Generalized CPMM

Unlike other Automated Market Makers, our Constant-Product AMM can handle multiple tokens simultaneously, whether their amounts are specified or not. This extends to LP tokens as well, which means we don't differentiate between LP join/exit and swap actions.

This article delves into the advanced workings of constant-product automated market makers (AMMs) with a focus on understanding the calculations involved. The explanations are provided with the presumption that the reader is already familiar with the basic concept of an CPMM.

Mathematical framework

Imagine that we have a pool consisting of N tokens. We can represent the states of this pool as a point within a vector space that has (N+1) dimensions. Specifically,(D , x 0 , x 1 , x 2 , . . .) (D, x_0, x_1, x_2, ...) (D ,

x 0 ,

x 1 ,

x 2 ,

...) , where eachx i x_i x i represents a token balance in the pool, and D represents the invariant. changes in invariant is credited as changes in LP token.

Any interaction with the pool, which we term as an "operation", will result in a change in the state of the pool, moving it from one point to another within this vector space. However, not every point within this space is legitimate. Only those points that satisfy the pool's equation can be considered as valid states. Therefore, any operation must result in a valid state.

Mathematically, this condition can be expressed as:∏ x w D (Σ w) = 1 \frac{\prod x^w}{D^{(\Sigma w)}} = 1 D (Σ w)

∏

x w

= 1

This implies that a valid state can be calculated if a user specifies the desired change (delta) for certain dimensions (tokens), leaving at least one dimension unknown. The core of this article revolves around figuring out these valid states, filling in the unknown dimensions based on the known ones.

The state of the pool before an operation is represented as vectora⃗ \vec{a} a , while the state after the operation is denoted as vectorb⃗ \vec{b} b .

a⃗ \vec{a} a is fully known, and our goal is to calculate the values forb⃗ \vec{b} b .

Calculating unknown dimensions in a zero-fee scenario

The computation becomes straightforward when there is no fee. Given the equation∏ x w D (Σ w) = 1 \frac{\prod x^w} {D^{(\Sigma w)}} = 1 D (Σ w)

∏

x w

= 1 , we can simplify it to∏ x w = 1 \prod x^w = 1 ∏

x w

= 1 considering D as just another token with its weight being the negative sum of other tokens' weights.

The product of the ratio of balance changes raised to their respective weights∏ (b i a i) w i \prod{(\frac{b_i}{a_i})}^{w_i} ∏

(a i

b i) w i has to remain 1, as is the case for botha⃗ \vec{a} a andb⃗ \vec{b} b .∏ a i w i = ∏ b i w i = 1 \prod {a_i}^{w_i} = \prod {b_i}^{w_i} = 1 ∏

a i w i

= ∏

b i w i

= 1 . This allows us to think in terms of balance change ratios.

Within this structure, we can separate the unknown ratios from the known ones as:

∏ u n k n o w n (b i a i) w i × ∏ k n o w n (b i a i) w i = 1 \prod_{unknown} {(\frac{b_i}{a_i})}^{w_i} \times \prod_{known} {(\frac{b_i}{a_i})}^{w_i}=1 ∏ u nkn o w n

(a i

b i) w i

× ∏ kn o w n

(a i

b i) w i

= 1

Assuming all the unknown ratios to be equal (b u n k n o w n a u n k n o w n \frac{b_{unknown}}{a_{unknown}} a u nkn o w n

b u nkn o w n)

∏ u n k n o w n (b i a i) w i = (b u n k n o w n a u n k n o w n) (∑ u n k n o w n w i) \prod_{unknown} {(\frac{b_i}{a_i})}^{w_i}=(\frac{b_{unknown}}{a_{unknown}})^{(\sum_{unknown} w_i)} ∏ u nkn o w n

(a i

b i) w i

= (a u nkn o w n

b u nkn o w n) (∑ u nkn o w n

w i)

we can derive them with ease:

(b u n k n o w n a u n k n o w n) = (∏ k n o w n (b i a i) w i) (− 1 ∑ u n k n o w n w i) (\frac{b_{unknown}}{a_{unknown}}) = {(\prod_{known} {(\frac{b_i}{a_i})^{w_i}})}^{(-\frac{1}{\sum_{unknown} w_i})} (a u nkn o w n

b u nkn o w n)

= (∏ kn o w n

(a i

b i) w i) (− ∑ u nkn o w n

w i

1)

Thus we can calculate b.

Calculating unknown dimensions with fees

To incorporate fees into the calculation of unknown dimensions, we have to adapt our approach. Unlike the case without fees, the computation becomes a bit more intricate.

When LP Token Balance Remains Unchanged (ΔD = 0)

ΔD = 0 In scenarios where the balance of the LP token stays constant, the fee applies to the increase in balance of tokens. The equation for this fee structure is as follows:

∏ (b i − f e e (b i − a i) a i) w i = 1 \prod{(\frac{b_i - fee(b_i - a_i)}{a_i})}^{w_i}=1 ∏

(a i

b i − f ee (b i − a i)) w i

= 1

wheref e e (x) = m a x (0 , x ∗ f e e F a c t o r) fee(x) = max(0, x * feeFactor) f ee (x)

= ma x (0 ,

x

∗ f ee F a c t or) .

in other words, the taker must deposit additionalf e e (b⃗ − a⃗) fee(\vec{b}-\vec{a}) f ee (b

− a) in addition to fair trade delta.

This approach mirrors traditional constant-product market makers, where fees are deducted based on a fixed rate from input tokens.

solving the above equation the same way, yields the equation below, which is the way the calculation is performed.

b u n k n o w n − f e e (b u n k n o w n − a u n k n o w n) a u n k n o w n = (∏ k n o w n (b i − f e e (b i − a i) a i) w i) (− 1 ∑ u n k n o w n w i) \frac{b_{unknown} - fee(b_{unknown} - a_{unknown})}{a_{unknown}} = {(\prod_{known} {(\frac{b_i - fee(b_i - a_i)}{a_i})^{w_i}})}^{(-\frac{1}{\sum_{unknown} w_i})} a u nkn o w n

b u nkn o w n − f ee (b u nkn o w n − a u nkn o w n)

= (∏ kn o w n

(a i

b i − f ee (b i − a i)) w i) (− ∑ u nkn o w n

w i

1)

When LP Token Balance Changes (ΔD ≠ 0)

ΔD > 0 In the case when the LP token balance changes, our objective is to apply the usual fee structure to token swaps, while not applying it to proportional deposits and withdrawals. To do this, we divide the requested vector change, denoted asr⃗ = b⃗ − a⃗ \vec{r}=\vec{b}-\vec{a} r

= b

− a , into two parts: one subjected to full fees (taxable) and another not subjected to fees (non-taxable).

The taxable component must maintain the LP token balance (ΔD = 0), and the non-taxable component must correspond to scalar multiplication, with a coefficient we callk k . So the request vector could be split as:

r⃗ = b⃗ − a⃗ = (1 k b⃗ − a⃗) + k − 1 k b⃗ \vec{r}=\vec{b}-\vec{a}=(\frac{1}{k}\vec{b} - \vec{a}) + \frac{k-1}{k}\vec{b} r

= b

− a

= (k

1 b

− a)

+ k

k − 1 b , wherek = D b D a k=\frac{D_b}{D_a} k

= D a

D b .

With this division of the request vector, we can then compute our desired result, by solving for∏ (b i − f e e (1 k b i − a i) a i) w i = 1 \prod{(\frac{b_i - fee(\frac{1}{k}b_i - a_i)}{a_i})}^{w_i}=1 ∏

(a i

b i − f ee (k

1 b i − a i)) w i

= 1

It's crucial to note that there are numerous ways to divide the request vector.

For example, we can also split it as:

r⃗ = b⃗ – a⃗ = (b⃗ – k a⃗) + (k – 1) a⃗ \vec{r}=\vec{b}-\vec{a}=(\vec{b} - k \vec{a}) + (k-1)\vec{a} r

= b

– a

= (b

– k a)

+ (k

– 1) a .

When 'k' is more significant than one, representing a deposit scenario, the form(1 k b⃗ – a⃗) (\frac{1}{k}\vec{b} - \vec{a}) (k

1 b

– a) will yield a lower fee. Conversely, when 'k' is less than one, implying a withdrawal, the form(b⃗ – k a⃗) (\vec{b} - k \vec{a}) (b

– k a) will yield a lower fee. The pool will automatically select the version that results in the least fee.

in other words, we solve for∏ (b i – f e e (1 k b i – a i) a i) w i = 1 \prod{(\frac{b_i - fee(\frac{1}{k}b_i - a_i)}{a_i})}^{w_i=1 ∏

(a i

b i – f ee (k

1 b i – a i)) w i

= 1 or∏ (b i – f e e (b i – k a i) a i) w i = 1 \prod{(\frac{b_i - fee(b_i - ka_i)}{a_i})}^{w_i=1 ∏

(a i

b i – f ee (b i – k a i)) w i

= 1 , depending onk k k

Last updated7 months ago On this page *Mathematical framework * Calculating unknown dimensions in a zero-fee scenario * Calculating unknown dimensions with fees