Underwriter Vault - Specification

Premia.Finance

April 6, 2023

1 Introduction

The purpose of the creation of the underwiter vault is to migrate users from Premia V2 to Premia V3. The `UnderwriterVault` satisfies the `ERC4626` vault standard and is enables `yearn` compatibility.

1.1 General overview and functionality

The vault's purpose is to cluster liquidity that will be used to underwrite options at prices which are close to Deribit's. The depositor's liquidity will be used to underwrite call and put options for a wide set of strikes and maturities.

1.2 User Stories

- As a depositor, I want to deposit collateral into the vault so that I can receive shares that can later be redeemed for the premium and spread that is made by selling options to buyers in addition to my original deposit.
- As a depositor, I want to be able to withdraw from a vault when there is capital available so that I can recover my collateral and realize the pro-rata P&L from the time spent in the vault.
- As a buyer, I want to receive a quote for an option purchase, so that I can know in advance what the premium will be for purchasing the options so that I can compare the cost with other outlets.
- As a buyer, I want to purchase options from the vault by paying a premium, so that I can receive long contracts.
- As a vault operator, I want to settle all options up until the current time, so that I increase the vault's liquidity / available assets, such that it will be available for depositors to withdraw or for buyers to purchase new options.

2 Algorithmic description

In the following sections, we introduce the storage variables and their update rules upon external calls that are necessary to maintain the vault.

2.1 ERC4626 state variables

Variable Name	Notation	Purpose
`totalAssets`	A	The total assets the vault holds is defined as the aggregate of the assets deposited, the premiums collected, the ask spreads earned and the collateral locked into short positions (`totalLockedAssets`). It is the sum of the `balanceOf` of the vault's reference asset and the `totalLockedAssets`.
`totalSupply`	S	`ERC20` total supply of the vault's shares. The total circulating supply of shares. This number can only increase upon a deposit (mint) or decrease upon a withdrawal (burn).

Table 1: Vault-specific state variables stored on-chain.

2.1.1 Updating `totalAssets` and `totalSupply`: LP interactions

`deposit()` User deposits λ amount of assets.

$$A_{n+1} \leftarrow A_n + \lambda$$
$$S_{n+1} \leftarrow S_n + \lambda p(t)^{-1}$$

where $p(t)$ is the `pricePerShare()` at the current time t defined in **??**.

`withdraw()` User withdraws λ amount of assets.

$$A_{n+1} \leftarrow A_n - \lambda$$
$$S_{n+1} \leftarrow S_n - \lambda p(t)^{-1}$$

`mint()` User mints α amount of shares using $\alpha p(t)$ amount of assets.

$$A_{n+1} \leftarrow A_n + \alpha p(t)$$
$$S_{n+1} \leftarrow S_n + \alpha$$

`redeem()` User redeems α amount of shares and receives $\alpha p(t)$ amount of assets.

$$A_{n+1} \leftarrow A_n - \alpha p(t)$$
$$S_{n+1} \leftarrow S_n - \alpha$$

2.2 Tracking existing option listings

The vault has to keep track of the short option positions, i.e. strike and maturity combinations, it holds. For this purpose we introduce the following storage variables in Table 2.

Variable Name	Notation	Purpose
`maturities`	$\mathcal{M}$	`DoublyLinkedList` which tracks the expired and unexpired maturities of unsettled options held by the vault.
`maturityToStrikes`	$\mathcal{K}$	Maps the maturity to the set of listed strikes.
`minMaturity`	$\mathcal{M}_{\min}$	The smallest (expired or unexpired) maturity in the set of listed `maturities`. By default 0 if no listings exist.
`maxMaturity`	$\mathcal{M}_{\max}$	The largest maturity in the set of `maturities`. By default 0 if no listings exist.

Table 2: State variables to keep track of expired and unexpired listings

Variables in Table 2 are updated whenever options are settled, in which case the strike-maturity pairs are removed, or whenever a long position was sold for a strike and maturity; in which case the strike and maturity is added to the storage variables. The corresponding updates can be found in subsubsection 2.7.2 and subsubsection 2.8.3.

2.3 Accounting of locked spreads

The price quoted by the vault consists of three components (1) the fair value of the option, (2) the spread charged by the vault and (3) the minting fee paid to the pool for minting the long and short optoins

$$\texttt{quotedPrice} = \texttt{fairOptionPrice} + \texttt{spread} + \texttt{mintingFee} \ .$$

Spreads collected from underwriting options can be regarded as profits *in expectation*[1]: the vault sells an option for a price higher than the fair value of the option. Naturally, the collection of spreads would therefore lead to an increase in the price per share. This in turn opens up arbitrage opportunities by front-running trades: before a large incoming trade an LP could deposit collateral, wait for the trade to be executed and then redeem his minted shares and collect his pro-rata share of the earned spreads. To ensure a fair distribution of profits and reward LPs for participating in the vault (and not only those at the time of underwriting) the vault disperses spreads linearly over the lifetime of the option.

[1] In expectation the vault will earn the spread, however the maturity-specific PnL of the vault's short exposure is a random variable which is determined at the option's maturity.

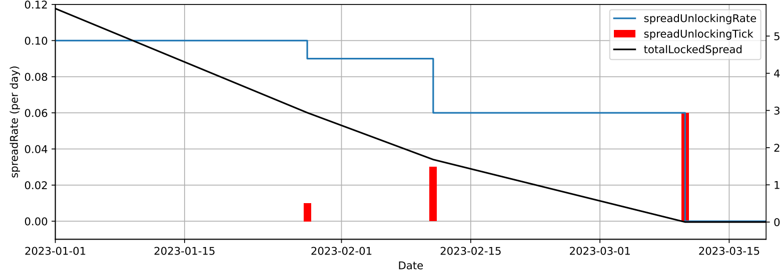

Figure 1: Adjustment of the spread unlocking rate

Example 1 (Dispersing the spread of a single option)**.** For simplicity, assume the minting fee is 0. Assume an option with a maturity of 10 days is worth 0.1 ETH and is sold for 0.11 ETH. In this case the spread of 0.01 ETH is linearly dispersed at a rate of $\frac{0.01}{10 \cdot 86400}$ ETH per second.[2]

In the accounting system *spread unlocking rates* are stored in seconds. Furthermore, ticks need to be stored to adjust the global spread dispersion rate at which the spread is unlocked. If an option expires all spreads related to that option were dispersed. Therefore the spread rate needs to be decremented everytime by the spread rate specific to an option once it expires.

Example 2 (Adjusting the spread unlocking rate)**.** For illustration and numerical ease, it is assumed in this example that rates are quoted in days and not seconds. Assume 0.28 ETH are linearly dispersed over 28 days, 1.56 over 52 days and 4.2 over 70 days. Then the aggregate spread unlocking rate is for the first 28 days 0.1 ETH per day, 0.09 for the following 14 days and 0.06 for the final 28 days. The corresponding rates are illustrated in Figure 1.

Table 3 lists the state variables required to enable linear spread dispersion.

[2]There are 86400 seconds in a day.

Variable Name	Notation	Purpose
`lastSpreadUnlockUpdate`	τ	Timestamp at which the `totalLockedSpread` and `spreadUnlockingRate` were last updated.
`totalLockedSpread`	ξ	The total amount of locked spread. To compute current amount of total locked spread the function `updateState` (see algorithm 1) has to be called.
`spreadUnlockingRate`	ϕ	Defines the aggregate rate at which the spreads are unlocked per unit of time between the `lastSpreadUnlockUpdate` (τ) and the next unexpired maturity.
`spreadUnlockingTicks`	$\phi(m)$	The amount of `spreadUnlockingRate` corresponding to options expiring at maturity m.

Table 3: State variables tracked to process the accounting of total locked spread.

Computing the current `totalLockedSpread` and `spreadUnlockingRate` Multiple maturities may have expired since last update of the spread state variables. In this case expired options would still be contained in `maturities` as they would not have been settled (calling `settle()` triggers an update of the spread state variables). are still contained in the set of maturities when those options have not been settled (explained in the next section). Algorithm 1 shows how to update the state variables relevant to spread dispersion.

In the while block starting at `Line` 4 we decrement ϕ by the spread unlocking tick $\phi(m)$ defined at maturity m as long the `block.timestamp` has crossed the maturity. To ensure that we have not crossed the last tick at `maxMaturity`, which would result in `next`$(\mathcal{M}, m)$ mapping to

2.3.1 Constructive computation of the `totalLockedSpread`

It is worth noting that the `spreadUnlockingRate` can be computed constructively using the `spreadUnlockingTicks`. The following relationship holds

$$\xi_n = \sum_{m \in \mathcal{M}} \phi(m) \max(\min(t, m) - \tau, 0)$$
$$\phi = \sum_{m \in \mathcal{M}} \phi(m) \mathbf{1}_{\{t \leq m\}}$$

where $\mathcal{M}$ is the set of maturities (unexpired and expired) and t is the current time.

Algorithm 1 Algorithm to compute current `totalLockedSpread` and `spreadUnlockingRate` by crossing the spreadUnlockingTicks.

```
 1: procedure UPDATESTATE
 2:     m ← minMaturity
 3:     t ← block.timestamp
 4:     while m ≤ t ∧ m ≠ 0 do
 5:         ξ ← ξ − φ · (m − τ)
 6:         φ ← φ − φ(m)                         ▷ "Tick crossing."
 7:         τ ← m                    ▷ Set τ to last crossed maturity.
 8:         m ← next(M, m)                 ▷ Set the next maturity.
 9:     end while
10:     ξ ← ξ − φ · (t − τ)    ▷ Decrement totalLockeSpread by the amount of
        unlocked spread since the last crossed maturity.
11:     τ ← t
12:     return {ξ, φ, τ}
13: end procedure
```

2.4 The fair value of the vault's option book

We want the vault's price per share not to be arbitrageable. For this reason we need to evaluate the value of the vault's short option positions. Options that are listed may be expired and unsettled or may be settled. The implementation therefore includes a function for expired and one for unexpired options.

`getTotalLiabilitiesExpired()` `getTotalLiabilitiesExpired()` computes the exercise value of every option that has expired and was not settled by the vault. The exercise value is the value that is owed to the option holder by the vault. Note that the exercise value is quoted in terms of the underlying for calls and in terms of the quote asset for the put.

$$\mathcal{L}^{\texttt{expired}}(t) = \begin{cases} \sum_{m \in \mathcal{M}^{\texttt{expired}}} \sum_{k \in \mathcal{K}(m)} s(k,m) \; (1 - \frac{k}{S_m})^+ & \text{if call} \\ \sum_{m \in \mathcal{M}^{\texttt{expired}}} \sum_{k \in \mathcal{K}(m)} s(k,m) \; (k - S_m)^+ & \text{if put} \end{cases}$$

where

- $s(k, m)$ is the amount of shorts with strike k maturing at m,
- S_m denotes the spot price at maturity m,
- $\mathcal{M}^{\texttt{expired}}$ denotes the set of expired maturities, i.e.

$$\mathcal{M}^{\texttt{expired}} = \{m \in \mathcal{M} : m \leq t\} \; .$$

`getTotalLiabilitiesUnexpired()` `getTotalLiabilitiesUnexpired()` computes for all unexpired option listings the option's *fair value*. "The fair value of an option is the price or premium at which both the buyer and the writer of the

option should expect to break even, neglecting the effect of commissions and other trading costs and after an adjustment for risk.” We compute an option’s fair value by using the Black-Scholes formula which uses the implied volatility derived from a parametric volatility surface.

$$\mathcal{L}^{\texttt{unexpired}}(t) = \begin{cases} \sum_{m\in\mathcal{M}^{\texttt{unexpired}}} \sum_{k\in\mathcal{K}(m)} s(k,m) \ \dfrac{\texttt{BS}(S_t,k,\tau,r,\sigma(k,\tau))}{S_t} & \text{if call} \\ \sum_{m\in\mathcal{M}^{\texttt{unexpired}}} \sum_{k\in\mathcal{K}(m)} s(k,m) \ \texttt{BS}(S_t,\tau,k,r,\sigma(k,\tau)) & \text{if put} \end{cases}$$

- r is the risk-free rate,
- t is the current time,
- τ is the time to maturity defined as $\tau = \max(m - t, 0)$[3]
- $\mathcal{M}^{\texttt{unexpired}}$ is the set of unexpired maturities

$$\mathcal{M}^{\texttt{expired}} = \mathcal{M} \setminus \mathcal{M}^{\texttt{unexpired}} ,$$

- $\sigma(k,\tau)$ is the implied volatility with time to expiration τ and strike k queried from the volatility oracle.

`getTotalLiabilities()` The vault’s total liabilities are then defined as the sum of the expired and unexpired liabilities

$$\mathcal{L}(t) = \mathcal{L}^{\texttt{unexpired}}(t) + \mathcal{L}^{\texttt{expired}}(t)$$

`getTotalFairValue()` Above we defined the fair value of the options that were sold to the option holders / counterparties. However, we are interested in the fair value of the vault’s option book $\mathcal{F}(t)$, i.e. the short positions. A single call / put option is secured by 1 unit of the underlying / k units of the quote asset.

$$\mathcal{F}(t) = -\mathcal{L}(t) + \begin{cases} \sum_{m\in\mathcal{M}} \sum_{K\in\mathcal{K}(m)} s(k,m) & \text{if call} \\ \sum_{m\in\mathcal{M}} \sum_{K\in\mathcal{K}(m)} s(k,m)\cdot k & \text{if put} \end{cases}$$

In subsection 2.7 it will become apparent that the fair value of the vault’s option book can be defined as the difference between the total locked assets and the liabtilities the vault holds.

$$\mathcal{F}(t) = L_n - \mathcal{L}(t)$$

[3] An option that is listed by the vault may have expired and not have been settled. In this case the option’s fair value is defined as the exercise value at time of maturity. See subsection 2.8 for further details.

2.5 Fee collection

tl;dr LPs are subject to management and performance fees. Management fees are paid to the vault by LPs for managing the LPs assets. Performance fees are payments made by LPs to the vault for generating positive returns. Fee collection is triggered upon a transfer either between two users or a redemption and is only paid on the transferred / redeemed amount. Thus, if a user redeems / transfers Δ shares and the total fee burden in shares is x then $\Delta + x$ shares will be deducted from the user's share balance. Fees are collected such that the the vault's price per share remains unaffected by burning the shares owed by the user upon a transfer / redemption. This is equivalent to the vault realizing fees at the current price per share, however, exposure in the corresponding asset (`base` / `quote`) is still given.

Variable Name	Notation	Purpose
`netUserDeposits`	$(\texttt{user} \mapsto d_n)$	A mapping from user addresses to the amount of collateral deposited at time of deposit. The mapping is used to calculate the average price per share at which the user minted shares at.
`timeOfDeposit`	$(\texttt{user} \mapsto t_n)$	A mapping from user addresses to the UTC timestamp (in seconds) at which the user deposited. Used to compute the correct amount of management fees owed by a user.
`managementFeeRate`	χ	The per annum ad volarem rate applied to compute management fees owed.
`performanceFeeRate`	γ	The ad volarem rate applied to compute performance fees; if a user's return was positive.
`protocolFees`	Σ	Tracks the total amount of fees collected. Reset to zero whenever fees are claimed by the deployer.

Table 4: Mappings / variables stored on-chain to compute / track management and performance fees owed to the vault.

2.5.1 Management fees

The payment of management fees is triggered through a transfer or a redemption and is only paid on the transferred / redeemed amount. Assume that the user's time of deposit is t_n and wants to transfer / redeem s shares. Let t be the current UTC timestamp in seconds. Then the management fee in shares that

is owed to the vault is defined as

$$\xi^{\texttt{user}}(s) = (t - t_n)\,\omega^{-1}\chi s$$

where ω denotes the number of seconds in a year.

Todo: not critical, but in the unlikely event that management fees are greater than the actual balance we would catch an underflow.

2.5.2 Performance fees

Denote by μ the average price per share at which the user minted shares and let s be the amount of shares the user wants to transfer / redeem. We can compute the user's return from buying vault shares as $r = \frac{\mu}{p} - 1$ where p denotes the current price per share. The user's performance fee in shares owed to the vault is then defined as

$$\gamma^{\texttt{user}}(s) = \mathbf{1}_{\{r>0\}}\gamma r s \ .$$

Thus, if the user's investment is in the money, i.e. has a positive return, then the user has to burn $\gamma^{\texttt{user}}(s)$ in order to transfer / redeem s shares. If the user is out of the money, i.e. has a negative return, the user does not have to burn any shares.

2.5.3 Total fee burden

If the user transfers / redeems s shares his total fee burden is defined as the sum of the user's management and performance fees $\xi^{\texttt{user}}(s) + \gamma^{\texttt{user}}(s)$. Note that a user's performance is independent of the management fees, i.e. the deduction of management fees does not decrease the user's performance.

2.5.4 The maximum of transferable shares

Any transaction of the `ERC20` vault shares is subject to the payment of outstanding fees. Therefore, the maximum of shares that are transferable in general (flash transactions excluded) does not equal the balance of shares held by the user. Let s denote the total balance of the user's shares. Then then maximum amount of transferable shares the user holds is defined as

$$\Xi^{\texttt{user}}(s) = s - \left(\xi^{\texttt{user}}(s) + \gamma^{\texttt{user}}(s)\right) \ .$$

2.5.5 `pricePerShare` invariance upon fee collection

Fees are collected such that the `pricePerShare` remains unaffected. This is peformed by burning the amount of shares that are due at the time of transfer and deducting the share amount valued in assets from `totalAssets`. Furthermore, `protocolFees` are incremented by the asset amount.

Example 3 (Transferring shares)**.** Assume an LP deposited collateral to mint 10 vault shares 6 months ago. The LP's performance is 125% (equivallently 25% in returns). The LP transfers 2 vault shares to another account. By doing so the LP has to burn additionally $2 \cdot 0.25 \cdot \gamma$ in shares as performance fees and $\frac{6}{12} \cdot 2 \cdot m$ shares as management fees. The total shares deducted from the LP's balance is $2 \cdot (1 + 0.25\gamma + 0.5\chi)$.

The invariance is illustrated mathematically by showing that burning $\Delta < S$ shares and deducting shares valued in assets from total assets does not change the price per share $p(t) = \frac{A}{S}$.

$$\frac{A - \Delta p(t)}{S - \Delta} = \frac{A - \Delta \frac{A}{S}}{S - \Delta} = \frac{A(S - \Delta)}{S(S - \Delta)} = p(t)$$

Thus, we obtain the following update for `totalAssets` and `totalSupply`

$$\begin{aligned} A_{n+1} &\leftarrow A_n - p(t) \cdot (\xi^{\texttt{user}}(s) + \gamma^{\texttt{user}}(s)) \\ S_{n+1} &\leftarrow S_n - (\xi^{\texttt{user}}(s) + \gamma^{\texttt{user}}(s)) \end{aligned}$$

where `totalSupply` is decreased by calling the `burn()` method.

2.5.6 Updating `timeOfDeposit`

Minting / receiving shares A user's time of deposit timestamp is only updated when the user mints or receives shares through a transactoin. The update is defined through a share weighted average[4] of the time of deposit t_0 and the current timestamp t_1. Let x denote the user's balance before receiving y shares. Then the updated timestamp is defined as

$$t_{n+1} \leftarrow \frac{xt_n + yt_*}{x + y} \ .$$

Through this update the user will always owe the correct amount of management fees, i.e. transferring shares does not increase nor decrease the management fees owed to the vault. The following equation shows that management fees owed at any time t ($\geq t_{n+1} \geq t_* \geq t_n$) past the last time of deposit update is equal to the management fees owed at times t_0 and t_1 if the shares would have been treated seperately.

$$\begin{aligned} \xi^{\texttt{user}}(x + y, t_{n+1}) &= \omega^{-1}\chi \cdot (t - t_{n+1})(x + y) \\ &= \omega^{-1}\chi \cdot (t \cdot (x + y) - (xt_n + yt_*)) \\ &= \omega^{-1}\chi \cdot (x \cdot (t - t_n) + y \cdot (t - t_*)) \\ &= \xi^{\texttt{user}}(x, t_n) + \xi^{\texttt{user}}(y, t_*) \end{aligned}$$

[4] Equivalent to a linear interpolation based on the share amounts.

Transferring / redeeming shares When a user transfers or redeems shares the time of deposit does not need to be updated as the same timestamp can be used to compute future managament fees. Note that if a user redeems all of his shares and at a later point in time receives or mints new shares the share-weighted average of the time of deposit will ensure that the new time of deposit is the time at which the new shares were minted.

2.5.7 Updating `netUserDeposit`

Minting / receiving shares Let x be the share balance before receiving / minting shares, y denote the received / minted amount, and p denote the price per share at the time of mint / receival. Whenever a user mints or receives y shares `netUserDeposit` is incremented by the asset amount $\Delta = py$ used to mint the new shares.

$$d_{n+1} \leftarrow d_n + \Delta = d_n + py$$

Let μ_n denote the user's average price per share at time n and x the users. The following equation verifies that the updated average price per share paid by the user is the share-weighted average of past prices per share

$$\mu_{n+1} = \frac{d_{n+1}}{x+y} = \frac{d_n + \Delta}{x+y} = \mu_n \frac{x}{x+y} + p\frac{y}{x+y} \ .$$

Transferring / redeeming shares Again assume the user's initial balance of vault shares is x. Whenver a user transfers or redeems $y \leq x$ shares the `netUserDeposit` d_n is decreased proportional to the amount of shares

$$d_{n+1} \leftarrow d_n \frac{x - y - (\gamma^{\texttt{user}}(y) + \xi^{\texttt{user}}(y))}{x} \ .$$

Using this update rule the new average price per share μ_{n+1} remains constant as the net user deposit is reduced proportional amount by the same amount as the balance

$$\mu_{n+1} = \frac{d_{n+1}}{x - y - (\gamma^{\texttt{user}}(y) + \xi^{\texttt{user}}(y))} = \frac{d_n}{x} = \mu_n$$

Even more observe that the user is indifferent, i.e. paying performance fees through multiple transactions / redemptions is equal to paying the performance fee by having a single transaction / redemption of the max transferable shares. Assume that the price per share is constant, and that the user first redeems y shares and then x shares. Then the following relationship holds true

$$\begin{aligned}
\gamma^{\text{user}}(x+y) &= \mathbf{1}_{\{r>0\}}\gamma r \cdot (x+y) \\
&= \mathbf{1}_{\{r>0\}}\gamma r x + \mathbf{1}_{\{r>0\}}\gamma r y \\
&= \gamma^{\text{user}}(x) + \gamma^{\text{user}}(y)
\end{aligned}$$

where equality two is true since the average price per share before and after the transaction / redemption are equal and we assumed the price per share to be constant (the return r did not change).

2.5.8 Frequently asked questions

Why is there no functionality that collects the management fees from all users? This functionality was originally implemented, but later on deprecated as deducting the management fee from all users would require reducing the price per share. A decrease in the price per share comes with some side effects. Assume user A wants to buy from user B Δ shares at the current price per share of p. The transfer of shares would trigger a collection of management fees and thus a decrease in the price per share. Thus, user A would have paid the management fees of user B. We therefore decided to collect management fees at transfer / redemption instead of collecting them globally. Note that charging management fees this way is beneficial to the vault: management fees charged decay linearly[5], not geometrically. Therefore, this method collects more fees.

Why are only fees paid on the the transferred amount and not the full balance, i.e. `maxTransferableShares`, at time of transfer? When fees are collected the vault collects the fees by burning the shares owed to the vault and realizes the shares at the current price per share. Since we assume that the vault has positive expected value due to the spread collection taking place we are not interested in collecting the fees. An alternative design could be not to burn the shares at all. In this case the vault would become and LP itself and only redeem those shares when the vault operator is willing to do so. Furthermore, if requested the current fee design can be changed such that all management and performance fees are charged whenever a user transfers / redeems.

2.6 `pricePerShare`: exchanging between collateral and vault shares

At the core of the vault is the conversion from assets to shares and vice versa. Different to other vaults we cannot directly compute the exchange rate as the ratio of total assets against total shares as the vault is held liable to a potential future payout. Furthermore, spreads are locked and dispersed over the lifetime of the option. It's therefore necessary account for those two positions when computing the `pricePerShare`. Including these considerations we define the price per share as

$$p(t) = \frac{A_n - \xi(t) - \mathcal{L}(t)}{S_n}$$

Observe that defining the price per share this way ensures that the price per share stays constant whenever a trade is processed. This is shown for trades (including the offset of premiums and spreads) in subsubsection 2.7.6.

[5] This is the reason why management fees in shares can be greater than the total balance of a user.

2.7 `trade`: buying options from the vault

A user can buy call / put options for a single underlying from a call / put vault given that a pool with the corresponding maturity, strike and option type exists. A user can never sell optoins back to the vault.

Trades need to fulfill pass filters based on the option's delta and the days to expiry to be executed. Table 5 summarises the parameters stored on-chain that define the filters.

Variable Name	Notation	Purpose
`minDTE`	–	The minimum days to expiration that an option needs to have for the sale / trade to be processed.
`maxDTE`	–	The maximum days to expiration that an option can have for the sale / trade to be processed.
`minDelta`	–	The minimum Black-Scholes delta that an option needs to have for the sale / trade to be processed.
`maxDelta`	–	The maximum Black-Scholes delta that an option can have for the sale / trade to be processed.

Table 5: Set of parameters that control the space of admissible trades.

To charge a spread the vault uses just as in the Premia v2 version the c-level function. Table 6 summarises the variables that are used to parametrise the spread adjustment.

Variable Name	Notation	Purpose
`minCLevel`	$c_{\min}$	The minimum c-level. Default: 1.0.
`maxCLevel`	$c_{\max}$	The maximum c-level. Default: 1.2.
`alphaCLevel`	α	
`hourlyDecayDiscount`	λ	Rate in seconds at which the c-level decays.
`lastTradeTimestamp`	t_0	Timestamp in seconds that stores the last time a trade was executed. Used to compute the decay of the c-level.
`totalLockedAssets`	L	The amount of collateral locked due to underwriting options used to compute the vault's utilisation and available assets. E.g. if 2.3 put options with \$500 strike are underwritten then \$1150 units of the quote are locked in short put contracts.

Table 6: c-level parameters

2.7.1 c-level: adjustments to the fair option value to earn a spread

Depending on the vault's utilisation and the last time a trade occured the spread will be increased or decreased. The function $\tilde{c} : [0,1] \to [c_{\min}, c_{\max}]$ defines the intermediate c-level as a function of the vault's utilisation $x \in [0,1]$

$$\tilde{c}\,(x) = \frac{e^{-\alpha(1-x)}\left(\beta e^{\alpha(1-x)} + c_{\max}\alpha - \beta\right)}{\alpha}$$

where β is defined as

$$\beta = \frac{\alpha\,(c_{\min}e^{\alpha} - c_{\max})}{e^{\alpha} - 1}\ .$$

To obtain the final c-level function $c : [0,1] \to [c_{\min}, c_{\max}]$ the intermediate c-level is decayed by

$$c(x) = \max(c(x) - \lambda \cdot (t - t_0), c_{\min})$$

Assuming the minting fee is zero for simplicity, the quoted price q is then defined as a function of utilisation $x \in [0,1]$, time to maturity $\tau \geq 0$, strike k and the risk-free rate r

$$q(x, S_t, k, \tau, r) = c(x)\ \mathtt{BS}(S_t, k, \tau, r, \sigma(k,\tau))$$

Separating the quoted price into the fair option price and collected spread we can define the spread in terms of the Black-Scholes price and the c-level

$$\underbrace{q(x, S_t, k, \tau, r)}_{\texttt{quotedPrice}} = \underbrace{\mathtt{BS}(S_t, k, \tau, r, \sigma(k,\tau))}_{\texttt{fairOptionPrice}} + \underbrace{(c(x) - 1)\ \mathtt{BS}(S_t, k, \tau, r, \sigma(k,\tau))}_{\texttt{spread}}$$

Note that whenever the minimum c-level $c_{\min}$ is greater or equal to zero the spread is non-negative. For a minimum c-level less than one the quoted price can become less than the options fair value. In this case the option is sold at a discount giving the LP exposure to an option position with negative expected value.

2.7.2 `addListing()`: updating the set of listings upon a trade

Whenever a trade occurs the vault checks whether the option maturity and strike were previously added. In case the listing does not exist the maturity-strike pair gets added to the set of `maturities` and `maturitiesToStrikes`

$$\mathcal{M} = \mathcal{M} \cup \{m\}$$
$$\mathcal{K}(m) = \mathcal{K}(m) \cup \{k\}$$

Furthermore, the vault checks whether `minMaturity` and `maxMaturity` need to be updated.

2.7.3 Updating `totalAssets`

The vault's `totalAssets` are incremented by the c-level adjusted fair option price. It is not incremented by the minting fee as the minting fee is transferred from the user to the pool for minting the options and therefore is balanced as a transit item within the vault.

$$A_{n+1} \leftarrow A_n + c(x)\texttt{BS}(S_t, k, \tau, r, \sigma(k, \tau))$$

2.7.4 Updating `totalLockedAssets`

Whenever a trade is processed `totalLockedAssets` is incremented by the amount

$$L_{n+1} \leftarrow L_n + \begin{cases} s(k,m) & \text{if call} \\ s(k,m) \cdot k & \text{if put} \end{cases}$$

2.7.5 Updating spread state variables (`afterBuy()`)

The hook `afterBuy` was introduced to update spread state variables whenever a trade was processed. Before incrementing spread state variables the procedure `updateState` is called as otherwise inconsistencies between the `totalLockedSpread` and `spreadUnlockingRate` may be introduced[6]. Afterwards, the `totalLockedSpread` is incremented by the amount of spread earned $\gamma \geq 0$, and the `spreadUnlockingRate`

[6] Please ask if this is unclear.

and `spreadUnlockingTick` at maturity m are increased by the spread dispersion rate $\frac{\gamma}{m-t}$.

$$\begin{aligned} \xi &\leftarrow \xi + \gamma \\ \phi &\leftarrow \phi + \frac{\gamma}{(m-t)} \\ \phi(m) &\leftarrow \phi(m) + \frac{\gamma}{(m-t)} \end{aligned}$$

2.7.6 Price invariance

A trade should not increase nor decrease the trade as otherwise it would be subject to frontrunning / arbitrage opportunities. Let t_- and t_+ be the immediate time before and after the trade is processed and $p(t_-)$, $p(t_+)$ denote the corresponding prices per share[7], $\mathcal{L}(t_-), \mathcal{L}(t_+)$ the total liablities and $\xi(t_-)$, $\xi(t_+)$ the total locked spreads. Furthermore, assume π denotes the option's fair value and z the spread paid due to the c-level. Then we observe that the prices per share before and after the trade are equal

$$\begin{aligned} p(t_+) &= \frac{A_{n+1} - \xi(t_+) - \mathcal{L}(t_+)}{S_{n+1}} \\ &= \frac{(A_n + \pi + z) - (\xi_n(t_-) + z) - (\mathcal{L}(t_-) + \pi)}{S_n} \\ &= \frac{A_n - \xi(t_-) - \mathcal{L}(t_-)}{S_n} \\ &= p(t_{-1}) \end{aligned}$$

Thus, the vault is protected from front-running / arbitrageurs.

2.8 Settlement of expired options

Short options need to be settled for the remaining collateral to be released. The liberation of collateral through settlement makes collateral available and lowers the vault's utilisation. Settling expired options can be triggered through a keeper or a user at any time.

2.8.1 Updating `totalAssets`

During settlement the vault's `totalAssets` are decreased by the option's exercise value

$$A_{n+1} \leftarrow A_n - \begin{cases} s(k,m)(1 - \frac{k}{S_m})^+ & \text{if call} \\ s(k,m)(k - S_m)^+ & \text{if put} \end{cases}$$

where again $s(k,m)$ denotes the number of short contracts with strike k and expiry m and S_m denotes the spot price at time m.

[7] More precisely $p(t_-)$ and $p(t_+)$ can be viewed as the left and right limit.

2.8.2 Updating `totalLockedAssets`

`totalLockedAssets` need to be decreased by the collateral value of the short position

$$L_{n+1} \leftarrow L_n - \begin{cases} s(k,m) & \text{if call} \\ s(k,m)\cdot k & \text{if put} \end{cases}$$

which is equal to the amount of shorts $s(k,m)$ for calls and the amount of short adjusted by the strike $s(k,m)\cdot k$ for puts. Note that we do not increment `totalAssets` by the released collateral since we never deducted the locked assets which are deposited as a security into the short position from `totalAssets`. See subsection 2.7 for more details.

2.8.3 `removeListing()`: updating the set of listings upon settlement

Moreover, every maturity $m \in \mathcal{M}$ that was settled is removed from the set of `maturities` $\mathcal{M}$ and every strike specific to the maturity $m \in \mathcal{M}$ is removed from the set of strikes $\mathcal{K}(m)$

$$\begin{aligned} \mathcal{M} &= \mathcal{M} \setminus \{m\} \\ \mathcal{K}(m) &= \mathcal{K}(m) \setminus \{\forall k : k \in \mathcal{K}(m)\} = \emptyset \end{aligned}$$

Furthermore, `minMaturity` is incremented to the next maturity in the set of `maturities` and set to 0 in case the set is empty. Likewise, `maxMaturity` is set to 0 in case there are no more active listings.

It is worth noting that because maturities and strikes are removed from the option lattice the vault has to update it's spread state variables before removing them as otherwise the vault is not able to decrement the spread unlocking rate ϕ using the spread unlocking tick $\phi(m)$ specific to the removed maturity $m \in \mathcal{M}$ anymore. Therefore, the method `settle` calls the method `_updateState` prior to removing any maturity-strike pairs from the option lattice.