Pendle Technical Specification

1. Introduction

Pendle Finance leverages on the base lending layer created by prominent DeFi protocols such as Aave and Compound, which has shown incredible growth and community acceptance. The protocol builds on this layer by separating the future cash flows from these lending protocols' yield tokens and tokenizing it. This allows future yield to be traded without affecting ownership of the underlying asset.

Ownership of the future cash flows is guaranteed by the smart contract, so there is no need to worry about collateral or counterparty risk, as long as the underlying lending protocol is not compromised.

2. Expectations on How It Works

The protocol has been designed to provide 2 main functionalities: 1) to tokenize their yield; and 2) to be able to trade that tokenized yield. We've designed the contracts where it is straightforward for users to tokenize their yield bearing tokens, producing Future Yield Tokens (XYT) and Ownership Tokens (OT) and to be able to trade them with other ERC20 tokens.

As such, here are the following characteristics of the Pendle smart contracts:

2.1 Yield Tokenization

- For tokenization functions, the user solely interacts with the core routing contract PendleRouter.sol. The PendleRouter.sol contract routes to the relevant Forge contracts for tokenization.
- Users should be able to tokenize any Yield Bearing Token (such as aTokens or cTokens) as long as they are officially supported by the underlying yield protocol, namely Aave and Compound.
- The user can deposit and lock his Yield Bearing Token into a Forge, and the Forge mints Future Yield Tokens (XYTs) and Ownership Tokens (OTs), which is transferred back to the user.
- Users can create any yield contract with any expiry that is in quantums of x days (current 7 days) since UNIX timestamp 0 (1st Jan 1970)
- The user can redeem his accrued interest at any time given his XYT holdings.
- The user can redeem the underlying Yield Bearing Token from the Forge as long as they provide the equal amounts of XYT and OT tokens to redeem the specified amount of Yield Bearing Token.
- Should the yield contract of the XYT token expire, the user has the choice to renew the yield contract, setting a new expiry, or to redeem the Yield Bearing Token by only returning back the OT tokens.

- Should the yield contract of the XYT token expire, the XYT tokens themselves will still exist but will be considered valueless.

2.2 Future Yield Trading via AMMs

- Users should be able to swap between XYTs and a base token
- Users will need to only provide token allowance to the core routing contract, PendleRouter.sol, for trading.
- Pendle will permissionlessly support the creation of markets of all legitimate XYTs in the network.
- Users can add liquidity to any Market permissionlessly and get back an LP token as a representation of their stake.
- Users can add liquidity for a Market for a single token or for both tokens.
- Users are still able to earn and redeem interest even if the user has used those XYTs to contribute liquidity into a Market.
- Users can withdraw their liquidity at any point as long as they provide the specific LP token for that Market to redeem their stake.
- Users will pay a 0.35% swap fees
- Pendle protocol will charge 1/7 of the 0.35% swap fees, that goes to the Treasury

3. Scope of current contract system

- Governance module has not been developed. Gnosis Safe Multisigs will be used as the governance role until the Governance module is deployed
- Emergency handler contracts have not been deployed. Gnosis Safe Multisigs will be used as the emergency handlers in the mean time.

4. Protocol Overview

The protocol consists of several smart contracts that covers operations on routing, tokenization of future yields (Forge), trading on future yield AMMs (Markets), and Governance. Only the Forges and Markets hold user funds, the user is able to redeem back his funds at any point of time. The user will interact with only PendleRouter.sol.

4.1 PendleRouter.sol

The main entry point for users to perform tokenization operations and to perform trades across the different future yield markets. The Pendle contract itself acts as a router and communicates with other Pendle contracts.

4.2 PendleData.sol

Stores the data across forges and markets.

4.3 PendleForgeBase.sol and Pendle*Forge.sol

PendleForgeBase.sol implements the common logic for a forge, while the Pendle*Forge.sol contracts implements the specific implementation for Aave, AaveV2 and Compound

4.4 PendleMarketFactoryBase.sol and Pendle*MarketFactory.sol

Used to create new market contracts. Pendle*MarketFactory.sol are specific market factories for Aave and Compound, extending from the common PendleMarketFactoryBase.sol

4.5 PendleMarket.sol

4.5.1 AMM Model

The Pendle AMM follows the formula:

$$x^{\alpha_{i+1}} y^{\beta_{i+1}} = x_i^{\alpha_{i+1}} y_i^{\beta_{i+1}}$$

where
α: $the\ weight\ of\ x\ at\ time\ = i + 1$
β: $the\ weight\ of\ y\ at\ time\ = i + 1$
x_i and y_i: $the\ equilibrium\ point\ of\ x\ and\ y\ at\ time\ = i$

At time = 0, ɑ and β are initiated at 0.5, the quantity of token_x and token_y is initiated by the market creator which translates to x_0 and y_0. The curve will be similar to that of the Uniswap's constant product curve.

When a swap happens, the equilibrium point of token_x and token_y shifts along the curve.

At subsequent time step time = *i*, ɑ and β will change with the following formulas:

$$\alpha_{i+1} = \alpha_i - \epsilon_i$$

$$\beta_{i+1} = \beta_i + \epsilon_i$$

where

$$\epsilon_i = \frac{\alpha_i \beta_i (1 - R_i)}{R_i \alpha_i + \beta_i}$$

$$R_i = \frac{p(t_{now})}{p(t_{eq})}$$

$$p(t) = \frac{ln(3.14 \cdot t + 1)}{ln(4.14)}$$

$$t(T) = \frac{T_{end} - T}{T_{end} - T_{start}}$$

$$t_{now} = t(T_{now})$$

$$t_{eq} = t(T_{eq})$$

t: *time to maturity from* 1 *to* 0

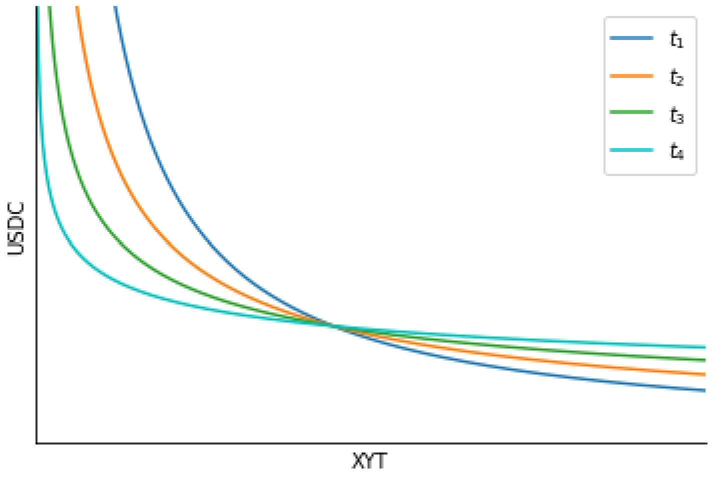
T: *UNIX timestamp*

T_{start}: *contract start timestamp*

T_{end}: *contract end timestamp*

$T_{end} - T_{start}$ *is coded as* $3600 * 24 * 180$ (*representing* 180 *days long contracts*)

The changing weights of token_x and token_y changes the shape of the curve.



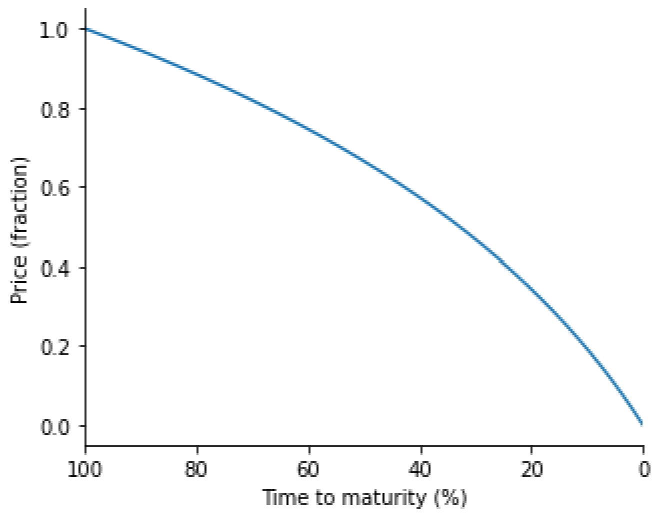
The pivot point for curve shifting is the equilibrium point of token_x and token_y at the time of curve shift. Swapping between token_x and token_y changes the equilibrium point, and the pivot point for curve shift will vary accordingly.

At time close to T_{end}, the slope of the curve will be near zero. And reaches zero at T_{end}.

4.5.2 Curve Shifting Algorithm

The Pendle curve shifting algorithm closely resembles the concept of option/bond pricing models, where price decays more drastically towards the end of the contract than at the beginning. The magnitude of price decay over time is represented by the *p(t)* formula outlined in the section above.

The motivation for the constants chosen for the above formula is the fact that the price loses about 1/3 of its original value (at the beginning of the contract period) when time to maturity reaches 50%.



The function of price (fraction) w.r.t. time remaining to maturity

Swap actions through the Pendle AMM will trigger curve shifts, as detailed in swapAmountOut and swapAmountIn functions.

The `_updateWeight()` function outlines the curve shifting formula.

The `_curveShift()` function will check if it has been more than curveShiftBlockDelta, to do the curveShift (which will only be done once in a block, before any swapping occurs)

With curve shifting called for any swap action and the addition of limiting a maximum of one `_updateWeight()` per block, the profitability of conducting a sandwich attack will be diminished. There are two main reasons behind this, the frequency of curve shifts will be relatively high (we are planning to set it to be roughly once a day), as such the magnitude of every curve shift is smaller than otherwise. On top of that, `_updateWeight()` will only be called by the first swap transaction of a block, mitigating the possibility of a one transaction flashloan sandwich attack.

4.5.3 Liquidity provision (LP) token calculation

Taking inspiration from Balancer, for a generic `joinMarketByAll()`, the LP tokens received by the depositor is the ratio of deposited tokens to the current reserves of the token in the pool. In other words:

$$inAmount \ = (\frac{totalLp + \ outAmountLp}{totalLp} - 1) \cdot B_{in}$$

derives to

$$outAmountLp \ = \ (\frac{inAmount}{B_{in}}) \ \cdot \ totalLp$$

As mentioned, the swapFee variable will have to be introduced in the case of `joinMarketSingleToken()`. In other words, the formula for LP token calculation for single token pool join is then (adapting from Balancer):

$$outAmountLp \ = \ totalLp \ \cdot \ ((1 + \frac{(inAmount - inAmount \cdot (1-W_{in}) \cdot swapFee)}{B_{in}})^{W_{in}} - 1)$$

where

B_{in}: $inTokenReserve.balance$

W_{in}: $inToken\ Reserve.weight$

4.5.4 Liquidity withdrawal calculations

Without considering the `swapFee`, liquidity withdrawal formula is the inverse of that provision (without swapFee considerations).

With `swapFee`, adapting from Balancer, the formula for `outAmountToken` given inAmountLp is:

$$outAmountToken \;=\; B_{out} \cdot (1 - (1 - \frac{inAmountLp}{totalLp})^{\frac{1}{W_{out}}}) \cdot (1 - (1 - W_{out}) \cdot swap$$

where
B_{out}: $outTokenReserve.balance$
W_{out}: $outTokenReserve.weight$

4.5.6 Swapping

Swap function consist of several parts:

1. `spotPrice`:
   `spotPrice` is a crucial part of the calculation for swapping. The calculation of spot price is implemented as `_calcSpotPrice` and it takes the following formula:

$$spotPrice \;=\; (\frac{B_{in}}{W_{in}} \div \frac{B_{out}}{W_{out}}) \cdot (\frac{1}{(1-swapFee)})$$

2. `maxPrice`:
   `maxPrice` is the maximum price before the transaction is reverted. This is set by users as the ‘slippage tolerance’ multiplied by spotPrice.

3. `swapAmountIn` or `swapAmountOut` (taking reference from Balancer):
   a. `swapAmountIn`:
      `swapAmountIn` returns outAmount of the output token, which utilizes the following formula:

$$outAmount = B_{out}(1 - (\frac{B_{in}}{B_{in} + inAmount \cdot (1 - swapFee)})^{\frac{W_{in}}{W_{out}}})$$

   b. `swapAmountOut`:
      `swapAmountOut` returns the inAmount of the input token, which utilizes the following formula:

$$inAmount = B_{in} \cdot ((\frac{B_{out}}{B_{out} - outAmount})^{\frac{W_{out}}{W_{in}}} - 1) \cdot (\frac{1}{(1 - swapFee)})$$

4.5.5 Fees

There are two implemented fees in the Pendle contracts:

swapFee

> `swapFee` is equivalent to the liquidity provider fee when traders execute swaps on the Pendle AMM. It will be implemented as a constant 0.35% across all liquidity pools until the Pendle governance is enacted for future modification. Within this swapFee, the Pendle protocol will take 1/7 as protocol fees, which is taken as LP tokens and sent to the Treasury address.

forgeFee

> Every interest payment for XYT holders will be subject to a forge fee of 3% to the protocol. The governance can withdraw this fee amount anytime to the Treasury address

4.5.6 Market Locking

A market will enter a locking phase right before expiry, determined by `setLockParams`. `setLockParams` sets the `lockNumerator` and the `lockDenominator`.

`lockDuration` is then calculated by `contractDuration * (lockNumerator / lockDenominator)`. `lockStartTime` will be initiated when the new market is bootstrapped, calculated by `contractExpiry - lockDuration`. When `currentTime > lockStartTime`, the market then enters the said locking phase.

During the locking phase, most functions will be disabled, including:

- `addMarketLiquidityDual`
- `addMarketLiquiditySingle`
- `removeMarketLiquiditySingle`
- `swapExactIn`
- `swapExactOut`

The only function that is not disabled is

- `removeMarketLiquidityDual`

4.6 PendleFutureYieldToken.sol

This contract inherits from the ERC20 standard, thus implements all ERC20 functions (both required and optional). In addition, it has implemented mint() and burn() functions as well for increasing or decreasing the token supply.

4.6.1 Extension

- Stores a reference to the Forge that minted this XYT token.
```
  address public override forge;
```

- Stores a reference to the underlying asset (e.g. USDT) that was used as a basis to create the Yield Bearing Token.
```
  address public override underlyingAsset;
```

- Stores a reference to the underlying Yield Bearing Token (e.g. aUSDT) that was used to tokenize and mint this XYT token.
```
  address public override underlyingYieldToken;
```

- A function that's called before any token transfer, to settle due interests on the current holder before transfer.
```
  function _beforeTokenTransfer(address from, address to) internal
override {
```

4.7 PendleOwnershipToken.sol

Similar to PendleFutureYieldToken.sol, this contract inherits from the ERC20 standard, thus implements all ERC20 functions (both required and optional). In addition, it has implemented mint() and burn() functions as well for increasing or decreasing the token supply.

4.7.1 Extension

- Stores a reference to the Forge that minted this XYT token.
```
  address public override forge;
```

- Stores a reference to the underlying asset (e.g. USDT) that was used as a basis to create the Yield Bearing Token.
```
  address public override underlyingAsset;
```

- Stores a reference to the underlying Yield Bearing Token (e.g. aUSDT) that was used to tokenize and mint this XYT token.
```
  address public override underlyingYieldToken;
```

4.8 PendleLiquidityMining.sol

The Pendle liquidity mining mechanism cover three main aspects:

1. Users to stake PendleLP to be eligible for Pendle liquidity mining rewards
2. Mining rewards distribution will be tabulated at the end of 7 day epochs, which claiming will become available
3. The entitled rewards will be vested across 5 epochs (including the staking epoch) with 20% released at the end of each epoch

One staking contract will be created for an underlyingYieldBearingToken against a baseToken across multiple expiries. To illustrate,

- Same staking contract for *XYT-aDAI-13Dec2020 / USDT* and *XYT-aDAI-26Mar2021 /USDT*
- Separate staking contracts for *XYT-aDAI-13Dec2020 / USDT* and *XYT-aDAI-13Dec2020 / ETH*
- Separate staking contracts for *XYT-aDAI-13Dec2020 / USDT* and *XYT-aUSDC-13Dec2020 / USDT*

4.8.1 LP Token Staking and Withdrawal

- Stakes the LP token specifying the XYT contract expiry and exact amount. Also note that any staking action will settle the previously accrued, available for claiming but is unclaimed liquidity incentives for the staker.

```
  function stake(uint256 expiry, uint256 amount) external returns
(address);
```

- Withdraws the LP token specifying the XYT contract expiry and exact amount. Also note that any staking action will settle the previously accrued, available for claiming but is unclaimed liquidity incentives for the withdrawer.

```
  function withdraw(uint256 expiry, uint256 amount) external;
```

4.8.2 Epoch based distribution

The liquidity incentive distribution mechanism includes a 7-days-epoch system. Users are free to stake or withdraw LP tokens from the staking contract during the epoch.

4.8.3 Rewards vesting

The accrued rewards in each epoch will be vested in equal proportions across 5 epochs (current epoch + 4 subsequent epochs).

5. Actors

`Governance`

Pendle Finance contracts where user funds are not involved, and where certain parameters can be changed, is governed by a DAO — specifically the PendleGovernance DAO contract.

On initial launch, Governance will be represented by an M of N multisig. However, once the DAO contracts are ready, the DAO will replace the multisig and take its place as the governing entity of Pendle Finance.

`EOAs/Contracts`

Can call any external/public view/pure functions on any of the Pendle contracts. However, when it comes to state changing functions, they are only able to interact with the PendleRouter and Liquidity Mining contracts for actions such as future yield tokenization, trading and staking.

6. Trust Model

↓ Trusts →	Pendle Finance	Yield Protocols	ERC20 Tokens	EOA / Contracts	Integrations
Pendle Finance	■	YES	YES	NO	NO
Yield Protocols	NO	■	NO	NO	NO
ERC20 Tokens	NO	NO	■	NO	NO
EOA / Contracts	YES	YES	YES	■	YES
Integrations	YES	YES	YES	NO	■

6.1 Pendle Finance

The Pendle Finance contracts need to trust:

- The yield bearing protocols that it chooses to integrate (i.e. Aave, Compound Finance, Harvest Finance), as it needs to trust that the yield protocols execute the forwarded requests such as interest redemption, return the correct information when queried, and to not act malicious
- The ERC20 tokens that are added by users into the Pendle Markets, as Pendle cannot verify that the token contracts have implemented the ERC20 standard in a non-malicious way. The immediate risk, however, should be isolated within the Pendle Market AMM contract where the ERC20 token is provided as liquidity.

6.2 Yield Protocols

Yield protocols such as Aave, Compound Finance, and Harvest Finance do not need to trust Pendle Finance or any actors in Pendle's ecosystem as it treats Pendle like any other user of its platform.

6.3 ERC20 Tokens

ERC20 tokens such as USDT, DAI, or WBTC do not need to trust Pendle Finance or any actors in Pendle's ecosystem as it treats Pendle like any other user of its platform. However, protocols such as Pendle Finance and users need to fully trust the ERC20 token to not unexpectedly revert valid token transfers or to not act in any malicious way.

6.4 EOA / Contracts

Externally owned accounts and 3rd party contracts need to trust:

- Pendle Finance to execute operations as intended.
- The yield bearing protocols as the user can be both a direct and indirect user of the yield protocol within the use of Pendle Finance. Therefore, as a user of Pendle Finance, the user expects the integrated yield protocol to not act maliciously as Pendle will have very little protections in place should the integrated yield protocol be compromised.
- The ERC20 tokens that are being used to add liquidity to a Pendle Market or for trading. The user needs to trust that the token has implemented the ERC20 standard with no malicious code that can result in the user losing funds.
- 3rd party integrations that have integrated Pendle Finance and where the user is using the integration instead (e.g. a Wallet, Aggregation Service, etc.). Pendle Finance has no control on what the 3rd party integrations do when they integrate Pendle Finance, and so the user will need to trust that they do not act maliciously.

6.5 Integrations

3rd party Dapps that have integrated Pendle Finance need to trust:

- Pendle Finance to execute operations as intended.
- The yield bearing protocols as the dapp indirectly uses the yield protocol along with the integration of Pendle Finance.
- The ERC20 tokens that are being supported by Pendle Finance for trading. The dapp needs to trust that the token has implemented the ERC20 standard with no malicious code that can exploit their own platform.