Pendle V2

1. Overview

- PendleV2 is a complete and standalone smart contract system for tokenising and trading the yield of yield generating mechanisms in DeFi
- PendleV2 consists of 5 relatively independent components:
    - Liquid reward token (LYT), a.k.a ERC4000: a standard for any mechanism that generates yield in DeFi
    - LYT Depository: a system to hold LYTs for any protocols that want to work with LYTs
    - Yield Splitter: a protocol for splitting a LYT into yield tokens (YTs) and Ownership Tokens (OTs)
    - Yield Market: a system to enable efficient yield trading through trading OTs and YTs with any token with relatively low slippage
    - Pendle Governance: a system to
        - Incentivise liquidity providers for PendleV2 Yield Trading
        - Give utility to the PENDLE token through incentive boosting and voting on incentive allocation
- Although the last 3 components (PendleV2 Yield Splitter, PendleV2 yield Trading and Pendle Governance) are specific to yield trading and other applications on top of yield trading, LYT and LYT Depository are at the general infrastructure level for DeFi that could have a wide range of other applications.

LYT Overview

- A LYT is a yield bearing token that constantly accrues yield to its holder
- Any mechanism that generates yield can be made into a LYT, including
    - Lending position in a money market (cToken, aToken)
    - Farming position in a vault (Yearn, Harvest, Ribbon)
    - Liquidity provision position in an AMM (UniswapV2, UniswapV3, Curve, KyberDMM)
    - Staking position in a rebasing currency (Ohm, Wonderland)
    - Liquid staking position (Lido)
    - Liquidity farming position (Sushi Onsen, protocols' pool2)
- With a standard behaviour, LYT can be easily integrated into any other protocols or products that work with a yield bearing token.
- Detailed specs

LYT Depository Overview

- Holds LYTs for anyone who wants to keep a LYT balance
- Allows users to flashloan LYTs
- Will be used by PendleV2 Yield Splitter and PendleV2 Yield Market
- Detailed specs

Yield Splitter Overview

- Any LYT can be splitted into YTs and OTs, with respect to an expiry
- YTs get all the yield from the LYT until the expiry
- OTs have the right the redeem the initial capital after the expiry
- Detailed specs

Yield Market Overview

- There are 4 components in PendleV2's Reward Market:
  - OT AMM: An AMM that enables efficient trading between a LYT and its corresponding OT
  - YT Pseudo-AMM: A system to enable trading between a LYT and its corresponding YT, while ultilising the OT AMM behind the scene
  - LYT Trading Router: A routing system to trade a LYT with any other token, through other AMMs
  - Yield Market Router: The user facing router to trade OT/YT against any other tokens, with support for advanced order types like limit orders
- Detailed specs

Pendle Governance Overview

- Follows a similar mechanism to veCRV, PENDLE tokens could be locked up for up to 4 years to mint xPENDLE
- xPENDLE can be used to boost one's liquidity incentives in all of PendleV2's OT pools
- xPENDLE can also be used to vote for the allocation of PENDLE incentives to the different pools
- Detailed specs

2. LYT

2.1 LYT overview

- LYTs are Yield Generating Tokens, which is another more general standard for tokens that generate other tokens, as users hold it over time
- LYTs are basically a standard for any reward generating mechanism in DeFi

2.2 Yield Generating Tokens

Interface for Yield Generating Token:

- `address[] rewardTokens`
  - List of reward tokens that a user will get by holding the YGT
- `claimReward(bool[] isClaiming, address user)`
- `delegateRewardDestination(address)`
- `accruedRewards(address user) returns uint256[]`
- All the ERC20 interfaces

2.3 Generalising DeFi's reward generating mechanisms

2.3.1 Overview

- There are multiple protocols/strategies for generating reward in DeFi:

    - Lend tokens to a money market (Compound/Aave)
    - Provide liquidity to an AMM
    - Stake some tokens to farm some incentive tokens (all liquidity mining programs)
    - Stake some tokens to a strategy/vault to get more of the same tokens, or other tokens (Yearn, Harvest, Autofarm,...)
- Let's try to generalise all these reward generating mechanisms into a general framework

2.3.2 Let's define some basic terms

For each reward generating mechanism, let's define the following terms:

- **Deposit assets**: the asset(s) that are deposited, to generate the reward
    - E.g: In providing liquidity to Uniswap, the **deposit assets** are the two pool tokens
- **Accounting asset**: the asset whose balance is used to calculate the reward
    - In most cases, the **accounting asset** is the same as the **deposit asset**
        - E.g: Compound, Aave, liquidity mining,...
    - In providing liquidity to an AMM, the **accounting asset** is usually the **liquidity**
- **Reward assets**: the assets that are gained from having a position of the **accounting asset in the protocol over time**

2.3.3 The two types of reward generating mechanisms:

1. **Accounting assets** generates more **accounting assets** over time (with auto compound)
    - In other words, the reward asset is the same as the **accounting assets**, and it's compounded into the **accounting assets** balance of the user
    - Let's define some more terms:
        - **pool shares**: an asset that the user gets after depositing **accounting assets**, that represents their share in the reward generating pool
            - As the pool generates more reward in terms of **accounting assets**, 1 pool share can exchange for more **accounting assets**
            - E.g:
                - In Compound: pool shares = cDAI
                - In AaveV2: pool shares = `scaledBalanceOf`
                - For UniswapV2 LP: pool shares = LP tokens
        - **exchange rate**: how many **accounting asset **is 1 pool share worth
            - = total amount of **accounting assets** in the pool / total supply of pool shares
            - E.g:
                - In Compound: `exchangeRateCurrent`
                - In Aave: normalised income
                - In UniswapV2: total liquidity / totalSupply of Lp token
    - How is reward calculated ?
        - When user deposits **deposit assets**:
            - Calculate how much **accounting assets** are equivalent to the **deposit assets**
            - Mint **pool shares** to user according to current **exchange rate**
        - When user withdraw:
            - Calculate how much **accounting assets** is the **pool share** worth, based on current **exchange rate**

        - Convert the **accounting assets** to **deposit assets**
2. **Accounting assets** generate other reward assets over time (simple interest)
    - The pool generates some **reward assets** over time
    - The **reward assets** are distributed equally among the users, proportionally to their **accounting asset** balance

2.4 LYT's interface

LYT has all the interfaces from YGT:

- `address[] rewardTokens`
- `claimReward(bool[] isClaiming, address user)`
- `delegateRewardDestination(address)`
- `accruedRewards(address user) returns uint256[]`
- All the ERC20 interfaces

With some additional interfaces:

- `uint256[] rewardIndexes`
- `uint256 exchangeRate`
- `address depositAssets[]`
- `mint(uint256[] depositAssetAmounts, address to) returns (uint256 poolShares)`
- `burn(uint256 poolShares) returns (uint256[] depositAssetAmounts)`
- `updateAccounting(bool forceUpdate)`
    - Update global reward accounting (exchange rate and rewardIndexes)
    - if `forceUpdate` is true, bypass the caching mechanism and always update the indexes

2.5 How LYT works

- When depositing **deposit assets**, the amount of LYT minted will be the same as the pool share of the reward generating mechanism
- When withdrawing **deposit assets**, the amount of **accounting assets**-equivalent the user gets will be proportional to their pool shares
- As such, LYT will be worth more of the **accounting asset** overtime
    - This also means that when transfering LYTs, the compounded reward on the accounting assets is transfered together with the tokens
- For the other reward tokens (with simple interest), they are accrued to each user proportionally to their pool shares
    - When transfering LYTs, the accrued reward tokens stay with the sender

2.6 Caching of rewards

- In the actual implementation, there could be settings for a `cachingThreshold` percentage for the exchange rate and for each rewardIndex
- If the exchange rate or the rewardIndex doesn't exceed the threshold, doesn't update the indexes

3. LYT Depository

3.1 Overview

- This is like a bank, where any "protocol" (each identified by an address) could use to store their LYTs
  - Each "protocol" could have multiple users
- If many protocols use the LYT Depository, it will be more efficient to transfer LYTs across them (since it will just be an "internal bank transfer")

3.2 Interface

- `deposit(address lyt, address account, uint256 amount)`
  - Deposit funds for a certain account
  - Note: use balanceOf() to check the deposit amount
- `withdraw(address lyt, uint256 amount, address destination)`
  - Withdraw LYTs to the destination, from the msg.sender's account
- `flashloan(address lyt, uint256 amount, address receiver, bytes data)`
  - Flashloan an amount of lyt, to be sent to `receiver`
  - After sending the amount, call a callback function `receiver.handleFlashloan(lyt, amount, data)`
  - If the receiver is not in a whitelist, charge a fees on the flashloan

4. Yield Splitter

4.1 How it works

- When tokenising `l` LYTs which is worth `a = l * lyt.exchangeRate()` accounting assets, we will mint `a` OTs and `a` YTs
- Each YT is a Yield Generating Token
  - `address[] rewardTokens`
    - The rewardTokens array here is lyt.rewardTokens with another extra item which is the lyt token itself
    - Let's call lyt the **primary reward token** and the other reward tokens as **secondary reward tokens**
  - `claimReward(bool[] isClaiming, address user)`
    - The `isClaiming` array has another element for the lyt reward
  - `delegateRewardDestination(address)`
  - `accruedRewards(address user) returns uint256[]`
  - All the ERC20 interfaces
- Accounting for YT's rewards:
  - For the **secondary reward tokens**:
    - Save `lastRewardIndexes` for each user
    - Before transfering YT: accrue the secondary reward tokens to the sender and receiver
      - accruedRewards[user][rewardTokenIndex] += ytBalance * (currentRewardIndex - lastRewardIndex)
    - Save lastRewardIndexesBeforeExpiry to use as the estimated rewardIndexes at the expiry
  - For the **primary reward token**:
    - Save `lastExchangeRate` for each user

        - Before transfering YT: accrue the primary reward token to the sender and receiver
            - `accruedRewards[user][primayRewardTokenIndex] += ytBalance * (currentExchangeRate/lastExchangeRate - 1) / currentExchangeRate`
        - Save lastExchangeRateBeforeExpiry to use as the estimated exchangeRate at the expiry

5. Yield Market

5.1 OT AMM

5.1.1 Overview

- This is a an AMM for trading OT against its corresponding LYT
- The AMM design aims to achieve a few things
    - It's capital efficient, which allows for trading relatively large size with low slippage
    - It preserves a consistent interest rate over time (interest rate continuity), if no trades happen
        - It means that people should only trade with the AMM if they think the interest will change, and not because of other factors like underlying asset price or time
    - It will dynamically change its formula over time to approach a constant sum formula (1 OT = 1 accounting asset) after the expiry
    - It allows for explicitly setting the tradeoff between capital efficiency and the reasonable trading range

5.1.2 Virtual accounting asset balance

- Although the actual tokens sitting in the AMM's account are OT and LYTs, from the AMM's perspective, we will pretend that we have OT and accounting assets instead.
- As such, this is really an AMM for trading OT against accounting assets
- In terms of how the AMM's logic works, at any point in time, we will pretend that we have `a` accounting assets and `o` OTs, where `a = lytBalance * lyt.exchangeRate()`
- Although `a` will increase by itself, we will treat it as if it's just somebody sending accounting assets into the AMM.

5.1.3 The parameters

- There are 3 important parameters in the AMM:
    - `f_scalar0`: a scalar factor to adjust the initial capital efficiency (by adjusting the slope of the exchange rate graph)
    - `r_anchor0`: an initial anchor rate to anchor the initial AMM formula to be more capital efficient around that interest rate
    - `f_fees0`: the initial fees factor

5.1.4 The definitions

- Time left until expiry

$$t = \frac{timeToExpiry}{contractDuration}$$

- The `p` parameter

$$p = \frac{otAmount}{otAmount + accountingAssetAmount} = \frac{o}{o + a}$$

- Scalar factor

$$f_{scalar} = \frac{f_{scalar0}}{t}$$

- The marginal exchange rate, which is basically the marginal price of accounting assets in OTs

$$marginalExchangeRate = e_{marginal} = \frac{1}{f_{scalar}} \times ln(\frac{p}{1-p}) + r_{anchor}$$

- The marginal interest rate

$$r_{marginal} = (e_{marginal} - 1) \times \frac{1year}{timeToExpiry}$$

- The liquidity fees

$$f_{fees} = f_{fees0} \times t$$

5.1.5 Adjusting r_anchor for interest rate continuity

- After every trade, we will save the marginal interest rate post-trade as `lastRate = r_last`
- Before every trade, we will adjust `r_anchor` such that the pre-trade marginal interest rate will be exactly the same as `lastRate`
    - We calculate the new marginal interest rate pre-adjustment `r_beforeAdjustment` (using old anchor rate `r_anchorOld`)
    - Then, we calculate the newly adjusted anchor rate `r_anchorNew`

$$r_{anchorNew} = r_{anchorOld} - (r_{beforeAdjustment} - r_{last}) \times \frac{timeToExpiry}{1year}$$

5.1.6 Swapping logic

- Let's say the current reserve has `o` OTs and `a` accounting assets
- Let's say a user Alice wants to swap in `d_o` OTs (which could be negative)
- First, we adjust the anchor rate as per the previous section
- Then, we calculate the `p` parameter for the trade, which we call `p_trade`

$$p_{trade} = \frac{o + d_o}{o + a}$$

- Then, we calculate the exchange rate according to this formula:

$$e_{trade} = \frac{1}{f_{scalar}} \times ln(\frac{p_{trade}}{1 - p_{trade}}) + r_{anchor} \pm f_{fees}$$

  - The fees is added if `d_o` is positive, and subtracted if it's negative

- Then, we simply calculate `d_a` using the `e_trade`

$$d_a = -\frac{d_o}{e_{trade}}$$

- After the trade, we calculate and save the `r_last`

5.1.7 Adding/removing liquidity

- For the very first liquidity addition:
  - The user can set how much OT and accounting assets to bootstrap the liquidity with
  - The user will be given back the same amount of LP tokens as the accounting asset amount
- Liquidity is added/removed proportionally in terms of OT and accounting assets
- As such, we need to convert accounting assets to and from LYT accordingly

5.1.8 TWAP for OT prices

- Use the same approach as UniswapV3 to store the culmulative sums of `price * time` in an array
- Link to Uniswap docs: https://uniswap.org/blog/uniswap-v3#advanced-oracles

5.2 YT Pseudo-AMM

- This is a routing mechanism to let users trade YT against it's own LYT

1. Swap exact `YIn` YT to LYT:
   - Flashloan some LYT
   - swap exactly `YIn` OT out using the OT AMM
   - Redeem YT + OT to get LYT
   - Repays flashloan
2. Swap YT to get exact `L` LYT?
   - Flashloan a bunch of LYT
   - Calculate how much `yIn` YT to use to get exactly `L` LYT in the end
     - TODO: Solve equations for `yIn`
   - Follow the same steps as 1
3. Swap LYT to exact `YOut` YT
   - Flashloan a bunch of LYT
   - Tokenise LYT to get exactly `YOut` YT and `YOut` OT
   - Swap exact `YOut` OT in to get LYT
   - Transfer LYT from user to cover flashloan
4. Swap exact LYT to get YT

    - Flashloan a bunch of LYT
    - Calculate how much `yOut` YT to mint from LYT
        - TODO: Solve equations for `yOut`
    - Follow the same steps as 3

5.3 LYT Trading Router:

- This is a routing system to swap LYT against any token
- We will be using a number of other AMMs, which could include
    - UniswapV2 clones [to be supported first]
    - UniswapV3
- Each of these AMMs will be wrapped to have these two interfaces:
    - SwapExactIn(address tokenIn, address tokenOut, uint256 amountIn, address[] path)
    - SwapExactOut(address tokenIn, address tokenOut, uint256 amountOut, address[] path)
- To swap LYT to ANYTOKEN:
    - Steps:
        - Redeem LYT to get its deposit assets
        - Swap the deposit assets to ANYTOKEN
    - Swap exact LYT to ANYTOKEN:
        - Just get the deposit tokens and SwapExactIn each of them into ANYTOKEN
    - Swap LYT to exact `a` ANYTOKEN:
        - If there are more than one deposit assets, we need to calculate the `a1` = amount of ANYTOKEN from deposit asset 1, and `a2` = amount of ANYTOKEN from deposit asset 2
            - Such that, `a1 + a2 = a` and the amounts of deposit asset 1 and deposit asset 2 needed to SwapExactOut to `a1` and `a2` are of the same proportion as the LYT's composition
            - How ? Either by closed form formula or binary searching
- To swap ANYTOKEN to LYT:
    - Steps:
        - Swap ANYTOKEN to the deposit assets
        - Mint the LYT with the deposit assets
    - Swap exact ANYTOKEN to LYT:
        - Similarly, if there are more than one deposit assets, we needs to calculate the proportion of ANYTOKEN to swap to the deposit assets
            - Either by closed form formula or binary searching
    - Swap ANYTOKEN to exact LYT:
        - Just SwapExactOut to get the deposit assets to mint the exact amount of LYT

5.4 Yield Market Router:

- This is the user facing router to execute all the trades
- Swapping logic:
    - For ANYTOKEN vs YT, we just swap ANYTOKEN vs LYT and LYT vs YT
    - For ANYTOKEN vs OT, we just swap ANYTOKEN vs LYT and LYT vs OT
- Order settings:
    - Slippage:

        - We can add slippage settings in the swap functions, and revert when they are not satisfied
            - For swapping exact in: have a `minOutAmount` setting
            - For swapping exact out: have a `maxInAmount` setting
    - Deadline:
        - Beyond a certain timestamp, the trade will auto revert
    - [Nice to have, to be added later] Limit orders using 0x:
        - A user can sign a limit order on 0x to trade YT/OT against anything
        - Arbitragers can trigger these limit orders (if they are within the range of the AMM) to get the spread
        - Market makers can use these limit orders to market make
    - [Nice to have, to be added later] Pendle native limit order based on implied yield
        - A user can sign a limit order to trade when implied yield is lower/higher than a value
        - Arbitragers can trigger these limit orders (if they are within the range of the AMM) to get the spread
    - TODO: Study how 1inch does their limit orders

6. Pendle Governance

6.1 xPENDLE

- xPENDLE is minted by locking PENDLE for a duration (from 1 week to 4 years)
    - `xPendleMinted = pendleLocked * lockingDuration / fourYears`
- xPENDLE balance decays linearly over time
    - Each user's xPENDLE position is represented by `a` and `b`, which are the parameters for the line representing their xPENDLE balance over time
        - `w_user(t) = b - a * t`
- Actions user can do:
    - Create new lock:
        - This is when a user starts locking for the first time
        - We initiate a new `a` and `b` for them
    - Increase amount:
        - User adds more PENDLE to their current locking schedule
        - Update `a` and `b`
        - Mint extra xPENDLE (equal to increase in b)
    - Increase unlock time:
        - Update `a` and `b` to reflect longer unlock time
        - Mint extra xPENDLE (equal to increase in b)
    - Withdraw:
        - When `w_user(t)` is <= 0, give PENDLE back to the user
- Total xPENDLE supply:
    - The total supply `W(t) = B - A * t` where B is the sum of all `b` and `A` is the sum of all `A`
    - We just keep track of `A` and `B`

6.2 Liquidity Mining pools' PENDLE allocation

- There is a fixed rate `R` of PENDLE per second for liquidity mining

- There are multiple liquidity mining pool types, each identified by a string id (bytes32)
- Each pool type has a weight `w_poolType`
- There are multiple pools in each pool type, each identified by the pool address
- Each pool has a weight `w_pool`
- Each pool will get PENDLE incentives at a rate proportional to `w_poolType * w_pool`

6.3 Voting on liquidity mining pools

- xPENDLE holders can allocate their vote among a number of pools
- `w_poolType` and `w_pool` will be decided based on the xPENDLE votes, on a weekly basis

6.4 Boosted liquidity incentives for xPENDLE holders

- Within each pool, a user will have a balance of `b_u` liquidity, and `w_u` xPENDLE
- let
  - `B = sum of all b_u` in the pool
  - `W = sum of all w_u` in the pool
  - `b_u_boosted = min(0.4 * b_u + 0.6 * B * w_u/W, b_u)`
- User's portion of the liquidity mining rewards will be proportional to `b_u_boosted`
- This means that the more xPENDLE one has, the more portion of liquidity rewards one can get
- One detail: `w_u` is taken at the last user action (deposit/withdraw/redeem) & applied until the next action
  - Therefore, it's good to apply a boost and do nothing until getting more xPENDLE
  - When the xPENDLE expires, anyone can "kick" the user out