Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Replicating Liquidity Provision Strategies in Uniswap

Bachelor's Thesis

Amir Dellali

dellalia@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Robin Fritsch
Prof. Dr. Roger Wattenhofer

June 15, 2022

Acknowledgements

I would like to thank my supervisor, Robin Fritsch for the continuous discussion we had about the topic, which have helped me understand the topic in a level of detail I would not have achieved without this support and feedback. I would also like to thank Prof. Dr. Wattenhofer, and the department in general for being able to work on this topic.

Abstract

By estimated the liquidity space of the 0.03% USDC-ETH-Pool of Uniswap v3, we investigated the conditions under which a liquidity provider replicating the behavior of other providers is profitable, considering various sources of losses, including gas fees and impermanent loss. We found that, in general, this replication behavior can profitable when reducing the amount of adjustments done by the replicating agent, leading to profits when provided with a sufficient degree of starting capital.

Contents

Chapter 1

Introduction

Decentralized exchanges (DEXs) facilitate the swapping of cryptocurrency tokens on the blockchain while foregoing the need for intermediaries. This provides greater autonomy for the user and trustlessness when compared to centralized exchanges (CEXs). By directly interacting with smart contracts, users avoid giving up custody of their tokens. DEXs have become integral to the functioning of DeFi markets, with Uniswap v3 in turn being the largest one in terms of 24-hour volume at the time of writing.

Different mechanisms are used to implement DEXs, the main ones being order books and automated market makers (AMMs). While order books work by matching trades between users directly, usually relying on a peer-to-peer off-chain network, AMMs are a concept unique to DeFi, swapping tokens in accordance to a mathematical relationship specific to the protocol. Constant product market makers (CPMMs) like Uniswap use the following relationship:

$$x \cdot y = k, \tag{1.1}$$

with the token reserves x (of token X) and y (of token Y), and a constant k, which is related to the total liquidity L contained in the pool by $\sqrt{k} = L$ [1]. While Uniswap v1 and v2 are pure CPMMs, the behavior of Uniswap v3 is considerably more complex due to the introduction of the concept of concentrated liquidity [1], which is central to the innovation of this version. Concentrated liquidity gives users the ability to invest their assets within constrained price regions, leading to much higher capital efficiency when compared to preceding versions. This innovation necessitated the discretization of the price space to individual ticks, which act as the start- and endpoints of a position. A tick index i corresponds to the price

$$p(i) = 1.0001^i \tag{1.2}$$

A liquidity position can thus be described as a tuple of lower and upper tick, and the contained liquidity: $P = (t_l, t_u, L)$. The position P is only active (i.e.

only accrues fees) when it envelopes the price $p = \frac{y}{x}$ rounded to the closest valid $p(i)$:

$$active(P, p) = t_l \leq \lfloor \log_{1.0001} p \rceil < t_u \tag{1.3}$$

This property makes Uniswap v3 fundamentally different from ealier version. Instead of a swap being governed by the CPMM formula for the entirety of the tick space, equation 1.1 now only holds within the region between two initialized ticks, changing k and the token reserves x, y when crossing said region, essentially splitting the price range into multiple fragments of different liquidity.

Only ticks with indices that are multiples of some tick spacing, which is determined by the fee structure of the pool, can be initialized, while the price tick can still take on all possible values.

The newfound ability for users to choose position ranges quickly highlighted the question of how proactively one should manage their invested tokens. A user might decide to concentrate their liquidity in a narrow range around the price at the time of investment, representing a larger portion of the in-range liquidity and thus gaining a proportionally higher fraction of swap fees, at the risk of this position not earning any fees if the price moves outside of this narrow range. Should the user now readjust their position bounds, incurring a loss due to gas fees, wait for the price to come back into range, or would it have been more profitable to create a broader position in the first place?

This issue of the degree of active management has led to the development of a multitude of strategies with the aim of providing certain heuristics to the liquidity provider. This has, in a sense, only pushed the decision problem one (figurative) layer up, however. Instead of managing their investment on a per-position basis, liquidity providers now manage various strategies, some of which might provide higher or lower returns under specific market conditions, gas prices, the size of the investment, whether the pool in question swaps between stablecoins, etc. One might argue that the management of strategies has even caused a greater degree of complexity for the average end user.

Given these circumstances, an intriguing question to ask is whether it is possible to forego the use of individual strategies in favor of simply replicating the behavior of other liquidity providers, acting (loosely speaking) as a sort of index fund for active management.

Referring to the examples given above, a sharp increase in gas prices is likely to make various strategies nonviable, requiring manual intervention on behalf of the investor actively managing them, while being reflected by simply replicating market behavior.

Rather than adjusting strategies when providing liquidity for stablecoins, an investor might decide to replicate instead, or might do so as an alternative to

having to actively monitor market conditions or price data. Given the sophistication of liquidity providers on the largest Uniswap v3 pools, one could imagine that their behavior in aggregate leads to a valuable strategy.

Chapter 2

Related Work

2.1 The Question of Active Management

Since the introduction of Uniswap, ways that liquidity providers can choose to actively manage their liquidity, and more generally the recommended degree of activity, have been suggested, even leading to the creation of protocols with the aim of optimizing liquidity allocation. Most of these are concerned with Uniswap v2 however, with a comparatively minor amount of research being done concerning liquidity provision on Uniswap v3. One of the few approaches exploring this problem is presented in [2], which formalizes the relationship between higher liquidity concentration, and correspondingly higher fee revenue, and the risk of price movements making the position inactive.

2.2 On Market Efficiency

An underlying assumption of replicating other liquidity providers is that, as a whole, one expects them to exhibit some degree of efficiency in terms of liquidity provision. Replicating the behavior of irrational actors makes little sense, so this question is central to the idea of replicating other LPs. Most focus concerning Uniswap and market efficiency is focused on pools, and on the existence of arbitrage opportunities between different liquidity pools and protocols, not on whether liquidity providers active their liquidity in an optimal manner. However, there is enough research to indicate that a high amount of liquidity providers (at least in larger pools) are using sophisticated strategies [3], leading to the erosion of risk-free returns (which would indicate obvious inefficiencies). Generally speaking, given the difficulty of of finding profitable strategies, this work assumes that, at least when considering the largest liquidity pools available on Uniswap, the strategies of other actors have achieved a level of sophistication sensible of replication.

Chapter 3

Data Collection & Exploration

The Uniswap v3 subgraph [4] was used as the primary data source for block-level position adjustments and swaps. Changes in a pool's liquidity are represented in the GraphQL schem as `PositionSnapshot` entities, containing (among other data) the new liquidity of the position if it underwent an adjustment at that block. If a liquidity position undergoes multiple changes in a single block, the `PositionSnapshot` represents the final state at the end of it.

Swaps are represented in the `Swap` entity, which contains the input and output amount, as well as the price tick after the swap occurred. As it is relatively common for multiple swaps to occur in a single block, they are ordered by their `logIndex` within the array of transactions.

Each `PositionSnapshot` and `Swap` belongs to a `Transaction` entity which contains the `gasPrice` field. We calculate the mean gas price over all transactions contained in a block to retrieve historical gas prices with block-level granularity.

This work investigates the USDC-WETH 0.03% pool specifically.

Chapter 4

Experimental Setup

Position replications and swap fees are computed separately in two different models. Due to high computational workload involved, calculations were done on a 5-node TPU cluster hosted on Google Compute Engine, heavily utilizing parallel processing for most simulations.

4.1 Position Replication

Position replications utilize the JAX library [5], which is a wrapper around the popular Numpy [6] library but providing the additional functionality of just-in-time compilation using Tensorflow's XLA compiler, and automatic differentiation.

The liquidity of each tick in the pool at a certain block number is incrementally built up by applying the changes specified by each position snapshot up to that point. An internal representation of each position is maintained, against which every snapshot is compared in order to obtain a delta between the new and old liquidity of that position. This liquidity delta is then applied to the pool state, resulting in the new liquidity of each tick.

This real pool state is approximated given a certain set of restrictions and parameters, which depend on the specific type of approximation used, as explained in 5.

Due to the Uniswap protocol's extensive usage of high-precision numbers and JAX's restriction of only supporting 32-bit floats when using the TPU backend, all liquidity and token amounts are internally multiplied with a precision reduction factor of 10^{-4} to avoid overflowing the float32 type. This does not lead to a significant reduction in precision of the output, which was tested empirically.

The replicated positions are then written to a JSON file, following the `PositionSnapshot` specification to facilitate seamless comparison during when calculating swap fees.

4.2 Swap Fee Calculation

Swap fees are calculated using the Uniswap Typescript SDK, which implements a range of functionalities, including (most notably for our purposes) functions for liquidity calculation and the `Pool` class, specifically its `swap` method. By calling the `computeSwapStep` function internally, this method considers the changes in liquidity incurred when swapping ticks. While a swap fee calculation using the spot price (which does not consider tick crossings) is likely accurate enough for most applications, our use-case necessitates this function because we are specifically concerned with the fraction of the total liquidity our replicated position represent at tick.

Following a similar procedure as during position replication (4.1), the pool's entire liquidity at each block is adjusted according to the changes specified by the block's position snapshots. Each swap is then executed against the pool (in order of `logIndex`), yielding a new price tick that the next swap will start from. The total fee generated is split up among all the liquidity providers at each initialized tick, in accordance with the fraction of the total liquidity they provide. After the swaps have been executed, the pool state is updated with the positions produced in the replication step. This order of operations is necessary to ensure temporal consistency, as we should only be able to replicate a block once it has appeared on the blockchain. The replicated positions thus only affect the subsequent blocks, avoiding data leakage.

Chapter 5

Methods

5.1 Histogram Approximation

5.1.1 Limited Token Amounts

The real token amounts contained in a position P_i denoted by lower price $p_{l,i}$, upper price $p_{u,i}$, and contained liquidity L_i, depend on the pool's current price p and are given as follows:

$p \leq p_{l,i}$:

$$x_i = L\frac{\sqrt{p_{u,i}} - \sqrt{p_{l,i}}}{p_{l,i} \cdot p_{u,i}} \qquad y_i = 0$$

$p \geq p_{u,i}$:

$$x_i = 0 \qquad y_i = L(\sqrt{p_{u,i}} - \sqrt{p_{l,i}})$$

$p_{l,i} < p < p_{l,i}$:

$$x_i = L\frac{\sqrt{p_{u,i}} - \sqrt{p}}{\sqrt{p} \cdot p_{u,i}} \qquad y_i = L(\sqrt{p} - \sqrt{p_{l,i}})$$

Thus, the total token amounts of all positions at state S can be written in vector form as:

$$\boldsymbol{T}_S = \begin{bmatrix} x_{tot} \\ y_{tot} \end{bmatrix} = \sum_{P_i \in \{P_1, P_2, \ldots, P_n\}} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{5.1}$$

In order to directly compare the amount of tokens in the replicator's positions to the amount contained in the pool, both are converted to units of token x by taking the dot product of the token vector and the price:

$$x_S = \boldsymbol{T}_S{}^{\intercal} \cdot \begin{bmatrix} 1 \\ \frac{1}{p} \end{bmatrix} \tag{5.2}$$

This is done for each position individually, and for the total pool state, producing the share of tokens (and liquidity) represented by each position. The goal of each replication is to replicate this share of liquidity, under consideration of a few cost minimization strategies.

Cost minimization strategies

Replicating every tick exactly would be highly inefficient, which is why a number of cost minimization strategies were implemented.

1. Create approximations using the median of all values within a given `bin_width`. In particular JAX's `nanmedian` function was chosen, which ignores NaN values. This is necessary because some values will be set to NaN when reducing the tick space around the price, as outlined in item 4.

2. Only replicate the pool every nth block. The average Ethereum block time is between 12 and 14 seconds, which is much lower than the frequency with which the average liquidity provider (even those conducting active management) adjusts their positions. By only replicating every `nth_block`, a replicator can reduce the number of adjument operations considerably (mints/burns), while retaining a high degree of accuracy in terms of replication.

3. Only change the pool state once it diverges enough from what its optimal replication would be. By considering the divergence from the optimal replication (defined here as a percentage of the real liquidity), one can avoid the problem of adjusting ones positions if the real state only diverges by a small amount. The divergence is calculated by considering a `max_err_factor`, which represents the percentage that the summed absolute value of the liquidity difference at each tick (between the approximation and the real tick state) can exceed before changing.

4. Limit replication to a restricted region around the price tick. Given the assumption that the price tick is unlikely to move more than some amount in the span of n blocks, one should be able to achieve the similar rates of of return while simultaneously reducing the number of position adjustments by limiting the replicated region to within a range around the price tick. Due to the changing nature of the price tick however, this might instead lead to an increase in the number of mints/burns. This tradeoff is explored later by adjusting the `price_range` parameter of the model.

Simulations and Interpretation

The pool's liquidity state was replicated over its entire history, starting from block number 12370624 (May 4, 2021) and ending at 1654197822 (June 2, 2022), corresponding to a total number of x data points (position snapshots or swaps) in that time span. A sweep of the parameters specified above was conducted, totaling 17380 combinations.

The result of every strategy is comprised of 6 values that determine the performance of each:

- `feesTokenX`: The resulting fee amount comprised of token X.
- `feesTokenY`: The resulting fee amount comprised of token Y.
- `totalMints`: The number of times a position was minted (before this change, the position contained 0 liquidity).
- `totalBurns`: The number of times a position was burned (after this change, the position contained 0 liquidity).
- `totalAdds`: The number of times liquidity was added to an existing position.
- `totalRemoves`: The number of times liquidity was removed from an existing position, without burning it (i.e. reaching 0 liquidity).

While adding and removing liquidity are technically done by the same mechanism as minting and burning (the Uniswap protocol does not differentiate between these operations), they were considered separately in this analysis due to a difference in gas costs when minting/burning liquidity for a new position vs. doing so for an existing one.

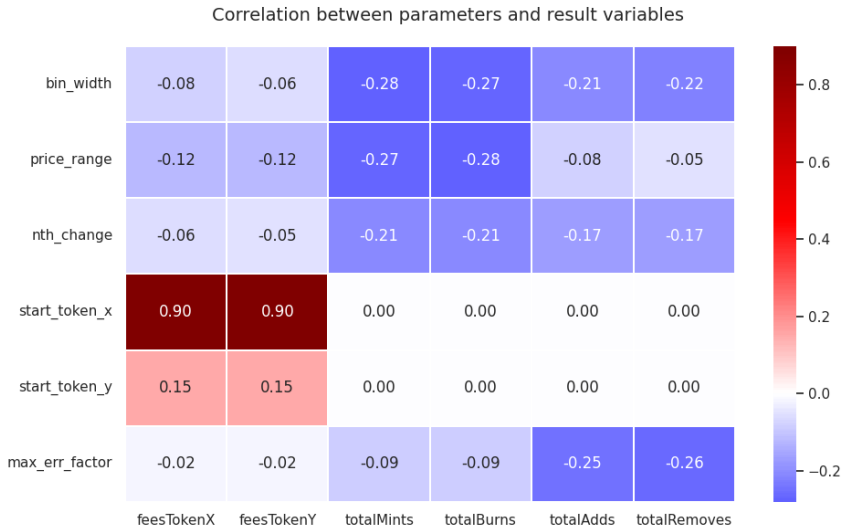


Figure 5.1: Correlation between parameters and results.

The correlation between simulation parameters and results is shown in figure 5.1. As one would expect, there is a strong negative correlation between `bin_width` and the number of position adjustment operations. This relationship is stronger towards mints and burns than to additions and removals of liquidity, which can be explained by considering that once the entire tick range is filled with positions, further changes in the liquidity do not cause more mints or burns (once a position has been initialized, it rarely reaches 0 liquidity again if the considered price range is large enough). The total revenue as measured by `feesTokenX` and `feesTokenY` is reduced by a smaller degree, which can be attributed to the higher degree of liquidity dilution and reduced replication accuracy when compared to a strategy with a smaller `bin_width`.

A similar pattern can be observed for the `price_range` parameter. While an increase will dilute the invested liquidity, decreasing fee revenues, the number of mints and burns are simultaneously reduced when compared to a smaller value. A small `price_range` leads to ticks on the edges of the considered interval coming in and out of range repeatedly, increasing the amount of mints/burns in favor of what would have been adds/removes for a larger range.

Only considering changes at every `nth_change` multiple of the block number leads to an expected decrease in the number of operations, also affecting the fee revenue negatively.

The amounts of initially invested token amounts `start_token_x` and `start_token_y` have no impact on the number of operations, while exhibiting a direct increase in revenue.

`max_err_factor` affects the revenue only marginally, while decreasing the

number of additions and removals considerably. Mints and burns are reduced to a smaller degree, which can again be explained by the considered tick range usually filling up completely (i.e. being entirely covered by replicated positions). After that point, every change in liquidity will be reflected by additions and removals.

In order to compare different experiments directly, a strategy's fee revenue in terms of token X (USDC in this case) will be used as a stand-in for the total value of the fees in both tokens combined, in order to simplify formulas and visualizations. This can be justified by the high degree of correlation between the the two as seen in figure 5.2.

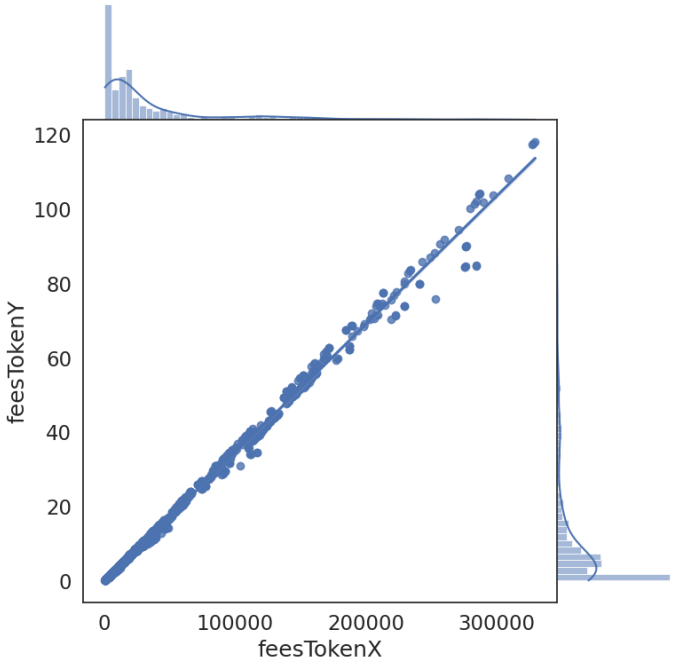
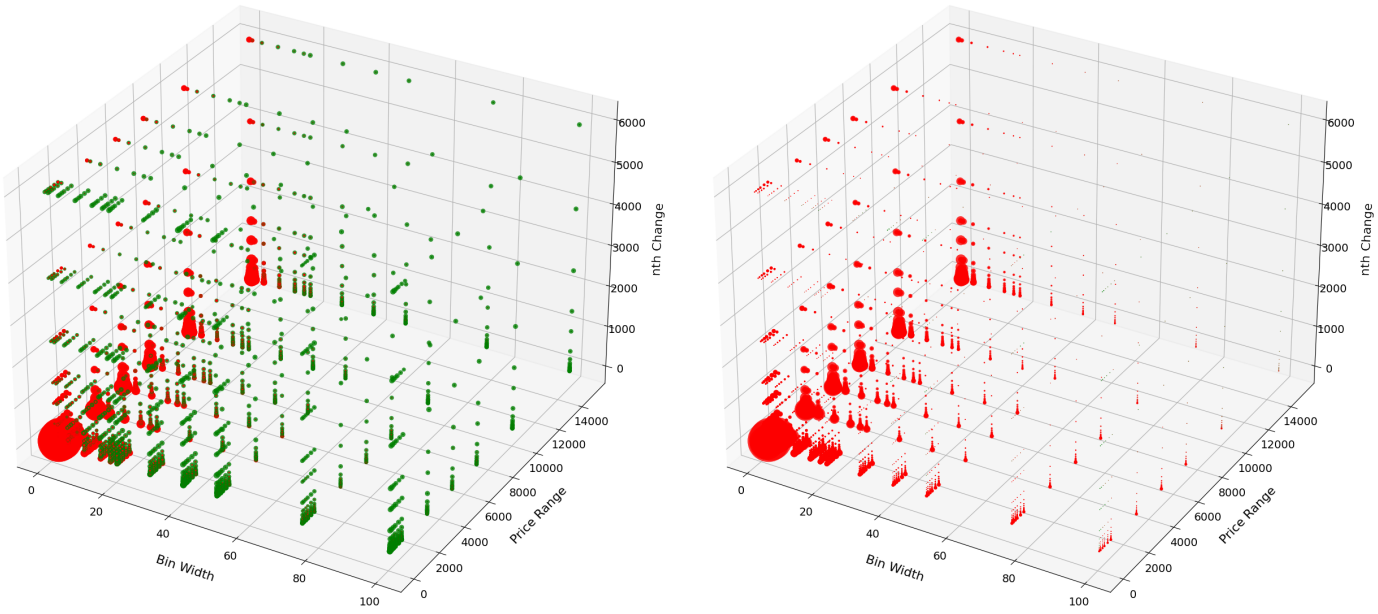


Figure 5.2: Joint plot between token denominations of fee revenue.

Motivated by the increased revenue observed for smaller price ranges, one might be interested in the point at which the loss due to position adjustments, including mints, burns, additions removals, and swaps, exceeds the gain in fees.

Cost Sources and Breakeven Points

The point at which these various strategies reach profitability was established by first collating historical block-level gas prices on the Ethereum mainnet with the adjustment operations conducted at that block, and second, by considering



(a) All investments.

(b) Investments with values of less than $20000.

Figure 5.3: Profitability of various parameter combinations. Green points are profitable, while red ones are not, with sizes indicating profit relative to the highest absolute value.

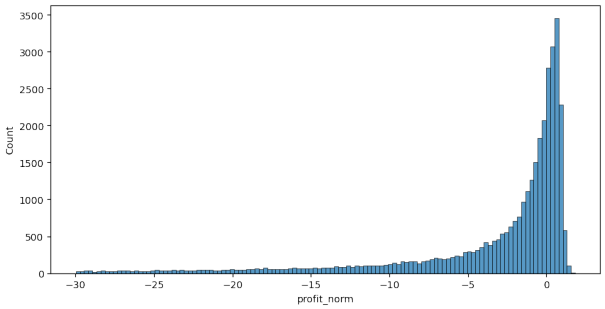
various theoretical gas prices, taken as an average across the pool's history. Intuitively, there should be a tradeoff between the losses due to increased gas fees from higher degrees of active management on one hand, and the associated rise in returns from better replication or concentration of liquidity on the other.

Another critical aspect to consider besides gas fees however, is the degree of impermanent loss [7] an LP will incur over time for a given strategy. One might suspect that this will be one of, if not the biggest source of losses for replicating strategies as presented here, given that every position adjustment is accompanied by a swap (except in the case of the price being out or range of all positions), meaning that losses are actually realized, as opposed to remaining unrealized as would have been the case had they not been withdrawn. For most liquidity providers, impermanent loss already exceeds fee revenues [8].

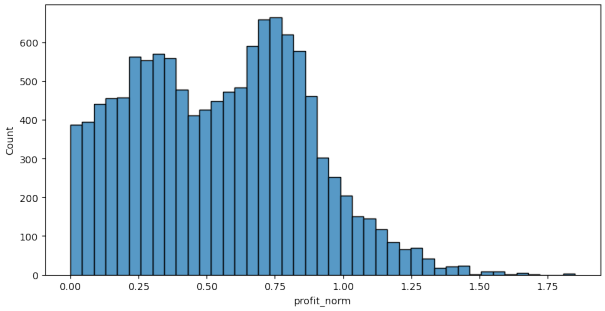
The overall profit is calculated as:

$$\texttt{profit}_X = \texttt{fee\_revenue}_X - \texttt{op\_cost}_X - \texttt{imp\_loss}_X \tag{5.3}$$

The total profit in USDC over a range of parameters is shown in figure 5.3a.



(a) Normalized profits. Note that there are smaller values than -30 (the minimum is -3413.02), which are omitted for visualization purposes.



(b) Positive normalized profits.

Figure 5.4: Histogram of normalized profits. (b) shows the positive section of (a) in more detail.

What is obvious from this figure is that less active strategies are much more likely to be profitable, when compared to highly active ones. There is an especially high influence of `bin_width`, with small values of this parameter accruing the highest deficits, and largely preventing profitability, except for very high values of `nth_change`. The points in this plot include all investment sizes, while figure 5.3b shows that the investment amount has considerable impact on a strategies profitability, with those starting with a value of less than $20000 almost never being profitable, regardless of parameter choices.

Given this high degree of dependency of profitability on investment size, this relation was next explored by normalizing total profits with respect to the investment value required to achieve them, calculating normalized profits in the following manner:

$$\texttt{profit\_norm} = \frac{\texttt{profit}_X}{\texttt{investment\_tvl}_X} \tag{5.4}$$

As can be seen in figure 5.4a, the normalized profits form a long-tailed distribution, with 32.1% having a positive normalized profit. Normalized profits

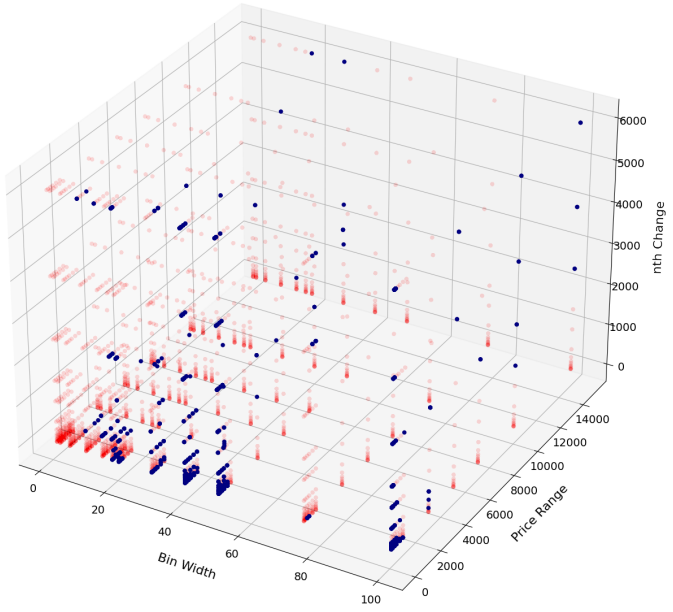


Figure 5.5: Scatter plot of strategies in parameter space, highlighting those with a normalized profit greater than 1 in blue.

represent return on investment, meaning that strategies with a high normalized profit are of particular interest. Figure 5.5 highlights the parameter distribution of strategies with a normalized profit greater than 1, showing a much higher density at bin widths larger than 20 and, most importantly, small price ranges. This is due to the high concentration of liquidity around the price occurring for these values, giving the investor a larger share of fee revenues. It stands to reason that this should only hold if the price shows relatively little movement between adjustments. This can either be due to the nature of the pool (tokens with a particularly volatile nature), or due to our own strategy (choosing `nth_change` such that the price moves a lot between adjustments - even if the tokens themselves are not highly volatile). The latter case is seen in this figure, which shows that the density of viable strategies (those highlighted in blue) is much higher for low values of `nth_change`, dropping of considerably above 1000.

This relationship between `price_range` and the normalized profit only occurs at high values (defined here as being greater than 0), reversing for negative normalized profits, as shown in figure 5.6.

While this shows that profitability is possible for a given set of parameters, this strategy can be improved:

One simple (in terms of formulation) improvement over changing all replicated

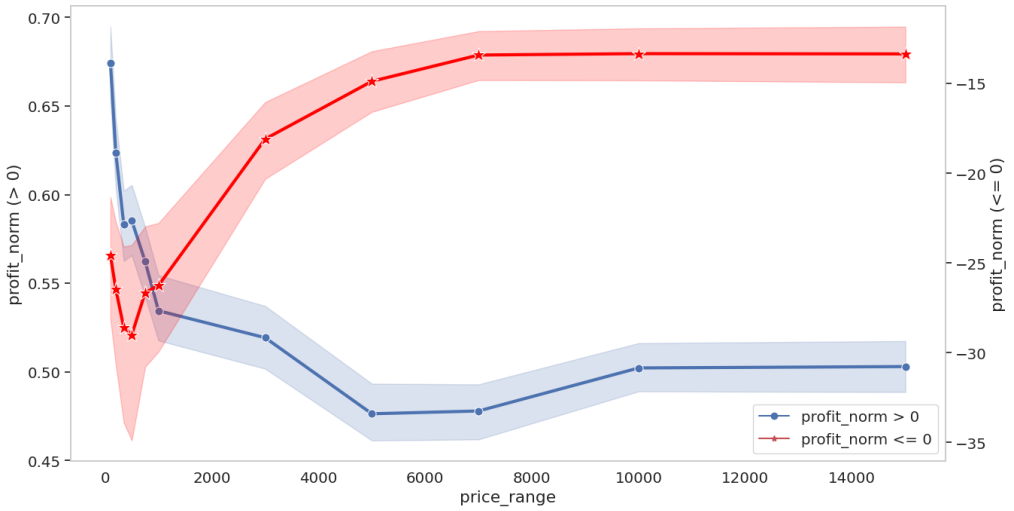


Figure 5.6: Behavior of normalized profit for varying values of `price_range`, differentiated by whether `norm_profit` is positive. Note the different scales

positions when a certain threshold is reached, is to only adjust those positions that are affected most by changes. With the algorithm given above, even the adjustment of a single bin leads to liquidity being withdrawn from all other positions, due to 100% of the available liquidity being invested at all times. This effect can however be mitigated somewhat by increasing the error threshold.

In order to perform selective tick updates, it is necessary for the liquidity provider to reserve a set amount of tokens used for covering liquidity deltas not covered through swaps. A modification of the binning algorithm described above was used to explore this scenario.

5.1.2 Reserved Liquidity and Unlimited Token Amounts

The general goal of this algorithm is to replicate a constant percentage of the in-range liquidity, ignoring token amounts. By working in units of liquidity rather than concrete token denominations, the complexity and cost associated with swapping is avoided. One can imagine this strategy as representing an investor with unlimited reserves, whose aim it is to minimize gas fees (by minimizing the number of position adjustments and swaps), at the expense of not gaining fees from these amounts. While unrealistic, considering this scenario can provide meaningful information about the independent contribution of parameters, and most relevant for this section, on the reserve size needed to minimize operational costs.

Implementation Details

Positions are adjusted in such a manner so as to keep the fraction of the unlimited approximation replicated at a constant value. Note that this differs from the approach given above in the critical aspect that the total invested amount can increase. Informally, this method seeks to answer the question: "*If I wanted to represent p% of the unlimited approximation at every block, what size reserve would I need?*", while the above asks: *"Given a set amount of tokens, how should I distribute them such that all positions represent the same q% of their unlimited approximation?"*. While q can change across blocks (and almost always does), p is constant, meaning, that as the TVL (total value locked) in a pool increases (decreases) over time, the amount invested by the LP will follow suit, increasing or decreasing in lockstep.

Apart from changing liquidity amounts, this implementation is mostly identical to the prior one. One problem that becomes apparent when using the same `nanmedian` approach as described in the limited token scenario, is that, due to not having an upper limit on the amount of invested liquidity, a decrease in the price range (or increase in bin width) will almost always lead to a much higher fee return. A replication that constrains the tick space tightly around the price will set all other ticks to N/A, excluding them from providing any information to the median calculation. This means that a high amount of liquidity around the price (which is usually the case) will be propagated across the entire bin, producing an unrealistically large amount of liquidity to replicate. An alternative is to set out-of-range ticks to 0. The disadvantage of this method is its tendency to lead to many 0-liquidity estimations (specifically, when $2 \cdot \texttt{price\_delta} < 0.5 \cdot \texttt{bin\_width} \cdot \texttt{tick\_spacing}$), preventing us from gaining information about a large section of the parameter space. Instead, the approach taken here is to retain the `nanmedian` method, but normalizing the simulated profit by the reserve size needed to achieve it (both converted to a common token, USDC in this case)[1]:

$$\texttt{profit\_norm} = \frac{\texttt{fee\_revenue}_X - \texttt{op\_cost}_X}{\texttt{reserve}_X} \tag{5.5}$$

This is argued to be a sensible approach, given that the fee revenue per position and swap is directly proportional to the fraction of total liquidity the position represents in the crossed ticks.

[1] Note that the X subscript refers to the token type the variable is converted into (either X or Y in the X/Y pool). Before conversion, `fee_revenue` and `reserve` consist of both token types, while `op_cost` is in ETH (specifically Gwei), at least on the Ethereum blockchain.

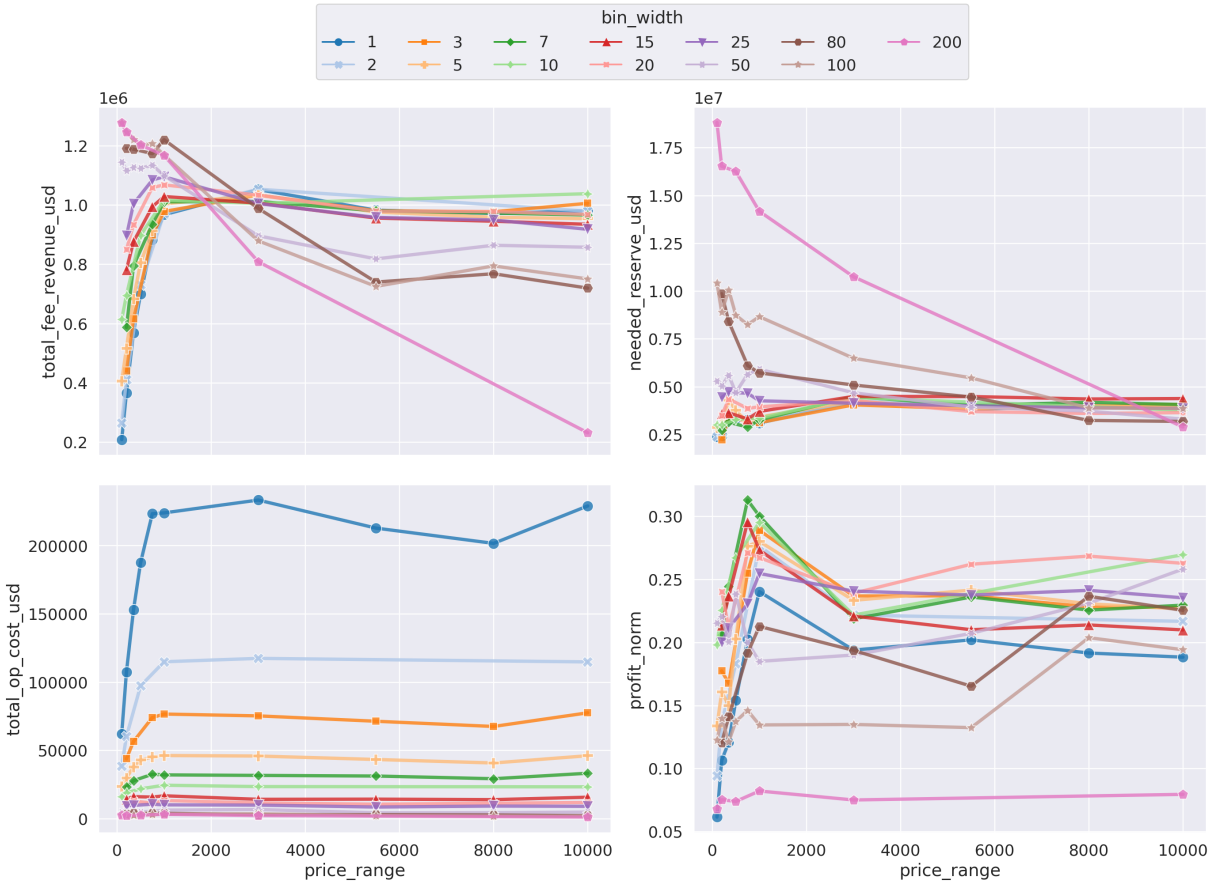


Figure 5.7: Results of replicating 1% of liquidity, $\delta_{max} = 0.5$, $n_{adj} = 1000$. (Top left) Total fee revenue in USD. (Top right) Amount of reserve utilized in USD. Note the order of magnitude difference to fee revenues. (Bottom left) Sum gas fees for position adjustments (mints, burns, adds, removes) in USD. (Bottom right) Normalized total profit as seen in equation 5.5.

.

Simulated Reserve Sizes and Profits

In contrast to Uniswap v2, the token ratios present in each pool do not always follow the price, meaning that the distribution of tokens within a reserve will may not always contain the specific token amounts needed to replicate every new change. This discrepancy is increased even further, when only considering a selected range around the current price. Figure 5.8 shows the development of the token ratio against that of the price, using a `price_range` of 500. While generally following price movements, the tokens ratios of the positions surrounding the price display a high degree of variance. Given this, the choice was made to distribute the tokens in the replicator's reserves according to the initial price, with the total

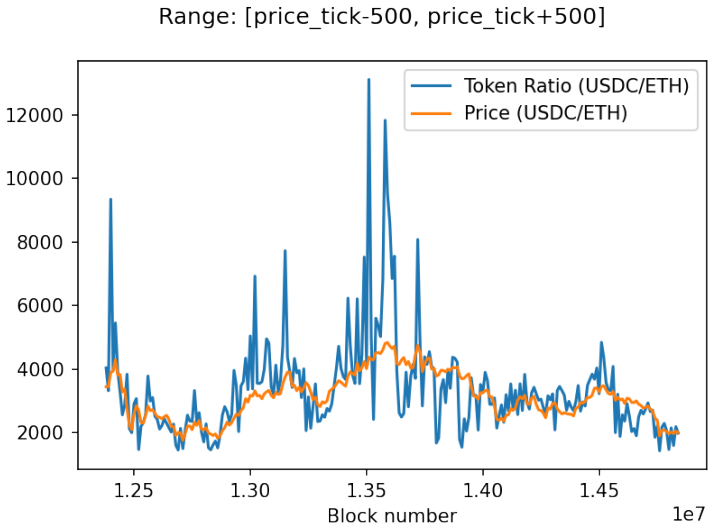


Figure 5.8: Ratio of tokens vs. price over block numbers.

.

value of the reserve chosen to satisfy a given replication percentage.

Figure 5.7 shows the output of a simulation replicating 1% of the liquidity within its range. Price range and bin width were varied while keeping other parameters at the values `price_range` = 0.5, `nth_change` = 1000. One can see that fee revenues increase for a reduction in price range (for the reason mentioned in 5.1.2) up until a point, where strategies with a bin width smaller than 50 (not necessarily the least upper bound) peak, before sharply declining, while the rest continue rising. This indicates that there is a tradeoff between the overapproximation of liquidity (due to 5.1.2), which dominates for positions with a large bin width, and the replicative power, which is restricted by a smaller price range (especially in combination with a small bin width). In these regimes, the issue with median estimation is less relevant due to small bin widths. This interaction might seem counter-intuitive, especially given the pattern observed in the limited-liquidity scenario, but makes more sense when one considers that this method (in contrast limited liquidity) does not concentrate liquidity in smaller price ranges, but replicates the same percentage, just as it would have done with a larger price range.

5.2 Selective Position Adjustment

One obvious inefficiency of the approach outlined so far is the lack of selective position adjustments. Once the maximum error threshold is reached, every bin is updated, even if the actual change experienced by this position is minimal.

The total number of position adjustments thus has an upper bound of $\texttt{n\_bins} * \frac{\texttt{n\_blocks}}{\texttt{nth\_change}}$, representing the main source of costs.

One can reformulate the approach outlined above, to only adjust the positions most affected by change. By reconsidering `max_err_factor` as being per position (instead of per state), only those positions whose absolute value has changed by more than the given fraction will be adjusted.

The reserve sizes calculated in 5.1.2 represent the maximum reserve amount necessary to replicate a given $p\%$ of the in-range liquidity at every point in time. While required to achieve this percentage at every block, a more realistic scenario would most likely not choose the maximum reserve sizes, in favor of simply not replicating those outliers. By choosing the maximum theoretical reserve, a replicating liquidity provider would need to either leave a large amount of tokens uninvested (to avoid changing the chosen percentage value), or invest those tokens, changing p and introducing a higher amount swapping overall.

Because of this consideration, reserve sizes were recalculated for a given set of percentages to replicate, choosing not the maximum token amount at each adjustment (as done in section 5.1.2), but such that 80 % of position adjustments can be done without swapping. On average, this leads to a reduction in reserve sizes of 30 %, which is consistent across different parameter configurations.

While it was unfortunately not possible to investigate this approach for a parameter set as large as the one used for the approximation method outlined above, mostly due to computational constraints, it was found that choosing the reserve in such a manner reduces the amount of liquidity additions and removals only when considering very small values of `max_error_factor` and `nth_change`, which is sensible, considering that these parameters were chosen with the idea of reducing adjustment operations in the first place. Overall, the range of parameters for which a selective position adjustments make sense, only includes strategies that are subject to high losses to begin with, making the reduction in gas fees negligible.

Chapter 6

Conclusion

By designing and optimizing a model to approximate the liquidity space of a Uniswap v3 pool, the profitability and overall behavior of replicating other liquidity providers was explored. It was shown that replicating the liquidity space in a very detailed fashion was generally unprofitable, due to the high number of mints and burns required to achieve said replication. By approximating the tick space to a coarser degree, however, strategies with net profit could be determined. In conclusion, a replicating agent requires a certain amount of minimal capital for all strategies, but given this minimum amount, there is a clear tendency for slower moving strategies (with fewer position adjustments) leading to higher profits.

Bibliography

[1] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, and D. Robinson, "Uniswap v3 core," 2021.

[2] M. Neuder, R. Rao, D. J. Moroz, and D. C. Parkes, "Strategic liquidity provision in uniswap v3," 2021. [Online]. Available: https://arxiv.org/abs/2106.12033

[3] L. Heimbach, E. Schertenleib, and R. Wattenhofer, "Risks and returns of uniswap v3 liquidity providers," 2022. [Online]. Available: https://arxiv.org/abs/2205.08904

[4] TheGraph. (2021) Uniswap v3 mainnet subgraph. [Online]. Available: https://thegraph.com/hosted-service/subgraph/uniswap/uniswap-v3

[5] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[7] H. J. Kim, S. Choi, Y. T. Yoon, and S. Yoo, "Impermanent Loss and Gain of Automated Market Maker Smart Contracts," 2 2022. [Online]. Available: https://www.techrxiv.org/articles/preprint/Impermanent_Loss_and_Gain_of_Automated_Market_Maker_Smart_Contracts/19196960

[8] S. Loesch, N. Hindman, M. B. Richardson, and N. Welch, "Impermanent loss in uniswap v3," 2021. [Online]. Available: https://arxiv.org/abs/2111.09192