Invariant - math specification

Invariant Labs

Introduction

This paper presents the mathematical specification for the Invariant protocol, a Decentralized Exchange with a Concentrated Liquidity Mechanism. This specification is designed to extensively explore mathematical types and their interrelationships, with a focus on their use in the implementation in Aleph Zero.

In contrast to other CLAMM projects, such as Uniswap, Invariant uses a decimal system instead of a binary system for state representation. This approach has been purposefully designed to facilitate integrations with other protocols

An essential aspect in this paper is the definition of core data types and their associated value domains. These domains play a crucial role in determining key parameters, including the choice of primitive data types and precision, which is indicated by the number of decimal places. The selection of these parameters always entails a trade-off. Choosing a larger primitive data type increases computational demands and storage requirements but allows for a wider range of values. Similarly, precision, represented by the number of decimal places, requires a greater bit allocation for the decimal type. This increased precision enables the detection of minimal differences, which is often necessary.

The Invariant protocol is centered on the concept of permissionlessness, where users have complete control over specific pairs. As such, our specification devotes significant attention to addressing possible edge cases, including maximum and minimum values, to comprehensively cover all potential scenarios.

Contents

Core Types

1 TokenAmount

The TokenAmount type stores **integers** token amount. TokenAmount exist in two variations, dependent on the token type. The **x** represents the quantity of tokens for token **X**, while **y** signifies the quantity of tokens for token **Y**.

1.1 Primary details

- *primitive type* - u128
- *decimals* - 0
- *symbols* :
  - x - amount of token X
  - y - amount of token Y
  - a - generic form of either x or y, when the formula is applied to both types of tokens
  - $\delta(a)$ - decimals of TokenAmount type

1.2 Domain

The TokenAmount's domain is derived from the underlying **Balance** type defined in the **PSP22** token standard.

Max value:
$a_{max} = 2^{128} - 1$

Min value:
$a_{min} = 0$

2 SqrtPrice

The SqrtPrice type stores the **square root** of the y-to-x ($\frac{y}{x}$) token ratio.

2.1 Primary details

- *primitive type* - u128
- *decimals* - 24
- *symbols* :
  - p - price defined as the ratio of $\frac{y}{x}$ tokens
  - $\sqrt{p}$ - sqrt root of price
  - t - integer value denoting the price-associated tick
  - $s(t)$ - tick spacing, specifies the multiple of the associated tick
  - $\delta(\sqrt{p})$ - decimals of SqrtPrice type

2.2 Domain

The factor determining the possible price range, and consequently, the range of the square root of prices, is the extreme ratio of tokens we assume may occur. Therefore, let's start by determining the limiting ratio of y-to-x tokens.

$max(\frac{y}{x}) = 2^{32} - 1$
$min(\frac{y}{x}) = \frac{1}{max(\frac{y}{x})} = \frac{1}{2^{32}-1}$

In the presented context, the maximum and minimum values are reciprocally related, rendering the designation of tokens as X or Y inconsequential.

Now we can determine the range of the square root of prices.

$\sqrt{p_{max}} = \sqrt{max(\frac{y}{x})} * 10^{\delta(\sqrt{p})} = \sqrt{2^{32} - 1} * 10^{24}$
$\sqrt{p_{min}} = \frac{1}{\sqrt{p_{max}}} * 10^{\delta(\sqrt{p})} = \frac{1}{\sqrt{2^{32}-1}} * 10^{24}$

Having the price range and a tick multiplier of 1.0001, we can define the maximum and minimum tick

$t_{max} = \log_{\sqrt{1.0001}} \sqrt{p_{max}} = 221818$
$t_{min} = \log_{\sqrt{1.0001}} \sqrt{p_{min}} = -\log_{\sqrt{1.0001}} \sqrt{p_{max}} = -221818$

Thus we can determine the constants present in source code.

```
pub const MAX_SQRT_PRICE: u128 = 65535383934512647000000000000;
pub const MIN_SQRT_PRICE: u128 = 15258932000000000000;
```

Therefore the tick limit is:

```
pub const MAX_TICK: i32 = 221818;
pub const MIN_TICK: i32 = -MAX_TICK;
```

2.3 Conversion from Tick to $\sqrt{p}$

The function calculates the root of the price at a tick with an accuracy of $\frac{1}{2}\delta(\sqrt{p})$ (12 decimal places). As a result, the product contains $\delta(\sqrt{p})$ (24 decimals), with the last $\frac{1}{2}\delta(\sqrt{p})$ digits being zeros due to computational unit optimization.

```
pub fn calculate_sqrt_price(tick_index: i32) -> SqrtPrice
```

Domain:
$tick\_index = t \in \langle t_{min}, t_{max} \rangle$
$result = \sqrt{p} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$

Formula:
$\sqrt{p} = \sqrt{1.0001^t}$

Max value:
$\sqrt{p_{\max}} = \sqrt{1.0001^{t_{\max}}} * 10^{\delta(\sqrt{p})}$
$2^{95} < \sqrt{p_{\max}} = \sqrt{1.0001^{221818}} * 10^{24} < 2^{96}$

Min value:
$\sqrt{p_{\min}} = \sqrt{1.0001^{t_{\min}}} * 10^{\delta(\sqrt{p})}$
$2^{63} < \sqrt{p_{\min}} = \sqrt{1.0001^{-221818}} * 10^{24} < 2^{64}$

Max intermediate operations:
$\sqrt{p_{\max}} = \sqrt{1.0001^{t_{\max}}} * 10^{\frac{1}{2}\delta(\sqrt{p})} * 10^{\frac{1}{2}\delta(\sqrt{p})}$
$2^{95} < \sqrt{p_{max}} < 2^{96}$

Max intermediate type:
u128

Edge cases:

- $\sqrt{p_{\max}}$
- $\sqrt{p_{\min}}$

2.4 Conversion from $\sqrt{p}$ to Tick

The function calculates the tick aligned to the tick spacing based on the square root of the price.

```
pub fn get_tick_at_sqrt_price(sqrt_price: SqrtPrice, tick_spacing: u16) -> i32
```

1. convert $\sqrt{p}$ to Q32.32
2. calculate $\log_2 \sqrt{p}$ using iterative approximation algorithm
3. calculate the absolute value of the tick changing base rule. The result has an asymmetric accuracy of 1
4. choose either $|tick|$ or $|tick + 1|$, select a sign, and then align the tick to tick spacing

All conversion operations employ unchecked math for compute unit optimizations. The conversion is tested across the entire domain, covering prices at each tick. We expect the functions to perform correctly within our pricing domain.

Domain:
$sqrt\_price = \sqrt{p} \in \langle \sqrt{p_{\min}}, \sqrt{p_{\max}} \rangle$
$tick\_spacing = s(t) \in \langle 1, 100 \rangle$
$result = t \in \langle t_{min}, t_{max} \rangle$

Formula:
$t = \lfloor \frac{\log_{\sqrt{1.0001}} \sqrt{p}}{t(s)} \rfloor * t(s)$

Max value:
$\log_{\sqrt{1.0001}} \sqrt{p_{max}} = 221818$

Min value:
$\log_{\sqrt{1.0001}} \sqrt{p_{min}} = -221818$

Max intermediate operations:
$\sqrt{p_{\max}} = \sqrt{1.0001^{t_{\max}}} * 10^{\frac{1}{2}\delta(\sqrt{p})} * 10^{\frac{1}{2}\delta(\sqrt{p})}$
$2^{95} < \sqrt{p_{max}} < 2^{96}$

Max intermediate type:
u128

Edge cases:

- $t_{\min}$
- $t_{\max}$

3 Liquidity

The liquidity type represents a value that indicates the ability to exchange. Liquidity is determined by the product of the amount of tokens X, the amount of tokens Y, or the sum of X and Y, and the level of concentration. It is associated with a specific price range. As a result, L can be either active or inactive, depending on whether the current price is within or outside the liquidity range.

3.1 Primary details

- *primitive type* - u128
- *decimals* - 6
- *symbols*:
  - L - value of Liquidity type
  - t - integer value denoting the price-associated tick
  - $s(t)$ - tick spacing, specifies the multiple of the associated tick
  - $\mathcal{L}(s(t))$ - function that assigns the liquidity limit per one tick for a specified tick spacing

3.2 Domain

The domain of Liquidity value depends on price accuracy, price range, and TokenAmount primitive type.

Max value:
$L_{max} = 2^{128} - 1$

Min value:
$L_{min} = 0$

3.3 Liquidity limit per Tick

It is required to establish a liquidity limit per one tick to prevent the cumulative value from potentially causing an overflow after multiple tick crossings.

Formula:
$\mathcal{L} = \frac{L * s(t)}{2 * t_{max} + 1}$

Max value:
$s(t)_{max} = 100$
$\mathcal{L}_{max} = \frac{L_{max} * s(t)_{max}}{2 * t_{max} + 1}$
$2^{115} < \mathcal{L}_{max} = \frac{(2^{128} - 1) * 100}{2 * 221818 + 1} = \frac{(2^{128} - 1) * 100}{443637} < 2^{116}$

Min value:
$s(t)_{min} = 1$
$\mathcal{L}_{min} = \frac{L_{max} * s(t)_{min}}{2 * t_{max} + 1}$
$2^{109} < \mathcal{L}_{min} = \frac{(2^{128} - 1) * 1}{2 * 221818 + 1} = \frac{2^{128} - 1}{443637} < 2^{110}$

Max intermediate type:
u128

4 Percentage

The type represents a percentage and is used to simplify interest calculations for various types.

4.1 Primary details

- *primitive type* - u64
- *decimals* - 12
- *symbols* :
  - ρ - percentage value
  - $\delta(\rho)$ - decimals of percentage type

4.2 Domain

Max value:
$\rho_{max} = 2^{64} - 1$
$\rho_{max} = (2^{64} - 1) * 10^{-\delta(\rho)} * 100\%$
$10^9\ \% < \rho_{max} = (2^{64} - 1) * 10^{-12} * 100\% < 10^{10}\ \%$

Min value:
$\rho_{min} = 0$
$\rho_{min} = 0\%$

5 FeeGrowth

FeeGrowth is used to calculate the fee amount within a specified price range. FeeGrowth represents TokenAmount in X or Y per unit of liquidity.

5.1 Primary details

- *primitive type* - u128
- *decimals* - 28
- *symbols* :
  - f - FeeGrowth value
  - λ - tickmap search range is the maximum number of ticks, including tick spacing, that can differentiate the current tick from the target tick within one swap step
  - ϕ - amount of fee denominated TokenAmount type
  - t_l - lower tick index
  - t_u - upper tick index
  - t_c - current tick index
  - f_g - amount of FeeGrowth global, which increases with each swap step
  - $f_{o,i}$ - amount of FeeGrowth associated with t_i
  - $f_i(t_l, t_u)$ - amount of FeeGrowth inside between t_l and t_u
  - $s(t)$ - tick spacing associated with t
  - ς - minimal amount of swap steps required to reach f_{max}
  - $\delta(f)$ - decimals of FeeGrowth type
  - $\delta(L)$ - decimals of Liquidity type
  - $\delta(\sqrt{p})$ - decimals of Price type
  - $\delta(bp)$ - decimals of basis point

FeeGrowth is a type of parameter that can be interpreted as a counter. Therefore, the FeeGrowth leverages the mechanism of underflow and overflow to calculate the difference in the counter state.

- f_g - overflow can occur in a specific pool after increasing FeeGrowth during a swap
- $f_{o,i}$ - underflow can occur in a specific t_i during tick cross, after f_g has overflowed
- $f_i(t_l, t_u)$ - underflow can occur within a specific price range between t_l and t_u, after f_g has overflowed

5.2 Domain

FeeGrowth value is a parameter that accumulates as a result of multiple performed swaps. Let's shift our focus to two aspects separately:

- determine the range of f values in a single swap step
- determine the number of these swap steps

The range of f in a single swap step is calculated by $\frac{\phi}{L}$ quotient. To determine the range of f_g, we should examine Δa tokens required to change square root of price from $\sqrt{p_l}$ to $\sqrt{p_u}$

Let's consider Δy between $\sqrt{p_l}$ and $\sqrt{p_u}$: $\Delta y = |\sqrt{p_u} - \sqrt{p_l}| * L$

It can be transformed to: $\frac{\Delta y}{L} = |\sqrt{p_u} - \sqrt{p_l}|$

Finally, we can determine: $f = \frac{\phi}{L} = \frac{\Delta y * P}{L} = P * |\sqrt{p_u} - \sqrt{p_l}|$

Min value:
The minimum value is determined by the precision composite of $\delta(\sqrt{p})$ and $\delta(bp)$
$f_{min} = \sqrt{\Delta p}_{min} * P_{min}$
$f_{min} = 10^{-\delta(\sqrt{p})} * 10^{-\delta(bp)}$
$f_{min} = 10^{-24} * 10^{-4} = 10^{-28}$

Max value:
$f_{max} = \varsigma * \sqrt{\Delta p}_{max} * 10^{\delta(bp)} = \varsigma * (\sqrt{p}_{u,max} - \sqrt{p}_{l,min}) * 10^{\delta(bp)}$
$f_{max} = \varsigma * (\sqrt{1.0001^{t_{max}}} - \sqrt{1.0001^{t_{max-\lambda*s(t)_{max}}}}) * 10^{\delta(\sqrt{p})+\delta(bp)}$
$\frac{f_{max}}{\varsigma} = (\sqrt{1.0001^{221818}} - \sqrt{1.0001^{196218}}) * 10^{24+4} = 47313 * 10^{28}$
$2^{127} < f_{max} = 719215 * 47313 * 10^{28} < 2^{128} - 1$

5.3 Conversion from fee to FeeGrowth

Function calculates f based on ϕ and L.

```
pub fn from_fee(liquidity: Liquidity, fee: TokenAmount) -> FeeGrowth
```

Formula:
$f = \frac{\phi}{L}$

Domain:
$liquidity = L \in \langle 1, L_{max} \rangle$
$fee = \phi \in \langle 0, \phi_{max} \rangle$
$result = f \in \langle 0, f_{max} \rangle$

Max value:
$\frac{f_{max}}{\varsigma} = (\sqrt{1.0001^{t_{max}}} - \sqrt{1.0001^{t_{max-\lambda*s(t)_{max}}}}) * 10^{\delta(\sqrt{p})+\delta(bp)}$
$2^{108} < \frac{f_{max}}{\varsigma} = (\sqrt{1.0001^{221818}} - \sqrt{1.0001^{196218}}) * 10^{28} < 2^{109}$

Min value:
$f_{min} = \frac{\phi_{min}}{L_{max}} = \frac{0}{2^{128} - 1} = 0$

Max intermediate operations:
$\phi_{max} * 10^{\delta(f)} * 10^{\delta(L)}$
$2^{240} < (2^{128} - 1) * 10^{28} * 10^{6} < 2^{241}$

Max intermediate type:
U256

Edge cases:

- f_{max}
- f_{min}
- $\phi = 0$
- $L = 0$

5.4 Conversion from FeeGrowth to fee

Function calculates ϕ based on f and L.

```
pub fn to_fee(self, liquidity: Liquidity) -> TokenAmount
```

Formula:
$\phi = f * L$

Domain:
$FeeGrowth = f \in \langle 0, f_{max} \rangle$
$liquidity = L \in \langle 0, L_{max} \rangle$
$result = \phi \in \langle 0, a_{max} \rangle$

Max value:
$\phi_{max} = f_{max} * L_{max} < a_{max}$
$2^{127} < \phi_{max} = f_{max} * L_{max} < 2^{128} - 1$

Min value:
$f_{min} * L_{min} = 0 * L_{min} = 0$

Max intermediate operations:
$f_{max} * L_{max}$
$2^{255} < f_{max} * L_{max} < (2^{128} - 1) * (2^{128} - 1) < 2^{256} - 1$

Max intermediate type:
U256

Edge cases:

- ϕ_{max}
- ϕ_{min}
- $f = 0 \vee L = 0$

5.5 Calculate f inside between t_l and t_u

Calculates $f_i(t_l, t_u)$ for specified tick range between t_l and t_u.

```
pub fn calculate_fee_growth_inside(
    tick_lower: i32,
    tick_upper: i32,
    tick_current: i32,
    fee_growth_global_x: FeeGrowth,
    fee_growth_global_y: FeeGrowth,
    tick_lower_fee_growth_outside_x: FeeGrowth,
    tick_lower_fee_growth_outside_y: FeeGrowth,
    tick_upper_fee_growth_outside_x: FeeGrowth,
    tick_upper_fee_growth_outside_y: FeeGrowth,
) -> (FeeGrowth, FeeGrowth)
```

Domain:
$tick\_lower = t_l \in \langle t_{min}, t_{max} \rangle$
$tick\_upper = t_u \in \langle t_{min}, t_{max} \rangle$
$tick\_current = t_c \in \langle t_{min}, t_{max} \rangle$
$tick\_upper > tick\_lower$
$fee\_growth\_global\_x = f_g \in \langle 0, f_{max} \rangle$
$fee\_growth\_global\_y = f_g \in \langle 0, f_{max} \rangle$
$tick\_lower\_fee\_growth\_outside\_x = f_{o,l} \in \langle 0, f_{max} \rangle$
$tick\_lower\_fee\_growth\_outside\_y = f_{o,l} \in \langle 0, f_{max} \rangle$
$tick\_upper\_fee\_growth\_outside\_x = f_{o,u} \in \langle 0, f_{max} \rangle$
$tick\_upper\_fee\_growth\_outside\_y = f_{o,u} \in \langle 0, f_{max} \rangle$
$result = f_i(t_l, t_u) \in \langle 0, f_{max} \rangle$

Formula:
$$f_i(t_l, t_u) = \begin{cases} f_g - f_{o,l} - f_{o,u} & t_l \leq t_c < t_u \\ f_{o,l} - f_{o,u} & t_c \leq t_l < t_u \\ f_{o,u} - f_{o,l} & t_l \leq t_u < t_c \end{cases}$$

Max value:
$f_{i,max}(t_l, t_u) = f_g - f_{o,l} - f_{o,u}$
$f_{i,max}(t_l, t_u) = f_{max} - f_{min} - f_{min}$
$f_{i,max}(t_l, t_u) = 2^{128} - 0 - 0 = 2^{128}$

Min value:
$f_{i,min}(t_l, t_u) = f_g - f_g - 0 = 0$

Max intermediate operations:
$f_{max} - f_{min} - f_{min}$
$2^{128} - 0 - 0 = 2^{128}$

Max intermediate type:
u128

Edge cases:

- controlled underflow of $f_i(t_l, t_u)$

6 SecondsPerLiquidity

SecondsPerLiquidity represents the time difference denominated in seconds per liquidity unit.

6.1 Primary details

- *primitive type* - u128
- *decimals* - 24
- *symbols*:
    - L - value of Liquidity type

- θ - value of SecondsPerLiquidity type
- θ_g - value of SecondsPerLiquidity global, which increases with each swap step
- $\theta_{o,i}$ - value of SecondsPerLiquidity outside associated with t_i
- $\theta_i(t_l, t_u)$ - value of SecondsPerLiquidity inside between t_l and t_u
- $\delta(\theta)$ - decimals of SecondsPerLiquidity type

SecondsPerLiquidity is a type of parameter that can be interpreted as a counter. Therefore, the SecondsPerLiquidity leverages the mechanism of underflow and overflow to calculate the difference in the counter state.

- θ_g - overflow can occur in a specific pool after increasing SecondsPerLiquidity during a swap
- $\theta_{o,i}$ - underflow can occur in a specific t_i during tick cross, after θ has overflowed
- $\theta_i(t_l, t_u)$ - underflow can occur within a specific price range between t_l and t_u, after θ_g has overflowed

6.2 Domain

Max value:
$\theta_{max} = 2^{128} - 1$

Min value:
$\theta_{min} = 0$

6.3 Calculate global SecondsPerLiquidity

Calculates the number of seconds elapsed between the last swap per unit of liquidity.

```
pub fn calculate_seconds_per_liquidity_global(
    liquidity: Liquidity,
    current_timestamp: u64,
    last_timestamp: u64,
) -> SecondsPerLiquidity
```

Domain:
$liquidity = L \in \langle 1, L_{max} \rangle$
$current\_timestamp = T_c \in \langle 0, 2^{64} - 1 \rangle$
$last\_timestamp = T_l \in \langle 0, 2^{64} - 1 \rangle$
$T_c > T_l$
$T_c - T_l \leq 315360000$
$result = \theta \in \langle 0, \theta_{max} \rangle$

Formula:
$$\theta_g = \frac{\Delta T}{L} = \frac{T_c - T_l}{L}$$

Max value:
$\Delta T_{max} = 60 * 60 * 24 * 365 * 10 = 315360000$
$$2^{127} < \frac{\Delta T_{max} * 10^{\delta(\theta)} * 10^{\delta(L)}}{L_{min}} = \frac{315360000 * 10^{30}}{1} < 2^{128}$$

Min value:
$$\frac{\Delta T_{min}}{L} = \frac{0}{L} = 0$$

Max intermediate operations:
$\Delta T_{max} * 10^{\delta(\theta)} * 10^{\delta(L)}$
$2^{127} < \Delta T_{max} * 10^{24} * 10^{6} < 2^{128}$

Max intermediate type:
u128

Edge cases:

- $T_c \leq T_l$
- $L = 0$
- θ_{min}
- θ_{max}

6.4 Calculate θ inside between t_l and t_u

Calculates the number of seconds per liquidity unit within a specified tick range in a liquidity pool.

```
pub fn calculate_seconds_per_liquidity_inside(
    tick_lower: i32,
    tick_upper: i32,
    tick_current: i32,
    tick_lower_seconds_per_liquidity_outside: SecondsPerLiquidity,
    tick_upper_seconds_per_liquidity_outside: SecondsPerLiquidity,
    pool_seconds_per_liquidity_global: SecondsPerLiquidity,
) -> SecondsPerLiquidity
```

Domain:
$tick\_lower = t_l \in \langle t_{min}, t_{max} \rangle$
$tick\_upper = t_u \in \langle t_{min}, t_{max} \rangle$
$tick\_current = t_c \in \langle t_{min}, t_{max} \rangle$
$tick\_lower\_seconds\_per\_liquidity\_outside = \theta_{o,l} \in \langle 0, \theta_{max} \rangle$
$tick\_upper\_seconds\_per\_liquidity\_outside = \theta_{o,u} \in \langle 0, \theta_{max} \rangle$
$pool\_seconds\_per\_liquidity\_global = \theta_g \in \langle 0, \theta_{max} \rangle$
$result = \theta_i \in \langle 0, \theta_{max} \rangle$

Formula:

$$\theta_i(t_l, t_u) = \begin{cases} \theta_g - \theta_{o,l} - \theta_{o,u} & t_l \leq t_c < t_u \\ \theta_l - \theta_{o,u} & t_c \leq t_l < t_u \\ \theta_u - \theta_{o,l} & t_l \leq t_u < t_c \end{cases}$$

Max value:
$\theta_i(t_l, t_u) = \theta_g - \theta_{o,l} - \theta_{o,u}$
$\theta_i(t_l, t_u) = \theta_{max} - \theta_{min} - \theta_{min} = \theta_{max} - 0 - 0 = \theta_{max}$
$\theta_i(t_l, t_u) = 2^{128} - 1$

Min value:
$\theta_i = \theta_g - \theta_g - 0$
$\theta_i = 0$

Edge cases:

- controlled underflow of $\theta_i(t_l, t_u)$

Relationships Between Types

7 Calculate Δx

Calculates the token amounts of X needed to change the price between points A and B, corresponding to their respective price roots and the liquidity between these points. The order of prices at points A and B can be ignored since the formula uses absolute values.

```
pub fn get_delta_x(
    sqrt_price_a: SqrtPrice,
    sqrt_price_b: SqrtPrice,
    liquidity: Liquidity,
    rounding_up: bool,
) -> TokenAmount
```

Domain:
$sqrt\_price\_a = \sqrt{p_a} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$sqrt\_price\_b = \sqrt{p_b} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity = L \in \langle 0, L_{max} \rangle$
$result = \Delta x \in \langle 0, 2^{125} \rangle$

Formula:
$$\Delta x = \frac{L * |\sqrt{p_a} - \sqrt{p_b}|}{\sqrt{p_a} * \sqrt{p_b}}$$

Max value:
$$\Delta x_{max} = \frac{L_{max} * |\sqrt{p_{max}} - \sqrt{p_{min}}|}{\sqrt{p_{max}} * \sqrt{p_{min}}} * 10^{-(\delta(L))}$$

$$2^{124} < \Delta x_{max} = \frac{(2^{128} - 1) * 10^{-6} * (\sqrt{1.0001^{221818}} - \sqrt{1.0001^{-221818}})}{(\sqrt{1.0001^{221818}} * \sqrt{1.0001^{-221818}})} < 2^{125}$$

Min value:
$$\Delta x_{min} = \frac{L_{min} * |\sqrt{p_{min}} - (\sqrt{p_{min}} + 10^{-(\delta(\sqrt{p}))})|}{\sqrt{p_{min}} * (\sqrt{p_{min}} + 10^{-(\delta(\sqrt{p}))})} * 10^{-(\delta(L))} = 0, \text{ when } L_{min} > 0$$

$$\Delta x_{min} = \frac{1 * (\sqrt{1.0001^{-221818}} - (\sqrt{1.0001^{-221818}} + 10^{-24}))}{(\sqrt{1.0001^{-221818}} * (\sqrt{1.0001^{-221818}} + 10^{-24}))} * 10^{-6} = \lfloor 2^{-4} \rfloor = 0, \text{ when } L_{min} > 0$$

Max intermediate operations:
$$\frac{L_{max} * |\sqrt{p_{max}} - \sqrt{p_{min}}|}{10^{\delta(L)}} * 10^{\delta(\sqrt{p})} + \sqrt{p_{max}} * \sqrt{p_{min}}$$

$$\frac{L_{max} * |\sqrt{p_{max}} - \sqrt{p_{min}}|}{10^{6}} * 10^{24} + \sqrt{p_{max}} * \sqrt{p_{min}}$$

$$2^{283} < \frac{(2^{128} - 1) * (\sqrt{1.0001^{221818}} - \sqrt{1.0001^{-221818}}) * 10^{24}}{10^{6}} * 10^{24} +$$

$$+ (\sqrt{1.0001^{221818}} * \sqrt{1.0001^{-221818}}) * 10^{24} < 2^{284}$$

Max intermediate type:
U320

Edge cases:

- Δx_{max}
- Δx_{min}

8 Calculate Δy

Calculates the token amounts of Y needed to change the price between points A and B, corresponding to their respective price roots and the liquidity between these points. The order of prices at points A and B can be ignored since the formula uses absolute values.

```
pub fn get_delta_y(
    sqrt_price_a: SqrtPrice,
    sqrt_price_b: SqrtPrice,
    liquidity: Liquidity,
    rounding_up: bool,
) -> TokenAmount
```

Domain:
$sqrt\_price\_a = \sqrt{p_a} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$sqrt\_price\_b = \sqrt{p_b} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity = L \in \langle 0, L_{max} \rangle$
$result = \Delta y \in \langle 0, 2^{125} \rangle$

Formula:
$\Delta y = L * |\sqrt{p_a} - \sqrt{p_b}|$

Max value:
$\Delta y_{max} = L_{max} * |\sqrt{p_{max}} - \sqrt{p_{min}}| * 10^{-(\delta(L)+\delta(\sqrt{p}))}$
$2^{124} < \Delta y_{max} = (2^{128} - 1) * 10^{-6} * (\sqrt{1.0001^{221818}} - \sqrt{1.0001^{-221818}}) < 2^{125}$

Min value:
$\Delta y_{min} = L * |\sqrt{p_a} - \sqrt{p_b}| * 10^{-(\delta(L)+\delta(\sqrt{p}))} = 0,$
$\Delta y_{min} = \lfloor 1 * 10^{-30} \rfloor = 0, \text{ when } L > 0 \wedge |\sqrt{p_a} > \sqrt{p_b}| > 0$

Max intermediate operations:
$L_{max} * (\sqrt{p_{max}} - \sqrt{p_{min}})$
$2^{223} < (2^{128} - 1) * (\sqrt{1.0001^{221818}} - \sqrt{1.0001^{-221818}}) * 10^{24} < 2^{224}$

Max intermediate type:
U256

Edge cases:

- Δy_{max}
- Δy_{min}

9 Calculate $\sqrt{p_t}$ altering x

Calculates the square root price after adding or subtracting an amount of X tokens to/from the liquidity pool, while considering liquidity in a specific range.

```
pub fn get_next_sqrt_price_x_up(
    starting_sqrt_price: SqrtPrice,
    liquidity: Liquidity,
    x: TokenAmount,
    add_x: bool,
) -> SqrtPrice
```

Domain:
$starting\_sqrt\_price = \sqrt{p_s} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity = L \in \langle 1, L_{max} \rangle$
$x \in \langle 0, x_{max} \rangle$
$result = \sqrt{p_t} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$

Formula:

$$\sqrt{p_t} = \begin{cases} \frac{L * \sqrt{p_s}}{L + x * \sqrt{p_s}}, & \sqrt{p_t} < \sqrt{p_s} \\ \frac{L * \sqrt{p_s}}{L - x * \sqrt{p_s}}, & \sqrt{p_t} \geq \sqrt{p_s} \end{cases}$$

Max value:

$$2^{95} < \frac{L_{max} * \sqrt{p_{max}}}{L_{max} - x_{min} * \sqrt{p_{max}}} < 2^{96}$$

Min value:

$$2^{63} < \frac{L_{max} * \sqrt{p_{min}}}{L_{max} + x_{min} * \sqrt{p_{min}}} < 2^{64}$$

Max intermediate operations:

$$\frac{L_{max} * \sqrt{p_{max}} + 10^{\delta(l)}}{10^{\delta(l)}} * 10^{\delta(\sqrt{p})} + (L_{max} + x_{max} * \sqrt{p_{max}})$$

$$2^{283} < \frac{(2^{128} - 1) * \sqrt{1.0001^{221818}} * 10^{24} + 10^6}{10^6} * 10^{24} + (2^{128} - 1) + (2^{128} - 1) * \sqrt{1.0001^{221818}} * 10^{24} < 2^{284}$$

Max intermediate type:
U320

Edge cases:

- $\sqrt{p_{max}}$
- $\sqrt{p_{min}}$

10 Calculate $\sqrt{p_t}$ altering y

Calculates the square root price after adding or subtracting an amount of Y tokens to/from the liquidity pool, while considering liquidity in a specific range.

```
fn get_next_sqrt_price_y_down(
    starting_sqrt_price: SqrtPrice,
    liquidity: Liquidity,
    y: TokenAmount,
    add_y: bool,
) -> SqrtPrice
```

Domain:
$starting\_sqrt\_price = \sqrt{p_s} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity = L \in \langle 1, L_{max} \rangle$
$y \in \langle 0, y_{max} \rangle$
$result = \sqrt{p_t} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$

Formula:
$$\sqrt{p_t} = \begin{cases} \sqrt{p_s} - \frac{y}{L}, & \sqrt{p_t} < \sqrt{p_s} \\ \sqrt{p_s} + \frac{y}{L}, & \sqrt{p_t} \geq \sqrt{p_s} \end{cases}$$

Max value:
$$2^{95} < \sqrt{p_{max}} - \frac{y_{min}}{L_{max}} < 2^{96}$$

Min value:
$$2^{63} < \sqrt{p_{min}} + \frac{y_{min}}{L_{max}} < 2^{64}$$

Max intermediate operations:
$y_{max} * 10^{2*\delta(\sqrt{p})}$
$2^{287} < (2^{128} - 1) * 10^{24} * 10^{24} < 2^{288}$

Max intermediate type:
U320

Edge cases:

- $\sqrt{p_{max}}$
- $\sqrt{p_{min}}$

11 Calculate $\sqrt{p_t}$ altering a_{in}

Calculates the square root price after adding or subtracting an amount of X or Y tokens to/from the liquidity pool, while considering liquidity in a specific range.

```
fn get_next_sqrt_price_from_input(
    starting_sqrt_price: SqrtPrice,
    liquidity: Liquidity,
    amount: TokenAmount,
    x_to_y: bool,
) -> SqrtPrice
```

Domain:
$starting\_sqrt\_price = \sqrt{p_s} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity = L \in \langle 1, L_{max} \rangle$
$amount = x \vee y \in \langle 0, a_{max} \rangle$
$result = \sqrt{p_t} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$

Formula:
$$\sqrt{p_t} = \begin{cases} \frac{L * \sqrt{p_s}}{L + x * \sqrt{p_s}}, & \sqrt{p_t} > \sqrt{p_s} \\ \sqrt{p_s} + \frac{y}{L}, & \sqrt{p_t} \leq \sqrt{p_s} \end{cases}$$

Max value:
$$\begin{cases} 2^{95} < \frac{L_{max} * \sqrt{p_{max}}}{L_{max} - x_{min} * \sqrt{p_{max}}} < 2^{96} \\ 2^{95} < \sqrt{p_{max}} - \frac{y_{min}}{L_{max}} < 2^{96} \end{cases}$$

Min value:
$$\begin{cases} 2^{63} < \frac{L_{max} * \sqrt{p_{max}}}{L_{max} + x_{min} * \sqrt{p_{max}}} < 2^{64} \\ 2^{63} < \sqrt{p_{max}} + \frac{y_{min}}{L_{max}} < 2^{64} \end{cases}$$

Max intermediate operations:
$x_{max} * 10^{2*\delta(\sqrt{p_s})}$
$2^{287} < (2^{128} - 1) * 10^{24} * 10^{24} < 2^{288}$

Max intermediate type:
U320

Edge cases:

- $a = 0$
- $L = 0$
- $\sqrt{p_{max}}$, when $\sqrt{p_s} > \sqrt{p_t}$
- $\sqrt{p_{max}}$, when $\sqrt{p_t} > \sqrt{p_s}$
- $\sqrt{p_{min}}$, when $\sqrt{p_s} > \sqrt{p_t}$
- $\sqrt{p_{min}}$, when $\sqrt{p_t} > \sqrt{p_s}$

12 Calculate $\sqrt{p_t}$ altering a_{out}

Calculates the square root price after adding or subtracting an amount of X or Y tokens to/from the liquidity pool, while considering liquidity in a specific range. $\sqrt{p_t}$ is returned based on specified amounts of tokens that you receive from the pool, in contrast to the version where you specify a_{in}.

```
fn get_next_sqrt_price_from_output(
    starting_sqrt_price: SqrtPrice,
    liquidity: Liquidity,
    amount: TokenAmount,
    x_to_y: bool,
) -> SqrtPrice
```

Domain:
$starting\_sqrt\_price = \sqrt{p_s} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity = L \in \langle 1, L_{max} \rangle$
$amount = x \vee y \in \langle 0, a_{max} \rangle$
$result = \sqrt{p_t} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$

Formula:
$$\sqrt{p_t} = \begin{cases} L * \dfrac{\sqrt{p_s}}{L - x * \sqrt{p_s}}, & \sqrt{p_t} > \sqrt{p_s} \\ \sqrt{p_s} - \dfrac{y}{L}, & \sqrt{p_t} \leq \sqrt{p_s} \end{cases}$$

Max value:
$$\begin{cases} 2^{95} < \sqrt{p_{max}} - \dfrac{y_{min}}{L_{max}} < 2^{96} \\ 2^{95} < L_{max} * \dfrac{\sqrt{p_{max}}}{L_{max} - x_{min} * \sqrt{p_{max}}} < 2^{96} \end{cases}$$

Min value:
$$\begin{cases} 2^{63} < \sqrt{p_{min}} + \dfrac{y_{min}}{L_{max}} < 2^{64} \\ 2^{63} < L_{max} * \dfrac{\sqrt{p_{min}}}{L_{max} + x_{min} * \sqrt{p_{min}}} < 2^{64} \end{cases}$$

Max intermediate operations:
$x_{max} * 10^{2*\delta(\sqrt{p})}$
$2^{287} < (2^{128} - 1) * 10^{24} * 10^{24} < 2^{288}$

Max intermediate type:
U320

Edge cases:

- $a = 0$
- $L = 0$
- $\sqrt{p_{max}}$, when $\sqrt{p_s} > \sqrt{p_t}$
- $\sqrt{p_{max}}$, when $\sqrt{p_t} > \sqrt{p_s}$
- $\sqrt{p_{min}}$, when $\sqrt{p_s} > \sqrt{p_t}$
- $\sqrt{p_{min}}$, when $\sqrt{p_t} > \sqrt{p_s}$

13 Calculate (Δx, Δy) between t_l and t_u

Calculate the required amounts of token X and Y when adding or removing liquidity from the pool within a specified price range and liquidity delta. The price range is determined by lower and upper ticks, and the liquidity direction is indicated by the liquidity sign.

```
pub fn calculate_amount_delta(
    current_tick_index: i32,
    current_sqrt_price: SqrtPrice,
    liquidity_delta: Liquidity,
```

```
    liquidity_sign: bool,
    upper_tick: i32,
    lower_tick: i32,
) -> (TokenAmount, TokenAmount, bool)
```

Domain:
$current\_tick\_index = t_c \in \langle t_{min}, t_{max} \rangle$
$current\_sqrt\_price = \sqrt{p_c} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity\_delta = \Delta L \in \langle 0, L_{max} \rangle$
$upper\_tick = t_u \in \langle t_{min}, t_{max} \rangle$
$lower\_tick = t_l \in \langle t_{min}, t_{max} \rangle$
$upper\_tick > lower\_tick$
$\sqrt{p}_l = \sqrt{1.0001^{t_l}}$
$\sqrt{p}_u = \sqrt{1.0001^{t_u}}$
$result = (\Delta x, \Delta y) \in (\langle 0, x_{max} \rangle, \langle 0, y_{max} \rangle)$

Formula:

$$(\Delta x, \Delta y) = \begin{cases} (\Delta x(\sqrt{p_l}, \sqrt{p_u}, \Delta L), 0) & t_c < t_l < t_u \\ (\Delta x(\sqrt{p_c}, \sqrt{p_u}, \Delta L), \Delta y(\sqrt{p_l}, \sqrt{p_c}, \Delta L)) & t_l < t_c < t_u \\ (0, \Delta y(\sqrt{p_l}, \sqrt{p_u}, \Delta L)) & t_l < t_u \leq t_c \end{cases}$$

Max value:
$(2^{124}, 0) < (\Delta x(\sqrt{p_{min}}, \sqrt{p_{max}}, \Delta L_{max}), 0) < (2^{125}, 0)$
$(0, 2^{124}) < (0, \Delta y(\sqrt{p_{min}}, \sqrt{p_{max}}, \Delta L_{max})) < (0, 2^{125})$

Min value:
$(\Delta y(\sqrt{p_{min}}, \sqrt{p_{min}}, \Delta L_{min}), 0) = (0, 0)$
$(0, \Delta y(\sqrt{p_{min}}, \sqrt{p_{min}}, \Delta L_{min})) = (0, 0)$

Max intermediate operations:
$\frac{\Delta L_{max} * |\sqrt{p_{max}} - \sqrt{p_{min}}|}{10^{\delta(L)}} * 10^{\delta(\sqrt{p})} + \sqrt{p_{max}} * \sqrt{p_{min}}$
$2^{283} < \frac{(2^{128} - 1) * (\sqrt{1.0001^{221818}} - \sqrt{1.0001^{-221818}}) * 10^{2*24}}{10^6} + (\sqrt{1.0001^{221818}} * \sqrt{1.0001^{-221818}}) *$
$10^{24} < 2^{284}$

Max intermediate type:
U320

Edge cases:

- Δx_{max}
- Δy_{max}
- Δx_{min}
- Δy_{min}
- $\Delta L = 0$

14 Examine $\sqrt{p}$ sensitivity

Determines whether a given amount of tokens (X or Y) is sufficient to cause a change in the square root price of a pool, considering liquidity, fee percentage on pool, swap direction, and whether the specified amount concerns the input or output tokens.

```
pub fn is_enough_amount_to_push_price(
    amount: TokenAmount,
    current_sqrt_price: SqrtPrice,
    liquidity: Liquidity,
    fee: Percentage,
    by_amount_in: bool,
    x_to_y: bool,
) -> bool
```

Domain:
$amount = a \in \langle 0, a_{max} \rangle$
$current\_sqrt\_price = \sqrt{p_s} \in \langle \sqrt{p_{min}}, \sqrt{p_{max}} \rangle$
$liquidity = L \in \langle 1, L_{max} \rangle$
$fee = P \in \langle 0, P_{max} \rangle$
$result = i \in \langle 0, 1 \rangle$

Formula:
$i = \sqrt{p_s} \neq \sqrt{p_t}$

Max intermediate operations:
$a_{max} * 10^{2*\delta(p)}$
$2^{287} < (2^{128} - 1) * 10^{24} * 10^{24} < 2^{288}$

Max intermediate type:
U320

Edge cases:

- P_{max}
- $L = 0$
- a_{min}
- a_{max}

15 Calculate swap step in continuous range $\in \langle \sqrt{p_s}, \sqrt{p_t} \rangle$

Calculate the result of a token exchange within a specified range, taking into account factors such as current and target prices, liquidity, input or output amounts (X or Y), and fees. This calculation determines the square root of the price after the swap and provides information on the input and output amounts, as well as the fees associated with the input amount

```
pub fn compute_swap_step(
    current_sqrt_price: SqrtPrice,
    target_sqrt_price: SqrtPrice,
    liquidity: Liquidity,
    amount: TokenAmount,
    by_amount_in: bool,
```

```
    fee: Percentage,
) -> SwapResult

pub struct SwapResult {
    pub next_sqrt_price: SqrtPrice,
    pub amount_in: TokenAmount,
    pub amount_out: TokenAmount,
    pub fee_amount: TokenAmount,
}
```

Calculating a single step within the swap follows the algorithm below:

1. Determine whether L = 0. If it does, proceed directly to $\sqrt{p_t}$ and return to indicate that no swap has been executed.
2. Calculate the available amount of $a_{\sqrt{p_s}\to\sqrt{p_t}}$ tokens X or Y, swapping from $\sqrt{p_s}$ to $\sqrt{p_t}$. If $a_{\sqrt{p_s}\to\sqrt{p_t}} \leq a$, execute a complete swap from $\sqrt{p_s}$ to $\sqrt{p_t}$ and return the SwapResult.
3. In case $a_{\sqrt{p_s}\to\sqrt{p_t}} \geq a$, the swap will come to a halt somewhere between $\sqrt{p_s}$ and $\sqrt{p_t}$. Therefore, it is necessary to calculate the next square root of price: $\sqrt{p_n}$ altering a_{in} or a_{out}.
4. After establishing $\sqrt{p_n}$ as the swap end-point, it will be used to calculate the $a_{\sqrt{p_s}\to\sqrt{p_n}}$ token amounts of X or Y, required to change the square root of price from $\sqrt{p_s}$ to $\sqrt{p_n}$.
5. Having the details of $\sqrt{p_n}$ and $a_{\sqrt{p_s}\to\sqrt{p_n}}$, the SwapResult object can be constructed and then returned.

Domain:
$current\_sqrt\_price = \sqrt{p_s} \in \langle\sqrt{p_{min}}, \sqrt{p_{max}}\rangle$
$target\_sqrt\_price = \sqrt{p_t} \in \langle\sqrt{p_{min}}, \sqrt{p_{max}}\rangle$
$liquidity = L \in \langle 0, L_{max}\rangle$
$amount = a \in \langle 0, a_{max}\rangle$
$fee = P \in \langle 0, 10^{\delta(P)}\rangle$

$next\_sqrt\_price = \sqrt{p_n} \in \langle\sqrt{p_{min}}, \sqrt{p_{max}}\rangle$
$amount\_in = a_{in} \in \langle 1, a_{max}\rangle$
$amount\_out = a_{out} \in \langle 1, a_{max}\rangle$
$fee\_amount = \phi \in \langle 0, a_{max}\rangle$

Edge cases:

- $L = 0$
- $P_{min} \Rightarrow \phi = 0$
- $P_{max} \Rightarrow a_{in} = 0$
- $a_{in,max}$
- $a_{out,max}$
- $\sqrt{p_t} = \sqrt{p_n}$
- $\sqrt{p_t} \neq \sqrt{p_n}$