Uniswap v3 Core

March 2021

Hayden Adams
hayden@uniswap.org

Noah Zinsmeister
noah@uniswap.org

Moody Salem
moody@uniswap.org

River Keefer
river@uniswap.org

Dan Robinson
dan@paradigm.xyz

ABSTRACT

Uniswap v3 is a noncustodial automated market maker implemented for the Ethereum Virtual Machine. In comparison to earlier versions of the protocol, Uniswap v3 provides increased capital efficiency and fine-tuned control to liquidity providers, improves the accuracy and convenience of the price oracle, and has a more flexible fee structure.

1 INTRODUCTION

Automated market makers (AMMs) are agents that pool liquidity and make it available to traders according to an algorithm [5]. Constant function market makers (CFMMs), a broad class of AMMs of which Uniswap is a member, have seen widespread use in the context of decentralized finance, where they are typically implemented as smart contracts that trade tokens on a permissionless blockchain [2].

CFMMs as they are implemented today are often capital inefficient. In the constant product market maker formula used by Uniswap v1 and v2, only a fraction of the assets in the pool are available at a given price. This is inefficient, particularly when assets are expected to trade close to a particular price at all times.

Prior attempts to address this capital efficiency issue, such as Curve [3] and YieldSpace [4], have involved building pools that use different functions to describe the relation between reserves. This requires all liquidity providers in a given pool to adhere to a single formula, and could result in liquidity fragmentation if liquidity providers want to provide liquidity within different price ranges.

In this paper, we present Uniswap v3, a novel AMM that gives liquidity providers more control over the price ranges in which their capital is used, with limited effect on liquidity fragmentation and gas inefficiency. This design does not depend on any shared assumption about the price behavior of the tokens. Uniswap v3 is based on the same constant product reserves curve as earlier versions [1], but offers several significant new features:

- *Concentrated Liquidity*: Liquidity providers (LPs) are given the ability to concentrate their liquidity by "bounding" it within an arbitrary price range. This improves the pool's capital efficiency and allows LPs to approximate their preferred reserves curve, while still being efficiently aggregated with the rest of the pool. We describe this feature in section 2 and its implementation in Section 6.
- *Flexible Fees*: The swap fee is no longer locked at 0.30%. Rather, the fee tier for each pool (of which there can be multiple per asset pair) is set on initialization (Section 3.1). The initially supported fee tiers are 0.05%, 0.30%, and 1%. `UNI` governance is able to add additional values to this set.
- *Protocol Fee Governance*: `UNI` governance has more flexibility in setting the fraction of swap fees collected by the protocol (Section 6.2.2).
- *Improved Price Oracle*: Uniswap v3 provides a way for users to query recent price accumulator values, thus avoiding the need to checkpoint the accumulator value at the exact beginning and end of the period for which a TWAP is being measured. (Section 5.1).

- *Liquidity Oracle*: The contracts expose a time-weighted average liquidity oracle (Section 5.3).

The Uniswap v2 core contracts are non-upgradeable by design, so Uniswap v3 is implemented as an entirely new set of contracts, available here. The Uniswap v3 core contracts are also non-upgradeable, with some parameters controlled by governance as described in Section 4.

2 CONCENTRATED LIQUIDITY

The defining idea of Uniswap v3 is that of *concentrated liquidity*: liquidity bounded within some price range.

In earlier versions, liquidity was distributed uniformly along the

$$x \cdot y = k \tag{2.1}$$

reserves curve, where x and y are the respective reserves of two assets X and Y, and k is a constant [1]. In other words, earlier versions were designed to provide liquidity across the entire price range $(0, \infty)$. This is simple to implement and allows liquidity to be efficiently aggregated, but means that much of the assets held in a pool are never touched.

Having considered this, it seems reasonable to allow LPs to concentrate their liquidity to smaller price ranges than $(0, \infty)$. We call liquidity concentrated to a finite range a *position*. A position only needs to maintain enough reserves to support trading within its range, and therefore can act like a constant product pool with larger reserves (we call these the *virtual reserves*) within that range.

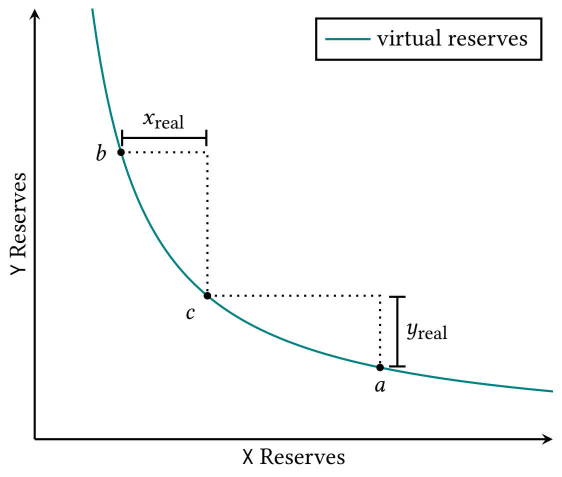

Figure 1: Simulation of Virtual Liquidity

Specifically, a position only needs to hold enough of asset X to cover price movement to its upper bound, because upwards price movement[1] corresponds to depletion of the X reserves. Similarly, it only needs to hold enough of asset Y to cover price movement to its lower bound. Fig. 1 depicts this relationship for a position on a range $[p_a, p_b]$ and a current price $p_c \in [p_a, p_b]$. x_{real} and y_{real} denote the position's real reserves.

When the price exits a position's range, the position's liquidity is no longer active, and no longer earns fees. At that point, its liquidity is composed entirely of a single asset, because the reserves of the other asset must have been entirely depleted. If the price ever reenters the range, the liquidity becomes active again.

The amount of liquidity provided can be measured by the value L, which is equal to $\sqrt{k}$. The real reserves of a position are described by the curve:

$$(x + \frac{L}{\sqrt{p_b}})(y + L\sqrt{p_a}) = L^2 \tag{2.2}$$

This curve is a translation of formula 2.1 such that the position is solvent exactly within its range (Fig. 2).

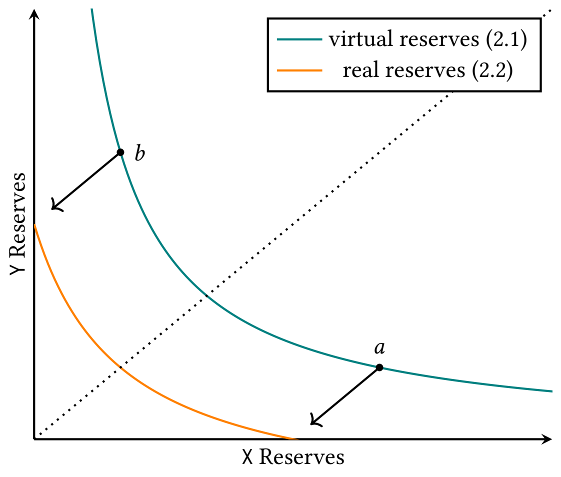

Figure 2: Real Reserves

Liquidity providers are free to create as many positions as they see fit, each on its own price range. In this way, LPs can approximate any desired distribution of liquidity on the price space (see Fig. 3 for a few examples). Moreover, this serves as a mechanism to let the market decide where liquidity should be allocated. Rational LPs can reduce their capital costs by concentrating their liquidity in a narrow band around the current price, and adding or removing tokens as the price moves to keep their liquidity active.

2.1 Range Orders

Positions on very small ranges act similarly to limit orders—if the range is crossed, the position flips from being composed entirely of one asset, to being composed entirely of the other asset (plus accrued fees). There are two differences between this *range order* and a traditional limit order:

- There is a limit to how narrow a position's range can be. While the price is within that range, the limit order might be partially executed.
- When the position has been crossed, it needs to be withdrawn. If it is not, and the price crosses back across that range, the position will be traded back, effectively reversing the trade.

[1]We take asset Y to be the unit of account, which corresponds to `token1` in our implementation.

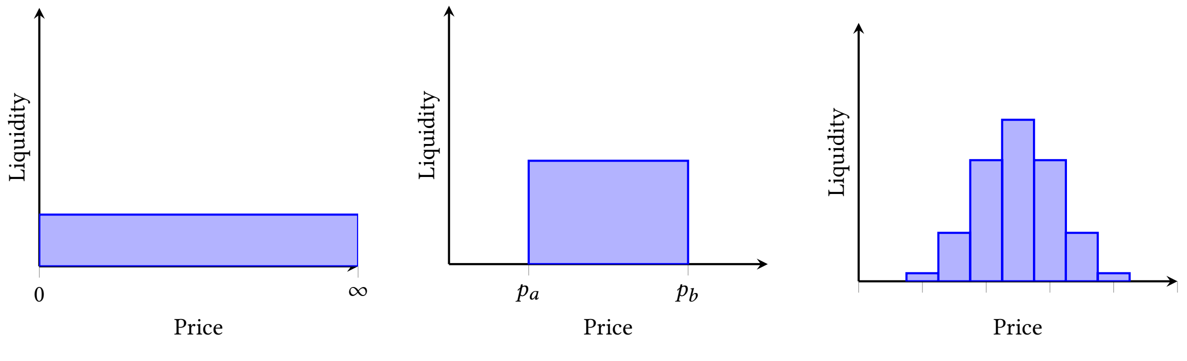

Figure 3: Example Liquidity Distributions

3 ARCHITECTURAL CHANGES

Uniswap v3 makes a number of architectural changes, some of which are necessitated by the inclusion of concentrated liquidity, and some of which are independent improvements.

3.1 Multiple Pools Per Pair

In Uniswap v1 and v2, every pair of tokens corresponds to a single liquidity pool, which applies a uniform fee of 0.30% to all swaps. While this default fee tier historically worked well enough for many tokens, it is likely too high for some pools (such as pools between two stablecoins), and too low for others (such as pools that include highly volatile or rarely traded tokens).

Uniswap v3 introduces multiple pools for each pair of tokens, each with a different swap fee. All pools are created by the same factory contract. The factory contract initially allows pools to be created at three fee tiers: 0.05%, 0.30%, and 1%. Additional fee tiers can be enabled by UNI governance.

3.2 Non-Fungible Liquidity

3.2.1 Non-Compounding Fees. Fees earned in earlier versions were continuously deposited in the pool as liquidity. This meant that liquidity in the pool would grow over time, even without explicit deposits, and that fee earnings compounded.

In Uniswap v3, due to the non-fungible nature of positions, this is no longer possible. Instead, fee earnings are stored separately and held as the tokens in which the fees are paid (see Section 6.2.2).

3.2.2 Removal of Native Liquidity Tokens. In Uniswap v1 and v2, the pool contract is also an ERC-20 token contract, whose tokens represent liquidity held in the pool. While this is convenient, it actually sits uneasily with the Uniswap v2 philosophy that anything that does not need to be in the core contracts should be in the periphery, and blessing one "canonical" ERC-20 implementation discourages the creation of improved ERC-20 token wrappers. Arguably, the ERC-20 token implementation should have been in the periphery, as a wrapper on a single liquidity position in the core contract.

The changes made in Uniswap v3 force this issue by making completely fungible liquidity tokens impossible. Due to the custom liquidity provision feature, fees are now collected and held by the pool as individual tokens, rather than automatically reinvested as liquidity in the pool.

As a result, in v3, the pool contract does not implement the ERC-20 standard. Anyone can create an ERC-20 token contract in the periphery that makes a liquidity position more fungible, but it will have to have additional logic to handle distribution of, or reinvestment of, collected fees. Alternatively, anyone could create a periphery contract that wraps an individual liquidity position (including collected fees) in an ERC-721 non-fungible token.

4 GOVERNANCE

The factory has an `owner`, which is initially controlled by UNI tokenholders.[2] The owner does not have the ability to halt the operation of any of the core contracts.

As in Uniswap v2, Uniswap v3 has a protocol fee that can be turned on by UNI governance. In Uniswap v3, UNI governance has more flexibility in choosing the fraction of swap fees that go to the protocol, and is able to choose any fraction $\frac{1}{N}$ where $4 \leq N \leq 10$, or 0. This parameter can be set on a per-pool basis.

UNI governance also has the ability to add additional fee tiers. When it adds a new fee tier, it can also define the `tickSpacing` (see Section 6.1) corresponding to that fee tier. Once a fee tier is added to the factory, it cannot be removed (and the `tickSpacing` cannot be changed). The initial fee tiers and tick spacings supported are 0.05% (with a tick spacing of 10, approximately 0.10% between initializable ticks), 0.30% (with a tick spacing of 60, approximately 0.60% between initializable ticks), and 1% (with a tick spacing of 200, approximately 2.02% between ticks.

Finally, UNI governance has the power to transfer ownership to another address.

5 ORACLE UPGRADES

Uniswap v3 includes three significant changes to the time-weighted average price (TWAP) oracle that was introduced by Uniswap v2.

Most significantly, Uniswap v3 removes the need for users of the oracle to track previous values of the accumulator externally. Uniswap v2 requires users to checkpoint the accumulator value at both the beginning and end of the time period for which they

[2]Specifically, the owner will be initialized to the Timelock contract from UNI governance, 0x1a9c8182c09f50c8318d769245bea52c32be35bc.

wanted to compute a TWAP. Uniswap v3 brings the accumulator checkpoints into core, allowing external contracts to compute on-chain TWAPs over recent periods without storing checkpoints of the accumulator value.

Another change is that instead of accumulating the sum of prices, allowing users to compute the arithmetic mean TWAP, Uniswap v3 tracks the sum of *log* prices, allowing users to compute the *geometric mean* TWAP.

Finally, Uniswap v3 adds a liquidity accumulator that is tracked alongside the price accumulator. This liquidity accumulator can be used by other contracts to inform a decision on which of the pools corresponding to a pair (see section 3.1) will have the most reliable TWAP.

5.1 Oracle Observations

As in Uniswap v2, Uniswap v3 tracks a running accumulator of the price at the beginning of each block, multiplied by the number of seconds since the last block.

A pool in Uniswap v2 stores only the most recent value of this price accumulator—that is, the value as of the last block in which a swap occurred. When computing average prices in Uniswap v2, it is the responsibility of the external caller to provide the previous value of the price accumulator. With many users, each will have to provide their own methodology for checkpointing previous values of the accumulator, or coordinate on a shared method to reduce costs. And there is no way to guarantee that every block in which the pool is touched will be reflected in the accumulator.

In Uniswap v3, the pool stores a list of previous values for the accumulator. It does this by automatically checkpointing the accumulator value every time the pool is touched for the first time in a block, cycling through an array where the oldest checkpoint is eventually overwritten by a new one, similar to a circular buffer. While this array initially only has room for a single checkpoint, anyone can initialize additional storage slots to lengthen the array, extending to as many as 65,536 checkpoints.[3] This imposes the one-time gas cost of initializing additional storage slots for this array on whoever wants this pair to checkpoint more slots.

The pool exposes the array of past observations to users, as well as a convenience function for finding the (interpolated) accumulator value at any historical timestamp within the checkpointed period.

5.2 Geometric Mean Price Oracle

Uniswap v2 maintains two price accumulators—one for the price of `token0` in terms of token1, and one for the price of `token1` in terms of token0. Users can compute the time-weighted arithmetic mean of the prices over any period, by subtracting the accumulator value at the beginning of the period from the accumulator at the end of the period, then dividing the difference by the number of seconds in the period. Note that accumulators for `token0` and `token1` are tracked separately, since the time-weighted arithmetic mean price of `token0` is not equivalent to the reciprocal of the time-weighted arithmetic mean price of `token1`.

Using the time-weighted *geometric* mean price, as Uniswap v3 does, avoids the need to track separate accumulators for these ratios. The geometric mean of a set of ratios is the reciprocal of the geometric mean of their reciprocals. It is also easy to implement in Uniswap v3 because of its implementation of custom liquidity provision, as described in section 6. In addition, the accumulator can be stored in a smaller number of bits, since it tracks $\log P$ rather than P, and $\log P$ can represent a wide range of prices with consistent precision.[4] Finally, there is a theoretical argument that the time-weighted geometric mean price should be a truer representation of the average price.[5]

Instead of tracking the cumulative sum of the price P, Uniswap v3 accumulates the cumulative sum of the current tick index ($log_{1.0001}P$, the logarithm of price for base 1.0001, which is precise up to 1 basis point). The accumulator at any given time is equal to the sum of $log_{1.0001}(P)$ for every second in the history of the contract:

$$a_t = \sum_{i=1}^{t} \log_{1.0001}(P_i) \tag{5.1}$$

We want to estimate the geometric mean time-weighted average price (p_{t_1,t_2}) over any period t_1 to t_2.

$$P_{t_1,t_2} = \left(\prod_{i=t_1}^{t_2} P_i\right)^{\frac{1}{t_2-t_1}} \tag{5.2}$$

To compute this, you can look at the accumulator's value at t_1 and at t_2, subtract the first value from the second, divide by the number of seconds elapsed, and compute 1.0001^x to compute the time weighted geometric mean price.

$$\log_{1.0001}\left(P_{t_1,t_2}\right) = \frac{\sum_{i=t_1}^{t_2} \log_{1.0001}(P_i)}{t_2 - t_1} \tag{5.3}$$

$$\log_{1.0001}\left(P_{t_1,t_2}\right) = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1} \tag{5.4}$$

$$P_{t_1,t_2} = 1.0001^{\frac{a_{t_2}-a_{t_1}}{t_2-t_1}} \tag{5.5}$$

5.3 Liquidity Oracle

In addition to the time weighted average price, Uniswap v3 also tracks an accumulator of the current value of L (the virtual liquidity currently in range) at the beginning of each block. This can be used by on-chain contracts to make their oracles stronger (such as by evaluating which fee-tier pool to use the oracle from). This liquidity accumulator's values are checkpointed along with the price accumulator.

[3]The maximum of 65,536 checkpoints allows fetching checkpoints for at least 9 days after they are written, assuming 13 seconds pass between each block and a checkpoint is written every block.

[4]In order to support tolerable precision across all possible prices, Uniswap v2 represents each price as a 224-bit fixed-point number. Uniswap v3 only needs to represent $log_{1.0001}P$ as a signed 24-bit number, and still can detect price movements of one tick, or 1 basis point.

[5]While arithmetic mean TWAPs are much more widely used, they should theoretically be less accurate in measuring a geometric Brownian motion process (which is how price movements are usually modeled). The arithmetic mean of a geometric Brownian motion process will tend to overweight higher prices (where small percentage movements correspond to large absolute movements) relative to lower ones.

6 IMPLEMENTING CONCENTRATED LIQUIDITY

The rest of this paper describes how concentrated liquidity provision works, and gives a high-level description of how it is implemented in the contracts.

6.1 Ticks and Ranges

To implement custom liquidity provision, the space of possible prices is demarcated by discrete *ticks*. Liquidity providers can provide liquidity in a range between any two ticks (which need not be adjacent).

Each range can be specified as a pair of signed integer *tick indices*: a lower tick (i_l) and an upper tick (i_u). Ticks represent prices at which the virtual liquidity of the contract can change. We will assume that prices are always expressed as the price of one of the tokens—called `token0`—in terms of the other token—`token1`. The assignment of the two tokens to `token0` and `token1` is arbitrary and does not affect the logic of the contract (other than through possible rounding errors).

Conceptually, there is a tick at every price p that is an integer power of 1.0001. Identifying ticks by an integer index i, the price at each is given by:

$$p(i) = 1.0001^i \tag{6.1}$$

This has the desirable property of each tick being a .01% (1 basis point) price movement away from each of its neighboring ticks.

For technical reasons explained in 6.2.1, however, pools actually track ticks at every *square root price* that is an integer power of $\sqrt{1.0001}$. Consider the above equation, transformed into square root price space:

$$\sqrt{p}(i) = \sqrt{1.0001}^i = 1.0001^{\frac{i}{2}} \tag{6.2}$$

As an example, $\sqrt{p}(0)$—the square root price at tick 0—is 1, $\sqrt{p}(1)$ is $\sqrt{1.0001} \approx 1.00005$, and $\sqrt{p}(-1)$ is $\frac{1}{\sqrt{1.0001}} \approx 0.99995$.

When liquidity is added to a range, if one or both of the ticks is not already used as a bound in an existing position, that tick is *initialized*.

Not every tick can be initialized. The pool is instantiated with a parameter, `tickSpacing` (t_s); only ticks with indexes that are divisible by `tickSpacing` can be initialized. For example, if `tickSpacing` is 2, then only even ticks (...-4, -2, 0, 2, 4...) can be initialized. Small choices for `tickSpacing` allow tighter and more precise ranges, but may cause swaps to be more gas-intensive (since each initialized tick that a swap crosses imposes a gas cost on the swapper).

Whenever the price crosses an initialized tick, virtual liquidity is kicked in or out. The gas cost of an initialized tick crossing is constant, and is not dependent on the number of positions being kicked in or out at that tick.

Ensuring that the right amount of liquidity is kicked in and out of the pool when ticks are crossed, and ensuring that each position earns its proportional share of the fees that were accrued while it was within range, requires some accounting within the pool. The pool contract uses storage variables to track state at a *global* (per-pool) level, at a *per-tick* level, and at a *per-position* level.

6.2 Global State

The global state of the contract includes seven storage variables relevant to swaps and liquidity provision. (It has other storage variables that are used for the oracle, as described in section 5.)

Type	Variable Name	Notation
`uint128`	`liquidity`	L
`uint160`	`sqrtPriceX96`	$\sqrt{P}$
`int24`	`tick`	i_c
`uint256`	`feeGrowthGlobal0X128`	$f_{g,0}$
`uint256`	`feeGrowthGlobal1X128`	$f_{g,1}$
`uint128`	`protocolFees.token0`	$f_{p,0}$
`uint128`	`protocolFees.token1`	$f_{p,1}$

Table 1: Global State

6.2.1 *Price and Liquidity.*

In Uniswap v2, each pool contract tracks the pool's current reserves, x and y. In Uniswap v3, the contract could be thought of as having *virtual reserves*—values for x and y that allow you to describe the contract's behavior (between two adjacent ticks) as if it followed the constant product formula.

Instead of tracking those virtual reserves, however, the pool contract tracks two different values: `liquidity` (L) and `sqrtPrice` ($\sqrt{P}$). These could be computed from the virtual reserves with the following formulas:

$$L = \sqrt{xy} \tag{6.3}$$

$$\sqrt{P} = \sqrt{\frac{y}{x}} \tag{6.4}$$

Conversely, these values could be used to compute the virtual reserves:

$$x = \frac{L}{\sqrt{P}} \tag{6.5}$$

$$y = L \cdot \sqrt{P} \tag{6.6}$$

Using L and $\sqrt{P}$ is convenient because only one of them changes at a time. Price (and thus $\sqrt{P}$) changes when swapping within a tick; liquidity changes when crossing a tick, or when minting or burning liquidity. This avoids some rounding errors that could be encountered if tracking virtual reserves.

You may notice that the formula for liquidity (based on virtual reserves) is similar to the formula used to initialize the quantity of liquidity tokens (based on actual reserves) in Uniswap v2. before any fees have been earned. In some ways, liquidity can be thought of as virtual liquidity tokens.

Alternatively, liquidity can be thought of as the amount that `token1` reserves (either actual or virtual) changes for a given change in $\sqrt{P}$:

$$L = \frac{\Delta Y}{\Delta \sqrt{P}} \tag{6.7}$$

We track $\sqrt{P}$ instead of P to take advantage of this relationship, and to avoid having to take any square roots when computing swaps, as described in section 6.2.3.

The global state also tracks the current tick index as `tick` (i_c), a signed integer representing the current tick (more specifically, the nearest tick below the current price). This is an optimization (and a way of avoiding precision issues with logarithms), since at any time, you should be able to compute the current tick based on the current `sqrtPrice`. Specifically, at any given time, the following equation should be true:

$$i_c = \left\lfloor \log_{\sqrt{1.0001}} \sqrt{P} \right\rfloor \tag{6.8}$$

6.2.2 Fees. Each pool is initialized with an immutable value, `fee` (γ), representing the fee paid by swappers in units of hundredths of a basis point (0.0001%).

It also tracks the current protocol fee, ϕ (which is initialized to zero, but can changed by UNI governance).[6] This number gives you the fraction of the fees paid by swappers that currently goes to the protocol rather than to liquidity providers. ϕ only has a limited set of permitted values: 0, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9, or 1/10.

The global state also tracks two numbers: `feeGrowthGlobal0` ($f_{g,0}$) and `feeGrowthGlobal1` ($f_{g,1}$). These represent the total amount of fees that have been earned per unit of virtual liquidity (L), over the entire history of the contract. You can think of them as the total amount of fees that would have been earned by 1 unit of unbounded liquidity that was deposited when the contract was first initialized. They are stored as fixed-point unsigned 128x128 numbers. Note that in Uniswap v3, fees are collected in the tokens themselves rather than in liquidity, for reasons explained in section 3.2.1.

Finally, the global state tracks the total accumulated uncollected protocol fee in each token, `protocolFees0` ($f_{p,0}$) and `protocolFees1` ($f_{p,1}$). This is an unsigned `uint128`. The accumulated protocol fees can be collected by UNI governance, by calling the `collectProtocol` function.

6.2.3 Swapping Within a Single Tick. For small enough swaps, that do not move the price past a tick, the contracts act like an $x \cdot y = k$ pool.

Suppose γ is the fee, i.e., 0.003, and y_{in} as the amount of `token1` sent in.

First, `feeGrowthGlobal1` and `protocolFees1` are incremented:

$$\Delta f_{g,1} = y_{in} \cdot \gamma \cdot (1 - \phi) \tag{6.9}$$

$$\Delta f_{p,1} = y_{in} \cdot \gamma \cdot \phi \tag{6.10}$$

Δy is the increase in y (after the fee is taken out).

$$\Delta y = y_{in} \cdot (1 - \gamma) \tag{6.11}$$

If you used the computed virtual reserves (x and y) for the `token0` and `token1` balances, then this formula could be used to find the amount of `token0` sent out:

$$x_{end} = \frac{x \cdot y}{y + \Delta y} \tag{6.12}$$

But remember that in v3, the contract actually tracks liquidity (L) and square root of price ($\sqrt{P}$) instead of x and y. We could compute x and y from those values, and then use those to calculate the execution price of the trade. But it turns out that there are simple formulas that describe the relationship between $\Delta\sqrt{P}$ and Δy, for a given L (which can be derived from formula 6.7):

$$\Delta\sqrt{P} = \frac{\Delta y}{L} \tag{6.13}$$

$$\Delta y = \Delta\sqrt{P} \cdot L \tag{6.14}$$

There are also simple formulas that describe the relationship between $\Delta\frac{1}{\sqrt{P}}$ and Δx:

$$\Delta\frac{1}{\sqrt{P}} = \frac{\Delta x}{L} \tag{6.15}$$

$$\Delta x = \Delta\frac{1}{\sqrt{P}} \cdot L \tag{6.16}$$

When swapping one token for the other, the pool contract can first compute the new $\sqrt{P}$ using formula 6.13 or 6.15, and then can compute the amount of `token0` or `token1` to send out using formula 6.14 or 6.16.

These formulas will work for any swap that does not push $\sqrt{P}$ past the price of the next initialized tick. If the computed $\Delta\sqrt{P}$ would cause $\sqrt{P}$ to move past that next initialized tick, the contract must only cross up to that tick—using up only part of the swap—and then cross the tick, as described in section 6.3.1, before continuing with the rest of the swap.

6.2.4 Initialized Tick Bitmap. If a tick is not used as the endpoint of a range with any liquidity in it—that is, if the tick is uninitialized—then that tick can be skipped during swaps.

As an optimization to make finding the next initialized tick more efficient, the pool tracks a bitmap `tickBitmap` of initialized ticks. The position in the bitmap that corresponds to the tick index is set to 1 if the tick is initialized, and 0 if it is not initialized.

When a tick is used as an endpoint for a new position, and that tick is not currently used by any other liquidity, the tick is initialized, and the corresponding bit in the bitmap is set to 1. An initialized tick can become uninitialized again if all of the liquidity for which it is an endpoint is removed, in which case that tick's position on the bitmap is zeroed out.

6.3 Tick-Indexed State

The contract needs to store information about each tick in order to track the amount of net liquidity that should be added or removed when the tick is crossed, as well as to track the fees earned above and below that tick.

The contract stores a mapping from tick indexes (`int24`) to the following four values:

Type	Variable Name	Notation
`int128`	`liquidityNet`	ΔL
`uint128`	`liquidityGross`	L_g
`uint256`	`feeGrowthOutside0X128`	$f_{o,0}$
`uint256`	`feeGrowthOutside1X128`	$f_{o,1}$

Table 2: Tick-Indexed State

[6]Technically, the storage variable called "protocolFee" is the denominator of this fraction (or is zero, if ϕ is zero).

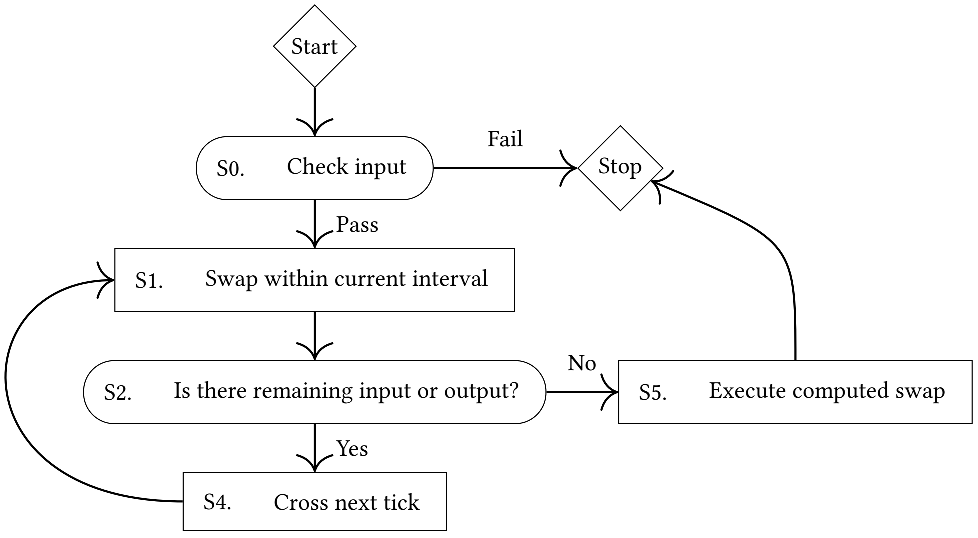
Start
S0. Check input
Fail
Stop
Pass
S1. Swap within current interval
S2. Is there remaining input or output?
No
S5. Execute computed swap
Yes
S4. Cross next tick

Figure 4: Swap Control Flow

Each tick tracks ΔL, the total amount of liquidity that should be kicked in or out when the tick is crossed. The tick only needs to track one signed integer: the amount of liquidity added (or, if negative, removed) when the tick is crossed going left to right. This value does not need to be updated when the tick is crossed (but only when a position with a bound at that tick is updated).

We want to be able to uninitialize a tick when there is no longer any liquidity referencing that tick. Since ΔL is a net value, it's necessary to track a gross tally of liquidity referencing the tick, `liquidityGross`. This value ensures that even if net liquidity at a tick is 0, we can still know if a tick is referenced by at least one underlying position or not, which tells us whether to update the tick bitmap.

`feeGrowthOutside{0,1}` are used to track how many fees were accumulated within a given range. Since the formulas are the same for the fees collected in `token0` and token1, we will omit that subscript for the rest of this section.

You can compute the fees earned per unit of liquidity in token 0 above (f_a) and below (f_b) a tick i with a formula that depends on whether the price is currently within or outside that range—that is, whether the current tick index i_c is greater than or equal to i:

$$f_a(i) = \begin{cases} f_g - f_o(i) & i_c \geq i \\ f_o(i) & i_c < i \end{cases} \tag{6.17}$$

$$f_b(i) = \begin{cases} f_o(i) & i_c \geq i \\ f_g - f_o(i) & i_c < i \end{cases} \tag{6.18}$$

We can use these functions to compute the total amount of cumulative fees per share f_{i_l,i_u} in the range between two ticks—a lower tick i_l and an upper tick i_u:

$$f_{i_l,i_u}(0) = f_g - f_b(i_l) - f_a(i_u) \tag{6.19}$$

$f_{o,1}$ needs to be updated each time the tick is crossed. Specifically, as a tick i is crossed in either direction, its f_o (for each token) should be updated as follows:

$$f_o(i) := f_g - f_o(i) \tag{6.20}$$

f_o is only needed for ticks that are used as either the lower or upper bound for at least one position. As a result, for efficiency, f_o is not initialized (and thus does not need to be updated when crossed) until a position is created that has that tick as one of its bounds. When f_o is initialized for a tick i, the value—by convention—is chosen as if all of the fees earned to date had occurred below that tick:

$$f_o := \begin{cases} f_g & i_c \geq i \\ 0 & i_c < i \end{cases} \tag{6.21}$$

Note that since f_o values for different ticks could be initialized at different times, comparisons of the f_o values for different ticks are not meaningful, and there is no guarantee that values for f_o will be consistent. This does not cause a problem for per-position accounting, since, as described below, all the position needs to know is the growth in g within a given range since that position was last touched.

Finally, the contract also stores `secondsOutside` (t_o) for each tick. This can be thought of as the amount of time spent on the other side of this tick (relative to the current price), and can be used to compute the number of seconds that have been spend above or below a tick for a particular range. This value is not used within the contract, but is tracked for the convenience of external contracts that want to know how many seconds a given position has been active.

As with f_a and f_b, the time spent above (t_a) and below (t_b) a given tick is computed differently based on whether the current price is within that range, and the time spent within a range (t_r) can be computed using the values of t_a and t_b

$$t_a(i) = \begin{cases} t - t_o(i) & i_c \geq i \\ t_o(i) & i_c < i \end{cases} \tag{6.22}$$

$$t_b(i) = \begin{cases} t_o(i) & i_c \geq i \\ t - t_o(i) & i_c < i \end{cases} \tag{6.23}$$

$$t_r(i_l, i_u) = t - t_b(i_l) - t_a(i_u) \tag{6.24}$$

The number of seconds spent within a range between two times t_1 and t_2 can be computed by recording the value of $t_r(i_l, i_u)$ at t_1 and at t_2, and subtracting the former from the latter.

Like f_o, t_o does not need to be tracked for ticks that are not on the edge of any position. Therefore, it is not initialized until a position is created that is bounded by that tick. By convention, it is initialized as if every second since the Unix timestamp 0 had been spent below that tick:

$$t_o(i) := \begin{cases} t & i_c \geq i \\ 0 & i_c < i \end{cases} \tag{6.25}$$

As with f_o values, t_o values are not meaningfully comparable across different ticks. t_o is only meaningful in computing the number of seconds that liquidity was within some particular range between some defined start time (which must be after t_o was initialized for both ticks) and some end time.

6.3.1 Crossing a Tick. As described in section 6.2.3, Uniswap v3 acts like it obeys the constant product formula when swapping between initialized ticks. When a swap crosses an initialized tick, however, the contract needs to add or remove liquidity, to ensure that no liquidity provider is insolvent. This means the ΔL is fetched from the tick, and applied to the global L.

The contract also needs to update the tick's own state, in order to track the fees earned (and seconds spent) within ranges bounded by this tick. The `feeGrowthOutside{0,1}` and `secondsOutside` values are updated to both reflect current values, as well as the proper orientation relative to the current tick:

$$f_o := f_g - f_o \tag{6.26}$$

$$t_o := t - t_o \tag{6.27}$$

Once a tick is crossed, the swap can continue as described in section 6.2.3 until it reaches the next initialized tick.

6.4 Position-Indexed State

The contract has a mapping from user (an `address`), lower bound (a tick index, `int24`), and upper bound (a tick index, `int24`) to a specific `Position` struct. Each `Position` tracks three values:

Type	Variable Name	Notation
`uint128`	`liquidity`	l
`uint256`	`feeGrowthInside0LastX128`	$f_{r,0}(t_0)$
`uint256`	`feeGrowthInside1LastX128`	$f_{r,1}(t_0)$

Table 3: Position-Indexed State

`liquidity` (l) means the amount of virtual liquidity that the position represented the last time this position was touched. Specifically, `liquidity` could be thought of as $\sqrt{x \cdot y}$, where x and y are the respective amounts of virtual `token0` and virtual `token1` that this liquidity contributes to the pool at any time that it is within range. Unlike pool shares in Uniswap v2 (where the value of each share grows over time), the units for liquidity do not change as fees are accumulated; it is always measured as $\sqrt{x \cdot y}$, where x and y are quantities of `token0` and token1, respectively.

This `liquidity` number does not reflect the fees that have been accumulated since the contract was last touched, which we will call *uncollected fees*. Computing these uncollected fees requires additional stored values on the position, `feeGrowthInside0Last` ($f_{r,0}(t_0)$) and `feeGrowthInside1Last` ($f_{r,1}(t_0)$), as described below.

6.4.1 setPosition. The `setPosition` function allows a liquidity provider to update their position.

Two of the arguments to `setPosition` —`lowerTick` and `upperTick`— when combined with the `msg.sender`, together specify a position.

The function takes one additional parameter, `liquidityDelta`, to specify how much virtual liquidity the user wants to add or (if negative) remove.

First, the function computes the uncollected fees (f_u) that the position is entitled to, in each token.[7] The amount collected in fees is credited to the user and netted against the amount that they would send in or out for their virtual liquidity deposit.

To compute uncollected fees of a token, you need to know how much f_r for the position's range (calculated from the range's i_l and i_r as described in section 6.3) has grown since the last time fees were collected for that position. The growth in fees in a given range per unit of liquidity over between times t_0 and t_1 is simply $f_r(t_1) - f_r(t_0)$ (where $f_r(t_0)$ is stored in the position as `feeGrowthInside{0,1}Last`, and $f_r(t_1)$ can be computed from the current state of the ticks). Multiplying this by the position's `liquidity` gives us the total uncollected fees in token 0 for this position:

$$f_u = l \cdot (f_r(t_1) - f_r(t_0)) \tag{6.28}$$

Then, the contract updates the position's `liquidity` by adding `liquidityDelta`. It also adds `liquidityDelta` to the `liquidityNet` value for the tick at the bottom end of the range, and subtracts it from the `liquidityNet` at the upper tick (to reflect that this new liquidity would be added when the price crosses the lower tick going up, and subtracted when the price crosses the upper tick going up). If the pool's current price is within the range of this position, the contract also adds `liquidityDelta` to the contract's global `liquidity` value.

Finally, the pool transfers tokens from (or, if `liquidityDelta` is negative, to) the user, corresponding to the amount of liquidity burned or minted.

The amount of `token0` (ΔX) or `token1` (ΔY) that needs to be deposited can be thought of as the amount that would be sold from the position if the price were to move from the current price (P) to the upper tick or lower tick (for `token0` or `token1`, respectively). These formulas can be derived from formulas 6.14 and 6.16, and depend on whether the current price is below, within, or above the range of the position:

[7] Since the formulas for computing uncollected fees in each token are the same, we will omit that subscript for the rest of this section.

$$\Delta Y = \begin{cases} 0 & i_c < i_l \\ \Delta L \cdot (\sqrt{P} - \sqrt{p(i_l)}) & i_l \le i_c < i_u \\ \Delta L \cdot (\sqrt{p(i_u)} - \sqrt{p(i_l)}) & i_c \ge i_u \end{cases} \tag{6.29}$$

$$\Delta X = \begin{cases} \Delta L \cdot (\frac{1}{\sqrt{p(i_l)}} - \frac{1}{\sqrt{p(i_u)}}) & i_c < i_l \\ \Delta L \cdot (\frac{1}{\sqrt{P}} - \frac{1}{\sqrt{p(i_u)}}) & i_l \le i_c < i_u \\ 0 & i_c \ge i_u \end{cases} \tag{6.30}$$

REFERENCES

[1] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2020. *Uniswap v2 Core.* Retrieved Feb 24, 2021 from https://uniswap.org/whitepaper.pdf

[2] Guillermo Angeris and Tarun Chitra. 2020. Improved Price Oracles: Constant Function Market Makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies (AFT '20).* Association for Computing Machinery, New York, NY, United States, 80–91. https://doi.org/10.1145/3419614.3423251

[3] Michael Egorov. 2019. *StableSwap - Efficient Mechanism for Stablecoin Liquidity.* Retrieved Feb 24, 2021 from https://www.curve.fi/stableswap-paper.pdf

[4] Allan Niemerg, Dan Robinson, and Lev Livnev. 2020. *YieldSpace: An Automated Liquidity Provider for Fixed Yield Tokens.* Retrieved Feb 24, 2021 from https://yield.is/YieldSpace.pdf

[5] Abraham Othman. 2012. *Automated Market Making: Theory and Practice.* Ph.D. Dissertation. Carnegie Mellon University.

DISCLAIMER