Curve pools multiple allocation fairness strategies

Stefan Duprey, Tom Suter
SwissBorg
stefan@swissborg.com, tom@swissborg.com

***Abstract*—Fairness in IOUs distribution. Shrimp vs whales.**

I. Swap invariant

Automate market makers or DEX do come in multiple flavours.

A. Different AMM flavours

1) Constant sum:

$$\sum x_i = cte \tag{1}$$

2) Constant product:

$$\prod x_i = cte \tag{2}$$

3) Mix of both:

$$An^n \sum x_i + D = DAn^n + \frac{D^{n+1}}{n^n \prod x_i} \tag{3}$$

where x_i represents the balance of token i in the pool, A is the amplification coefficient and n is the number of tokens in the pool as explained in [1]. Let's write $Ann = Ann$, the invariant equation becomes :

$$Ann \sum x_i + D = DAnn + \frac{D^{n+1}}{n^n \prod x_i} \tag{4}$$

B. Curve : a mix of both

1) Swapping token with Curve: When a trader wants to know the amount x_j of the j token he will get for swapping dx token of x_i of the i token. The updated amount of token i in the pool can be calculated as $x_i \leftarrow x_i + dx$.
Since the tokens quantities always need to follow the stable swap invariant I-A3, the updated quantity for x_j can be found by solving :

$$x_j^2 + (b - D)x_j - c = 0 \tag{5}$$

where :

$$b = \sum_{i \neq j}^{n} x_i + \frac{D}{Ann} \tag{6}$$

and

$$c = \frac{D^{n+1}}{\prod_{i \neq j}^{n} x_i An^n n^n} \tag{7}$$

This second order polynomial roots can be found easily using a discriminant and square roots. Due to blockchain limited computations abilities, an approximation can be computed easily using Newton's method.
This approximation will converge to the proper root if we take as starting point the previous x_j value.
The converging algorithm can be written as :

$$x_{j_{n+1}} = x_{j_n} - \frac{x_{j_n}^2 + (b - D)x_{j_n} - c}{2x_{j_n} + b - D} = \frac{x_{j_n}^2 + c}{2x_{j_n} + b - D} \tag{8}$$

The quantity $dy = x_{j0} - x_{j\,final}$ where x_{j0} is the initial quantity of token j in the pool and $x_{j\,final}$ is the approximation.
Check out the implementation of the above algorithm in the $get\_y(i, j, x, xp)$ function of 3poolSwap contract [**?**]. The variable naming is consistent with the code.

Let's call that function :

$$D = getD(x_1, ..., x_n) \tag{9}$$

2) The stable invariant D: The stable invariant D is calculated by solving equation I-A3 for D, given all other parameters are constant.
The function $f(D)$, which is a polynomial function of degree $n + 1$ can be represented as :

$$f(D) = \frac{D^{n+1}}{n^n \prod x_i} + (Ann - 1)D - AnnS = 0 \tag{10}$$

3) Adding liquidity to a Curve pool: When depositing d ethers, we split the quantities according to the target allocations and deposit the matching quantities $d * \omega_i$ in each pool, and will receive IOUs Lps tokens to reflect their share of the pool.
The Lps token quantity is computed as :
We take the convention that token 1 is the same for each pool and is the one that is added.
We also assume there that there are n tokens on each m pools just for simplification purposes.
For each pool the

$$\text{old balance} \leftarrow \text{new balance}$$

$$(x_1^1, ..., x_n^1) \leftarrow (x_1^1 + d * \omega_1, ..., x_n^1)$$

$$(x_1^i, ..., x_n^i) \leftarrow (x_1^i + d * \omega_i, ..., x_n^i)$$

$$(x_1^m, ..., x_n^m) \leftarrow (x_1^m + d * \omega_m, ..., x_n^m)$$

We compute the theoretical change in the invariant D for all m pools.

$$\Delta D^1 = getD(x_1^1 + d * \omega_1, ..., x_n^1) - getD(x_1^1, ..., x_n^1)$$

$$\Delta D^i = getD(x_1^i + d * \omega_i, ..., x_n^i) - getD(x_1^i, ..., x_n^i)$$

$$\Delta D^m = getD(x_1^m + d * \omega_m, ..., x_n^m) - getD(x_1^m, ..., x_n^m)$$

Fees are then deducted based on a comparison between this theoretical growth by comparing the actual new balances and taken directly on the token amounts in the pool.

$$ideal\_x_i^m = \frac{(D^i + \Delta D^i)}{D^i} * x_i^m$$

where $D^i = getD(x_1^i, ..., x_n^i)$.
By denoting $new\_balance^m(i)$ the i^{th} component of the vector
$(x_1^m + D * \omega_m, x_2^m, ..., x_n^m)$ of new balances and $ideal\_balance^m(i)$ the i^{th} component of the vector
$(\frac{(D^m+\Delta D^m)}{D^m} * x_1^m, ..., \frac{(D^m+\Delta D^m)}{D^m} * x_n^m)$ of ideal balances.
Fees are deducted from the theoretical new balances :

$$new\_balance\_minus\_fee^m(i) \tag{11}$$

$$\leftarrow$$

$$new\_balance^m(i) - fee*$$

$$|new\_balance^m(i) - ideal\_balance^m(i)|$$

A new invariant taking the fees into account is then computed:

$$D\_new\_minus\_fees^m = getD(new\_balance\_minus\_fee^m) \tag{12}$$

The awarded l_p^m tokens amount for the m pools is then:

$$\Delta L_p^m = L_p^m * \frac{D\_new\_minus\_fees^m - D^m}{D^m} \tag{13}$$

where L_p^m is the actual amount of Lp tokens in the m pool.
All this can be checked in Curve smart contract $add\_liquidity$ function.
Let's call the function $get\_Lp(d, Lp_0, x_1, ..., x_n)$ which computes the provided liquidity tokens after depositing an amount d in a pool with an initial amount Lp_0 and an initial token amount $(x_1, ..., x_n)$.

4) Valuing the Lp tokens: The valuation method for the Lp tokens can be found in Curve smart contract through the get_virtual_price() method.

$$lpvalue = \frac{D}{Lp} \tag{14}$$

where Lp is the total amount of liquidity provider tokens in the pool.

5) Withdrawing liquidity to a Curve pool: We will here assume that the withdrawn token is the token 1 for all pools. When withdrawing an amount L of local tokens of the pool i, whose total lp amount is lp^i the pool invariant changes to :

$$D\_new^i = D\_previous^i * (1 - \frac{L}{lp^i}) \tag{15}$$

We then compute the new quantity x_1^i of token 1 for the pool i with the same methodology as in I-B1.
We find the new quantity $x\_new_1^i$ that will solve equation I-A3 with the new invariant D_new^i and the same old balances for all tokens j other than the token to be withdrawn ($j \neq 1$). By denoting $new\_balance^m(i)$ the i^{th} component of the vector $(x\_new_1^i, x_2^i, ..., x_n^m)$ of new balances after the withdrawal and $ideal\_balance^m(i)$ the i^{th} component of the vector $(\frac{D\_new^m}{D^m} * x_1^m, ..., \frac{D\_new^m}{D^m} * x_n^m)$ of ideal balances.
Fees are deducted from the theoretical new balances to the old balances :

$$old\_balance\_minus\_fee^m(i) \tag{16}$$

$$\leftarrow$$

$$old\_balance^m(i) - fee*$$

$$|new\_balance^m(i) - ideal\_balance^m(i)|$$

We then compute the new quantity x_1^i of token 1 for the pool i with the same methodology as in I-B1 but for the balances $old\_balance\_minus\_fee$: we find the new quantity $x\_new_{fee}_1^i$ that will solve equation I-A3 with the new invariant D_new^i and $old\_balance\_minus\_fee$ for all tokens j other than the token to be withdrawn ($j \neq 1$).

Stable invariant and balances and for each pool i are updated to D_new^i and
$(x\_new\_fee_1^i, x\_old\_fee_2^i, ..., x\_old\_fee_{m)}$ and the returned quantity of token 1 is $x\_new\_fee_1^i - x\_old\_fee_1^i$.
The exact computation can be checked in Curve smart contract function $remove\_liquidity$.
Let's call the function $redeem\_Lp(L, x_1, ..., x_n)$ which computes the provided liquidity tokens after redeeming an amount L in a pool with an initial token amount $(x_1, ..., x_n)$.

II. Mitigating slippage for the meta strategy while maintaining a target allocation

A. Meta strategy IOUs

1) Slippage in: For a target allocation of $(\omega_1, .., \omega_m)$ between our m pools, we define the entry slippage in as :

$$slip\_in = \sum_{pools\ i} \omega_i * slip\_in_i \tag{17}$$

where the local slippage for pool i is defined as:

$$slip\_in_i = \frac{get\_Lp(\omega_i * d, lp^i, x_1^i, ..., x_n^i) * get\_virtual\_price()}{D * \omega_i} \tag{18}$$

where the virtual price is computed with the new stable invariant and the new amount of Lps tokens after the deposit d.

If we define the metastrategy value as :

$$V = \sum_{poolsi} lp_i * get\_virtual\_price_i() \tag{19}$$

We see that

$$d * slip\_in = \sum_i \Delta lp_i * lpvalue_i = \Delta V(d)$$

2) Slippage out: Let's define the slippage out for a user withdrawing a L of the metastrategy IOUs pool.
For a target allocation of $(\omega_1, .., \omega_m)$ between our m pools, we define the exit slippage in as :

$$slip\_out = \sum_{pools\ i} \omega_i * slip\_out_i \tag{20}$$

where the local out slippage for pool i is defined as:

$$slip\_out_i = \frac{redeem\_Lp(\omega_i * L, x_1^i, ..., x_n^i)}{L * get\_virtual\_price() * \omega_i} \tag{21}$$

We see that the $get\_virtual\_price()$ is the lp^i token value for the pool i.

3) Target allocation and IOUs: The metastrategy IOUs minting amount subsequent to a deposit d and a target allocation $(\omega_1, ..., \omega_m) =$ for m pools is defined by:

$$IOUs\_minted = IOUs * \frac{\Delta V}{V} \tag{22}$$

The share of the metastrategy IOUs token following a deposit d is:

$$shares = \frac{IOUs\_minted}{IOUs + IOUs\_minted} = \frac{\Delta V}{V + \Delta V} \tag{23}$$

4) Immediate slippage in/slippage out: One key metrics for our meta strategy is the direct slippage the customer would incur if withdrawing directly the share he got by depositing an amount d. Let's write the value obtained as:

$$withdrawal\_value = get\_in(d) \tag{24}$$

Where the withdrawal value is directly computed from the minted IOUs by:

$$withdrawal\_value = shares * (V + \Delta V) = \Delta V \tag{25}$$

where shares is defined in 23. The customer will immediately withdraw the added value he got from the deposit.

$$Eths = get\_out(AV) \tag{26}$$

The total slippage can be written as :

$$\frac{get\_out(get\_in(d))}{d} \tag{27}$$

5) Optimal penalizer: In order to mitigate too big immediate in/out slippages, that would lead to the possibility of a financial attack through flash loans.
The idea here is to implement a penalizing mechanism where the in/out immediate slippage estimation will come modify the minted IOUS to prevent such attacks.
We modify the get_in function to add a penalizing term to the minted IOUs.
The optimal penalizer is found by solving the optimization problem:

$$\min_{p \in]1-b\,;b]} \left| \frac{get\_out(get\_in(d,p)) - d}{d} \right| \tag{28}$$

where b is the maximal penalizing bound we are ready to implement.

6) Heuristic: The problem 28 can be solved with numerical optimization algorithms. But as we cannot compute analytically the derivative, they tend to be computationally intensive and not appropriate to be coded in solidity on the blockchain. We therefore do propose the following heuristic, which is the simple renormalization of the minted IOUs by the slippage in and out.

$$p = \frac{1}{slip\_in * slip\_out} \tag{29}$$

The problem only resides in the evaluation of the $slip\_out$ which is defined by the minted IOUs itself. We therefore propose a recursive evaluation of the $slip\_out$ until reaches a stable value.

III. Optimizating allocation to keep the optimal APY when depositing

In the following, we describe the numerical optimization mentioned before When a deposit d is split according to $(x_1, ..., x_n)$ between multiple pools. We compute the total value using the newly minted IOUs per pool $lp\_minted^i(x_i * d)$ by the deposited amount $x_i * d$ into pool i.

A. Keeping the ratios between pools constant

1) Keeping the ratios between 2 pools constant: When depositing an amount d between 2 pools with the target ratio (x_1^*, x_2^*).
In the case of two pools, the exact analytical solution for the equation I-A3 can be found. We do not need a Newton numerical approximation as in 10, as the polynomial order for D is 3 and the Cardan formulas give us immediately the new invariant D.
The derivatives of that function can also easily be computed.
The optimal deposit allocation problem can be stipulated as follow:
We have in initial allocation between the Curve pool which is optimal.
(lp^1, lp^2) is such that $(\frac{lp^1}{lp^1+lp^2}, \frac{lp^2}{lp^1+lp^2}) = (x_1^*, x_2^*)$.
We want to find the optimal deposit allocation (x_1, x_2) such that newly minted lp^i for each pools do keep the already optimal allocation $(\frac{lp^1}{lp^1+lp^2}, \frac{lp^2}{lp^1+lp^2}) = (x_1^*, x_2^*)$.
If we note $c = \frac{lp^1}{lp^2}$, we see that

$$(x_1^*, x_2^*) = (\frac{1}{1 + \frac{1}{c}}, \frac{1}{1+c}) \tag{30}$$

If we keep the lps ratio between the pool constant, the optimal allocation is kept.
By adding $x_1 * d$ ethers to the pool 1, we note the newly added quantity of lp token for the pool i : $lp\_minted^i(x_i * d)$.
We see that if we keep the lp ratio constant :

$$\frac{lp^1 + lp\_minted^1}{lp^2 + lp\_minted^2} = c = \frac{lp^1}{lp^2} \tag{31}$$

Then the optimal allocation is still valid after a deposit (x_1, x_2).

$$\frac{lp^1 + lp\_minted^1}{lp^1 + lp\_minted^1 + lp^2 + lp\_minted^2} = \frac{1}{1 + \frac{1}{c}} = x_1^* \tag{32}$$

The equation is directly equivalent:

$$\frac{lp\_minted^1}{lp^1} = \frac{lp\_minted^2}{lp^2} \tag{33}$$

Given a deposit amount d, the opimization problem is then :

$$\text{Find } x_1 \text{such that} \tag{34}$$

$$\frac{lp\_minted^1(x_1 * d)}{lp^1} =$$

$$\frac{lp\_minted^2((1-x_1) * d)}{lp^2}$$

Using the analytical expression of lp_minted^i and its derivatives, one can use the Newton algorithm to find the x_1 root respecting the optimal allocation. lp_minted^i can be found using the share value deposited $x_i * d$ in the StableSwap invariant analytical form derived from I-A3:

$$D = \sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} - \sqrt[3]{\frac{1/27 * p^3}{(\sqrt{1/27p^3 + 1/4q^2}) - 1/2q)}} \tag{35}$$

where $p = 4*x*y*(4*A-1)$ and $q = -16*A*x*y*(x+y)$. we can rewrite III-A1

$$\frac{LP^1_{pool}}{LP^1_{Strat}} * (D_1^1 - D_0^1)/D_0^1 = \frac{LP^2_{pool}}{LP^2_{Strat}} * (D_1^2 - D_0^2)/D_0^2 \tag{36}$$

where D_1 is the StableSwap invariant evaluated with the new quantities deposited.

2) Exact optimal equation between n pools: Optimization When depositing an amount d of deposit between the pools of the meta strategy, we want to keep the initial target allocation, which optimized the global APY for the metastrategy.
One must find the optimal allocation of the deposit d such that the optimal allocation $(x_1^*, ..., x_n^*)$ still holds.
The problem in its most general form can be expressed as:

$$\min_{\sum_i x_i = 1} \left[\sum_{pools\ i} \left(\frac{lp^i + lp\_minted^i(x_i * d)}{\sum_{pools\ j} lp^j + lp\_minted^j(x_j * d)} - x_i^* \right)^2 \right] \tag{37}$$

where

$$x_i^* = \frac{lp^i}{\sum_j lp^j}$$

The optimization problem can be solved numerically with advanced non-linear solver, but those kind of solvers won't exist in Solidity. A natural starting point

$$(x_{10}, ..., x_{i0}, ..., x_{n0}) = (x_1^*, ..., x_i^*, ..., x_n^*) \tag{38}$$

Simplifying the optimization for low deposit Let's write $lp^j = l_j$ and $lp\_minted^j(x_j * d) = f_j(x_j)$. The optimization problem can be rewritten:

$$\min_{\sum_i x_i = 1} \left[\sum_i \left(\frac{l_i + f_i(x_i)}{\sum_j l_j + f_j(x_j)} - \frac{l_i}{\sum_j l_j} \right)^2 \right] \tag{39}$$

We now make the assumption that the total minted amount of tokens due to the deposit d is small compared to the total strategies lp tokens.

$$\frac{\sum_j f_j(x_j)}{\sum_j l_j} << 1 \tag{40}$$

A first order Taylor expansion gives us the following approximation:

$$\frac{1}{\sum_j (l_j + f_j(x_j))} = \frac{1}{\sum_j l_j} * \frac{1}{1 + \frac{\sum_j f_j(x_j)}{\sum_j l_j}} \tag{41}$$

$$\approx \frac{1}{\sum_j l_j} * (1 - \frac{\sum_j f_j(x_j)}{\sum_j l_j})$$

We obtain the simplified optimization problem:

$$\min_{\sum_i x_i = 1} \left[\sum_i \left(\frac{f_i(x_i)}{L} - \frac{1}{L^2} (\sum_j f_j(x_j))(l_i + f_i(x_i)) \right)^2 \right] \tag{42}$$

We see that we have another second order term $\frac{1}{L^2}(\sum_j f_j(x_j))(f_i(x_i))$ that we can drop so that the problem becomes :

$$\min_{\sum_i x_i = 1} \left[\sum_i \left(\frac{f_i(x_i)}{L} - \frac{1}{L^2} (\sum_j f_j(x_j))(l_i) \right)^2 \right] \tag{43}$$

By naming $f(x) = \begin{pmatrix} f_1(x_1) \\ \vdots \\ f_i(x_i) \\ \vdots \\ f_n(x_n) \end{pmatrix}$ and $a_i = \begin{pmatrix} -\frac{l_i}{L^2} \\ \vdots \\ -\frac{l_i}{L^2} + \frac{1}{L} \\ \vdots \\ -\frac{l_i}{L^2} \end{pmatrix}$

$$\min_{\sum_i x_i = 1} \left[\sum_i \left(a_i^t * f \right)^2 \right] \tag{44}$$

$$\sum_i \left(a_i^t * f \right)^2 = \sum_i f^t * a_i * a_i^t * f = f^t * (\sum_i a_i * a_i^t) * f$$

The problem then becomes :

$$\min_{\sum_i x_i = 1} \left[f^t * A * f \right] \tag{45}$$

where $A = \sum_i a_i * a_i^t$. A is obviously symmetric.
If we call $J(x) = f^t * A * f$, let's rewrite the new functional with Lagrange multiplier:

$$J'(x) = J(x) + \lambda(g(x) - 1) \tag{46}$$

where $g(x) = \sum_j x_j$, there comes the new functional.

$$\begin{cases} \nabla_x \left[J(x) + \lambda * (g(x) - 1) \right] = 0 \\ \nabla_\lambda \left[J(x) + \lambda * (g(x) - 1) \right] = 0 \end{cases} \tag{47}$$

By using that

$$\nabla_x \left[x^t * A * x \right] = (A + A^t) * x \tag{48}$$

There comes :

$$2A * f(x) * \nabla_x f(x) + \lambda \nabla_x g = 0 \tag{49}$$

IV. Conclusion

We have managed to simplify the optimization problem and transport it to EVM like blockchain languages.
The solution can scale up to at least ten pools without being too computationally intensive.

References

[1] M. Egorov, "Curve whitepaper," *https://curve.fi/files/stableswap-paper.pdf*.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[3] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, pp. 22–23, 2013.

[4] "Aave white paper," *https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf*.

[5] "Aave documentation," *https://docs.aave.com/portal/*.
#code#code#code

[] "Curve smart contract code," *https://etherscan.io/address/0xbebc44782c7db0a1a60cb6fe97d0b483032ff1c7#code*.