- 1 Removal + 1 Addition

```solidity
pragma solidity ^0.5.16;

import "./InterestRateModel.sol";
import "./SafeMath.sol";

/**
  * @title Compound's WhitePaperInterestRateModel Contract
  * @author Compound
  * @notice The parameterized model described in section 2.4 of the original Compound Protocol whitepaper
  */
contract WhitePaperInterestRateModel is InterestRateModel {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock);

    /**
     * @notice The approximate number of blocks per year that is assumed by the interest rate model
     */
    uint public constant blocksPerYear = 2102400;

    /**
     * @notice The multiplier of utilization rate that gives the slope of the interest rate
     */
    uint public multiplierPerBlock;

    /**
     * @notice The base interest rate which is the y-intercept when utilization rate is 0
     */
    uint public baseRatePerBlock;

    /**
     * @notice Construct an interest rate model
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     */
    constructor(uint baseRatePerYear, uint multiplierPerYear) public {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = multiplierPerYear.div(blocksPerYear);

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock);
    }

    /**
     * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
```

```solidity
pragma solidity ^0.5.16;

import "./InterestRateModel.sol";
import "./SafeMath.sol";

/**
  * @title Compound's WhitePaperInterestRateModel Contract
  * @author Compound
  * @notice The parameterized model described in section 2.4 of the original Compound Protocol whitepaper
  */
contract WhitePaperInterestRateModel is InterestRateModel {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock);

    /**
     * @notice The approximate number of blocks per year that is assumed by the interest rate model
     */
    uint public constant blocksPerYear = 6000000;

    /**
     * @notice The multiplier of utilization rate that gives the slope of the interest rate
     */
    uint public multiplierPerBlock;

    /**
     * @notice The base interest rate which is the y-intercept when utilization rate is 0
     */
    uint public baseRatePerBlock;

    /**
     * @notice Construct an interest rate model
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     */
    constructor(uint baseRatePerYear, uint multiplierPerYear) public {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = multiplierPerYear.div(blocksPerYear);

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock);
    }

    /**
     * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market (currently unused)
     * @return The utilization rate as a mantissa between [0, 1e18]
     */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }

    /**
     * @notice Calculates the current borrow rate per block, with the error code expected by the market
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
        uint ur = utilizationRate(cash, borrows, reserves);
        return ur.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
    }

    /**
     * @notice Calculates the current supply rate per block
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @param reserveFactorMantissa The current reserve factor for the market
     * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRate(cash, borrows, reserves);
        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }
}
```

```
     * @param reserves The amount of reserves in the market (currently unused)
     * @return The utilization rate as a mantissa between [0, 1e18]
     */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }

    /**
     * @notice Calculates the current borrow rate per block, with the error code expected by the market
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
        uint ur = utilizationRate(cash, borrows, reserves);
        return ur.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
    }

    /**
     * @notice Calculates the current supply rate per block
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @param reserveFactorMantissa The current reserve factor for the market
     * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRate(cash, borrows, reserves);
        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }
}
```