CERTORA

Formal Verification of Opyn Gamma Protocol

Summary

This document describes the specification and verification of Opyn Gamma Protocol using Certora Prover. The work was undertaken from October 5,2020 until December 28, 2020. The rules are currently being integrated into Opyn's CI.

The scope of the project was the new Gamma protocol. The protocol enables any user to create arbitrary option tokens representing the right to buy or sell a certain asset at a predefined price (strike price) at or before expiry.

The Certora Prover proved that the implementation of the following contracts: MarginVault.sol, Controller.sol, MarginPool.sol, MarginVault.sol, Whitelist.sol, Otoken.sol are correct with respect to the formal rules written by the Opyn and Certora teams. The mathematical formulas for computing collateralization of an account, which are implemented in MarginCalculator.sol, have been shown to be correct (see Appendix A). During the verification process, the Certora Prover and the team's manual review discovered a number of bugs in the code listed in the table below. All the high- and medium issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. These limitations are currently being handled by the Certora development team. The next section formally defines high level specifications of Gamma Protocol. All the rules are publically available in https://github.com/opynfinance/GammaProtocol/tree/certora/accounting.

List of Main Issues Discovered

Issue	Rule broken	Description	Severity	Mitigation
Denial of service on an option series		Vault can be in a valid state but nobody else is able to mint more options. The attacker would mint enough oTokens so that their total supply reaches MAX_UINT, and then add them as long in their vault.	High	Add a check to not allow storing shorts and longs of the same strike price in a vault
Can manipulate		Oracle's price feed is based on a list	High	Check that the

ChainLinkPricer to give bad prices		of consecutive rounds, each one containing a monotonically increasing timestamp and a suggested price for that timestamp. However, some rounds can be invalidated by Chainlink, which sets their timestamp to zero. Due to a lack of sanitation, a user might set an expiry price for an option from a different date, using such invalid rounds.		requested round's timestamp is non-zero (valid round) - Update Chainlink price-feed API to v3, which naturally prevents this issue.
Inverting deposits can go wrong	inverse Add & Remove Collateral (6), inverse Add & Remove Short (9), inverse Add & Remove Long (12)	In the MarginVault library, one must pass the correct and same index in both add and remove operations in order to return to the original state. This can lead to suboptimal user experience	Medium	Stronger requirements in add to vault operations
Deposit functions are not additive	additive Add Collateral (5), additive Add Short (8), additive Add Long (12)	In the MarginVault library, array sizes could overflow	Low	Since it's unrealistic to overflow the arrays, the requirement is reflected only in the spec
Asset arrays could contain invalid assets	valid entity index (1. II.)	In the MarginVaultLibrary, address "0" can be added and take up an array element slot	Low	Should make sure "0"cannot be added as an asset to the vault
Gas inefficiency in runActions		By running a sequence of operations that are surely to lead to a violation, the revert happens late, leading to more gas waste by the user	Low	
getProceed may interpret a negative amount as a positive one		Every call to getExcessCollateral returns two values: the excess collateral or the missing collateral for solvency, and a boolean flag indicating which case we are in. This flag should always be checked	Low	Unreachable code

		whenever getExcessCollateral is called.		
intToUint() semantics are confusing		The SignedConverter class is converting int data type to uint datatype by taking the absolute value of the number. This could lead to wrong usages if the behavior is not properly documented.	Low	

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, but provide no warranty of any kind, express or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

1. ✔ indicates the rule is formally verified on the latest commit, We write ✔* when the rule was verified on a simplified version of the code (or under some assumptions).
2. ❌ indicates the rule is violated in the version under test.
3. ✍ indicates the rule is not yet formally specified.
4. 🔁 indicates the rule is beyond the scope of this project.
5. We define ε to be the allowed rounding error, token-conversion error as a small percentage.
6. We define $\mathbf{L}$ to be some realistic value of supply of an asset.
7. We use Hoare triples of the form {p} C {q}, which means that if program C executes starting in any state satisfying p, then it will end in a state satisfying q. In Solidity, p is similar to require and q is similar to assert.

   The syntax {p} (C_1 ~ C_2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states.

   As a special case, $C_1 \sim_{op} C_2$, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of MarginVault

A library that manipulates vaults, verified independently of the calling contracts.

Functions

totalCollateral(Vault v) : uint
Returns the total collateral in vault *v*.

totalShortAmount(Vault v) : uint
Returns the total amount of short oTokens in vault *v*.

totalLongAmount(Vault v) : uint
Returns the total amount long oTokens in vault *v*.

addCollateral(Vault v, address asset, uint x, uint index) : bool
Returns true when successfully increasing the collateral balance at index *index* in vault *v* by *x asset* tokens.

removeCollateral(Vault v, address asset, uint x, uint index) : bool
Returns true when successfully decreasing the collateral balance at index *index* in vault *v* by *x asset* tokens.

addShort(Vault v, address short, uint x, uint index) : bool
Returns true when successfully increasing the short balance at index *index* in vault *v* by *x short tokens*.

removeShort(Vault v, address short, uint x, uint index) : bool
Returns true when successfully decreasing the short balance at index *index* in vault *v* by *x short tokens*.

addLong(Vault v, address long, uint x, uint index) : bool
Returns true when successfully increasing the long balance at index *index* in vault *v* by *x long tokens*.

removeLong(Vault v, address long, uint x, uint index) : bool
Returns true when successfully decreasing the long balance at index *index* in vault *v* by *x long tokens*.

Properties

1. **Validity state of a vault ✔**
   A vault *v* is in a valid state if for each of the entities (collateral, long, short):

   I. The length of the list of amounts of entity equals to the length of the list of addresses of this entity. ✔

   ```
   v.longAmounts.length = v.longOtkens.length ⋀
   v.shortAmounts.length = v.shortOtkens.length ⋀
   v.collateralAmounts.length = v.collateralAssets.length
   ```

   II. The entity amount at index *i* is greater than zero iff there is a non zero address at index *i* of this entity. ✔

   ```
   v.longAmounts[i] > 0 ⇔ v.longOtokens[i] ≠ 0 ⋀
   v.shortAmounts[i] > 0 ⇔ v.shortOtokens[i] ≠ 0 ⋀
   v.collateralAmounts[i] > 0 ⇔ v.collateralAssets[i] ≠ 0
   ```

2. **Change to a single entity ✔**
   Each operation `op` changes at most one entity: short, long or collateral.

```
{
       l = v.longAmounts[i] ⋀
       s = v.shortAmounts[i] ⋀ c = v.collateralAmounts[i]
}
       op
{
      (v.longAmounts[i] ≠ l
         ⇒ v.shortAmounts[i] = s ⋀ v.collateralAmounts[i] = c )
      ⋀
      (v.shortAmounts[i] ≠ s
         ⇒ v.longAmounts[i] = l ⋀ v.collateralAmounts[i] = c )
      ⋀
      (v.collateralAmounts[i] ≠ c
         ⇒ v.longAmounts[i] = l ⋀ v.shortAmounts[i] = s )
}
```

3. **Success characteristic of addCollateral ✔**
   One should be able to add *i* collateral on a valid vault *v* under realistic circumstances.

```
{ x > 0
   ⋀ ( i < v.collateralAmounts.length
        ⇒ ( v.collateralAssets[i] = a
             ⋀ v.collateralAmounts[i] + x < MAX_UINT) )
}
   r = addCollateral(v,a,x,i)
{ r }
```

4. **Integrity of addCollateral ✔**
   When successfully adding *x* collateral to a vault at index *i*, the `totalCollateral` increases by *x.*

```
{ b = totalCollateral(v) }
      addCollateral(v,a,x,i);
{ totalCollateral(v) = b + x }
```

5. **Additive addCollateral ✔**
   Adding collateral to a vault is additive, i.e., it can be performed either all at once or in steps.

```
addCollateral(v,a,x); addCollateral(v,a,y) ~ addCollateral(v,a,x+y)
{ totalCollateral₁(v) = totalCollateral₂(v) }
```

6. **Inverse addCollateral and removeCollateral ✔**
   Adding and removing collateral are inverse operations.

```
{ b = totalCollateral(v) }
      addCollateral(v,a,x,i); removeCollateral(x,a,x,i)
{ totalCollateral(v) = b }
```

7. **Integrity of addShort ✔**
   Similar to 4.

8. **Additive addShort ✔**
   Similar to 5.

9. **Inverse addShort and removeShort ✔**
   Similar to 6.

10. **Integrity of addLong ✔**
    Similar to 4.

11. **Additive addLong ✔**
    Similar to 5.

12. **Inverse addLong and removeLong ✔**
    Similar to 6.

Verification of Controller

Controller, the main contract of the Gamma Protocol, interacts with all sub contracts (oTokens, MarginPool, etc…). The properties of Controller are verified using an assumption there are at most two oTokens and one collateral type token. These assumptions are safe given that it is shown that each operation changes at most one vault. Each vault contains at most one short oToken and one long oToken of the same collateral. In addition, it is assumed that the Controller and MarginPool contracts are not recipients of any token.

Functions

storedBalance(address asset) : uint
The tracked asset balance of the system, i.e., `MarginPool.getStoredBalance(asset)`

vault(address owner, uint i) : vault
Returns the *i*-th vault of a user *owner*.

excessAsset(address owner, uint vaultId, address asset) : int
The excess amount the *owner* has in his *vaultId*, denominated in *asset.* Theoretically, it can be negative, as it converts the shorts oTokens' obligation to *asset*.

openVault(address owner, uint256 vaultId) : void
Opens a new vault *vaultId* inside the account of the *owner.*

depositLong(address owner, uint256 vaultId, address from, uint256 index, address oToken, uint256 amount) : void
Deposits *amount* of long *oToken* into *vaultId* of *owner.*

withdrawLong(address owner, uint256 vaultId, address receiver, uint256 index, address oToken, uint256 amount) : void
Withdraws *amount* of long *oToken* from *vaultId* of *owner.*

depositCollateral(address owner, uint256 vaultId, address from, uint256 index, address asset, int256 amount) : void
Deposits *amount* of collateral *asset* into *vaultId* of *owner*

withdrawCollateral(address owner, uint256 vaultId, address receiver, uint256 index, address asset, int256 amount) : void
Withdraws *amount* of collateral *asset* from *vaultId* of *owner.*

mintOtoken(address owner, uint256 vaultId, address receiver, uint256 index, address oToken, uint256 amount) : void
Mints *amount* of short *oToken* from a *vaultId* of *owner* which creates an obligation that is recorded in the vault. Sends the minted token to *receiver*.

burnOtoken(address owner, uint256 vaultId, address from, uint256 index, address oToken, uint256 amount) : void
Burns *amount* of *oToken* and reduces the minted *oToken* obligation in *vaultId* of *owner*.

redeem(address receiver, address oToken, uint256 amount) : void
Redeems an *amount* of *oToken* after expiry, transferring the payout to *receiver*.

settleVault(address owner, uint256 vaultId, address receiver) : void
Settles vault *vaultId* of *owner* after expiry, removing the net proceeds/collateral after both long and short oToken payouts have been settled and transfers excess collateral to *receiver*.

Properties

13. Valid balance of the system ✔*

The balance of the system in an external *asset* is correlated with the tracked asset balance.

I. `storedBalance(asset) ≤ asset.balanceOf(MarginPool)`

II. `storedBalance(asset) ≤ asset.totalSupply()`

14. Valid balance with respect to total collateral ✔*

The sum of a collateral asset across vaults matches the asset balance stored in the margin pool (before expiry).

$$\texttt{storedBalance(asset)} = \sum_{(v,i) \in V\,asset} \texttt{v.collateralAmounts[i]}$$

where V_{asset} is the set of vaults which contains asset as collateral, i.e.,

$V_{asset} \equiv$ `{ (v,i) | v ∈ Vaults.  v.collateralAssets(i) = asset }`

This rule is proven by showing that:

1. change in vault balances => equal change in balance in pool
2. change in pool balances => equal change in a specific vault balances

15. Valid balance of long oTokens ✔*

The sum of a long oToken across vaults matches the asset balance stored in the margin pool.

$$\texttt{storedBalance(otoken)} = \sum_{(v,i) \in V otoken} \texttt{v.longAmounts[i]}$$

where V_{otoken} is the set of vaults which contains oToken as a long oToken, i.e.,

$V_{oToken} \equiv \{ (v,i) \mid v \in Vaults.\ \ v.longOtokens(i) = otoken\}$

This rule is proven by showing that:

1. change in vault balances => equal change in balance in pool
2. change in pool balances => equal change in a specific vault balances

   (Note this fails if someone mints oTokens to the pool.)

16. Valid supply of short oToken ✔*

The sum of a short *oToken* across vaults matches the supply of that short oToken, for an oToken before expiry.

$$\texttt{otoken.totalSupply()} = \sum_{(v,i) \in V otoken} \texttt{v.shortAmounts[i]}$$

where V_{otoken} is the set of vaults which contains *oToken* as a short oToken, i.e.,

$V_{oToken} \equiv \{ (v,i) \mid v \in Vaults.\ \ v.shortOtokens(i) = oToken\}$

1. change in vault balances => equal change in balance in pool
2. change in pool balances => equal change in a specific vault balances

   (Note this fails if someone mints oTokens to the pool.)

17. No effect on other vault ✔*

All operations can affect at most one vault (of the same or other user)

$\{ v_1 = vault(a,i) \wedge v_2 = vault(b,j) \wedge (a \neq b \vee i \neq j) \}$

0p

$\{ vault(a,i) = v_1 \vee vault(a,i) = v_2 \}$

18. Order of operations ✔*

The order of operations does not matter and the end result should be the same if the same operations are successfully performed butin different orders

```
op1; op2  ~  op2; op1
```

19. Solvency of the system ✔*

The asset balance of the system is more than the obligations in this asset.

Solvency is defined as:

$\texttt{storedBalance(asset)} \geq \sum_{o \in O\_asset.} \texttt{obligation(o)}$

where O_{asset} is the set of oTokens with `asset` as `ColleteralAsset`:

$O_{asset} \equiv \{ o \in oTokens.\ \ o.CollateralAsset = asset \}$

This is proved for different cases of oTokens:

I. For call options:

$\texttt{obligation(o)} \equiv \texttt{(o.totalSupply() - storedBalance(o))}$

II. For put options:

$\texttt{obligation(o)} \equiv$
$\texttt{(o.totalSupply() - storedBalance(o)) * o.strikePriceIn(asset)}$

Given, a system that is in a solvent state and a vault *v* that is valid (collateralized) and in pre-expiry state, any change to the vault resulting in a valid vault leaves the system in a solvent state

$\{\ \texttt{storedBalance(asset)} \geq \sum_{o \in O\_asset.} \texttt{obligation(o)} \wedge \texttt{isValid(v)} \wedge \texttt{isPreExpiry(v)}\ \}$

op_v^*

$\{\ \texttt{isValid(v)} \Rightarrow \texttt{storedBalance(asset)} \geq \sum_{o \in O\_asset.} \texttt{obligation(o)}\ \}$

where op_v^* is a sequence of operations on the vault *v*.

20. Integrity of collateral being withdrawn from the margin pool ✔*

Collateral assets can only be transferred out of the margin pool by certain functions.

```
{ b = storedBalance(asset) }

      op

{ b > storedBalance(asset) ⇒

      op in { withdrawCollateral(), redeem(), settleVault() }

}
```

21. Integrity of options being withdrawn from the margin pool ✔*

Options can only be transferred out of the margin pool by certain functions.

```
{ b = storedBalance(oToken) }

      op

{ b > storedBalance(oToken) ⇒

      op in { withdrawLong(), settleVault() }

}
```

22. Integrity of redeem ✔*

Redeem of a short oToken reduces the total supply and payout as expected.

```
{ bSupply = oToken.totalSupply() ∧
  bBalance = storedBalance(oToken)  ∧
  bReceiver = asset.balanceOf(receiver) ∧
  p = payout(oToken, amount) ∧
  oToken.CollateralAsset = asset  }

     redeem(receiver, oToken, amount)

{ bSupply - oToken.totalSupply() = amount ∧
  bBalance - storedBalance(oToken) = p ∧
  asset.balanceOf(receiver) - bReceiver = p

}
```

23. Integrity of settleVault ✔*

Settlement of a vault yields payout as expected.

```
{ bSupply = oToken.totalSupply() ∧
  bReceiver = asset.balanceOf(receiver) ∧
  p = excessAsset(owner, vaulted, asset) ∧
  vault(owner, vaultId).shortOtoken = oToken ∧
  oToken.CollateralAsset = asset
}

  settleVault(owner, vaultId, receiver)

{ bSupply = oToken.totalSupply() ∧
  asset.balanceOf(receiver) - bReceiver = p
}
```

Verification of Whitelist

A module that keeps track of all valid oToken and Collateral addresses.

Functions

isWhitelistedOtoken(address token) : bool
Returns true if *token* is a whitelisted oToken.

isWhitelistedCollateral(address token) : bool
Returns true if *token* is a whitelisted collateral.

Properties

24. Privileged operations ✔

State changing functions are privileged: can only be run by a single address.

25. Only whitelisted oTokens can be used in the system ✔

```
{ b =  oToken.totalSupply() ∧ p = storedBalance(otoken) }
      op
{ (b ≠  oToken.totalSupply() ∨ p  ≠ storedBalance(otoken)) ⇒
       isWhitelistedOtoken(oToken) }
```

where `op` is any operation of the controller.

26. Only whitelisted oTokens can be in vaults ✔

```
(v.longOtokens(i) = otoken  ∨ v.shortOtokens(i) = otoken)
       ⇒ isWhitelistedOtoken(otoken)
```

27. An address can not be both collateral and oToken ✔

```
!(isWhitelistedOtoken(a) ∧ isWhitelistedCollateral(a))
```

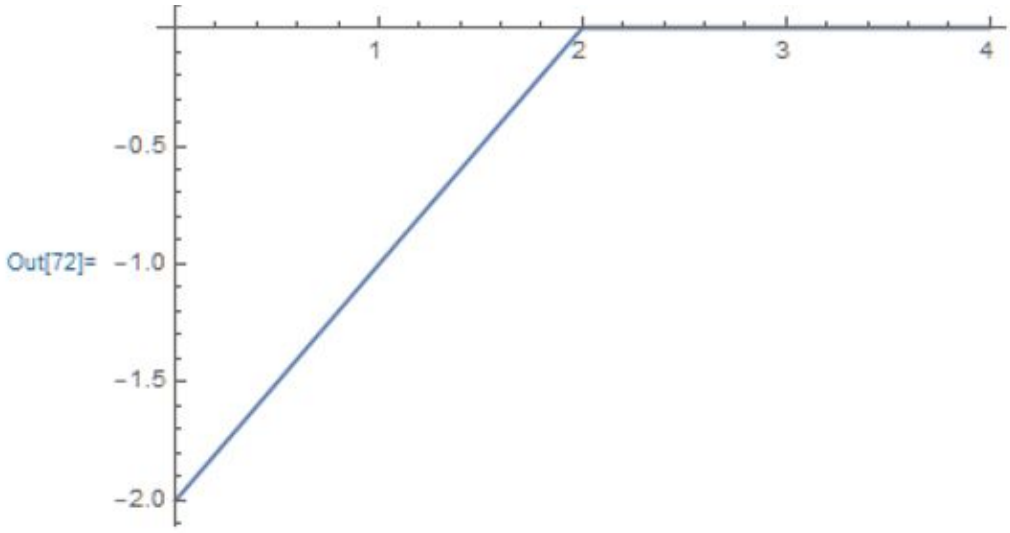
Appendix - Computing collateral requirements for option spreads

To understand how to correctly write a general equation for a put/call spread position (short+long), it is useful to look at the "final price vs profit" graph for each option.

We begin with put type options, since it is the simpler case of the two.

Put

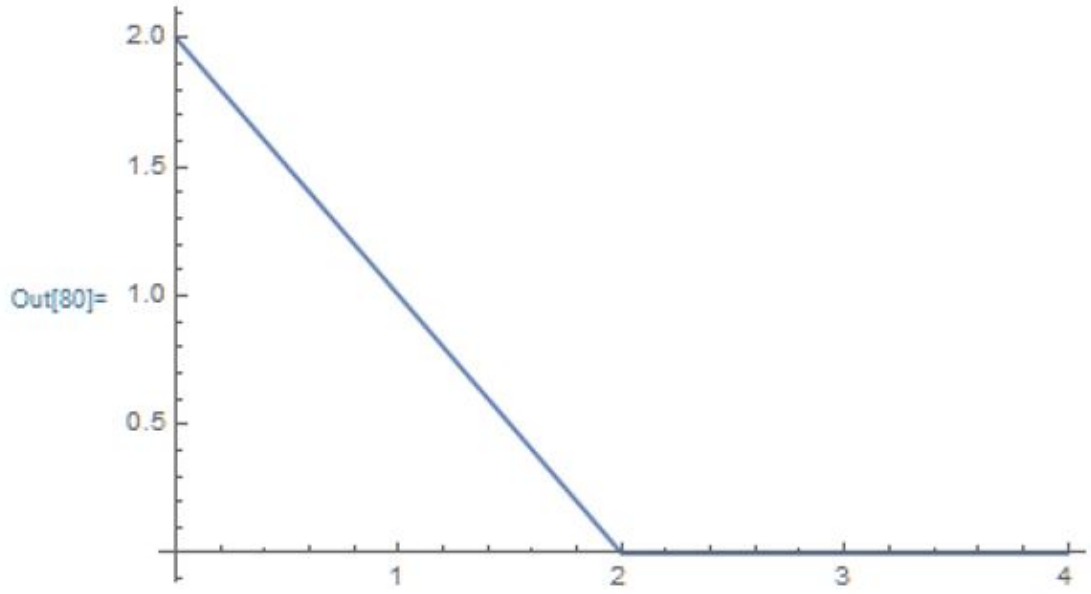
A standard graph for a short put would look like:



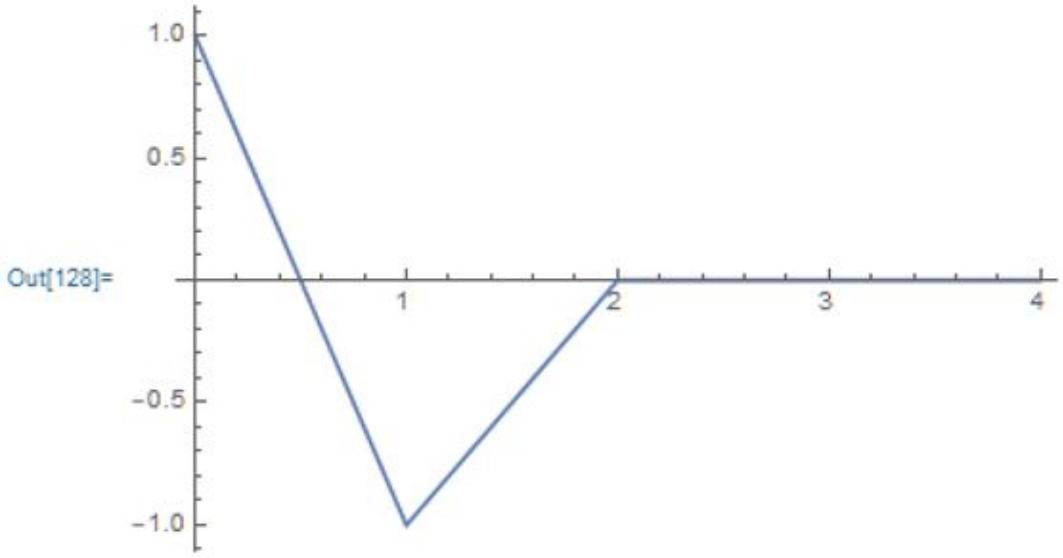
In this graph, if the final price of the underlying asset (x axis) drops below 2 (in the strike asset), the short position (the one who "wrote" the option) begins to lose money, since there is a commitment to buy the asset at a price of 2. In the worst case for the option writer, the asset is worth zero, so the writer has a -2 balance for every option produced.[1]

The long put is just the negation of the short one, which corresponds to the other side of the deal (the option buyer).

[1]On the internet, the graph saturates at some small positive value, which corresponds to the option premium. In Opyn, the premium is implicit (it is determines by the exchanges/markets), so we can ignore it.



A spread graph (short+long) can look something like this:



So, at some range of final prices for the underlying asset the spread holder would lose, and in the other he would gain.

The general formula for a spread put graph is:

$$T_{put}(x) = shortAmount * \big((x - shortPrice) - \Theta(x - shortPrice) * (x - shortPrice)\big) + \\ -longAmount * \big((x - longPrice) - \Theta(x - longPrice) * (x - longPrice)\big)$$

With $\Theta(x) = \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases}$ being the usual theta function.

Next, with Mathematica's MinValue[] function we find the minimum of $T_{put}(x)$ with respect to the final price for some general amounts and prices (under the constraints that they are all real and positive). This would tell us what is the minimal amount of collateral we need to prevent a situation where the position is under-collateralized.

The result is:

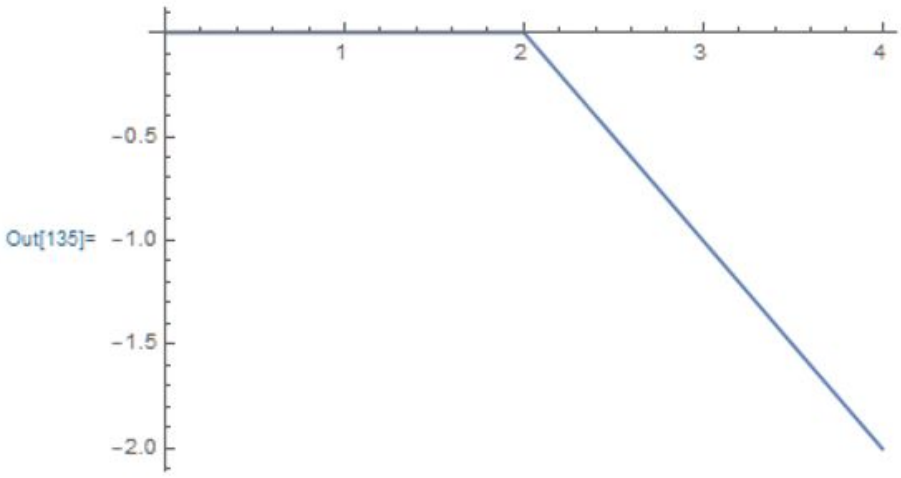
```
In[132]:= Assuming[{longPrice > 0, longAmount > 0, shortPrice > 0, shortAmount > 0},
           FullSimplify[MinValue[{debitPut, longPrice > 0, longAmount > 0, shortPrice > 0, shortAmount > 0, x ≥ 0, x < Max[shortPrice, longPrice]}, x]]]

Out[132]= { longAmount longPrice - shortAmount shortPrice   longAmount ≤ shortAmount && longAmount longPrice < shortAmount shortPrice
          { shortAmount (longPrice - shortPrice)            longAmount > shortAmount && longPrice < shortPrice
          { 0                                               True
```

This indeed looks very much like Opyn's calculation.

Call

For a call option, the short graph looks like:



Potentially, the loss is infinite in the units of the strike asset. Therefore, in the call case, we must have collateral in the underlying asset.

The general formula for a spread call graph (the put formula just flipped horizontally around the short\long prices) is:

$$T_{call}(x) = shortAmount\big((shortPrice - x) - \Theta(shortPrice - x) * (shortPrice - x)\big) + \\ -longAmount\big((longPrice - x) - \Theta(longPrice - x) * (longPrice - x)\big)$$

Because we need to denote the collateral in the underlying asset, we want to minimize $\frac{T_{call}(x)}{x}$.

The result is:

Out[•]=

$$\begin{cases} longAmount - shortAmount & (longAmount < shortAmount \;\&\&\; longAmount\, longPrice < shortAmount\, shortPrice \;\&\&\; \sqrt{5}\; shortAmount < 2\, longAmount + shortAmount) \;||\; \\ & (\sqrt{5}\; shortAmount \geq 2\, longAmount + shortAmount \;\&\&\; longAmount\, longPrice \leq shortAmount\, shortPrice) \\ shortAmount\left(-1 + \frac{shortPrice}{longPrice}\right) & (longAmount\, longPrice > shortAmount\, shortPrice \;\&\&\; (longAmount == shortAmount \;||\; \sqrt{5}\; shortAmount \geq 2\, longAmount + shortAmount)) \;||\; \\ & (longAmount\, longPrice \geq shortAmount\, shortPrice \;\&\&\; \sqrt{5}\; shortAmount < 2\, longAmount + shortAmount \;\&\&\; longAmount < shortAmount) \;||\; \\ & (longAmount > shortAmount \;\&\&\; longPrice > shortPrice) \\ 0 & True \end{cases}$$

Opyn's calculation is simply the maximum of the three possible solutions, so it is a sufficient bound to prevent under-collateralization. The tight bound involves the golden ratio, for some unknown reason.