Muffin

April 2022

1 Introduction

Muffin is an automated market maker (AMM) that supports concentrated liquidity and multiple fee tiers in one pool. Other new features include limit range orders, EMA price oracles, internal accounts, and single-contract design.

We will describe each feature's detail and explain some key concepts of the technical implementation and the derivations of the related formulas.

2 Features

2.1 Concentrated liquidity

Muffin inherits the existing design of concentrated liquidity. Traditionally, `x*y=k` AMMs force liquidity providers (LPs) to put liquidity on the price range from 0 to infinity. It is wasteful because a large part of the liquidity can never be put into service. Concentrated liquidity means that now LPs can select their own price ranges to concentrate their capital there as a way to provide a great amount of liquidity to the pool. LPs can accrue fees from the swaps that are executed in their selected price ranges.

2.2 Multiple fee tiers in one pool

The typical design of AMM has been "one fixed fee tier for one pool" and to create several pools with different fee tiers to suit the market needs.

In Muffin, we change the design to "multiple fee tiers for one pool" and create only one pool to serve the needs of different fee tiers for a token pair. The hierarchical change enables a data structure that can afford more choices of fee tiers and make cross-tier order optimization more gas-efficient.

On a data structure level, each tier of a pool in Muffin represents a percentage swap fee, and has its independent liquidity, price, and tick states, acting like an "inner pool" by itself. Despite that, the tiers of a pool don't have their own tick spacing settings and TWAP oracles (or technically, *price accumulators*). Instead, they share the same one together in a pool, reducing extra data read and write when executing a swap in multiple fee tiers.

2.2.1 Choices of fee tiers

For LPs, the experience of adding or removing liquidity in Muffin is similar to doing that in Uniswap V3: pick a fee tier, then pick a price range. The sole difference is they are now given more fee tiers to choose from.

Unlike the typical AMM design, the available choices of fee tiers are not preset on the protocol level. Instead, they can be uniquely set for each pool. For example, we can have a set of much lower fee tiers for stablecoin pairs, such as 0.5, 1, 3, and 5 bps, as opposed to some 5, 20, 40, and 60 bps which you may expect in ETH-USDC or a more exotic pair. In general, the more mean-reverting the token pair is, the lower fee it's better to have, and vice versa.

To retain flexibility, Muffin's core contract does not preset any rules or logic about which pool should have what fee tiers. At launch, the action of creating tiers will be gated and can only be performed by the "governance" contract. It can later be replaced by a "controller" contract where we can define the logic and proxy the action of creating tiers to being permissionless.

There is an upper limit on how many tiers are available in a pool. The limit is a protocol-level setting, i.e. applied to all the pools in the protocol. The number is decided solely based on its impact on the swap's gas cost, which is more of an economic consideration than a technical constraint. For example, on Ethereum, Muffin will offer up to 6 fee tiers per pool, and we'll practically set up 3-4 fee tiers per pool considering the current gas price. This number will be tuned up further when Muffin is deployed on L2s or other chains with lower gas prices.

2.2.2 Order optimization

For traders, when you swap on Muffin, the protocol will split your order into several smaller orders and route them into different tiers to give you an optimal rate. The split path is calculated on-chain, as opposed to the conventional routers pre-determining the path off-chain before submitting the transaction.

The calculation takes account of the price, the liquidity level, and the percentage swap fee of each tier in the pool. Suppose Alice wants to swap Δ units of token X for token Y in a pool of n tiers. The protocol then splits her order and routes δ_i units of token X to each tier, i.e. $\sum_{i=1}^{n} \delta_i = \Delta$.

Using each tier's current virtual reserves x_i and y_i, its liquidity $L_i = \sqrt{x_i y_i}$, and its percentage swap fee $(1 - \gamma_i)$, we can formulate the total output amount of token Y, which is our objective function to maximize.

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{n} \left(y_i - \frac{L_i^2}{x_i + \gamma_i \delta_i} \right) \\ &\text{subject to} \sum_{i=1}^{n} \delta_i = \Delta \end{aligned}$$

Solving this optimization problem gives us the formula of the sub-order's optimal size δ_i^*, which we use to determine how the order is split on-chain. Full derivation is in the appendix section.

$$\delta_i^* = \frac{L_i}{\sqrt{\gamma_i}} \frac{\Delta + \sum_{j=1}^{n} \frac{x_j}{\gamma_j}}{\sum_{j=1}^{n} \frac{L_j}{\sqrt{\gamma_j}}} - \frac{x_i}{\gamma_i}, \qquad i = 1, ..., n$$

2.2.3 Combining with concentrated liquidity

The calculation in the last section assumes the tiers' liquidity levels are always unchanged during a swap, which is not the case in a concentrated liquidity pool. To incorporate this into concentrated liquidity, here we adopt a greedy strategy:

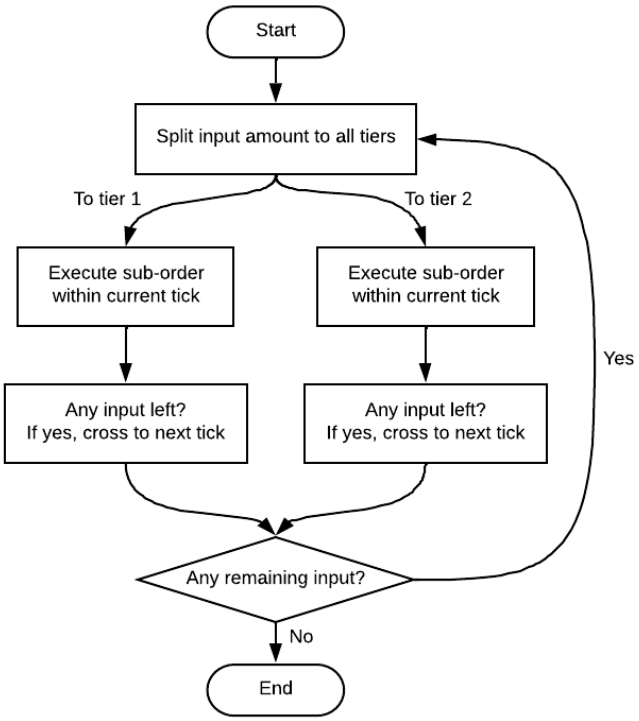


Figure 1: Flowchart of swap

1. When an order comes, the protocol splits the order by considering each tier's current price, liquidity level, and percentage swap fee.

2. Each tier executes its corresponding sub-order within its current tick space and stops when its price hits the boundary of its current tick space.

3. Those stopped tiers, which could not finish the whole sub-order within their current tick space, will cross to their next tick space and, at the same time, adjust their liquidity level.

4. The remaining input tokens are gathered, and the whole process repeats until all input tokens are swapped.

Note that, in principle, such greedy strategy does not produce an exact optimal solution, but it approximates well under a reasonable number of steps, therefore keeping the gas cost to perform it on-chain low enough.

This on-chain order optimization is not mandatory. It is still possible to be more cost-effective for tiny orders to pre-calculate the split path off-chain, or even not do any order splitting at all, considering the gas price.

In addition, Muffin allows traders to specify which tiers to include and exclude when splitting their orders. Sometimes, a tier can have a very low amount of liquidity to the extent that fetching its state from chain and including it in the optimization calculation reversely cause more gas cost than what the optimization can save for the trader. In this case, traders should better exclude those tiers in their swap.

2.2.4 Observations

Doing some simulations, we can see the graph below is what the tier's price movements would be like when the incoming orders are split in the proposed way. Each colored line in the graph represents a different γ, i.e. one minus the percentage swap fee.

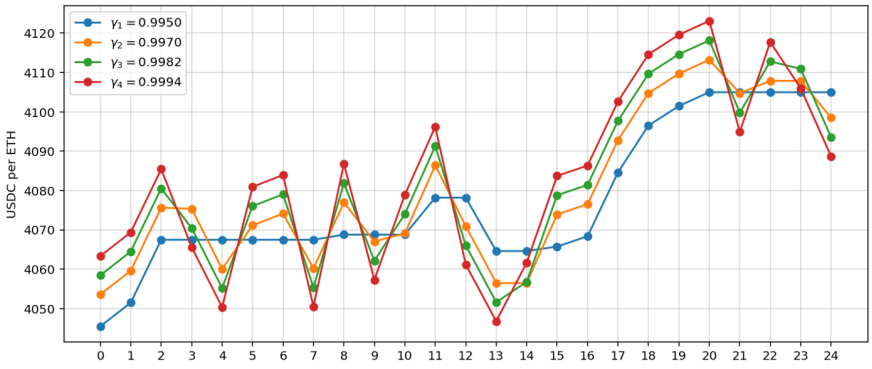


Figure 2: Example of tiers' price movements

We can see the prices of the lower fee tiers swing up and down more vigorously, and are more sensitive to the market change, comparing to the prices of the higher fee tiers. This matches our intuition. Intuitively, the pool should distribute more trades to the tiers with lower fees in order to optimize the exchange rate. And for an AMM, each trade is an impact on price. So, more trades on the lower fee tiers lead to more price fluctuations there.

Also, we can see the pattern that each price has a maximum distance to another. Technically, the relationship between each two tiers' prices is approximately $\frac{\gamma_j}{\gamma_i} \leq \frac{P_i}{P_j} \leq \frac{\gamma_i}{\gamma_j}$ for $\gamma_i > \gamma_j$. The full derivation is in the appendix section.

2.3 Unified TWAP

It is now common to see the liquidity of a token pair scattered into pools of different fee tiers. However, the current design of TWAP oracle only keeps track of the price of only one pool. It is not desirable to have multiple oracles for multiple pools for one token pair since it dilutes the resistance to manipulation. This problem exacerbates when there are more fee tiers for a token pair.

Here in Muffin, each pool tier has its own liquidity and price. But instead of having an independent TWAP for each tier, each pool provides a unified TWAP — a time-weighted average of liquidity-weighted average price, P_{t_1,t_2}. It preserves the resistance to manipulation even if liquidity is fragmented into different tiers. Note that the average is a geometric mean.

$$P_{t_1,t_2} = \prod_{i=t_1}^{t_2} \left[\left(\prod_{j=1}^{n} P_j^{L_j} \right)^{\frac{1}{\sum_{j=1}^{n} L_j}} \right]^{\frac{1}{t_2 - t_1}}$$

Technically, the protocol stores the cumulative sum of the liquidity-weighted average of the tiers' tick indices from the moment that the pool is created to the present time. Let's denote it by a_t.

$$a_t = \sum_{i=t_0}^{t} \left(\frac{\sum_{j=1}^{n} L_{ij} \log_{1.0001} P_{ij}}{\sum_{j=1}^{n} L_j} \right)$$

We can show that, by knowing a_t at two different timestamps t_1 and t_2, we can compute back P_{t_1,t_2}, i.e. the TWAP over this period.

$$\log_{1.0001} P_{t_1,t_2} = \frac{\sum_{i=t_1}^{t_2} \left(\frac{\sum_{j=1}^{n} L_{ij} \log_{1.0001} P_{ij}}{\sum_{j=1}^{n} L_j} \right)}{t_2 - t_1}$$

$$\log_{1.0001} P_{t_1,t_2} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

$$P_{t_1,t_2} = 1.0001^{\frac{a_{t_2} - a_{t_1}}{t_2 - t_1}}$$

2.4 EMA price oracle

In the past, oracle users (e.g. lending protocols) had to rely on either Chainlink oracles or AMM's TWAPs with the help of keepers to record the price accumulator value a_t periodically (e.g. TWAP oracles maintained by Keep3r Network). These two methods are available to the popular token pairs, but they are not economically viable for long-tail tokens. Later, Uniswap v3 introduced one-call oracles that users can query prices with just one contract call, although the oracles require an upfront gas cost to initialize.

To progress, Muffin provides 20-min and 40-min EMA price oracles on every pool, such that oracle users can query a time-weighted average price of a reasonable time period without the need to do two contract calls, require the help from keepers, or prepay any upfront gas cost to initialize data storage slots. The EMA oracles are free and available from the moment the pool is created.

The rationale behind using 20 and 40 minutes is that we expect the major on-chain oracle users are lending protocols, which are generally using 15-min and 30-min TWAPs. And we lengthen the periods a bit to 20-min and 40-min to make the oracles more secure, considering the fact that EMA is more attack-prone than SMA as it gives the recent data points greater weights in the calculation.

Technically, the contract stores the EMA of the liquidity-weighted average of the tick indices of all the tiers in the pool, defined as:

$$EMA_T = (1 - u^t)g + u^t \cdot EMA_{T-t}$$

$$u = 1 - a = 1 - \frac{2}{N+1}$$

$$g = \frac{\sum_{i=1}^{n} L_i \log_{1.0001} P_i}{\sum_{i=1}^{n} L_i}$$

where a is the smoothing factor and is equal to $\frac{2}{N+1}$ by definition, N is the number of seconds of the EMA period, i.e. 2400 and 1200, t is the seconds elapsed between the last update and the present time, and g is the current liquidity-weighted average of the tick indices. We can get the price by calculating 1.0001 raised to the power of the EMA tick.

2.5 Limit range orders

2.5.1 Range order

Uniswap V3 introduced a new order type called "range order". It refers to traders creating a position to sell an underlying token for another along the AMM's liquidity curve when the price moves.

Depositing tokens to a narrow price range can imitate a traditional limit order. However, traders need to withdraw liquidity once they entirely sell out their tokens; otherwise, their positions will automatically buy back what they have sold if the price moves backward.

2.5.2 Limit range order and settlement

Muffin offers LPs to set their range order to "limit range order", as long as the width of the position's tick range matches the one that the protocol requires. In general, we'll set the required width to as narrow as one tick spacing of the pool to imitate the traditional limit order as much as possible.

During a swap, when a tier's price crosses a tick on which some LPs' limit-range-order positions has just fully converted into a single token currency, the protocol will "settle" these positions as if the LPs have just withdrawn the liquidity in the same tx of the swap. Their position's liquidity will not be put into service again even if the price moves back. The position will stay in the form of one token currency. It feels similar to traditional limit orders being filled and executed.

LPs of limit range orders also accrue fees from the swaps that happen inside their position's range before their positions are "settled". Technically, when the positions are being settled, the protocol will snapshot the values of the *fee growths per unit of liquidity* in their tick range. Then, when LPs close their positions, the snapshot is used to calculate how many fees the positions had accrued before the settlement.

Each pool tier can set its own range width requirement for limit range orders, and can also disable this feature. One caveat of enabling it is that it may increase the gas cost for a tick-crossing swap. On Ethereum, a realistic setting would be to enable it on only one tier per pool, assuming normal users care about the limit-order-esque functionality more than the fees they can earn during their positions being filled. In any case, this setting is changeable.

2.5.3 Average execution price

We may be curious on the average execution price of limit range orders, since we are now exchanging tokens on a curve of price instead of a fixed price point. To find it, we can simply use the formulas for calculating how many tokens we should deposit and can withdraw for a single-sided position.

$$\text{Average price} = \frac{\Delta y}{\Delta x} = \frac{l\left(\sqrt{P_{upper}} - \sqrt{P_{lower}}\right)}{l\left(\frac{1}{\sqrt{P_{lower}}} - \frac{1}{\sqrt{P_{upper}}}\right)} = \sqrt{P_{lower}}\sqrt{P_{upper}}$$

2.6 Single-contract design

In contrast with the typical way of creating one contract per pool, all pools in Muffin stay inside one contract under one contract address.

With this design, token reserves of all pools are stored under one address. It reduces the gas cost of multihop swaps, as we no longer need to transfer intermediate tokens across addresses. It also removes the need to call multiple pool contracts and have an external router to orchestrate these hops. Besides multihop swap, any zaps or multicall that interact with multiple pools can now have a lower gas cost.

2.7 Internal accounts

Muffin allows users to deposit ERC-20 tokens into their own internal accounts in the Muffin's contract. Users can use the accounts to pay and receive tokens in swapping and adding/removing liquidity. Together with the previously mentioned single-contract design, the primary goal is to reduce the need for token transfers for frequent traders and active liquidity providers, reducing their gas expenditure.

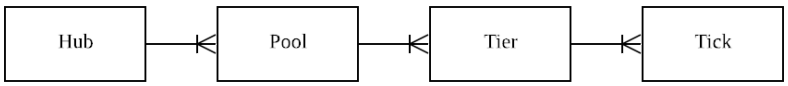
3 Technical concepts

The rest of the paper is about the technical implementation of Muffin in Solidity. We will highlight some noteworthy key concepts here.

3.1 Data hierarchy

As mentioned, we use a single contract to store all pools and handle all swaps and adding/removing liquidity there for all the pools. We call this contract the "hub", i.e.:

The hub contains all the pool states. Each pool represents a token pair and contains various tiers. And each tier represents a swap fee and has its own price, liquidity and tick states.



3.2 Price

Each tier has its independent price. Mathematically, the price of a tier is equal to the ratio of the tier's virtual reserve of an underlying token to that of another, i.e. $P = \frac{y}{x}$. To make calculation convenient in the contract, we store the square root price.

The square root price in the contract is a UQ56.72 fixed-point number stored as a `uint128` variable. Although it has 72 fraction bits, our actual minimum supported square root price is about 2^{-56}, because when the value goes smaller, it is difficult to calculate its corresponding tick index on-chain accurately. In most cases, a UQ56.72 fixed-point is spacious and precise enough to represent most of the realistic price ratios in the market.

3.3 Ticks

Price is partitioned into ticks. Each tick represents a 0.01% price difference to its adjacent ticks, i.e. $P_i = 1.0001^i$ where P is the price and i is the tick index, or specifically $\sqrt{P_i} = \sqrt{1.0001}^i$ as we store square root price in the contract. So, $i = \log_{\sqrt{1.0001}} \sqrt{P_i}$.

The supported tick range in Muffin is $[-776363, 776363]$, which the corresponding square root prices maximally fit inside the price range $[2^{-56}, 2^{56}]$. Therefore, the exact supported price range in Muffin is $\left[1.0001^{-776363}, 1.0001^{776363}\right]$.

A few optimizations are done to make finding the next initialized tick easier. Each tier use nested bitmaps to track which ticks are initialized. A tick being initialized means there are positions using it as their price boundary. Also, each tick state tracks its adjacent upper and lower initialized ticks, working as a doubly-linked list. In addition, each tier state caches the next initialized ticks below and above its current tick.

3.4 Tick spacing

Ticks are used for LP positions to represent their upper and lower price boundaries. However, not all ticks can be used. There is a tick spacing setting for each pool, and only ticks that are multiples of that tick spacing can be used.

When a pool is created, it will use the default tick spacing that is set in the protocol.

This default can be changed. Note that the pool's tick spacing can also be changed, in contrast to the immutable nature in other existing AMMs. Also, each pool has only one tick spacing, which applies to all the tiers in the pool.

3.5 Fee

Each tier has its own independent percentage swap fee. To make calculation convenient in the contract, rather than storing the percentage value $(1-\gamma)$, we store the square root of one minus it, i.e. $\sqrt{\gamma}$. In the code, we use the variable name `sqrtGamma` to represent it. It is a 6-digit decimal fixed point stored as an `uint24` variable.

3.6 Liquidity

The amount of active liquidity in a pool tier is measured by the value L, defined as $L = \sqrt{xy}$, where x and y are the current virtual reserves of the two underlying tokens.

The liquidity in the tier state is stored as an `uint128`, but in the tick, position and settlement states, liquidity is stored as a multiple of 2^8 as an `uint96` with a variable name suffix `D8` to indicate this. The primary reason is that we can store liquidity using few bits, and therefore can pack variables more tightly to reduce the amount of `SLOAD` and `SSTORE` operations.

3.7 Fee growths

Each tier keep tracks of two values `feeGrowthGlobal{0,1}`, denoted by f_g. They are the total amounts of fees earned per unit of virtual liquidity throughout the entire history of the tier. Simply put, these are the amounts of fees you would earn if you provide one unit of full-range liquidity in the tier from the moment it is created.

Also, each tick tracks two values `feeGrowthOutside{0,1}`. We denote it by f_o. They are the amounts of the fee growths per unit of liquidity in the range from the tick itself outwards to the end of the supported tick range. You can think of the "outward" direction as the direction at which you stand on the current price looking at the tick and towards the end of the whole price range.

These values are used to find out the fee growths per unit of liquidity inside a particular tick range, `feeGrowthInside{0,1}`, f_r. Specifically, the f_r of a tick range from a to b when the tier's current tick is i is defined as:

$$f_{r,a,b} = \begin{cases} f_g - f_{o,a} - f_{o,b} & a \le i < b \\ f_{o,a} - f_{o,b} & i < a \\ f_{o,b} - f_{o,a} & i \ge b \end{cases}$$

These values f_r are then used to calculate the amount of fees earned by a position. We will cover it in a section below.

3.8 Positions

Each position in a pool is distinct by the combination of position owner address, position reference id, tier id, and position's lower and upper tick boundaries.

3.8.1 Position reference id

Position reference id, or `positionRefId` in the code, is an arbitrary number set by the position owner. It is expected that the position owner is in fact a "position manager" contract, and the position manager can make use of `positionRefId` to separate positions

of the same tick range on the same pool tier that are however owned by different end-users.

3.8.2 Conversion between liquidity and tokens

The quantities of the underlying tokens, x and y, of a position with l units of liquidity are defined as:

$$x = l\left(\frac{1}{\sqrt{P}} - \frac{1}{\sqrt{P_{upper}}}\right)$$

$$y = l\left(\sqrt{P} - \sqrt{P_{lower}}\right)$$

where $P = \min\{\max\{P_{now}, P_{lower}\}, P_{upper}\}$, in which P_{lower} and P_{upper} and the lower and upper price boundaries of the position, and P_{now} is the current price of the pool tier. Rearranging the terms, when an LP is creating a position using x and y units of underlying tokens, the amount of liquidity that the LP receives, l, is defined as:

$$l = \min\left\{\frac{x}{\left(\frac{1}{\sqrt{P}} - \frac{1}{\sqrt{P_{upper}}}\right)}, \frac{y}{\left(\sqrt{P} - \sqrt{P_{lower}}\right)}\right\}$$

3.8.3 Fee calculation

When a position is created, it will snapshot the value of the *fee growth per unit of liquidity* in the position's tick range for each underlying token, i.e. the position's `feeGrowthInside{0,1}` in the code. The protocol can then calculate the position's unclaimed fees by calculating the delta between the snapshot value f_{rs} and the current value f_{rc}, and then multiplying it by the position's liquidity l, i.e.:

$$\text{fee unclaimed} = (f_{rc} - f_{rs}) \cdot l$$

3.8.4 Accounting

When adding liquidity to a position, the position's `feeGrowthInside{0,1}` are updated so as to accrue fees without the immediate need to claim and move the fees from the position to the position owner's internal account, therefore reducing unnecessary `SSTORE` operations. When removing liquidity, the `feeGrowthInside{0,1}` remain unchanged this time, and partial fees are transferred to the position owner's internal account proportionally to the amount of liquidity removed.

$$\text{fee\_out} = \begin{cases} 0 & l' \geq l \\ (f_{rc} - f_{rs})\,(l - l') & l' < l \end{cases}$$

$$f_{rs} := \begin{cases} f_{rc} - \frac{l}{l'}\,(f_{rc} - f_{rs}) & l' \geq l \\ f_{rs} & l' < l \end{cases}$$

LPs can also choose to collect all unclaimed fees at once when removing liquidity, regardless of the amount of liquidity being removed. In this case, the `feeGrowthInside{0,1}` are directly replaced by the current values f_{rc}.

$$\text{fee\_out} = (f_{rc} - f_{rs}) \cdot l$$

$$f_{rs} := f_{rc}$$

4 Appendices

4.1 Deriving sub-order's optimal size

First, we start from the constant product function of the x*y=k AMM:

$$(x + \gamma \cdot x_{in})(y - y_{out}) = L^2$$

Rearranging the terms, we have:

$$y_{out} = y - \frac{L^2}{x + \gamma \cdot x_{in}}$$

Suppose we have a n-tier pool for the tokens X and Y. We want to sell Δ units of token X to the pool. The pool splits our order and eventually we're selling δ_i units of token X to each tier i, i.e. $\sum_{i=1}^{n} \delta_i = \Delta$. We want to maximize the total output amount of token Y we receive from the pool. Using the formula of y_{out} above, we can formulate the optimization problem:

$$\begin{aligned} \text{minimize} \quad & f(\delta_1, ..., \delta_n) = \sum_{i=1}^{n} \left(\frac{L_i^2}{x_i + \gamma_i \delta_i} \right) \\ \text{subject to} \quad & g(\delta_1, ..., \delta_n) = \sum_{i=1}^{n} \delta_i - \Delta = 0 \end{aligned}$$

There are several ways to solve it. Here, we use a more elementary approach. First, we solve the constraint function for one of the variables, say, δ_n:

$$\delta_n = \Delta - \sum_{i=1}^{n-1} \delta_i$$

Then, we substitute this expression back to the f, so we get a new optimization problem with one less variable.

$$\begin{aligned} f(\delta_1, ..., \delta_{n-1}) &= \sum_{i=1}^{n} \frac{L_i^2}{x_i + \gamma_i \delta_i} \\ &= \sum_{i=1}^{n-1} \frac{L_i^2}{x_i + \gamma_i \delta_i} + \frac{L_n^2}{x_n + \gamma_n \delta_n} \\ &= \sum_{i=1}^{n-1} \frac{L_i^2}{x_i + \gamma_i \delta_i} + \frac{L_n^2}{x_n + \gamma_n \left(\Delta - \sum_{i=1}^{n-1} \delta_i \right)} \end{aligned}$$

Then, we take the partial derivative with respect to δ_i:

$$\frac{\partial f}{\partial \delta_i} = \frac{L_i^2 \gamma_i}{(x_i + \gamma_i \delta_i)^2} + \frac{L_n^2 \gamma_n}{\left(x_n + \gamma_n \left(\Delta - \sum_{i=1}^{n-1} \delta_i \right) \right)^2} = 0, \qquad i = 1, ..., n-1$$

For simplicity, we denote the second term above by λ. Then, we'll get a set of equations for δ_i:

$$\delta_i = \frac{L_i}{\sqrt{\gamma_i}\sqrt{\lambda}} - \frac{x_i}{\gamma_i}, \qquad i = 1, ..., n$$

We substitute these equations to the constraint g:

$$\sum_{i=1}^{n} \delta_i = \sum_{i=1}^{n} \left(\frac{L_i}{\sqrt{\gamma_i}\sqrt{\lambda}} - \frac{x_i}{\gamma_i} \right) = \Delta$$

$$\frac{1}{\sqrt{\lambda}} = \frac{\Delta + \sum_{i=1}^{n} \frac{x_i}{\gamma_i}}{\sum_{i=1}^{n} \frac{L_i}{\sqrt{\gamma_i}}}$$

Then, substitute this back to δ_i and we have the solution to the optimization problem, i.e. the token amount δ_i we should sell to each tier i respectively to get an overall best swap rate.

$$\delta_i^* = \frac{L_i}{\sqrt{\gamma_i}} \frac{\Delta + \sum_{j=1}^{n} \frac{x_j}{\gamma_j}}{\sum_{j=1}^{n} \frac{L_j}{\sqrt{\gamma_j}}} - \frac{x_i}{\gamma_i}, \qquad i = 1, ..., n$$

Similarly, we can find the solution to an "exact output" order, i.e. having an exact desired amount of token to buy instead of sell.

$$\delta_i^* = y_i - \frac{L_i}{\sqrt{\gamma_i}} \frac{-\Delta + \sum_{j=1}^{n} y_j}{\sum_{j=1}^{n} \frac{L_j}{\sqrt{\gamma_j}}}, \qquad i = 1, ..., n$$

Note that δ_i^* needs to be non-negative but is however unbounded in the optimization problem. In practice, when a solution is found negative, we set it to zero and then exclude that tier from the optimization problem and solve it again. The worse case asymptotic complexity here is $O(n^2)$, nonetheless the gas usage is still relatively low.

4.2 Relationship between tiers' prices

Let's go further to see the relationship between tier's prices. Specifically, if an order selling the quote currency is split and routed to a set of tiers, the post-swap prices P of any two tiers from the set will behave in this relationship $P_i : P_j = \gamma_j : \gamma_i$. We can show it by using the price formula of AMM, $P = y/x = (L/x)^2$, to express P with the λ defined above.

$$P_i = \left(\frac{L_i}{x_i + \gamma_i \delta_i}\right)^2 = \left(\frac{L_i}{x_i + \gamma_i \left(\frac{L_i}{\sqrt{\gamma_i}\sqrt{\lambda}} - \frac{x_i}{\gamma_i}\right)}\right)^2 = \left(\frac{\sqrt{\lambda}}{\sqrt{\gamma_i}}\right)^2 = \frac{\lambda}{\gamma_i}$$

$$\frac{P_i}{P_j} = \frac{\frac{\lambda}{\gamma_i}}{\frac{\lambda}{\gamma_j}} = \frac{\gamma_j}{\gamma_i}$$

Using the same approach, we can know the relationship between the tiers' prices after being sold the base currency is $P_i : P_j = \gamma_i : \gamma_j$. These lead to the conclusion below for $\gamma_i > \gamma_j$:

$$\frac{\gamma_j}{\gamma_i} \leq \frac{P_i}{P_j} \leq \frac{\gamma_i}{\gamma_j}$$

Note that this is derived without considering the implementation we use to take account of the change in tier's liquidity during a swap in a concentrated liquidity design (as described in section 2.2.3). As a result, the actual price movements will approximately but not perfectly exhibit this pattern.