Implementing Automated Market Makers with Constant Ellipse

Yongge Wang
UNC Charlotte

February 17, 2021

Abstract

This paper describe the implementation details of constant ellipse based automated market makers (CoinSwap).

1 Introduction

Decentralized finance (DeFi or open finance) is implemented through smart contracts (DApps) which are stored on a public distributed ledger (such as a blockchain) and can be activated to automate execution of financial instruments and digital assets. The immutable property of blockchains guarantees that these DApps are also tamper-proof and the content could be publicly audited. Recently, Wang [6] proposed a constant ellipse/circle based automated market makers that improves existing schemes such as LMSR [1, 2], LS-LMSR [4], and constant product (e.g., Uniswap [5]) based automated market makers.

2 Constant ellipse/circle automated market makers

Recently, Wang [6] proposed automated market makers based on constant ellipse/circle cost functions. That is, the automated market maker's cost function is defined by

$$C(\mathbf{q}) = \sum_{i=1}^{n}(q_i - a)^2 + b\sum_{i\neq j} q_i q_j \tag{1}$$

where a, b are constants. In constant ellipse/circle automated market makers, the price function for each token is

$$P_i(\mathbf{q}) = \frac{\partial C(\mathbf{q})}{\partial q_i} = 2(q_i - a) + b\sum_{j\neq i} q_j.$$

For automated market makers, we only use the first quadrant of the coordinate plane. By adjusting the parameters a, b in the equation (1), one may keep the cost function to be concave (that is, using the upper-left part of the ellipse/circle) or to be convex (that is, using the lower-left part of the ellipse/circle).

For automated market makers based on the cost function $C(\mathbf{q}) = (x-10^9)^2+(y-10^9)^2$, if the initial deposit to the market is 10^6 coins (if the decimal value for the token is 18, then this is equivalent to 10^{24} token unit) for each token, then the market cost is $C(\mathbf{q}_0) = 2\cdot 10^{12}\cdot 999^2$. At market state $\mathbf{q}_0$, a trader may use 1001002 token B coins to purchase 1000000 token A coins with a resulting market state $\mathbf{q}_1 = (0, 2001002)$ and a resulting market cost $C(\mathbf{q}_1) = C(\mathbf{q}_0)$. This tangent line slope stays in the interval $[-1.002005014, -0.9979989980]$. Note that the boundary price is reached when the balance of one token becomes zero. Assume that the cost function is $r\cdot 10^{14} = (x-10^9)^2 + (y-10^9)^2$ and $x = 0$. Then the price of the first token reaches the highest value of

$$\frac{\partial y}{\partial x} = \frac{10^9}{\sqrt{r\cdot 10^{14} - 10^{18}}} = \frac{100}{\sqrt{r-1000}}$$

The token price fluctuation could be adjusted by revising the value of the circle radius. The following table shows some examples.

value r	minimal price	maximum price
10001	0.01000000000	100.0000000
10010	0.03162277660	31.62277660
10100	0.1000000000	10.00000000
10500	0.2236067977	4.472135956
11000	0.3162277660	3.162277660
15000	0.7071067814	1.414213562
17000	0.8366600265	1.195228609
19000	0.9486832980	1.054092553
19900	0.9949874374	1.005037815
19990	0.9994998752	1.000500375
19999	0.9999499985	1.000050004

3 Implementing constant ellipse based AMM – Approach I

As an example, we use the circle $(x-c)^2+(y-c)^2=r^2$ to show how to establish a token pair swapping market in this section. Specifically, we use $c=10^9$ and $r\cdot 10^{14}=16000\cdot 10^{14}$ (that is, $r=16000$) for illustration purpose in this section. Each token pair market maintains constants λ_0 and λ_1 which are determined at the birth of the market. Furthermore, each token market also maintains a non-negative multiplicative scaling variable μ which is the minimal value so that the following equation holds.

$$(\mu\lambda_0 x_0-10^9)^2+\left(\mu\lambda_1 y_0-10^9\right)^2\le 16000\cdot 10^{14} \tag{2}$$

where $\mu\lambda_0 x_0<10^9$ and $\mu\lambda_0 y_0<10^9$. This ensures that we use the lower-left section of the circle for the automated market. The tangent line's slope at the point (x,y) for the pool circle is

$$\frac{\partial(\lambda_1 y)}{\partial(\lambda_0 x)}=-\frac{10^9-\mu\lambda_0 x}{10^9-\mu\lambda_1 y}$$

Thus for the reserves (x_0,y_0), the (λ_0,λ_1)-weighted relative price of the tokens at this market is

$$P_{y/x}(x_0,y_0)=\frac{\lambda_0(10^9-\mu\lambda_0 x_0)}{\lambda_1(10^9-\mu\lambda_1 y_0)}. \tag{3}$$

That is, at market (x_0,y_0), Δ_x token A coins could be swapped for $\Delta_y=\Delta_x P_{y/x}(x_0,y_0)$ token B coins.

Each token market also maintains a non-negative multiplicative scaling variable μ and a total liquidity supply amount variable Ω. The value of Ω changes each time when a liquidity provider adds liquidity to or removes liquidity from the market. The liquidity value Ω should be defined in such a way that the following conditions are satisfied. Assume that the current market state is $\mathbf{q}=(x,y)$ with a market liquidity value $\Omega_{\mathbf{q}}$. If a customer moves the current market state to a new state $\mathbf{q}'=(e\cdot x,e\cdot y)$, then the new market liquidity value is $\Omega_{\mathbf{q}'}=e\cdot\Omega_{\mathbf{q}}$. This could be achieved in several ways. For example, Uniswap defines $\Omega(x,y)=\sqrt{xy}$. In this section, we use the following approach. Let $\pi(x,y)=ax+by$ for some constants a,b. For a given market state $\mathbf{q}=(x,y)$ with liquidity value Ω, if one moves the market state from $\mathbf{q}=(x,y)$ to $\mathbf{q}'=(e\cdot x,e\cdot y)$, then the mew market liquidity for $\mathbf{q}'$ is defined in such a way that

$$\frac{\Omega'}{\Omega}=\frac{\pi(e\cdot x,e\cdot y)}{\pi(x,y)}=e. \tag{4}$$

The reader should be aware of the fact that when the market moves on, we normally do not have $\Omega(x,y)=\pi(x,y)$. It should also be noted that an alternative function $\pi(x,y)=\sqrt{x^2+y^2}$ could also be used in the equation (4) for certain applications.

Though in several other DEX applications (such as Uniswap), a liquidity provider normally provides equivalent values of token A and token B to the market each time, this is generally not true for CoinSwaps. As an example, assume that the current market state is $\mathbf{q}=(x_0,y_0)$ with a total supply Ω. A liquidity provider can add some

liquidity $(\delta x_0, \delta y_0)$ for some $\delta > 0$ to the market without changing the current (λ_0, λ_1)-weighted relative price $\delta = P_{y/x}(x_0, y_0)$ in (3). That is, we need to keep

$$-P_{y/x}((1+\delta)x_0, (1+\delta)y_0) = \frac{10^9 - \mu'\lambda_0(1+\delta)x_0}{10^9 - \mu'\lambda_1(1+\delta)y_0} = \frac{10^9 - \mu\lambda_0 x_0}{10^9 - \mu\lambda_1 y_0} \tag{5}$$

where $x = x_0 + \Delta_x$, $y = y_0 + \Delta_y$, and $\mu' > \mu$. It is straightforward to show that

$$\mu' = \frac{\mu}{1+\delta}.$$

As an example, assume that when the market is set up, one token A's value is equivalent to three token B's price. That is, we set $\lambda_0 = 3$ and $\lambda_1 = 1$. Now assume at some time with market (x_0, y_0), one token A's value is equivalent to two token B's price. That is,

$$\frac{\partial(\lambda_1 y)}{\partial(\lambda_0 x)} = -\frac{10^9 - 3\mu x_0}{10^9 - \mu y_0} = -\frac{2}{3}$$

which means $y_0 = 4.5x_0 - \frac{10^9}{2\mu}$. A liquidity provider can select a $\delta > 0$ and add $(\delta x_0, \delta y_0)$ tokens to the market. Generally do not have $\delta x_0 = 2\delta y_0$ since $y_0 = 4.5x_0 - \frac{10^9}{2\mu}$ does not guarantee $x_0 = 2y_0$.

3.1 Adding/removing liquidity and swapping

Establishing a token pair market. When a token pair market is not established yet, a liquidity provider can establish the token pair market by depositing x_0 coins of token A and y_0 coins of token B. The assumption is that the market value of x_0 coins of token A is equivalent to the market value of y_0 coins of token B. The token pair constant variables λ_0, λ_1 are defined as $\lambda_0 x_0 \sim \lambda_1 y_0$. Note that λ_0, λ_1 are extensively used by the swapping algorithm. Thus it is better to have simple/smaller λ_0, λ_1. This could be calculated by the smart contract using continuous fraction. It is recommended that the liquidity providers should provide the values of λ_0, λ_1 when establishing a new market (or by vote from the community). Let μ be the non-negative number such that the equation (2) holds. Let the liquidity of the token pair market be defined as $\Omega_0 = 10^{18} \cdot \frac{\lambda_0 x_0 + \lambda_1 y_0}{2}$. As a return, the liquidity provider receives Ω_0 liquidity coins for this token pair market. In our implementation, it is required that each liquidity provider should put at least one coin for each token in the market. That is, for an ERC token with 10^{18} decimals, the liquidity provider should put at least 10^{18} units of token A and 10^{18} units of token B. This requirement insures that $\mu \leq 10^9$.

Data representations: We have the following conventions:

- x, y are represented as `uint96` though the right-most 18 digits are considered as decimals. For 96-binary bits, one can represent numbers small than 79228162514264337593543950335. So with 18 decimal ERC20 tokens, the liquidity could contain 79 billion coins of each token which are sufficient for most tokens.
- Liquidity value Ω: this is the inherited value `totalSupply` from the ERC20 and is of type `uint256`.
- `rSquare` is of type `uint16` (where $r^2 = (10000 + \texttt{rSquare}) \cdot 10^{14}$). This allows us the let `rSquare` takes values from 1 to 9999. That is, we can select the price fluctuation from $[0.01, 0.9999499985]$. In this paper, we use $\texttt{rSquare} = 6000$ as an example. That is, $r^2 = 16000 \cdot 10^{14}$ and the weighted price of a token (relative the weighted price of the other token) is allowed to change from 0.7745966692 to 1.290994449. Note that we assume the weighted price of one token relative the weighted price of the other token is one at the market set up time.
- λ_0, λ_1 are of type `uint16`. This implies that both λ are smaller than 65535. In order to support tokens with smaller decimals (less than 18), we also need to store $D_0 = 10^{18-d_0}$ and $D_1 = 10^{18-d_1}$ where $d_0, d_0 \leq 18$ are decimals of the two tokens respectively. We use 56-bits to store D_0 and 56-bits to store D_1.
- μ is of type `uint56` where the right-most 7 digits are fractional part. That is, $\mu \leq 7205759403.7927936$

As a summary, a 224-bits variable is used to store these parameters related to the circle as follows

ICO(8)	D_0 (56)	D_1 (56)	rSquare (16)	λ_0 (16)	λ_1 (16)	μ (56)

In the CoinSwap library, the function `getReservesAndmu` returns the value $D_0\lambda_0\mu \| D_1\lambda_1\mu$ in a 256-bit variable. That is, each $D_i\lambda_i\mu$ takes 128-bits. It is noted that λ is 16-bits, μ is 56-bits. Thus, to avoid overflow, it is required that $D_i < 2^{56} = 72057594037927936$. In other words, we have $18 - d_i \leq 16$ and the underlying tokens must have at least two decimals.

Remarks: It is required that $\mu\lambda x < 10^9$ and μ has at most 7 fractional digits (see the following data type for μ). Thus we need to have $\lambda x < 10^{16}$. Furthermore, the maximal value for λ is $2^{16} - 1$. If μ takes the minimal value and λ takes the maximal value, the maximal value that x can take is $5^{16} = 152587890625$ which is stall smaller than the supposed maximal value 79228162514. In other words, No overflow should happen if we require the balances for each token is smaller than the maximal value of 96-bits.

After the market is set up, the values of λ_0, λ_1 are fixed for the duration of the market. During the computation, we need to compute $\mu\lambda_0 x$, $(\mu\lambda_0 x - 10^9)^2$, $\mu\lambda_1 y$, and $(\mu\lambda_1 y - 10^9)^2$. Since we require $\mu\lambda_0 x < 10^9$ and $\mu\lambda_1 y < 10^9$. This means that each of $\mu\lambda_0 x$ and $\mu\lambda_1 y$ can be represented by a $25 + 9 = 34$ digits (without decimal point). That is, $\mu\lambda_0 x$ and $\mu\lambda_1 y$ can be represented with 113-bit binary strings (note that 25 decimal parts can be represented with a 83-bits binary string). Thus the values $(\mu\lambda_0 x - 10^9)^2$ and $(\mu\lambda_1 y - 10^9)^2$ could be held in a 256-bit binary string.

Since it is challenging to support float computation in Solidity, for easy calculation of (2), we can multiply 10^{36} to both side of the equation so that x, y can then be treated as `uint`. That is,

$$(\mu\lambda_0 x_0 \cdot 10^{18} - 10^{27})^2 + \left(\mu\lambda_1 y_0 \cdot 10^{18} - 10^{27}\right)^2 = 16000 \cdot 10^{50}$$

Adding liquidity. Assuming that the current market status is $\mathbf{q}_0 = (x_0, y_0)$ and the total supply of the token pair market is Ω_0. If a liquidity provider would like to add Δ_x token A coins to the market, then she/he also needs to add $\Delta_y = \frac{\Delta_x y_0}{x_0}$ token B coins to the market. The new total supply of the token pair market becomes

$$\Omega_1 = \frac{\Omega_0(\lambda_0 x_1 + \lambda_1 y_1)}{\lambda_0 x_0 + \lambda_1 y_0}$$

where $x_1 = x_0 + \Delta_x$ and $y_1 = y_0 + \Delta_0$. As a return, the liquidity provider will receive $\Omega_1 - \Omega_0$ liquidity coins for this token pair market. The variable μ is adjusted in such a way that the following equation holds

$$(\mu\lambda_0 x_1 - 10^9)^2 + \left(\mu\lambda_1 y_1 - 10^9\right)^2 = 16000 \cdot 10^{14} \tag{6}$$

with $\mu\lambda_0 x_1 < 10^9$ and $\mu\lambda_1 y_1 < 10^9$.

The minted liquidity and the adjusted μ could be computed alternatively as follows. Assume that $x_0 \neq 0$. Then we have $\mu_0\lambda_0 x_0 = \mu\lambda_0 x_1$. That is, $\mu = \frac{\mu_0 x_0}{x_1}$ and $\Omega_1 = \frac{\Omega_0 x_1}{x_0}$.

Removing liquidity. Assuming that the current market status is $\mathbf{q}_0 = (x_0, y_0)$ and the total supply of the token pair market is Ω_0. If a liquidity provider would like to convert Δ_Ω liquidity coins back to native coins, the liquidity provider will receive (Δ_x, Δ_y) tokens where $\Delta_x = \frac{\Delta_\Omega x_0}{\Omega_0}$ and $\Delta_y = \frac{y_0 \Delta_\Omega}{\Omega_0}$. The new total supply of the token pair market becomes $\Omega = \Omega_0 - \Delta_\Omega$. The multiplicative scaling variable μ is adjusted in such a way the equation (6) holds for $x_1 = x_0 - \Delta_x$ and $y_1 = y_0 - \Delta_y$ with $\mu\lambda_0 x_1 < 10^9$ and $\mu\lambda_1 y_1 < 10^9$.

Swapping. Assuming that the current market status is $\mathbf{q}_0 = (x_0, y_0)$ and the multiplicative scaling variable is μ. For each transactions, the customer should pay 0.03% transaction fee. Thus if a customer submits Δ_x token A coins to the market, the customer receives Δ_y token B coins such that

$$\left(\mu\lambda_0(x_0 + 0.997\Delta_x) - 10^9\right)^2 + \left(\mu\lambda_1(y_0 - \Delta_y) - 10^9\right)^2 \leq \left(\mu\lambda_0 x_0 - 10^9\right)^2 + \left(\mu\lambda_1 y_0 - 10^9\right)^2 \tag{7}$$

where $\mu\lambda_0(x_0 + 0.997\Delta_x) < 10^9$. Note that equation (7) is equivalent to the following equation.

$$\left(10^{25}\mu\lambda_0(10^3 \cdot x_0 + 997\Delta_x) - 10^{37}\right)^2 + \left(10^{28}\mu\lambda_1(y_0 - \Delta_y) - 10^{37}\right)^2 \leq \left(10^{28}\mu\lambda_0 x_0 - 10^{37}\right)^2 + \left(10^{28}\mu\lambda_1 y_0 - 10^{37}\right)^2 \tag{8}$$

where we try to convert the fractional parts of μ and x_0, y_0 into integer parts. That is, μ has 7 fractional digits and x_0, y_0 have 18 fractional digits. In order to compute the value Δ_y using the equation (8), let

$$r_0 = \left(10^{28}\mu\lambda_0 x_0 - 10^{37}\right)^2 + \left(10^{25}\mu\lambda_1 y_0 - 10^{37}\right)^2 - \left(10^{25}\mu\lambda_0(1000 \cdot x_0 + 997\Delta_x) - 10^{37}\right)^2.$$

Then we need to have

$$10^{37} - 10^{28}\mu\lambda_1(y_0 - \Delta_y) \leq \sqrt{r_0}.$$

That is,

$$10^{18}\Delta_y \leq \frac{10^{28}\mu\lambda_1 y_0 + \sqrt{r_0} - 10^{37}}{10^3 \cdot (10^7\mu)\lambda_1}.$$

By the discussion in the preceding paragraphs, $10^{25}\mu\lambda_0 x_0$ and $10^{25}\mu\lambda_1 y_0$ could be represented using 113-bit binary strings. Thus $10^{28}\mu\lambda_0 x_0$ and $10^{28}\mu\lambda_1 y_0$ could be represented using 123-bit binary strings. On the other hand, 10^{37} could also be represented by a 123-bit binary string. In a summary, r_0 could be represented by a 247-bit binary string. Thus in order to get 10^{18} fractional digit accuracy for Δ_y, it is sufficient for $\sqrt{r_0}$ to have accuracy until the decimal point.

Given Δ_y, in order to compute Δ_x, we have the following formula:

$$r_1 = \left(10^{28}\mu\lambda_0 x_0 - 10^{37}\right)^2 + \left(10^{28}\mu\lambda_1 y_0 - 10^{37}\right)^2 - \left(10^{28}\mu\lambda_1(y_0 - \Delta_y) - 10^{37}\right)^2.$$

Then we need to have

$$10^{37} - 10^{25}\mu\lambda_0(10^3 x_0 + 997\Delta_x) \leq \sqrt{r_1}.$$

That is,

$$10^{18} \cdot \Delta_x \geq \frac{10^{37} - \sqrt{r_1} - 10^{28}\mu\lambda_0 x_0}{997 \cdot (10^7\mu)\lambda_0}.$$

Similar analysis as in the preceding paragraphs shows that, r_1 has at most 247-bits and, in order to get 10^{18} fractional digit accuracy for Δ_x, it is sufficient for $\sqrt{r_1}$ to have accuracy until the decimal point.

3.2 Protocol fees

For each transaction, traders pay 0.30% fees on all traders. The system collects 0.05% protocol fee (included in the 0.30% transaction fee) when protocol fee is turned on. If this protocol fee is collected each time when a transaction is done, it would cost too much gas fee. Thus we adopt the mechanism that was taken by Uniswap that this fee is only calculated when liquidity is deposited or withdrawn or if a command for calculating protocol fee is received. Assume that the current market state is (x_0, y_0), the total liquidity supply amount is Ω, and the current multiplicative scaling variable is μ_0. Let μ be the number such that equation (2) holds with $\mu\lambda_0 x_0 < 10^9$ and $\mu\lambda_1 y_0 < 10^9$. By equation (2), the accumulated transaction fees are $\frac{\mu_0 x_0 - \mu x_0}{\mu_0}$ coins of token A and $\frac{\mu_0 y_0 - \mu y_0}{\mu_0}$ coins of token B. Thus we need to calculate the accumulated protocol fee Δ_Ω such that

$$\frac{\Delta_\Omega}{\Omega + \Delta_\Omega} = \frac{1}{6}\frac{\frac{(\mu_0 - \mu)\lambda_0 x_0}{\mu_0} + \frac{(\mu_0 - \mu)\lambda_1 y_0}{\mu_0}}{\lambda_0 x_0 + \lambda_1 y_0} = \frac{\mu_0 - \mu}{6\mu_0}.$$

That is,

$$\Delta_\Omega = \frac{\Omega(\mu_0 - \mu)}{5\mu_0 + \mu}. \tag{9}$$

Transfer the liquidity amount Δ_Ω to the given protocol fee address and increase the total supply liquidity to $\Omega + \Delta_\Omega$. The new multiplicative scaling variable becomes μ.

3.3 Cumulative price

CoinSwap employs the cumulative price mechanism for price oracle services and the smart contract records the cumulative price ratio $P_{y/x}$ of the equation (3). Let $t_0, t_1, \cdots, t_m$ be the price checkpoints where the corresponding reserves are $(x_0, y_0), \cdots, (x_m, y_m)$. Then the smart contract records cumulative prices at time t_m as the time-and-(λ_0, λ_1)-weighted average of the prices at these checkpoint times:

$$p_m = \sum_{i=1}^{m} \frac{\Delta_i \lambda_0 (10^9 - \mu\lambda_0 x_i)}{\lambda_1 (10^9 - \mu\lambda_1 y_i)} \tag{10}$$

where $\Delta_i = t_i - t_{i-1}$. The (λ_0, λ_1)-weighted price for the time period $[t_{m_1}, t_{m_2}]$ is then computed as

$$p_{m_1,m_2} = \sum_{i=m_1+1}^{m_2} \frac{\Delta_i \lambda_0 (10^9 - \mu\lambda_0 x_i)}{\Delta_{m_1,m_2} \lambda_1 (10^9 - \mu\lambda_1 y_i)} \tag{11}$$

where $\Delta_{m_1,m_2} = t_{m_2} - t_{m_1}$.

3.4 Calculation of the multiplicative variable

In the algorithms of Sections 3.1 and 3.2, given (x_0, y_0), one needs to find the multiplicative scaling variable μ such that the equation (2) holds. To reduce the computation on the blockchain and to reduce the gas fee cost, the calculation of μ could be done by the client (e.g., traders, liquidity providers, etc.). That is, the automated market smart contract only needs to verify the correctness of the value μ. For the circle

$$(\mu\lambda_0 x_0 \cdot 10^{25} - 10^{34})^2 + \left(\mu\lambda_1 y_0 \cdot 10^{25} - 10^{34}\right)^2 = r \cdot 10^{64} \tag{12}$$

and a point (x_0, y_0) with $x_0 + y_0 \neq 0$, the value of μ is computed as follows. First the equation (12) can be converted to the following equation

$$10^{36} \left(x_0^2\lambda_0^2 + y_0^2\lambda_1^2\right) (10^7\mu)^2 - 2 \cdot 10^{52} \cdot (x_0\lambda_0 + y_0\lambda_1) (10^7\mu) + 2 \cdot 10^{68} - r \cdot 10^{64} = 0.$$

That is

$$10^7 \cdot \mu = \left\lceil 10^{32} \cdot \frac{10^{20} (x_0\lambda_0 + y_0\lambda_1) \pm \sqrt{10^{40} (x_0\lambda_0 + y_0\lambda_1)^2 - 10^{36}(20000 - r) \cdot (x_0^2\lambda_0^2 + y_0^2\lambda_1^2)}}{10^{36}(x_0^2\lambda_0^2 + y_0^2\lambda_1^2)} \right\rceil. \tag{13}$$

For the two solutions in equation (13), one uses the μ such that $\mu\lambda_0 x_0 < 10^9$ and $\mu\lambda_1 y_0 < 10^9$. That is, $\lambda_0 x_0 < 10^{16}$ and $\lambda_1 y_0 < 10^{16}$. That is, the value inside $\sqrt{\cdot}$ has no fractional digits and is smaller than $10^{5+2(16+18)} = 10^{73}$ which requires at most 245-bits. Since the denominator in (13) has at least 36 digits and there is a constant scale 10^{32} in the numerator, In order to keep all digits in $10^7 \cdot \mu$ significant, we need the square root to be approximated at least at $O(10^3)$. This is easily achieved using Newton's approach.

3.5 Some practical considerations

Calculation of λ_0 and λ_1: The values of λ_0 and λ_1 are determined by the liquidity that the user decides to put in. For example, if the user decides to put in $\frac{x}{10^{18}}$ shares of token A and $\frac{y}{10^{18}}$ shares of token B where the market values of these A tokens is equivalent to B tokens. The user interface should help the user to calculate λ_0 and λ_1 so that $\lambda_0 x \sim \lambda_1 y$ where λ_0, λ_1 are 16 bits. Without loss of generality, we may assume that $x \geq y$. Let $x = x_0 x_1 \cdots x_m . x_{m+1} \cdots$ and $y = y_0 y_1 \cdots y_m . y_{m+1} \cdots$ are binary representations of x and y. It is noted that y_0 may be zero in some cases. The system should take $\lambda_x = y_0 \cdots y_{16}$ and $\lambda_y = x_0 \cdots x_{16}$.
Circle parameters: The default parameter for the circle is $\texttt{rSqaure} = 16000 \times 10^{14}$. In the router smart contracts, a user needs to provide the

$$\texttt{circle} = \texttt{rSquare} \cdot 2^{32} + \lambda_0 \cdot 2^{16} + \lambda_1$$

parameter for adding liquidity. For a pair of tokens A and B, the token with smaller address is defined as the token0. However, these information should be transparent to the end users. Thus the system should compare the address of token A and token B. If token A address is less than token B, the system should set $\lambda_0 = \lambda_x$. Otherwise, it should set $\lambda_0 = \lambda_y$. After the value of `circle` is calculated, the system can automatically use this value for adding liquidity.
It should be noted that the first transaction on each block needs to invoke the cumulative price update process. Thus it will pay a slightly high gas fee. The `revisemu` process is only invoked in the functions `mint`, `burn`, and `skim`. After each swap transaction, the circle radius may decrease. Thus unless these functions are called (e.g., a `skim` function call without adding/removing liquidity), the swap transactions are carried out on a circle with smaller radius which means the price fluctuation is slightly larger–but the difference may not be quite observable).
It is noted that a user may carry out a sequence of actions in one transaction: make a swap to reduce the number of one token; add the liquidity based on the new coin ratios; make another swap to cancel the first swap impact. Can a use make a profit from this?

4 Implementing constant ellipse based AMM – Approach II

Section 3 discusses how to implement constant ellipse based AMM using the circle $(x-c)^2+(y-c)^2=r^2$ with a multiplicative scaling variable μ. This section provides an implementation without the scaling variable μ. The cost function for the AMM is defined as $C(\mathbf{q})=(x-10^9)^2+(y-10^9)^2$ with $x,y<10^9$. Each token pair market maintains a constant λ_0 which is determined at the birth of the market and maintains a total liquidity supply amount variable Ω. For the function $\pi(x,y)$ that is used to infer market liquidity values, we use $\pi(x,y)=\frac{x+y\lambda_0}{2}$ as in Section 3. As we have mentioned in Section 3, one may also use $\pi(x,y)=\sqrt{x^2+y^2}$ or $\pi(x,y)=\sqrt{xy}$. But then the calculation could be more complicated.

Establishing a token pair market. When a token pair market is not established yet, a liquidity provider can establish the token pair market by depositing x_0 coins of token A and y_0 coins of token B. The assumption is that the market value of x_0 coins of token A is equivalent to the market value of y_0 coins of token B. The token pair constant variable λ_0 is defined as $\lambda_0 x_0\sim\lambda_1 y_0$. That is, at the birth of the token pair market, one coin of token A is worthy of λ_0 coins of token B. Let the total supply of the token pair market be $\Omega_0=10^{18}\cdot\frac{\lambda_0x_0+\lambda_1y_0}{2}$. As a return, the liquidity provider receives Ω_0 liquidity coins for this token pair market.

Adding liquidity. Assuming that the current market status is $\mathbf{q}_0=(x_0,y_0)$ and the total supply of the token pair market is Ω_0. If a liquidity provider would like to add Δ_x token A coins to the market, then she/he also needs to add $\Delta_y=\frac{\Delta_x y_0}{x_0}$ token B coins to the market. The new total supply of the token pair market becomes

$$\Omega_1=\frac{\Omega_0(\lambda_0x_1+\lambda_1y_1)}{\lambda_0x_0+\lambda_1y_0}$$

where $x_1=x_0+\Delta_x$ and $y_1=y_0+\Delta_0$. As a return, the liquidity provider will receive $\Omega_1-\Omega_0$ liquidity coins for this token pair market.

Removing liquidity. Assuming that the current market status is $\mathbf{q}_0=(x_0,y_0)$ and the total supply of the token pair market is Ω_0. If a liquidity provider would like to convert Δ_Ω liquidity coins back to native coins, the liquidity provider will receive (Δ_x,Δ_y) tokens where $\Delta_x=\frac{\Delta_\Omega x_0}{\Omega_0}$ and $\Delta_y=\frac{\Delta_\Omega y_0}{\Omega_0}$. The new total supply of the token pair market becomes $\Omega=\Omega_0-\Delta_\Omega$.

Swapping. Assuming that the current market status is $\mathbf{q}_0=(x_0,y_0)$. For each transaction, the customer should pay 0.03% transaction fee. Thus if a customer submits Δ_x token A coins to the market, the customer receives Δ_y token B coins such that

$$\left(\lambda_0(x_0+0.997\Delta_x)-10^9\right)^2+\left(\lambda_1(y_0-\Delta_y)-10^9\right)^2\le\left(\lambda_0x_0-10^9\right)^2+\left(\lambda_1y_0-10^9\right)^2 \tag{14}$$

where $\lambda_0(y_0-\Delta_y)<10^9$.

Protocol fee calculation. Protocol fees should be be transferred each tie when a swap transaction is made. Otherwise, we need to store value $\left(\lambda_0x_0-10^9\right)^2+\left(\lambda_1y_0-10^9\right)^2$ of the last time that the protocol fee was calculated and the calculation could be quite complicated. Assuming that the current market status is $\mathbf{q}_0=(x_0,y_0)$ and the total supply of the token pair market is Ω_0. If a client made a transaction of Δ_x, then the protocol fee is $0.003\Delta_x$. Then the protocol fee is

$$\Omega=\frac{0.003\Omega_0\lambda_0\Delta_x}{\lambda_0x_0+\lambda_1y_0}$$

5 Square roots with Babylonian/Newton method

For a given $a>0$, we need to know $x=\sqrt{a}$. We start with a guess $x_1>0$ and compute the sequences

$$x_{n+1}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right). \tag{15}$$

In other words, we use the arithmetic mean to approximate the geometric mean. This approach was used by the ancient Babylonians circa 1000 BC and is equivalent to the Newton's method for finding the root of $f(x)=0$ from a guess

x_n by approximating $f(x)$ as its tangent line $f(x_n) + f'(x_n)(x - x_n)$. That is,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{16}$$

is a better approximation than x_n. If we replace $f(x)$ with $x^2 - a$, we get (15). A simple analysis shows that no matter whether $x_1 > \sqrt{a}$ or not, we have $x_2 > \sqrt{a}$. Furthermore, we have $x_2 > x_3 > \cdots > x_{n+1} > \sqrt{a}$. By using Taylor-expansion, it can been shown that the error roughly squares (i.e., halves) after each iteration. In other words, the number of accurate digits approximately doubles on each iteration. However, this convergence property is not preserved if rounding happens. In most Solidity implementation of the Babylonian method, one round x_n to the integer digit each time. Thus the convergence may take linear time. The best practice is to have at least one more digit than the required accuracy. The other challenge is that Solidity only supports `uint256`. For an integer of 256 bits, the operation a/x_n only returns the integer part. That is, we cannot calculate fractional part of the square roots directly.

In the following, we propose an approach to deal with this situation. Given $x_n = x'_n + \frac{x''_n}{10^l}$ where x'_n and x''_n are integers. Let $q = a/x'_n$ and $r = a\%x'_n$. That is, $a = qx'_n + r$. Then we should have

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{q(x'_n + \frac{x''_n}{10^l}) - \frac{qx''_n}{10^l} + r}{x'_n + \frac{x''_n}{10^l}}\right) = \frac{1}{2}\left(x_n + q + \frac{r - \frac{qx''_n}{10^l}}{x'_n + \frac{x''_n}{10^l}}\right) \tag{17}$$

In other words, we first calculate q using the `uint256` division operator in Solidity. Then we compute $\frac{10^l r - qx''_n}{10^l x'_n + x''_n}$. Let $|x|_b$ to denote the number of binary bits of an integer x. In practice, we may assume that $|a|_2 \leq 256$ and take $128 \geq |x'_n|_b \geq \frac{|a|_2}{2} - 8$. Then we have

- $|10^l r|_b \leq |x'_n|_b + |x''_n|_b \leq 128 + |x''_n|_b$.
- $|q|_b + |x'_n|_b \leq |a|_2$. That is, $|qx''_n|_b \leq |q|_b + |x''_n|_b \leq |a|_2 - |x'_n|_b + |x''_n|_b \leq \frac{|a|_2}{2} + 8 + |x''_n|_b \leq 136 + |x''_n|_b$.

Now assume that we want to have 50-bit fractional precision. Then we can set $||x''_n||_b = 51$. Then we will be able to compute $\frac{10^l r - qx''_n}{10^l x'_n + x''_n}$ with at least 50 bits precision by computing $2^{51} \cdot (10^l r - qx''_n)/(10^l x'_n + x''_n)$ using Solidity division.

5.1 Montgomery algorithm for 512-bit division

Section 5 presents an algorithm to compute square roots of a `uint256` with 50-bits fractional accuracy with Solidity division. For the calculation of μ in (13), we need to compute the square root of a 250-bits integer with at least 33 accurate fractional digits (that is, 100 binary fractional bits). This can be achieved by computing the square root of $a \cdot 10^{66}$ without accurate digit until decimal point. In other words, we need to calculate $\frac{a*10^{66}}{x_n}$ for the iteration (15) where a is 250-bits and $a * 10^{66}$ can only be represented as a 512-bit integer. We employed the division algorithm described in [3]. The reader is referred to [3] for algorithm details. The core part of the algorithm is based on Newton's method (16) with $f(x) = \frac{1}{x} - a$. The iteration will be able to compute the reciprocal $\frac{1}{a}$ for a. Since $f'(x) = -\frac{1}{x^2}$, (16) could be re-written as

$$x_{n+1} = x_n - \frac{\frac{1}{x_n} - a}{-\frac{1}{x_n^2}} = x_n + (x_n - dx_n^2) = x_n(2 - ax_n).$$

It is straightforward to show that the iteration has the quadratic convergence also (the error rate squares after each iteration).

6 Smart contracts for CoinSwap based on constant circle model

We have implemented a proof of concept constant circle automated market (called CoinSwap) using the approach I in Section 3. The implementation is based on the Uniswap V2 architecture. The `CoinSwapFactory.sol` is similar to the Uniswap Factory contracts. The major new component is the `CoinSwapPair.sol` contract. `CoinSwapPair.sol` contains the following storage variables

- address factory: The address who calls constructor factory = msg.sender.
- address token0 and address token1.
- uint public priceCumulative is for the cumulative price storage.
- uint public circleData represents the following value

$$\texttt{ICO} \cdot 2^{216} + D_0 \cdot 2^{160} + D_1 \cdot 2^{104} + r \cdot 2^{88} + \lambda_0 \cdot 2^{72} + \lambda_1 \cdot 2^{56} + \mu$$

  where D_0 and D_1 are 56-bits each, ICO is 8-bits, r is 16-bits, λ_0, λ_1 are 16-bits each and μ is 56-bits. The value $\texttt{ICO} = 0$ for regular pairs and $\texttt{ICO} > 0$ for initial coin offers (that is, the purpose of the pair is to sell ICO tokens instead of regular AMM trading). These parameters define the constant circle

$$(x - 10^9)^2 + (y - 10^9)^2 = (10000 + r) \cdot 10^{14}.$$

  Furthermore, $\lambda_0 x_0 \sim \lambda_1 y_0$ and μ is the multiplicative scaling variable. $\mu\lambda_0$ and $\mu\lambda_1$ are stored there to save gas cost for the multiplication during swapping process.

- uint224 reserve represents the following value

$$\texttt{reserve0} * 2^{128} + \texttt{reserve1} * 2^{32} + \texttt{blockTimestampLast}$$

  where reserve0 and reserve1 are 96-bits each and blockTimestampLast is 32-bits.

CoinSwapPair.sol contains the following major functions:

- constructor() sets the factory as the message sender.
- initialize(token0, token1, circle): this function can only be called by factory and it initializes the two token addresses and sets the circle parameter within the circleData as

$$\texttt{circle} = \texttt{ICO} \cdot 2^{216} + D_0 \cdot 2^{160} + D_1 \cdot 2^{104} + r \cdot 2^{88} + \lambda_0 \cdot 2^{72} + \lambda_1 \cdot 2^{56}.$$

- ICOmanage(unlocked, circleData). The pair owner may be able to reset of ICO token selling prices. In order to call this function, it is required to have $\texttt{ICO} > 0$. That is, this function cannot be called for regular token pairs with $\texttt{ICO} = 0$.
- revisemu(balance0, balance1,μ) function re-computes the new value μ and updates it.
- getReserves() returns reserve0, reserve1, blockTimestampLast.
- _update(balance0, balance1, reserve0, reserve1): update cumulative price, let reserve0 = balance0, reserve0 = balance0.
- _mintFee(reserve0, reserve1): calculate and transfer protocol fee to feeTo.
- mint(to): If this is for establishing for a token pair market (that is, totalSupply = 0), then it calls revisemu() to calculate the value of μ, $\mu\lambda_0$,$\mu\lambda_1$, transfers the liquidity Ω_0 to the liquidity provider and calls the _update(). If this is for adding liquidity, it first calls _mintFee to transfer the accumulated protocol fee to the address feeTo. Then it revisemu() and transfers the liquidity $\Omega_1 - \Omega_0$ to the liquidity provider and calls the _update().
- burn(to) first calls _mintFee to transfer the accumulated protocol fee to feeTo. Then it removes the liquidity, revisemu(), and calls _update().
- swap(amount0Out, amount1Out, to, calldata data): checks whether equation (8) is satisfied. If the condition is satisfied, do the swap transaction and call _update()

Table 1: Gas cost Uniswap V2/Coinswap with liquidity size (40000000,10000000)

function	mine()	swap()	swap()[1st]	addΩ	removeΩ	add ETH	full removal	partial removal
Uniswap V2	141106	89894	101910	216512	140613	223074	123339	180355
Uniswap V2O	132410	88224	100051	207368	97319	213930	122061	137061
CoinSwap	109722	89348	96294	185442	67127	192027	98805	144283
Gas Saving	22.24%	0.61%	5.51%	14.35%	31.92%	13.92%	19.89%	20.00%

7 Gas cost and comparison

We compare the gas cost against Uniswap. During the implementation of CoinSwap, we find out that some of the optimization techniques that we used in Coinswap may be used to reduce the gas cost in Uniswap. Thus we compare the gas cost for Uniswap V2, our optimized version of Uniswap V2, and Coinswap. Table 1. In a summary, CoinSwap has an average 15% gas-saving over Uniswap V2.

References

[1] R. Hanson. Combinatorial information market design. *Information Systems Frontiers*, 5(1):107–119, 2003.

[2] R. Hanson. Logarithmic markets coring rules for modular combinatorial information aggregation. *The Journal of Prediction Markets*, 1(1):3–15, 2007.

[3] E.W. Mayer. Efficient long division via montgomery multiply. *arXiv preprint arXiv:1303.0328*, 2013.

[4] A. Othman, D.M. Pennock, D.M. Reeves, and T. Sandholm. A practical liquidity-sensitive automated market maker. *ACM Tran. Economics and Computation (TEAC)*, 1(3):1–25, 2013.

[5] Uniswap. Uniswap v2 core, March 2020. `https://uniswap.org/whitepaper.pdf`.

[6] Yongge Wang. Automated market makers for decentralized finance (defi). *arXiv preprint arXiv:2009.01676*, 2020.