Curve Stableswap: From Whitepaper to Vyper

Curve Research[*]

January 11, 2023

Introduction

The stableswap invariant was derived by Michael Egorov and promulgated in the whitepaper, “StableSwap - efficient mechanism for Stablecoin liquidity”. The whitepaper clearly explained the invariant and its implications for DeFi; however, there are differences with how it is implemented in practice, currently across hundreds of live contracts across Ethereum and other layer 2s and chains.

Particularly important details for the practitioner but not given in the whitepaper are: 1) implementation of fees, both for exchanges and adding liquidity, 2) the practical solution procedures for the invariant and related quantities in integer arithmetic.

The practitioner seeking to understand the live functionality of the stableswap pools must look toward the vyper code for help, which while very readable, has minimal comments and explanation (indeed some comments are even wrong!). To understand the vyper code, the reader must have a solid grasp of the whitepaper in order to translate to the appropriate variables and understand various tweaks needed for implementation.

This note seeks to close the gap between the whitepaper and the vyper contracts. It seeks to give a consistent derivation of the necessary mathematics, using the notation and language of the contracts. At the same time, it points out and explains the “grungy” changes to calculations needed to ensure secure and safe operation on the Ethereum Virtual Machine.

[*]Curve Research is a community organization funded through the Curve DAO grants program and is not affiliated with Curve Finance (Swiss Stake GmbH). Neither Curve Research nor Curve DAO are responsible for any damages that result from use of the provided information or guarantee its accuracy.

Preliminaries (notation and conventions)

Stableswap equation

This is the original stableswap equation:

$$A \cdot n^n \sum_i x_i + D = A \cdot n^n \cdot D + \frac{D^{n+1}}{n^n \prod_i x_i}$$

In the vyper code, the amplification constant A actually refers to $A \cdot n^{n-1}$, so the equation becomes:

$$A \cdot n \sum_i x_i + D = A \cdot n \cdot D + \frac{D^{n+1}}{n^n \prod_i x_i}$$

This is the form we use for all our derivations.

Coin balances

We denote the coin balances (as in the contracts) with x_i, x_j etc. In the context of a swap, i is the "in" index and j is the "out" index.

The quantities though are in special units. They are *not* native token units. For example, if x_i represents the USDC amount, one whole token amount would not be 1000000 as might be assumed from the 6 decimals for USDC. Instead x_i would be 1000000000000000000 (18 zeroes). All the x balances should be assumed to be in the same units as D. For lack of a better term, sometimes we will call these *virtual* units (as the amount of D per LP token is often referred to as "virtual price") and we will call the x balances *virtual balances.*

While putting balances into virtual units often involves only a change of decimals, this is not the correct way of thinking about the process. The stableswap equation assumes a 1:1 peg between coins. This means the balances being used must reflect the proper value in the common units of D being used. For the example of USDC, this means adjusting simply the decimals. For a rebasing token however, it may not be. Indeed, for metapools, when exchange the basepool's tokens for the main stable, the basepool token conversion into D units must take into account the accrued yield. This is done by multiplying the amount by the basepool virtual price.

Solving for D

Since the arithmetic mean is greater than the geometric mean (unless the balances x_i are equal, in which case the means are identical), the form of the equation suggests that there ought to be a D in-between the means that satisfies the equation.

To see this rigorously, we use the auxiliary function:

$$f(D) = A \cdot n \cdot (D - A \cdot n \sum_i x_i) + D \cdot (\frac{D^n}{n^n \prod_i x_i} - 1)$$

Let $P = n\,(\prod_i x_i)^{\frac{1}{n}}$ and $S = \sum_i x_i$. This is a continuous function (away from zero balances) with $f(P) < 0$ and $f(S) > 0$. So there is a D such that $P < D < S$ and $f(D) = 0$. In particular, note

$$f'(D) = A \cdot n + (n+1)\frac{D^n}{n^n \prod_i x_i} - 1$$

the derivative of f, is positive (assuming $A >= 1$), so f is strictly increasing and there is a unique D that solves $f(D) = 0$.

Newton's method

The stableswap contracts utilize Newton's method to solve for D. It is easy to check $f'' > 0$, i.e. f is convex. An elementary argument shows that this guarantees convergence of Newton's method starting with initial point S to the solution.

The vyper code (from 3Pool) is:

```
@pure
@internal
def get_D(xp: uint256[N_COINS], amp: uint256) -> uint256:
    S: uint256 = 0
    for _x in xp:
        S += _x
    if S == 0:
        return 0

    Dprev: uint256 = 0
    D: uint256 = S
    Ann: uint256 = amp * N_COINS
    for _i in range(255):
        D_P: uint256 = D
        for _x in xp:
            D_P = D_P * D / (_x * N_COINS)  # If division by 0, this will be borked: only wi
        Dprev = D
        D = (Ann * S + D_P * N_COINS) * D / ((Ann - 1) * D + (N_COINS + 1) * D_P)
        # Equality with the precision of 1
        if D > Dprev:
            if D - Dprev <= 1:
                break
        else:
            if Dprev - D <= 1:
                break
    return D
```

This code is used with minimal difference between all the stableswap contracts. For safety, later versions choose to revert if the 255 iterations are exhausted before converging.

The iterative formula is easily derived:

$$\begin{aligned}
d_{k+1} &= d_k - \frac{f(d_k)}{f'(d_k)} \\
&= d_k - \frac{An(d_k - A\sum_i x_i) + d_k(\frac{d_k^n}{n^n \prod_i x_i} - 1)}{\frac{(n+1)d_k^n}{n^n \prod_i x_i} + An - 1} \\
&= \frac{An\sum_i x_i + \frac{nd_k^{n+1}}{n^n \prod_i x_i}}{\frac{(n+1)d_k^n}{n^n \prod_i x_i} + An - 1} \\
&= \frac{AnS + nD_p(d_k)}{\frac{D_p(d_k)}{d_k} + An - 1} \\
&= \frac{(AnS + nD_p(d_k))d_k}{D_p(d_k) + (An-1)d_k}
\end{aligned}$$

where $S = \sum_i x_i$ and $D_p(d_k) = \frac{d_k^{n+1}}{n^n \prod_i x_i}$

Rate of convergence

Convergence follows from convexity of f. However we need much better than that, we need to reduce the distance to the solution by half each time, otherwise 255 iterations is not sufficient. Also, in practice, exceeding more than half a dozen iterations is not sufficiently gas efficient enough to be competitive. We will in fact demonstrate quadratic convergence, which means the Netwon estimate will double in accuracy on each iteration.

First, to see convergence is straightforward. First recall that $f(P) < 0$ and $f(S) > 0$ and that there is exactly one zero in $[P, S]$ for f.

We have the first derivative:

$$f'(D) = A \cdot n + (n+1)\frac{D^n}{n^n \prod_i x_i} - 1$$

and the second derivative:

$$f''(D) = n(n+1)\frac{D^{n-1}}{n^n \prod_i x_i}$$

With $A >= 1$, $f' > 0$ and $f'' > 0$ everywhere, and in particular on $[P, S]$ the domain of interest.

The formula for Newton's method is:

$$d_{k+1} = d_k - \frac{f(d_k)}{f'(d_k)}$$

Since f is convex, its graph lies about every tangent line and in particular, for every iteration of Newton's method, $f(d_{k+1}) > 0$ (since the tangent line approximation intersects the y-axis at d_{k+1}). Since $f'(d_k) > 0$ also, we see that d_{k+a} is always to the left of d_k. The solution D we are seeking is always to the left of any iterate (because $f(D) = 0$ while $f(d_k) > 0$) so the sequence d_k converges to c. We claim $f(c) = 0$. This can be seen from the iterative formula. Since $d_k - d_{k+1}$ get arbitrarily small, $\frac{f(d_k)}{f'(d_k)}$ gets arbitrarily small. But the denominator $f'(d_k)$ has a max on $[c, S]$ so the numerator must be getting arbitrarily small, i.e. $f(d_k) \to 0$, which implies $f(c) = 0$.

For quadratic convergence, we first need to derive an inequality using a couple applications of the mean-value theorem.

Let $\delta_k = d_k - c$. Then $f'(\eta_k)(\delta_k) = f(d_k) - f(c) = f(d_k)$. Rewriting, $\delta_k = \frac{f(d_k)}{f'(\eta_k)}$.

Then the Newton iteration can be rewritten as:

$$
\begin{aligned}
\delta_{k+1} &= \delta_k - \frac{f(d_k)}{f'(d_k)} \\
&= \frac{f(d_k)}{f'(\eta_k)} - \frac{f(d_k)}{f'(d_k)} \\
&= \frac{f(d_k)(f'(d_k) - f'(\eta_k))}{f'(\eta_k)f'(d_k)} \\
&= \frac{f(d_k)f''(\xi_k)(d_k - \eta_k)}{f'(\eta_k)f'(d_k)} \\
&= \frac{f''(\xi_k)}{f'(d_k)}(d_k - \eta_k)\delta_k \\
&\le \frac{f''(\xi_k)}{f'(d_k)}\delta_k^2 \\
&\le \frac{f''(d_k)}{f'(d_k)}\delta_k^2 \\
&\le \frac{n(n+1)d_k^{n-1}}{(n+1)d_k^n + (An-1)n^n \prod_i x_i}\delta_k \cdot \delta_k \\
&\le \frac{n(n+1)d_k^{n-1}}{(n+1)d_k^n}\delta_k \cdot \delta_k \\
&\le n\frac{\delta_k}{d_k} \cdot \delta_k \\
&\le n\frac{\delta_k}{c} \cdot \delta_k
\end{aligned}
$$

Note that $\frac{\delta_k}{c}$ is the distance of our estimate to the solution, as a percentage of the solution.

In particular, if we can get $n\frac{\delta_0}{c} < m$ for the first estimate, then for subsequent estimates, we find:

$$\delta_k = m^{2^k - 1}\delta_0$$

So if m is less than 1, say 1/2, this gives incredibly fast convergence.

Integer arithmetic

The swap equation

The stableswap equation allows you to solve for any coin balance given the other balances and the value of D. This is a fundamental property needed for enabling swap functionality. Since this is not derived in the whitepaper, we go through it here.

The stableswap equation can be re-written in the form:

$$An\left(x_j + \sum_{k\neq j} x_k\right) + D = AnD + \frac{D^{n+1}}{n^n x_j \prod_{k\neq j} x_k}$$

where j is the out-token index.

Let's denote $\sum_{k\neq j} x_k$ by S' and $\prod_{k\neq j} x_k$ by P'.

Then we have, after some re-arranging

$$x_j + S' + \frac{D}{An} = D + \frac{D^{n+1}}{An^{n+1} x_j P'}$$

This becomes

$$x_j^2 + \left(S' + \frac{D}{An} - D\right) x_j = \frac{D^{n+1}}{An^{n+1} P'}$$

or

$$x_j^2 + bx_j = c$$

where $b = S' + \frac{D}{An} - D$ and $c = \frac{D^{n+1}}{An^{n+1}P'}$.

This quadratic equation can be solved by Newton's method:

$$\begin{aligned} x_j &:= x_j - \frac{x_j^2 + bx_j - c}{2x_j + b} \\ &:= \frac{x_j^2 + c}{2x_j + b} \end{aligned}$$

Note the actual vyper code cleverly defines b as our b without the $-D$ term. This allows b to be defined as a `uint256` since otherwise it could be negative (although of course $2x_j + b$ is always positive).

The vyper code should be understandable now:

```
@view
@internal
def get_y(i: int128, j: int128, x: uint256, xp_: uint256[N_COINS]) -> uint256:
    # x in the input is converted to the same price/precision

    assert i != j       # dev: same coin
    assert j >= 0       # dev: j below zero
    assert j < N_COINS  # dev: j above N_COINS

    # should be unreachable, but good for safety
    assert i >= 0
    assert i < N_COINS

    amp: uint256 = self._A()
    D: uint256 = self.get_D(xp_, amp)
    c: uint256 = D
    S_: uint256 = 0
    Ann: uint256 = amp * N_COINS

    _x: uint256 = 0
    for _i in range(N_COINS):
        if _i == i:
            _x = x
        elif _i != j:
            _x = xp_[_i]
        else:
            continue
        S_ += _x
        c = c * D / (_x * N_COINS)
    c = c * D / (Ann * N_COINS)
    b: uint256 = S_ + D / Ann  # - D
    y_prev: uint256 = 0
    y: uint256 = D
    for _i in range(255):
        y_prev = y
        y = (y*y + c) / (2 * y + b - D)
        # Equality with the precision of 1
        if y > y_prev:
            if y - y_prev <= 1:
                break
        else:
            if y_prev - y <= 1:
                break
    return y
```

So given all the normalized balances (the out-token balance doesn't matter), we can compute the balance of the out-token that satisfies the stableswap equation for the given D and other balances.

This is what's done in the `get_dy` function in the stableswap contract:

```
@view
@external
def get_dy(i: int128, j: int128, dx: uint256) -> uint256:
    # dx and dy in c-units
    rates: uint256[N_COINS] = RATES
    xp: uint256[N_COINS] = self._xp()

    x: uint256 = xp[i] + (dx * rates[i] / PRECISION)
    y: uint256 = self.get_y(i, j, x, xp)
    dy: uint256 = (xp[j] - y - 1) * PRECISION / rates[j]
    _fee: uint256 = self.fee * dy / FEE_DENOMINATOR
    return dy - _fee
```

The key logic is given in the lines:

```
y: uint256 = self.get_y(i, j, x, xp)
dy: uint256 = (xp[j] - y - 1) * PRECISION / rates[j]
```

As usual the `xp` balances are the virtual balances, the token balances normalized to be in the same units as `D` with any rate adjustment to compensate for changes in value, e.g. accrued interest.

So by using `get_y` on the in-token balance increased by the swap amount `dx`, we can get the new out-token balance and subtract from the old out-token balances, which gives us `dy`. This then gets adjusted to native token units with the fee taken out.

The `get_dy` isn't actually what's used to do the exchange, but the `exchange` function does the identical logic while handling token transfers and other fee logic, particularly sweeping "admin fees", which are the fees going to the DAO. In any case, the amount `dy` is the same.

Note that an extra "wei" is subtracted in `xp[j] - y - 1`. This is because `y` might be truncated due to the integer division, effectively rounding down. So we compensate by increasing `y` by 1. Defending against value leakage like this protects the pool from attackers potentially draining the pool, although it may very slightly penalize the user in some cases.

Fees

Fees enter into the picture in a couple different ways. During an exchange, the out-token amount received by the user is reduced. This is added back to the pool, which increases the stableswap invariant (the invariant increases when a coin balance increases, as can be checked using the usual calculus). This effectively increases the balances for LPs when they redeem their LP tokens.

The other case is when liquidity is added to the pool (if liquidity is removed in an imbalanced way, fees also apply in that case). When adding liquidity, swap fees are deducted for coin amounts that differ from the "ideal" balance (same proportions as the pool). The reduced input amounts are then used to mint LP tokens.

Exchange

Adding and removing liquidity

Balanced deposits and withdrawals

Removing one coin

Useful formulas

Price

Use the auxiliary function:

$$f(x_1, x_2, ..., x_n, D) = An \sum_i x_i + D - AnD - \frac{D^{n+1}}{n^n \prod_i x_i}$$

When $f(x_1, ..., x_n, D) = 0$, we have the stableswap equation. We will be supposing D fixed in what follows, so you can consider f to be a function of just the x_i's.

Computing the partials of f, we get:

$$\begin{aligned}
\frac{\partial f}{\partial x_k} &= An - \frac{\partial}{\partial x_k}\left(\frac{D^{n+1}}{n^n \prod x_i}\right) \\
&= An - \frac{\partial}{\partial x_k}\left(\frac{D^{n+1}}{n^n \prod_{i \neq k} x_i} \cdot \frac{1}{x_k}\right) \\
&= An + \frac{D^{n+1}}{n^n \prod_{i \neq k} x_i} \cdot \frac{1}{x_k^2} \\
&= An + \frac{D^{n+1}}{n^n \prod_i x_i} \cdot \frac{1}{x_k}
\end{aligned}$$

When restricting f to the level set given by $f(x_1, ..., x_n) = 0$, we must have $\frac{\partial f}{\partial x_i} dx_i + \frac{\partial f}{\partial x_j} dx_j = 0$.

Now the price is

$$
\begin{aligned}
-\frac{\partial x_j}{\partial x_i} &= \frac{\frac{\partial f}{\partial x_i}}{\frac{\partial f}{\partial x_j}} \\
&= \frac{An + \frac{D^{n+1}}{n^n \prod_k x_k} \cdot \frac{1}{x_i}}{An + \frac{D^{n+1}}{n^n \prod_k x_k} \cdot \frac{1}{x_j}} \\
&= \left(\frac{x_j}{x_i}\right) \frac{\left(Anx_i + \frac{D^{n+1}}{n^n \prod_k x_k}\right)}{\left(Anx_j + \frac{D^{n+1}}{n^n \prod_k x_k}\right)} \\
&= \left(\frac{x_j}{x_i}\right) \frac{\left(\frac{An^{n+1} x_i \prod_k x_k}{D^{n+1}} + 1\right)}{\left(\frac{An^{n+1} x_j \prod_k x_k}{D^{n+1}} + 1\right)}
\end{aligned}
$$

Some sanity checks:

- when A is 0, the price is $\frac{x_j}{x_i}$, the price for a constant product AMM.
- when $A \to \infty$, price is 1, the price for a constant sum AMM.
- when $x_j = x_i$, price is 1.

Slippage

Depth

https://etherscan.io/address/0xbebc44782c7db0a1a60cb6fe97d0b483032ff1c7
https://github.com/curvefi/curve-contract/blob/d808ed824ad6008d554dc7a70c0bbcb2ba8b9349/contracts/pools