Problem

- Solidity lacks native support for fixed-point arithmetic
- Implementing advanced math functions is hard
- Few math libraries use the latest v0.8 features

Every Solidity developer is a mathematician (well, sort of)

How to calculate decimal value?

Asked 14 days ago Modified 13 days ago Viewed 23 times

0

```
function claim() public {
    uint myWai = 10000000000;
    uint totalWei = 20000000000;
    uint myPercentage = myWai/totalWei * 100;

    payable(msg.sender).transfer(12345/100*myPercentage);
}
```

I expect that 'myWay/totalWei' will be '0.5' but actually it return '0'. How to calculate decimal numbers??

My solidity version is 0.8

solidity Edit tags

Share Edit Follow Close Flag

asked Apr 4 at 16:09

jjjoo

3 ● 1

Solution: PRBMath!

- Modern math lib operating with 18-decimal numbers
- Has logarithms, exponentials, square roots and more
- Gas efficient, but still user-friendly
- Well documented and commented

Developer Experience

- NatSpec comments. Everywhere.
- Complementary comments in the function bodies.
- Clear installation and usage guides.

```solidity
/// @notice Calculates the binary logarithm of x.
///
/// @dev Based on the iterative approximation algorithm.
/// https://en.wikipedia.org/wiki/Binary_logarithm#Iterative_approximation
///
/// Requirements:
/// - x must be greater than zero.
///
/// Caveats:
/// - The results are nor perfectly accurate to the last digit, due to the lossy precision of the iterative approximation.
function log2(int256 x) internal pure returns (int256 result) {
    require(x > 0);
    unchecked {
        // This works because log2(x) = -log2(1/x).
        int256 sign;
        if (x >= SCALE) {
            sign = 1;
        } else {
            sign = -1;
            // Do the fixed-point inversion inline to save gas. The numeretor is SCALE * SCALE.
            assembly {
                x := div(100000000000000000000000000000000000, x)
            }
        }

        // Calculate the integer part of the logarithm and add it to the result and finally calculate y = x * 2^(-n).
        uint256 n = PRBMathCommon.mostSignificantBit(uint256(x / SCALE));

        // The integer part of the logarithm as a signed 59.18-decimal fixed-point number. The operation can't overflow
        // beacuse n is maximum 255, SCALE is 1e18 and sign is either 1 or -1.
        result = int256(n) * SCALE;

        // This is y = x * 2^(-n).
        int256 y = x >> n;

        // If y = 1, the fractional part is zero.
        if (y == SCALE) {
            return result * sign;
        }

        // Calculate the fractional part via the iterative approximation.
        // The "delta >>= 1" part is equivalent to "delta /= 2", but shifting bits is faster.
        for (int256 delta = int256(HALF_SCALE); delta > 0; delta >>= 1) {
            y = (y * y) / SCALE;

            // Is y^2 > 2 and so in the range [2,4)?
            if (y >= 2 * SCALE) {
                // Add the 2^(-m) factor to the logarithm.
                result += delta;

                // Corresponds to z/2 on Wikipedia.
                y >>= 1;
            }
        }
        result *= sign;
    }
}
```

frankie
@FrankieIsLost

Replying to @0xTomoyo

Highly recommend PRBMath for a cleanly written and commented fixed point arithmetic library. They use the same algo but include an explanation for the bitshifts: setting an initial guess to the closest power of 2 greater than x github.com/hifi-finance/p...

```solidity
// Set the initial guess to the closest power of two that is higher than x.
uint256 xAux = uint256(x);
result = 1;
if (xAux >= 0x100000000000000000000000000000000) {
    xAux >>= 128;
    result <<= 64;
}
if (xAux >= 0x10000000000000000) {
    xAux >>= 64;
    result <<= 32;
}
if (xAux >= 0x100000000) {
    xAux >>= 32;
    result <<= 16;
}
if (xAux >= 0x10000) {
    xAux >>= 16;
    result <<= 8;
}
if (xAux >= 0x100) {
```

Error Handling

Custom errors are the future of Solidity error reporting.

```
pragma solidity >=0.8.4;

/// @notice Emitted when the input is less than or equal to zero.
error PRBMathSD59x18__LogInputTooSmall(int256 x);

/// @notice Emitted when one of the inputs is MIN_SD59x18.
error PRBMathSD59x18__MulInputTooSmall();

/// @notice Emitted when the intermediary absolute result overflows SD59x18.
error PRBMathSD59x18__MulOverflow(uint256 rAbs);

/// @notice Emitted when the intermediary absolute result overflows SD59x18.
error PRBMathSD59x18__PowuOverflow(uint256 rAbs);
```

Testing and Security

- ~1,400 test cases
- ~96% test coverage
- Audit coming later this year

Use Cases

- Percentages and proportionalities
- AMM curves
- Interest rates
- Derivative pricing
- TWAP oracles
- Custom trading algos

```solidity
pragma solidity >=0.8.4;

import "@prb/math/contracts/PRBMathUD60x18.sol";

contract Percentage {
    using PRBMathUD60x18 for uint256;

    function getPercentage(uint256 a, uint256 b) public pure returns (uint256)
    {
        uint256 oneHundredPercent = PRBMathUD60x18.fromUint(100);
        return a.mul(oneHundredPercent).div(b);
    }
}
```

Yield Space AMM

$$\left(\frac{y}{x}\right)^t = \left(\frac{\left(x_{start}^{1-t} + y_{start}^{1-t} - x^{1-t}\right)^{\frac{1}{1-t}}}{x}\right)^t$$

```solidity
pragma solidity >=0.8.4;

import "@prb/math/contracts/PRBMathUD60x18.sol";

contract YieldSpace {
    using PRBMathUD60x18 for uint256;

    uint256 internal constant k = 7927447996;
    uint256 internal constant g = 10526315789473684 21;

    function yieldSpace(uint256 x_s, uint256 y_s, uint256 x, uint256 ttm) external pure
returns (uint256 y) {
        uint256 t = k.mul(ttm);
        uint256 a = 1e18 - g.mul(t);
        y = (x_s.pow(a) + y_s.pow(a) - x.pow(a)).pow(a.inv());
    }
}
```

Future of PRBMath

- Moving to Foundry (fuzzing)
- More type safety via user-defined value types
- User-defined operators

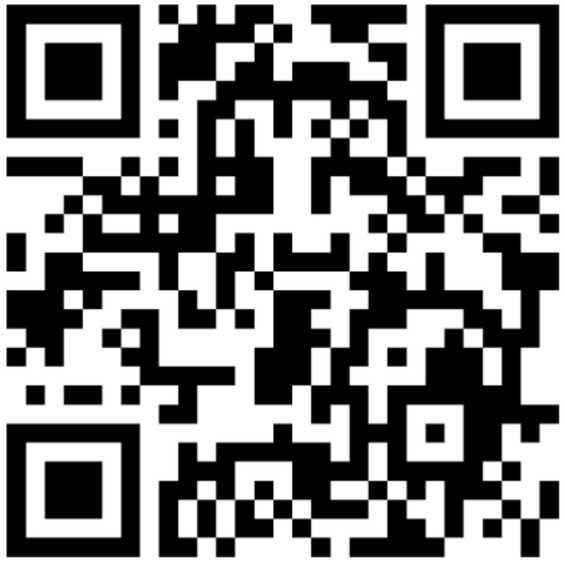
```
pragma solidity >=0.8.4;

import { UD60x18 } from "@prb/math/UD60x18.sol";

contract Add {
    function add(UD60x18 a, UD60x18 b) public pure returns (uint256)
{       return a.add(b);
        // in the future: "return a + b";
    }
}
```

Thanks for Coming

PRBMath is released under the “Unlicense” license.
Find me on Twitter and GitHub @PaulRBerg.



```
yarn add @prb/math
```

```
forge install paulrberg/prb-math
```