# Financial Instrument Pricing System Documentation

**Author:** Asmit Chauhan

## 1. Introduction and Overview

This document details the design and implementation of a C++-based pricing system for two financial instruments: ValueNote and DeliveryContract.

* A **`ValueNote`** represents a fixed-income security, akin to a bond, where an investor lends money to an issuer in return for periodic interest payments and principal repayment at maturity. My system provides functionalities to calculate its price based on an effective rate and vice-versa, along with first and second-order sensitivities, under three different rate conventions (Linear, Cumulative, Recursive).
* A **`DeliveryContract`** is a derivative instrument that gives the seller the option to deliver the "most economical" ValueNote from a predefined basket at a future expiration date. The buyer pays a price based on this delivered note's RelativeFactor. My system prices this contract by modeling the future effective rates of ValueNotes using a geometric Brownian motion driven by a single common random factor, approximating the resulting price ratios quadratically, and then calculating the expected minimum ratio. My system also calculates the sensitivity of the DeliveryContract price with respect to the volatility (σᵢ) and today's price (VP₀ᵢ) of each underlying ValueNote.

My system is designed with object-oriented principles to promote modularity, maintainability, and extensibility. It aims for accuracy in financial calculations while providing a framework for more complex derivative pricing.

## 2. Implementation Approach

My system is primarily composed of two C++ classes: ValueNote for the underlying asset and DeliveryContract (inheriting from a Basket base class) for the derivative.

### 2.1. `ValueNote` Class

Encapsulates properties and calculations for a single ValueNote.

**Attributes:** notional (N), maturity (M), valueRate (VR), paymentFreq (PF), effectiveRateVolatility (ERV).

**Constructor:** Validates input parameters (N, M, VR, PF > 0).

**Core Pricing and Rate Conversion Methods (Q1.1):** For Linear, Cumulative, and Recursive conventions.

* eRateToPriceLinear(eRate)
* priceToERateLinear(price)
* eRateToPriceCumulative(eRate)
* priceToERateCumulative(price, tolerance, maxIter): Uses bisection method.
* eRateToPriceRecursive(eRate)
* priceToERateRecursive(price, tolerance, maxIter): Uses bisection method.

**Sensitivity Methods - First Derivatives (Q1.2):** dP/dER and dER/dP.

* eRateToPriceSensitivityLinear(eRate)
* priceToERateSensitivityLinear(price)
* eRateToPriceSensitivityCumulative(eRate)
* priceToERateSensitivityCumulative(price)
* eRateToPriceSensitivityRecursive(eRate)
* priceToERateSensitivityRecursive(price)

**Sensitivity Methods - Second Derivatives (Q1.3):** d²P/dER² and d²ER/dP².

* eRateToPriceSecondDerivativeLinear(eRate)
* priceToERateSecondDerivativeLinear(price)
* eRateToPriceSecondDerivativeCumulative(eRate)
* priceToERateSecondDerivativeCumulative(price)
* eRateToPriceSecondDerivativeRecursive(eRate)
* priceToERateSecondDerivativeRecursive(price)

**DeliveryContract-related Methods (Q2):**

* forwardPrice(expirationTimeYears, riskFreeRate): Calculates forward price. riskFreeRate is input as a percentage and scaled by /100.0.
* getRiskAdjustedEffectiveRateCumulative(forwardPrice, expirationTimeYears): Calculates risk-adjusted effective rate based on the quadratic equation from the problem statement.
* getRiskAdjustedEffectiveRate(forwardPrice, expirationTimeYears, convention): Wrapper, currently supports "Cumulative".

### 2.2. `Basket` Class

Protected base class for DeliveryContract.

**Attributes:** basket (vector of ValueNote), standardizedValueRate, expirationTime (months), relativeFactorConvention, effectiveRateConvention, riskFreeInterestRate (as percentage), numberOfNotes.

**Constructor:** Initializes common parameters.

**Methods:** createBasket() (user input), getBasket().

### 2.3. `DeliveryContract` Class

Inherits from Basket and implements the derivative pricing logic and advanced sensitivities.

**Private Attributes:** relativeFactors, quadraticApprox (for price ratios, not directly stored but computed locally in computeAndPrintPriceAndProbabilities), forwardPrices, adjustedEffectiveRates.

**Key Methods:**

* computeRelativeFactors(): Based on relativeFactorConvention.
* computeForwardPricesAndAdjustedRates(): Pre-calculates forward prices and risk-adjusted ERs for all notes.
* fitQuadratic(z\_samples, ratios, vnIndex): Fits quadratics to price ratios using WLS (Weighted Least Squares).
* findIntersections(quadratic\_approximations): Finds intersections of price ratio quadratics.
* evaluateQuadratic(coeffs, z): Evaluates a quadratic.
* findOptimalValueNote(z, quadratic\_approximations): Finds the note with min ratio at z.
* integrateQuadraticWithNormalWeight(a, b, c, z1, z2): Analytical integration helper for (az² + bz + c)φ(z).
* computeAndPrintPriceAndProbabilities(): Core pricing method for the contract, computes quadraticApproxLocal.
* fitQuadraticForDeriv(z\_samples, y\_values, vnIndex): Helper to fit quadratics to derivative values (same logic as fitQuadratic).
* dAdjustedER\_dVP0(noteIndex): Calculates d(ER\*\_Tᵢ)/dVP₀ᵢ analytically using implicit differentiation of the quadratic defining ER\*\_Tᵢ (risk-adjusted effective rate).
* integrateDeliveryContractSensitivity(derivQuads, priceQuads): Generic function to integrate sensitivities of the optimal note.
* volatilitySensitivityForNote(noteIndex, sigma\_value): Calculates ∂(DeliveryContractPrice)/∂σᵢ.
* priceSensitivityForNote(noteIndex, VP0\_value): Calculates ∂(DeliveryContractPrice)/∂VP₀ᵢ.
* promptAndPrintUserSensitivities(): Orchestrates user input and prints detailed sensitivities.
* Various print utilities: printRelativeFactors(), printForwardPrices(), printRiskAdjustedEffectiveRates(), printQ1TableForFirstNote().

### 2.4. `main()` Function

Handles user input for global parameters, instantiates DeliveryContract, and orchestrates the calculation flow:

**1.** createBasket()

**2.** computeRelativeFactors()

**3.** printQ1TableForFirstNote()

**4.** printRelativeFactors()

**5.** computeForwardPricesAndAdjustedRates()

**6.** printForwardPrices()

**7.** printRiskAdjustedEffectiveRates()

**8.** computeAndPrintPriceAndProbabilities()

**9.** promptAndPrintUserSensitivities()

Includes a try-catch block for standard exceptions.

## 3. Mathematical Framework and Derivations

This section details the mathematical models and analytical derivations used in my system.

### 3.1. `ValueNote` Pricing Models and Sensitivities (Q1)

#### 3.1.1. Linear Model

**Price Formula:**

P(ER) = N \* (1 - (ER \* M) / 100)

where:

* N: Notional amount (this->notional)
* ER: Effective rate (%) (eRate parameter)
* M: Maturity in years (this->maturity)

**Derivatives:**

* dP/dER = - N \* M / 100
* d²P/dER² = 0

#### 3.1.2. Cumulative Model

**Price Formula:**

VP₀ = ( Σ\_{i=1}^{n-1} [ CF / (1 + r)ⁱ ] ) + [ (CF + N) / (1 + r)ⁿ ]

where:

* CF = (N \* VR) / (100 \* f): Periodic interest payment (interest in code)
* r = ER / (100 \* f): Periodic effective rate
* N: Notional amount (this->notional)
* VR: Value Rate (%) (this->valueRate)
* f: Payment frequency (this->paymentFreq)
* n = M \* f: Total number of payments (n in code)

**First Derivative (dVP₀/dER₀):**

dVP₀/dER₀ = - (1 / (100 \* f)) \* ( Σ\_{i=1}^{n-1} [ i \* CF / (1+r)ⁱ⁺¹ ] + [ n \* (CF+N) / (1+r)ⁿ⁺¹ ] )

(Implemented in ValueNote::eRateToPriceSensitivityCumulative)

**Second Derivative (d²VP₀/dER₀²):**

d²VP₀/dER₀² = (1 / (100 \* f)²) \* ( Σ\_{i=1}^{n-1} [ i\*(i+1)\*CF / (1+r)ⁱ⁺² ] + [ n\*(n+1)\*(CF+N) / (1+r)ⁿ⁺² ] )

(Implemented in ValueNote::eRateToPriceSecondDerivativeCumulative)

**Inverse Sensitivities (dER₀/dVP₀ and d²ER₀/dVP₀²):**

Calculated using standard inverse function derivative rules:

dER₀/dVP₀ = 1 / (dVP₀/dER₀)

d²ER₀/dVP₀² = - (d²VP₀/dER₀²) / (dVP₀/dER₀)³

#### 3.1.3. Recursive Model

**Price Formula (as implemented in ValueNote::eRateToPriceRecursive):**

Let FV be the sum of periodic interest payments (interest), each discounted back to the time of the next interest payment, and all accumulated up to the final interest payment.

FV\_0 = 0

For k = 1 to n-1:

FV\_k = (FV\_{k-1} + interest) / (1 + ER \* m\_k / 100)

(where m\_k is time period t\_{k+1} - t\_{k})

FV\_final\_interests = FV\_{n-1} + interest (value of all interests at time of last interest payment)

VP₀ = (FV\_final\_interests + notional) / (1 + ER \* M\_total / 100)

(where M\_total is the total maturity in years)

**Derivatives (as implemented in ValueNote::eRateToPriceSensitivityRecursive and eRateToPriceSecondDerivativeRecursive):**

The derivatives dFV/dER, d²FV/dER² are calculated iteratively alongside FV.

dFV\_k/dER = (dFV\_{k-1}/dER) / B\_k - (FV\_{k-1} + interest) \* (m\_k/100) / B\_k², where B\_k = (1 + ER \* m\_k / 100).

d²FV\_k/dER² = (d²FV\_{k-1}/dER²)/B\_k - 2\*(dFV\_{k-1}/dER)\*(m\_k/100)/B\_k² + 2\*(FV\_{k-1}+interest)\*(m\_k/100)²/B\_k³.

The final price derivatives dVP₀/dER and d²VP₀/dER² are then calculated using these accumulated dFV/dER and d²FV/dER² for FV\_final\_interests and applying the quotient rule for the final discounting step over M\_total.

#### 3.1.4. Effective Rate Calculation (Price-to-Rate)

**Bisection Method: Used for Cumulative and Recursive models (priceToERateCumulative, priceToERateRecursive).**

**Why did I use Bisection Method?**

**1. Guaranteed Convergence:** If a root exists in the [low, high] interval and the function is continuous.

**2. Robustness:** Requires only function continuity and an interval bracketing the root.

**3. Domain Knowledge for ER:** Natural bounds (e.g., 0-100%), also the monotonically decreasing nature of price w.r.t. ER, which can be verified from the formulae given.

**4. Implementation Simplicity.**

// Snippet from ValueNote::priceToERateCumulative

double low = 0.0; double high = 100.0; // Initial search range

while (high - low > tolerance && i < maxIter) {

double mid = (low + high) / 2.0;

double diff = eRateToPriceCumulative(mid) - price;

if (diff < 0) high = mid; else low = mid;

// ...

}

return (low + high) / 2.0;

### 3.2. `DeliveryContract` Price Calculation

#### 3.2.1. Forward Price and Risk-Adjusted Effective Rate

**Forward Price (VP\_T) for ValueNote i (ValueNote::forwardPrice):**

VP₀ᵢ = eRateToPriceCumulative(valueRateᵢ) (Price today using its own valueRate as ER)

CompoundedValue = VP₀ᵢ \* (1 + (riskFreeInterestRate/100.0) \* expirationTimeYears)

AccruedInterestPV = Σ\_{j where t\_j <= T} [ interest\_j / (1 + (riskFreeInterestRate/100.0) \* t\_j) ]

FutureValueOfAccrued = AccruedInterestPV \* (1 + (riskFreeInterestRate/100.0) \* expirationTimeYears)

VP\_Tᵢ = CompoundedValue – FutureValueOfAccrued

**Risk-Adjusted Effective Rate (ER\*\_Tᵢ) for ValueNote i (ValueNote::getRiskAdjustedEffectiveRateCumulative):**

Solved from the quadratic equation provided in the problem statement (p.4):

A\*(ER\*\_Tᵢ)² + B\*(ER\*\_Tᵢ) + C = 0

where coefficients A, B, C are:

A\_coeff = 0.5 \* d²P/dER² \* exp(σᵢ² \* T)

B\_coeff = dP/dER - (d²P/dER²) \* ER\_T\_unadj

C\_coeff = 0.5 \* (d²P/dER²) \* ER\_T\_unadj² - (dP/dER) \* ER\_T\_unadj

(**Note:** Out of the two roots, the chosen root is the one closer to ER\_T\_unadj. This ensures that unrealistic roots are omitted, such as negative or too large ones.)

#### 3.2.2. Quadratic Approximation of Price-to-RelativeFactor Ratio

* The Price-to-RelativeFactor ratio for each ValueNote k is approximated:

R\_k(z) = VP\_k(ER\_Tz\_k) / RF\_k ≈ q\_k(z) = a\_kz² + b\_kz + c\_k

where ER\_Tz\_k = ER\*\_T\_k \* exp(σ\_k√T z - 0.5σ\_k²T).

* Coefficients (a\_k, b\_k, c\_k) found via WLS (DeliveryContract::fitQuadratic).

#### 3.2.3. Integration Approach for Contract Price

* DeliveryContract Price = E[min\_k R\_k(z)] = ∫ min\_k (q\_k(z)) φ(z) dz
* Calculated by summing analytical integrals over intervals [z\_m, z\_{m+1}] defined by intersections of q\_k(z):

Price = Σ\_m ∫[z\_m, z\_{m+1}] q\_{opt,m}(z) \* φ(z) dz

using DeliveryContract::integrateQuadraticWithNormalWeight.

### 3.3. `DeliveryContract` Price Sensitivities (Q2.4)

**My Unique Approach:**

The sensitivity of the DeliveryContract Price (P\_DC) w.r.t. a parameter θ (e.g., σᵢ or VP₀ᵢ) is ∂P\_DC/∂θ.

If P\_DC = ∫ F(z, θ) φ(z) dz where F(z, θ) = min\_k q\_k(z, θ).

Then ∂P\_DC/∂θ = ∫ [ ∂F(z, θ) / ∂θ ] φ(z) dz.

In an interval where q\_j(z,θ) is minimal, ∂F/∂θ = ∂q\_j/∂θ.

The term ∂q\_j(z,θ)/∂θ (which is ∂(VP\_j(ER\_Tz\_j)/RF\_j)/∂θ) is itself approximated by a quadratic in z, let's call it dq\_j(z).

So, ∂P\_DC/∂θ = Σ\_m ∫[z\_m, z\_{m+1}] dq\_{opt,m}(z) φ(z) dz.

The intervals [z\_m, z\_{m+1}] are determined by intersections of the original q\_k(z)(the original quadratic fit for the DeliveryContract Price). We thus reuse the same optimal ValueNotes between intersections as used for Price Calculation. The final integration is facilitated by the generic integration function for Sensitivity of DeliveryContract price w.r.t any variable θ.

#### 3.3.1. Volatility Sensitivity (`∂P\_DC/∂σᵢ`)

This is DeliveryContract::volatilitySensitivityForNote(i, sigma\_value).

**1.** For each ValueNote k and each sample z\_s, calculate D\_ks = ∂R\_k(z\_s)/∂σᵢ.

∂R\_k(z\_s)/∂σᵢ = (1/RF\_k) \* (dVP\_k/dER\_Tz\_k) \* (∂ER\_Tz\_k/∂σᵢ).

∂ER\_Tz\_k/∂σᵢ is non-zero only if k = i, where it is ER\_Tz\_i \* (√T z\_s - σᵢT).

**2.** Fit a quadratic dq\_k(z) to the samples D\_ks for each k (using fitQuadraticForDeriv). This dq\_k(z) approximates ∂R\_k(z)/∂σᵢ.

**3.** Use integrateDeliveryContractSensitivity(derivQuads, priceQuads) to sum ∫ dq\_{opt,m}(z) φ(z) dz.

#### 3.3.2. Price Sensitivity (`∂P\_DC/∂VP₀ᵢ`)

This is DeliveryContract::priceSensitivityForNote(i, VP0\_value).

**1.** For each ValueNote k and sample z\_s, calculate D\_ks = ∂R\_k(z\_s)/∂VP₀ᵢ.

∂R\_k(z\_s)/∂VP₀ᵢ = (1/RF\_k) \* (dVP\_k/dER\_Tz\_k) \* (∂ER\_Tz\_k/∂VP₀ᵢ).

∂ER\_Tz\_k/∂VP₀ᵢ = (∂ER\*\_T\_k/∂VP₀ᵢ) \* exp\_term\_k(z\_s). This is non-zero only if k = i.

**2.** ∂ER\*\_T\_i/∂VP₀ᵢ is calculated analytically in dAdjustedER\_dVP0(i):

It uses implicit differentiation on the quadratic Q(ER\*\_Tᵢ, VP\_Tᵢ) = A(ER\*\_Tᵢ)² + B(ER\*\_Tᵢ) + C\_orig = 0 (from getRiskAdjustedER).

dER\*\_Tᵢ/dVP\_Tᵢ = -(∂Q/∂VP\_Tᵢ) / (∂Q/∂ER\*\_Tᵢ).

The code dAdjustedER\_dVP0 implements this as:

dQ\_dER\_adj = 2 \* d2P\_adj \* ER\*\_Tᵢ (This d2P\_adj is d²P/dER² at ER\*\_Tᵢ)

dQ\_dFwd = 1.0 (Assuming VP\_Tᵢ term is linear with coeff 1 in relevant quadratic).

dER\*\_Tᵢ/dVP\_Tᵢ = -1.0 / dQ\_dER\_adj.

Then, ∂ER\*\_T\_i/∂VP₀ᵢ = (dER\*\_Tᵢ/dVP\_Tᵢ) \* dVP\_Tᵢ/dVP₀ᵢ = (dER\*\_Tᵢ/dVP\_Tᵢ) \* (1 + (riskFreeRate/100)\*T).

**3.** Fit quadratic dq\_k(z) to samples D\_ks for each k.

**4.** Use integrateDeliveryContractSensitivity(derivQuads, priceQuads) to sum integrals.

## 4. Key Assumptions

* **Single Factor Model:** All ValueNote effective rates driven by a single z.
* **Quadratic Approximation:** R\_k(z) and ∂R\_k(z)/∂θ are well-approximated by quadratics over z ∈ [-3, 3].
* **Integration Range:** z ∈ [-3, 3] (Z\_MIN, Z\_MAX) covers sufficient probability mass. NUM\_SAMPLES = 2000 is adequate.
* **Bisection Method Convergence:** Assumed for price-to-rate calculations.
* **Risk-Adjusted ER Quadratic:** The specific quadratic form from the problem statement (p.4) and its implementation in getRiskAdjustedEffectiveRateCumulative are used. The derivation in dAdjustedER\_dVP0 for ∂ER\*\_Tᵢ/∂VP\_Tᵢ is based on this specific implementation.

## 5. Implementation Challenges and Solutions

### 5.1. Numerical Stability and Correctness

**Risk-Adjusted Effective Rate (ER\*\_T):**

* **Challenge**: Quadratic for ER\*\_T might have no real solutions or unstable solutions.
* **Solution:** Check discriminant (B\*B - 4\*A\*C < 0). If no real solution, print error and return unadjusted ER\_T. Choose root closer to unadjusted ER\_T.

**WLS Quadratic Fit (fitQuadratic, fitQuadraticForDeriv):**

* **Challenge:** Potential singularity if determinant of the WLS system is near zero.
* **Solution:** Check abs(det) < 1e-12; if singular, fall back to a constant fit (c = sy/s0, a=0, b=0).

**Sensitivity dAdjustedER\_dVP0:**

* **Challenge:** Potential division by zero if dQ\_dER (denominator 2 \* d2P \* adjER) is near zero.
* **Solution:** The code includes a debug print for dQ\_dER.

## 6. Error Handling

**1. Input Validation:** ValueNote constructor throws std::invalid\_argument.

**2. Numerical Stability Checks:** WLS determinant, ER\*\_T discriminant.

**3. Unsupported Conventions:** getRiskAdjustedEffectiveRate and computeRelativeFactors throw std::invalid\_argument for unsupported conventions.

**4. Exception Handling in `main()`:** Catches std::exception. Also printing debugging statements after each method call.

## 7. Design Patterns and Architectural Elements

* **Object-Oriented Programming with Inheritance:** ValueNote, Basket, DeliveryContract(Inherited from Basket) classes.
* **Strategy Pattern (Implicit):** String-based dispatch for conventions.
* **Modularity**: Separation of concerns, helper numerical functions.

## 8. Scalability, Maintainability, and Production Readiness

**Scalability:**

* DeliveryContract Price: O(N\_notes \* NUM\_SAMPLES + N\_notes²) (approx).
* DeliveryContract Sensitivities: Each sensitivity calculation (volatilitySensitivityForNote or priceSensitivityForNote) has a similar complexity to pricing the contract itself, as it involves sampling derivatives for all notes and fitting quadratics. This is performed for each note for which sensitivity is requested.
* **Maintainability and Product Readiness:** Structured code. Robust logging. Printing functions for intermediate calculations (like forward price and risk-adjusted ER) which are not part of the result.

## 9. Future Enhancements

* Explicit Strategy Pattern for conventions, including modelling for Linear and Recursive conventions.
* More sophisticated financial models (multi-factor, other processes).
* Alternative numerical methods (e.g., splines for approximation).
* Parallelization for sampling loops or per-note sensitivity calculations.
* GUI/API, database integration.

## 10. Validating the correctness of my Approach

**ValueNote Calculations:** Compared against manual examples, checked boundary conditions and monotonicity.

**DeliveryContract Price:** Tested with simple cases (1 note, identical notes).

**Sensitivities:**

* ValueNote sensitivities: Compared with finite differences.
* DeliveryContract sensitivities: Compared results of volatilitySensitivityForNote / priceSensitivityForNote with finite differences of the full DeliveryContract price (i.e., re-price contract with σᵢ ± h or VP₀ᵢ ± h).
* Numerical Stability: Tested with extreme (but valid) parameters.
* Debug Output: [DEBUG] statements in code aid runtime validation.

## 11. Summary of Challenges Faced

* **Mathematical Complexity:** Deriving and implementing analytical derivatives by hand first.
* **Numerical Stability**: WLS, root finding, division by small numbers.
* **Formula Interpretation:** Translating problem statement to code accurately.
* **Algorithm Design for DeliveryContract**: Multi-step pricing and sensitivity calculations were very intricate.
* **Correctness of dAdjustedER\_dVP0:** Ensuring the implicit differentiation correctly matches the specific quadratic form used for ER\*\_T.

## 12. Results Analysis

**DeliveryContract price**: less than or equal to the forward price of any single ValueNote in the basket, when normalized by its Relative Factor. The reason can be:

* In financial terms, an option on the minimum of several assets is always less than or equal to the price of an option on any single one of those assets, assuming comparable terms. Also, the given pricing model has many benefits, in terms of optionality and mitigation of risk, which lowers the expected cost of delivery.

**Zero Values for all except one** **DeliveryContract sensitivity:**

* This seem to be corresponding to the zero delivery probabilities of all but one ValueNote, which might imply that one ValueNote is superiorly beneficial than others.

## 13. Conclusion

This C++ system implements a comprehensive solution for pricing ValueNotes and DeliveryContracts, including advanced analytical sensitivities for the contract price. Key refinements include accurate risk-free rate handling and a more rigorous approach to contract sensitivities via integration of ratio derivatives. The object-oriented design facilitates this complexity. For production of this system, further hardening in error handling, configuration, and automated testing would be beneficial.