WIP: PowerMaker – Oracle-Free Power^2 Perpetuals

Robert Leifke
University of Michigan
rleifke@umich.edu

Kyle Scott
University of Michigan
kyscott18@umich.edu

April 2024

Abstract

This paper describes a mechanism for constructing Power^2 Perpetuals without oracles and its implementation on the Ethereum Virtual Machine. Thus offering a means of accessing convexity on any token without trusted intermediaries.

1 Introduction

A **Power^2 Perpetual** describes a perpetual future that tracks an index of two and whose payoff is quadratic.

> If the price of ETH goes up by 2%, then a trader's gains will go up by 4%. If the price goes up by 4%, then a trader's gains will go up by 16% and so on.

Beyond leverage, power^2 perpetuals expose traders to the variance of a token's returns which otherwise requires traders to manage a portfolio of traditional options. It has shown to be a hedge against Uniswap (Clark, 2023).

Unfortunately the oracle dependency of current power perpetual designs like Squeeth has limited the offering to ETH. As a solution, we construct a power perpetual by lending the LP shares of a constant function market maker (CFMM) whose trading function matches that of the payoff of its replicated options portfolio (Angeris, 2021). This means the borrowed LP shares of a power^2 perpetual CFMM behave like power^2 perpetuals themselves. Thereby providing a mechanism for pooling liquidity to power perpetuals inexpensively on a blockchain.

2 Capped Power^2 Market Maker

The unique CFMM implements the *capped power* invariant introduced in Replicating Monotonic Payoffs. The trading function that replicates a power^2 perpetual , "Squeeth" is:

$$\varphi(R_1, R_2) = R_1 - \left(k - \frac{1}{2}R_2\right)^2 \tag{1}$$

where the replication holds between initial price 0 and the strike k. The invariant corresponds to a portfolio value $V(p)$ that is concave with p being the price of the speculative asset in terms of base asset:

$$V(p) = \begin{cases} 2p * k - p^2 & 0 \leq p \leq k \\ k^2 & p > k \end{cases} \tag{2}$$

2.1 Constructing a Convex Payoff

Convexity is achieved by borrowing the LP share. Liquidity providers receive a share representing their deposit into the underlying liquidity pool. This share is then deposited in a specialized lending pool and made available to borrowers.

Those wishing to receive the power^2 perpetual payoff are the borrowers of the liquidity shares. This second party of users determines the maximum amount of speculative asset that will ever be in a liquidity provider shares i. i can be found by taking $\lim_{p \to 0^+} V(p) = 2k$ with a value $I(p) = 2p * k$. Power^2 perpetuals are then constructed by taking i speculative tokens as collateral and borrowing one share of the underlying liquidity pool. This position with $I(p)$ collateral and $V(p)$ debt is $N(p)$

$$N(p) = \begin{cases} p^2 & 0 \leq p \leq k \\ 2p * k - k^2 & p > k \end{cases} \tag{3}$$

It is important to note that because the liquidity shares are non-fee accruing, $I(p) \geq V(P)\ \forall\ p \geq 0$, so that once the reserves in the borrowed LP share run out, then payoff ends. The payoff is guaranteed until the strike price k.

2.1.1 Avoiding pool manipulation

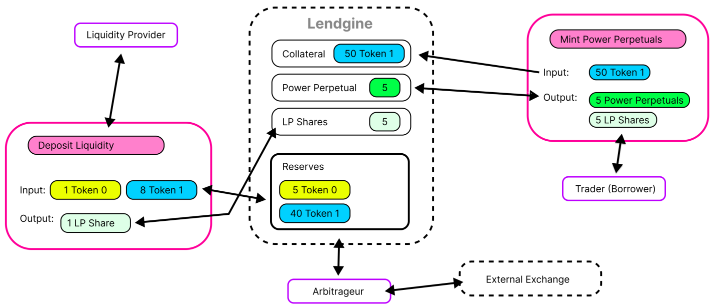
The mechanism differs from the system described in the *Replicating Monotonic Payoff* paper where the LP share splits the speculative and base token. In that system, the long payoff of the speculative is given to the holder of the base, and the short is holding the speculative, receiving funding in the base asset. For our use case, that mechanism is susceptible to an internal oracle flash attack when implemented.

3 Solidity Implementation

PowerMaker is a suite of smart contracts on the EVM that facilitates the creation of oracle-free power^2 perpetuals. The smart contracts are divided among core contracts implementing critical accounting logic and a periphery that interacts with core contracts for extra safety checks and ease-of-use functionality. All contracts are compatible with compiler version SOLC 0.8.7.

The primary contract in PowerMaker is `Lendgine.sol`. It is the single source of truth and accounting across all PowerMaker instances. Each `lendgine` instance can thought of as the power^2 perpetual itself. It manages the borrowing of the CFMM LP shares between a liquidity provider and trader.

3.1 `lendgine` logic



Each *power*^{2} perpetual token "P2" is a single `lendgine` instance with the receipt token being a fungible ERC-20 with the same base token, risky token, and strike. The minting of a *power*^{2} perpetual is done in conjunction with a flash swap. When the borrower is ready to redeem the underlying value of the power perpetual and thereby repay the debt to the protocol, a flash swap is executed to convert the LP share to collateral and then pay it all back.

Lendgines takes advantage of the fact that the underlying token balances per liquidity are bounded. Any amount above this bound can be used as collateral to borrow liquidity and therefore be always over-collateralized and never at risk of liquidations.

3.1.1 Strikes

Every `lendgine` P2 token is bounded. That means the implicit leverage stops once it reaches its `strike`. Then all the reserves of a given base token like USDC is converted into the speculative token ETH. Once this happens, the payoff is linear to the spot price of the speculative token. Although there are no expirys, strikes cannot be avoided without an external margin system. The `strike` has been arbitrarily set to be two times the price of the initial entry price when someone swaps into the power perpetual. This way the underlying asset can increase by 100% and the theoretical gains could be 400% assuming no fees paid.

3.1.2 Borrow rate

For the convexity, a continuous fee is paid from borrowers to liquidity providers. The funding is determined by a interest rate curve whose logic is held in `JumpRate.sol`.

The borrow rate between liquidity providers and the power[2] perpetual holders is calculated as:

$$multiplier * min(U_a, kink) + jumpMultiplier * max(0, U_a - kink) \quad (4)$$

The value for all three can be found by finding the kink or point at which the rates jump at a given utilization of the pool. Since Deribit's ATM 180 day implied volatility for ETH is 78.5%, the kink was also set at 78.5% leading to the following values:

uint256	**kink**	.785 ether
uint256	**multiplier**	1.375 ether
uint256	**jumpMultiplier**	44.5 ether

3.2 `invariant()` logic

The capped power[2] market maker invariant is is implemented in `Pair.sol`.

```
function invariant(uint256 amount0, uint256 amount1, uint256 liquidity)
    public view override returns (bool) {

    if (liquidity == 0) return (amount0 == 0 && amount1 == 0);

    uint256 scale0 = FullMath.mulDiv(amount0 * token0Scale, 1e18, liquidity);
    uint256 scale1 = FullMath.mulDiv(amount1 * token1Scale, 1e18, liquidity);

    if (scale1 > 2 * upperBound) revert InvariantError();

    uint256 a = scale0 * 1e18;
    uint256 b = scale1 * upperBound;
    uint256 c = (scale1 * scale1) / 4;
    uint256 d = upperBound * upperBound;

    return a + b >= c + d;
}
```

The `scale0` and `scale1` variables are calculated by dividing the amount0 and amount1 variables by the liquidity variable, and then multiplying by 1e18. The `strike` variable is a constant that defines the maximum value that the scale1 variable can take.

Given the limitations of the EVM the invariant has the be implemented in a way where it equals zero or is greater than to avoid precision loss. The function `invariant()` checks whether the pool reserves satisfies that condition. If the invariant condition is not satisfied, the function will revert with an `InvariantError()`.

3.2.1 Pool Deployments

Important to note is that the `Pair.sol` contract is abstract, meaning that it is not directly deployable. Instead, it is inherited by a lendgine. For a better developer experience, all pair deployments share the same factory address via the `create3` opcode.

3.3 Periphery

The periphery contracts consists of a lendgine router and a liquidity manager, both meant to make interacting with core contracts safer and more user friendly. Because of PowerMaker's permission-less nature, periphery contracts have no special privileges with core and are completely replaceable by other periphery contracts with possible a different set of features.

3.3.1 `LiquidityManager.sol`

The liquidity manager custodies liquidity positions and fully supports adding and removing liquidity from a lendgine. It also contains the same ease-of-use features found in lendgine router.

3.3.2 `LendgineRouter.sol`

The lendgine router allows for easy minting and burning of P2 tokens. It contains logic for general actions such as handling WETH, supporting EIP-2612 tokens, slippage checks, and staleness checks. More specifically, the lendgine router allows for maximum leverage on P2 by facilitating the borrowing of liquidity, and selling the underlying tokens into more collateral to borrow more liquidity. Liquidity provider positions are represented as a struct with extra variables to account for accrued funding rewards in an algorithm introduced by [6].

uint256	**liquidity**
uint256	**rewardPerLiquidityPaid**
uint256	**tokensOwed**

4 Future Work

4.1 Optimal Borrow Rate through LVR

The capped power^2 market maker like any other CFMM exposes the liquidity provider to loss-versus-rebalancing (LVR). This is the aggregate loss to arbitrageurs incurred by LPs via rebalancing. We can use the LVR to find the optimal fee per block and construct a methodology for the optimal borrow rate. In an efficient market, the continuous fee paid to liquidity providers offsets their LVR. For a capped power^2 market maker, that should be exactly equal the variance of the underlier σ^2.

References

[1] Guillermo Angeris, Alex Evans, Tarun Chitra, *Replicating Monotonic Payoffs*, Papers 2111.13740.pdf, arxiv.org, September 2021.

[2] Guillermo Angeris, Tarun Chitra, Alex Evans, *Replicating Market Makers*, Papers 2103.14769, arXiv.org, March 2021.

[3] Estelle Sterrett, Alexander Angel, Matt Czernik *Primitive RMM-01*, primitive.xyz, October 2021.

[4] Guillaume Lambert, *On-chain Implied Volatility and Uniswap v3*, lambert-guillaume.medium.com, November 2021.

[5] Guillermo Angeris, Tarun Chitra *Improved Price Oracles: Constant Function Market Makers*, Papers 2003.10001, arxiv.org, June 2020.

[6] Bogdan Batog, Lucian Boca, Nick Johnson *Scalable Rewards Distribution on the Ethereum Blockchain*, https://uploads-ssl.webflow.com

[7] Fredico Magnani, *Notes on Replicating Monotonic Payoffs without Oracles*, https://github.com/fedemagnani/Notes-Replicating-Monotonic-Payoffs-Without-Oracles/

[8] Robert Leifke, Kyle Scott, *PowerMaker codebase*, https://github.com/numotrade/power-maker/

[9] Clark, Joseph, Spanning with Power Perpetuals (January 3, 2023). Available at SSRN: https://ssrn.com/abstract=4317072 or http://dx.doi.org/10.2139/ssrn.4317072

[10] Deribit, Implied Volatility, https://metrics.deribit.com/futures/BTC, April 2024.