KittenSwap : Navigating the Design Space of DEX

Sep 6, 2020 [WORKING DRAFT]

KittenSwap is a hybrid AMM+Book DEX (www.kittenswap.org) currently in active development. The name might be a meme, however KittenSwap is a serious project.

In this draft I will review typical AMM designs and challenges, and describe how KittenSwap chooses to solve the problem.

Some details are deliberately left out in this draft, to prevent potential immediate copycats.

1 The Uniswap model, and my view

I will begin with a review of Uniswap. Conside the token pair A/B.

Let $\{q_A, q_B\}$ be the current amount of A and B tokens in the contract, and q_{LP} be the total supply of LP shares.

There are three kinds of common events:

1. LP-DEPOSIT : the LP is swapping $A+B$ for LP shares.
2. LP-WITHDRAW : the LP is swapping LP shares for $A+B$.
3. TRADER-SWAP : the trader is swapping between A and B.

and I would take a new **unified view**: all three events are swapping tokens, therefore they can be processed by one single KittenSwap$(\cdots)$ function under a **unified model**. More on that in later sections.

Some formulas. For LP-DEPOSIT based on B, we have:

$$\delta_A = \delta_B \cdot q_A/q_B + \epsilon$$

$$\delta_{LP} = \delta_B \cdot q_{LP}/q_B$$

For LP-WITHDRAW, we have:

$$\delta_A = \delta_{LP} \cdot q_A/q_{LP}$$

$$\delta_B = \delta_{LP} \cdot q_B/q_{LP}$$

For TRADER-SWAP from B to A, we have:

$$Q(q_A, q_B) = q_A \cdot q_B = (q_A - \delta_A)(q_B + \lambda\delta_B) = Q(q'_A, q'_B)$$
$$\implies \quad \delta_A = \frac{\lambda\delta_B \cdot q_A}{q_B + \lambda\delta_B} \quad \text{or} \quad \delta_B = \frac{\delta_A \cdot q_B}{\lambda q_A - \lambda\delta_A}$$

Notice Uniswap applies $\lambda = 0.997$ for the incoming token B. **I believe it's better to apply λ for the outgoing token A instead**. Because if A is in demand, leaving some A instead of B in the pool is better for LPs. Please correct me if I made a mistake.

Here is the relevant code in the UniswapV2Router02 contract:

```
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal
    pure returns (uint amountOut) {
   ...
   uint amountInWithFee = amountIn.mul(997);
   uint numerator = amountInWithFee.mul(reserveOut);
   uint denominator = reserveIn.mul(1000).add(amountInWithFee);
   amountOut = numerator / denominator;
}
```

The marginal price $p = p_{A/B}$ is:

$$p_{A/B} = \left.\frac{\partial\delta_B}{\partial\delta_A}\right|_{\delta_A=0} = \frac{q_B}{\lambda q_A}$$

as expected.

Uniswap is a special case of CFMMs. The constant function Q is a weighted geometric mean in Balancer, and a mixture of sum and product in Curve.

2 KittenSwap : the dynamic AMM pool

One of the drawbacks of Uniswap is the LP has a fixed 50-50 exposure to both tokens. The situation is improved in Balancer, however the token weights are still fixed at pool creation.

KittenSwap purposes a **new solution**, where the LP is free to deposit either of the pair tokens, and there is no swap-at-deposit loss as in the case of Balancer.

That is, in Balancer when you deposit A into an A/B pool, you are forced to swap part of A into B, taking a loss. KittenSwap solves this.

The KittenSwap solution is to introduce a **dynamic weight** w, and changing it at each LP-DEPOSIT and LP-WITHDRAW event. This is best explained using a table of events.

KittenSwap	q_A	q_B	w	$p_{A/B}$	remark
HIDDEN	FOR	NOW	LET'S	WAIT FOR	RELEASE

And the perfect circle is completed.

When LPs deposit and withdraw, the $p_{A/B}$ does not change, hence LPs **have** 0 **loss** in the process, and only the w changes dynamically.

Careful readers will notice some hidden problems and corner cases, moreover there can be further improvements. Here we will deliberately conceal our full solution, as mentioned before.

Basically, a KittenSwap pool has two kinds of shares LPA and LPB (instead of one single share as in Uniswap). The LP gets LPA shares for depositing A, and LPB shares for depositing B, and the prices of LPA and LPB shares are dynamically adjusted after each TRADER-SWAP event.

The formulas for adjusting share prices are not obvious. I will show a glimpse of my closed-form formulas to demonstrate the depth of the problem.

Let the initial pool state $\Omega = \{q_A, q_B, w\}$, and let the initial prices of pool shares be $1\,[LPA] = 1[A]$, $1\,[LPB] = 1\,[B]$.

If a trader purchases δ_A amount of A token from the pool, then the price of LPA becomes $1\,[LPA] = a\,[A] + b\,[B]$, where the formula for a is $[HIDDEN-FOR-NOW]$ and the formula for b is $[HIDDEN-FOR-NOW]$.

For instance, if $\Omega = \{q_A = 2, q_B = 1, w = 1\}$ and $\delta_A = 1$ then:

$$a = [HIDDEN-FOR-NOW], \quad b = [HIDDEN-FOR-NOW]$$

and as a sanity check, we have:

$$2\,[LPA] + 1\,[LPB] = 1\,[A] + 2\,[B]$$

because $\{q'_A = 1, q'_B = 2\}$ after the trade, and the value of outstanding pool shares $(2\,[LPA] + 1\,[LPB])$ must equal the value of the pool $(1\,[A] + 2\,[B])$.

3 On the impermanent loss problem

The impermanent loss problem is usually claimed to be solved using an external oracle, which in fact brings in other problems.

KittenSwap plans to avoid an external oracle. **KittenSwap will be an oracle itself**, and capable of providing price feeds to other cryptoprojects.

This requires sophisticated designs because this is known to be difficult problem involving game theory between miners and arbitrageurs and traders and LPs. More on that later.