Analysis of Trading Functions in Automated Market Making

by

A.R. Husain Cornelissen

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Friday July 7, 2023 at 14:00

Student number:	5247985	
Project duration:	April 20, 2023 - July 7, 2023	
Thesis committee:	Dr. N. Parolya,	TU Delft, supervisor
	Dr. R. J. Fokkink,	TU Delft

Preface

This thesis was written under the supervision of Dr. N. Parolya, on behalf of the statistics department at the EEMCS faculty, TU Delft.

During the covid pandemic, I became interested in blockchain development, and specifically automated market makers. The topic of this thesis originated from practical issues I had trying to find an explicit expression for the Uniswap V3 trading function in order to use it in an optimization. The problem fascinated me so much, that I decided that I wanted to write my thesis about it.

I would like to express my special gratitude to Dr. N. Parolya, for his willingness and effort to guide me through this topic that is outside of his expertise. His broad range of knowledge and patience has been of indispensable value to me.

I would also like to thank Dr. R.J Fokkink for taking place in my graduation committee.

A.R. Husain Cornelissen
Delft, June 2023

Abstract

Automated Market Makers (AMMs) are a novel type of market makers that eliminate the need for a counterparty in a trade. This thesis analyses the properties of several types of AMMs, and in particular the concentrated liquidity market maker. An axiomatic definition of AMMs is provided, and two types of constant function market makers (CFMMs), called the constant sum market maker (CSMM) and constant product market maker (CPMM), are explored. The concentrated liquidity market maker, which improves liquidity provision compared to the CPMM, is thoroughly analyzed, and it is shown that it's trading function can be formulated as a composition of functions. This thesis also conjectures that the concentrated liquidity trading function can be approximated by taking an infinite number of compositions. Additionally, a simulation study is conducted using transaction mocking. The simulation study supports the conjecture, and brings several other noteworthy properties of the concentrated liquidity market maker to light.

Contents

1 Introduction

Markets have been around for millennia. Originally a gathering place for farmers to sell their goods to the inhabitants of a town, markets have evolved massively over time. With the rise of financial markets, traded goods have evolved into financial derivatives and securities, and the physical gathering place has been replaced by extremely low-latency digital markets. These developments have made financial markets incredibly efficient, and the goods traded on them increasingly liquid. However, an element that hasn't changed over time is the need for a counterparty in every trade. The rise of blockchain technology has facilitated the creation of decentralised financial markets. These financial markets allow users to trade assets in a permissionless manner, without the existence of a central authority. An innovation that has emerged with decentralised finance is the concept of Automated Market Makers (AMMs). AMMs have transformed the way trading occurs by eliminating the traditional need for a counterparty in a trade. Furthermore, previous research [9] has demonstrated that the AMM outperforms markets utilizing limit order books in terms of market quality.

At the core of an AMM is an algorithm that determines the prices of assets based on their relative supply and demand, called the trading function. Unlike tradition order book exchanges, where buyers and sellers must match their orders to execute trades, AMMs utilize pools of funds that traders interact with. These pools contain pairs of assets, and users can swap one asset for another by trading against the pool.

Although trading functions must satisfy certain market requirements for the market to function properly, this can be accomplished in many different forms. Previous works have already compared different AMMs [6],[7]. It is generally agreed upon that the most common form of AMM, called the constant product market maker(CFMM), possesses many favourable economic and analytic properties. However, more recently a new type of AMM, allowing liquidity providers to place range orders has emerged. This type of AMM, called the concentrated liquidity AMM, allows liquidity to be used more efficiently than in a CFMM. This type of AMM is most notably implemented in Uniswap V3 [23], and will from here on be mentioned by that name

In this thesis, the trading function of Uniswap V3 is analysed and compared to other types of AMMs. Firstly, an axiomatic definition for AMMs is constructed in Chapter 2, which generalises previously given definitions for AMMs to include Uniswap V3. Then, in Chapter 3, the trading function of several types of market makers are introduced and analysed. The mechanics behind concentrated liquidity will be explained and an expression for the trading function will be derived. Subsequently, in Chapter 4, a simulation study into the properties of the Uniswap V3 trading function will be performed. Finally, the research will be concluded and discussed in Chapters 5 and 6.

2

Axioms for Automated Market Makers

In this chapter, an axiomatic definition for an automated market maker is constructed. This is done by generalising the works of Bartoletti et al. [6] and Bichuch and Feinstein [7] to incorporate more general types of liquidity pools, characterised by a liquidity vector $\vec{l}$. Furthermore, both the transitions between states of the blockchain, and the properties of the input-output relation are axiomatised within the same framework.

Firstly, two types of tokens are distinguished. An atomic token represents a single crypto asset that can be traded, e.g. Ethereum or Bitcoin. On the other hand, a minted token is an unordered pair of atomic tokens, i.e. (t_0, t_1) for t_0, t_1 tokens of the atomic type. As will be seen further on in this section, converting atomic tokens to minted tokens is one of the operations defined on an AMM, which is a process known as minting [2].

We define an automated market maker (AMM) by it's trading function:

Definition 2.1 (Trading function)**.** *A trading function is a function*

$$G : \mathbb{R}_+^n \to \mathbb{R}_+ \tag{2.1}$$

that determines the output of a trade from an input vector $\vec{v} \in \mathbb{R}^n$.

Remark. Where possible, the trading function will be denoted as $G(x)$, making the reserves implicit

In order to reduce redundancy, only one direction of trading is considered, i.e swapping units of token t_0 for token t_1. However, trading units of t_1 for t_0 is a symmetric operation with a symmetric trading function.

Note that for AMMs with only two reserves, the trading function is defined as

$$G : \begin{bmatrix} x \\ r \\ s \end{bmatrix} \in \mathbb{R}_+^3 \to \mathbb{R}_+ \tag{2.2}$$

Where r, s are the reserves of tokens t_0 and t_1 respectively, and x is the amount of token t_0 inputted. However, there are some notable exceptions to this, which will be discussed in detail in Section 3.2

Definition 2.2 (liquidity pool). *A liquidity pool is a set* $\{\vec{l}, G(x)\}$, *containing a vector* $\vec{l} \in \mathbb{R}^{n-1}$, *together with a trading function*

$$G(x) : \begin{bmatrix} x \\ \vec{l} \end{bmatrix} \in \mathbb{R}^n_+ \to \mathbb{R}_+$$

Note that a liquidity pool can be seen as an individual realization of an AMM. By using these definitions for AMMs and liquidity pools, the construction that defines the relation between input and output is separated from the definition of individual realizations of AMMs.

For the sake of completeness, the interaction between users and liquidity pools is also modelled. Therefore, the notion of a wallet is introduced, which tracks the assets and the corresponding quantities a user possesses:

Definition 2.3 (Wallet). *Let* T *be a set of tokens, both atomic and minted. Let* U *be a set of unique identifiers, also known as users. Then for every* $u \in U$, *the Wallet of* u *on the token space* T *is defined as a mapping*

$$W_u : T \to \mathbb{R}_{\geq 0} \tag{2.3}$$

As shown by Bartoletti et al. [6], the interactions between users and liquidity pools can be modelled as a so-called Labelled Transition System [15].

Definition 2.4 (Labelled Transition System). *A labeled transition system is a tuple* (S, L, T, p_0), *where*

- S *is a set of states*
- L *is a set of labels*
- R *is a transition relation, where* $R \subseteq S \times L \times S$
- p_0 *is a initial state*

In our case, S represents a finite set containing wallets and liquidity pools:

$$\{W_{u_1}, W_{u_2}, ..., W_{u_n}, \{\vec{l_1}, G_1(x)\}, \{\vec{l_2}, G_2(x)\}, ..., \{\vec{l_n}, G_n(x)\}\} \tag{2.4}$$

Furthermore, L represent the transactions on the blockchain and p_0 is the state where there are no liquidity pools in the system and all wallets hold only tokens of the atomic type.

There are three types of transition relations, which are all triggered by transactions:

1. Deposit: A user can deposit units of atomic tokens t_0 and t_1 to a liquidity pool $\{\vec{l_i}, G_i(x)\}$ in order to receive some units of the minted token (t_0, t_1).
2. Redeem: A user can redeem units of the minted token (t_0, t_1) in order to receive some amounts of the atomic tokens t_0 and t_1 from the liquidity pool $\{\vec{l_i}, G_i(x)\}$.
3. Swap: A user can deposit x units of the token t_0 to the liquidity pool $\{\vec{l_i}, G_i(x)\}$ in order to receive $G_i(x)$ units of the token t_1.

A liquidity pool is created by depositing a positive amount of the atomic tokens t_0 and t_1 to a newly generated structure $\{\vec{l_{n+1}}, G_{n+1}(x)\}$. From that point on, only transactions involving the tokens t_0 and t_1 will be accepted by the pool.

Note that depositing and redeeming atomic tokens are operations that respectively add and remove liquidity from the liquidity pool. Since the process of swapping is the main topic of interest in this thesis, only this operation will be elaborated upon in further chapters.

3

The trading function

In Chapter 2, it was established that an AMM is characterised by it's trading function, which determines the output of a trade given the input. In this chapter, several trading functions and their properties will be examined. A class of AMMs called constant function market makers (CFMMs) will be introduced, and the properties their trading functions satisfy will be proven. Subsequently, the definition for a concentrated liquidity AMM is given and it's trading function will be described. The chapter is concluded by giving a new expression for the trading function of the concentrated liquidity AMM as a composition of explicit functions.

Let t_0 and t_1 be two atomic tokens, with reserves r_0 and r_1 respectively. In order to prevent redundancy, only swapping x units of t_0 for units of t_1 will be considered from now on, as the other direction can be deduced by symmetry. For the sake of simplicity, trading fees are assumed to be zero.

Definition 3.1 (Properties of the trading function)**.** *A trading function $G : \mathbb{R}_+^n \to \mathbb{R}_+$ may satisfy any the following properties*

- *Output-boundedness: : For all x, r_0, r_1 such that $x \geq 0$ and $r_0, r_1 > 0$:*

$$G(x, r_0, r_1) < r_1 \tag{3.1}$$

- *monotonicity : For $x_1 \geq x_0, r_0 \geq r'_0, r'_1 \geq r_1$:*

$$G(x_1, r_0, r_1) \geq G\left(x_0, r'_0, r_1\right) \tag{3.2}$$

- *Strict monotonicity : For one of the above inequalities being strict:*

$$G\left(y, r'_0, r_1\right) > G(x_1, r_0, r_1) \tag{3.3}$$

- *Additivity : For $G(x, r_0, r_1) = a, G\left(y, r_0 + x, r_1 - ax\right) = b$*

$$\Rightarrow G\left(x + y, r_0, r_1\right) = \frac{ax + by}{x + y} \tag{3.4}$$

- *Reversibility: For $G(x, r_0, r_1) = a$:*

$$G(ax, r_1 - ax, r_0 + x) = x \tag{3.5}$$

- *Homogeneity: For all a, x, r_0, r_1 such that $x \geq 0$ and $a, r_0, r_1 > 0$:*

$$G(ax, ar_0, ar_1) = a \cdot G(x, r_0, r_1) \tag{3.6}$$

- *Continuity at x_0: For all $r_0, r_1 > 0$, $G(x, r_0, r_1)$ is continuous at $x = x_0$*
- *differentiability at x_0: For all $r_0, r_1 > 0$, the derivative $G'(x, r_0, r_1)$ is defined at $x = x_0$*

- *Analyticity at x_0: For all $r_0, r_1 > 0$, $G^{(n)}(x_0, r_0, r_1)$ exists for every $n \in \mathbb{N}$*
- *Concaveness: For all x, r_0, r_1 such that $x \geq 0$ and $r_0, r_1 > 0$:*

$$\lim_{x \to x_0} \sup G(x, r_0, r_1) \leq G(x_0, r_0, r_1) \tag{3.7}$$

Remark. Whenever a property is defined in Definition 3.1, it is given for $G : \mathbb{R}^3_+ \to \mathbb{R}_+$. Note that the definition is analogous for higher dimensions.

Remark. Some of the properties in Definition 3.1 are stronger than others. For example, (strict monotonicity) → (monotonicity), (analyticity) → ((continuity) and (differentiability))

Before introducing several trading functions, the price of a token must be defined. In particular, the difference between an external and an internal price oracle.

Definition 3.2 (External price oracle)**.** *An external price oracle for tokens t_0, t_1 is a function $P_e : \{t_1\} \to \mathbb{R}$ which determines the market price of t_1 in terms of t_0, based on external parameters. By definition, the external price oracle cannot be determined by an AMM.*

Definition 3.3 (Internal price oracle)**.** *An internal price oracle for tokens t_0, t_1 is a function $P : \mathbb{R}^n \to \mathbb{R}$ that determines the price of an asset based on the parameters of the associated liquidity pool $\{\vec{l}, G(x)\}$*

Remark. When referring to the price of a token, the internal price oracle is meant. The external price oracle will be explicitly named when used.

In order to analyse the properties of various trading functions, the notion of slippage must also be defined

Definition 3.4 (Slippage)**.** *The slippage of a trade measures the difference between the price of an asset before a trade, and the actual price paid for the asset in a trade. I.e., for a trade $(x, \vec{l})$, the slippage $S : \mathbb{R}^n_+ \to \mathbb{R}$ is*

$$S = \frac{G(x)}{xP(\vec{l})} \tag{3.8}$$

3.1. Constant Function Market Makers

In this section, a specific class of AMMs, called constant function market makers (CFMMs) will be introduced. Subsequently, two types of CFMMs will be discussed, namely the constant sum market maker(CSMM) and the constant product market maker (CPMM). The mathematical and economic properties of both of these types of AMMs will be discussed. Furthermore, it will be shown which of the properties from Definition 3.1 hold.

Definition 3.5. *A Constant Function Market Maker is an Automated Market Maker such that for some constant function $\phi : \mathbb{R}^2 \to \mathbb{R}$, any swap satisfies:*

$$\phi(r_0 + x, r_1 - G(x)) = \phi(r_0, r_1) \tag{3.9}$$

Intuitively, a CFMM ensures that a certain measure of value, defined by the constant function, is being maintained regardless of the trades performed. Therefore, providing that the user agrees with the measure of value, little value can be extracted from the AMM. It has been shown by Angeris and Chitra [4] that under fairly general assumptions, the internal and external price oracles of a CFMM are equal.

3.1.1. Constant Sum Market Maker

Definition 3.6. *A constant sum market maker has the constant function:*

$$\phi(r_0, r_1) = r_0 + r_1 \tag{3.10}$$

Therefore, swapping x units of t_0 for $G(x)$ units of t_1 is only valid if:

$$\phi(r_0 + x, r_1 - G(x)) = \phi(r_0, r_1) \iff G(x) = x \tag{3.11}$$

and the price of t_1 in terms of t_0 is given by

$$P(r_0, r_1) = 1$$

It is trivial to verify that the trading function for a CSMM is strictly monotonic, additive, reversible, homogeneous, continuous, analytic and concave. However, it is not output bounded, since $G(x)$ may exceed r_1 whenever $r_0 > r_1$.

There are several significant drawbacks to a CSMM which limit it's use. Due to the linear output function $G(x) = x$, rational users will drain one of the assets from the AMM whenever $P_e(t_1) \neq 1$. Furthermore, due to the lack of output-boundedness, the liquidity cannot grow unbounded. In fact, the bound is determined by the first deposit. Despite these drawbacks, CSMMs are being used in liquidity pools of so-called stable-coins. These are coins which are supposed to have an equal price, because the price is pegged to an external currency. An example of a CSMM implementation is Mstable [18].

3.1.2. Constant Product Market Maker

Definition 3.7. *A constant product market maker has the constant function:*

$$\phi(r_0, r_1) = r_0 \cdot r_1 \tag{3.12}$$

From the definition it follows that swapping x units of t_0 for t_1 is only valid if:

$$(r_1 - G(x))\,(r_0 + x) = r_0 r_1 \tag{3.13}$$

Solving for $G(x)$ gives the trading function as:

$$G(x, r_0, r_1) = \left(x \frac{r_1}{r_0 + x}\right) \tag{3.14}$$

The price function $P : \mathbb{R}^2_+ \to (0, \infty)$ is given by

$$P(r_0, r_1) = \frac{r_0}{r_1} \tag{3.15}$$

Therefore, the marginal price after a trade of size x is given by:

$$P\,(x, r_0, r_1) = \frac{r_0 + x}{r_1 - G(x)} \tag{3.16}$$

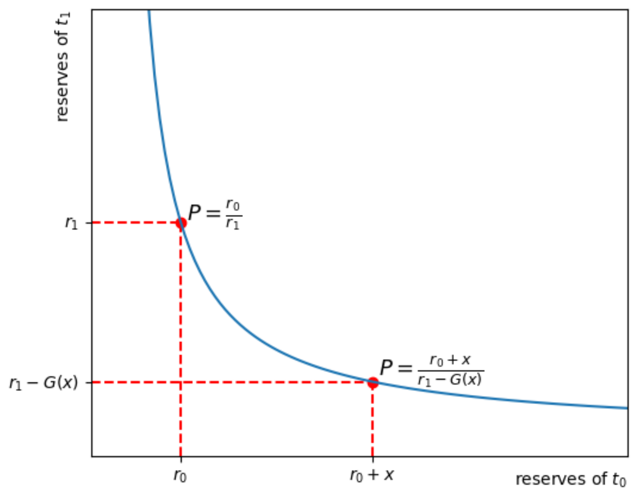


Figure 3.1: The swapping process for a Constant Product Market Maker

In appendix A it is proven that the trading function of a Constant Product Market Maker satisfies all of the proposed properties from Definition 3.1. Thus it is output-bounded, strictly monotonic, additive, reversible, homogeneous, continuous, concave and analytic. The possession of these properties automatically implies all the other properties.

Since the trading function is output-bounded, unlimited liquidity can be added. This allows for scalability of the CPMM, which is a large benefit over the CSMM. Furthermore, since the price function $P : \mathbb{R}^2_+ \to (0, \infty)$ is surjective, any price difference between the external and internal price oracle will be closed due to arbitrage. A possible downside to this type of price function, is that slippage grows linearly with the input amount x. Therefore, users are disincentivized to make large swaps. Due to the many favourable economic properties of the CPMM, it is used in many decentralised financial markets, most notably in Uniswap V2. [3]

Research into mixing CPMMs and CSMMs, by Port and Tiruviluamala [21], has shown that combining the properties of CPMMs and CSMMs can lead to lower slippage and higher stability of the AMM. Several implementations that mix these two AMMs exist, most notably Curve [8].

3.2. Uniswap V3

3.2.1. Concentrated liquidity

A key concept in the definition of Uniswap V3 is that of concentrated liquidity. Recall that in a constant function market maker, all liquidity is deployed on the price interval $(0, \infty)$. This is a rather inefficient method of allocating capital, since at any given price point only a fraction of the total liquidity is available. Uniswap V3 aims to reduce this inefficiency by introducing concentrated liquidity, allowing liquidity providers to provide liquidity on any price interval (P_a, P_b) for $P_a, P_b \in (0, \infty)$. This results in the creation of many different liquidity ranges, each with distinct liquidity amounts.

On a price interval (P_a, P_b), only enough liquidity needs to be provided in order to facilitate trading on that specific range. Therefore, the liquidity pool can act like a CPMM with much larger reserves. These larger reserves are called 'virtual reserves'. This works under the condition that once the price falls outside of the interval (P_a, P_b), the liquidity inside that interval is defined to be composed of just a single asset. Therefore, one of the assets has zero liquidity and the next price interval is entered. Figure 3.2 shows the difference between virtual reserves and the real reserves held.

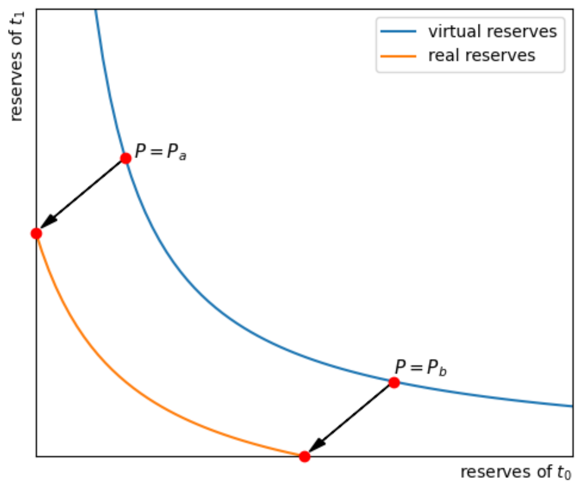


Figure 3.2: Virtual and real reserves

The Uniswap V3 trading function is defined by it's real reserves [23]

Definition 3.8 (real reserves)**.** *The real reserves x, y on a liquidity range $[P_a, P_b]$ in Uniswap V3 must satisfy*

$$\left(x + \frac{L}{\sqrt{P_b}}\right)\left(y + L\sqrt{P_a}\right) = L^2 \tag{3.17}$$

Remark. The real reserves are a translation of the concentrated liquidity reserves given by $r_0 \cdot r_1 = c$, such that the liquidity is exactly solvent on it's liquidity range, as can be seen in Figure 3.2.

Remark. In this chapter, calculations within liquidity ranges will be performed using the concentrated liquidity.

3.2.2. Ticks

In Uniswap V3, the price of an asset cannot be any arbitrary value. Instead, the price moves in so-called ticks. This means that at all times $P = 1.0001^i$ for some $i \in \mathbb{Z}$. This has the desirable property that the price always moves in steps of %0.01 percent, also known as a basis point. However, due to reasons shown in Section 3.2.3, the square root price of integer powers are tracked instead, so $p(i) = \sqrt{1.0001^i}$. Thus, the price moves in ticks of approximately 0.5 basis points, since $\sqrt{1.0001} \approx 1.00005$. An implication of this is that liquidity can only be provided between two, not necessarily adjacent, ticks. Therefore, the liquidity only has to be updated at ticks.

For the sake of simplicity, the price is assumed to be continuous in this section. Therefore the use of ticks is disregarded.

3.2.3. Within-interval swaps

Firstly, swapping within a liquidity range will be considered. Let t_0 and t_1 be two atomic tokens with virtual reserves of r and s respectively on some liquidity range $[P_a, P_b]$. As before, only trading x units of t_0 for $G(x)$ units of t_1 will be considered.

Denote r_0,r_1 and s_0,s_1 as the reserves before and after a swap of size x respectively. Similarly, let $\sqrt{P_0}$ and $\sqrt{P_1}$ be the square-root prices before and after a trade respectively. Within a liquidity range, the liquidity pool acts identical to a CPMM [14]. Define the liquidity on the interval $[P_a, P_b]$ as

$$L := \sqrt{rs} \tag{3.18}$$

As in a CPMM, $r \cdot s = c$ for some constant c. Therefore, the liquidity is constant within a liquidity range, and only needs to be updated when entering a new liquidity range. Furthermore, define the square root price as

$$\sqrt{P} := \sqrt{\frac{r}{s}} \tag{3.19}$$

And the change in price from a trade of input size x as

$$\Delta\sqrt{P} := \sqrt{P_1} - \sqrt{P_0} \tag{3.20}$$

The square root of the price is tracked, since this allows us to take advantage of the relationships shown in Theorems 3.1 and 3.2.

Remark. Note that the liquidity and the price must be balanced, i.e. $\sqrt{P} = \sqrt{\frac{r}{s}}$ as stated. Also note that when trading token t_0 for token t_1, the liquidity will be entirely composed of token t_1, since this is the asset required. Therefore, the liquidity can also be defined as

$$L := \min(\sqrt{P \cdot s}, \sqrt{\frac{1}{P} \cdot r}) \tag{3.21}$$

. This ensures that the liquidity and the price are balanced when the ratio of assets is not equal to the price. Additionally, taking the minimum makes sure that when trading in either direction, there is enough liquidity of the required asset.

As will be shown, the entire swapping process can be characterised using only L and $\sqrt{P}$, instead of using the reserves. The following two theorems show that the changes in input and output can be described in terms of these two variables.

Theorem 3.1. $x = L\Delta\sqrt{P}$

Proof.

$$\begin{aligned}
&r_1 - r_0 = r_1 - r_0 \\
&\Rightarrow \sqrt{r_1^2} - \sqrt{r_0^2} = r_1 - r_0 \\
&\Rightarrow \sqrt{\tfrac{s_1 r_1 r_1}{s_1}} - \sqrt{\tfrac{s_0 r_0 r_0}{s_0}} = r_1 - r_0
\end{aligned}$$

Using the fact that $L = \sqrt{s_1 r_1} = \sqrt{s_0 r_0} = \sqrt{sr}$

$$\begin{aligned}
&\Rightarrow \sqrt{sr}\left(\sqrt{\tfrac{r_1}{s_1}} - \sqrt{\tfrac{r_0}{s_0}}\right) = r_1 - r_0 \\
&\Rightarrow \sqrt{sr}\left(\sqrt{P_1} - \sqrt{P_0}\right) = r_1 - r_0 \\
&\Rightarrow \sqrt{sr} = \tfrac{r_1 - r_0}{\sqrt{P_1} - \sqrt{P_0}} \\
&\Rightarrow L = \tfrac{x}{\Delta\sqrt{P}} \\
&\Rightarrow x = L\Delta\sqrt{P}
\end{aligned}$$

□

Theorem 3.2. $G(x) = -L\Delta(\frac{1}{\sqrt{P}})$

Proof. From Theorem 3.1, we have $L = \frac{x}{\Delta\sqrt{P}} \implies x = L\Delta\sqrt{P}$

$$\begin{aligned}
\text{Using } \sqrt{P} = \sqrt{\tfrac{r}{s}} \Rightarrow G(x) &= -\sqrt{sr}\left(\sqrt{\tfrac{s_1}{r_1}} - \sqrt{\tfrac{s_0}{r_0}}\right) \\
&= -L\left(\frac{1}{\sqrt{\frac{r_1}{s_1}}} - \frac{1}{\sqrt{\frac{r_0}{s_0}}}\right) \\
&= -L\left(\frac{1}{\sqrt{P_1}} - \frac{1}{\sqrt{P_0}}\right) \\
&= -L\Delta\left(\frac{1}{\sqrt{P}}\right)
\end{aligned}$$

□

Similarly, $G(x)$ can be calculated using the current price P_0, the liquidity L, and the input x.

$$\begin{aligned}
G(x) = -L\Delta\frac{1}{\sqrt{P}} &= -L\left(\frac{1}{\sqrt{P_1}} - \frac{1}{\sqrt{P_0}}\right) \\
&= -L\left(\frac{1}{\sqrt{P_0} + \Delta\sqrt{P}} - \frac{1}{\sqrt{P_0}}\right) \\
&= -L\left(\frac{1}{\sqrt{P_0} + \frac{x}{L}} - \frac{1}{\sqrt{P_0}}\right)
\end{aligned}$$

3.2.4. Cross-interval swaps

If a certain liquidity range does not contain enough liquidity to satisfy an order, the price will move into the next liquidity range. Note that this will only happen if there is a liquidity range with a lower bound equal to the upper bound of the current liquidity range. If there is an interval with $L = 0$ and positive length, the order will only be partially executed.

Suppose liquidity is provided on the interval $[\sqrt{P_a}, \sqrt{P_b}]$ and assume $P \geq P_b$. This means that there are no reserves of t_1 left in this price interval, i.e. $s = 0$ on $[\sqrt{P_a}, \sqrt{P_b}]$. Thus the position is fully in t_0.

Recall $G(x) = -L\Delta\frac{1}{\sqrt{P}}$. Thus, if the position is fully in t_0:

$$G(x) = -L\Delta\frac{1}{\sqrt{P}} = -L\left(\frac{1}{\sqrt{P_b}} - \frac{1}{\sqrt{P_0}}\right) \tag{3.22}$$

Similarly, when trading over the entire interval:

$$G(x) = -L\Delta\frac{1}{\sqrt{P}} = -L\left(\frac{1}{\sqrt{P_b}} - \frac{1}{\sqrt{P_a}}\right) \tag{3.23}$$

In reality, often only L and $\sqrt{P}$ are known. However, once a liquidity range is entered, $\sqrt{P} = \sqrt{P_a}$ by definition. Therefore,

$$\sqrt{P_b} = \frac{L\sqrt{P_a}}{L - \sqrt{P_a}x} \tag{3.24}$$

To conclude, if a liquidity range does not contain enough liquidity to satisfy an order, the virtual reserves are added to the total amount swapped, and the next liquidity range is being entered at $\sqrt{P_b}$ as shown above. Subsequently, the liquidity, the input and the total output are updated, and the process is repeated

3.2.5. Explicit form of the Uniswap V3 trading function

Let's consider the swapping process across multiple liquidity ranges as described. Let x_1 be the amount of input tokens. Assume there are n liquidity ranges denoted as $[\sqrt{P_0}, \sqrt{P_1}]$, $[\sqrt{P_1}, \sqrt{P_2}]$, ..., $[\sqrt{P_{n-1}}, \sqrt{P_n}]$, with corresponding liquidity vectors $\vec{l}_1, ..., \vec{l}_n$, where $\sqrt{P_0}$ is the initial price. Let $s_1, ..., s_n$ be the reserves of t_1 in each price interval, and assume $\sum_{i=1}^{n} s_i \geq G(x)$, i.e. there is enough liquidity in the liquidity ranges to fulfill the order.

Firstly, the trading function will be described iteratively. For each liquidity range i s.t. $1 \leq i \leq n$, perform the following steps:

- Determine $\sqrt{P_i'}$
  - If $x_i \geq s_i$, set $\sqrt{P_i'} = \sqrt{P_i}$
  - Otherwise, set $\sqrt{P_i'} = \frac{l_i\sqrt{P_i}}{l_i - \sqrt{P_i}x}$ (3.24)
- Calculate $\Delta\sqrt{P_i'} = \sqrt{P_i'} - \sqrt{P_{i-1}'}$
- Calculate $\Delta\frac{1}{\sqrt{P_i'}} = \left(\frac{1}{\sqrt{P_i'}} - \frac{1}{\sqrt{P_{i-1}'}}\right)$
- Calculate the output for this iteration $G_i(l_i, x,) = -l_i\Delta\frac{1}{\sqrt{P_i'}}$ (3.2)
- Update the total output $T_i(x) = T_{i-1}(x) + G_i(l_i, x)$
- Update the remaining input $x_{i+1} = x_i - l_i\Delta\sqrt{P_i}$
- Terminate if $x_{i+1} \leq 0$

Now assume that the final price is known and equal to P_n. The output of each iteration can be expressed as the composition of functions

$$G_i(\vec{l}, x) = G_i(l_i, x_{i-1} - G_{i-1}^{-1}(G_{i-1}(...G_2^{-1}(G_2(l_i, x_1 - G_1^{-1}(G_1(l_i, x_1))...)))))$$

And the total output of a trade of size x is:

$$T_n(x) = \sum_{i=1}^{n} G_i(\vec{l}, x)$$

Note that in this characterisation, the number of compositions, n, is dependent on the output of the functions $G_i(\vec{l}, x)$. An explicit form of the trading function can therefore only be found if this dependence is eliminated. Therefore, the following conjecture is stated.

Conjecture 3.1. *Let $[P_0, P_1]$ be a liquidity range, with liquidity l. Let $G_0(l, x)$ denote the output of a trade with input x on the liquidity range. Furthermore, assume that $\sum_{i=1}^{n} s_i \geq G_0(l, x)$. Dividing $[P_0, P_1]$ into n intervals*

$$[P_0 + \frac{j}{n}(P_1 - P_0), P_0 + \frac{j+1}{n}(P_1 - P_0)]$$

for $0 \leq j \leq n$, with corresponding liquidity $l_j = \frac{L}{n}$. Then as $n \to \infty$:

$$\sum_{i=1}^{\infty} G_i(\vec{l}, x) \approx G_0(l, x) \tag{3.25}$$

This conjecture states that the output from an infinite number of liquidity range is approximately equal to the output from a single liquidity range on the same domain. Although proving this conjecture is beyond the scope of this thesis, in Chapter 4 this conjecture will be supported using simulated trades.

4 Simulating the Uniswap V3 trading function

As mentioned earlier, the primary objective of this thesis is to determine the shape and properties of the trading function of Uniswap V3. The previous chapter employed an analytic approach based on the Uniswap V3 whitepaper [23]. In this chapter, an alternative approach is employed, wherein the trading function is simulated under various initial conditions.

A newly devised simulation method called transaction mocking will be introduced, which enables the precise simulation of the trading function as implemented, with complete control over the initial conditions. Using simulations, it will demonstrated that, to a great extent, the shape of the trading function can be characterized by the number of liquidity ranges and their distribution. Finally, Monte Carlo simulations [22] will be conducted to investigate the distribution of the trading function and observe the effects of increasing the number of liquidity ranges.

4.1. Method of transaction mocking

In implementations, an Automated Market Maker consists of a collection of smart contracts [24] deployed on a blockchain. Users interact with the AMM using transactions. Using a message call, a user can call one of the functions in a smart contract in order to perform a desired operation. Alternatively, a contract creation transaction deploys a smart contract that can interact with other smart contracts given a set of conditions. In order to prevent ambiguity in smart contracts and perform calculations with maximum accuracy, a specialised programming language called Solidity [1] has been developed for smart contracts.

To analyze the properties of the Uniswap V3 trading function, it is crucial to have a reliable and accurate method for simulating a large number of transactions. The method that has been devised will be called transaction mocking. Essentially, the implementation of Uniswap V3 is replicated and deployed on a local instance of a blockchain. Subsequently, liquidity pools and transactions are simulated to recreate the precise trading conditions desired. The outputs of these transactions are then read from the blockchain. For the simulations, a lightweight version of Uniswap V3 containing the trading logic and functionality, was cloned [16]. Instead of deploying a local Ethereum blockchain, which is complex and computationally demanding, the mocking process was conducted using the testing infrastructure of Uniswap V3. This testing infrastructure allows the mocking of blockchain addresses, smart contracts, and transactions without the need to run an actual blockchain instance.

For each simulated trade, a new smart contract is deployed. This smart contract generates two new tokens and proceeds to mock a liquidity pool with the exact parameters required for that particular simulation. Subsequently, a function within the smart contract is executed, mocking a wallet and performing swap transactions of a predetermined size on the liquidity pool. Throughout the execution of these transactions, the transaction logs of the mocked blockchain are recorded. From these logs, the output of each swap is filtered and processed using Python. The relevant code can be found in Appendix A.

4.2. Reducing the parameter space

In order to perform simulations, the domain of inputs needs to be specified. In order to reduce redundancy and increase the significance of the results, it is desirable to reduce the dimension of the parameter space as much as possible. This allows the cause of observed effects to be determined more accurately, and makes interpretation of results easier. Therefore, it will be shown that varying the input and initial price have a predominantly linear effect on the output. Subsequently, these variables can be held constant while performing the simulations for liquidity ranges.

As shown in Section 3.2, the input parameters of a trade are:

- a sequence of price intervals $[\sqrt{P_i}, \sqrt{P_{i+1}}]$ with corresponding output reserves l_i for $1 \leq 0 \leq n-1$
- The input amount $x \geq 0$
- An initial price $\sqrt{P_{initial}}$

Firstly, the case where there is not enough liquidity in the price intervals combined to completely fulfill the trade, i.e. $\sum_{i=1}^{n} s_i < G(x)$, can be disregarded. In this scenario, only part of the trade will be executed, meaning that this case is equivalent to having a lower input x.

4.2.1. Varying the input

The effect of varying the input amount x on the output is investigated. Note that the relationship between the input and output of a trade is heavily dependent on other parameters, such as the liquidity ranges. Therefore, only the output curve for a single liquidity range is considered in this simulation. The effect of altering the distribution and number of liquidity ranges will be shown in further simulations.

In Figure 4.1a, it can be seen that varying the input in a single liquidity range result in a nearly linear relationship between input and output. Performing least-squares regression [20] shows that the deviation from linearity is very minor, although interestingly it seems to be a perfectly quadratic curve. A natural response to this is to fit a quadratic polynomial to the data. As shown in Figure 4.1c, this results in a residual in the form of a third-order polynomial. Similarly, polynomial regression of order three results in a fourth order residual, and in general nth order regression results in a residual of order $n+1$ for up to at least $n = 8$. Above $n = 8$, limitations in tick spacing and the numerical stability of polynomial regression prevent any further analysis.

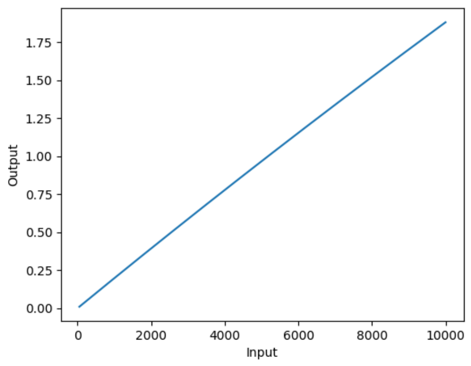
The observed curvature makes economic sense, since it shows that the trading function is concave, i.e. $G''(x) < 0$. It is necessary for the trading function to be concave, since this means that the price increases as the input x increases.

By comparing the polynomial fit for different orders, it can be seen that the coefficients of the lower order terms stay nearly identical when performing higher order regressions. Thus, it is observed that the first seven terms of the power series are approximately:

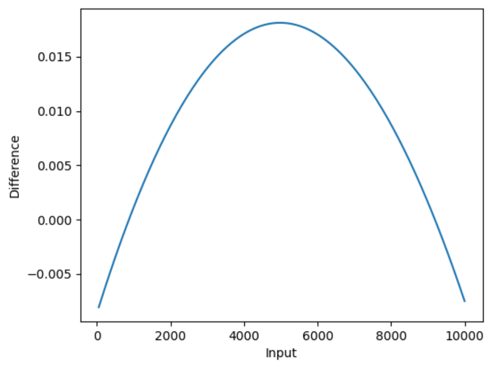
$$0.000198x - 1.138 \cdot 10^{-9}x^2 + 6.517 \cdot 10^{-15}x^3 - 3.7331 \cdot 10^{-20}x^4 + 2.136 \cdot 10^{-21}x^5 + 1.223 \cdot 10^{-30}x^6 \tag{4.1}$$

Where the coefficients of the terms are denoted by $a_0, ..., a_6$. Note that the linear term $a_1 \approx 0.0002$, since the initial price is 5000. Also note that the constant term oscillates seemingly randomly in the order of 10^{-15}. Since a constant term is not expected, this is treated as error an thus left out. Dividing each of the coefficients a_n by a_{n-1}, a very constant common ratio of $r \approx 5.725$ emerges, with the first five terms being separated by less than 0.0001%. Therefore, the following input-output relation is proposed for $P_{initial} = 5000$:

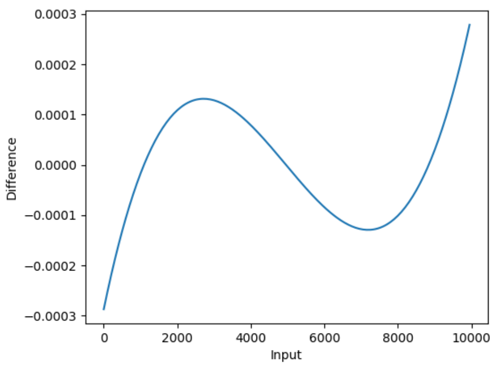
$$G(x) = 0.0002 \sum_{n=0}^{\infty} (5.725 \cdot 10^{-6})^n \cdot x^n \tag{4.2}$$



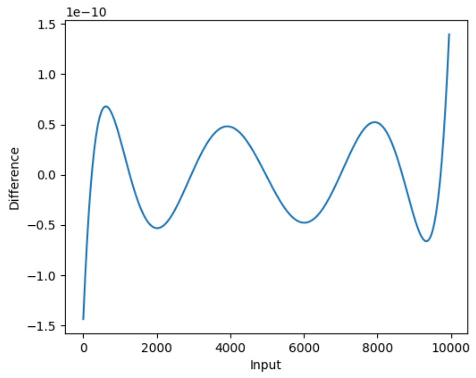
(a) Varying the input for equidistant liquidity ranges



(b) The residuals from linear regression, $MSE \approx 6 \cdot 10^{-6}$



(c) Quadratic regression, $MSE \approx 1.2 \cdot 10^{-8}$



(d) Sixth order regression, $MSE \approx 1.8 \cdot 10^{-21}$

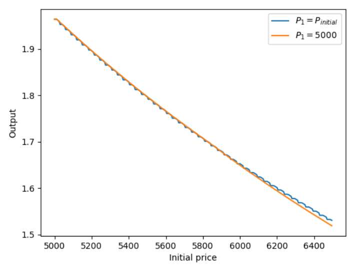
Figure 4.1: Varying the input on a single liquidity range, with residuals of first, second, fourth and sixth order polynomial regression shown

Similar power series can be constructed for other initial prices. Since the non-linear terms are extremely small, the input-output relation can be assumed to be linear on a single price interval for all practical purposes. An implication of this is that when varying the initial price, any significant non-linear effects can be attributed to other factors, such as the placement of liquidity ranges. Thus, when simulating over a discrete input, such as the number of liquidity ranges, the initial price can be varied to investigate the differences in the shape of the trading function. Furthermore, when simulating over a continuous input, such as a distribution of liquidity ranges, the input can be kept constant as it will not influence the shape of the output distribution.

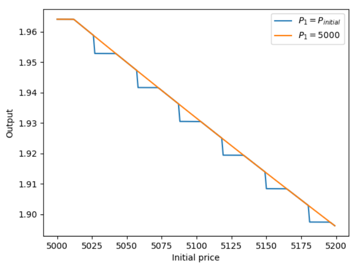
4.2.2. Varying the initial price

The effect of shifting the initial price $\sqrt{P_{initial}}$ will be investigated. Equations 3.22 and 3.23 suggest that varying $\sqrt{P_{initial}}$, for $\sqrt{P_1} \leq \sqrt{P_{initial}} \leq \sqrt{P_2}$ is equivalent to setting a shorter price interval $[\sqrt{P_{initial}}, \sqrt{P_2}]$. Fix $P_1 = 5000$, $P_2 = 7000$ and the input $x = 10000$. Note that since the relative difference between outputs is of interest, the exact price interval is irrelevant. In Figure 4.2, the output for various values of $P_{initial}$ between 5000 and 6500 is plotted, for both price intervals $[P_1, P_2]$ and $[P_{initial}, P_2]$.

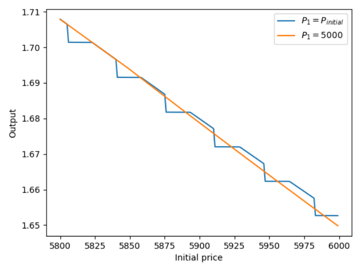
There are two surprising effects that can be observed from the output of this simulation. Firstly, as can be seen on the smaller interval in Figure 4.2b, the output for $P_1 = P_{initial}$ deviates from the constant price interval periodically. The period of each deviation is approximately 15 units of input, and the interval between each oscillation is also around 15 units. Note that this effect is not due to the step size of $P_{initial}$, since the step size is 1. Furthermore, since the trade is performed within a liquidity range, this effect is independent of



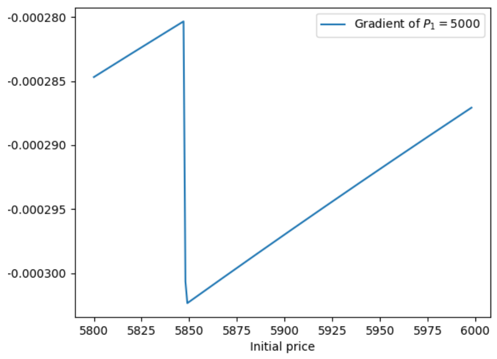
(a) $P_{initial} \in [5000, 6500]$



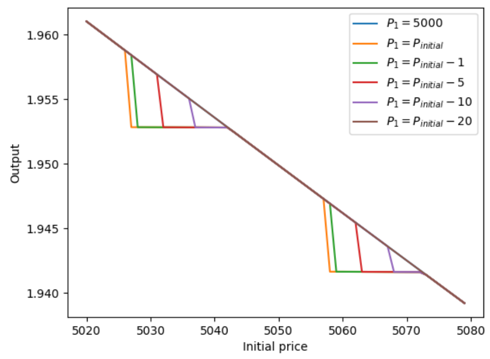
(b) $P_{initial} \in [5000, 5200]$



(c) $P_{initial} \in [5800, 6000]$



(d) The gradient of the fixed price interval for $P_{initial} \in [5800, 6000]$



(e) Output for several distances between P_1 and $P_{initial}$

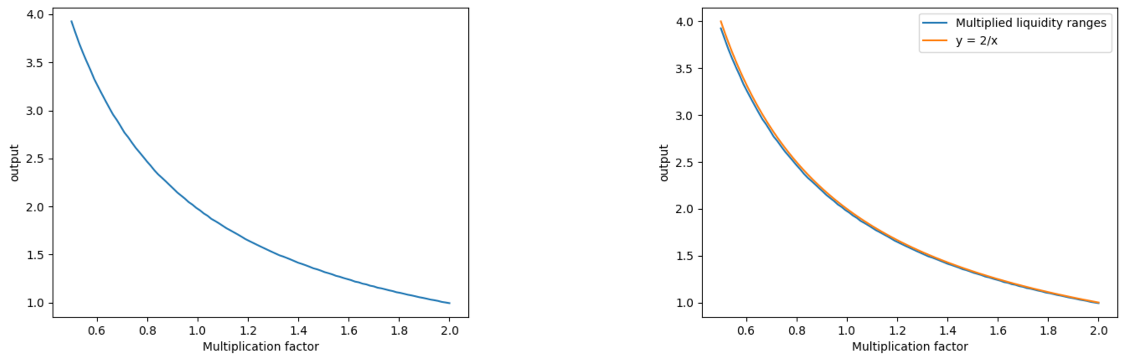
Figure 4.2: Varying the initial price

the distribution of liquidity ranges. Additionally, the magnitude of the effect stays equal for different prices and liquidity ranges. This suggests that the effect is solely dependent on the relation between the bounds of the liquidity ranges and the current price. In Figure 4.2b it can be seen that for $P_1 = 5000$, the output stays constant until approximately $P_{initial} = 5011$. This suggests that at certain values for P_1, the output stays constant for values of $P_{initial}$ close to P_1. In order to investigate this effect more closely, the output was plotted for several price intervals depending on $P_{initial}$. In Figure 4.2e, it can be seen that the length and the intensity of the shift is directly proportional to the distance between P_1 and $P_{initial}$. The exact cause of the effect is not yet known to the author.

The second effect that can be observed, is that the direction of the two output relations seem to diverge, starting around $P = 5850$. This effect can be observed better when looking at the smaller interval in Figure 4.2c. From Figure 4.2d, it can be seen that this effect is caused by a sudden jump in the gradient of the output curve for $P_1 = 5000$, i.e. the marginal price. This jump is due to the ratio of the reserves shifting. This effect has been described in more detail in Remark 3.2.3. At $P = 5850$, the liquidity calculated in terms of s will suddenly be smaller than that in terms of r. Therefore, the gradient of the price curve will change. The reason why this effect doesn't occur for $P_1 = P_{initial}$, is that the reserves are almost fully in terms of s at the start of a liquidity range. Therefore, the liquidity will not be smaller in terms of s than r in the first part of the liquidity range. Since, for $P_i = P_{initial}$, the current price is always near the start of the liquidity range, the effect doesn't occur.

4.2.3. Varying the length of all liquidity ranges

Firstly, varying both the length and position of a liquidity range is observed, i.e. multiplying P_i by a factor c for all $1 \le i \le n$. Let $P_1 = 5000$, and $P_2 = 6000$ at multiplication factor $c = 1$. Set $P_{initial} = P_1$. In Figure 4.3a, the output can be seen for multiplication factor $0.5 \le c \le 2$. Interestingly, Figure 4.3b shows that the output greatly resembles the output of a CPMM with constant product $r_0 r_1 = 2$. As before, this effect can be explained by Remark 3.2.3. Since both P_a and P_b are being multiplied by a factor x, Theorem 3.2 states that $G(x) = L\Delta\frac{1}{\sqrt{P}}$ should be multiplied by a factor $\frac{1}{\sqrt{x}}$. However, since the liquidity stays equal, the ratio of assets in the pool does not change. Therefore, if before the trade the reserves were s and $r = \sqrt{P_1}s$, and the price is

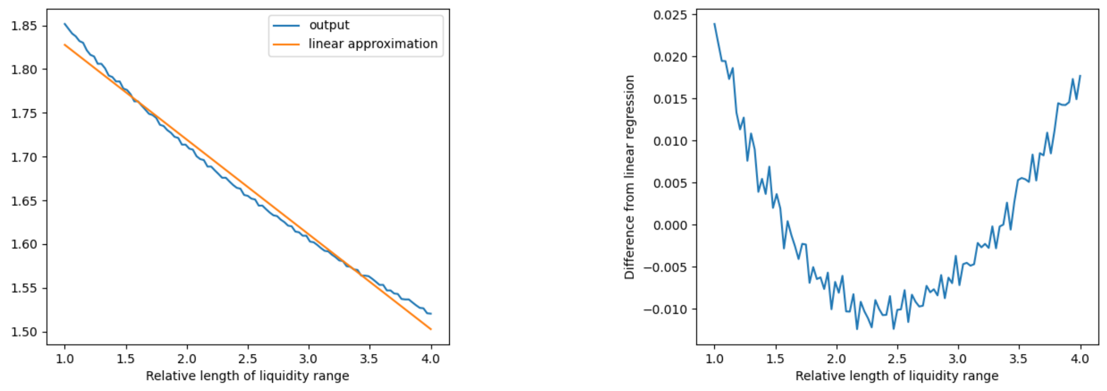


(a) The output for a range of multiplication factors

(b) Comparison with the trading function of a CPMM with $r_0 r_1 = 2$

Figure 4.3: Multiplying all liquidity ranges

multiplied by a factor x, the liquidity will be multiplied by another factor $\frac{1}{\sqrt{x}}$. Thus, the output will change with a factor $\frac{1}{\sqrt{x}} \cdot \frac{1}{\sqrt{x}} = \frac{1}{x}$. In this particular case, the constant factor is approximately 2, since the initial price is 5000 and the input is 10000.



(a) Output curve compared with the linear least-squaures approximation

(b) Difference between the output and the least-squares approximation

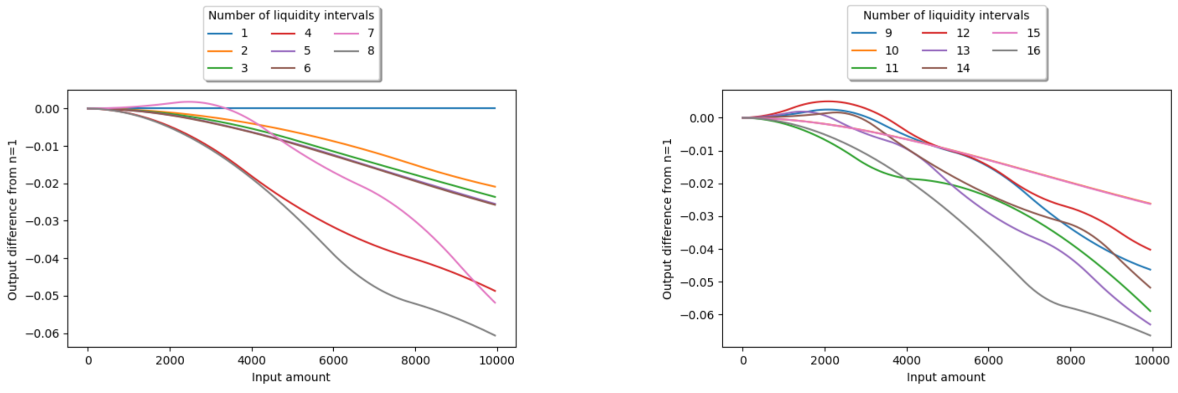
Figure 4.4: Multiplying the length of liquidity ranges

Subsequently, multiplying the length of a liquidity range $P_{i+1} - P_i$, by a factor c, while keeping the start of the liquidity range equal, is considered. As can be seen in Figure 4.4a, the relationship between the length of liquidity ranges and the output is nearly linear, with a quadratic residual. When performing quadratic least-squares regression, the residual seems to be in the form of a third order polynomial. At higher orders, no pattern can be spotted anymore, although this is likely due to the high level of noise. This result is strikingly similar to the input change shown in Figure 4.1a, although with significantly more noise in the data. This makes sense, since varying the length of a liquidity range is essentially opposite to varying the input. Increasing the length of a liquidity range increases the price, and thus decreases the output.

4.3. Varying the number of liquidity ranges

The effect of varying the number of liquidity ranges on the input-output curve is considered. Instead of varying the number of liquidity ranges for a fixed price, the effect will be observed for a price curve as in Figure 4.1a. This allows us to examine the nonlinear effects, and the possible convergence of the trading function more closely.

As before, simulations occur on the price interval $[5000, 6000]$. On this interval, n evenly spaced liquidity ranges are created for $1 \leq n \leq 16$, with equal liquidity on each interval and equal total liquidity for each iteration. Subsequently, trades take place over these liquidity ranges for 200 input values between 0 and 10000. In Figure 4.5, the deviations of each output curve from $n = 1$ can be seen.



(a) n=1 to n=8 (b) n=9 to n=16

Figure 4.5: Deviations from n=1 for different numbers of liquidity ranges

From Figure 4.5, it can be seen that the simulations for $n = 2, 3, 5, 6, 10$ and 15 closely resemble each other, and move in a predictable, monotonic manner. As of yet, no clear explanation has presented itself why this is the case. Furthermore, it can be seen that the simulations for $n = 4, 8$ and 16 have a very similar shape and direction, where the deviations are more pronounced for the simulations with a higher number of liquidity ranges. The same can be said for $n = 7$ and $n = 12$. However, this pattern of similarity for multiples of liquidity ranges seems to break for $n = 6$ and $n = 12$, as these have a completely different shape. Due to the chaotic structure of these output relations, no conclusions about convergence in output as n increases can be made.

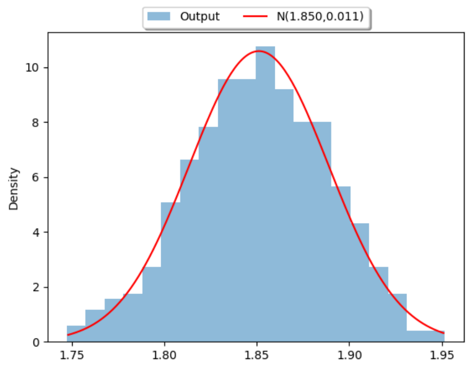
4.4. Monte Carlo simulations over a uniform distribution of liquidity ranges

In this section, the distribution of outputs is examined by performing Monte Carlo simulations [22] over a given input distribution. Previously, it was established that varying the total length of the liquidity ranges, $P_N - P_1$ for $N - 1$ liquidity ranges, has a predominantly linear impact on the output. Therefore, fix $P_1 = 5000$ and $P_N = 6000$. Subsequently, n independent samples of size $N - 2$ over the distribution Uniform$(5000, 6000)$ are generated. These points will become the values $P_2, P_3, ..., P_{N-1}$, i.e. the boundaries between the price intervals. Using these values, the output of the trading function is generated from a fixed input, and equal liquidity on each interval.

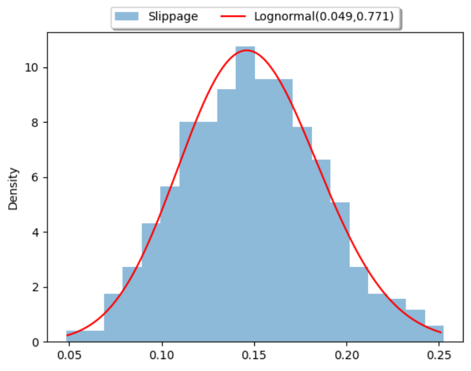
In Figure 4.6a, the outputs for $N = 5$, i.e. 4 liquidity ranges, with sample size $n = 500$ are shown. Overlayed is the best fitting normal distribution. Performing the Kolmogorov-Smirnov test [17], the output is normally distributed with $p = 0.94$.

Note that in the case that there would be no slippage at all, the output of all simulations would be 2, as the starting price is $P_1 = 5000$, and the input is $x = 10000$. Therefore, all data points i can be transformed to $2 - i$ to obtain the absolute slippage, i.e. the reduction in output due to the price increase while trading. Note also that the data seems to be somewhat left skewed. Therefore, a log-normal distribution is fitted to the transformed data. In Figure 4.6b, it can be seen that the log-normal fits the data very well, with a p-value from the Kolmogorov-Smirnov test of 0.99. Note that, because the output cannot exceed 2, the log-normal should technically be truncated at that value. However, since this is approximately 5 standard deviations from the mean, truncation makes no realistic difference.

In order to test for convergence in distribution as the number of liquidity ranges increases, 500 simulations over a uniform distribution have been performed for n liquidity ranges, with $4 \leq n \leq 16$. Assuming that the data is log-normally distributed for all n, a logarithmic transformation is applied to make the observations

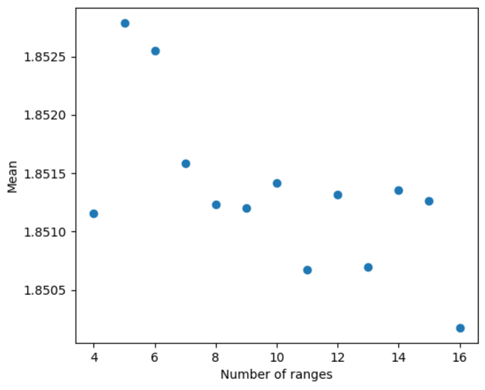


(a) The normal distribution fitted to the output, $p = 0.94$

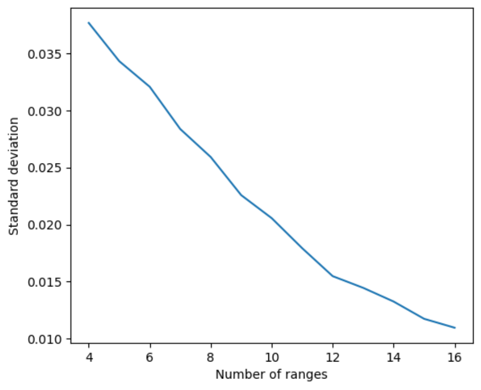


(b) The lognormal fitted to the slippage, $p = 0.99$

Figure 4.6: Monte Carlo simulations over a uniform distribution, with sample size $n = 500$



(a) Mean output



(b) Standard deviation of output

Figure 4.7: Monte Carlo simulations over a uniform distribution, for $4 \leq n \leq 16$

normally distributed. As can be seen in Figure 4.7, the standard deviation of the output steadily decreases as the number of liquidity ranges increases. This is to be expected, as increasing the number of liquidity ranges decreases the variance in the length of the liquidity ranges. On the other hand, there is no clear relationship between the number of liquidity ranges and the mean output.

Therefore, One-way Analysis of Variance (ANOVA) [13] is used to test the null hypothesis:

$$H_0 : \mu_1 = \mu_2 = ... = \mu_{16} \tag{4.3}$$

Against the alternative hypothesis

$$H_1 : \mu_1 \neq \mu_2 \neq .... \neq \mu_{16} \tag{4.4}$$

Where the usual assumptions for ANOVA of homoscedasticity and normality of residuals apply. Performing the one-way ANOVA test gives a p-value of 0.97 for the results under the null-hypothesis. This means that with high likelihood, the means for different numbers of liquidity ranges are equal.

To conclude, in this section it has been shown that when trading over uniformly distributed liquidity ranges, the output has a lognormal distribution. Additionally, the mean outputs for $4 \leq n \leq 16$ are shown to be equal with high likelihood. Since, logically $\sigma \to 0$ as $n \to \infty$, this result is a strong indication that the output for n liquidity ranges converges in value as $n \to \infty$. Furthermore, a high p-value for the one-way ANOVA test is also a strong indication that $\mu_1 = \mu_2 = ... = \mu_\infty$.

5 Conclusion

In this thesis, the mathematical properties of Automated Market Makers, and in particular Uniswap V3, have been investigated. An axiomatic definition for an AMM, that generalises the works of Bartoletti et al. [6] and Bichuch and Feinstein [7] was constructed. These definitions allow Constant Function Market Makers to be accurately described, as well as AMMs based on concentrated liquidity. Furthermore, both the trading function and the transition between states of the blockchain are described within the same framework.

Several Automated Marker Makers were investigated. Although the Constant Sum Market Maker has useful properties for assets with a stable price, the existence of a bound on liquidity, and the constant price make it unsuitable for most types of assets. The trading function of Constant Product Market Makers posseses many desirable mathematical properties, such as output-boundedness, additivity and analyticity. Among others, these properties allow for the unbounded growth of liquidity and minimise the difference between the internal and external price oracle due to arbitrage. However, a big limitation of the CPMM is that liquidity is always deployed on the price interval $(0,\infty)$. This limitation is tackled by the trading function of Uniswap V3, which utilises the concept of concentrated liquidity to allow users to deploy liquidity on an arbitrary interval.

Due to concentrated liquidity, the trading function of Uniswap V3 behaves similar to a CPMM within a price interval. Whenever a new price interval is reached, the previous interval becomes completely composed of a single asset, and the liquidity, input and output are updated. Therefore, the trading function is determined iteratively, and can be described using a composition of function. However, the number of functions composed is dependent on the functions itself. This thesis conjectures that the trading function can be approximated by taking an infinite number of compositions, while letting the size of each liquidity interval tend to zero.

Additionally, the Uniswap V3 trading function was also simulated using the devised method of transaction mocking. it was shown that changing the input amount within a price interval has a predominantly linear effect on the output, and that the precise input-output relation can be represented as a power series. The effect of varying the initial price was observed, for both fixed and dynamic price intervals. Although the output for a fixed price interval was similar to varying the price, the dynamic price intervals produce jumps in output that have yet to be fully explained. Furthermore, it was found that when moving the initial price past half of the price interval, the gradient can jump due to liquidity calculations.

Simulations were also used to test the conjecture from Chapter 3. By varying the number of liquidity ranges, no convergence of the input-output relation can be spotted directly. By performing Monte Carlo simulations over a uniform distribution of liquidity ranges, It was shown that the output of each simulation run is log-normally distributed with a very high significance. Furthermore, it was shown that the true mean for different numbers of liquidity ranges is very likely to be equal for at least up to 16 liquidity ranges. Combining these results with the fact that the standard deviation naturally goes to zero as the number of liquidity ranges increases, convergence in value as $n \to \infty$ is likely. This supports the statement of the conjecture.

6 Discussion

During the writing of this thesis, some research questions could not be fully answered, and some assumptions were made. These shortcomings are discussed in this section. In the simulation study, several obtained results could not be fully explained. Most significantly, the cause of the jumps in output observed when $P_{initial}$ is near P_1 could not be identified. Although it could be established that the effect is directly linked to the distance between the initial price and the start of the price interval, and that it occurs independent of other factors such as input and price, the result contradicts the theoretical result that fixed and variable starts of price intervals should give equal results. Another effect that could not be fully explained, is the shape of the input-output relation for different numbers of liquidity ranges. Especially the very close grouping of the outputs for $n = 2, 3, 5, 6, 10$ and 15 is notable, but no clear cause was established.

The simulation study was also slightly limited in it's scope due to the simulation setup. Although the setup was very accurate, since the programming language Solidity is designed to not allow any rounding errors, this part of the design also limited the simulations. The absence of floating point numbers prevented the use of very small liquidity intervals, which meant that simulations could not reliable be performed for more than 16 liquidity intervals. Although there is most likely a method to utilize smaller liquidity intervals, as this is also done in practice, the author was unable to find it. This shortcoming in the simulation setup meant that convergence for the number of liquidity intervals could not be tested for more than 16 liquidity intervals.

6.1. Recommendations

6.1.1. Analysing the trading function using dynamical systems theory

In this thesis, an expression for the Uniswap V3 trading function was found as a composition of functions. Expressions of this form are well studied in the theory of iterated functions and discrete dynamical systems. A useful continuation of this research would be to investigate this expression using the tools provided by dynamical systems theory, and see which properties of the trading function can be deduced, given that the number of iterations are known. Furthermore, simulations for different numbers of liquidity ranges have made it plausible that the trading function converges as the number of liquidity ranges increases to infinity. If convergence can be proven, the theory of infinite compositions of analytic functions [12] could be used to approximate the trading function using an infinite number of trading functions acting on an infinitely small price interval. This approximation would be independent of the number of iterations n.

6.1.2. Principal Component Analysis for simulations

The trading function of Uniswap V3 has multiple highly correlated input parameters, such as the initial price, price intervals and the input amount. By varying these parameters independently, it was made plausible that some of these parameters have equivalent effects, and thus can be disregarded. However, in order to find the parameter space that has the largest, independent influence on the output, dimensionality reduction can be applied. By performing Principal component analysis (PCA) [11] on a large dataset using all input parameters, the principal components that contain the largest variance of data and are orthogonal can be found. This would allow for better interpretation of the simulation results, using fewer figures.

A
Proofs of CPMM properties

Output-Boundedness

Let $x_1 r_0, r_1 > 0$. Then

$$G(x, r_0, r_1) = x \cdot \frac{r_1}{r_0 + x} < \frac{x r_1}{x} = r_1$$

Strict Monotonicity

let $x_1 > x_0, r_1' > r_1, r_0' < r_0$ Then

$$G\left(x_1, r_1', r_0'\right) = x_1 \cdot \frac{r_1'}{r_0' + x_1} > x_0 \frac{r_1}{r_0 + x_0} = G(x_0, r_1, r_0)$$

Additivity

Let $x_0, x_1, r_0, r_1 > 0$ and

$$G(x_0, r_0, r_1) = x_0 \cdot \frac{r_1}{r_0 + x_0} = a$$

$$G(x_1, r_0 + x_0, r_1 - a x_0) = x \cdot \frac{r_1 - a x_0}{r_0 + x_0 + x_1} = b$$

Then

$$G(x_0 + x_1, r_0, r_1)$$

$$\begin{aligned}
&= \frac{r_1}{r_0+x_0+x_1} \\
&= \frac{r_1(r_0+x_0)(x_0+x_1)}{(r_0+x_0+x_1)(r_0+x_0)(x_0+x_1)} \\
&= \frac{1}{x_0+x_1} \frac{r_0 r_1 x_0 + r_1 x_0^2 + r_1 x_0 x_1}{(r_0+x_0)(r_0+x_0+x_1)} \\
&= \frac{1}{x_0+x_1} \left(\frac{r_1 x_0}{r_0+x_0} + \frac{r_0 r_1 x_1}{(r_0+x_0)(r_0+x_0+x_1)} \right) \\
&= \frac{ax+by}{x+y}
\end{aligned}$$

Homogeneity

let $a, x, r_0, r_1 > 0$.
Then,

$$G(ax, ar_0, ar_1) = ax \cdot \frac{ar_1}{ar_0 + ax} = a \cdot G(x, r_0, r_1)$$

Reversibility

let $x, r_0, r_1 > 0$ and $G(x, r_0, r_1) = a$.
Then,

$$\begin{aligned} G(ax, r_1 - ax, r_0 + x) &= ax \frac{r_0 + x}{(r_1 - ax) + ax} \\ &= ax \frac{r_0 + x}{r_1} \\ &= ax \left(\frac{r_1}{r_0 + x} \right)^{-1} \\ &= xG(x, r_0, r_1)\, G^{-1}(x, r_0, r_1) = x \end{aligned}$$

Continuity

Clearly, $G(x) = x \cdot \frac{r_1}{r_0 + x}$ for $r_0, r_1 > 0$ is continuous $\forall x \in [0, \infty)$

Concaveness

let $r_0, r_1 > 0, x \geq 0$. It needs to be shown that the second derivative of G is non-positive on the domain

$$\begin{aligned} G(x) &= \frac{r_1 x}{r_0 + x} \\ G'(x) &= \frac{(r_0 + x)\, r_1 - r_1 x}{(r_0 + x)^2} \\ &= \frac{r_0 r_1}{(r_0 + x)^2} \\ G''(x) &= -\frac{2 r_0 r_1}{(r_0 + x)^3} \leq 0 \quad \text{for all } r_0, r_1 > 0, x \geq 0 \end{aligned}$$

Therefore, $G(x, r_0, r_1)$ is concave.

Analyticity

In order to show that $G(x)$ is analytic, it needs to be demonstrated that it has derivatives of all orders, and each derivative is continuous.

From proof A, it can easily be seen that

$$G^{(n)}(x) = \frac{(-1)^{n+1} n! r_0 r_1}{(r_0 + x)^n}$$

is the nth derivative of $G(x)$ with respect to x.
Clearly, each of these derivatives is continuous for $x \geq 0, r_0, r_1 > 0$. Therefore, $G(x)$ is analytic.

B
Solidity code

Note that this code cannot be run by itself. It needs to be added to the testing infrastructure of Kuznetsov [16] and compiled using Foundry [10]

```
    // SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.14;

import "forge-std/Test.sol";
import "./ERC20Mintable.sol";
import "./UniswapV3Pool.Utils.t.sol";

import "../src/interfaces/IUniswapV3Pool.sol";
import "../src/lib/LiquidityMath.sol";
import "../src/lib/TickMath.sol";
import "../src/UniswapV3Factory.sol";
import "../src/UniswapV3Pool.sol";

contract UniswapV3PoolSwapsTest is Test, UniswapV3PoolUtils {
    ERC20Mintable weth;
    ERC20Mintable usdc;
    UniswapV3Factory factory;
    UniswapV3Pool pool;
    event SwapCompleted(int256 amount0Delta, int256 amount1Delta);
    bool transferInMintCallback = true;
    bool transferInSwapCallback = true;
    bytes extra;

    function setUp() public {
        usdc = new ERC20Mintable("USDC", "USDC", 18);
        weth = new ERC20Mintable("Ether", "ETH", 18);
        factory = new UniswapV3Factory();

        extra = encodeExtra(address(weth), address(usdc), address(this));
    }

    function testBuyETHConsecutivePriceRanges4() public {
        uint16[][] memory lowerbounds = [%Liquidity intervals inserted here%];
        uint16[][] memory upperbounds = [%Liquidity intervals inserted here%];
        uint256 amount0 = 0.6 ether;
        uint256 amount1 = 3000 ether;
        uint256 currentPrice = 5000;
        for (uint256 i = 0; i < lowerbounds.length; i++) {
            multiplePriceRanges(
                lowerbounds[i],
                upperbounds[i],
                amount0,
                amount1,
                currentPrice,
                uint24(i)
            );
        }
    }

    function multiplePriceRanges(
        uint16[4] memory lowerbounds,
        uint16[4] memory upperbounds,
        uint256 amount0,
        uint256 amount1,
        uint256 currentPrice,
        uint24 i
    ) public {
        LiquidityRange[] memory liquidityRanges = new LiquidityRange[](4);
        uint256 scaled_amount1;
        liquidityRanges[0] = liquidityRange(
                lowerbounds[0],
                upperbounds[0],
                amount0,
                amount1,
                currentPrice
            );
        for (uint24 j = 0; j < lowerbounds.length; j++) {
            liquidityRanges[j] = liquidityRange(
                lowerbounds[j],
                upperbounds[j],
                amount0,
                amount1,
                currentPrice
            );
        }

        (
            LiquidityRange[] memory liquidity,
            uint256 poolBalance0,
            uint256 poolBalance1
        ) = setupPool(
                PoolParams({
                    balances: [uint256(2.4 ether), 120000 ether],
                    currentPrice: 5000,
                    liquidity: liquidityRanges,
                    transferInMintCallback: true,
                    transferInSwapCallback: true,
                    mintLiqudity: true
                }),
                uint24(3000 + i)
            );

        uint256 swapAmount = 10000 ether; // 10000 USDC
        usdc.mint(address(this), swapAmount);
        usdc.approve(address(this), swapAmount);

        (int256 userBalance0Before, int256 userBalance1Before) = (
            int256(weth.balanceOf(address(this))),
            int256(usdc.balanceOf(address(this)))
        );

        (int256 amount0Delta, int256 amount1Delta) = pool.swap(
            address(this),
            false,
            swapAmount,
            sqrtP(61060),
            extra
        );
        emit SwapCompleted(amount0Delta, amount1Delta);

        // assertEq(
        //     amount0Delta,
        //     -1.806151062659754714 ether,
        //     "invalid ETH out"
        // );
        // assertEq(
        //     amount1Delta,
        //     9938.146841864722991247 ether,
        //      "invalid USDC in"
        // );
    }


    function uniswapV3SwapCallback(
        int256 amount0,
        int256 amount1,
        bytes calldata data
    ) public {
        if (transferInSwapCallback) {
            IUniswapV3Pool.CallbackData memory cbData = abi.decode(
                data,
                (IUniswapV3Pool.CallbackData)
            );

            if (amount0 > 0) {
                IERC20(cbData.token0).transferFrom(
                    cbData.payer,
                    msg.sender,
                    uint256(amount0)
                );
            }

            if (amount1 > 0) {
                IERC20(cbData.token1).transferFrom(
                    cbData.payer,
                    msg.sender,
                    uint256(amount1)
                );
            }
        }
    }

    function uniswapV3MintCallback(
        uint256 amount0,
        uint256 amount1,
        bytes calldata data
    ) public {
        if (transferInMintCallback) {
            IUniswapV3Pool.CallbackData memory cbData = abi.decode(
                data,
                (IUniswapV3Pool.CallbackData)
            );

            IERC20(cbData.token0).transferFrom(
                cbData.payer,
                msg.sender,
                amount0
            );
            IERC20(cbData.token1).transferFrom(
                cbData.payer,
                msg.sender,
                amount1
            );
        }
    }

    ////////////////////////////////////////////////////////////////////////////

    ///INTERNAL

    ////////////////////////////////////////////////////////////////////////////
    function setupPool(
        PoolParams memory params,
        uint24 feeparam
    )
        internal
        returns (
            LiquidityRange[] memory liquidity,
            uint256 poolBalance0,
            uint256 poolBalance1
        )
    {
        weth.mint(address(this), params.balances[0]);
        usdc.mint(address(this), params.balances[1]);

        pool = deployPool(
            factory,
            address(weth),
            address(usdc),
            feeparam,
            params.currentPrice
        );

        if (params.mintLiqudity) {
            weth.approve(address(this), params.balances[0]);
            usdc.approve(address(this), params.balances[1]);

            uint256 poolBalance0Tmp;
            uint256 poolBalance1Tmp;
            for (uint256 i = 0; i < params.liquidity.length; i++) {
                (poolBalance0Tmp, poolBalance1Tmp) = pool.mint(
                    address(this),
                    params.liquidity[i].lowerTick,
                    params.liquidity[i].upperTick,
                    params.liquidity[i].amount,
                    extra
                );
                poolBalance0 += poolBalance0Tmp;
                poolBalance1 += poolBalance1Tmp;
            }
        }

        transferInMintCallback = params.transferInMintCallback;
        transferInSwapCallback = params.transferInSwapCallback;
        liquidity = params.liquidity;
    }
}
```

List of Terms and Acronyms

Glossary

arbitrage The process of taking advantage of price difference in different markets, by buying an asset from the cheaper market and selling it in the more expensive market. 8

Bitcoin The first implementation of a blockchain, proposed by Nakamoto [19]. 3

blockchain A distributed, peer-2-peer, computer network. Often used for the trade of crytocurrencies. 1, 3, 4, 13, 21

Ethereum First and largest blockchain making use of smart contracts (as of 2023). 3, 13

smart contract A self executing computer program that runs based on a set of specified rules. Implemented on a smart blockchain, e.g. Ethereum. Smart contracts are deployed and interacted with through blockchain transactions. 13

Acronyms

AMM Automated Market Maker. 1, 5–7, 13, 21

CFMM Constant Function market Maker. 1, 5, 6

CPMM Constant Product Market Maker. 6, 8, 9, 21

CSMM Constant Sum Market Maker. 6–8

Bibliography

[1] *Solidity Language website*. URL https://docs.soliditylang.org/en/latest/. Accessed July 1st, 2023.

[2] What is crypto minting? how is crypto minting different from crypto mining? 2022. URL https://zipmex.com/learn/crypto-minting/. Accessed July 1st, 2023.

[3] H. Adams, N. Zinsmeister, and D. Robinson. Uniswap v2 Core. 2020. URL https://www.semanticscholar.org/paper/Uniswap-v2-Core-Adams-Zinsmeister/3bf68dddcd4e817e50539a1382da701defef04a0.

[4] G. Angeris and T. Chitra. Improved Price Oracles: Constant Function Market Makers. 2020. doi: https://doi.org/10.2139/ssrn.3636514.

[5] G. Angeris and T. Chitra. An Analysis of Uniswap markets. 2021. URL https://cryptoeconomicsystems.pubpub.org/pub/angeris-uniswap-analysis/release/15.

[6] M. Bartoletti, Y. Chiang, and A. Lluch-Lafuente. A theory of Automated market makers in Defi. 2021. URL https://doi.org/10.46298/lmcs-18(4:12)2022.

[7] M. Bichuch and Z. Feinstein. Axioms for Automated Market Makers: A Mathematical Framework in FinTech and Decentralized Finance. 2022. URL https://arxiv.org/pdf/2210.01227.pdf.

[8] *Curve documentation*. Curve Finance. URL https://resources.curve.fi. Accessed July 1st, 2023.

[9] C. de Vries. A comparison of the limit order book and automated market maker. Master's thesis, 2022. URL https://repository.tudelft.nl/islandora/object/uuid:d8a0e235-5804-4735-87ee-36cf9aa28e0a?collection=education.

[10] *Github - Foundry*. Foundry, 2023. URL https://github.com/foundry-rs/foundry. Accessed June 22th 2023.

[11] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901. doi: 10.1080/14786440109462720.

[12] John Gill. A mathematical note: Convergence of infinite compositions of complex functions. *Comm. Analytic Theory of Continued Fractions*, XIX, 01 2012.

[13] Ellen R Girden. *ANOVA: Repeated measures*. Number 84. Sage, 1992.

[14] Ivan Kuznetsov. Uniswap v3 Development book. 2022. URL https://uniswapv3book.com/. Accessed June 15th, 2023.

[15] R. Keller. Formal verification of parallel programs. *Communications of the ACM*, 1976. URL https://doi.org/10.1145/360248.360251.

[16] Ivan Kuznetsov. GitHub - Jeiwan/uniswapv3-code: Uniswap V3 clone. GitHub, 2023. URL https://github.com/Jeiwan/uniswapv3-code. Accessed may 10th, 2023.

[17] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[18] *Mstable documentation*. Mstable. URL https://docs.mstable.org/. Accessed July 1st, 2023.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[20] *Least squares polynomial fit method.* Numpy. URL `https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html`. Accessed July 1st, 2023.

[21] A. Port and N. Tiruviluamala. Mixing constant sum and constant product market makers. 2022. URL `https://doi.org/10.48550/arXiv.2203.12123`.

[22] C. Robert and G. Casella. *Monte Carlo statistical methods.* Springer Science & Business Media, 2013.

[23] Uniswap. Uniswap V3 whitepaper, 2021. URL `https://uniswap.org/whitepaper-v3.pdf`.

[24] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, and Rui Qin. An overview of smart contract: Architecture, applications, and future trends. pages 108–113, 06 2018. doi: 10.1109/IVS.2018.8500488.