How Liquidity Position Earnings Grow in UniswapV3

February 14, 2025

Introduction

When collecting tokens via the `collect(...)` function in the uniswapV3 protocol the value of `position.tokensOwed` along with `amountRequested` [an input parameter] determines the amount which a liquidity position holder is able to collect. For those interested in the the programmatic details of liquidity investments it may be of interest to understand in detail how this value changes.

Firstly, `position.tokensOwed` is only affected via the `_udpate(...)` function in the pool contract (or `pool::_udpatePosition(...)` in an abreviated notation). `pool::_udpate(...)` function is a private function which is only called by `pool::_modifyPositio` `(...) -- yet another private function which is only invoked via \lstinline`pool::mint(...)— or `pool::burn(...)`. Note that, though both of the latter functions are marked as 'external', because these functions utilize the `_modifyPosition(...)` the `noDelegateCall` modifier is applied and this has the effect that only contracts in the uniswap V3 protocol are able to call `pool::mint(...)` and `pool::burn(...)` (this largely amounts to the `NonfungiblePositionManager.sol` contract which, in addition to having its own functions for minting/ burning, also has the functions `addLiquidity` and `decreaseLiquidity` for modifying invested liquidity amounts once a position has already been minted).

That was a mouth-full. Unfortunately, it gets worse. Again, our aim here is to narrow our focus to simply what affects the value of `position.tokensOwed` which, by whatever route it is called, is ultimately only affected by `pool::_udpatePosition(...)` .

The `pool::_udpatePosition(...)` function will update the parameter `position.tokensOwed` as follows,

$$\begin{aligned}
&if\ \Delta L == 0\{L_{Next} = self.L \\
&\quad\}else\{L_{Next} = self.L + \Delta L\} \\
&position.tokensOwed = \frac{(fGI - position.fGIL) * self.L}{Q128} \\
&position.fGIL = fGI
\end{aligned} \tag{1}$$

where the following abbreviations have been used:

$$\begin{aligned}
&\text{position [or 'self']} = positions.get(owner, tickLower, tickUpper) &&: \text{liquidity position stored in Position.sol library} \\
&\Delta L = deltaLiquidity &&: \text{amount of liquidity being added/subtracted} \\
&position.L = position.liquidity &&: \text{amount of liquidity owned by position} \\
&fGI = feeGrowthInside &&: \text{all-time fee-growth per unit liquidity inside position} \\
&position.fGIL = position.feeGrowthInsideLast &&: \text{last updated value of fGI [see above]} \\
&* Q128 = 0x100000000000000000000000000000000 &&: [= 340282366920938463463374607431768211456]
\end{aligned} \tag{2}$$

later we'll also use the following abbreviations,

$$
\begin{aligned}
fGO &= feeGrowthOut &&: \\
fGG &= feeGrowthGlobal &&: \\
LG &= liquidityGross &&: \\
L_{Next} &= liquidityNext &&: \\
t_0 &= slot0.tick &&: \\
t_L &= tick_{Lower}, t_U = tick_{Upper} && \\
& &&: *Q96 = 0x1000000000000000000000000 \quad : [= 79228162514264337593543950336]
\end{aligned}
\tag{3}
$$

Actually, the calculations for `tokensOwed` shown above are computed in the `positions.update(...)` function of the `Position.sol` peripherial contract [`Position::positions.update(...)` is called by the `pool::_update(...)`]. `The user might also wonder how \lstinline` fGI — was calculated to begin with. In the pool contract this calculation is performed with the function `getFeeGrowthInside(...)` which is defined in yet another peripherial contract called `Ticks.sol`. Finally, the astute reader might wonder how the quantities `fGI` and `self.L` will be affected when liquidity is not actively being added or subtracted, yet when token swaps are occuring whithin or across the boundaries of the position.

Hence, the need for a flow-chart.

The uniswapV3 protocol is complex, and it is unfortunately not entirely possible to isolate most if any single function within the core pool contract without being lured down a rabbit hole that leads to a series of interconnected tunnels – other core and peripherial contracts which are designed to aid the pool contract in executing operations. Of course there is always the theoretical understanding of what is occuring, and really the aspiring developer / quantitative investor ought to have a grasp of both. The following chart will hopefully aid in this by putting the operations involved when adding/ subtracting liquidity to a position in a graphical sequential [*and color-coded] flow-graph. When appropriate, mathematical notation will be used instead of code so as to more clearly demonstrate what is occuring, i.e. $self.L = self.L + \Delta L$ may be more intuitively readable than the coded lines,

```
[Position.sol :: update(...)]:
        uint128 liquidityNext;
        if (liquidityDelta == 0) {
            require(_self.liquidity > 0, 'NP'); // disallow pokes for 0 liquidity
  positions
            liquidityNext = _self.liquidity;
        } else {
            liquidityNext = LiquidityMath.addDelta(_self.liquidity, liquidityDelta);
        }
        ...
        ...
        if (liquidityDelta != 0) self.liquidity = liquidityNext;
```

which would perhaps lead one who suffers a mild form of OCD to examine the `LiquidityMath.sol` contract in detail. Details are important, but so too is it vital to be able to narrow down what is ultimately of interest, then approach the details such as dealing with potential overflow calculations (whaat `LiquidityMath.addDelta` does) or of attempting to update positions with zero liquidity at a later point.

Included in the flowish-graph at the most relevant point is an outline of how `fGI` and other relevant factors used to calculate its value are affected by the execution of the `pool::swap(...)` function. Before jumping into the flow-graph, let us first classify parameters which are altered when adding/ subtracting liquidity directly 'ACTIVE parameters' as they require an act of investing liquidity to change them, and parameters which are altered only when swapping as 'PASSIVE parameters' as they will [potentially] create passive income once a position has been minted. The following summarizing notes are good to have in mind as one who is new to the v3 protocol attempts to work through the flow-chart:

- `position.tokensOwed = (fGI - position.fGIL) * position.L/Q128` grows via positions.update in `pool::_update(...)`] i.e. only when minting/ burning or adding/decreasing liquidity through the NonfungiblePositionManager.sol contract. But IN BETWEEN minting and burning how does the quantity (fGI - position.fGIL)*position.L increase or decrease [passively]?

- `fGI` will change with `fGG` and `tick.fGO` [`tick.feeGrowthOut`] (see eqn. (6))
  - `fGG`
  - `fGO` grows via swapping via ticks.cross. $\longrightarrow$ PASSIVE value

- `position.fGIL`: in `pool::_update(...)`, if $\Delta L \neq 0$, in the call `ticks.update(...)`, at the very end after `position.tokensOwed` has been calculated, we have `position.fGIL = fGI` where `fGI = ticks.getfGIn(...)`. Hence, `position.fGIL` is altered when adding/ decreasing liquidity[1]

- `position.L`: basically the same as above (just the stipulation that $\Delta L \neq 0$ arises slightly differently): when `pool::_update(...)` is called with $\Delta L \neq 0$ (so when adding or decreasing liquidity) we have `position.L = position.L` $+ \Delta L$

we then can make the classifications:

$$\begin{aligned} fGI, fGG, fGO &\quad\longrightarrow\quad \text{PASSIVE values} \\ position.fGIL &\quad\longrightarrow\quad \text{ACTIVE value} \\ position.L &\quad\longrightarrow\quad \text{ACTIVE value} \end{aligned} \tag{4}$$

[1] we might add 'or minting non-zero liquidity positions' to this, but minting non-zero liquidity positions is not actually allowed.

UniswapV3Pool.sol :: burn(...)

```
    function burn( tL, tU, x)
    external override lock returns (x0, x1) {
        (position, x0Int, x1Int) =_modifyPosition(
                    owner      : msg.sender,
                    tickLower: tL,
                    tickUpper: tU,
                    liquidityDelta: -int256(x).toInt128()  );

        x0 = uint256(-x0Int);
        x1 = uint256(-x1Int);

        if (x0 > 0 || x1 > 0) {
            (position.tokensOwed0, position.tokensOwed1) = (
                position.tokensOwed0 + uint128(x0),
                position.tokensOwed1 + uint128(x1)
            );
        }
```

_modifyPosition

UniswapV3Pool.sol :: _modifyPosition(...)

```
    function _modifyPosition( params) ...noDelegateCall
        returns (position, amount0,  amount1{...
        position = _updatePosition(owner, tL, tU,liquidityDelta,t0);

```

UniswapV3Pool.sol :: _updatePosition(...)

_updagtePosition

```
f _updatePosition( owner, tL, tU, liquidityDelta, t)  {
    position = positions.get(owner, tL, tU);
    if (liquidityDelta != 0) {
     [update ticks/ flip tick bit map]
```

**if $\Delta L \neq 0$ then* `Tick.sol :: update(..)` *is used to update t_L and t_U. Tick bitmaps are also updated, but we won't be concerned with that here.*

$ticks.update(t, t_C, \Delta L, fGG_0, fGG_1, sPLC, t_C, time, up, maxL$

$$
\begin{aligned}
&info = self\\
&lGB = info.lG\\
&lGA = lGB + \Delta L\\
&if\ lGB == 0: \quad \textit{non-zero if position minted -- will 'positions.get' error if no position has been minted?}\\
&\quad ift < t_C\{\\
&\qquad info.fGO_0 = fGG_0\\
&\qquad info.fGO_1 = fGG_1\\
&\qquad info.sPLO = sPLC\\
&\qquad info.tCO = tC\\
&\qquad info.sO = time \quad \textit{time spent on left side of tick0}\\
&\qquad \}\\
&info.init = true\\
&\quad \}\\
&info.lG = lGA\\
&info.lNet = up\\
&\qquad ?info.lNet- = \Delta L\\
&\qquad : info.lNet+ = \Delta L
\end{aligned} \tag{5}
$$

```
    (fGI0, fGI1) =  ticks.getfGIn(tL, tU, t, fGG0,  fGG1);
*fGG SLOADed
```

$ticks.getfGIn(t_L, t_U, t = t_0, fGG)$

**Note: t.fGO = fGG only if t.lG == 0 (so when minting)*

$$
\begin{aligned}
&if\ t_0 \ >= t_L\{\\
&\quad fGB = t_L.fGO\\
&\qquad \}else\{fGB = fGG - t_L.fGO\}\\
&\\
&if\ t_0 \ < t_U\{\\
&\quad fGA = t_U.fGO\\
&\qquad \}else\{fGA = fGG - t_U.fGO\}\\
&\\
&fGI = fGG - fGB - fGA
\end{aligned} \tag{6}
$$

***fGI_scenarios**

$t_0 < t_L$:

$fGI = fGG - [fGG - t_L.fGO] - [t_U.fGO] = \Delta_{LU} \ldots$

$t_L.LG = 0$, $t_U.LG > 0$

$\ldots = fGG - t_U.fGO$

$t_L.LG > 0$, $t_U.LG = 0$

$\ldots = t_L.fGO - fGG$

$t_L.LG = 0, > 0$, $t_U.LG = 0, > 0$

$\ldots = 0, \Delta_{LU}$

$t_L < t_0 < t_U$:

$fGI = fGG - [t_L.fGO] - [t_U.fGO] = fGG - \Delta_{LU} = \ldots.$

$t_L.LG = 0$

$\ldots = t_U.fGO$

$t_L.LG > 0$

$\ldots = fGG - \Delta_{LU}$

$t_U < t_0$:

$fGI = fGG - [t_L.fGO] - [fGG - t_U.fGO] = -\Delta_{LU}$

$t_L.LG = 0$, $t_U.LG > 0$

$\ldots = -(fGG - t_U.fGO)$

$t_L.LG > 0$, $t_U.LG = 0$

$\ldots = -(t_L.fGO - fGG)$

$t_L.LG = 0, > 0$, $t_U.LG = 0, > 0$

$\ldots = 0, -\Delta_{LU}$

**FGG growth

***How does FGG grow in pool?

swap

```
while (state.xSpecRem != 0 && state.sqrtP != sqrtP):
  (state.P, step.xIn, step.xOut, step.feeAmount) =
computeSwapStep(...
```

$$if\ cache.fP > 0:$$

$$\Delta = \frac{step.feeAmount}{cache.fP}$$

$$step.feeAmount\ -= \Delta$$

$$state.protocolFee\ += \Delta$$

$if\ state.L$ >0{

$$state.fGG\ += \frac{step.feeAmount}{Q128} * state.L$$

```
if (state.sqrtP == step.sqrtPNext) {
    if (step.initialized) {
        ...
        if (!cache.cLO) {
```

(cache.tC . cache.sPLC) – observeSingle(...):

$$\Delta t = bT() - last.bT$$

$$cache.tC = t_0 * \Delta t$$

$$cache.sPLC = last.sPLC + \frac{\Delta t << 128}{L < 0 ? L : 1}$$

```
LNet   =  ticks.cross(
            step.tickNext,
            (zF1 ? state.fGG1 : fGG0),
            (zF1 ? fGG1 : state.fGG),
            cache.sPLC,
            cachetC,
            cache.bT );
 if (zF1) liquidityNet = -liquidityNet;
 state.L = L + LNet
```

ticks.cross

$$L_{Net} = t_{Next}.LNet$$

$$t_{Next}.fGO = fGG - t_{Next}.fGO$$

$$t_{Next}.sPLO = sPLC - t_{Next}.tCO$$

$$t_{Next}.tCO = time - t_{Next}..sO$$

** $t_{Next}.fGO$ only updated with tickss.cross – so when swapping*

** state.fGG only updated with step.feeAmount which is calculated with computSwapStep [fGG set to state.fGG after while loop]*

_updatePosition

```
position.update(liquidityDelta, fGIn0, fGIn1);
```

position.update(self, ΔL, fGI)

$$
\begin{aligned}
&if\ \Delta L == 0\{L_{Next} = self.L\\
&\quad \}else\{L_{Next} = self.L + \Delta L\}\\
tokensOwed &= \frac{(fGI - self.fGIL) * self.L}{Q128}\\
self.fGI &= fGI
\end{aligned}
\tag{7}
$$

```
[if DeltaL < 0: clear ticks]
```

end _modifyPosition

* NOTE: _updatePosition just as well could have been called AFTER calculating x_0, x_1???

```
/// @return amount0 the amount of token0 owed to the pool, negative if the pool should pay the recipient
/// @return amount1 the amount of token1 owed to the pool, negative if the pool should pay the recipient
```

$$
\begin{aligned}
&if\ t_0 < t_{Lower}:\\
&\qquad x_0 = \frac{\sqrt{P_U} - \sqrt{P_L}}{\sqrt{P_U}\sqrt{P_L}} * \Delta L = \left(\frac{1}{\sqrt{P_L}} - \frac{1}{\sqrt{P_U}}\right) * \Delta L\\
&if\ t_L < t_0 < t_U:\\
&\qquad x_0 = \frac{\sqrt{P_U} - \sqrt{P_0}}{\sqrt{P_U}\sqrt{P_0}} * \Delta L = \left(\frac{1}{\sqrt{P_0}} - \frac{1}{\sqrt{P_U}}\right) * \Delta L\\
&\qquad x_1 = \left(\sqrt{P_0} - \sqrt{P_L}\right) * \Delta L\\
&\qquad L = L + \Delta L\\
&if\ t_U < t_0:\\
&\qquad x_1 = \left(\sqrt{P_U} - \sqrt{P_L}\right) * \Delta L
\end{aligned}
\tag{8}
$$

```
"current tick is below the passed range; liquidity can only become in range by crossing from left to right, when we'll need more token0 (it's becoming more valuable) so user must provide it"
```

....Why do we care about all this? Consider the `pool::collect` function,

pool :: collect

```
f collect( recipient , tL, tU, xR) external override lock returns ( x0,  x1) {
        position = positions.get(msg.sender , tL, tU);

        x = xR > position_xOwed ? position.xOwed : xR;

        if ( x > 0) {
            position.tokensOwed -= x;
            TransferHelper.safeTransfer(token0 , recipient , x);}
```

so we see it is the calculated value of `position.tokensOwed` which will limit how much we are able to collect. This ultimately is how we make money off of UniswapV3. We could also use this above analsysi to track the pools calculation of the factoryOwners cut via `protocolFees.token0`/ `protocolFees.token1` (see `pool :: collectProtocol`).