Maverick AMM

Maverick Research Team

March 2023

1 Introduction

Maverick AMM is a range-based AMM where LPs can choose to have their liquidity automatically move to stay near the price. This liquidity movement increases capital efficiency and allows LPs to make directional bets on price while still collecting fee.

In order to provide liquidity in Maverick, LPs make two choices:

- Which range of prices to LP in
- How their liquidity should shift in price as price shifts

LPs can select between four Modes when adding their liquidity:

- **Mode Right** - liquidity moves with price as price increases, but doesn't move when price decreases
- **Mode Left** - liquidity moves with price as price decreases, but doesn't move when price increases
- **Mode Both** - liquidity moves with price as it increases and decreases
- **Mode Static** - liquidity does not move

This document explains how liquidity is added to the smart contract and how liquidity is moved when LPs select one of the three movement Modes.

2 Bins

The smallest available price range is a "bin". The bin spacing is a configurable parameter that is set when a pool is initialized. A typical bin spacing for a volatile asset pair is 2%.

A bin has a lower price of p_l and an upper price of p_u. The swap invariant for a bin is

$$L^2 = \left(B + \frac{L}{\sqrt{p_u}}\right)\left(A + L\sqrt{p_l}\right), \tag{1}$$

where B is the amount of base token in the bin and A is the amount of quote token in the bin.

2.1 Bin Structure

The AMM smart contract stores the following properties for each bin:

- `binID` - Unique ID of the bin
- `reserveA` - Quantity of quote tokens in the bin
- `reserveB` - Quantity of base tokens in the bin
- `kind` - One of `STATIC`, `RIGHT`, `LEFT`, `BOTH`,
- `lowerTick` - Defined as $\texttt{lowerTick} = s_{tick} \log_{1.0001}(p_l)$, where $s_{tick} \in \mathbb{N}_{>0}$ is the tick spacing of the pool
- `mergeId` - ID of bin that this bin has merged in to
- `totalSupply` - Quantity of LP tokens that have been minted for this bin
- `mergeBinBalance` - Quantity of mergeID bin LP tokens this bin has a claim to (only non-zero when bin has been merged)
- `balances` - LP token balance for each Position NFT ID

2.2 Adding Liquidity to a Bin

A user can choose to add liquidity to a bin that is either all quote, all base, or a mix of base and quote. When adding liquidity, the user cannot change the price, so they must add base and/or quote in the same proportion that the bin already contains.

When a user becomes an LP by adding liquidity, LP tokens are minted and increment the bin's `totalSupply` amount. The number of LP tokens (`newSupply`) that are minted is calculated by expressing the base and/or quote being added as a percentage of the bin's existing reserves and multiplying that by the current `totalSupply` of LP tokens.

There are several possible states that a bin might be in when an LP adds liquidity. Table 1 summarizes how many LP tokens are minted depending on the state.

To add liquidity to Maverick, an LP will first mint a free NFT Position. The NFT Position has a unique ID and this ID is what is associated with a user's LP token balance. Users can choose to mint a new Position at any time, but one Position is sufficient to store all of the LP token balances across all bins and pools in which an LP is participating.

2.3 Moving and Merging Bins

Non-static bins (i.e., bins with a `kind` property of `RIGHT`, `LEFT`, or `BOTH`) may move right or left with price as traders swap with the pool. When this happens, the `lowerTick` value of the bin changes. The new `lowerTick` value is set to the new tick position of the bin.

If a non-static bin moves to a tick where there is already another bin of the same kind, the contract executes a merge procedure described below. At the end of that

Table 1: Number of LP tokens `newSupply` minted for a contribution of a new quote tokens and/or b new base tokens to the bin. a or b has to be non-zero.

	$p < p_l$	$p_l < p < p_u$	$p > p_u$
`reserveA` $== 0$, `reserveB` $== 0$	a	$\max\{a, b\}$	b
`reserveA` $== 0$, `reserveB` > 0	-	-	$\frac{b \text{ totalSupply}}{\text{reserveB}}$
`reserveA` > 0, `reserveB` $== 0$	$\frac{a \text{ totalSupply}}{\text{reserveA}}$	-	-
`reserveA` > 0, `reserveB` > 0 $a > 0$, $b > 0$	-	$\min\left\{\frac{a \text{ totalSupply}}{\text{reserveA}}, \frac{b \text{ totalSupply}}{\text{reserveB}}\right\}$	-
`reserveA` > 0, `reserveB` > 0 $a > 0$, $b == 0$	-	$\frac{a \text{ totalSupply}}{\text{reserveA}}$	-
`reserveA` > 0, `reserveB` > 0 $a == 0$, $b > 0$	-	$\frac{b \text{ totalSupply}}{\text{reserveB}}$	-

procedure, only one of the two bins will still be "active" and the other bin's reserve will be transferred to that active bin. As described in the next section, the non-active, "merged" bin, will still exist and will still be the `binID` the user references to withdraw their liquidity.

As part of the movement/swapping procedure, the contract ensures that there will never be two active bins of the same `kind` at the same `lowerTick` value. When a bin is moved, the contract checks to see if there would be more than one active bin of the same `kind` at the same `lowerTick`. If there would be, then a merge procedure happens: the `mergeId` of merged bin is updated with the `binID` of the bin that is active at this tick.

For the sake of discussion, consider an example where the contract is merging bin with `binID` = i_m into bin with `binID` = i_a. Bins can only merge when they only hold either all base or all quote. For this example, assume both bins are all quote. Table 2 summarizes the state of these two bins before and after the merge.

The notable aspect of this process is that the merged bin now holds an LP token balance of `mergeBinBalance` in the active bin, but the LP `balances`, which track the LP balance of each user in each bin, have remained the same. The active bin now has a `totalSupply` of LP tokens equal to its original `balance` and the `mergeBinBalance` that the merged bin now holds in it. The last row of the table

Table 2: State change of two bins as they merge together.

binID	i_m pre merge	i_a pre merge	i_m post merge	i_a post merge
reserveA	a_m	a_a	0	$a_a + a_m$
reserveB	0	0	0	0
lowerTick	t_m	t_a	t_a	t_a
mergeId	-	-	i_a	-
totalSupply	LP_m	LP_a	LP_m	$LP_a + LP_a \frac{a_m}{a_a}$
mergeBinBalance	-	-	$LP_a \frac{a_m}{a_a}$	-
balances[user]	bal_m	bal_a	bal_m	bal_a
balances[0]	0	0	0	$LP_a \frac{a_m}{a_a}$

shows that, in the binID = i_a bin, the LP token balance of any merged bins gets stored in index 0. This is a special reserved index value that no user will be assigned.

Tracking the balances this way makes merges computationally tractable on chain because it means that the contract does not have to iterate through all LPs' positions to update balances on a merge. Instead, as discussed in the next section, the LPs still claim their LP tokens on the original, now-merged, bin, and the contract recursively traverses merged bins to get to the active bin where the reserves are all held.

The mechanisms that cause a bin to move are related to the time-weighted-average-price (TWAP) and are discussed in the Pool section.

2.4 Removing Liquidity

An LP can remove liquidity from a bin by passing to the contract their NFT Position ID, the bin's binID, and the amount of LP balance they want to remove from that bin.

For an active bin, the process is straightforward. The contract checks to make sure the Position has at least the amount of LP balance the user is trying to withdraw. If the balance is sufficient, the bin disperses a pro-rata amount of reserve out back to the user:

$$A_{out} = \frac{\texttt{amount}}{\texttt{totalSupply}}\texttt{reserveA}, \tag{2}$$

$$B_{out} = \frac{\texttt{amount}}{\texttt{totalSupply}}\texttt{reserveB}. \tag{3}$$

For a merged bin, the process is similar, but the calculation is recursive. Continuing the example from Table 2, say an LP in the merged bin, binID = i_m wants to remove amount of their LP balance from that now-merged bin. That merged bin no longer has any reserves directly associated with it. But it does possess an LP balance in the active bin with binID = i_a and those balances correspond to reserve amounts,

$$A_{i_a} = \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}}\texttt{reserveA}_\texttt{a}, \tag{4}$$

$$B_{i_a} = \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}}\texttt{reserveB}_\texttt{a}. \tag{5}$$

The LP is removing `amount` from bin `binID` = i_m. So the net amount the user will receive is

$$A_{out} = \frac{\texttt{amount}}{\texttt{totalSupply}_\texttt{m}} \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}} \texttt{reserveA}_\texttt{a}, \tag{6}$$

$$B_{out} = \frac{\texttt{amount}}{\texttt{totalSupply}_\texttt{m}} \frac{\texttt{mergeBinBalance}_\texttt{m}}{\texttt{totalSupply}_\texttt{a}} \texttt{reserveB}_\texttt{a}. \tag{7}$$

A user must first call `migrateBinsUpStack` for any `binIDs` that the user wants to remove that are merged more than one level deep. The migrate function will move the bins up the linked list of merged bins until each migrated bin is pointing to an active bin. As part of that function, `mergeBinBalance` of the bins being migrated will be updated along with the balances of the bin that was the head bin, i.e. `balances[0]` of the old head bin.

3 Pool

Pools are created permissionlessly by any user and that first user has the option of configuring the pool parameters.

3.1 Data Structure

A pool is parameterized by:

- `fee` — the proportion of swap that stays in the pool, e.g., 0.1%
- `tickSpacing` — the width of a bin on a geometric grid of 1 basis point ticks where bin width = $1.0001^{\texttt{tickSpacing}}$
- `lookback` — TWAP look back in seconds, default is 3 hours
- `tokenA` — quote token in the pool
- `tokenB` — base token in the pool

A pool tracks the following state elements:

- `state.activeBin` — lowest tick of the set of bins that contain the current price
- `state.status` — lock signal to protect against reentrancy
- `state.binCounter` — highest `binID` that has been initialized
- `state.protocolFeeRatio` — the proportion of the swap fee that is retained by the Maverick protocol
- `binPositions` — mapping from (`lowerTick`, `kind`) to active `binId` if a bin exists at that (`lowerTick`, `kind`)

- `binMap` — bit mask of which ticks have active bins and which kinds of bins are in those ticks
- `bins` — list of bin objects indexed by `binId`
- `binBalanceA` — the sum A reserves in all active bins
- `binBalanceB` — the sum B reserves in all active bins

3.1.1 binMap

The `binMap` is multilevel data structure designed for efficient bin look-ups. Conceptually, the `binMap` is a collection of four-bit nybbles at every possible tick position with each bit representing one of the four bin `kinds` (`STATIC`, `RIGHT`, `LEFT`, `BOTH`). In order to reduce memory usage, however, a sparse data structure is used that only maintains a minimal collection of 256-bit words (holding 64 nybbles representing 64 ticks), which are indexed with a simple 32-bit-integer-to-word `hashmap`. Only non-zero words are kept in the hashmap.

Fast look-ups can be made "up" or "down" relative to a reference tick position by computing the index of the current word and offset from the reference tick position (`index, offset = divmod(tick, 64)`). If the index is not present in the hashmap, then the index is incremented (up) or decremented (down) until an active word is found, or an iteration limit is reached. If an active word is found, then the position of the least-significant bit (up) or most-significant bit (down) is used to determine both the bin `kind` and the tick position. If the active index matches the reference index, then the most-significant- or least-significant-bit search is done relative to the reference-tick offset.

3.2 Swapping

3.2.1 Callback Mechanics

Swapping is callback-based, which allows users to "flash swap" such that they can collect the proceeds of their swap before they have to transmit what they owe to the contract. Users can swap by specifying either the exact amount they want to receive of a given token or the exact amount they want to swap in.

The contract will disburse the proceeds to the user. In calling a swap on the pool, the user had to provide a callback function. To pay the contract what it is owed for the swap, the contract will call the user-provided callback, which will need to transmit the user's tokens to the contract in order for the transaction to complete.

3.2.2 Swapping in a Bin

For any given position in price, there can be up to 4 active bins. To compute a swap efficiently, the contract first determines the amount of reserve for all bins present at the current price. Then the swap computation is performed on this aggregate amount of reserve. After the swap is complete, the input and output token amounts

for the aggregate swap amount are distributed to each of the participating bins in proportion to the bin's reserve amounts.

The process is as follows:

1. Lookup in `binMap` the bins that are active at the current price. This set of bins is $\mathscr{I}$, where the number of bins is at most 4, $|\mathscr{I}| \leq 4$
2. Compute the total aggregate A and B amounts in these bins —
$$A = \sum_{i \in \mathscr{I}} A_i, \quad B = \sum_{i \in \mathscr{I}} B_i \tag{8}$$
3. Compute the aggregate liquidity by solving this quadratic for L
$$0 = \left(\sqrt{\frac{p_l}{p_u}} - 1\right) L^2 + \left(\frac{A}{\sqrt{p_u}} + B\sqrt{p_l}\right) L + AB \tag{9}$$
4. Compute sqrt price, $\sqrt{P}$
$$\sqrt{P} = \frac{A + L\sqrt{p_l}}{B + L/\sqrt{p_u}} \tag{10}$$
5. Extract fee from the token balance coming in and set that aside as either A_{fee} or B_{fee}
6. Extract the protocol fee from the total fee
$$A_{protocol} = A_{fee}\texttt{state.protocolFeeRatio} \tag{11}$$
$$B_{protocol} = B_{fee}\texttt{state.protocolFeeRatio} \tag{12}$$
7. Use the identities to compute ΔA and ΔB
$$\Delta\sqrt{P} = \frac{\Delta A}{L} \tag{13}$$
$$\Delta\frac{1}{\sqrt{P}} = \frac{\Delta B}{L} \tag{14}$$
8. Apportion these amounts to the active bins
$$A_{i,new} = A_i + \frac{A_i}{A}\left(\Delta A + A_{fee} - A_{protocol}\right) \tag{15}$$
$$B_{i,new} = B_i + \frac{B_i}{B}\left(\Delta B + B_{fee} - B_{protocol}\right) \tag{16}$$
9. Increment the `binBalanceA` and `binBalanceB` balances with $\Delta A + A_{fee} - A_{protocol}$ and $\Delta B + B_{fee} - B_{protocol}$, respectively
10. Update TWAP with the new ending price of the swap

At the end of the swap, the `binBalanceA` and `binBalanceB` balances will reflect the sum A and B balances across all bins. The amount of protocol fee set aside is not explicitly tracked. Instead, the protocol fee is the difference between the ERC20 A and B balances and the bin balances.

3.2.3 Rounding

The contract is designed to round appropriately in order to remain solvent with respect to the pool's balance according to `tokenA` and `tokenB` ERC20 contracts. In particular, the contract must ensure that

- The token balances according to the ERC20 contracts is always greater than or equal to both the binBalance and the sum of the bin's reserves
- The binBalance is greater than or equal to the sum of the bins' reserves

$$\sum_i \texttt{reserveA}_i \leq \texttt{binBalanceA} \leq ERC_A \tag{17}$$

$$\sum_i \texttt{reserveB}_i \leq \texttt{binBalanceB} \leq ERC_B \tag{18}$$

3.2.4 Swapping Through Ticks

A swap may be large enough that it will swap an entire bin. If this happens, the swap-in-a-bin process described above will be repeated again for the next adjacent bin set with any remaining assets that are remaining to be swapped.

3.3 Moving Bins and TWAP

The contract tracks the TWAP of the pool by registering the price of the pool at the end of each swap, but the value is overwritten for swaps in the same block. The TWAP is stored in the log price domain and, ultimately, the pool only needs to know which tick the TWAP is in because that dictates when non-static bins move left or right with the price.

After a swap, the contract checks to see if any bins need to be moved. If so, then the move proceeds. Within a block, no time passes between operations, so the TWAP will also not change for the duration of the block. Because of this, no bins will move beyond the first swap in a block, as any subsequent checks for movement will find the bins already in line with the TWAP. In other words, all movement checks within a block are governed by the TWAP change that occurred in the previous block. These mechanisms mean that a swapper cannot move liquidity using a swap inside of a single block. This makes the movement robust to large inner-block flash swap operations that may significantly move the price.

That is, in the case of a large two-step flash swap that moved the price up and then back down inside the block, none of the dynamic liquidity bins would move in response and the TWAP would be unaffected by the large price excursion. For liquidity to move, a swapper would have to leave their capital on chain for at least one block period, which would leave that liquidity exposed to arbitrageurs, thereby discouraging any such toxic liquidity movement manipulations.

Finally, when an LP starts a pool, they have the option to choose the TWAP lookback period. Longer periods further blunt any liquidity manipulation attack surface. The suggested default liquidity lookback period for a pool is 3 hours.

Notation:

- t_c — `lowerTick` of the bin that contains the current price
- t_p — `lowerTick` of the bin that contains the previous price
- $t_{twap,c}$ — `lowerTick` of the bin that contains the current TWAP
- $t_{twap,p}$ — `lowerTick` of the bin that contains the previous TWAP
- $t_{bin,c}$ — `lowerTick` of the moving bin after the move
- $t_{bin,p}$ — `lowerTick` of the moving bin before the move

The movement conditions are

- If $t_c == t_p$ and $t_{twap,c} == t_{twap,p}$ no bins move
- Only bins within one tick of price or exactly the previous TWAP will move; other bins stay "stranded" until the price moves within one bin of their position
- The target right-most tick where `RIGHT` or `BOTH` bins will move to is $\text{target}_{right} = \min\{t_c - 1, t_{twap,c}\}$
- All `RIGHT` and `BOTH` bins from $\min\{t_p - 1, t_{twap,p}\}$ to $\text{target}_{right} - 1$ will be moved to target_{right}
- The target left-most tick where `LEFT` or `BOTH` bins will move to is $\text{target}_{left} = \max\{t_c + 1, t_{twap,c}\}$
- All `LEFT` and `BOTH` bins from $\max\{t_p + 1, t_{twap,p}\}$ down to $\text{target}_{left} + 1$ will be moved left to target_{left}

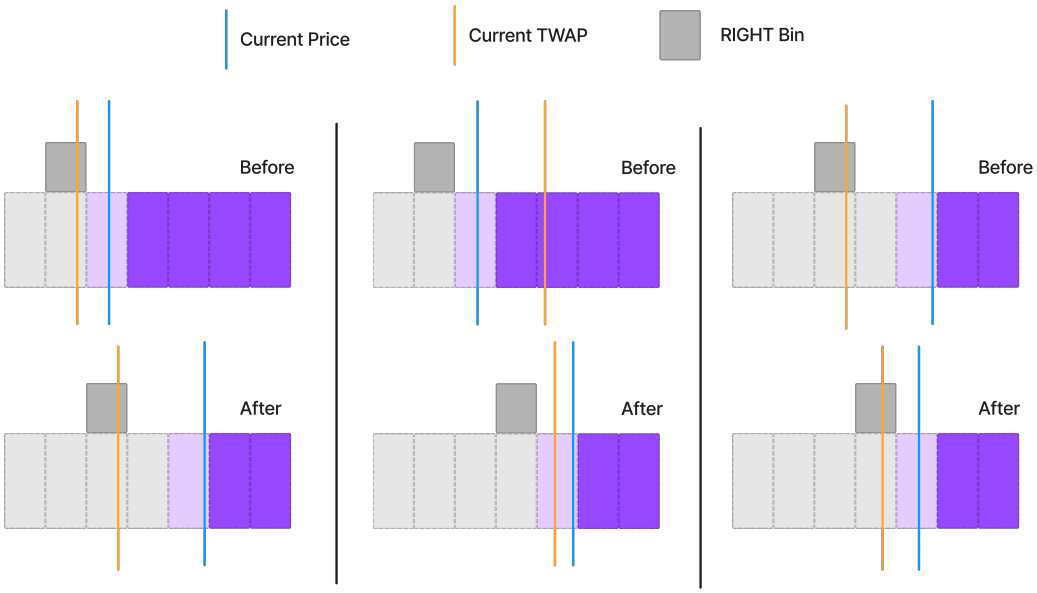

Figure 1: Example situations where a bin would move right.