1 Introduction

1.1 Overview

The Raydium AMM is an on-chain smart contract built on the Solana blockchain. Leveraging the high-speed and low transaction fees of Solana, it provides a fast and cost-efficient tool for market participants to access and provide liquidity to trade and swap digital assets based on the "constant product" in a permissionless and decentralized manner.

The special feature of the Raydium AMM that differentiates it from other AMMs such as Uniswap is that the program also shares its liquidity according to the Fibonacci sequence in the form of limit orders on OpenBook, the primary central limit order book (CLOB) of Solana. This allows for deeper liquidity on the order book while simultaneously earning extra fees for the liquidity providers through third-party order flow.

Whether the traditional AMM operations or when sharing liquidity on OpenBook, 1.the principle of a constant product is maintained as:

$$Rx * Ry = K \tag{1}$$

Where Rx and Ry represent the amount of X and Y in associated market reserve vaults deposited by liquidity providers. And then the Δx amount of X can be swapped for Δy amount of Y such as:

$$(Rx + \Delta x) * (Ry - \Delta y) = Rx * Ry = K \tag{2}$$

The open-source code can be found here: https://github.com/raydium-io/raydium-amm

2 Program Overview

The structural design of the AMM requires pools to be combined with an OpenBook market when created.

2.1 Program ID

OpenBook Program ID:

Cluster	Address
Mainnet-Beta	srmqPvymJeFKQ4zGQed1GFppgkRHL9kaELCbyksJtPX
Devnet	EoTcMgcDRTJVZDMZWBoU6rhYHZfkNTVEAfz3uUJRcYGj
Devnet for Immunefi	EoTcMgcDRTJVZDMZWBoU6rhYHZfkNTVEAfz3uUJRcYGj

Note: refer to https://github.com/openbook-dex/program

AMM Program ID:

Cluster	Address
Mainnet-Beta	675kPX9MHTjS2zt1qfr1NYHuzeLXfQM9H24wFSUt1Mp8
Devnet	HWy1jotHpo6UqeQxx49dpYYdQB8wj9Qk9MdxwjLvDHB8
Devnet for Immunefi	AMMjRTfWhP73x9fM6jdoXRfgFJXR97NFRkV8fYJUrnLE

2.2 Account Structure

This section primarily introduces the layout of the main accounts under the ownership of the AMM program.

2.2.1 AMM Info

The AMM Info Account is a PDA (Program Derived Address) account created during the initialization of the pool using the following string as seeds:

$[program\_id, market\_account,$ "amm_associated_seed"].

Its primary purpose is to store the pool's state, fee rates, order placement parameters, associated vaults and mint accounts along with their decimals, the amount of the pool's lp_mint, and certain accounts related to the associated OpenBook market. These accounts might include the open orders account, market account, and OpenBook program id, among others.

```
pub struct AmmInfo {
    pub status: u64,
    pub nonce: u64,
    pub order_num: u64,
    pub depth: u64,
    pub coin_decimals: u64,
    pub pc_decimals: u64,
    pub state: u64,
    pub reset_flag: u64,
    pub min_size: u64,
    pub vol_max_cut_ratio: u64,
    pub amount_wave: u64,
    pub coin_lot_size: u64,
    pub pc_lot_size: u64,
    pub min_price_multiplier: u64,
    pub max_price_multiplier: u64,
    pub sys_decimal_value: u64,
    pub fees: Fees,
    pub out_put: OutPutData,
    pub token_coin: Pubkey,
    pub token_pc: Pubkey,
    pub coin_mint: Pubkey,
    pub pc_mint: Pubkey,
```

```
    pub lp_mint: Pubkey,
    pub open_orders: Pubkey,
    pub market: Pubkey,
    pub serum_dex: Pubkey,
    pub target_orders: Pubkey,
    pub withdraw_queue: Pubkey,
    pub token_temp_lp: Pubkey,
    pub amm_owner: Pubkey,
    pub lp_amount: u64,
    pub client_order_id: u64,
    pub padding: [u64; 2],
}
```

2.2.2 Target Orders

The Target Account is also a PDA (Program Derived Address) account created during the initialization of the pool using the following string as seeds:

$[program\_id, market\_account,$ "target_associated_seed"].

Its primary purpose is to store intermediate variables used during order placement in the pool. This includes the price and quantity of each order, the client order id associated with each order, as well as the quantities of x and y in the pool after the previous PNL collection.

```
pub struct TargetOrders {
    pub owner: [u64; 4],
    pub buy_orders: [TargetOrder; 50],
    pub padding1: [u64; 8],
    pub target_x: u128,
    pub target_y: u128,
    pub plan_x_buy: u128,
    pub plan_y_buy: u128,
    pub plan_x_sell: u128,
    pub plan_y_sell: u128,
    pub placed_x: u128,
    pub placed_y: u128,
    pub calc_pnl_x: u128,
    pub calc_pnl_y: u128,
    pub sell_orders: [TargetOrder; 50],
    pub padding2: [u64; 6],
    pub replace_buy_client_id: [u64; MAX_ORDER_LIMIT],
    pub replace_sell_client_id: [u64; MAX_ORDER_LIMIT],
    pub last_order_numerator: u64,
    pub last_order_denominator: u64,
    pub plan_orders_cur: u64,
    pub place_orders_cur: u64,
```

```
    pub valid_buy_order_num: u64,
    pub valid_sell_order_num: u64,
    pub padding3: [u64; 10],
    pub free_slot_bits: u128,
}
```

3 Features

The Raydium AMM program contains the traditional AMM features, such as initialize pool, add and remove liquidity, and swap.

In addition, there are some admin features like admin set params, admin cancel AMM orders, Admin update config, admin migrate to OpenBook, admin withdraw PNL (Profit and loss), and MSRM tokens.

For the special feature of sharing liquidity on OpenBook, an on-chain state machine is cranked by the monitor step function.

3.1 Initialize Pool

The program allows anyone to create and initialize a pool associated with an unused OpenBook market and with params of pool open time and initialize tokens amounts to be deducted.

During this period, some associated accounts will be created and initialized, the pool LP mint decimals value will be set to the token coin decimals value. And the liquidity amount is calculated according to the following formula:

$$liquidity = \sqrt{init_{pc} * init_{coin}} \tag{3}$$

To ensure pool liquidity is never empty, a small amount of liquidity ($10^{lpdecimals}$) is locked, the initialized pool liquidity is:

$$liquidity_{init} = liquidity - 10^{lpdecimals} \tag{4}$$

3.2 Add and Remove Liquidity

Like other AMMs, the Protocol also supports adding liquidity by depositing with a specified base token amount and minting corresponding LP tokens and removing liquidity by withdrawing the specified amount and burning the corresponding LP tokens.

When adding or removing liquidity, the amount is proportional to the quantity of tokens and liquidity ratio without changing the price of the pool.

When adding liquidity, we use a specified quantity of a particular coin, as an example:

$$deduct\_amount_{pc} = \frac{(specified\_amount_{coin} * pool\_amount_{pc})}{pool\_amount_{coin}} \tag{5}$$

$$deduct\_amount_{lp} = \frac{(specified\_amount_{coin} * pool\_amount_{lp})}{pool\_amount_{coin}} \tag{6}$$

When removing liquidity:

$$calculate\_amount_{token} = \frac{\left(withdraw\_amount_{lp} * pool\_amount_{token}\right)}{pool\_amount_{lp}} \quad (7)$$

However, because liquidity is shared to the corresponding OpenBook market according to the pre-defined parameters, and the fact that the portion of liquidity shared on the OpenBook is not updated in real-time by the AMM when trades are executed, it's necessary to calculate and collect Profit and Loss (PNL) before performing liquidity addition or removal calculations. This involves using the quantities of the new pool vaults for calculations. Additionally, after liquidity updates, tracking and recording the quantities of the pool vaults are essential for future calculations. The process of calculating PNL is quite complex and is detailed as follows:

a) When orders from the pool are filled on the OpenBook market or when users perform swaps, the presence of fees causes the overall K value of the pool to increase. The increased value of K is credited to liquidity providers as fees earned:

$$K_{before} \xrightarrow{order\ filled\ or\ swap(\Delta K)} K_{after}$$

b) ΔK consists of two parts, with one part going to liquidity providers in the pool and the other part serving as the protocol fee, or PNL. Since the price of the pool is unchanged when adding or removing liquidity, the calculation of PNL does not affect the price. In this process, we use x_{last} and y_{last} to represent the quantities of vaults after the last PNL collection, $x_{current}$ and $y_{current}$ as the current vault quantities of the pool, and x_{last} and y_{last} as the quantities of the pool's vaults after deducting ΔK (Δx, Δy).

$$Price_{current} = \frac{x_{current}}{y_{current}} = \frac{x_{after}}{y_{after}} \quad (8)$$

$$x_{after} * y_{after} = x_{last} * y_{last} \quad (9)$$

Then: $x_{after} * \frac{x_{after} * y_{current}}{x_{current}} = x_{last} * y_{last}$ =>

$$x_{after} = \sqrt{x_{last} * y_{last} * \frac{x_{current}}{y_{current}}} \quad (10)$$

Then:

$$y_{after} = x_{after} * \frac{y_{current}}{x_{current}} \quad (11)$$

And final we get: $\Delta x = x_{current} - x_{after}$, $\Delta y = y_{current} - y_{after}$

c) Assuming α represents the protocol fee rate collected from the pool, then according to the result from b): $x_{pnl} = \alpha * \Delta x$, $y_{pnl} = \alpha * \Delta y$

d) After deducting PNL, the quantity of the pool's vault becomes:

$$x_{pool} = x_{current} - x_{pnl} \quad (12)$$

$$y_{pool} = y_{curretn} - y_{pnl} \quad (13)$$

Then, combining x_{pool} and y_{pool} with formulas (5), (6), and (7), the corresponding amounts are calculated. Finally, updating x_{last} and y_{last} to facilitate the next calculation.

3.3 Swap

In Raydium, the swap operation is same with other constant product AMMs, maintaining a

constant K value, as specified in formula (2). However, the difference in Raydium is that it cancels AMM orders on the OpenBook according to the swap direction:

a) Cancel the bid orders owned by AMM while swaping from coin to pc;
b) Cancel the ask orders owned by AMM while swaping from pc to coin;

Note: Only the swap fee is deducted during swaps. The pool does not calculate the pending PNL because the complexity of calculation would require more compute units. Instead, the PNL is calculated and collected during add, remove liquidity, or IdleState. Handling PNL in this way only impacts the collection of PNL but does not impact fees accrued to users.

3.4 Monitor Step

When sharing liquidity with OpenBook, due to compute units limitations (200,000 compute units) in a single Solana transaction (early Solana versions) the process must be divided into some submodules to complete the whole operation. Here, a mechanism similar to a state machine is used for computation. The data is saved, updated, and processed based on the current on-chain data in each state. Several main states are involved: IdleState, PlanOrdersState, PlaceOrdersState, CancelAllOrdersState. The diagram below illustrates the state transition summary with brief explanations for each state:

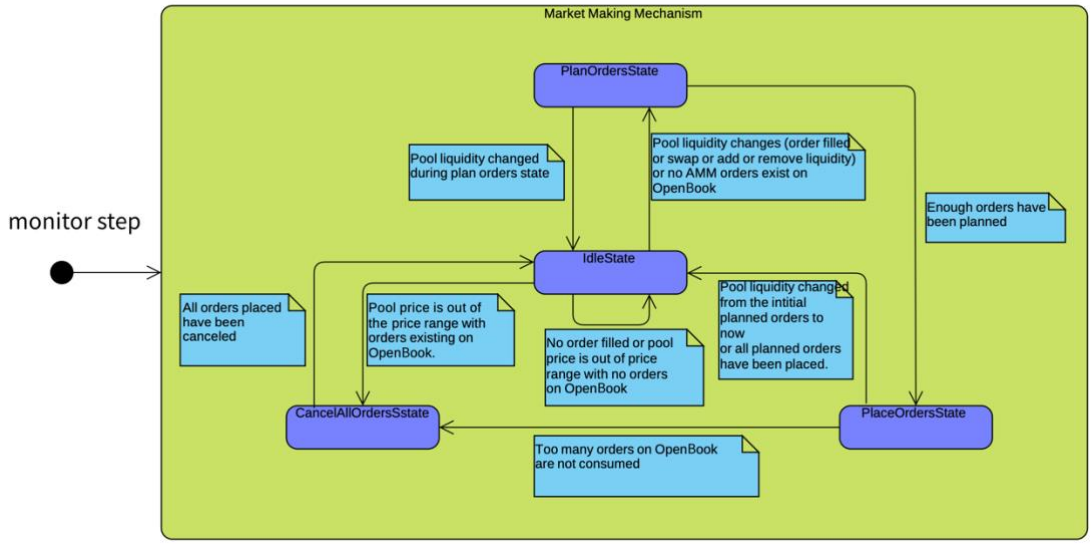

Fig1: State machine diagram of the liquidity shared on OpenBook

3.4.1 IdleState

This is the initial state of the state machine. The AMM evaluates the on-chain data to determine whether to plan orders, cancel orders, or remain in the idle state and then saves the result of decision.

The decisions are:

a) The state will change to PlanOrdersState when the pool liquidity changes (AMM orders on the OPENBOOK are filled, liquidity is added or removed, or assets are swapped from the vaults), or there are no AMM orders on OpenBook.

b) The state will change to CancelAllOrdersState if the pool price is outside of the price range limitations with AMM orders existing on OpenBook.
c) The state will remain in IdleState if the pool price is outside of the price range limitations with no AMM orders existing on OpenBook.

Note: There is a check for K, which will change to CancelAllOrdersState to cancel all AMM orders if it's found that the value of K has decreased.

3.4.2 PlanOrdersState

The primary function of this state is for the AMM to determine the price and the size of the limit orders placed on the OpenBook based on the pool liquidity and price.

The main objective is to maintain the constant product formula (1) throughout the whole process. Specifically, using amount of both assets within the pool represented as x and y, combined with the constant K value to determine the order size according to the price. And to track and record the new value of x and y after the order is complete. The process iterates continuously until calculations for all price levels are finished.

The details are as follows:

a) Calculate the grid value based on the pool price, the depth and order num stored in the AMM Info account:

$$grid = \frac{price_{current} * depth}{100 * order_{num}} \tag{14}$$

b) Calculate the max bid price and the min ask price according to the trade fee rate and the separate (an arbitrary buffer that accounts for slippage):

$$price_{\max\_bid} = price_{current} * (1 - trade\_fee\_rate - seperate) \tag{15}$$

$$price_{\min\_ask} = price_{current} * (1 + trade\_fee\_rate + seperate) \tag{16}$$

Then calculate each additional order's price and size based on the grid value and the Fibonacci sequence.

c) Calculating the quantity without considering the impact of fees. According to the formula (1) let's take buy orders as an example first:

$$\Delta y = \frac{x_{lastbid}}{price_{currentbid}} - y_{lastbid}$$

$$\Delta x = price_{currentbid} * \Delta y = x_{lastbid} - price_{currentbid} * y_{lastbid}$$

So, after the order placed the pool assets become $y_{lastbid} + \Delta y$ and $x_{lastbid} - \Delta x$:

$$y_{lastbid} + \Delta y = \frac{x_{lastbid}}{price_{currentbid}}$$

$$x_{lastbid} - \Delta x = price_{currentbid} * y_{lastbid}$$

That is also maintaining the constant product:

$$(x_{lastbid} - \Delta x) * (y_{lastbid} + \Delta y) = x_{bid} * y_{bid}$$

Then the sell orders:

$$\Delta y = y_{lastask} - \frac{x_{lastask}}{price_{lastask}}$$

$$\Delta x = price_{currentask} * \Delta y = price_{currentask} * y_{lastask} - x_{lastask}$$

So, after the order is placed the pool assets become $x_{lastask} + \Delta x$ and $y_{lastask} - \Delta y$

$$x_{lastask} + \Delta x = price_{currentask} * y_{lastask}$$

$$y_{lastask} - \Delta y = \frac{x_{lastask}}{price_{currentask}}$$

That is also maintaining the constant product:

$$(x_{lastask} + \Delta x) * (y_{lastask} - \Delta y) = x_{ask} * y_{ask}$$

The volume (size) of the buy order is:

$$vol_{bid} = \frac{x_{lastbid}}{price_{currentbid}} - y_{lastbid} \quad (17)$$

The volume (size) of the sell order is:

$$vol_{ask} = y_{lastask} - \frac{x_{lastask}}{price_{currentask}} \quad (18)$$

d) Use the value of x and y saved in the Step C and repeat the calculation method in Step C until all the orders are calculated and saved.

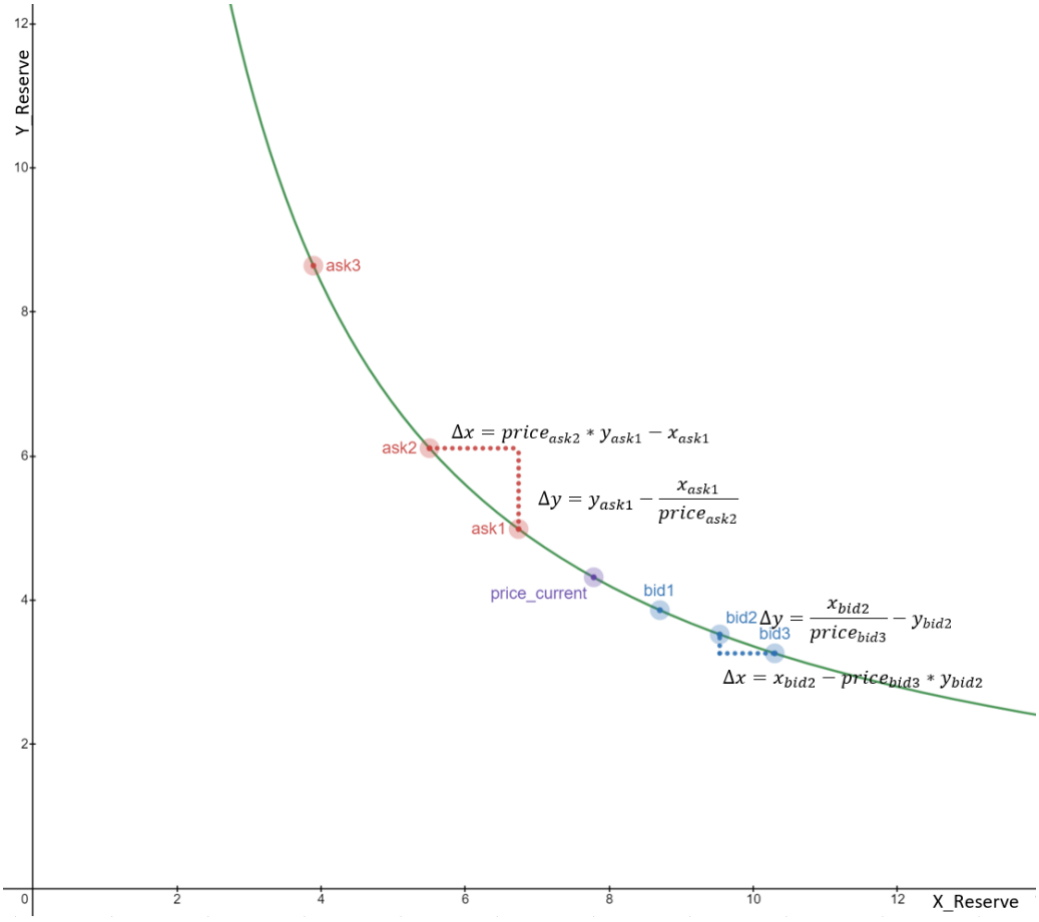

Fig2: The AMM X-Y curve's reserves each price point

3.4.3 PlaceOrdersState

In this stage, the AMM will place or replace orders on OpenBook based on the price and volume calculated during the PlanOrdersState. It's important to note that from the beginning of plan

orders to the end of all orders placed, the pool liquidity must not change to avoid disrupting the constant product formula and resulting in losses of pool assets.

If there pool liquidity changes during this time, including adding or removing liquidity, or successful swaps or filled orders, the process will be interrupted and the revert to IdleState and restart the process.

3.4.4 CancelAllOrdersState

This state cancels all of the AMM orders on the OpenBook and the assets shared on OpenBook are settled and returned to the AMM vaults.

4 Convert Mathematics

In the case that the pc and coin have different decimals, they are converted to the decimals of pc and coin to the same amount (sys_decimals_value). These conversions add a lot of complex and redundant operations and may not be necessary. There are some temporary variables saved in the target account, and it is currently not easy to optimize.

We have established the following two values:

$$multiplier_{pc} = 10^{decimals_{pc}} \tag{19}$$

$$multiplier_{coin} = 10^{decimals_{coin}} \tag{20}$$

Usually, if the pc decimal is larger than the coin decimal:

$$sys\_decimals\_value = multiplier_{pc}$$

Otherwise:

$$sys\_decimals\_value = multiplier_{coin}$$

But if the $sys\_decimals\_value$ is smaller than $\frac{(multiplier_{pc} * coinlotsize_{openbook})}{multiplier_{coin} * pclotsize_{openbook}}$

$$sys\_decimals\_value = \frac{(multiplier_{pc} * coinlotsize_{openbook})}{multiplier_{coin} * pclotsize_{openbook}}$$

Convert token amount with its decimals to amount with sys_decimals_value:

$$x \text{ or } y = \frac{(amount_{token} * sys\_decimals\_value)}{multiplier_{token}} \tag{21}$$

Convert amount with sys_decimals_value to token amount with its decimals:

$$amount_{token} = \frac{(x \text{ or } y * multiplier_{token})}{sys\_decimals\_value} \tag{22}$$

AMM coin lot size is equal to OpenBook coin lot size:

$$coinlotsize_{amm} = coinlotsize_{openbook} \tag{23}$$

Convert OpenBook pc lot size to AMM pc lot size:

$$pclotsize_{amm} = \frac{sys\_decimals\_value * pclotsize_{openbook} * multiplier_{coin}}{coinlotsize_{openbook} * multiplier_{pc}} \tag{24}$$

Convert AMM coin lot size to OpenBook coin lot size:

$$pclotsize_{openbook} = \frac{pclotsize_{amm} * coinlotsize_{openbook} * multiplier_{pc}}{sys\_decimals\_value * multiplier_{coin}} \quad (25)$$

Convert OpenBook price to AMM price:

$$price_{amm} = price_{openbook} * pclotsize_{amm} \quad (26)$$

Convert AMM price to OpenBook price:

$$price_{openbook} = \frac{price_{amm}}{pclotsize_{amm}} \quad (27)$$

Convert OpenBook volume to AMM volume:

$$coinvol_{amm} = \frac{coinvol_{openbook} * coinlotsize_{openbook} * sys\_decimals\_value}{multiplier_{coin}} \quad (28)$$

Convert AMM volume to OpenBook volume:

$$coinvol_{openbook} = \frac{coinvol_{amm} * multiplier_{coin}}{coinlotsize_{openbook} * sys\_decimals\_value} \quad (29)$$