position at some constant price.

If we take 1 ETH and spread it across the price range of 3000 DAI per ETH to 5000 DAI per ETH using range-bound liquidity and $x * y = k$, that 1 ETH will be sold at approximately 3868.23 DAI per ETH.

When the market price of ETH becomes 5000 DAI per ETH, a user with range-bound liquidity between 3000 DAI per ETH and 5000 DAI per ETH will experience a value loss of 1131.77 DAI. Here we assume the user to liquidate these holdings to 100% DAI at the price of 5000 DAI per ETH with zero price slippage.

To cover these losses, we highlight here a few key options for the liquidity provider:

1) Choose to not provide bidirectional liquidity and favor a buy-and-hold strategy

2) Buy-and-hold an option, perpetual, or other derivatives which are inherently less liquid than naked assets

3) Buy-and-hold the underlying asset in the bidirectional liquidity pool which the LP expects to appreciate over time

1), 2), and 3) in the current context of decentralized exchanges involve diminishing liquidity available for assets such as WETH, DAI, and other fungible tokens in favor of improving individual portfolio outcomes.

Having a way to express 3) by providing range-bound liquidity to a smart contract means A) liquidity can be provided by both traders and fee collectors B) bidirectional liquidity providers require less active management to become delta neutral C) more price diversity in the market with discrete liquidity curves for each trading direction.

This is the clear motivation for the introduction and existence of directional automated market makers (DAMMs): to support the needs of on-chain liquidity seeking a gas-efficient buy-and-hold strategy across various market conditions.

1. High-level overview

Directional liquidity AMMs, defined as a smart contract allowing for liquidity to be exclusively traded between

$$token_0 => token_1 \oplus token_1 => token_0$$

Directional automated market makers have two key traits:

1) Irreversible liquidity swaps

2) Discrete liquidity curves

The Poolshark Protocol delivers these qualities with two definitive variants, *Cover Pools* and *Limit Pools*.

Limit Pools allow for price priority, where liquidity providers can undercut the market on one side and traders receive the lowest price possible.

Cover Pools enable the liquidity provider to unlock liquidity as the price moves against the asset which they provided to the liquidity pool.

Limit Example :

Bob provides ETH at a 0.01% discount to the rest of the market to exit his ETH position.

Cover Example :

Alice provides DAI in a range of 3000 to 5000 DAI per ETH to offset the potential impermanent loss.

1.1 Limit Pools

In the current bidirectional range AMMs as of writing, when the LP position has reached the desired price, it needs to be withdrawn. This is a drastic departure from how traditional limit orders work on centralized exchanges. *Limit Pools* provide a closed-form solution for mirroring one-way fills: a *Claim Tick*.

The concept of a claim tick is such that there is some price tick T up to which our position has been filled since the time of creation. As the pool crosses each tick and fills the user's Position P, it will mark the global total fee growth at which the tick was crossed.

Since there exist discrete curves for each trading direction, the smart contract can retrieve the lowest price for each trading direction. As a result, users executing liquidity swaps will receive the best price available in the smart contract. Liquidity providers receive the benefit of irreversible "take-profit" range orders that can be used for a variety of scenarios.

1.2 Cover Pools

To cover this loss of 1131.77 DAI as mentioned in the *Motivation* section, the solution proposed in this whitepaper is to purchase the same amount of ETH back from the market that was sold over this range at the same average price of 3868.23 DAI per ETH.

For each bit of ETH that the liquidity provider receives from the market, they ought to hold this ETH to profit from a continual directional increase in price (i.e. impermanent gain). From this point on we shall refer to this description as a *Cover Pool*.

Cover Pools gradually unlock liquidity as the price continues to move against the asset the LP provided. In the case here, the user provides 3868.23 DAI to create a *Cover LP Position* with a lower bound of 3000 DAI per ETH and an upper bound of 5000 DAI per ETH. As the price continues to increase from ETH to DAI and decrease for DAI to ETH, the smart contract will unlock more of the user's LP position to be auctioned off to the market.

The smart contract requires a reference price to determine which liquidity should be unlocked at any given time. One way to inform this reference price is via a time-weighted average price (TWAP) oracle. Oracle data will ideally originate on-chain to maximize the trustlessness of the protocol. A longer TWAP sample (e.g. last 1 hour) will make the *Cover Pool* less reactive to changes in price. Conversely, a shorter TWAP sample (e.g. last 10 minutes) will make the *Cover Pool* more reactive to price changes but better able to handle short-term volatility.

In addition to creating a positive spread on the latest TWAP price, the *Cover Pool* will also need to dynamically adjust to the current market price, which is likely plus or minus the current TWAP sample. To account for this adjustment, *Cover Pools* will implement the concept of *Gradual Dutch Auctions* to adjust the pool price each consecutive block.

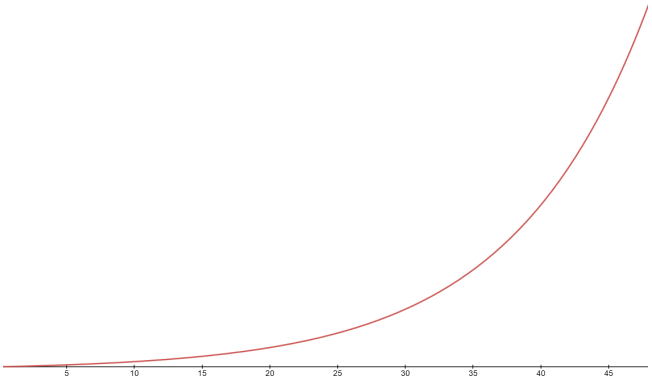

Figure 1. Cover example auction curve

Each time the TWAP moves, a liquidity auction will initiate and create a positive spread against the TWAP sample equal to the *tickSpacing*. For example, if the ETH-DAI *Cover Pool* in our example has sourced a TWAP 0.1% (10 basis points) above 3000 DAI per ETH the ETH to DAI price will start at 3000 + 0.2% and the DAI to ETH price will start at 3000 DAI per ETH. This is done to get slightly ahead of the current market price so the pool can ensure those with a liquidity position in the pool get filled. To be clear, this positive TWAP spread does not alter the price at which the *Cover LP Position* gets filled but rather slightly speeds up the liquidity unlocking to increase the likelihood of user liquidity getting filled.

When liquidity is unlocked by the TWAP, it is unlikely this TWAP will fit the natural market price without any adjustment. To account for this delta, *Cover Pools* will implement the concept of *Gradual Dutch Auctions* to increase or decrease the pool price as needed each consecutive block a tick or set of ticks has its liquidity auctioned off. Increases in pool price will happen over time while decreases in pool price will occur as a result of slippage from incoming swaps against the *CoverPool*.

2. Implementation Details

Directional liquidity relies on a time-keeping mechanism to tell liquidity providers the price tick closest to fill completion, which we will refer to as the *Claim Tick*, containing any other data needed to fulfill the successful removal of liquidity from the smart contract. Given the *Claim Tick* is known and verified as correct (i.e. it has seen some update since the user created their Position), $x * y = k$ math can be used to determine the amount a Position should receive of both *token*0 and *token*1.

The next tick in progression after the *ClaimTick* in question ought to not have an $epochLast$ less than or equal to that of the liquidity position's stamped $epochLast$. Otherwise, the user will have claimed from the wrong tick and the smart contracts should revert the transaction or continue naively cycling to find the correct *ClaimTick*.

This simple mechanism of liquidity position tracking persists throughout, however, there are a variety of special cases to handle. Where the pool price is at the time of mint, collect, or burn will mean a different approach to updating liquidity values at $Tick_{\text{lower}}$ and $Tick_{\text{upper}}$ for the *Position*. It is important to note that *Cover Pools* and *Limit Pools* have completely separate mechanics outside of the similarities outlined in this section.

2.1 Limit Pool Contracts

[Coming in v1.1.0]

2.2 Cover Pool Contracts

When first launching a *Cover Pool*, we must first ensure that the *Range Pool* we are referencing has a sufficient TWAP sample length for any user minting a *Cover LP Position*. If our pool uses a TWAP sample length of 10 minutes, we must ensure there is 1 minute's worth of data currently available.

Once this check passes, we can initialize the pool and allow users to start adding liquidity. By initializing the pool, $Tick_{min}$ and $Tick_{max}$, the minimum and maximum price ticks, are created in addition to a $Tick_{latest}$, a *Tick* representing the current TWAP sample.

To prevent the state of any liquidity position from being fragmented, we must not allow users to add liquidity on both sides of the current TWAP. For LPs seeking to provide *token*1 in exchange for *token*0, they must provide liquidity at a price greater than the current TWAP, assuming the pool price is represented as

$$price_{pool} = \frac{token_1\ reserves}{token_0\ reserves}$$

Likewise, LPs seeking to provide *token*1 in exchange for *token*0 must provide liquidity at a price lesser than the current TWAP. Users seeking to exchange at a more favorable price than the current TWAP should instead use a *Limit Position*.

Having discrete liquidity curves means having separate liquidity data for each swap direction. We will refer

to this distinction as $pool0$ and $pool1$, for which $pool0$ will contain $token0$ and $pool1$ will contain $token1$. *Position* and *Tick* data will also have distinct contract storage to track the activity of each pool and apply exactly-once liquidity position fills.

2.2.1 Position Updates

Cover introduces a brand new concept of $amountIn\Delta$ and $amountOut\Delta$. These values represent filled amounts and unfilled amounts respectively for an auction or auctions. $amountIn\Delta$ represents the incoming token amounts to the *CoverPool*, while $amountOut\Delta$ represents outgoing token amounts being offered to traders.

Additionally we introduce the concept of $amountIn\Delta_{max}$, which is the max incoming token amount the liquidity provider(s) expects over some range. In the same vein, $amountOut\Delta_{max}$ is the max outgoing token amount the liquidity provider(s) expect to be consumed. When the edge of a position range is reached, we proportionally distribute $amountIn\Delta$ based on the $amountIn\Delta_{max}$- and $amountOut\Delta_{max}$- present at that tick.

Upon minting an LP position in a *Cover Pool*, a $+\Delta_{liquidity}$, or *positive liquidityDelta*, is added to $Tick_{start}$, while conversely a $-\Delta_{liquidity}$ is added to $Tick_{end}$. If the LP mints a position for $token1$ to $token0$, the $Tick_{start}$ will be at the lower bound, and the $Tick_{end}$ will be at the upper bound. Vice versa will be true if the LP is depositing $token0$ for $token1$.

Utilizing standard $x*y=k$ math and virtual reserves, we can calculate the amount of liquidity L that a *Position* holds using the following formula[1]:

$$L=\sqrt{(x+\frac{L}{\sqrt{price_{\text{upper}}}})(y+L\sqrt{price_{\text{lower}}})}$$

where $\text{price}_{\text{lower}}$ is strictly less than $\text{price}_{\text{upper}}$ and L represents the square root of k in the equation $x*y=k$.

If there isn't already a *Position* that exists for a user with a chosen $Tick_{\text{lower}}$ and $Tick_{\text{upper}}$, liquidity will be added to the respective ticks and the *Position* will have two values initialized, *epochLast* and *claimPriceLast*. The former tells us at what *Epoch* the position was created, while *claimPriceLast* tells us up to what price the user has claimed their expected *Position* token output.

If a *Position* already exists for a user with a chosen $Tick_{\text{lower}}$ and $Tick_{\text{upper}}$, the current *Position* must be updated first before we can create the new *Position*. This is implemented so the *Position* can be partially filled and then have more liquidity deposited thereafter. Once the *Position* is updated, it will be stored with a $Tick_{\text{lower}}$ representing what remains if $token1$ was initially deposited into the *Positon*, or $Tick_{\text{upper}}$ if $token0$ was initially deposited into the *Position*.

2.2.2 Tick Auctions

For a single constant *Tick* T, it is unlikely the current TWAP will fully inform the pool of what is a competitive market price. To remedy this, the concept of a *TickAuction* is implemented to naturally fit the *Cover Pool* price to that of other bidirectional liquidity pools.

For simplicity the initial version of Cover applies a linearly increasing auction price starting at the price tick being unlocked and improving *tickSpread* basis points. For example, if the *tickSpread* is 20 basis points, the auction curve will give a maximum of 20 basis points of price improvement. If the auction is still not filled once that maximum is reached, any remaining amount will be marked unfilled which increases the $amountOut\Delta$ being rolled over to the next auction.

2.2.3 Price Tracking

Price is tracked using an on-chain time-weighted average price oracle as determined by the user launching the pool. Each time the *auctionLength* or $\frac{3}{4}twapLength$ is passed, whichever is lesser, we will sync the state of each price tick between the previous TWAP tick and the new TWAP tick, progressing the *ClaimTick* for liquidity providers in-range. Curve math can then be used to determine how much each of $token0$ and $token0$ the *Position* is owed. This is accomplished by checking *claimPriceLast*, measuring the $amountIn\Delta_{max}$ and $amountOut\Delta_{max}$ covered in this latest position update, and then granting the proper $amountIn\Delta$ and $amountOut\Delta$ present at the tick.

Once the "syncing" process is carried out, a new *StashTick* will be initialized for any range that is partially crossed. This *StashTick* will contain any Δ amounts needed for users to exit their *CoverPosition*. The price for $pool1$ will start at a positive spread of *tickSpacing* in favor of the trader, while the price for $pool0$ will do the same relative to the *newLatestTick*. This creates the potential for the liquidity unlock to get slightly ahead of the TWAP and thus increase the chances of successfully filling a given auction.

2.2.4 Delta Math

Δ amounts are aggregated across auctions onto either a *StashTick* for within a position range or a *FinalTick* for when a position end boundary is crossed. Separate Δ_{max} values will be kept for 1) uncrossed position boundaries or Δ_{max}-values 2) crossed position boundaries Δ_{max} and 3) stashed tick boundaries or Δ_{max}stashed.

The Δ values for a given *Stash Tick* can be determined via the following equation:

$$\Delta\, stashed = \Delta * \frac{\Delta_{max}stashed}{\Delta_{max}stashed+\Delta_{max}final}$$

The Δ values for a given *Final Tick* can be determined via the following equation:

$$\Delta\, final = \Delta * \frac{\Delta_{max}final}{\Delta_{max}stashed+\Delta_{max}final}$$

Acknowledgments

Major thanks to all the wonderful people I have met at each and every Ethereum conference. Without your inspiration and dedication to the space, this wouldn't have been possible. Together we can build a brighter future.

Big thanks to 0xnexusflip for being a constant sounding board and equal thanks to everyone on my team who believed in me.

Shouts out to kassandra.eth for being my initial gateway into the concept of crossing price ticks and managing liquidity.

Big thanks to Geoff Hamilton from Variant for pushing me to talk to other protocols more and receive genuine feedback.

Shouts out to Dan Ugolino for giving me the conviction bidirectional liquidity would not work for decentralized options protocols.

Big thanks to Zach Hamm for being a constant advocate for creating better open and permissionless financial tooling for everyone.

References

[1] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. 2021. Uniswap v3 Core. Retrieved Jan 10, 2023, from https://uniswap.org/whitepaper-v3.pdf

[2] Frankie, Dan Robinson, Dave White, and andy8052. 2022. Gradual Dutch Auctions. Retrieved Jan 10, 2023, from https://www.paradigm.xyz/2022/04/gda