Online accompaniement for "Collaborative and parametric insurance on the Ethereum blockchain"

Pierre-Olivier Goffard [*1] and Stéphane Loisel[†2]

[1] Université de Strasbourg, Institut de Recherche Mathématique Avancée, Strasbourg, France

[2] CNAM, Paris, France

June 20, 2025

We are going to review the solidity code of the smart contract discussed in the paper entitled "Collaborative and parametric insurance on the Ethereum blockchain". The core mechanism of our parametric insurance is implemented in the contract `InsuranceLogic.sol`. It relies on two helper contract, `PricingLogic.sol` that encodes the pricing formula and `ModelPointsLogic.sol` that performs the model points aggregation of the portfolio. We first have a look at the helper contracts `PricingLogic.sol` in Section 1 and `ModelPointsLogic.sol` in **??**.

1 `PricingLogic.sol`

We start by indicating the version of the compiler and the name of the smart contract in code fragment 1.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

contract PricingLogic {
    ---
}
```

Code Fragment 1: Declaring a compiler and define `PricingLogic` contract.

[*]Email: goffard@unistra.fr.

[†]Email: stephane.loisel@lecnam.net.

Solidity is a high-level programming language designed for smart contract development, where instructions are terminated with semicolons. The language is continuously evolving, with each stable release tied to a specific compiler version. This versioning ensures backward compatibility, allowing developers to compile code written in older Solidity versions and maintain stability for existing contracts. The following code fragments are intended to populate the space between the brackets following the contract declaration. In Code Fragment 2, we declare several variables. When declaring a variable in Solidity, one must specify its type,

```
uint256 public constant factor = 10 ** 4; // Numerical precision
address public owner; // Ethereum address of the owner of the contract


// Define a struct to hold polynomial coefficients
struct PolyCoeffs {
    int256 c0;
    int256 c1;
    int256 c2;
    int256 c3;
    int256 c4;
}

// Mapping from station name to polynomial coefficients
mapping(string => PolyCoeffs) public stationCoeffs;
```

Code Fragment 2: Variable declaration of the `PricingLogic.sol` contract.

visibility, and name. A variable of type `address` stores a 20-byte value that represents the location of accounts (both externally owned accounts and smart contracts) on the Ethereum network. In Solidity, numbers can only be stored as integer values in variables of type `uint.x`, where x specifies the bit length of the number. Other basic types include strings, booleans, and fixed-size byte arrays[1]. A `struct`is a custom data type that stores the coefficient of the polynomial use to smooth the probability of the triggering evet as a function of the day of the year. Mappings in Solidity function as data dictionaries. Here, we create a mapping in order to retrieve the coefficinet of the polynomial asociated to the probability of the triggering events in Marseille or Strasbourg. The constructor method in 3 initialize the contract's data.

[1] See https://docs.soliditylang.org/en/latest/types.html

```
constructor() {
    owner = msg.sender;

    // Initialize coefficients for "MARSEILLE-MARIGNANE"
    stationCoeffs["MARSEILLE-MARIGNANE"] = PolyCoeffs(
       61866466488584768,    // 6.08088853e-02 * 1e11
        1279067435587755,     // 1.30072825e-03 * 1e11
        -22081841812020,      // -2.22982823e-05 * 1e11
        109525424150,         // 1.10635770e-07 * 1e11
        -160548258           // -1.62373694e-10 * 1e11
    );

    // Initialize coefficients for "STRASBOURG-ENTZHEIM"
    stationCoeffs["STRASBOURG-ENTZHEIM"] = PolyCoeffs(
        76579341005774688,     // 7.53896117e-02 * 1e11
        -322886972669433,       // -2.16506423e-04 * 1e11
        13084236350914,        // 1.17177460e-05 * 1e11
        -66687593698,          // -6.10554025e-08 * 1e11
        91781331               // 8.44071936e-11 * 1e11
    );
}
```

Code Fragment 3: Constructor method of `PricingLogic`.

Here it sets the value of coefficient of the polynomial approximation of the triggering event probability in Marseille and Strasbourg. The conract only have one mehod called `getProbability` taht takes as argument `T` the day of the year and `station` which indicates the location. The code is provided in code fragment **??**.

This method returns the probability that that the level of precipitation exceeds 5mm on a particular day and at a particular location. This method will be called within the `InsuranceLogic` contract to provide a quote to customer.

2 ModelPointsLogic.sol

The second auxiliary contract is called `ModelPointsLogic`, see code fragmet 5.

```
    function getProbability(uint256 T, string memory station) public view returns (uint)
         {
        // Retrieve the coefficients for the given station
        PolyCoeffs memory coeffs = stationCoeffs[station];

        // // Scale T by 1e10 for fixed-point arithmetic
        int256 scaledT = int256(T);
        int result = coeffs.c4; // Start with the highest degree coefficient

        // // Evaluate the polynomial using Horner's method
        result = result * scaledT + coeffs.c3;
        result = result * scaledT + coeffs.c2;
        result = result * scaledT + coeffs.c1;
        result = result * scaledT + coeffs.c0;

        // Return the result scaled down by 1e18
        return uint256(result * int256(factor) / 1e18);
    }
```

Code Fragment 4: Declaring a compiler and define a smart contract.

```
pragma solidity 0.8.19;

contract ModelPointsLogic {
    ---
}
```

Code Fragment 5: Declaring a compiler and define `ModelPointsLogic` contract.

The goal of this smart contract is to monitor our parametric insurance portfolio by keeping track of the mean, variance and skewness of the liability distribution. The liability of the parametric insurance portfolio corresponds to the pay out asociated to model points that reprsents a group of contract. The variable of `ModelPointsLogic` are given in code fragment 6.

The variable `mu` is the mean of the liability, the variable `sn2` is the variance of the liability and `gamma1_unormalized`

```
address public owner; // Ethereum address of the owner of the contract
uint256 public constant factor = 10 ** 4; // Numerical precision
uint256 public mu = 0;
uint256 public sn2 = 0;
uint256 public gamma1_unormalized = 0; // Skewness times std to the power 3 of the
    liability
uint256 public Nt = 0 ; // Counter for the number of insurance contracts
uint256 public Nt_MP = 0; // Counter for the number of model points ever created
uint256 public Nt_MP_active = 0; // Counter for the number of active model points
```

Code Fragment 6: Variables of the `ModelPointsLogic` contract.

is the skewness of the liability time the stadard deviation to the power 3/2 which allows us to just sum of the unormalized skewness coeficients associated to each model points. We keep track of the number of parametric insurance cintract, the number of model points ever created but also and more importantly the number of active model points that is those for which the teh event date is still ahead of us. Note that the number of active model points must be higher than a threshold (5 in our implementation) in order to use the normal approximation to calculate the solvency capitals. In code fragment 7 we define two data structure to store the informations of parametric insurance contracts and model points respectively.

Sets of parametric isnurance contracts and model points are then indexed using mappings. The constructor method is not so inetresting for the `ModelPointsLogic` contract and so we omit it here. Code fragments 8, 9 and 10 show the `addInsuranceContract` method.

When a new parametric insurance contract is added to the portfolio, the portfolio of model points is updated along with the moments of the liability. These informations are then forwarded to the `InsuranceLogic` contract that calculates the solvency capital requirements. We also have a method to remove a policy from the portfolio as it can be closed or cancelled. The solidity code is provided in coe fragaments 11 and 12.

```
// Define a struct to represent an insurance contract
struct InsuranceContract {
    string station;
    uint256 T; // day in a year
    uint256 p; // probability
    uint256 l; // compensation
    string status; // status of the contract

}

// // Define a struct to represent a model point
struct ModelPoint {
    string station;
    uint256 T; // day in a year
    uint256 p; // probability
    uint256 l_sum; // Sums of the compensations
    uint256 mean; // mean of the Model Point
    uint256 variance; // variance of the Model Point
    uint256 skew_unormalized; // skewness times std to the power 3 of the Model
        Point
}

// Mapping from uint to an array of InsuranceContract
mapping(uint256 => InsuranceContract) public insuranceContracts;

// Mapping from uint to an array of ModelPoint
mapping(uint256 => ModelPoint) public modelPoints;
```

Code Fragment 7: Data structures of the `ModelPointsLogic` contract.

As the smart contract may be going under a reset if the surplus is falling below the Minimum Capital Required then all the remaining active policies are being cancelled. The portfolio then needs to be updated which is possible with the `cancelAllActiveContracts` method of code fragment 13.

```
function addInsuranceContract(string memory station, uint256 T, uint256 p, uint256 l
    ) public {
    // Calculate mean, variance, and skewness of the payout
    uint256 mean;
    uint256 variance;
    uint256 skew_unormalized;

    // Increment the contract counter
    Nt++;
    // Add the new insurance contract to the mapping
    insuranceContracts[Nt] = InsuranceContract({
        station: station,
        T: T,
        p: p,
        l: l,
        status: "active"
    });
```

Code Fragment 8: Part 1 of addInsuranceContract method.

3 InsuranceLogic.sol

The core mechanism of our parametric insurance solution on the blockchain is implemented in InsuranceLogic. We start by declaring the dependencies of our contract with respect to PricingLogic and ModelPointsLogic in code fragment 14.

This is done using interface in which we also indicate the state variables and methods from PricingLogic and ModelPointsLogic that we will be using. We then declare our smart contract together with its variables in code fragment 15.

We keep track of the total number of protocol token, the exchange of token against ETH and the surplus. We have the initial values of the parameters corresponding to the safety loading and the risk level associated to the solvency capitals. We monitor the number of parametric insurance contract as well as the number of model points. The moments of the liability are derived from the informations provided by ModelPointsLogic. Code fragment 16 defines a data structure to store the information associated to a parametric insurance contract.

```
// Check if a model point with the same T and station exists
bool modelPointExists = false;
for (uint256 i = 1; i <= Nt_MP; i++) {
    if (keccak256(abi.encodePacked(modelPoints[i].station)) == keccak256(abi.
        encodePacked(station)) && modelPoints[i].T == T) {
        // Update the mean, variance and skewness of the liability
        if (modelPoints[i].l_sum == 0) {
            Nt_MP_active++;
        }
        mu = mu - modelPoints[i].mean;
        sn2 = sn2 - modelPoints[i].variance;
        gamma1_unormalized = gamma1_unormalized - modelPoints[i].
            skew_unormalized;
        // Update the mean, variance, and skew of the existing model point
        uint256 l_sum = modelPoints[i].l_sum + l;
        modelPoints[i].l_sum = l_sum;
        mean = (p * l_sum) / factor;
        variance = (p * (factor - p)* l_sum * l_sum) / (factor * factor);
        skew_unormalized = p * l_sum * l_sum * l_sum / factor - 3 * mean *
            variance - mean * mean * mean;
        modelPoints[i].mean = mean;
        modelPoints[i].variance = variance;
        modelPoints[i].skew_unormalized = skew_unormalized;
        mu = mu + modelPoints[i].mean;
        sn2 = sn2 + modelPoints[i].variance;
        gamma1_unormalized = gamma1_unormalized + modelPoints[i].
            skew_unormalized;
        modelPointExists = true;
        break;
    }
}
```

Code Fragment 9: Part 2 of addInsuranceContract method.

```
        // If no model point exists, create a new one
        if (!modelPointExists) {
            Nt_MP++;
            Nt_MP_active++;
            mean = (p * l) / factor;
            variance = (p * (factor - p) * l * l) / (factor * factor);
            skew_unormalized = p * l * l * l / factor - 3 * mean * variance - mean *
                mean * mean;

            modelPoints[Nt_MP] = ModelPoint({
                station: station,
                T: T,
                p: p,
                l_sum: l,
                mean: mean,
                variance: variance,
                skew_unormalized: skew_unormalized

            });
            mu = mu + mean;
            sn2 = sn2 + variance;
            gamma1_unormalized = gamma1_unormalized + skew_unormalized;
        }
    }
```

Code Fragment 10: Part 3 of `addInsuranceContract` method.

We further define a mapping to retrieve the policies in our portfolio. Surplus providers lock fund in the sart contrcat in exchange of what they get protocol token. Code fragment 17

```
function removeInsuranceContract(uint256 index, string memory newStatus) public {
    // require(msg.sender == owner, "Only the owner can remove contracts");
    require(index > 0 && index <= Nt, "Invalid contract index");
    require(
        keccak256(abi.encodePacked(insuranceContracts[index].status)) == keccak256(
            abi.encodePacked("active")),
        "Contract is not active"
    );
    require(
        keccak256(abi.encodePacked(newStatus)) == keccak256(abi.encodePacked("
            cancelled")) ||
        keccak256(abi.encodePacked(newStatus)) == keccak256(abi.encodePacked("closed
            ")),
        "Invalid status"
    );

    // Update the status of the contract
    insuranceContracts[index].status = newStatus;

    // Retrieve the contract details
    string memory station = insuranceContracts[index].station;
    uint256 T = insuranceContracts[index].T;
    uint256 l = insuranceContracts[index].l;
```

Code Fragment 11: Part 1 of removeInsuranceContract method.

```
        // Update the corresponding model point
        for (uint256 i = 1; i <= Nt_MP; i++) {
            if (keccak256(abi.encodePacked(modelPoints[i].station)) == keccak256(abi.
                encodePacked(station)) && modelPoints[i].T == T) {
                mu = mu - modelPoints[i].mean;
                sn2 = sn2 - modelPoints[i].variance;
                gamma1_unormalized = gamma1_unormalized - modelPoints[i].
                    skew_unormalized;

                // Update the model point's l_sum, mean, and variance
                modelPoints[i].l_sum -= l;
                if (modelPoints[i].l_sum == 0) {
                    Nt_MP_active--;
                    modelPoints[i].mean = 0;
                    modelPoints[i].variance = 0;
                    modelPoints[i].skew_unormalized = 0;
                } else {
                    uint256 mean = (modelPoints[i].p * modelPoints[i].l_sum) / factor;
                    uint256 variance = (modelPoints[i].p * (factor - modelPoints[i].p) *
                         modelPoints[i].l_sum * modelPoints[i].l_sum) / (factor * factor)
                        ;
                    uint256 skew_unormalized = modelPoints[i].p * modelPoints[i].l_sum *
                         modelPoints[i].l_sum * modelPoints[i].l_sum / factor - 3 * mean
                        * variance - mean * mean * mean;
                    modelPoints[i].mean = mean;
                    modelPoints[i].variance = variance;
                    modelPoints[i].skew_unormalized = skew_unormalized;
                    mu = mu + mean;
                    sn2 = sn2 + variance;
                    gamma1_unormalized = gamma1_unormalized + skew_unormalized;
                }
                break;
            }
        }
    }
```

Code Fragment 12: Part 2 of removeInsuranceContract method.

```
function cancelAllActiveContracts() public {
        for (uint256 i = 1; i <= Nt; i++) {
            if (keccak256(bytes(insuranceContracts[i].status)) == keccak256(bytes("
                active"))) {
                removeInsuranceContract(i, "cancelled");
            }
        }
    }
```

Code Fragment 13: cancelAllActiveContracts method.

```
pragma solidity 0.8.19;

interface IPricingLogic {
    function getProbability(uint256 T, string memory station) external view returns (
        uint);
}
interface IModelPointsLogic {
    function sn2() external view returns (uint256);
    function gamma1_unormalized() external view returns (uint256);
    function Nt_MP_active() external view returns (uint256);
    function addInsuranceContract(string memory station, uint256 T, uint256 p, uint256 l
        ) external;
    function removeInsuranceContract(uint256 index, string memory newStatus) external;
    function cancelAllActiveContracts() external;
}
```

Code Fragment 14: Dependency declaration of InsuranceLogic to PricingLogic and ModelPointLogic .

```
contract InsuranceLogic{
    address public owner; // Ethereum address of the owner of the contract
    uint256 public constant factor = 10 ** 4; // Numerical precision
    uint256 public totalSupply; // Total supply of tokens
    uint256 public rt = 10000; // Exchange rate of token against ETH
    uint256 public Xt; // Reserve to calculate the exchange rate
    uint16 public eta = 1000; // 10% loading (1000 basis points) to reward the token
        holders
    uint16 public qAlphaSCR = 25758; // standard normal quantile of order alpha = 0.995
    uint16 public qAlphaMCR = 10364; // standard normal quantile of order alpha = 0.995
    uint256 public Nt; // Counter for the number of insurance contracts
    uint256 public Nt_MP_active; // Solvency Capital Requirement
    uint256 public SCR; // Solvency Capital Requirement
    uint256 public MCR; // Minimum Capital Requirement
    uint256 public mu = 0; // mean of the liability distribution
    uint256 public sn = 0; // std of the liability distribution
    uint256 public gamma1 = 0; // Skewness of the liability distribution
    uint256 public l_tot = 0; // Sum of all the potential payouts
    uint256 public Nt_MP_active_min = 5; // Minimum number of model points to trigger
        SCR and MCR
```

Code Fragment 15: Contract and variable declaration of InsuranceLogic.

```
struct InsuranceContract{
        address customer; // Address of the policyholder
        uint256 T; // Day of the year between 1 and 365
        bytes32 station; // Description of the insured event
        uint256 l; // Payout amount
        uint256 p; // Probability of the event
        uint16 eta; // loading of the premium at the underwritting time
        uint256 refund; // Refund amount in case of bankruptcy
        uint8 status; // Status of the contract 0 = open, 1 = closed and settled, 2 =
            closed without compensation, 3 = refunded
    }
    mapping(uint256 => InsuranceContract) public insuranceContracts; // Mapping of
        insurance contracts
```

Code Fragment 16: InsuranceContract data structure

```
address[] public investorAddresses; // Array to store investor addresses
mapping(address => uint256) public Yt; // Balance of investors in tokens
```

Code Fragment 17: Array of investorAddresses and token holding